

**FFRU: A TIME- AND SPACE-EFFICIENT CACHING
ALGORITHM**

A Dissertation
Submitted to
the Temple University Graduate Board

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

by
Benjamin Garrett
August, 2021

Examining Committee Members:

Dr. Richard Beigel, Advisory Chair, Computer & Information Sciences
Dr. Yuan Shi, Computer & Information Sciences
Dr. Pei Wang, Computer & Information Sciences
Dr. Daniel Szyld, External Member, Mathematics

©

by

Benjamin Garrett

August, 2021

All Rights Reserved

ABSTRACT

FFRU: A TIME- AND SPACE-EFFICIENT CACHING ALGORITHM

Benjamin Garrett

DOCTOR OF PHILOSOPHY

Temple University, August, 2021

Dr. Richard Beigel, Chair

Cache replacement policies have applications that are nearly ubiquitous in technology. Among these is an interesting subset which occurs when referentially transparent functions are memoized, eg. in compilers, in dynamic programming, and other software caches. In many applications the least recently used (LRU) approach likely preserves items most needed by memoized function calls. However, despite its popularity LRU is expensive to implement, which has caused a spate of research initiatives aimed at approximating its cache miss performance in exchange for faster and more memory efficient implementations.

We present a novel caching algorithm, *Far From Recently Used (FFRU)*, which offers a simple, but highly configurable mechanism for providing lower bounds on the usage recency of items evicted from the cache. This algorithm preserves the constant time amortized cost of insertions and updates and minimizes the memory overhead needed to administer the eviction guarantees. We study the cache miss performance of several memoized optimization problems which vary in the number of subproblems generated and the access patterns exhibited by their recursive calls. We study their cache miss performance using LRU cache replacement, then show the performance of FFRU in these same problem scenarios. We show that for equivalent minimum eviction age guarantees, FFRU incurs fewer cache misses than LRU, and does so using less memory.

We also present some variations of the algorithms studied (Fibonacci, KMP, LCS, and Euclidean TSP) which exploit the characteristics of the cache replacement algorithms being employed, further resulting in improved cache miss performance. We present a novel implementation of a well known approximation algorithm for the Euclidean Traveling Salesman Problem due to Sanjeev Arora. Our implementation of this algorithm outperforms the currently known implementations of the same. It has long remained an open question whether or not algorithms relying on geometric divisions of space can be implemented into practical tools, and our powerful implementation of Arora's algorithm establishes a new benchmark in that arena.

ACKNOWLEDGEMENTS

If you want to go fast, go alone. If
you want to go far, go together.

African proverb

I cannot say enough to express gratitude to my advisor, Richard Beigel, for having demonstrated unbounded patience with me and for having helped me develop this project into what it is. I thank him for having encouraged me to own my work, to let my results speak for themselves, and to trust my critical thought. I have learned more from him than from anyone else in my life, lessons that will forever shape my approach to computer science, to career pursuits, and to life. It has been an honor to work with him.

I warmly thank my readers, Justin Shi, Pei Wang, and Daniel Szyld. I especially thank Justin Shi for his numerous thought provoking questions about the direction of my research efforts, as well as his priceless advice about how to navigate the nuances of doctoral studies and research. I must thank Julie Skrocki for having figured out my correct deadlines.

I would like to thank Bjarne Stroustrup for providing some useful perspective on some of the wider implications and applications of the caching algorithm presented herein. I owe much gratitude to Michael Davidson, who provided me with very worthwhile and effective strategies making sure this endeavor came to fruition and for improving my ability to navigate better the subtleties of interpersonal collaboration. To my dear friend, coach, and mentor, Alwyn Callender, I am deeply indebted for his perspicacious suggestions on how best to find balance while I concluded my research and for some helpful feedback on early drafts of this thesis. To my good friend Lior Grinberg I am grateful for his advice during the final moments when I closed out this project. Special thanks go out to Manish Chakrabarti, whose questions regarding my research gave me useful perspective, and whose feedback helped me improve the exposition of this thesis.

I am grateful to David Petrou, Carter Schonwald, and Eric Martinez for having commented on and provided pointed feedback on unpublished drafts of the work reported in this thesis. Their insight has been invaluable as I neared the end of this journey and sought to gain a wider appreciation for what was accomplished during this project. David Petrou and Dushyanth Narayanan were pursuing their doctorate degrees in computer science when I was still working on my bachelor degree, and I remember being awestruck by the breadth and depth of their intellect, both in computer science and in a variety of topics. I am grateful to have met them, because part of the reason why I originally decided to pursue this Ph.D. was to try and follow in their footsteps.

I owe a huge debt of gratitude to Ariella Mordukhayeva, who provided immeasurable assistance with typesetting and proofreading this document as well as my defense presentation slides. She has not only displayed a tour de force of \LaTeX debugging prowess, but she has also given me an unlimited supply of encouragement throughout the many phases of this project. I thank Sukanya Raj for her support and for being a good friend who patiently listened to me elaborate on my project over many iterations of successes and challenges.

To the Brothers family, Rob, Julie, Maya, and Jack, I wish to express my heartfelt thanks. They opened up their home to me during a period of this project when I really needed it. They have been there for me over various challenging moments when it became difficult to balance the needs of my doctoral studies with those of my personal and professional life. Rob Brothers has been an enduring ally and friend to me throughout this journey.

I would like to thank my loving wife, Padmini, for having giving me her unbridled support and encouragement. She has been right by my side through thick and thin, offering up on numerous occasions priceless wisdom gained through her own doctoral travails. I thank my two brilliant sons for having kept me grounded during the final years of this project, for their unlimited supply of hugs, and for always reminding me to think outside the box. I would like to thank Nandini Biswas, whose gentle and positive energy always put a

smile on my face especially during the final years of this project. To Amitabha and Sibani I am grateful for their never ending encouragement.

I wish to acknowledge my father, whose guidance, warmth, and eternal love stay with me every day and have given fuel to my perseverance. My father's solid work ethic has always inspired me to pour every ounce of my energy into this project. I know he would be proud of the milestones I have reached up to this point.

And finally I literally and figuratively would not be here if it weren't for my mother. She has been an endless source of uplifting energy and sunshine, always reminding me to hold my head high and to believe in myself. None of this would have been possible without all that she has done for me.

This dissertation was typeset using the L^AT_EX typesetting system originally developed by Leslie Lamport, based on T_EX created by Donald Knuth. This research includes calculations carried out on Temple University's HPC resources and thus was supported in part by the National Science Foundation through major research instrumentation grant number 1625061 and by the US Army Research Laboratory under contract number W911NF-16-2-0189.

To Anand and Aarav.

TABLE OF CONTENTS

ABSTRACT	iv
ACKNOWLEDGEMENT	vi
DEDICATION	ix
LIST OF FIGURES	xiii
LIST OF TABLES	xv
LIST OF ALGORITHMS	xvii
1 INTRODUCTION	1
1.1 Challenges	1
1.2 Contributions	4
1.3 Remaining Chapters	6
2 BACKGROUND	9
2.1 Cache Replacement Policies	9
2.1.1 Clairvoyant Caching and Bélády’s Anomaly	10
2.1.2 Random	11
2.1.3 FIFO, Second Chance, and Clock Variants	11
2.1.4 LRU	12
2.1.5 LRU approximations	13
2.1.6 Adaptive Replacement Cache	14
2.2 Traveling Salesman Problem	15
2.2.1 Variations of the TSP	15
2.2.2 Applications of the TSP	15
2.2.3 Defining the TSP	16
2.2.4 Complexity of the TSP	17
2.2.5 Heuristic Approaches to the TSP	17
2.2.6 Algorithmic Approaches to the TSP	18

2.2.7	Concorde	20
2.2.8	Arora's PTAS	21
2.3	Conventions & Notation	22
2.3.1	Pseudocode Listings	22
2.3.2	Cache Replacement Policies	23
2.3.3	Approximation Algorithms	24
3	FAR FROM RECENTLY USED (FFRU) CACHE REPLACEMENT	26
3.1	Far From Recently Inserted (FFRI)	26
3.1.1	Cuckoo Clock Caching	27
3.1.2	Drunken Cuckoo Clock Caching	28
3.1.3	d -Drunken Cuckoo Caching	28
3.2	Far From Recently Used	28
3.2.1	FFRU Absolute	28
3.2.2	FFRU Relative	29
3.3	Parameter Constraints & Performance Guarantees	30
3.4	Run Time Analysis	31
3.4.1	Average Probes per Insertion into Cuckoo Hash Tables with Deletions	32
3.4.2	Average Probes per Insertion into Cuckoo Hash Tables without Deletions	35
4	VERIFYING FFRU'S PERFORMANCE GUARANTEES	39
4.1	Verifying Recency Guarantees for Fixed Operation Sequences	40
4.2	Fibonacci	42
4.2.1	Version 0	42
4.2.2	Version 1	47
4.2.3	Version 2	48
4.2.4	Conclusions	52
4.3	KMP String Matching	55
4.3.1	Developing a Recursive Version	56
4.3.2	Prefix-suffix Function	59
4.3.3	Failure Function	65
4.3.4	Conclusions	65
4.4	Longest Common Subsequence	67
4.4.1	Cost Function	69
4.4.2	Oblivious LCS	70
4.4.3	Traceback Function	75
4.4.4	Conclusions	78
4.5	Arora's Algorithm using a100	81

4.5.1	2 Points per Border, all subproblems	81
4.5.2	1 Point per Border, all subproblems	82
4.5.3	2 Points per Border, limited subproblems	82
4.5.4	Conclusions	83
5	IMPLEMENTING ARORA'S TSP PTAS	88
5.1	Perturbing Input Points	89
5.2	Partitioning Strategies	90
5.3	Portal Points	92
5.4	Cost Function	92
5.4.1	Solving the Base Case - Leaf Partitions	95
5.4.2	Solving non-base cases	96
5.4.3	Subproblems involving one sibling point	99
5.4.4	Subproblems involving two sibling points	100
5.4.5	Dead-end scenarios due to impossible child partition sub- problems	104
5.5	Traceback	104
5.5.1	Base case	106
5.5.2	Non-base case	107
5.6	Schema Tables and the Catalan Numbers	107
5.7	Automatic Table Generation	110
5.8	Further Implementation Considerations	111
5.9	Performance without limiting cache space	113
5.9.1	Configuration Scenarios Tested	114
5.9.2	Solution Quality	115
5.10	Run Time and Cache Miss Performance	119
5.11	Areas for Improvement in Our Implementation	129
5.12	Conclusions	129
6	CONCLUSIONS	131
	REFERENCES	135

LIST OF FIGURES

3.1	Average # of probes / insertion with cuckoo hashing, with deletions	37
3.2	Linear regression of average probes / insertion as a function of $1/(0.97 - \alpha)$	37
3.3	Bounding average probes / insertion as a function of $1/(0.97 - \alpha)$	38
3.4	Bounding average probes / insertion as a function of α	38
4.1	Versions 0A, 0B, and 0C, Cache Size 1	46
4.2	Version 1A, Bounding the Critical Cache Size	49
4.3	Fibonacci Version 2B, confirming our LRU observation programmatically	55
4.4	Fibonacci Version 2B, LRU and FFRU compared	56
4.5	Fibonacci Version 2B, FFRU variants compared	57
4.6	KMP's prefix-suffix function memoized with LRU	63
4.7	KMP's prefix-suffix, comparing LRU to FFRU	63
4.8	KMP's prefix-suffix, FFRU all three variants	64
4.9	LCS Cost Version 2, comparison between LRU and FFRU	70
4.10	LCS Cost Version 2, FFRU all three variants	72
4.11	OLCS Cost Version 4, comparison between LRU and FFRU	76
4.12	OLCS Cost Version 4, FFRU all three variants	77
4.13	LCS Traceback Version 2, critical cache size as a function of problem size	79
4.14	OLCS Traceback Version 4, critical cache size as a function of problem size	80
4.15	LCS Traceback Version 2, comparison between LRU and FFRU	81
4.16	LCS Traceback Version 2, FFRU all three variants	82
4.17	OLCS Traceback Version 4, comparison between LRU and FFRU	83
4.18	OLCS Traceback Version 4, FFRU all three variants	84
4.19	Arora, <i>burma14</i> , 2 points per border, comparison between LRU and FFRU	85

4.20	Arora, <i>burma14</i> , 2 points per border, FFRU all three variants	85
4.21	Arora, <i>pr136</i> , 1 point per border, comparison between LRU and FFRU	86
4.22	Arora, <i>pr136</i> , 1 point per border, FFRU all three variants	86
4.23	Arora, <i>u2319</i> , 2 points per border, limited subproblems, comparison between LRU and FFRU	87
4.24	Arora, <i>u2319</i> , 2 points per border, limited subproblems, FFRU all three variants	87
5.1	A Partitioned Instance	91
5.2	A Partitioned Instance with Portal Points	93
5.3	Partition and portal points with local indices	97
5.4	Left and right partitions with common sibling points	100
5.5	Portal pairing involving one sibling point	101
5.6	Portal pairing involving no sibling points	102
5.7	Portal pairing involving two sibling points entered from left partition	103
5.8	Portal pairing involving two sibling points entered from right partition	103
5.9	Anomaly scenario - 3 pairings, 3 sibling points needed	105
5.10	Anomaly scenario - 4 pairings, 4 sibling points needed	105
5.11	A Solved Instance	107
5.12	Canonical schemata for one, two, three, and four tour segments.	109
5.13	Cache misses as a function of problem size - 2 portals/border, all subproblems	128
5.14	Cache misses as a function of problem size - 1 portal/border, all subproblems	128
5.15	Cache misses as a function of problem size - 2 portals/border, limited subproblems	129

LIST OF TABLES

3.1	Constraints on κ , d , and a	30
3.2	Performance guarantees.	30
3.3	Number of timestamps that are inspected.	33
3.4	Average Run Time for Insertions using 4 Cuckoo Tables.	33
3.5	Average Run Time for Retrievals using 4 Cuckoo Tables.	34
4.1	FFRU Parameters used with KMP Prefix-suffix function.	64
4.2	FFRU Parameters used with LCS Cost Version 2.	71
4.3	FFRU Parameters used with OLCS Cost Version 4.	78
5.1	Growth of canonical schemata.	110
5.2	a100 configuration scenarios tested.	114
5.3	Average approximation factor achieved by a100	115
5.4	Approximation factor (α), 2 portals/border, all subproblems.	116
5.5	Approximation factor (α), 1 portal/border, all subproblems.	116
5.6	Approximation factor (α), 2 portals/border, limited subproblems.	117
5.7	Approximation factor (α), 2 portals/border, limited subproblems (cont.).	118
5.8	Run time and cache misses, 2 portals/border, all subproblems.	120
5.9	Run time and cache misses, 1 portal/border, all subproblems.	121
5.10	Run time and cache misses, 1 portal/border, all subproblems (cont.).	122
5.11	Run time and cache misses, 2 portals/border, limited subproblems (cont.).	123
5.12	Run time and cache misses, 2 portals/border, limited subproblems (cont.).	124
5.13	Run time and cache misses, 2 portals/border, limited subproblems (cont.).	125
5.14	Run time and cache misses, 2 portals/border, limited subproblems (cont.).	126

5.15 Run time and cache misses, 2 portals/border, limited subproblems (cont.)	127
---	-----

LIST OF ALGORITHMS

1	Procedure to measure average # of probes per insertion using a k -ary cuckoo hash table with deletions	34
2	Procedure to measure average # of probes per insertion using a k -ary cuckoo hash table without deletions	36
3	Fibonacci Version 0A	44
4	Fibonacci Version 0B	45
5	Fibonacci Version 0C	46
6	Fibonacci Version 1A	48
7	Fibonacci Version 2A	51
8	Fibonacci Version 2AR	54
9	KMP Iterative String Search	59
10	KMP prefix-suffix iterative version	62
11	KMP prefix-suffix recursive version	62
12	KMP failure function iterative version	66
13	KMP failure function recursive version	66
14	KMP recursive string search	67
15	LCS Cost Version 1	73
16	LCS Cost Version 2	73
17	Oblivious LCS Cost Version 1	74
18	Oblivious LCS Cost Version 2	74
19	Oblivious LCS Cost Version 3	74
20	Oblivious LCS Cost Version 4	75
21	Oblivious LCS Cost Version 5	75

22	Oblivious LCS Cost Version 6	75
23	LCS Traceback	80
24	High level procedure for Arora's PTAS	89
25	Perturbing and re-scaling input points	90
26	Recursive Cost Function	94
27	Traceback procedure to obtain the shortest salesman tour . . .	106
28	Schema table source code generator	111

CHAPTER 1

INTRODUCTION

Computers are useless. They can only give you answers.

Pablo Picasso

1.1 Challenges

Research on cache replacement policies has seen decades of activity for one simple reason. Despite the growing amounts of memory available in primary and secondary storage devices, the memory appetite of information systems keeps growing as well. It is commonplace today to speak of the memory resources of systems in terms of terabytes, petabytes, and exabytes, however that does not mean we are ready to throw terabytes of memory at every single task. For example when waiting for source code to compile and build, software engineers want faster completion times, but not at the expense of bringing the available memory on their systems to its knees.

For decades the gap in performance between processor speed and memory speed has grown despite ongoing attempts to cope with it. The average access time of DRAM hovers around 60 ns, whereas a typical ARM11 MPCore processor clock period is around 2 ns. Even though embedded SRAM can

attempt to mitigate this bottleneck, allowing memory retrievals every clock cycle, this type of memory is very expensive and has severely limited capacity¹. As a result when designing fast systems the memory access time is the critical performance bottleneck.

Techniques for solving problems quickly while limiting the memory requirements thus indeed remain relevant. In software engineering modern build systems make heavy use of caching techniques to automate and speed up the software build process². Contemporary enterprise database systems like Redis, MongoDB, and Cassandra feature the ability to deploy cache replacement policies like LRU, and research on novel caching techniques with databases continues (for an example see [Chavan and Sane, 2011]).

Evidence of the continued importance of caching/memoization techniques can be seen by the growth in programming language constructs, which allow annotations to mark functions for automatic memoization. Michie coined the term memoization to denote the augmentation of functions with a cache of argument-result pairs [Michie, 1968]. Hughes later applied applied memoization, in the original sense of Michie, to functional programs [Hughes, 1986]. Almost all of the popular languages embraced by industry now offer automatic memoization as either a built-in language feature or standard library, owing to the heightened awareness on software engineering teams of the possibility to gain performance advantages through memoization. In functional programming settings, lazy values are automatically cached for performance advantages. Memoization is fundamental to the implementation of lazy data structures. Using the *call-when-needed* programming construct known as the “think”, one can delay the execution of a function by wrapping it in a no-argument lambda expression, eg. $g = () \rightarrow f(x)$, which is

¹For a good discussion of cache memory performance tradeoffs and further motivation of the performance gap between processor speed and memory access speed, consult [Hennessy and Patterson, 2017].

²To glimpse some of the pain points related to tracking down third party artifacts and their transitive dependencies, see [Winters et al., 2020], which discusses the importance of caching to manage complexity in build systems.

then automatically memoized behind the scenes. Provided a function exhibits referential transparency, using memoization when memory resources permit can often speed up applications dramatically. For some background on the functional programming theoretical underpinnings of lazy values, check out [Michaelson, 1989] and [Okasaki, 1998]. For a treatment on how automatic memoization in Scala can be elegantly accomplished via functors, see [Chiusano and Bjarnason, 2014]. Dynamically typed languages like Javascript, Python, and Closure also support annotating functions for automatic function value caching (see [Halloway and Bedra, 2018] and [Emerick et al., 2012]). Automatic memoization was first introduced by Peter Norvig [Norvig, 1991], with applications to context free language parsing algorithms like CYK and Earley’s algorithm³. In other contexts like telecommunications systems (DNS caches, routing table management algorithms), operating systems, compilers, gaming applications, and many other settings, caches are employed to derive improvements in execution speed at the cost of some hopefully limited amount of memory overhead.

To address the challenges of effectively maintaining caches, many cache replacement policies have been developed, with the LRU family of algorithms having seen experimental results that make it often the preferred method. Many approximation algorithms exist which reduce the time and/or memory resources compared to LRU. However, in these various alternative caching algorithms the evicted items do not provide the same guarantee on the minimum usage age of evicted items the way that LRU does. The ability of LRU to adapt to changing degrees of popularity of cached items and its clear lower bound on the minimum usage age of evicted items has cemented its position as a “gold standard” in cache replacement policies, also because evicting items “farthest in the past” is like a mirror image of Bélády’s “farthest in the future” idealized cache replacement policy. Nonetheless, the implementation costs associated with the pure, exact version of LRU, are a disadvantageous aspect of

³For a good discussion of these algorithms and some theoretical implications, see [Floyd and Beigel, 1994].

this caching algorithm.

This is particularly true in hardware and operating systems contexts where high associativity renders infeasible the usage of exact LRU⁴. One could avoid excessive memory overhead by implementing LRU as a priority queue, however this would suffer the amortized $O(\log n)$ insertion and retrieval times. The preferred approach is usually to place the items into a hash table which is tied to a linked list, so that insertion and retrievals, as well as updates to the recency ranks of the items, are all done in $O(1)$ time. However, the memory overhead with this approach involves an additional $O(n)$ for the linked list. There have been many attempts to mitigate the memory overhead issue, but these typically lose the recency guarantees on how old the items are when they are evicted.

1.2 Contributions

The main contribution of this project arises in the form of a novel family of cache eviction algorithms (FFRI, FFRU Absolute, FFRU Relative), which provides the ability to configure guarantees on both the minimum recency of items in the cache and the maximum load factor of the same. Our goal is to match or exceed the performance of LRU under a variety of application workload scenarios and to do so using less overall memory overhead. To establish this, we take an empirical approach, by solving memoized optimization problems and noting their cache miss growth as a function of decreasing cache size or increasing problem size. When we compare LRU with three variants of our proposed caching algorithm, *Far From Recently Used (FFRU)*, we look at the number of cache misses incurred by caches having equivalent recency guarantees. Our expectation is that FFRU incurs no greater cache misses than those incurred by LRU.

In the course of verifying the performance of our new caching algorithm

⁴For a good explanation of set associativity in hardware caching and its relevance to LRU, consult [Patterson and Hennessy, 2013].

(FFRU), this thesis presents novel techniques for solving recursive, memoized problems when the cache replacement policy in use evicts items that have not been recently accessed or inserted, where recency can be measured in terms of time (*time recency*) or the number of cache operations (*item recency*). As we investigate several chosen optimization problems, we perform algorithm optimizations which exploit the access patterns of LRU's eviction policy and achieve on several occasions notable improvements in their cache miss performance. Some of the problems we studied are straight forward to memoize, since they are by default recursive algorithms. This includes Fibonacci, LCS, and Arora's TSP PTAS. However, in order to memoize the KMP string matching algorithm we first convert it into recursive form, which then allows us to study the effect of storing only a subset of the values it needs to perform string matching. We also reformulate these problems in the attempts of achieving improved cache miss performance. Our reformulations of the Fibonacci algorithm, the KMP string matching algorithm, and the longest common subsequence algorithm, show the improved cache performance achievable when the algorithm is designed with a specific cache replacement policy in mind.

Our work on one of the optimization problems used for testing FFRU provides a separate set of contributions of its own, so we devote an entire chapter to our implementation of it. We are referring to the Euclidean traveling salesman problem. Many exact and approximate methods have been developed for solving the TSP. A notable and interesting subset of these is the category of algorithms which rely on geometric divisions of space in order to find their solutions. It has long remained an open question whether or not such algorithms can be implemented into practical tools. One such algorithm is Arora's polynomial-time approximation scheme (PTAS) for the Euclidean TSP, which partitions the space of data points then finds a short tour of these via dynamic programming. Arora's algorithm generates exponentially many subproblems, but with the help of abundant memory resources, memoization tables are used to tackle them all in linear time. We present a powerful implementation of Arora's algorithm (which we affectionately name **a100**) whose performance

establishes a new benchmark concerning its capabilities.

We discuss methods for addressing the implementation challenges surrounding this algorithm. Germane to the discussion of implementation considerations is the issue of how to generate subproblems in Arora’s recursive, divide and conquer strategy. To this end we introduce a useful formalism, termed *canonical schema*, which serves as the basis for discussing how to generate the large, static tables that map parent subproblems to child subproblems. As yet we’ve not seen other implementations of Arora’s algorithm address this issue, hence we devote space to describing a practical approach for encoding canonical schema mappings by means of automatic source code generation, that is performed offline. Our discussion of canonical schema is useful as it would inform future efforts to build upon the scalability of the present implementation.

Our implementation of Arora’s algorithm shows some promising signs of being able to mature into a more robust TSP approximation tool, one that could potentially serve to generate candidate tours that other methods like Lin-Kernighan could then refine into optimal or nearly optimal tours quickly. We provide results for our solver under three different configuration scenarios, in order to demonstrate that it has the ability either to provide high quality solutions, i.e. tour length is close to optimal or is optimal, or to provide approximate solutions to large instances, i.e. well above 100,000 data points, in a feasible amount of time. With respect to presently known implementations of Arora’s algorithm, our implementation yields shorter run times, higher solution quality, and most notably the ability to generate approximate tours for much larger TSP instances.

1.3 Remaining Chapters

In chapter 2 we survey related work in the areas of cache replacement algorithms and the traveling salesman problem. We establish some conventions and useful terminology that renders the discussion of empirical results

and conjectures in later chapters more succinct. In particular we adopt the conventions presented in [Garey and Johnson, 1979] with regards to how to specify the solution quality of an approximation algorithm with respect to a known optimal solution.

In chapter 3 we present a novel cache replacement algorithm, called Far From Recently Used (FFRU), and describe its performance in terms of recency and load factor guarantees. We define the configuration parameters of FFRU and discuss constraints on choosing values for these as well as performance tradeoffs involved. We also present empirical evidence which substantiates the run time analysis of insertions into an FFRU cache. The three caching algorithm variants that we present – FFRI, FFRU Absolute, FFRU Relative – each have certain special cases based on the selection of their configuration parameters. We break down the run time and memory analysis of these variants according to these various parameter selection cases.

In chapter 4 we present evidence that FFRU exhibits recency and load factor performance as predicted in the preceding chapter. We explain techniques used in ensuring the correct functioning of our implementation, including a framework for determining the ages of evicted items that makes use of *fixed operation sequences*. We use these to simulate various workloads on LRU and FFRU caching. Using an offline analysis of logs detailing the evictions that occurred during the execution of fixed operation sequences, we are able to verify the minimum eviction age for each caching algorithm that we studied. The second method by which we verify the performance of FFRU is by solving recursive, memoized optimization problems. We investigate several variants of the Fibonacci algorithm, the KMP algorithm, the LCS algorithm, and Arora’s algorithm using our implementation, giving notable results arising from having measured their cache performance using LRU and FFRU. We also investigated the cache miss performance of the Needleman-Wunsch sequence alignment algorithm and algorithms for the Levenshtein distance (edit distance), although we omit the reporting of empirical results for these two problems, because their access patterns were identical to those of one of our reformulations of

the LCS algorithm. In this chapter we establish that for each problem studied we observe for a wide variety of input types, problem sizes, and cache sizes, that memoized problems incur cache misses with FFRU that are less than or equal to those incurred by LRU, when the two algorithms are equipped with equivalent recency guarantee configurations.

In chapter 5 we present our implementation of Arora’s PTAS for the Euclidean TSP. We discuss relevant design decisions encountered while implementing this algorithm, which includes the challenge of automatically generating the large and complicated schema mapping tables that are needed to resolve individual subproblems. We present empirical results in terms of solution quality and run time on the performance of our implementation when memoized using an amount of cache memory greater than the number of subproblems. In these experiments having plenty of memory allows us to focus on the run time performance and solution quality of our implementation by itself, i.e. without the influence of a memory management policy. Our implementation of Arora’s algorithm provides the ability to fine tune the ability to search for high quality TSP solutions, or instead to produce crude approximations very quickly. To demonstrate this variability of performance, we present empirical results arising from three ways of configuring our implementation. When solution quality is the priority, our solver yields solutions that are on average within 3% of optimality. When run time speed is the priority, our solver is capable of yielding approximate solutions to large instances. The largest instances our solver tackled during experimentation was 100,000 points in around 30 minutes to within 15% of optimality.

In chapter 6 we draw some conclusions from the work conducted in this project and suggest some future extensions to this research.

CHAPTER 2

BACKGROUND

Savoir pour prévoir, afin de pouvoir.
(From knowledge comes prediction
and from prediction action.)

A. Comte

In this chapter we provide references to previous results and some seminal work about cache replacement algorithms and the traveling salesman problem. An exhaustive survey of either topic would exceed the scope of this chapter. We restrict our attention to results that are most relevant to the work done in this thesis. We also present some notation and conventions that will be used in later chapters to facilitate the reporting of our empirical results.

2.1 Cache Replacement Policies

Cache replacement policies attempt to manage memory resources, with the overreaching goal of providing convenient access to important previously stored items while avoiding storing items which won't be needed in the near future. Applications store the results of computations in caches because of the assumption that the results of those computations will need to be obtained more than once. The applications of cache replacement policies can be found

across a wide variety of technology settings, including database applications (see for example [O’Neil et al., 1993] on disk buffering), telecommunications (for proxy servers and DNS implications consult [Forouzan, 2009], which discusses time to live purging of DNS name resolution using caching, and see [Taher et al., 2018] for some recent work on caches in named data networks), operating systems, compilers, and many other domains. Since many research initiatives in caching algorithms are aimed at reducing the number of page faults in operating systems and database contexts, several of the approaches among those surveyed below are particularly germane to *page replacement algorithms*. Although cache replacement strategies and page replacement strategies share the goal of making the best possible eviction choice, the latter often take into account whether or not a cached page has been modified, as an indicator of its popularity. However, there are many contexts, notably in compilers and in memoized software functions, when cached values are invariant, since they represent the result of some deterministic computation. In such cases any strategy based on checking whether cached items have been modified is liable to lose some of its advantages.

2.1.1 Clairvoyant Caching and Bélády’s Anomaly

Given the goal of maximizing the likelihood that a given item is in the cache when requested, the most efficient eviction policy would be to evict the item whose future request time is *farthest in the future*. László Bélády presents the *clairvoyant algorithm* in [Belady, 1966], also known as Bélády’s algorithm, or the optimal cache replacement policy. Since it is generally impossible to know in advance the request times of items for an application into the future, Bélády’s algorithm is considered impossible to implement for online performance. However, it is still considered useful as a benchmark when post-facto, i.e. offline, comparisons of cache eviction policies are able to be carried out. Bélády’s anomaly [Belady et al., 1969] refers to a phenomenon exhibited by certain page replacement algorithms, notable FIFO, in which the number of

page frames, i.e. the size of the cache, can result in an increase in page faults.

2.1.2 Random

The simplest solution to the problem of selecting an item for eviction is not to make the choice at all. Using linear feedback shift registers has been used to implement random cache replacement, although in one study the reported performance was 22% worse than LRU on average [Al-Zoubi et al., 2004]. See [Baroughi and Naderi, 2020] for a recent performance comparison of random replacement to LRU and MRU.

2.1.3 FIFO, Second Chance, and Clock Variants

The FIFO cache replacement strategy places cached items into a simple queue, and replaces the item that has been in memory the longest. Although simple to implement and fast, this strategy requires an additional $O(n)$ memory overhead to maintain the queue data structure. Furthermore, repeated accesses to a given item do not alter its position in the queue, making it a poor choice for applications where certain cached items are more popular than others. FIFO cache replacement was shown to have miss ratios that were 12-20% higher than LRU on average [Smith and Goodman, 1983]. And it has been shown to suffer from Bélády’s anomaly in [Belady, 1966].

The Second Chance page replacement strategy is a modification of FIFO that was designed for virtual memory. It checks the oldest item in a FIFO queue. If that item hasn’t been referenced it is evicted, otherwise the item is moved to the front of the line and is set as unreferenced. If everything in the FIFO queue has been referenced since the last clock tick¹, then the oldest item is evicted, namely the one that had been moved to the front of the line and whose referenced bit was reset.

A slightly faster implementation of Second Chance is known as the Clock

¹Second chance and FIFO are both *page* replacement algorithms, which is why clock ticks determine how often such data structures are administrated.

replacement algorithm [Corbato, 1969], which maintains a pointer to cached items. Cached items are stored in a circular linked list. When a page fault occurs, if the item pointed to has not been referenced it is discarded, otherwise it's dirty bit is reset and the pointer is advanced to the next item. Variants on the general clock approach include Gclock [Smith, 1978], Clock-pro [Jiang et al., 2005], and working set [Carr and Hennessy, 1981].

