

On Algorithmic Progress in Data Structures and Approximation Algorithms

by

Jeffery Li

SB in Mathematics and in Computer Science and Engineering
Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

© 2024 Jeffery Li. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Jeffery Li
Department of Electrical Engineering and Computer Science
May 17, 2024

Certified by: Jayson Lynch
Research Scientist, MIT CSAIL, Thesis Supervisor

Certified by: Neil Thompson
Research Scientist, MIT CSAIL, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

On Algorithmic Progress in Data Structures and Approximation Algorithms

by

Jeffery Li

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE

ABSTRACT

In the big data regime, computer systems and algorithms must process large amounts of data, making many traditional exact algorithms too costly to run. To work around this, researchers have developed approximation algorithms, which trade off some accuracy for asymptotic improvements in runtime, and data structures, which can efficiently store and answer multiple queries about a dataset. This naturally leads to the question, how have approximation algorithms and data structures improved over the years? Here, we provide some insight into this question, looking into trends in algorithmic and data structure progress, tradeoffs between speed and accuracy or between runtimes of specific data structure operations, and specific problems of interest. Our analysis is based on a dataset of around 300 approximation algorithms and around 250 data structures. For both fields, we find that research is still fairly active even to the present day, even though significant or asymptotic gains for data structures have been slowly on the decline. Improvements have also been fairly heterogeneous – some problems see a lot of work and improvements put into them, while others have not seen as much progress. In addition, of the problems that have both exact and approximation algorithms, around $\frac{1}{6}$ of the problems have seen approximation algorithms have immensely large average yearly improvement rates compared to exact algorithms, while around $\frac{1}{2}$ of the problems have seen approximation algorithms have minimal improvement over exact algorithms. For data structures, we find that only 4 out of the 28 abstract data types in our dataset have ever had a tradeoff between storage requirements and/or runtimes of specific operations, with only 2 still existing in the present, suggesting that improvements generally build off of each other without increasing space usage or time required for other operations. This research helps us understand how approximation algorithms and data structures have progressed through the years and how they are now.

Thesis supervisor: Jayson Lynch

Title: Research Scientist, MIT CSAIL

Thesis supervisor: Neil Thompson

Title: Research Scientist, MIT CSAIL

Acknowledgments

I'd like to thank my advisors, Jayson Lynch and Neil Thompson, for all of the guidance and support, particularly when I joined the Algorithms Wiki project two years ago. As a mentor, Jayson has not only provided lots of feedback for and insights into the data collection process for the Algorithms Wiki and the other recent theses that have come out of the different aspects of the project, but has also helped with some other research endeavors outside of the Algorithms Wiki project, such as those in the fall 2023 semester edition of Erik Demaine's "Algorithmic Lower Bounds" course. Neil's expertise on data analysis methodology has been invaluable for the data analysis portion of this project, and his feedback has helped improve many of the figures used in this thesis.

I'd also like to thank Cecilia Chen and Liva Olina for helping greatly with the data collection for approximation algorithms, and Andrew Lucas and Grace Jiang for helping with the creation and editing of many of the figures for approximation algorithms and data structures, respectively, in this thesis. This project would not have been possible without the hard work they put in. Thank you to many previous students and researchers for helping with data collection for older datasets, which I used for data analysis for this project, and to Hayden Rome and Damian Tontici, previous MEng students who also wrote theses on projects part of the Algorithms Wiki, for providing me with some of their code to help with data analysis.

In addition, I'd like to thank Erik Demaine, not only for providing knowledge and insights into approximation algorithms and data structures, but also for running several amazing theoretical computer science courses featuring large, collaborative open problem sessions. The two that stood out to me were "Advanced Data Structures," which was super helpful for the data structures part of this project, and "Algorithmic Lower Bounds," which introduced me to many more ways complexity theory can be fun and helped me contribute to original research, particularly in Tetris and asynchronous cellular automata puzzles, among other work currently in progress.

Lastly, on a more personal note, I'd like to thank many of the friends I met during my time at MIT, particularly through the Next 4W community and through MIT Sport Taekwondo. You all have inspired me throughout the years and helped me grow a lot personally, and I hope that I've also helped you all as a friend and/or as a role model.

Two joint-authored papers based on the results of this thesis, one on approximation algorithms and one on data structures, are in the works.

Contents

Title page	1
Abstract	3
Acknowledgments	5
List of Figures	9
1 Introduction	11
1.1 Prior Work	12
1.2 Objectives and Outline	13
2 Background	15
2.1 Algorithmic Problems	15
2.2 Approximation Algorithm Metrics	16
2.2.1 Time Complexity	16
2.2.2 Approximation Error	17
2.3 Data Structure Problems and Metrics	18
3 Methods	21
3.1 Scope	21
3.2 Data Collection	24
3.3 Data Processing	25
3.3.1 Classes for Approximation Algorithms	25
3.3.2 Classes for Data Structures	28
4 Progress in Approximation Algorithms	29
4.1 Overview	29
4.2 Parameterization in Approximation Algorithms	31
4.3 Speed-Accuracy Tradeoffs in Practice	35
5 Progress in Data Structures	43
5.1 Overview	43

5.2	Tradeoffs Between Query Times and/or Space	46
5.3	Most Interesting Problems	47
5.3.1	Approximate Membership Query	47
5.3.2	Priority Queue	51
5.3.3	Ordered Associative Array, Non-Comparison	54
6	Conclusion and Future Work	59
	References	63

List of Figures

4.1	For each decade, the total number of approximation algorithms, split based on whether there was a significant improvement versus no significant improvement. Note that the "2020s" bar only includes algorithms up to the present (2024), so there is not a substantial dip in the 2020s.	30
4.2	For each decade, the percentage of problem families that have seen a significant improvement from approximation algorithms.	30
4.3	The percentage of the 28 problem families with any approximation algorithm, based on type of parameterization. Blue region indicates problem families with approximation algorithms whose error can be made arbitrarily small; red region indicates problems with no such approximation algorithms	32
4.4	Distribution of polynomial-time approximation schemes by type: Fully Polynomial-Time (Randomized), Efficient Polynomial-Time (Randomized), Polynomial-Time (Randomized), Quasi-Polynomial-Time	33
4.5	Number of each type of PTAS, along with number of non-parameterized and other parameterized approximation algorithms, over the decades	34
4.6	An accumulation graph for the number of each type of PTAS, along with number of non-parameterized and other parameterized approximation algorithms, over the decades	34
4.7	Distribution of problem families by best time complexity class at 3 levels of error (exact, constant-term/factor error, any reasonable error)	36
4.8	Evolution of best runtime for All-Pairs Shortest Paths at 3 levels of error (exact, constant-term/factor error, any reasonable error)	38
4.9	Evolution of best runtime for Geometric Traveling Salesman Problem at 3 levels of error (exact, constant-term/factor error, any reasonable error)	38
4.10	Distributions of compound growth rates based on problem size (n) and error tolerance, considering only the problem families with both exact and approximation algorithms. Red indicates a decrease relative to lower error tolerance, green indicates an increase relative to lower error tolerance	41
5.1	The number of data structures in our dataset that fall under each decade	44
5.2	For each decade, the number of data structures with a significant improvement versus with no significant improvement	44

5.3	For each decade, the percentage of ADTs that have seen a significant improvement	45
5.4	Evolution of best time and space complexities for AMQ filters supporting <code>delete</code> . The line represents the asymptotically best complexity over time. The points off the line (indicated with a black outline) show data structures built that are not-optimal and thus (presumably) were designed for other purposes.	49
5.5	Evolution of best time and space complexities for AMQ filters not supporting <code>delete</code> . The line represents the asymptotically best complexity over time. The points off the line (indicated with a black outline) show data structures built that are not-optimal and thus (presumably) were designed for other purposes.	50
5.6	Evolution of best time and space complexities for priority queues. The line represents the asymptotically best complexity over time.	53
5.7	Evolution of best time and space complexities for non-comparison ordered associative arrays when the universe size U is small. The line represents the asymptotically best complexity over time.	56
5.8	Evolution of best time and space complexities for non-comparison ordered associative arrays when the universe size U is large. The line represents the asymptotically best complexity over time.	57

Chapter 1

Introduction

In the field of theoretical computer science, one of the main goals is to produce algorithms with low asymptotic runtimes, or time complexity. Generally speaking, the lower the asymptotic time complexity, the faster one can solve a problem. Typically, the problems being considered are of the following form: design an algorithm that, given an input, outputs an answer that *exactly* satisfies a set of conditions and/or optimizes for specific metrics. However, there are a few interesting variants that can be considered.

The first variant that we consider are *approximation problems*. There are many algorithmic problems where the best-known exact algorithm has a prohibitively large time complexity (such as quasipolynomial or exponential). Occasionally, when working with larger datasets, even quadratic-time algorithms may become too computationally intensive. One way to cope with this computational intractability, particularly for optimization problems (where the goal is to optimize for specific metrics), is to relax the optimization condition to allow for algorithms which output answers that are *approximately* optimal or correct, allowing for some error, but have an asymptotically better runtime than existing exact algorithms. Such algorithms are called *approximation algorithms*. In this case, the focus is on both how fast the algorithm runs and what guarantees we have on the accuracy of the algorithm's output.

The second variant that we consider are *data structure problems*. In this case, rather than

outputting a single answer, we are asked to maintain a data structure that processes and answers specific queries from a user, like inserting or deleting elements or finding the minimum or maximum of a set of objects. In particular, these differ from algorithmic problems because we now have sequences of *operations*, whose correct answers may depend on previous operations. In addition, the user does not necessarily see the state of the data structure; there is an "abstraction barrier" that separates the internal state of the data structure from the user of the data structure.

1.1 Prior Work

In analyzing the state of improvements in computing after the end of Moore's law, Leiserson et al. [1] argue that progress will come in three main areas – hardware architecture, software, and algorithms. In particular, if we can solve or approximate an algorithmic problem with a faster algorithm, then we are improving our effective computing power (albeit with some slight cost in accuracy). As such, it is important to review the state of improvements in algorithms and see from where we can possibly get additional gains in efficiency.

The rate of algorithmic progress has been studied before. The first paper addressing this topic at a large scale is due to Sherry and Thompson in 2021 [2]. The authors analyzed improvements in time complexity for 113 different exact problem families, which are groups of algorithms that solve a similar problem. They split up asymptotic runtimes into several *classes*, like "linear," "quadratic," or "exponential," and analyzed algorithmic improvement rates using these time complexity classes.

Recently, Rome [3], [4] analyzed analyzed improvements in auxiliary space complexity for 118 different exact problem families, using similar complexity classes for their analysis. In addition, the authors analyzed time-space tradeoffs for these problem families, noting what percentage of problem families had these tradeoffs (which occur when there are multiple "best" algorithms using slightly more time for less space or slightly more space for less time)

at any point within the past several decades.

Much more recently, Tontici [5] performed a similar analysis for parallel algorithms, or algorithms which utilize multiple resources like processors to run multiple steps in parallel, thus achieving a faster runtime. The authors analyzed tradeoffs between the amount of speedup achievable by parallelism and the amount of work required, noting that more parallelization often incurs work overhead due to communication costs, along with the amount of speedup that is currently possible based on real-world processor data.

Of note is that these papers analyze *exact* algorithmic problems; these papers exclude approximation problems and most data structure problems from their analyses. No paper preceding or following these papers have analyzed approximation problems and data structure problems at a large scale; previous sources like [6] (for data structures) and [7] (for approximation algorithms) have only focused on specific problems or techniques.

1.2 Objectives and Outline

In this thesis, we will analyze both approximation problems and data structure problems, focusing on how the best approximation algorithms and data structures have improved over time and whether there exist tradeoffs between approximation factors and time complexity for algorithms solving specific approximation problems and tradeoffs between query times for data structures solving data structure problems.

The work done for this thesis is part of the Algorithms Wiki project¹.

The rest of the thesis is organized as follows. In Chapter 2, we give an overview of some relevant concepts and terms related to approximation algorithms and data structures. In Chapter 3, we discuss our methodology for data collection, processing, and analysis both approximation algorithms and data structures. Chapter 4 presents our results and analysis for approximation algorithms, and Chapter 5 presents our results and analysis for data

¹algorithm-wiki.csail.mit.edu

structures. We give some concluding remarks in Chapter 6.

Chapter 2

Background

The fields of approximation algorithms and data structures are quite broad, with a wide variety of concepts and terms developed for discussing the accuracy and efficiency of approximation algorithms and data structures. In this chapter, we give some background on some of these concepts that are most relevant to the work in this thesis. We give a definition of algorithmic problems, particularly in relation to approximation algorithms, in Section 2.1, discuss approximation algorithm metrics in Section 2.2, and discuss data structure problems and metrics in Section 2.3.

2.1 Algorithmic Problems

We first give an overview on the types of algorithmic problems we chose to focus on when analyzing approximation algorithms. Here, we broaden our scope compared to what is usually studied for approximation algorithms.

