HEURISTICS FOR GENERATING ADDITIVE SPANNERS

by

Michael J. Letourneau

B.Sc. (Honours), Brock University, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Michael J. Letourneau 2004
SIMON FRASER UNIVERSITY
August 2004

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Michael J. Letourneau

Degree: Master of Science

Title of thesis: Heuristics for Generating Additive Spanners

Examining Committee: Dr. Lou Hafer
Chair

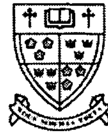
Dr. Arthur Liestman, Senior Supervisor

Dr. Thomas Shermer, Supervisor

Dr. Hovhannes Harutyunyan, External Examiner
Associate Professor of Computer Science,
Concordia University, Montreal, QC

Date Approved: August 5th, 2004

SIMON FRASER UNIVERSITY



Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Bennett Library
Simon Fraser University
Burnaby, BC, Canada

Abstract

Given an undirected and unweighted graph G, the subgraph S is an additive spanner of G with delay d if the distance between any two vertices in S is no more than d greater than their distance in G. It is known that the problem of finding additive spanners of arbitrary graphs for any fixed value of d with a minimum number of edges is NP-hard. Additive spanners are used as substructures for communication networks which are subject to design constraints such as minimizing the number of connections in the network, or permitting only a maximum number of connections at any one node.

In this thesis, we consider the problem of constructing good additive spanners. We say that a spanner is "good" if it contains few edges, but not necessarily a minimum number of them. We present several algorithms which, given a graph G and a delay parameter d as input, produce a graph S which is an additive spanner of G with delay d.

We evaluate each of these algorithms experimentally over a large set of input graphs, and for a series of delay values. We compare the spanners produced by each algorithm against each other, as well as against spanners produced by the best-known constructions for those graph classes with known additive spanner constructions. We highlight several algorithms which consistently produce spanners which are good with respect to the spanners produced by the other algorithms, and which are nearly as good as or, in some cases, better than the spanners produced by the constructions. Finally, we conclude with a discussion of future algorithmic approaches to the construction of additive spanners, as well as a list of possible applications for additive spanners beyond the realm of communication networks.

To Stanley "Uncle Willie" Wilson,
who knows a thing or two about experimentation.

spanner

2a. *A hand-tool, usually consisting of a small bar of steel, having an opening, grip, or jaw at the end which fits over or clasps the nut of a screw, a bolt, coupling, etc., and turns it or holds it in position; a wrench.*

2b. *Colloq. phr.* **to throw a spanner in the works** *and varr.: to cause disruption, to interfere with the smooth running of something.*

The Oxford English Dictionary
Second Edition, 1989

Acknowledgments

First and foremost, I would like to thank my Senior Supervisor, Dr. Art Liestman, for all his assistance, encouragement, and direction throughout my whole thesis process, as well as for all that pie. Can't forget the pie.

I would like to thank my Supervisor, Dr. Tom Shermer, for his guidance and numerous insights, and my Examiner, Dr. Hovhannes Harutyunyan of Concordia University, for agreeing to examine my thesis. I would also like to thank Pritam Ranjan and especially Matt Pratola of the Department of Statistics at SFU for their help with the statistical portions of this thesis.

I would like to thank Dr. Brian Ross, Dr. Tom Jenkyns, and especially Dr. Sheridan Houghten, all of Brock University, for their efforts in my undergraduate education. Without them, I would not have been inspired to undertake graduate studies.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for the generous financial support they have provided in the form of a Postgraduate Scholarship.

I would like to thank my wonderful friends for being so wonderful and for making so much of this experience fun. I only hope I have made their various experiences as much fun as they have made mine.

Last, but certainly not least, I would like to thank my parents for all the love, care, and encouragement they have provided me, and continue to provide me.

Contents

Approval ii

Abstract iii

Dedication iv

Quotation v

Acknowledgments vi

Contents vii

List of Tables x

List of Figures xi

List of Algorithms xiii

1 Introduction 1

2 Foundations 3

2.1 Definitions 4

2.2 Known Additive Spanner Constructions 6

2.3 Terminology 9

3 **Experimental Design** 11
3.1 Experiments 11
3.1.1 Primary Experiment 12
3.1.2 Secondary Experiments 12
3.2 Graphs 14
3.2.1 Structured Graphs 14
3.2.2 Random Graphs 14
3.2.3 Further Graphs 15
3.3 Baseline 16

4 **Algorithms** 17
4.1 On "Greedy" Algorithms 17
4.1.1 On Analysis 18
4.2 Algorithm Terminology 19
4.3 1-Stage Algorithms 20
4.3.1 Far Edge Deletion 20
4.3.2 Tainting 24
4.3.3 Neighbourhood Tainting 29
4.3.4 Popular Edges 32
4.3.5 Low Degree Pairs 37
4.4 2-Stage Algorithms 40
4.4.1 Spanning Tree Algorithms 42
4.4.2 Patching 46
4.4.3 Analysis 57
4.5 Decomposition Algorithms 57
4.5.1 Algorithmic Argument 57
4.5.2 Structural Argument 58
4.6 Baseline Algorithm 59

5 **Experimental Analysis** 60
5.1 Experiments 60

5.1.1 Test Graphs 61
5.1.2 Algorithms 63
5.1.3 Shuffling 63
5.2 Implementation 65
5.2.1 Evaluation System 65
5.2.2 Implementation Choices and Limitations 66
5.3 Primary Criterion 67
5.3.1 Competitive Rankings 68
5.3.2 Edge ratio rankings 76
5.3.3 Comparisons against known constructions 76
5.3.4 List of good heuristics 89
5.4 Secondary Criteria 90
5.4.1 Average Delay 90
5.4.2 Maximum Degree 99
5.4.3 Running Time 99
5.5 Summary 101

6 Conclusions and Future Work **106**
6.1 Conclusions 106
6.2 Future Work 107

Bibliography **110**

Appendix — Experimental Data **113**

List of Tables

5.1 Graphs used for experiments 63
5.2 Table of algorithms and their labels 64
5.3 "Top 10" fractions for all algorithms over all grids 69
5.4 "Top 10" fractions for all algorithms over all hypercubes 70
5.5 "Top 10" fractions for all algorithms over all X-trees 71
5.6 "Top 10" fractions for all algorithms over all pyramids 72
5.7 "Top 10" fractions for all algorithms over all random graphs 73
5.8 "Top 10" fractions for all algorithms over all graphs 75
5.9 Edge ratios for all algorithms over all grids. 77
5.10 Edge ratios for all algorithms over all hypercubes. 78
5.11 Edge ratios for all algorithms over all X-trees. 79
5.12 Edge ratios for all algorithms over all pyramids. 80
5.13 Edge ratios for all algorithms over all random graphs. 81
5.14 Edge ratios for all algorithms over all graphs. 82
5.15 "Good" heuristics according to the primary criterion 89
5.16 Average delay for the good algorithms over all hypercubes 90
5.17 Average delay for the good algorithms over all grids 92
5.18 Average delay for the good algorithms over all X-trees 92
5.19 Average delay for the good algorithms over all pyramids 92
5.20 Average delay for the good algorithms over all random graphs 93

List of Figures

2.1 Basic spanner types 5

2.2 Highways in a supergrid 8

2.3 Spanning an infinite grid with highways and tiles 8

4.1 Edge Distance 20

4.2 Far Edge Deletion 22

4.3 Tainting when $d_S(w,u) = d_S(w,v)$ 26

4.4 Tainting when $|d_S(w,u) - d_S(w,v)| = 1$ 26

4.5 Tainting when $1 < |d_S(w,u) = d_S(w,v)| \leq R$ 27

4.6 Neighbourhood Tainting 30

4.7 Low Degree Pair Selection 39

4.8 2-Stage Algorithm Process 41

4.9 Patching 48

4.10 Repairing 51

4.11 Distinct-tree patching 55

5.1 Experiment implementation structure. 67

5.2 Key to the plots. 83

5.3 Spanner edges vs. delay (parameter), for XT_9. 84

5.4 Spanner edges vs. delay (parameter), for P_5. 85

5.5 Spanner edges vs. delay (parameter), for Q_9. 86

5.6 Spanner edges vs. delay (parameter), for $G_{16,16}$. 87

5.7 Average delay vs. parameter delay for Q_9 91

5.8 Average delay vs. parameter delay for XT_7 94

5.9 Average delay vs. parameter delay for XT_9 95
5.10 Average delay vs. parameter delay for $R_{200}(0.10)$ 96
5.11 Average delay vs. parameter delay for $R_{200}(0.30)$ 97
5.12 Average delay vs. parameter delay for $R_{200}(0.50)$ 98
5.13 Maximum degree vs. parameter delay for $R_{200}(0.15)$ 100
5.14 Running Time vs. parameter delay for Q_8 102
5.15 Running Time vs. parameter delay for Q_9 103
5.16 Average Running Time vs. Instance Size ($|V|$) for the Hypercubes . . 104

List of Algorithms

1 Far Edge Deletion 23
2 Edge Removal with Tainting 29
3 Simple Neighbourhood Tainting 31
4 k-Neighbourhood Deletion 33
5 Popular Edge Selection 34
6 Partial Popularity Assignment 35
7 Low Degree Pairs 38
8 Breadth-First Search Spanning Tree 43
9 Depth-First Search Spanning Tree 44
10 Simple Spanning Tree 45
11 Popular Edges Spanning Tree 46
12 Simple Patching 47
13 Farthest Pair Patching 49
14 Farthest Edge Patching with Tie-Breaking 50
15 Simple Repairing (recursive component only) 52
16 Distinct Tree Construction 56
17 Random Edges 59

Chapter 1

Introduction

> The journey of a thousand miles
> begins with a cash advance.
> (colloquial)

In this thesis, we will consider the problem of generating good *additive graph spanners.* We will defer their formal definition until Chapter 2 and instead simply describe an additive spanner as a graph which models a more complicated graph in such a way that the distances between pairs vertices in the spanner and the distances between the same pairs in the original graph differ by at most a set amount. A spanner generally has fewer edges than the graph it models, but it still retains the general character of that graph.

Additive spanners were originally motivated by problems in the field of computer and communication networks. They are used in situations where a network having certain properties is desired, but where construction costs or hardware limitations prevent this network from being feasibly constructed. By constructing a network based on a spanner which resembles the desired network, instead of the desired network itself, one can mitigate the factors preventing the construction of the desired network at a known cost.

In this thesis, we take an *experimental* approach to evaluating algorithms for generating good spanners of arbitrary graphs. We will present an experimental framework which can be used to evaluate such algorithms. This framework includes a set of

criteria for determining if a spanner is "good", along with a set of graphs for which prospective algorithms can be used to generate additive spanners. We will also present a series of algorithms for generating good spanners and evaluate them extensively within this framework. Using this information, we find several algorithms which perform well in all tested cases, and which will likely produce good spanners in all cases. Finally, we conclude the thesis with a discussion of possible new spanner algorithms as well as possible uses for additive spanners, both in the realm of communication networks as well as beyond it.

Chapter 2

Foundations

Would the two of you stop being
amazed by the mathematics?!
C. J. Cregg, *The West Wing*
"The Leadership Breakfast"
Season 2, Episode 11

This chapter will cover the material which underlies the main work of this thesis. Previous work on spanners will be discussed, and some terminology will be defined.

All graphs in this thesis will be undirected, unweighted, simple graphs, and all graph-theoretic terminology will follow the style of West [25]. The naming convention $G_x = (V_x, E_x)$ will be used to denote that V_x is the vertex set of the graph G_x, and E_x is the edge set of G_x. The notation S_x will be used to denote a graph that is a spanner of G_x, and the subscripts will be omitted when the meaning of the symbols is clear from the context. The notation $d_x(u, v)$ denotes the distance between u and v in graph x. More terminology will be defined in Section 2.3.

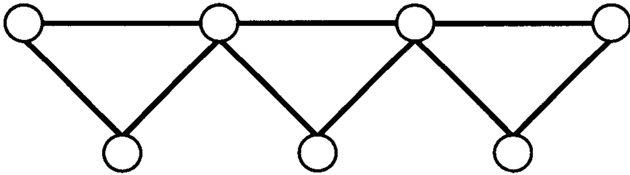
Given a graph G, S is a spanner of G if S is a spanning subgraph of G and the distance between any pair of vertices in S is bounded by some function of their distance in G. More formally, as defined by Liestman and Shermer [15], we say that a graph $G = (V, E)$ contains an $f(x)$-spanner $S = (V, E')$ if S is a spanning subgraph of G and $\forall u, v \in V, d_S(u, v) \leq f(d_G(u, v))$.

2.1 Definitions

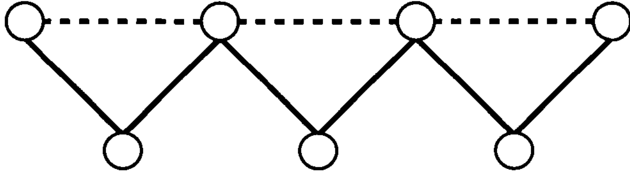
Under the original definition of spanners (due to Peleg and Schäffer [19]) the distance bound was of the form $f(x) \leq tx$. In particular, every edge (u, v) in G corresponded to a path from u to v in S of length at most t. Thus, the distance in the spanner between any two vertices u and v was no more than t times longer than the path in the original graph. Spanners of this form would come to be known as *multiplicative* spanners, or tx-spanners, where t is called the *stretch* of the spanner. Their multiplicative nature, however, implies that the distances between distant vertices in G can grow to be extremely large in the spanner. This is not desirable when such a spanner is used to model many types of communications/interconnection networks.

After proposing the more general definition of $f(x)$-spanners, Liestman and Shermer devised *additive* spanners which ensured that the distances between any pair of vertices in S would never be much greater than their distance in G. Additive spanners are spanners with a distance bound of the form $f(x) = d + x$; we call such a spanner a $(d + x)$-spanner, or simply an *additive d-spanner*. In the case of additive d-spanners, d is also called the *delay* of the spanner. Thus, the distance between any pair of vertices u, v in an additive d-spanner is no more than d greater than their distance in the original graph, regardless of whether or not there is an edge $(u, v) \in E_G$. While additive spanners do not suffer from the same "explosion" of distances found in multiplicative spanners, they are conceptually more complicated, since the delays between all vertex pairs must be bounded, not just the delays between pairs connected by edges in the original graph. Figure 2.1 illustrates the difference between multiplicative and additive spanners; dashed lines indicate edges present in the graph but not in the spanners.

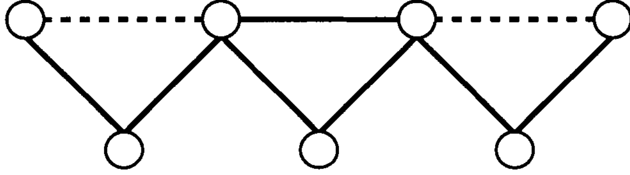
Other special cases of $f(x)$-spanners have been studied, such as (α, β)-spanners [7], where the distance bound is of the form $f(x) = \alpha x + \beta$, and tree spanners, where each spanner is also a spanning tree. The spanning tree condition imposes an additional constraint on the spanners which we do not consider here; the interested reader is invited to consult Kratsch *et al.* [12] for more details. In this thesis, we shall focus strictly on *additive* spanners.



(a) A graph G



(b) A $2x$-spanner of G (multiplicative)



(c) A $(2+x)$-spanner of G (additive)

Figure 2.1: Basic spanner types

2.2 Known Additive Spanner Constructions

The original concept of spanners (*i.e.* multiplicative spanners) is due to Peleg and Ullman [20], and was formalized by Peleg and Schäffer [19]. Multiplicative spanners have been studied by Liestman and Shermer[14, 16], Heydemann *et al* [9], and Richards and Liestman [22], as well as many others. Additive spanners have also been extensively studied by Liestman and Shermer [13, 15, 17] as well by Kratsch *et al* [12], Dor *et al* [6], Elkin and Peleg [7], Bollobás *et al* [1], and others. Liestman and Shermer presented additive spanner constructions for several popular topologies for networks and parallel computers: hypercubes, X-trees, pyramids, and grids. Kratsch *et al* [12] presented several classes of graphs which support additive tree-spanners (*i.e.* spanners which are trees,) namely distance-hereditary graphs, interval graphs, and asteroidal-triple free graphs. The work described here was restricted to considering graphs which have parameterized constructions.

A small amount of work has been done in constructing algorithms to build spanners. Farley *et al* [8] used a simple heuristic to generate tx-spanners of weighted graphs for use as multicast trees. They used their algorithm to create spanners of graphs which modeled the structure of internet domains, and then analyzed the multicast properties of those spanners. Although this work originally motivated this thesis, our experimental work differs significantly, since we are interested in comparing several spanner generating algorithms and since we are interested in generating $(d + x)$-spanners.

The constructions for hypercubes, X-trees, and pyramids are parameterized in such a way as to provide tradeoffs between vertex degree and spanner delay. Liestman and Shermer give two constructions for hypercubes. Each construction labels each vertex in the hypercube with a standard binary label where each vertex is connected to all other vertices that differ in exactly one dimension in their labels. In the spanner, certain edges are retained, depending on the labels of the vertices they join. By retaining only the edges occurring every j dimensions, one construction is able to attain a maximum delay of 2^j. The other construction uses the product of spanners from the first construction to form spanners of larger cubes with maximum delay

$2^j + 2$, but which uses roughly 1/2 the edges used by the 2^j construction as applied to that larger cube.

The constructions for X-trees and pyramids work in a similar fashion to those for the hypercube. They break the graph down into a hierarchy and then periodically retain edges at certain points in the hierarchy. The length of this period is variable, and longer periods result in fewer edges and higher delays in the spanners.

Liestman and Shermer's construction for grids differs somewhat from the other constructions, as it describes what they call *highway* spanners. In order to provide an example of their constructions, we will describe their construction for grids. Highway spanners consist of regularly-spaced rows and columns of vertices within which all grid edges are present; these rows and columns are called *highways.* These highways form what might be called a "super-grid" embedded in the actual grid. The rectangular regions between the highways are then filled-in with a single repeated pattern of edge connections, called a *tile.* Figure 2.2 illustrates the path between two vertices in a highway spanner.

The maximum delay of a highway spanner is defined by the increase in the distance between each pair of vertices in the spanner and their distance in the original graph. The distance between each vertex pair in the spanner depends on their relative positions within the super-grid; Liestman and Shermer showed that the maximum delay of a highway spanner of a grid is defined by the distances between the vertices in a tile and the vertices in the nine tiles centered on that tile. For example, if for an infinite grid G and its highway spanner S, for vertices u, v, w such that:

- $d_G(u, v) < d_G(u, w)$,
- v and w are in the same relative position in S with respect to their tiles, and
- the relative position of the tiles containing v and w with respect to the tile containing u are the same (e.g. v and w are "to the right" of u),

then the increase in distance between u and w in the spanner will be no greater than the increase in distance between u and v. Thus, by determining the spanner properties of the tile used to create a highway spanner of the infinite grid and considering only

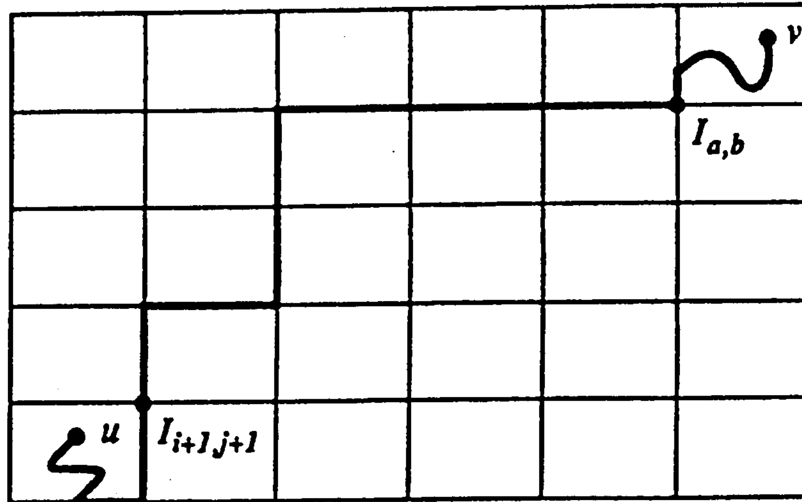


Figure 2.2: Highways in a supergrid

the relationships between any vertex in an arbitrary tile T and any vertex in the nine tiles centered on T, the maximum delay of the spanner can be determined. A spanner with that same maximum delay for an $m \times n$ finite grid can then be obtained by taking an $m \times n$ subgrid of the infinite grid's highway spanner and possibly adding some edges near the borders. Figure 2.3 illustrates a portion of such a spanner of an infinite grid; note that in Liestman and Shermer's terminology, a *prototile* is a tile that is repeated throughout the spanner.

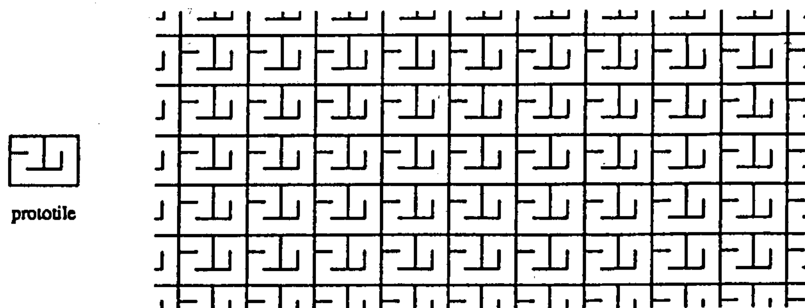


Figure 2.3: Spanning an infinite grid with highways and tiles

2.3 Terminology

Here we present definitions of terms which will be used throughout this thesis. First, we review some definitions from the preamble of this chapter, and then we add other definitions.

All graphs in this thesis will be undirected, unweighted, simple graphs. The naming convention $G_x = (V_x, E_x)$ will be used to denote that V_x is the vertex set of the graph G_x, and E_x is the edge set of G_x. The notation S_x will be used to denote a graph that is a spanner of G_x, and the subscripts will be omitted when the meaning of the symbols is clear from the context. The notation $d_G(u, v)$ denotes the distance between u and v in graph G, and the notation $d_G(v)$ indicates the degree of vertex v in graph G. $P_G(u, v)$ denotes a shortest path of vertices in G starting at u and ending at v.

Given a graph G, S is an $f(x)$-spanner of G if S is a spanning subgraph of G and if for all $u, v \in V$ where $x = d_G(u, v)$, $d_S(u, v) \leq f(x)$. A tx-spanner is a spanner where $f(x) = tx$, where $t \in \mathbf{Z}^+$; it is also called a *multiplicative* spanner with *stretch* t. A $(d + x)$-spanner is a spanner where $f(x) = x + d$, where $d \in \mathbf{Z}^+$; it is also called an *additive* spanner with *delay* d, or an additive d-spanner.

The *delay between any two vertices* u and v is defined as $d_S(u, v) - d_G(u, v)$. The *delay for a vertex* u is defined as $\max_{v \in V}[d_S(u, v) - d_G(u, v)]$; it is the largest delay to any other vertex from u. The *(maximum) delay for a spanner* S of a graph G is defined as $\max_{u,v \in V}[d_S(u, v) - d_G(u, v)]$, which is to say that it is the largest increase in distance between any pair of vertices in S over their distance in G.

In the descriptions of our algorithms for constructing spanners, we refer to the spanner S as it undergoes construction. Each algorithm is given an input graph G and a maximum delay parameter d and is expected to produce a graph S which is a spanner of G with delay $\leq d$. At times during the construction of the spanner there will be many vertex pairs with delay $> d$; we say that any such pair is *overdelayed*, and the algorithm must ensure that no pairs are overdelayed in the final spanner that it returns. The *overdelay* of any two vertices u, v is $\max\{d_S(u, v) - d_G(u, v) - d, 0\}$. An overdelay of zero indicates that u and v are not overdelayed.

Some algorithms presented here will deal with lists of vertices or edges. We extend our set notation to express this list concept; $A[i]$ indicates the ith element for the set A under some list ordering. The nature of that ordering will be given for each such reference.

Finally, we note that we use the terms *dense* and *sparse* to refer to the average degree of or number of edges in a graph. We do not use these terms in any particular technical sense, except to indicate that a graph with high average degree or more edges is *dense*, and one with low average degree or less edges is *sparse*.

Chapter 3

Experimental Design

All models are wrong. Some models are useful.

George Box

In order to determine which of the proposed heuristics are most suitable for generating good additive spanners of arbitrary graphs, a series of experiments were designed. Each experiment evaluates the spanners produced by the heuristics on the basis of some measurable feature of their structure. Some of these features relate to the performance of the algorithms themselves, while the rest of them relate to the qualities of the spanners produced as models of networks.

In this chapter, the experiments will be discussed, along with the graphs on which the algorithms were tested. A baseline for determining if an algorithm is worthy of future consideration will also be presented.

3.1 Experiments

We present several experiments here. Each experiment measures some feature of the spanner produced by a given algorithm on a given graph. In order to make the experimental analysis as significant as possible, we present one experiment which will be used as the primary criterion for determining "good" spanners. All other experiments

presented will be considered secondary and will be used to differentiate only among those spanners which are equally "good" according to the primary criterion.

3.1.1 Primary Experiment

Since, for an arbitrary graph and any integer delay $d > 0$, the problem of finding a $(d + x)$-spanner with a minimum number of edges is NP-hard [15], we feel that the number of edges in the spanner would be most appropriate measure for determining if the spanner is "good". Any heuristic consistently producing spanners with relatively few edges, as compared to the other heuristics and, where possible, to the previously known constructions, is considered to be "good". The primary goal of each heuristic presented here will be the production of a spanner with a minimum number of edges.

We do note that the primary motivation of Liestman and Shermer [15] in their additive spanner constructions was not to minimize the number of edges, but rather to first minimize the maximum degree, then to minimize the average degree and number of edges. We have chosen to use the number of edges as our primary criterion since it is the variable being optimized in the problem known to be NP-hard.

3.1.2 Secondary Experiments

Although we have chosen number of edges as our primary "goodness" criterion, it is not the only criterion which could be used to evaluate the spanners produced by our algorithms. If we use spanners to model "real-world" networks, then we may wish to evaluate qualities of the spanners which relate to issues of network performance. We may also wish to evaluate the performance of the algorithms themselves.

Since the primary goal of the heuristics given in this thesis is not to optimize these parameters, the heuristics will have their optimization as, at most, secondary goals.

Average Delay

Every non-trivial spanner induces a delay between some vertex pairs in the graph. The delay of an additive spanner is the maximum delay over all pairs of vertices in the

graph. That definition, however, does not speak to the average delay over all pairs of vertices. We define the average delay of a spanner S of a graph G as:

$$\overline{d_S} = \frac{\sum_{i,j \in V} d_S(i,j) - d_G(i,j)}{|V|^2}$$

For spanners modeling networks, the average delay will give some indication of the overall effectiveness of a spanner-generating heuristic. For example, suppose that two different heuristics H_1 and H_2, given the same input graph G and the same maximum delay d, produce, respectively, two different spanners S_1 and S_2 where $\overline{d_{S_1}} > \overline{d_{S_2}}$ and where $|E_{S_1}| \approx |E_{S_2}|$. Spanners produced by H_2 may be more desirable since they use roughly the same number of edges as those produced by H_1 but do a better job of preserving distances between vertices on average.

Maximum Degree

In real-world networks, equipment is subject to certain physical, design, and/or cost constraints. If one could disregard these real-world constraints, then all networks could directly connect all nodes to all other nodes, and thus be modeled as complete graphs. Under these constraints, hardware may only permit a certain maximum number of connections per node, and thus maximum degree over all vertices in a spanner will be an interesting parameter to observe[1]. In this case, a heuristic producing spanners with lower maximum degree will be preferable.

