8-20-2017

Data-Driven Abstraction

Vivian Mankau Ho
Louisiana State University and Agricultural and Mechanical College, whe1@lsu.edu

DATA-DRIVEN ABSTRACTION

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The School of Electrical Engineering and Computer Science

by
Vivian Mankau Ho
December 2017

Acknowledgments

It is a great pleasure to thank my advisor Professor Supratik Mukhopadhyay for his time, patience, and guidance during my years of graduate study, without whose support this thesis would not have been possible. His comprehensive knowledge of computer science has made him an invaluable resource for learning.

I would like to thank Professors Gerald Baumgartner and Bijaya B. Karki for joining my thesis committee, willing to help and give their best suggestions, and sharing their expertise. I would also like to thank Brian Peterson for his great help with collecting sample data for experiments.

Table of Contents

List of Tables

List of Figures

ABSTRACT
Given a program analysis problem that consists of a program and a property of interest, we use a data-driven approach to automatically construct a sequence of abstractions that approach an ideal abstraction suitable for solving that problem. This process begins with an infinite concrete domain that maps to a finite abstract domain defined by statistical procedures resulting in a clustering mixture model. Given a set of properties expressed as formulas in a restricted and bounded variant of CTL, we can test the success of the abstraction with respect to a predefined performance level. In addition, we can perform iterative abstraction-refinement of the clustering by tuning hyperparameters that determine the accuracy of the cluster representations (abstract states) and determine the number of clusters. Our methodology yields an induced abstraction and refinement procedure for property verification.

Chapter 1
Introduction

In the first chapter we shall present some background information and motivation of abstraction-based verification.

1.1 History and Motivation

1.1.1 Program Analysis and Abstract Interpretation

The overall goal of program analysis, is that given a program P and a set Φ of properties, we hope to discern whether all such properties are true or false on the program. There are three basic steps towards this goal:

1. Set up a mathematical model for the program P;

2. Use a formal language to express the properties Φ of interest;

3. Apply an approach to check whether $P \models \Phi$ (i.e., whether P satisfies properties Φ).

Abstract interpretation is a sound tool that can be used to formally verify properties of programs. It formalizes the idea of semantics-based approximation of a program with possible loss of information providing a sound framework for program analysis.

The theoretical framework of abstract interpretation was first introduced by Patrick and Radhia Cousot [8]. Abstract interpretation provides a practical and effective method to verify properties of both finite and infinite state systems (programs).

The principle of abstract interpretation is that it involves an infinite domain of program states we refer to as a concrete domain mapping to a finite height lattice we refer to as an abstract domain. From there, the abstract lattice is explored using different techniques to identify fix points. Lastly, since the abstraction and concretization mappings satisfy a Galois connection, we are guaranteed that said fix points have a meaningful interpretation in the concrete domain. An abstraction thus consists of

- a concrete domain and an abstract domain
- a Galois connection that relates the abstract domain to the concrete domain
- a set of (sound) abstract transitions on the abstract domain (that form an abstract transition system).

Classical abstraction frameworks, such as predicate abstraction[1] [15], require the user to input predicates that enable the creation of a "good" abstraction of the program. This step necessitates that the user has a thorough understanding of the program. However, the author of a program and the person verifying it may be different; as such, the latter may not be well-versed in the intricacies of the program. In classical abstraction-based program analysis [8], the behavior of a program analysis tool is not quantitatively characterized in terms of true positive rate (TPR), false positive rate (FPR), and the number of abstract states. There are no "hyperparameters" to tune to obtain "better abstractions" eventually approaching an optimal "operating point" for a given program analysis problem.

[1] We consider predicate abstraction closest to our work in the conventional verification literature.

1.1.2 Data-Driven Abstraction

In this work, we propose a new technique, called data-driven abstraction, based on the classical abstract interpretation framework [8], to effectively and efficiently verify properties of programs with minimal manual intervention.

We state *our problem* as follows: given an instrumented program P and a set of properties Φ (we use temporal formulas to naturally express Φ), we apply data-driven abstraction approach to automatically generate an ideal abstraction that determines how to abstract concrete behaviors. The abstraction then induces a program analyzer such that the temporal formulas in Φ are verified as precisely and fast as possible, depending on the abstraction.

The verification of such temporal formulas will be executed on an abstract domain (abstract model) which is one key ingredient to form an abstraction of the concrete domain of the program. The size of the abstract domain may be heavily reduced but with properties safely preserved. That is to say, if we can answer questions about the program in the abstract domain, then we can answer questions about the program in the original, concrete domain. This one-direction implication is regarded as *weak preservation* of properties under the abstraction. We further wish the converse direction of the implication to hold as well. The implications on both directions form *strong preservation* of properties under the abstraction that is an equivalence of verification of properties on the concrete and abstract domains.

Our data-driven abstraction approach outputs strongly preserving abstract models constructed through abstract domain refinements that can be used to verify properties of programs. Our method aims to

- determine the accuracy of the abstraction in terms of an *error probability* ($\mathcal{E}$);

- balance verification accuracy and the cost of computation in terms of a quantitative performance measure ($\mathcal{S}$) (see its definition in (5.10)).

We present both a theoretical framework for data-driven abstraction and a practical tool for program analysis, using a clustering technique based on a distance metric that results in an abstract cluster domain. Our technique applies a data driven, query guided refinement process that refines the abstract model by tuning the hyperparameters that in turn results in modification of the abstract cluster domain.

Our data-driven abstraction framework is composed of the following steps. A flow chart of our data-driven abstraction approach is shown in Figure 1.1.

1. The user instruments a program P to sample states occurring at the instrumented program points during the execution of the program. The (instrumented) program itself is treated as a generative model.

2. Given a set of sampled program states, we show the existence of an abstract domain consisting of clusters of states where states having similar behavior belong to the same cluster. We show the existence of a Galois connection between the concrete domain of program states and the abstract domain of clusters.

3. We then provide a procedure for drawing from each cluster a sample of program states (call it a cluster sample) that satisfy a criterion depending on two hyperparameters β and ϵ that modulate the characteristics of the clusters.

4. For each cluster, using the sample of program states drawn above, we estimate a probability density function (PDF) representing the distribution

of states corresponding to that cluster. Each cluster can now be implicitly approximated by a PDF.

5. The clusters form the basis of an abstract transition system (ATS), and the PDFs form the basis of an abstract density transition system (ADTS), approximating the program.

6. Queries are specified in the verification logic, a restricted and bounded variant of CTL. The ATS (and/or ADTS) allows us to answer queries about the program with an error probability $\mathcal{E}$ that depends on the hyperparameters β and ϵ.

7. We define a quantitative performance measure $\mathcal{S}$ for the program analyzer, in terms of TPR, FPR, and the size of ADTS[2]. In case $\mathcal{S}$ falls below a threshold δ, we can refine the abstraction by tuning the hyperparameters β and ϵ towards achieving an optimal operating point for the given program analysis problem.

In abstraction-based analysis, the design of an analyzer for a program analysis problem essentially depends on the construction of an abstract domain (and thus an abstraction). A successful abstraction is closely related to the property of interest. Our framework is flexible since the users need only to provide program points of interest.

1.2 Contributions

Since we verify properties of a program over abstract states (clusters), we need to know how to compute the abstraction mapping α, i.e., how to determine which

[2]The size of ADTS is the same as the size of ATS, or the number of PDFs, or the number of abstract clusters.

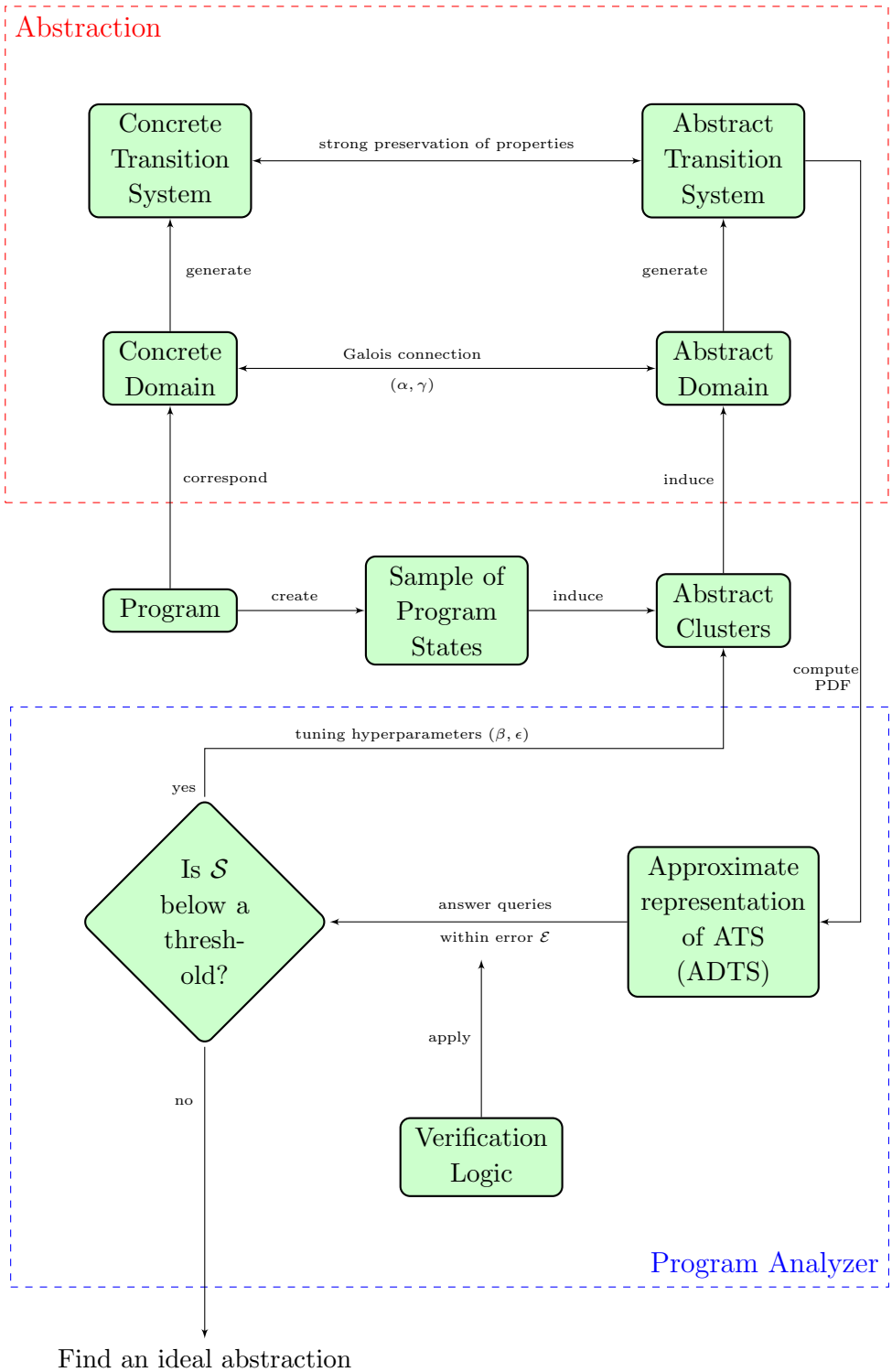


FIGURE 1.1. A flow chart of data-driven abstraction approach.

abstract state does most precisely represent a given concrete program state. To determine such an abstract state may be computationally expensive or even undecidable. A novel technique used in our framework is to provide a symbolic representation of a possibly infinite set of states constituting a cluster by means of a PDF. We use kernel density estimation (KDE), a non-parametric technique, to compute the PDF of a random variable representing the distribution of program states in a cluster, based on the random sampled data drawn from the concrete program. Using the PDFs representing individual clusters, we can probabilistically determine α, i.e. determine the cluster in the abstract domain that most precisely represents a given program state in the concrete domain.

In summary, this work makes the following contributions:

- It introduces a new paradigm of automated program analysis based on data-driven abstraction. The key ideas are to treat a concrete program as a generative process and symbolically approximate a possibly infinite set of states using a PDF. An abstraction of the concrete program is automatically generated from data sampled from the instrumented program using a clustering technique.

- The data-driven abstraction paradigm can result in provably sound abstractions. The abstractions can be created with minimal understanding of the program under analysis, since the knowledge of the program can be obtained and updated by drawing samples of program states.

- It introduces the verification logic, a restricted and bounded variant of CTL, for specifying properties of programs.

- Based on the data-driven abstraction framework, we provide a program analysis tool that can verify, within an error probability, if a given program sat-

isfies properties specified in the verification logic. We experimentally demonstrate the effectiveness of the tool on a test suite of programs from GNU coreutils, diffutils, and grep.

- It quantitatively characterizes the behavior of a program analyzer in terms of its true positive rate, false positive rate, and the number of abstract states.

- Our approach allows one to acquire different abstractions by tuning the hyperparameters, and provides quantitative evaluation of the performance of the tool/analyzer at different abstraction levels. Based on a performance measure, one can determine an "optimal operating point" that serves as an "ideal" abstraction suitable to solve a program analysis problem.

- It provides an automatic approach to tune a program analyzer in accuracy and cost.

1.3 Plan of the Manuscript

This work is organized as follows. We first introduce in Chapter 1, the history and foundation of abstraction interpretation and motivation of creating data-driven abstraction, explain key challenges to quantitate data-driven abstraction process, and summarize our contributions. In Chapter 2, we briefly review the mathematical foundation needed to understand the abstraction interpretation framework. In Chapter 3 we shall build up a thorough and rigorous formalization of the data driven abstraction approach. In what follows in Chapter 4 we study a bounded CTL logic in order to specify properties on programs. Then we systematically define algorithms according to the procedures of our data-driven abstraction in Chapter 5, in particular algorithms to design abstract domains through statistical mechanics and domain refinement. We implement the data-driven abstraction approach and

experimentally validate our approach on standard C programs in Chapter 6. In particular, we analyze the procedure how to tune the hyperparameters to obtain an ideal abstraction. Finally, we summarize a few main lines of research works in the scope of abstract interpretation which is closest to the subject of this work in Chapter 7, and point out a direction that may further extend this work in Chapter 8. We conclude the work with Appendix A, that includes the quantitative evaluations of performance of our approach on a test suite of C programs selected from the open-source GNU coreutils, diffutils, and grep.

Chapter 2
Preliminaries

In this chapter we establish our notation and recall some basic facts from set theory, lattice theory, and temporal logic. Throughout this manuscript, we denote by $\mathbb{R}$ the set of real numbers, $\mathbb{N} = \{1, 2, 3, \dots\}$, and $\overline{\mathbb{N}} = \{0\} \cup \mathbb{N} \cup \{\infty\}$.

2.1 Lattice Theory

In this section, we review the basic terminology and notation of orderings and lattices, and may also discuss some points of elementary logic. These topics are very well known, and all definitions and propositions presented in this section can be found in many standard textbooks, for example, [10].

2.1.1 Orderings

Let $R \subseteq X \times X$ be a relation on a set X. Then

1. R is *total* if for every element $x \in X$ there exists $y \in X$ such that $(x, y) \in R$.

2. R is *reflexive* on X if $(x, x) \in R$, for every $x \in X$.

3. R is *symmetric* if $(y, x) \in R$ whenever $(x, y) \in R$.

4. R is *anti-symmetric* if $(x, y) \in R$ and $(y, x) \in R$ together imply $x = y$.

5. R is *transitive* if $(x, y) \in R$ and $(y, z) \in R$ together imply that $(x, z) \in R$.

6. R is an *equivalence relation* on X if R is reflexive, symmetric, and transitive.

7. R is a *partial order* on X if R is reflexive, anti-symmetric, and transitive.

A *partially ordered set* (poset for short) is a set X equipped with a partial order $\leq \subseteq X \times X$, denoted by $(X, \leq)$. If, in addition, for all $x, y \in X$, at least one of $x \leq y$ and $y \leq x$ holds, a poset $(X, \leq)$ is called a *totally ordered set*.

Let $(X, \leq)$ be a poset and $Y \subseteq X$. We define

$$\begin{aligned} Y^u &:= \{x \in X \mid (\forall y \in Y)\ y \leq x\} \\ Y^\ell &:= \{x \in X \mid (\forall y \in Y)\ x \leq y\}. \end{aligned}$$

The sets Y^u and Y^ℓ are the sets of all *upper bounds* and *lower bounds* of Y, respectively. It is easy to see that $\emptyset^u = \emptyset^\ell = X$. A set Y is *bounded* if it has both an upper bound and a lower bound.

Consider a poset $(X, \leq)$ again. An element $\bot \in X$ is said to be a *bottom element* (least element) of X if $\bot \leq x$ for all $x \in X$. An element $\top \in X$ is a *top element* (greatest element) of X if $x \leq \top$ for all $x \in X$. For the power set $\mathcal{P}(X)$, we have $\bot = \emptyset$ and $\top = X$. Since $\emptyset^u = X$, $\bigvee \emptyset$ exists if and only if X has a bottom element $\bot$. In this case, $\bigvee \emptyset = \bot$. Similarly, if X has a top element $\top$, $\bigwedge \emptyset = \top$.

