ON LABELED PATHS

by

NATHAN MICHAEL WIEGAND

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the Doctor of Philosophy in the
Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2010

ABSTRACT

Labeled graph theory is the marriage of two common problem domains to computer science—graph theory and automata theory. Though each has been independently studied in depth, there has been little investigation of their intersection, the labeled paths.

This dissertation examines three results in the area of labeled path problems. The first result presents an empirical analysis of two context-free labeled all-pairs shortest-path algorithms using MapReduce as the experimental platform. The second and third results examine labeled paths in the context of formal languages beyond the context free languages. The second result is a lower bound on the length of the longest shortest path when the formal language constraining the path is a member of the control language hierarchy. Finally, the third result presents a labeled all-pairs shortest-path algorithm for each level of the infinite K_{Linear}-Hierarchy.

LIST OF ABBREVIATIONS AND SYMBOLS

LLNF	Labeled Linear Normal Form
$A \underset{C}{\overset{*}{\Rightarrow}} \sigma$	The production A derives the string σ if it is controlled by the language C.
CFL-APSP	Context-free language constrained all-pairs shortest path
$u \rightsquigarrow v$	The path from u to v.
$u \overset{L}{\rightsquigarrow} v$	The path from u to v which is validly labeled in L.

ACKNOWLEDGMENTS

I would like to thank both Dr. Richard Borie and Dr. Phillip Bradford for not only the invaluable lessons I have learned but also for their efforts relating to this dissertation. I would also like to thank Google, Inc. for allowing me to use a very generous number of CPU hours of their MapReduce system for the experiment described in Chapter 3. Finally, I would like to thank my friends and family for helping me maintain focus and obtain my goal.

CONTENTS

ABSTRACT ii

LIST OF ABBREVIATIONS AND SYMBOLS iii

ACKNOWLEDGMENTS. iv

LIST OF TABLES. vii

LIST OF FIGURES viii

1 INTRODUCTION 1

1.1 Introduction 1

1.2 Works of Barrett, et al. 4

1.3 Fast Dyck and Semi-Dyck Constrained Shortest Paths on DAGs 8

1.4 Structure of The Dissertation 11

2 DISTRIBUTED ALGORITHM FOR CFL-APSP AND COMPLEXITY OF WIRELESS MESH ROUTING METRICS 12

2.1 A Distributed Context-Free Language Constrained Shortest Path Algorithm 12

2.2 Complexity Results on Labeled Shortest Path Problems from Wireless Routing Metrics 19

2.3 Conclusion 27

3 EMPIRICAL ANALYSIS OF CFL-APSP ALGORITHMS 28

3.1 Introduction 28

3.2 BT-FastBJM 29

3.3 Experiment Structure 30
3.4 Results 37
3.5 Conclusion 40

4 LOWER BOUNDS ON CLH_k ROUTING TABLES 46
4.1 Introduction 46
4.2 Successor Matrices 49
4.3 Compression of Routing Tables for CLH_k Labeled Paths 54
4.4 Conclusion 57

5 LABELED PATH ALGORITHMS FOR THE k-HIERARCHY 58
5.1 Labeled Normal Form 61
5.2 $K_{Linear,2}$ APSP Algorithm 63
5.3 Analysis of the $K_{Linear,2}$ APSP Algorithm 73
5.4 K_{Linear}-Hierarchy APSP Algorithms 74
5.5 Analysis of the $K_{Linear,k}$ APSP Algorithm 79
5.6 Conclusion 79

6 CONCLUSION AND FUTURE WORK 81
6.1 Further Open Questions 82
6.2 Conclusion 84

BIBLIOGRAPHY 85

LIST OF TABLES

3.1 Context-Free Languages 35

3.2 Context-Free Languages 36

4.1 $len(k)$ table for longest common substrings 56

LIST OF FIGURES

2.1 Heap Element Distribution Intersecting Elements of a Row/Column Cross . 16
2.2 Multi-Radio Multi-Hop Wireless Network Example 21

3.1 Path probability vs. edge/vertex ratio for graphs with 25 vertices 38
3.2 Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 1-parenthesis language 39
3.3 Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 2-parenthesis language 40
3.4 Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 3-parenthesis language 41
3.5 Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 4-parenthesis language 42
3.6 Number of comparisons for both BJM and FastBJM with edge probability 0.25 and simple language 42
3.7 Number of comparisons for both BJM and FastBJM with edge probability 0.25 and palindrome language 43
3.8 Ratios of graphs with no negative cycles. 43
3.9 Ratios of graphs with no negative cycles. 44
3.10 Running times of `BJM`, `FastBJM`, `BT-FastBJM` compared over different formal languages 45

4.1 *Widget* — a graph G with a long shortest path. 50
4.2 Special graph with a large successor matrix but a lot of redundancy 53

CHAPTER 1 – INTRODUCTION

This dissertation is concerned with aspects of labeled graph theory. Labeled graph theory is the marriage of two common problem domains to computer science—graph theory and automata theory. More specifically, this dissertation examines several algorithms and complexity results which pertain to shortest-path problems in labeled graphs.

1.1 Introduction

Graph theory is an important area in computer science, with numerous applications including operating systems, programming languages, and networking. Traditionally, the most common graph theoretic model has been that of the weighted directed graph. However, there are many problems which cannot be fully modeled without additional graph properties, such as processor assignment for scheduling problems or wireless channels for network simulations. To express such constraints, labeled graphs may be used. While there exist a number of algorithms for a variety of labeled graph theory problems, the field has seen surprisingly little coherent examination. This chapter examines several fundamental papers in the area, particularly those relating to language constrained shortest paths.

The survey written by Zwick (Zwick 2001) contains a large number of algorithms for solving path problems and also for approximating solutions to some of these problems. It provides a brief discussion of myriad path problems, including those over directed and undi-

rected graphs, and those over weighted and unweighted graphs. This is yet more evidence that the area of graph problems, especially that of finding paths, is rich and still interesting. Many of these problems deserve investigation in a context where formal language constraints are applied.

Section 1.1.1 covers several definitions which are necessary to better understand the topics contained within this text. Section 1.1.2 is concerned with existing applications of formal language constrained graph problems. Section 1.2 contains a discussion of the works of Barrett, et al. (Barrett, Jacob, and Marathe 2000; Barrett, Bisset, Jacob, Konejevod, and Marathe 2002; Barrett, Bisset, Holzer, Konjevod, Marathe, and Wagner 2007) which relate to the TRANSIMS project for transportation networks. It also provides a discussion of polynomial time algorithms for solving both the regular-language constrained shortest path problem and the context-free language constrained shortest path problem. Further, it discusses other theoretical results, including the NP-Completeness of some of the variants of formal-language constrained path problems. Section 1.3 presents a discussion of the work of Bradford and Choppella (Bradford and Choppella 2010) which provides an $O(n^\omega \log n)$ algorithm for computing shortest paths on DAGs with either a Dyck or Semi-Dyck language, where ω is the best exponent of running time for matrix multiplication.

1.1.1 Definitions

A formal language constrained graph algorithm has two components: the graph, and the formal language which provides constraints. The graph has edges labeled with symbols from some alphabet Σ which is associated with a formal language L. This graph may be directed, or undirected, and may or may not have edge weights. Formally, $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V \times \Sigma$.

A formal-language constrained shortest path is the shortest path in a graph G such that the concatenation of the edge labels along the path forms a valid string in some formal language L. We use the notation $u \overset{L}{\rightsquigarrow} v$ to denotes the path between two vertices, u and v

that the concatenation of the edge labels along the path yields a valid string in the formal language L.

The formal language greatly affects the complexity of the problem. For example, linear regular languages—those that can be defined by the regular expression $a_0^+ a_1^+ ... a_k^+$—are much simpler than those defined by context-free grammars, and context sensitive languages are harder yet. Sections 1.2 and 1.2.2 discuss these complexities further.

We assume that context-free languages are modeled via context-free grammars which are in Chomsky Normal Form. That is, every production has either two non-terminals on the right hand side, or exactly one terminal. It is known that all context-free grammars can be expressed in Chomsky Normal Form.

1.1.2 Applications of Formal Language Constrained Graph Problems

Graph problems have been a major area of interest to computer scientists since perhaps the inception of the field. Labeled graph problems, like their unlabeled counterparts, are more than just academic curiosities and have many applications. Section 1.2 discusses the application of these problems to the TRANSIMS project for transportation route planning. Section 1.3 shows that program analysis can be addressed using particular classes of context-free language constrained path problems, namely the Dyck and Semi-Dyck classes of context-free languages.

Another important application of these techniques is with use in the field of database theory. Yannakakis, in (Yannakakis 1990), showed that query processing can be done using regular language constrained shortest paths. His paper provides a very similar algorithm to Barrett, et al.'s regular-language constrained shortest path algorithm (see Section 1.2.1). In addition to this algorithm, he also addresses more applied concerns of query processing, such as memory limitations and parallelization.

1.2 Works of Barrett, et al.

The TRANSIMS (Barrett, Bisset, Jacob, Konejevod, and Marathe 2002; Barrett, Bisset, Holzer, Konjevod, Marathe, and Wagner 2007) project is funded by the Department of Transportation and by the Environmental Protection Agency. The purpose of this project is to evaluate the economic and social impact of building new roads in metropolitan areas. Barrett, et al.'s work involves using very large graphs which model all the streets, pedestrian paths, and railways in several urban centers to compute route planning given certain constraints. Having this graph alone is not sufficient for the purpose, as there are implicit constraints in such a transportation network. For example, an edge in this graph represents any of roads, train tracks, or pedestrian paths, but a car should not be routed via a train track and a train should not attempt to take a walking-bridge. So, to solve this problem, the graph is labeled with a symbol from $\{r, t, w\}$—road, train, and pedestrian respectively.

Given such a graph representing a metropolitan area (Portland, Oregon in the original study) the task of TRANSIMS is to use activity information from demographic data to find optimal travel choices for each individual traveler. In order to evaluate possible paths with various travel requirements (pedestrian only, or car only, etc.) Barrett, et al. developed a technique of finding regular-language constrained shortest paths (See the next subsection). Having such a shortest path algorithm, it is necessary to model modes of travel with regular expressions. A traveler who wishes to travel from home to work only by walking to and from a train station can be modeled using the linear regular expression: $w^+t^+w^+$. A traveler who wishes to drive to work can be modeled by: r^+.

The TRANSIMS project finds the shortest paths between some source and some destination for millions of routes. The method that Barrett gives for finding such constrained shortest paths is such that any shortest path algorithm can be used after the graph has been modified to include the regular expressions. The papers (Barrett, Bisset, Jacob, Konejevod, and Marathe 2002) and (Barrett, Bisset, Holzer, Konjevod, Marathe, and Wagner 2007)

exploit this by using several shortest path algorithms, and even shortest path approximations. Also, since for their purposes they are evaluating linear regular expressions (those of the form $a_0^+ a_1^+ ... a_k^+$ for each $a_i \in \Sigma$), the regular expression can be implicitly computed. The algorithm would take as input a string $a_0 a_1 ... a_k$. Then, the shortest path algorithm will remember its current position in the string and only transition to those edges which are allowable. That is, if the current position in the string is i then the only edges allowable are those labeled with the label at position i or $i + 1$ in the string.

Barrett, et al.'s paper **??** evaluates an implementation of the regular language constrained shortest path problem with both Dijkstra's shortest path algorithm, and also the Sedgewick-Vitter heuristic. The latter works by biasing the search using Euclidean distances. This will not work well for a standard shortest path on labeled graphs, but is very conducive to the TRANSIMS project as it models a city.

In a technical report published in 2007, Barrett evaluates the regular language constrained algorithm further using two other modified versions of the shortest path algorithm. The first is with a goal-directed (A^*) search algorithm, the second with a bidirectional search. The former works by evaluating Euclidean distance and biasing towards the goal and can be modified to give approximate results using the Sedgewick-Vitter Heuristic. The latter is the standard Dijkstra's algorithm except that the search takes place simultaneously from both the source and destination.

1.2.1 Algorithms for RE and CFL Constrained Shortest Paths

For the TRANSIMS project Barrett, et al. developed an algorithm (Barrett, Jacob, and Marathe 2000) for solving the regular language constrained shortest path problem. The algorithm takes as input a graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V \times \Sigma$ and a deterministic finite automata, R, corresponding to a regular language L over some alphabet Σ.

The algorithm computes $G \times R$ in the following way. A new graph, G' is created with

vertex set $V(G) \times V(R)$. That is, the vertex set of G' consists of tuples (g_i, r_j). An edge exists in G' between vertices (g_i, r_j) and (g_k, r_ℓ) if and only if there are identically labeled edges between g_i and g_k in G and between r_j and r_ℓ in R. The edges in G' have the same edge weight as the edges in G. By taking the product of the two graphs, and requiring that the edges in the new graph are labeled with the same label as the edges in the corresponding graphs, all paths in the new graph (from the start state to a final state) are validly labeled. Also, since there are only edges between vertices in the new graph that correspond to edges in G, there are no paths that can be taken which do not correspond to paths in G. Therefore, for each labeled shortest path in G there exists an unlabeled shortest path in G'. Since only validly labeled paths exist in G', the edges in G' need not be labeled at all, and thus G' is conducive to any of the many shortest path algorithms for weighted graphs.

Also, in the work where Barrett, et al. presented the algorithm for the regular language constrained shortest path problem, an algorithm was presented for solving the context-free language constrained shortest path problem. This algorithm takes as input a graph, $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V \times \Sigma$. Like the regular-language version of the problem, Σ is the alphabet for the formal language L. In this case, the formal language must be a context-free language defined by a grammar which is in Chomsky-Normal Form (CNF). CNF grammars are those which have either exactly two non-terminals on the right hand side of a production, or exactly one terminal. The only non-terminal which can derive ϵ is S and then it may do so only when S is not used on the right hand side of any production. The context free grammar $C = (N, \Sigma, P, S)$, where N is the set of non-terminals, Σ is the alphabet, P is the set of productions (in CNF), and S is the start non-terminal. The paper presents the algorithm as the following dynamic programming recurrence:

$$D[u][v][t] = \begin{cases} w(u, v, t) & \text{if } (u, v, t) \in E \\ \infty & \text{otherwise} \end{cases}$$

$$D[u][v][A] = \min \begin{cases} \min\limits_{A \to BC} \{ \min\limits_{k \in V} \{ D[u][k][B] + D[k][v][C] \} \} \\ D[u][v][t] \text{ if } (A \to t) \in P \\ 0 \text{ if } u = v \text{ and } A = S \text{ and } (S \to \epsilon) \in P \end{cases}$$

where u and v are vertices; A, B, and C are non-terminals; and t is a terminal. The function w returns the weight of the edge between u and v labeled with t.

An algorithm which implements the above dynamic programming recurrences was presented in Bradford and Thomas (Bradford and Thomas 2009). The algorithm sets at least one value of the table per iteration. The table is of size $|V|^2(|N| + |\Sigma|)$. The value of $|\Sigma|$ are set in the first equation above and are never changed. Each iteration of the algorithm permanently sets one entry in the D table. Because there are $|V|^2|N|$ entries in the table which can be set during the computation of the second recurrence above, the algorithm need only execute $|V|^2|N|$ times in order to be certain of completion. If after this number of executions the table is still unstable, then there are no validly labeled shortest paths (or in the case of negative edge weights, there might be a negatively weighted cycle). So, this algorithm is $O(|V|^5|N|^2|P|)$ worst case.

Also in the work which presents the $O(|V|^5|N|^2|P|)$ algorithm, Barrett, et al. sketches an algorithm which Bradford and Thomas (Bradford and Thomas 2009) state explicitly which, using a priority queue, is able to solve the context-free language shortest path problem in $O(|V|^3|N||P|)$ (calling it the FastBJM algorithm). A more in-depth discussion of these algorithms is provided in Section 2.1, along with a parallel version of the FastBJM algorithm which can solve the problem on up to $O(|V||N|)$ compute nodes.

1.2.2 Theoretical Results

In addition to polynomial time algorithms for both the regular and context-free language constrained shortest path problems, Barrett, et al. show several results for other classes of graphs and formal languages, and also for the version of the problem where only simple

(acyclic) paths are considered. The paper shows that when considering simple paths, even finding labeled paths constrained by free finite languages is NP-Complete. The proof follows from a reduction from the Hamiltonian Path problem, which is NP-Complete.

A finite language is a formal language L if and only if there are finitely many words in the language. A free finite language is a finite language which is considered part of the input to the algorithm. To reduce the Hamiltonian Path problem to the free finite-language constrained simple path problem, simply modify the input graph G of the Hamiltonian Path problem by giving all edges in G the same label, say a. Now, query the free finite-language constrained simple path algorithm for a path in this newly labeled G (with n vertices) that is validly labeled from the language $\{a^{n-1}\}$. This enforces that the path contain exactly n vertices since it finds only simple paths. Therefore, free finite-language constrained simple paths is NP-Complete (for the class of graphs that are NP-Complete under the Hamiltonian Path problem). This result means that anything more complex than this (including regular languages) must be at least NP-Hard.

The paper also shows, by a clever reduction from 3-SAT, that the regular-language simple path problem is NP-Hard for other classes of graphs including cliques, interval graphs, chordal graphs, complete meshes, and permutation graphs. However, when the graph class is limited to treewidth-k graphs, the problem becomes polynomial again for formal languages at least as restrictive as regular languages.

When context sensitive languages are considered, finding simple paths becomes PSPACE-Complete, but finding shortest paths becomes undecidable.

1.3 Fast Dyck and Semi-Dyck Constrained Shortest Paths on DAGs

Bradford and Choppella are currently preparing a work (Bradford and Choppella 2010) that constrains the general labeled graph problem on two fronts: a specific class of graphs, and a specific class of grammar. Firstly, their algorithm operates on DAGs—directed acyclic graphs. Since these graphs are acyclic, they avoid some of the problems associated with

finding shortest paths in the presence of negative edge weights, namely negatively weighted cycles. Secondly, their algorithm takes as part of its input a context free language which is restricted to either a Dyck language or a Semi-Dyck language. The paper states that both syntactic unification and program analysis can be reduced to path problems over Dyck and Semi-Dyck languages. Further, in inter-procedural dataflow analysis Dyck and Semi-Dyck constrained paths can be used to determine which procedure calls match which returns.

Both Dyck languages and Semi-Dyck languages are parenthesis languages. That is, every terminal γ has an inverse γ^{-1}. An example of this would be the familiar parenthesis: '(' and ')'. A Semi-Dyck language is the parenthesization which is used in mathematical expressions. The following is an example of a Semi-Dyck language with one type of parentheses:

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow \epsilon \\ S &\rightarrow \text{`(' } S \text{ `)'} \end{aligned}$$

A Dyck language is not as restricted and has that $\gamma\gamma^{-1} = \gamma^{-1}\gamma = \epsilon$. That is, it allows for either parenthesis to 'open' the expression and enforces that its opposite 'close' the expression. The grammar for a Dyck language with one parenthesis type is as follows:

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow \epsilon \\ S &\rightarrow \text{`(' } S \text{ `)'} \\ S &\rightarrow \text{`)' } S \text{ `('} Barret, etal.'s \end{aligned}$$

By restricting the classes of both the input graph and the input formal language, Bradford and Choppella are able to improve the running time over general label constrained graph problems and achieve a running time of $O(|V|^{\omega} \log |V|)$, where ω is the best possible exponent

for the matrix multiplication algorithm of a $|V| \times |V|$ matrix. At the time of this paper, ω is known to be at most 2.376 which was shown by Coppersmith and Winograd (Coppersmith and Winograd 1987).