Running Time

Since efficient algorithms are preferable over inefficient ones, we will measure the running time of our algorithm over input graphs of various sizes, and for different delay parameter values. Using this information, we can attempt to extrapolate the algorithm's running time over larger input instances. Naturally, we consider lower running times to be preferable.

---

[1]We note that, in some cases (*i.e.* networks with high-connectivity hubs), it may be preferable to have a few vertices with high degree, and the rest with low degree.

3.2 Graphs

In order to determine the performance of the heuristics on graphs in general, we use them to produce spanners of several types of graphs. We consider two major groups of graphs in this thesis which represent what we feel are the two "opposite ends of the spectrum" of graph types.

3.2.1 Structured Graphs

Our structured graphs represent four classes of graphs for which we have parameterized constructions for producing additive spanners[2]: hypercubes, X-trees, pyramids, and multidimensional grids. The spanner constructions for these classes, due to Liestman and Shermer [13, 15], are believed to be "good" constructions, according to both our primary measure of goodness, the number of edges in the spanner, as well as according to our secondary measures of average delay and maximum degree. Comparisons between the spanners produced by these constructions and the spanners produced by the heuristics will determine if the heuristics are capable of producing spanners equivalent to (or better than) those produced by a mathematically good construction.

3.2.2 Random Graphs

These graphs represent the other extreme in the definition of graphs. While the structured graphs discussed previously are very well defined and represent networks with a highly regular topology, random graphs describe networks with a highly irregular topology.

There are many different random graph models for describing networks; a sampling of them can be found in Zegura *et al* [26] and Rajaraman [21]. It is beyond the scope of this thesis to consider heuristics on any significant fraction of the available models.

[2]We wish to note that we are considering only those classes of graphs with *explicit* constructions for additive spanners. Other constructs may exist which, incidentally, are also additive spanners, but we do not consider those constructs here.

This thesis will focus on a single class of random graphs, where there is a uniform probability p of an edge existing between any two vertices.

3.2.3 Further Graphs

There are many more types of structured graphs used to model networks (eg. rings, deBruijn graphs, Kautz (di)graphs, trees of rings, cube-connected-cycles, ...) than are considered in this thesis. Likewise, there are many more random graph models of real-world networks (eg. internet graphs [26] and small world graphs [24].) There are two reasons why the choices of graphs in this thesis are limited to the graphs described above.

First, we are more interested in the algorithmic aspects of developing additive spanners for general graphs than in developing algorithms for generating spanners for any particular class of graphs. The structured graphs we intend to use are those graphs for which there are known, parameterized constructions for additive spanners. Comparing the heuristically-generated spanners against the constructed spanners allows the heuristics to "compete", not only against each other, but also against mathematically verified constructions. The random graph model chosen for this thesis is the most random model available and is, possibly, the model most diametrically opposed to the given structured graphs.

Second, since we are concerned with experimental results, some consideration must be given to the quantity of experimental data to be managed. As the number of graph models increases, not only must more computation time be devoted to producing the results, but there is also the problem of having more experimental results to analyse.

While experimental data on other graph models may be of interest for understanding the properties of those graphs, that data will not provide significant insight into the algorithmic structure of the general problem of generating additive graph spanners. Considering other structured classes would not provide a comparison against spanners produced by known constructions; we consider the spanners produced by classes with known constructions to be good, and we already consider those classes with known parameterized constructions. Likewise, additional random graph classes

will have more structure in their graph models, and this structure will not be present in all graphs. Without including a whole set of random graph classes which, in total, provides a representative model of general graphs, additional experimental data would not give a complete picture of the performance of the heuristics on general graphs. Including such a set would produce a volume of data whose analysis is beyond the scope of this thesis.

3.3 Baseline

The experiments proposed above describe the characteristics which will determine if a generated spanner is good. If an algorithm produces consistently good spanners of the graphs in our set of "test" graphs, then it will be considered to be a likely candidate for producing good spanners on general graphs. In order to determine what may make an algorithm unworthy of future consideration, we propose that the spanners produced by each algorithm be compared against the spanners produced by a relatively unintelligent algorithm. Any heuristic which produces spanners which are no better than the spanners produced by this baseline algorithm should be rejected as not being suitable for producing good spanners of general graphs.

Chapter 4

Algorithms

> The "Richard Feynman
> Problem-Solving Algorithm":
> 1.) Write down the problem.
> 2.) Think very hard.
> 3.) Write down the answer.
>
> Murray Gell-mann

This chapter describes each of the algorithms studied in this thesis. Each of these algorithms is a form of "greedy" algorithm. First, a description of why the work was restricted to greedy algorithms will be presented, followed by the actual algorithms. Each of the algorithms will be classified according to its rough structure. A worst-case running-time analysis will be presented for each algorithm.

4.1 On "Greedy" Algorithms

One purpose of this thesis was to determine what basic techniques, if any, would be suitable for constructing good additive spanners of general graphs. Since most of the work to-date on developing additive spanners has focused on developing spanners for specific classes of graphs, there is very little information already available on such techniques. We feel that "greedy" algorithms—those which work by the repeated application of a rule which makes locally optimal choices in the hope that a globally

optimal solution will arise from those choices—would be a suitable class of relatively simple algorithms to consider. The operation of "greedy algorithms" is typically more straightforward than more complicated forms of algorithms, and they tend to be more amenable to analysis.

We by no means desire to imply that "greedy" algorithms are the best class of algorithms for constructing additive spanners of general graphs. There are many other algorithmic approaches to the problem which deserve study, and a small summary of them will be presented in Section 6.2.

4.1.1 On Analysis

Each algorithm will be presented along with a worst-case analysis of its running time. Since these algorithms are intended to be implemented, they will have average-case running times that will usually be significantly better than their worst-case times, however, average-case analyses will not be presented here. The reasons for this are twofold:

- Since these algorithms are intended for use on general graphs, their average case analysis will necessarily be quite complex. The analysis would make this thesis much more complicated without necessarily providing much more insight into the problem of additive spanners of general graphs.
- Since extensive experimental results will be presented in this thesis, a practical examination of the running time of these algorithms will provide what we expect will be a useful examination of their average-case running times.

We note that, in general, to determine if a graph S is a $(d+x)$-spanner of G, it is necessary to compare the distance between each vertex pair in S to the corresponding distance between their counterparts in G. One reasonably efficient way to do this is to run an APSP (All-Pairs Shortest Paths) algorithm on both S and G and to compare the lengths of the paths determined by those algorithms. The best known on-line, dynamic APSP algorithm requires $\Omega(|V|^2 \log |V|)$ time for an update [5], and the distance comparisons obviously require $\Omega(|V|^2)$ time. We consider any algorithm

which, for any single edge deletion, makes a decision in $< O(|V|^2 \log |V|)$ time to be reasonably efficient.

Throughout our algorithms, we use an APSP algorithm to perform various computations of overdelayed vertex pairs. For our analyses, we assume an efficient data structure. In our implementations, we use the well-known Floyd-Warshall algorithm, which returns a $|V| \times |V|$ matrix of distances between vertices. When determining overdelayed vertex pairs, we use the information in the distances matrix to produce a list containing the overdelayed pairs. This list is constructed by traversing the distance matrix in a standard row-major traversal, and thus this traversal ordering will influence the order of the pairs in the list.

4.2 Algorithm Terminology

Each of the algorithms to be presented has been classified in one of two general classes:

- *1-stage algorithm*
- *2-stage algorithm*

Each class is a rough categorization, and is intended only to improve the presentation of the algorithms. The definitions for these classes are not precise, and they are intended only to provide an indication of general similarities between the algorithms in them.

A third class of algorithms, which we call *decomposition algorithms*, was also considered. We will present observations on why such algorithms are not appropriate for evaluation in this thesis.

Finally, before we present the algorithms, we note that all of the algorithms take as input a graph G and a delay parameter d. They return a graph S which is a $(d+x)$-spanner of G. S is not guaranteed to be an optimal $d+x$-spanner in any sense of the word "optimal".

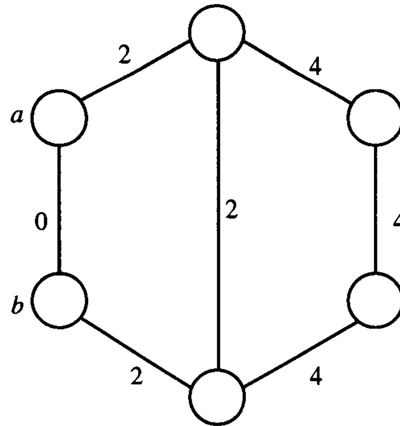


Figure 4.1: Edge Distance — the distances of every edge from edge (a, b) are listed beside that edge.

4.3 1-Stage Algorithms

Roughly speaking, a *1-stage algorithm* creates a spanner through the repeated application of a single rule or process to the graph. These represent the simplest algorithms to be considered here.

4.3.1 Far Edge Deletion

Let G be a reasonably dense graph. It seems unlikely that two edges that are "distant" from each other in G would be on a shortest path between some pair of vertices. In other words, the deletion of one of the edges will likely not have any effect on the deletability of the other.

Let $e = (u_e, v_e), f = (u_f, v_f) \in E$ and define the *(edge-)distance* between e and f as follows:

$$d_e(e, f) = \min\left[d(u_e, u_f) + d(v_e, v_f), d(u_e, v_f) + d(v_e, u_f)\right]$$

That is, the distance between two edges is the smaller of the sums of the shortest distances between each pair of vertices of the two edges. Figure 4.1 illustrates the concept of edge-distance.

We also define the concept of the *farthest* edge from some edge $e \in E$ to be:

$$\text{FarEdge}_E(e) = \arg\max_{e' \in E} d_e(e, e')$$

which is that e' has the greatest edge-distance from e of any edge in E.

We use this concept in the algorithm to identify possible edges to delete. After deleting some edge $e \in E_S$, we take $e' = \text{FarEdge}_S(e)$ and attempt to delete e'. We maintain a list of "deletable" edges, and we only delete edges on that list. Edges whose deletion cause the spanner to be overdelayed are restored to the spanner and are removed from the list of deletable edges. Figure 4.2 illustrates this process, and Algorithm 1 gives an implementation of it.

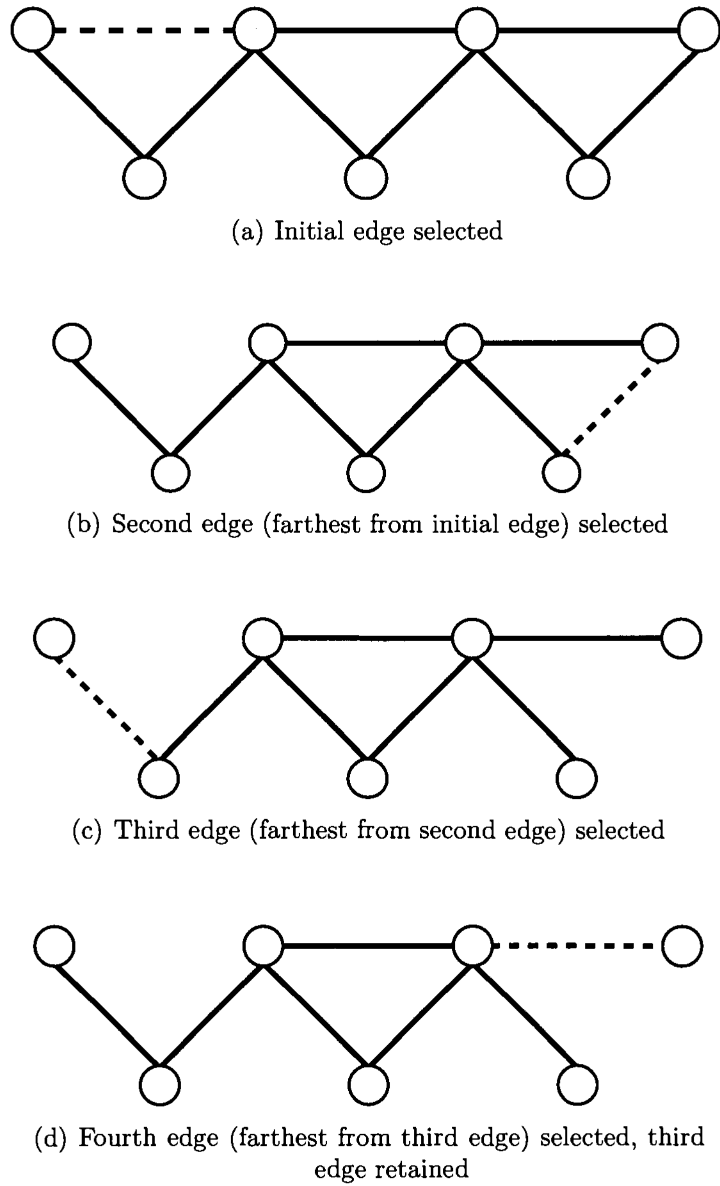
The algorithm maintains a set of *deletable* edges. Initially the spanner contains all edges from the original graph, and all edges are deletable. As the spanner is developed, certain edges will be identified as being necessary to maintain the acceptable delays within the spanner. Those edges are removed from the set of deletable edges so that they will not be deleted later.

Actual edge deletions occur within the *select-delete cycle.* Exactly one edge is deleted in each iteration of the select-delete cycle. In the first iteration, an arbitrarily-chosen edge is selected and deleted from the spanner. In subsequent iterations, the edge in the set of deletable edges which is farthest from the most recently deleted edge is selected and deleted from the spanner. This process repeats until there are no edges left in the set of deletable edges.

The algorithm has one control parameter: *count_max*, which is a constant. After *count_max* iterations of the select-delete cycle, a call is made to the APSP algorithm, and the delays between all vertices in S are determined. For each overdelayed pair v_1, v_2, a shortest path in G between v_1 and v_2 is determined, and all of the edges on that path which are not in the spanner are:

- Returned to the spanner.
- Removed from the set of deletable edges

A similar APSP/edge-restoration pass is done at the very end of the algorithm, in order to ensure that the final spanner meets its delay constraint.



(a) Initial edge selected

(b) Second edge (farthest from initial edge) selected

(c) Third edge (farthest from second edge) selected

(d) Fourth edge (farthest from third edge) selected, third edge retained

Figure 4.2: Far Edge Deletion — dashed line indicates candidate edge

Algorithm 1 Far Edge Deletion

Require: *count_max* = the number of edges deleted before APSP is run

$S \leftarrow G$
$e \leftarrow E_S[0]$
$E_S \leftarrow E_S \setminus e$
count $\leftarrow 0$
$E_R \leftarrow \emptyset$
loop
  if count = count_max **then**
    if S is overdelayed **then**
      while $\exists v_1, v_2 \in V$ such that v_1, v_2 are overdelayed **do**
        Find a shortest path P from v_1 to v_2
        for all $e_r \in P, e_r \notin E_S$ **do**
          $E_S \leftarrow E_S \cup \{e_r\}$
          $E_R \leftarrow E_R \cup \{e_r\}$
        end for
      end while
    end if
    count $\leftarrow 0$
  end if
  if $(E_S \setminus E_R) = \emptyset$ **then**
    break
  else
    $e \leftarrow \text{FarEdge}_{E_S \setminus E_R}(e)$
    $E_S \leftarrow E_S \setminus e$
    count $\leftarrow$ count + 1
  end if
end loop
if S is overdelayed **then**
  while $\exists v_1, v_2 \in V$ such that v_1, v_2 are overdelayed **do**
    Find a shortest path P from v_1 to v_2
    for all $e_r \in P, e_r \notin E_S$ **do**
      $E_S \leftarrow E_S \cup \{e_r\}$
      $E_R \leftarrow E_R \cup \{e_r\}$
    end for
  end while
end if

A simple analysis of this algorithm reveals that its running time is dominated by the APSP algorithm. In the worst case, since APSP is run every *count_max* edge deletions, and since there are $|E|$ edges, it follows that it requires time $\frac{|E|}{count_max} \times O(|V|^3) = O(|E||V|^3)$. In practice, this algorithm's running time is similar to the other algorithms' running times when $count_max \approx 25$.

We observe that the process of restoring edges to the spanner to correct overdelays may appear to require more running time than the APSP algorithm. Specifically, if there are $O(|V|^2)$ overdelayed vertex pairs, then for each vertex pair, it will be necessary to determine a shortest path in G between the vertices. Normally, finding a shortest path requires $O(|V|^2)$ steps in the worst case, and so it would appear that this process requires $O(|V|^2) \times O(|V|^2) = O(|V|^4)$ steps. We note, however, that since G is invariant, we can precompute all shortest paths in G at the start of the algorithm with a single APSP call. We can simply perform a $O(|V|)$ step lookup to determine this path when it is needed. Thus, this edge restoration phase requires at most $O(|V|^3)$ steps.

4.3.2 Tainting

We now introduce *tainting*[1] as a process of estimating the "area of effect" of removing an edge. More precisely, it involves estimating the differences in the distances between all vertex pairs in a graph without an edge e with respect to the same graph with the edge e.

The motivation for tainting comes from the observation that the most efficient means known of determining the exact effect of an edge deletion (*i.e.* the change in the distances between all pairs of vertices) in an arbitrary graph requires $\Omega(|V|^2)$ time, and currently is not easy to implement efficiently [5]. The "tainting" process is an attempt to produce an estimate of the delay added to all vertex pairs by the deletion of an edge. We desired that it should be easy to implement, and should run

[1] The term "tainting" was taken from the fact that in this algorithm some vertices in the spanner may be incorrectly assigned a delay after an edge deletion; we say that such vertices are "tainted" by the deletion.

in time at most $O(|V|^2)$.

Consider a graph G and an edge $e = (u, v) \in E_G$. Let $S = (V, E \setminus \{e\})$, that is, S is G without edge e. Within G and S, consider any vertex pair $w, x \in V$ for which $d_G(w, x) \neq d_S(w, x)$. Since e is the only edge in G which is not in S, it follows that the shortest path from w to x in G must include e, and without loss of generality, we can assume that $P = w \ldots uv \ldots x$. Further, there must be no other path $P' \neq P$ from w to x in G which is of length $d_G(w, x)$ and does not include e, since if there were, then $d_G(w, x) = d_S(w, x)$. Finally, we note that we use $l(P)$ to denote the length of a path P, where the length of the path is the number of edges in it.

Now, for any vertex $w \in V$, we consider the relationship between $d_S(w, u)$ and $d_S(w, v)$. Note that the path from u to v that e makes possible in G is replaced in S by a path of length $d_S(u, v)$. For future reference let $R = d_S(u, v) - 1$ (which is the delay which exists between u and v in S without e,) and let the shortest path from u to v in S be $ur_1r_2 \ldots r_{R-1}v$, where $r_1, r_2, \ldots r_{R-1} \in V$.

Suppose that $d_S(w, u) = d_S(w, v)$. Figure 4.3 illustrates this case. This implies that e cannot have been included in any shortest path to/from w in G, otherwise that path would contain a subpath of the form $w \ldots uv$ or of the form $w \ldots vu$, and that subpath could be replaced with an equivalent shorter subpath of the form $w \ldots v$ or $w \ldots u$, respectively. Thus, in this case, since e is not included in any shortest path to/from w, the distance from w to any other vertex will not change due to the deletion of e.

Suppose that $|d_S(w, u) - d_S(w, v)| = 1$; this implies that any shortest path in G to/from w containing e can be replaced by an equivalent shortest path in S not containing e. Figure 4.4 illustrates this case. If the path contained a subpath of the form $w \ldots uv$, it could be replaced with an equivalent subpath of the form $w \ldots v$, not including e, which was the same length as the original subpath containing e. Likewise, a similar case holds for paths with subpaths of the form $w \ldots vu$. Thus, in this case, since any shortest path to/from w containing e can be replaced with a path of the same length not containing e, the distance from w to any other vertex will not change due to the deletion of e.

Suppose that $1 < |d_S(w, u) - d_S(w, v)| \leq R$; this implies that any shortest path in

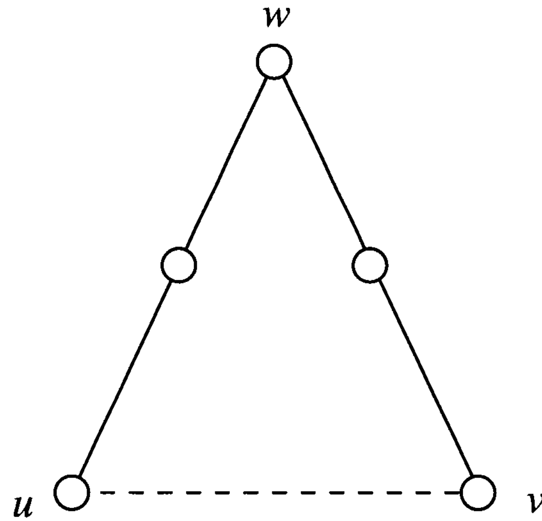


Figure 4.3: Tainting when $d_S(w,u) = d_S(w,v)$

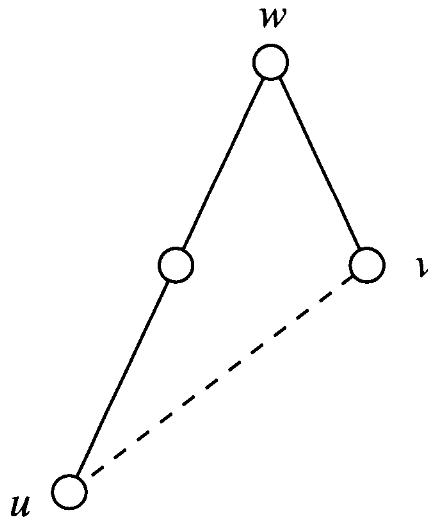


Figure 4.4: Tainting when $|d_S(w,u) - d_S(w,v)| = 1$

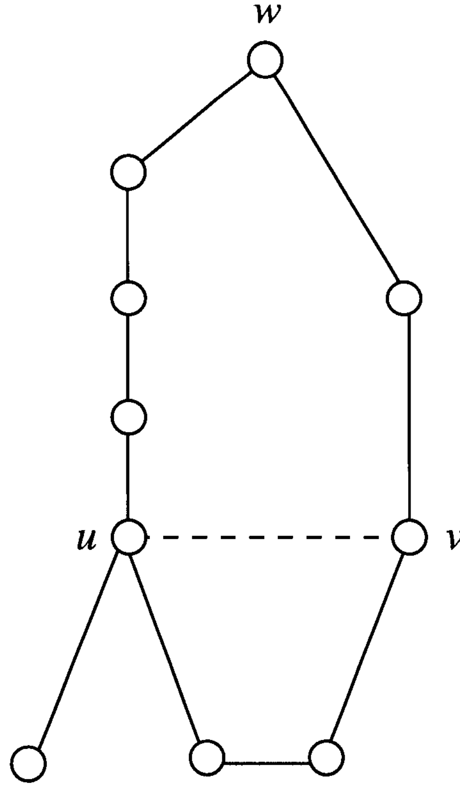


Figure 4.5: Tainting when $1 < |d_S(w,u) = d_S(w,v)| \leq R$

G to/from w containing e can be replaced by a path from $w \in S$ containing exactly $|d_S(w,u) - d_S(w,v)| - 1$ more edges. Figure 4.5 illustrates this case. We assume, without loss of generality, that $d_S(w,u) < d_S(w,v)$. Any such shortest path in G would contain a subpath of the form $P = w \ldots uv$, where $l(w \ldots u) = d_G(w,u) = d_S(w,u)$, and thus $l(P) = d_G(w,u) + 1$. Such a subpath in G can be replaced with a subpath in S of the form $P' = w \ldots v$ not containing e, where $l(P') = d_S(w,v)$. Comparing the differences in the two subpaths reveals that $l(P') - l(P) = d_S(w,v) - (d_S(w,u)+1) = d_S(w,v) - d_S(w,u) - 1$. A similar argument establishes the difference in distance for the case where $d_S(w,u) > d_S(w,v)$. Combining those two cases gives the result that any shortest path to/from $w \in S$ will be at most $|d_S(w,u) - d_S(w,v)| - 1$ longer than the path it replaces in G.

Finally, we note that $|d_S(w,u) - d_S(w,v)| > R$ cannot be the case. Since $R = d_S(u,v) - 1$, there will always be a subpath of the form $w \ldots ur_1r_2 \ldots r_{R-1} \ldots v$ which

simply follows the shortest path from u to v in S that replaces the edge (u, v) in G. Thus, no vertex pair will experience a delay of more than R for any edge deletion.

We summarize the above observation by formally defining the *taint* of a vertex:

Observation 1 *For any graph $G = (V, E)$, for any edge $e = (u, v) \in E$, in a spanner $S = (V, E \setminus \{e\})$, for any $w \in V$,*

$$taint(w) = \begin{cases} |d_S(w,u) - d_S(w,v)| & \textit{if } 1 < |d_S(w,u) - d_S(w,v)| \leq d_S(u,v) - 1 \\ 0 & \textit{otherwise} \end{cases}$$

and where $taint(w) \geq \max_{x \in V}[d_S(w,x) - d_G(w,x)]$.

Using this observation, we can devise an algorithm which works by estimating the delays produced by deleting edges from a spanner. This algorithm works by keeping track of the taints for each vertex as each edge is deleted. An accumulated taint count is kept for each vertex, and whenever an edge e is deleted, each vertex's accumulated taint count will be incremented with the taint caused by deleting e. If the deletion of an edge will cause any vertex's accumulated taint count to exceed d, then that edge will not be deleted. At a suitable point in the operation of the algorithm, an APSP algorithm will be run, and the accumulated taints for each vertex will be set to the actual delays for those vertices.

To compute the actual taints at any step in the algorithm, we label each vertex w in S with both $d_S(w, u)$ and $d_S(w, v)$; this can be accomplished simply by constructing two spanning trees in S by breadth-first search, one each rooted at u and v. For each vertex w, we can then determine $taint(w)$ by subtracting $d_S(w, v)$ from $d_S(w, u)$.

In the algorithm itself, the edge list is traversed in the order the list's data structure presents it. Initially, all vertices have an accumulated taint of 0. Each edge e will be deleted iff there is no vertex whose accumulated taint will exceed d after e is deleted. Once the edge list has been completely traversed, the APSP algorithm will be run and the taints for each vertex will be set to their respective delays. Traversals of the edge list will continue until a complete pass is made where no edges are deleted. If no edges are deleted, then the algorithm terminates. Algorithm 2 presents this tainting algorithm.

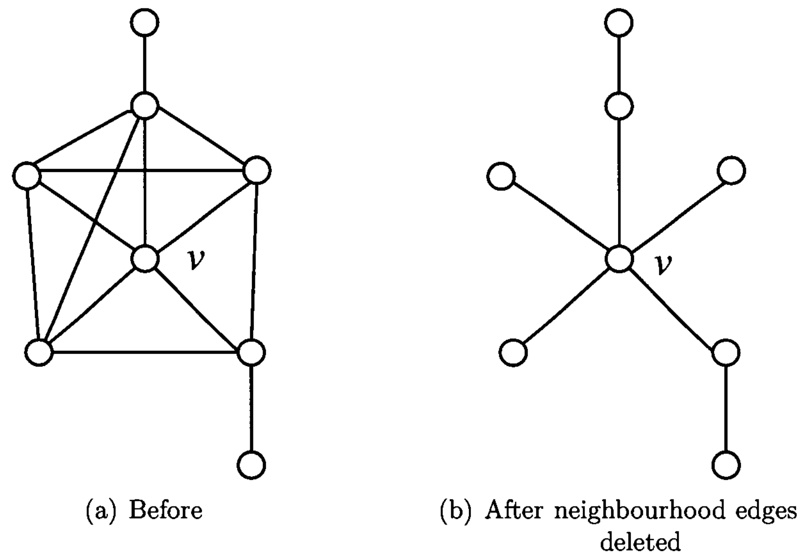
Algorithm 2 Edge Removal with Tainting

```
deleted ← true
Run APSP on G
while deleted = true do
  deleted ← false
  Run APSP on S; For all v ∈ V, set taint for v ← max_{u∈V} d_S(v,u) − d_G(v,u)
  for e ∈ E_S do
    Determine taints if e is deleted
    if e can be deleted without having any vertex taint exceed d then
      Delete e
      Update taints.
      deleted ← true
    end if
  end for
end while
```

A worst-case analysis reveals that the worst case would occur if exactly one edge could be deleted in each pass of the outer while-loop. In such a case, the algorithm would consider the tainting resulting from the deletion of each edge remaining in the list in each pass. Overall, this will require considering the tainting of $O(|E|^2)$ edges. For each tainting, two spanning trees are built, requiring $O(|E|)$ steps. Multiplying the two stages together, we see that the worst-case requires $O(|E|^3)$ steps. In practice, however, the algorithm runs much faster, since many edges can be deleted in each pass through the edge-list.