The partial order $\leq$ is called a *well-order* on X (and $(X, \leq)$ is a well-ordered set) provided that every non-empty subset of X has a least element.

An element $\mu \in X$ is called the *least upper bound* of Y, denoted by $\bigvee_X Y$ ($\bigvee Y$ for short), if

1. $\mu \in Y^u$; and
2. $y \leq z$ for every $y \in Y$ implies that $\mu \leq z$ (i.e. μ is smallest among Y^u).

Similarly, an element $\nu \in X$ is the *greatest lower bound* of Y, denoted by $\bigwedge_X Y$ ($\bigwedge Y$ for short), if

1. $\mu \in Y^\ell$; and
2. $y \geq z$ for every $y \in Y$ implies that $\nu \geq z$ (i.e. ν is greatest among Y^ℓ).

Since not every subset $Y \subseteq X$ has either an upper or a lower bound, least upper bounds and greatest lower bounds do not exist in every case.

Let $(X, \leq)$ be a poset and $A \subseteq X$. Then A is a *downset* of X if

$$\forall y \in A \, (\forall x \in X \, (x \leq y) \Rightarrow x \in A)$$

Let $x \in X$. We define

$$\downarrow x := \{y \in X \mid y \leq x\}.$$

By the transitivity of the order $\leq$, we see that $\downarrow x$ is a downset for any $x \in X$.

2.1.2 Lattices

For the reader who is interested in the subject of lattices, one of good reference books is [10].

Let us define two binary operations on X:

- *meet* $\wedge : X \times X \to X$, $x \wedge y$ is given by the greatest lower bound of $\{x, y\}$;
- *join* $\vee : X \times X \to X$, $x \vee y$ is given by the least upper bound of $\{x, y\}$.

A *lattice* is a non-empty poset X in which every pair of elements has a least upper bound and a greatest lower bound, that is, $x \vee y$ and $x \wedge y$ exist for every pair $x, y \in X$. A *complete lattice* is a non-empty poset X in which every arbitrary subset has a least upper bound (a.k.a *join*) and a greatest lower bound (a.k.a *meet*), that is, $\bigvee Y$ and $\bigwedge Y$ exist for all subsets $Y \subseteq X$.

The least upper bound and the greatest lower bound of the empty set are the bottom and top elements of a complete lattice, respectively.

Example 2.1. Consider the set $(\mathbb{Z}, \leq)$ of integers with the standard partial order relation $\leq$. This set is not a complete lattice since $\mathbb{Z}$ has neither a least upper bound nor a greatest lower bound. But we can lift up this set to the extension

$(\mathbb{Z} \cup \{\pm\infty\}, \leq)$ with

$$-\infty < k < \infty \ (\forall k \in \mathbb{Z})$$

which is a complete lattice.

Let $(X, \leq)$ and $(Z, \subseteq)$ be lattices. We say that a function $f : X \to Z$ is *monotone* if for $x \leq y$ we have $f(x) \leq f(y)$. Monotone maps need not preserve join or meet ($\vee$ or $\wedge$); on the other hand, any map between lattices preserving join and meet is monotone. The function f is *continuous* if for every non-empty subset $Y \subseteq X$, whenever $\bigvee Y$ exists, we have $\bigvee f(Y)$ exists and

$$\bigvee f(Y) = f(\bigvee Y).$$

Similarly, f is *co-continuous* if for every non-empty subset $Y \subseteq X$, whenever $\bigwedge Y$ exists, we have $\bigwedge f(Y)$ exists and

$$\bigwedge f(Y) = f(\bigwedge Y).$$

2.2 Temporal Logic

In this section, we shall introduce a logic to specify a variety of properties of transition systems for programs. We choose temporal logic for this purpose, since temporal logic is a formal language to express sequences of transitions between program states in a program.

One of the simplest branching-time temporal logics is the standard computation tree logic (CTL) [13, 12]. CTL combines temporal operators with path quantification over runs and describes properties of a computation tree. We define a logic CTL_k that restricts and bounds CTL on k steps with $k \in \overline{\mathbb{N}}$ (see examples of CTL_k formulas in Chapter 6 and Appendix A). If $k = \infty$, CTL_k formulas are exactly full CTL formulas. A specification or property of a program is then written as a CTL_k formula.

2.2.1 Define CTL_k Logic

In CTL_k formulas consists of *bounded path quantifiers* and *temporal operators*.

1. There are two bounded path quantifiers

$$\mathbf{A}_k \longleftrightarrow \text{for all } k\text{-paths}$$

$$\mathbf{E}_k \longleftrightarrow \text{there exists a } k\text{-path}$$

   (we will define a k-path soon)

2. and there are the following basic temporal operators

$$\mathbf{X} \longleftrightarrow \text{next time}$$

$$\mathbf{F} \longleftrightarrow \text{sometime in the future}$$

$$\mathbf{G} \longleftrightarrow \text{globally in the future}$$

$$\mathbf{U} \longleftrightarrow \text{until.}$$

Similar to CTL [13], each bounded path quantifier must be immediately followed by exactly one of the operators **X**, **F**, **G**, or **U**, e.g., $\mathbf{E}_k\mathbf{F}$. Formulas in CTL_k consists of

- state formulas (being true or false in a particular state)
- path formulas (being true or false along a particular path).

Let AP be a finite set of atomic propositions. The choice of atomic propositions usually does not involve time. The logic CTL_k is the set of (state) formulas φ inductively formed by the following grammar (syntax):

$$\varphi_k ::= ap \mid \neg\varphi_k \mid \varphi_k^1 \vee \varphi_k^2 \mid \mathbf{E}_k\mathbf{X}(\varphi_{k-1}) \mid \mathbf{E}_k\mathbf{G}(ap) \mid \mathbf{E}_k(ap_1 \,\mathbf{U}\, ap_2) \tag{2.1}$$

where $ap \in \mathrm{AP}$. CTL_k formulas are evaluated at states, and they can deal with both some or all paths, starting from the state they are evaluated at.

2.2.2 Semantics of CTL_k

We define the semantics of CTL_k with respect to a Kripke structure.

Definition 2.2 (Kripke structure [7])**.** A *Kripke structure* M is a triple (S, R, L) where

1. S is a set of states;

2. $R \subseteq S \times S$ is the transition relation; and

3. $L : S \to \mathcal{P}(\mathrm{AP})$ is the proposition labeling function, that assigns to each state a set of atomic propositions true in that state;

where we assume R is total, that is, for each state $\vec{s} \in S$ there exists a state $\vec{s}' \in S$ such that $(\vec{s}, \vec{s}') \in R$.

Definition 2.3 (k-path)**.** Let $k \in \overline{\mathbb{N}}$. We define a *$k$-path* in M to be a path of length $k+1$, i.e., a sequence of states $\pi = \vec{s}_0\vec{s}_1 \cdots \vec{s}_k$, such that for every $0 < i \leq k$, $(\vec{s}_{i-1}, \vec{s}_i) \in R$, where $\vec{s}_0$ is the current state and $\vec{s}_i$ is a successor state of $\vec{s}_{i-1}$.

The relation $M, s \models \varphi$ means that the CTL_k formula φ is true for state s in the structure M. The notation M is often omitted since we refer to the same model most of the time. The satisfiability of each primary CTL_k formula is defined in the

following:

$$\begin{aligned}
\vec{x} \models ap &\iff ap \in L(\vec{x}) \\
\vec{x} \models \neg\varphi_k &\iff \vec{x} \not\models \varphi_k \\
\vec{x} \models \varphi_k^1 \vee \varphi_k^2 &\iff \vec{x} \models \varphi_k^1 \text{ or } \vec{x} \models \varphi_k^2 \\
\vec{x} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1}) &\iff \exists \pi = \vec{s}_0\vec{s}_1\cdots\vec{s}_k \text{ with } \vec{s}_0 = \vec{x} \text{ s.t. } \vec{s}_1 \models \varphi_{k-1} \\
\vec{x} \models \mathbf{E}_k\mathbf{G}(ap) &\iff \exists \pi = \vec{s}_0\vec{s}_1\cdots\vec{s}_k \text{ with } \vec{s}_0 = \vec{x} \text{ s.t. } \vec{s}_i \models ap, \forall 0 \le i \le k \\
\vec{x} \models \mathbf{E}_k(ap_1\mathbf{U}ap_2) &\iff \exists \pi = \vec{s}_0\vec{s}_1\cdots\vec{s}_k \text{ with } \vec{s}_0 = \vec{x} \text{ and } \exists\, 1 \le j \le k \text{ s.t.} \\
&\qquad \vec{s}_i \models ap_1, \forall 0 \le i < j \text{ and } \vec{s}_j \models ap_2.
\end{aligned}$$

Denote by the operator $\mathbf{X}^\ell$ the ℓ-time concatenations of the next time operator $\mathbf{X}$. Other than the above primary operators, we are particularly interested in the following operators in CTL_k for our implementation purpose:

$$\begin{aligned}
\vec{x} \models \mathbf{E}_k\mathbf{F}(ap) &\iff \exists \pi = \vec{s}_0\vec{s}_1\cdots\vec{s}_k \text{ with } \vec{s}_0 = \vec{x} \text{ and } \exists\, 0 \le j \le k \text{ s.t.} \vec{s}_j \models ap \\
\vec{x} \models \mathbf{A}_k\mathbf{G}(ap) &\iff \forall \pi = \vec{s}_0\vec{s}_1\cdots\vec{s}_k \text{ with } \vec{s}_0 = \vec{x},\ \vec{s}_i \models ap, \forall 0 \le i \le k \\
\vec{x} \models \mathbf{E}_k\mathbf{X}^k(ap) &\iff \exists \pi = \vec{s}_0\vec{s}_1\cdots\vec{s}_k \text{ with } \vec{s}_0 = \vec{x} \text{ s.t. } \vec{s}_k \models ap.
\end{aligned}$$

These operators states that

1. $\mathbf{E}_k\mathbf{F}(ap)$ is true at $\vec{x}$ if and only if ap is true *somewhere* on some k-path starting from $\vec{x}$, i.e., it can reach a state within k steps from $\vec{x}$ satisfying ap;

2. $\mathbf{A}_k\mathbf{G}(ap)$ is true at $\vec{x}$ if and only if ap is true *everywhere* on *every* k-path starting from $\vec{x}$;

3. $\mathbf{E}_k\mathbf{X}^k(ap)$ is true at $\vec{x}$ if and only if ap is true at the last state on some k-path starting from $\vec{x}$, i.e., it can reach a state k steps away from $\vec{x}$ satisfying ap.

Chapter 3
Theory

The abstraction process involves a deep understanding of mathematical structures of a program and their relationships. This chapter formally builds the mathematical foundation of abstraction theory for our data-driven abstraction process.

3.1 Semantics of Programs

The theory of abstract interpretation [8] is used to approximate the semantics of a program. The semantics of a program is an infinite mathematical model of what the program does. It usually computes all possible program states occurring in the execution of a program for all possible input, which may practically be uncomputable. The basic idea of abstraction in the framework of Cousot [8] is that a (finite or infinite) set of program states is approximated by a finite set of abstract states (a superset of the set of program states). This thus requires a computable set of program states that correctly interprets the program.

We shall develop a mathematical semantics of programs in this section, since it is the foundation (or base) on which an abstraction is built.

3.1.1 Program States and Distances

We introduce a few concepts related to the semantics of a program, following [8] with some simplifications.

A program $P = (\Sigma, \mathcal{L}, \mathcal{T}, L)$ consists of a set Σ of all program states, a set $\mathcal{L}$ of statement locations, a set $\mathcal{T}$ of transitions, and the proposition labeling function L on Σ. We define a *program state* to be a valuation of all program variables. For

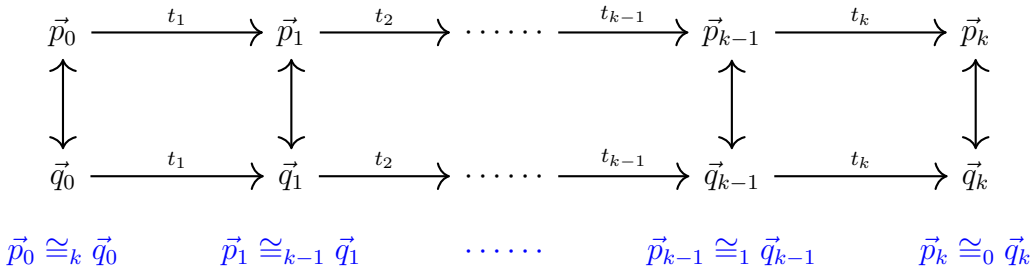


FIGURE 3.1. A sequence of transitions of a program

simplicity, we will denote a program state $\vec{x} \in \Sigma$ (consisting of ℓ variables) as

$$\vec{x} = (x_1, \ldots, x_\ell), \quad x_j \in \mathbb{R}, \forall 1 \leq j \leq \ell.$$

Each *transition* in $\mathcal{T}$ is a relation $t \subseteq \Sigma \times \Sigma$ over program variables that relates program states to their successors after the transition. For states $\vec{p}, \vec{q} \in \Sigma$ and a transition $t \in \mathcal{T}$, we say $\vec{p} \xrightarrow{t} \vec{q}$ if $(\vec{p}, \vec{q}) \in t$.

We will then define *bounded bisimilarity* between program states and a distance metric over program state space, that respects the bisimilarity relation.

Let $k \in \overline{\mathbb{N}}$. The idea of $\vec{p}$ being k-step bisimilar to $\vec{q}$ is that $\vec{p}$ can imitate the behaviour of $\vec{q}$ for up to k steps, and vice versa, and "divergence" takes place after k steps, as shown in Figure 3.1 (for simplicity, we consider only one successor of a state for each transition).

Definition 3.1 (k-step bisimilarity)**.** The k-step bisimilarity on Σ is defined to be the largest symmetric relation $\cong_k \subseteq \Sigma \times \Sigma$ such that for any two states $\vec{p}, \vec{q} \in \Sigma$, $\vec{p} \cong_k \vec{q}$ implies that $L(\vec{p}) = L(\vec{q})$, and $k \in \overline{\mathbb{N}}$ is the largest number for which the following hold:

1. for any sequence of transitions of length k,

$$\vec{p}_0 \xrightarrow{t_1} \vec{p}_1 \xrightarrow{t_2} \cdots\cdots \xrightarrow{t_{k-1}} \vec{p}_{k-1} \xrightarrow{t_k} \vec{p}_k, \quad \vec{p}_0 := \vec{p},$$

there exists a sequence of transitions of length k,

$$\vec{q}_0 \xrightarrow{t_1} \vec{q}_1 \xrightarrow{t_2} \cdots\cdots \xrightarrow{t_{k-1}} \vec{q}_{k-1} \xrightarrow{t_k} \vec{q}_k, \quad \vec{q}_0 := \vec{q},$$

such that

$$\vec{p}_j \cong_{k-j} \vec{q}_j, \quad \forall 1 \leq j \leq k;$$

2. the same as the previous statement with the roles of $\vec{p}_j$ and $\vec{q}_j$ interchanged.

Remark 3.2. It is worth mentioning that

1. Two states $\vec{p}$ and $\vec{q}$ are 0-step bisimilar if $L(\vec{p}) = L(\vec{q})$ and they are not 1-step bisimilar.

2. If two program states are k-step bisimilar, they are not s-step bisimilar for any $s \in \overline{\mathbb{N}}$ with $s > k$.

The performance of our clustering algorithm (cf. Algorithm 2) under unsupervised settings depends critically on a good and meaningful distance metric (or pseudometric) over the input program state space. Our method is to use the information of k-step bisimilarity to find meaningful clusters. More precisely, given a bounded bisimilarity relation between program states, we would like to construct a clustering which assigns bisimilar (at least k-step bisimilar for some k) pairs of states into the same cluster. Therefore, a good metric needs to properly reflect the k-step bisimilarity relationships between program states in order to maximize the clustering performance, that is, a good metric should assign small distances between bisimilar pairs of states (bisimilar states end up close to each other).

Definition 3.3 (Distances between program states)**.** Let $\vec{p}, \vec{q} \in \Sigma$. The *distance between program states* is a function $d : \Sigma \times \Sigma \to [0, 1]$ defined by

$$d(\vec{p}, \vec{q}) = 2^{-k} \tag{3.1}$$

where $k \in \overline{\mathbb{N}}$ is the number such that $\vec{p}$ and $\vec{q}$ are k-step bisimilar.