2.1.4 LRU

Least recently used caching always evicts the item whose most recent usage time is the farthest in the past with respect to the other items that are in the table. As such conceptually it can be considered to perform the same strategy as Bélády's algorithm, but with the direction of time reversed. Given that many applications exhibit *locality of reference* it is widely believed that LRU is well suited to most problems. Sleator and Tarjan showed that the number of cache misses incurred by LRU can be bounded by a constant factor with respect to those incurred by an offline determination of Bélády's algorithm [Sleator and Tarjan, 1985]. Some useful discussions of LRU can be found in [Kleinberg and Tardos, 2006]. When implementing LRU, one could opt to avoid memory overhead by using a priority queue, however in practice this isn't feasible, as insertions and retrievals suffer amortized logarithmic slow down. Another more popular approach is to store items in a hash table and keep their ranks stored in a linked list, which preserves the amortized constant time insertion and update costs, but incurs a $O(n)$ memory overhead cost to maintain the doubly linked list of ranks. LRU has been shown to be free from Belady's Anomaly [Belady et al., 1969]. In [Panagakis et al., 2008] it was shown LRU is insensitive to the length of local correlations in access patterns, provided that the memory of such access pattern correlations was smaller than that of the cache. This result was used to corroborate LRU's good suitability for web caching applications.

2.1.5 LRU approximations

PLRU Variants

In **bit-PLRU** using only one bit per cached item, the cache can be partitioned into those most recently used and not most recently used, with the latter set available for eviction. On access the MRU bit for an item is set, placing it into the set of items most recently used. When a cache miss occurs the first non-MRU item found is evicted. In comparison studies it was shown to sometimes perform slightly better than LRU in some cases, and given its efficient operation it has found use in embedded systems. See [So and Rechtschaffen, 1988] and [Al-Zoubi et al., 2004] for further discussion.

Tree-PLRU (PLRUt) uses a binary tree to approximate LRU. In an N -way associative cache it requires $N - 1$ bits. The tree bits encode the paths to the leaves corresponding to each cached item. On a cache hit the bits on the path leading to the chosen item are flipped in order to protect it from subsequent eviction. On a cache miss the item pointed to in the PSLRU tree is evicted. It was shown in [Al-Zoubi et al., 2004] that this strategy is only slightly worse than LRU. The **Modified Pseudo LRU (MPLRU)** policy [Ghasemzadeh et al., 2006] is another variant using a tree-based approach and has given similar performance to PLRUt. The **SIDE** algorithm also makes use of $O(N \log K)$ bits for counters, where N is the cache size and K is the degree of set associativity [Deville, 1990]. The performance of the SIDE algorithm is up to 20% worse than LRU.

LRU/K

The LRU/K algorithm [O’Neil et al., 1993] keeps track of the K^{th} most recent access time for every item in the cache. When an eviction must occur, the item whose K^{th} most recent access time is longer ago than any other item’s K^{th} most recent access time, is the item to be evicted. The mentality behind this approach is that popular items have frequent requests and should be kept in preference over items accessed more intermittently. It was originally devised

as a strategy to keep popular database pages in memory. According to this terminology conventional LRU would be termed LRU/1, since it adjusts item ranks based on the time of the last access. From their empirical results they found that LRU/2 most exploits the advantages of their method. It requires two delicate tuning parameters, *Correlated Reference Period* which determines how far apart in time repeated references to an item are to count as separate instances and *Retained Information Period* which is how long after eviction an item’s reference history is retained. Empirical evidence and discussion found on LRU/ K would suggest that it’s particularly suited to database settings, especially given that it’s tuning parameters are based on access times.

2Q and Variants

The 2Q buffer management algorithm [Johnson and Shasha, 1994] builds on ideas of LRU/ K by maintaining a two tiered approach. Like LRU, LRU/ K also suffers from some overhead needed to implement the priority queue that ranks items in terms of their latest reference time. The main buffer for cached items is maintained as an LRU priority queue. When an item therein is re-referenced it is deemed “hot” and is placed into the second buffer, which is implemented as a FIFO queue. Once in the second buffer, if an item is re-referenced it gets moved back into the first buffer. One advantage of 2Q over LRU/ K is that it offers amortized constant time insertions and updates, although it’s memory overhead remains a potentially detracting factor, just as with LRU. 2Q was shown in empirical studies to outperform LRU and Gclock and to have comparable performance to LRU/2 using Zipfian distributed inputs. The *low inter-reference recency set* [Jiang and Zhang, 2002] builds on ideas of 2Q and shows slight improvements over LRU in certain scenarios.

2.1.6 Adaptive Replacement Cache

The Adaptive Replacement Cache (ARC) shows improved performance over LRU for certain applications by keeping track of both the most recently

used and most frequently used items [Megiddo and Modha, 2003]. It also tracks the eviction history for both sets, the sizes of which are smaller than that of the cache and must be configured. See [Megiddo and Modha, 2004] for a comparison of ARC with several other caching algorithms. For a good survey of different categories of caching algorithm see [Zebchuk et al., 2008].

2.2 Traveling Salesman Problem

2.2.1 Variations of the TSP

The traveling salesman problem (TSP) is a famous and classic problem that has fascinated researchers for over a hundred and fifty years. It has generated enormous attention from algorithms experts seeking a testing ground for their optimization strategies. The TSP has been formulated in numerous ways, in various numbers of dimensions, and using different types of constraints. Some of the variants that have been formulated and studied include: MAX TSP, bottleneck TSP, TSP with multiple visits (TSPM), Messenger problem, Clustered TSP, Generalized TSP (GTSP), m -salesmen TSP, Time dependent TSP, Period TSP, Delivery man problem, Black and white TSP, Angle TSP, Film-copy Deliverer problem, Selective TSP, Resource constrained TSP, Serdyukov TSP, Ordered Cluster TSP, Precedence constrained TSP, k -Peripatetic Salesman Problem, Covering Salesman Problem, TSP with time windows, Moving target TSP, Remote TSP, Distances one and two, and asymmetrical distance matrices. There are many more. For a comprehensive treatment of some of the most important variations of the TSP, consult [Punnen, 2007] and [Worboys, 1986].

2.2.2 Applications of the TSP

Although solving the TSP has considerable theoretical importance, this problem also has many concrete and practical applications. Apart from the obvious task of routing a salesman through a given number of cities cover-

ing the shortest distance, there are applications of the Traveling Salesman Problem to be found in biology, astronomy, health care, construction, industrial manufacturing, and the arts. Some of the applications of the TSP include: Machine Scheduling, Cellular Manufacturing, Arc Routing Problems, Frequency Assignment, Structuring of Matrices, Data analysis in psychology, X-Ray crystallography, Overhauling gas turbine engines, Warehouse order-picking problems, and Wall paper cutting. See [Cook, 2011] for a fascinating discussion of many of the most common applications of the TSP.

2.2.3 Defining the TSP

The Traveling Salesman Problem, in its simplest form, can be stated as follows: given n nodes in some space having k dimensions, determine a closed path of minimal cost that visits each node exactly once. Costs are typically given by some distance value $d_{(i,j)}$ for each pair (i, j) of distinct nodes, hence the cost of the minimal length salesman tour would be the sum $\sum d_{\sigma(k)}$, of the distances along the tour. Since the distances are known, or at least are easy to determine, then solving the problem amounts to determining the function $\sigma : \{1..n\} \rightarrow \{1..n\}$ which gives the ordering of nodes to visit corresponding to such a path.

The Metric TSP can be defined as follows. Given a graph with weights assigned to the edges, find a minimum weight tour that visits each vertex at least once. This is called metric TSP because it is equivalent to solving the original TSP (without revisiting vertices) on the “metric completion” of the graph. By metric completion, we mean that we put an edge between every pair of nodes in the graph with length equal to the length of the shortest path between them. The shortest path function on a connected graph forms a metric. A metric is an assignment of length to every pair of nodes, u , v , and w , such that $d(u, y) \geq 0$, $d(u, v) = 0$ iff $u = v$, $d(u, v) = d(v, u)$, $d(u, v) \leq d(u, w) + d(w, v)$.

The Euclidean TSP is a special case of the Metric TSP and can be defined

as follows. Given n points in the d -dimensional Euclidean metric space (for some fixed d), find a minimum length tour that visits them all. The points are typically taken from \mathfrak{R}^2 and the metric used is typically the l_2 norm, although higher dimensions may be used (i.e. \mathfrak{R}^d where $d > 2$) and other norms are possible (i.e. l_p for other values of p).

2.2.4 Complexity of the TSP

Although it is simple to define, the Traveling Salesman Problem is considerably difficult to solve. Papadimitriou showed that exact optimization is NP-Hard in [Papadimitriou, 1977]. It has been shown that approximating the optimum within any constant factor is also NP-Hard [Sahni and Gonzalez, 1976], [Vazirani, 2001]. Trevisan has shown that the Euclidean TSP becomes MAX-SNP-hard in $O(\log n)$ dimensions [Trevisan, 1997].

2.2.5 Heuristic Approaches to the TSP

There exist a number of solution heuristics for the TSP. Although, as opposed to algorithms, heuristics provide no guarantees on solution quality or run time bounds. Perhaps the simplest heuristic for finding a short tour of a set of points is through trial and error, i.e. generating some number of different tours, calculating the length of each, and keeping the shortest one found. However, since the space of possible tours grows as the factorial of the number of cities, any trial and error approach that stops after having considered a constant and fixed number of tour candidates will have progressively decreasing solution quality performance as the number of cities grows.

For this reason heuristics for solving the TSP often perform what is known as hill-climbing, wherein the vast space of solutions is viewed as a landscape one can traverse via local search. One way to perform local search when solving the TSP is to perform k -opt moves. A k -opt move selects k edges in the candidate TSP tour and looks at all possible ways of reconnecting the vertices directly accessed by these edges that still results in a valid tour. If a shorter

tour is formed by reconnecting these vertices differently, then this new tour becomes the current candidate tour, otherwise that given k -opt move leaves the candidate tour unchanged. This process of considering “neighboring tours” continues until some stopping criteria has been met, which is typically either a distance threshold, a predefined number of trials, or a predefined run time threshold. There have been many strategies devised for solving the TSP that fall under the rubric of local search heuristics. For a good survey of some of the most popular probabilistic techniques see [Greco, 2008], which include genetic algorithms, bio-inspired algorithms, Ant Colony Optimization (ACO), Evolutionary Computing, Particle Swarm Optimization (PSO), and Neural Networks. Dorigo gives an in depth survey and history of Ant Colony Optimization in [Dorigo and Stützle, 2004], and he places ACO in the context of other probabilistic approaches. An advantage to these techniques is that with the proper tuning of their algorithmic parameters, they are often able to solve many problem instances quickly. It remains challenging, however, to establish their algorithmic complexity and provide tight and consistent solution quality bounds, since they are sometimes subject to getting trapped in suboptimal minima.

2.2.6 Algorithmic Approaches to the TSP

In the category of algorithmic approaches to the TSP, the Nearest Neighbor algorithm is one of the simplest approaches. It starts at a given city and at every step adds the nearest city to the most recently added city. As such it is a greedy algorithm. The problem with this approach is that the final city is often very far from the original city. As a result this algorithm rarely finds a shortest-possible solution, and it is easy for an adversary to construct a TSP instance for which the nearest neighbor algorithm chooses a tour that is much longer than the optimal tour. The worst-case guarantee of nearest neighbor is $1 + \log(n)/2$ times the cost of an optimal tour for an n -city TSP.

Another greedy approach is to grow separate subpaths by adding the short-

est edge to one of the subpaths at every step. Like with nearest neighbor, one can begin with any city. This strategy allows for slight variations on how successive cities are chosen. However, despite numerous attempts to improve its performance in handling certain particularly hard instances, the worst-case guarantee of this algorithm is $1/2 + \log(n)/2$ times the optimal tour length for instances that satisfy the triangle inequality. Thus, it is a slight improvement over nearest neighbor.

There are a variety of different insertion algorithms, i.e. algorithms that grow a tour by starting with a complete tour of just three points and adding points using some selection policy. Some popular selection policies are cheapest, nearest, farthest, and random. Using this mechanism the tour grows until it includes all of the instance points. Cheapest and nearest insertion have been shown to produce tours no worse than twice the length of optimal solutions when the triangle inequality holds.

Another well-known algorithm for solving the TSP is to determine the minimum spanning tree (MST) of the instance points, then perform a depth-first-search traversal of the tree. This algorithm guarantees that the resulting tour is never longer than twice the length of the MST, and the length of the MST is always less than the length of the optimal tour.

In addition to tour growing algorithms, there are also tour improvement algorithms. A notable family of such algorithms is due to Shen Lin & Brian Kernighan [Lin and Kernighan, 1973], and it starts by selecting an initial tour, then it makes a number of adjustments, known as k -opt moves. A k -opt move considers k edges at a time and, for a given set of k edges determines the best way to connect the vertices. Using k -opt moves is the same technique for iteratively improving tours that was discussed above in the context of heuristic solution techniques, except that in this case all possible k -opt moves are attempted until no further improvements have been found. The Lin-Kernighan and the family of algorithms later developed due to Lin, Kernighan, & Helsgaun, all rely on a fundamental result, which is that an optimal tour will never cross itself [Flood, 1956]. Implementations have been developed for $k = 2$ up

to $k = 10$, and one of the most recent ones was able to solve a 24,978-city instance to optimality.

Finally, the algorithm with the best solution quality guarantee is due to Christofides, who designed an approximation algorithm that runs in polynomial time and for every instance of the metric TSP computes a tour of cost at most $3/2$ times the optimum [Christofides, 1976].

Christofides' algorithm solves the TSP by creating a minimum spanning tree of the input points. It then finds a perfect matching with minimum weight in the complete graph over the vertices that have odd degree. It combines the edges from the perfect matching with those of the minimum spanning tree to form a multigraph. It then forms an Eulerian circuit in that multigraph. Finally, the algorithm finds the salesman tour of the original points by skipping visited nodes in the Eulerian circuit. For a good comparison of empirical results arising from different approximation algorithms to the TSP, see [He and Xiang, 2017].

2.2.7 Concorde

Perhaps the most successful of all known implemented TSP solution methods, indeed the one adopted by the Concorde implementation (considered by many to be the current state-of-the-art of TSP solvers), is linear programming. There exists an enormous amount of research concerning these methods. For a good survey of the prominent LP methods, consult [Cook, 2011]. The Concorde TSP Solver is a software implementation, written in C, using an ensemble of linear programming strategies aimed at finding upper and lower bounds on the optimal tour lengths of TSP instances. It is able to generate tours using those bounds, and has been able to find the optimal tour to the entire TSPLIB collection. The solver has also been applied to problems like vehicle routing, protein function prediction, gene mapping, others having computational difficulty germane to the TSP. The Concorde solver has indeed produced formidable results in solving the traveling salesman problem. Nevertheless the

parameters dictating its operation are numerous, and for certain configurations of Concorde there are TSP instances that have shown themselves to be particularly difficult or not possible to be solved. Despite producing remarkable empirical results, this solver exhibits considerable variability in run times, owing to the fact that certain TSP instances are much harder than others. At present we are unaware of any proven guarantees on the run time needed by Concorde to produce a tour having a degree of optimality under some chosen threshold. An in depth treatment of its capabilities and the concepts underlying its construction can be found in [Applegate et al., 2006].

2.2.8 Arora’s PTAS

Arora showed a non-deterministic $(1 + \epsilon)$ -approximation with $n^{O(1/\epsilon)}$ running time, which he later improved to $n(\log n)^{O(1/\epsilon)}$, for the geometric TSP [Arora, 1998]. This algorithm computes a salesman tour of length no more than $(1 + \epsilon)$ times the optimal salesman tour length with probability $\frac{1}{2}$, independent of ϵ . The run times depend on ϵ , but for each fixed ϵ the run time is polynomial in the input length. Arora’s algorithm can be made to work deterministically, although this increases the run time by a factor of $n^{O(1)}$ with respect to the non-deterministic approach². Arora’s seminal result proves that a polynomial run time algorithm can generate tours that can be made arbitrarily close to the optimal length, although his algorithm exhibits a crucial aspect of many approximation techniques, which is that run time and memory requirements increase dramatically as ϵ decreases. His techniques also extend other problems, such as Minimum Steiner Tree, k -TSP, k -MST, and several other Euclidean optimization problems. Mitchell shortly thereafter published a similar algorithm in [Mitchell, 1999]. Although unlike Arora’s algorithm, Mitchell’s algorithm does not scale to work with metric spaces having higher dimensions and does not extend to other Euclidean problems.

²Note that the implementation we present in chapter 3 uses the non-deterministic approach, which accounts for the variability in solution quality encountered in our empirical results.

Survey of Known Implementations

Much research concerning Arora’s algorithm has remained at the theoretical level, and few attempts have been made to implement Arora’s algorithm into a robust, practical tool. We are aware of two such cases. Zhao et al. [Zhao Weizhong and Daming, 2007] produced a Java implementation and conducted a cursory study of the effect of varying a few of the algorithm’s main parameters. However, it remains unclear what capabilities their solver has, because empirical results are given for only three of the smallest of problem instances found in the TSPLIB, namely *ulysses16*, *ulysses22*, and *eil51*. They varied four of the parameters associated with Arora’s algorithm, but obtained results ranging from 2% to 40% of optimality, and their run times vary greatly.

Rodeker et al. [Rodeker et al., 2009] produced a C++ implementation of Arora’s algorithm and compared the performance of their implementation with that of the Concorde solver, as well as several well-known algorithms, such as Greedy, Nearest Neighbor, and Lin-Kernighan. They appear to have managed the increasing memory requirements inherent in Arora’s dynamic programming algorithm, although the largest instances their solver appears to be able to handle are around 200 points, with high variability in solution quality as measured in percentage of optimality, and high variability in run times.

2.3 Conventions & Notation

2.3.1 Pseudocode Listings

In all pseudocode algorithm listings herein, unless otherwise stated we assume that sequences, lists, and arrays have indices that start with zero, as they do in contemporary programming languages. We adopt the \leftarrow symbol to denote variable assignment, and the operator $=$ to denote the test for equality. In all algorithm listings we also assume that invalid inputs, eg. array indices out of bounds, are handled properly, and we omit such error handling logic

from our exposition of the algorithms discussed herein.

2.3.2 Cache Replacement Policies

Unless otherwise stated we generally use n to refer to the problem instance size with respect to some optimization problem P , and we assume C_S refers to the cache size used with respect to some cache replacement strategy S . When the cache replacement strategy is irrelevant to the given context, the subscript is omitted and we simply use C to represent the cache size.

Definition 2.1

- A **cache miss** happens when a memoized procedure consults the cache and does not find the desired key therein.
- $M_S(P)$ denotes the number of cache misses encountered when solving problem P using cache replacement strategy S . The subscript is omitted if the caching strategy is irrelevant to the conjecture in question.
- We let $\hat{C}(P)$ denote the **critical cache size** for problem P , which is the smallest cache size for which the only cache misses that occur when solving P are those not due to eviction of previously inserted keys.
- We let $\hat{M}(P)$ denote the **critical cache misses**, which is the minimum number of cache misses that arise when solving P . It follows that when $C(P) \geq \hat{C}(P)$, then $M_S(P) = \hat{M}(P)$.

An intuitive perspective on the notion of the critical cache misses is that it is a measure of how many total subproblems are encountered when solving P . As such it can be a convenient metric when attempting to establish the run time complexity of a memoized procedure, given that the base case is usually completed in $O(1)$ time. Likewise the critical cache size lets us know how much memory would be needed to solve a problem such that every subproblem is able to be stored in memory. This is why these two metrics omit mention of a cache replacement strategy, S .

Total cache misses versus cache hit ratio

When characterizing caching algorithm performance, sometimes a useful metric is to normalize the number of cache misses occurred by taking into account how many cache hits occurred as well. This is called the *cache miss ratio*. In the results we present, we are sometimes interested in the total number of cache misses as a function of cache size, and sometimes interested in the number of cache misses as a function of *problem size*. This is because in addition to showing that our algorithms produce fewer cache misses than LRU, we also want to demonstrate the effect of modifying the memoized functions themselves in order to achieve improved cache miss performance.

2.3.3 Approximation Algorithms

In this section we intend to establish the terminology for evaluating the results of approximation algorithms, the five categories in Garey & Johnson [Garey and Johnson, 1979] and specify what is the case for TSP with and without triangular inequality. Five categories: $R_{min}(\Pi) = 1$ with difference guarantee; $R_{min}(\Pi) = 1$ fully polynomial scheme; $R_{min}(\Pi) = 1$ polynomial scheme; $1 < R_{min}(\Pi) < \infty$; $R_{min}(\Pi) = \infty$.

In order to discuss the results of approximation methods³, a measure of solution quality is needed. For our purposes, solution quality is measured by determining the difference in length between the obtained solution and the optimal solution. We use $OPT(I)$ to denote the quality associated with an optimal solution for a given problem, and $ALG(I)$ to denote the quality produced by the approximation algorithm under consideration. Ideally we would like to provide a guarantee that $ALG(I) \leq \alpha OPT(I)$ where $\alpha \geq 1$ and α is as close to unity as possible. The I above denotes any instance of the optimization problem being solved. In this context, we refer to α as the approximation factor. Often approximation factors are stated as percentages. For example,

³See [Hochbaum and Hochbaum, 1997] for a comprehensive survey of approximation algorithms for optimization problems.

if we solve a problem instance for which $ALG(I) = \alpha OPT(I)$ where $\alpha = 1.05$ then we say we've solved instance to within 5% of optimality⁴. Approximation factors may depend only on some constant factor inherent in the approximation algorithm being used. In this case we represent the approximation factor as $\alpha = (1 + \epsilon)$ where $\epsilon > 0$. In the case of a $(1 + \epsilon)$ -approximation algorithm, we call such an algorithm a polynomial time approximation scheme, PTAS. If an algorithm is polynomial in n and $1/\epsilon$ then we call such an algorithm a fully polynomial time approximation scheme, FPTAS. This is one of the best scenarios for approximation algorithms. Other less favorable situations exist where the approximation factor, α , may be some function of n , where n is the size of the instance. Note that Arora's algorithm is a PTAS and not a FPTAS, since it is not polynomial in $1/\epsilon$.

⁴In the research surrounding strategies for solving the TSP, one finds both styles of expressing the degree of optimality obtained. We opted for the style found in Gary & Johnson, since it seems to be present across discussions of a variety of problem types.

CHAPTER 3

FAR FROM RECENTLY USED (FFRU) CACHE REPLACEMENT

Veritatis simplex oratio est. (The
language of truth is simple.)

Seneca — *Epistulae morales ad
Lucilium*

In this chapter we present a novel caching algorithm and analyze its performance. Since our caching algorithm is built as an extension of cuckoo hashing, we present the results from experiments done on cuckoo hash tables in order to quantify the average number of probes per insertion. These results inform our run time analysis of insertions with FFRI & FFRU.

3.1 Far From Recently Inserted (FFRI)

In this section we discuss the basic framework for FFRI and FFRU caching and present a caching algorithm that always evicts a not-recently-inserted item. In FFRI caching, an item's timestamp is set when it is inserted and is

not updated on subsequent references. Items and their timestamps are stored in a cuckoo hash table. We define the following parameters for FFRI and FFRU caching.

Definition 3.1

- N is the table size, i.e. the total number of memory locations able to hold an item. This is not the same as capacity (see **load factor** below).
- k is the number of cuckoo tables allocated. Note that N is some multiple of k .
- $\kappa \leq N$ is the number of distinct timestamps. Every active cache item has a timestamp $t \in \{0, \dots, \kappa - 1\}$.
- $1 < d < \kappa$ is the number of “recent” timestamps.
- a denotes the timestamp update threshold factor, such that aN is the maximum items per timestamp. We require aN to be an integer.
- S_t is the number of items having timestamp t , i.e. we maintain item counts for all timestamps.
- α is the **load factor**, i.e. the ratio of occupied table slots to the total number of table slots; $(1 - \alpha)N$ table entries are unoccupied.
- ϕ is the **freshness factor**; the ϕN most recently inserted data items are still in the table.

The memory needed for administering the timestamps is:

$$N \log_2(\kappa) + c\kappa \tag{3.1}$$

Where c is the size of integers used for each of S_t . We describe how FFRI caching works by discussing its behavior for different choices of d .

3.1.1 Cuckoo Clock Caching

We begin with the simplest scenario, namely when $d = \kappa - 1$, the highest valid value for d . In this case the assignment of successive timestamps is sequential mod κ . At time t , locations with timestamp $(t + 1) \bmod \kappa$ are

considered to be available, in that new items can be stored there, possibly evicting an item if the given table slot is already occupied. When S_t reaches aN , the time changes to $(t + 1) \bmod \kappa$. This new timestamp has the fewest entries.

3.1.2 Drunken Cuckoo Clock Caching

Another special case is when $d = 2$, it's lowest valid value. In this case the choice of which successive timestamp is to be considered the current one is not sequential. We must now keep track of the previous time, s , as well as the current time, t . Locations with any timestamp other than s or t are considered available. When S_t reaches aN , we let $s = t$ and we assign t to be the timestamp with the fewest entries. Note that this is the same as d -Drunken Cuckoo Caching with $d = 2$.

3.1.3 d -Drunken Cuckoo Caching

This is the general case, namely when $2 \leq d \leq \kappa - 1$. The assignment of successive timestamps is not sequential. We maintain a queue of the d most recent timestamps, which includes the current timestamp t . Locations with any timestamps other than the d most recent are considered available. When S_t reaches aN , we discard the oldest timestamp in the queue, and we assign t to be the timestamp with the fewest entries.

3.2 Far From Recently Used

3.2.1 FFRU Absolute

In this section we explain how to modify FFRI to evict an item that was not recently used, where **recency** is measured by number of operations since the item was used (call this the **time-recency**). An item's timestamp must be updated whenever it is referenced.

In order for FFRI/FFRU to work correctly it is necessary that each recent timestamp be assigned to exactly N/κ items. When an item in the table with timestamp i is referenced and its timestamp is updated, S_i decreases to $N/\kappa - 1$. If i is one of the recent timestamps, we rectify this situation by choosing a pseudorandom item with a non-recent timestamp and changing its timestamp to i . To choose the pseudorandom item, we can employ a strategy inspired by linear probing as follows. Suppose that the item referenced has key x and it was hashed using hash function f_j into cuckoo hash table j , i.e., the item is in position $f_j(x)$ in table j . Then we look in position $f_j(x)$ in all of the tables except j for a cell that is currently available. If we find one, then we change its key to i . If we don't find one then we look in position $f_j(x) + 1 \bmod N/k$, and so on, as with linear probing. Note that the value of $f_j(x)$ does not need to be recomputed.

3.2.2 FFRU Relative

Given the definition of FFRU Absolute just discussed, suppose hypothetically that a single item was referenced many times in a row. That would make all other items non-recent and therefore fair game for eviction. In some situations we would prefer to preserve a number of most recent items regardless of how many operations occurred after they were referenced. We call this the **item-recency**.

Suppose that we are preserving the d most recent timestamps. This is the most general case. If an item with one of the $\lceil d/2 \rceil$ most recent timestamps is referenced then we don't update its timestamp. If an item with one of the other $\lfloor d/2 \rfloor$ of the d most recent timestamps is referenced, then we do update its timestamp. The resulting item-recency and time-recency are $\frac{1}{2}$ of the recency obtained by FFRU Absolute.

Table 3.1: Constraints on κ , d , and a .

Variant	FFRI and FFRU (Absolute & Relative)
Cuckoo Clock Caching	$1/\kappa < a < 1/(\kappa - 1)$
Drunken Cuckoo Clock Caching	$1/\kappa < a < 1/2$
d -Drunken Cuckoo Caching	$1/\kappa < a < 1/d$

Table 3.2: Performance guarantees.

Variant	FFRI & FFRU Absolute	FFRU Relative
Cuckoo Clock Caching	$\alpha \leq (\kappa - 1)a$ $\phi \geq (\kappa - 2)(\kappa a - 1)$	$\alpha \leq (\kappa - 1)a$ $\phi \geq \frac{(\kappa-2)(\kappa a-1)}{2}$
Drunken Cuckoo Clock Caching	$\alpha \leq 2a$ $\phi \geq \frac{\kappa a - 1}{\kappa - 2}$	$\alpha \leq 2a$ $\phi \geq \frac{\kappa a - 1}{2(\kappa - 2)}$
d -Drunken Cuckoo Caching	$\alpha \leq da$ $\phi \geq \frac{(d-1)(\kappa a - 1)}{\kappa - d}$	$\alpha \leq da$ $\phi \geq \frac{(d-1)(\kappa a - 1)}{2(\kappa - d)}$

3.3 Parameter Constraints & Performance Guarantees

Table 3.1 lists the parameter constraints that all versions and variants thereof must satisfy, and table 3.2 lists the performance guarantees provided by each version and variant. We must choose N , κ , d , and a such that $\phi > 0$, otherwise undefined behavior results. When $d = \kappa - 1$, the largest cache size for which $\phi \leq 0$ is $N = \kappa d$. When $N \leq \kappa d$ and $N \bmod \kappa = 0$ then $\phi = 0$, because all timestamp counts become equal when the cache is full. For certain cache sizes when $N < \kappa d$, we sometimes find $\phi < 0$. The smallest such case is $N = 4$, $\kappa = 3$, $d = 2$, which yields $\phi N = -1$. Under this configuration, $aN = 1$, and a point is reached when there are three items in the cache and all timestamp counts are 1. When the fourth item is inserted, if a collision occurs the cache

eviction policy tries to find a candidate for eviction, despite the cache not being full. No eviction should be warranted, but it attempts eviction anyway, because the current timestamp has count equal to aN . However, all the other timestamp counts equal aN , so the eviction attempt has undefined behavior. Therefore, the freshness guarantees risk being compromised again and the load factor suffers as well. Therefore when $N < \kappa d$ and $a\kappa < 1$ then $\phi < 0$, because all timestamps become equal but the cache doesn't reach capacity.

3.4 Run Time Analysis

In this section we examine the run time analysis of FFRI & FFRU. The underlying storage data structure used with FFRI & FFRU is cuckoo hashing. Cuckoo hashing was originally presented in [Pagh and Rodler, 2004]. The storage for a k -ary cuckoo hash table having size N is divided into k separate tables. For each table $j \in \{1, \dots, k\}$ the hash function, $f_j(x) \bmod N/k$, determines the position that key x would occupy if placed in table j . We denote the position of key x in table j by

$$p_j(x) = f_j(x) \bmod N/k \quad (3.2)$$

On insertion key x is placed into one of the k tables. If none of the k positions $p_j(x)$ is available, then we choose one of these positions and replace the existing key, x' , with key x . The displaced item, key x' , is placed into one of the other $k-1$ tables, in position $p_{j'}(x')$. If none of those positions is available the process continues, thereby causing a series of item displacements until every key has found a position. The number of memory accesses, i.e. probes, needed for inserting a key depends on how many item displacements, as just described, occur before every item is hashed into one of its k possible positions. The advantage of this insertion strategy is that on retrieval of key x , it is guaranteed to be found in exactly one of the $j \leq k$ positions $p_j(x)$. In other words no more than k probes are ever required to locate an item (or to determine that it is not in the table).

The method just described used by cuckoo hashing for determining a table position is a type of closed hashing. Linear probe hashing, when implemented using closed hashing, is known to require on average $1/(1 - \alpha)$ probes per insertion, where α (the load factor) is the fraction of table full [Cormen et al., 2009]. When analyzing the insertion time of k -ary cuckoo hash tables, [Fotakis et al., 2003] determined using empirical evidence that as k increases, there is a threshold load factor, α , at which the average number of probes per insertion begins to increase quickly.

In order to determine how long insertions take using cuckoo hashing, we conducted two experiments inspired by [Fotakis et al., 2003]. We wish to analyze the number of probes required to insert an item into a k -ary cuckoo hash table. In the following sections we describe two procedures which we use to generate empirical data about the average number of probes per insertion.

3.4.1 Average Probes per Insertion into Cuckoo Hash Tables with Deletions

Algorithm 1 describes the procedure by which we measured the average number of probes per insertion into a k -ary cuckoo hash table with using deletions. As with the previous method we accumulate the results across all trials, and we obtain the average afterwards. Similarly we initialize a new hash table before every trials, and we stop collecting probe counts above a threshold when the number of probes exceeds the cache size, N . Within the loop involving deletions followed by insertions, the load factor is $\alpha = j/N$, just as it is within the innermost loop in Algorithm 2.

Figure 3.1 shows the average number of probes per insertion when $N = 4096$, $k = 4$, $max_trials = 100$, and $max_iterations$. The asymptote at $\alpha = 97\%$ is shown, because [Fotakis et al., 2003] reported this threshold for $k = 4$. We re-scale the x-axis as $1/(0.97 - \alpha)$ and use log-log regression to determine $y = b + mx$, yielding $m \approx 0.73$, $b \approx 0.53$ (figure 3.2). We find the smallest c such that for all x in our empirical data, $c + 1/(0.97 - \alpha)$ is

Table 3.3: Number of timestamps that are inspected.