An *algorithmic problem* is defined as a relation between two sets of values, the *input values* and the *output values*, and an *algorithm* is a (finite) sequence of steps used to solve an algorithmic problem. In the context of approximation algorithms, the outputs to our algorithmic problems are related by a real-valued *metric* that defines the "size" of the output. For example, the possible outputs to an algorithmic problem may be the real numbers,

with the metric being the identity function, or the possible outputs may be the cycles of a particular weighted graph that visit every vertex, with the metric being the sum of the weights of the edges in the cycle. Note that this not only includes optimization problems, where the goal is to find a valid output to the algorithmic problem that optimizes for the metric and is usually the main point of study in the field of approximation algorithms, but also estimation and numerical-analysis based problems, where the goal may be to find a value that is sufficiently close to the correct answer; these arise when it may be impossible or computationally intractable to compute the exact answer (such as the real-valued roots of general polynomials or non-linear equations).

2.2 Approximation Algorithm Metrics

When analyzing approximation algorithms, the two main metrics we consider are *time complexity* and *approximation error*.

2.2.1 Time Complexity

Time complexity, or *runtime*, describes how much time, or how many steps, it takes for an algorithm to produce an output. Generally speaking, the smaller the time complexity, the more efficient the algorithm is. As is standard in analysis of algorithms, we use *asymptotic notation* for time complexities, so we generally ignore constant factors (or terms that don't change when the input size changes) and lower-order terms. For example, an algorithm that takes $2n^3 + 3n$ steps to solve a problem would have a time complexity of $O(n^3)$, as we ignore the constant factor 2 on the term $2n^3$ and the lower-order term $3n$.

For many problems, it is common to express the time complexity in terms of the input size n ; however, this is not always the case. For example, many algorithms for graph problems have their time complexities written in terms of the number of vertices $|V|$, when the size of the input is the number of vertices plus the number of edges, or $|V| + |E|$. We follow

previous papers [3]–[5] in standardizing time complexities in terms of the problem size, so in the case of algorithms for graph problems, we use $n = |V| + |E|$.

2.2.2 Approximation Error

Approximation error describes the amount by which the answer produced by the algorithm is off from the correct or optimal solution, based on the metric defined by the problem. There are several different types of approximation errors that are of interest. The most common type is *multiplicative error*, or the ratio between the value of the answer produced by an algorithm and the value of the optimal solution. For example, one can consider the maximum-weight matching problem, where one is given a graph with a weight assigned to each edge and is asked to find a *matching* (or a set of edges such that no two edges share a vertex) whose sum of edge weights is maximized. For this problem, the algorithm that consists of greedily adding the largest available edge to the current matching is guaranteed to produce a matching whose weight is at least $\frac{1}{2}$ times the weight of the maximum-weight matching, and so we would say that the multiplicative error is $\frac{1}{2}$. During analysis, we normalize multiplicative errors to be greater than 1 (i.e. by reciprocating errors that are less than 1). This is because of the differences in how errors are stated, particularly for maximization and minimization problems.

Another type of error is *additive error*, or the absolute difference between the value of the answer produced by an algorithm and the value of the optimal solution. Additive errors are fairly common in iterative algorithms, which iteratively find values that get closer and closer to the desired solution until the change in values between iterations becomes sufficiently small. That said, this type of error occasionally pops up in algorithms for combinatorial or graph-theoretic problems.

An even rarer form of error is *additive-multiplicative error*; in this case, an algorithm may produce an answer that has both an additive error component and a multiplicative error component.

In addition, approximation errors can be *parameterized* or *non-parameterized*. By "parameterized," we mean that there is an additional parameter in the algorithm, typically denoted by ε , that can control for the desired error in the solution produced by the algorithm. A special subset of algorithms with such a parameter are the **polynomial-time approximation schemes** (PTAS), which have a multiplicative approximation factor of $1 + \varepsilon$ and a runtime that is polynomial in the problem size for every fixed value of ε , such as $O(n^{1+1/\varepsilon})$. More restrictive versions include *efficient polynomial-time approximation schemes* (EPTAS), which require the running time to be of the form $O(f(\varepsilon)n^c)$ where c is independent of ε and f is any computable function (an example of such a runtime is $O(2^{1/\varepsilon}n)$), and *fully polynomial-time approximation schemes* (FPTAS), which require the running time to be polynomial in both n and $1/\varepsilon$ (like $O(n + 1/\varepsilon^2)$); less restrictive versions which occasionally pop up are *quasi-polynomial-time approximation schemes* (QPTAS), which only require the runtime to be quasi-polynomial (i.e. $n^{\text{polylog}(n)}$) in the problem size for every fixed value of ε , such as $O(n^{(\log n)^{1/\varepsilon}})$.

2.3 Data Structure Problems and Metrics

As data structures are defined differently than algorithms, we require a different definition for a data structure problem. Here, we turn to the notion of **abstract data types (ADTs)**. An ADT is an abstract structure that consists of a set of possible objects it can interact with and collection of operations on these objects. Each operation has a specification that describes any constraints on what the input and/or output should look like and what the operation is intended to do. An example of an ADT is a *stack*, which is intended to keep track of a set of objects and has three operations:

- **build**, which creates a new instance with some elements already in the set,
- **push**, which adds an object to this set, and

- `pop`, which removes the most recently added object from the set, if there are any objects in the set.

We can thus define data structure problems as follows: given the specification for an ADT and its operations, design a concrete implementation of the ADT, which we will call a *data structure*. An example of a data structure implementing a stack is a linked list, which can be thought of as a sequence of elements with each element except for the last element containing a pointer to the next element in the list.

Similar to exact algorithms, the main metrics we consider are *time complexity* and *space complexity*. For time complexity, we consider the amount of time (asymptotically) required for each operation. As ADTs always have multiple operations, this means that data structures will have multiple time complexities associated with them, opening up possibilities to analyze tradeoffs between the time complexities of different operations. Space complexity is more straightforward – we consider the amount of information the data structure needs to store after performing a sequence of operations. The space complexity is usually linear in the number of objects the data structure is keeping track of at a given point in time; however, some data structures store slightly more information asymptotically to achieve a speedup in one or more operations.

Chapter 3

Methods

In this chapter, we discuss our approach to collecting and analyzing the data used to obtain the results in subsequent chapters. We will cover the scope and process of our data collection in Sections 3.1 and 3.2 and how our data is processed and used to produce our analysis and results in 3.3.

3.1 Scope

First, we considered which problems we should analyze. For approximation algorithms, we started with the set of 140 problem families used in Sherry and Thompson [2], Rome [3], [4], and Tontici [5]. Following previous analyses, we define a **problem family** as a collection of various formulations of the same algorithmic problem. These formulations have the same general goal, but each varies slightly in terms of restrictions on input values, type or format of the output, or even the metric being considered to judge the output in some cases. We call these **problem variations**.

Different problem families have different ways the variations are related to each other – some problem families have variations that are subsets of or overlap with other variations, while other problem families have variations that are disjoint from each other, and some may have both. An example of both can be found by considering the *Travelling Salesman*

Problem (TSP), which is the following problem: Given a graph $G = (V, E)$ with each edge having an associated distance, or weight, determine the route that visits each city exactly once, returns to the original city, and optimizes for the sum of the weights of the edges in the cycle. Three of the variations that we have data on are the following:

- *Metric Travelling Salesman Problem (Metric TSP)*, in which the distances must satisfy the *triangle inequality*¹, and we wish to minimize the sum of the weights,
- *Geometric Travelling Salesman Problem (Geometric TSP)*, in which the cities must lie in Euclidean space, with distances governed by the Euclidean metric, and we wish to minimize the sum of the weights,
- *Metric Maximum Travelling Salesman Problem (Metric Maximum TSP)*, in which the distances must satisfy the triangle inequality, and we wish to *maximize* the sum of the weights.

Here, notice that Geometric TSP is a special case of Metric TSP where the metric being used to determine distances is the Euclidean metric. On the other hand, Metric TSP and Metric Maximum TSP are disjoint variations; the former seeks to *minimize* the sum of the edge weights, whereas the latter seeks to *maximize* the sum of the edge weights. We restrict our attention to variations which were analyzed in previous papers, like Metric or Geometric TSP. Different algorithms are most useful for different variations or may only solve specific variations; we made sure to account for this in our analysis.

We use the same set of **118 problem families** used in Rome [3], [4] for our analysis, and many of our figures regarding the effects of approximation algorithms for problem families consider all 118 problem families. However, it is important to note that many problem families currently do not see any benefits from approximation algorithms. Out of the 118 problem families, 60 did not have any approximation algorithms that we were able to

¹if $d(v, w)$ denotes the shortest distance between v and w , then for any three cities v , w , and x , $d(v, w) \leq d(v, x) + d(x, w)$

collect, either because it is difficult or impossible to define a "metric" to judge outputs by (particularly for decision, or yes/no, problems like the Graph Isomorphism Problem) or because there hasn't been sufficient interest for designing approximation algorithms for these problems (likely because exact algorithms are already efficient enough). Once we filtered out for relevant approximation algorithms, particularly algorithms that did not have any issues regarding having theoretical runtime or approximation factor analysis or solving a variant that is directly comparable to a variant solved by the exact algorithms in the Algorithms Wiki database, 30 more problem families did not have any relevant approximation algorithms for us to analyze. As such, we end up with **28** problem families that have any relevant approximation algorithms.

For data structures, we didn't have a previous set of problems we could go off of, so we followed a methodology similar to Sherry and Thompson [2]. We looked through the 57 textbooks used in Sherry and Thompson's analysis and took notes on any data structures or ADTs that we saw in the textbooks. From our notes, we determined a set of **28 ADTs** that we use for our data structures analysis, along with a few more primitive ADTs (like graphs and sparse matrices) that we collected some data on but did not end up analyzing.

Model of Computation

The majority of algorithms and data structures we have collected operate under either the word RAM or the real RAM models of computation. The word RAM model is a random-access machine (RAM) that operates on words, or groups, of $O(\log n)$ bits [8]. Here, the space complexity is usually measured in words rather than bits. The real RAM model operates on real numbers exactly instead of using floating point numbers [9]. In the real RAM model, each memory cell contains a real number of any size, with exact precision [9].

Sometimes, the analysis of certain algorithms or data structures uses other models of computation, such as PRAMs (for parallel algorithms). We note the model of computation for each algorithm in our data tables.

3.2 Data Collection

Most of the data collection took place between June 2023 and March 2024. Although approximation algorithms were not the main focus in previous works, we were able to extract about 80 approximation algorithms from the database of algorithms collected for Rome [3], [4]. For the rest, we collected approximation algorithms by looking through papers, mostly through Google Scholar and Wikipedia, and using references in the papers we found. We collected data structures through the textbooks used in Sherry and Thompson’s analysis [2], tracing citations through Wikipedia, and through references in the papers we found. We noted down some algorithms and data structures that didn’t fall under our current scope, instead of discarding them, so that they be used for future projects. In total, we considered 1020 approximation algorithms and 316 data structures, but after filtering out for relevancy, we have 332 approximation algorithms and 246 data structures making up our analyzable dataset.

For approximation algorithms, the most relevant information we extracted for each algorithm is the following:

- The problem family and variation the algorithm is for
- The author and year of publication of the paper
- The worst-case time complexity
- Any parameter definitions (such as number of vertices or edges in a graph or number of elements in the input)
- Approximation error term, error type (additive, multiplicative, additive-multiplicative), whether or not the error is parameterized, whether or not the algorithm is a PTAS, and a brief description of the approximation error

- Other metadata, such as whether or not the algorithm is randomized, heuristic-based, parallel, quantum, or GPU-based; for this analysis, we exclude parallel, quantum, and GPU-based algorithms.

For data structures, the most relevant information we extracted for each algorithm is the following:

- The ADT the data structure is for
- The name of the data structure
- The author and year of publication of the paper introducing or discussing the data structure
- The worst-case space complexity, along with the unit of space being considered (i.e. a bit or a word) and the model of computation (typically Word RAM or Real RAM)
- For each operation, the time complexity of the operation as noted in literature, along with any notes on the time complexity (such as whether the time complexity is amortized, influenced by randomization, has any notable constant factors, etc.)

3.3 Data Processing

Similar to Sherry and Thompson [2], Rome [3], [4], and Tontici [5], we created a classification scheme for time and space complexities and approximation errors to allow for a standardized way of analyzing improvements in time and space complexity or approximation. Like in Rome [3], [4], we will also consider intermediate values between these classifications - for example, a data structure with a query time of $O(\sqrt{\log n})$ would be considered 2.5 in our classification.