Proposition 3.4. *The function d defines a pseudometric on the program state space Σ.*

Proof. Since a state $\vec{p}$ is infinitely bisimilar to itself,

$$d(\vec{p}, \vec{p}) = \lim_{k \to \infty} 2^{-k} = 0.$$

However, for two distinct states $\vec{p}$ and $\vec{q}$, it is possible that $\vec{p}$ can imitate everything that $\vec{q}$ can do and vice versa, so $\vec{p}$ and $\vec{q}$ are ∞-bisimilar and thus $d(\vec{p}, \vec{q})$ is zero even for $\vec{p} \neq \vec{q}$. Next, since $\cong_k$ is a symmetric relation, $d(\vec{p}, \vec{q}) = d(\vec{q}, \vec{p})$. Last, d is subadditive because if $\vec{p} \cong_{k_1} \vec{q}$ and $\vec{q} \cong_{k_2} \vec{r}$ for some $k_1, k_2 \in \overline{\mathbb{N}}$, then $\vec{p} \cong_{k_3} \vec{r}$ with $k_3 = \min(k_1, k_2)$. It follows that d defines a pseudometric on the program state space Σ. □

3.1.2 The Concrete and Abstract Domains of a Program

In this section we equip the set of program states and its power set with a partial ordering for each, and use them to define the concrete and abstract domains required for abstract interpretation.

Definition 3.5 (Concrete domain)**.** Let P be a program with corresponding space Σ of program states. The *concrete domain* for P, $\mathcal{C} = (\Sigma, \leq)$, is the set Σ of program states equipped with a partial order $\leq$ where for two states $\vec{x} = (x_1, \ldots, x_\ell)$ and $\vec{y} = (y_1, \ldots, y_\ell)$, $\vec{x} \leq \vec{y}$ if $x_j \leq y_j$ for all $1 \leq j \leq \ell$.

In this work, we restrict the concrete domain on the real numbers, although a user may define a well ordering with respect to their particular domain.

Our proposed data-driven technique depends on the concept of a cluster, a well-known concept in machine learning. Our technique does not learn clusters from a set of points in the traditional sense. Instead we iteratively construct clusters in an abstract domain based on *root* points. Each cluster consists of a downset of the root. We shall define the abstract domain for a program P as a set of clusters.

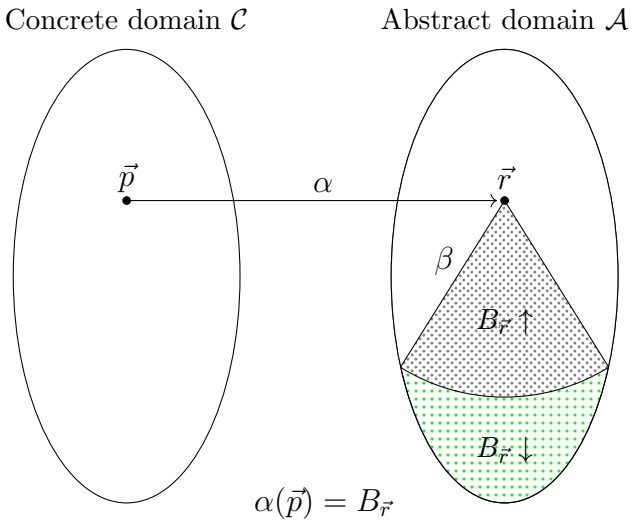


FIGURE 3.2. Inducing abstract clusters $B_{\vec{r}} = B_{\vec{r}}\uparrow \cup B_{\vec{r}}\downarrow$

Definition 3.6 (Abstract cluster rooted at a point)**.** Let $\vec{r} \in \mathcal{C}$. The *abstract cluster rooted at* $\vec{r}$ is the set of program states in the downset $\downarrow \vec{r}$:

$$B_{\vec{r}} = \downarrow \vec{r} := \{\vec{q} \in \mathcal{C} \mid \vec{q} \leq \vec{r}\}.$$

Fix a $\beta > 0$. Partitioning $B_{\vec{r}}$ into two disjoint sets, we define the *upper abstract cluster rooted at* $\vec{r}$ as

$$B_{\vec{r}}\uparrow = \{\vec{q} \in B_{\vec{r}} \mid d(\vec{r}, \vec{q}) \leq \beta\}.$$

Similarly, we define the *lower abstract cluster rooted at* $\vec{r}$ as

$$B_{\vec{r}}\downarrow = \{\vec{q} \in B_{\vec{r}} \mid d(\vec{r}, \vec{q}) > \beta\}.$$

As is shown in Figure 3.2, the cluster $B_{\vec{r}}$ in $\mathcal{A}$ rooted at $\vec{r}$ consists of all states in the downset of the root. When the context is clear, we will simply refer to $B_{\vec{r}}$ as a cluster, $B_{\vec{r}}\uparrow$ as an upper cluster, and $B_{\vec{r}}\downarrow$ as a lower cluster. We will also refer to a cluster without specifying its root.

Lemma 3.7. *Clusters and upper clusters rooted at some program states are non-empty.*

Proof. Let $B_{\vec{p}}$ be a cluster rooted at $\vec{p}$. Since $\vec{p} \leq \vec{p}$, $\vec{p} \in B_{\vec{p}}$. Thus, $B_{\vec{p}} \neq \emptyset$. Moreover, since $d(\vec{p}, \vec{p}) = 0 < \beta$, $\vec{p} \in B_{\vec{p}}\uparrow$. Hence, $B_{\vec{p}}\uparrow \neq \emptyset$. □

The following two lemmas establish clearly that a cluster can be split into two unique partitions.

Lemma 3.8. *For a root point $\vec{p} \in \mathcal{C}$, $B_{\vec{p}}\uparrow \cup B_{\vec{p}}\downarrow = B_{\vec{p}}$.*

Proof. This is clear by the definition of a cluster. □

Lemma 3.9. *For a root point $\vec{p} \in \mathcal{C}$, $B_{\vec{p}}\uparrow \cap B_{\vec{p}}\downarrow = \emptyset$.*

Proof. Let $\vec{p} \in \mathcal{C}$ and suppose $B_{\vec{p}}\uparrow \cap B_{\vec{p}}\downarrow \neq \emptyset$. Then there exists a program state $\vec{e} \in B_{\vec{p}}\uparrow \cap B_{\vec{p}}\downarrow$. Then $d(\vec{p}, \vec{e}) \leq \beta$ and $d(\vec{p}, \vec{e}) > \beta$. This is a contradiction since β is fixed. □

Furthermore, because β is fixed, we have

Lemma 3.10. *The root of a cluster uniquely determines the cluster.*

Next, we establish a hierarchy among all clusters in a domain.

Lemma 3.11. *For root points $\vec{p}, \vec{q} \in \mathcal{C}$ we have $\vec{q} \leq \vec{p}$ if and only if $B_{\vec{q}} \subseteq B_{\vec{p}}$.*

Proof. For $\Rightarrow$: Assume $\vec{q} \leq \vec{p}$. Let $\vec{x} \in B_{\vec{q}}$. So $\vec{x} \leq \vec{q}$ and thus $\vec{x} \leq \vec{p}$. By the definition of a cluster, we have $\vec{x} \in B_{\vec{p}}$ and hence $B_{\vec{q}} \subseteq B_{\vec{p}}$.

For $\Leftarrow$: Assume $B_{\vec{q}} \subseteq B_{\vec{p}}$. In case $B_{\vec{q}} = B_{\vec{p}}$, it is clear that $\vec{q} = \vec{p}$. For the case where $B_{\vec{q}} \subset B_{\vec{p}}$ we want to show $\vec{q} < \vec{p}$. Suppose that $\vec{p} \leq \vec{q}$. It follows from the first part of this lemma that $B_{\vec{p}} \subseteq B_{\vec{q}}$. This yields a contradiction. □

We now define the abstract domain as the set of all clusters rooted at points in the concrete domain.

Definition 3.12 (Abstract domain)**.** For a program P with concrete domain $\mathcal{C} = (\Sigma, \leq)$, the *abstract domain*

$$\mathcal{A} = (2^{\Sigma}, \subseteq)$$

is defined to be the power set of Σ, where any $B_{\vec{p}}$ in $\mathcal{A}$ is a cluster rooted at some state $\vec{p} \in \mathcal{C}$, and all such $B_{\vec{p}}$ are partially ordered by inclusion $\subseteq$.

Assume the concrete domain maintains a supremum and an infimum element respectively:

$$\vec{\top}_{\mathcal{C}} = (\top_1, \ldots, \top_m) \in \mathcal{C}, \quad \vec{\bot}_{\mathcal{C}} = (\bot_1, \ldots, \bot_m) \in \mathcal{C}.$$

It follows that $\mathcal{A}$ defines a supremum element

$$\top_{\mathcal{A}} = \bigcup_{\vec{p} \in \mathcal{C}} \{\vec{p}\} = \mathcal{C}.$$

The singleton set containing $\vec{\bot}_{\mathcal{C}}$ is the infimum of $\mathcal{A}$, i.e. $\bot_{\mathcal{A}} = \{\vec{\bot}_{\mathcal{C}}\}$.

Lemma 3.13. *The abstract domain $\mathcal{A}$ defines a complete lattice.*

Proof. Since there are an infimum and a supremum in $\mathcal{A}$, for all subsets $X \subseteq \mathcal{A}$ there exist a meet and a join. Thus, $\mathcal{A}$ is a complete lattice. □

3.2 Abstraction and Property Preservation

This section introduces the construction abstractions used to verify properties of complex programs.

3.2.1 Abstraction and Concretization Mappings

To ensure a valid abstract interpretation (cf. [8]) we must relate the abstract domain to the concrete domain by means of a Galois connection which consists of an abstraction mapping α and a concretization mapping γ.

We begin by defining the abstraction mapping via taking a state in the concrete domain and mapping it to the most appropriate cluster (i.e. the cluster corresponding to the closest root above it).

Definition 3.14 (Abstraction Mapping)**.** Let P be a program with corresponding space Σ of program states. Let $\mathcal{C}$ be the concrete domain for P, and $\mathcal{A}$ the abstract domain with $C_1, \ldots, C_n$ the n clusters of $\mathcal{A}$, each with unique respective root $\vec{r}_1, \ldots, \vec{r}_n$. We assume that $d(\vec{r}_i, \vec{r}_j) \geq \epsilon$ for $i \neq j$, where ϵ is a hyperparameter. The *abstraction mapping* $\alpha : \mathcal{C} \to \mathcal{A}$ is defined by $\alpha(\vec{p}) = C_i$, $\vec{p} \in \mathcal{C}$, where C_i (for some $1 \leq i \leq n$) is determined such that

1. $\vec{p} \leq \vec{r}_i$; and

2. if $\vec{p} \leq \vec{r}_j$ then $d(\vec{p}, \vec{r}_i) \leq d(\vec{p}, \vec{r}_j)$ for all $j \neq i$ with ties broken arbitrarily.

The hyperparameter ϵ is the least distance between any two root roots (see Section 5.1). Definition 3.14 maps a state $\vec{p}$ in the concrete domain to the cluster in abstract domain $\mathcal{A}$ whose root is greater than or equal to $\vec{p}$ and is closer to $\vec{p}$ than any other root. In particular, $\alpha(\vec{p})$ identifies the element in $\mathcal{A}$ that most precisely represents the state $\vec{p}$. As shown in Figure 3.2, state $\vec{p}$ is mapped under α to the cluster $B_{\vec{r}}$.

Since a cluster is defined by a root point, we have a natural definition of a concretization mapping from a cluster in the abstract domain to a state in the concrete domain; specifically, a cluster is mapped to its root in the concrete domain.

Definition 3.15 (Concretization Mapping)**.** Let $B_{\vec{p}} \in \mathcal{A}$ be a cluster rooted at a state $\vec{p} \in \mathcal{C}$. The *concretization mapping* $\gamma : \mathcal{A} \to \mathcal{C}$ is defined by $\gamma(B_{\vec{p}}) = \vec{p}$.

We now prove some important properties required of the abstraction and concretization maps to ensure a Galois connection.

Lemma 3.16. *The abstraction mapping $\alpha : \mathcal{C} \to \mathcal{A}$ is monotone, i.e., order preserving.*

Proof. Let $\vec{p}, \vec{q} \in \mathcal{C}$ with $\vec{p} \leq \vec{q}$. We need to show $\alpha(\vec{p}) \subseteq \alpha(\vec{q})$. Suppose $\alpha(\vec{p}) = B_{\vec{r}_i}$ and $\alpha(\vec{q}) = B_{\vec{r}_j}$ for some root points $\vec{r}_i$ and $\vec{r}_j$. Then $\vec{p} \leq \vec{r}_i$ and $\vec{q} \leq \vec{r}_j$. In view of Lemma 3.11, to prove the monotonicity of α, it remains to show $\vec{r}_i \leq \vec{r}_j$. If $\vec{q} < \vec{r}_i$, then by Definition 3.14,

$$d(\vec{q}, \vec{r}_j) < d(\vec{q}, \vec{r}_i).$$

But then

$$d(\vec{p}, \vec{r}_j) \leq d(\vec{p}, \vec{r}_i).$$

This contradicts to Definition 3.14, since $\vec{r}_i$ is supposed to be closer to $\vec{p}$ than any other root. Therefore, the only possibility is that $\vec{q} \geq \vec{r}_i$. It follows that $\vec{p} \leq \vec{r}_i \leq \vec{q} \leq \vec{r}_j$ and so $B_{\vec{r}_i} \subseteq B_{\vec{r}_j}$. □

Lemma 3.17. *The concretization mapping $\gamma : \mathcal{A} \to \mathcal{C}$ is order preserving.*

Proof. Let $B_{\vec{p}}, B_{\vec{q}} \in \mathcal{A}$ be clusters rooted at $\vec{p}$ and $\vec{q}$ respectively with $B_{\vec{p}} \subseteq B_{\vec{q}}$. By Lemma 3.11, $\vec{p} \leq \vec{q}$. Thus, in view of Definition 3.15, $\gamma(B_{\vec{p}}) = \vec{p} \leq \vec{q} = \gamma(B_{\vec{q}})$. □

3.2.2 The Galois Connection

For a program P, we define the concrete domain $\mathcal{C} = (\Sigma, \leq)$ as an infinite state space of program states and the corresponding abstract domain $\mathcal{A} = (2^{\Sigma}, \subseteq)$ as the set of clusters ordered by inclusion.

We have properly defined correspondence (α, γ) between the concrete and abstract domains. Next we must verify the pair of monotone mappings α and γ form a Galois connection (named after Évariste Galois) between the concrete and the abstract domains to ensure a valid abstract interpretation.

A Galois connection framework will require partial orderings on the concrete and the abstract domains, which clearly have.

Definition 3.18 (Galois Connection [24])**.** For partially ordered sets $\mathcal{C}$ and $\mathcal{A}$, a pair (α, γ) of monotone maps, $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{C}$, is a *Galois connection* if and only if for any $A \in \mathcal{A}$ and $\vec{p} \in \mathcal{C}$, $\alpha(\vec{p}) \subseteq A \Leftrightarrow \vec{p} \leq \gamma(A)$.

We may find it convenient to observe that (α, γ) forms a Galois connection if and only if $\vec{p} \leq \gamma(\alpha(\vec{p}))$ and $\alpha(\gamma(A)) \subseteq A$ for any $\vec{p} \in \mathcal{C}$ and $A \in \mathcal{A}$.

The partial order relations on the concrete and abstract domains reflect their respective relative precisions. For example, $\vec{p} \leq \vec{q}$ implies that $\vec{q}$ is an approximation of $\vec{p}$ (or $\vec{q}$ carries less information than $\vec{p}$); and $B_{\vec{p}} \subseteq B_{\vec{q}}$ implies that $B_{\vec{p}}$ is more precise than $B_{\vec{q}}$. This brings out a basic yet important condition for the abstraction and concretization maps, that is, they preserve the exact approximation orderings (monotonicity). Galois connection then relates the relative precisions of the concrete and abstract domains in the sense that an abstract state A approximates a concrete state $\vec{p}$ if $\alpha(\vec{p}) \subseteq A$ (i.e. $\vec{p} \leq \gamma(A)$).