The paper begins by describing Nykänen and Ukkonen's (Nykänen and Ukkonen 2002) algorithm for exact path length algorithm. This algorithm is then improved from an $O(n^3)$ running time (where $n = |V|$)to the $O(n^\omega \log n)$ running time by replacing key components with more efficient ones—namely, replacing lists in an adjacency matrix with balanced binary trees. The key to this algorithm's efficiency is restricting the possible edge weights in the graph to members of the set $\{-1, 0, 1\}$. These weights naturally represent a parenthesis and its inverse.

In the case of only a single type of parenthesis, a zero-weighted path in this graph corresponds to a valid parenthesization. However, the problem becomes a bit more challenging again when multiple types of parentheses are allowed. Bradford and Choppella address this problem by creating new graphs for each type of parenthesis. Provided a boolean matrix for one of these graphs, raising it to the n^{th} power will show whether or not there exists a path from u to v for all $u, v \in V$ by containing a 1 (if there is a path) in the cell corresponding to u and v in the new matrix. This does not solve the problem, however, as these separate graphs (made by considering a single parenthesis type at a time) may not yield a correct solution to the overall problem. Bradford and Choppella address this problem by providing a method of combining these separate graphs in a manner which does allow for correctly labeled paths in the original graph.

Bradford and Choppella's result shows that both Dyck and Semi-Dyck constrained shortest path problems can be solved in $O(n^\omega \log n)$ on DAGs. This result is important to the areas of program analysis and syntactic unification and likely other areas as well. However, it does merit further investigation to see if this bound is true for all context-free languages over DAGs, or if it is just a convenient property of Dyck languages that allows this sub-n^3 bound.

1.4 Structure of The Dissertation

The rest of the dissertation is structured as follows. Chapter 2 contains a discussion of previously published work that I did in this area in conjunction with other authors. These works include a distributed CFL-APSP algorithm and a complexity result for wireless routing metrics. Chapter 3 provides an in-depth empirical analysis of various CFL-APSP algorithms in the face of random graphs and context-free grammars with various properties. Chapter 4 proves a lower bound on the size of a successor matrix (routing table) when the formal language is constrained to be from the Control Language Hierarchy. Chapter 5 provides an algorithm for finding shortest paths in graphs where the formal language comes from the K_{linear}-hierarchy. And finally, Chapter 6 holds some final remarks and future work to be considered.

CHAPTER 2 – DISTRIBUTED ALGORITHM FOR CFL-APSP AND COMPLEXITY OF WIRELESS MESH ROUTING METRICS

The purpose of this chapter is to further motivate the rest of the dissertation by presenting work that I have done with other authors in this area that has already been published. The first result is a distributed algorithm for solving the context-free language constrained shortest path problem. The paper that presented this algorithm (Ward, Wiegand, and Bradford 2008) was published in the International Conference on Parallel Processing. The second result is a complexity result on labeled shortest path problems relating to wireless routing metrics. This result appears in the journal Computer Networks (Ward and Wiegand 2010).

2.1 A Distributed Context-Free Language Constrained Shortest Path Algorithm

Barrett, et al. in (Barrett, Jacob, and Marathe 2000) provide the following dynamic programming recurrence to solve the CFL-APSP problem:

$$\delta[u][v][t] = \begin{cases} w(u,v,t) & \text{if } (u,v,t) \in E \\ \infty & \text{otherwise} \end{cases}$$

$$\delta[u][v][A] = \min \begin{cases} \min\limits_{A \to BC} \{ \min\limits_{k \in V} \{ \delta[u][k][B] + \delta[k][v][C] \} \} \\ \delta[u][v][t] \text{ if } (A \to t) \in P \\ 0 \text{ if } u = v \text{ and } A = S \text{ and } (S \to \epsilon) \in P \end{cases}$$

This recurrence algorithm requires a dynamic programming table δ to be a $|V| \times |V| \times (|N| + |\Sigma|)$ table. The entry $\delta[u][v][t]$ corresponds to the weight of the edge, if there is one, between the two vertices u and v that is labeled with the terminal t. Entry $\delta[u][v][A]$ corresponds to the weight of the shortest path between u and v which is validly labeled from strings derivable from the non-terminal A. Thus, the entry $\delta[u][v][S]$ corresponds to the weight of the shortest path between u and v which is validly labeled with strings from the input language L. The second equation from above assigns each value in the δ-table exactly one time.

2.1.1 BJM and FastBJM

Bradford and Thomas (Bradford and Thomas 2009) explicitly provide an algorithm for the above dynamic programming recurrences. Algorithm 2.1 implements the first equation from above.

Algorithm 2.1 InitializeMatrixD(G)

```
1: for all u, v ∈ V × V do
2:    for all A ∈ N do
3:       D[u][v][A] = ∞
4:    end for
5: end for
6: if (S → ϵ) ∈ P then
7:    for all v ∈ V do
8:       D[v][v][S] = 0
9:    end for
10: end if
11: for all (u, v, t) ∈ V × V × Σ do
12:    if (u, v, t) ∈ E then
13:       for all (A → t) ∈ P do
14:          D[u][v][A] = w(u, v, t)
15:       end for
16:    end if
17: end for
```

The second equation can more or less be translated directly. This yields an $O(|V|^5|N|^2|P|)$ algorithm which solves the CFL-APSP problem, as seen in Algorithm 2.2.

Algorithm 2.2 BJM(G)

1: InitializeMatrixD(G)
2: **for all** (i from $1 \ldots |V|^2|N|$) **do**
3: **for all** $(u, v, A) \in V \times V \times N$ **do**
4:

$$D[u][v][A] = \min \left\{ \begin{array}{r} D[u][v][A], \\ \min\limits_{(A \to BC) \in P} \{ \min\limits_{k \in V} \{ D[u][k][B] + D[k][v][C] \} \} \end{array} \right.$$

5: **end for**
6: **end for**

The calculation of the minimum in line 4 of Algorithm 2.2 requires $O(|V||P|)$ work since one must evaluate all productions and look at all vertices. The intuition behind this step is that $u \overset{A}{\rightsquigarrow} v$ must be a single edge, or it must be through an intermediate vertex k which has $u \overset{B}{\rightsquigarrow} k$ and $k \overset{C}{\rightsquigarrow} v$ for the production $A \to BC$. Thus, because of the outer two loops, this algorithm is $O(|V|^5|N|^2|P|)$.

If we restrict the graph to having non-negative edge weights, then we can modify the above algorithm to run in $O(|V|^3|N||P|)$. The aforementioned work by Barrett, et al. also provides this, asymptotically better, algorithm. It works by using a priority queue to examine the $|V|^2|N|$ elements of the D table in order of the smallest current estimate of the final distance. Using a Fibonacci Heap (Fredman and Tarjan 1987), we can obtain the desired amortized bound. However, we can obtain the exact bound desired by using a Brodal Queue, which has worst case $O(\log n)$ time for a $extractMin$ operation (Brodal 1996).

Algorithm 2.3 presents what Bradford and Thomas call FastBJM. Line 9, $P_D(u, v_0, A)$ is the pointer to the value $D[u][v_0][A]$. Line 4 gets the current minimum value from the heap. This value is a tuple consisting of a pointer to an entry in the D table and the actual minimum distance, $w(|u \overset{A}{\rightsquigarrow} v|)$. Thus, this value is permanently set. Next, we check to see which keys to decrease by finding all other paths of which this is a direct sub-component.

Note that $decreaseKey$ is a constant time operation in a Brodal Queue (or amortized constant in the case of a Fibonacci Heap). However, $extractMin$ is $O(\log n)$ time. Since

Algorithm 2.3 FastBJM(G)

```
1: InitializeMatrixD(G)
2: InitializeHeapH(G)
3: while H ≠ ∅ do
4:    (key, (u, v, X)) ← extractMin(H)
5:    D[u][v][X] = key
6:    for all (A → BC) ∈ P do
7:       if B = X then
8:          for all v_0 ∈ V do
9:             L ← P_D(u, v_0, A)
10:            val ← D[u][v][B] + D[v][v_o][C]
11:            if val < D(L) then
12:               decreaseKey(H, L, val)
13:            end if
14:         end for
15:      end if
16:      if C = X then
17:         for all u_0 ∈ V do
18:            L ← P_D(u_0, v, A)
19:            val ← D[u_0][u][B] + D[u][v][C]
20:            if val < D(L) then
21:               decreaseKey(H, L, val)
22:            end if
23:         end for
24:      end if
25:   end for
26: end while
```

the heap initially has size $|V|^2|N|$ and each iteration of the *while*-loop removes one element from the heap, the *while*-loop will iterate $O(|V|^2|N|)$ times. Thus, the total work for this algorithm is $O\big(|V|^2|N|(\log|V|^2|N| + |V||P|)\big) = O\big(|V|^3|N||P|\big)$.

2.1.2 A Distributed FastBJM Algorithm

Since the FastBJM algorithm is dominated by a cubic term, the actual work required can quickly grow impractical. In such a case, it would be necessary to solve this problem in parallel. Our paper provides an algorithm which allows the distribution of work onto at most $O(|V||N|)$ nodes. It does so by a clever distribution of a global priority queue.

As noted before, the outer-most loop of FastBJM will iterate at most $O(|V|^2|N|)$ times

since the heap has that many elements. The next inner loop will loop $|P|$ times. Lastly, there are the inner most loops which loop $|V|$ times. So, in order to effectively distribute this algorithm, we need to spread the work out as evenly possible. It needs to be the case that each node is responsible for a constant amount of work during each round of the algorithm. If we were to naively spread responsibility of the heap to the nodes by making one node responsible for all information about a single vertex in the graph, then that node could be responsible for $O(|V|)$ work during the round. So, instead we distribute the heap itself in such a way that each node will be responsible for a constant amount of information that may be used during each round.

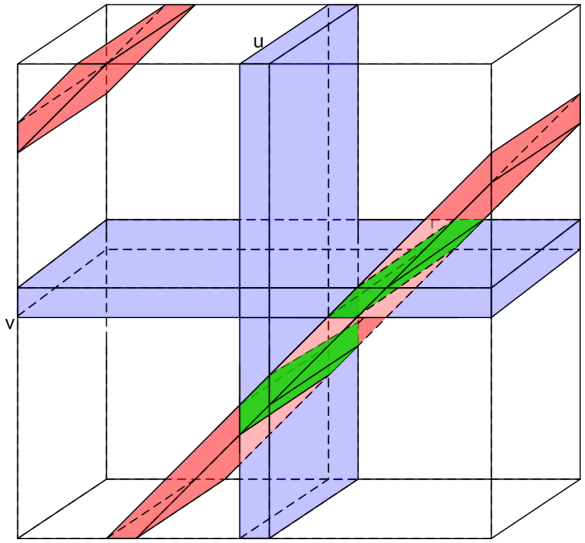

Figure 2.1. Heap Element Distribution Intersecting Elements of a Row/Column Cross

Figure 2.1 shows how the heap is distributed. Each node in the graph is responsible for the heap elements which correspond to a diagonal through the D-table. The slices (in blue) that are labeled u and v correspond to those rows and columns of the D-table corresponding to paths starting at u, and those paths which terminate at v. The diagonal through this space intersects these two slices in at most two places. This means that each node only has information about two paths involving u or v, one which begins at u and terminates at v_o

(the green intersection with the u column above) and one which begins at u_o (the green intersection with the v row above) and terminates at v.

Algorithm 2.4 D-FastBJM(G): (Code for node p_i)

```
 1: InitializeMatrixD(G)
 2: InitializeHeapH(G)
 3: Construct Minimum Spanning Tree of Communication Graph
 4: while H ≠ ∅ do
 5:    (key, (u, v, X)) ← globalExtractMin(H)
 6:    D[u][v][X] = key
 7:    for all (A → BC) ∈ P do
 8:       if B = X then
 9:          v_0 ← (u + i)  mod |V|
10:          L ← P_D(u, v_0, A)
11:          val ← D[u][v][B] + D[v][v_o][C]
12:          if val < D(L) then
13:             decreaseKey(H, L, val)
14:          end if
15:       end if
16:       if C = X then
17:          u_0 ← (v − i)  mod |V|
18:          L ← P_D(u_0, v, A)
19:          val ← D[u_0][u][B] + D[u][v][C]
20:          if val < D(L) then
21:             decreaseKey(H, L, val)
22:          end if
23:       end if
24:    end for
25: end while
```

Given that each node only does a constant amount of work per round, it is now necessary to explain how the minima are communicated so that the global state is correctly maintained. The key lies in $globalExtractMin()$ as defined in Algorithm 2.5 along with the communication structure.

For the communication structure we assume no particular underlying network topology. Line 3 in D-FastBJM creates a minimum spanning tree in this network. The minimum spanning tree of a network of size n can be constructed in $O\left(\sqrt{n}\log^* n\right)$ via an algorithm

Algorithm 2.5 globalExtractMin() — Code for node p_i

```
1: if Children = ∅ then
2:    (key, id) ← peekMin()
3:    send <peek, (key, id)>
4: else
5:    for all c ∈ Children do
6:       receive <peek, (key, id)_c>
7:    end for
8:    (minKey, minID) ← min{min{(key, id)_c}, (peekMin(), i)}
9:    if Parent ≠ ∅ then
10:      {COMMENT This is not the root node of the minimum spanning tree}
11:      send <peek, (minKey, minID)>
12:      receive <heapMin, (gMin, gID)>
13:      if gID = i then
14:         extractMin()
15:      end if
16:      return gMin
17:   else
18:      {COMMENT This is the root node of the minimum spanning tree}
19:      broadcast <heapMin, (minKey, minID)>
20:      if minID = i then
21:         extractMin()
22:      end if
23:      return minKey
24:   end if
25: end if
```

credited to Garay, Kutten, and Peleg (Garay, Kutten, and Peleg 1998). Thus, the root of the tree will act as the leader of each round.

The *globalExtractMin*() method behaves differently for each of leaf, internal, and root nodes in the communication network. First, the leaf nodes will *peekMin*() getting the current minimum key-id tuple from its own heap, then it will send a **peek** message containing this tuple to its parent. All internal nodes will listen for all of the **peek** messages from its children. It will then find the minimum value of all of the values it received along with the minimum from *peekMin*() with its own heap. Finally, the parent node computes the minimum from its children and itself. This will be the global minimum value for this round. The next phase of the algorithm is for the root to broadcast the global minimum tuple to the

whole network. Each node receives this broadcast message and checks the global minimum to see if it was the originator of the value. If so, then the value is actually removed from its own heap.

By using the minimum spanning tree in this way, each node only sends a single message. Thus, there are n messages sent—where n is the number of compute nodes. Each node must wait on all of its children to send **peek** messages. Let Δ be the maximum degree of any node. So, each node must wait on at most $O(\Delta)$ messages. Further, this algorithm is not complete until all nodes have received the broadcast message. This will be complete after $O(h)$ where h is the height of the minimum spanning tree.

Therefore, D-FastBJM solves the CFL-APSP problem on $|V|$ nodes with $O(|V|^3|N|)$ messages, since there are $|V|^2|N|$ rounds with $O(|V|)$ messages each. Finally, each node does $O(\Delta|V|^2|N||P|)$ work, for a total of $O(\Delta|V|^3|N||P|)$ work.

Our paper also describes a means for distributing the work onto fewer than $|V|$ nodes by partitioning more of the heap to each node. The paper also discusses how to use $|V||N|$ nodes by further splitting the heap.

2.2 Complexity Results on Labeled Shortest Path Problems from Wireless Routing Metrics

Routing in computer networks is a well studied, though still active, area of research. We can consider a network as if it were a graph, with computers and routers represented by vertices and the links between them represented by edges. It might seem natural to assume that the shortest path between any two points is the optimal route that a message from one computer to another should take. However, the problem is more complex than that. Different links have different band-width or they may be physically far apart and the speed that the message can traverse that link must be taken into consideration. These are but a few of the variables which greatly increase the difficulty in creating routing algorithms.

One approach to routing is to create a routing metric which can assess the cost of dif-

ferent routes through the network, taking into account bandwidths, physical distances, hop-count, throughput, packet-delay, etc. Routing in wireless networks has several additional constraints, one being channel assignment.

Our paper, "Complexity Results on Labeled Shortest Path Problems from Wireless Routing Metrics" (Ward and Wiegand 2010), argues that finding shortest paths over a metric that exploits channel assignment is tantamount to finding labeled shortest paths. We use this to explore the complexity of two routing metrics: Weighted Cumulative Expected Transmission Time (WCETT) (Draves, Padhye, and Zill 2004) and Metric for Interference and Channel-switch (MIC) (Yang, Wang, and Kravets 2005).

2.2.1 Two Routing Metrics for Wireless Mesh Networks

Effectively evaluating the cost of using communications links is the basis for determining routing paths in networks. Though in wireless networks, accurately calculating the cost of a route is difficult. Not only must one consider the qualities of the individual links, such as their band-width or packet drop rate, but one must also take into consideration the interference between the links. Interference between routing links becomes particularly interesting when we allow routing nodes to have multiple wireless radios which can be tuned to separate, non-interfering channels.

In order to facilitate the discussions of WCETT and MIC, consider the example provided in Figure 2.2. Here, W represents the edge weight and C represents the channel that the wireless radio is operating on. In this example each node has exactly two radios which are tuned to different channels. The edges between the nodes represent two nodes having a possible communication link between them. That is, the nodes are within range of one another and they have radios on the same channel. The following are two metrics which attempt to route packets in such a network taking into consideration channel assignment.

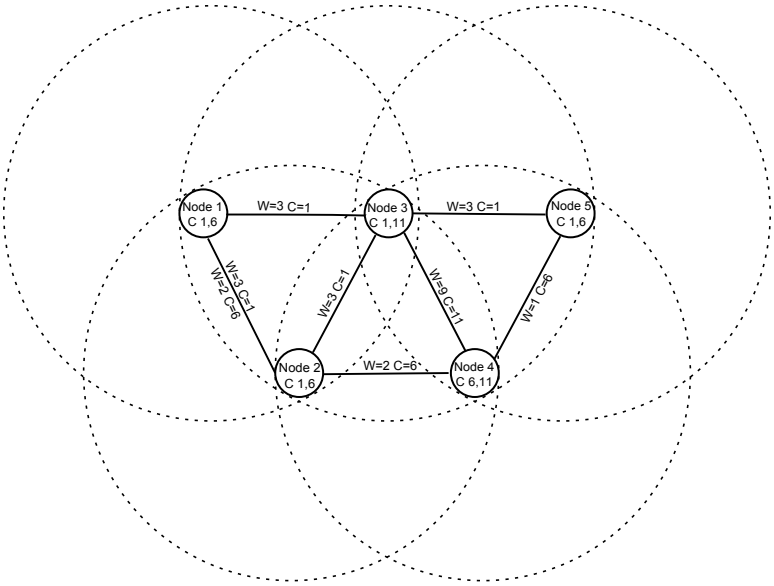

Figure 2.2. Multi-Radio Multi-Hop Wireless Network Example

2.2.2 WCETT

Weighted Cumulative Expected Transmission Time (WCETT) is a metric defined in (Draves, Padhye, and Zill 2004). It was designed to take into account not only the estimates of transmission time across a given link, but also the channel information in the network. The metric consists of two components, the first being the sum of Expected Transmission Times (ETT) for each link along a given path. ETT takes into account the expected number of times a packet must be transmitted before it is successfully received along with the bandwidth of the links. This will yield an estimate of the total transmission time of packets along the entire route. The other component of WCETT is designed to account for channel usage. It is the total ETT of the most used (by ETT) channel in the path.