3.3.1 Classes for Approximation Algorithms

For time complexities, we use the following classes:

1. Constant – $O(1)$
2. Polylogarithmic – $O(\log^c n)$ for some constant $c > 0$
3. Linear – $O(n)$
4. Quasilinear – $O(n \log^c n)$ for some constant $c > 0$
5. Quadratic – $O(n^2)$
6. Cubic – $O(n^3)$
7. Supercubic but still polynomial – $O(n^c)$ for some constant $c > 3$
8. Superpolynomial – $\omega(n^c)$ for any constant $c > 0$

For approximation factors, as there are both additive errors and multiplicative errors, and some algorithms have both types of errors, we use two different classifications for additive errors and multiplicative errors. For additive errors, we use the following classes:

1. Sub-constant – $o(1)$ (for example, $1/n^c$ for any $c > 0$)
2. Constant – $O(1)$
3. Logarithmic – $O(\log n)$
4. Polylogarithmic with exponent greater than 1 – $O(\log^c n)$ for some constant $c > 1$
5. Square root – $O(\sqrt{n})$
6. Linear – $O(n)$
7. Quadratic or greater – $\Omega(n^2)$

For multiplicative errors, we use the following classes:

1. Error factor decreasing toward 1 – $1 + o(1)$ (for example, $1 + (1/n^c)$ for any $c > 0$)

2. Controllable arbitrarily small error – $1 + \varepsilon$
3. Multiplicative error equal to 2
4. Any constant ($O(1)$) greater than 2
5. $O(\log \log n)$
6. Logarithmic – $O(\log n)$
7. Polylogarithmic with exponent greater than 1 – $O(\log^c n)$ for some constant $c > 1$
8. Square root – $O(\sqrt{n})$
9. Linear – $O(n)$
10. Greater than linear – $\omega(n)$

Besides $1 + \varepsilon$ multiplicative error, we treat any error with an additional error parameter term ε (which allows for the algorithm user to control the error) as slightly smaller or larger than the version without the error parameter term, depending on the effect of ε on the error, and subtract or add 0.01 from the class. In other words, additive errors of $O(\varepsilon)$, $O(\varepsilon n)$, and $O(\varepsilon n^2)$ would be considered 1.99, 5.99, 6.99, respectively, and a multiplicative error of $2 + \varepsilon$ would be considered 3.01. This is because having an error parameter allows for the error to be smaller than any value in the class above and/or approach the value in the class below even though it won't be put in the class below. For example, we can make ε smaller than any constant, and we can make $2 + \varepsilon$ approach 2 by letting ε approach 0, but $2 + \varepsilon$ cannot be equal to 2 because the error parameter must usually be positive. For $1 + \varepsilon$ multiplicative error, because all PTASes have a $1 + \varepsilon$ multiplicative error, the algorithms with this error are special enough that we separate them out into their own class.

3.3.2 Classes for Data Structures

For data structures, there is a wider variety of time and space complexities, ranging from than logarithmic or between polylogarithmic and linear to exponential or greater. Thus, we use the following classes for both time and space complexities:

1. Constant – $O(1)$
2. $O(\log \log n)$
3. Logarithmic – $O(\log n)$
4. Polylogarithmic with exponent greater than 1 – $O(\log^c n)$ for some constant $c > 1$
5. Square root – $O(\sqrt{n})$
6. Linear – $O(n)$
7. $O(n \log n)$
8. $O(n \log^c n)$ for some constant $c > 1$
9. Quadratic – $O(n^2)$
10. Superquadratic but still polynomial – $O(n^c)$ for some $c > 2$
11. Exponential or greater – $\Omega(c^n)$ for some constant $c > 0$

Chapter 4

Progress in Approximation Algorithms

4.1 Overview

When talking about advances in approximation algorithms, it first makes sense to consider when researchers have been devising new approximation algorithms, both in general and with significant (asymptotic) improvements. We thus analyze both how many approximation algorithms have been created per decade and how many significant improvements have been made per decade, and compare some of these figures to those for exact algorithms.

Figure 4.1 shows the distribution of approximation algorithms in our dataset that fall within each decade. Notably, many of the approximation algorithms were introduced in the 1990s and beyond, with growth even into the 2000s and 2010s. This suggests that the field of approximation algorithms is still very active, with many approximation algorithms being designed even in the present day. Of note is the moderate number of approximation algorithms in and before the 1940s – many of these algorithms are iterative algorithms used for computing roots of an equation or eigenvalues of a matrix.

The distribution of improvements in approximation algorithms tells a similar story – see Figure 4.1, which show the number of approximation algorithms with a significant improvement versus with no significant improvement, and Figure 4.2, what proportion of problem

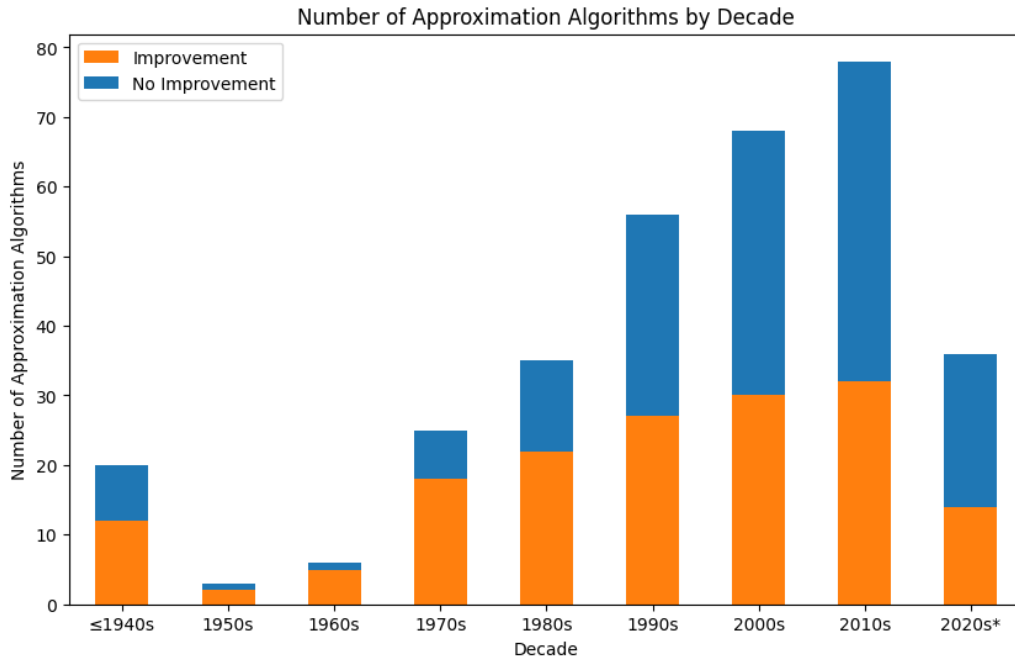


Figure 4.1: For each decade, the total number of approximation algorithms, split based on whether there was a significant improvement versus no significant improvement. Note that the "2020s" bar only includes algorithms up to the present (2024), so there is not a substantial dip in the 2020s.

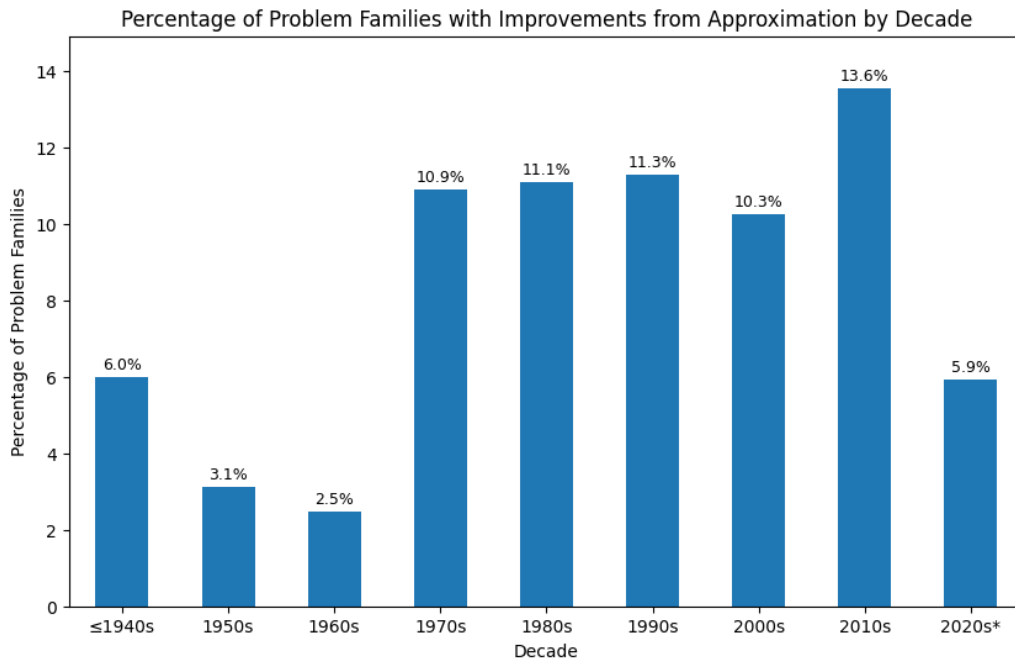


Figure 4.2: For each decade, the percentage of problem families that have seen a significant improvement from approximation algorithms.

families, out of the problem families considered in previous analyses [2]–[5], had significant improvements in each decade. Here, we define a *significant improvement* as having an algorithm that has a better runtime, additive error, or multiplicative error class, based on the classes defined in Section 3.3.1, at the end of the decade compared to the beginning of the decade. Here, we see that the amount of improvements has been consistently around 10-14% since the 1970s, which is relatively high considering that many problem families have no approximation algorithms for them. In addition, there is still significant activity in finding improvements even into and beyond the 2010s, both in terms of output (producing approximation algorithms) and variety (across problem families).

4.2 Parameterization in Approximation Algorithms

One of the main ways the tradeoff between speed and accuracy shows up in approximation algorithms is through the *parameterization of the approximation error* ε . By having at least one error parameter ε (if not more) that can freely be adjusted by the algorithm user, we are able to get arbitrarily close to the desired solution at the cost of higher runtime, or speed up the computations in our algorithm at the cost of a slightly worse approximation guarantee. Some natural questions to ask here include: How many problem families have an algorithm that has an approximation error parameter, and how many have an algorithm that allows us to get arbitrarily close to the desired solution through either an additive ε error or a multiplicative $(1 \pm \varepsilon)$ error? Of the types of polynomial-time approximation schemes (PTASes), which type appears the most often in our dataset? We answer these questions and more in this section.

We first discuss the questions related to approximation algorithms with *any* type of parameterized approximation. Figure 4.3 shows the proportion of the 28 problem families with any approximation algorithm that have an algorithm with an approximation error parameter, and the proportion that have an algorithm that allows us to get arbitrarily close

Percentage of Problem Families by Type of Parameterization

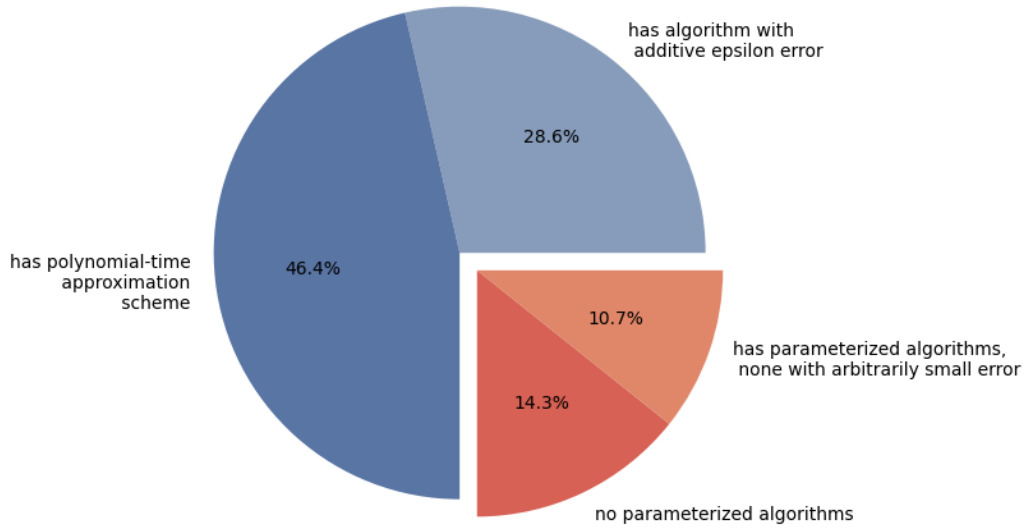


Figure 4.3: The percentage of the 28 problem families with any approximation algorithm, based on type of parameterization. Blue region indicates problem families with approximation algorithms whose error can be made arbitrarily small; red region indicates problems with no such approximation algorithms

to the desired solution. Interestingly, a majority (75%) of these problem families have approximation algorithms with arbitrarily small parameterized error. Many of these families seem fall under two flavors: either they are numerical analysis-based problems with iterative algorithms where the additive error goes to 0 as more and more iterations are performed, or they are NP-complete or other graph-based problems for which PTASes are most useful. The remaining families are split somewhat evenly between having a parameterized approximation algorithm and not having a parameterized approximation algorithm, indicating that parameterization can still be useful outside of making the error arbitrarily small, but it's not always considered when designing approximation algorithms.