Proposition 3.19. *The maps α and γ defined in Definitions 3.14 and 3.15 satisfy a Galois connection.*

Proof. Let $\vec{x} \in \mathcal{C}$ be such that $\alpha(\vec{x}) = B_{\vec{x}} \in \mathcal{A}$, a cluster rooted at the point $\vec{x}$. To prove that (α, γ) forms a Galois connection, it suffices to prove that for any $\vec{b} \in \mathcal{C}$,

$$\alpha(\vec{b}) \subseteq \alpha(\vec{x}) \Longleftrightarrow \vec{b} \leq \gamma(\alpha(\vec{x})).$$

Since $\alpha(\vec{x}) = B_{\vec{x}}$ and $\gamma(B_{\vec{x}}) = \vec{x}$, we have

$$\begin{aligned} \vec{b} \leq \gamma(\alpha(\vec{x})) &\iff \vec{b} \leq \gamma(B_{\vec{x}}) \\ &\iff \vec{b} \leq \vec{x} \\ &\iff \alpha(\vec{b}) \subseteq \alpha(\vec{x}), \end{aligned}$$

where we use the fact that α and γ are order preserving. □

Lemma 3.20. *The abstraction mapping $\alpha : \mathcal{C} \to \mathcal{A}$ is continuous.*

Proof. Suppose that T is a subset of $\mathcal{C}$ such that the least upper bound of T, $\bigvee T$, exists. By the definition of least upper bound, $\vec{t} \leq \bigvee T$ for all $\vec{t} \in T$. It follows from the monotonicity of α that $\alpha(\vec{t}) \subseteq \alpha(\bigvee T)$. This implies that $\alpha(\bigvee T)$ is an upper bound for the set $\alpha(T) = \{\alpha(\vec{t}) \mid \vec{t} \in T\}$. Let A be another upper bound of $\alpha(T)$. Then $\alpha(\vec{t}) \subseteq A$ and so $\vec{t} \leq \gamma(A)$ for all $\vec{t} \in T$ since (α, γ) is a Galois connection. The equation $\vec{t} \leq \gamma(A)$ for all $\vec{t} \in T$ implies that $\gamma(A)$ is an upper bound of T. By the definition of least upper bound, we have $\bigvee T \leq \gamma(A)$. By the monotonicity of α and the definition of Galois connections,

$$\alpha(\bigvee T) \subseteq \alpha(\gamma(A)) \subseteq A$$

which proves that $\alpha(\bigvee T)$ is the least upper bound of $\alpha(T)$, i.e.

$$\alpha(\bigvee T) = \bigvee \alpha(T) = \bigvee \{\alpha(\vec{t}) \mid \vec{t} \in T\}.$$

□

Corollary 3.21. *The concretization map $\gamma : \mathcal{A} \to \mathcal{C}$ is co-continuous.*

Proof. It is sufficient to show γ preserves greatest lower bounds. We have shown in Lemma 3.20 that α preserves least upper bounds. By the duality principle, if for any $W \subseteq \mathcal{A}$, $\bigwedge W$ exists, then

$$\gamma(\bigwedge W) = \bigwedge \{\gamma(A) \mid A \in W\}.$$

□

3.2.3 Abstract Transition System

In abstraction-based verification, the weak preservation means for a formula φ defined in a temporal language:

$$\mathcal{A} \models \varphi \Longrightarrow \mathcal{C} \models \varphi.$$

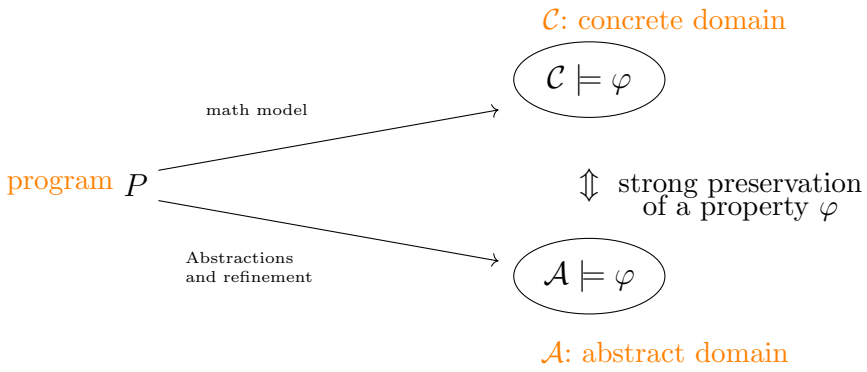


FIGURE 3.3. Verification big picture

It is obvious that abstract verification of a property on the abstract side might produce false positives because an abstract domain hides certain details of the concrete domain. Therefore we would like to construct an abstraction that makes properties *strongly preserved* (cf. Figure 3.3), that is, a formula φ holds on the abstract domain if and only if it holds on the concrete domain:

$$\mathcal{A} \models \varphi \iff \mathcal{C} \models \varphi. \tag{3.2}$$

Property preservations between the concrete and abstract domains only depend on the abstract domain itself, but not on how to abstractly interpret atomic propositions and logical operators (for our case check (2.1)) on the abstract domain.

The verification of a temporal property will be executed in a sound (and complete) abstraction. For that we should now introduce abstract transitions and abstract transition systems.

Definition 3.22 (Abstract Transitions)**.** For each transition $t \subseteq \Sigma \times \Sigma$, we define an abstract transition $t^{\#} \subseteq 2^{\Sigma} \times 2^{\Sigma}$ such that $t^{\#} = \alpha \circ t \circ \gamma$.

In this case, we say that the abstract transition $t^{\#}$ is a *sound and complete* approximation on $\mathcal{A}$ for the concrete transition t.

While soundness (correctness) is a very basic requirement for our data driven abstraction approach (in fact, for any approximation approach), a strict loss of

information may build up in the abstract-side computations. *Completeness* in abstract interpretation aims to ensure no such loss of precisions. It has been shown that strong preservation of abstraction-based verification can be carried via sound and complete abstract interpretation (for example, see [26]).

Transition systems constitute a mathematical object to formalize the evolution of a program. We need to generate an abstract transition system which strongly preserves desired properties of the concrete transition system.

Definition 3.23. A (concrete) *transition system* (CTS) is a pair of the form $(\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is the concrete domain, and $\mathcal{T}$ is the set of transitions on $\Sigma \times \Sigma$. Similarly, we define the corresponding *abstract transition system* (ATS) to be $(\mathcal{A}, \mathcal{T}^{\#})$, where $\mathcal{A}$ is the abstract domain, and $\mathcal{T}^{\#}$ is the set of abstract transitions on $2^{\Sigma} \times 2^{\Sigma}$ as defined in Definition 3.22.

Thus, the domains $(\mathcal{C}, \mathcal{A})$ and the Galois connection (α, γ) together with the ATS $\mathcal{T}^{\#}$ induce a sound and complete abstraction. We will use such an abstraction to check for temporal properties of programs.

Chapter 4
Verification Logic

As we have seen the approach to bring about preservation of properties from concrete domain to abstract domain is to iteratively refine the abstract model such that each abstract state (in our case, abstract cluster) represents a set of concrete states which all agree on certain properties. In order to construct a good abstract model that can preserve some set of logical formulas, each abstract state should be consistent with all formulas in the logic semantics. This coincides with the construction of a quotient of the abstract model under some behavioural equivalence that can be defined in terms of program states and transition systems of a program (we have defined state bisimilarity for this purpose).

We have defined, in Section 2.2, a logic CTL_k, a fragment of CTL that expresses properties that hold true over sequences of transitions of length k with $k \in \overline{\mathbb{N}}$. Therefore, in this chapter, we investigate the correspondence between the behavioural equivalence defined by k-bisimilarity between states and the logical equivalence induced by the temporal language CTL_k.

Theorem 4.1. *Let $\vec{p}, \vec{q} \in \Sigma$ and φ_k a CTL_k formula for a $k \in \overline{\mathbb{N}}$. Let $s \in \overline{\mathbb{N}}$ with $s \geq k$. Then the following are equivalent:*

1. *$\vec{p} \cong_s \vec{q}$; and*
2. *$\vec{p} \models \varphi_k$ if and only if $\vec{q} \models \varphi_k$.*

Remark 4.2. If two program states $\vec{p}$ and $\vec{q}$ are s-step bisimilar with $s > k$, then they agree on the same CTL_k formulas, too. So it suffices to prove Theorem 4.1 for $s = k$.

Proof of Theorem 4.1. We first prove statement 1 $\Rightarrow$ statement 2:

$$\vec{p} \cong_k \vec{q} \Longrightarrow (\vec{p} \models \varphi_k \Longleftrightarrow \vec{q} \models \varphi_k)$$

i.e., prove k-step bisimilar program states satisfy exactly the same set of CTL_k formulas. Since the logic CTL_k is defined as a set of state formulas (cf. the CTL_k syntax (2.1)), it suffices to prove it by the induction on the structures of CTL_k formulas and check it case by case.

For the base case let ap be an atomic proposition. Since $\vec{p} \cong_k \vec{q}$, by definition of k-bisimilarity, $L(\vec{p}) = L(\vec{q})$. This implies that $\vec{p} \models ap$ if and only if $\vec{q} \models ap$.

The following are the inductive steps. Assume that (statement 1 $\Rightarrow$ statement 2) holds for two CTL_k formulas φ_k, ϕ_k (this is the induction hypothesis).

(1) Consider $\neg\varphi_k$:

$$\begin{aligned} \vec{p} \models \neg\varphi_k &\iff \vec{p} \not\models \varphi_k \\ \text{(by induction hypothesis)} &\iff \vec{q} \not\models \varphi_k \\ &\iff \vec{q} \models \neg\varphi_k. \end{aligned}$$

(2) Consider $\varphi_k \vee \phi_k$:

$$\begin{aligned} \vec{p} \models (\varphi_k \vee \phi_k) &\iff (\vec{p} \models \varphi_k) \vee (\vec{p} \models \phi_k) \\ \text{(by induction hypothesis)} &\iff (\vec{q} \models \varphi_k) \vee (\vec{q} \models \phi_k) \\ &\iff \vec{q} \models (\varphi_k \vee \phi_k). \end{aligned}$$

(3) Consider $\mathbf{E}_k\mathbf{X}(\varphi_{k-1})$. Assume $\vec{p} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$. Then there is a transition t and a state $\vec{p}^{(1)}$ such that $\vec{p} \xrightarrow{t} \vec{p}^{(1)}$ and $\vec{p}^{(1)} \models \varphi_{k-1}$. The bisimilarity $\vec{p} \cong_k \vec{q}$ implies that

$$\forall \vec{p}^{(1)} \left((\vec{p} \xrightarrow{t} \vec{p}^{(1)}) \Longrightarrow \exists \vec{q}^{(1)} \left((\vec{q} \xrightarrow{t} \vec{q}^{(1)}) \wedge (\vec{p}^{(1)} \cong_{k-1} \vec{q}^{(1)}) \right) \right) \tag{4.1}$$

By the induction hypothesis, $\vec{p}^{(1)} \models \varphi_{k-1}$ if and only if $\vec{q}^{(1)} \models \varphi_{k-1}$. This implies that $\vec{q} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$. The converse direction, i.e., to show $\vec{q} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1}) \Rightarrow \vec{p} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$ follows from the fact that $\cong_k$ is a symmetric relation.

(4) Consider $\mathbf{E}_k\mathbf{G}(ap)$. Assume $\vec{p} \models \mathbf{E}_k\mathbf{G}(ap)$. Let $\vec{p}^{(0)} := \vec{p}$ and $\vec{q}^{(0)} := \vec{q}$. Then there is a k-path $\pi_1 = \vec{p}\,\vec{p}^{(1)} \cdots \vec{p}^{(k)}$ such that $\vec{p}^{(i)} \models ap$ for all $0 \leq i \leq k$ (every state on π_1 satisfies ap). Since $\vec{p} \cong_k \vec{q}$, from the equation (4.1) we see that, for π_1 there exists a corresponding k-path $\pi_2 = \vec{q}\,\vec{q}^{(1)} \cdots \vec{q}^{(k)}$ such that for all $0 \leq i \leq k$, $\vec{p}^{(i)} \cong_{k-i} \vec{q}^{(i)}$. By the induction hypothesis, $\vec{p}^{(i)} \models ap$ if and only if $\vec{q}^{(i)} \models ap$ for all $0 \leq i \leq k$. This implies that $\vec{q} \models \mathbf{E}_k\mathbf{G}(ap)$. The converse direction, i.e., to show $\vec{q} \models \mathbf{E}_k\mathbf{G}(ap) \Rightarrow \vec{p} \models \mathbf{E}_k\mathbf{G}(ap)$ follows by symmetry.

(5) Consider $\mathbf{E}_k(ap_1\mathbf{U}ap_2)$. Suppose $\vec{p} \models \mathbf{E}_k(ap_1\mathbf{U}ap_2)$. Let $\vec{p}^{(0)} := \vec{p}$. Then there exists a path $\pi_1 = \vec{p}\,\vec{p}^{(1)} \cdots \vec{p}^{(k)}$ such that there is a $1 \leq j \leq k$ for which

- $\vec{p}^{(i)} \models ap_1$ for all $0 \leq i < j$; and
- $\vec{p}^{(j)} \models ap_2$.

Since $\vec{p} \cong_k \vec{q}$, there exists a corresponding path $\pi_2 = \vec{q}\,\vec{q}^{(1)} \cdots \vec{q}^{(k)}$ such that for all $1 \leq i \leq k$, $\vec{p}^{(i)} \cong_{k-i} \vec{q}^{(i)}$. By the induction hypothesis, we must have $\vec{q}^{(i)} \models ap_1$ for all $0 \leq i < j$, and $\vec{q}^{(j)} \models ap_2$. This implies that $\vec{q} \models \mathbf{E}_k(ap_1\mathbf{U}ap_2)$. The converse direction, i.e., to show $\vec{q} \models \mathbf{E}_k(ap_1\mathbf{U}ap_2) \Rightarrow \vec{p} \models \mathbf{E}_k(ap_1\mathbf{U}ap_2)$, is similar.

Next we shall prove statement 2 $\Rightarrow$ statement 1:

$$(\vec{p} \models \varphi_k \Longleftrightarrow \vec{q} \models \varphi_k) \Longrightarrow \vec{p} \cong_k \vec{q} \tag{4.2}$$

that is, to characterize program states up to k-step bisimilarity by CTL_k formulas: The proof of (4.2) can be done by induction on k.

For the base case ($k = 0$), since $\vec{p} \models ap \Leftrightarrow \vec{q} \models ap$, $L(\vec{p}) = L(\vec{q})$. By the definition of bisimilar states, we have $\vec{p} \cong_0 \vec{q}$.

The following is the induction step. Assume the statement (4.2) is true for k. We shall show that it is true for $k+1$.

To do so, we show that if $\vec{p} \not\cong_{k+1} \vec{q}$, then there is CTL_{k+1} formula φ such that $\vec{p} \models \varphi$, but $\vec{q} \not\models \varphi$. Let $\vec{p}_0 := \vec{p}$ and $\vec{q}_0 := \vec{q}$. The case that $\vec{p} \not\cong_{k+1} \vec{q}$ can only happen if there is a $(k+1)$-path starting from $\vec{p}$, say

$$\pi_{\vec{p}} = \vec{p}_0 \vec{p}_1 \cdots \vec{p}_k \vec{p}_{k+1}$$

but without a corresponding $(k+1)$-path starting from $\vec{q}$, say

$$\pi_{\vec{q}} = \vec{q}_0 \vec{q}_1 \cdots \vec{q}_k \vec{q}_{k+1}$$

such that

$$\vec{p}_j \cong_{k+1-j} \vec{q}_j,\ 0 \leq j \leq k+1 \tag{4.3}$$

(note: the roles of $\vec{p}_j$ and $\vec{q}_j$ can be interchanged). This can be divided into two possibilities:

- $\vec{p} \not\cong_k \vec{q}$;
- $\vec{p} \cong_k \vec{q}$ but $\vec{p} \not\cong_{k+1} \vec{q}$.

The former case follows immediately from the induction hypothesis. For the latter case, let $Q = \{\vec{q}_{k+1,1}, \vec{q}_{k+1,2}, \ldots, \vec{q}_{k+1,l}\}$ (for some $l \geq 1$) be the set of all successors of $\vec{q}_k$. Since the transition system $\mathcal{T}$ has finite range, Q is finite. Since there is no $(k+1)$-path starting from $\vec{q}$ satisfying the condition (4.3), there are atomic propositions ap_i such that

$$\vec{p}_{k+1} \models ap_i,\ \vec{q}_{k+1,i} \not\models ap_i,\ \forall 1 \leq i \leq l. \tag{4.4}$$

On the other hand, by the induction hypothesis, there are $\phi_j \in \mathrm{CTL}_{k-j}$ with $\vec{p}_j \models \phi_j$ such that for every $\vec{s}_j \in \Sigma$

$$\vec{s}_j \models \phi_j \iff \vec{s}_j \cong_{k-j} \vec{p}_j,\ \forall 0 \leq j \leq k. \tag{4.5}$$

Next we define a formula

$$\varphi := \phi_0 \wedge \mathbf{E}_{k+1}\mathbf{X}(\phi_1 \wedge (\mathbf{EX}(\phi_2 \wedge \cdots \wedge \mathbf{EX}(\phi_k \wedge \mathbf{EX}(ap_1 \wedge \cdots \wedge ap_l)\cdots)$$

where we use the notation **EX** to indicate there is a successor in some k-path without specifying the number k explicitly for simplicity. We see that $\varphi \in \mathrm{CTL}_{k+1}$. By the conditions (4.5) and (4.4), we conclude that $\vec{p} \models \varphi$, but $\vec{q} \not\models \varphi$. This proves that (4.2) is true for $k+1$. □

Remark 4.3. We have k-bisimilar transition systems preserve the same CTL_k formulas.