4.3.3 Neighbourhood Tainting

For any vertex v, consider $N(v)$, the (open) neighbourhood of v. We can try to produce a spanner by deleting any edge between vertices in $N(v)$. Consider the deletion of an edge $e_v = (w, x) \in E$, where $w, x \in N(v)$. In this case, since $w, x \in N(v)$, any shortest path in G including e_v must have a subpath of the form wx, and this subpath can be replaced with a subpath of the form wvx, which will increase the length of the path by at most 1. Further, we can repeat this process for any number of edges in $N(v)$, since for any such edge there will always be a length 2 replacement path



(a) Before

(b) After neighbourhood edges deleted

Figure 4.6: Neighbourhood Tainting

involving v. Figure 4.6 illustrates this idea.

Simple Approach

Using this observation, we create a candidate selection process which, along with the tainting process described in Section 4.3.2, will form a spanner-construction algorithm. In this algorithm, we select high degree vertices and attempt to delete edges within their neighbourhoods. Algorithm 3 describes the algorithm in more detail.

This algorithm maintains the same accumulated taint counts as the previous algorithm, and only deletes edges if their deletion will not result in any accumulated taint count exceeding d. It begins by considering vertices with the maximum degree, and proceeds to consider vertices of consecutively smaller degree, down to a minimum of 2. For a given vertex degree value, the algorithm locates all vertices of that degree. For each such vertex, the algorithm attempts to delete any edge with both vertices in the neighbourhood of that vertex. This deletion-by-tainting process is carried out for all vertices of the specified degree. Once all vertices of the specified degree have had their neighbourhoods processed in this fashion, the algorithm resets the taints using an APSP pass, and then another round of deletions are attempted for vertices with

Algorithm 3 Simple Neighbourhood Tainting

```
del ← 1
Run APSP on G
for deg ← max_{v∈V} d(v) to 2 do
  loop
    del ← 0
    for v ∈ V, d(v) = deg do
      T ← G[N(v)]
      for e ∈ E_T do
        Determine taints if e is deleted from S
        if e can be deleted from S without having any vertex taint exceed d then
          Delete e from S
          Update taints in S
          del ← 1
        end if
      end for
    end for
    Run APSP on S; Set taint for v ← max_{u∈V} (d_S(v, u) − d_G(v, u))
    if del = 0 then
      Exit loop
    end if
  end loop
end for
```

that degree. (We observe that while the deletion process had considered vertices of that degree in the previous pass, the tainting process may have incorrectly marked some edges in the neighbourhoods as undeletable.) If no deletions occur for that degree value, then the degree value is decremented by 1, and the process repeats, unless the new degree value is 1, in which case the algorithm terminates.

In the worst case, this algorithm requires $O(|E|^2|V|)$ steps. The outermost loop requires at most $|V|-2$ iterations, since $|V|-1$ is the maximum degree of any vertex in the input graph. The middle loop, which repeatedly looks for vertices with the specified degree, will be run at most once per edge, so it requires $O(|E|)$ iterations. Finally, each deletion candidate edge requires $O(|E|)$ steps to build the spanning trees. Thus, the algorithm requires $O(|V|-2) \times O(|E|) \times O(|E|) = O(|E|^2|V|)$ steps.

Extended Approach

We can extend the previous reasoning to other edges if we expand the distance from v to the vertices in its neighbourhood. Let $N_k(v)$ be the k-neighbourhood of a vertex v and define it as follows: $N_k(v) = \{v' \in V | d(v, v') = k\}$. Consider the construction of a spanner S by the deletion of any edge $e = (w, x)$ from a graph G where $w, x \in N_k(v)$. In this case, any shortest path in G containing e must contain a subpath wx, and that subpath may be replaced with a subpath of the form $u \ldots v \ldots x$. In this case, the length of the path will increase by at most $2k-1$. Algorithm 4 illustrates an extension of Algorithm 3 which considers k-neighbourhoods for $k = 1, 2, \ldots, \lceil d/2 \rceil$.

The analysis of this algorithm extends easily from the analysis of the simple neighbourhood tainting algorithm. Since this algorithm "wraps" that algorithm in a loop which considers all neighbourhood sizes from 1 to $\lceil d/2 \rceil$, it follows that this algorithm requires at most $O(|E|^2|V|) \times O(\lceil d/2 \rceil) = O(d|E|^2|V|)$ steps.

4.3.4 Popular Edges

One simple observation which one can make of creating a spanner can be expressed as "keep those edges that you need, discard those edges that you don't." While this may be regarded as a truism, we may use it to inform some of our heuristics. Consider the

---

Algorithm 4 k-Neighbourhood Deletion

```
del ← 1
Run APSP on G
for k ← 1 to k = ⌈d/2⌉ do
  for deg ← max_{v∈V} d(v) to 0 do
    loop
      del ← 0
      for v ∈ V, d(v) = deg do
        T ← G[N_k(v)]
        for e ∈ E_T do
          Determine taints if e is deleted from S
          if e can be deleted from S without having any vertex taint exceed d
          then
            Delete e from S
            Update taints in S
            del ← 1
          end if
        end for
      end for
      Run APSP on S; Set taint for v ← max_{u∈V} (d_S(v,u) − d_G(v,u))
      if del = 0 then
        Exit loop
      end if
    end loop
  end for
end for
```

---

following:

Between each vertex pair $u, v \in V$, there exists at least one, but possibly more than one, shortest path. If there exist more than one shortest path between u and v, then it follows that if all of the edges of some shortest path P are in a spanner S then, with respect only to u and v, no edge on any of those other paths need be in S.

Of course, the argument above fails to account for the importance of those other edges to paths not between u and v, but we can extend it to encode the "popularity" of all edges with respect to all the potential shortest paths in a graph. If, for each vertex pair $u, v, \in V$, we select one shortest path between u and v, then we can assign a popularity measure to each edge $e \in E$ as follows:

$$pop(e) = \frac{p(e)}{P(e)}$$

where $p(e)$ is the number of selected shortest paths in G which include e, and $P(e)$ is the total number of shortest paths in G which include e.

Once each edge is assigned a popularity, a spanner can be constructed by a simple algorithm which adds edges to the spanner in order of decreasing popularity until the spanner meets its delay constraint. Algorithm 5 illustrates the overall spanner-creation algorithm.

Algorithm 5 Popular Edge Selection

```
Assign popularities to all e ∈ E_G
E_S ← ∅
while S is overdelayed do
  Add (next) most popular edge to E_S
end while
```

While the spanner creation algorithm is relatively simple, there are serious drawbacks to the popularity assignment method described above. For example, in a highly regular graph such as a hypercube or a grid, there may be numerous shortest paths between any two vertices and enumerating the shortest paths in which every edge occurs for such a graph would be far too time-consuming. To mitigate this in our practical algorithm, we consider only a subset of all of the possible paths on which a given edge lies. Algorithm 6 illustrates how these popularity values are assigned.

Algorithm 6 Partial Popularity Assignment

Run APSP on G, storing distances in $dist_{(i,j)}$
for $i, j \in V$ **do**
  $paths_{(i,j)} \leftarrow 0$
  $chosen_{(i,j)} \leftarrow 0$
end for
for $e = (i, j) \in E_G$ **do**
  $paths_{(i,j)} \leftarrow 1$
  $chosen_{(i,j)} \leftarrow 1$
end for
for $dist = 0$ to $dist = \max_{i,j \in V} dist_{i,j}$ **do**
  for $i, j \in V, d_G(i, j) = dist$ **do**
    $best \leftarrow 0$
    for $k \in V, (k, j) \in E_G$ **do**
      if $(chosen_{(i,k)} + chosen_{(k,j)})/(paths_{(i,k)} + paths_{(k,j)}) > best$ **then**
        $best = (chosen_{(i,k)} + chosen_{(k,j)})/(paths_{(i,k)} + paths_{(k,j)})$
        $l \leftarrow k$
      end if
      for $e = (u, v) \in P_G(i, k)$ **do**
        $paths_{(u,v)} \leftarrow paths_{(u,v)} + 1$
      end for
      $paths_{(k,j)} \leftarrow paths_{(k,j)} + 1$
    end for
    for $e = (u, v) \in P_G(i, l)$ **do**
      $chosen_{(u,v)} \leftarrow chosen_{(u,v)} + 1$
    end for
    $chosen_{(l,j)} \leftarrow chosen_{(l,j)} + 1$
  end for
end for

This algorithm assigns popularities to edges by considering paths between all vertex pairs according to increasing path length. Since paths of length 1 (*e.g.* edges) will always be the shortest path between their vertices, their edges are "seeded" with an initial score of $1/1 = 1$ (*e.g.* one path using that edge / one path potentially using that edge.) The scores for longer paths will be built incrementally, starting with these scores for the shorter paths.

In order to compute the scores for the paths of length l, we let:

- $c(k,j)$ be the number of paths actually using edge (k,j),
- $C(i,k)$ be the number of paths of length $< l$ actually using the chosen path for $i \ldots k$,
- $p(k,j)$ be the number of paths potentially using edge (k,j), and,
- $P(i,k)$ be the number of paths of length $< l$ potentially using the chosen path for $i \ldots k$.

For $l = 2, 3, \ldots$, we consider all $i, j \in V$ where $d_G(i,j) = l$ and let $\lambda_{i,j} = \{k | d_G(i,k) = l-1, d_G(k,j) = 1\}$. Thus, $\lambda_{i,j}$ is a set of all vertices next-to-last on the shortest paths from i to j. For each $k \in \lambda_{i,j}$, a score is calculated as:

$$Score_k(i,j) = \frac{C(i,k) + c(k,j)}{P(i,k) + p(k,j)}$$

This score represents the popularity score of the path from i to k, via the most popular path from i to k, plus the popularity score of the edge from k to j. The highest scoring path over all k will be chosen as the path from i to j. All edges on the chosen path will have their "paths using" count incremented, all edges on all paths from i to j through each $k \in \lambda_{i,j}$ will have their "potential paths using" count incremented, and the sum of both of those counts for the chosen path will be stored as the counts for the path from i to j.

We consider the worst-case analysis of this algorithm in two parts, since it operates in two parts: popularity assignment followed by edge selection. Assignment of probabilities begins by seeding the counts for the paths corresponding to edges in

E_G, an operation requiring $O(|E|)$ steps. For each vertex pair i, j where $d_g(i,j) > 1$, $|\lambda_{i,j}| \leq |V| - 2$, so at most $|V| - 2$ paths from i to j will be considered, and each path requires following as many as $|V| - 1$ edges. There are $O(|V|^2)$ i, j pairings, and for each of those it may be necessary to consider $O(|V|)$ paths of $O(|V|)$ edges each, so at most $O(|V|^2 \times |V| \times |V|) = O(|V|^4)$ steps would be required to assign the probabilities.

To find a spanner by incrementally taking edges in order of decreasing popularity, we can sort the edges based on decreasing popularity and then perform a binary search on the edge list. For each value i probed in the binary search, we can construct the graph containing edges $0, 1, \ldots, i$, run APSP on that graph, and then evaluate the results to see if any vertex pair is overdelayed. The binary search will determine the smallest value of i such that the graph containing edges $0, 1, \ldots, i$ is a $(d+x)$-spanner of G, and that graph will be returned. This search will require $O(\log |E|)$ probes, each probe requiring an APSP requiring $O(|V|^3)$ steps, and then $O(|V|^2)$ steps to evaluate the results. Thus, in total, it will require $O(|V|^3 \log |E|)$ steps.

Since the $O(|V|^3 \log |E|)$ steps in the search phase is dominated by the $O(|V|^4)$ steps required to assign the probabilities, the algorithm will require $O(|V|^4)$ steps in the worst case. This worst case will require that the probability assignment phase be degenerate, and on average, the binary search phase will likely dominate the algorithm. Since the binary-search portion of the algorithm will always require $O(|V|^3 \log |E|)$ steps, the overall algorithm will require at least that many steps in any case. In practice, this algorithm runs in approximately the same time as the algorithms presented in Sections 4.4, 4.3.1, and 4.3.2.

4.3.5 Low Degree Pairs

The *low degree pairs* algorithm attempts to find a spanner with as few edges as possible along with a low maximum vertex degree. This algorithm provides a tradeoff between the number of edges in the spanner and its maximum degree. In the algorithm, edges where the sum of the degrees of its endpoints are low are added to the spanner until it meets its specified delay.

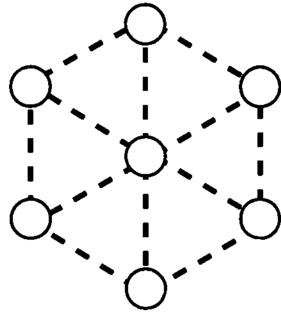
The algorithm initially has an empty spanner and begins by constructing a list of all edges in the graph. The algorithm adds edges to the graph based on their *degree sum*, which is $d_S(i) + d_S(j)$ for all edges $(i, j) \in E_V$. Initially, the algorithm sets its "current degree sum" to 0. It then traverses the edge list and adds any edge in the list with an appropriate degree sum. Vertex degree counts are maintained after each edge is added, so any edge's degree sum will be accurate at the time that edge is considered for addition. Once the edge list has been fully traversed, the "current degree sum" is increased by 1, and another traversal is begun. This pattern repeats until the spanner meets its maximum delay constraint. Figure 4.7 illustrates some steps in this process; Algorithm 7 implements the basics of the algorithm.

Algorithm 7 Low Degree Pairs

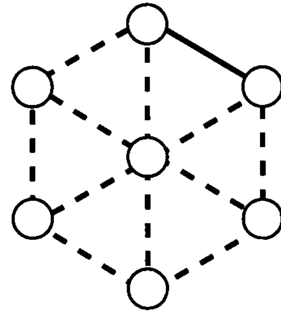
```
c ← 0
E_S ← ∅
while S is not a (d + x)-spanner of G do
  for e = (i, j) ∈ E_G, e ∉ E_S do
    if d_S(i) + d_S(j) = c then
      E_S ← E_S ∪ {e}
    end if
    if S is a (d + x)-spanner of G then
      Exit for loop.
    end if
    c ← c + 1
  end for
end while
```

In order to efficently implement the algorithm, we can use the same strategy we employed in the Popular Edges algorithm described in Section 4.3.4. In that strategy, we decompose the algorithm into two parts. The first part assigns an ordering to the edges based upon the order they are added to the spanner. This ordered list of edges can then be tested using binary-search to determine the smallest contiguous set of edges, starting at the start of the list, which comprise a $(d + x)$-spanner of G. We will use this method to consider the worst-case running time of the algorithm.

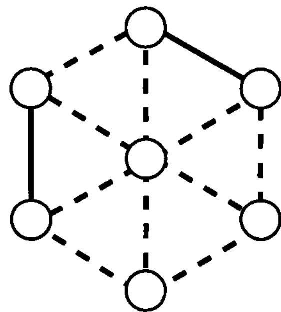
To assign the ordering to the edges, we can start with an edge e_0, assign it to position 0 in the ordering, and then search the edge list for e_1, where e_1 is the farthest



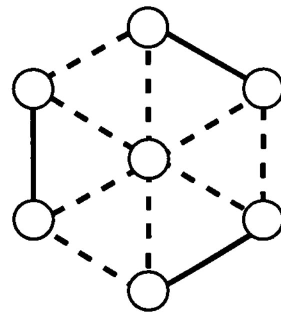
(a) Initially, no edges selected



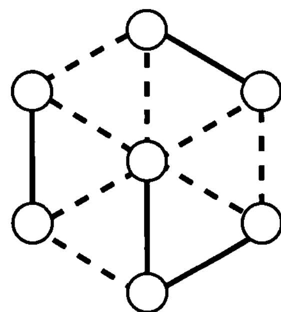
(b) First edge (degree sum = 0) selected



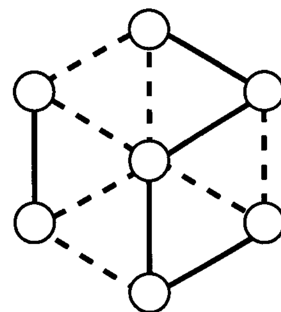
(c) Second edge (degree sum = 0) selected



(d) Third edge (degree sum = 0) selected



(e) Fourth edge (degree sum = 1) selected



(f) Fifth edge (degree sum = 2) selected

Figure 4.7: Low Degree Pair Selection

edge from 0. We can then assign e_1 to position 1 in the ordering and search for e_2. (Obviously, we ignore edges already assigned in the ordering). This process can then be repeated until all edges are assigned. Since there are $|E|$ edges, and each edge (except the last) requires a traversal of the edge list to find the farthest, we have $|E| \times O(|E|) = O(|E|^2)$ steps to assign the ordering to the edge list.

As in the analysis of the Popular Edges algorithm, performing a binary search on the edge list requires $O(|V|^3 \log |E|)$ steps. In the worst case, $|E| \simeq |V|^2$ and so, since $O(|E|^2)$ dominates $O(|V|^3 \log |E|)$ for $|E| \simeq |V|^2$, this algorithm requires at most $O(|E|^2)$ steps.

4.4 2-Stage Algorithms

A 2-stage algorithm is, roughly speaking, an algorithm which attempts to create a spanner by:

1. finding a spanning tree of the graph to serve as the initial spanner graph, then

2. adding edges to bring the delays between all vertex pairs in the spanner to within the given maximum delay.

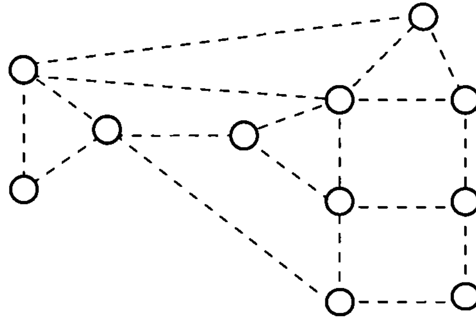
Figure 4.8 illustrates this process.

This structure naturally lends itself to a form of "modular" construction where each algorithm consists of:

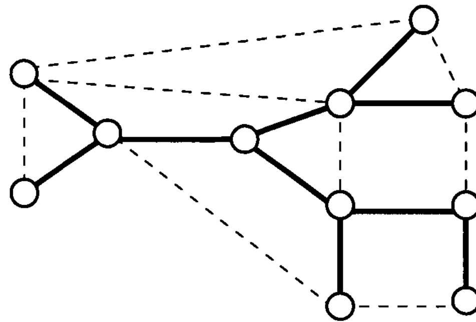
1. a spanning tree algorithm, followed by

2. an algorithm for choosing the remaining edges, which we call a *patching* algorithm.

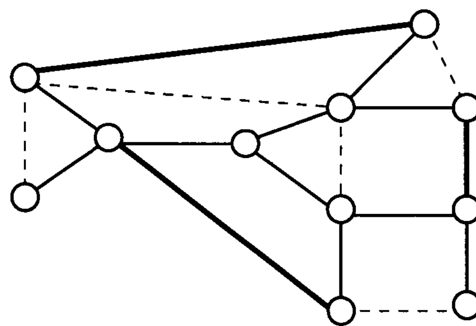
We will describe independently our four spanning tree algorithms, followed by our six patching algorithms. Each spanning tree algorithm will be paired with each patching algorithm to create a total of twenty-four candidate algorithms.



(a) Empty spanner, before spanning tree.



(b) Partially complete spanner, after spanning tree, before patching.



(c) Complete spanner, after patching.

Figure 4.8: 2-Stage Algorithm Process

4.4.1 Spanning Tree Algorithms

Since the graphs we are considering are unweighted, minimum spanning tree algorithms are of little use to us. Conventional literature gives short shrift to unweighted spanning tree algorithms; the most common approach is to choose a root and then add edges according to a straightforward breadth-first search. We considered several spanning tree algorithms, including breadth-first search, in order to determine if there are any which are clearly preferable or, equally, if there are any which are highly unsuitable.

Breadth-First Search

The first spanning-tree algorithm we consider is a straightforward breadth-first search. Let r be the chosen root vertex of the tree. We add all edges incident on r to the tree and then recursively repeat this addition procedure for every vertex just added to the tree. Algorithm 8 implements our breadth-first search spanning tree algorithm.

In our implementation of this algorithm, the root is chosen to be a vertex with maximum degree. When attempting to find a spanner with a minimum number of edges, this is likely going to be a good choice, since the high-degree vertex will provide a length 2 path between any vertices adjacent to it. This path will provide connections with a delay of at most 1; a delay of 1 will satisfy the maximum delay constraint for any spanner with non-zero delay. Additionally, these short paths will frequently be suitable for providing low delay paths between vertices deeper in the tree.

Since breadth-first search exploits high-degree vertices, it is not well suited, in general, to producing spanners with low maximum degree. Since our primary criterion for "good" algorithms is number of edges, and whereas low maximum degree is only a secondary criterion, we feel that this algorithm is still a good choice for evaluation, regardless of its behaviour with respect to maximum degree of the spanner.

Breadth first search is known to run in $O(|E|)$ steps in the worst case.

Algorithm 8 Breadth-First Search Spanning Tree

```
E_S ← ∅
V_T ← r
for v ∈ N(r) do
  e ← (r, v)
  E_S ← E_S ∪ e
  V_T ← V_T ∪ {v}
  Enqueue v
end for
while V_T ≠ V do
  Dequeue v
  for v' ∈ N(v), v' ∉ V_T do
    e ← (v, v')
    E_S ← E_S ∪ e
    V_T ← V_T ∪ {v'}
    Enqueue v'
  end for
end while
```

Depth-First Search

The second spanning-tree algorithm we consider is a straightforward depth-first search. Presented with a vertex v, it will attempt to find an edge from v to a vertex v' not in the tree, add that edge to the tree, and then recursively consider v'. If, after the recursion to v' is complete and the tree has not been built, the algorithm will attempt to locate another vertex adjacent to v which is not in the tree. Each recursion at a vertex will consider all edges incident on that vertex and will exit when either a tree is built or it has no more edges to consider. Algorithm 9 implements our depth-first search spanning tree algorithm.

Depth-first search is known to require $O(|E|)$ steps in the worst case.

Simple Tree

This algorithm builds a tree by taking a list of edges in the graph and traversing that list in some order, adding edges to the tree as it goes. It is designed to exploit the fact that our algorithms are being implemented using a data structure which efficiently

Algorithm 9 Depth-First Search Spanning Tree

Require: r to be the specified root vertex for the tree
  $E_S \leftarrow \emptyset$
  $V_T \leftarrow root$
  for $e = (root, v) \in N(root)$ **do**
    if $v \notin V_T$ **then**
      $V_T \leftarrow V_T \cup \{v\}$
      $E_T \leftarrow E_T \cup \{e\}$
      Recursively consider v
    end if
  end for

Recursive component

Require: v to be the specified vertex to search from
  for $e = (v, v') \in N(v)$ **do**
    if $v' \notin V_T$ **then**
      $V_T \leftarrow V_T \cup \{v\}$
      $E_T \leftarrow E_T \cup \{e\}$
      Recursively consider v'
    end if
  end for

supports such a traversal. The algorithm initially adds the first edge, along with its two vertices, to the tree. It then proceeds to repeatedly traverse the edge list, adding any edge to the tree which will add a vertex to the tree, until all vertices are in the tree. Algorithm 10 illustrates this algorithm.

There are certain, highly degenerate, cases where up to $|V|-2$ passes over the edges may be required to complete the tree. Such cases would occur when only one vertex is added to the tree per list pass. These cases are not likely to occur in practice, and can be avoided by using an intelligent ordering of the edge list for each traversal. In such a highly degenerate case, the algorithm would require $O(|E|) \times |V| - 1 = O(|E||V|)$ steps altogether.

Algorithm 10 Simple Spanning Tree

```
E_S ← ∅
V_T ← {V[1]}
while V_T ≠ V do
  for i = 1 to |V| do
    e = (u, v) = E_G[i]
    if u ∈ V_T, v ∉ V_T or u ∉ V_T, v ∈ V_T then
      V_T ← V_T ∪ {u, v}
      E_T ← E_T ∪ {e}
    end if
  end for
end while
```

Popular Edges Tree

This spanning tree algorithm takes advantage of the same reasoning used for the *Popular Edges* algorithm (see Section 4.3.4.) Each edge is first given a "popularity" value in the manner described in Algorithm 11, and then a tree is built by selecting the $|V| - 1$ most popular edges from E such that no edge creates a cycle in S.

Since this algorithm uses the popularity assignment algorithm described in Section 4.3.4, we know that this algorithm requires $O(|V|^3)$ steps to assign the popularity values. After that, the choice of the tree edges can be made using a single pass through

Algorithm 11 Popular Edges Spanning Tree

Assign popularities to edges
Sort edge list in order of decreasing popularity
for each $e = (u, v)$ in the edge list **do**
  if adding e to E_T would not create a cycle **then**
    $V_T \leftarrow V_T \cup \{u, v\}$
    $E_T \leftarrow E_T \cup \{e\}$
  end if
end for

the edge list, which will require $|E|$ steps, so the algorithm requires $O(|V|^3)$ steps in the worst case.

4.4.2 Patching

We consider here six different patching algorithms. Since these algorithms are used to finish the generation of the spanner after the spanning tree is constructed, they all begin by calling an APSP algorithm in order to determine which pairs are overdelayed. This list of overdelayed pairs is then used by each algorithm to determine which edges to restore to the graph. Section 4.1.1 describes the construction and ordering of this list in mode detail.

Simple Patching

Simple Patching works by considering each overdelayed pair of vertices in the order in which they are presented in the list of overdelayed pairs. For each overdelayed vertex pair (i, j), the distance between i and j is first determined to see if the pairs remain too far from each other — although this information is initially available from the list of overdelayed pairs, edges added to the graph to satisfy other vertex pairs may also serve to bring i and j closer together. If the pair is no longer overdelayed, then no action is taken. If the pair is still overdelayed, then all of the edges in E_G of a shortest path between i and j are added to E_S. Clearly, adding these edges between i and j will ensure that (i, j) is no longer overdelayed.

Figure 4.9 illustrates patching in general, and Algorithm 12 implements Simple Patching.