Variant	FFRI and FFRU (Absolute & Relative)
Cuckoo Clock Caching	1
Drunken Cuckoo Clock Caching	$\kappa - 2$
d -Drunken Cuckoo Caching	$\kappa - d$

Table 3.4: Average Run Time for Insertions using 4 Cuckoo Tables.

Variant	FFRI and FFRU (Absolute & Relative)
Cuckoo	
Clock Caching	$0.05 + \frac{1}{0.97 - (\kappa - 1)a}$
Drunken Cuckoo	
Clock Caching	$0.05 + \frac{1}{0.97 - 2a}$
d -Drunken	
Cuckoo Caching	$0.05 + \frac{1}{0.97 - da}$

greater than the measured average number of probes per insertion, yielding $c \approx 0.05$ (figure 3.3). Figure 3.4 shows the hypothesized bound with the original empirical data.

Observation 3.1 *When $N = 4096$, using a cuckoo hash table with $k = 4$, the average number of probes per insertion when deletions are supported is less than $0.05 + 1/(0.97 - \alpha)$.*

Table 3.3 shows the number of timestamps that must be checked every time the current timestamp must be changed, although this occurs so infrequently that it has a negligible effect on the average run time of insertions. Given the bound found above, we can now provide in table 3.4 the average run time formulae for all three versions and all three variants. For the analysis of retrievals we assume 4-ary cuckoo hash tables, which have 2.5 probes in the expected case. FFRU incurs a degradation as shown in table 3.5 due to the possibility of having to adjust timestamps as described in the preceding sections.

Table 3.5: Average Run Time for Retrievals using 4 Cuckoo Tables.

	FFRI	FFRU (Absolute & Relative)
Cuckoo		
Clock Caching	2.5	$2.5 + \frac{1}{1-(\kappa-1)a}$
Drunken Cuckoo		
Clock Caching	2.5	$2.5 + \frac{1}{1-2a}$
d -Drunken		
Cuckoo Caching	2.5	$2.5 + \frac{1}{1-da}$

Algorithm 1 Procedure to measure average # of probes per insertion using a k -ary cuckoo hash table with deletions

Input: $N, k, max_trials, max_iterations$

Output: μ , the average # of probes per insertion as a function of α

```

1: procedure AVE – PROBES( $N, k, max\_trials, max\_iterations$ )
2:   for  $j \leftarrow 0$  to  $N$  do
3:      $total\_probes[j] \leftarrow 0$ 
4:      $total\_insertions[j] \leftarrow 0$ 
5:    $max\_insertions \leftarrow N$ 
6:   while  $j < max\_insertions$  do
7:      $num\_trials \leftarrow 0$ 
8:     while  $num\_trials < max\_trials$  do
9:       Initialize a  $k$ -ary cuckoo hash table having  $N$  empty entries and  $\alpha = 0$ .
10:      Insert  $j$  random keys into the hash table.
11:       $i \leftarrow 0$ 
12:      while  $i < max\_iterations$  do
13:        Delete a randomly chosen key from the hash table.
14:        Insert a new random key into the the hash table.
15:        Measure the probes for this insertion as  $num\_probes$ .
16:         $total\_probes[j] \leftarrow total\_probes[j] + num\_probes$ 
17:         $total\_insertions[j] \leftarrow total\_insertions[j] + 1$ 
18:        if  $num\_probes > N$  and  $max\_insertions = N$  then
19:           $max\_insertions \leftarrow j$ 
20:           $i \leftarrow i + 1$ 
21:           $num\_trials \leftarrow num\_trials + 1$ 
22:         $j \leftarrow j + 1$ 
23:    $\mu \leftarrow$  new associative array
24:   for  $j \leftarrow 0$  to  $max\_insertions$  do
25:     INSERT( $\mu, (\alpha_j \rightarrow total\_probes[j]/total\_insertions[j])$ )
26:   return  $\mu$ 

```

3.4.2 Average Probes per Insertion into Cuckoo Hash Tables without Deletions

Algorithm 2 describes the procedure by which we measured the average number of probes per insertion into a k -ary cuckoo hash table without using deletions. We use two arrays, *total_probes* and *total_insertions*, to accumulate the results across all trials, and we obtain the average afterwards. Before every trial we initialize an empty k -ary cuckoo hash table. The value *max_insertions* represents the highest load factor, $\alpha = \text{max_insertions}/N$ that we deem feasible. If in any trial the number of probes needed for an insertion exceeds the cache size, N , then we cease to obtain probe counts for higher load factors, as the conditional statement on line 17 accomplishes.

By similar data analysis techniques as in the previous section we find a bound on the average number of probes per insertion when no deletions are involved. Since FFRI and FFRU perform evictions, this bound does not inform the run time analysis of FFRI and FFRU in the preceding section, although it shows that a difference occurs when the average number of probes per insertion is measured when only insertions and updates are performed.

Observation 3.2 *When $N = 4096$, using a cuckoo hash table with $k = 4$, the average number of probes per insertion when deletions are not supported is less than $0.05 + 1/(0.98 - \alpha)$.*

Algorithm 2 Procedure to measure average # of probes per insertion using a k -ary cuckoo hash table without deletions

Input: N, k, max_trials

Output: μ , the average # of probes per insertion as a function of α

```

1: procedure AVE – PROBES( $N, k, max\_trials$ )
2:   for  $j \leftarrow 0$  to  $N$  do
3:      $total\_probes[j] \leftarrow 0$ 
4:      $total\_insertions[j] \leftarrow 0$ 
5:    $num\_trials \leftarrow 0$ 
6:    $max\_insertions \leftarrow N$ 
7:   while  $num\_trials < max\_trials$  do
8:     Initialize a  $k$ -ary cuckoo hash table having  $N$  empty entries and  $\alpha = 0$ .
9:      $j \leftarrow 0$ 
10:    while  $j \leq max\_insertions$  do
11:       $key \leftarrow$  random value
12:      if  $key$  is not in hash table then
13:        Insert  $key$  into hash table
14:        Measure the probes for this insertion as  $num\_probes$ .
15:         $total\_probes[j] \leftarrow total\_probes[j] + num\_probes$ 
16:         $total\_insertions[j] \leftarrow total\_insertions[j] + 1$ 
17:        if  $num\_probes > N$  and  $max\_insertions = N$  then
18:           $max\_insertions \leftarrow j$ 
19:           $j \leftarrow j + 1$ 
20:         $num\_trials \leftarrow num\_trials + 1$ 
21:     $\mu \leftarrow$  new associative array
22:    for  $j \leftarrow 0$  to  $max\_insertions$  do
23:      INSERT( $\mu, (\alpha_j \rightarrow total\_probes[j]/total\_insertions[j])$ )
24:    return  $\mu$ 

```

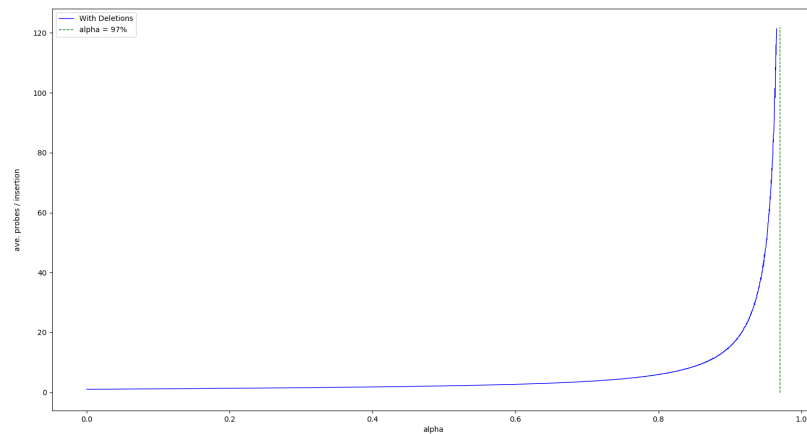


Figure 3.1: Average # of probes / insertion with cuckoo hashing, with deletions

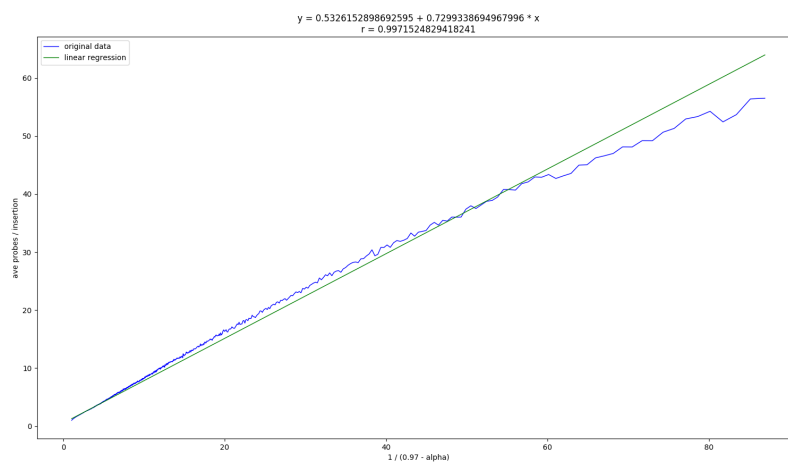


Figure 3.2: Linear regression of average probes / insertion as a function of $1/(0.97 - \alpha)$

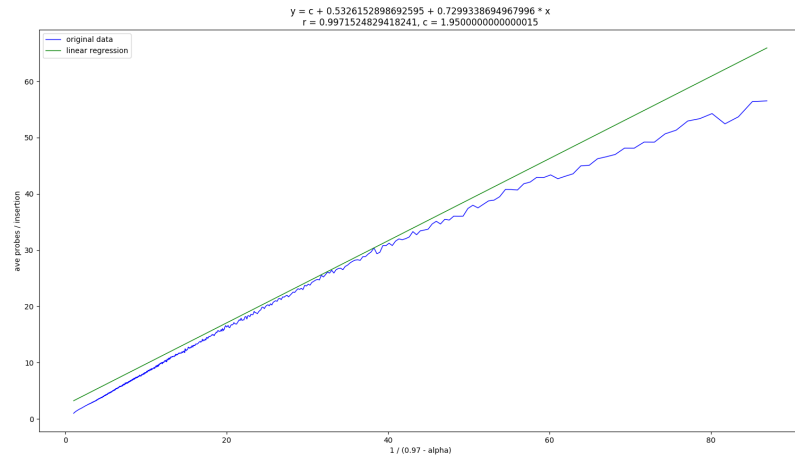


Figure 3.3: Bounding average probes / insertion as a function of $1/(0.97 - \alpha)$

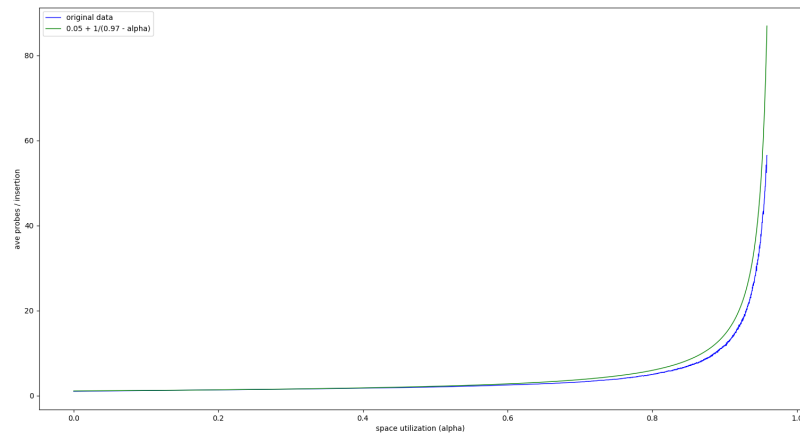


Figure 3.4: Bounding average probes / insertion as a function of α

CHAPTER 4

VERIFYING FFRU'S PERFORMANCE GUARANTEES

Gutta cavat lapidem, non vi sed
saepe cadendo. (A water drop
hollows a stone, not by force but by
falling often.)

Ovid

In this chapter we report empirical evidence to substantiate the cache miss performance of FFRI and FFRU. Our expectation is that FFRU always evicts an item having recency no less than ϕN . Furthermore, when LRU and FFRU are configured to provide equivalent recency guarantees, memoizing with FFRU should incur fewer cache misses than with LRU. In order to give support to these claims we employ two approaches.

The first approach is to perform a fixed sequence of cache operations on both LRU and FFRU. While performing these operations we log the items that are evicted and analyze this eviction log. In the analysis of the evicted items we compute the recency of each eviction, in order to verify that every evicted

item has recency greater than or equal to ϕN . This approach is discussed in section 4.1.

The second approach is to solve a variety of memoized optimization problems using both LRU and FFRU. We repeatedly solve a given problem with varying cache size or with varying problem size. For each trial we record the number of cache misses incurred, in order to verify that FFRU incurs no greater cache misses than LRU when the two are configured with equivalent recency guarantees. This approach is discussed in sections 4.2-4.5.

4.1 Verifying Recency Guarantees for Fixed Operation Sequences

A **fixed operation sequence** is a list of tuples having the form:

$$\begin{aligned} &(\textit{operation}_1, \textit{key}_1) \\ &(\textit{operation}_2, \textit{key}_2) \\ &(\textit{operation}_3, \textit{key}_3) \\ &(\textit{operation}_4, \textit{key}_4) \\ &\dots \end{aligned}$$

Where an *operation* is either an insertion or a retrieval on a caching module. A fixed operation sequence can be performed on a given caching module with the objective of recording when each cached item is evicted. Once all operations are performed and the corresponding eviction log is obtained, we can analyze this log to obtain for each evicted item its relative and absolute usage ages. Across any given range of operations the minimum eviction age should be at least ϕN . Using a logged sequence of insertions, accesses, and evictions, a separate cache analysis module maintains absolute timestamps for every operation (something FFRU does *not* maintain), so that it can compute the absolute eviction age of every evicted item. Additionally, the cache analysis

module maintains a priority queue of all items currently in the cache, in order to compute the relative eviction age of every evicted item.

The recency bound specified above holds true for both the absolute and the relative versions of FFRU. With Absolute FFRU the lower bound on ϕ determines the minimum *time recency* of any eviction, which is the difference between the most recent absolute timestamp when the evicted item was handled and the current absolute timestamp (if absolute timestamps were being tracked). With Relative FFRU the lower bound on ϕ determines the minimum *item recency* of any eviction, which is the number of items *still in the table* that have been handled more recently than the evicted item. Given this distinction between relative and absolute eviction age, LRU can be defined as always evicting the item having greatest relative usage age.

The second approach by which we may surmise the performance of a caching module is to solve memoized problems and observe the cache miss behavior with various cache size/problem size scenarios. With LRU the minimum eviction age is known by definition to be N , i.e. the size of the LRU cache. With FFRU caching instead we have the expression:

$$\phi N \geq N(d-1)(\kappa a - 1)/(\kappa - d) \quad (4.1)$$

which tells us how many of the items still in the cache are guaranteed to be recent. Therefore, if we configure FFRU, via appropriate values for κ and d , to provide for similar performance guarantees to a given LRU cache, then we would expect that the cache misses from the former module would remain within a constant factor of those from the latter. In the rest of this chapter, we study the cache performance of several memoized algorithms when using LRU and when using the variants of FFRU cache replacement policies with intention of confirming that claim.

In the following sections we observe that in practice the recency and load factor guarantees are respected when FFRU is used to solve the below mentioned memoized optimization problems.

4.2 Fibonacci

With the Fibonacci versions we benefit from having predictable access patterns, which allows us to realize a couple of notable cache performance improvements. We consider several variants on the classical formulation of the Fibonacci algorithm, which make two recursive calls at every level of recursion. We explore the effect of varying the order of recursive calls and of decreasing the problem size, in terms of its effect on the critical cache size and on its effect on cache misses as a function of problem size.

4.2.1 Version 0

We begin by analyzing the original version of Fibonacci. Algorithm 3 shows Fibonacci version 0A, which handles the larger subproblem first. This means that all of the recursive calls needed to satisfy the $n - 1$ case must be completed before recursive call for the $n - 2$ case begins, because we assume these procedures are executed on a single processor thread. We measure the cache misses of version 0A and note that only three values are needed to be stored before affecting the cache misses.

Observation 4.1 *When $C > 2$, for all positive n , $M_{LRU}(F_{0A}(n)) = n + 1$. It follows that, $\hat{C}_{LRU}(F_{0A}) = 3$ and $\hat{M}(F_{0A}(n)) = n + 1$.*

This observation is obtained by solving a given problem size repeatedly using increasing cache sizes and recording the cache misses that occur after each trial. Given the access pattern of variants of our version 0 of the Fibonacci algorithm, we note that nothing is gained by caching more than three values at a time using LRU. However, when the cache size is reduced by just one item, the performance changes notably. A corollary to this observation is that any cache replacement policy which provides for a minimum relative eviction age of 3 will incur the same number of cache misses as noted here, a point which will help us show that the performance guarantees of our FFRI/FFRU implementation are met.

Observation 4.2 *When $C = 2$, $n > 3$, $M_{LRU}(F_{0A}(n)) = M_{LRU}(F_{0A}(n-1)) + M_{LRU}(F_{0A}(n-4)) + 2$, with initial conditions $M_{LRU}(F_{0A}(1)) = 1$, $M_{LRU}(F_{0A}(2)) = 3$, $M_{LRU}(F_{0A}(3)) = 5$, $M_{LRU}(F_{0A}(4)) = 8$.*

This observation points out that the growth of the number of cache misses can be expressed by the recurrence relation $f_n = f_{n-1} + f_{n-4} + 2$, with initial conditions $f_1 = 1, f_2 = 3, f_3 = 5, f_4 = 8$. This is a non-homogeneous linear recurrence relation having constant coefficients, and as such it can be solved by symbolic differentiation. Given that this polynomial has several non-trivial roots, we obtain an approximate solution by evaluating at www.wolframcloud.com the Wolfram language expression given in the listing below, and we take the largest summand of the returned result.

```

1 N[RSolve[
2   {
3     f[n] == 2 + f[-1+n] + f[-4+n],
4     f[1] == 1,
5     f[2] == 3,
6     f[3] == 5,
7     f[4] == 8
8   },
9   {
10    f[n]
11  },
12  n
13 ],50]

```

Listing 4.1: Wolfram language to solve recurrence relation

We then use this strategy to formulate the following conjecture.

Observation 4.3 *When $C = 2$, $n > 1$, $M_{LRU}(F_{0A}(n)) = \text{round}(ab^n - 2)$ where $a \approx 2.63$ and $b \approx 1.38$.*

We verify this observation for large n and find that it converges exponentially fast, as the smaller terms become negligible.

Observation 4.4 *When $C < 2$, for all positive n , $M(F_{0A}(n)) = 2F(n+1) - 1$.*

When we cache less than 2 values we notice the above pattern in the cache misses. This observation holds regardless of caching strategy, given that a cache size of one or zero renders the eviction policy irrelevant. It's notable at any rate that caching one value yields no difference in cache miss performance in this scenario.

Algorithm 3 Fibonacci Version 0A

Input: n , a positive integer

Output: the n^{th} Fibonacci number

```

1: procedure  $F_{0A}(n)$ 
2:   if  $n = 0$  then
3:     return 0
4:   if  $n = 1$  then
5:     return 1
6:   return  $F_{0A}(n - 1) + F_{0A}(n - 2)$ 

```

Algorithm 4 shows Fibonacci Version 0B, which handles the smaller sub-problem first. This access pattern exploits LRU's priority queue behavior better, as we see in observation 4.5, which shows the improved critical cache size.

Observation 4.5 *When $C > 1$, for all positive n , $M_{LRU}(F_{0B}(n)) = n + 1$. It follows that, $\hat{C}_{LRU}(F_{0B}) = 2$ and $\hat{M}(F_{0B}(n)) = n + 1$.*

When the caching strategy is irrelevant however algorithm 4 does no better than algorithm 3, as seen in observation 4.6.

Observation 4.6 *When $C < 2$, for all positive n , $M(F_{0B}(n)) = 2F(n+1) - 1$.*

Algorithm 4 Fibonacci Version 0B

Input: n , a positive integer

Output: the n^{th} Fibonacci number

```

1: procedure  $\mathbf{F}_{0\mathbf{B}}(n)$ 
2:   if  $n = 0$  then
3:     return 0
4:   if  $n = 1$  then
5:     return 1
6:   return  $\mathbf{F}_{0\mathbf{B}}(n - 2) + \mathbf{F}_{0\mathbf{B}}(n - 1)$ 

```

Algorithm 5 shows Fibonacci Version 0C, which handles the smaller sub-problem first for even problem sizes. Checking the parity of the problem size has no advantage in terms of cache miss performance as observation 4.7 shows.

Observation 4.7 *When $C > 1$, for all positive n , $M_{LRU}(F_{0C}(n)) = n + 1$. It follows that, $\hat{C}_{LRU}(F_{0C}) = 2$ and $\hat{M}(F_{0C}(n)) = n + 1$.*

When we only cache one value, algorithm 5 exhibits a stepwise behavior, as shown in figure 4.2.1, albeit one having a regular pattern. As a consequence in observation 4.8 we notice two separate recurrence relations.

Observation 4.8 *When $C < 2$, when $n > 3$, $M(F_{0C}(n)) = M(F_{0C}(n-1)) + 1$ if n is even. When n is odd, $M(F_{0C}(n)) = 2M(F_{0C}(n-1))$, with initial cases $M(F_{0C}(1)) = 1$, $M(F_{0C}(2)) = 3$, $M(F_{0C}(3)) = 5$. When n is even, $M(F_{0C}(n)) = M(F_{0C}(n-1)) + 1 = 2M(n-2) + 1$. When n is odd, $M(n) = 2M(n-1) = 2M(n-2) + 2$.*

Using semilog regression we observe exponential growth, and by analyzing the growth rate we obtain the following closed form. We verify the correctness of this closed by direct comparison against the empirical data.

Observation 4.9 *When $C < 2$, for $n > 2$, $M(F_{0C}(n)) = \frac{7}{4}\sqrt{2}^{n+1} - 2$ when n is odd and $M(F_{0C}(n)) = \frac{7}{4}\sqrt{2}^n - 1$ when n is even.*

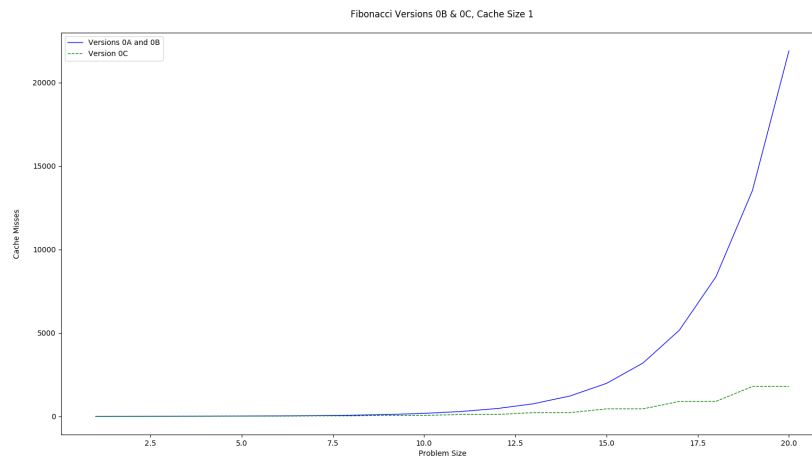


Figure 4.1: Versions 0A, 0B, and 0C, Cache Size 1

As figure 4.2.1 shows, the parity check employed by algorithm 5 yields an enormous advantage in cache miss performance.

Algorithm 5 Fibonacci Version 0C

Input: n , a positive integer

Output: the n^{th} Fibonacci number

```

1: procedure  $F_{0C}(n)$ 
2:   if  $n = 0$  then
3:     return 0
4:   if  $n = 1$  then
5:     return 1
6:   if  $n$  is odd then
7:     return  $F_{0C}(n - 1) + F_{0C}(n - 2)$ 
8:   else
9:     return  $F_{0C}(n - 2) + F_{0C}(n - 1)$ 

```

4.2.2 Version 1

In this series of versions of Fibonacci, we decrease the problem by half each time. We use the fact that when n is even,

$$F(n) = (F(\lfloor n/2 \rfloor + 1) + F(\lfloor n/2 \rfloor - 1)) \cdot (F(\lfloor n/2 \rfloor + 1) - F(\lfloor n/2 \rfloor - 1)) \quad (4.2)$$

Otherwise, when n is odd,

$$F(n) = (F(\lfloor n/2 \rfloor + 1) + F(\lfloor n/2 \rfloor - 1)) \cdot F(\lfloor n/2 \rfloor + 1) - (-1)^{\lfloor n/2 \rfloor} \quad (4.3)$$

When we cache only one value, we observe just a linear growth in cache misses.

Observation 4.10 *When $C = 1$, for all positive n , $0.5n < M(F_{1A}(n)) < 1.3n$.*

When we measure the critical cache size, \hat{C} , and the critical cache misses, \hat{M} , for this version we observe logarithmic growth.

Observation 4.11 *For $n \geq 50$, $\hat{C}(F_{1A}(n)) \leq 4.17 \cdot \log_2(n) - 11.18$*

Observation 4.12 *For $n \geq 50$, $\hat{M}(F_{1A}) < 4.17 \cdot \log_2(n) - 11.20$.*

In order to arrive at observation 4.11, we obtain the running max of the critical cache size. We then compute the semi-log regression lines of the left and right endpoints of the the running max, with the x-axis logarithmic and the y-axis linear. We adjust the slopes of both regression lines until they bound the running max of critical cache size in sample data. We take the slope and intercept of the upper bounding line of running max. And then we confirm this observation for greater values of n . Figure 4.2.2 gives an illustration of what that procedure looks like. Repeating recursive calls in algorithm 6 does not improve the cache miss performance. Changing the order of calls affects the cache miss performance only slightly. At any rate the order of calls given in algorithm 6 gave the best performance of those tried. The values in observations 4.11 and 4.12 represent the magnitude of these problems from a

memoization standpoint. As such they place version 1 of Fibonacci low on the scale of difficulty of memoization problems, since the overall load on a caching algorithm grows slowly as larger problem sizes are solved.

Algorithm 6 Fibonacci Version 1A

Input: n , a positive integer

Output: the n^{th} Fibonacci number

```

1: procedure  $\mathbf{F}_{1A}(n)$ 
2:   if  $n = 0$  then
3:     return 0
4:   if  $n = 1$  or  $n = 2$  then
5:     return 1
6:    $k \leftarrow \lfloor n/2 \rfloor$ 
7:    $f_1 \leftarrow \mathbf{F}_{1A}(k + 1)$ 
8:    $f_2 \leftarrow \mathbf{F}_{1A}(k - 1)$ 
9:   if  $n$  is even then
10:    return  $(f_1 + f_2) \cdot (f_1 - f_2)$ 
11:  else
12:    return  $(f_1 + f_2) \cdot f_1 - (-1)^k$ 

```

4.2.3 Version 2

In this series of versions of Fibonacci, we achieve even fewer cache misses using LRU than with the previous versions. It was shown in [Gries and Levin, 1980] that Fibonacci numbers can be computed in time proportional to the logarithm of the problem size. Algorithm 7 takes into account both the parity of n and the parity of $\lfloor n/2 \rfloor$. In this approach we exploit the following identities of the Fibonacci sequence.

When both n and $\lfloor n/2 \rfloor$ are even,

$$F(n) = F(\lfloor n/2 \rfloor)(2F(\lfloor n/2 \rfloor + 1) - F(\lfloor n/2 \rfloor)) \quad (4.4)$$

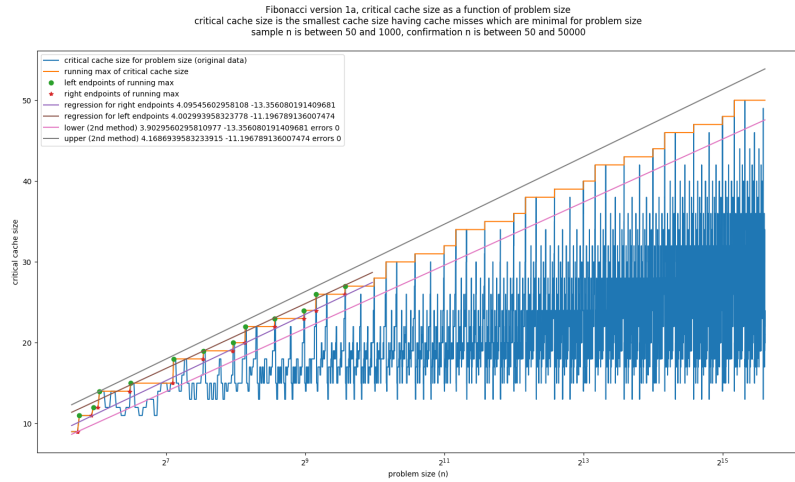


Figure 4.2: Version 1A, Bounding the Critical Cache Size

When n is even and $\lfloor n/2 \rfloor$ is odd,

$$F(n) = F(\lfloor n/2 \rfloor)(F(\lfloor n/2 \rfloor) + 2F(\lfloor n/2 \rfloor - 1)) \quad (4.5)$$

When n is odd and $\lfloor n/2 \rfloor$ is even,

$$F(n) = F(\lfloor n/2 \rfloor + 1)(2F(\lfloor n/2 \rfloor + 1) - F(\lfloor n/2 \rfloor)) - (-1)^{\lfloor n/2 \rfloor} \quad (4.6)$$

When both n and $\lfloor n/2 \rfloor$ are odd,

$$F(n) = (F(\lfloor n/2 \rfloor) + F(\lfloor n/2 \rfloor - 1))(F(\lfloor n/2 \rfloor) - 2F(\lfloor n/2 \rfloor - 1)) - (-1)^{\lfloor n/2 \rfloor} \quad (4.7)$$

We explore the effect in this algorithmic approach to Fibonacci of changing the order in which we perform the above parity checks and the effect of repeating certain recursive calls in order to exploit the eviction policy of LRU. The worst variant of version 2 is when we swap the order of lines 9 and 10 in algorithm 7. We called this version 2B, and with 4 cached values using LRU, we observe linear growth in cache misses.

Observation 4.13 *When $C = 4$, for $n > 4$, $M_{LRU}(F_{2B}(n)) \leq 1.25n - 1$.*

We analyze and confirm this observation programmatically by choosing a small range of data points for small instance sizes, fit a regression line, then confirm our observation for much larger sizes, as figure 4.2.4 illustrates. Without swapping lines 9 and 10 in algorithm 7, we find that when caching only 4 values with LRU, we see the performance improve to logarithmic growth in the cache misses.

Observation 4.14 *When $C > 4$, $n > 0$, $M_{LRU}(F_{2A}(n)) = 2\lfloor \log_2(n) \rfloor + 1$.*

And when we repeat recursive calls as shown in algorithm 8, we are able to achieve the same cache miss growth with one fewer cached item.

Observation 4.15 *When $C > 3$, $n > 0$, $M_{LRU}(F_{2AR}(n)) = 2\lfloor \log_2(n) \rfloor + 1$.*

The reason version 2AR improved over version 2A is because the repeated calls serve to adjust the position of the given subproblems within the LRU priority queue.

Algorithm 7 Fibonacci Version 2A

Input: n , a positive integer

Output: the n^{th} Fibonacci number

```

1: procedure  $\mathbf{F}_{2\mathbf{A}}(n)$ 
2:   if  $n == 0$  then
3:     return 0
4:   if  $n == 1$  then
5:     return 1
6:    $k \leftarrow \lfloor n/2 \rfloor$ 
7:   if  $n$  is even then
8:     if  $k$  is even then
9:        $f_1 \leftarrow \mathbf{F}_{2\mathbf{A}}(k)$ 
10:       $f_2 \leftarrow \mathbf{F}_{2\mathbf{A}}(k + 1)$ 
11:      return  $f_1 * (2 * f_2 - f_1)$ 
12:     else
13:        $f_3 \leftarrow \mathbf{F}_{2\mathbf{A}}(k)$ 
14:        $f_4 \leftarrow \mathbf{F}_{2\mathbf{A}}(k - 1)$ 
15:       return  $f_3 * (f_3 + 2 * f_4)$ 
16:     else
17:       if  $k$  is even then
18:          $f_5 \leftarrow \mathbf{F}_{2\mathbf{A}}(k + 1)$ 
19:          $f_6 \leftarrow \mathbf{F}_{2\mathbf{A}}(k)$ 
20:         return  $f_5 * (2 * f_5 - f_6) - (-1)^k$ 
21:       else
22:          $f_7 \leftarrow \mathbf{F}_{2\mathbf{A}}(k)$ 
23:          $f_8 \leftarrow \mathbf{F}_{2\mathbf{A}}(k - 1)$ 
24:         return  $(f_7 + f_8) * (f_7 + 2 * f_8) - (-1)^k$ 

```

When we compare FFRU with LRU, we restrict our attention to version 2, as it was our best performing version. We configure LRU with a cache size of 5, which means it always evicts the fifth oldest item. Likewise, with FFRU

we configure an equivalent guarantee by choosing $\kappa = 3$, $d = 2$, which yields:

$$\phi \geq \frac{5}{16} \tag{4.8}$$

$$\phi C_{FFRU} \geq 5 \tag{4.9}$$

In the case of version 2B, FFRU greatly outperforms LRU despite that their performance guarantees are similar, as shown in figure 4.2.4. This arises from the larger cache size that FFRU uses. If we look at figure 4.2.4 we see that all three variants exhibit the same cache miss behavior, which tells us that with this algorithm items are not being re-referenced.