Recall that we have defined a Galois connection (α, γ) of the abstract domain $\mathcal{A}$ into the concrete domain $\mathcal{C}$ in terms of the abstraction map $\alpha : \mathcal{C} \to \mathcal{A}$ and concretization map $\gamma : \mathcal{A} \to \mathcal{C}$. In the next theorem we investigate the Galois connection with which CTL_k formulas are strongly preserved between the concrete system and the abstract system. We use the abstract transition system $(\mathcal{A}, \mathcal{T}^{\#})$ to check for CTL_k formulas. Since the clusters in the abstract domain form a basis of the abstract transition system, we formulate the temporal property preservation (3.2) by the requirement (cf. [8]):

$$\forall \vec{p} \in \mathcal{C} \left((\mathcal{C} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathcal{A}) \Longrightarrow \forall \varphi_k \in \mathrm{CTL}_k((\mathcal{C}, \vec{p}) \models \varphi_k \Longleftrightarrow (\mathcal{A}, \alpha(\vec{p})) \models \varphi_k) \right). \quad (4.6)$$

Since we need to deal with property verification on abstract domain, we shall first define abstract atomic propositions. We would like to have as many atomic propositions as possible to be true in each abstract state (cluster). Thus, we define a valuation of atomic propositions on abstract states to be the set of atomic propositions ap such that

$$\alpha(\vec{p}) \models ap \Longleftrightarrow \vec{p} \models ap, \ \forall \vec{p} \in \mathcal{C}. \quad (4.7)$$

The following theorem relates the concrete transition system and the abstract transition system with respect to verification of CTL_k properties.

Theorem 4.4. *For a concrete transition system $(\mathcal{C}, \mathcal{T})$ where $\mathcal{C} = (\Sigma, \leq)$, $\vec{p} \in \Sigma$, a corresponding abstract transition system $(\mathcal{A}, \mathcal{T}^{\#})$ where $\mathcal{A} = (2^{\Sigma}, \subseteq)$, and a CTL_k formula φ_k, if $d(\vec{p}, \gamma(\alpha(\vec{p}))) \leq 2^{-k}$ then $(\mathcal{C}, \mathcal{T}), \vec{p} \models \varphi_k$ if and only if $(\mathcal{A}, \mathcal{T}^{\#}), \alpha(\vec{p}) \models \varphi_k$.*

Proof. We prove for any $\vec{p} \in \mathcal{C}$,

$$(\mathcal{C}, \vec{p}) \models \varphi_k \Longleftrightarrow (\mathcal{A}, \alpha(\vec{p})) \models \varphi_k$$

by the induction on the structure of CTL_k formula (cf. (2.1)).

For the base case let ap be an atomic proposition. By the definition (4.7), for any atomic proposition,

$$\vec{p} \models ap \Longleftrightarrow ap \in L(\vec{p}) \Longleftrightarrow ap \in L(\alpha(\vec{p})) \Longleftrightarrow \alpha(\vec{p}) \models ap.$$

The next are the inductive steps.

(1) Consider $\neg\varphi_k$. By induction hypothesis, we have

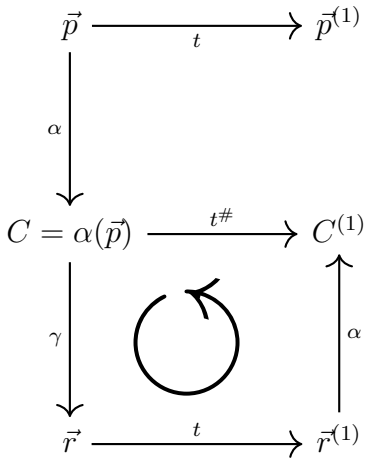
$$\vec{p} \models \neg\varphi_k \Longleftrightarrow \vec{p} \not\models \varphi_k \Longleftrightarrow \alpha(\vec{p}) \not\models \varphi_k \Longleftrightarrow \alpha(\vec{p}) \models \neg\varphi_k.$$

(2) Consider $\varphi_k \vee \phi_k$. By induction hypothesis we have

$$\begin{aligned} \vec{p} \models \varphi_k \vee \phi_k &\iff (\vec{p} \models \varphi_k) \vee (\vec{p} \models \phi_k) \\ &\iff (\alpha(\vec{p}) \models \varphi_k) \vee (\alpha(\vec{p} \models \phi_k) \\ &\iff \alpha(\vec{p}) \models \varphi_k \vee \phi_k. \end{aligned}$$

(3) Consider $\mathbf{E}_k\mathbf{X}(\varphi_{k-1})$. We first show

$$\vec{p} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1}) \Longrightarrow \alpha(\vec{p}) \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1}),\ \forall \vec{p} \in \mathcal{C}.$$



Since $\vec{p} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$, there are a state $\vec{p}^{(1)}$ and a transition t such that $\vec{p} \xrightarrow{t} \vec{p}^{(1)}$ and $\vec{p}^{(1)} \models \varphi_{k-1}$. Let $\vec{r} = \gamma(\alpha(\vec{p}))$. If $d(\vec{p}, \vec{r}) \leq 1/2^k$, we have $\vec{p} \cong_s \vec{r}$ for some $s \geq k$. Then there exists a state $\vec{r}^{(1)}$ such that $\vec{r} \xrightarrow{t} \vec{r}^{(1)}$ and $\vec{p}^{(1)} \cong_{s-1} \vec{r}^{(1)}$. Let $C = \alpha(\vec{p})$ and $C^{(1)} = \alpha(\vec{r}^{(1)})$. Define an abstract transition $t^{\#} := \alpha \circ t \circ \gamma$. Then $C \xrightarrow{t^{\#}} C^{(1)}$. Since $\vec{p}^{(1)} \cong_{s-1} \vec{r}^{(1)}$ and $\vec{p}^{(1)} \models \varphi_{k-1}$, by Theorem 4.1, $\vec{r}^{(1)} \models \varphi_{k-1}$. By induction hypothesis, $\alpha(\vec{r}^{(1)}) \models \varphi_{k-1}$, i.e. $C^{(1)} \models \varphi_{k-1}$. This implies that $C \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$.

Next we show

$$\alpha(\vec{p}) \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1}) \Longrightarrow \vec{p} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1}),\ \forall \vec{p} \in \mathcal{C}.$$

Let $C = \alpha(\vec{p})$. Since $C \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$, there exists an abstract transition $t^{\#}$ and some cluster $C^{(1)}$ such that $C \xrightarrow{t^{\#}} C^{(1)}$ and $C^{(1)} \models \varphi_{k-1}$. By the definition of an abstract transition, there must exist a transition t such that $t^{\#} = \alpha \circ t \circ \gamma$. Let $\vec{r} = \gamma(C)$. Then there exists a state $\vec{r}^{(1)}$ such that $\vec{r} \xrightarrow{t} \vec{r}^{(1)}$ and $\alpha(\vec{r}^{(1)}) = C^{(1)}$. Since $C^{(1)} \models \varphi_{k-1}$, by induction hypothesis, $\vec{r}^{(1)} \models \varphi_{k-1}$. If $d(\vec{p}, \gamma(\alpha(\vec{p}))) \leq 1/2^k$, we have $\vec{p} \cong_s \vec{r}$ for some $s \geq k$. Then there must exist a state $\vec{p}^{(1)}$ such that $\vec{p} \xrightarrow{t} \vec{p}^{(1)}$ and $\vec{p}^{(1)} \cong_{s-1} \vec{r}^{(1)}$. By Theorem 4.1, $\vec{p}^{(1)} \models \varphi_{k-1}$, and so $\vec{p} \models \mathbf{E}_k\mathbf{X}(\varphi_{k-1})$.

(4) Consider $\mathbf{E}_k\mathbf{G}(ap)$. We first show

$$\vec{p} \models \mathbf{E}_k\mathbf{G}(ap) \Longrightarrow \alpha(\vec{p}) \models \mathbf{E}_k\mathbf{G}(ap),\ \forall \vec{p} \in \mathcal{C}.$$

Since $\vec{p} \models \mathbf{E}_k\mathbf{G}(ap)$, there is a sequence of transitions $\vec{p} \xrightarrow{t_1} \vec{p}^{(1)} \xrightarrow{t_2} \vec{p}^{(2)} \cdots \xrightarrow{t_k} \vec{p}^{(k)}$ such that $\vec{p}^{(i)} \models ap$ for all $0 \leq i \leq k$, where $\vec{p}^{(0)} := \vec{p}$. Let $\vec{r} = \gamma(\alpha(\vec{p}))$. If $d(\vec{p}, \vec{r}) \leq 1/2^k$, we have $\vec{p} \cong_s \vec{r}$ for some $s \geq k$. Set $\vec{r}^{(0)} = \vec{r}$. Then there exist states $\vec{r}^{(i)}$ for $1 \leq i \leq k$ such that $\vec{r}^{(i-1)} \xrightarrow{t_i} \vec{r}^{(i)}$ and $\vec{p}^{(i)} \cong_{s-i} \vec{r}^{(i)}$. Let $C := C^{(0)} = \alpha(\vec{p})$ and $C^{(i)} = \alpha(\vec{r}^{(i)})$ for all $1 \leq i \leq k$. Define abstract transitions $t_i^{\#} := \alpha \circ t_i \circ \gamma$ for all $1 \leq i \leq k$. Then $C^{(i-1)} \xrightarrow{t_i^{\#}} C^{(i)}$. Since $\vec{p}^{(i)} \cong_{s-i} \vec{r}^{(i)}$ and $\vec{p}^{(i)} \models ap$, by Theorem 4.1, $\vec{r}^{(i)} \models ap$. By induction hypothesis, $\alpha(\vec{r}^{(i)}) \models ap$, i.e. $C^{(i)} \models ap$ for all $0 \leq i \leq k$. This implies that $C \models \mathbf{E}_k\mathbf{G}(ap)$.

Next we show

$$\alpha(\vec{p}) \models \mathbf{E}_k\mathbf{G}(ap) \Longrightarrow \vec{p} \models \mathbf{E}_k\mathbf{G}(ap),\ \forall \vec{p} \in \mathcal{C}.$$

Let $C = \alpha(\vec{p})$ and $C^{(0)} := C$. Since $C \models \mathbf{E}_k\mathbf{G}(ap)$, there exists a sequence of abstract transitions $C \xrightarrow{t_1^{\#}} C^{(1)} \xrightarrow{t_2^{\#}} C^{(2)} \cdots \xrightarrow{t_k^{\#}} C^{(k)}$ such that $C^{(i)} \models ap$ for all $0 \leq i \leq k$. By the definition of an abstract transition, there must exist transitions t_i such that $t_i^{\#} = \alpha \circ t_i \circ \gamma$ for all $1 \leq i \leq k$. Let $\vec{r} = \gamma(C)$. Then there exists states $\vec{r}^{(i)}$ such that $\vec{r}^{(i-1)} \xrightarrow{t_i} \vec{r}^{(i)}$ and $\alpha(\vec{r}^{(i)}) = C^{(i)}$ for all $1 \leq i \leq k$. Since $C^{(i)} \models ap$, by induction hypothesis, $\vec{r}^{(i)} \models ap$. If $d(\vec{p}, \gamma(\alpha(\vec{p}))) \leq 1/2^k$, we have $\vec{p} \cong_s \vec{r}$ for some $s \geq k$. Set $\vec{p}^{(0)} = \vec{p}$. Then there must exist states $\vec{p}^{(i)}$ such that $\vec{p}^{(i-1)} \xrightarrow{t_i} \vec{p}^{(i)}$ and $\vec{p}^{(i)} \cong_{s-i} \vec{r}^{(i)}$ for all $1 \leq i \leq k$. By Theorem 4.1, $\vec{p}^{(i)} \models ap$ for all $0 \leq i \leq k$, and so $\vec{p} \models \mathbf{E}_k\mathbf{G}(ap)$.

(5) Consider $\mathbf{E}_k(ap_1 \mathbf{U} ap_2)$. We first show

$$\vec{p} \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2) \Longrightarrow \alpha(\vec{p}) \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2),\ \forall \vec{p} \in \mathcal{C}.$$

Suppose $\vec{p} \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2)$. Let $\vec{p}^{(0)} := \vec{p}$. Then there exists a path $\pi_1 = \vec{p}\,\vec{p}^{(1)} \cdots \vec{p}^{(k)}$ such that there is a $1 \leq j \leq k$ for which

- $\vec{p}^{(i)} \models ap_1$ for all $0 \leq i < j$; and

- $\vec{p}^{(j)} \models ap_2$.

Let $\vec{r} = \gamma(\alpha(\vec{p}))$. If $d(\vec{p}, \vec{r}) \leq 1/2^k$, we have $\vec{p} \cong_s \vec{r}$ for some $s \geq k$. Set $\vec{r}^{(0)} = \vec{r}$. Then there exist states $\vec{r}^{(i)}$ for $1 \leq i \leq k$ such that $\vec{r}^{(i-1)} \xrightarrow{t_i} \vec{r}^{(i)}$ and $\vec{p}^{(i)} \cong_{s-i} \vec{r}^{(i)}$. Let $C := C^{(0)} = \alpha(\vec{p})$ and $C^{(i)} = \alpha(\vec{r}^{(i)})$ for all $1 \leq i \leq k$. Define abstract transitions $t_i^{\#} := \alpha \circ t_i \circ \gamma$. Then $C^{(i-1)} \xrightarrow{t_i^{\#}} C^{(i)}$. Since $\vec{p}^{(i)} \cong_{s-i} \vec{r}^{(i)}$ for all $1 \leq i \leq k$, and since $\vec{p}^{(i)} \models ap_1$ for all $0 \leq i < j$, and $\vec{p}^{(j)} \models ap_2$, by Theorem 4.1, $\vec{r}^{(i)} \models ap_1$ for all $0 \leq i < j$, and $\vec{r}^{(j)} \models ap_2$. By induction hypothesis, $\alpha(\vec{r}^{(i)}) \models ap_1$ for all $0 \leq i \leq j$, and $\alpha(\vec{r}^{(j)}) \models ap_2$. This implies that $C \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2)$.

Next we show

$$\alpha(\vec{p}) \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2) \Longrightarrow \vec{p} \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2),\ \forall \vec{p} \in \mathcal{C}.$$

Let $C = \alpha(\vec{p})$. Since $C \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2)$, there exists a sequence of abstract transitions $C \xrightarrow{t_1^{\#}} C^{(1)} \xrightarrow{t_2^{\#}} C^{(2)} \cdots \xrightarrow{t_k^{\#}} C^{(k)}$ such that there is a $1 \leq j \leq k$ for which $C^{(i)} \models ap_1$ for all $0 \leq i \leq j$ and $C^{(j)} \models ap_2$. By the definition of an abstract transition, there must exist transitions t_i such that $t_i^{\#} = \alpha \circ t_i \circ \gamma$ for all $1 \leq i \leq k$. Let $\vec{r} = \gamma(C)$. Then there exists states $\vec{r}^{(i)}$ such that $\vec{r}^{(i-1)} \xrightarrow{t_i} \vec{r}^{(i)}$ and $\alpha(\vec{r}^{(i)}) = C^{(i)}$ for all $1 \leq i \leq k$. Since $C^{(i)} \models ap_1$ for all $0 \leq i \leq j$ and $C^{(j)} \models ap_2$, by induction hypothesis, $\vec{r}^{(i)} \models ap_1$ for all $0 \leq i \leq j$ and $\vec{r}^{(j)} \models ap_2$. If $d(\vec{p}, \gamma(\alpha(\vec{p}))) \leq 1/2^k$, we have $\vec{p} \cong_s \vec{r}$ for some $s \geq k$. Set $\vec{p}^{(0)} = \vec{p}$. Then there must exist states $\vec{p}^{(i)}$ such that $\vec{p}^{(i-1)} \xrightarrow{t_i} \vec{p}^{(i)}$ and $\vec{p}^{(i)} \cong_{s-i} \vec{r}^{(i)}$ for all $1 \leq i \leq k$. By Theorem 4.1, $\vec{p}^{(i)} \models ap_1$ for all $0 \leq i \leq j$ and $\vec{p}^{(j)} \models ap_2$, and so $\vec{p} \models \mathbf{E}_k(ap_1 \mathbf{U} ap_2)$. □

Based on Theorem 4.4, to verify $(\mathcal{C}, \mathcal{T}), \vec{p} \models \varphi_k$, where $\mathcal{C} = (\Sigma, \leq)$, $\vec{p} \in \Sigma$, one needs to use the abstraction map α to determine the abstract cluster $\alpha(\vec{p})$ corresponding to $\vec{p}$, and if $d(\vec{p}, \gamma(\alpha(\vec{p}))) \leq 2^{-k}$ then we verify $(\mathcal{A}, \mathcal{T}^{\#}), \alpha(\vec{p}) \models \varphi_k$ on the abstract transition system. However, computing $\alpha(\vec{p})$ requires comparing among $d(\vec{p}, \vec{r})$ for each root $\vec{r} \geq \vec{p}$; this may be computationally expensive.