Now, we look specifically at the PTASes. Here, we use the strict definition of a PTAS from in Section 2.2.2; in other words, we do not consider algorithms with additive errors as being PTASes. Figure 4.4 shows the proportion of PTASes by type of PTAS. Here, we categorize each PTAS by the most specific category it falls under – for example, a fully polynomial-

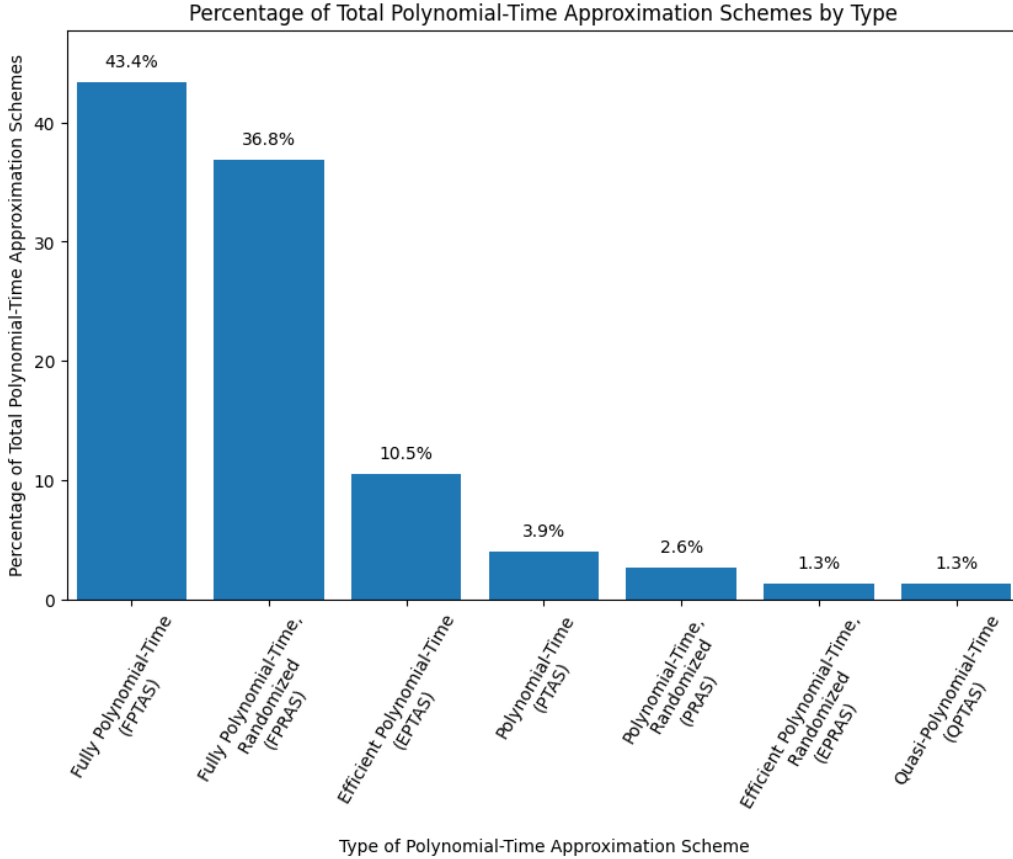


Figure 4.4: Distribution of polynomial-time approximation schemes by type: Fully Polynomial-Time (Randomized), Efficient Polynomial-Time (Randomized), Polynomial-Time (Randomized), Quasi-Polynomial-Time

time approximation scheme (FPTAS) will always be categorized as an FPTAS, an efficient polynomial-time approximation scheme (EPTAS) that is not an FPTAS will be categorized as an EPTAS, and so on. Of note is that there are significantly more FPTASes compared to any other type of PTAS. This makes sense based on the emphasis placed on polynomial runtimes for approximation algorithms, and the fact that no other types of PTASes besides FPTASes have guarantees on the runtime being polynomial in the reciprocal of the error parameter $\frac{1}{\epsilon}$, as even EPTASes can have runtimes that are exponential or worse in $\frac{1}{\epsilon}$. This means that once an FPTAS is designed for a specific problem, any new parameterized algorithms designed for that problem should generally be FPTASes in order for there to be a meaningful improvement in runtime. This is also supported in Figures 4.5 and 4.6 – while the other

types of PTAS are prevalent throughout the decades, with many PTASes and EPTASes appearing in the 1990s, FPTASes remain the most popular type of PTAS across all decades.

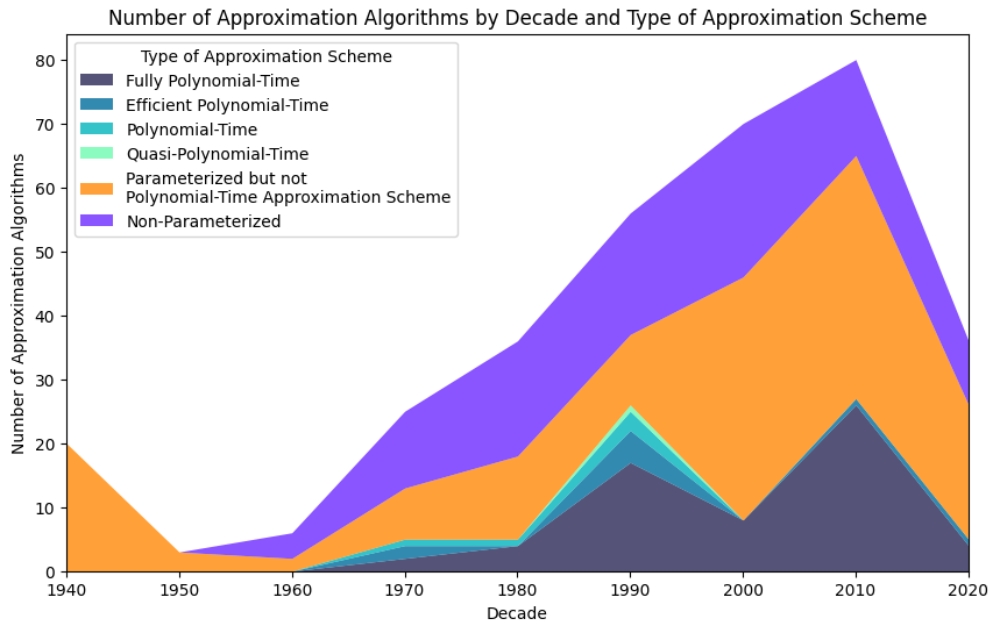


Figure 4.5: Number of each type of PTAS, along with number of non-parameterized and other parameterized approximation algorithms, over the decades

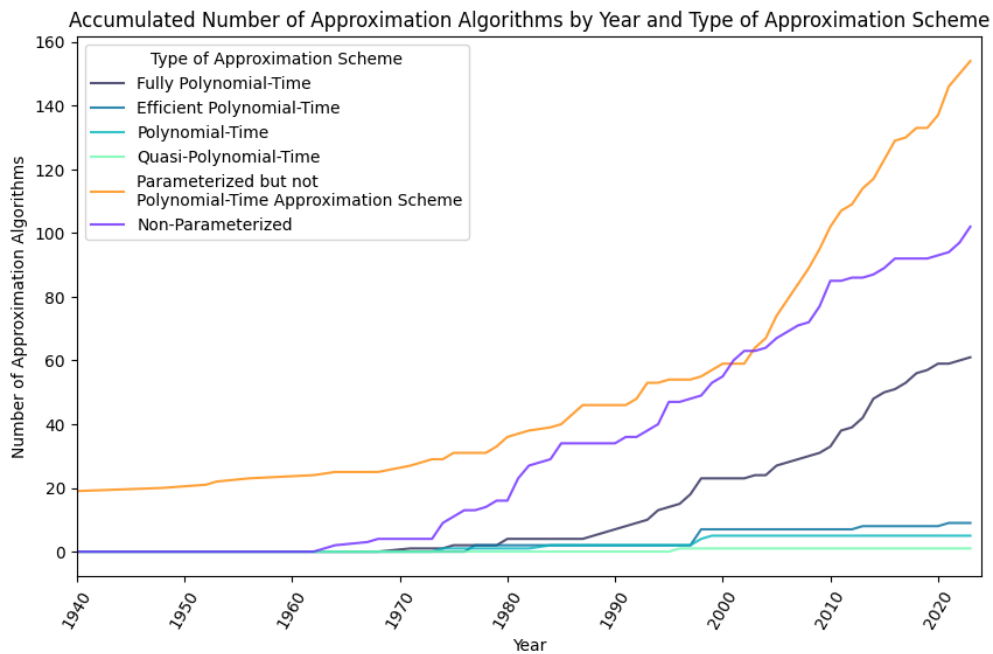


Figure 4.6: An accumulation graph for the number of each type of PTAS, along with number of non-parameterized and other parameterized approximation algorithms, over the decades

4.3 Speed-Accuracy Tradeoffs in Practice

Approximation algorithms are designed under the premise that, by sacrificing some accuracy, one can obtain results using significantly lower computation time. Thus, rather than asking how many problem families have a set of algorithms that show a tradeoff between speed and accuracy, we ask, how much faster can we solve specific problems if we decide to trade off varying amounts of accuracy? Here, we analyze the gains in speed at different levels of approximation errors, both from a theoretical and from a practical point of view.

First, we examine the algorithms with the best theoretical runtime, given different levels of approximation errors, across all problem families. Figure 4.7 shows the distribution of problem families based on the algorithm with the best theoretical runtime at 3 levels of error:

- No error – only exact algorithms are considered,
- Constant-term or better error – the error term does not increase when any parameter in the problem increases, besides possibly any specific approximation error parameters,
- Any reasonable error – all algorithms recorded in our dataset. By "reasonable," we try to avoid any trivial or nonsensical algorithms, such as those ignore the problem and output an arbitrary answer.

We note that quite a few of the problems in our dataset are NP-complete problems, meaning that no exact polynomial-time algorithms exist for these problems unless $P=NP$. This explains why the distribution for "no error" has a spike in the "superpolynomial" class that decreases by around 25-30% when moving to the other two distributions.

However, what's striking is the significant increase in the "linear" class – the percentage of problem families with a linear-time algorithm increases by around 18% (from around 37.9% to around 44.8%) when we allow for even a constant factor/term error. This highlights that approximation is not just used for NP-complete problems – it is also used for problems where the best exact algorithm runtime is a high-degree polynomial, a cubic, or even a quadratic

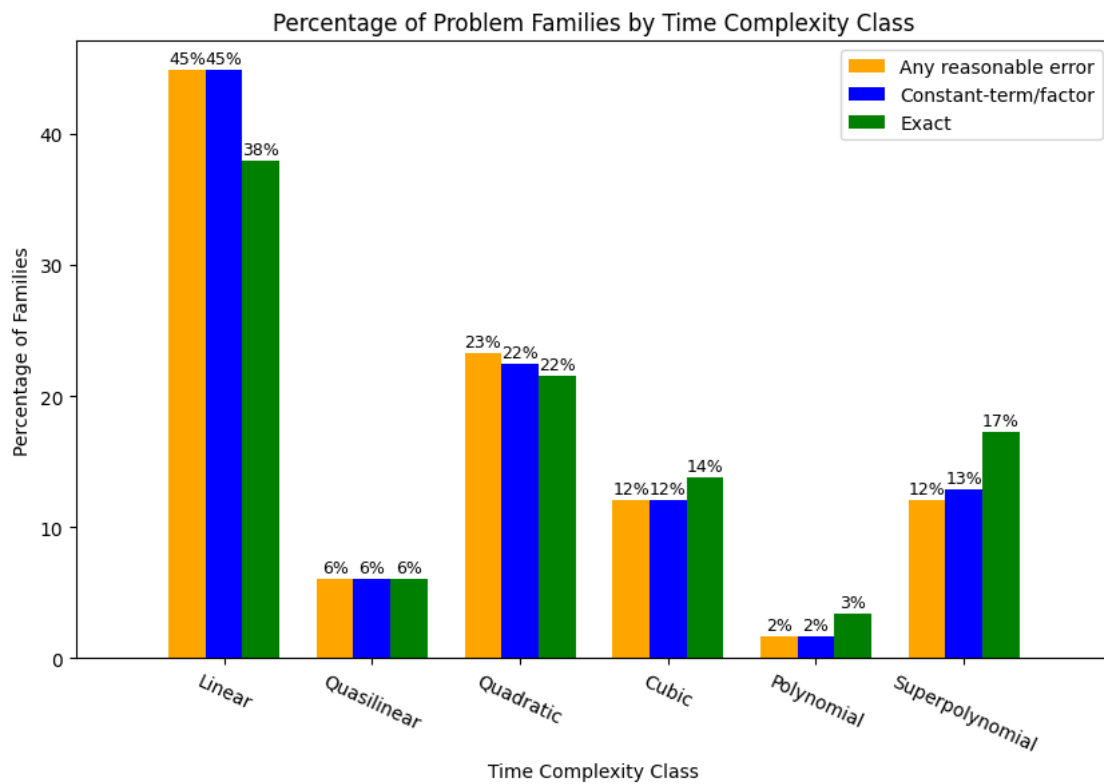


Figure 4.7: Distribution of problem families by best time complexity class at 3 levels of error (exact, constant-term/factor error, any reasonable error)

in the problem size. It also shows the practicality of approximation algorithms for many problems, as the runtime for these approximation algorithms is low even with a slight cost in error (i.e. a constant term or factor).