Observation 4.16 *When $C_{LRU} = 5$, $C_{FFRU} = 16$, $\kappa = 3$, and $d = 2$, for $n > 0$, $M_{FFRU}(F_{2B}(n)) \leq M_{LRU}(F_{2B}(n))$.*

Observation 4.17 *When $C_{LRU} = 5$, $C_{FFRU} = 16$, $\kappa = 3$, and $d = 2$, for $n > 0$, $M_{FFRU}(F_{2A}(n)) \leq M_{LRU}(F_{2A}(n))$.*

Observation 4.18 *When $C_{LRU} = 4$, $C_{FFRU} = 16$, $\kappa = 3$, and $d = 2$, for $n > 0$, $M_{FFRU}(F_{2AR}(n)) \leq M_{LRU}(F_{2AR}(n))$.*

With versions 2A and 2AR we find that the cache misses are identical for LRU and FFRU.

4.2.4 Conclusions

To summarize what we have accomplished in this series of experiments with the above variants of the Fibonacci algorithm, we have succeeded in obtaining notable improvements to the cache miss performance when computing Fibonacci numbers. When we incorporate specific knowledge about the access patterns of this recursive algorithm and we use those insights together with a known cache eviction policy, we are able to modify the memoized algorithm by either changing the order of recursive calls or by repeating them. This adjusts the position of items within an LRU or FFRU cache so that items likely to be needed again will not be evicted.

Additionally, we established empirically that when our best performing Fibonacci version, i.e. version 2, is memoized with LRU and FFRU caching, the latter never incurs more cache misses, when the two caching algorithms are configured with equivalent recency guarantees. It should be noted that in this case we measured cache misses as a function of *problem size*, instead of cache size, because the number of subproblems that must be cached is already very low, regardless of how large the problem is. Additionally, given that the access patterns for this problem follow a simple pattern, we observe no difference in the cache misses incurred by LRU compared to FFRU. In the following sections as we explore memoized problems having more complex access patterns this will not be the case. We will see a clear difference between the cache misses incurred by LRU and FFRU, because with LRU there is always only one candidate for eviction, whereas FFRU always evicts a pseudorandom non-recent item.

Algorithm 8 Fibonacci Version 2AR

Input: n , a positive integer

Output: the n^{th} Fibonacci number

```

1: procedure  $\mathbf{F}_{2\text{AR}}(n)$ 
2:   if  $n == 0$  then
3:     return 0
4:   if  $n == 1$  then
5:     return 1
6:    $k \leftarrow \lfloor n/2 \rfloor$ 
7:   if  $n$  is even then
8:     if  $k$  is even then
9:        $f_1 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
10:       $f_2 \leftarrow \mathbf{F}_{2\text{AR}}(k + 1)$ 
11:       $f_1 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
12:      return  $f_1 * (2 * f_2 - f_1)$ 
13:     else
14:        $f_3 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
15:        $f_4 \leftarrow \mathbf{F}_{2\text{AR}}(k - 1)$ 
16:        $f_3 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
17:       return  $f_3 * (f_3 + 2 * f_4)$ 
18:     else
19:       if  $k$  is even then
20:          $f_5 \leftarrow \mathbf{F}_{2\text{AR}}(k + 1)$ 
21:          $f_6 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
22:          $f_5 \leftarrow \mathbf{F}_{2\text{AR}}(k + 1)$ 
23:         return  $f_5 * (2 * f_5 - f_6) - (-1)^k$ 
24:       else
25:          $f_7 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
26:          $f_8 \leftarrow \mathbf{F}_{2\text{AR}}(k - 1)$ 
27:          $f_7 \leftarrow \mathbf{F}_{2\text{AR}}(k)$ 
28:          $f_8 \leftarrow \mathbf{F}_{2\text{AR}}(k - 1)$ 
29:         return  $(f_7 + f_8) * (f_7 + 2 * f_8) - (-1)^k$ 

```

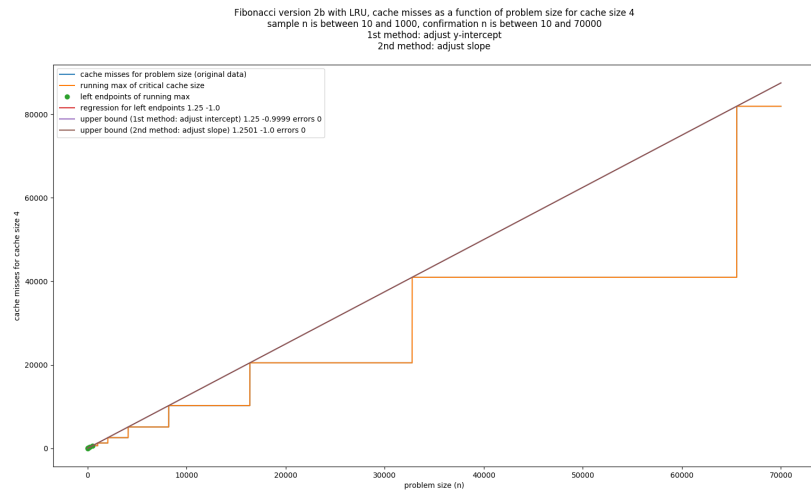


Figure 4.3: Fibonacci Version 2B, confirming our LRU observation programmatically

4.3 KMP String Matching

The Knuth-Morris-Pratt string matching algorithm finds all positions within text T where pattern P occurs. The original version of the KMP algorithm was presented in iterative form [Knuth et al.]. Given our objectives of exploring memoization performance we ideally want to express the algorithm in recursive form. We present a recursive variant of the Knuth-Morris-Pratt in this chapter and discuss its performance using LRU, FFRI, and FFRU cache replacement policies.

Problems like KMP, which include LCS (discussed later in this chapter), and other problems like shortest common supersequence, Levenshtein distance, sequence alignment, etc., all assume that sequences are drawn from some finite “alphabet” of symbols, eg. $\Sigma = \{a, b\}$. In the experiments on LCS and KMP in this chapter the size of the underlying alphabet from which instances are drawn, although finite, is allowed to grow as input sizes grow. Therefore, we use integers as our “symbols” to match on, meaning that if we needed an alphabet of size 20, for example, we would just use the set

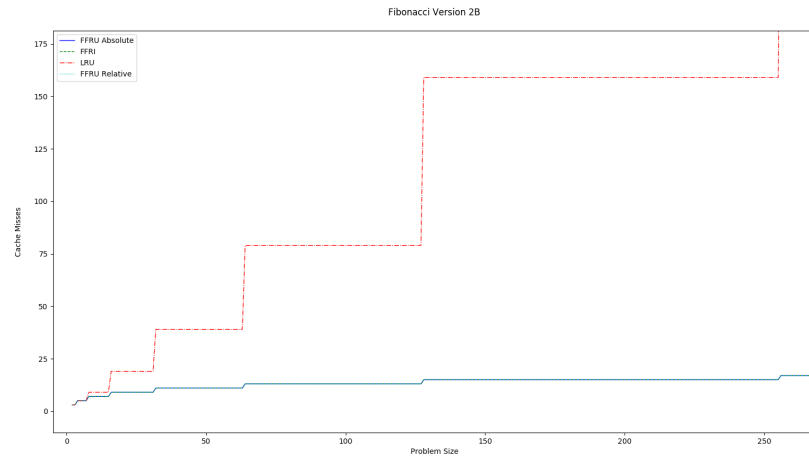


Figure 4.4: Fibonacci Version 2B, LRU and FFRU compared

$\{0, 1, \dots, 19\}$. In addition to analyzing the cache miss performance of these problems, we would like to find out what are the “hard” instances to these problems, which leads us to considering several types of input sequences.

4.3.1 Developing a Recursive Version

As a starting point for discussing our investigation of KMP, algorithm 9 shows the original iterative search algorithm. Given text T and pattern P , return all indices into T where P was found. Note that this search procedure allows for overlapping matches, so if P contains a prefix that is a proper suffix of the same, then the returned match indices could have spacing shorter in length than P . For example, if $P = abcdabc$ and $T = abcdabcdabc$, then the search would return $R = (0, 4)$.

The search mechanism relies on a pre-computed table, $next$, which is computed before traversing T . Table $next$ allows the index into T to skip past positions where a match with P could not occur. Knuth et al. [Knuth et al., 1977] develop the ideas behind computing $next$ by breaking it into two constituent concepts, which we’ll refer to as the **failure function** and the **prefix-suffix**

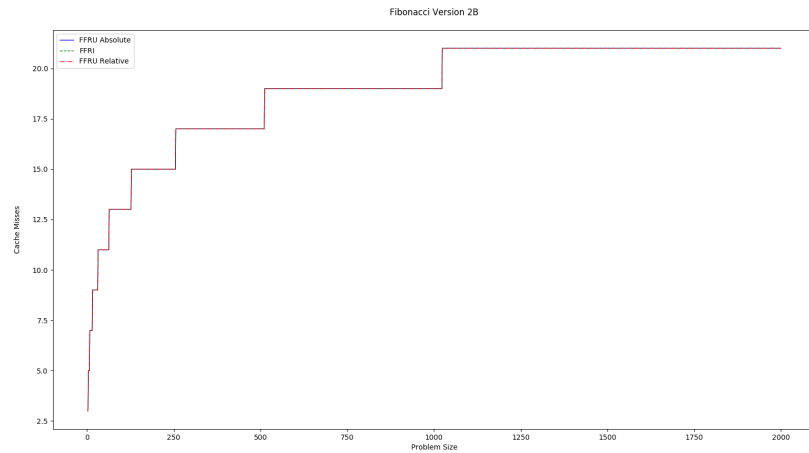


Figure 4.5: Fibonacci Version 2B, FFRU variants compared

function. These are described as follows.

Definition 4.1

- Given $P = (p_0p_2\dots p_{m-1})$, $f[q]$, the **prefix-suffix function**, is the length of the longest proper prefix of P that is also a proper suffix of $P[0..q]$.
- Given $P = (p_0p_2\dots p_{m-1})$, $next[t]$, the **failure function**, is the largest index i less than t such that the last t characters of the input matched P except for the i^{th} character.

As such $next[t]$ indicates the least amount of shift for which successive input characters can possibly match the portion of P already referenced. The above definition specifies *proper* prefixes and suffixes, because the longest prefix of any string that is also a suffix of the same is always the string itself. Given these two functions, Knuth et al. establish that $next$ depends on f , in that

$$next[j] = \begin{cases} f[j] & \text{if } P[j] \neq P[f[j]] \\ next[f[j]] & \text{if } P[j] = P[f[j]] \end{cases} \quad (4.10)$$

This relation serves as the basis for our investigation of a memoized, recursive

version of KMP¹. The idea here is that a potentially fortunate choice of cache replacement policy could allow the recursive version of KMP to enjoy the same run time efficiency as the iterative version, but with improved memory usage. This is because instead of computing table *next* before traversing text *T*, we make memoized calls to two new functions as we traverse the text, *T*, being searched, and a cache employing an eviction policy will only retain a limited set of the values that would comprise table *next*. We will call these new functions *PS* (for “prefix-suffix function”) and *FF* (for “failure function”), such that

$$PS(i) = f[i] \tag{4.11}$$

$$FF(i) = next[i] \tag{4.12}$$

Both of these functions rely only on the search pattern, *P*. As a consequence when looking at their memoized performance, we don’t report test results for the overall search algorithm. Instead we’re interested in the number of references to the pattern while calculating all of the table entries for a given pattern *P*. From this perspective we can then discuss potentially discuss “hard” inputs to this algorithm.

¹Note that this isn’t the first time KMP has been reformulated in recursive terms, although we aren’t aware of other investigations of memoized recursive formulations of KMP. See [Ager et al., 2002] for a reformulation of KMP using functional programming constructs.

Algorithm 9 KMP Iterative String Search

Input: $P = (p_0p_2\dots p_{m-1})$, $T = (t_0t_2\dots t_{n-1})$

Output: R is a sequence of array indices where P was found within T

```

1: procedure ITERATIVE – KMP – SEARCH( $P, T$ )
2:    $i \leftarrow 0$ 
3:    $j \leftarrow 0$ 
4:    $next \leftarrow$  COMPUTE – FAILURE – FUNCTION – TABLE( $P$ )
5:    $R \leftarrow \emptyset$ 
6:   while  $i < m$  and  $j < n$  do
7:     while  $-1 < i$  and  $P_i \neq T_j$  do
8:        $i \leftarrow next[i]$ 
9:      $i \leftarrow i + 1$ 
10:     $j \leftarrow j + 1$ 
11:    if  $i = m$  then
12:      add  $k - m$  to  $R$ 
13:       $i \leftarrow next[i]$ 
14:       $j \leftarrow j - i$ 
15:  return  $R$ 

```

4.3.2 Prefix-suffix Function

Algorithm 10 shows the iterative procedure that computes the entire prefix-suffix table. Although not directly implicated by the iterative search routine given above in algorithm listing 9, the values computed by this function would be needed for building a recursive version of the failure function. So, given the goal of memoizing the entire process, we would like a recursive version of this procedure, that let's us ask for one value of the prefix-suffix function at a time. This way if we vary the cache size and employ different cache eviction policies, we can gain insight as to how many of those values (and which ones) are really needed to be kept in memory before run time performance (as measured here by by cache miss increases) suffers prohibitively. The procedure showing our

recursive version of the prefix-suffix function is given in algorithm 11.

The base case occurs when we've reached the first character, in which case it is trivially the case that the longest proper prefix of one character that is also a proper suffix of the same is the empty string. In any other case, when considering the substring $P[0..q]$, we are able to build upon the prefix-suffix result of the substring $P[0..(q-1)]$. However, we notice that despite the iterative version making a pass through the entire pattern, in order to compute a single given value of the prefix-suffix, potentially we need only to inspect very few values. This is why this function is a good candidate to look at memoizing strategies using limited memory.

We solve this memoized prefix-suffix algorithm with LRU for a variety of input types. When Knuth et al. describe the KMP algorithm, they discuss what happens when $P = a^{n-1}b$. While they do not explicitly claim that this input is the hardest type for KMP, that would certainly appear so, since a possible mismatch wouldn't be noted until having reached the last character of the string. In our investigation of LRU, we find this input type to be the worst case of those we've tried, which includes random strings, as well as strings such as:

$$a^n \tag{4.13}$$

$$(ab)^{n/2} \tag{4.14}$$

$$a^{n/2}b^{n/2} \tag{4.15}$$

As the cache size, C_{LRU} , decreases we find the following hyperbolic growth in cache misses within a certain range of cache sizes:

Observation 4.19 *For inputs of type $a^{n-1}b$ for all positive n , when $1 \leq C_{LRU}(PS(n)) < \sqrt{n}$, then $n^2/2 \leq M_{LRU}(PS(n)) \cdot C_{LRU}(PS(n)) \leq n^2/2 + n\sqrt{n}$.*

In order to confirm this observation, we observe this relationship at certain smaller instances sizes. We fit curves to the data points using regression in order to obtain upper and lower bounds. We then test the observation for

larger instances. Despite the prefix-suffix function having linear run time performance, it still requires $O(n^2)$ memory before noticeable increases in cache misses slow it down. Figure 4.6 illustrates these fitted curves for several instance sizes.

We are interested in comparing the performance of LRU and the variants of FFRU that we describe in the sections above. When the eviction age lower bounds of LRU and FFRU, denoted $\phi_{LRU}N$ and $\phi_{FFRU}N$ respectively, offer equivalent guarantees, our expectation is the cache misses incurred with FFRU never exceed those of LRU.

Observation 4.20 *For inputs of type $a^{n-1}b$ for all positive n , when $1 \leq C_{LRU}(PS(n)) < \sqrt{n}$, and $\phi_{LRU}N_{LRU} \equiv \phi_{FFRU}N_{FFRU}$, $M_{FFRU}(PS(n)) \leq M_{LRU}(PS(n))$.*

With input type $a^{n-1}b$ all variants of FFRU outperform LRU, as figure 4.7 illustrates, and the margin of performance improves as the cache size decreases. To get a better look at the comparative performance of the three variants of FFRU, figure 4.8 shows small differences, owing to the fact that some references do occur, but the close similarities between the three variants tell us that this is still an instance of sparse dynamic programming, like in the case of Fibonacci. We select a variety of FFRU parameter settings that provide eviction age lower bound guarantees that are in the same range as those of the LRU cache miss data to be compared. These FFRU parameter settings are reported in table 4.1.

Algorithm 10 KMP prefix-suffix iterative version

Input: $P = (p_0p_2\dots p_{m-1})$

Output: table f specifies for each character i in P the length of the longest proper prefix of $P[0..i]$ that is also a proper suffix of the same

```

1: procedure COMPUTE – TABLE – F()
2:    $f[0] \leftarrow 0$ 
3:    $k \leftarrow 0$ 
4:    $q \leftarrow 1$ 
5:   while  $q < |P|$  do
6:     while  $0 < k$  and  $P_k \neq P_q$  do
7:        $k \leftarrow f[k - 1]$ 
8:     if  $P_k = P_q$  then
9:        $k \leftarrow k + 1$ 
10:     $f[q] \leftarrow k$ 
11:     $q \leftarrow q + 1$ 
12:  return  $f$ 

```

Algorithm 11 KMP prefix-suffix recursive version

Input: index q into globally available $P = (p_0p_2\dots p_{m-1})$

Output: the length of the longest proper prefix of $P[0..q]$ that is also a proper suffix of the same

```

1: procedure PS( $q$ )
2:   if  $q = 0$  then
3:     return 0
4:    $k \leftarrow \text{PS}(q - 1)$ 
5:   while  $0 < k$  and  $P_k \neq P_q$  do
6:      $k \leftarrow \text{PS}(k - 1)$ 
7:   if  $P_k = P_q$  then
8:     return  $k + 1$ 
9:   else
10:    return  $k$ 

```

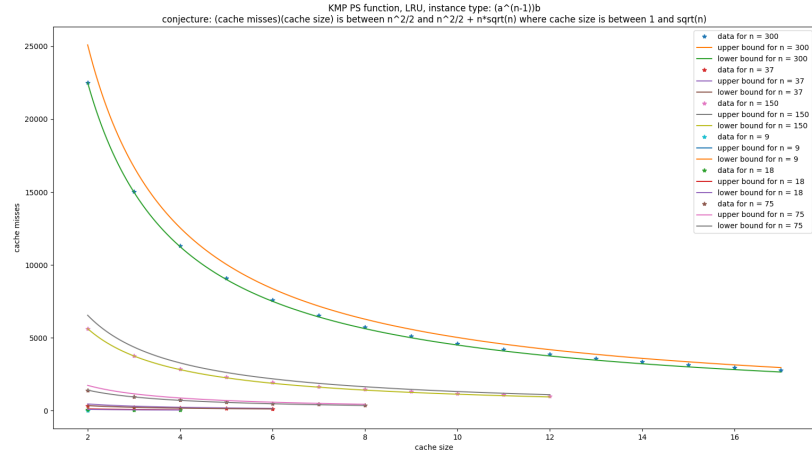


Figure 4.6: KMP's prefix-suffix function memoized with LRU

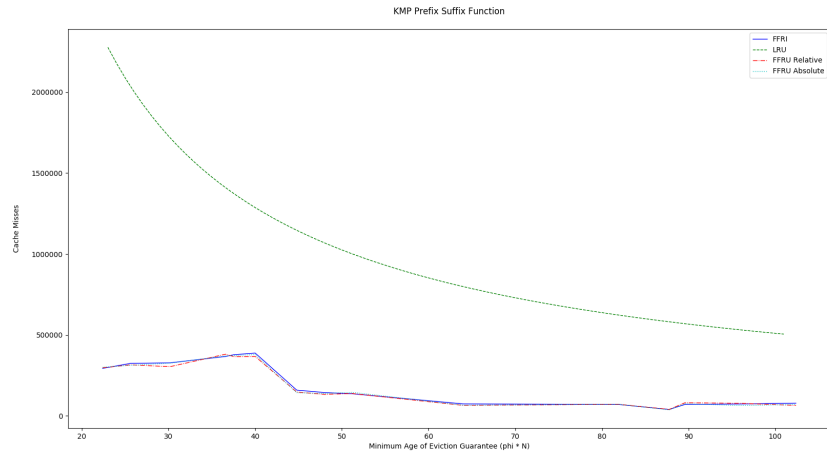


Figure 4.7: KMP's prefix-suffix, comparing LRU to FFRU

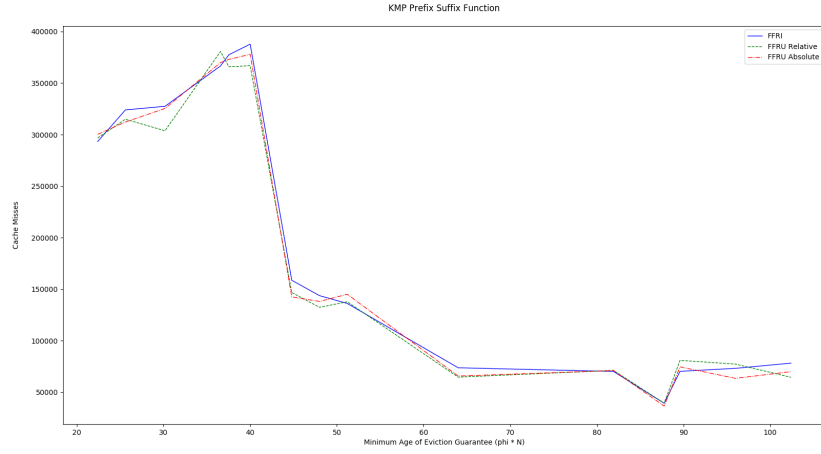


Figure 4.8: KMP's prefix-suffix, FFRU all three variants

Table 4.1: FFRU Parameters used with KMP Prefix-suffix function.

ϕN	cache size (N)	κ	d	a
22.4	64	3	2	0.45
25.6	64	4	3	0.3
30.17142857	64	9	7	0.1285714286
36.57142857	64	10	7	0.1285714286
37.54666667	64	8	5	0.18
40	64	10	6	0.15
44.8	128	3	2	0.45
48	128	5	4	0.225
51.2	128	4	3	0.3
64	256	7	6	0.15
81.92	256	6	5	0.18
87.77142857	512	8	7	0.1285714286
89.6	256	3	2	0.45
96	256	5	4	0.225
102.4	256	4	3	0.3

4.3.3 Failure Function

Algorithm 12 shows the iterative procedure that computes the entire failure function table, namely the one that algorithm 9 relies on as it traverses the search text T . It is this function whose recursive version would allow us to eliminate the need to generate the entire table up front. The choice of setting $next[0]$ to -1 is merely a convention, since this value isn't really needed. When the value -1 occurs in table $next$ it means none of the pattern has been matched, therefore the procedure cannot skip ahead in the search text. Empirical results suggest that FF has comparable cache miss performance to PS .

4.3.4 Conclusions

To summarize some takeaways from studying our recursive reformulation of KMP string matching, we do not require the entire prefix-suffix table nor the failure function table to be in memory. By caching only a subset of these values, we are able to perform efficient string matching with the same run time performance. As opposed to Fibonacci, our memoized reformulation of KMP exhibits cache miss performance when memory is limited that shows a notable difference between LRU and FFRU.

Another important theme arising from these results and the ones in the following sections has to do with how similar the cache miss behavior observed across the three variants of FFRU. In particular FFRI and FFRU incur very similar cache miss behavior, which indicates very low overlap of subproblems. This is because after most values in the cache are inserted, they are seldom retrieved again.

Algorithm 12 KMP failure function iterative version

Input: $P = (p_0p_2\dots p_{m-1})$

Output: $next[t]$ is the largest index i less than t such that the last t characters of the input matched P except for the i^{th} character

```

1: procedure COMPUTE – FAILURE – FUNCTION – TABLE( $P$ )
2:    $next[0] \leftarrow -1$ 
3:    $t \leftarrow 0$ 
4:    $k \leftarrow -1$ 
5:   while  $t < |P|$  do
6:     while  $-1 < k$  and  $P_k \neq P_t$  do
7:        $k \leftarrow next[k]$ 
8:      $k \leftarrow k + 1$ 
9:      $t \leftarrow t + 1$ 
10:    if  $P_k = P_t$  then
11:       $next[t] \leftarrow next[k]$ 
12:    else
13:       $next[t] \leftarrow k$ 
14:    return  $next$ 

```

Algorithm 13 KMP failure function recursive version

Input: index t into globally available $P = (p_0p_2\dots p_{m-1})$

Output: the largest index i less than t such that the last t characters of P matched except for the i^{th} character

```

1: procedure FF( $t$ )
2:   if  $t = 0$  then
3:     return  $-1$ 
4:    $k \leftarrow \mathbf{PS}(t - 1)$ 
5:   if  $P_k = P_t$  then
6:     return FF( $k$ )
7:   else
8:     return  $k$ 

```

Algorithm 14 KMP recursive string search

Input: $P = (p_0p_2\dots p_{m-1})$, $T = (t_0t_2\dots t_{n-1})$

Output: R is a sequence of array indices where P was found within T

```

1: procedure RECURSIVE – KMP – SEARCH( $P, T$ )
2:    $i \leftarrow 0$ 
3:    $j \leftarrow 0$ 
4:    $R \leftarrow \emptyset$ 
5:   while  $i < m$  and  $j < n$  do
6:     while  $-1 < i$  and  $P_i \neq T_j$  do
7:        $i \leftarrow \mathbf{FF}(i)$ 
8:        $i \leftarrow i + 1$ 
9:        $j \leftarrow j + 1$ 
10:    if  $i = m$  then
11:      add  $k - m$  to  $R$ 
12:       $i \leftarrow \mathbf{FF}(i)$ 
13:       $j \leftarrow j - i$ 
14:  return  $R$ 

```

4.4 Longest Common Subsequence

The Longest Common Subsequence (LCS) problem consists of determining the length of the longest subsequence common to two input sequences, $X = (x_0x_1\dots x_{n_1-1})$ and $Y = (y_0y_1\dots y_{n_2-1})$. For the purposes of discussing empirical results from our experimental trials, we'll refer to the problem size n as length of the longer of X and Y , although in the following results we only consider inputs where $|X| = |Y|$. As with our work with KMP, the *alphabet* from which we draw LCS problem instances consists of the non-negative integers. In trying to find hard inputs we explore a variety of types of sequences, and for each type we measure the critical cache size, \hat{C} , for a range of problem sizes. We find the critical cache size using binary search. In addition to random inputs

we initially considered simple inputs like these:

$$(X = 1^n, Y = 1^n) \quad (4.16)$$

$$(X = 0^n, Y = 1^n) \quad (4.17)$$

$$(X = 012\dots n, Y = n\dots 210) \quad (4.18)$$

$$(X = (01)^{n/2}, Y = (10)^{n/2}) \quad (4.19)$$

$$(X = 0^{n/2}1^{n/2}, Y = 1^{n/2}0^{n/2}) \quad (4.20)$$

$$(X = 0^{n/10}1^{n/10}2^{n/10} \dots 9^{n/10}, Y = 1^{n/10}2^{n/10}3^{n/10} \dots 9^{n/10}0^{n/10}) \quad (4.21)$$

$$(X = 0^{n/10}1^{n/10}2^{n/10} \dots 9^{n/10}, Y = 9^{n/10} \dots 2^{n/10}1^{n/10}0^{n/10}) \quad (4.22)$$

Of these input 4.22 produced the greatest cache misses for any given n . So, we investigated repeating symbols while increasing the number of total symbols, each time using the reverse string as the second input. But given that there's nothing special about the value 10 in input 4.22, we tried to generalize over that type of pattern with inputs of the form:

$$(X = 0^{n/a}1^{n/a}2^{n/a} \dots (a-1)^{n/a}, Y = (a-1)^{n/a} \dots 2^{n/a}1^{n/a}0^{n/a}), a = f(n) \quad (4.23)$$

As a starting point we looked at the case when $a = \sqrt{n}$:

$$(X = 0^{\sqrt{n}}1^{\sqrt{n}}2^{\sqrt{n}} \dots (\sqrt{n}-1)^{\sqrt{n}}, Y = (\sqrt{n}-1)^{\sqrt{n}} \dots 2^{\sqrt{n}}1^{\sqrt{n}}0^{\sqrt{n}}) \quad (4.24)$$

which generalizes to:

$$(X = 0^{n^c}1^{n^c}2^{n^c} \dots (n^c-1)^{n^c}, Y = (n^c-1)^{n^c} \dots 2^{n^c}1^{n^c}0^{n^c}) \quad (4.25)$$

where $0 \leq c \leq 1$, for which we investigated values of c in 0.01 increments. The form given by the expression in 4.25 simplifies to:

$$(X = 0^a1^a2^a \dots b^a(b+1)^{n-(b+1)a}, Y = (b+1)^{n-(b+1)a}b^a \dots 2^a1^a0^a) \quad (4.26)$$

where $a = \lfloor n^c \rfloor$ and $b = \lfloor n/a \rfloor$. Through regression analysis we conclude that the hardest input that we were able to find have the form:

$$(X = 0^a1^a2^{n-2a}, Y = 2^{n-2a}1^a0^a) \quad (4.27)$$

where $a = \lceil n/3 \rceil$.

4.4.1 Cost Function

We examine two methods of computing the length of the longest common subsequence of X and Y . For both methods we study the affect of changing the order of recursive calls. Algorithms 15 and 16 show the conventional approach, which makes the necessary recursive calls corresponding to reducing the length of X or of Y separately. We explore the effect of swapping the order of the subproblems visited, hence the two separate versions. Of these two versions, algorithm 16 performs better. For the hardest inputs we could find, we see that the critical cache size does not grow faster than the product of the lengths of the two input sequences.

Observation 4.21 *For $n \geq 10$, when X has the form $0^a 1^a 2^{n-2a}$, $a = \lceil n/3 \rceil$, Y is the reverse of X , then the running max of $\hat{C}(LCS_2(n))$ is between $0.270 \cdot n^2 + 1.055n + 4.085$ and $0.278 \cdot n^2 + 0.145n + 1.764$.*

We confirm this observation by fitting a regression based curve to a small portion of data for small input sizes, then check whether the bound holds true for larger input sizes.

Observation 4.22 *For $n \geq 10$, when X has the form $0^a 1^a 2^{n-2a}$, $a = \lceil n/3 \rceil$, Y is the reverse of X , and $\phi_{LRU} N_{LRU} \equiv \phi_{FFRU} N_{FFRU}$, $M_{FFRU}(LCS_2(n)) \leq M_{LRU}(LCS_2(n))$.*

Figure 4.4.1 shows that all three variants of FFRU clearly outperform LRU in terms of cache misses. Note that this figure illustrates the behavior of cache misses as a function of the minimum eviction age guarantee of the cache eviction policy in question and *not* the cache size. At the smaller cache sizes in figure 4.4.1 we see quite a bit of differences among the three variants of FFRU, which makes this problem type more complex than KMP. There isn't a clear favorite across the variants, and it varies across cache sizes which variants performs best. When LRU is used minimum eviction age guarantee happens also to be the cache size, but with the FFRU variants used the values along

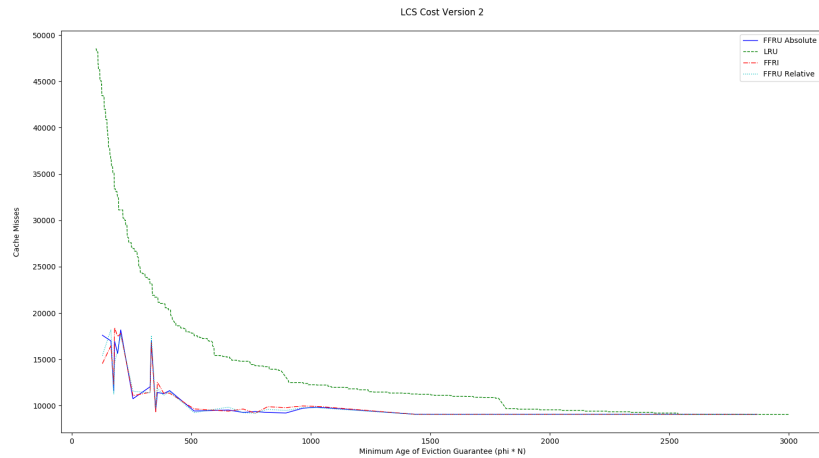


Figure 4.9: LCS Cost Version 2, comparison between LRU and FFRU

the x-axis are ϕN . Table 4.2 reports the FFRU configuration parameters used for the empirical data shown in figure 4.4.1.