Algorithm 12 Simple Patching

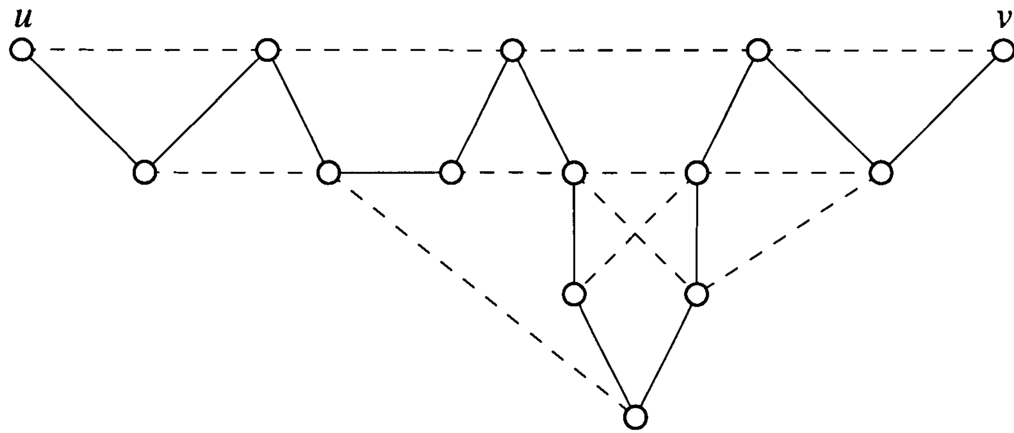
```
Determine overdelayed pairs; store in OD
for k = 1 to |OD| do
  (i, j) = OD[k]
  if d_S(i, j) > (d_G(i, j) + d) then
    for e ∈ P_G(i, j) do
      E_S ← E_S ∪ e
    end for
  end if
end for
```

In the worst case, at the start of Simple Patching, it may be that many of the vertex pairs are overdelayed, so we assume that $O(|V|^2)$ pairs are overdelayed. We also assume that, for the worst case, each repair will bring less than $|V|$ vertex pairs together. Since, for each vertex pair to be brought together by patching, we verify that the two vertices are overdelayed before adding the patch edges, we require $O(|E|)$ steps for each pair to determine that distance. Since, for any vertex pair, it takes only $O(|V|)$ steps to follow the shortest path between them in G and to add those path edges to S, we can observe that Simple Patching will require $O(|E|) \times O(|E| \times |V|^2) = O(|E||V|^2)$ steps in the worst case.

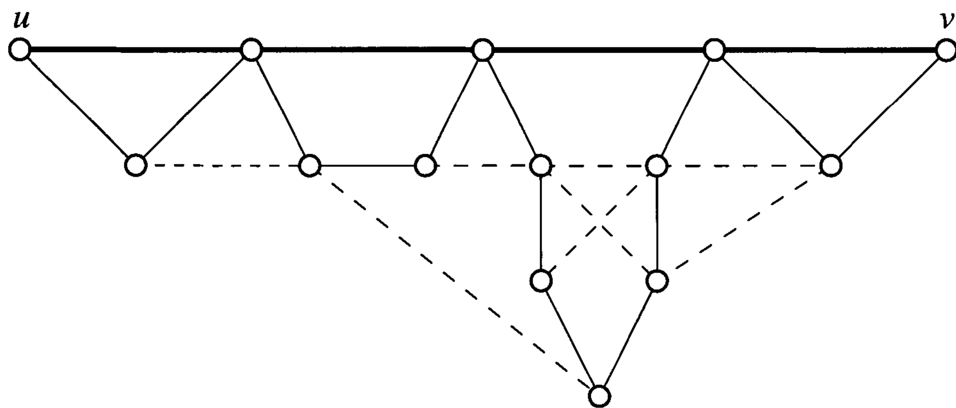
Farthest Pair Patching

Farthest Pair Patching is similar to Simple Patching, except that instead of considering the overdelayed vertex pairs in the order in which they occur in their data structure, they are considered in order of their overdelay (recall that the overdelay between two vertices i, j is $(d_S(i, j) - d_G(i, j) - d)$.) Note that we use *farthest*[2] to mean "most overdelayed", rather than the more intuitive meaning of "highly distant." By considering vertices which are most overdelayed first, adding a shortest-possible

[2]We apologize for the choice of the word "farthest." It was used as a "placeholder" name and not reconsidered until changing it had become impractical.



(a) Path from u to v, before patching.



(b) Path from u to v, after patching.

Figure 4.9: Patching — dashed lines indicate graph edges not present in the spanner

path between them will serve as a shortest path for those vertices on that path as well as an almost-shortest-possible path for vertex pairs i', j', where i' is near i and j' is near j. Algorithm 13 implements Farthest Pair Patching.

Algorithm 13 Farthest Pair Patching

```
Determine overdelayed pairs; store in OD
for i = 1 to i = |OD| do
  (i, j) = arg max_{(i,j)∈OD}(d_S(i, j) − d_G(i, j) − d)
  if d_S(i, j) > (d_G(i, j) + d) then
    for e ∈ P_G(i, j) do
      E_S ← E_S ∪ e
    end for
  end if
end for
```

In an efficient implementation, we would sort the overdelayed pairs list in order of decreasing delay, and then choose overdelayed pairs in-order from the sorted list. This would require $O(|V|^2 \log(|V|^2)) = O(2|V|^2 \log |V|)$ steps. If each vertex pair in the list required patching, it would require $O(|E|)$ steps to patch each of the $O(|V|^2)$ pairs, resulting in a total of $O(|E||V|^2)$ steps, just as Simple Patching requires. Despite having the extra sorting overhead, Farthest Pair Patching runs much faster in practice than does Simple Patching.

Farthest Pair Patching with Tie-Breaking

Farthest Pair Patching with Tie-Breaking is simply Farthest Pair Patching, except that if there exist more than one most overdelayed vertex pair, the pair with the lowest vertex degree sum will be chosen. This algorithm is expected to perform equivalently to Farthest Pair Patching, except that spanners produced by it may have a lesser maximum degree. Algorithm 14 implements Farthest Pair Patching with Tie-Breaking.

It is easy to see that Farthest Pair Patching with Tie-Breaking is almost identical to Farthest Pair Patching, and that it requires $O(|E||V|^2)$ steps in the worst case. In practice, its running time is similar to the regular Farthest Pair Patching algorithm.

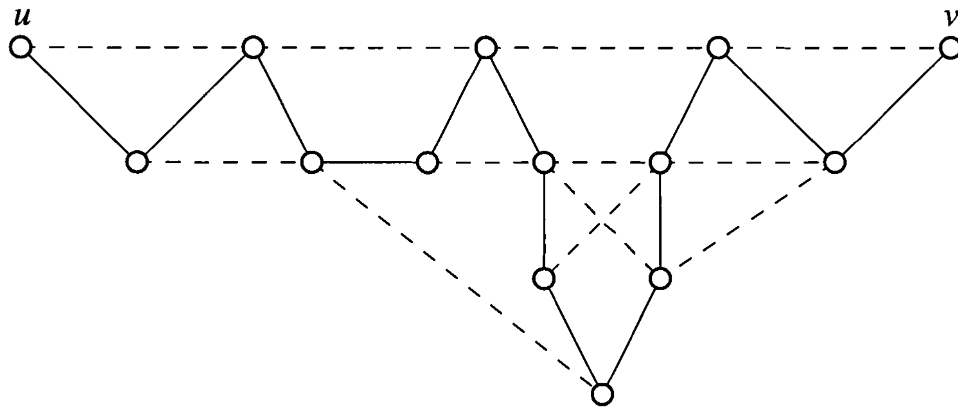
Algorithm 14 Farthest Edge Patching with Tie-Breaking

Determine overdelayed pairs; store in OD
for $i = 1$ to $i = |OD|$ **do**
  $OD' = \{(u,v)|(u,v) = \arg\max_{(u,v)\in OD}(d_S(u,v) - d_G(u,v) - d)$
  $(i,j) = \arg\min_{(i,j)\in OD'} d_S(i) + d_S(j)$
  if $d_S(i,j) > (d_G(i,j) + d)$ **then**
    for $e \in P_G(i,j)$ **do**
      $E_S \leftarrow E_S \cup e$
    end for
  end if
end for

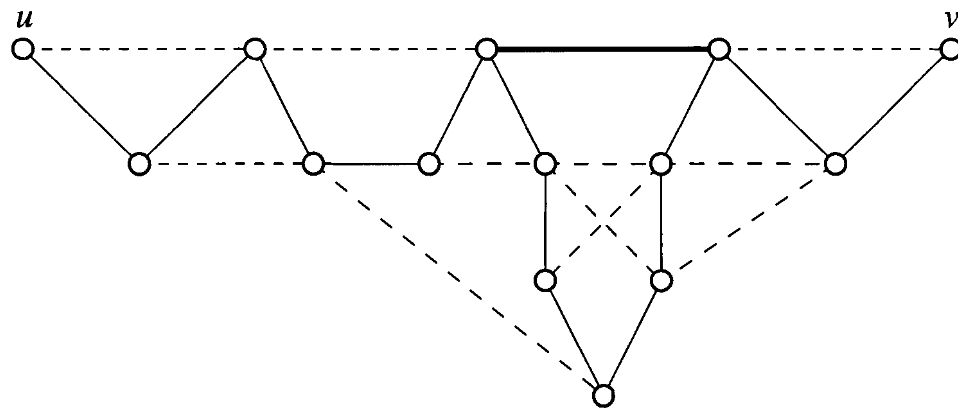
Simple Repairing

Repairing is a refinement of the basic patching procedure. Whereas patching ensures that the overdelayed pair is brought as close together as possible by adding all of the edges on the shortest path between the vertices in G, repairing selectively adds those edges along that path whose vertices are the most overdelayed. By adding such edges we bring together pairs of normally adjacent vertices that are separated by a large distance. By bringing together those previously distant vertices, we hope to reduce the delay between the vertices in our original overdelayed pair. Figure 4.10 illustrates repairing; note that to repair the vertex pair in the depicted graph to $d = 4$ requires only one edge, whereas the patching illustrated in Figure 4.9 requires four edges for the same graph and vertex pair, regardless of the value of d.

Simple Repairing selects vertex pairs in the same order as Simple Patching: by the order of the overdelayed pairs list. The distance between the chosen vertex pair (u, v) is verified to determine if it is still overdelayed, and the pair is ignored if it is no longer overdelayed. If the pair is still overdelayed, then the delays between all adjacent vertices in $P_G(u, v)$ are calculated and the edge between the vertex pair with the highest delay is added to S; let that edge be (i, j). This edge then partitions $P_G(u, v)$ into two subpaths (u, i) and (j, v); without loss of generality, we assume $d_G(u, i) < d_G(u, j)$ and $d_S(u, i) > d_S(j, v)$. If adding (i, j) to S is sufficient to make u and v no longer overdelayed, then the patching from u to v is complete. Otherwise, $d_S(u, i)$ and $d_S(j, v)$ are determined, and the longer of the two paths (being, under



(a) Path from u to v, before repairing.



(b) Path from u to v, after repairing.

Figure 4.10: Repairing — dashed lines indicate graph edges not present in the spanner.

our assumptions, the one from u to i) will be recursively repaired to a delay of $d_{u,i} = \lfloor d_{u,v}/2 \rfloor$ where $d_{u,v}$ is the maximum delay permitted for repairing u and v. If repairing the longer of the two subpaths still does not bring u and v close enough, then the shorter of the two subpaths will also be recursively repaired to a delay of $d_{j,v} = \lfloor d_{u,v}/2 \rfloor$.

This recursive repairing approach ensures that a path between u and v will always be repaired to a delay of at most d, where d is the delay parameter provided to the algorithm. This delay can be guaranteed by considering the nature of the subpaths repaired to delay $\lfloor d/2 \rfloor$: if the overdelays from u to i and from j to v are at most $\lfloor d/2 \rfloor$, then since (i,j) is present in the graph and since u, i, j, and v all lie on a shortest path in S from u to v, then the total delay along the repaired path is at most $2\lfloor d/2 \rfloor \leq d$. Algorithm 15 implements Simple Repairing.

Algorithm 15 Simple Repairing (recursive component only)

Require: u, v to be the vertices to be repaired between, $d_{u,v}$ to be the maximum delay between u and v
  for $e = (i,j) \in P_G(u,v)$ **do**
    $PD_e \leftarrow d_S(i,j) - d_G(i,j)$
  end for
  $e_{max} = \max_{e \in P_G(u,v)} PD_e$
  $E_S \leftarrow E_S \cup e_{max}$
  $(i,j) = e_{max}$
  if $d_S(u,v) > d_G(u,v) + d_{u,v}$ **then**
    Repair more overdelayed of $(u,i), (j,v)$ to delay $\lfloor d_{u,v}/2 \rfloor$
    if $d_S(u,v) > d_G(u,v) + d_{u,v}$ **then**
      Repair the other one to delay $\lfloor d_{u,v}/2 \rfloor$
    end if
  end if

In the worst case, Simple Repairing will need to make $O(|V|^2)$ repairs. Each vertex pair being repaired requires $O(|E|)$ steps to compute the distances between each of the $O(|V|)$ vertices on the shortest path between the pair being repaired. Taken in total, Simple Repairing requires $O(|V|^2) \times O(|E|) \times O(|V|) = O(|E||V|^3)$. Like the previous patching algorithms, Simple Repairing runs much faster in practice.

Farthest Pair Repairing

The difference between *Farthest Pair Repairing* and Simple Repairing is the same as the difference between Farthest Pair Patching and Simple Patching: Farthest Pair Repairing is identical to Simple Repairing except that it selects vertex pairs which are most overdelayed and then repairs the paths between them.

Similarly, Farthest Pair Repairing requires the same number of steps as Simple Repairing for each vertex pair being repaired, plus the overhead of sorting the overdelayed pairs list. It requires $O(|E||V|^3)$ steps in the worst case.

Distinct Tree Construction

Distinct Tree Construction takes a substantially different approach to patching a spanner. Instead of patching or repairing single paths within S, it attempts to create additional spanning trees which are nearly edge-disjoint from the edges already present in S. The motivation for this algorithm comes from our initial experimental observations which showed that spanning trees, especially those produced by a breadth-first search, created a good basis for additive spanners. In such a tree, vertex pairs near the root will generally have no or low overdelay, whereas vertex pairs located at or near the leaves will generally have high overdelay. If a second spanning tree is created which is rooted at a leaf of the first spanning tree, then this second tree should bring together many vertices which are distant in the first tree. If necessary, further such trees can be built to bring together other vertices until, ultimately, the spanner meets its delay constraint.

If the trees created in each step of this algorithm contained many of the same edges as the trees created in the previous stages, then the algorithm would quickly become inefficient, adding only a few new edges per tree at the expense of reconsidering a large number of edges. To prevent this from occuring, we require that the trees constructed in the algorithm be *distinct*. We define *distinct* to mean that edges are only added to a tree iff:

1. any edge from the root of the new tree does not occur in the previous tree built, and

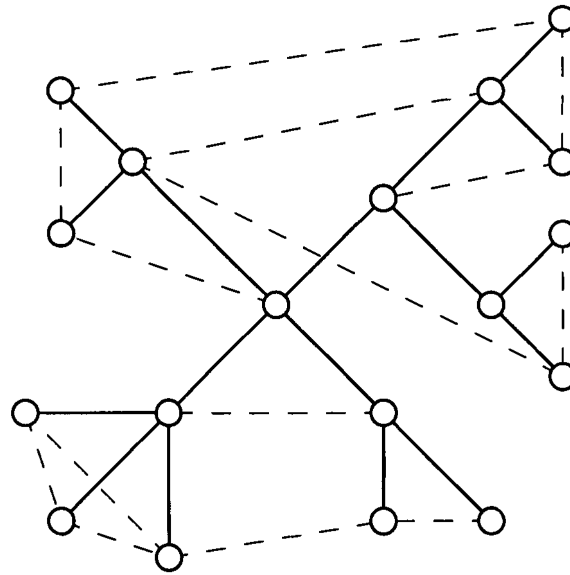
2. no other edge in the new tree exists in any previous tree.

The first condition attempts to ensure that the edge structure of the new tree differs from the previous tree built, in order to prevent it from building a tree which is substantially similar to the previous tree, and thus having only a minimal impact on the overall overdelays. The second condition ensures that the majority of the tree will be edge-disjoint from S as it exists when the tree is built. Since the second condition imposes strict edge-disjointness on most of the tree, there is a possibility that a complete spanning tree cannot be constructed. Such a situation would arise if there is a set of edges in S which form a cut in G and if the root of the new tree is not one of the vertices on a cut edge. In that case, it would not be possible to build a complete tree, since there would not be a path of edges not in S from the root to the other side of the cut. In such cases, the algorithm will accept such a "tree" as far as it can be built, and so the trees in this algorithm could be considered to be "best-effort trees". Figure 4.11 illustrates the creation of a distinct tree for patching; note that the tree cannot span the entire graph due to the edges used by the initial spanning tree.

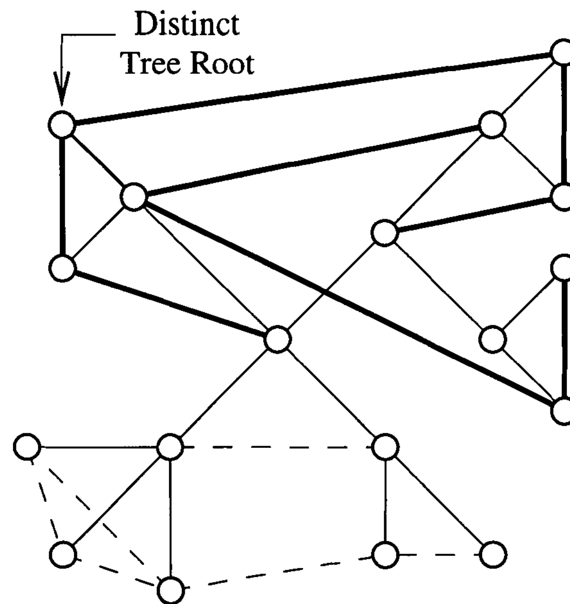
We wish to note that a strict disjointness condition was initially considered for this algorithm but, after the initial experimental analysis was performed, it was found to be too restrictive. The distinctness condition given here was used since it appeared to be the next least restrictive; it gives much more satisfactory experimental results.

The algorithm itself simply locates the most numerously overdelayed vertex in S, and then creates a distinct tree rooted at that vertex. After the tree is built, the algorithm calls APSP and determines if the spanner is still overdelayed. If it is, the algorithm repeats. Otherwise, it terminates. Algorithm 16 gives the implementation of Distinct Tree Construction.

In the worst case, after the construction of the original spanning tree, it may be the case that a spanning tree must be created from each of the $O(|V|)$ vertices in S. Since each tree in the patching phase is constructed by breadth-first search, each tree requires $O(|E|)$ steps to create it. In order to determine the most numerously overdelayed vertex in S, it would not be necessary to do more than perform an APSP, which requires $O(|V|^3) > O(|E|)$ operations. Since it is necessary to determine the



(a) Spanner — Initial spanning tree



(b) Spanner — After distinct-tree patching

Figure 4.11: Distinct-tree patching

Algorithm 16 Distinct Tree Construction

Require: L to be the set of leaves from the first-phase spanning tree.
  Let v be the most numerously overdelayed vertex in L.
  while S is overdelayed **do**
    for $e = (v, i) \in E_G, i \in L$ **do**
      $E_S \leftarrow E_S \cup \{e\}$
      $V_{tree} \leftarrow V_{tree} \cup \{i\}$
      Enqueue i
    end for
    while Vertex queue not empty **do**
      Dequeue i
      for $e = (i, j) \in E_G, e \notin E_S, j \notin V_{tree}$ **do**
        $E_S \leftarrow E_S \cup \{e\}$
        $V_{tree} \leftarrow V_{tree} \cup \{j\}$
        Enqueue j
      end for
    end while
    Let v be the most numerously overdelayed vertex in S.
    Let L be the leaves of the tree just constructed.
  end while

most numerously overdelayed vertex in S after each additional tree is created, there may be at most $O(|V|) \times O(|V|^3) = O(|V|^4)$ operations required.

4.4.3 Analysis

For all of the 2-stage algorithms constructible from the components given above, the worst-case running times of these algorithms are dominated by the running times of their patching phases. Given that, the worst-case running time of any such algorithm is equivalent to the worst-case running time of the patching algorithm it uses.

4.5 Decomposition Algorithms

One further class of algorithms which was considered for evaluation in this thesis was what we refer to as *decomposition algorithms*. The primary idea for these algorithms is that a $(d + x)$-spanner of an input graph G could be constructed by the following process:

1. Break G down into two components, G_1 and G_2, by taking a (most likely minimum) cut C.
2. Produce spanners of G_1 and G_2, being S_1 and S_2, respectively.
3. Merge S_1, S_2, and C to produce S.

Recursive application of this procedure would permit a large graph to be decomposed into many smaller graphs where each of which could be much more quickly spanned.

We have two arguments that support a contention that these algorithms are not suitable for creating good spanners as defined by our criteria.

4.5.1 Algorithmic Argument

This thesis presents algorithms for creating spanners of graphs where all vertex pairs in the spanner are connected by a path of length at most d greater than the corresponding paths in the spanner. Creating $(d+x)$-spanners of both G_1 and G_2 is not feasible, since

it would be possible for the shortest path in S between some vertex in S_1 and another vertex in S_2 to have delay $2d$. Thus, it must be that any path which crosses the cut must have a total delay of no more than d across both S_1 and S_2. In general, there are two simple methods which could be used to ensure those delays are maintained:

1. Create spanners of G_1 and G_2 where, for any vertex pair where one vertex is part of an edge in C, the delay between that pair is $d/2$; for all other pairs the delay is d.

2. Create spanners of G_1 and G_2 with delay $d/2$.

(Note: For odd d, $d/2$ may be substituted with $\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$ for the smaller and larger of G_1 and G_2, respectively.)

If the first approach is taken, then the delays used to create the spanners of G_1 and G_2 are no longer uniform for all vertex pairs in the two components. Solving this problem would require constructing algorithms which deal with a non-constant delay for all vertex pairs; such algorithms lie outside the scope of this thesis.

If the second approach is taken, then the delays in each component would decrease uniformly within each component. This is, in general, not desirable, since it could result in many edges being retained in the resultant spanner which are not actually necessary for a $(d + x)$-spanner but which are necessary for the $d/2 + x$-spanners of the two components.

4.5.2 Structural Argument

There are certain structural issues regarding decomposed spanners that make them unsuitable for consideration in this thesis. We note the following: If we wish to create a $(d+x)$-spanner of a graph G by decomposition, then we must create $(d/2+x)$-spanners of its components G_1 and G_2. If we apply the decomposition approach recursively to G_1 and G_2, then we will need to construct $(d/4 + x)$-spanners of their decomposed components. Further decompositions will reduce the delay of the component spanners even further.

For generating a $(d + x)$-spanner of any graph, at most $\lfloor \log_2 d \rfloor$ levels of decomposition are possible; any decomposition beyond that will result in a delay of 0 in the spanners of the decomposed components, and any spanner with delay 0 must be identical to the original graph. Since the largest delay being considered experimentally in this thesis is 9, this would permit at most 3 levels of decomposition, a value which would not be very suitable for trying to create spanners of large and relatively sparse graphs.

Further, decomposition of dense graphs will not be very efficient, since a dense graph will likely require a relatively large number of edges to create a cut. Since the graph is dense, it is likely that many of those cut edges will be very near other cut edges, and thus many of those edges could be removed from the spanner with an acceptable increase in delay. A simple decomposition algorithm would not be well suited to exploit the nature of such a dense cut, and algorithms which do so are beyond the scope of this thesis.

4.6 Baseline Algorithm

As described in Section 3.3, a baseline algorithm was desired in order to compare the performance of the heuristics against a relatively "dumb" algorithm. We present such a relatively dumb algorithm here.

This algorithm, which we call *random edges*, produces a spanner by randomly selecting edges from G until the maximum delay constraint is met.

Algorithm 17 Random Edges

$E_S \leftarrow 0$
while S is not a $(d + x)$-spanner **do**
  Select an edge $e \in E_G \setminus E_S$ with uniform probability.
  $E_S \leftarrow E_S \cup \{e\}$
end while

Chapter 5

Experimental Analysis

All life is an experiment. The more experiments you make the better.
Ralph Waldo Emerson
(attributed)

A series of experiments were conducted on each of the algorithms described in Chapter 4 in order to evaluate the spanners they produce. Each spanner was evaluated according to the criteria described in Chapter 3. In this chapter, we discuss the particulars of the experiments and the implementation of the algorithms. We also describe the results of the experiments, ranking the algorithms first against the primary criterion in order to determine a set of algorithms whose performance was generally good. We then rank those good algorithms against the secondary criteria. Finally, we summarize these results, highlighting algorithms which do well according to many of our criteria.

5.1 Experiments

In order to evaluate the algorithms proposed in Chapter 4, it was necessary to use each of them to produce spanners of the graphs described in Chapter 3. Since the graph classes discussed in that chapter are infinite, it was necessary to consider only

a finite subset of the graphs in those classes. It was also necessary to limit the range of delay parameter values considered, in order to keep the computation manageable. For the experiments which were conducted, spanners were created:

- by each of the 30 algorithms defined in Chapter 4,
- for each of 20 shuffles (see Section 5.1.3) of each the graphs listed in Table 5.1,
- for each of the fixed maximum delay parameter values $d = 1 \ldots 9$.

For each such spanner, a series of features of the spanner, as well as of the algorithm run that produced it, were measured and recorded. Those features included:

- Quantity of edges in the spanner
- Maximum degree of the spanner
- Maximum delay of the spanner (which may be $\leq d$)
- Average delay of the spanner
- Running time of the algorithm which created the spanner

The results for each algorithm/graph/delay triple were then averaged over the 20 shuffles, and that average value was used for further analysis.

5.1.1 Test Graphs

In the experiment runs, the input graphs from the four classes of structured graphs given in Section 3.2.1 were restricted to having at most 512 vertices, except for the grids, which were restricted to 256 vertices. Table 5.1 lists the graphs in the test set. The 512 vertex restriction came from an implementation restriction on the size of the graphs which could be tested; see Section 5.2 for more information. The 256 vertex restriction for grids was put in place to try to keep the number of instances of grids manageable. The grids were also restricted to being "square" (*i.e.* of the form $G_{n,\ldots,n}$ for the necessary number of dimensions), again to attempt to keep the number of input graphs manageable.

For the random graph model given in Section 3.2.2, three sizes of graphs were considered: those having 50, 100, and 200 vertices, respectively. Additionally, for each of those three sizes, a variety of edge-probabilities were considered; see Table 5.1 for a complete listing of the probabilities. For each combination of size and edge-probability, 5 different graphs were generated randomly; the results from these 5 graphs were averaged into one set of results in the final analysis.

Finally, we note that we required all graphs used in our experiments to be connected. While this is clearly the case for graphs in the four structured classes, this may not always be the case for the random graphs. The choice to require the graphs to be connected was motivated by the fact that the definition of additive graph spanners due to Liestman and Shermer [15] is unclear on the nature of spanners of disconnected graphs. One reasonable interpretation of the definition would indicate that a $(d + x)$-spanner of some disconnected graph G could be produced by producing $(d+x)$-spanners of each of the components of G. In this case, the problem of computing a good spanner of G can be likewise decomposed into several smaller, independent problems. Thus, the restriction to connected graphs ensures that the algorithms will be tested in the worst-case. Since each of the algorithms work on connected graphs, they may be easily modified to compute spanners of disconnected graphs.

In Table 5.1, we use the standard shorthand notations for denoting grids and hypercubes. We use the notation XT_n to denote the X-tree of n levels, and use P_n to denote the pyramid of n levels; we refer the interested reader to consult Liestman and Shermer [15] for the definitions of X-trees and pyramids. We use the notation $R_n(p) = (V, E_R)$ to refer to a subgraph of $K_n = (V, E_K)$ where each each edge from E_K is included in E_R with probability p. Thus, the notation $R_n(p)$ refers to a graph on n vertices where, for any pair of distinct vertices u and v, there is an edge from u to v with probability p.