Another interesting aspect is that there is little change from the "constant-term/factor error" distribution to the "any reasonable error" distribution; only one problem family (the *Set Cover Problem*) changes from superpolynomial to quadratic, and no other problem family changes from one class to another. This change only happens because the Set Cover Problem cannot be approximated better than $(1 - o(1)) \ln n$ unless $P = NP$ [10], but has many approximation algorithms with an $O(\log n)$ approximation factor. However, the fact that there are no other changes highlights the efficiency of constant-factor/term approximation algorithms for many of the problem families being considered, and that researchers generally don't aim to design faster approximation algorithms with more than constant error, likely due to the error factor or term being too large to be useful.

We take a look at a few specific problems. One notable problem is *All-Pairs Shortest Paths (APSP)*, a problem that has cubic-time exact algorithms but has seen quite a few approximation algorithms as well. Figure 4.8 shows how the best runtime, at the 3 levels of error mentioned earlier, has evolved over the years. Here, we specifically study the unweighted and undirected variant, which is the variant that all of our algorithms can solve. Of note is that the one truly subcubic-time algorithm in our figure is based on fast matrix multiplication, which is known to be impractical (see page 2 of [11]), while many of the approximation algorithms recorded in our dataset do not use fast matrix multiplication, including many with constant-term or better errors. This shows how approximation algorithms are important even in the polynomial-time regime, in that they help us bypass impractical methods (like fast matrix multiplication) and shave off polynomial factors in runtime with only a slight cost in accuracy.

Another notable problem is the *Traveling Salesman Problem (TSP)*, an NP-complete problem. Figure 4.9 shows how the best runtime, at the 3 levels of error mentioned earlier,

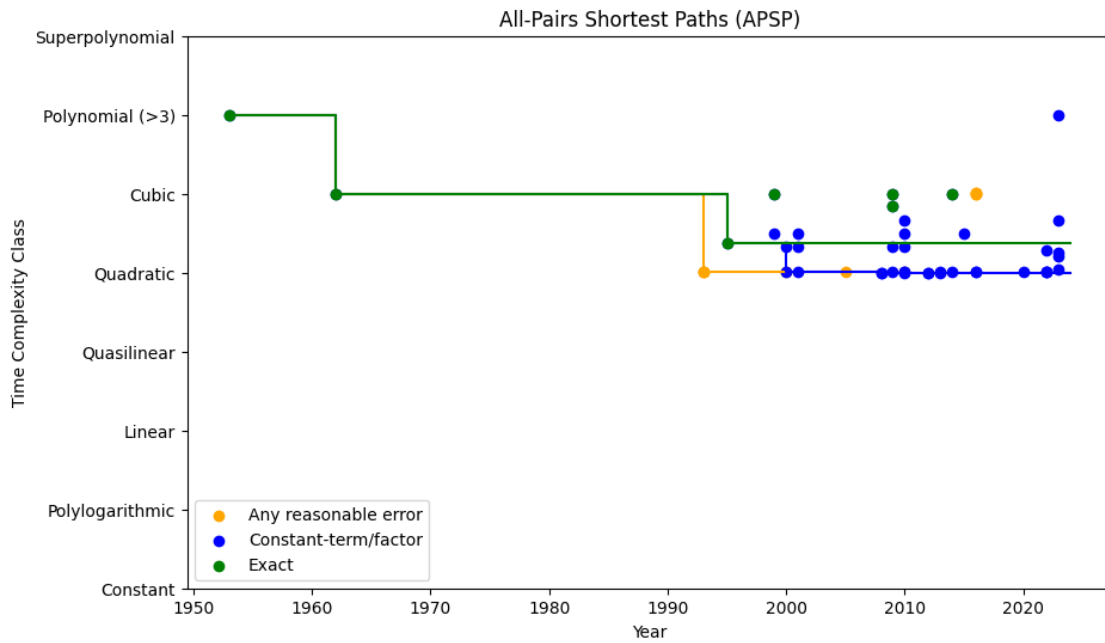


Figure 4.8: Evolution of best runtime for All-Pairs Shortest Paths at 3 levels of error (exact, constant-term/factor error, any reasonable error)

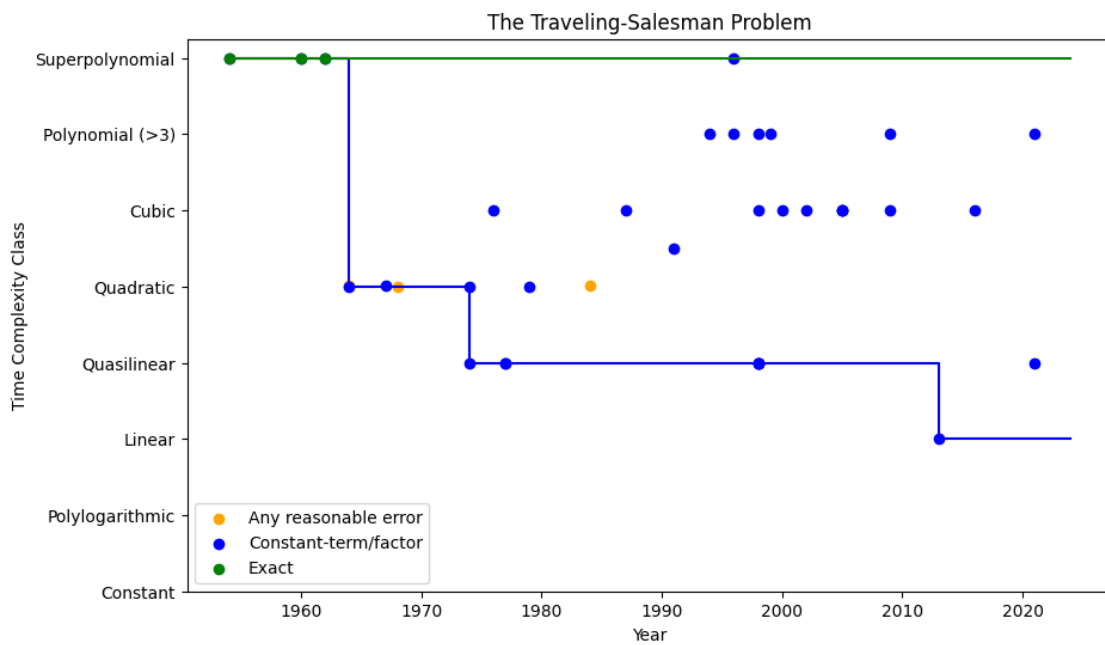


Figure 4.9: Evolution of best runtime for Geometric Traveling Salesman Problem at 3 levels of error (exact, constant-term/factor error, any reasonable error)

has evolved over the years. Here, we specifically study the geometric variant; in other words, the graph is embedded in Euclidean space, and edge weights are based on Euclidean distance between the two vertices. As an NP-complete problem, it's clear that the best exact algorithms should remain superpolynomial; however, it's interesting that much work has been done to reduce the runtime all the way down to linear in the input size, although many of the approximation algorithms that are linear or quasilinear in the input size are EPTASes with an exponential or greater dependency on the error parameter.

How do these theoretical runtime decreases translate to speedups in practice? Here, we look at how the speedup factors look like with numbers plugged in, and amortized over the years since the first algorithm for each problem. To do so, we use a slightly modified version of the **compound growth rate** defined by Tontici [5]; here, we compute the compound growth rate using the formula

$$\text{CGR} = (r_e/r_{a,\varepsilon})^{1/t} - 1,$$

where r_e is the numerical value associated with the best exact algorithm runtime, $r_{a,\varepsilon}$ is the numerical value associated with the runtime of the best approximation algorithm with error term at most ε , and t is the number of years since the first algorithm for the problem.

We present Figure 4.10, which shows the distribution of compound growth rates across only the problem families that have both exact and approximation algorithms (not across all 118 problem families) for 12 different combinations of problem size n (10^3 , 10^6 , and 10^9) and error tolerance ε (0.1, 2, 10, and any). Here, if the error tolerance depends on n , we plugged in the corresponding value of n to determine the error tolerance; in most cases, this just means that the approximation algorithms with logarithmic error are grouped with the approximation algorithms with constant error between 2 and 10, though some are grouped with the approximation algorithms with larger error. We also ignore constant factors in runtimes, even those generated by the error tolerance ε (as we believe that the values of ε

being considered are large enough that the effect is fairly small); as such, these figures are only rough estimates, but Sherry and Thompson [2] have shown that the effect of constant factors on these types of calculations are usually small, if not negligible.

Within the problem families that have both exact and approximation algorithms, we see some trends similar to those for time and space complexity improvements for exact algorithms observed by Sherry and Thompson [2] and by Rome [3], [4]. In particular, improvements are fairly heterogeneous across problem families – around half of the problems don’t see any noticeable speedups regardless of error size, even if the problem size increases to 10^9 , while quite a few problems observe moderately large or immeasurable speedup, particularly for larger problem sizes and larger error tolerance. Problem size and error tolerance somewhat affect the distribution of compound growth rates, but the effect is not large – the main changes occur when the error tolerance is increased from 2 to 10 and when the problem size increases from 10^3 to 10^6 , and from 10^6 to 10^9 . Otherwise, the distributions are fairly similar across all different combinations of problem size and error tolerance that we consider for this analysis.

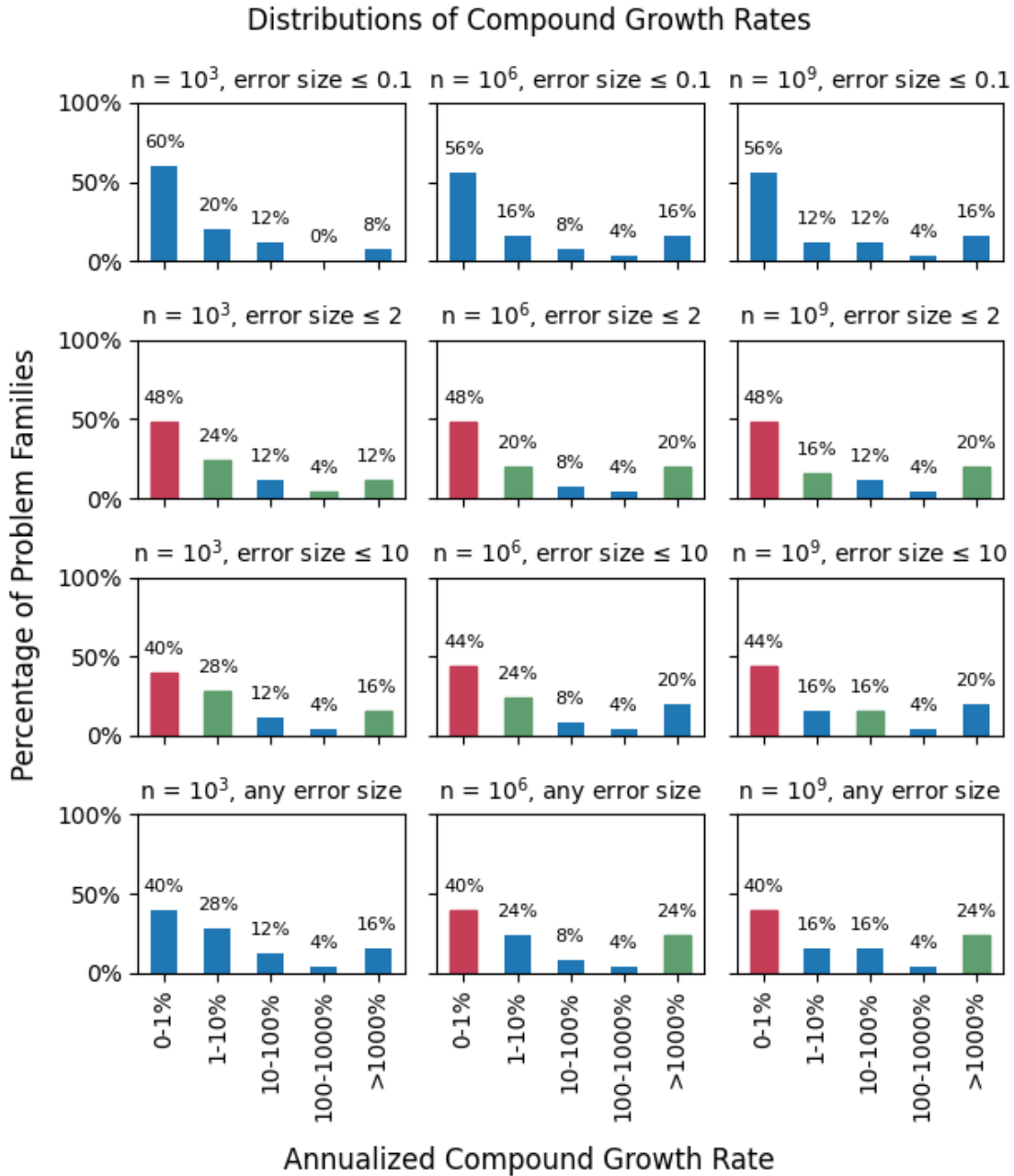


Figure 4.10: Distributions of compound growth rates based on problem size (n) and error tolerance, considering only the problem families with both exact and approximation algorithms. Red indicates a decrease relative to lower error tolerance, green indicates an increase relative to lower error tolerance

Chapter 5

Progress in Data Structures

5.1 Overview

Compared to research in common textbook algorithmic problems, research in common textbook data structure problems has a slightly different story. Overall progress in improving time and space complexities for data structures are still fairly heterogeneous, but are typically much smaller asymptotically compared to that of algorithms. Here, we give an overview of trends in both volume of and improvements in data structures, both as a whole and over the decades.