4.4.2 Oblivious LCS

For the sake of improved cache miss performance we found that making the third redundant recursive call, i.e. reducing the length of both strings, produced favorable results. Algorithms 17, 18, 19, 20, 21, 22 show different orders of calls made to what we call Oblivious LCS, which we present due to its superior cache miss performance. When compared with the conventional variants of LCS, namely algorithms 15 and 16, the oblivious versions incur much fewer cache misses, because they exploit the cache eviction policy being used.² We notice a slight advantageous difference in versions 1-4 compared to versions 5 and 6. Evidently making the redundant recursive call first incurs more cache misses, due to where it places the relevant subproblem in the LRU

²We investigated the Needleman–Wunsch algorithm for sequence alignment and the Levenshtein distance, but omitted reporting those results, as they exhibit the same pattern of recursive calls as Oblivious LCS does.

Table 4.2: FFRU Parameters used with LCS Cost Version 2.

ϕN	cache size (N)	κ	d	a
128	512	7	6	0.15
163.84	512	6	5	0.18
175.5428571	1024	8	7	0.1285714286
179.2	512	3	2	0.45
192	512	5	4	0.225
204.8	512	4	3	0.3
256	1024	7	6	0.15
327.68	1024	6	5	0.18
332.8	512	11	6	0.15
351.0857143	2048	8	7	0.1285714286
358.4	1024	3	2	0.45
384	1024	5	4	0.225
409.6	1024	4	3	0.3
512	2048	7	6	0.15
655.36	2048	6	5	0.18
716.8	2048	3	2	0.45
768	2048	5	4	0.225
819.2	2048	4	3	0.3
896	2048	10	8	0.1125
965.4857143	2048	9	7	0.1285714286
1024	2048	5	3	0.3
1433.6	4096	3	2	0.45
1638.4	4096	4	3	0.3
2129.92	4096	7	5	0.18
2402.986667	4096	8	5	0.18
2867.2	8192	3	2	0.45
3276.8	8192	4	3	0.3
4259.84	8192	7	5	0.18

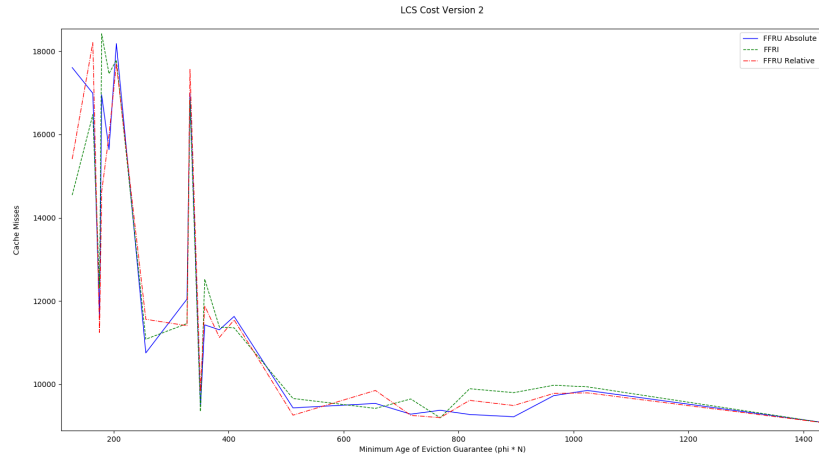


Figure 4.10: LCS Cost Version 2, FFRU all three variants

priority queue.

Observation 4.23 For $n > 0$, when $C \geq 2n + 2$, $M_{LRU}(OLCS_{1-4}(n)) = (n + 1)^2$, regardless of input type.

Observation 4.24 For $n > 0$, when $C \geq 4n - 5$, $M_{LRU}(OLCS_{5,6}(n)) = (n + 1)^2$, regardless of input type.

At any rate all six OLCS Cost versions perform much better than the conventional LCS cost algorithm with the added oblivious subproblem.

With regard to the comparison between LRU and FFRU, we measure cache misses with LRU and FFRU in the same range of eviction age lower bound guarantees.

Observation 4.25 For $n \geq 0$, regardless of input type, and $\phi_{LRU}N_{LRU} \equiv \phi_{FFRU}N_{FFRU}$, $M_{FFRU}(OLCS_4(n)) \leq M_{LRU}(OLCS_4(n))$.

Figure 4.4.2 illustrates the advantage of all three FFRU variants with respect to LRU with version 4 of OLCS cost. The cache misses with FFRU are never

greater than those with LRU for comparable freshness guarantees. The regular patterns observed in the LRU data are due to the fact that the eviction age of every item is the same. Table 4.3 lists the FFRU parameters used to generate the empirical results illustrated in figure 4.4.2. We zoom in on a small portion of the cache sizes tested and show in figure 4.4.2 that for this problem type there are only small variations in the performance exhibited by the three variants of FFRU. Something to note in the FFRU parameters listed in these tables is that the amount of memory overhead varies in the scenarios tested, but this was intentional in order to observe (and report) the behavior arising from a variety of degrees of memory usage.

Algorithm 15 LCS Cost Version 1

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{LCS}_1(i, j)$ 
2:   if  $X_i = Y_j$  then
3:     return  $\text{LCS}_1(i - 1, j - 1) + 1$ 
4:   else
5:      $c1 \leftarrow \text{LCS}_1(i, j - 1)$ 
6:      $c2 \leftarrow \text{LCS}_1(i - 1, j)$ 
7:     return  $\max(c1, c2)$ 

```

Algorithm 16 LCS Cost Version 2

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{LCS}_2(i, j)$ 
2:   if  $X_i = Y_j$  then
3:     return  $\text{LCS}_2(i - 1, j - 1) + 1$ 
4:   else
5:      $c2 \leftarrow \text{LCS}_2(i - 1, j)$ 
6:      $c1 \leftarrow \text{LCS}_2(i, j - 1)$ 
7:     return  $\max(c1, c2)$ 

```

Algorithm 17 Oblivious LCS Cost Version 1

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{OLCS}_1(i, j)$ 
2:    $c_1 \leftarrow \text{OLCS}_1(i, j - 1)$ 
3:    $c_2 \leftarrow \text{OLCS}_1(i - 1, j)$ 
4:    $c_3 \leftarrow \text{OLCS}_1(i - 1, j - 1) + (X_i = Y_j)$ 
5:   return  $\max(c_1, c_2, c_3)$ 

```

Algorithm 18 Oblivious LCS Cost Version 2

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{OLCS}_2(i, j)$ 
2:    $c_1 \leftarrow \text{OLCS}_2(i, j - 1)$ 
3:    $c_3 \leftarrow \text{OLCS}_2(i - 1, j - 1) + (X_i = Y_j)$ 
4:    $c_2 \leftarrow \text{OLCS}_2(i - 1, j)$ 
5:   return  $\max(c_1, c_2, c_3)$ 

```

Algorithm 19 Oblivious LCS Cost Version 3

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{OLCS}_3(i, j)$ 
2:    $c_2 \leftarrow \text{OLCS}_3(i - 1, j)$ 
3:    $c_1 \leftarrow \text{OLCS}_3(i, j - 1)$ 
4:    $c_3 \leftarrow \text{OLCS}_3(i - 1, j - 1) + (X_i = Y_j)$ 
5:   return  $\max(c_1, c_2, c_3)$ 

```

Algorithm 20 Oblivious LCS Cost Version 4

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{OLCS}_4(i, j)$ 
2:    $c_2 \leftarrow \text{OLCS}_4(i - 1, j)$ 
3:    $c_3 \leftarrow \text{OLCS}_4(i - 1, j - 1) + (X_i = Y_j)$ 
4:    $c_1 \leftarrow \text{OLCS}_4(i, j - 1)$ 
5:   return  $\max(c_1, c_2, c_3)$ 

```

Algorithm 21 Oblivious LCS Cost Version 5

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{OLCS}_5(i, j)$ 
2:    $c_3 \leftarrow \text{OLCS}_5(i - 1, j - 1) + (X_i = Y_j)$ 
3:    $c_1 \leftarrow \text{OLCS}_5(i, j - 1)$ 
4:    $c_2 \leftarrow \text{OLCS}_5(i - 1, j)$ 
5:   return  $\max(c_1, c_2, c_3)$ 

```

Algorithm 22 Oblivious LCS Cost Version 6

Input: index i into X , and index j into Y

Output: the length of the longest subsequence common to X and Y

```

1: procedure  $\text{OLCS}_6(i, j)$ 
2:    $c_3 \leftarrow \text{OLCS}_6(i - 1, j - 1) + (X_i = Y_j)$ 
3:    $c_2 \leftarrow \text{OLCS}_6(i - 1, j)$ 
4:    $c_1 \leftarrow \text{OLCS}_6(i, j - 1)$ 
5:   return  $\max(c_1, c_2, c_3)$ 

```

4.4.3 Traceback Function

The algorithm for the traceback of the oblivious version follows the same format as in algorithm 23, except that it calls OLCS_C instead of LCS_C , which

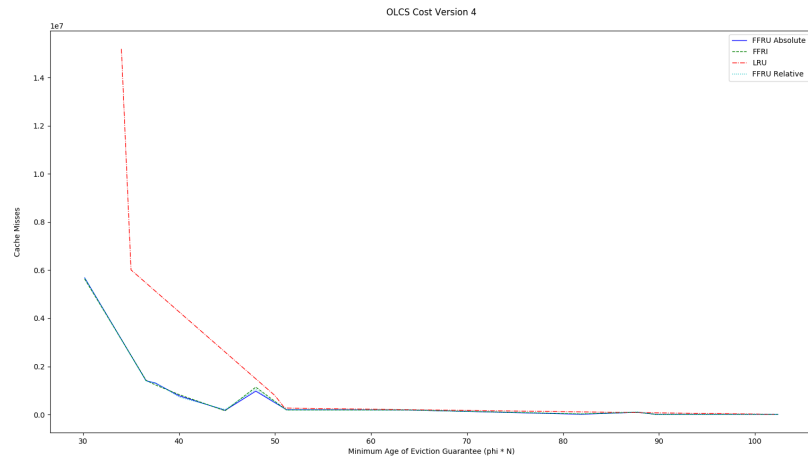


Figure 4.11: OLCS Cost Version 4, comparison between LRU and FFRU

refers to whichever version of the cost function is being used. Bounding the growth of the critical cache, \hat{C} , gives us a sense of how hard a memoized function is, since it indicates how much memory is needed before additional cache misses would occur due to eviction. Figure 4.4.3 shows the pattern of growth when we measure the critical cache size for problem instances having the form given by observations 4.26 and 4.27.

Observation 4.26 For $n \geq 10$, when using version 2 of LCS with LRU caching and X has the form $0^a 1^a 2^{n-2a}$, $a = \lceil n/3 \rceil$, and Y is the reverse of X , the maximum $\hat{C}(LCS_T(n)) \leq 0.611 \cdot n^2 + 1.456n - 3.965$.

Observation 4.27 For $n \geq 10$, when using version 2 of LCS with LRU caching and X has the form $0^a 1^a 2^{n-2a}$, $a = \lceil n/3 \rceil$, and Y is the reverse of X , the maximum $\hat{C}(LCS_T(n)) \geq 0.611 \cdot n^2 + 0.210n - 6.457$.

Given the pattern of growth observed in figure 4.4.3, we obtain in the above observations upper and lower bounds on the *maximum* critical cache size, because what we're bounding is the running maximum.

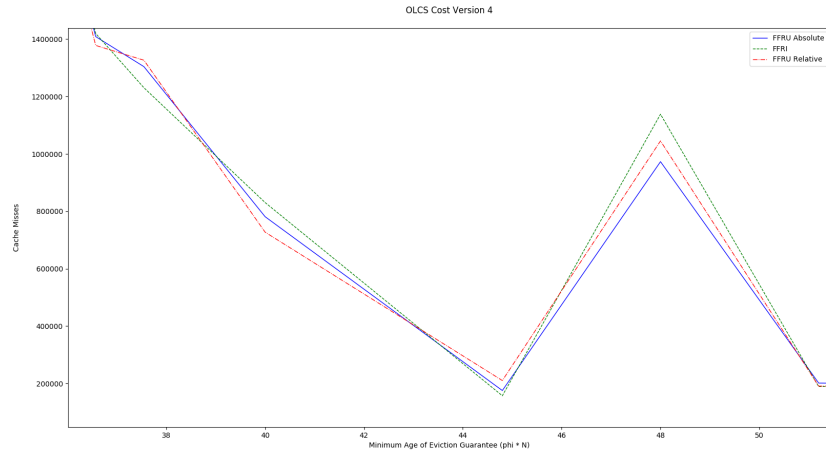


Figure 4.12: OLCS Cost Version 4, FFRU all three variants

We compare LRU to FFRU for LCS version traceback in figure 4.4.4, using the same hard input type as in the above observations, and we find as expected that for a large FFRU is able to solve these instances using much less memory. It should be pointed out that the memory overhead incurred in the various FFRU configurations tested varies greatly. But the advantage gained in terms of reduced cache misses seems not to be directly impacted by the degree of memory overhead incurred. As we will discuss further down, by varying the parameters to FFRU it is possible to achieve vastly different performance guarantees, each with its corresponding tradeoffs in terms of load factor, which in turn impacts the amortized cost of insertions and updates. When we zoom in here to the places where the differences between the FFRU variants occur in figure 4.4.4, we would expect that if many re-references happened, then FFRU would be better than FFRI.

Observation 4.28 For $n \geq 0$, when X has the form $0^a 1^a 2^{n-2a}$, $a = \lceil n/3 \rceil$, Y is the reverse of X , and $\phi_{LRU} N_{LRU} \equiv \phi_{FFRU} N_{FFRU}$, then $M_{FFRU}(LCS_{2T}(n)) \leq M_{LRU}(LCS_{2T}(n))$.

We present the OLCS traceback bound for version 4, i.e. algorithm 20,

Table 4.3: FFRU Parameters used with OLCS Cost Version 4.

ϕN	cache size (N)	κ	d	a
30.17142857	64	9	7	0.1285714286
36.57142857	64	10	7	0.1285714286
37.54666667	64	8	5	0.18
40	64	10	6	0.15
44.8	128	3	2	0.45
48	128	5	4	0.225
51.2	128	4	3	0.3
64	256	7	6	0.15
81.92	256	6	5	0.18
87.77142857	512	8	7	0.1285714286
89.6	256	3	2	0.45
96	256	5	4	0.225
102.4	256	4	3	0.3

since it was the best performing of the six variants in the previous section. Figure 4.4.4 shows the growth of the critical cache size as problem sizes increase.

Observation 4.29 For $n \geq 2$, when solving OLCS traceback version 4 with LRU caching and X has the form $0^a 1^a 2^{n-2a}$, $a = \lceil n/3 \rceil$, and Y is the reverse of X , when $M_{LRU}(OLCS_T(n)) = n^2 + 2n$, then $C \leq 0.667 \cdot n^2 + 1.686n - 0.208$.

Observation 4.30 For $n \geq 0$, regardless of input type, and $\phi_{LRU} N_{LRU} \equiv \phi_{FFRU} N_{FFRU}$, $M_{FFRU}(OLCS_T(n)) \leq M_{LRU}(OLCS_T(n))$.

4.4.4 Conclusions

To summarize we note that when computing the cost function, the oblivious variant exhibits the same improved cache miss performance regardless of input type. Conventional LCS instead incurs vastly different amounts of cache misses depending on the type of input sequences. The “hard” input types we discovered share the characteristic that the number of repeated symbols are a function of the cache size. However, contrary to our expectation the total

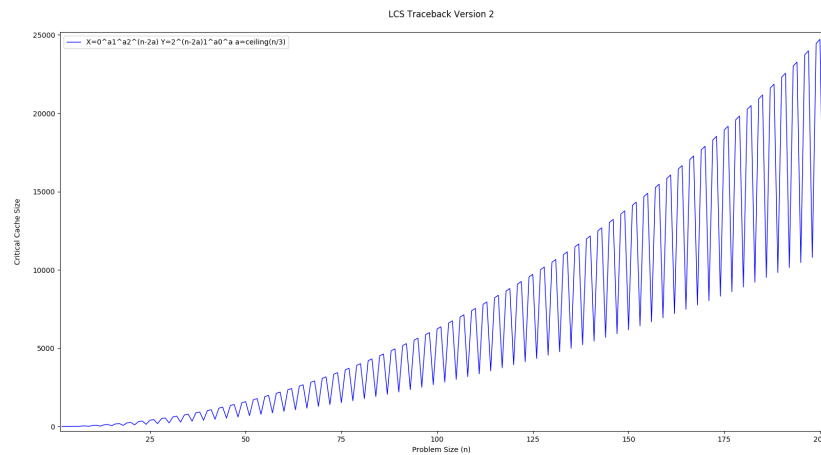


Figure 4.13: LCS Traceback Version 2, critical cache size as a function of problem size

number of subproblems, indicated by the *critical cache size*, \hat{C} , is considerably less than n^2 .

From the standpoint of sparseness of subproblems, this problem also exhibits strong overlap between FFRI and FFRU, owing to the fact that while attempting to find long subsequences, the LCS cost function is not encountering the same subproblem often. Nevertheless, the superiority of FFRU over LRU is once again demonstrated by the above performance comparisons.

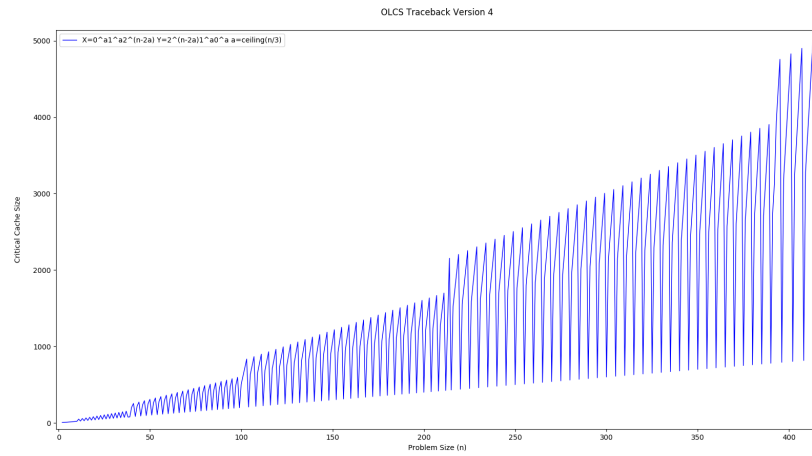


Figure 4.14: OLCS Traceback Version 4, critical cache size as a function of problem size

Algorithm 23 LCS Traceback

Input: $X = (x_0x_2\dots x_{n_1-1})$, $Y = (y_1y_2\dots y_{n_2-1})$

Output: a subsequence common to X and Y having maximal length