We will use an approach where each cluster C_i will be approximately represented by a *probability density function* (PDF) that determines the probability that C_i is the abstract cluster corresponding to a set of states $U \subseteq \Sigma$. To verify $(\mathcal{C}, \mathcal{T}), \vec{p} \models \varphi_k$, one needs to check if $(\mathcal{C}, \mathcal{T}), U \models \varphi_k$ where $U \subseteq \Sigma$ is a neighborhood of $\vec{p}$ and $(\mathcal{C}, \mathcal{T}), U \models \varphi_k$ if and only if for all $\vec{u} \in U$, $(\mathcal{C}, \mathcal{T}), \vec{u} \models \varphi_k$. Probability density functions can be estimated using a kernel density estimation (KDE) procedure [32]. We use Gaussian kernels to estimate the desired PDFs in the implementation.

Chapter 5
Algorithm for Program Analysis

In this chapter, we formally describe the process that was briefly introduced in Chapter 1. Our approach for verification is a data-driven approach. We assume a program P with the corresponding state space Σ.

5.1 Sampling for Cluster Roots

To induce the clusters in the abstract domain, we first sample program states as root points of clusters. Specifically, we construct a *net* N from the program state space Σ such that

- if $\vec{p}$ and $\vec{q}$ are in N, then $d(\vec{p}, \vec{q}) \geq \epsilon$ where ϵ is a hyperparameter.

The size of N, $n = |N|$, is finite but unbounded (we will estimate n in Section 5.4). Each state in N will be the root of a cluster. We apply the constant ϵ to ensure *unique* program states in the net for discrete distribution conditions. Each state in the net must be at least a distance of ϵ away from all other states in the net so that root points are "spread out" enough, and the resulting clusters rooted at these root points can cover the concrete domain (see Lemma 5.2).

Lemma 5.1. *Let $E_n[X]$ be the expectation of the number X of trials needed to get n root points. There exists a constant χ, where $0 \leq \chi \leq \frac{1}{\epsilon}$, such that*

$$E_n[X] \approx \frac{n}{(1 - \chi\epsilon)^{n(n-1)/2}}. \tag{5.1}$$

Proof. Recall that any two root points $\vec{r}_i$ and $\vec{r}_j$ in the net N satisfy the condition $d(\vec{r}_i, \vec{r}_j) \geq \epsilon$. Let $\overline{f}$ be the conditional probability density function (CPDF) for the distance $d(\vec{r}_i, \vec{r}_j)$. The function $\overline{f}$ is finite, bounded[1], and continuous on $[0, 1]$.

[1]Notice that the CPDF $\overline{f}$ and another CPDF $\overline{f}_i$ (we use it later) are not the Dirac delta function.

Then we have, using the mean value theorem for integrals,

$$\begin{aligned}\mathrm{pr}(d(\vec{r}_i, \vec{r}_j) \geq \epsilon) &= \int_{\epsilon}^{1} \overline{f}(v)\, dv \\ &= 1 - \int_{0}^{\epsilon} \overline{f}(v)\, dv \\ &= 1 - \epsilon \cdot \overline{f}(\overline{v})\end{aligned}$$

for some $\overline{v} \in (0, \epsilon)$. Set $\chi := \overline{f}(\overline{v})$. Since $0 \leq \epsilon\chi \leq 1$, then $0 \leq \chi \leq \frac{1}{\epsilon}$. Thus the probability to get n root points is

$$p = \mathrm{pr}(|N| = n) = (1 - \chi\epsilon)^{C_n^2} = (1 - \chi\epsilon)^{\frac{n(n-1)}{2}}.$$

Then the expected number of trials needed to get n root points is given by

$$E_n[X] \approx \frac{n}{p} = \frac{n}{(1 - \chi\epsilon)^{\frac{n(n-1)}{2}}}$$

where we apply the law of large numbers. □

5.2 Inducing the Abstract Domain

Assuming a net of the concrete state space, $N \subseteq \Sigma$, we construct a (sampling) abstract domain. For each $\vec{r} \in N$, we construct a cluster rooted at $\vec{r}$. Assume there exists the supremum element $\vec{\top}_{\mathcal{C}}$ in the concrete domain $\mathcal{C}$.

Lemma 5.2 (Finite Coverage of Concrete Domain)**.** *For a program P with the program state space Σ, a finite sample $N \subseteq \Sigma$ will induce a set of clusters*

$$\mathcal{C}_N = B_{\vec{\top}_{\mathcal{C}}} \cup \left(\bigcup_{\vec{r} \in N} B_{\vec{r}} \right)$$

that is a finite cover of the concrete domain $\mathcal{C}$.

It is natural to define a (sampling) abstract domain to be $\mathcal{A} := \mathcal{C}_N$.

Proof. Since N is a finite set, $\bigcup_{\vec{r} \in N} B_{\vec{r}}$ is also finite. We include a cluster induced by the supremum element $\vec{\top}_{\mathcal{C}}$ in the concrete domain $\mathcal{C}$. Since $B_{\vec{\top}_{\mathcal{C}}}$ covers all states in Σ, $\mathcal{C}_N$ is a finite subcover of the concrete domain $\mathcal{C}$. □

Identify Abstract States. The construction of an abstract domain essentially depends on determining abstract states (clusters) that most precisely represent the given (concrete) states. For a state $\vec{p} \in \Sigma$, if $\alpha(\vec{p}) = C_i$, then the root r_i is the closest to $\vec{p}$ among all root points $\vec{r} \geq \vec{p}$. The probability density function approximately representing C_i will provide a measure of the closeness of a set of states U to the root $\vec{r_i}$ of C_i. An abstract cluster C_i will correspond to a set of states $U \subseteq \Sigma$ if the probability of its closeness to the root $\vec{r_i}$ is greater than that of its closeness to all other roots $\vec{r}$ (ties broken arbitrarily). Let $\mathcal{F}$ be the set of all PDFs f_C corresponding to $C \in \mathcal{A}$.

5.3 Sampling for Clusters

To approximate a probability density function that can be used to estimate for each abstract cluster C_i with root $\vec{r_i}$, the likelihood that for a state $\vec{p}$, $\alpha(\vec{p}) = C_i$, we need to draw a finite sample for each cluster that is an approximate representation of state $\vec{p}$ such that $\alpha(\vec{p}) = C_i$ (we call this sample the cluster sample for C_i). Observe that if $\vec{p} \leq \vec{r_i}$ and $d(\vec{p}, \vec{r_i}) < \epsilon/2$ then $\alpha(\vec{p}) = C_i$ (by triangle inequality). For a state $\vec{p} \leq \vec{r_i}$ such that $\epsilon/2 \leq d(\vec{p}, \vec{r_i}) \leq 1$, it is possible that $\alpha(\vec{p}) = C_i$ with likelihood increasing with closeness to $\vec{r_i}$.

We form n cluster samples (for the n clusters) from the program state space Σ such that

- each cluster sample contains m program states (m can be pre-specified; we assume $m \geq 30$ for statistical significance); and
- if $\vec{r}$ is the root point of a cluster and $\vec{x}$ is any state in the cluster sample, then

$$d(\vec{r}, \vec{x}) \leq \beta, \quad \text{and} \quad \vec{x} \leq \vec{r} \tag{5.2}$$

  where β is a hyperparameter determining the accuracy of the sample.

Algorithm 1 Build a cluster sample

1: **procedure** *BuildCluster*$(S, \vec{r}, \beta)$
   Input: the root point $\vec{r}$ and the radius β of the cluster sample
   Output: a cluster sample $S_{\vec{r}}$ rooted at $\vec{r}$
2: Begin
3: $S_{\vec{r}} \leftarrow \{\vec{r}\}$
4: $\Sigma \leftarrow \Sigma \setminus \{\vec{r}\}$
5: **while** $|S_{\vec{r}}| < m$ **do**
6: $\vec{x} \leftarrow$ *choose*(Σ)
7: *close* $\leftarrow$ *false*
8: **if** $d(\vec{x}, \vec{r}) > \beta$ **or** $\vec{x} > \vec{r}$ **then**
9: *close* $\leftarrow$ *true*
10: break
11: **end if**
12: **if** *close* $=$ *false* **then**
13: $S_{\vec{r}} \leftarrow S_{\vec{r}} \cup \{\vec{x}\}$
14: $\Sigma \leftarrow \Sigma \setminus \{\vec{x}\}$
15: **end if**
16: **end while**
17: **end procedure**

This construction is to guarantee any state in a cluster sample (i.e. within β distance from the root point of the cluster) is k-step bisimilar (for some $k \in \overline{\mathbb{N}}$) to the root point, while those states beyond β distance from the root point share less and less bisimilarities, and thus are less and less likely to share the same CTL_k properties with the root point. Choosing an appropriate value for β depends on the coarseness of the desired abstraction.

Algorithm *BuildCluster*$(\Sigma, \vec{r}, \beta)$ (cf. Algorithm 1) is based on these conditions, that acquires a cluster sample of program states. We repeatedly choose valid candidate program states using the *choose* function (cf. line 6 in Algorithm 1).

5.4 Estimate the Number of Clusters

In any clustering technique, it is important and difficult to determine the number of clusters. We introduce Algorithm 2 to determine the number n of clusters constructed from observed data, using a stopping criterion. We fix a level of signif-

icance $\nu \in (0, 1]$ to indicate the fraction of unclustered program states. The basic idea is to make the probability of unclustered program states (i.e., states for which equation (5.2) does not hold true for any cluster) less than or equal to the level of significance ν.

Let us first observe some facts from probability theory. Let $\vec{p} \in \Sigma$ and $\vec{r}_i$ a fixed root point. We want to determine $d(\vec{p}, \vec{r}_i)$. Let $\overline{f}_i$ be the conditional probability density function (CPDF) for the distance $d(\,\cdot\,, \vec{r}_i)$ from a state to the root point $\vec{r}_i$. The function $\overline{f}_i$ is finite, bounded, and continuous on $[0, 1]$. Given a state $\vec{p}$ and a distance value $v \in [0, 1]$, $\overline{f}_i(v \,|\, \vec{p})$ tells us how likely the distance $d(\vec{p}, \vec{r}_i)$ equals v. Our goal is to determine the probability

$$\mathrm{pr}(0 \leq d(\vec{p}, \vec{r}_i) \leq \beta) = \int_0^\beta \overline{f}_i(v \,|\, \vec{p})\, dv.$$

By the mean value theorem for integrals, there exists a $\overline{v} \in (0, \beta)$ such that

$$\int_0^\beta \overline{f}_i(v \,|\, \vec{p})\, dv = \overline{f}_i(\overline{v} \,|\, \vec{p}) \cdot \beta = \chi_1 \cdot \beta \tag{5.3}$$

where we set $\chi_1 := \overline{f}_i(\overline{v} \,|\, \vec{p})$. Since $\overline{f}_i$ is bounded, χ_1 is a finite number. Moreover, $0 \leq \chi_1 \beta \leq 1$ since the equation (5.3) represents a probability. So $0 \leq \chi_1 \leq \frac{1}{\beta}$. The following proposition gives an upper bound for the number n of clusters.

Proposition 5.3. *An upper bound for the expected number n of clusters is given by*

$$E[n] = \log\left(\frac{\nu}{\varpi}\right), \quad \varpi := 2 - (c + \chi_1 \beta) - c\chi_1 \beta$$

where $\chi_1 \in [0, \frac{1}{\beta}]$ is the same constant as in (5.3), and c is the probability of a state being less than or equal to a root point[2]*.*

Proof. The construction of n cluster samples $\{S_{\vec{r}_i}\}_{i=1}^n$ is based on the condition (5.2). In view of (5.3), we see that, for a program state $\vec{x}$, the probability of $\vec{x}$

[2]When a partial order $\leq$ is given on the concrete domain $\mathcal{C}$, we have for any two element $\vec{x}, \vec{y} \in \mathcal{C}$, the probability $\mathrm{pr}(\vec{x} \leq \vec{y})$ is a constant.

being unclustered into a cluster sample $S_{\vec{r}_i}$ is given by

$$\mathrm{pr}(\vec{x} \notin S_{\vec{r}_i}) = (1 - \chi_1\beta) + (1 - c) - c\chi_1\beta = 2 - (c + \chi_1\beta) - c\chi_1\beta := \varpi$$

where c (a constant) is the probability of $\vec{x}$ satisfying the condition $\vec{x} \leq \vec{r}_i$. The probability of program states unclustered into any of the n cluster samples is supposed to be less than or equal to the level ν of significance. So we have

$$\mathrm{pr}(\cap_{i=1}^{n}(\vec{x} \notin S_{\vec{r}_i})) = \varpi^n \leq \nu$$

and thus

$$n \leq \log\left(\frac{\nu}{\varpi}\right). \tag{5.4}$$

□

Remark 5.4. Consider the inequality $\varpi^n \leq \nu$. If $\nu \to 0$, since c, χ_1, and β are fixed, we have $n \to \infty$. This implies that Algorithm 2 will not terminate if $\nu \to 0$. Therefore, we restrict the value of ν on $(0, 1]$.

Proposition 5.5. *The following hold true for Algorithm 2:*

1. *It determines the number n of clusters (this is also the number of root points) and generates n cluster samples.*
2. *Each state in a cluster sample is within β-distance from the root point of the cluster.*
3. *Any two root points of clusters are at least ϵ-distance away from each other.*

Proposition 5.6 (Time Complexity of Algorithm 2)**.** *Without loss of generality we assume the size of each cluster sample is the same. An upper bound on the expected run time of Algorithm 2 is given by*

$$\frac{\Gamma}{(1 - \chi\epsilon)^{\Gamma(\Gamma-1)/2}} + \frac{m\Gamma}{c\chi_1\beta}, \;\; \Gamma := \log\left(\frac{\nu}{\varpi}\right) \tag{5.5}$$

Algorithm 2 Determine the Number of Clusters

1: **procedure** $NOC(\Sigma,\ \beta,\ \epsilon,\ \nu)$
   n: the number of clusters; S_i: cluster samples, $i = 1, \ldots, n$
2: $i \leftarrow 1$, $N \leftarrow \emptyset$
3: $\vec{p}_i \leftarrow choose(\Sigma)$
4: $N \leftarrow N \cup \{\vec{p}_i\}$
5: $S_i \leftarrow BuildCluster(\Sigma, \vec{p}_i, \beta)$
6: **while** $\varpi^i > \nu$ **do**
7: $i \leftarrow i + 1$
8: **repeat**
9: $\vec{p}_i \leftarrow choose(\Sigma \setminus (S_1 \cup \cdots \cup S_{i-1}))$
10: $\min \leftarrow d(\vec{p}_i, \vec{p}_1)$
11: **for all** $j = 2 : (i-1)$ **do**
12: **if** $d(\vec{p}_i, \vec{p}_j) < \min$ **then** $\min \leftarrow d(\vec{p}_i, \vec{p}_j)$
13: **end for**
14: **until** $\min \geq \epsilon$
15: $N \leftarrow N \cup \{\vec{p}_i\}$
16: $S_i \leftarrow BuildCluster(\Sigma, \vec{p}_i, \beta)$
17: **end while**
18: $n \leftarrow i$
19: **return** n, N
20: **end procedure**

where c, χ_1 and ϖ are the same as in Proposition 5.3, and χ is the same as in Lemma 5.1.

Proof. In view of the condition (5.2), we see that the probability of a state being assigned to a cluster sample is $c\chi_1\beta$. So we need $\frac{m}{c\chi_1\beta}$ (the expected number) trials to cluster m states into a cluster sample, and thus need $n \cdot \frac{m}{c\chi_1\beta}$ trials to create n such cluster samples. On the other hand, (5.1) gives the expected number of trails needed to get n root points. Therefore, using the upper bound for n (cf. (5.4)), the expectation of the number of trails for Algorithm 2 to terminate is given by

$$\begin{aligned} E(\text{run\_time}) &= \frac{n}{(1-\chi\epsilon)^{n(n-1)/2}} + \frac{nm}{c\chi_1\beta} \\ &\leq \frac{\Gamma}{(1-\chi\epsilon)^{\Gamma(\Gamma-1)/2}} + \frac{m\Gamma}{c\chi_1\beta}, \ \ \Gamma := \log\left(\frac{\nu}{\varpi}\right). \end{aligned}$$

□

5.5 Abstract Density Map and Abstract Density Transition System

For each cluster C_i, we estimate a PDF f_{C_i} from its cluster sample. This f_{C_i} provides an approximate representation of C_i.

Definition 5.7 (Abstract density map)**.** We define an *abstract density map* $\eta : 2^{\Sigma} \to \mathcal{F}$ that maps $U \subseteq \Sigma$ to the PDF f_{C_i} for $C_i \in \mathcal{A}$ if

$$\int_U f_{C_i}(\vec{x})d\vec{x} \geq \int_U f_{C_j}(\vec{x})d\vec{x}$$

for all j (ties broken arbitrarily).