Formally, WCETT is defined as follows (Draves, Padhye, and Zill 2004):

Definition 2.1. *Let p be a path in the network graph $G = (V, E)$, let C be a set of valid edge labels (wireless channels), let β be a tunable parameter in the range $[0, 1]$, fixed for any instance of the problem, and let $ETT : E \to \mathbb{R}^+$ be the weight function on the edges. Then*

the cost function WCETT is computed over p as follows:

$$WCETT(p) = (1-\beta)\sum_{e \in p} ETT(e) + \beta \max_{j \in C}\{X_j(p)\}$$

$$X_j(p) = \sum_{e \in p, label(e)=j} ETT(e)$$

Now, once again consider Figure 2.2. Let the W values in the example stand for the ETT values for each link. Also, assume that $\beta = 0.5$. So, the shortest $WCETT$ path from Node 1 to Node 5: $1 \xrightarrow{1} 2 \xrightarrow{6} 4 \xrightarrow{6} 5$. The weight of this path is 6 and since $\beta = .5$ the $WCETT$ equation is now:

$$WCETT(p) = 3 + \beta \max_{j \in C}\{X_j(p)\}$$

The second term of this equation is 3×0.5 for a total cost of 4.5. The path from 1 to 5 through node 3 has a weight of 6. Note, that it may appear that the edge from 1 to 2 via the link which is over channel 6 would be better since it has smaller weight. Though, note that $X_6(p)$ would become 5, for an overall $WCETT(p) = 5$ for the path. However, if $\beta = 1$ these paths will have the same weight and the shortest path would have a cost of 5: $1 \xrightarrow{6} 2 \xrightarrow{6} 4 \xrightarrow{6} 5$

2.2.3 MIC

The Metric of Interference and Channel-switch (MIC) for shortest paths in wireless mesh networks was introduced in (Yang, Wang, and Kravets 2005) for the purpose of improving upon $WCETT$. It consists of two main components. The first is the sum of IRU values along the path. These IRU values are estimates of the total amount of wasted transmission time. The estimate is calculated as the expected transmission time multiplied by the number of links which interfere with the current channel. So, this component is an approximation of how much this path interferes with other paths.

The second component of the MIC metric is the sum of weights for nodes along the path, where the weight is determined by the channels that are used for receiving and transmitting. If two links on a path are on the same channel then they may interfere. So, these are given weight w_2, and links that do not have conflicting channels are given wieght w_1. Thus, this second part of the metric captures the likelihood of interference between links on the path.

Definition 2.2. *Let p, G, C, ETT, be defined as for $WCETT$, while $min(ETT)$ is the smallest edge weight in the graph. Then if $v \in V$, let $prev(v, p)$ be the node preceding v in the path p or $\emptyset$ if there is no such node, and define $CH(\emptyset) = \epsilon$ and $CH(v)$ be the channel used for node v . Let N_e be the size of the set of neighboring edges which link e interferes with (has the same channel assignment). Finally, $w_1, w_2 \in \mathbb{R}^+$, $O \leq w_1 < w_2$ are channel-switching costs. Then define the cost function MIC over p as follows:*

$$
\begin{aligned}
MIC(p) &= \frac{1}{|V| \times \min(ETT)} \sum_{e \in p} IRU_e + \sum_{e \in p} CSC_{e,p} \\
IRU_e &= ETT_e \times N_e \\
CSC_{v,p} &= \begin{cases} w_1 & \text{if } CH(prev(v,p)) \neq CH(v) \\ w_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Considering again Figure 2.2, let the W values be scaled ETT values for each edge. Now, let $w_1 = 5$ and $w_2 = 13$. Given this, the shortest path from node 1 to node 5 is: $1 \xrightarrow{1} 3 \xrightarrow{1} 5$. This path has a cost of 6 for the weights and a cost of 13 from the channel-switching cost. Thus, the total cost is 19. However, if we allow w_1 to be 1 the shortest path becomes: $1 \xrightarrow{6} 2 \xrightarrow{1} 3 \xrightarrow{11} 4 \xrightarrow{6} 5$ This path has a total weight of 18, where 15 is from the link weights and 3 is from channel switching.

Yang, et al. give an algorithm that computes shortest paths using this metric by augmenting the graph such that each vertex in the original graph is replaced with a number of vertices equal to twice the number of edges at that vertex.

2.2.4 Routing Metrics and Formal Language Shortest Paths

As has been stated previously in this dissertation, computing shortest paths on labeled graphs is polynomial when the input formal language is context-free and thus regular as well. However, (Barrett, Bisset, Jacob, Konejevod, and Marathe 2002) shows that for even very simple graph classes, the problem of finding cycle-free shortest paths (formally known as simple paths) is actually NP-Complete.

Once again, please consider the example in Figure 2.2. If we consider the channels to be symbols from some formal language (since there are a finite number of them) then we can label each edge in the graph with the symbol corresponding to that link. Now, in the case where we want to find a shortest path where no channel can be repeated consecutively on a path, this is equivalent to setting $w_2 = \infty$. Also, note that we can express the same problem as a labeled shortest path problem by expressing the constraint that there can be no two consecutive edges with the same label as a regular expression and then using the RE-Shortest Path algorithm sketched in Section 1.2.1.

Given this new constraint the shortest path is forced to take the expensive edge between nodes 3 and 4. This path is $1 \stackrel{1}{\rightarrow} 3 \stackrel{11}{\rightarrow} 4 \stackrel{6}{\rightarrow} 5$ which has a cost of 13. This shows an equivalence between the problems when $w_2 = \infty$. Now, we present an algorithm which will be equivalent in the case when w_2 is some real number.

Recall that the RE-Shortest Path algorithm works by computing the cross product of the input graph and the finite state machine representing the regular expression. This cross product means that any path in the new graph is validly labeled, that is in this case that no symbol occurs twice consecutively. In order to make this roughly equivalent to the general MIC algorithm, we simply add in the edges that were omitted during the cross product—those edges corresponding to an edge in the input graph but not to a valid transition in the finite state machine—and we give this edge weight w_2. The resulting graph is similar to the 'virtual nodes' provided in (Yang, Wang, and Kravets 2005).

Because of the relationship between MIC and the RE-Shortest Path problem, we know

that we can compute MIC in polynomial time, even for simple paths, on acyclic graphs (Mendelzon and Wood 1995). However, when the constraint for the graph to be acyclic, the problem becomes NP-Complete.

2.2.5 Most Used Label Shortest Paths

Recalling the $WCETT$ metric, the first component is simple to compute since it is simply a weighted shortest path problem. The second component, however, is more difficult since it attempts to minimize the most used channel. The first component does not fundamentally alter the complexity of computing a shortest path over $WCETT$; so we consider the following labeled shortest path problem corresponding to only the second term in the $WCETT$ metric (this proof is taken almost directly from our paper (Ward and Wiegand 2010)):

Definition 2.3. *The most used labeled shortest paths (MUL) decision problem: Input: Directed Multigraph $G = (V, E)$, a set of labels Σ, weight function $\Phi : E \rightarrow \mathbb{R}^*$, labeling $L : E \rightarrow \Sigma$, source $s \in V$, destination $t \in V$, maximum weight W.*
Output: True if $\exists$ path p such that $mul(p) \leq W$ where

$$mul(p) = \max_{j \in C}\{X_j(p)\}$$

$$X_j(p) = \sum_{e \in p, label(e)=j} \Phi(e)$$

False, otherwise.

Theorem 2.1. *The MUL problem is NP-Complete.*

Proof. $\Leftarrow$: First note that one can easily verify in polynomial time whether for a given path p it is the case that $mul(p) \leq W$. Thus the problem is in NP.

$\Rightarrow$: To show that the MUL problem is NP-Hard we reduce from K-Bin Packing (Garey and Johnson 1990)Suppose we have an instance of the K-Bin packing problem in which we have k unit bins $\{B_1, \cdots, B_k\}$ and n items $\{1, \cdots, n\}$ with weights $\{I_1, \cdots, I_n\}$. We

will construct the input to a MUL problem. Let $\Sigma = \{1, \cdots, k\}$, and create the graph $G = (V, E)$. Here $V = \{v_1, \cdots, v_{n+1}\}$. For each vertex $v_i (1 \leq i \leq n)$ we create edges $e_{i,j} = (v_i, v_{i+1}) (1 \leq j \leq k)$ in E. Then let $L(v_{i,j}) = j$ and $\Phi(v_{i,j}) = I_i$. Finally, set $s = v_1$, $t = v_{n+1}$, and $W = 1$.

We run MUL on $(G, \Sigma, \Phi, L, s, t, W)$. If MUL returns true, then the shortest path through G represents a bin-packing, as each element has a label representing the bin it is in, and $mul(p)$ is the total weight of all elements put in the most filled bin. □

Corollary 2.1. *The MUL Problem is Strong NP-Complete over general undirected graphs, directed acyclic graphs, and series parallel graphs. The MUL Problem is Strong NP-Complete for any fixed number of labels $k \geq 2$.*

Proof. Note that if any shortest *mul*-path contains a loop, we can remove the loop and it remains a shortest *mul*-path. Thus, the reduction in Theorem 2.1 holds if the edges are undirected. Further, the graph given in Theorem 2.1 is both a directed acyclic graph and a series parallel graph, and thus the problem is NP-Hard over these sets of input graphs as well. K-Bin packing is Strong NP-Hard for $k \geq 2$, and so MUL is Strong NP-Hard for any number of labels ≥ 2 over all the aforementioned graph classes. □

2.2.6 Final Remarks Regarding the Complexity Paper

This section has presented complexity results on the $WCETT$ metric, showing a reduction from K-Bin Packing meaning that computing the metric accurately is NP-Complete on general graphs. Included in the paper are proofs of the complexity of the MIC metric and a discussion of how hard approximating the metric is. This metric is also NP-Complete, but further it belongs to a class of problems—NPO PB-Complete—which cannot be approximated within $n^{1-\epsilon}$ for any ϵ. Thus, not only is MIC NP-Complete, but it is difficult to approximate well.

2.3 Conclusion

This chapter presented two important results in the realm of labeled path problems: a distributed algorithm for CFL-APSP, and complexity results on wireless routing metrics. The former result was presented at the 37^{th} International Conference on Parallel Processing (ICPP) and the latter was published in the Journal of Computer Networks.

These results build upon those presented in Chapter 1 to lay a solid foundation in the emerging area of Labeled Graph Theory. The remainder of this dissertation continues to build upon this foundation by examining the relationship between the CFL-APSP algorithms (Chapter 3), giving lower bounds on the computation time and space required for the CLH-APSP (Chapter 4), and finally by providing an algorithm for computing K_{Linear}-Hierarchy APSP shortest paths (Chapter 5).

CHAPTER 3 – EMPIRICAL ANALYSIS OF CFL-APSP ALGORITHMS

3.1 Introduction

This chapter presents an empirical analysis of the `BJM` and `FastBJM` algorithms for solving the labeled all-pairs shortest-path problem. The case is made that even though the asymptotic bounds on running time of the algorithms are substantially different, that the choice of which algorithm to use must be made given larger considerations, including what the sizes of the graph and input grammar are.

Recall that the `BJM` algorithm has a $O(|V|^5|N|^2|P|)$ running time and that the `FastBJM` algorithm has a $O(|V|^3|N||P|)$ running time where $|V|$ is the number of vertices in the input graph and $|N|$ and $|P|$ are respectively the number of nonterminals and productions in the input formal language. This $|V|^2|N|$ difference would seem to make clear the case that one should always choose the `FastBJM` algorithm to solve the LAPSP problem; however, as this chapter will show, this is not always the case.

The first thing that must be considered is that `FastBJM` will not operate on graphs that have negative edge weights. `BT-FastBJM` (see Section 3.2) is a modification of `FastBJM` to allow it to compute shortest paths on graphs with negative edge weights so long as there are no negative cycles—whether validly labeled or not. The `BJM` algorithm will work on graphs with negative edge weights so long as there are no validly labeled negative cycles. It would seem that this generality comes at a great cost.

Performing a detailed amortized analysis of these algorithms proves quite complex because of the number of variables that would be involved in such an analysis. For the graph, these variables would be the number of vertices, density of edges, edge weights, etc. But

we also must take into consideration the CFG which is also part of the input. This would yield yet more variables including the number of non-terminals, number of productions, and the number of terminals. Further, the likelihood that some string is in the language greatly affects the analysis of the algorithm since this would directly affect the number of valid labeled paths of that same length that may exist in the graph. Therefore, in order to further investigate these algorithms we performed an empirical analyses over modified Erdös-Rènyi random graphs along with several different context free grammars.

The `BJM` algorithm which was introduced in Chapter 2 is actually $\Theta(|V|^5|N|^2|P|)$ as presented. However, a simple observation can improve the running time for certain graphs. In each iteration of the outer loop, at least one element of the D table must be updated until all shortest paths are found. This is clearly the case since if the D table ever does not change in a single iteration then all subsequent iterations will also fail to update values in the D table since the new values would depend on the state of the table. Therefore, we can terminate the `BJM` algorithm early by noticing when there are no further updates to the table.

3.2 BT-FastBJM

The `BT-FastBJM` algorithm was created by Bradford and Thomas (Bradford and Thomas 2009) in order to address the problem that Barrett's FastBJM does not allow graphs with negative edge weights. This algorithm was constructed exactly as Johnson's algorithm (Cormen, Leiserson, Rivest, and Stein 2001; Johnson 1977a), using FastBJM as Johnson's algorithm uses Dijkstra's algorithm. The algorithm works by using the Bellman-Ford shortest path algorithm (Bellman 1958) to calculate the shortest paths through the graph without respect to the context-free grammar. The algorithm then uses these shortest paths to modify the weights of the original graph before calling `FastBJM`. The `BT-FastBJM` algorithm is given in Algorithm 3.2. Although seemingly complex, the extra complexity adds little to the overall running time since it is merely an up-front cost.

Algorithm 3.1 BT-FastBJM for finding labeled shortest paths in graphs with negative edge weights

Compute G'
  $V[G'] \leftarrow V[G] \cup \{s\}$ and $E[G'] \leftarrow E[G] \cup \{\ (s, v, \epsilon, 0) : v \in V[G]\ \}$
for all $(u, v, t) \in E[G]$ **do**
  $w'(u, v) \leftarrow \min_{t \in \Sigma}\{\ w(u, v, t)\ \}$
end for
{COMMENT:Replacing the weight function w with w' for Bellman-Ford}
if Bellman-Ford$(G'(w'), s)$ = FALSE **then**
  print "G's underlying unlabeled subgraph contains a negative weight cycle"
  Exit;
end if
{Bellman-Ford gives $\delta_{BF}(s, v)$ for all $v \in V[G']$ *ignoring* edge labels in G'}
for all $v \in V[G']$ **do**
  $h(v) \leftarrow \delta_{BF}(s, v)$
end for
for all $(u, v, t) \in E[G]$ **do**
  $\widehat{w}(u, v, t) \leftarrow w(u, v, t) + h(u) - h(v)$
end for
$\widehat{\delta} \leftarrow$ Fast_BJM$(G(\widehat{w}))$ {COMMENT:Labeled directed graph G with the weight function $\widehat{w}$ replacing w}
for all $(u, v, A) \in V[G] \times V[G] \times N$ **do**
  $D[u][v][A] \leftarrow \widehat{\delta}(u, v, A) + h(v) - h(u)$
end for
Return D

As was stated in the introduction, this algorithm will work on graphs with negative edge weights, so long as there are no negatively weighted cycles. This differs from the `BJM` algorithm in that the `BJM` algorithm will work on graphs with negatively weighted cycles, so long as they are not validly labeled. This distinction is important as it means there are potentially many more graphs which could be analyzed using the slower, `BJM` algorithm.

3.3 Experiment Structure

One of the main goals of this experiment is to be completely repeatable by another researcher. Because of this, we only collect data about the graphs (longest shortest path, average path length, etc.) and also hardware agnostic statistics about the algorithms (how many comparisons were evaluated for each algorithm). To facilitate the former, and to en-

sure repeatability, the experiment uses Knuth's uniform random number generator which is provided with the Stanford Graph Base (Knuth 1993). By using this random number generator and providing the seeds as part of the artifact of the experiment, we guarantee the repeatability of the experiment despite what compiler or libraries another researcher may have.

As a note, it would be interesting to further study these algorithms in a hardware specific manner to see how each performs with respect to such events as cache-misses, paging, or swapping. Because of the requirement to keep this particular experiment completely repeatable, this will be left for future research.

3.3.1 Random Graphs

A classical definition of a random graph is owed to Erdös and Rènyi (Erdös and Renyì 1960). As expounded upon by Newman (Newman 2002), there are numerous problems with Erdös-Rènyi random graphs as related to modeling graphs in applied settings. This class of random graphs is, however, very well studied and understood, with well-known results on many parameters ranging from connectedness to graph diameter. While it would be desirable in future work to use a graph model such as the one described in the work by Newman, we use a slightly modified version of the Erdös-Rènyi model for both simplicity of simulation and ease of interpretation.

To empirically analyze these algorithms, as well as to examine probabilities of various events, we trivially extend the concept of a typical Erdös-Rènyi random graph. An Erdös-Rènyi random graph $G(n, p) = (V, E)$, with $|V| = n$ and for each edge $(u, v) \in V \times V$ there is a probability p that $(u, v) \in E$. For the purpose of this research, we simply take the set of all potential edges to be from $V \times V \times \Sigma$ rather than $V \times V$. That is, for $(u, v, t) \in V \times V \times \Sigma$ there is a probability p that $(u, v, t) \in E$.

3.3.2 MapReduce

MapReduce (Dean and Ghemawat 2008), created at and used by Google, is a distributed programming paradigm that removes many of the normal concerns associated with writing distributed programs such as work distribution, communication, node failures, etc. The programmer can concern himself with only describing two functions: `map` and `reduce`.

`map` and `reduce` are similar to the more familiar `map` and `fold` from functional programming. In fact, `fold` was known as `reduce` in early functional languages like APL. `map` takes as input a key/value pair and yields some intermediate key/value pair as output. These intermediate pairs are then aggregated amongst all `map` instances. The intermediate keys can be thought of as keys in a dictionary. So, a list is generated for each key and all values that share that key are added to the list. Next each of these lists which share a common key are passed to a `reduce` instance. `reduce` takes this list of intermediate values which share a common key and computes some new, desired value. It then emits a new key/value pair as output. The result of a MapReduce computation is a list which is the aggregation of all of the final key/value pairs resulting from each `reduce` operation.

Algorithm 3.2 map(String key, String value)

```
1: for all w ∈ value do
2:     EmitIntermediate(w, "1")
3: end for
```

Algorithm 3.3 reduce(String key, Iterator values)

```
1: int result = 0;
2: for all v ∈ values do
3:     result += ParseInt(v);
4: end for
5: Emit(AsString(result));
```

Consider the example which is taken from the original MapReduce paper (Dean and Ghemawat 2008), shown in Algorithm 3.3.2 and Algorithm 3.3.2. This is an example of how the frequency of words in a corpus of documents can be calculated. A `map` instance is created

for each document in the corpus, with *key* being the name of the document and *value* being the contents of that document. The mapper then emits the word as the key along with the string "1". Each occurrence of a particular word is then grouped together since the word is used as the key. Then, this list is passed to `reduce`. Reduce then adds one for each element in the list, thus counting the frequency of that particular word. Finally, it emits that count. This count is automatically matched with the key that was its source so we get a list of tuples where the first element in the tuple is a word and the second is the number of times that word occurs in the corpus of documents.

For the purpose of this experiment, `map` and `reduce` are used as a convenient means of achieving a distributed Monte Carlo simulation. The input to `map` will be the parameters for each experiment: the seed for the random number generator, the number of vertices, the edge likelihood, and the context-free grammar. Each `map` instance will create a new labeled-random graph using the seed and other provided parameters. Then it will run each algorithm, `BJM` and `FastBJM`, and collect the following relevant statistics per each:

- number of comparisons computed in the execution of the algorithm
- number of memory allocations used in the algorithm
- number of rounds required for the table to stabilize (`BJM` only).