Figure 5.1 shows the distribution of data structures in our dataset that fall within each decade. Somewhat surprisingly, this distribution is fairly similar to the one for approximation algorithms – many of the data structures were designed fairly recently, particularly in the 1990s to 2010s, suggesting that research in data structures remains active in this present day.

Slightly different from before is the amount of asymptotic improvements for any query or space complexity for these data structures – Figures 5.2 and 5.3 show the number of data structures with a significant improvement versus with no significant improvement, and what proportion of problems have had such improvements in each decade (Here, the first discovered

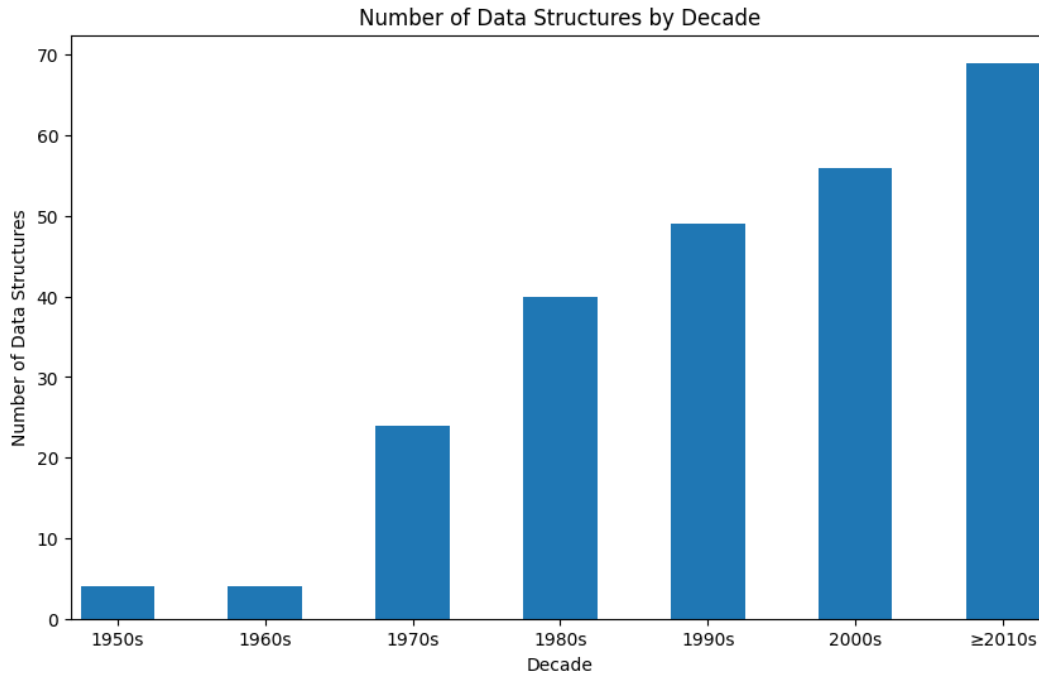


Figure 5.1: The number of data structures in our dataset that fall under each decade

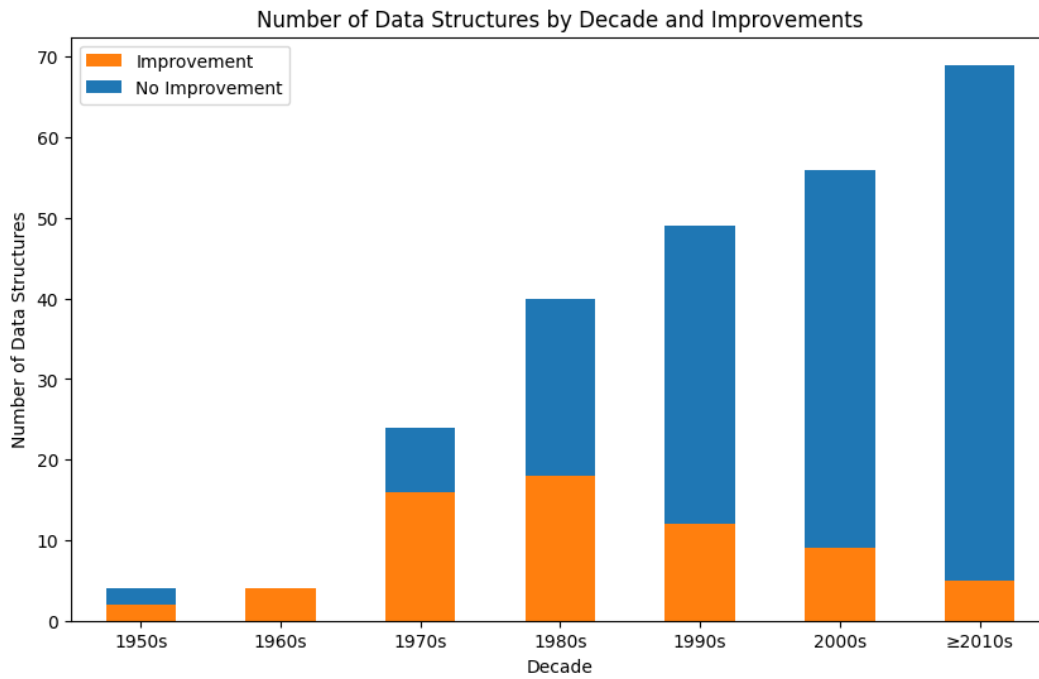


Figure 5.2: For each decade, the number of data structures with a significant improvement versus with no significant improvement

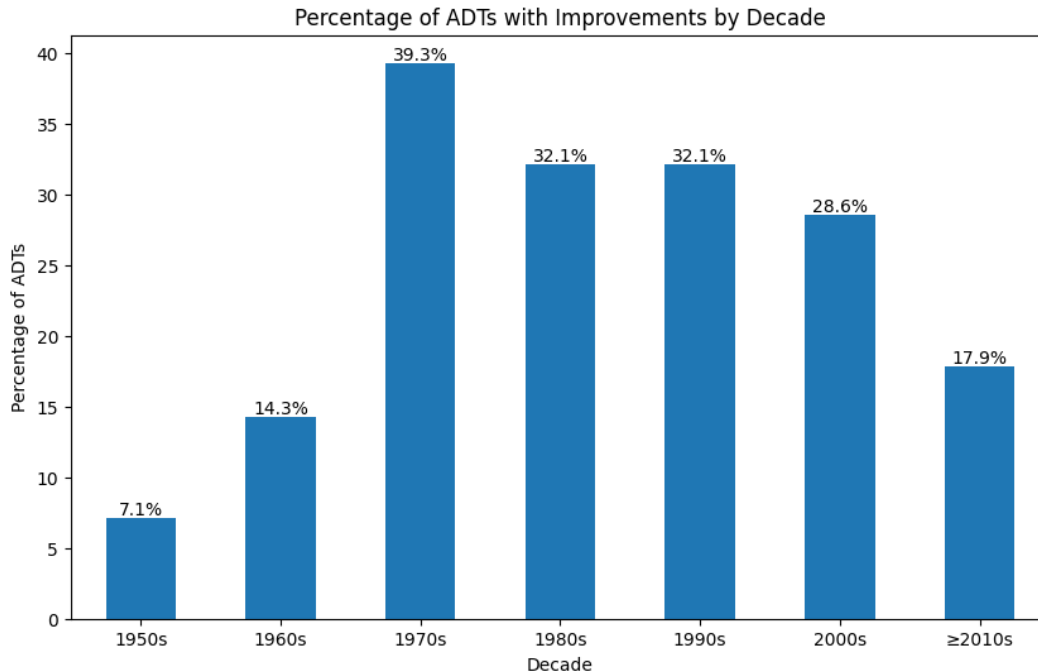


Figure 5.3: For each decade, the percentage of ADTs that have seen a significant improvement (data structure of each ADT counts as an improvement). Here, many of the improvements were achieved around the 1970s and 1990s, and while there still have been some improvements in recent years, many more data structures being created don't have significant improvements over previous data structures, and fewer ADTs have seen improvements in recent years. As it turns out, **50% (14)** of the ADTs in our dataset have seen an improvement beyond the first data structure, and around **28.5% (8)** of the ADTs have seen at least two improvements beyond the first data structure.

Together, these suggest that while there is still very active research in designing data structures, achieving asymptotic improvements is not the only priority. This is likely due to the first data structure designed for many problems already being asymptotically optimal or otherwise being difficult to improve on, and so researchers optimize for other properties, such as constant factors, simplicity or practicality of implementation in specific real-world systems, or compatibility with other models like the external memory or cache-oblivious model.

5.2 Tradeoffs Between Query Times and/or Space

Even more striking is the small proportion of ADTs that have had any nontrivial tradeoffs between time complexities of any of its operations and/or space complexities. As it turns out, only 4 ADTs in our dataset have seen any such tradeoffs at any point over the years, and only 2 such tradeoffs still exist today:

- **Priority Queue** – This ADT keeps track of a set of elements, each with an associated priority. When queried, the priority queue must be able to add and update elements, or remove and/or output the element with the highest priority. An example of a data structure for this ADT is the *binary heap*. This tradeoff is discussed in Section 5.3.2.
- **Non-Comparison Ordered Associative Array** – This ADT is designed to keep track of a set of key/value pairs. By *ordered*, we want the structure to keep track of the keys based on an ordering imposed on the keys, and by *non-comparison*, we mean that the keys are nonnegative integers from the set $[0, U]$, which we call the *universe*¹. An example of a data structure for this ADT is the *van Emde Boas tree*. This tradeoff is discussed in Section 5.3.3.
- **Monotone Priority Queue** – This ADT is similar to the priority queue, but has the guarantee that the priorities of the outputted elements must form a monotone sequence. In other words, if the priority queue is designed to remove the element with the minimum priority, then the priorities on the removed elements should be monotonically increasing; similarly, if the priority queue is designed to remove the element with the maximum priority, then the priorities on the removed elements should be monotonically decreasing. An example of a data structure for this ADT is the *bucket queue*. For this ADT, the introduction of *multi-level bucket queues* in 1979,

¹This roughly follows the convention in distinguishing between comparison sorting, which uses comparisons between values in order to sort, and non-comparison sorting, which uses properties of the values, like the values being integers, to sort

which improves on the `delete` runtime at the cost of the `insert` runtime, caused a tradeoff with single-level bucket queues until the introduction of *two-level radix heaps* in 1990, and later, the introduction of *heap-on-top (HOT) queues* in 1999 augmented with the heaps designed by Thorup in 2000, along with Thorup's heaps themselves, re-opened a tradeoff with previous data structures, between runtimes for `insert` and `delete_max`, that still seems to exist to this day.

- **External Memory Range Search** – This ADT keeps track of a set of elements, each associated with a point in a space like \mathbb{R}^d , and must handle queries of the form "given a region of space called the *range*, how many elements lie in the range?" Our dataset only has two data structures for this ADT: the *O-tree*, which performs better on `insert`, `delete`, and `find` operations, and the *Cross Tree*, which performs better on `range search` operations.

5.3 Most Interesting Problems

Here, we wish to highlight specific ADTs which have seen lots of study and/or progress over the years and show some specific improvement graphs for these ADTs.

5.3.1 Approximate Membership Query

One notable ADT is the *Approximate Membership Query (AMQ) filter*. The goal of an AMQ filter is to keep track of a set of elements approximately and space-efficiently, trading off some accuracy of information about the set for a smaller amount of required storage for the data structure. The main operations for an AMQ filter include the following:

- `build` – Creates a new instance with n elements already in the set.
- `insert` – Inserts an element into the set.

- **find** – Determines whether an element is *definitely not* in the set or *probably* in the set. In other words, **find** is allowed to have false positives but not false negatives; if an element is in the set, then **find** must always return "yes", but if an element is not in the set, then **find** should return "no" with probability $1 - \varepsilon$, where ε is some constant determined before the creation of the instance of the AMQ filter.

Some AMQ filters also support **delete**, which deletes an element from the set.

Figures 5.4 and 5.5 show how the best space complexity and the best runtimes for each of the operations has evolved over the years, for both the version that supports **delete** and the version that does not support **delete**. Of note is the space complexity graph – the true space complexity for most AMQ filters is slightly smaller than linear; assuming ε is a constant, AMQ filters typically require $\Omega(n)$ *bits* to store n elements, which translates to roughly $\Omega(n/\log n)$ words in the Word RAM model. As shown in the figures, the earliest AMQ filter, which is the Bloom filter, achieves a space complexity of $O(n)$ bits along with the best known runtime for all operations ($O(n)$ for build, $O(1)$ for all other operations). Despite this, many other data structures, including many variants of the Bloom filter, have still been developed over the years, even some with slightly higher space complexity or higher runtimes for specific operations, with many being designed or optimized for specific purposes like networking or key-value storage or having additional features like scalability or ability to be parallelized.