Based on the abstract density map, we define an abstract density transition system (ADTS) as follows.

Definition 5.8 (ADTS)**.** For each abstract transition $t^{\#} \in \mathcal{T}^{\#}$, we define an *abstract density transition* $\theta^{\#} \subseteq \mathcal{F} \times \mathcal{F}$ such that $(f_{C_i}, f_{C_j}) \in \theta^{\#}$ if and only if $(C_i, C_j) \in t^{\#}$. Let $\Theta^{\#}$ be the set of all abstract density transitions. We define the *abstract density transition system* (ADTS) to be the pair $(\mathcal{F}, \Theta^{\#})$.

To verify $(\mathcal{C}, \mathcal{T}), U \models \varphi_k$, we verify if $(\mathcal{F}, \Theta^{\#}), \eta(U) \models \varphi_k$. The ADTS $(\mathcal{F}, \Theta^{\#})$ and $(\mathcal{C}, \mathcal{T})$ are related by the following theorem.

Theorem 5.9. *For a program P with the state space Σ and $U \subseteq \Sigma$, and a* CTL_k *formula φ_k, if $\eta(U) = f_C \in \mathcal{F}$ and $f_C \models \varphi_k$ in the density transition system, then in the concrete program $P, U \models \varphi_k$ with an error probability $\mathcal{E}$ upper bounded by $1 - (c'\epsilon)/2^{k+1}$, where $c' \in [0, \frac{2^{k+1}}{\epsilon}]$ is a constant.*

Proof. Let $U \subseteq \Sigma$ and $\eta(U) = f_{C_{\vec{r}_i}}$ where $C_i := C_{\vec{r}_i}$ is a cluster rooted at $\vec{r}_i$. Suppose $f_{C_i} \models \varphi_k$. Then in the concrete domain, $U \models \varphi_k$ with an error probability $\mathcal{E}$ given by $\mathrm{pr}(U \not\models \varphi_k)$. We determine an upper bound for $\mathcal{E}$ in the following.

Notice that $U \not\models \varphi_k$ if there exists a $\vec{p} \in U$ such that $\vec{p} \not\models \varphi_k$. Thus,

$$\mathcal{E} = \mathrm{pr}((\vec{p} \in U) \wedge (\vec{p} \not\models \varphi_k)) \leq \mathrm{pr}(\vec{p} \not\models \varphi_k) = 1 - \mathrm{pr}(\vec{p} \models \varphi_k).$$

We have shown, in Theorem 4.4, that the statement

$$\alpha(\vec{p}) = C_i \quad \text{and} \quad d(\vec{p}, \vec{r}_i) \leq \frac{1}{2^k} \tag{5.6}$$

together with the statement

$$(\mathcal{A}, \mathcal{T}^{\#}), C_i \models \varphi_k \tag{5.7}$$

implies the statement

$$(\mathcal{C}, \mathcal{T}), \vec{p} \models \varphi_k. \tag{5.8}$$

By the assumption, the statement (5.7) is true and so

$$\text{statement (5.6)} \wedge \text{statement (5.7)} = \text{statement (5.6)}.$$

Thus we can simplify this implication as

$$\text{statement (5.6)} \Longrightarrow \text{statement (5.8)}.$$

The error probability $\mathcal{E} \leq 1 - \mathrm{pr}((5.8))$, where $\mathrm{pr}((5.8))$ is the probability that the statement (5.8) holds true. By probability theory, we see that $\mathrm{pr}((5.8)) \geq \mathrm{pr}((5.6))$. Notice that if $\vec{p} \leq \vec{r}_i$ and $d(\vec{p}, \vec{r}_i) \leq \epsilon/2$ then $\alpha(\vec{p}) = C_i$ (by triangle inequality). For $\vec{p} \leq \vec{r}_i$ such that $\epsilon/2 < d(\vec{p}, \vec{r}_i) \leq 1$, it is possible that $\alpha(\vec{p}) = C_i$ with likelihood increasing with closeness to $\vec{r}_i$. In view of (5.3), there exists a $v' \in (0, \epsilon/2)$ such that

$$\mathrm{pr}(0 \leq d(\vec{p}, \vec{r}_i) \leq \frac{\epsilon}{2}) = \chi_2 \cdot \frac{\epsilon}{2}, \quad \chi_2 := \overline{f}_i(v' \,|\, \vec{p}).$$

Therefore,

$$\mathrm{pr}(\alpha(\vec{p}) = C_i) \geq c \cdot \frac{\chi_2 \epsilon}{2}$$

where c is the same constant as given in Proposition 5.3. Similarly, there exists a constant χ_3 such that

$$\text{pr}(0 \leq d(\vec{p}, \vec{r}_i) \leq \frac{1}{2^k}) = \frac{\chi_3}{2^k}.$$

Notice that both $\frac{\chi_2 \epsilon}{2}$ and $\frac{\chi_3}{2^k}$ are between zero and one. Hence,

$$\text{pr}((5.6)) = \text{pr}(\alpha(\vec{p}) = C_i) \cdot (d(\vec{p}, \vec{r}_i) \leq \frac{1}{2^k}) \geq \frac{c\chi_2\chi_3\epsilon}{2^{k+1}}.$$

Set $c' = c\chi_2\chi_3$. Then $0 \leq c' \leq \frac{2^{k+1}}{\epsilon}$. It follows that

$$\text{pr}((5.8)) \geq \frac{c'\epsilon}{2^{k+1}} \quad \text{and} \quad \mathcal{E} \leq 1 - \frac{c'\epsilon}{2^{k+1}}.$$

□

Remark 5.10. When we fix k and ϵ, the error probability $\mathcal{E}$ can be made arbitrarily small by repeating the verification process as many times as needed.

5.6 Abstraction Refinement

Given a program and a set of properties, our approach generates an abstraction on which we verify properties. The associated ATS (or ADTS) allows us to answer queries about the program within an error probability $\mathcal{E}$.

Performance Evaluation. To evaluate the performance of the program analyzer based on the data-driven abstraction framework, we check the correctness and accuracy of the abstraction applied to verify properties on a validation set of program states. This includes the following steps:

1. For a state $\vec{x}$, using PDFs to identify a cluster C that approximates the abstract state $\alpha(\vec{x})$;

2. For a given property φ_k, verify $C \models \varphi_k$;

3. Go through the same process for each state in the validation set;

We can compare the verification results obtained based on the ADTS with the ground truth, and can classify them in one of the four categories, i.e., true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN):

ground truth	verification results	decision
$\vec{x} \models \varphi_k$	$C \models \varphi_k$	TP
$\vec{x} \models \varphi_k$	$C \not\models \varphi_k$	FN
$\vec{x} \not\models \varphi_k$	$C \models \varphi_k$	FP
$\vec{x} \not\models \varphi_k$	$C \not\models \varphi_k$	TN

(5.9)

We now present a way to offer a viable performance measure of the program analysis tool at different abstraction levels. We consider a weighted sum of true positive rate (TPR) and false positive rate (FPR), and the number n of clusters:

$$\mathcal{S} := w_1 \log_2(TPR + 1) - w_2 \log_2(FPR + 1) - w_3 \log_2(n) \tag{5.10}$$

where

$$TPR = \frac{TP}{TP + FN} \quad \text{and} \quad FPR = \frac{FP}{FP + TN}$$

are calculated based on (5.9), and w_i's are weighting factors in $[0, 1]$ that determine the relative strength of TPR, FPR, and the number n of clusters, respectively. We can change these parameters to control the relative importance of each term to our observation. An abstraction enables ideal performances when it has high TPR, low FPR, and relatively small number n of clusters (for computational efficiency). We observe the relation between these three terms and the hyperparameter β, and create an evaluation curve by plotting the weighted sum $\mathcal{S}$ against the (modified) hyperparameter $\log_2 \beta$. This curve illustrates the variation in performance of the program analyzer created based on the data-driven abstraction framework as its discrimination hyperparameter is varied.

Refinement. We compute $\mathcal{S}$ for an abstraction generated by our method using (5.10). If the number $\mathcal{S}$ is below a threshold δ, we vary β and the least distance ϵ between cluster root points to generate distinct levels of abstractions, on which we can again answer queries. Based on the computation of the quantitative measure $\mathcal{S}$, we may iteratively repeat this process until $\mathcal{S}$ reaches a stable and optimal value that induces an ideal abstraction for the given program analysis problem.

Chapter 6
Experiments

In this chapter, we summarize our experiments and present the verification results obtained by applying the data-driven abstraction framework on a test suite of C programs selected from the open-source GNU coreutils, diffutils, and grep. This will certify the claims we made in this work.

6.1 Experimental Strategy

In this section, we provide details on the strategy used in our experiments.

Experimental Setup. For our experiments we utilized a desktop computer running Ubuntu Linux 4.4.0-53. The data-driven abstraction framework is implemented in Python and MATLAB.

6.1.1 Generate Sample Data

First we introduce a dynamic analysis approach to generate concrete data.

Programs were selected for instrumentation if they fulfilled several criteria. We preferred programs that did not alter the system in a way that would be difficult to clean up after a test run. Programs that utilized generic text file input were selected, as we can vary the input easily by simply providing a variety of input files, generated from dictionary data. We picked programs that were intended to take command line input and run to completion with no user input during execution. Therefore, we are able to run each instrumented program many times with little difficulty.

For our experiments we require data on variable values throughout the execution of a program. To gather this data we have instrumented several programs

TABLE 6.1. Programs instrumented to examine execution steps and variable values.

	Name	Package	Documented purpose
1	base64	coreutils	Base64 encode or decode.
2	cat	coreutils	Concatenate files to standard output.
3	cmp	diffutils	Compare two input files byte by byte.
4	comm	coreutils	Compare sorted input files line by line.
5	cp	coreutils	Copy a source to a destination.
6	csplit	coreutils	Output piece of input file separated by input pattern.
7	dd	coreutils	Copy, convert, and format a file.
8	diff	diffutils	Compare input files line by line.
9	du	coreutils	Summarize disk usage of input file(s).
10	fmt	coreutils	Reformat file by paragraph.
11	fold	coreutils	Wrap lines in input file.
12	grep	grep	Search for pattern in input source.
13	truncate	coreutils	Shrink or extend the size of input file.
14	wc	coreutils	Print word count for input file.

(Table 6.1), mostly selected from the open-source GNU coreutils, diffutils, and grep.

Within an instrumented program, we record two types of information. The first is the states of certain variables, global and local, in a set of meaningful code locations[1] of the program, for a variety of inputs. The first data set allows us to compare the actual values of the variables at particular locations. The second is the sequential set of statements that were actually executed in a run. The second data set, which we will call the traces, allows us to compare any two code locations across any of our runs and determine bisimilarity (we call the resulting values *bisimilarity data*).

We specifically limit ourselves to boolean and numeric variables, as we can draw clusters around these variables, and meaningfully discuss the distance between one variable value and another. While we have examined measuring strings and byte arrays, we are not utilizing those here (see Section 8.2 for more discussion about extension to other program types).

[1]By a meaningful code location of a program, we mean the code represents a statement that has side-effect.

Generating trace data. The instrumentation allows us to collect data (a trace) from one individual run of a program. This trace will consist of the exact sequence of significant statements executed in the program, and the values stored in the variables at every meaningful location in those programs (we call the resulting values *trace data*). For some programs, we instrumented only the main method, for others we instrumented some supporting methods as well. We would instrument supporting methods if the program was more modular and contained a smaller main method, or if experimental results showed that merely instrumenting the main method did not provide enough information to verify any properties. By executing these instrumented programs with a variety of inputs, we are able to acquire a sample of traces from the population of all traces available. As the set of all traces might be infinite, if we consider the set of all possible input files to be infinite, we cannot examine all possible traces.

We also need to get as much variety as possible in the traces that we execute. There are two basic kinds of variations we put on user input. The first was using generated file input to be used by the program. The second was to vary the argument flags that alter the manner in which the program executes. While file input can be varied automatically, not every flag combination is relevant for every program. For example, consider a program with a flag to trigger verbose output, and a flag that caused the program to execute in quiet mode and produce no output. Combining both of those flags in a single execution would not be meaningful. We created a file that contains a set of reasonable execution commands, each of which uses a different flag combinations, and one or more locations to insert generated file input. We created the flag combinations by reading the help data provided for each program, but for the programs with many flags, not every combination was

used. Then we were able to run the entire file on a set of generated inputs, to create a large set of varying execution traces.

We can see a fragment of instrumented code in Figure 6.1. The `openStateFile` and `finishState` functions open and close the file used to store variable state and trace prefix information. The `nvpair` array stores information for each variable that we instrument. And `appStmt` is used to store a unique identifier for each transitional statement in code, and record the actual variable values after each such statement. By transitional statement, we mean assignment statements that change variables' values, or function calls. Any execution that affects memories is viewed as a transition. For example, the statement `x = 15;` would be considered transitional as it alters the value of the variable `x`. A conditional such as `if(x > 10)` would not be considered a transitional statement as no variable's state is altered. Moreover, a variable declaration is not a transitional statement. In all likelihood any meaningful `if block` would contain transitional statements, so while the statement itself would not be recorded, its impact on control flow would as we instrument the body of the `if block`. After making these modifications to the source file(s) we build new executables that will continue to perform their original purposes, as well as saving trace and variable information to a file every time the program is run.

When we store each state name, and each variable value, it is written to a file as the program is executing. After the program completes execution for a specific input, the files containing the data collected during the trace are moved to a collection directory, which will store the other trace and statement sequence data files created for a specific set of tests.

Determining bisimilarity. In addition to the values of variables, we also constructed a sequence of the statements executed for each trace. The sequence of

```
1  int main (int argc, char **argv)
2  {
3    bool number=false;
4    bool squeeze_blank=false;
5    openStateFile();
6    appStmt("BN");
7    ...
8    nvpair* locals = malloc(sizeof(nvpair)*20);
9    ...
10   locals[1] = (nvpair){"LOCAL:squeeze_blank",
11     "bool", &squeeze_blank, sizeof(bool)};
12   locals[2] = (nvpair){"LOCAL:number",
13     "bool", &number, sizeof(bool)};
14   ...
15   switch(c){
16     case 'b':
17       number = true;
18       appStmt("CH");
19     case 's':
20       squeeze_blank = true;
21       appStmt("CM");
22   ...
23   if (! (number || show_ends || show_nonprinting
24   || show_tabs || squeeze_blank))
25   {
26     insize = MAX (i, outsize);
27     appStmt("EA");
28     ...
29   }
30   ...
31   finishState();
32 }
```

FIGURE 6.1. Code snippets from cat.c in linux coreutils package. This code has been edited for formatting, and includes highlighted lines for added instrumentation code.

statements executed at each point in the trace is saved along with the variable values. These statement sequences are used to construct a bisimilarity measurement for each point/state in the trace. One trace corresponds to one total sequence of statements executed, and each point in the trace has executed some prefix of that total sequence. After running the entire experiment, we can compare those saved

Trace x complete sequence:	Trace y complete sequence:
A,B,C,T,U,X,U,X,Y,Z	N,M,O,T,U,X,U,Z
Trace x state $\vec{p}$ prefix:	Trace y state $\vec{q}$ prefix:
A,B,C,T	N,M,O,T
Trace x state $\vec{p}$ suffix:	Trace y state $\vec{q}$ suffix:
U,X,U,X,Y,Z	U,X,U,Z
	$Bisimilar(\vec{p}, \vec{q}) = 3$

TABLE 6.2. Bisimilarity calculation for two states, $\vec{p}$ and $\vec{q}$, dropped in one location from two traces, x and y.

prefixes, along with the complete sequence, and create a suffix that represents the statements that will be executed moving forward from any point. Bisimilarity between two states is calculated by determining how long an identical sequence exists between the suffixes calculated for those two states, as seen in Table 6.2. The computed bisimilarity values that compare each trace point are then stored in a bisimilarity matrix. When a state is compared with itself, its bisimilarity is considered to be infinite, however because all our experiments were run on terminating programs, all calculated bisimilarity values are initially finite integers. An additional calculation step is performed for a special case where a pair of entire traces match, i.e., for two states the same sequence of statements executed all the way up to the end of code, they are considered to be infinite bisimilar to each other. These bisimilarity values are reset in the matrix to a marker for an infinite value.

Summary of the dynamic analysis process. As a general rule, in order to instrument more programs, one would need to instrument the program to save the states of variables, and the sequence of statements executed for one execution of the program. Then one would develop a set of input arguments to run the program, and properties that vary over those input arguments. After this, by executing the pro-

gram over each argument in the set, one can verify any property at any location, and compare any execution's bisimilarity values to any other execution in our set.