Further, it will collect the following information about the random graph:

- seed used to generate the graph
- number of vertices in the graph
- edge likelihood
- average labeled path length
- length of the longest validly labeled shortest path in the graph
- is the graph L-strongly connected

- number of extant labeled shortest paths
- is there a negative cycle

Definition 3.1. *Let G be a labeled graph and L be a formal language, then G is L-strongly connected if $\forall u, v \in V$ there exists a validly labeled path from u to v.*

The intermediate key which is returned by `map` is the same as original key however with the *seed* field removed. The intermediate value is a structure containing all of above enumerated statistics and information. Because of the nature of grouping the intermediate values by the intermediate key, a single `reduce` instance will have a list containing the statistics on all graphs generated with the same parameters. This is equivalent to the data collected during each instance of a Monte Carlo simulation. The `reduce` function will then calculate such statistics as the longest validly labeled shortest path encountered, the average path length, and statistics about the number of comparisons, allocations, and assignments required for each algorithm. Thus, on aggregate, the output of the `reduce` operations will be the statistics given per each parameter triple.

The keys for our map reduce are from $Grammars \times Seeds \times Vertices \times EdgeProbabilities \times Weights$. There are 20 unique seeds per experiment, meaning that each experiment is run 20 times with the same generative parameters. $Vertices$ is the set $\{10, 12, 14, \ldots, 100\}$. $EdgeProbabilities$ is the set $\{0.05, 0.15, 0.25, \ldots, 0.95\}$. Finally, $Weights$ is the set of tuples $\{(0, 100), (-5, 100), (-10, 100), (-100, 100)\}$. Since there are 6 grammars (described in Section 3.3.3), this yields $220,800$ experiments. Also, several smaller experiments were ran for more focussed, granular data. These experiments were run at Google on their internal MapReduce system utilizing over 700 CPU hours of donated compute time.

To summarize,by using MapReduce, we can write two simple functions—`map` and `reduce`—and then run a massive number of Monte Carlo simulations in parallel.

3.3.3 Context-Free Grammars

For our experiment, we used six context free grammars. Table 3.1 and Table 3.2 presents these grammars, each in Chomsky Normal Form. The first grammar represents the context-free language which is equivalent to the regular language $(a)+$, that is one or more as. This grammar was chosen since it would be guaranteed that if there is a path between two vertices then there will be a valid string in the language created by concatenating the labels of that path. Thus, this grammar degenerates into the case of simply finding shortest paths. The second grammar defines the language of palindromes over three terminals. We chose this grammar as one in which we are less likely to generate valid strings than in the parenthesis language. The next four grammars represent the languages over $1, 2, 3$, and 4 types of parenthesis respectively.

Simple language			Palindrome language			1-Parenthesis language		
S	$\rightarrow$	AS	S	$\rightarrow$	AX	S	$\rightarrow$	VW
S	$\rightarrow$	a	S	$\rightarrow$	BY	S	$\rightarrow$	XW
A	$\rightarrow$	a	S	$\rightarrow$	CZ	S	$\rightarrow$	XY
			X	$\rightarrow$	SA	S	$\rightarrow$	VY
			Y	$\rightarrow$	SB	V	$\rightarrow$	XS
			Z	$\rightarrow$	SC	W	$\rightarrow$	YS
			A	$\rightarrow$	a	X	$\rightarrow$	(
			B	$\rightarrow$	b	Y	$\rightarrow$	)
			C	$\rightarrow$	c			
			S	$\rightarrow$	a			
			S	$\rightarrow$	b			
			S	$\rightarrow$	c			

Table 3.1. Context-Free Languages

2-Parenthesis language			3-Parenthesis language			4-Parenthesis language		
S	$\rightarrow$	AB	S	$\rightarrow$	AB	S	$\rightarrow$	AB
S	$\rightarrow$	CD	S	$\rightarrow$	CD	S	$\rightarrow$	CD
S	$\rightarrow$	ES	S	$\rightarrow$	ES	S	$\rightarrow$	ES
S	$\rightarrow$	FS	S	$\rightarrow$	FS	S	$\rightarrow$	FS
S	$\rightarrow$	AG	S	$\rightarrow$	AG	S	$\rightarrow$	AG
S	$\rightarrow$	CH	S	$\rightarrow$	CH	S	$\rightarrow$	CH
S	$\rightarrow$	IS	S	$\rightarrow$	IS	S	$\rightarrow$	IS
S	$\rightarrow$	JS	S	$\rightarrow$	JS	S	$\rightarrow$	JS
E	$\rightarrow$	AB	S	$\rightarrow$	MS	S	$\rightarrow$	MS
F	$\rightarrow$	CD	S	$\rightarrow$	KN	S	$\rightarrow$	KN
G	$\rightarrow$	SB	S	$\rightarrow$	OS	S	$\rightarrow$	OS
H	$\rightarrow$	SD	E	$\rightarrow$	AB	S	$\rightarrow$	PQ
I	$\rightarrow$	AG	F	$\rightarrow$	CD	S	$\rightarrow$	RS
J	$\rightarrow$	CH	G	$\rightarrow$	SB	S	$\rightarrow$	PU
A	$\rightarrow$	(	H	$\rightarrow$	SD	S	$\rightarrow$	TS
B	$\rightarrow$	)	I	$\rightarrow$	AG	E	$\rightarrow$	AB
C	$\rightarrow$	{	J	$\rightarrow$	CH	F	$\rightarrow$	CD
D	$\rightarrow$	}	M	$\rightarrow$	KL	G	$\rightarrow$	SB
			N	$\rightarrow$	SL	H	$\rightarrow$	SD
			O	$\rightarrow$	KN	I	$\rightarrow$	AG
			A	$\rightarrow$	(	J	$\rightarrow$	CH
			B	$\rightarrow$	)	M	$\rightarrow$	KL
			C	$\rightarrow$	{	N	$\rightarrow$	SL
			D	$\rightarrow$	}	O	$\rightarrow$	KN
			K	$\rightarrow$	[	R	$\rightarrow$	PQ
			L	$\rightarrow$	]	T	$\rightarrow$	PU
						U	$\rightarrow$	SQ
						A	$\rightarrow$	(
						B	$\rightarrow$	)
						C	$\rightarrow$	{
						D	$\rightarrow$	}
						K	$\rightarrow$	[
						L	$\rightarrow$	]
						P	$\rightarrow$	$<$
						Q	$\rightarrow$	$>$

Table 3.2. Context-Free Languages

3.3.4 Implementation

The `BJM` and `FastBJM` algorithms were each implemented in `C`. This included an implementation of a Fibonacci heap. While most of the implementation details will be omitted here, for brevity's sake, the methodology for counting comparisons need be discussed. Such a matter would seem trivial. In order to count a comparison, simply increment a counter on the line before the comparison. However, this is not necessarily a safe way to count since the compiler is free to optimize the comparisons away or reorder them. So, in order to count the comparisons we use a technique described in(Knuth 1993).

```
if((++counter), x > 0) {
  ...
}
```

This technique uses several features of the `C` language. First, in an expression with commas, every sub-expression is evaluated and the sub-expression after the last comma is returned as the value for the whole expression. Thus, in the case of our example, `x>0` will be the sub-expression that determines whether or not the `if` statement passes, just as was desired. Since the first subexpression is also evaluated, this causes a counter to be incremented. Because this all takes place within the body of the comparison, the optimizer cannot remove or restructure the comparison or its counter since there is a side-effect, the incrementing of the counter. This technique was used throughout the source code of the algorithms to count all comparisons evaluated in each algorithm.

3.4 Results

Recall that the `BJM` algorithm only performs $O(|V|^5|N|^2|P|)$ in the worst case. Whenever the D-table stabilizes, it means the lengths of all shortest paths have been found and the algorithm terminates. Consider Figure 3.1. This figure shows the percentage of validly labeled paths in a graph with 25 vertices. Note that the horizontal access is the ratio of the

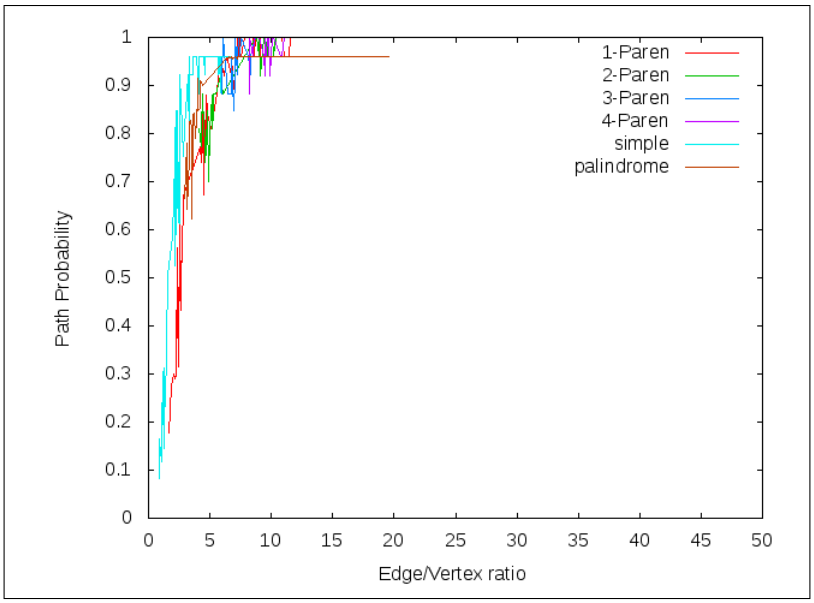

Figure 3.1. Path probability vs. edge/vertex ratio for graphs with 25 vertices

number of edges in the graph to the number of vertices (25 in this case). This ratio was chosen as opposed to using the edge-probabilities to the Erdös-Rènyi random graph model because we believe it to be a more stable metric. It is important to note that in situations where the edge to vertex ratio is even moderate, the number of extant paths is very high.

Figures 3.2 through 3.5 show the comparisons required for the `BJM` and the `FastBJM` algorithm on graphs using 25% edge probability. Each graph represents a parenthesis language with a different number of parenthesis. Note that the line representing `FastBJM` looks perfectly smooth. This is because `FastBJM` is in fact $\Theta(|V|^3|N||P|)$.

The absolute best case for the `BJM` algorithm is when there are many, very short, valid paths. Consider Figure 3.6. This figure shows that the `BJM` performs considerably better than the `FastBJM` algorithm when using the Simple language. The Simple language is equivalent to the regular language $a*$. Thus, all edges are labeled with a single terminal and any string of any length is considered valid. Therefore, this is equivalent to the standard all-pairs shortest path problem.

Figure 3.8 shows the percentage of graphs which can be handled by `BJM` and `BT-FastBJM` over a series of random graphs with varying edge probabilities, with 10 vertices, with edge

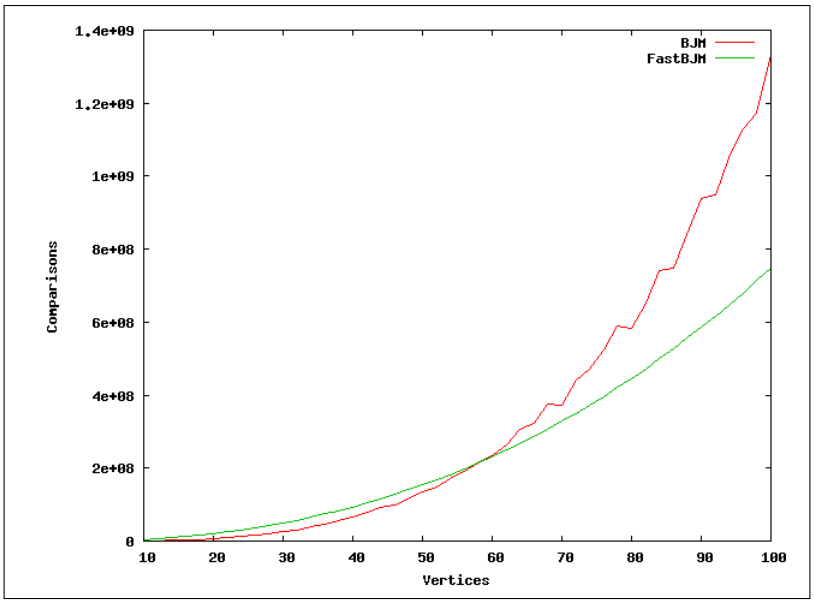

Figure 3.2. Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 1-parenthesis language

weights sampled uniform randomly from the range $[-5, 100]$, and with a single-parenthesis language. Figure 3.9 shows the same graph, but with a three-parenthesis language. Notice that as the probabilities, which are input to the Erdös-Rènyi random graph model, increase, the percentage of graphs which do not have negative cycles decreases.

Comparing these two graphs, we infer that the generality benefit of `BJM` over `BT-FastBJM` is strongly dependent on the complexity of the grammar which is to be matched. This is intuitively correct, as we would expect that for any given negative cycle in the graph, the more complex the grammar, the less likely it is to have a proper labeling and, thus, cause `BJM` to fail. The more complicated the grammar, the less likely that a shortest path would exist between any two vertices. So, whenever the edge/vertex ratio is high, this would seem to indicate (as is reinforced by Figure 3.1) the more likely that a valid path would exist. In the case where there are negative edge weights, this will also increase the likelihood of negatively weighted cycles.

It is also important to note that for graphs under 50 vertices, fewer comparisons are needed for the `BJM` algorithm as opposed to the `FastBJM` algorithm.

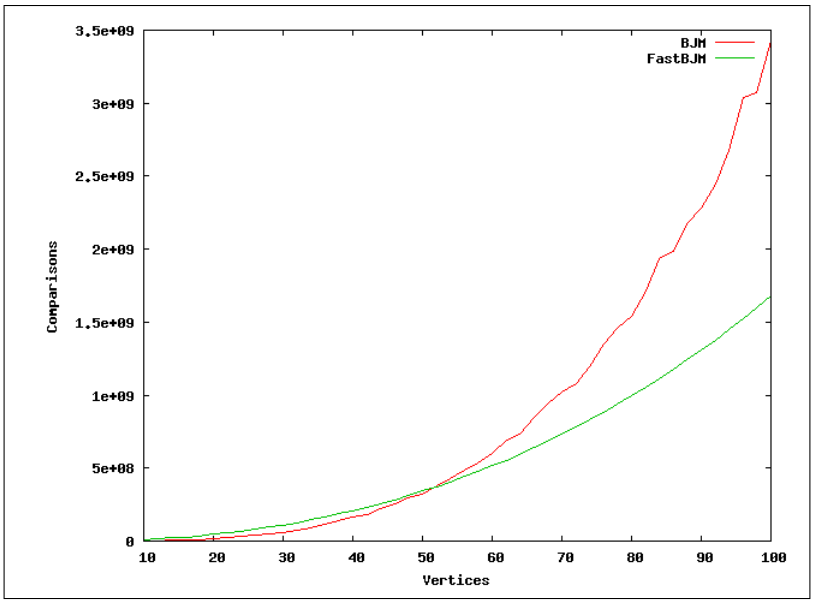

Figure 3.3. Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 2-parenthesis language

While the number of comparisons for each algorithm is high, it is important to note that comparisons are comparatively cheap operations. Reading and writing from RAM is a much slower process. We have conducted a few small experiments toward measuring the running times of the respective algorithms, which yield some surprising results. See Figure 3.10. `BJM` routinely performs faster than `FastBJM` on these graphs of up to 90 vertices on several different context-free grammars. This improvement is probably due to being able to place large portions of the D-table into either the `L1` or `L2` cache or the CPU. This result is very interesting and requires further investigation, as it would seem to imply that even though the number of comparisons is much higher, the `BJM` algorithm might be the better choice in practice, at least for small graphs.

3.5 Conclusion

This chapter has presented an empirical analysis of the `BJM`, `FastBJM`, and `BT-FastBJM` algorithms. The simulations demonstrate that although the worst-case of the `BJM` algorithm is extremely costly, the algorithm performs well in practice. Further, the algorithm should

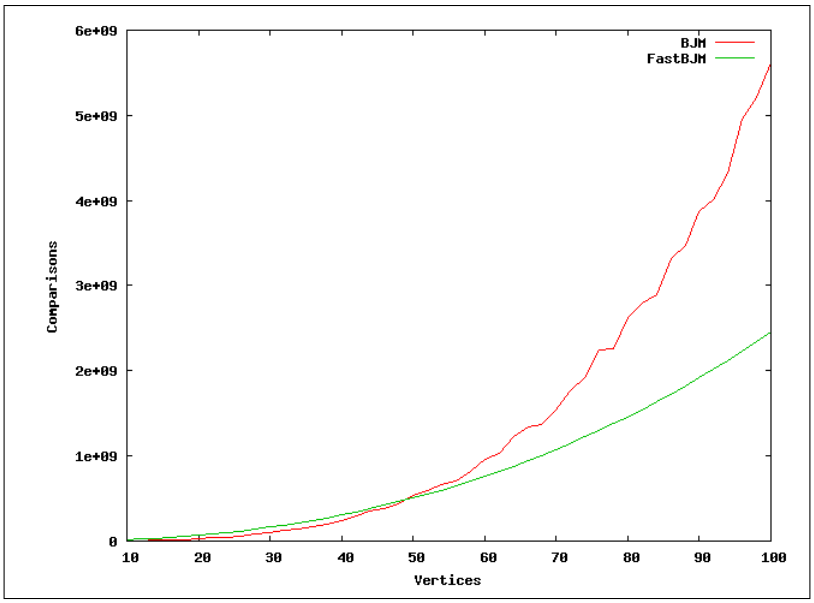

Figure 3.4. Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 3-parenthesis language

receive consideration in the face of graphs with negative edge-weights since it will successfully find shortest paths on a higher percentage of graphs than `BT-FastBJM`. Although the results from this chapter have addressed only the averages of number of comparisons, running time, etc., a future extension to this research is planned which will provide a much more thorough statistical analysis.

This work could not have been completed without the donation of computer time by Google, Inc. These resources proved invaluable.