```

1: procedure  $\text{LCS}_T(X,Y)$ 
2:    $i \leftarrow |X| - 1$ 
3:    $j \leftarrow |Y| - 1$ 
4:    $S \leftarrow \emptyset$ 
5:   while  $i \geq 0$  and  $j \geq 0$  do
6:     if  $X_i = Y_j$  then
7:       append  $X_i$  to  $S$ 
8:        $i \leftarrow i - 1$ 
9:        $j \leftarrow j - 1$ 
10:    else if  $\text{LCS}_C(i - 1, j) > \text{LCS}_C(i, j - 1)$  then
11:       $i \leftarrow i - 1$ 
12:    else
13:       $j \leftarrow j - 1$ 
14:  return  $S$ 

```

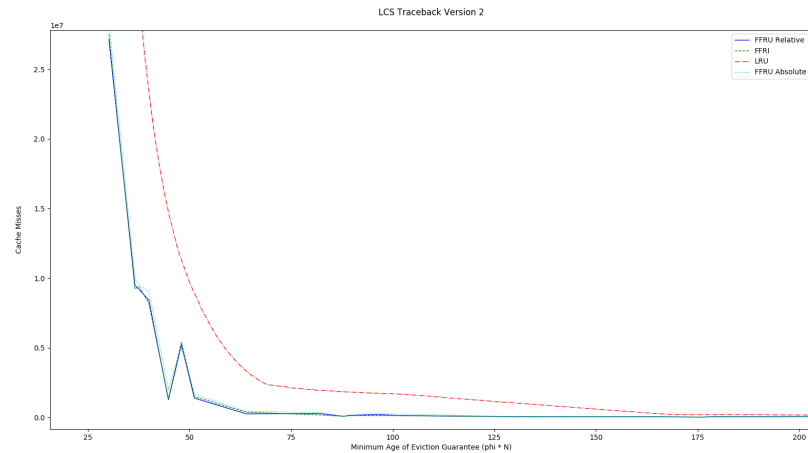


Figure 4.15: LCS Traceback Version 2, comparison between LRU and FFRU

4.5 Arora's Algorithm using a100

When we solved TSP instances using Arora's algorithm we observe a growth rate of cache misses using LRU that grows exponentially as the cache size decreases. In this section we present a few representative illustrations which show the comparative performance of LRU and the three variants of FFRU.

4.5.1 2 Points per Border, all subproblems

In figure 4.5.1 we show a comparison of cache miss performance when using a100 to solve burma14 using 2 portal points per border and covering all encountered subproblems in each partition. We observe a similar advantageous difference in cache misses here as with the previous problems studied. When we zoom in just the three FFRU variants in figure 4.5.1, we finally notice the effect of re-referencing occurring, because FFRI does slightly worse than the two FFRU variants.

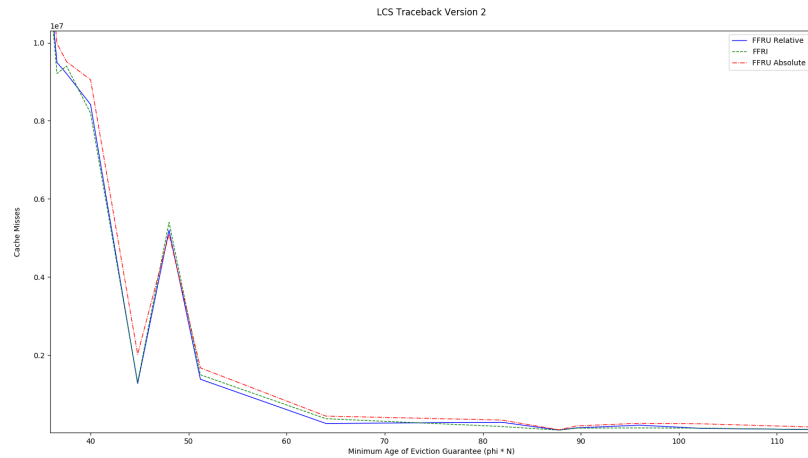


Figure 4.16: LCS Traceback Version 2, FFRU all three variants

4.5.2 1 Point per Border, all subproblems

Likewise in figure 4.5.2 we see slightly different behavior between FFRI and the two variants of FFRU (Absolute and Relative). Notice that at a few points FFRI did slightly worse than LRU, and this doesn't surprise us, because we've seen with many instances that Arora's algorithm revisits subproblems plenty of times and the re-referencing of those proves advantageous for the variants that update timestamps upon access. Figure 4.5.2 shows this more clearly, namely the clear advantage the FFRU has over FFRI due to the re-referencing of subproblems by this algorithm.

4.5.3 2 Points per Border, limited subproblems

Figure 4.5.3 shows a 2319 point instance, u2319, being solved for a range of effective cache sizes. This third case shows the instance being solved using our fastest a100 configuration, which limits how many total subproblems are visited. The important takeaway in these cases is that we see FFRU outperforming LRU in terms of cache misses when the effective eviction age

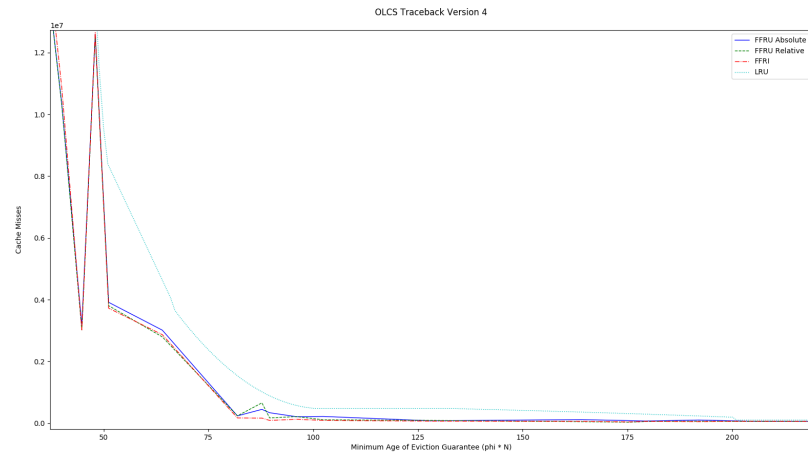


Figure 4.17: OLCS Traceback Version 4, comparison between LRU and FFRU

guarantees are comparable. In this scenario an additional observation we make is that when we look more closely at the comparative performance of the three variants of FFRU in figure 4.5.3, we see at many points FFRU Absolute is the clear winner over FFRU Relative and FFRU. This suggests that in this configuration scenario of **a100**, certain subproblems are being re-referenced but not in short repeated bursts.

4.5.4 Conclusions

In the final section of this chapter we studied the cache miss performance of our memoized implementation of Arora’s PTAS, the implementation of which we describe in the following chapter. That algorithm can be configured by choosing the number of portal points per border. The first two subsections above show results for 2 portals per border and 1 portal per border respectively. The remaining subsection shows results for a configuration option whereby few subproblems per partition are invoked. As the cache size decreases there is a clear exponential increase in the number of cache misses incurred. In all three cases FFRU clearly outperforms LRU.

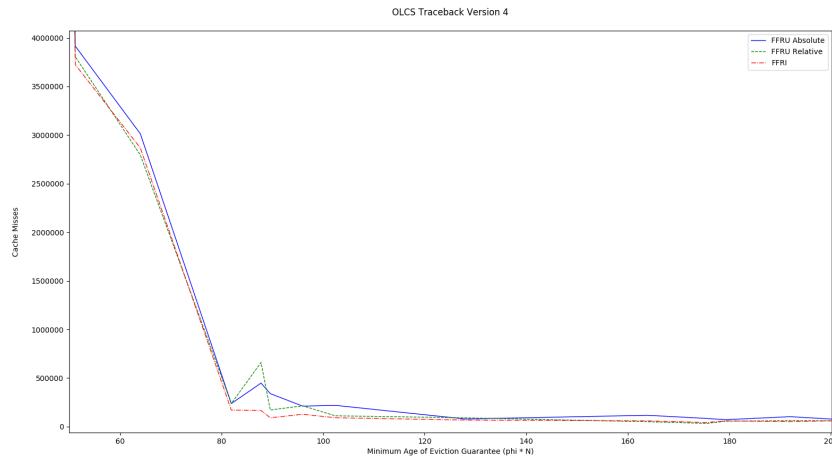


Figure 4.18: OLCS Traceback Version 4, FFRU all three variants

The expectation is that FFRU never incurs more cache misses than LRU, and we observe confirmation of this expectation in every problem scenario explored. As opposed to the previous problems, Arora’s PTAS shows for a few cache sizes that the cache misses incurred by FFRI are slightly higher than those with LRU. This to be expected in certain cases, because it indicates that for the cache size and problem size in question, there were certain retrievals repeatedly encountering cache misses.

We have investigated the performance of FFRU in a limited number of application scenarios, and we have observed a clear cache miss performance advantage with FFRU compared to LRU. However, we acknowledge that we are not yet able to make a more general claim, like $M_{FFRU}(P) \leq M_{LRU}(P)$ for all problems P . In particular comparing LRU’s performance to FFRU’s performance in a wide variety of industry and theoretical settings is still needed before being able to establish the suitability of this caching algorithm.

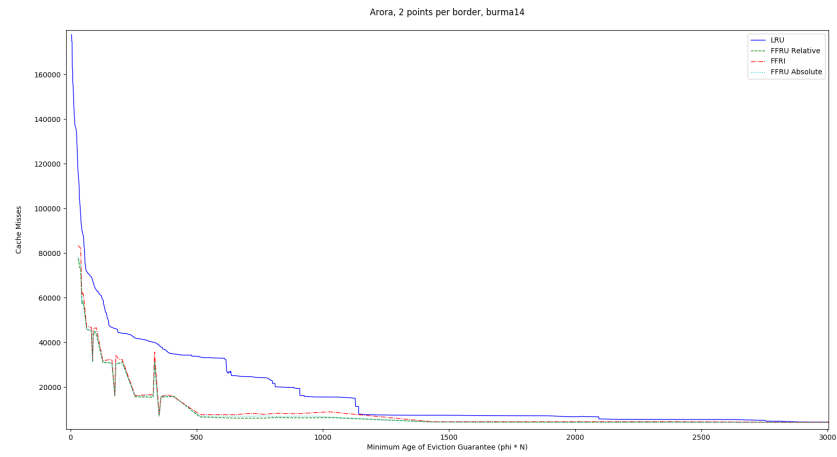


Figure 4.19: Arora, *burma14*, 2 points per border, comparison between LRU and FFRU

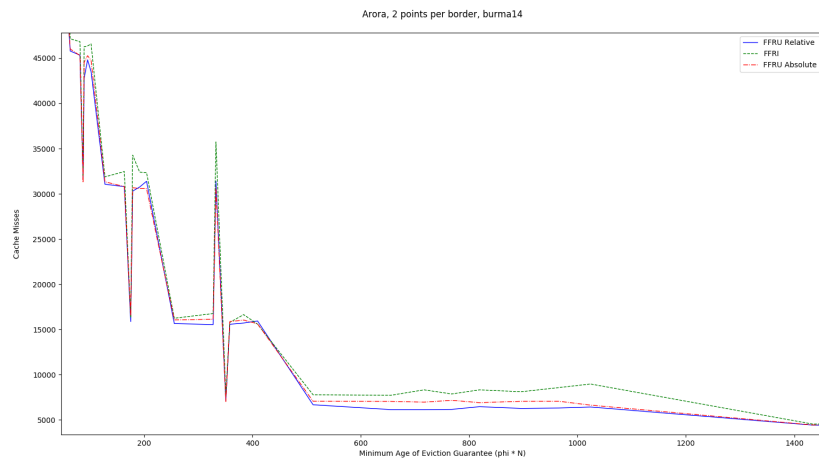


Figure 4.20: Arora, *burma14*, 2 points per border, FFRU all three variants

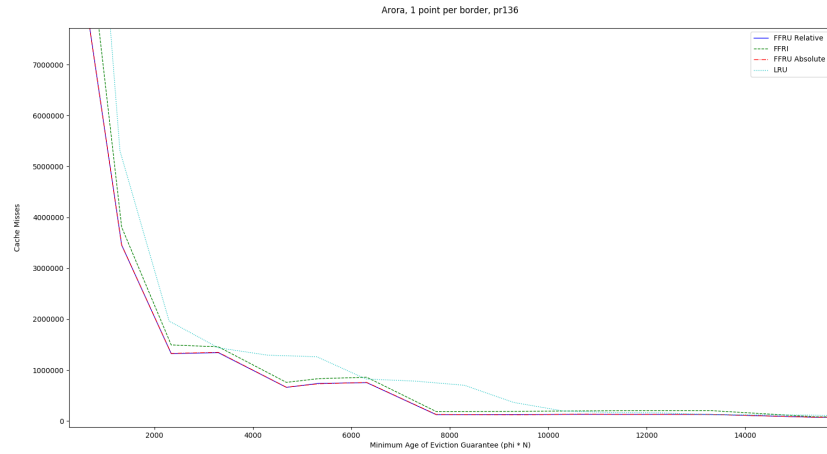


Figure 4.21: Arora, *pr136*, 1 point per border, comparison between LRU and FFRU

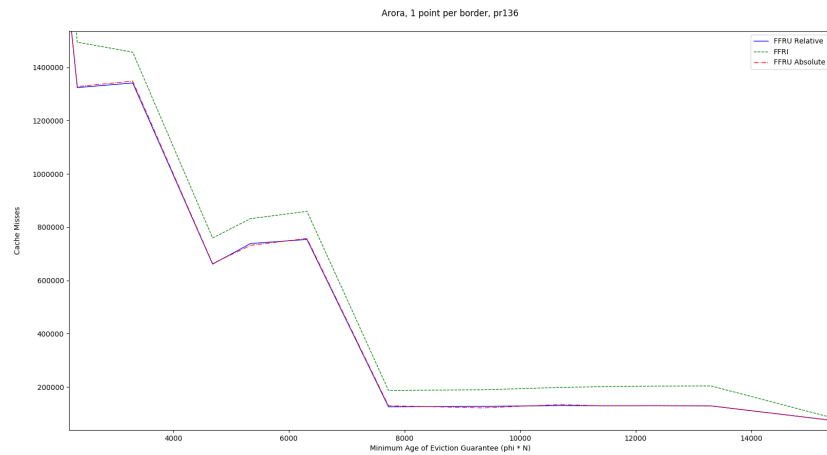


Figure 4.22: Arora, *pr136*, 1 point per border, FFRU all three variants

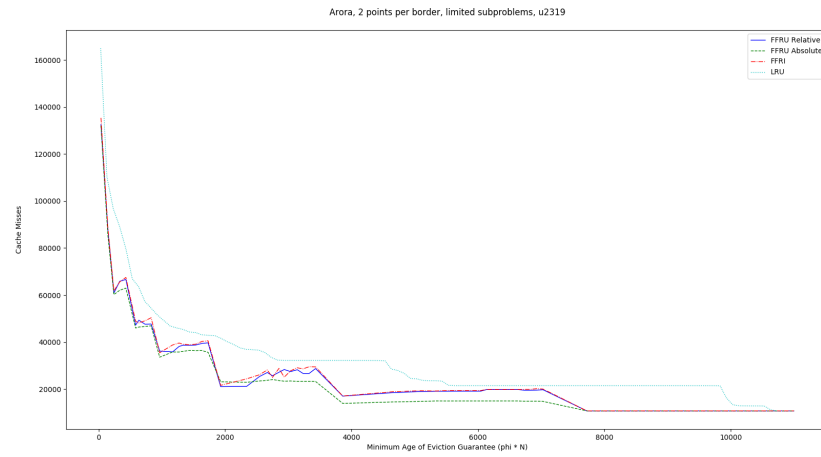


Figure 4.23: Arora, *u2319*, 2 points per border, limited subproblems, comparison between LRU and FFRU

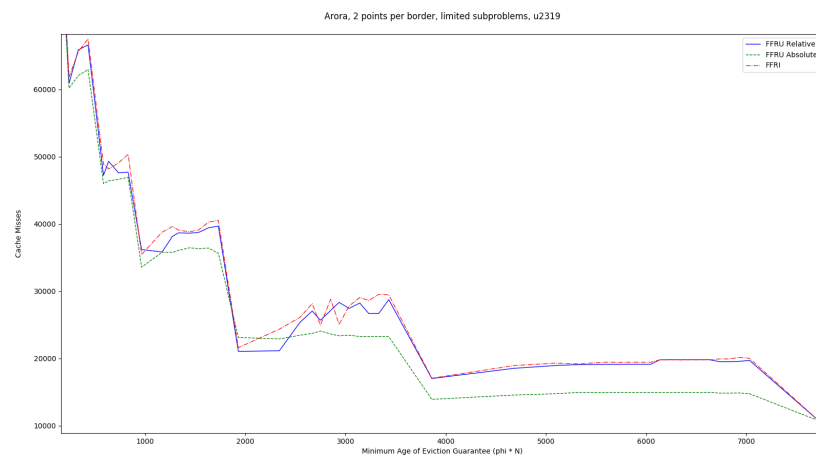


Figure 4.24: Arora, *u2319*, 2 points per border, limited subproblems, FFRU all three variants

CHAPTER 5

IMPLEMENTING ARORA'S TSP PTAS

If you don't know where you're
going, any road will take you there.

George Harrison — *Any Road*

In this section we review how Arora's algorithm works and explain the details of various design decisions taken in our implementation of this algorithm, which we affectionately call **a100**¹. Given a sequence of points $P = (p_1, p_2, \dots, p_n)$ Arora's algorithm returns an ordering of that sequence $T = (P[t_1], P[t_2], \dots, P[t_n])$ such that the sum

$$\sum_{j=1}^{n-1} d(T_j, T_{j+1}) + d(T_n, T_1) \tag{5.1}$$

has minimal length, where $d(p_1, p_2)$ is the Euclidean distance. Algorithm 24 lists the five main stages of Arora's PTAS. The following sections expand on the procedures mentioned therein.

¹The **a** obviously stands for "Arora", and the 100 refers to the author, who implemented a100, and whose name is featured on the one hundred dollar bill.

Algorithm 24 High level procedure for Arora’s PTAS

Input: $P = (p_1, p_2, \dots, p_n)$, $p_i \in \mathbb{R}^2$, input data points

Output: $T = (P[t_1], P[t_2], \dots, P[t_n])$, a tour P having minimal length

- 1: **procedure SOLVE – ARORA**(P)
 - 2: $\hat{P} \leftarrow$ **PERTURB – DATA – POINTS**(P)
 - 3: $G \leftarrow$ **PARTITION – INSTANCE**(\hat{P})
 - 4: $G_p \leftarrow$ **PORTALIZE**(G)
 - 5: $S_{best} \leftarrow$ **GET – BEST – DISTANCE**(G_p)
 - 6: $T \leftarrow$ **GET – BEST – TOUR**(R_p, S_{best})
 - 7: **return** T
-

5.1 Perturbing Input Points

The first step involves re-scaling and slightly moving each data point as detailed in algorithm 25. The original input points P are assumed to be taken from \mathbb{R}^2 , however Arora’s algorithm needs to operate on integer values. We must also ensure that each nonzero inter-node distance is at least 8 units, and that the maximum inter-node distance is $O(n)$. Additionally, given a design decision related to capping the number of portal points to two for this implementation (see discussions below about portal points), we ensure all data points are not multiples of three, so that they never occur along partition lines. As a consequence no leaf partition has width less than 3, in order to accommodate one or two portal points along any edge of a partition².

²We discuss later in the chapter the importance of limiting the number of portals per border when the schema tables are explained.

Algorithm 25 Perturbing and re-scaling input points

Input: $P = (p_1, p_2, \dots, p_n)$, $p_i \in \mathbb{R}^2$, input data points

Output: $\hat{P} = (\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n)$, $\hat{p}_i \in \mathbb{Z}^2$ perturbed and re-scaled input points

- 1: **procedure** **PERTURB – DATA – POINTS**(P)
 - 2: $scaling_constant \leftarrow 192$
 - 3: $min_dist \leftarrow \text{MIN}(\text{DIST}_{1 \leq j < k \leq n}(p_j, p_k))$
 - 4: $scaling_factor \leftarrow \text{MIN}(1, scaling_constant/min_dist)$
 - 5: $rescaled_real_points \leftarrow (1 \leq j \leq n | P_j \cdot scaling_factor)$
 - 6: $integer_points \leftarrow \text{ROUND}(1 \leq j \leq n | rescaled_real_points_j)$
 - 7: **return** $integer_points$
-

After this step is complete, the perturbed points must lie on a grid of even values. The effect of this transformation of data points was shown in [Arora, 1998] to introduce a degree of error into the final solution that is bounded by a constant additive. This procedure can be completed in $O(n)$ time. The *scaling_constant* used in algorithm 25 was chosen because it yields decent performance in practice on the instances we studied. Generalizing our perturbation strategy to do away with such hard-coded constants is an area where our implementation could be improved.

5.2 Partitioning Strategies

The second step in algorithm 24 consists of partitioning the space where the perturbed points lie. In [Arora, 1998] Arora discusses partitioning the space into a quad tree by dividing every bounding box into four subdivisions. Arora refines this approach and discusses slicing every bounding box in two to form a binary tree. In our implementation we cut every partition into two pieces, because it greatly simplifies the creation of the table used in the dynamic programming procedure discussed below. The procedure **PARTITION-INSTANCE** returns a tree, $G = (V, E)$, of partitions, where each vertex V is a bounding box and each vertex has two edges pointing to

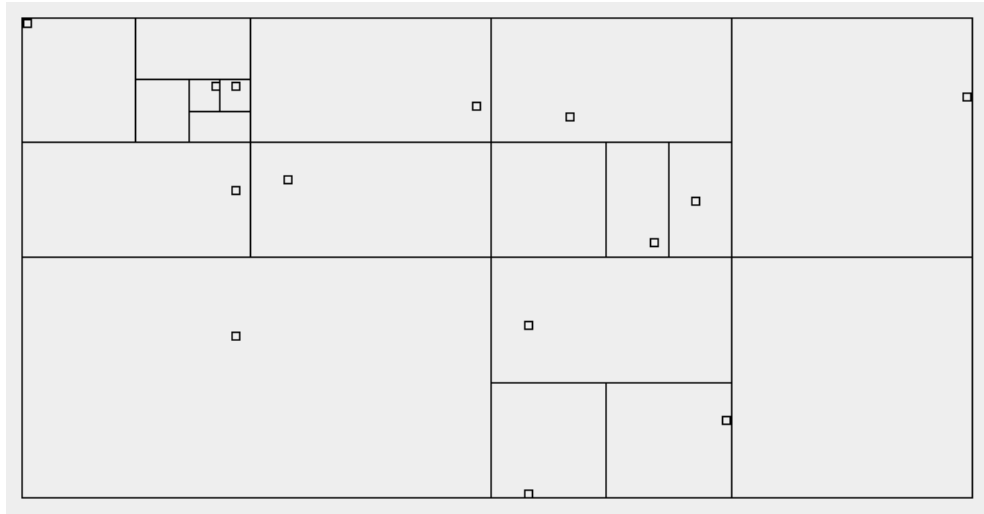


Figure 5.1: A Partitioned Instance

left and right child partitions.

We found no noticeable advantage to varying the amount by which we cut partitions, eg. exactly at the midpoint versus a $1/3$ - $2/3$ split, therefore we cut each one exactly in half, selecting a vertical or horizontal cut depending on which whether the partition in question is landscape or portrait. One consequence of this design decision is the need to account for landscape versus portrait partitions in the schema table, the creation of which is discussed below.

As child partitions are formed from parent partitions, the number of data points contained in them decreases until the partitions at the lowest level of the tree contain either zero or one data point. One clear advantage however was gained by adjusting the partitioning strategy to ensure that no leaf partition was empty of data points. Doing this greatly reduces run time. As such we guarantee exactly n leaf partitions, resulting in $2n - 1$ total partitions. Since each partition takes a constant amount of time to create, this portion of the algorithm can be completed in $O(n)$ time. Figure 5.1 illustrates an instance after the data points have been partitioned.

5.3 Portal Points

After the space is partitioned, the third step in algorithm 24 consists of setting portal points along the borders between partitions. The purpose of these portal points is that as we consider various tours of the data points, we constrain the tour to pass through the portal points that have been placed along the edges of the partitions. We define a **salesman path** to be a sequence of path points, with perturbed data points and portal points occurring in alternating order. Instead of directly finding the shortest tour of the original data points, the algorithm finds a salesman path of minimal length. Having found the shortest salesman path, a tour can be formed by removing the portal points. The number of portal points placed has an impact over the error introduced by having tour deviate to pass through them. The fewer portal points are used, the greater the chance that error will be introduced into the final solution. However, as greater portal points are used, the algorithm takes longer to terminate, because it must consider every combination of portal points as it searches for a shortest tour. In practice we found that little was gained in terms of solution quality when two points per border are chosen instead of one. Since the number of partitions is always less than $4n$ and it takes constant time to place portal points along the partition edge, so this portion of the algorithm takes $O(n)$ time. Figure 5.2 illustrates an instance after the portal points have been set along the borders of the partitions. In this figure we place two portal points along each partition border. We note that certain partitions have more than two portal points placed along a given edge, and this is because those partitions border more than one partition along that edge.

5.4 Cost Function

The fourth step in algorithm 24 consists of determining the length of the shortest salesman path and is where the real work of Arora's algorithm is ac-

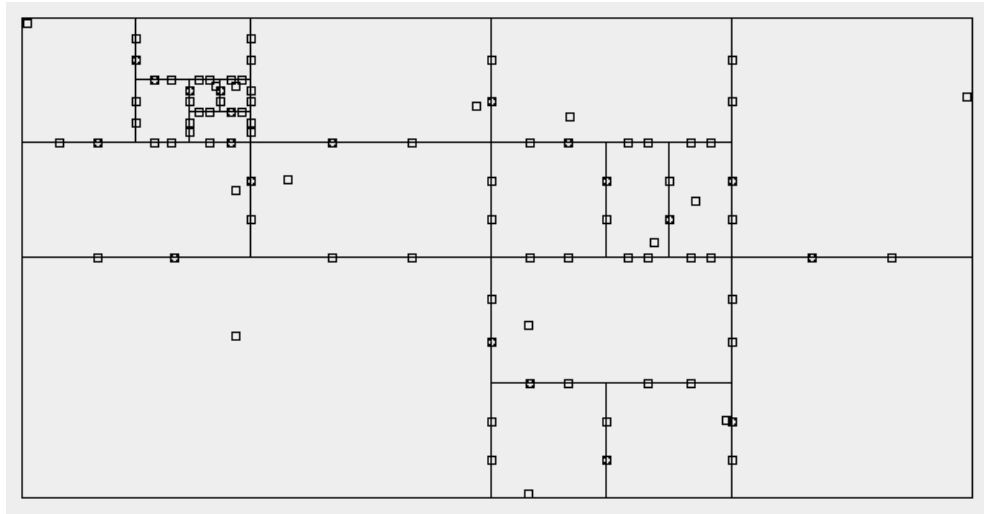


Figure 5.2: A Partitioned Instance with Portal Points

completed. Algorithm 26 outlines the principle steps involved in determining the cost of the shortest salesman path. It recursively solves subproblems that are defined in terms of the portal points placed along the edges of the partition in question. For a given partition, for a given subset of the portals associated with that partition, and for a given order of those portal points, referred to below as pairings, the algorithm determines the length of the shortest salesman path segment set that utilizes the given portal points. We refer to the subset of portal points as pairings because their order matters. The number of portal points given by pairings is always even, because they are considered to be entry and exit points to the partition.

We recall that \hat{G} is the entire portalized and partitioned problem instance, which as described above cuts every region into a left and a right sub-region. The top level recursive calls to **GET-BEST**, then, consist of considering every combination of sibling portal points adjoining the left and right partitions. It takes as input the portalized partition tree \hat{G} and returns a sequence S_{best} consisting of k indices, s_1, s_2, \dots, s_k which indicate the top level portal points that are used to form the shortest tour. In our implementation $k \in 2, 4$ but

it can be any even number, since it indicates which top level portals the best tour enters and leaves. The objective of this step is to find the distance of the shortest portal respecting salesman tour, i.e., the tour of all data points that passes through portal points as it crosses partition boundaries. Knowing the distance is not what the final step needs in order to find actual shortest salesman tour. Instead, knowing which top level portal points lead to the best portal respecting tour allows the final step to consult the memoization table and reconstruct the sequence of subproblems that find the shortest tour. In terms of run time this step requires reaching leaf nodes from the root of the tree. There are $O(n)$ leaves. Reaching a leaf node requires $\log(n)$ steps, which is the height of a binary tree having n leaves. Solving the base case at the leaf level takes constant time. At each node in the tree there are $c = \frac{1}{\epsilon}$ subproblems, and this constant depends on how many portal points are at each node. Therefore, this step of the algorithm takes $O(n(\log n)^c) = O(n(\log n)^{\frac{1}{\epsilon}})$ steps to complete.

Algorithm 26 Recursive Cost Function

Input: P is the current partition, $R = \{r_0..r_k\}$ are the current portal pairings

Output: the length of the segment set corresponding to the shortest salesman path passing through P and utilizing the pairings R

```

1: procedure GET – BEST( $P, R$ )
2:   if  $P$  is a leaf node then
3:     return LEAF-COST( $P, R$ )
4:   else
5:     for each valid parent-child mapping  $m_i \in \Psi$  do
6:       for each valid combination of sibling portals  $c_j$  do
7:          $(R_{left}, R_{right}) \leftarrow$  GET – CHILD – INDICES( $R, m_i, c_j$ )
8:          $cost_{left} \leftarrow$  GET – BEST( $P_{left}, R_{left}$ )
9:          $cost_{right} \leftarrow$  GET – BEST( $P_{right}, R_{right}$ )
10:         $cost_{i,j} \leftarrow cost_{left} + cost_{right}$ 
11:   return min( $cost$ )

```

5.4.1 Solving the Base Case - Leaf Partitions

If the partition is a leaf, then determining the minimum cost is straightforward, although there are several cases to consider, based on how many pairings are given and whether the leaf contains a data point. If there is only one pairing, i.e. two portal points chosen, p_1 and p_2 , and one data point p_d , **LEAF-COST** returns the sum of their distances, namely

$$d(p_1, p_d) + d(p_d, p_2) \quad (5.2)$$

If there is more than one pairing, then one must consider all of the possible ways of connecting the portals of those pairings with the data point, and the number of possible cases to consider is proportional to the number of pairings given. In the case of two pairings and one data point, there are two cases to consider. To illustrate this, let the pairings be (p_1, p_2) , (p_3, p_4) , and let the data be p_d . Then, **LEAF-COST** returns $\min(d_1, d_2)$ where

$$d_1 = d(p_1, p_d) + d(p_d, p_2) + d(p_3, p_4) \quad (5.3)$$

$$d_2 = d(p_1, p_2) + d(p_3, p_d) + d(p_d, p_4) \quad (5.4)$$

Likewise, three pairings necessitate a three-way comparison using the same reasoning, and **LEAF-COST** returns $\min(d_1, d_2, d_3)$ where

$$d_1 = d(p_1, p_d) + d(p_d, p_2) + d(p_3, p_4) + d(p_5, p_6) \quad (5.5)$$

$$d_2 = d(p_1, p_2) + d(p_2, p_d) + d(p_d, p_3) + d(p_4, p_5) \quad (5.6)$$

$$d_3 = d(p_1, p_2) + d(p_3, p_4) + d(p_5, p_d) + d(p_d, p_6) \quad (5.7)$$

The four-pairing case follows the same pattern, namely **LEAF-COST** returns $\min(d_1, d_2, d_3, d_4)$ where

$$d_1 = d(p_1, p_d) + d(p_d, p_2) + d(p_3, p_4) + d(p_5, p_6) + d(p_7, p_8) \quad (5.8)$$

$$d_2 = d(p_1, p_2) + d(p_3, p_d) + d(p_d, p_4) + d(p_5, p_6) + d(p_7, p_8) \quad (5.9)$$

$$d_3 = d(p_1, p_2) + d(p_3, p_4) + d(p_5, p_d) + d(p_d, p_6) + d(p_7, p_8) \quad (5.10)$$

$$d_4 = d(p_1, p_2) + d(p_3, p_4) + d(p_5, p_6) + d(p_7, p_d) + d(p_d, p_8) \quad (5.11)$$

5.4.2 Solving non-base cases

In algorithm 26 above, when the subproblem involves a partition that is not a leaf partition, two sets of recursive subproblems are generated, because we divide every partition into two child partitions. A subproblem is identified by partition P and a set of portal pairings R .

Portal pairings

One pairing in R is a 2-tuple of portal points. The pairings of portal points at which the candidate tour segment enters and leaves the partition in question can be represented as:

$$R_{parent} = \{(p_{1,1}, p_{1,2}), (p_{2,1}, p_{2,2}), \dots, (p_{k,1}, p_{k,2})\} \quad (5.12)$$

Note that the ordering of the pairings within R , as well as the ordering of portals in each pairing, are both immaterial. What is important is topology that the pairings indicate. Nevertheless in order to manage the task of mapping parent pairings to child pairings, we adopt a canonical ordering by sorting pairings by lowest portal index and by sorting tuples similarly.

Local versus global portal indices

Each $p_{i,j}$ in R is simply an index indicating a portal point within the partition. The values of k are constrained as indicated above because we limit the number of times the tour can cross any partition side to two crossings. This limits the possible number of pairings of a subproblem to four pairings. Let the values shown in figure 5.3 be called **global indices** that identify the portal points across all partitions. For convenience we adopt the convention of numbering the portal points clockwise starting from the upper left corner. Given the example just mentioned, the **local indices** for the partition depicted would be

$$(0, 1, 2, \dots, 17) \quad (5.13)$$

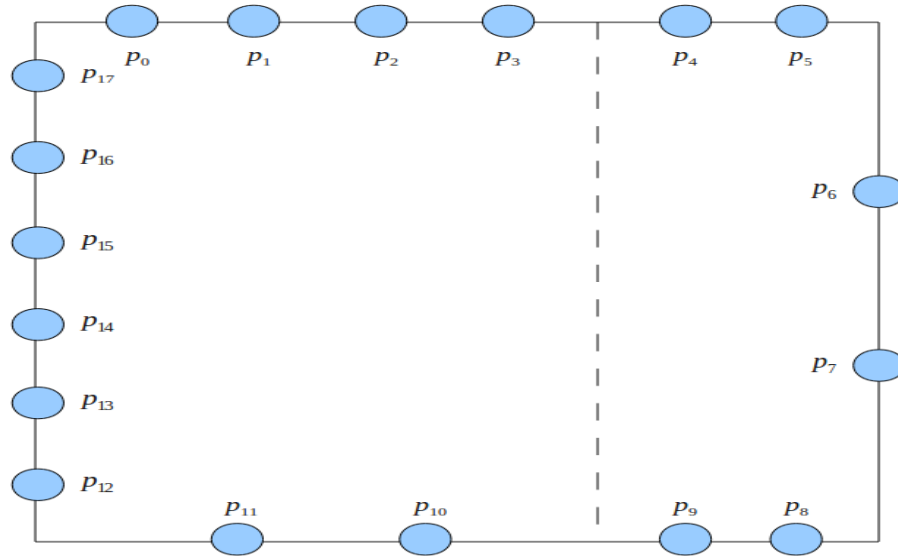


Figure 5.3: Partition and portal points with local indices

However, in general it isn't always necessarily the case that the indices match the values as this example shows. This local portal indexing convention allows for a pairing at any level in the partition hierarchy to be unambiguously expressed. In figure 5.4 we show the portal points which two child partitions share, and we denote these portal points as **sibling portal points**. Using the convention just described, we can extend it to allow the unambiguous indexing of the sibling points also. According to that convention the local indices for the left child partition are:

$$(0, 1, 2, 3, 18, 19, 20, 21, 10, 11, 12, 13, 14, 15, 16, 17) \quad (5.14)$$

And the local indices for the right child partition are:

$$(4, 5, 6, 7, 8, 9, 21, 20, 19, 18) \quad (5.15)$$

Ultimately the importance between the two systems can be summed up as follows. Since portal points are only stored one time in memory, we need global indices to unambiguously identify them across all partitions. However,

each subproblem must determine the portal pairings of the child partitions (discussed below) by means of a precomputed table, which necessitates referring to the portals from the standpoint of their relative position within the partition in question.

Determining child pairings using the Canonical Schema Table Ψ

When **GET-BEST** solves a non-base case, it determines the portal pairings for the subproblems of its left and right child partitions. When problem instances are partitioned, the dividing lines between the child partitions can be either vertical or horizontal. Without loss of generality we refer to the two subproblems as *left* and *right* child partitions, regardless of whether the partition orientation is landscape or portrait.

To determine the child portal pairings from the parent pairings is a trivial matter to the naked eye. One can look at the pairings that were chosen, see the sibling partition boundary, and observe easily the topology that would result from applying the given parent pairings to the partition in question. However, in order to automate this process a very large table of parent to child canonical schema mappings is required.

Definition 5.1

- **Canonical schema**, $C = \{\{c_{0,0}, c_{0,1}\}, \dots, \{c_{k,0}, c_{k,1}\}\}$, is a set of sets indicating a portal connection topology within a partition.
- **Canonical schema table**, $\Psi : (C_{parent}, S) \rightarrow \{(C_{i,L}, C_{i,R})\}$, is a mapping from a parent canonical schema C_{parent} and information about where the sibling partition boundary lies, given by S , to a set of valid 2-tuples of child canonical schemata.

A canonical schema serves to encode the topology of which portal points connect to one another within a partition. As such it employs indices that must then be mapped to the local partition indices that are discussed above. As an example consider a partition having four portal points chosen for use

and let their *local partition indices* be p_0, p_1, p_2 , and p_3 . Suppose we wish to connect the first portal with the fourth and the second with the third. The resulting canonical schema that represents this topology is, then, $\{\{0, 3\}, \{1, 2\}\}$. The utility of this encoding is that captures the relevant connection arrangement without regard to how many total portals comprise a given partition. Therefore, canonical schemata can be applied to any partition. The canonical schema table, Ψ , encodes all valid child topology possibilities arising from (a) the parent canonical schema and (b) the location of the sibling partition boundary with respect to the parent portal connections. To explain this concept we'll use some examples.

5.4.3 Subproblems involving one sibling point

Consider the partition illustrated in figure 5.4. It shows a parent partition having 18 portals points and left and right child partitions respectively having 16 and 10 portal points. Suppose for example that the parent portal pairing is $\{\{p_1, p_7\}\}$. Using our definitions above, this corresponds to a canonical schema of $\{\{0, 1\}\}$. In order to determine the child portal connection topology options that result from choosing these parent portals, we also need to know that the first portal lies in the left partition and the second one lies in the right side. Using this information the canonical schema table determines that there is only one valid child topology corresponding to this parent topology, namely $\{\{0, 1\}, \{0, 1\}\}$. As a result the left and right child partitions will have pairings which share one common portal point as shown in figure 5.5. It follows from this example that four subproblems are generated, because there are four sibling portal points in the figure shown, namely p_{18}, p_{19}, p_{20} , and p_{21} .

We notice that in the parent partition the number of crossings is two, i.e. the number of times the salesman tour segment enters or leaves the partition boundaries. Likewise the number of crossings is two in both child partition subproblems. In order for Arora's algorithm to bound the search space, the number of crossings must not exceed a predefined limit. We discuss below the

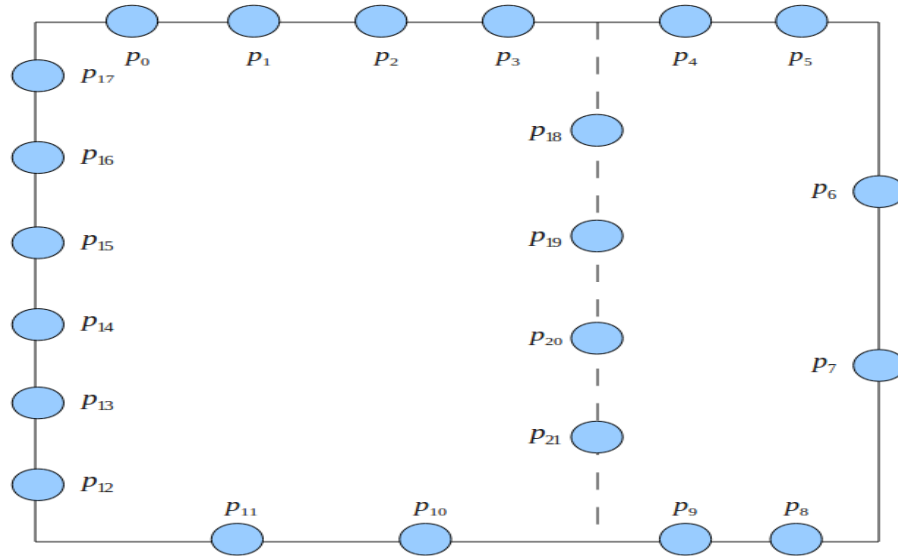


Figure 5.4: Left and right partitions with common sibling points

consequences of allowing the maximum number of crossings to increase, but for now assume that no more than two border crossings are allowed in these examples. Also notice that given the position of the parent portal points in figure 5.5, it is unavoidable to cross the sibling partition boundary. However it is not always the case that the sibling partition boundary is crossed in every generated subproblem, as we illustrate in the following section. Also notice that in figure 5.5, if the parent portal points had instead been p_1 and p_{13} , then the resulting subproblems generated would involve two crossings of the sibling partition boundary, in order to ensure that the resulting salesman tour includes the data points contained in the right side partition.

5.4.4 Subproblems involving two sibling points

As an example of a parent pairing generating multiple child topology options, consider the partition illustrated in figure 5.6. In the illustration the pairings are $\{\{p_6, p_7\}, \{p_{14}, p_{16}\}\}$. Using our definitions above, this corre-

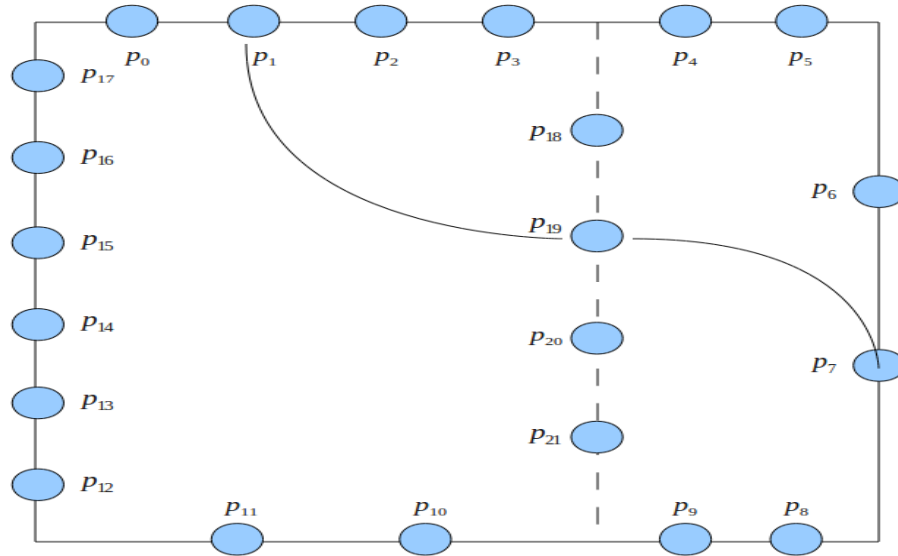


Figure 5.5: Portal pairing involving one sibling point

sponds to a canonical schema of $\{\{0, 1\}, \{2, 3\}\}$. As in the previous example with only the canonical schema we don't know which actual portal points are used. What we need to know is which ones lie to the left of the sibling partition boundary and which ones lie to the right. We see that the first two portals in the given parent pairing lie in the right child partition and the second two portals lie in the left side. Using this information the canonical schema table determines that several topology possibilities result, which are illustrated in the following figures. The first option in figure 5.6 yields left and right child canonical schemata $\{\{0, 1\}\}$ and $\{\{0, 1\}\}$. The second option in figure 5.7 yields left and right child canonical schemata $\{\{0, 1\}, \{2, 3\}\}$ and $\{\{0, 3\}, \{1, 2\}\}$. The third option in figure 5.8 yields left and right child canonical schemata $\{\{0, 3\}, \{1, 2\}\}$ and $\{\{0, 1\}, \{2, 3\}\}$. Note that the first option listed is valid, despite that it uses no sibling portal points. This is because there are portal connections on both sides, which ensures that all data points can be incorporated into the final solution. Since the first option involves no sibling portal points, it follows that only subproblem is generated for that case.

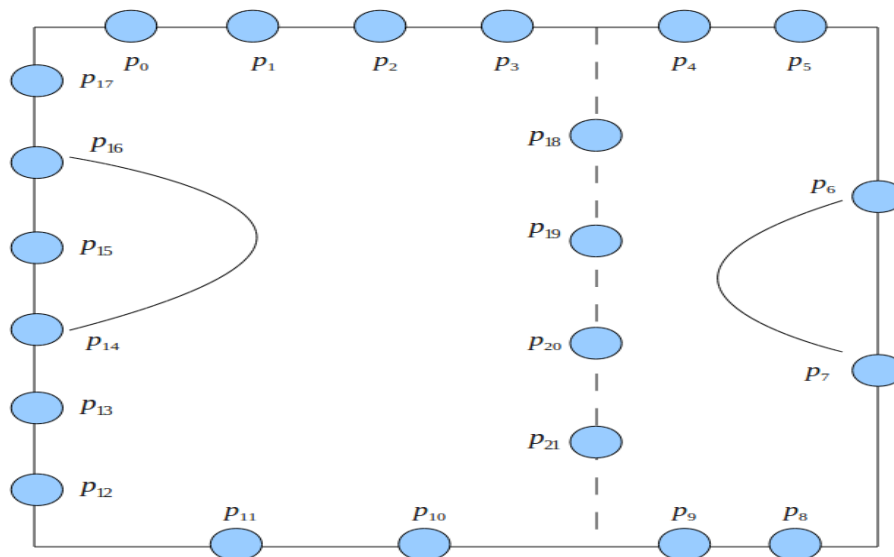


Figure 5.6: Portal pairing involving no sibling points

Figures 5.7 and 5.8 show the other subproblems that occur as a result of this configuration of portal pairings. Note that it is *not* possible for *both* tour segments to cross the sibling partition boundary, because this would result in four total crossings at the sibling boundary, which is above the limit of two used in our implementation. In figures 5.7 and 5.8 no sibling points are depicted, however let there be k siblings shared between these two partitions be. Then there would be

$$\binom{k}{2} \tag{5.16}$$

subproblems generated per child for this given parent pairing.

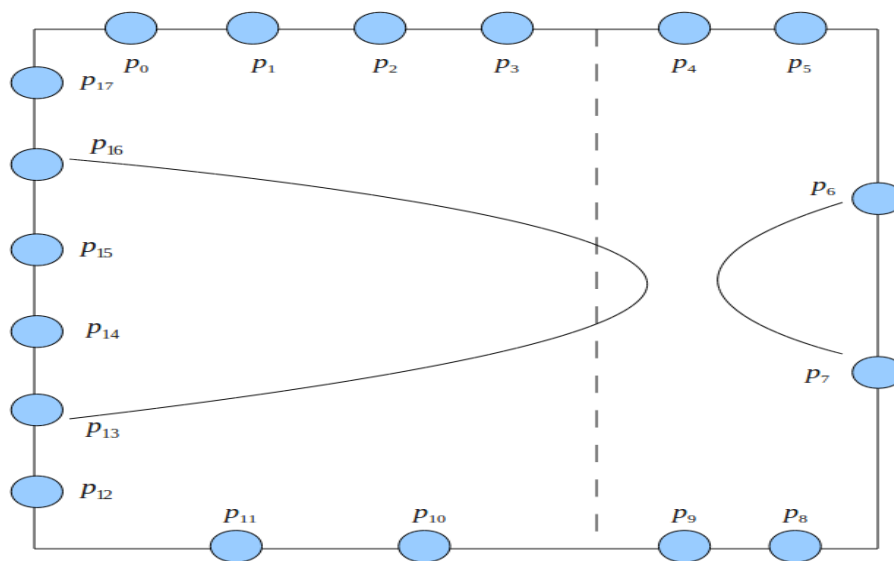


Figure 5.7: Portal pairing involving two sibling points entered from left partition

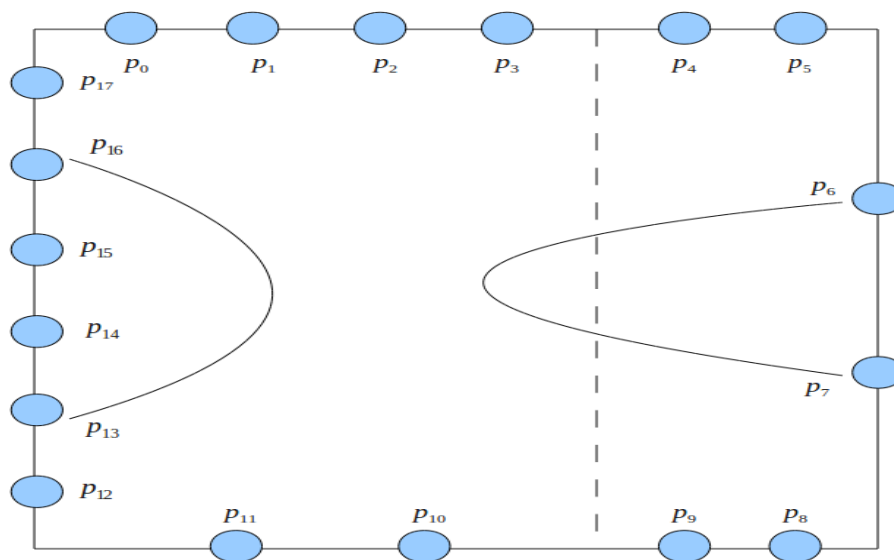


Figure 5.8: Portal pairing involving two sibling points entered from right partition

5.4.5 Dead-end scenarios due to impossible child partition subproblems

In the above scenarios we've noted that the maximum number of crossings per partition border must be two in all cases. This allows for subproblems involving a maximum of four portal pairings, i.e. eight portal points. In this section we examine situations in which a given parent subproblem becomes impossible to solve, due to the limit on the number of allowable crossings per partition border.

In the first such example figure 5.9 shows a parent subproblem involving pairings $\{\{p_2, p_5\}, \{p_7, p_{16}\}, \{p_8, p_{10}\}\}$. Given the topology that these pairings have with respect to the where the sibling partition boundary occurs, three crossings would result, which is not allowed due to the limit imposed.

In the second example figure 5.10 shows a parent subproblem involving pairings $\{\{p_2, p_5\}, \{p_6, p_{15}\}, \{p_7, p_{14}\}, \{p_8, p_{10}\}\}$. Likewise, given the topology that these pairings have with respect to the where the sibling partition boundary occurs, four crossings would result, which is not allowed due to the limit imposed.

5.5 Traceback

The fifth and final step in algorithm 24 consists of determining the **salesman tour** corresponding to the best distance found in the previous step. The sequence S_{best} , together with a memoization table that will have been filled in, is used by the final step, **GET-BEST-TOUR**, to generate the indices of the shortest tour. Since the costs associated with each subproblem are already known and stored in the memoization table, each node is visited once and constant time is spent at each. There are $O(n)$ nodes, so assuming every needed subproblem is still cached it takes $O(n)$ steps to complete this portion of the algorithm.

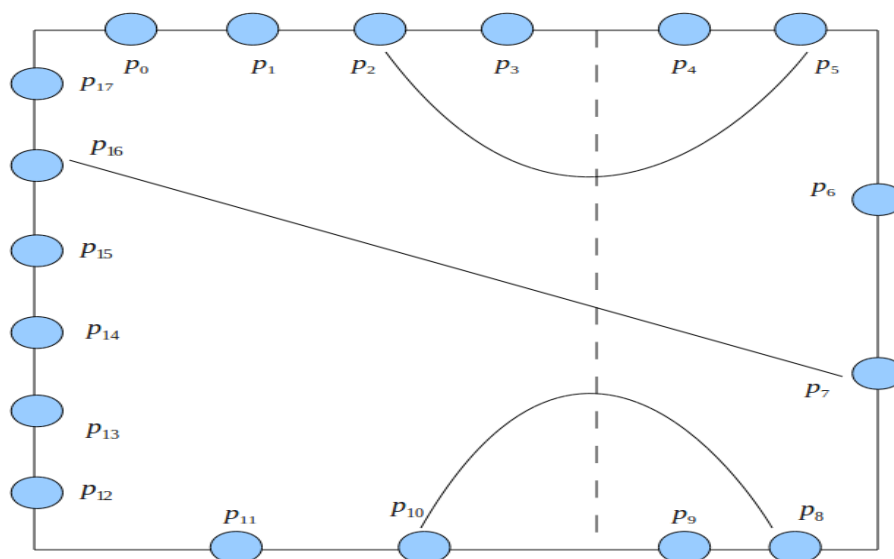


Figure 5.9: Anomaly scenario - 3 pairings, 3 sibling points needed

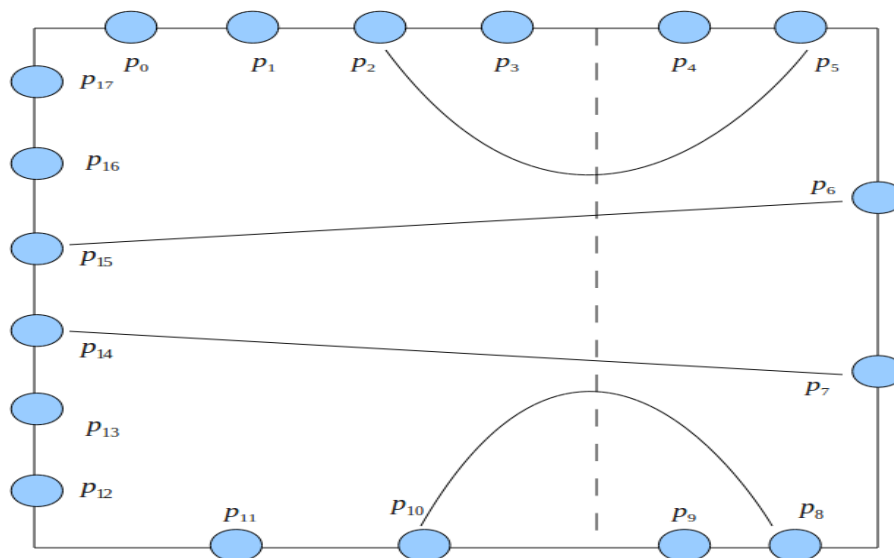


Figure 5.10: Anomaly scenario - 4 pairings, 4 sibling points needed

5.5.1 Base case

In algorithm 27 once recursion reaches a leaf partition, i.e. a partition having no children partitions, the action to take is straightforward. The input parameter R specifies the portal pairings that for the given partition P were found to yield the best combination of connections leading to a salesman path having the shortest length able to be found using the given constraints. Therefore, at the leaf level the only thing to be done is return the line segments that are indicated by R and the data point contained in partition P . These individual line segments, once the entire traceback procedure is finished, can then be assembled into a salesman path.

Algorithm 27 Traceback procedure to obtain the shortest salesman tour

Input: P is the current partition, $R = (r_0..r_k)$ are the current portal pairings

Output: $T = (P[t_1], P[t_2], \dots, P[t_n])$, a tour P having minimal length