Graph Class	Subclass	Graphs Used in Experiments
Grids	2-dimensional	$G(4,4)$, $G(5,5)$, ..., $G(16,16)$
	3-dimensional	$G(3,3,3)$, $G(4,4,4)$, ..., $G(6,6,6)$
	4-dimensional	$G(2,2,2,2)$, $G(3,3,3,3)$, $G(4,4,4,4)$
Hypercubes		Q_3, Q_4, ..., Q_9
X-trees		XT_2, XT_3, ..., XT_9
Pyramids		P_2, P_3, P_4, P_5
Random	$p = 0.05$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.10$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.15$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.20$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.25$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.30$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.40$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$
	$p = 0.50$	$5 \times R_{50}(p)$, $5 \times R_{100}(p)$, $5 \times R_{200}(p)$

Table 5.1: Graphs used for experiments

5.1.2 Algorithms

The algorithms which were used in the experiments were the algorithms described in Chapter 4. Table 5.2 summarizes the names of the algorithms and gives the short-form labels which will be used to indicate them on the plots. Figure 5.2 (see page 83) gives the key to the plots which will be used through the remainder of this thesis. In the case of a 2-stage algorithm, its name is a combination of its spanning tree algorithm and its patching algorithm. The table and figure also lists the short-form label for when values from known constructions will be used in the plots.

5.1.3 Shuffling

In order to conduct the experiments, it was necessary to implement the algorithms. Implementing the algorithms requires certain decisions to be made regarding the order in which the graph data will be handled. These decisions may have an unintended effect on the operation of the algorithm. Although steps were taken to try to reduce

Algorithm	Label
Far Edge Deletion	FED
Tainting	ST
Neighbourhood Tainting	NT
Popular Edges	PE
Low Degree Pairs	LDP
Random Edges (baseline algorithm)	RE
BFS Tree / Simple Patching	BFSP
BFS Tree / Farthest Pair Patching	BFFP
BFS Tree / Farthest Pair Patching with Tie-Breaking	BFTB
BFS Tree / Simple Repairing	BFSR
BFS Tree / Farthest Pair Repairing	BFFR
BFS Tree / Distinct Tree Construction	BFDT
DFS Tree / Simple Patching	DFSP
DFS Tree / Farthest Pair Patching	DFFP
DFS Tree / Farthest Pair Patching with Tie-Breaking	DFTB
DFS Tree / Simple Repairing	DFSR
DFS Tree / Farthest Pair Repairing	DFFR
DFS Tree / Distinct Tree Construction	DFDT
Simple Tree / Simple Patching	STSP
Simple Tree / Farthest Pair Patching	STFP
Simple Tree / Farthest Pair Patching with Tie-Breaking	STTB
Simple Tree / Simple Repairing	STSR
Simple Tree / Farthest Pair Repairing	STFR
Simple Tree / Distinct Tree Construction	STDT
Popular Edges Tree / Simple Patching	PESP
Popular Edges Tree / Farthest Pair Patching	PEFP
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	PETB
Popular Edges Tree / Simple Repairing	PESR
Popular Edges Tree / Farthest Pair Repairing	PEFR
Popular Edges Tree / Distinct Tree Construction	PEDT
(values from known constructions)	C

Table 5.2: Table of algorithms and their labels

such effects during implementation (see Section 5.2), we felt it was necessary to further mitigate these effects by "shuffling" each input graph several times, running the algorithm on each shuffle, and then averaging the results over all the shuffles.

Within our experimentation structure, each graph was represented as an adjacency matrix. For a graph G, each shuffle of G was produced by:

1. Numbering the vertices of the graph as $1, 2, \ldots, |V_G|$ according to the order they were presented in the adjacency matrix.
2. Producing a pseduorandom permutation of $1, 2, \ldots, |V_G|$ as $p_1, p_2, \ldots p_{|V_G|}$.
3. Producing a new shuffled adjacency matrix representing the graph with the vertices in the order $p_1, p_2, \ldots p_{|V_G|}$.

This process results produces a graph identical to the original graph, except that the vertices are presented in a different order within the data structure.

In order to maintain consistency of the shuffles across the algorithm/graph/delay triples, a single set of 20 graph shuffles was generated for each graph G in the set of test graphs, and this set of shuffles was used each time an algorithm was tested on graph G.

5.2 Implementation

To perform the experiments, each algorithm was implemented. In order to produce spanners which can be compared against each other, it was necessary to implement the algorithms in a reasonably consistent manner. In this section, we describe our implementation of the algorithms and the system used to evaluate their performance. We also describe some of the implementation choices which were made and the limitations those choices imposed on our overall system.

5.2.1 Evaluation System

Each algorithm was written as a function according to a well-defined specification. This function takes as input:

- The input graph G, including the number of vertices in the graph (as the value n) and the graph itself (as an an $n \times n$ adjacency matrix), and
- A maximum delay parameter d

The function is required to produce a $(d + x)$-spanner of G and to return it as an $n \times n$ adjacency matrix.

Each algorithm was implemented as a function of this form, and was called by a "wrapper" program. The wrapper program took care of necessary I/O activity (e.g. loading the graph data from disk), much of the memory allocation, as well as instrumenting the program for measuring the running time of the algorithm. The wrapper program, with an algorithm compiled in, given as input a graph and maximum delay, would output the spanner produced by the algorithm on that input, along with data from the wrapper program recorded by the instrumented program run. This wrapper program, in conjunction with the spanner heuristic function specification, allowed for consistent implementation of the algorithms, as well as consistent measurement of their runtime properties.

An additional suite of utility programs were written to perform data analysis on and collation of the amassed spanner data. Figure 5.1 illustrates the structure of the system we describe here.

5.2.2 Implementation Choices and Limitations

All of the programs were written in C. For each algorithm, an attempt was made to implement the algorithm in an efficient manner, but it was necessary to balance that efficiency against ensuring the correctness of the implementation. Since the primary criterion used to determine if a given algorithm was "good" was the number of edges in the spanners it produces and not the running time of the algorithm, implementation decisions were always made in favour of correctness.

The experiments themselves were carried out on a series of Pentium 4 class workstations running Redhat Linux. The distribution of the jobs was handled by the program *autoson* [18]. The machines were identical, except that had two different

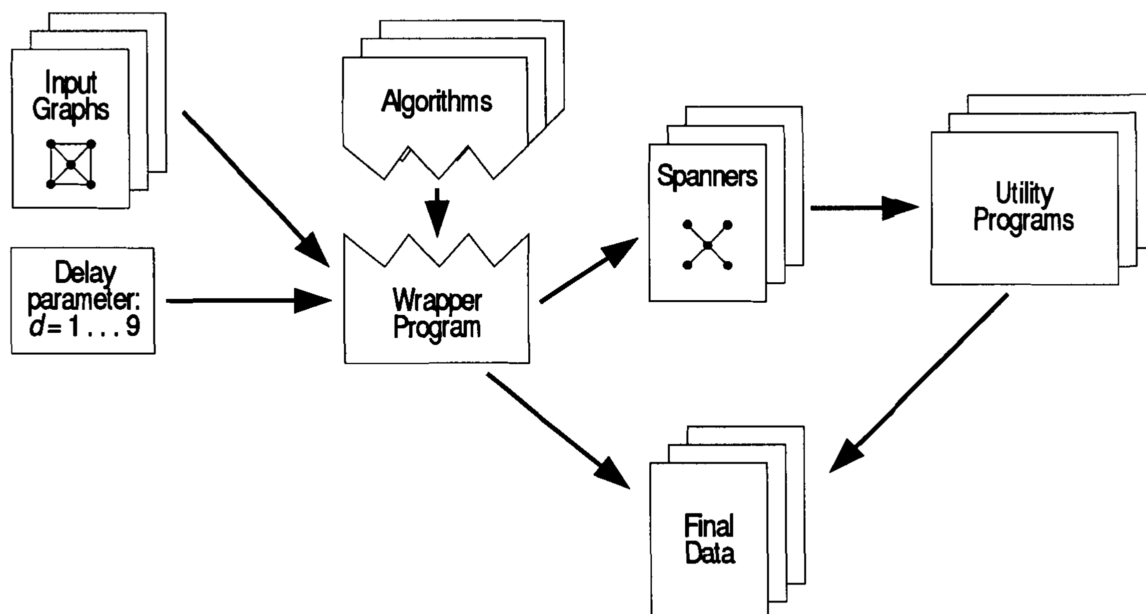


Figure 5.1: Experiment implementation structure.

processor speeds. In order to normalize the running times for all the experiments, a series of identical experiment runs were conducted on both machines and their times were compared. These experiment runs were chosen to have significant running times (generally $>$ 10 seconds) in order to eliminate any effect caused by the OS on runs with small running times. In all cases, the running times of the instances on the slower machines were slower by a factor $\approx$ 1.35. We have multiplied the running times of the runs performed on the faster machines by 1.35 in order to make them comparable to the running times of the runs performed on the slower machines.

5.3 Primary Criterion

In this section, we compare the numbers of edges in the spanners produced by the various algorithms for the graphs given in Table 5.1. We will compare them according to the various classes of graphs (grids, hypercubes, pyramids, X-trees, random graphs,) as well as in total for all the graphs. We will use two different means of ranking them and will, from that information, produce a list of heuristics which consistently produce

spanners which are good by our primary criterion, namely that they contain few edges.

5.3.1 Competitive Rankings

Our first method of comparison is by ranking the algorithms based on the number of edges in the spanners they created for a particular graph/delay-parameter combination. The spanner with the fewest edges will be ranked first, the spanner with the next fewest edges will be second, and so on. Since there are a total of 159 graphs and 9 delay values considered for each, this results in a total of 1431 comparisons. We will not present each comparison here. Instead, we will summarize them over the various graph classes, as well as over the whole of the test graphs.

Each summary presents the percentage of times each algorithm was present in the "top 10" of the rankings; that is, the percentage of the total rankings for that graph class in which the spanner produced by the given algorithm had fewer edges than at least 20 other algorithms. We include the baseline Random Edges algorithm in these comparisons.

Tables 5.3, 5.4, 5.5, 5.6, and 5.7 give the rank summaries for the grid, hypercube, X-tree, pyramid, and random graph classes, respectively. The most noticeable result in these rankings is that the BFS Tree / Farthest Pair Repairing algorithm always appears in the top 10 rankings, and is the only algorithm to do so.

BFS Tree / Simple Repairing and BFS Tree / Farthest Pair Patching with Tie-Breaking each appear consistently in the top 10 rankings of four of the five graph classes. In their remaining classes, being respectively random graphs and grids, they appear in 98% and 82% of top 10 rankings.

Other algorithms to appear consistently in the top 10 rankings for some graph class(es) are:

- BFS Tree / Simple Patching, for the pyramid and X-tree classes.
- BFS Tree / Farthest Pair Patching, for the class of random graphs.
- Simple Tree / Simple Repairing, for the class of hypercubes.
- Simple Tree / Farthest Pair Repairing, also for the class of hypercubes.

Algorithm	Appears in Top 10
BFS Tree / Distinct Tree Construction	13.333%
BFS Tree / Farthest Pair Patching with Tie-Breaking	82.222%
BFS Tree / Farthest Pair Patching	77.222%
BFS Tree / Farthest Pair Repairing	100.000%
BFS Tree / Simple Patching	96.667%
BFS Tree / Simple Repairing	100.000%
Far Edge Deletion	11.111%
DFS Tree / Distinct Tree Construction	11.111%
DFS Tree / Farthest Pair Patching with Tie-Breaking	11.111%
DFS Tree / Farthest Pair Patching	11.111%
DFS Tree / Farthest Pair Repairing	15.556%
DFS Tree / Simple Patching	16.667%
DFS Tree / Simple Repairing	38.889%
Tainting	13.333%
Low Degree Pairs	11.111%
Simple Tree / Distinct Tree Construction	13.333%
Simple Tree / Farthest Pair Patching with Tie-Breaking	31.111%
Simple Tree / Farthest Pair Patching	16.667%
Simple Tree / Farthest Pair Repairing	47.222%
Simple Tree / Simple Patching	45.556%
Simple Tree / Simple Repairing	90.000%
Neighbourhood Tainting	11.111%
Popular Edges	11.111%
Popular Edges Tree / Distinct Tree Construction	12.222%
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	44.444%
Popular Edges Tree / Farthest Pair Patching	46.667%
Popular Edges Tree / Farthest Pair Repairing	95.556%
Popular Edges Tree / Simple Patching	61.111%
Popular Edges Tree / Simple Repairing	96.667%
Random Edges (baseline)	11.111%

Table 5.3: “Top 10” fractions for all algorithms over all grids

Algorithm	Appears in Top 10
BFS Tree / Distinct Tree Construction	30.159%
BFS Tree / Farthest Pair Patching with Tie-Breaking	100.000%
BFS Tree / Farthest Pair Patching	90.476%
BFS Tree / Farthest Pair Repairing	100.000%
BFS Tree / Simple Patching	90.476%
BFS Tree / Simple Repairing	100.000%
Far Edge Deletion	11.111%
DFS Tree / Distinct Tree Construction	17.460%
DFS Tree / Farthest Pair Patching with Tie-Breaking	17.460%
DFS Tree / Farthest Pair Patching	17.460%
DFS Tree / Farthest Pair Repairing	26.984%
DFS Tree / Simple Patching	17.460%
DFS Tree / Simple Repairing	33.333%
Tainting	15.873%
Low Degree Pairs	11.111%
Simple Tree / Distinct Tree Construction	26.984%
Simple Tree / Farthest Pair Patching with Tie-Breaking	96.825%
Simple Tree / Farthest Pair Patching	65.079%
Simple Tree / Farthest Pair Repairing	100.000%
Simple Tree / Simple Patching	77.778%
Simple Tree / Simple Repairing	100.000%
Neighbourhood Tainting	11.111%
Popular Edges	11.111%
Popular Edges Tree / Distinct Tree Construction	23.810%
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	26.984%
Popular Edges Tree / Farthest Pair Patching	26.984%
Popular Edges Tree / Farthest Pair Repairing	49.206%
Popular Edges Tree / Simple Patching	26.984%
Popular Edges Tree / Simple Repairing	65.079%
Random Edges (baseline)	11.111%

Table 5.4: "Top 10" fractions for all algorithms over all hypercubes

Algorithm	Appears in Top 10
BFS Tree / Distinct Tree Construction	25.000%
BFS Tree / Farthest Pair Patching with Tie-Breaking	100.000%
BFS Tree / Farthest Pair Patching	97.222%
BFS Tree / Farthest Pair Repairing	100.000%
BFS Tree / Simple Patching	100.000%
BFS Tree / Simple Repairing	100.000%
Far Edge Deletion	36.111%
DFS Tree / Distinct Tree Construction	19.444%
DFS Tree / Farthest Pair Patching with Tie-Breaking	19.444%
DFS Tree / Farthest Pair Patching	19.444%
DFS Tree / Farthest Pair Repairing	23.611%
DFS Tree / Simple Patching	19.444%
DFS Tree / Simple Repairing	20.833%
Tainting	16.667%
Low Degree Pairs	12.500%
Simple Tree / Distinct Tree Construction	23.611%
Simple Tree / Farthest Pair Patching with Tie-Breaking	34.722%
Simple Tree / Farthest Pair Patching	27.778%
Simple Tree / Farthest Pair Repairing	44.444%
Simple Tree / Simple Patching	33.333%
Simple Tree / Simple Repairing	45.833%
Neighbourhood Tainting	9.722%
Popular Edges	12.500%
Popular Edges Tree / Distinct Tree Construction	20.833%
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	81.944%
Popular Edges Tree / Farthest Pair Patching	79.167%
Popular Edges Tree / Farthest Pair Repairing	87.500%
Popular Edges Tree / Simple Patching	69.444%
Popular Edges Tree / Simple Repairing	87.500%
Random Edges (baseline)	12.500%

Table 5.5: "Top 10" fractions for all algorithms over all X-trees

Algorithm	Appears in Top 10
BFS Tree / Distinct Tree Construction	30.556%
BFS Tree / Farthest Pair Patching with Tie-Breaking	100.000%
BFS Tree / Farthest Pair Patching	97.222%
BFS Tree / Farthest Pair Repairing	100.000%
BFS Tree / Simple Patching	100.000%
BFS Tree / Simple Repairing	100.000%
Far Edge Deletion	25.000%
DFS Tree / Distinct Tree Construction	19.444%
DFS Tree / Farthest Pair Patching with Tie-Breaking	22.222%
DFS Tree / Farthest Pair Patching	22.222%
DFS Tree / Farthest Pair Repairing	22.222%
DFS Tree / Simple Patching	19.444%
DFS Tree / Simple Repairing	19.444%
Tainting	13.889%
Low Degree Pairs	22.222%
Simple Tree / Distinct Tree Construction	22.222%
Simple Tree / Farthest Pair Patching with Tie-Breaking	41.667%
Simple Tree / Farthest Pair Patching	33.333%
Simple Tree / Farthest Pair Repairing	50.000%
Simple Tree / Simple Patching	36.111%
Simple Tree / Simple Repairing	47.222%
Neighbourhood Tainting	13.889%
Popular Edges	0.000%
Popular Edges Tree / Distinct Tree Construction	22.222%
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	91.667%
Popular Edges Tree / Farthest Pair Patching	86.111%
Popular Edges Tree / Farthest Pair Repairing	91.667%
Popular Edges Tree / Simple Patching	91.667%
Popular Edges Tree / Simple Repairing	97.222%
Random Edges (baseline)	0.000%

Table 5.6: "Top 10" fractions for all algorithms over all pyramids

Algorithm	Appears in Top 10
BFS Tree / Distinct Tree Construction	65.741%
BFS Tree / Farthest Pair Patching with Tie-Breaking	100.000%
BFS Tree / Farthest Pair Patching	100.000%
BFS Tree / Farthest Pair Repairing	100.000%
BFS Tree / Simple Patching	93.519%
BFS Tree / Simple Repairing	97.685%
Far Edge Deletion	39.815%
DFS Tree / Distinct Tree Construction	0.463%
DFS Tree / Farthest Pair Patching with Tie-Breaking	1.389%
DFS Tree / Farthest Pair Patching	1.852%
DFS Tree / Farthest Pair Repairing	7.870%
DFS Tree / Simple Patching	0.000%
DFS Tree / Simple Repairing	3.704%
Tainting	0.000%
Low Degree Pairs	0.000%
Simple Tree / Distinct Tree Construction	5.556%
Simple Tree / Farthest Pair Patching with Tie-Breaking	41.667%
Simple Tree / Farthest Pair Patching	26.852%
Simple Tree / Farthest Pair Repairing	68.981%
Simple Tree / Simple Patching	71.296%
Simple Tree / Simple Repairing	85.185%
Neighbourhood Tainting	0.000%
Popular Edges	0.000%
Popular Edges Tree / Distinct Tree Construction	5.093%
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	23.148%
Popular Edges Tree / Farthest Pair Patching	25.926%
Popular Edges Tree / Farthest Pair Repairing	36.111%
Popular Edges Tree / Simple Patching	32.407%
Popular Edges Tree / Simple Repairing	47.222%
Random Edges (baseline)	0.000%

Table 5.7: "Top 10" fractions for all algorithms over all random graphs

Table 5.8 gives the rank summaries over all the input graphs. We can see that the algorithms BFS Tree / Farthest Pair Repairing, BFS Tree / Simple Repairing, and BFS Tree / Farthest Pair Patching with Tie-Breaking which appear consistently in the rankings for several graph classes also do very well overall. Specifically, they appear in 100%, 99%, and 94% of the top 10 rankings, respectively. Of the remaining algorithms which appeared consistently in the rankings for some graph class, the remaining two BFS Tree algorithms, BFS Tree / Farthest Pair Patching and BFS Tree / Simple Patching, appeared in 91% and 95% of the top 10 rankings. The Simple Tree / Simple Repairing and Simple Tree / Farthest Pair Repairing algorithms each consistently appeared in the rankings for one graph class, and appeared in 81% and 61% of the top 10 rankings over all graphs

Two other algorithms, Popular Edges Tree / Simple Repairing and Popular Edges Tree / Farthest Pair Repairing did not appear consistently in the top 10 rankings for any graph class, but did appear in 73% and 66% of the top 10 rankings overall.

The table also lists two measures of the the overall performance of the algorithms. These results summarize the results given in the tables for each of the individual graph classes. The first measure gives the number of graph classes in which the given algorithm appeared in 100% of the top ten results for each delay/graph input instance in that class. We say that this 100% measure indicates how *exceptional* a given algorithm is. Clearly, if the algorithm consistently appears in the top ten results for a given graph class, it is likely a good algorithm for generating spanners of graphs in that class.

The second measure is similar to the first. It indicates the number of graph classes in which the algorithm appeared in at least 50% of the top 10 results for each delay/graph input instance. We say that this 50% measure indicates how *acceptable* a given algorithm is. If an algorithm appears less than 50% of the time in the top ten results for a given graph class, then the algorithm is likely a bad choice for generating spanners of graphs in that class.

Algorithm	Appears in Top 10	Class Results 100%	50%
BFS Tree / Distinct Tree Construction	37.743%	0	1
BFS Tree / Farthest Pair Patching with Tie-Breaking	94.356%	4	5
BFS Tree / Farthest Pair Patching	91.182%	1	5
BFS Tree / Farthest Pair Repairing	100.000%	5	5
BFS Tree / Simple Patching	95.414%	2	5
BFS Tree / Simple Repairing	99.118%	4	5
Far Edge Deletion	26.102%	0	0
DFS Tree / Distinct Tree Construction	9.347%	0	0
DFS Tree / Farthest Pair Patching with Tie-Breaking	9.877%	0	0
DFS Tree / Farthest Pair Patching	10.053%	0	0
DFS Tree / Farthest Pair Repairing	15.344%	0	0
DFS Tree / Simple Patching	10.935%	0	0
DFS Tree / Simple Repairing	21.340%	0	0
Tainting	8.995%	0	0
Low Degree Pairs	7.760%	0	0
Simple Tree / Distinct Tree Construction	13.757%	0	0
Simple Tree / Farthest Pair Patching with Tie-Breaking	43.563%	0	1
Simple Tree / Farthest Pair Patching	28.395%	0	1
Simple Tree / Farthest Pair Repairing	61.199%	1	3
Simple Tree / Simple Patching	56.790%	0	2
Simple Tree / Simple Repairing	80.952%	1	3
Neighbourhood Tainting	6.878%	0	0
Popular Edges	6.349%	0	0
Popular Edges Tree / Distinct Tree Construction	12.522%	0	0
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	42.152%	0	2
Popular Edges Tree / Farthest Pair Patching	43.210%	0	2
Popular Edges Tree / Farthest Pair Repairing	66.490%	0	3
Popular Edges Tree / Simple Patching	49.383%	0	3
Popular Edges Tree / Simple Repairing	73.192%	0	4
Random Edges (baseline)	6.349%	0	0

Table 5.8: "Top 10" fractions for all algorithms over all graphs

5.3.2 Edge ratio rankings

In this section, we rank the algorithms based on the *edge ratios* of the spanners they produce. For a spanner S of a graph G, we say that the *edge ratio* of S is

$$ER(S) = \frac{|E_S|}{|E_G|}.$$

Put simply, the edge ratio of a spanner is the fraction of edges retained from the original graph. Since this measure is normalized against the size of the original input graph, we can use it to compare the sizes of spanners produced of graphs with different sizes.

Tables 5.9, 5.10, 5.11, 5.12, and 5.13 give the average edge ratio values over the grid, hypercube, X-tree, pyramid, and random graph classes, respectively. In most cases, we see that the algorithms which produced the spanners with the lowest edge ratio are the same algorithms which appeared in most of the top 10 rankings of number of edges. While this is not especially surprising, we do note that, for each graph class, the algorithms with the lowest average edge ratio were not necessarily the same as those which most consistently appeared in the top 10 rankings of numbers of edges. This indicates that the algorithms with good average edge ratios may perform poorly in certain circumstances, and it also implies that placing high on the top 10 rankings list indicates that the algorithm may do well in most cases, but may not always be in the first position in that ranking.

Table 5.14 gives the average degree ratio of the spanners produced for all of the test graphs. In general the BFS Tree family of 2-stage algorithms dominate the list, except for BFS Tree / Disjoint Tree Construction, which appears in 22nd place. We also note that Popular Edges Tree / Farthest Pair Repairing, Simple Tree / Simple Repairing, and Popular Edges Tree / Farthest Pair Repairing perform well in general.

5.3.3 Comparisons against known constructions

In this section, we discuss the relative performance of the algorithms against the existing additive spanner constructions of Liestman and Shermer [15, 13].

Algorithm	Edge Ratio
BFS Tree / Simple Repairing	0.6655
BFS Tree / Farthest Pair Repairing	0.6708
Popular Edges Tree / Simple Repairing	0.6755
BFS Tree / Simple Patching	0.6798
Popular Edges Tree / Farthest Pair Repairing	0.6803
Simple Tree / Simple Repairing	0.6811
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.6820
BFS Tree / Farthest Pair Patching	0.6844
Popular Edges Tree / Simple Patching	0.6918
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	0.6938
Popular Edges Tree / Farthest Pair Patching	0.6942
Simple Tree / Farthest Pair Repairing	0.6943
Simple Tree / Simple Patching	0.6944
Simple Tree / Farthest Pair Patching with Tie-Breaking	0.6966
DFS Tree / Simple Repairing	0.7037
DFS Tree / Farthest Pair Repairing	0.7086
Simple Tree / Farthest Pair Patching	0.7118
DFS Tree / Simple Patching	0.7203
DFS Tree / Farthest Pair Patching	0.7226
DFS Tree / Farthest Pair Patching with Tie-Breaking	0.7241
Far Edge Deletion	0.7347
Popular Edges Tree / Distinct Tree Construction	0.7825
BFS Tree / Distinct Tree Construction	0.7834
Simple Tree / Distinct Tree Construction	0.7870
Low Degree Pairs	0.8022
Tainting	0.8071
Popular Edges	0.8285
DFS Tree / Distinct Tree Construction	0.8289
Random Edges (baseline)	0.8446
Neighbourhood Tainting	1.0000

Table 5.9: Edge ratios for all algorithms over all grids.

Algorithm	Edge Ratio
BFS Tree / Simple Repairing	0.5195
BFS Tree / Farthest Pair Repairing	0.5202
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.5213
BFS Tree / Farthest Pair Patching	0.5245
BFS Tree / Simple Patching	0.5261
Simple Tree / Simple Repairing	0.5278
Simple Tree / Farthest Pair Repairing	0.5292
Simple Tree / Farthest Pair Patching with Tie-Breaking	0.5300
Popular Edges Tree / Simple Repairing	0.5354
Simple Tree / Simple Patching	0.5357
Simple Tree / Farthest Pair Patching	0.5377
Popular Edges Tree / Farthest Pair Repairing	0.5401
Popular Edges Tree / Simple Patching	0.5447
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	0.5459
Popular Edges Tree / Farthest Pair Patching	0.5464
DFS Tree / Simple Repairing	0.5511
DFS Tree / Farthest Pair Repairing	0.5525
DFS Tree / Simple Patching	0.5593
DFS Tree / Farthest Pair Patching	0.5594
DFS Tree / Farthest Pair Patching with Tie-Breaking	0.5595
Far Edge Deletion	0.5721
Low Degree Pairs	0.6206
BFS Tree / Distinct Tree Construction	0.6381
Simple Tree / Distinct Tree Construction	0.6563
Popular Edges	0.6755
Popular Edges Tree / Distinct Tree Construction	0.6789
Random Edges (baseline)	0.6891
DFS Tree / Distinct Tree Construction	0.6925
Tainting	0.7189
Neighbourhood Tainting	1.0000

Table 5.10: Edge ratios for all algorithms over all hypercubes.