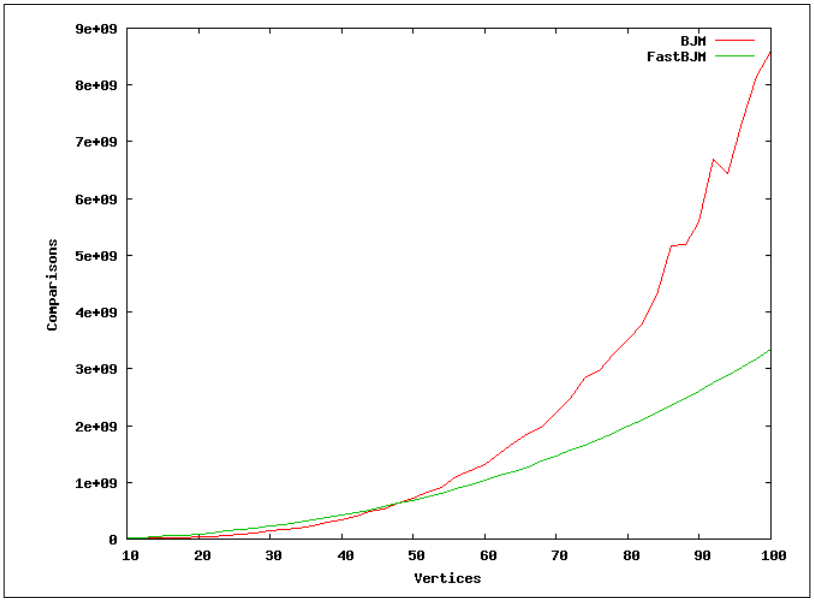

Figure 3.5. Number of comparisons for both BJM and FastBJM with edge probability 0.25 and 4-parenthesis language

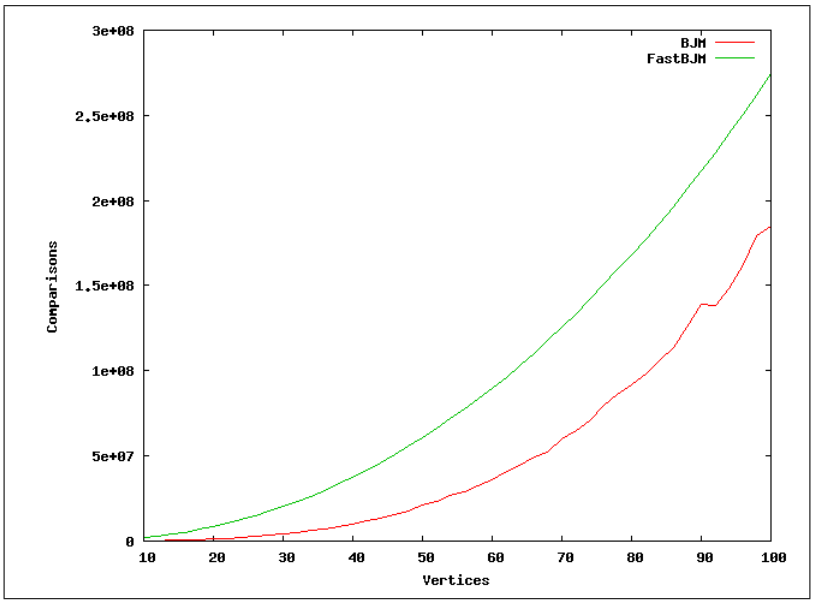

Figure 3.6. Number of comparisons for both BJM and FastBJM with edge probability 0.25 and simple language

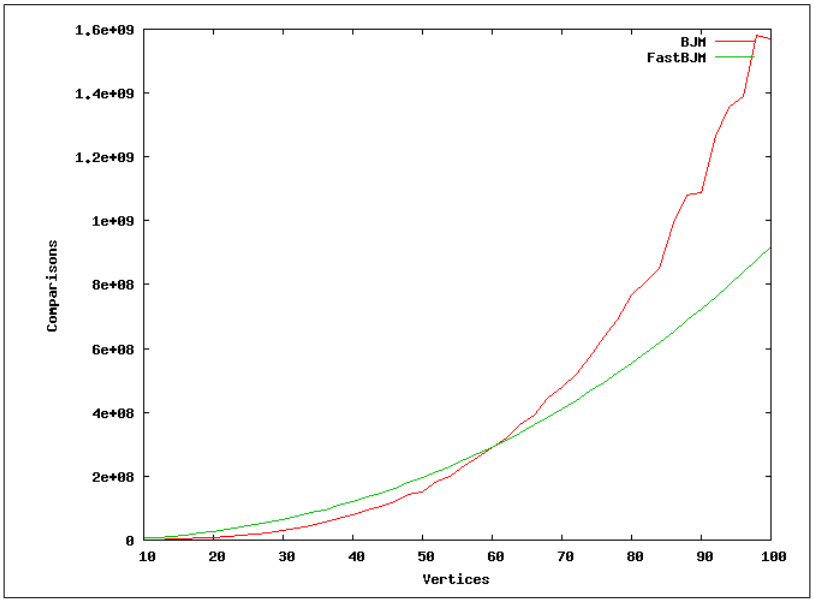

Figure 3.7. Number of comparisons for both BJM and FastBJM with edge probability 0.25 and palindrome language

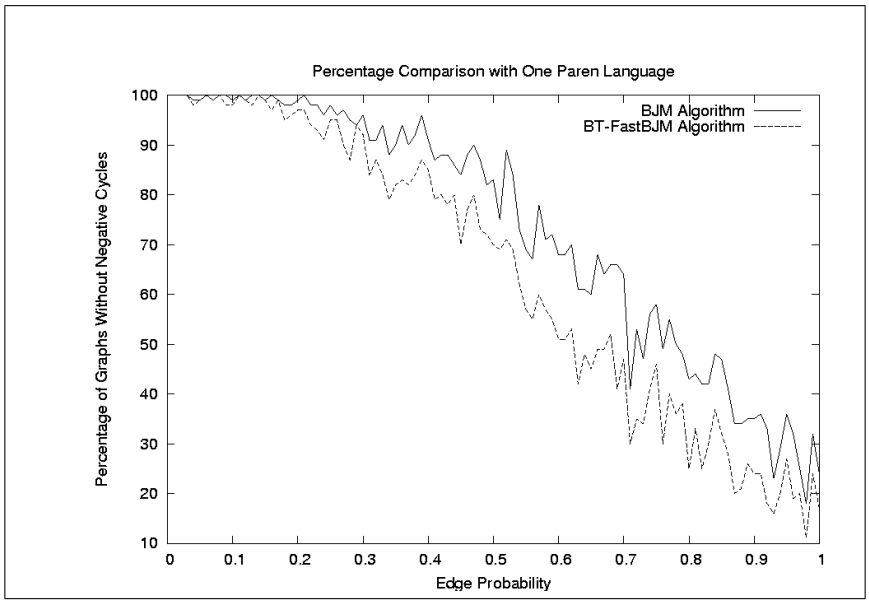

Figure 3.8. Ratios of graphs with no negative cycles.

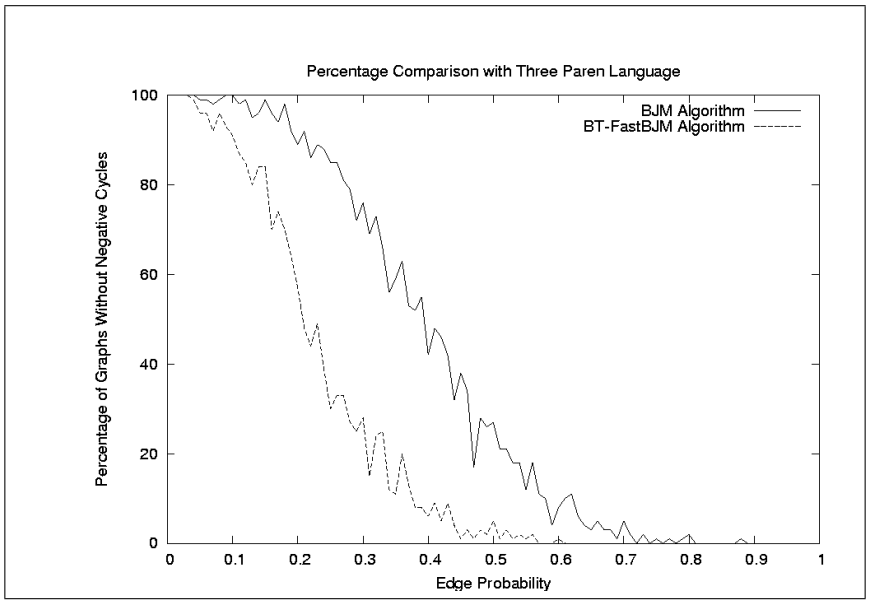

Figure 3.9. Ratios of graphs with no negative cycles.

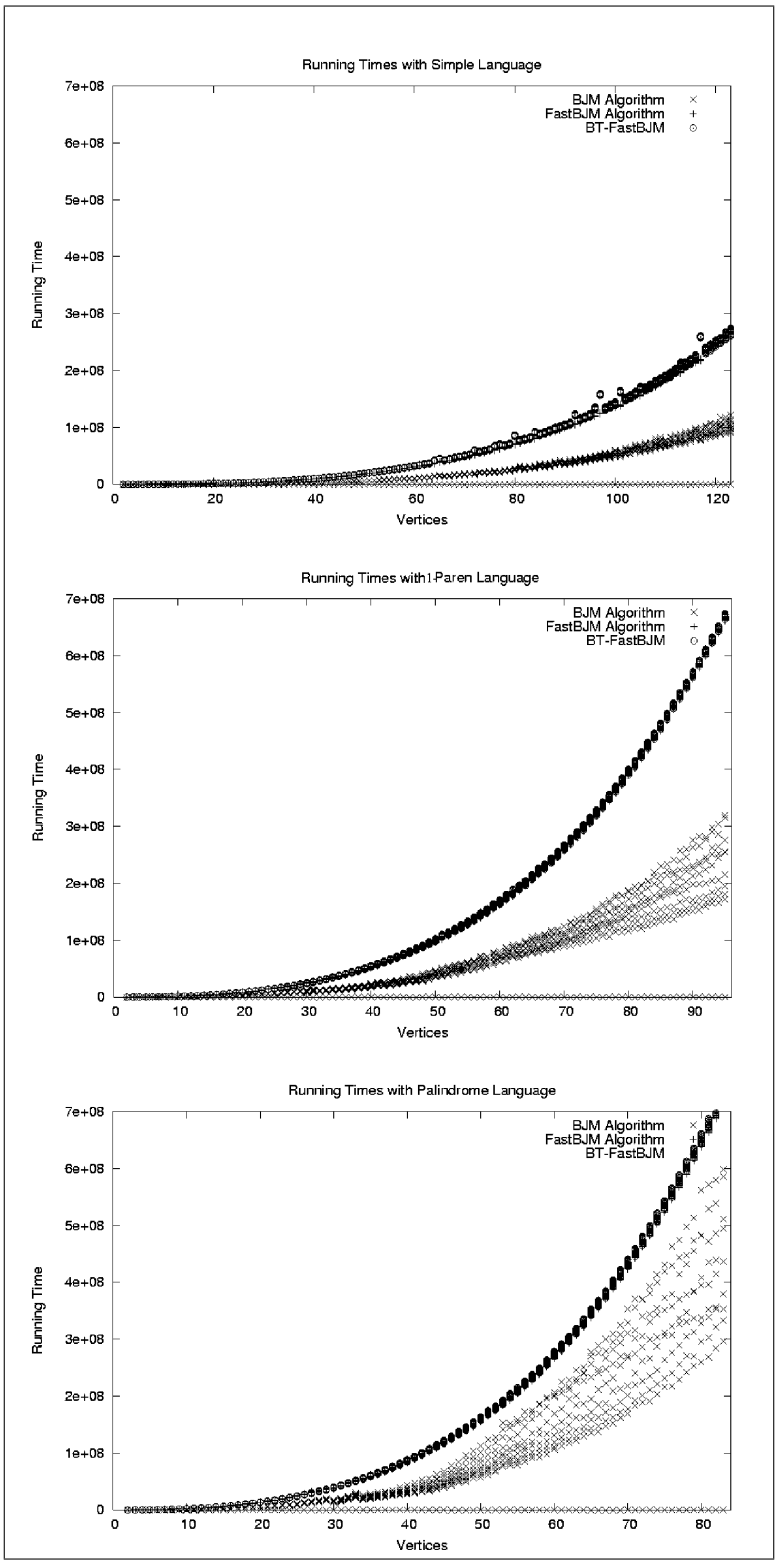

Figure 3.10. Running times of `BJM`, `FastBJM`, `BT-FastBJM` compared over different formal languages

CHAPTER 4 – LOWER BOUNDS ON CLH_k ROUTING TABLES

4.1 Introduction

This chapter is concerned with a special type of shortest path problems. In this context, a shortest path between two vertices u and v in some graph G is the path from u to v which has minimum total sum of edge weights. We assume that there are unit edge weights and thus a shortest path is that path which has the fewest edges. Further, the all-pairs shortest path problem (APSP) is that where we want to find the shortest paths between all possible pairs of vertices. Many algorithms are available to solve different instances of these problems (Floyd 1962) (Johnson 1977b). However, this chapter is concerned with labeled path problems. Recall that in this problem, we take as input the graph $G = (V, E)$ and also some formal language L. $E \subseteq V \times V \times \Sigma$ where Σ is the alphabet of L. In other words, we allow G to be a multigraph with labeled edges so long as if there are two edges e_1 and e_2 between vertices u and v, then the labels of these edges are necessarily distinct. Given such a graph G and language L, the LAPSP problem is that of finding a shortest path between all vertices u and v such that the concatenation of the edge-labels along the path forms a valid word in the language L.

In this chapter, we are concerned specifically with the languages that fall between context-free languages (CFL) and context-sensitive languages (CSL) in the Chomsky hierarchy. Weir(Weir 1992) described an infinite hierarchy of languages which fit between CFL and CSL, the Control Language Hierarchy (CLH).

The CLH is an inductively built hierarchy with context-free languages at its base. The basic notion of a CLH grammar is that it begins with a (necessarily) overly general context-

free language and then filters the derivations of the productions via a control language. Formally, a CLH grammar is a pair, (LDG, CL), of a labeled distinguished grammar and a control language.

A labeled distinguished grammar (LDG) is:

$$G = (N, T, I, S, P)$$

where N is a set of non-terminals, T a set of terminals, I a set of production-labels, $S \in N$ the start symbol, and P the set of labeled distinguished productions.

A labeled distinguished production is:

$$l : A \rightarrow X_1 ... \check{X}_i ... X_n$$

where l is the label for this production, $A \in N$, $X_1, ..., X_{i-1}, X_{i+1} ... X_n \in N \cup T$, and $\check{X}_i \in N \cup T \cup \{\epsilon\}$. $\check{X}_i$ is the distinguished element.

Following Weir, for all $\alpha_1, \alpha_2 \in \{(N \cup \Sigma) \times I^*\}^*$ and $w \in I^*$ and $A \in N$, then

$$\alpha_1 < A, w > \alpha_2 \quad \Rightarrow_C \quad \alpha_1 < X_1, \epsilon > \cdots < \check{X}_i, wl > \cdots < X_n, \epsilon > \alpha_2$$

where $\check{X}_i$ is distinguished. Since only the distinguished non-terminal contributes to string for this production, the parse tree is uniquely specified. Let $L(G, C)$ be the language generated by controlling the grammar G with control language C.

Given a labeled distinguished grammar, in order to build the CLH we also require a control language. A control language is the set of valid words which are formed by concatenation of the production labels during the parse. The control language itself can be generated via some grammar, thus giving rise to the following definition of the Control Language Hierarchy.

Let $\text{CLH}_1 = CFL$, that is the base case of the recursion is the set of context-free languages. Then, let CLH_k be the class of all languages which can be generated by a

LDG controlled by grammars which are in the set CLH_{k-1}. Weir shows that the language $L_k = \{a_1^n a_2^n ... a_{2^k}^n | a_i \in \Sigma, n \geq 1\}$ is in CLH_k but not CLH_{k-1}. We use this fact to prove a bound on the size of a successor matrix for the CLH LAPSP.

4.1.1 Example CLH_3 Grammar

The following example is taken from (Weir 1992). It defines a 3-level grammar (G_1, G_2, G_3) which matches exactly the language $\{a_1^n a_2^n \cdots a_8^n | n \geq 0\}$. Let $G_1 = (\{S_1\}, \{a_1, \ldots, a_8\}, \{l_1, \ldots, l_5\}, S_1, P_1)$, $G_2 = (\{S_2, T\}, \{l_1, \ldots, l_5\}, \{l_6, \ldots, l_9\}, S_2, P_2)$, and $G_3 = (\{S_2, R\}, \{l_6, \ldots, l_9\}, S_3, P_3)$. Define the productions from the grammars as follow:

$$P_1 = \left\{ \begin{array}{lcl} l_1 : S_1 & \rightarrow & a_1 \check{S_1} a_8 \\ l_2 : S_1 & \rightarrow & a_2 \check{S_1} a_7 \\ l_3 : S_1 & \rightarrow & a_3 \check{S_1} a_6 \\ l_4 : S_1 & \rightarrow & a_4 \check{S_1} a_5 \\ l_5 : S_1 & \rightarrow & \check{\epsilon} \end{array} \right\}$$

$$P_2 = \left\{ \begin{array}{lcl} l_6 : S_2 & \rightarrow & \check{T} l_5 \\ l_7 : T & \rightarrow & l_1 \check{T} l_4 \\ l_8 : T & \rightarrow & l_2 \check{T} l_3 \\ l_9 : T & \rightarrow & \check{\epsilon} \end{array} \right\}$$

$$P_3 = \left\{ \begin{array}{lcl} S_2 & \rightarrow & l_6 R l_9 \\ R & \rightarrow & l_7 R l_8 \\ R & \rightarrow & \epsilon \end{array} \right\}$$

G_3 controls the derivations of G_2 by constraining which productions can occur in what order. So, in this case, $G_3 = \{l_6 l_7^n l_8^n l_9 | n \geq 0\}$, meaning that the production labeled l_6 must

be matched, followed by n matchings of the production labeled l_7 then n matchings of the production labeled by l_8. Finally l_9 must be matched. Continuing, notice that this means that $L(G_2, G_3) = \{l_1^n l_2^n l_3^n l_4^n l_5 | n \geq 0\}$, where $L(G, C)$ is the language generated by LDG G controlled via controlling grammar C. Finally, using this to control the base grammar, it can be seen that $L(G_1, L(G_2, G_3)) = \{a_1^n a_2^n \cdots a_8^n | n \geq 0\}$.

4.2 Successor Matrices

A successor matrix (Seidel 1995), S, can be thought of as a routing table. Element $S_{i,j} = k$ means that k is the next vertex in the shortest path from i to j. This is similar to the predecessor matrix that is used in the Floyd-Warshall algorithm (Floyd 1962). In the undirected, unlabeled version of the APSP problem, the sucessor matrix need only be $|V|^2$ in size. This is due to the optimal substructure of the problem — that in an undirected graph, if p is the shortest path from i to j then if $i \rightarrow k$ is the first edge on the path then the shortest path $k \rightsquigarrow j$ is completely contained within the path $i \rightsquigarrow j$. Given this, let us now explore the successor matrices for CLH_k labeled graphs.

4.2.1 Longest Shortest Paths

In an unweighted APSP problem, the longest shortest path between any pair of points is $O(|V|)$. This is not the case, however, for the CLH_k. Recall that $\text{L} = \{a_1^m a_2^m ... a_{2^k}^m | m \geq 1\}$ $\in \text{CLH}_k$. So, for a given instance of the LAPSP(G,L) problem, how long can the longest shortest path be? Consider Figure 4.1. This graph was constructed in order to demonstrate that very long shortest paths can exist. Each *ear* is labeled with a different symbol, and is a path of a different length. Specifically, the ear labeled with symbol a_1 is n long, the ear labeled with a_2 is $n-1$ long, etc. In general, the ear labeled with the symbol a_j is $n-j+1$ long. Let us now investigate how many vertices are in this graph.