```

1: procedure GET – BEST – TOUR( $\hat{P}$ )
2:   if  $P$  is a leaf node then
3:     add relevant line segments to  $T$ 
4:   else
5:     for each valid parent-child mapping  $m_i \in \Psi$  do
6:       for each valid combination of sibling portals  $c_j$  do
7:          $(R_{left}, R_{right}) \leftarrow$  GET – CHILD – INDICES( $R, m_i, c_j$ )
8:          $cost_{left} \leftarrow$  GET – BEST( $P_{left}, R_{left}$ )
9:          $cost_{right} \leftarrow$  GET – BEST( $P_{right}, R_{right}$ )
10:         $cost_{i,j} \leftarrow cost_{left} + cost_{right}$ 
11:        if  $cost_{i,j}$  is the lowest seen so far then
12:           $\hat{R}_{left} \leftarrow R_{left}$ 
13:           $\hat{R}_{right} \leftarrow R_{right}$ 
14:        GET – BEST – TOUR( $P_{left}, \hat{R}_{left}$ )
15:        GET – BEST – TOUR( $P_{right}, \hat{R}_{right}$ )
16:   return  $T$ 

```

connections that do not allow the tour to intersect itself. It turns out that the number of such combinations given n portal points is given by C_n , which is the n^{th} Catalan number. The Catalan numbers form a sequence of natural numbers that occur in various counting problems, especially those that are recursively defined. This number sequence can be described by the expression:

$$C_n = \frac{1}{n+1} \binom{2n}{n} \quad (5.17)$$

Asymptotically, C_n grows as:

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}} \quad (5.18)$$

and this has implications for how many border crossings to allow per side. If we limit the number of crossings to one per partition side, then the maximum number of portal points involved in a pairing would be four. Four portal points means two pairings, or two separate tour segments. If we allow for up to three segments, this means a maximum of six total portal points along the four borders of a partition. Allowing up to four segments means a maximum of eight total portal points, and so on. Given that we wish to restrict the portal pairing to only allow tour paths which do not intersect, this is akin to enumerating the expressions containing n pairs of parentheses which are correctly matched. We illustrate this by listing the first several canonical schemata as the number of tour segments increases. Figure 5.12 illustrates the topology of the first several canonical schemata. This figure accounts for allowing a maximum of two crossings per border, or up to four tour segments per partition. As the number of allowable tour segments per partition grows, the total number of canonical schemata that must be accounted for in a schema table mapping grows dramatically, as table 5.1 shows.





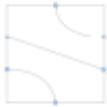


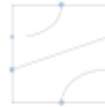





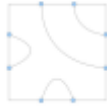
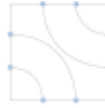






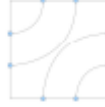
 Schema 1 (0, 1)	 Schema 2 (0, 1, 2, 3)	 Schema 3 (0, 3, 1, 2)	 Schema 4 (0, 1, 2, 3, 4, 5)
 Schema 5 (0, 1, 2, 5, 3, 4)	 Schema 6 (0, 3, 1, 2, 4, 5)	 Schema 7 (0, 5, 1, 2, 3, 4)	 Schema 8 (0, 5, 1, 4, 2, 3)
 Schema 9 (0, 1, 2, 3, 4, 5, 6, 7)	 Schema 10 (0, 1, 2, 3, 4, 7, 5, 6)	 Schema 11 (0, 1, 2, 5, 3, 4, 6, 7)	 Schema 12 (0, 1, 2, 7, 3, 4, 5, 6)
 Schema 13 (0, 1, 2, 7, 3, 6, 4, 5)	 Schema 14 (0, 3, 1, 2, 4, 5, 6, 7)	 Schema 15 (0, 3, 1, 2, 4, 7, 5, 6)	 Schema 16 (0, 5, 1, 2, 3, 4, 6, 7)
 Schema 17 (0, 5, 1, 4, 2, 3, 6, 7)	 Schema 18 (0, 7, 1, 2, 3, 4, 5, 6)	 Schema 19 (0, 7, 1, 2, 3, 6, 4, 5)	 Schema 20 (0, 7, 1, 4, 2, 3, 5, 6)
	 Schema 21 (0, 7, 1, 6, 2, 3, 4, 5)	 Schema 22 (0, 7, 1, 6, 2, 5, 3, 4)	

Figure 5.12: Canonical schemata for one, two, three, and four four segments.

Table 5.1: Growth of canonical schemata.

Max # of segments, n	C_n	Total schemata
1	1	1
2	2	3
3	5	8
4	14	22
5	42	64
6	132	196
7	429	625
8	1430	2055

5.7 Automatic Table Generation

Given that we cut partitions into two children and that the cut direction depends on which dimension is longer for a given partition, a full canonical schema table must account for both portrait and landscape orientations. In source code we implemented our canonical schema table as a struct containing a four dimensional array of structures, as shown in the listing below. The canonical schemata themselves are hardcoded one time and need not be repeated; what the schema table maps is canonical schema index numbers. These can be stored in one byte, because our schema tables only account for up to six tour segments per partition, which table 5.1 tells us only needs to be able to index 196 canonical schemata.

Since these schema table mappings are considerably large and manually coding them would risk errors, we automate their creation by using a separate program to automatically generate the source code which initializes the schema tables. Algorithm 28 shows how we generate the source for these schema tables. The concept behind this algorithm is to generate every combination of left and right child schema, see if they match, and only keep the valid mappings. There may be faster methods of generating a schema table, but this approach gave us confidence that the resulting table would be accurate and complete. Extending this procedure to be able to generate larger tables would require adding the canonical schemata corresponding to the number of pairings that one wishes

to support.

```

1 struct child_schema {
2   int8_t left;
3   int8_t right;
4   int8_t num_siblings;
5 };
6 struct child_schema_table {
7   // parent schema, orientation, num left points, ith entry
8   struct child_schema cs[22][8][8][5];
9   // number of schemata, orientation, num left points
10  int8_t ns[22][8][8];
11 };

```

Listing 5.1: Schema table structure declarations

Algorithm 28 Schema table source code generator

Input: M is the maximum allowable number of tour segments per partition

Output: None.

Side effect: automatically generates source code wherein parent canonical schemata are matched to corresponding left and right child schemata