Algorithm	Edge Ratio
BFS Tree / Simple Repairing	0.6052
BFS Tree / Farthest Pair Repairing	0.6086
Popular Edges Tree / Farthest Pair Repairing	0.6156
Popular Edges Tree / Simple Repairing	0.6162
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.6176
BFS Tree / Farthest Pair Patching	0.6182
BFS Tree / Simple Patching	0.6191
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	0.6262
Popular Edges Tree / Farthest Pair Patching	0.6270
Popular Edges Tree / Simple Patching	0.6323
Far Edge Deletion	0.6385
Simple Tree / Farthest Pair Repairing	0.6399
Simple Tree / Simple Repairing	0.6417
Simple Tree / Simple Patching	0.6560
Simple Tree / Farthest Pair Patching with Tie-Breaking	0.6564
Simple Tree / Farthest Pair Patching	0.6578
DFS Tree / Farthest Pair Repairing	0.6758
DFS Tree / Simple Repairing	0.6800
DFS Tree / Farthest Pair Patching	0.6962
DFS Tree / Simple Patching	0.6962
DFS Tree / Farthest Pair Patching with Tie-Breaking	0.6964
BFS Tree / Distinct Tree Construction	0.7118
Tainting	0.7141
Popular Edges Tree / Distinct Tree Construction	0.7203
Simple Tree / Distinct Tree Construction	0.7376
Low Degree Pairs	0.7946
DFS Tree / Distinct Tree Construction	0.7949
Random Edges (baseline)	0.8253
Neighbourhood Tainting	0.8457
Popular Edges	0.8604

Table 5.11: Edge ratios for all algorithms over all X-trees.

Algorithm	Edge Ratio
BFS Tree / Simple Repairing	0.4735
Popular Edges Tree / Simple Repairing	0.4801
BFS Tree / Farthest Pair Repairing	0.4802
BFS Tree / Simple Patching	0.4820
Popular Edges Tree / Farthest Pair Repairing	0.4831
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.4850
BFS Tree / Farthest Pair Patching	0.4861
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	0.4909
Popular Edges Tree / Farthest Pair Patching	0.4911
Popular Edges Tree / Simple Patching	0.4914
Simple Tree / Farthest Pair Repairing	0.5105
Simple Tree / Simple Repairing	0.5127
Far Edge Deletion	0.5176
Simple Tree / Simple Patching	0.5241
Simple Tree / Farthest Pair Patching with Tie-Breaking	0.5250
Simple Tree / Farthest Pair Patching	0.5253
DFS Tree / Farthest Pair Repairing	0.5429
DFS Tree / Simple Repairing	0.5485
DFS Tree / Farthest Pair Patching	0.5590
DFS Tree / Farthest Pair Patching with Tie-Breaking	0.5598
DFS Tree / Simple Patching	0.5610
BFS Tree / Distinct Tree Construction	0.5937
Popular Edges Tree / Distinct Tree Construction	0.5984
Tainting	0.6196
Simple Tree / Distinct Tree Construction	0.6292
Low Degree Pairs	0.6866
DFS Tree / Distinct Tree Construction	0.6895
Random Edges (baseline)	0.7169
Popular Edges	0.7307
Neighbourhood Tainting	0.7585

Table 5.12: Edge ratios for all algorithms over all pyramids.

Algorithm	Edge Ratio
BFS Tree / Farthest Pair Repairing	0.2202
BFS Tree / Farthest Pair Patching	0.2206
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.2212
BFS Tree / Simple Repairing	0.2228
BFS Tree / Simple Patching	0.2240
Simple Tree / Simple Repairing	0.2393
Simple Tree / Simple Patching	0.2416
Simple Tree / Farthest Pair Repairing	0.2422
Simple Tree / Farthest Pair Patching with Tie-Breaking	0.2445
Simple Tree / Farthest Pair Patching	0.2452
Popular Edges Tree / Simple Repairing	0.2453
Popular Edges Tree / Simple Patching	0.2472
Popular Edges Tree / Farthest Pair Repairing	0.2503
Popular Edges Tree / Farthest Pair Patching	0.2520
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	0.2528
Far Edge Deletion	0.2557
DFS Tree / Simple Repairing	0.2690
DFS Tree / Simple Patching	0.2715
DFS Tree / Farthest Pair Repairing	0.2721
DFS Tree / Farthest Pair Patching	0.2752
DFS Tree / Farthest Pair Patching with Tie-Breaking	0.2758
BFS Tree / Distinct Tree Construction	0.2884
Simple Tree / Distinct Tree Construction	0.3291
Popular Edges Tree / Distinct Tree Construction	0.3379
DFS Tree / Distinct Tree Construction	0.3627
Low Degree Pairs	0.3764
Random Edges (baseline)	0.4231
Tainting	0.4830
Popular Edges	0.5346
Neighbourhood Tainting	0.7101

Table 5.13: Edge ratios for all algorithms over all random graphs.

Algorithm	Edge Ratio
BFS Tree / Simple Repairing	0.4608
BFS Tree / Farthest Pair Repairing	0.4624
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.4679
BFS Tree / Simple Patching	0.4688
BFS Tree / Farthest Pair Patching	0.4690
Popular Edges Tree / Simple Repairing	0.4761
Simple Tree / Simple Repairing	0.4801
Popular Edges Tree / Farthest Pair Repairing	0.4802
Simple Tree / Farthest Pair Repairing	0.4851
Popular Edges Tree / Simple Patching	0.4858
Popular Edges Tree / Farthest Pair Patching with Tie-Breaking	0.4879
Popular Edges Tree / Farthest Pair Patching	0.4879
Simple Tree / Simple Patching	0.4886
Simple Tree / Farthest Pair Patching with Tie-Breaking	0.4899
Simple Tree / Farthest Pair Patching	0.4960
Far Edge Deletion	0.5081
DFS Tree / Simple Repairing	0.5083
DFS Tree / Farthest Pair Repairing	0.5103
DFS Tree / Simple Patching	0.5183
DFS Tree / Farthest Pair Patching	0.5203
DFS Tree / Farthest Pair Patching with Tie-Breaking	0.5211
BFS Tree / Distinct Tree Construction	0.5576
Simple Tree / Distinct Tree Construction	0.5818
Popular Edges Tree / Distinct Tree Construction	0.5820
Low Degree Pairs	0.6115
DFS Tree / Distinct Tree Construction	0.6230
Tainting	0.6501
Random Edges (baseline)	0.6562
Popular Edges	0.6974
Neighbourhood Tainting	0.8547

Table 5.14: Edge ratios for all algorithms over all graphs.

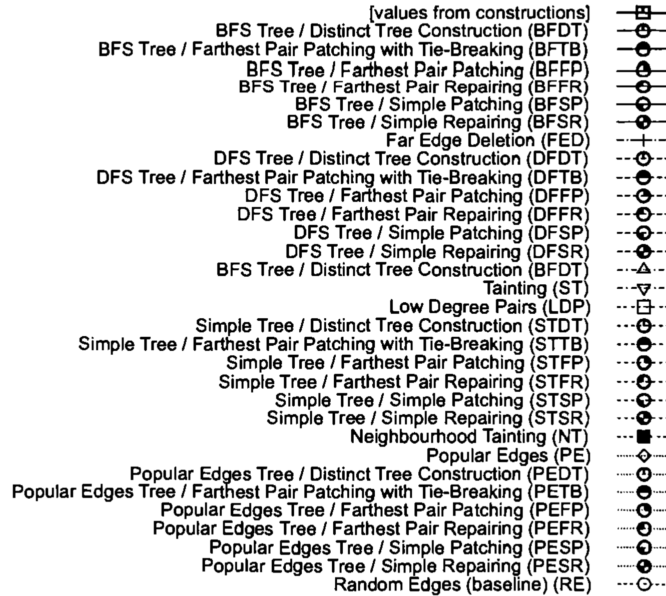


Figure 5.2: Key to the plots.

Figures 5.3, 5.4, 5.5, and 5.6 plot the number of edges in the spanners against the delay parameter for which the spanner was constructed for the input graphs XT_9, P_5, Q_9, and $G_{16,16}$ respectively. In each figure, the theoretical constructions are illustrated with a heavy line and square data points symbols with a vertical line in them, and the rest of the algorithms are illustrated with finer lines, and different data point symbols; refer to Figure 5.2 for a complete key. Since there are two different constructions for hypercubes, we illustrate them both in Figure 5.5. Figures 5.5 and 5.6 do not illustrate the case where $d = 1$. This case is omitted for the sake of clarity; since those graphs are bipartite, they do not contain any non-trivial spanner with $d = 1$.

For X-trees, the best spanners produced by the algorithms are no better than the spanners produced by the constructions, with the exception of the case where $d = 1$; the construction for X-trees does not produce spanners where $d = 1$. The difference between the number of edges in the best spanners and the number of edges in the constructed spanners does diminish as the delay parameter increases, however this can be attributed to fact that as the permissible delay increases, the minimum number of edges in the spanner decreases. Once the permissible delay is large enough, a spanning

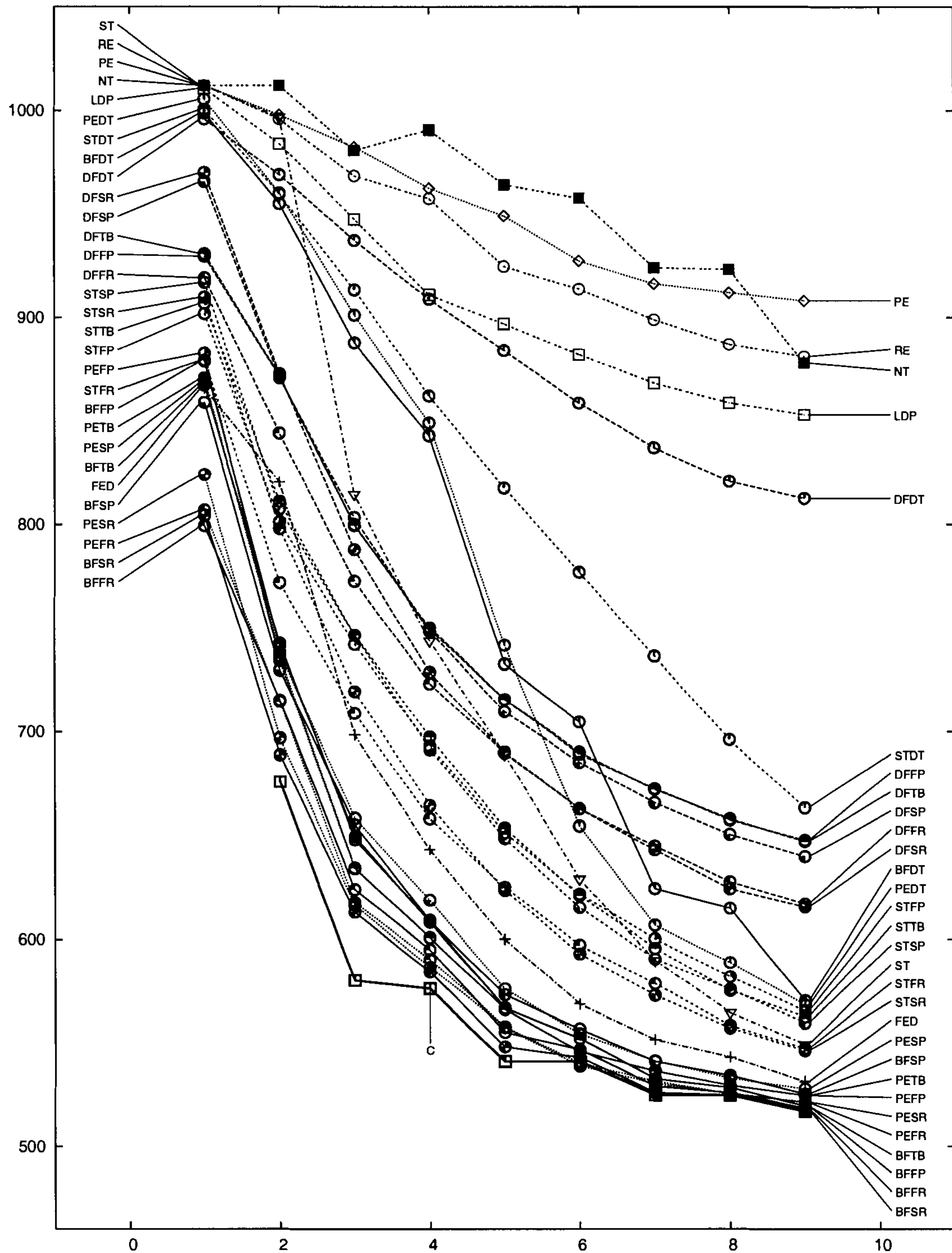


Figure 5.3: Spanner edges vs. delay (parameter), for XT_9.

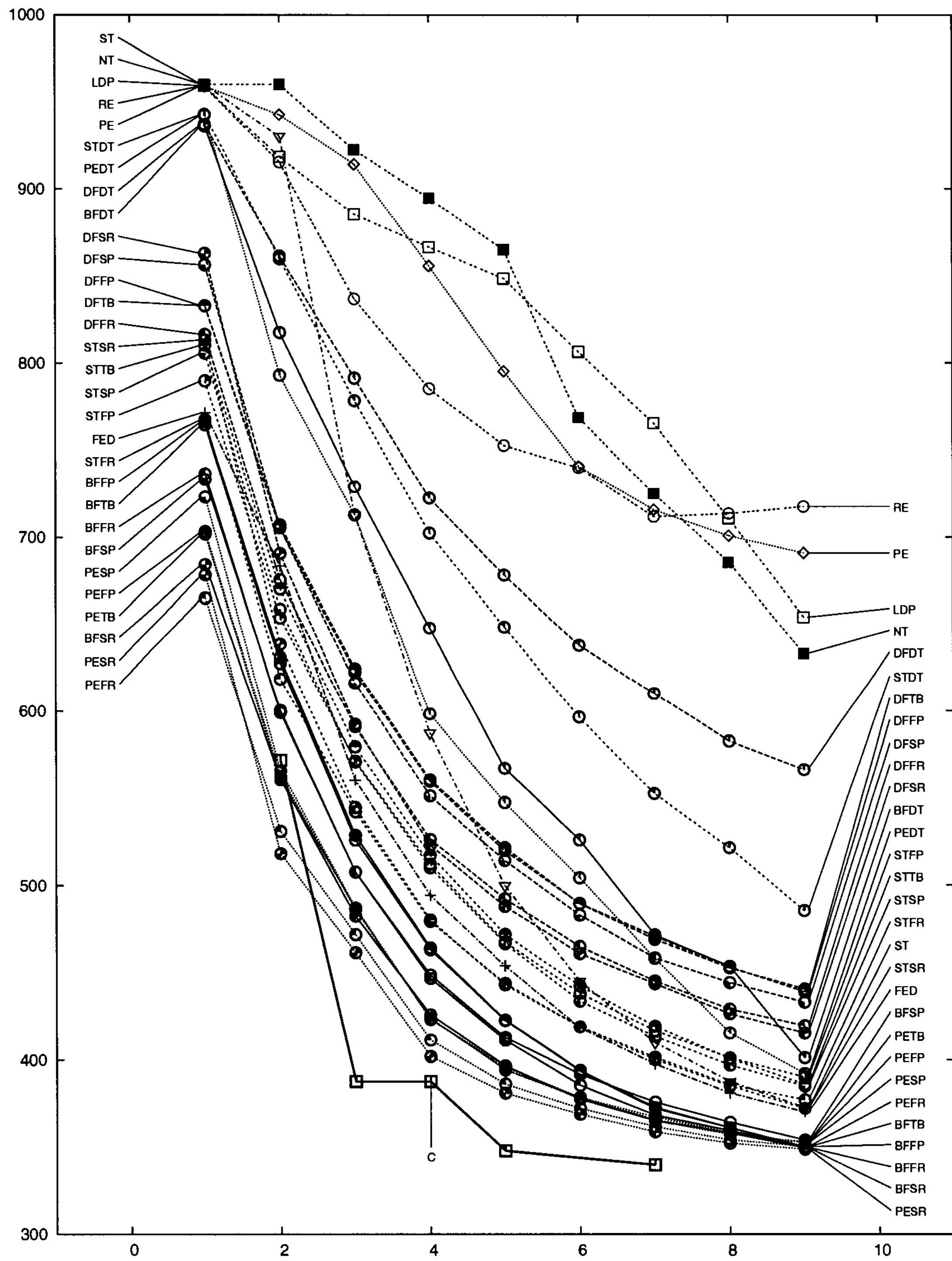


Figure 5.4: Spanner edges vs. delay (parameter), for P_5.

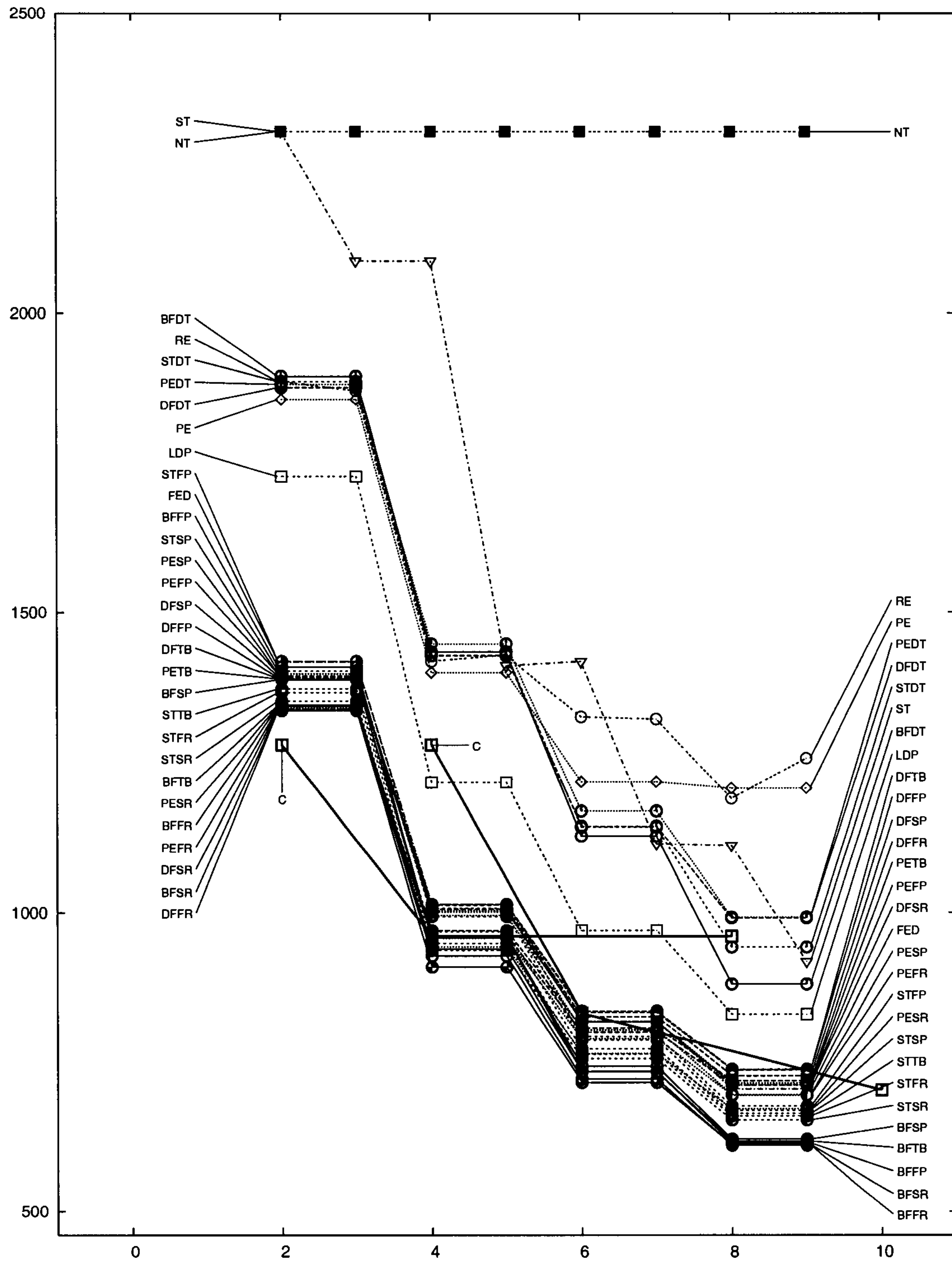


Figure 5.5: Spanner edges vs. delay (parameter), for Q_9.

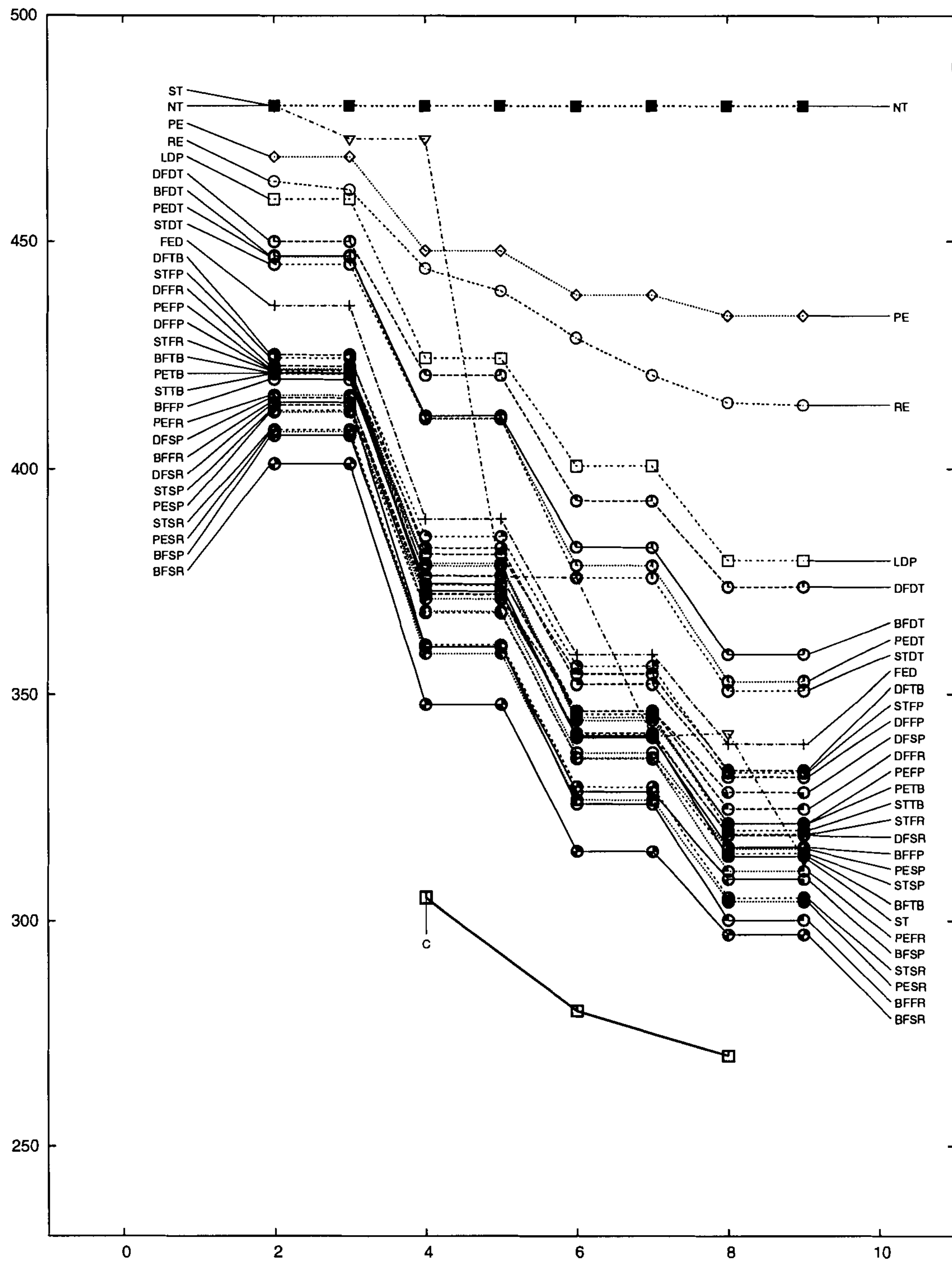


Figure 5.6: Spanner edges vs. delay (parameter), for $G_{16,16}$.

tree will serve as a $(d+x)$-spanner, and since a spanning tree is a smallest possible spanning subgraph, no construction will produce a smaller spanner. This works to the advantage of the 2-stage algorithms when d is large, since they first construct a spanning tree, and this tree will either be a spanner in its own right, or will require only a small number of additional edges to meet its delay requirement.

In the case of pyramids, we note that the results are somewhat similar to the results for X-trees. Like X-trees, the constructions for pyramids do not produce spanners with $d = 1$. We do note, however, that relative to the spanner with $d = 2$ produced by the construction, several of our algorithms produce spanners with fewer edges. Those well-performing algorithms are part of the Popular Edges family of 2-stage algorithms which performed well on the pyramid graphs, as illustrated in Table 5.6.

In the case of hypercubes, we note that for $d \geq 4$, a significant number of algorithms produced spanners with fewer edges than either of the constructions. A variety of algorithms are in this group of top-performers, although it should be noted that they are all 2-stage algorithms. We cannot say to what delay value this performance trend will continue out, but the observation made previously about the convergence of spanning trees and optimal spanners as delays increase also holds in this case.

For grids, the comparison we will make is not as direct as the comparisons for the previous graph classes. The difference results from the nature of the spanner constructions provided by Liestman and Shermer in [15]. A description of their basic construction is given in Section 2.2; we present a capsule review of it here. The construction subdivides an infinite grid into rectangular areas by periodically designating sequences of horizontal and vertical edges as *highways* and including those highway edges in the spanner. The areas between these highways are then populated with edges in a fixed pattern which they call a *tile*, where each tile is a $(d+x)$-spanner of a finite grid of the size of the "spaces" between the highways. These tiles are placed repetitively throughout the grid. The resultant graph is a $(d+x)$-spanner of the infinite grid. It is then possible to take a finite subgraph of this $(d+x)$-spanner of the infinite grid which is a $(d+x)$-spanner of the desired grid.

Liestman and Shermer present several different types of tiles, and each has its particular advantages. We constructed $(d+x)$-spanners using their "comb" tiles for

Heuristic	Overall Rankings "Top 10"	Edge Ratio
BFS Tree / Simple Patching	3	4
BFS Tree / Farthest Pair Patching	5	5
BFS Tree / Farthest Pair Patching with Tie-Breaking	4	3
BFS Tree / Simple Repairing	2	1
BFS Tree / Farthest Pair Repairing	1	2
Simple Tree / Simple Repairing	6	7
Simple Tree / Farthest Pair Repairing	9	9
Popular Edges Tree / Simple Repairing	7	6
Popular Edges Tree / Farthest Pair Repairing	8	8

Table 5.15: "Good" heuristics according to the primary criterion

$d = 4, 6, 8$; their constructions do not provide a tile for $d = 2$. As is clear from the plot, our heuristics do not perform as well as the construction on the grid indicated. Although we do not present a comparison of our spanners for grids with more than two dimensions here, we believe that the spanners produced by our heuristics for such graphs would not do significantly better against the spanners from Liestman and Shermer's constructions than our two-dimensional grid spanners did against the spanners from their two-dimensional grid constructions.

5.3.4 List of good heuristics

Table 5.15 gives the heuristics that are good, according to the primary criteria. These heuristics represent the top nine heuristics from the overall rankings, according to both the "top 10" listings for all graphs and the average edge ratio listing for all graphs. Additionally, these algorithms, in general, did well on the hypercube graphs, which was the only class of graphs where the heuristics did significantly better than the constructions. We will analyse these algorithms with the secondary criteria.