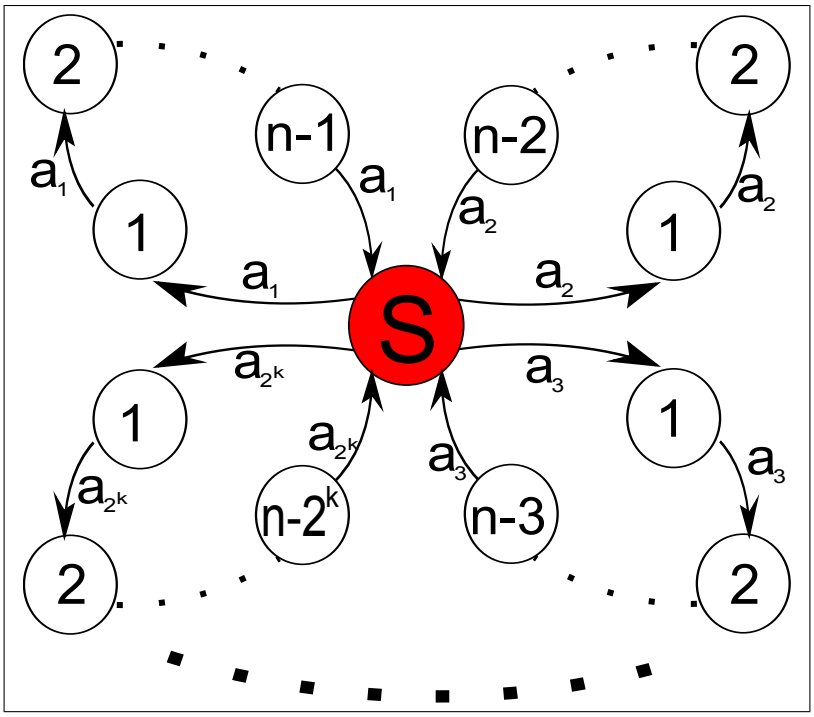

Figure 4.1. *Widget* — a graph G with a long shortest path.

$$
\begin{aligned}
|V| &= 1 + \sum_{i=1}^{2^k} (n - i) \\
&= 1 + \sum_{i=1}^{2^k} n - \sum_{i=1}^{2^k} i \\
&= 1 + 2^k n - \frac{2^k(2^k + 1)}{2} \\
&= 1 + 2^k n - 2^{2k-1} - 2^{k-1} \\
&\in \Theta(n)
\end{aligned}
$$

Now, we consider finding the shortest path from S to S given that the input language was L. Since L $= \{a_1^m a_2^m ... a_{2^k}^m | m \geq 1\}$, it means that $\forall i \in [1, 2^k]$, each a_i must appear the

same number of times—the least common multiple of the length of the ears.

$$\begin{aligned} lcm(n, n-1, ..., n-2^k) &\leq \prod_{i=1}^{2^k}(n-i+1) \\ &\in O(n^{2^k}) \\ lcm(n, n-1, ..., n-2^k) &\geq \prod_{i=1}^{2^k} \frac{n-i+1}{\prod_{j=2}^{2^k} j^{2^k}} \\ &\in \Omega(n^{2^k}) \\ \text{Therefore, } lcm(n, n-1, ..., n-2^k) &\in \Theta(n^{2^k}) \end{aligned}$$

So, let $m = lcm(n, n-1, ..., n-2^k)$. Thus, the shortest valid string in L_k is $2^k m$ long. Since each ear contributes one type of symbol and each edge contributes exactly one symbol, a valid path from $S \rightsquigarrow S$ in this graph must be of length $2^k m$. Since $m \in \Theta(n^{2^k})$, this path has length $\Omega(n^{2^k})$.

Theorem 1. *Given a graph G with labels from Σ which is constrained by $L_k \in CLH_k$, the longest shortest path between any two vertices in G is $\Omega(|V|^{2^k})$.*

4.2.2 Lower Bound on the Size of the Successor Matrix

Consider an unlabeled, unweighted graph. Intuition might lead one to believe that it would require $|V|^3$ space to store all possible shortest paths between vertices since there are $|V|^2$ paths and the maximum path length is $|V|$. However, this is not the case. Since there cannot be any cycles in such a graph, the principle of optimal substructure holds and there is much redundant information. Thus, the routing table representing all shortest paths can be represented in $|V|^2$ space.

The previous sections showed that the length of the longest shortest CLH_k labeled path is at least $\Omega(n^{2^k})$. In the ordinary meaning of a successor matrix, we cannot represent a cycle. However, a CLH_k labeled path can have numerous cycles. So, let us simply consider

some successor matrix that does act as a correct routing table for a CLH_k labeled path. In other words, it will have to allow for the fact that there are cycles, and will do so by having redundancies. That is, there will be more than one row corresponding to each vertex. The purpose of this section is to investigate how large such a successor matrix would need to be.

Let the successor matrix, M be $m \times |V|, m \geq |V|$. In a normal successor matrix, the value in the matrix at position row i and column j is the next vertex on the shortest path from i to j. In this modified version of the successor matrix, let $M_{i,j} = k$ be the row number corresponding with the next step in the shortest path from $i' \rightsquigarrow j$ where $i' = i \pmod{|V|}$.

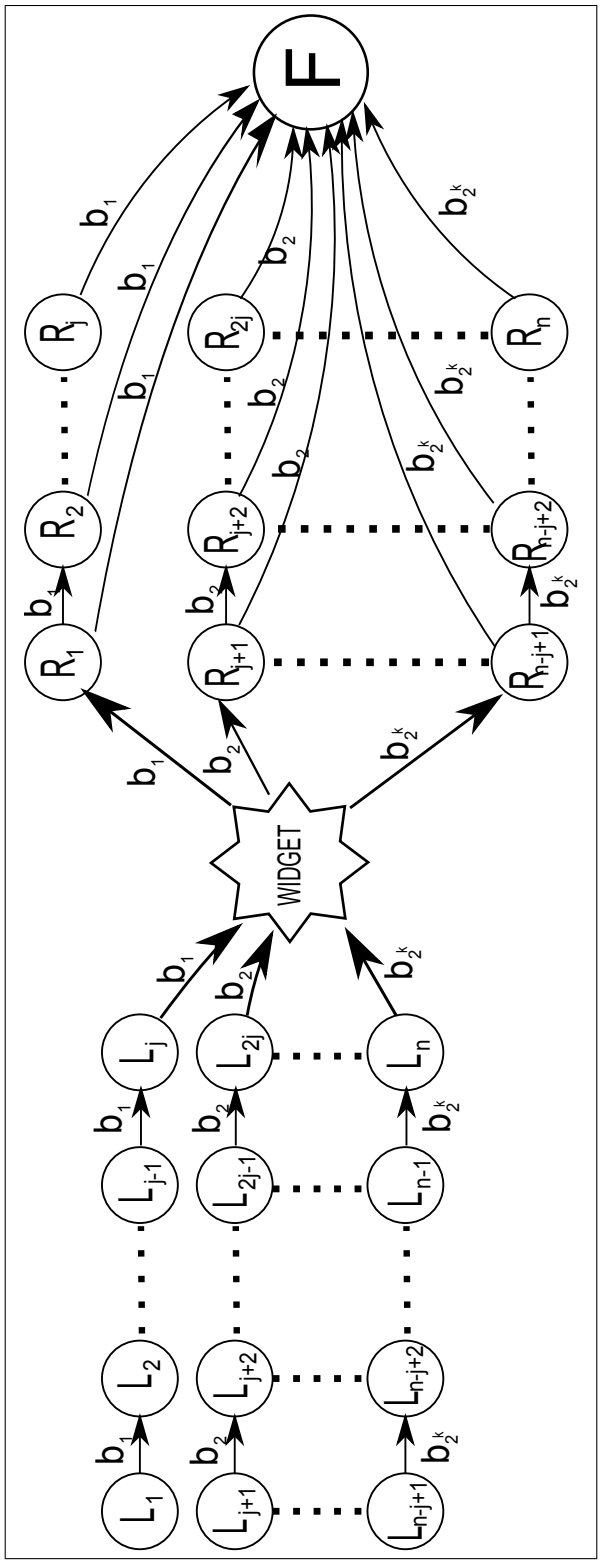

Figure 4.2. Special graph with a large successor matrix but a lot of redundancy

Consider such a matrix M for the graph G in Fig. 4.2. Let the formal language which

constrains the shortest paths in this graph be $\{b_k^j L b_k^j | \forall k \in [1, 2^k], L \text{ as from Figure 4.1}, j \geq 1\}$. Now, assume that the column of M which corresponds to the vertex labeled "F" is lost. This means that all shortest paths that terminate at "F" are lost (based on the definition of the routing table). It should be clear that to rebuild any one shortest path it will take time and space $O(length(path))$. Now we show that, unlike the case of the unlabeled, unweighted graphs, we require this time and space to rebuild each shortest path which was lost.

First, consider the two shortest paths: $p_1 = L_1 \rightsquigarrow F$ and $p_2 = L_2 \rightsquigarrow F$. Note that these two paths share many common features. In fact, p_2 is completely contained within p_1 but for the fact that p_1 has ends with $R_1 \rightarrow F$ and p_2 ends with $R_1 \rightarrow R_2 \rightarrow F$. In the previous subsection it was shown that the length of the shortest path through "Widget" was $\Omega(n^{2^k})$. So, each of these paths p_1 and p_2 is of length $\Omega(n^{2^k})$. Recall that with this model of the successor matrix these paths are required to be stored independently since they are disjoint.

Now, note that like p_1 and p_2 above,$\forall p_i = L_i \rightsquigarrow F, i \in [1, n]$, the paths are disjoint in at least the last edge. Thus each path must be stored independently. There are n such paths and each is of length $\Omega(n^{2^k})$ so the successor matrix for this graph must be $\Omega(n^{2^k+1})$ in size.

4.3 Compression of Routing Tables for CLH_k Labeled Paths

The previous section showed that the successor matrix for a CLH_k LAPSP must be very large, $\Omega(n^{2^k+1})$. However, simply looking at how the example graph was constructed demonstrates that even though the successor matrix is large, we can represent the all possible paths from a vertex on the left to the vertex labeled "F" in a much shorter space by simply routing the path to the *widget* and then from the widget to the F vertex. We simply need to recall the path through the widget.

4.3.1 Longest Common Sub-Path

The well known problem of finding the longest common subsequence is NP-Complete (Wagner and Fischer 1974). Luckily, however, the longest common sub-path is a degenerate case,

and can be solved in polynomial time as will be shown because of its relation to the longest common substring. The longest common sub-path of a set of paths is the longest path which is fully contained within all paths from the set. Recall that a subsequence is a sequence that is produced via deleting elements from a larger sequence. Since the only operation is deletion, the order of the elements in the subsequence must be the same as the order of those same elements in the original sequence. It should be clear that a substring is a special case of this, but where all elements in the subsequence are contiguous in the original string. We will now show that there is a relationship between sub-paths and substrings.

Let us now show that any algorithm which solves the longest common substring problem can be applied to the longest common sub-path problem for labeled graphs. Since in our graph model each edge has exactly two end points, we can uniquely define a path by remembering the edges that made up that path. Define $f(e)$ to be a unique symbol for an edge. Thus:

$$\Lambda = \{f(e)|e \in E\}$$

$$|\Lambda| = |E|$$

Thus there is a bijection from edges to these new symbols. Note that these are not the same as the labels along the edges since these labels must only be distinct between pairs of vertices. These new symbols are globally distinct. There is exactly one edge with each symbol s.

Since we can uniquely define a path by recording the edges that make up that path we can uniquely define the same path by recording the symbols associated with each edge in the path. Therefore we can simply record the string of symbols corresponding to edges in the path and use it to define the path. Even assuming a naive linear lookup to convert an edge to a symbol, we can convert all possible paths to strings in $\Omega(|V|^{2^k+1}) \times |E|$. Since $|E| \leq |V|^2|\Sigma|$ we know that this means we can convert all paths to strings in $\Omega(|V|^{2^k+3}|\Sigma|)$.

Given the set of shortest paths between all pairs of vertices, we convert them as above to

a set of strings, Ψ. For any of these strings s, all substrings t are unique paths $f^{-1}(t)$ in the graph as well, though perhaps not validly labeled. Therefore, finding the longest common substrings will be equivalent to finding the longest common sub-path.

The longest common substring problem is formally: given K strings $\{s_1, s_2, \ldots, s_K\}$ and let $m = \sum_{i=1}^{K} |s_i|$, for $k \in [2, K]$ define $len(k)$ to be the length of the longest common substring of at least k of these strings (Gusfield 2007). Borrowing an example from the same work: Consider the set of strings $\{sandollar, sandlot, handler, grand, pantry\}$. Then the table of $len(k)$ value is the following:

k	$len(k)$	one substring
2	4	sand
3	3	and
4	3	and
5	2	an

Table 4.1. $len(k)$ table for longest common substrings

Such a table can be computed in $O(m)$ time (Hui 1992) using generalized suffix trees. Also, it should be clear that having such a table will be very advantageous for compressing a set of strings.

Given our list of strings, we want to find the maximum amount of redundancy. Then we want to remove all but one of them since we will need one remaining substring in order to rebuild all of the other strings. Thus, we desire the k such that $(k-1) \times len(k)$ is maximized. That is, the k such that the maximum amount of characters can be removed from the set of strings as a whole. Translating this back to paths by taking each character c and replacing it with the edge $f^{-1}(c)$, we can rebuild the original paths, though where the redundancy lies, we will replace the edge with a pointer and a marker saying where to return to. Clearly this does not violate our constant time lookup of the next hop. Also, in our example above of the Figure 4.2, the path through the region labeled "widget" must only be stored once, for a net savings of $n^{2^k}(n-1)$.

4.4 Conclusion

This chapter showed that a successor matrix for the CLH_k-LAPSP problem has size at least $\Omega(n^{2^k+1})$ where k is the position in the Control Language Hierarchy and n is $O(|V|)$. Further, it provides a way to maintain such a routing table in a smaller space while maintaining a constant look up by finding longest common sub-paths. While this technique may not be optimal, it appears to be novel and does yield significant space savings.

CHAPTER 5 – LABELED PATH ALGORITHMS FOR THE k-HIERARCHY

Khabbaz in (Khabbaz 1974) defines a geometric hierarchy of languages which are more powerful than the context-free languages, but do not have as much power as the context-sensitive languages. He defined his hierarchy in the following way:

$$\begin{aligned} \mathscr{L} &= \text{Labeled linear grammars} \\ \mathscr{L}_1 &= CFL \\ \mathscr{L}_i &= \{L(\mathbf{G}, C) | \mathbf{G} \in \mathscr{L}, C \in \mathscr{L}_{i-1}\} \end{aligned}$$

Recall that a linear grammar is one of the following form:

$$X \rightarrow \alpha Y \beta \text{ where } \alpha, \beta \in \Sigma^*$$

$$Y \rightarrow \gamma \text{ where } \gamma \in \Sigma^*$$

In other words, the i^{th} level of the hierarchy is defined as the languages which can be recognized by a linear grammar which is controlled by a language which is in the $(i-1)^{th}$ level of the hierarchy.

For the purposes of this dissertation, Khabbaz's geometric hierarchy will be called the K-Hierarchy.

It might seem that building the hierarchy upon the linear grammars would limit the power considerably. However, each level does have surprising power. For example, the language $L = \{a^n b^n c^n d^n, \Sigma = \{a, b, c, d\}\}$ is not context free. We now show that this language is in

$\mathscr{L}_2$.

$$\begin{aligned} \ell_1 &: A \to aAd \\ \ell_2 &: A \to bAc \\ \ell_3 &: A \to \epsilon \end{aligned}$$

This grammar will recognize all strings in L, but it is overly general—it will also match strings which are not in L. So, a constraint language placed upon the labels of the parse tree is used to remove undesirable strings. In this case, we let the constraint language be the context-free language $\{\ell_1^n \ell_2^n \ell_3\}$. This means that the production labeled ℓ_1 is applied first and applied n times. Thus the derivation after this is done is:

$$A \Rightarrow \underbrace{aa\ldots a}_{n} A \underbrace{d\ldots dd}_{n}$$

Then apply the productions labeled ℓ_2 n times. The derivation becomes:

$$A \Rightarrow \underbrace{aa\ldots a}_{n} \underbrace{bb\ldots b}_{n} A \underbrace{cc\ldots c}_{n} \underbrace{d\ldots dd}_{n}$$

Lastly, apply the production labeled ℓ_3 a single time. This production is the ϵ production which derives the empty string. Thus this language is obviously $\{a^n b^n c^n d^n | n \geq 0, \Sigma = \{a, b, c, d\}\}$ and in $\mathscr{L}_2$ since the control language is in $\mathscr{L}_1 = CFL$. Formally:

$$A \underset{\ell_1^n \ell_2^n \ell_3}{\overset{*}{\Rightarrow}} a^n b^n c^n d^n$$

For another example of the power of the second level of this hierarchy, recall that the language $\{ww | w \in \Sigma^*\}$ is not context free when $|\Sigma| \geq 2$. For this example, let $\Sigma = \{a, b\}$.

Consider the following labeled linear grammar:

$$\begin{aligned}
\ell_1 &: A \to aAa \\
\ell_2 &: A \to bAb \\
\ell_3 &: A \to aAb \\
\ell_4 &: A \to bAa \\
\ell_5 &: A \to a \\
\ell_6 &: A \to b \\
\ell_7 &: A \to \epsilon
\end{aligned}$$

This language, sans labels, is obviously the regular language $(a|b)*$. However, utilizing the labels we can restrict it to only the desired language. First observe that each production matches one character at the beginning of the string and one at the end. So, keeping track of the labels of the productions will give us a record of what character was matched forward (and its equivalent backwards). Thus we can just have a slightly modified palindromic grammar to define a control language:

$$\begin{aligned}
S &\to X\ell_7 \\
X &\to \ell_1 X \ell_1 \\
X &\to \ell_2 X \ell_2 \\
X &\to \ell_3 X \ell_4 \\
X &\to \ell_4 X \ell_3 \\
X &\to \ell_1 \\
X &\to \ell_2 \\
X &\to \epsilon
\end{aligned}$$

This is a standard palindromic grammar but for the productions which match an ℓ_3 against an ℓ_4 (and vice versa). This is because the production labeled ℓ_3 in turn matches the string aAb and the production labeled ℓ_4 in turn matches the string bAa. Thus, combined they match $abAab$.

Consider the string "ababbaababba". This string is in $\{ww\}$ since we can let $w = ababba$. So, applying the labeled linear grammar from above to this language, we get the following control word that uniquely specifies the parse: $\ell_1\ell_2\ell_3\ell_4\ell_2\ell_1\ell_7$. Applying the control grammar from above, we see that this control word is in fact accepted, and thus the original string is accepted.

Note that both of these example grammars which are in $\mathscr{L}_2$ are also in CLH_2. In fact, the K-Hierarchy is completely contained within the CLH hierarchy. Simply consider each of the productions from the control grammars in the K-Hierarchy to be distinguished productions from the CLH. The distinguished productions each have a distinguished non-terminal which in this case would be the only non-terminal on the right hand side since we only consider linear grammars. In fact, Weir(Weir 1992) had exactly this extension in mind when he built the Control Language Hierarchy.

5.1 Labeled Normal Form

Often, for algorithmic convenience we convert context-free grammars to Chomsky Normal Form (Hopcroft, Motwani, and Ullman 2006). Every context-free grammar can be converted into an equivalent Chomsky Normal Form (CNF) representation. CNF was used for both the `BJM` and `FastBJM` algorithms from Chapter 2.

The base of the K-Hierarchy, $\mathscr{L}_1$, is exactly the context-free languages. However, in order to define the higher levels of the hierarchy we require that the CFGs have labeled productions. So, it is necessary to create a parallel to CNF which maintains these labelings in a meaningful way.

Since the algorithms in this chapter focus on only linear grammars, we introduce the

notion of a Labeled Normal Form (LNF), which differs from the normal definition of a linear grammar only in that there is at most one terminal on each side of the non-terminal in the production.

Note that there are techniques for removing unit productions from normal grammars. Thus, we assume that the linear grammar has no productions of the form $A \to B$.

In order to convert some production of a labeled linear grammar to LNF, we first telescope the production such that it has at most one terminal on each side of the non-terminal in the production, giving each new production a unique label.

$$\begin{aligned} \ell &: X \to abcYdefg \\ &\overset{\text{telescope}}{\mapsto} \\ \ell_0 &: X \to aX_1g \\ \ell_1 &: X_1 \to bX_2f \\ \ell_2 &: X_2 \to cX_3e \\ \ell_3 &: X_3 \to Yd \end{aligned}$$

Note that this would work exactly the same if there were only only terminals on the right hand side. Next, translate each of these new productions of the form $\ell_k : X_k \to tX_{k+1}u$ to be in LNF, with exactly one terminal per production. This would become $\ell_{k,1} : X_k \to tX_{k,2}$ and $\ell_{k,2} : X_{k,2} \to X_{k+1}u$. Thus, applied to the productions from the example above:

$$\ell \;:\; X \rightarrow abcYdefg$$

$$\overset{\text{LNF}}{\Rightarrow}$$

$$\ell_{0,1} \;:\; X \rightarrow aX_{0,2}$$

$$\ell_{0,2} \;:\; X_{0,2} \rightarrow X_1 g$$

$$\ell_{1,1} \;:\; X_1 \rightarrow bX_{1,2}$$

$$\ell_{1,2} \;:\; X_{1,2} \rightarrow X_2 f$$

$$\ell_{2,1} \;:\; X_2 \rightarrow cX_{2,2}$$

$$\ell_{2,2} \;:\; X_{2,2} \rightarrow X_3 e$$

$$\ell_{3,1} \;:\; X_3 \rightarrow Yd$$

Then, we replace every instance of ℓ in the control language with $(\ell_{0,1}\ell_{0,2}\ldots\ell_{n,1}\ell_{n,2})$. Note that when applying the rewrite rules from above that the new non-terminals and the new production labels introduced must be unique. Otherwise, the language could be changed. Note that application of these new productions yields the exact same matching as the old production. Also note that since all occurrences of ℓ were replaced by the concatenation of all of the new labels we added, then the control language will also enforce the same language.