```

1: procedure GENERATE – SCHEMA – TABLES( $M$ )
2:    $A \leftarrow$  all canonical schemata having 1 to  $M$  pairings.
3:   for each canonical schema  $S_{left} \in A$  do
4:     for each canonical schema  $S_{right} \in A$  do
5:       if  $S_{left}$  matches  $S_{rightCatalan}$  then
6:          $S_{parent} \leftarrow$  GET – PARENT – SCHEMA( $S_{left}, S_{right}$ )
7:         generate source code encoding mapping  $S_{parent} \rightarrow (S_{left}, S_{right})$ 

```

5.8 Further Implementation Considerations

In this section we examine some implementation decisions that we confronted, as well as some variations on the basic algorithm itself that gave us insight into where potential improvements could be made.

Identifying Subproblems for Memoization

A common issue with caching memoized subproblems is about how to identify them in the cache. In many industry settings this issue is resolved by using built-in programming language constructs that generate a hashcode for the objects being cached, and this is often computed based on the object's location in memory. Such an approach is not robust when trying to distribute the computation across devices, and a preferable approach is to generate cache keys using the semantic content of the item being cached (when possible).

Our caching module makes no attempts to abstract over these issues, as both keys and values are given as arrays of unsigned bytes. This leaves it to the module using the cache to prepare its keys and values by manually loading those arrays in a way that's meaningful to the application in question. In the case of Arora's algorithm, as we've discussed in the sections above, what identifies a subproblem are two things: the partition P and the portal pairings R . These two pieces of information constitute the key being cached and the cost associated with P and R is the value stored for that key. When we limit the schema table to allow a maximum of four segments per partition, the number of portal points we store is no longer than eight values. Therefore, we need to store the partition index and the portal indices in order to retrieve the cost for the given subproblem. For reasonably small instances, eg. $n \leq 200$ data points, we are able to identify the partition using an 8-bit integer and each local portal point index can also fit into 8-bits, which means 9-byte keys are sufficient. When we solve larger instances, the number of partitions grows, as well as the number of portal points that a given partition can contain, which increasing the number of bits per partition index and the number of bits per portal index. For medium sized instances we use 18 bits to identify subproblems and 27 bits to identify subproblems when solving large instances.

Caching distances

The base case for both the recursive cost function and the traceback procedure checks the distances between data points and portal points. We noticed that caching the distances between data points and portal points yielded a margin improvement in the run time performance of our implementation.

Parallelization Strategies

Our implementation at present does not support computing subproblems simultaneously on multiple threads. However, given that siblings in the partition tree are non-overlapping, the modifications needed to parallelize our implementation might not be extensive. One approach chooses the number of execution threads as a power of two, 2^k , such that subproblems on partitions at depth $k - 1$ become delegated to separate threads on sibling partitions at depth k . A more ambitious idea takes into account that an unfortunate partitioning of the data points could result in an imbalanced partition tree, rendering the strategy of multi-threading the execution of siblings at a given depth a bad idea. The number of different binary trees on n nodes is C_n , the n^{th} Catalan number. Therefore, if multiple partition trees were created in the attempt of landing on a balanced partition tree, the partitioning procedure could be slowed down by a factor of C_n .

5.9 Performance without limiting cache space

In this section we present empirical results from solving TSP instances using **a100**. The aim here is to surmise the capabilities of our solver alone, i.e. without attempting to limit memory usage through a cache replacement policy. In Chapter 4, we examine the effect of solving instances with limited memory and employing cache replacement policies. In the results reported in this section a total of 180 different instances were tested, ranging in size from 10 to 100,000 data points. Most of these are published data sets having prior

Table 5.2: **a100** configuration scenarios tested.

	portals/border	sibling points used
2 portals/all subproblems	2	all possible
1 portal/all subproblems	1	all possible
limited subproblems	2	one set per schema mapping

empirical results for comparison. We also generated many random instances. In the tables below we follow the naming convention for TSP instances consisting of including the number of data points in the instance name. Problem sizes larger than that were killed after 30 minutes of run time.

5.9.1 Configuration Scenarios Tested

Arora’s algorithm involves choosing several parameters, such as the maximum number of crossings per border, the number of portal points used, the shift (and possible rotation) of the data points before partitioning, and the number of subproblems to explore at every recursive step. This allows for a large number of possible configuration scenarios, of which we chose three for the results reported below. Table 5.2 summarizes the differences between the three types of tests we ran with a100. The first two scenarios operate exactly as described in the algorithms above. In the scenario labeled “limited subproblems” we only choose one set of sibling portal points for every valid schema mapping. In algorithms 26 and 27 this effectively means that the inner for loop in each iterates only once. The significance of this scenario is that it corresponds to having made one single “guess”³ at the best sibling portal to use at each level of recursion. As the results show this approach sometimes results in a fortunate outcome in terms of tour length, and the run times (and thus scalability) of this approach allow for quite large instances to be approximated.

³In Arora’s original description of the algorithm, he refers to the portal selection as a *guess*. In the configuration scenario in question, we are essentially making only one guess.

Table 5.3: Average approximation factor achieved by **a100**.

	Average α
2 portals/all subproblems	1.037
1 portal/all subproblems	1.079
limited subproblems	1.324

5.9.2 Solution Quality

We measure solution quality as the approximation factor that was described in Chapter 2, which is the ratio of the length of the tour found by **a100** to the length of the optimal tour, when the latter is known. In a few cases the optimal tour length wasn't available, although best known tours were reported to be within less than 0.01% of the bound provided by linear programming methods. In those cases the approximation factor reported uses the best known tour length. Table 5.3 shows the average degree of approximation achieved for each configuration scenario explored. Tables 5.4, 5.5, 5.6, and 5.7 list the raw data used to obtain those averages.

For two portals per border, the average approximation factor achieved was 1.037 for instances having as many as 136 data points. For one portal per border, the average approximation factor achieved was 1.079 for instances having as many as 442 data points. When we limited the number of subproblems to using only one choice of sibling points per schema mapping, the average approximation factor achieved was 1.324 for instances having as many as 100,000 data points. In none of the three scenarios does it appear that α increases as a function of problem size. Certain small instances appeared to be “hard”, i.e. yielded low solution quality, whereas in certain other cases the chosen shift/perturbation of points and resulting partitioning yielded good performance, even in the limited subproblems scenario. In this latter case **a100** was able to achieve solutions within 10% of optimality for several instances, and even solved the *mona-lisa100K* instance to within 15% of optimality in less than 30 minutes.

Table 5.4: Approximation factor (α), 2 portals/border, all subproblems.

instance	tour length	α	instance	tour length	α
<i>burma14</i>	31	1.000	<i>rat99</i>	1,254	1.028
<i>ulysses16</i>	74	1.000	<i>kroA100</i>	23,002	1.081
<i>ulysses22</i>	75	1.000	<i>kroB100</i>	22,781	1.029
<i>wi29</i>	27,601	1.000	<i>kroC100</i>	21,752	1.048
<i>dj38</i>	6,816	1.024	<i>kroD100</i>	22,574	1.060
<i>att48</i>	35,079	1.046	<i>kroE100</i>	22,963	1.041
<i>eil51</i>	440	1.022	<i>rd100</i>	8,317	1.051
<i>berlin52</i>	7,804	1.034	<i>lin105</i>	15,392	1.070
<i>st70</i>	693	1.021	<i>pr107</i>	45,492	1.027
<i>eil76</i>	571	1.047	<i>pr124</i>	63,239	1.071
<i>pr76</i>	111,757	1.033	<i>pr136</i>	99,109	1.024

Table 5.5: Approximation factor (α), 1 portal/border, all subproblems.

instance	tour length	α	instance	tour length	α
<i>wi29</i>	27,749	1.005	<i>pr136</i>	101,738	1.051
<i>dj38</i>	7,568	1.137	<i>pr144</i>	61,008	1.042
<i>att48</i>	36,591	1.091	<i>ch150</i>	7,268	1.113
<i>eil51</i>	445	1.035	<i>kroA150</i>	28,395	1.071
<i>berlin52</i>	8,191	1.086	<i>kroB150</i>	27,685	1.059
<i>st70</i>	699	1.030	<i>pr152</i>	82,844	1.124
<i>eil76</i>	580	1.063	<i>u159</i>	45,257	1.076
<i>pr76</i>	113,183	1.046	<i>qa194</i>	10,776	1.152
<i>gr96</i>	572	1.117	<i>rat195</i>	2,409	1.037
<i>rat99</i>	1,269	1.040	<i>d198</i>	17,851	1.131
<i>kroA100</i>	23,217	1.091	<i>kroA200</i>	32,061	1.092
<i>kroB100</i>	23,935	1.081	<i>kroB200</i>	31,779	1.080
<i>kroC100</i>	21,988	1.060	<i>tsp225</i>	4,023	1.042
<i>kroD100</i>	22,946	1.078	<i>gil262</i>	2,595	1.091
<i>kroE100</i>	23,039	1.044	<i>pr264</i>	54,414	1.107
<i>rd100</i>	8,428	1.065	<i>a280</i>	2,775	1.073
<i>eil101</i>	689	1.073	<i>pr299</i>	51,637	1.072
<i>lin105</i>	15,627	1.087	<i>lin318</i>	47,296	1.125
<i>pr107</i>	47,835	1.080	<i>linhp318</i>	47,296	1.125
<i>pr124</i>	69,666	1.180	<i>pcb442</i>	54,963	1.082
<i>ch130</i>	6,640	1.087			

Table 5.6: Approximation factor (α), 2 portals/border, limited subproblems.

instance	tour length	α	instance	tour length	α
<i>burma14</i>	36	1.156	<i>rat783</i>	11,376	1.292
<i>ulysses16</i>	80	1.082	<i>zi929</i>	132,459	1.389
<i>ulysses22</i>	83	1.094	<i>lu980</i>	14,903	1.314
<i>wi29</i>	29,941	1.085	<i>dsj1000</i>	26,466,570	1.418
<i>dj38</i>	8,407	1.263	<i>pr1002</i>	363,744	1.404
<i>att48</i>	39,534	1.179	<i>u1060</i>	305,237	1.362
<i>eil51</i>	471	1.096	<i>vm1084</i>	342,601	1.432
<i>berlin52</i>	10,103	1.339	<i>pcb1173</i>	78,158	1.374
<i>st70</i>	840	1.245	<i>d1291</i>	75,086	1.478
<i>eil76</i>	676	1.239	<i>rl1304</i>	388,301	1.535
<i>pr76</i>	128,270	1.186	<i>rl1323</i>	413,261	1.529
<i>gr96</i>	639	1.248	<i>nrv1379</i>	74,004	1.307
<i>rat99</i>	1,529	1.254	<i>fl1400</i>	26,990	1.341
<i>kroA100</i>	29,380	1.380	<i>u1432</i>	183,631	1.200
<i>kroB100</i>	27,008	1.220	<i>fl1577</i>	35,762	1.607
<i>kroC100</i>	30,989	1.493	<i>rw1621</i>	36,347	1.395
<i>kroD100</i>	26,673	1.253	<i>d1655</i>	85,397	1.375
<i>kroE100</i>	25,926	1.175	<i>vm1748</i>	464,676	1.381
<i>rd100</i>	9,975	1.261	<i>u1817</i>	79,202	1.385
<i>eil101</i>	763	1.188	<i>rl1889</i>	473,853	1.497
<i>lin105</i>	18,658	1.297	<i>mu1979</i>	129,801	1.494
<i>pr107</i>	49,176	1.110	<i>d2103</i>	114,664	1.425
<i>pr124</i>	75,088	1.272	<i>u2152</i>	88,657	1.380
<i>bier127</i>	148,043	1.252	<i>u2319</i>	256,376	1.095
<i>ch130</i>	7,747	1.268	<i>pr2392</i>	504,666	1.335
<i>pr136</i>	111,080	1.148	<i>pcb3038</i>	181,296	1.317
<i>pr144</i>	74,976	1.281	<i>nu3496</i>	132,887	1.382
<i>ch150</i>	8,828	1.352	<i>fl3795</i>	46,053	1.601
<i>kroA150</i>	34,586	1.304	<i>fnl4461</i>	237,535	1.301
<i>kroB150</i>	33,673	1.289	<i>ca4663</i>	1,824,086	1.414

Table 5.7: Approximation factor (α), 2 portals/border, limited sub-problems (cont.).

instance	tour length	α	instance	tour length	α
<i>pr152</i>	100,728	1.367	<i>rl5915</i>	861,342	1.523
<i>u159</i>	54,773	1.302	<i>rl5934</i>	854,444	1.537
<i>qa194</i>	12,070	1.291	<i>tz6117</i>	557,509	1.412
<i>rat195</i>	2,871	1.236	<i>eg7146</i>	246,701	1.431
<i>d198</i>	18,881	1.196	<i>pla7397</i>	32,296,611	1.388
<i>kroA200</i>	37,880	1.290	<i>ym7663</i>	339,307	1.424
<i>kroB200</i>	36,542	1.241	<i>pm8079</i>	155,898	1.357
<i>gr202</i>	572	1.040	<i>ei8246</i>	284,368	1.379
<i>ts225</i>	165,342	1.306	<i>ar9152</i>	1,182,504	1.412
<i>tsp225</i>	5,112	1.325	<i>ja9847</i>	713,623	1.451
<i>pr226</i>	90,072	1.121	<i>gr9882</i>	418,356	1.390
<i>gil262</i>	3,168	1.332	<i>kz9976</i>	1,505,639	1.418
<i>pr264</i>	62,042	1.263	<i>fl10639</i>	712,802	1.369
<i>a280</i>	3,409	1.322	<i>rl11849</i>	1,323,654	1.434
<i>pr299</i>	63,747	1.323	<i>usa13509</i>	27,523,901	1.377
<i>lin318</i>	57,680	1.372	<i>brd14051</i>	618,037	1.317
<i>linhp318</i>	57,680	1.372	<i>mo14185</i>	584,344	1.367
<i>rd400</i>	20,026	1.311	<i>ho14473</i>	233,142	1.317
<i>fl417</i>	18,127	1.528	<i>d15112</i>	2,083,460	1.324
<i>pr439</i>	147,287	1.374	<i>it16862</i>	763,813	1.371
<i>pcb442</i>	70,187	1.382	<i>d18512</i>	842,179	1.305
<i>d493</i>	44,026	1.258	<i>vm22775</i>	755,927	1.328
<i>u574</i>	47,934	1.299	<i>sw24978</i>	1,179,515	1.379
<i>rat575</i>	8,673	1.281	<i>bm33708</i>	1,299,898	1.355
<i>p654</i>	46,080	1.330	<i>pla33810</i>	92,526,026	1.399
<i>d657</i>	64,134	1.311	<i>ch71009</i>	6,184,003	1.354
<i>gr666</i>	4,142	1.048	<i>pla85900</i>	189,348,165	1.330
<i>u724</i>	54,381	1.298	<i>mona-lisa100K</i>	6,628,388	1.151
<i>uy734</i>	105,371	1.332			

5.10 Run Time and Cache Miss Performance

Tables 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, and 5.15 report the raw data listing run times and cache miss totals for the instances tested using the three configuration scenarios of **a100** that we explored. When the parameters of our implementation are adjusted for obtaining good quality solutions, eg. within 5% of optimality, the run times become prohibitively high for instances above 200 data points. When approximations within 10% of optimality are acceptable, our implementation can obtain solutions rather quickly and consistently for data sets of up to around 450 points. When the parameters of our implementation are adjusted for reducing the number of subproblems visited, that's when our implementation becomes truly scalable in run time performance, albeit with significant sacrifices in solution quality. The initial choice of orientation and shift of the data points before partitioning is likely to be a crucial factor in allowing our solver to obtain better solutions. Therefore, an area to be explored is to use our parameter settings adjusted for high speed combined with several shifts and rotations of the data points. That could result in greatly improved solution quality with only a constant additive in run time. When we limit the subproblems visited, the bulk of the time is spent in setting portal points and in partitioning the instance. In the other two scenarios tested the bulk of the time is spent obtaining the cost of the shortest tour.

In terms of the cache misses that occur, figures 5.10, 5.10, and 5.10 illustrate, for the three configuration scenarios tested, the rate of increase that happens as problem sizes increase. In the first two cases even though problems vary in difficulty there is a clear explosion in cache misses as problem sizes increase. In the case of limited subproblems we see a more gradual increase, suggesting the scalability of this option.

Table 5.8: Run time and cache misses, 2 portals/border, all subproblems.

instance	total time (sec)	cache misses
<i>random10</i>	0.444275	1264
<i>burma14</i>	0.44369	4287
<i>ulysses16</i>	0.977672	199762
<i>random20</i>	0.710591	146695
<i>ulysses22</i>	2.164056	492637
<i>wi29</i>	0.594852	92347
<i>random30</i>	0.63352	162375
<i>dj38</i>	34.288559	5798212
<i>random40</i>	4.310083	1378061
<i>att48</i>	4.986496	1417845
<i>random50</i>	4.631562	1538483
<i>eil51</i>	4.768721	1621361
<i>berlin52</i>	293.536629	31541755
<i>random60</i>	16.363247	5857362
<i>random70</i>	157.089442	19707228
<i>st70</i>	12.201555	3586481
<i>eil76</i>	95.150107	13899716
<i>pr76</i>	3.620041	1280241
<i>random80</i>	111.446084	16733703
<i>random90</i>	138.851729	27420242
<i>rat99</i>	54.76024	10789899
<i>random100</i>	131.44615	22113673
<i>kroA100</i>	43.331197	14527147
<i>kroB100</i>	316.931973	15791793
<i>kroC100</i>	16.572458	4652159
<i>kroD100</i>	71.193981	10487992
<i>kroE100</i>	42.080023	10816897
<i>rd100</i>	81.963075	13273783
<i>lin105</i>	42.035804	7218938
<i>pr107</i>	1.913265	558233
<i>random110</i>	128.521855	19125982
<i>random120</i>	111.111805	18837249
<i>pr124</i>	70.521321	22080578
<i>pr136</i>	17.375342	5072446

Table 5.9: Run time and cache misses, 1 portal/border, all subproblems.

instance	total time (sec)	cache misses
<i>random10</i>	0.442839	94
<i>random20</i>	0.446769	3230
<i>wi29</i>	0.501845	2359
<i>random30</i>	0.446759	3744
<i>dj38</i>	0.571435	52740
<i>random40</i>	0.481214	22473
<i>att48</i>	0.445306	21295
<i>random50</i>	0.471856	21699
<i>eil51</i>	0.452218	26621
<i>berlin52</i>	0.920635	234012
<i>random60</i>	0.516012	73637
<i>random70</i>	0.890702	192803
<i>st70</i>	0.492169	55229
<i>eil76</i>	0.680832	149769
<i>pr76</i>	0.450978	20781
<i>random80</i>	0.683069	155243
<i>random90</i>	0.765612	221515
<i>gr96</i>	1.929057	601900
<i>rat99</i>	0.641254	137799
<i>random100</i>	0.786839	214967
<i>kroA100</i>	0.614557	128513
<i>kroB100</i>	1.404107	245526
<i>kroC100</i>	0.517718	75330
<i>kroD100</i>	0.723081	152251
<i>kroE100</i>	0.599902	124382
<i>rd100</i>	0.703176	153920
<i>eil101</i>	1.430207	335263
<i>lin105</i>	0.586476	104412
<i>pr107</i>	0.442432	13974
<i>random110</i>	0.820011	221179
<i>random120</i>	0.761679	201203
<i>pr124</i>	0.716201	230322
<i>bier127</i>	140.01324	10902434
<i>random130</i>	2.763805	835729
<i>ch130</i>	2.118787	811255
<i>pr136</i>	0.508422	64298

Table 5.10: Run time and cache misses, 1 portal/border, all subproblems (cont.).

instance	total time (sec)	cache misses
<i>gr137</i>	10.072184	1968275
<i>random140</i>	1.334788	440236
<i>pr144</i>	4.166754	2954606
<i>random150</i>	11.625614	2872149
<i>ch150</i>	3.212551	939915
<i>kroA150</i>	1.120474	329788
<i>kroB150</i>	1.349027	432484
<i>pr152</i>	2.724821	1121814
<i>u159</i>	0.893602	305412
<i>random160</i>	4.847981	1139656
<i>random170</i>	8.555562	2294416
<i>random180</i>	10.190972	2662410
<i>random190</i>	28.0782	5188566
<i>qa194</i>	18.1527	3786904
<i>rat195</i>	3.503327	813341
<i>d198</i>	11.302157	2095234
<i>random200</i>	8.302201	2509494
<i>kroA200</i>	2.090698	772851
<i>kroB200</i>	5.802944	1418063
<i>random220</i>	23.217693	4233861
<i>tsp225</i>	5.877902	1557686
<i>gr229</i>	62.480743	8336406
<i>random240</i>	21.443264	5228811
<i>random260</i>	47.462579	6796331
<i>gil262</i>	39.236887	8497010
<i>pr264</i>	1.340845	473533
<i>random280</i>	49.805961	9307800
<i>a280</i>	6.30503	1964324
<i>pr299</i>	6.989829	2116047
<i>random300</i>	44.081897	7454967
<i>lin318</i>	30.802472	6560429
<i>linhp318</i>	30.898	6560429
<i>random350</i>	219.255829	22412749
<i>random400</i>	208.189237	25046990
<i>pcb442</i>	119.880631	17056428

Table 5.11: Run time and cache misses, 2 portals/border, limited sub-problems (cont.).

instance	total time (sec)	cache misses
<i>random10</i>	0.424162	18
<i>burma14</i>	0.46377	30
<i>ulysses16</i>	0.47687	39
<i>random20</i>	0.422682	42
<i>ulysses22</i>	0.465886	51
<i>wi29</i>	0.465975	56
<i>random30</i>	0.426436	81
<i>dj38</i>	0.459277	105
<i>random40</i>	0.428999	87
<i>att48</i>	0.43936	106
<i>random50</i>	0.431649	154
<i>eil51</i>	0.472064	116
<i>berlin52</i>	0.474722	163
<i>random60</i>	0.427536	193
<i>st70</i>	0.469389	174
<i>random70</i>	0.436047	206
<i>eil76</i>	0.474014	226
<i>pr76</i>	0.475181	221
<i>random80</i>	0.430619	204
<i>random90</i>	0.431317	269
<i>gr96</i>	0.426358	308
<i>rat99</i>	0.478055	419
<i>kroA100</i>	0.470552	280
<i>kroB100</i>	0.479733	226
<i>kroC100</i>	0.483919	296
<i>kroD100</i>	0.469753	244
<i>kroE100</i>	0.478394	253
<i>rd100</i>	0.420771	290
<i>random100</i>	0.430593	231
<i>eil101</i>	0.468714	264
<i>lin105</i>	0.480219	341
<i>pr107</i>	0.475588	413
<i>random110</i>	0.432506	399
<i>random120</i>	0.432652	375
<i>pr124</i>	0.478003	373
<i>bier127</i>	0.483441	409

Table 5.12: Run time and cache misses, 2 portals/border, limited sub-problems (cont.).

instance	total time (sec)	cache misses
<i>ch130</i>	0.48026	337
<i>random130</i>	0.430344	354
<i>pr136</i>	0.482479	448
<i>gr137</i>	0.472887	471
<i>random140</i>	0.43147	397
<i>pr144</i>	0.464467	401
<i>ch150</i>	0.477999	502
<i>kroA150</i>	0.471261	404
<i>kroB150</i>	0.474631	339
<i>random150</i>	0.430655	379
<i>pr152</i>	0.48527	419
<i>u159</i>	0.477554	432
<i>random160</i>	0.429548	609
<i>random170</i>	0.430074	545
<i>random180</i>	0.433193	478
<i>random190</i>	0.435718	552
<i>qa194</i>	0.486139	697
<i>rat195</i>	0.479925	790
<i>d198</i>	0.477352	659
<i>kroA200</i>	0.470095	596
<i>kroB200</i>	0.476092	683
<i>random200</i>	0.436571	668
<i>gr202</i>	0.485946	1257
<i>random220</i>	0.438859	817
<i>ts225</i>	0.469027	556
<i>tsp225</i>	0.480252	890
<i>pr226</i>	0.471906	638
<i>gr229</i>	0.472933	1185
<i>random240</i>	0.432237	754
<i>random260</i>	0.433423	810
<i>gil262</i>	0.486148	728
<i>pr264</i>	0.467543	919
<i>a280</i>	0.483722	916
<i>random280</i>	0.433945	914
<i>pr299</i>	0.489953	1245
<i>random300</i>	0.440295	910

Table 5.13: Run time and cache misses, 2 portals/border, limited sub-problems (cont.).

instance	total time (sec)	cache misses
<i>lin318</i>	0.490147	1067
<i>linhp318</i>	0.476587	1067
<i>random350</i>	0.443977	1055
<i>rd400</i>	0.515255	1277
<i>random400</i>	0.446233	1181
<i>fl417</i>	0.489471	1547
<i>gr431</i>	0.499756	1618
<i>pr439</i>	0.488926	1414
<i>pcb442</i>	0.488019	1356
<i>random450</i>	0.444812	1590
<i>d493</i>	0.492087	1756
<i>random500</i>	0.45428	1809
<i>att532</i>	0.50183	2019
<i>ali535</i>	0.49153	2130
<i>u574</i>	0.496873	1972
<i>rat575</i>	0.465661	2753
<i>random600</i>	0.462823	2213
<i>p654</i>	0.512764	2523
<i>d657</i>	0.536183	2928
<i>gr666</i>	0.507453	2873
<i>random700</i>	0.478452	2169
<i>u724</i>	0.517875	2388
<i>u734</i>	0.537618	3114
<i>rat783</i>	0.528535	3961
<i>random800</i>	0.495146	2762
<i>random900</i>	0.514872	3601
<i>zi929</i>	0.548544	5627
<i>lu980</i>	0.499383	2812
<i>dsj1000</i>	0.576934	4740
<i>random1000</i>	0.528127	3114
<i>pr1002</i>	0.566561	3885
<i>u1060</i>	0.591727	5955
<i>vm1084</i>	0.555389	4099
<i>pcb1173</i>	0.616408	5092
<i>random1200</i>	0.574463	4325
<i>d1291</i>	0.605569	4852

Table 5.14: Run time and cache misses, 2 portals/border, limited sub-problems (cont.).

instance	total time (sec)	cache misses
<i>rl1304</i>	0.628199	5297
<i>rl1323</i>	0.613523	4773
<i>nrw1379</i>	0.660213	6143
<i>fl1400</i>	0.638544	6282
<i>random1400</i>	0.615777	4824
<i>u1432</i>	0.628393	5775
<i>fl1577</i>	0.687664	6802
<i>random1600</i>	0.692061	6028
<i>rw1621</i>	0.536907	2874
<i>d1655</i>	0.708673	7201
<i>vm1748</i>	0.726826	6388
<i>random1800</i>	0.743825	5989
<i>u1817</i>	0.70354	7805
<i>rl1889</i>	0.797197	6640
<i>mu1979</i>	0.860584	8073
<i>random2000</i>	0.823442	7600
<i>d2103</i>	0.762181	8810
<i>u2152</i>	0.782816	9154
<i>u2319</i>	0.79948	10731
<i>pr2392</i>	1.038637	9570
<i>random2500</i>	1.075242	10793
<i>random3000</i>	1.334445	11605
<i>pcb3038</i>	1.372929	11397
<i>nu3496</i>	0.998721	12770
<i>random3500</i>	1.667924	12942
<i>fl3795</i>	1.547532	15496
<i>random4000</i>	2.015858	17109
<i>fnl4461</i>	2.398364	19445
<i>ca4663</i>	2.530927	28072
<i>random5000</i>	2.862496	21818
<i>rl5915</i>	3.444655	23091
<i>rl5934</i>	3.522624	26744
<i>random6000</i>	4.04534	24522
<i>tz6117</i>	3.885315	25258
<i>random7000</i>	5.244103	29074
<i>eg7146</i>	5.438541	44874

Table 5.15: Run time and cache misses, 2 portals/border, limited sub-problems (cont.).

instance	total time (sec)	cache misses
<i>pla7397</i>	4.437727	33694
<i>ym7663</i>	6.230499	38645
<i>random8000</i>	6.792605	33123
<i>pm8079</i>	2.734395	25366
<i>ei8246</i>	7.236932	41070
<i>random9000</i>	8.076343	39817
<i>ar9152</i>	4.893452	37499
<i>ja9847</i>	9.604251	48136
<i>gr9882</i>	8.701355	48945
<i>kz9976</i>	10.287256	54282
<i>random10000</i>	10.123077	43919
<i>fl10639</i>	11.017062	57692
<i>rl11849</i>	12.175853	54769
<i>usa13509</i>	18.041182	87623
<i>brd14051</i>	19.247134	75634
<i>mo14185</i>	18.776937	69815
<i>ho14473</i>	5.009122	42228
<i>d15112</i>	22.979666	85278
<i>it16862</i>	26.790909	101392
<i>d18512</i>	34.264042	95430
<i>random20000</i>	39.721757	95072
<i>vm22775</i>	45.265587	130940
<i>sw24978</i>	59.794845	132566
<i>random30000</i>	105.347772	132960
<i>bm33708</i>	116.991816	228,311
<i>pla33810</i>	98.348389	180369
<i>random40000</i>	231.451925	179340
<i>random50000</i>	388.2521	233951
<i>random60000</i>	600.06444	273965
<i>random70000</i>	806.234255	327693
<i>ch71009</i>	871.174714	395804
<i>random80000</i>	1216.134349	390467
<i>pla85900</i>	917.263725	487745
<i>random90000</i>	1530.93131	418909
<i>random100000</i>	1803.230146	469515
<i>mona-lisa100K</i>	1774.370209	518159

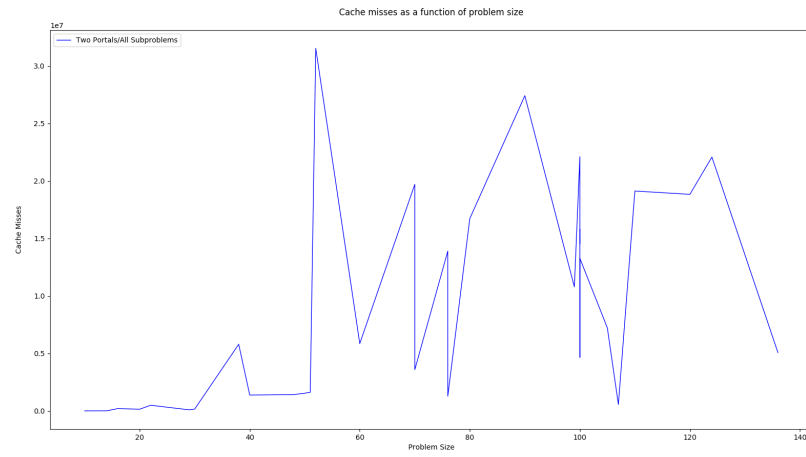


Figure 5.13: Cache misses as a function of problem size - 2 portals/border, all subproblems

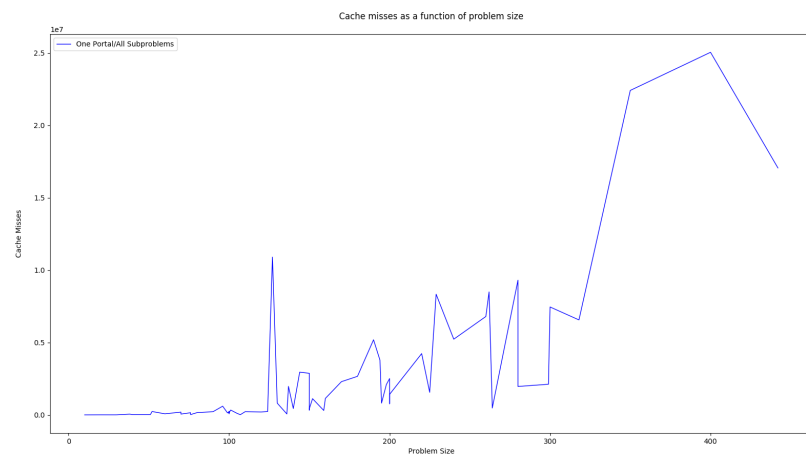


Figure 5.14: Cache misses as a function of problem size - 1 portal/border, all subproblems

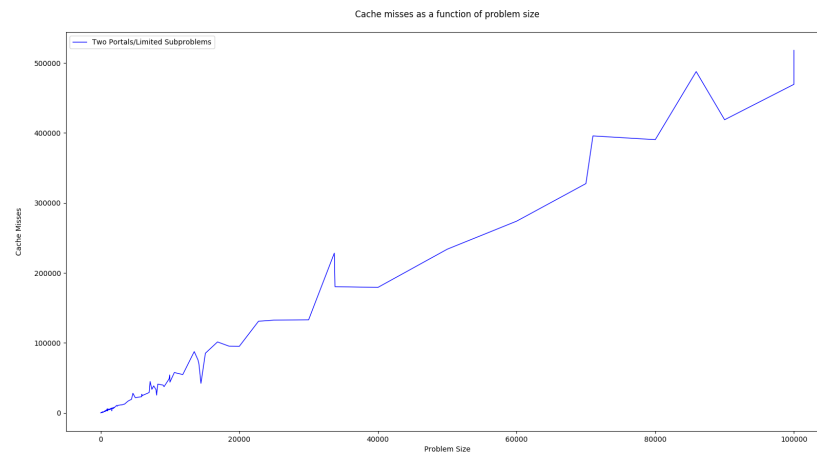


Figure 5.15: Cache misses as a function of problem size - 2 portals/border, limited subproblems

5.11 Areas for Improvement in Our Implementation

We don't not explore multiple shifts and rotations of the data points in our implementation. We merely rescale them and slightly move them until partitioning can be done such that no border coincides with data points, i.e. all data points must be properly contained within partitions. A future improvement of this implementation, one that could improve solution quality with only a constant multiple in runtime, depending on how many shifts and rotations of the data set are explored before partitioning the data.

5.12 Conclusions

In this chapter we have presented a novel implementation of Arora's well-known traveling salesman problem algorithm. We employed several effective software design and implementation strategies in order to render manageable the complexities inherent in turning this solution approach into a concrete

tool. In the empirical data section we demonstrated the capabilities of **a100** both as a tool for obtaining good quality solutions to instances of modest size, and as a tool for obtaining crude approximations to very large instances.

As opposed to the experiments in the previous chapter, all of the experiments in this chapter made use of a cache having enough memory to hold all of the subproblems encountered while solving the given instance. In other words for the experiments in this chapter, it was always the case that the cache size C was greater than the critical cache size, \hat{C} , for the given problem instance. These empirical results show that our implementation have the flexibility to be used either to emphasize solution quality or run time speed. It should be noted that when determining that a given problem instance was to be deemed “infeasible”, this decision was taken when the run time had exceeded what was considered a reasonable amount of time. That is to say that memory was not the limiting factor when determining the instance sizes able to be solved.

CHAPTER 6

CONCLUSIONS

Go on till you come to the end;
then stop.

Lewis Carroll — *Alice's Adventures
in Wonderland / Through the
Looking-Glass*

This thesis has presented several innovative strategies for rendering recursive, memoized problems more efficient both in terms of run time performance and memory requirements. We have shown that the caching algorithm proposed in this thesis generates fewer cache misses than LRU under a variety of workloads. In particular we have presented the following contributions:

- **FFRU** is a flexible, highly configurable mechanism for providing exact bounds on cache load factor, α , and recency of cached items, ϕN . The performance of our FFRU implementation confirms the expectation that for comparable eviction age lower bounds, FFRU never generates greater cache misses than LRU when solving memoized problems, because the usage ages of candidates for eviction span a greater range than those of LRU. Our implementation of FFRU caching was done using cuckoo hash tables. These provide guaranteed constant time retrieval speed and amortized constant time insertions. Since we wished to optimize memory

usage, we opted for 4 cuckoo tables because this choice leads to the ability to support high load factors, eg. in certain configurations exceeding 99%. Through empirical investigations we observed that when we limit the number of bits per cached item to one byte the maximum achievable recency factor (ϕ) as cache sizes increase converges to approximately 0.996. This can be viewed as a degree of approximation to LRU. Using this terminology, classic LRU would be the case in which $\phi = \frac{N-1}{N}$. In other words, with just one byte of administrative data per cache item, we are able to achieve very close approximation behavior to LRU. Another concern relates to the effect of the load factor, α , which increases as ϕ increases. A common complaint of cuckoo hashing (and many other hashing algorithms) is that insertion times can degenerate considerably when high load factors occur, eg. for $\alpha \geq 0.95$. Although given the high number of choices for κ and d in our algorithm, we observed empirically that as cache size increases when we limit κ to fit into one byte, it is always feasible to choose κ and d such that $\alpha \leq 0.90$, thereby preserving the expected amortized hash table operation performance.

- Several reformulations of classic optimization problems have yielded improved cache miss performance due to taking design decisions that exploited the cache replacement strategy generically characterized as evicting items that are not recently used. By leveraging several chosen factorizations of the Fibonacci recurrence, our best reformulation of Fibonacci shows cache miss growth that is logarithmic as problem sizes increase using a *constant* sized LRU/FFRU cache. Our recursive reformulation of KMP maintains the efficiency of the original version while needing only a fraction of the memory that the original iterative version uses. Our tweak to the classic LCS algorithm, i.e. *OLCS*, yields improved cache miss performance over the original using less cache memory. And in the process of investigating the LCS variants we discovered several families of particularly hard inputs from the standpoint of LRU cache

miss performance.

- The implementation of Arora’s algorithm, **a100**, presented in this thesis demonstrates run time and solution quality performance that clearly outperforms the currently known implementations of this approximation scheme. There has been suggestions (see [Applegate et al., 2006] for one such example) that the class of random Euclidean TSP instances drawn uniformly from a square bounding box constitutes in some ways a challenging set of instances. In this regard, **a100** appears to offer promising empirical results, both in terms of the growth in run time as a function of problem size and the solution quality obtained. The ability of our solver to generate a solution to the well known *MonaLisa100K* to within 15% of optimality in less than 30 minutes provides some encouragement regarding the potential of this implementation to mature into a more robust and practical tool.

These contributions provide quantitatively measurable performance benefits in certain problem domains. The methods and strategies demonstrated herein can be extended to other problem contexts.

Future Work

Techniques in this thesis open several directions for future work. Despite the saturation of research in cache replacement policies, this research topic remains ever-present both in theoretical and applied contexts.

Sparseness of dynamic programming problems: It would be interesting to further explore the performance of FFRU (and related caching algorithms) when the degree of overlap in encountered subproblems is the parameter being controlled. Such results might inform which is the preferred variant of FFRU to be deployed for a given application context. The optimization problems investigated in this study showed considerable sparseness of subproblem patterns, therefore a proper follow-up study would apply FFRU to a wider variety of workloads, including those encountered in real-world applications.

Applying FFRU to other hash table algorithms: Our implementation of FFRU runs as an extension of cuckoo hashing, which is known to require certain limits on load factor in order to avoid the risk of insertions failing due to the inability to re-build the cuckoo tables. Given the high load factors – and hence low memory overhead – inherent in FFRU’s design, it would be worth investigating the viability of implementing FFRU over top other approaches to hash table maintenance.

Hardware/OS applications of FFRU: Given the importance of caching techniques on page fault reduction in operating systems, databases, and telecommunications devices, it would be interesting to deploy a hardware version of FFRU to glean insight as to its usefulness in such settings. Likewise, there are several opportunities to test the viability of FFRU in open source Linux distributions.

Extending a100 to operate deterministically: In terms of future areas for investigation into the viability of our implementation of Arora’s algorithm, it would be interested to explore a form of the deterministic version, namely one that performs an exhaustive number of shifts in the data points prior to partitioning. This conjecture arises due to the high variability of solution quality we report, suggesting that certain cases could very well benefit from simply shifting and/or rotating the data points several times to see which one yields the shortest tour. Note that the run time of such a modification would only increase by a constant multiple of the current run times, which would depend on how many shifts and rotations were chosen.

Employing a100 for generating candidate solutions: While our implementation of Arora’s algorithm by itself might not threaten to become the new state-of-the-art in Euclidean TSP solvers, it would be worth investigating the potential of techniques like those exhibited by **a100** of generating decent candidate solutions that can be further refined, using other optimization heuristics.

REFERENCES

- [Ager et al., 2002] Ager, M. S., Danvy, O., and Rohde, H. K. (2002). On Obtaining Knuth, Morris, and Pratt’s String Matcher by Partial Evaluation. In *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ASIA-PEPM ’02, pages 32–46, New York, NY, USA. Association for Computing Machinery.
- [Al-Zoubi et al., 2004] Al-Zoubi, H., Milenkovic, A., and Milenkovic, M. (2004). Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. In *Proceedings of the 42nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, New York, NY, USA. Association for Computing Machinery.
- [Applegate et al., 2006] Applegate, D. L., Bixby, R. E., Chvatál, V., and Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.
- [Arora, 1998] Arora, S. (1998). Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems. *Journal of the ACM*, pages 2–11.
- [Baroughi and Naderi, 2020] Baroughi, A. S. and Naderi, M. (2020). High Performance Application-oriented Memory Management on Multicore Systems. In *2020 6th Iranian Conference on Signal Processing and Intelligent Systems (ICSPIS)*, pages 1–6.

- [Belady, 1966] Belady, L. A. (1966). A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101.
- [Belady et al., 1969] Belady, L. A., Nelson, R. A., and Shedler, G. S. (1969). An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine. *Communications of the ACM*, 12(6):349–353.
- [Carr and Hennessy, 1981] Carr, R. W. and Hennessy, J. L. (1981). WS-CLOCK—a Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 87–95, New York, NY, USA. Association for Computing Machinery.
- [Chavan and Sane, 2011] Chavan, H. and Sane, S. (2011). Mobile Database Cache Replacement Policies: LRU and PRRRP. In Meghanathan, N., Kaushik, B. K., and Nagamalai, D., editors, *Advances in Computer Science and Information Technology*, pages 523–531, Berlin, Heidelberg. Springer Berlin Heidelberg.
- [Chiusano and Bjarnason, 2014] Chiusano, P. and Bjarnason, R. (2014). *Functional Programming in Scala*. Manning Publications Co., USA, 1st edition.
- [Christofides, 1976] Christofides, N. (1976). Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University.
- [Cook, 2011] Cook, W. J. (2011). *In Pursuit of the Traveling Salesman*. Princeton University Press.
- [Corbato, 1969] Corbato, F. (1969). A Paging Experiment with the Multics System. In *Festschrift: In Honor of PM Morse*. MIT Press, pages 217–228.
- [Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and

- Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.
- [Deville, 1990] Deville, Y. (1990). A Low-Cost Usage-Based Replacement Algorithm for Cache Memories. *SIGARCH Comput. Archit. News*, 18(4):52–58.
- [Dorigo and Stützle, 2004] Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. MIT Press, Cambridge, MA.
- [Emerick et al., 2012] Emerick, C., Carper, B., and Grand, C. (2012). *Clojure Programming – Practical LISP for the Java World*. O’Reilly.
- [Flood, 1956] Flood, M. M. (1956). The Traveling-Salesman Problem. *Operations Research*, 4(1):61–75.
- [Floyd and Beigel, 1994] Floyd, R. W. and Beigel, R. (1994). *The Language of Machines: An Introduction to Computability and Formal Languages*. Computer Science Press, Inc., USA.
- [Forouzan, 2009] Forouzan, B. (2009). *TCP/IP Protocol Suite*. McGraw-Hill, Inc., USA, 4th edition.
- [Fotakis et al., 2003] Fotakis, D., Pagh, R., Sanders, P., and Spirakis, P. (2003). Space Efficient Hash Tables With Worst Case Constant Access Time. In *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science*, pages 271–282.
- [Garey and Johnson, 1979] Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. Mathematical Sciences Series. W. H. Freeman.
- [Ghasemzadeh et al., 2006] Ghasemzadeh, H., Mazrouee, S., and Kakooee, M. R. (2006). Modified pseudo LRU replacement algorithm. *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS’06)*, pages 368–376.

- [Greco, 2008] Greco, F., editor (2008). *Traveling Salesman Problem*. In-Teh.
- [Gries and Levin, 1980] Gries, D. and Levin, G. (1980). Computing Fibonacci Numbers (and Similarly Defined Functions) in Log Time. *Information Processing Letters*, 11(2):68–69.
- [Halloway and Bedra, 2018] Halloway, S. and Bedra, A. (2018). *Programming Clojure*. Pragmatic Bookshelf, 3rd edition.
- [He and Xiang, 2017] He, Y. and Xiang, M. (2017). An Empirical Analysis of Approximation Algorithms for the Euclidean Traveling Salesman Problem. *CoRR*, abs/1705.09058.
- [Hennessy and Patterson, 2017] Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition.
- [Hochbaum and Hochbaum, 1997] Hochbaum, D. and Hochbaum, E. (1997). *Approximation Algorithms for NP-hard Problems*. Computer science. PWS Publishing Company.
- [Hughes, 1986] Hughes, R. J. M. (1986). A Novel Representation of Lists and Its Application to the Function "Reverse". *Information Processing Letters*, 22(3):141–144.
- [Jiang et al., 2005] Jiang, S., Chen, F., and Zhang, X. (2005). CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05*, page 35, USA. USENIX Association.
- [Jiang and Zhang, 2002] Jiang, S. and Zhang, X. F. (2002). LIRS: An Efficient Low Inter-reference Recency Set Replacement to Improve Buffer Cache Performance. volume 30, pages 31–42.
- [Johnson and Shasha, 1994] Johnson, T. and Shasha, D. (1994). 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm.

- In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- [Kleinberg and Tardos, 2006] Kleinberg, J. and Tardos, É. (2006). *Algorithm Design*. Alternative Etext Formats. Pearson/Addison-Wesley.
- [Knuth et al., 1977] Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350.
- [Lin and Kernighan, 1973] Lin, S. and Kernighan, B. W. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516.
- [Megiddo and Modha, 2003] Megiddo, N. and Modha, D. S. (2003). ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, USA. USENIX Association.
- [Megiddo and Modha, 2004] Megiddo, N. and Modha, D. S. (2004). Outperforming LRU with an Adaptive Replacement Cache Algorithm. *Computer*, 37(4):58–65.
- [Michaelson, 1989] Michaelson, G. (1989). *An Introduction to Functional Programming through Lambda Calculus*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition.
- [Michie, 1968] Michie, D. (1968). “Memo” Functions and Machine Learning. *Nature*, 218(5136):19–22.
- [Mitchell, 1999] Mitchell, J. S. B. (1999). Guillotine Subdivisions Approximate Polygonal Subdivisions: A Simple Polynomial-Time Approximation Scheme for Geometric TSP, k -MST, and Related Problems. *SIAM Journal on Computing*, 28(4):1298–1309.

- [Nicola et al., 1992] Nicola, V. F., Dan, A., and Dias, D. M. (1992). Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing. *SIGMETRICS Perform. Eval. Rev.*, 20(1):35–46.
- [Norvig, 1991] Norvig, P. (1991). Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics - COLI*, 17(1):91–98.
- [Okasaki, 1998] Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.
- [O’Neil et al., 1993] O’Neil, E. J., O’Neil, P. E., and Weikum, G. (1993). The LRU-K Page Replacement Algorithm for Database Disk Buffering. *SIGMOD Record*, 22(2):297–306.
- [Pagh and Rodler, 2004] Pagh, R. and Rodler, F. F. (2004). Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144.
- [Panagakis et al., 2008] Panagakis, A., Vaios, A., and Stavrakakis, I. (2008). Approximate Analysis of LRU in the Case of Short Term Correlations. *Computer Networks*, 52(6):1142–1152.
- [Papadimitriou, 1977] Papadimitriou, C. H. (1977). The Euclidean Traveling Salesman Problem is NP-Complete. *Theoretical Computer Science*, 4(3):237–244.
- [Patterson and Hennessy, 2013] Patterson, D. A. and Hennessy, J. L. (2013). *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.
- [Punnen, 2007] Punnen, A. P. (2007). *The Traveling Salesman Problem: Applications, Formulations and Variations*, pages 1–28. Springer, Boston, MA, USA.

- [Rodeker et al., 2009] Rodeker, B., Cifuentes, M., and Favre, L. (2009). An Empirical Analysis of Approximation Algorithms for Euclidean TSP. In *Proceedings of the 2009 International Conference on Scientific Computing, CSC*, pages 190–196, Las Vegas, Nevada.
- [Sahni and Gonzalez, 1976] Sahni, S. and Gonzalez, T. (1976). P-Complete Approximation Problems. *Journal of the ACM*, 23(3):555–565.
- [Sleator and Tarjan, 1985] Sleator, D. D. and Tarjan, R. E. (1985). Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208.
- [Smith, 1978] Smith, A. J. (1978). Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247.
- [Smith and Goodman, 1983] Smith, J. E. and Goodman, J. R. (1983). A Study of Instruction Cache Organizations and Replacement Policies. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, ISCA '83*, pages 132–137, New York, NY, USA. Association for Computing Machinery.
- [So and Rechtschaffen, 1988] So, K. and Rechtschaffen, R. (1988). Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709.
- [Taher et al., 2018] Taher, S. J., Ghazali, O., and Hassan, S. (2018). A Review on Cache Replacement Strategies in Named Data Network. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 10(2–4):53–57.
- [Trevisan, 1997] Trevisan, L. (1997). When Hamming Meets Euclid: The Approximability of Geometric TSP and MST (Extended Abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, STOC '97*, pages 21–29, New York, NY, USA. Association for Computing Machinery.

- [Vazirani, 2001] Vazirani, V. V. (2001). *Approximation Algorithms*. Springer-Verlag, Berlin, Heidelberg.
- [Winters et al., 2020] Winters, T., Manshreck, T., and Wright, H. (2020). *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media.
- [Worboys, 1986] Worboys, M. (1986). The Traveling Salesman Problem (A Guided Tour of Combinatorial Optimisation), edited by E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. *The Mathematical Gazette*, 70(454):327–328.
- [Zebchuk et al., 2008] Zebchuk, J., Makineni, S., and Newell, D. (2008). Re-examining Cache Replacement Policies. In *2008 IEEE International Conference on Computer Design*, pages 671–678. IEEE.
- [Zhao Weizhong and Daming, 2007] Zhao Weizhong, F. H. and Daming, Z. (2007). Improvement and Implementation of a Polynomial Time Approximation Scheme for Euclidean Traveling Salesman Problem. *Journal of Computer Research and Development*, 44(10):1790.
- [Zhou et al., 2001] Zhou, Y., Philbin, J., and Li, K. (2001). The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 91–104, USA. USENIX Association.