The choice of having nine algorithms in this list was partially arbitrary. These algorithms hold the top nine positions on both of the overall rankings, and there is a significant difference in the positions of the algorithms following them on each list. They also appear in at least 60% of the "top 10" rankings. While the other algorithms may have application in certain situations, these algorithms are the most

Algorithm	Average Delay
Simple Tree / Farthest Pair Repairing	0.9888
BFS Tree / Simple Patching	0.9900
Simple Tree / Simple Repairing	0.9911
BFS Tree / Farthest Pair Patching	1.0032
BFS Tree / Simple Repairing	1.0108
BFS Tree / Farthest Pair Repairing	1.0173
BFS Tree / Farthest Pair Patching with Tie-Breaking	1.0175
Popular Edges Tree / Simple Repairing	1.0383
Popular Edges Tree / Farthest Pair Repairing	1.0462

Table 5.16: Average delay for the good algorithms over all hypercubes

likely to produce good spanners of general graphs, at least according to the primary criteria (*i.e.* the number of edges in their spanners.)

5.4 Secondary Criteria

In this section, we evaluate the heuristics given in Table 5.15 on the secondary criteria given in Section 3.1.2.

5.4.1 Average Delay

Overall, there was very little difference in the average delay of the spanners produced by the algorithms for the hypercubes; Table 5.16 illustrates this. There was very little difference in the performance of the algorithms over different delay parameters, and while this table gives the average delay for the spanners as averaged over all input delay parameters (*i.e.* for $d = 1 \ldots 9$), the results for the individual delay parameters are all very similar. Figure 5.7 plots the average delay of the spanners for Q_9 versus their delay parameter, and is illustrative of the general pattern. The distinction gets slightly greater as d increases, but not significantly.

There was some distinction between the average delays of the grids, X-trees and pyramids. Tables 5.17, 5.18 and 5.19 illustrate this distinction; again, there was almost no distinction between the algorithms over the different delay parameters. In

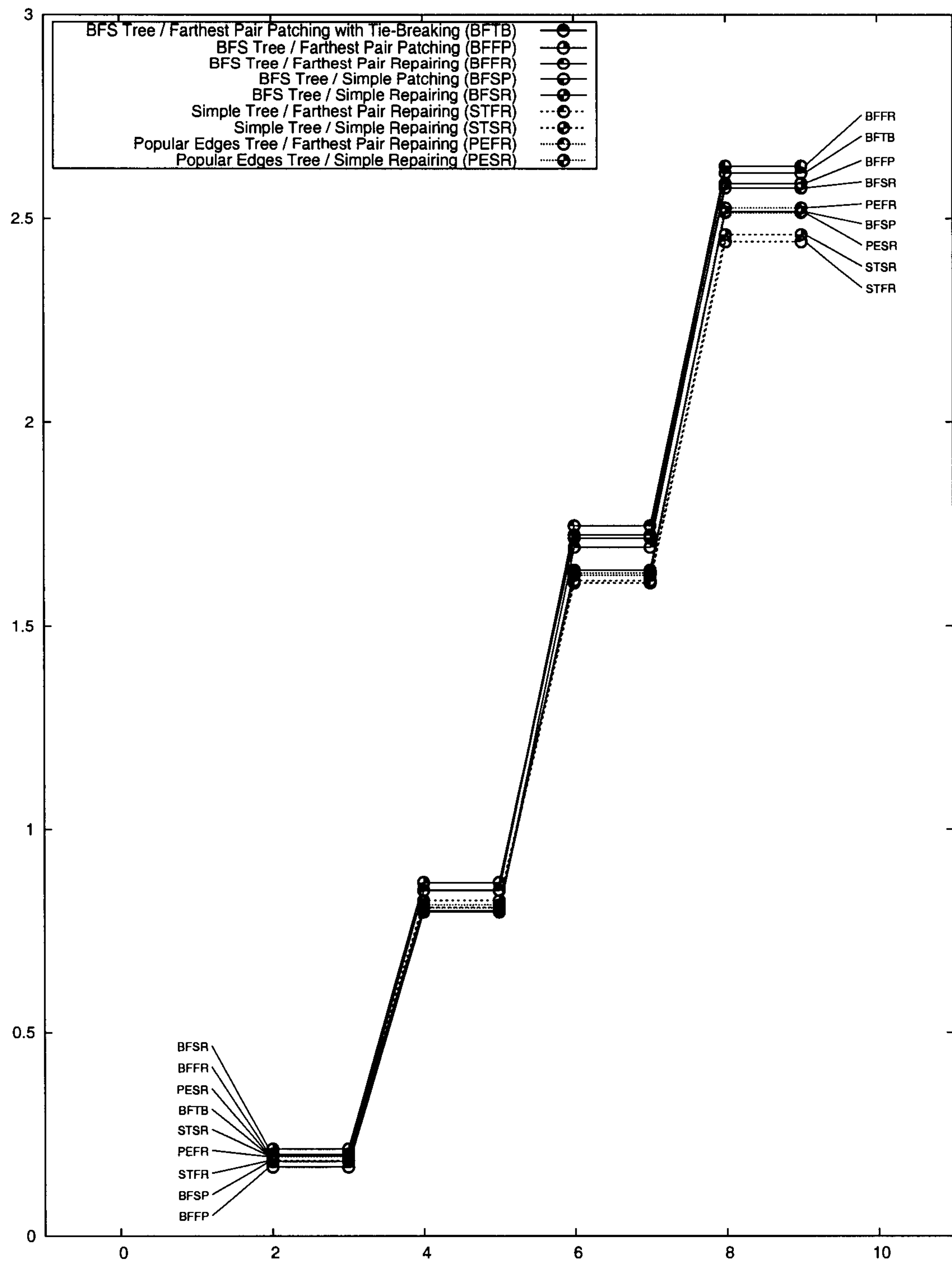


Figure 5.7: Average delay vs. parameter delay for Q_9

Algorithm	Average Delay
BFS Tree / Simple Patching	0.6485
BFS Tree / Farthest Pair Patching	0.6681
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.6775
BFS Tree / Simple Repairing	0.7075
BFS Tree / Farthest Pair Repairing	0.7220
Simple Tree / Farthest Pair Repairing	0.7360
Simple Tree / Simple Repairing	0.7625
Popular Edges Tree / Simple Repairing	0.7815
Popular Edges Tree / Farthest Pair Repairing	0.8088

Table 5.17: Average delay for the good algorithms over all grids

Algorithm	Average Delay
BFS Tree / Simple Patching	0.7740
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.8106
BFS Tree / Farthest Pair Patching	0.8110
BFS Tree / Simple Repairing	0.8224
BFS Tree / Farthest Pair Repairing	0.8486
Popular Edges Tree / Simple Repairing	0.8763
Simple Tree / Farthest Pair Repairing	0.8959
Popular Edges Tree / Farthest Pair Repairing	0.9017
Simple Tree / Simple Repairing	0.9182

Table 5.18: Average delay for the good algorithms over all X-trees

Algorithm	Average Delay
BFS Tree / Simple Patching	0.7582
BFS Tree / Farthest Pair Patching	0.7833
BFS Tree / Simple Repairing	0.7840
BFS Tree / Farthest Pair Patching with Tie-Breaking	0.7860
BFS Tree / Farthest Pair Repairing	0.8013
Popular Edges Tree / Simple Repairing	0.9287
Popular Edges Tree / Farthest Pair Repairing	0.9695
Simple Tree / Farthest Pair Repairing	0.9761
Simple Tree / Simple Repairing	0.9973

Table 5.19: Average delay for the good algorithms over all pyramids

Algorithm	Average Delay
BFS Tree / Simple Patching	1.1121
BFS Tree / Farthest Pair Patching	1.1133
BFS Tree / Farthest Pair Patching with Tie-Breaking	1.1133
BFS Tree / Farthest Pair Repairing	1.1141
BFS Tree / Simple Repairing	1.1144
Simple Tree / Farthest Pair Repairing	1.6151
Simple Tree / Simple Repairing	1.6180
Popular Edges Tree / Simple Repairing	1.7012
Popular Edges Tree / Farthest Pair Repairing	1.7182

Table 5.20: Average delay for the good algorithms over all random graphs

general, BFS Tree / Simple Patching had a smaller delay than the others, but three other BFS Tree algorithms had almost identical average delays which were not much greater. We do wish to note that the overall distinction in average delay at higher delay parameter values tends to disappear as the number of edges in the source graphs increases. This can be seen by comparing figures 5.8 and 5.9.

There was a good deal of distinction of average delays within the random graphs. In general, the BFS Tree algorithms performed uniformly well as compared to the others. Table 5.20 illustrates the overall difference; again, there was almost no distinction between the algorithms over the different delay parameters. Figures 5.10, 5.11, and 5.12 show that the distinction decreases as the number of edges in the original graph increases, but also that the BFS Tree algorithms continue to perform better than the others.

Summarizing the results given above, it appears that the BFS Tree based algorithms do slightly better overall than the others, with BFS Tree / Simple Patching being the best by a slight margin. They are most effective on the random graphs, and there is almost no distinction between the effectiveness of the algorithms on hypercubes. We do not have a concrete explanation of why there is little distinction in average delay for the hypercubes, but suspect that their highly regular structure may result in different algorithms producing similar spanners.

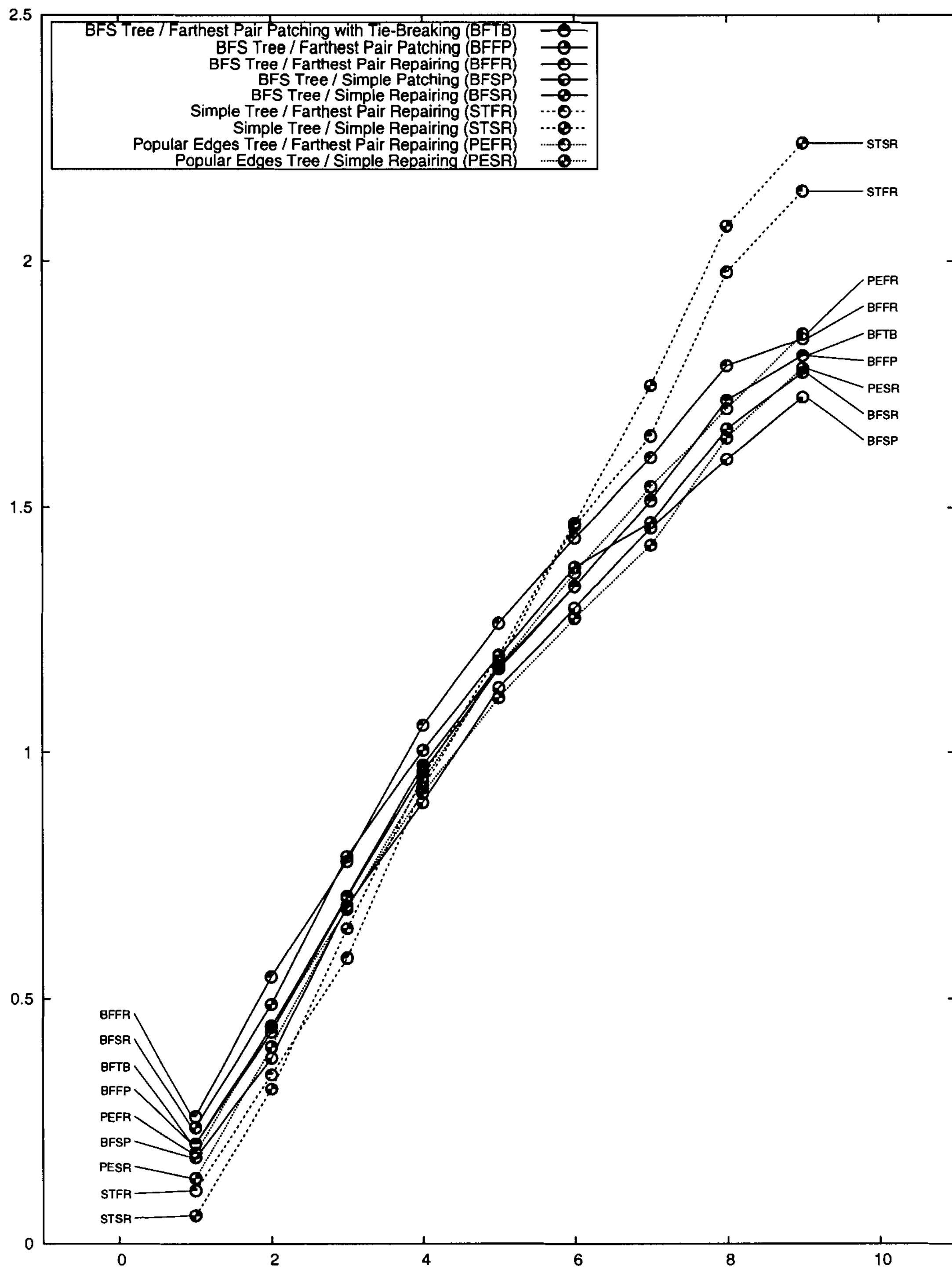


Figure 5.8: Average delay vs. parameter delay for XT_7

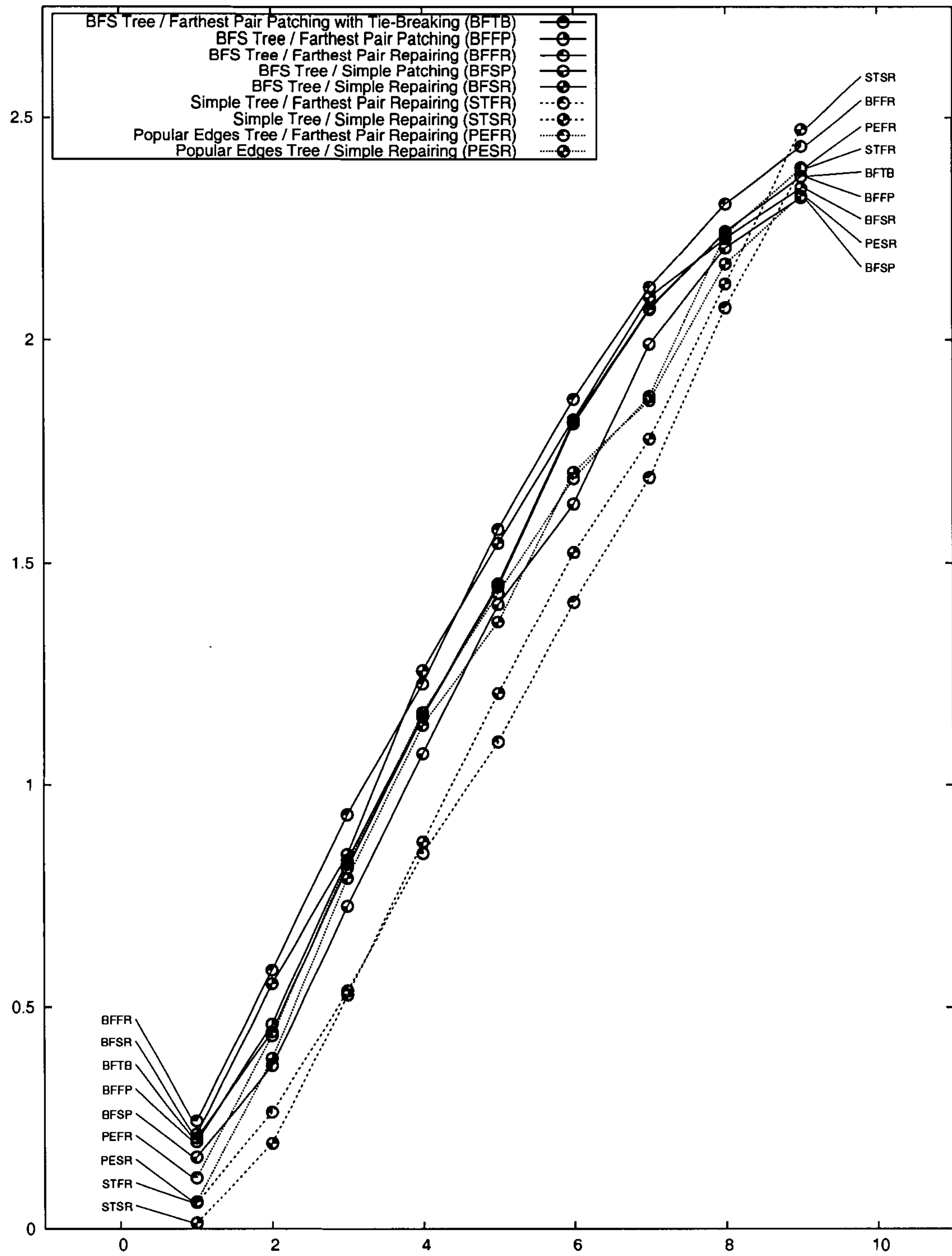


Figure 5.9: Average delay vs. parameter delay for XT_9

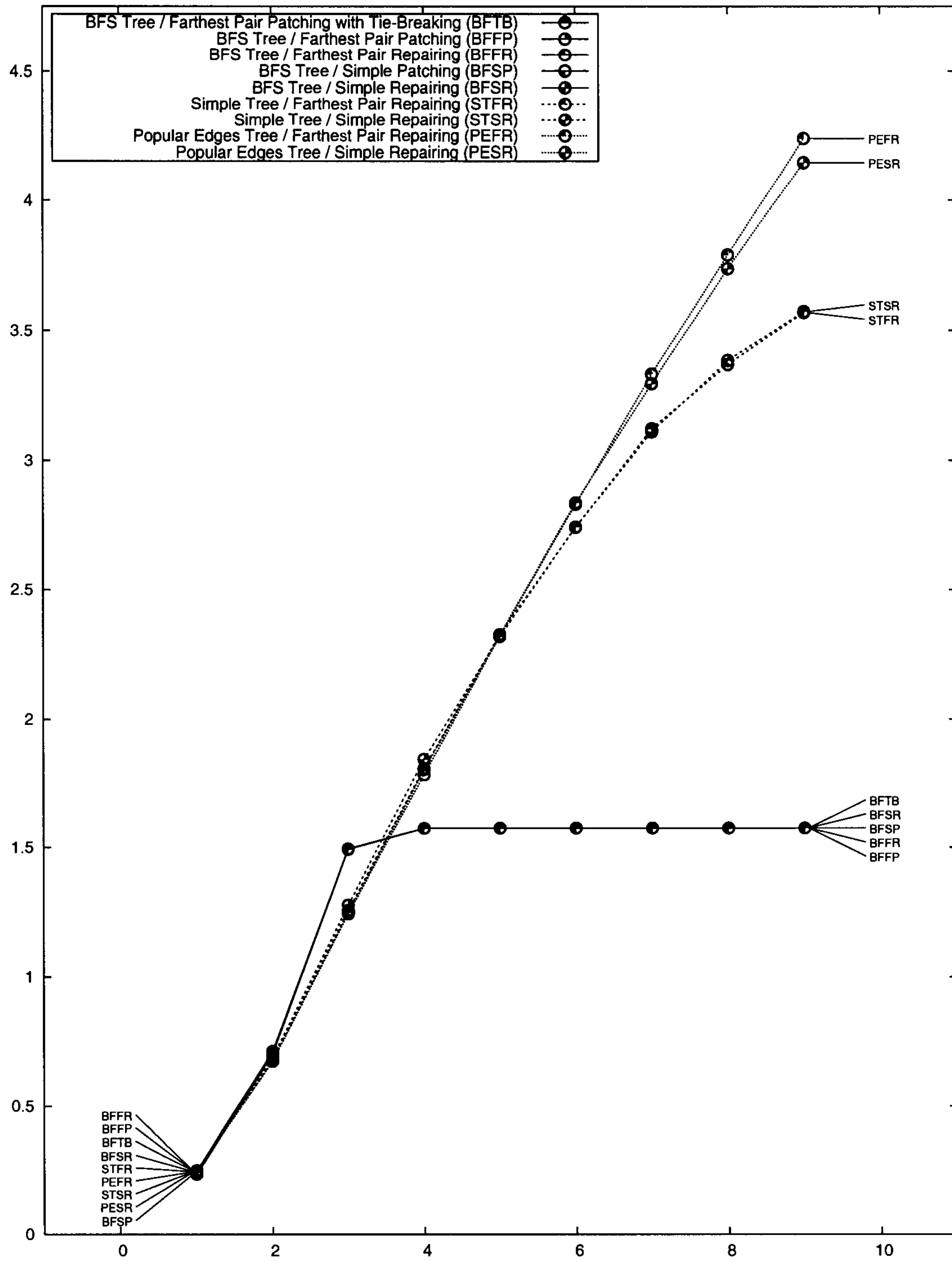


Figure 5.10: Average delay vs. parameter delay for $R_{200}(0.10)$

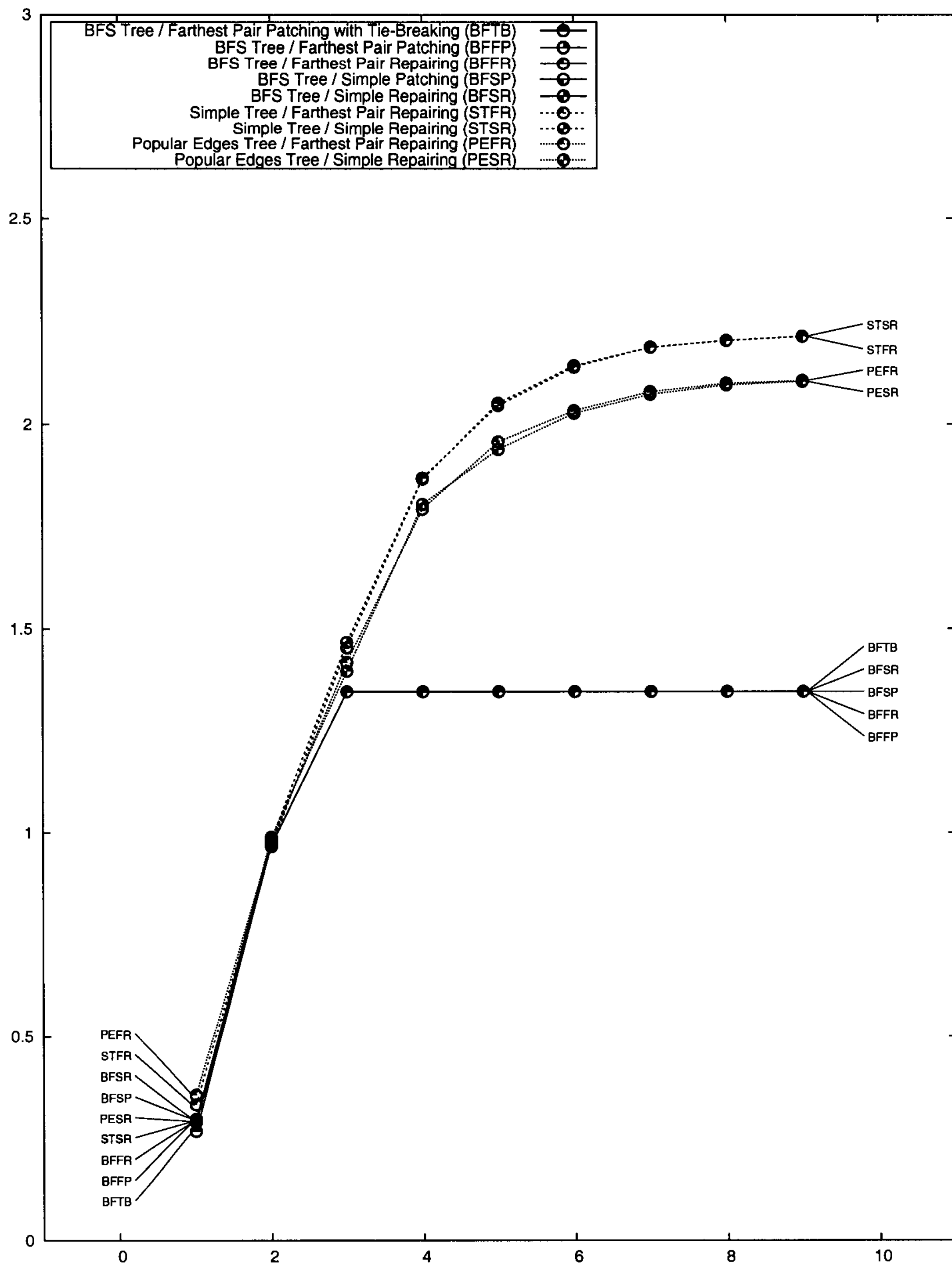


Figure 5.11: Average delay vs. parameter delay for $R_{200}(0.30)$

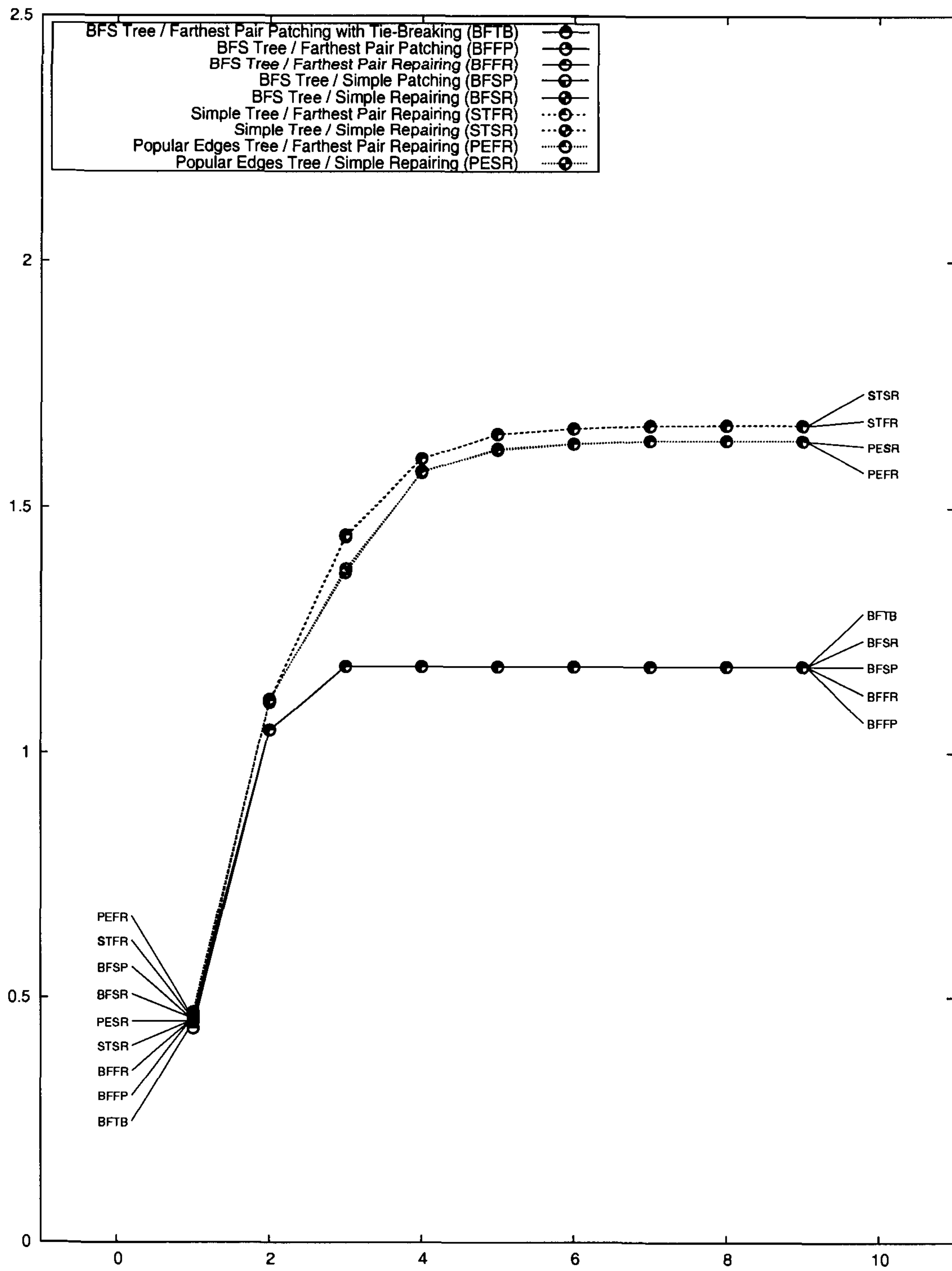


Figure 5.12: Average delay vs. parameter delay for $R_{200}(0.50)$

5.4.2 Maximum Degree

In general, the results for maximum degree show that the BFS Tree algorithms perform exceptionally poorly on this metric. This is not unexpected, however, since the breadth-first search algorithm that we use intentionally chooses a vertex of maximum degree as the root of the tree. Even if a root of smaller degree were chosen, it would not likely have a significant effect on the maximum degree of the spanner, simply because a breadth-first search attempts to follow all edges from any given candidate vertex.