Each production in the original grammar is transformed into $|\alpha| + |\beta|$ new productions.

5.2 $K_{Linear,2}$ APSP Algorithm

The K-Hierarchy requires that the CFG which is to be controlled have labels from some set, Γ. It is required that each production have exactly one label, however it is not required that those labels be distinct. However, as can be seen from the previous examples, much power is had in grammars which do have distinctly labeled productions. Thus, let us define the

K'-Hierarchy as the K-Hierarchy but with an added restriction that the CFG at the base of the hierarchy have productions which are uniquely labeled. Further, let L_k mean a language in level k of this modified hierarchy.

With this new constraint, the control word has more power. The control word is prescriptive of how the parse should take place. Since each character in the word corresponds to a unique production, one can parse some string in the base, controlled grammar, by applying the productions in the order prescribed by the control word. This important observation yields itself well to the act of finding labeled shortest paths.

We now make another important observation. Linear languages have parse trees such that every node is a leaf or has exactly one child node that is not a leaf. That is, all of the internal (non-leaf) nodes in the tree lie along a single path from the root. Since the underlying grammar being controlled has uniquely labeled productions, this parse tree also uniquely specifies a single string in the language L_k.

Now, note that because of the uniqueness and structure of the parse tree of the linear language, we can instead use $Rev(C)$—the reverse of the controlling language C—and achieve the same parse tree, but upside down. That is, the root is now the deepest internal node of the tree and vice versa, and also that leaf nodes which were the left children are now the right and vice versa. By doing this, we can use the pre-order traversal of this new tree to prescriptively parse strings in L_k from the inside out. Also, note that any linear grammar can be trivially reversed by simply reversing each production. Thus, if $X \to \alpha Y \beta$ is some production, then $Rev(X) \to Rev(\beta) Y Rev(\alpha)$. This crucial observations lends itself handily to building a labeled all pairs shortest path algorithm.

Again, consider the example of from before of the grammars which match $\{a^n b^n c^n d^n | n \geq 0\}$.

Controlled grammar(G):

$$\begin{aligned} \ell_1 &: A \rightarrow aAd \\ \ell_2 &: A \rightarrow bAc \\ \ell_3 &: A \rightarrow \epsilon \end{aligned}$$

Controlling grammar(C):

$$\begin{aligned} S &\rightarrow Tl_3 \\ T &\rightarrow l_1 T l_2 \\ T &\rightarrow \epsilon \end{aligned}$$

Now, $Rev(C)$ becomes:

$$\begin{aligned} S &\rightarrow l_3 T \\ T &\rightarrow l_2 T l_1 \\ T &\rightarrow \epsilon \end{aligned}$$

So, by looking at strings derived from C it can be seen that the string $s =$ "*aabbccdd*" which is in G would be parsed inside out. The first production of $Rev(C)$ causes the production labeled l_3 to match. In this case, this is an epsilon production, so the string being generated would not get any longer. Next, some production of the non-terminal T must be evaluated. In this case, since the s is non-empty, we apply the first production of T. This causes the production l_2 to attempt to match in the string. So now, the generated string is "bc". Next, T is evaluated again leaving $l_2 T l_1 l_1$ yet to be parsed. Applying the production

labeled with l_2 again we see that the string produced becomes "bbcc". Now, let this T be the epsilon production, so the symbols that are remaining are $l_1 l_1$. Thus, apply the production labeled l_1 two times. This yields the string "aabbccdd" which is exactly what was expected.

The basic idea of the algorithm is that it will start at each vertex then attempt to form a path backward and forward from this point by applying the productions with labels from $Rev(C)$. Because of this, each production which is applied from the underlying, controlled grammar must cause this path to expand only at the ends of this path. In other words, if we were to parse the string representing the path, then it would be parsed left to right. Further, in order to remove any ambiguity in figuring out which end of the string to apply a singleton production, let us consider grammars that have only one singleton production that can be matched in any parse. This is the case with linear grammars, as is the grammar being controlled. For the sake of convenience, allow Λ to be the grammar being controlled, a context-free grammar that has uniquely labeled productions and which are linear. Also, let Υ be the linear language which controls Λ.

We do make one constraining assumption upon the controlling grammar. In the traditional K-Hierarchy, the top most controlling grammar may be any context free grammar. However, to facilitate the construction of the algorithm, we will restrict this grammar to being linear as well. Though this may seem to be quite restrictive, the power of the hierarchy that enforces this requirement is enough to contain all of the previous examples. We call this new hierarchy that is controlled at the highest level by a linear grammar the K_{Linear}-Hierarchy, and refer to level k of the hierarchy as $K_{Linear,k}$.

So, Υ can be used to prescriptively parse Λ to yield a string which is not context-free, but is Λ_Υ (the language Λ controlled by Υ which is in $\mathscr{L}_2$). Also, since Λ is linear, application of the productions—even in the reverse order prescribed by $Rev(\Upsilon)$—will cause the string to be built a terminal at a time, maintaining contiguity. Thus, for some candidate path the next production which is applied from the controlled grammar will dictate which end of the path should be extended. For example, if the production in the controlled grammar which

is being prescriptively applied is $X \to aY$ then the path must grow by having a new edge precede the extant path. Symmetrically, $X \to Yb$ would cause the path to have a new edge (should one exist with label b) appended to the end. This is because of the nature of the linear grammars and the fact that we are applying the productions in the reverse of their normal order. Since the singleton production can only happen once, it must be the very first production applied in this reverse order strategy.

Having discussed how a path is expanded based upon which type of controlled production is applied, let us now consider how such a production is chosen. Since the control grammar, Υ, is linear it can be parsed recursively. Because of this, we can simulate such recursion by using stacks. Thus, in addition to storing a candidate-path between vertices u and v it is necessary to also store the stack ρ which is the stack corresponding to the recursion mentioned previously. Further, we insert this candidate path and its stack into a heap which supports $extractMin$ and use the current length of the path as the key.

We begin the algorithm by attempting to build candidate paths from every vertex in the graph. So, we insert the path of length 0 with endpoints v and v into the heap with the stack that contains only the start symbol of the controlling grammar, S_Υ. Then, $extractMin$ from the heap will return one of these candidate paths. The top of the stack, here S_Υ, will be popped and then a new candidate path (tuple of endpoints and the parse stack) will be created for each production of the non-terminal S_Υ. For each, we push the components of the production onto the stack from right to left.

Eventually, the top of the stack which was part of the tuple removed from the heap via $extractMin$ will be a terminal ℓ_Υ. This terminal corresponds to exactly one production in the controlled grammar Λ. That controlled production expands the path as described above and the algorithm continues. However, there is one further case that must be considered. The case where the controlled production is of the form $X \to c$. Since we require the language to be linear, such a production must be matched last in a normal parse tree, of first in our reversed language. Thus, it is only it is only necessary to attempt to check the

edges out of some node u to see if any are labeled with c. Only those edges out of u need be checked since this is being done for all vertices and if there is an edge into u it must have been added to the paths for the vertex which points to u via this edge.

When a candidate path, extracted via $extractMin$ from the heap, has an empty stack it means that the control string was successfully parsed which in turn means that the string representing the path was validly parsed by Λ_Υ. In this event, we record the length of this string in a distance table.

Algorithm 5.1 $K_{Linear,2}$ APSP Algorithm: input (G, L, lengthThreshold)

```
 1: D := |V| × |V| table
 2: heap := CreateEmptyHeap()
 3: for all v ∈ V, ℓ ∈ Σ do
 4:    heap.insert({0, (v, v, 0, S_Υ)})
 5: end for
 6: longestShortestPath := 0
 7: while isNotEmpty(heap) do
 8:    {key, (u, v, dist, ρ)} := extractMin(heap)
 9:    if key > lengthThreshold + |N_Υ| then
10:       break loop
11:    end if
12:    if DTableSize > |V|^2 AND dist > longestShortestPath + 1 then
13:       break loop
14:    end if
15:    if first(ρ) is a terminal then
16:       handleTerminal(heap, {key, (u, v, dist, ρ)})
17:    else
18:       handleNonTerminal(heap, {key, (u, v, dist, ρ)})
19:    end if
20: end while
```

5.2.1 Correctness

Theorem 5.1. *The $K_{Linear,2}$ APSP Algorithm terminates.*

Proof. Let us first begin by noticing the nature of the heap. The *key* of each heap element can be thought of as the penalized current estimate of the length of some validly labeled shortest path. In order to understand the nature of the penalty, consider the case where the

Algorithm 5.2 handleTerminal(heap, $\{key, (u, v, dist, \rho)\}$))

```
1: t := ρ.pop()
2: if p is of the form X → aY then
3:    for all k ∈ V do
4:       if (k, u, a) ∈ E then
5:          heap.insert({dist + 1, (k, v, dist + 1, ρ)})
6:       end if
7:    end for
8: end if
9: if p is of the form X → Yb then
10:   for all k ∈ V do
11:      if (v, k, b) ∈ E then
12:         heap.insert({dist + 1, (u, k, dist + 1, ρ)})
13:      end if
14:   end for
15: end if
16: if p is of the form X → c then
17:   if dist > 0 then
18:      halt. There must be a parse error
19:   end if
20:   if c = ϵ then
21:      heap.insert({0, (u, v, 0, ρ)})
22:      continue {COMMENT: since this will not lengthen the path}
23:   end if
24:   for all k ∈ V do
25:      if (k, u, c) ∈ E then
26:         heap.insert({1, (k, v, 1, ρ)})
27:      end if
28:   end for
29: end if
30: if ρ is empty then
31:   {COMMENT: The path between u and v is validly labeled}
32:   if D[u][v] = undefined then
33:      DTableSize := DTableSize + 1
34:   end if
35:   D[u][v] := min(D[u][v], dist)
36:   if D[u][v] > longestShortestPath then
37:      longestShortestPath := D[u][v]
38:   end if
39: end if
```

control language is:

$$S \rightarrow Sa$$

$$S \rightarrow a$$

Also, assume that there was no penalty and the key of the heap was just the length of the current path. Then, when some heap element is removed from the heap via $extractMin$ the state of ρ would need to be examined. Now, consider the case when the element on top of the stack is the non-terminal S. Thus, $handleNonTerminal$ is called, causing two new candidate paths to be added to the heap, each with the same key, but with different stacks. One has the original S replaced with Sa and the other has it replaced with a. Now, because the key is just the length of the path, these two new heap elements will have exactly the same key, but only one will be removed via $extractMin$. In the event that the former was removed first, it would be expanded exactly as before. In this case, an infinite loop could arise where the path length does not increase but the size of ρ would grow toward infinity.

In order to alleviate this problem, a penalty is imposed. Whenever a non-terminal is on top of the stack causing new candidate paths to be added to the heap, we set the new key of these elements to be one larger than the previous key. This in effect acts like a penalty on expanding such non-terminals and biasing the search toward those heap-elements which have terminals on the top of their stacks. Whenever a path actually gets expanded, its key is reset to be the distance. Thus, the penalty only lasts as long as non-terminals on ρ continue to be expanded.

Algorithm 5.3 handleNonTerminal(heap, $\{key, (u, v, dist, \rho)\}$)

```
1: X := ρ.pop(){COMMENT: this is a NonTerminal}
2: for all X → η ∈ P_control do
3:     σ := concatenate(η, ρ)
4:     heap.insert({key + 1, (u, v, dist, σ)})
5: end for
```

Consider Line 9 of Algorithm 5.2. Because of the bias toward terminal productions, the

only way that the minimum element in the key could be this large is if all smaller possible paths had already been considered. If $key > lengthThreshold + |N_\Upsilon|$ this means that either the path is longer than the threshold or that it has continually been penalized. If it were the case that the path were longer than *lengthThreshold* then we should terminate. This guard is put in place because it is possible that a string will always be in the matching state. For example, consider a cyclic graph with an even number of vertices. Thus, the path from u to u would be even length. Now, if the input formal language only matches odd-length strings, then this algorithm would never terminate, because for every cycle through the path, the corresponding string would only have even length. In the case of the BJM algorithm from Chapter 1, it was known that the longest possible shortest path was $|V|^2|N|$, thus lending itself to a threshold which also contained all correct solutions. By Chapter 4, we know that the longest shortest path for level k in the K_{Linear} is $\Omega(|V|^{2^k})$. If an upper bound can be proven for the longest shortest path then the threshold for this algorithm can be set to also contain all valid solutions.

If the path length was not longer than length threshold, then it might be the case that it could be lengthened later. However, it must be the case that only $|N_\Upsilon|$ non-terminals can be expanded on the path before a terminal is added because otherwise, this would be a loop that could continue forever. Since the key gets larger whenever one of these non-terminals is expanded, it must be the case that eventually this threshold is reached.

However, the algorithm may terminate sooner than when the distance hits *threshold*. Line 12 will lead to the algorithm terminating when all shortest paths are known. This is discussed more in the proof of Theorem 5.3.

Also, the algorithm may terminate when the heap is empty. This happens when there are no valid candidate paths remaining to be built upon. □

Theorem 5.2. *The $K_{Linear,2}$ APSP Algorithm finds only valid paths.*

Proof. For the sake of contradiction, assume that the algorithm finds paths which are invalid, that is that the string formed by the concatenation of the labels along the path is not in the

input formal language. For this to be the case then there must have been some point when the string was still valid and then the next edge was added (either to the beginning or the end) of the path causing it to be invalid. An invalid path can only be added to the heap in Lines 5, 12, or 26 of Algorithm 5.2. Note that cases 5 and 12 are symmetric, so we need only consider Line 5.

At this point in the code p must be the single production of the underlying grammar which is labeled with t. This is based off of our assumption that all productions in the grammar were uniquely labeled. First, note that because this terminal was the first thing in ρ it must be the correct production to apply because of the aforementioned uniqueness of the parse tree. In this line, the production must be of the form $X \to aY$, so the next character to prepend to the current string (representing the concatenation of all of the edge labels along this candidate path) must be this character a because of the prescriptive nature of the control language. Finally, Line 4 verifies that there is some vertex k which has an edge to u labeled with a. Thus, there is no way an invalid string could be added. $\Rightarrow\Leftarrow$

The other instance where such an invalid edge could be added is in Line 26. However, Line 26 only gets applied when the candidate path is empty and this is the first of the controlled productions being applied. Thus, because of the assumed validity of the input grammars, this must also be correct.$\Rightarrow\Leftarrow$ □

Theorem 5.3. *The $K_{Linear,2}$ APSP algorithm finds the shortest valid paths.*

Proof. Obviously, by line 35 of Algorithm 5.2, $D[u][v]$ is the length of the shortest valid path found during the execution of the algorithm.

Consider the two termination cases: the distance is greater than a threshold or the table is full and the distance is one bigger than the longest shortest path found.

1. Threshold

   As was stated in Theorem 5.1, the threshold is used to terminate the algorithm in the case of a possibly infinitely long accepting path. If the threshold is set too low, then

some valid paths will not be found. Therefore, for this proof we will assume that the threshold is set large enough to encompass all valid finite shortest paths.

2. D-Table is full and the distance is one bigger than the longest shortest path

   First, let us consider when the entries in the D-table can be updated. The first time that any accepting path between u and v is found, its length is recorded in $D[u][v]$. The question now is, when can this value be modified? Note that since we always perform $extractMin$ from the heap we are always working on the current minimum estimate of the shortest paths. So, the only possible time when $D[u][v]$ can be overwritten is when there are two candidate paths which have exactly the same length. Because we assume that the only productions which can be applied are of the form $X \rightarrow aY$ or $X \rightarrow Yb$ because unit productions have been removed, the distance must increase by exactly one for any path. Thus, the value of $D[u][v]$ can never decrease.

   Because a row of this D-table can never be overwritten, we can ignore all accepting paths which have endpoints u and v. Thus, in the case of Line 12 of Algorithm 5.2, the algorithm can terminate when all shortest paths have been found. That is, there is a shortest distance between all u and v in the D-table and the next candidate from the heap is more than one larger in length.

□

5.3 Analysis of the $K_{Linear,2}$ APSP Algorithm

In order to facilitate the analysis of the algorithm, let us begin by making several simplifying assumptions. Assume that the grammar being controlled is not only linear but also contains at most one terminal per production. Thus, the application of any production will cause the candidate path to grow by exactly one edge. Second, let us assume that the control language is normal-form right-linear – all productions are of the form $X \rightarrow aY$ or $X \rightarrow a|\epsilon$. Note that the right-linear languages are exactly the regular languages.

Now, note that the size of stack ρ will never be more than two. And the first element in ρ is always a control terminal rather than a control non-terminal. Thus, this will mean that every $extractMin$ causes some path to grow by exactly one in length. Therefore, to analyze this scenario we need only consider how long such valid paths can be. By Chapter 4 and by extension the first section in this chapter, we know that the longest shortest path for level k in the K_{Linear}-Hierarchy is $\Omega(|V|^{2^k})$. So, for level 2 of the hierarchy, this shortest string is $\Omega(|V|^4)$. Therefore, finding $K_{Linear,2}$ APSP paths is $\Omega(|V|^6)$ since we must consider paths between all pairs of vertices.

5.4 K_{Linear}-Hierarchy APSP Algorithms

Recall that the path is built from the inside-out simply by reversing the control string. This effectively reversed the entire language so that the path could be built in this manner. This effect carries on up the hierarchy, there is no need to reverse any of the higher control grammars since the language itself was reversed by simply reversing the control grammar at level 2 of the hierarchy.

Given this, the algorithm for $K_{Linear,2}$-APSP can be easily extended to encompass the infinite hierarchy by making a few observations. First, recall the purpose of the stack ρ in the algorithm. The stack allows the algorithm to simulate the recursive parsing of the labels on the controlled grammar. Whenever a non-terminal was on the top of the stack, new candidate paths were created for every possible production from that non-terminal. By requiring that all grammars in the hierarchy have uniquely labeled productions, this expansion of all possible productions is alleviated since the control-grammar that is higher in the hierarchy prescribes exactly which production must be applied. Only the top-most control grammar would yield new candidate paths being added to the heap since there it is unknown which of its productions would be applied next.

The $K_{Linear,2}$ algorithm is extended by utilizing $k-1$ stacks for the instance of the problem which operates on languages from level k of the hierarchy. Label these stacks $\rho_{k-1}, \ldots, \rho_1$

where ρ_{k-1} is for the highest control grammar. Each ρ_i corresponds to control grammar, Υ_i. Each stack ρ_i prescribes what should happen to the non-terminal on the top of stack ρ_{i-1}, for $i > 1$. Whenever a terminal is on top of ρ_1, the stack corresponding to the lowest control grammar, the production corresponding to this terminal is applied exactly as in the $K_{Linear,2}$ algorithm.

The algorithm begins by inserting zero-length candidate paths into the heap, exactly as the $K_{Linear,2}$ algorithm does. However, in this case, the heap elements are extended to contain a copy of each stack, $\rho_{k-1}, \ldots, \rho_1$. In the previous algorithm, the start symbol of the control language was initially placed upon the stack. Here, the same thing is done, but only on the top-most stack.