Figure 5.4: Evolution of best time and space complexities for AMQ filters supporting delete. The line represents the asymptotically best complexity over time. The points off the line (indicated with a black outline) show data structures built that are not-optimal and thus (presumably) were designed for other purposes.

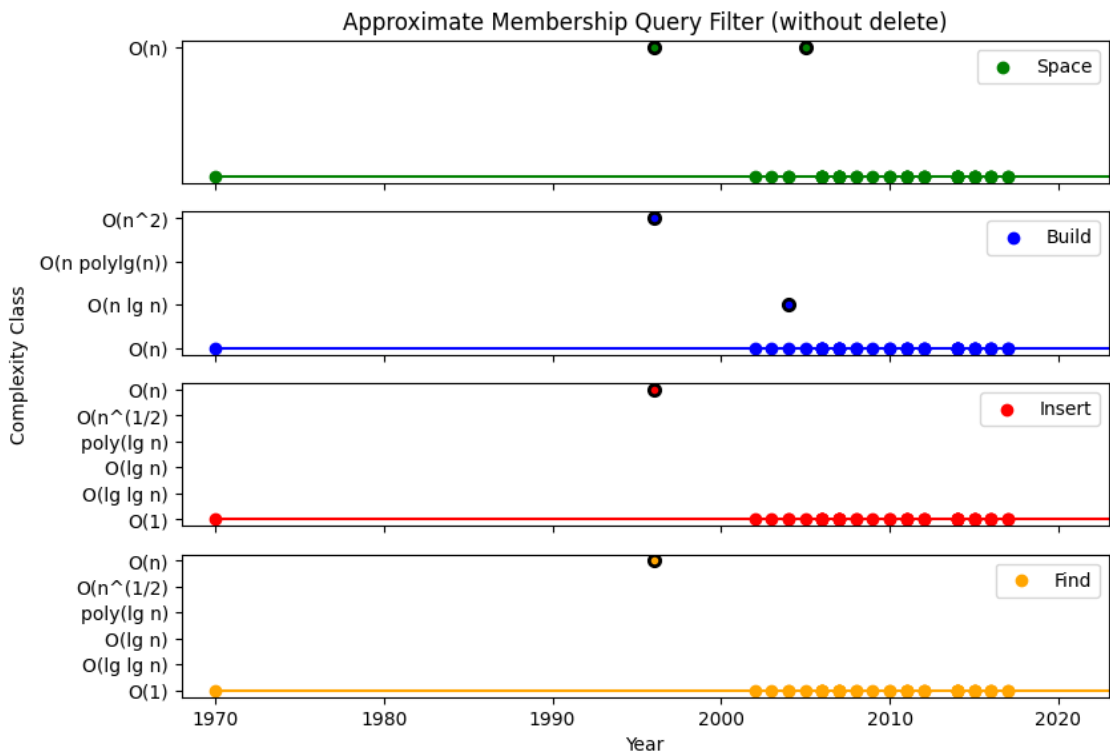


Figure 5.5: Evolution of best time and space complexities for AMQ filters not supporting `delete`. The line represents the asymptotically best complexity over time. The points off the line (indicated with a black outline) show data structures built that are not-optimal and thus (presumably) were designed for other purposes.

5.3.2 Priority Queue

Another notable ADT is the *Priority Queue*. Here, a priority queue is a structure that keeps track of a set of elements, each with an associated priority. When queried, the priority queue must remove and output the element with the highest priority. The main operations for a priority queue include the following:

- **build** – Creates a new instance with n elements (each with an associated priority) already in the set.
- **insert** – Inserts an element, with a specified priority, into the set.
- **delete** – Delete an arbitrary element from the set, if it exists.
- **delete_max** – Delete the element with the maximum priority from the set.
- **union** – Combines two priority queues into a single priority queue.
- **find_max** – Returns the element with the maximum priority (without removing it from the set).
- **update** (or **increase_key**) – Updates (typically increases) the priority of a specified element.

There exist variants of priority queues such as double-ended priority queues; however, for this subsection, we focus on single-ended priority queues.

Figure 5.6 shows how the best space complexity and the best runtimes for each of the operations (other than **delete_max**) has evolved over the years. Compared to AMQ filters, priority queues have seen quite a few more asymptotic improvements over the years. Interestingly, all of these improvements did not happen simultaneously, highlighting the significant incremental progress in driving down the runtime for many operations from $O(\log n)$ to $O(1)$ in the 1970s and 1980s. Of note is the **soft heap**, which is the data structure with an $O(1)$

runtime for the `delete` operation – the soft heap achieves constant (amortized) runtime for many operations by "corrupting" some of the priorities of the elements in the heap, which makes soft heaps slightly more unpredictable compared to other data structures. Otherwise, no other priority queue beats $O(\log n)$ runtime for `delete`.

As mentioned earlier, the priority queue ADT is one of the few ADTs that has seen a tradeoff, albeit very temporarily; the introduction of *pairing heaps*, which slightly improved on runtimes for `union` and `update` at the cost of a suboptimal `insert` runtime, and *bottom-up skew heaps*, which significantly improved the `union` runtime but did not beat the best `update` runtime, in 1986 created a tradeoff that lasted until the introduction of *Fibonacci heaps* in 1987. Besides these two years, there has remained one asymptotically optimal data structure throughout the years.

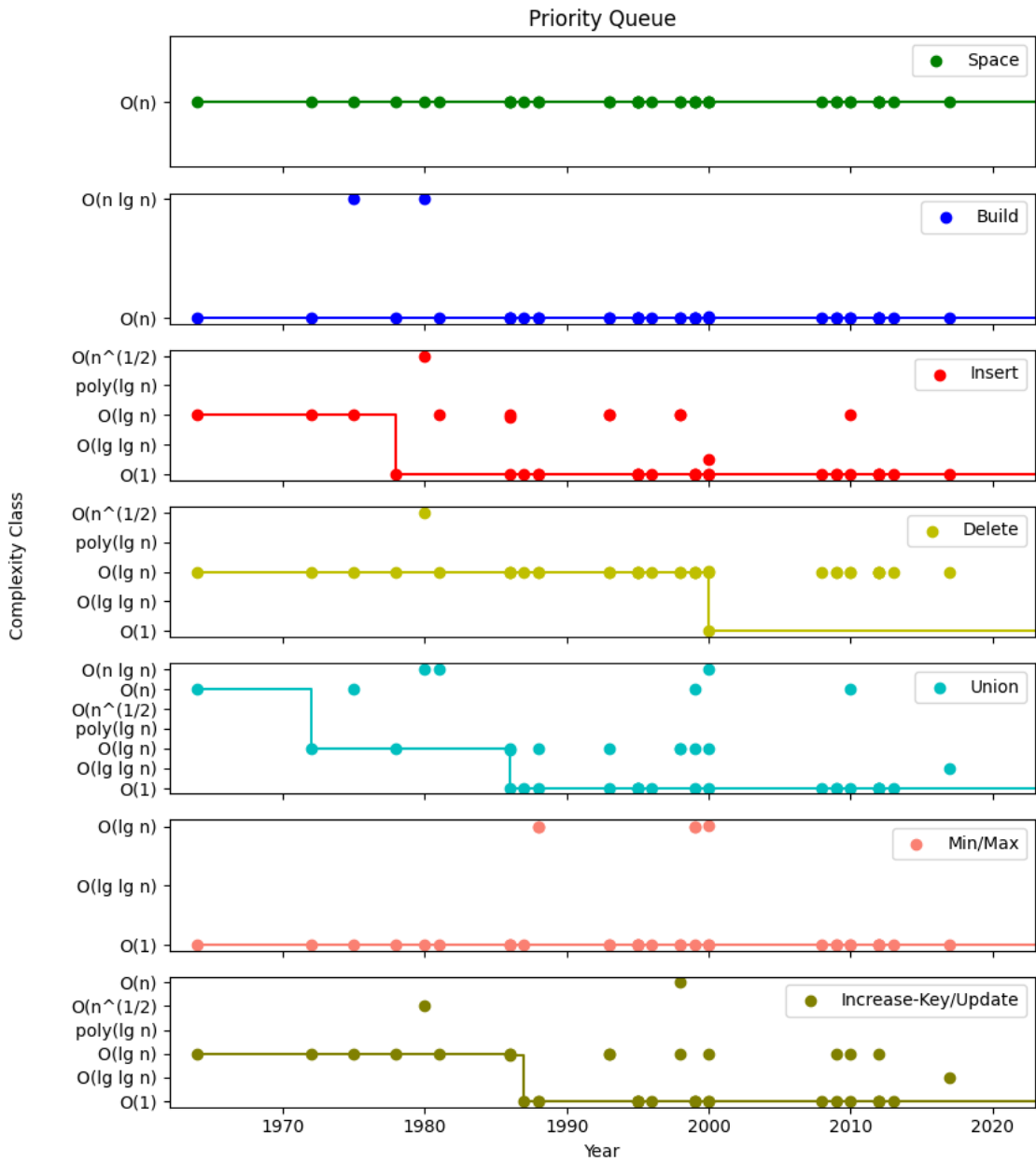


Figure 5.6: Evolution of best time and space complexities for priority queues. The line represents the asymptotically best complexity over time.

5.3.3 Ordered Associative Array, Non-Comparison

Finally, we consider the *Non-Comparison Ordered Associative Array* ADT. Recall that this ADT is designed to keep track of a set of key/value pairs where the keys are nonnegative integers from the set $[0, U]$, which we call the *universe*. The main operations for a non-comparison ordered associative array include the following:

- **build** – Creates a new instance with n key/value pairs already in the set.
- **insert** – Inserts a key/value pair into the set.
- **delete** – Delete an arbitrary key/value pair from the set, if it exists.
- **find** – Find the value associated with a given key.
- **successor/predecessor** – For a given possible key k , find the successor or predecessor key/value pair. The successor is the pair with the smallest key greater than k , and the predecessor is the pair with the largest key smaller than k .
- **union** – Combines two non-comparison ordered associative arrays into one.
- **find_min/find_max** – Returns the key/value pair with the lowest or highest key, respectively.

Figures 5.7 and 5.8 show how the best space complexity and the best runtimes for each of the operations has evolved over the years. Here, we consider two different reasonable scenarios for the universe size U – one where U is polynomial in the number of elements n , and one where U is exponential in n . One oddity is that the first data structure, van Emde Boas trees, for this ADT shows how to achieve an $O(1)$ runtime for **find_min/find_max** via storing the minimum and maximum values as attributes and updating as necessary, while few later data structures, if any, seem to support performing **find_min/find_max** operations, at least based on the descriptions found in the original papers introducing them. As it seems

straightforward to augment these data structures to support performing `find_min/find_max` operations in $O(1)$ time in a way similar to van Emde Boas trees, but the runtime is not explicitly mentioned in the literature, we have excluded the `find_min/find_max` graph from these figures.

Interestingly, the universe size U changes how the improvements on the runtimes of operations look – if the universe size is small (i.e. polynomial in n), then the current best data structure (the **y-fast tree**) was found fairly early on, with the data structures found afterward performing worse for most operations, while if the universe size is large (i.e. exponential in n), then there have been many incremental improvements over the years, with the latest designed data structure (the **dynamic fusion tree**) being the current best.

As mentioned earlier, and similar to priority queues this ADT is another one of the few ADTs that has seen a tradeoff, albeit very temporarily; the introduction of *p-fast tries* and *q-fast tries*, which slightly improved on space complexity and the runtime for `build` at the cost of suboptimal runtimes for all other operations, in 1981 created a tradeoff that lasted until the introduction of *y-fast tries* in 1982. Besides these two years, there has remained one asymptotically optimal data structure throughout the years.