Of the remaining algorithms, Popular Edges Tree / Farthest Pair Repairing generally does the best, followed closely by Popular Edges Tree / Simple Repairing. In some cases, most notably the 2-dimensional grids, they produce spanners with the same maximum degree as the other good algorithms. In some other cases, they produce spanners with much smaller degrees; Figure 5.13 illustrates $R_{200}(0.15)$, where they produce spanners with maximum degree $< 1/2$ the maximum degree of the spanners produced by the BFS Tree algorithms. This figure also illustrates the maximum degree of the spanners produced by the Low Degree Pairs heuristic; its results are included as a point of comparison. Clearly the lowest maximum degree obtained by the good algorithms is still higher than that which can be obtained by an algorithm which attempts to balance the number of edges in the spanner with the maximum degree of the spanner.

5.4.3 Running Time

In general, the BFS Tree / Farthest Pair Patching algorithm has the lowest running time. The algorithms BFS Tree / Farthest Pair Patching with Tie-Breaking and BFS Tree / Farthest Pair Repairing are tied for second lowest running time. This pattern holds for nearly all of the input graphs considered. Of the remaining algorithms, Simple Tree / Farthest Pair Repairing and Popular Edges Tree / Farthest Pair Repairing are the next fastest algorithms in a modest majority of the cases. All of the remaining algorithms tend to be much slower than the first two. Figure 5.14 illustrates the running time for algorithms on Q_8. The fastest algorithm required 2-3

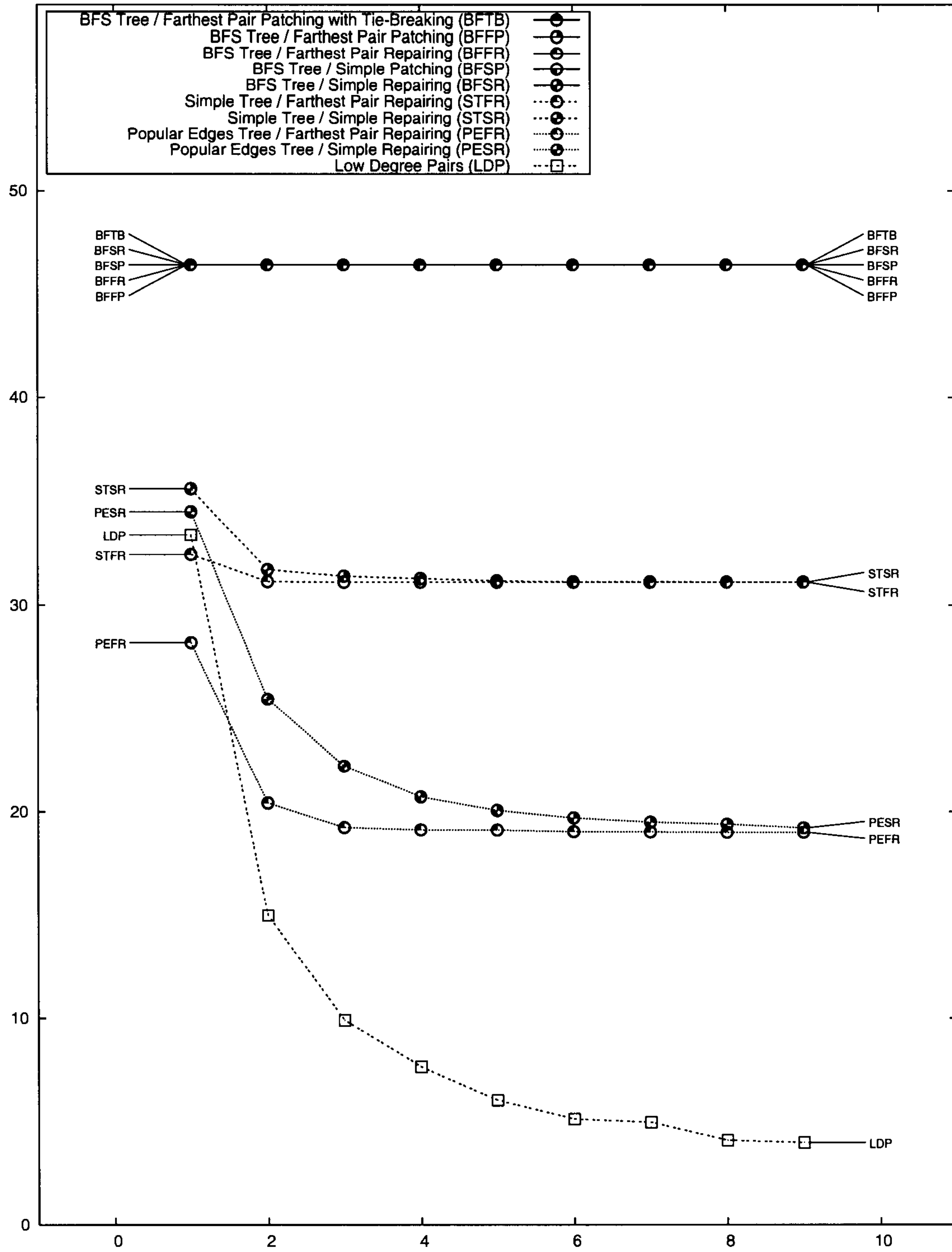


Figure 5.13: Maximum degree vs. parameter delay for $R_{200}(0.15)$

seconds, whereas the slowest algorithm required 35 seconds in one case, and never much less than 15 seconds. This situation is amplified in Q_9 (as illustrated in Figure 5.15,) where the fastest algorithm required never more than 18 seconds, but the worst algorithm required almost 800 seconds!

Figure 5.16 illustrates the trend in running times for the good algorithms over all the instances of hypercubes. While the times used in this graph are averaged over all delay parameter values, the trend is the same for the specific delay values. The running times for the four algorithms with "Simple" second stages grow much faster than for the other five "Farthest" based algorithms. This data indicates that judicious choice of the vertex pairs to be patched in the second stage is critical for running time.

5.5 Summary

Considering just the primary criteria, the best-performing algorithms are BFS Tree / Farthest Pair Repairing and BFS Tree / Simple Repairing, which dominate both the "top 10" and the edge ratio rankings. Beyond that, the other algorithms identified in Table 5.15 are also good algorithms.

Considering the secondary criteria, the BFS Tree algorithms did well with respect to average delay, while they did extremely poorly with respect to maximum degree. The various Farthest Pair Patching and Farthest Pair Repairing algorithms did well with respect to running time, while the other Simple Patching and Simple Repairing algorithms did poorly, often extremely so.

If one wanted to identify a single algorithm as an overall winner, the best choice would likely be BFS Tree / Farthest Pair Repairing. If one wanted an algorithm which was good overall, but which also had reasonable maximum delay results, then Popular Edges Tree / Farthest Pair Repairing would be a suitable choice.

In general, these experimental results reveal certain trends. First, the 2-stage algorithms proposed here are clearly dominant against the 1-stage algorithms. Since all of these algorithms are the results of much careful thought, it would seem likely that the overall 2-stage approach, creation of a spanning tree followed by adding additional edges as necessary, is a good approach for constructing algorithms for producing good

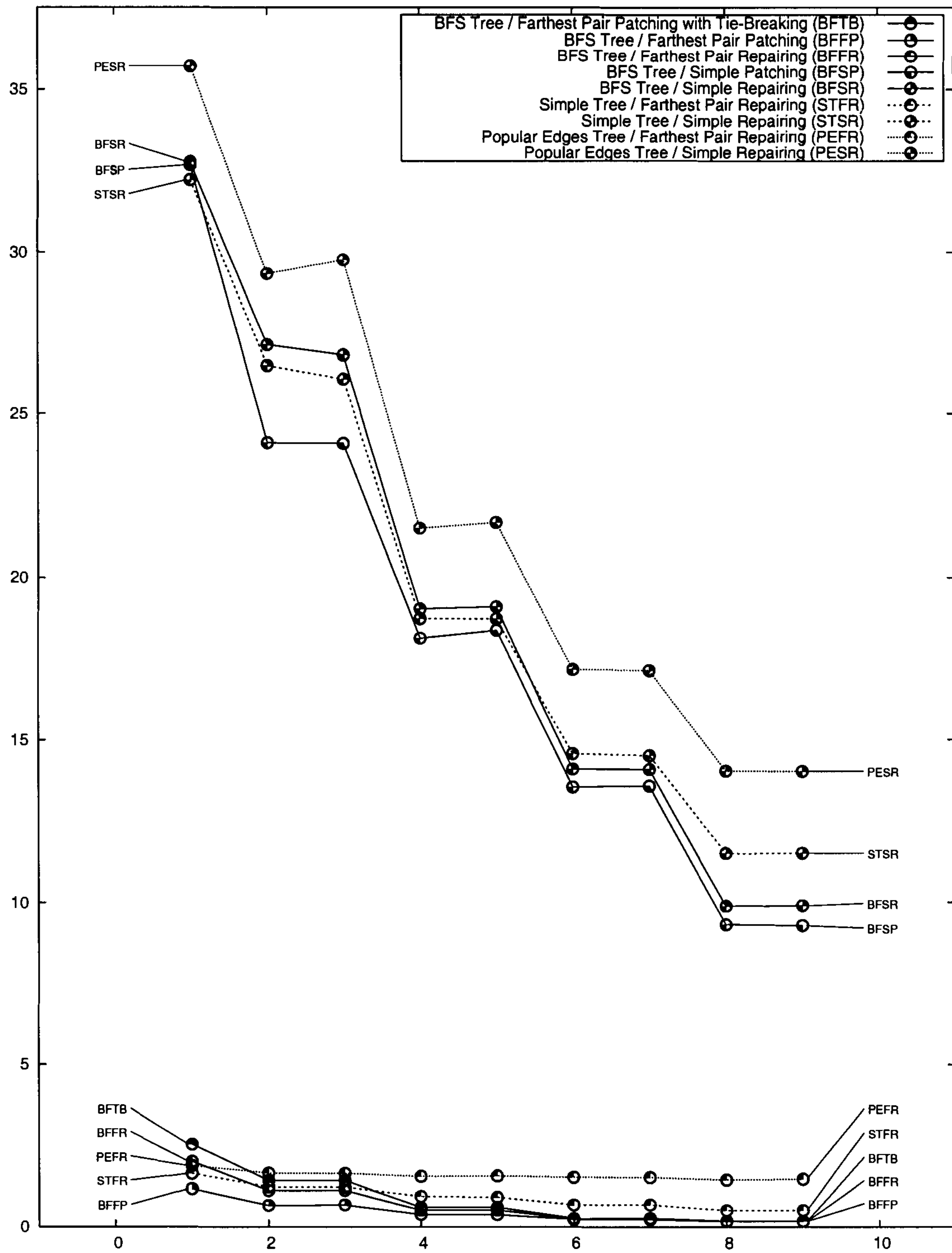


Figure 5.14: Running Time vs. parameter delay for Q_8

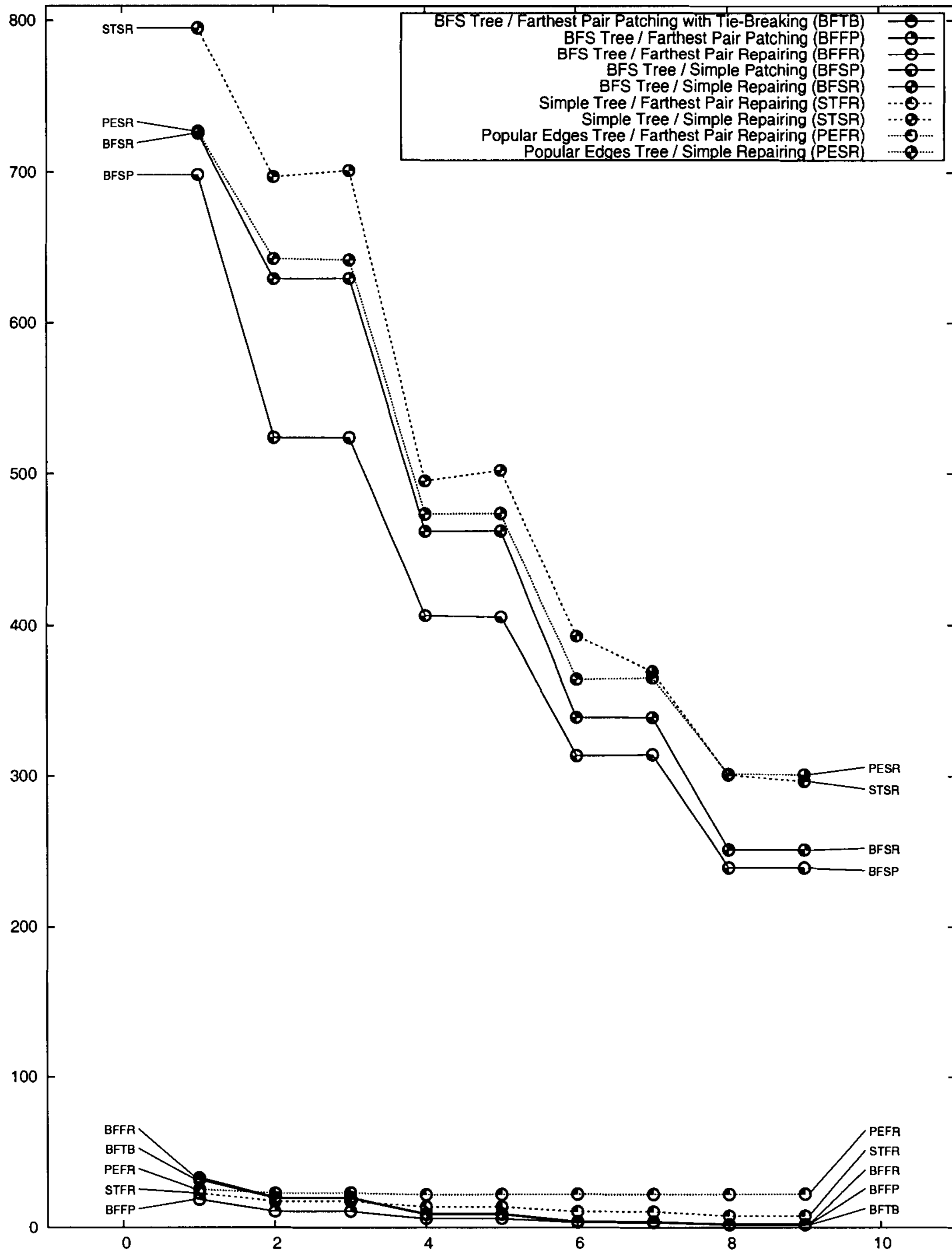


Figure 5.15: Running Time vs. parameter delay for Q_9

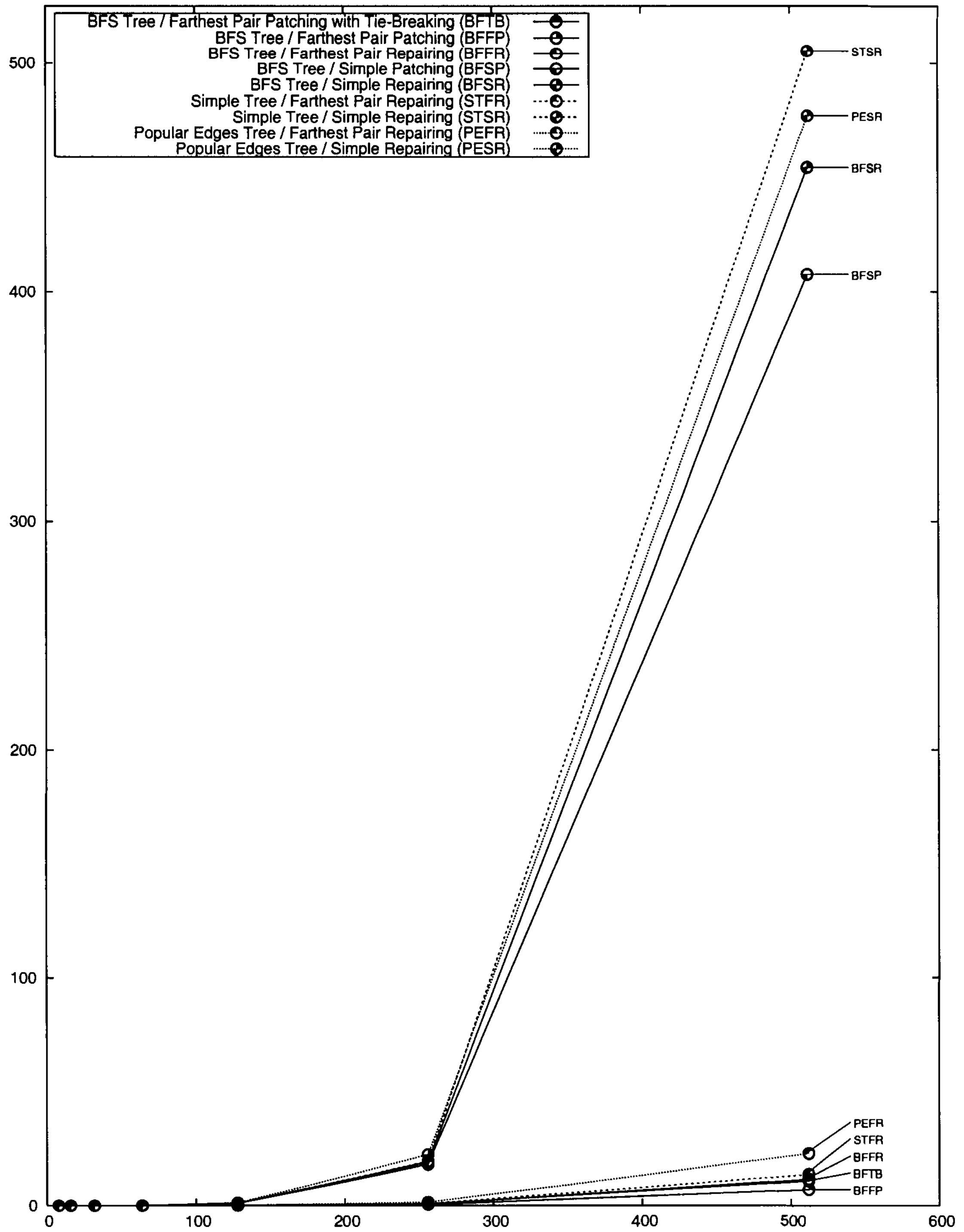


Figure 5.16: Average Running Time vs. Instance Size ($|V|$) for the Hypercubes

spanners. The second major trend is that the "repairing" approach to the second stage of the 2-stage algorithms is a good choice, as it tends to produce smaller spanners. Finally, careful selection of the vertex pairs in the second stage is very important in order to keep the running time low.

Chapter 6

Conclusions and Future Work

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> Sir Winston Churchill,
> on the Battle of Egypt

6.1 Conclusions

In this thesis, we have developed a large number of algorithms for building additive spanners. We have also attempted to determine which of these algorithms create good spanners, where the quality of a spanner is defined primarily by the number of edges that it contains. We presented results of computational experiments which indicate that our BFS Tree / Farthest Pair Repairing algorithm performs well by producing spanners with few edges and low average delay, and it does so in low overall running time.

We have also identified a general approach, namely the 2-stage approach, that works well in most cases. We have determined that, for the first stage of the 2-stage process, spanning trees built by breadth-first search tend to produce good results, except that they tend to have high maximum degree, which thus forces the spanner

to have high maximum degree as well. We have also identified a general second stage patching technique, repairing, which tends to produce good results, and observed that the choice of patching algorithm is critical to the running time of the overall algorithm.

We would be remiss if we didn't summarize some of the limitations of our work here. Since our method of evaluating the algorithms is experimental, as opposed to analytical, we cannot make absolute claims as to their effectiveness. While the algorithms we have presented here are "good" according to our criteria, they do not provide provably optimal spanners, nor do they provably approximate them. Also, the spanners produced by our algorithms do not necessarily deal with certain networking issues like efficient routing. They differ from the spanners produced by the known constructions insofar as the constructions use methods of removing edges which are highly regular, and hence they can easily specify alternate routing schemes. The spanners produced by our algorithms may be ill-suited for the kinds of simple, structured routing one can use in a regular topology such as a hypercube.

6.2 Future Work

The most obvious future direction for this work would be to evaluate more algorithms for producing good spanners. It would be interesting to consider other graph classes to evaluate the algorithms on, especially those of random graph models which model real world networks where spanners may have applications. Extending the range of algorithms to consider stochastic approaches such as genetic algorithms and ant-colony optimization may also be informative. We feel that continuing this research into such areas as on-line and distributed algorithms for generating spanners may eventually lead to their application as structures for emerging networking problems, such as broadcast structures for ad-hoc networks.

Since the general problem of producing optimal spanners of arbitrary graphs is NP-hard, it may be interesting to determine if polynomial-time approximation algorithms exist which can produce spanners with a near-optimal number of edges. We believe that the 2-stage process may provide a framework for developing such algorithms.

We also feel that the Tainting process described in Section 4.3.2 (or a modification thereof) may provide a useful starting point for such work.

Spanners were originally motivated by problems in network design, and one of the main considerations in the constructions presented by Liestman and Shermer [13, 15] was not minimizing the number of edges, but instead bounding the maximum degree at each node. This concern stems from the nature of real-world hardware being limited by the number of connections it can support. The algorithms considered here deal with the known NP-hard optimization problem for additive spanners. We suggest that the following problem be studied in detail:

- Let $d, \Delta \geq 1$ be integers. For $\overline{d} > 0$, does there exist a $(d + x)$-spanner S of a graph G such that the average delay of S w.r.t. G is at most $\overline{d}$ and such that the maximum degree of S is $\leq \Delta$?

This question considers if there exists a spanner with fixed maximum degree and delay which has a given number of edges. We feel that, like the existing decision question regarding additive spanners, this question will also prove to be NP-Complete. It engenders several optimization questions that may be of interest and which may be applicable to existing network problems. We also suggest the development of algorithms to generate degree-constrained spanners with low average delay.

In Section 4.5 we presented arguments why a decomposition approach, where the input graph is split into two (or more) components by a cut and spanners are constructed separately for each component, would not work well for the problem we considered here. Recall that one of the issues that prevented us from considering this approach was that it would require spanner-generating algorithms which have non-constant delays; these non-constant delays were necessary to allow vertex pairs separated by the cut to have the same maximum delay as pairs lying solely within either of the components. We feel that the decomposition approach has merit, and we feel that non-constant delay algorithms should be studied in order to determine if the decomposition approach would be a workable approach to the overall additive spanner generation problem.

Finally, we feel that additive spanners provide a useful construction for understanding the overall structure of a graph without having to deal with the complete graph itself. Since they can give an "idea" of a graph, we feel that additive spanners may have useful applications outside of the realm of network communications. They may have applications in areas such as Web searching and data mining, where large amounts of information need to be dealt with in a human-comprehensible manner. We hope that the algorithms we present here may be useful enough to be applied to problems in those areas.

Bibliography

[1] Béla Bollobás, Don Coppersmith, and Michael Elkin. Sparse distance preservers and additive spanners. In *Proccedings of the fourteenth annual ACM-SIAM symposium on Discrete Algorithms*, pages 414–423, 2003.

[2] Paul R. Cohen. *Empirical Methods for Artificial Intelligence.* MIT Press, 1995.

[3] Thomas H. Cormen, Charles R. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[4] Camil Demetrescu and Giuseppe F. Italiano. What do we learn from experimental algorithmics? In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, number 1893 in Lecture Notes in Computer Science, pages 36–51. Springer-Verlag, 2000.

[5] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamc all pairs shortest paths. In *Proceedings of the thirty-fifth ACM symposium on Theory of Computing*, pages 159–166, June 2003.

[6] Dorit Dor, Shay Halpern, and Uri Zwick. All-pairs almost shortest paths. *SIAM Journal of Computing*, 29(5):1740–1759, 2000.

[7] Michael Elkin and David Peleg. $(1+\epsilon, \beta)$-spanner constructions of general graphs. In *Proceedings of the thirty-third annual ACM symposium on the Theory of Computing*, pages 173–182, 2001.

[8] Arthur M. Farley, Andrzej Proskurowski, Daniel Zappala, and Kurt Windisch. Spanners and message distribution in networks. *Discrete Applied Mathematics*, 137(2):159–171, March 2004.

[9] Marie-Claude Heydemann, Joseph G. Peters, and Dominique Sotteau. Spanners of hypercube-derived networks. *SIAM Journal of Discrete Mathematics*, 9(1):37–54, 1996.

[10] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[11] David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. November 2001.

[12] Dieter Kratsch, Hoàng-Oanh Le, Haiko Müller, Erich Prisner, and Dorothea Wagner. Additive tree spanners. Technical Report 52, Mathematics and Computer Science, University of Konstanz, 1998.

[13] Arthur L. Liestman and Thomas C. Shermer. Additive spanners for hypercubes. *Parallel Processing Letters*, 1(1):35–42, 1991.

[14] Arthur L. Liestman and Thomas C. Shermer. Two-dimensional grid spanners. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, pages 211–214, 1991.

[15] Arthur L. Liestman and Thomas C. Shermer. Additive graph spanners. *Networks*, 23:343–363, 1993.

[16] Arthur L. Liestman and Thomas C. Shermer. Grid spanners. *Networks*, 23:123–133, 1993.

[17] Arthur L. Liestman and Thomas C. Shermer. Degree-constrained network spanners with nonconstant delay. *SIAM Journal of Discrete Mathematics*, 8(2):291–321, 1995.

[18] Brendan D. McKay. `autoson` — a distributed batch system for UNIX workstation networks (version 1.3). Technical Report TR-CS-96-03, The Australian National University, Department of Computer Science, 1996.

[19] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[20] David Peleg and Jeffery D. Ullman. An optimal synchronizer for the hypercube. In *Proceedings of the sixth annual ACM symposium on the Principles of Distributed Computing*, pages 77–85, 1987.

[21] Rajmohan Rajaraman. Topology control and routing in ad hoc networks: A survey. *SIGACT News*, 33:60–73, 2002.

[22] Dana Richards and Arthur L. Liestman. Degree-constrained pyramid spanners. *Journal of Parallel and Distributed Computing*, 25:1–6, 1995.

[23] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, second edition, 2001.

[24] Duncan J. Watts. *Small Worlds.* Princeton University Press, 1999.

[25] Douglas B. West. *Introduction to Graph Theory.* Prentice-Hall, 1996.

[26] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM 1996*, pages 594–602, 1996.

Appendix

Experimental Data

The raw data used in producing this thesis is too extensive to print here. We have included the data on a CD-ROM which is attached to the inside back cover of this thesis.

If the CD-ROM is not there, or if you are reading this thesis in a fashion which would not include the CD-ROM, please contact the author by email for the data.

The `README` file on the CD-ROM gives a detailed description of the format of the data files.