During each iteration of the algorithm, we $extractMin$ as before. However, instead of only branching based on whether the symbol on the top of the stack was a non-terminal or a terminal, we must investigate each stack. Since the stack ρ_1 corresponds to productions that should be applied to the base, controlled grammar, it is examined first. All terminals on the top of this stack are applied in the manner described above. This examination terminates for stack ρ_1 when either it is empty, or a non-terminal is on the top of the stack. In the previous algorithm, whenever a non-terminal was on top of the stack, it was necessary to create candidate paths for all possible productions of that non-terminal. However, here we know exactly which production of the non-terminal to apply by looking at the terminal on the top of the stack one level higher in the control grammars.

So, we apply all terminals on top of the stack ρ_1, then examine the stack ρ_2 to see which production in Υ_1 should be expanded. This pattern continues up the hierarchy. Beginning at stack ρ_2 and continuing to ρ_{k-2}, if there is a terminal on top of the stack ρ_i we remove this terminal, and apply the production corresponding to it in the lower control language, Υ_{i-1}, removing the non-terminal (if the stack is not already empty) from the top of ρ_{i-1} and replacing it with the production corresponding the terminal from ρ_i. Applying such a production might have caused a terminal to be the top of stack ρ_{i-1}, so after each production

is expanded, we recursively examine the stacks which are lower in the hierarchy, removing terminals and applying appropriate productions where necessary.

For ρ_{k-1}, which corresponds to the highest control grammar, it is not known which of the productions of a non-terminal on top of the stack should be examined. Thus, just as in the case of $K_{Linear,2}$, we insert new candidate paths into the heap for all possible productions of this non-terminal.

Whenever, all of the stacks are empty, it means that the path is validly labeled and the distance of this path is recorded in the D-Table just as before.

Algorithm 5.4 General $K_{Linear,k}$ APSP Algorithm: input (G, L, lengthThreshold)

```
1: D := |V| × |V| table
2: heap := CreateEmptyHeap()
3: for all v ∈ V, ℓ ∈ Σ do
4:    heap.insert({0, (v, v, 0, ρ_{k-1} = S, ρ_{k-2} = ∅, ···, ρ_1 = ∅)})
5: end for
6: longestShortestPath := 0
7: while isNotEmpty(heap) do
8:    {key, (u, v, dist, ρ_{k-1}, ρ_{k-2}, ..., ρ_1)} := extractMin(heap)
9:    if dist > lengthThreshold then
10:      break loop
11:   end if
12:   if DTableSize > |V|^2 AND dist > longestShortestPath + 1 then
13:      break loop
14:   end if
15:   for i ∈ [1, k − 1] do
16:      if first(ρ_i) is a terminal then
17:         handleTerminal(heap, i, {key, (u, v, dist, ρ_{k-1}, ρ_{k-2}, ..., ρ_1)})
18:      end if
19:   end for
20:   handleNonTerminal(heap, k − 1, {key, (u, v, dist, ρ_{k-1}, ρ_{k-2}, ..., ρ_1)})
21: end while
```

5.4.1 Correctness of $K_{Linear,k}$ APSP

Theorem 5.4. *The $K_{Linear,k}$ APSP algorithm terminates.*

Proof. The argument for termination of the algorithm is exactly the same as for $K_{Linear,2}$. The path length must grow beyond some boundary, or it might be the case that a cycle of

Algorithm 5.5 handleControlledProduction(heap, $\{key, (u, v, dist, \rho_{k-1}, \rho_{k-2}, \ldots, \rho_1)\})$)

```
1: t := ρ_1.pop()
2: if p is of the form X → aY then
3:    for all k ∈ V do
4:       if (k, u, a) ∈ E then
5:          heap.insert({dist + 1, (k, v, dist + 1, ρ_{k−1}, ρ_{k−2}, ..., ρ_1)})
6:       end if
7:    end for
8: end if
9: if p is of the form X → Yb then
10:   for all k ∈ V do
11:      if (v, k, b) ∈ E then
12:         heap.insert({dist + 1, (u, k, dist + 1, ρ_{k−1}, ρ_{k−2}, ..., ρ_1)})
13:      end if
14:   end for
15: end if
16: if p is of the form X → c then
17:   if dist > 0 then
18:      halt. There must be a parse error
19:   end if
20:   if c = ϵ then
21:      heap.insert({0, (u, v, 0, ρ_{k−1}, ρ_{k−2}, ..., ρ_1)})
22:      continue {COMMENT: since this will not lengthen the path}
23:   end if
24:   for all k ∈ V do
25:      if (k, u, c) ∈ E then
26:         heap.insert({1, (k, v, 1, ρ_{k−1}, ρ_{k−2}, ..., ρ_1)})
27:      end if
28:   end for
29: end if
30: if ∀i ∈ [1, k − 1], ρ_i is empty then
31:   {COMMENT: The path between u and v is validly labeled}
32:   if D[u][v] = undefined then
33:      DTableSize := DTableSize + 1
34:   end if
35:   D[u][v] := min(D[u][v], dist)
36:   if D[u][v] > longestShortestPath then
37:      longestShortestPath := D[u][v]
38:   end if
39: end if
```

Algorithm 5.6 handleTerminal(heap, i, $\{key, (u, v, dist, \rho_{k-1}, \rho_{k-2}, \ldots, \rho_1)\}$)

```
1: while top(ρ_i) is a terminal do
2:    if i = 1 AND top(ρ_1) is a terminal then
3:       handleControlledProduction(heap, {key, (u, v, dist, ρ_{k-1}, ρ_{k-2}, ..., ρ_1)})
4:       Continue
5:    end if
6:    label := ρ_i.pop()
7:    next := Production labeled with 'label' from control grammar i − 1
8:    ρ_{(i−1)}.pop()
9:    ρ_{(i−1)} = concatenate(next, ρ_{i−1})
10:   handleTerminal(heap, i - 1, {key, (u, v, dist, ρ_{k-1}, ρ_{k-2}, ..., ρ_1)})
11: end while
```

Algorithm 5.7 handleNonTerminal(heap, $\{key, (u, v, dist, \rho_{k-1}, \rho_{k-2}, \ldots, \rho_1)\}$)

```
1: X := ρ_{k−1}.pop() {COMMENT: this is a NonTerminal}
2: for all X → η ∈ P_{Υ_{k−1}} do
3:    σ := concatenate(η, ρ_{k−1})
4:    heap.insert({key + 1, (u, v, dist, σ, ρ_{k−2}, ..., ρ_1))})
5: end for
```

non-terminal expansions is occurring, adding penalties to the path. In either case, the keys must eventually get larger and pass this threshold. □

Theorem 5.5. *The $K_{Linear,k}$ APSP algorithm finds the lengths of only valid paths.*

Proof. Again, the argument is as for $K_{Linear,2}$. In order for for an invalid path to be added, it must have had a point at which it became invalid. That is, the candidate path was valid, but then application of some production caused it to become invalid. However, since each production is prescribed by the level above it, we know that these must be valid so long as the control level above it is valid. Thus we need only consider the grammar, Υ_{k-1}. However, we assume that this grammar is correct and therefore it must prescribe correct production applications. Thus, it is impossible to add an incorrect candidate path. □

Theorem 5.6. *The $K_{Linear,k}$ algorithm finds the lengths of shortest valid paths.*

Proof. The argument for this follows exactly from that of Theorem 5.3. □

5.5 Analysis of the $K_{Linear,k}$ APSP Algorithm

Again, note that Chapter 4 presented a graph that had a very long shortest path. In fact, this path is $\Omega(|V|^{2^k})$ in length. The k corresponds to the level of the K_{Linear} hierarchy that contains the constraining language. Thus, our algorithm for finding shortest paths between all pairs of vertices must be $\Omega(|V|^{2^k+2})$ since there are $|V|^2$ such paths.

While this does merit a more rigorous analysis, such an examination would require a detailed look at the controlling grammars being used. For example, note the discussion of the analysis of the $K_{Linear,2}$ APSP algorithm. The simplifying assumption was made that the control grammar be in normal-form right linear (i.e. a regular language) because it meant that each application would cause the path to grow by exactly one. If this assumption is not made then the expansion of non-terminals must occur causing new candidate paths to be added that are no longer than the original path. Thus, the frequency with which this happens greatly affects the running time of the algorithm.

5.6 Conclusion

This chapter has introduced two new concepts: the labeled normal form (LNF), and the K_{Linear}-Hierarchy APSP algorithm. The LNF is an extension of the Chomsky Normal Form to take into account production labels. By utilizing grammars represented in LNF, the K_{Linear}-Hierarchy APSP algorithm can find labeled shortest distances in graphs where the input formal language is from the K_{Linear}-Hierachy, such as $\{a_1^n a_2^n \cdots a_{2^k}^n | n \geq 1\}$.

In order to more deeply understand this algorithm several things need to be investigated further. For example, is there a natural threshold after which no shortest-valid path can exist like for the context-free instance of the problem? If such a threshold does exist, what is an upper bound on the running time of this algorithm? How can this algorithm be extended back to the K-Hierarchy, that is remove the constraint that the productions be uniquely labeled?

Another question which remains pertains to the power of the languages in the K_{Linear}-Hierarchy. In the K-Hierarchy, $\mathscr{L}_1 = CFL$, however here the base of the hierarchy is only those context free languages that can be matched with linear grammars. Is it possible that the power that would appear to be lost due to this restriction to linear controlled grammars can be reclaimed higher in the hierarchy? That is, is there some k such that $CFL \subset K_{Linear,k}$? If this is the case, then this new hierarchy would eventually be as powerful as the original, though at a higher level.

CHAPTER 6 – CONCLUSION AND FUTURE WORK

Labeled path problems are varied and complex. The works of Barrett, et al. (Barrett, Jacob, and Marathe 2000; Barrett, Bisset, Holzer, Konjevod, Marathe, and Wagner 2007; Barrett, Bisset, Jacob, Konejevod, and Marathe 2002) laid a foundation for solving these problems. The works presented in this dissertation, along with those of Bradford (Bradford and Choppella 2010) and Ward (Ward 2008), have built upon this foundation and provide evidence of how interesting and rich a field of study these problems may be.

Chapter 2 presented previously published works that I did with Dr. Charles Ward that were included as part of his dissertation. These included a distributed algorithm for finding context-free labeled shortest paths (Ward, Wiegand, and Bradford 2008) and also proofs of the complexity of certain routing metrics for wireless mesh networks (Ward and Wiegand 2010). The former was presented at the 37^{th} International Conference on Parallel Processing in Portland, Oregon. The latter appeared in the journal Computer Networks.

Chapter 3 discussed an empirical analysis of the two context-free labeled shortest path algorithms. Though the `BJM` algorithm has asymptotically worse running time, in practice it is sometimes the better choice. The analysis was completed by doing a distributed Monte Carlo simulation using MapReduce. I would again like to thank Google, Inc. for allowing me to borrow compute time on their internal MapReduce system. It was an invaluable tool for the purpose of this research.

The analysis was done using modified Erdös-Renyì random graphs. Additional analysis of these algorithms should be done in the light of different graph models. For example, how do the algorithms perform when we restrict the graphs to being series-parallel or tree-width

k? Also, large graphs like those in social networks would prove interesting for research. The random graph model presented in (Barabsi, laszlo Barabasi, Albert, and Jeong 1999) would be used to execute this analysis.

These algorithms were analysed in a machine-agnostic way so as to be completely repeatable by other researchers. However, only counting the number of comparisons each algorithm uses might miss some of the nuance of these algorithms. There is still room for further research about these algorithms in a machine-specific context, such as how each are affected by caching, paging, etc.

Further, in a similar vein, it would be interesting to do an empirical analysis of the distributed CFL-APSP algorithm presented in Chapter 2.

Chapter 4 provided a proof of the lower bound on the length of the longest labeled shortest path for graphs which are constrained by languages from the Control Language Hierarchy. This bound was then used to show a lower bound on the size of a routing table which would allow constant time lookup of next-hop lookup. Further, a method for compressing the routing table was proposed.

Lastly, Chapter 5 presents two algorithms for finding label constrained all-pairs shortest paths. The first finds paths when the language is constrained to be in the second level of the modified Khabbaz hierarchy, $K_{Linear,2}$. The second is an extension of the first to the entire hierarchy instead of just level 2. While these algorithms were proven correct, it still remains to do a more detailed analysis of their running times. It would be interesting to perform an empirical analysis similar to that of Chapter 3.

6.1 Further Open Questions

Though this is the second dissertation written on the topic of labeled paths ((Ward 2008)) the area is still replete with fruitful problems. This section will discuss possible avenues of further investigation into the area of labeled path problems and labeled graph theory as a whole.

In the Regular Language constrained all-pairs shortest path algorithm by Barrett, *et al.*, presented in Chapter 1, can we provide an algorithm that takes a regular expression defining the language rather than a state machine? The naive approach to this would be to simply transform a regular expression to a finite state machine before calling the original algorithm. However, it remains to be seen how we could utilize the regular expression directly to find the shortest paths.

For Chapter 3, several grammars were chosen for the analysis of the algorithms. Instead, it would be an interesting area of study to consider randomly generated CFGs and to investigate their properties.

The BJM algorithm computes the all-pairs shortest-path matrix by taking as input both the input graph and the language specified as a CFG in Chomsky Normal Form. Can we provide an algorithm which takes the language as a push down automata instead? Recall that a push down automata (PDA) is essentially a finite state machine which also has access to a single stack. Because of the parallel of the PDA with finite state machines, is there a similar way of finding shortest paths via a graph product?

There exist algorithms for computing shortest paths in graphs using Google's MapReduce technology. However, there are several limitations with computing labeled shortest paths. Most of the existing MapReduce graph algorithms work via a breadth first search strategy where a MapReduce job will merely move the search a single step forward. Thus, it requires $O(|V|)$ iterations of the MapReduce in order to find the shortest path. We know that the longest CFL labeled shortest path is $O(|V|^2|N|)$. This could be a limiting factor for this iterative style of MapReduce. Further investigation should be done into how one could efficiently compute shortest paths using MapReduce.

Beyond labeled paths are many other labeled graph problems. In Chapter 3 we discussed the use of modified Erdös-Renyì random graphs. Random graphs are a heavily studied area (Durrett 2006; Janson, Łuczak, and Ruciǹski 2000). Thus it would be very interesting to

study random labeled graphs and their properties such as labeled-connectedness, labeled diameter, etc.

Google's PageRank algorithm works by estimating the probability that a web-surfer behaving randomly will arrive at a given webpage. Thus, this process can be modeled with random walks. The algorithm assumes a democratic view of the internet, that is that there is no bias in how the network operates. However, a world without network neutrality could have it such that if the surfer originated from within network A then he could not access a document that is from network B, or that there is some penalty for doing so. Can we modify the notion of page rank to take this into account by using formal languages? If so, how could we use this result to accurately model a non-neutral Internet?

Finally, the languages in the Control Language Hierarchy can be recognized via embedded push down automata. Can we utilize these automata to find CLH_k labeled shortest paths?

6.2 Conclusion

This dissertation has further built upon the area of labeled graph problems. Though this is the second dissertation written on this topic the area is still ripe for research as was shown in the last section. Many more papers will be written in the area of labeled graph theory.

BIBLIOGRAPHY

Barabsi, A.-L., A. laszlo Barabasi, R. Albert, and H. Jeong (1999). Mean-field theory for scale-free random networks.

Barrett, C., K. Bisset, M. Holzer, G. Konjevod, M. Marathe, and D. Wagner (2007). Label constrained shortest path algorithms: An experimental evaluation using transportation networks. Technical report, Virginia Tech (USA), Arizona State University (USA), and Karlsruhe University (Germany). Work presented at the workshop on the DIMACS Shortest-Path Challenge.

Barrett, C., K. Bisset, R. Jacob, G. Konejevod, and M. Marathe (2002). Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In *ESA 2002, LNCS 2461*, pp. 126–138.

Barrett, C., R. Jacob, and M. Marathe (2000). Formal language constrained path problems. *SIAM Journal on Computing 30*(3), 809–837.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics 16*(1), 87–90.

Bradford, P. G. and V. Choppella (2010). Fast dyck and semi-dyck constrained shortest paths on dags. *In Preparation*.

Bradford, P. G. and D. A. Thomas (2009). Labeled shortest paths in digraphs with negative and positive edge weights. *Informatique Théorique et Applications (ITA) 43(3)*, 567–583.

Brodal, G. S. (1996). Worst-case efficient priority queues. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, pp. 52–58. Society for Industrial and Applied Mathematics.

Coppersmith, D. and S. Winograd (1987). Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, New York, NY, USA, pp. 1–6. ACM.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms*. McGraw-Hill.

Dean, J. and S. Ghemawat (2008, January). Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 107–113.

Draves, R., J. Padhye, and B. Zill (2004). Routing in multi-radio, multi-hop wireless mesh networks. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, New York, NY, USA, pp. 114–128. ACM.

Durrett, R. (2006). *Random Graph Dynamics (Cambridge Series in Statistical and Probabilistic Mathematics)*. New York, NY, USA: Cambridge University Press.

Erdös, P. and A. Renyì (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci 5*, 17–61.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM 5*(6), 345.

Fredman, M. L. and R. E. Tarjan (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*(3), 596–615.

Garay, J. A., S. Kutten, and D. Peleg (1998). A sub-linear time distributed algorithm for minimum-weight spanning trees. In *SIAM J. COMPUT*, pp. 659–668.

Garey, M. R. and D. S. Johnson (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Gusfield, D. (2007, January). *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press.

Hopcroft, J. E., R. Motwani, and J. D. Ullman (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Hui, L. C. K. (1992). Color set size problem with application to string matching. In *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, London, UK, pp. 230–243. Springer-Verlag.

Janson, S., T. Łuczak, and A. Ruciǹski (2000). *Random graphs*. New York, NY, USA: Wiley.

Johnson, D. B. (1977a). Efficient algorithms for shortest paths in sparse networks. *J. ACM 24*(1), 1–13.

Johnson, D. B. (1977b). Efficient algorithms for shortest paths in sparse networks. *J. ACM 24*(1), 1–13.

Khabbaz, N. A. (1974). A geometric hierarchy of languages. *J. Comput. Syst. Sci. 8*(2), 142–157.

Knuth, D. E. (1993). *The Stanford GraphBase: a platform for combinatorial computing*. New York, NY, USA: ACM.

Mendelzon, A. O. and P. T. Wood (1995). Finding regular simple paths in graph databases. *SIAM Journal on Computing 24*(6), 1235–1258.

Newman, M. E. J. (2002, Feb). Random graphs as models of networks.

Nykänen, M. and E. Ukkonen (2002). The exact path length problem. *J. Algorithms 42*(1), 41–53.

Seidel, R. (1995). On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences 51*(3), 400 – 403.

Wagner, R. A. and M. J. Fischer (1974). The string-to-string correction problem. *J. ACM 21*(1), 168–173.

Ward, C. B. (2008). *Labeled Path Problems, a dissertation*. University of Alabama.

Ward, C. B. and N. M. Wiegand (2010). Complexity results on labeled shortest path problems from wireless routing metrics. *Computer Networks 54*(2), 208 – 217. Wireless Multi-Hop Networking for Infrastructure Access.

Ward, C. B., N. M. Wiegand, and P. G. Bradford (2008). A distributed context-free language constrained shortest path algorithm. *ICPP 2008*.

Weir, D. J. (1992). A geometric hierarchy beyond context-free languages. *Theoretical Computer Science* (104), 235–261.

Yang, Y., J. Wang, and R. Kravets (2005). Designing routing metrics for mesh networks. *Proceedings of the IEEE Workshop on Wireless Mesh Networks (WiMesh). IEEE Press*.

Yannakakis, M. (1990). Graph-theoretic methods in database theory. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, New York, NY, USA, pp. 230–242. ACM Press.

Zwick, U. (2001). Exact and approximate distances in graphs - a survey. In *ESA 2001*, pp. 33–48.