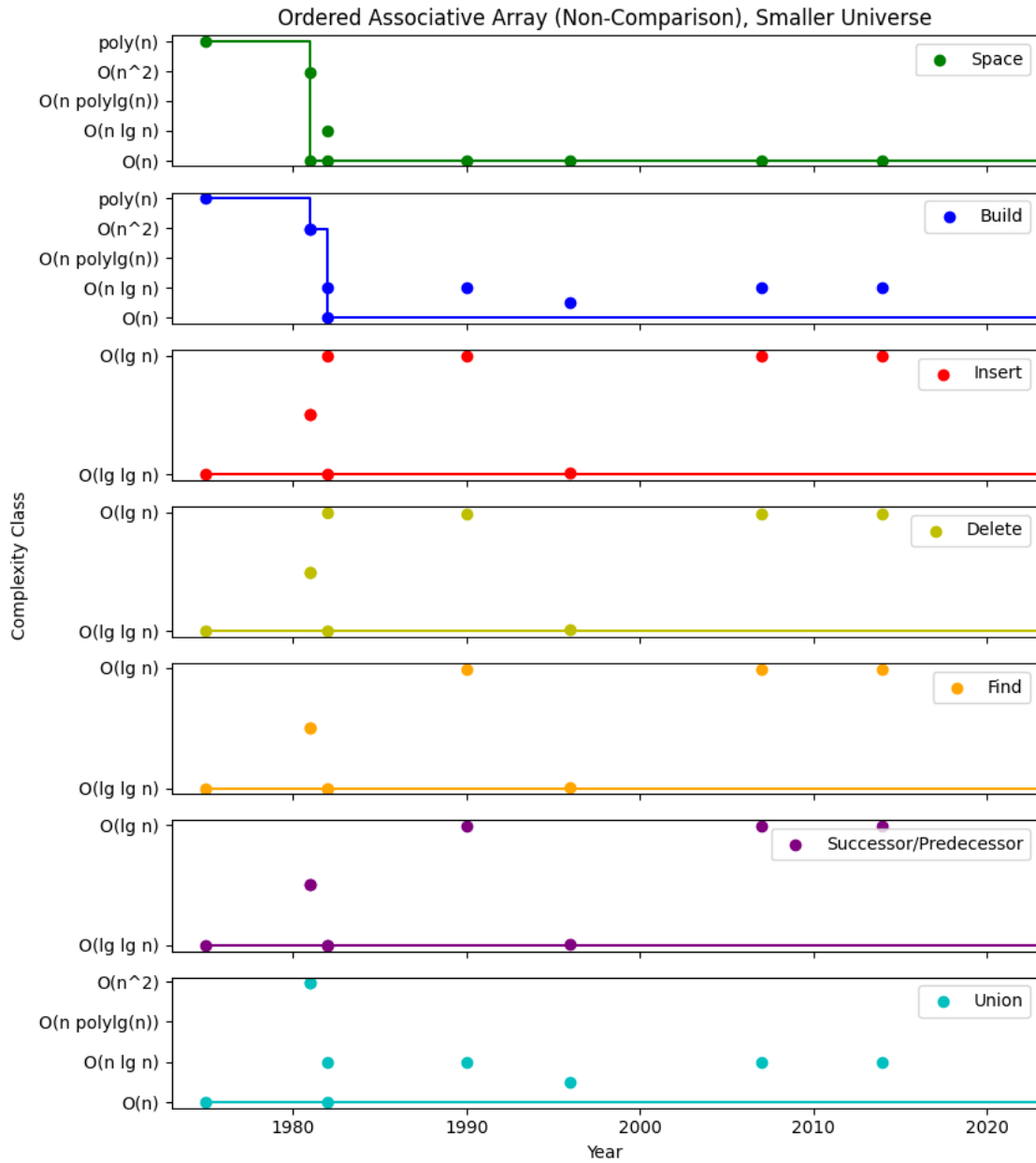


Figure 5.7: Evolution of best time and space complexities for non-comparison ordered associative arrays when the universe size U is small. The line represents the asymptotically best complexity over time.

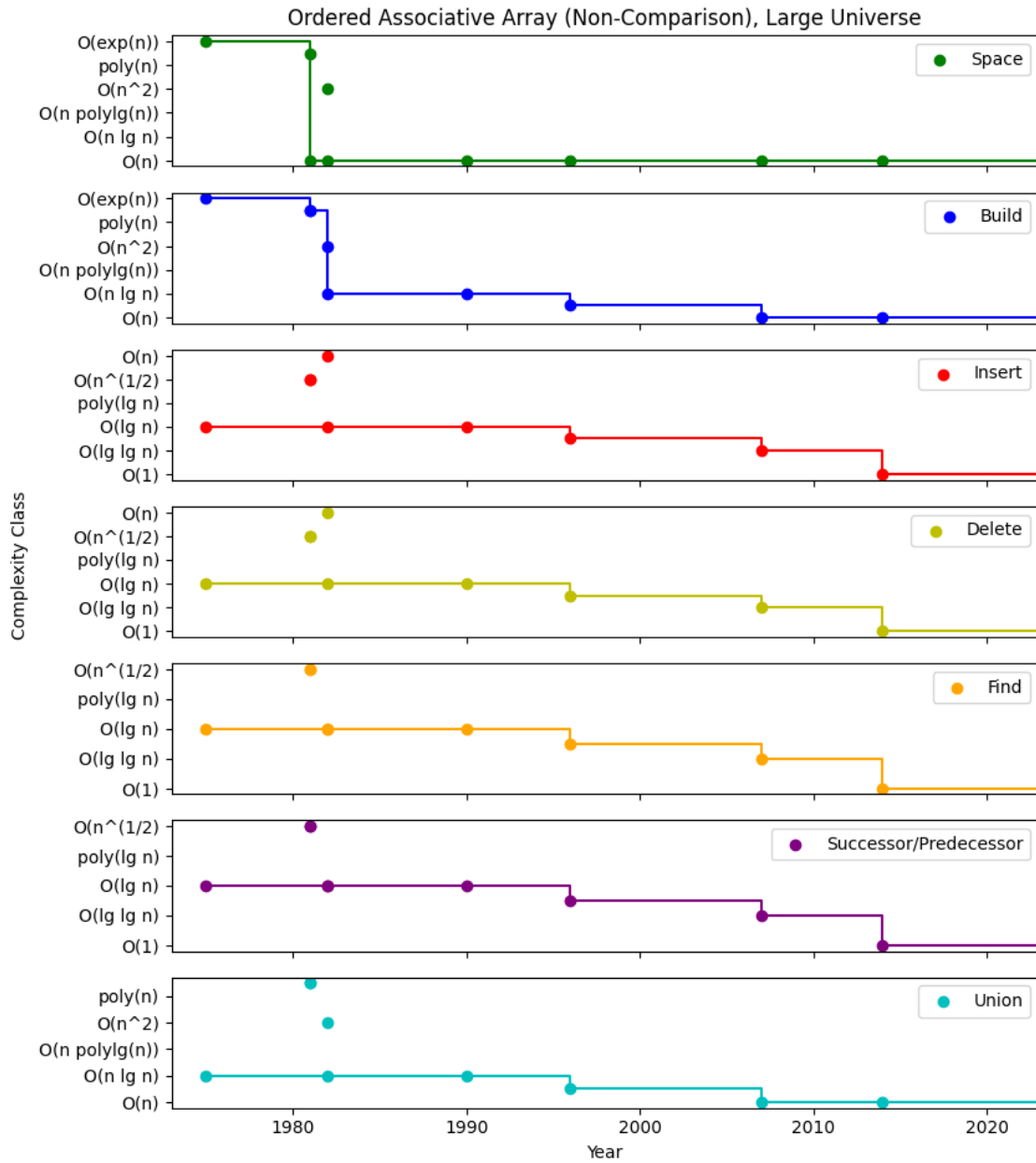


Figure 5.8: Evolution of best time and space complexities for non-comparison ordered associative arrays when the universe size U is large. The line represents the asymptotically best complexity over time.

Chapter 6

Conclusion and Future Work

In this thesis, we analyzed overall trends in research in approximation algorithms and data structures, particularly related to problems that are commonly studied in textbooks or in computer science classes. This continues off of previous work done in meta-analyzing algorithmic research [2]–[5] and considers some new directions, both in circumventing theoretical lower bounds for exact algorithms via approximation, and in branching out to structures that must repeatedly store and retrieve information based on the user’s needs.

Like parallel algorithms [5], approximation algorithms have seen much work and many improvements after the decline in improvements in sequential algorithms, particularly in the 1990s and beyond. In terms of the tradeoff between runtime and accuracy, one important aspect of many algorithms is an approximation parameter ε , which allows for the user to control the approximation factor at the cost of a higher runtime. Many problem families have at least one algorithm with such an approximation parameter, and of these problem families, quite a few have at least one algorithm that can output an answer that is arbitrarily close to the correct solution, like a PTAS or an algorithm with ε additive error, by allowing ε to be arbitrarily small. More practically, we note some interesting results – for example, by allowing for constant-factor or constant-term error, most problem families whose best exact algorithms run in exponential time now have polynomial-time algorithms, which is

to be expected, but we also see a spike in proportion of problem families that have linear-time or better algorithms, which indicates that approximation algorithms are also somewhat common even for problems that can already be solved in polynomial time. Of course, the set of problem families being considered for this and previous analyses does not contain many NP-complete problems, so this analysis may not be fully representative of the landscape of approximation algorithms research, but it still provides some interesting insight into approximation algorithms research in the most common textbook algorithmic problems.

Data structures research seems to tell a different story. Like algorithmic problems, data structure problems are fairly heterogeneous in terms of improvement rates, but few data structure problems have seen large asymptotic improvements. Somewhat surprisingly, even fewer data structure problems have any sort of tradeoff between space complexity and/or time complexities of specific operations, suggesting that most improvements, if they do occur, build off of previous best data structures without increasing space usage or time required for other operations. This is not to say that research in data structures is slow – we highlighted some data structure problems which have seen a lot of activity and improvements over the years. However, it’s likely that other aspects of data structures have greater importance in research, such as simplicity or practicality of implementation in real-world systems.

There are a few directions this research can be taken in in the future. First, for both approximation algorithms and data structures, it would be interesting to perform similar analysis but with an expanded set of problems. Particularly for our approximation algorithms analysis, NP-complete problems are not well-represented among the problems that we analyzed, since we stuck to the set of problems from Sherry and Thompson’s analysis [2] to be able to perform most of our analyses, so we could perform a similar analysis with more NP-complete problems or other problems commonly studied in the field of approximation algorithms. Since most, if not all, NP-complete problems see immeasurable gains from approximation algorithms, we would likely have to shift our perspective on how to analyze the algorithms we collect, such as looking instead at the largest problem size we can solve within

a given number of steps, or looking at how small the error parameter can be while still being asymptotically faster than the best exact algorithm, but the analysis would be more general and more applicable to the field as a whole. For data structures, while we believe we covered many of the data structures and data structure problems taught in textbooks and courses, including some taught in graduate-level courses, it's possible that we may have missed some that are important in real-world systems or in current research.

Also, for approximation algorithms, one could also look at gaps between known upper and lower bounds on best time complexity, similar to Liu's lower bounds analysis for exact algorithms in [12], [13]. In particular, some problems have known inapproximability results, such as the one for the Set Cover Problem discussed in Section 4.3, so we can highlight where there is still open area for research among the problems that we are considering.

Lastly, for data structures, we could consider some possible approaches to analyze the improvement rates of data structures solving the same problem. Because most data structures are required to handle many operations, it seems difficult to assign a single numerical value representing the speedup of one data structure over another because the runtime is dependent on the sequence of operations the data structure must handle, and the sequence of operations can vary greatly; for example, it may be balanced between all types of operations, or it contains much more of a particular type of operation, and it can force the data structure to handle fewer (i.e. $O(1)$ or $O(\log n)$) items during most of the sequence or many items (i.e. $O(n)$) during most of the sequence. It would be interesting to see how an analysis of improvement rates would look, particularly for sequences of operations that appear the most often.

Both approximation algorithms and data structures have seen impressive improvements and activity throughout the years, and it'll be interesting to see how these fields continue to advance.

References

- [1] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, “There’s plenty of room at the top: What will drive computer performance after moore’s law?” *Science*, vol. 368, no. 6495, eaam9744, 2020. DOI: [10.1126/science.aam9744](https://doi.org/10.1126/science.aam9744). eprint: <https://www.science.org/doi/pdf/10.1126/science.aam9744>. URL: <https://www.science.org/doi/abs/10.1126/science.aam9744>.
- [2] Y. Sherry and N. C. Thompson, “How fast do algorithms improve?[point of view],” *Proceedings of the IEEE*, vol. 109, no. 11, pp. 1768–1777, 2021.
- [3] H. Rome, “The space race: Progress in algorithm space complexity,” *Master’s thesis, Massachusetts Institute of Technology*, 2023.
- [4] H. Rome, J. Lynch, J. Li, C. Falor, and N. C. Thompson, “How fast are algorithms reducing the demands on memory? A survey of progress in space complexity,” *Working Paper*, 2024.
- [5] D. Tontici, “Progress in parallel algorithms,” *Master’s thesis, Massachusetts Institute of Technology*, 2024.
- [6] J. L. Bentley and J. H. Friedman, *A survey of algorithms and data structures for range searching*. Computer Science Department, Carnegie-Mellon Univ., 1978.
- [7] A. Srinivasan, “Approximation algorithms via randomized rounding: A survey,” *Series in Advanced Topics in Mathematics, Polish Scientific Publishers PWN*, pp. 9–71, 1999.

- [8] M. L. Fredman and D. E. Willard, “Blasting through the information theoretic barrier with fusion trees,” in *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, 1990, pp. 1–7.
- [9] M. I. Shamos, *Computational geometry*. Yale University, 1978.
- [10] I. Dinur and D. Steurer, “Analytical approach to parallel repetition,” *CoRR*, vol. abs/1305.1979, 2013. arXiv: [1305.1979](https://arxiv.org/abs/1305.1979). URL: <http://arxiv.org/abs/1305.1979>.
- [11] F. L. Gall, “Faster algorithms for rectangular matrix multiplication,” *CoRR*, vol. abs/1204.1111, 2012. arXiv: [1204.1111](https://arxiv.org/abs/1204.1111). URL: <http://arxiv.org/abs/1204.1111>.
- [12] E. Liu, “A metastudy of algorithm lower bounds,” *Master’s thesis, Massachusetts Institute of Technology*, 2021.
- [13] E. Liu, Y. Sherry, W. Kuszmaul, J. Lynch, and N. C. Thompson, “How close are algorithms to being optimal?” *Working Paper*, 2023.