6.1.2 Clustering Data

We next cluster the trace data based on the hyperparameters β and ϵ, which determine the quality of the abstraction, according to the clustering algorithm (Algorithm 2). Kernel density estimation (bandwidth = 0.15) is used to estimate PDFs to approximately represent the abstract clusters. Suppose we are going to verify a property φ_k for a fixed $k \in \overline{\mathbb{N}}$. Then we would set $\beta = 1/2^{\overline{k}}$, $\overline{k} \in \overline{\mathbb{N}}$, and $\epsilon \geq 2\beta$ (to guarantee clusters do not mutually overlap), and tune these hyperparameters (by varying $\overline{k}$ and ϵ) until the quality of the abstraction is satisfactory. While tuning, we would like to achieve a trade-off between the quality of the abstraction and the number of clusters which affect computational efficiency.

6.1.3 Verify Properties

Properties of interest are specified as CTL_k formulas. Atomic propositions are boolean formulas on the values of variables from the instrumented program. At each location in a program's execution at which we record the variable values, we can also calculate the truth value of the atomic propositions based on those variable values.

For example, if we examine the code in Figure 6.1, we can see that the variables `number` and `squeeze_blank` may each take on two possible values, both will have the value false at initialization, and each may take on the value true based on the execution of a switch statement. We can construct a CTL_k property φ_k using this variable, as seen in Figure 6.2. It means there exists some k-path (i.e. within k steps of execution) such that either `number=true` or `squeeze_blank=true` somewhere on that path.

$$\mathbf{E}_k\mathbf{F}(\text{number} = T \ \vee \ \text{squeeze\_blank} = T)$$

FIGURE 6.2. A property based on the value of the `number` and `squeeze_blank` variables somewhere on a k-step execution (from the current state) on the program *cat.c*.

The property in Figure 6.2 uses the CTL_k concept of *Finally*, meaning the property itself is true for those traces based on a location within k steps ahead. Suppose we would like to verify the property in Figure 6.2 for $k = 10$. Therefore, if a trace was to execute the statement on line 17 in Figure 6.1, `number=true` in a state on a path of length 10 from the current state, we could say that the property's value would be true. When we proceed to clustering data from multiple executions, we will see the relationships between multiple traces and the predictability of property values made clear.

By tracking the value of a property through the lifetime of a complete program execution, we can then move forward through that execution, and state what the property will be at later states. We can, for example, say that a given property will be true at the tenth step from the current state, or that a property will be true for everywhere on a 20-step execution from the current state, based on the complete trace information. When looking ahead at a property's value at some point later in the trace, we use the variable k to represent a specific number of execution steps in the future. Because property values are based on the values of variables, we can calculate the values of properties at every point for which we have variable values stored after the program has completed execution.

Remark 6.1. Our properties were chosen to be experimentally interesting, which we define as having significant variation over the experimental trace set. Whether these properties are those that correspond to ones that a programmer might find interesting is more difficult to determine. Often properties do relate to some of the variables that are most directly impacted by input to the program, such as boolean

variables that are set based on flags used at the terminal. An example might be the `-s` flag used in the `cat` program, which changes the way that output is processed. This flag is represented in the `cat` program as a global boolean variable, and we use it as a component in some of the properties that we attempt to verify.

6.1.4 Determine an Ideal Abstraction

Recall that we cluster the trace data (with hyperparameters $\beta = 2^{-\overline{k}}$, $\overline{k} \in \overline{\mathbb{N}}$, and $\epsilon \geq 2\beta$), and then use the resulting ADTS to verify a given property φ_k. This abstraction may be too rough to verify φ_k, for example, there may be an unacceptable number of false positives. We then perform abstraction refinement by tuning the hyperparameters β and ϵ to generate a new abstraction. We may repeat the process until the abstraction can verify the properties accurately or within an acceptable error probability. At that point, the abstraction is then an ideal abstraction with respect to a given program analysis problem.

To generate an ideal abstraction, we would like to determine optimal values for the hyperparameters β and ϵ that determine the quality of the abstraction. This amounts to tuning the values of $\overline{k}$ and $\epsilon \geq 2 \cdot 2^{-\overline{k}}$ until the performance measure $\mathcal{S}$ cannot be improved anymore. This is the point where the values of the hyperparameters β and ϵ reach an optimal value (and thus provide an optimal operating point of the program analyzer). Towards this goal, we involve the following steps (assume $\epsilon = 2\beta$ for simplicity):

- Initialize $\overline{k}$ (usually set it to be zero);
- Keep increasing $\overline{k}$ by a step size;
- For one $\overline{k}$ value that determines one β value (and thus one ϵ value), construct clusters with respect to the β and ϵ values which form an abstraction.

- Evaluate the performance measure $\mathcal{S}$ for each constructed abstraction;
- Observe each resulting $\mathcal{S}$ value. When the $\mathcal{S}$ values converge to a value above a threshold, and tend to be stable (with relatively small amount of deviations due to noise data), we stop increasing $\overline{k}$;
- Choose the maximum from all observed $\mathcal{S}$ values and find the corresponding $\overline{k}$, β, ϵ values which are optimal;
- The abstraction with respect to the optimal β and ϵ values is thus an ideal abstraction, and hence provide an optimal operating point of the program analyzer.

6.2 Experimental Results

We present the results of performance evaluations of the program analyzer based on the data-driven abstraction framework, and empirically analyze how to determine an ideal abstraction with respect to a given program analysis problem.

6.2.1 A Result on the Program *cat*

The following is one of our experimental results taken for the program `cat.c` (for detailed experimental results for this program and that for the rest of the C programs in the test suite we refer the reader to Appendix A). The sample data was drawn at location main_EH in the instrumented `cat.c` program. Suppose we want to verify a property

$$\varphi_k = \mathbf{E}_k\mathbf{F}(\text{show\_ends} = T),\ k = 42 \tag{6.1}$$

where `show_end` is a boolean variable in the program `cat.c`. The property φ_k ($k = 42$) holds true on a state if there exists a path of length 42 from that state such that `show_end=true` at some state in that path.

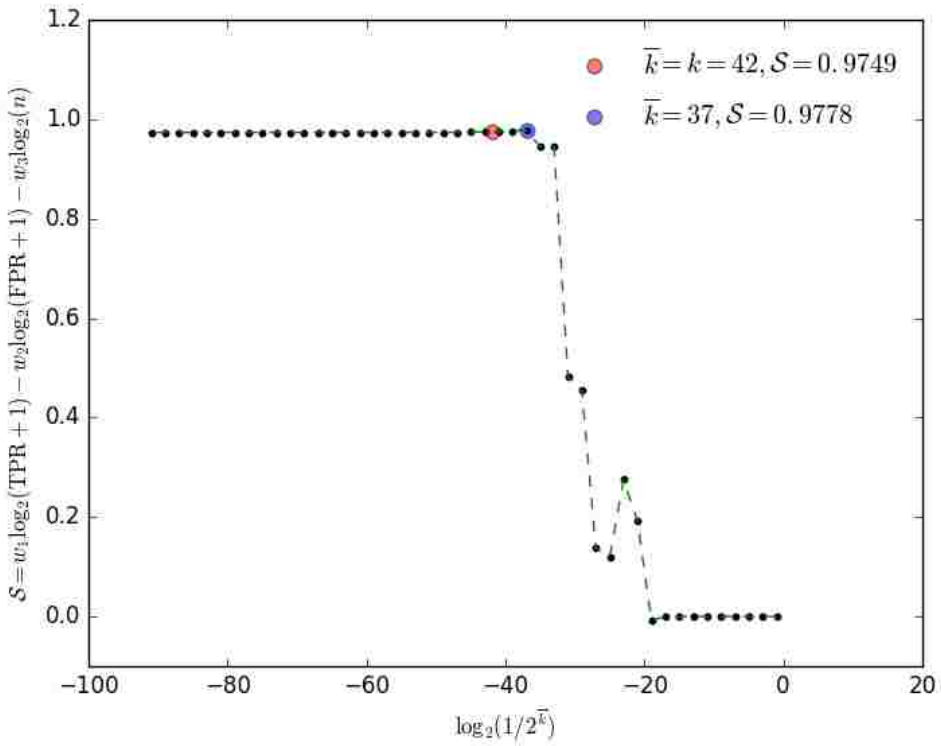


FIGURE 6.3. Performance measure on *cat*: $w_1 = w_2 = 1$, $w_3 = 0.005$

Figure 6.3 provides quantitative evaluations for the performance of the program analyzer for a set of program states (called the validation set; each state corresponding to a neighborhood around itself) for different abstractions used to verify the property (6.1), along with an optimal operating point. It describes a plot of $\mathcal{S}$ values for the different abstractions with respect to β on a log scale. Table 6.3 describes statistics about the performance of the program analyzer for different abstractions. We notice two important things:

1. an optimal $\overline{k}$ value (we set $\epsilon = 2(2^{-\overline{k}})$ for this experiment) for which the value of the performance measure $\mathcal{S}$ reaches its maximum (highlighted by blue bullet); and

2. the value of the performance measure $\mathcal{S}$ for the case $\overline{k} = 42$ (highlighted by red bullet).

TABLE 6.3: Statistical results for program *cat* at location main_EH

$\overline{k}$	n	average m	φ_k $(k = 42)$ TPR	φ_k $(k = 42)$ FPR	$\mathcal{S}$	execution time (in sec)
1	1	1158	0.2	0.2	−0.005	0.3477
19	3	386	0	0	−0.01	0.3572
25	4	288	0.2	0.097	0.118	0.3645
29	6	192	0.4667	0.0606	0.454	0.3751
31	9	127	0.5	0.0629	0.48	0.3962
33	14	82	1.0	0.0257	0.944	0.4327
37	22	52	1.0	0	0.978	0.5018
39	27	42	1.0	0	0.976	0.5516
42	32	36	1.0	0	0.975	0.6225
49	39	29	1.0	0	0.973	0.6837
59	39	27	1.0	0	0.973	0.6861
69	39	26	1.0	0	0.973	0.6890
79	39	25	1.0	0	0.973	0.7041
89	39	24	1.0	0	0.973	0.7223

It can be seen from Figure 6.3 and the statistical result in Table 6.3, that the performance of the abstraction ($\mathcal{S} = 0.9778$) is optimal when $\overline{k} = 37$ (with true positive rate of 1 and false positive rate of 0; see Figure 6.3 for the values of the parameters w_1, w_2, and w_3). For $\overline{k} = 42$ (this is the k value corresponding to the property (6.1)), the value of $\mathcal{S}$ reduces slightly to 0.9749, and reduces to 0.973 for $\overline{k} > 42$ and stabilizes to that value while the execution time keeps increasing. The

value $\overline{k} = 37$ provides the optimal operating point (highlighted by blue bullet) among all observed abstractions.

We also demonstrate empirically that even when we increased $\overline{k}$ to 90 or further, both the ratio of TPR and FPR, and the number of generated clusters did not have a significant improvement anymore. The performance of the constructed abstraction stabilized after the hyperparameter β passed a critical point (in this example, the critical point occurred at $\overline{k} = 37$).

6.2.2 Time Complexity

To estimate the time complexity of the data-driven abstraction approach, we consider the time to construct abstractions plus the time to verify properties of interest. The latter is polynomial in the size and the structure of the given property φ_k. The former has been determined in Proposition 5.6, where we have shown the clustering algorithm terminates and provided an upper bound (5.5) of the expected time for the clustering algorithm to terminate. Thus, the run time to construct abstractions is polynomial in the size of the sample data[2], and is parametric to the pair of hyperparameters β and ϵ.

In Table 6.3, we record a total time for one entire application of our approach on a program, including the time to cluster the sample data, compute PDFs to approximately represent each cluster, verify the given three properties, and evaluate the performance measure $\mathcal{S}$ for the approach.

We will summarize a trend in the execution times for applying our approach on real programs in Section A.3.

[2]By the size of sample data we mean the number of program states drawn from the concrete system, that in turn also depends on the structure of the program P

6.2.3 A Remark

Our goal is to detect bisimilarities between early execution states, to use those bisimilarities to build clusters of common states, and to determine the values of properties at future trace locations based on those similarities.

Because we track multiple properties throughout the lifetime of the entire program, not all properties can be meaningfully set and verified at every location we monitor. Since we work on numeric and boolean data, at the beginning of a program, before such variables are set, there have not yet been any branches in execution, no meaningful distinction can yet be drawn between any two traces, and therefore we have very little information of bisimilarities and little ability to determine property values based on those similarities. Similarly, near the end of the program all remaining suffixes will begin to converge, which also limits the ability to distinguish traces via the bisimilarity measurement. Some programs we worked with proved more problematic than others, however in every program there were at least a few locations, and in some programs there were many, in which we were able to use bisimilarity values to build clusters on which we can verify properties of interest.

Chapter 7
Related Work

The construction of property-preserving abstractions for concrete systems has been one of popular research topics in program analysis.

In counterexample-guided abstraction refinement (CEGAR) [1], the idea is to investigate and analyze counterexamples, and use the feedback to identify new predicates and construct more precise abstractions, while our clustering technique attempts to refine the abstract model by tuning the hyperparameters based on a distance metric. Moreover, our approach provides a quantitative evaluation of the program analyzer, yet the CEGAR framework does not do so.

Mauborgne and Rival [20, 27] constructed partitions of the collection of traces for abstraction and refine them using dynamic partitioning (in trace-based partitioning, elements in a disjunction are related to the history of the concrete computation). Contrasting, in our data-driven abstraction framework, abstraction refinement is performed by tuning hyperparameters. Laviron and Logozzo [18] incrementally refined abstract transfer functions through syntactic and semantic hints. The key idea is to determine the constraints that determine the precision of the analysis and use them to refine the transfer functions in the abstract domain. Our data driven abstraction framework automatically constructs a sequence of abstractions that approach an ideal abstraction suitable for solving a program analysis problem. In [30], Sharma et al. theoretically quantified the precision of an abstraction using VC dimension [5]. They investigated trade-offs between bias and variance to understand how the performance of a program analyzer varies across different abstractions. They used cross validation to determine the trade-off between bias and

variance for Microsoft's program analyzer Yogi and obtained significantly improved results compared to the original version. In contrast, our data-driven abstraction framework quantifies the performance of a program analyzer using a particular abstraction in terms of TPR, FPR, and the number of abstract states. It uses this quantification to determine an optimal operating point for a given program analysis problem. There has been some research on generating [25, 28, 14, 29], pre-conditions and invariants from source code using a data-driven approach.

Yogi [23] uses Synergy [17], an algorithm that combines a procedure for finding bugs with one for proof search in a way that the information gained from the former is fed back to the latter and vice-versa. In addition, Yogi uses Dash [2] that performs abstraction refinement through test generation. In contrast, our data-driven abstraction framework uses samples of states obtained from an instrumented program to estimate PDFs that approximately represent abstract states.

Zhang et al. [33] used a counter-example guided query-driven analysis that performs iterative refinement to identify the cheapest abstraction (minimal set of parameters) or prove that no such abstraction can prove the query. They used an efficient counterexample-driven technique to search through an infinite space of abstractions at a finer level of granularity to obtain one that is optimal. In contrast, our technique is data-driven that refines abstractions by tuning hyperparameters.

In [21], the author suggested a method to identify an optimal abstract interpreter. Convergence in the finite, abstract domain historically [8] requires the use of a widening operator. Widening operators are typically greedy for over-approximation to guarantee convergence and must be designed explicitly for each abstraction thus requiring a bit of creativity. Our clustering technique allows an infinite state space in the abstract domain by using a boolean algebra and its corresponding transition system. While [21] removed the necessity of widening through

a snipping operation (based on edges in dependence graphs) and thus removed recursive structures, the optimal abstract interpretation arises during the trickling phase where the designer chooses a specific abstraction for specific elements (leaves) of a data structure (DAG). This technique was not substantiated with data. Our clustering removes the necessity of the designer playing any role in the development of the optimal abstraction.

Liang et al. [19] used machine learning techniques to determine the coarsest possible abstraction needed to answer a set of points-to analysis queries. They implemented both coarsening and refining approaches, respectively. In contrast, our framework uses a data-driven approach to obtain an optimal operating point (abstraction) for a given program analysis problem. We also focus on verification of temporal properties specified an a restricted and bounded version of CTL. Chen et. al. [6] provided a PAC learning-based framework for creating a model abstracting a program. In contrast, our framework uses a data-driven approach to obtain an ideal abstraction suitable for solving a program analysis problem. In [4], the authors provided an automatic technique that infers a static analyzer, from a dataset containing programs, approximating their behavior. In contrast, our approach uses states sampled from an instrumented program to approximate its semantics. In [3], the authors presented a generative model for programs. We approximate the semantics of a program using a data-drive approach.

B.S. Gulavani et al. [16] automatically and dynamically refined abstract interpretations using a combination of interpolated widening and counter-example guided refinement and using a DAG independence of the domain and the chosen operators. However, we automatically refine abstractions by tuning hyperparameters.

There has been application [31, 9, 22] of abstract interpretation to probabilistic settings. Probabilistic abstract interpretation is based on the assumption that the

distribution of a program is known. In contrast, our approach uses a nonparametric density estimation technique to approximately represent abstract states. We sample a program and learn the distribution of that program from the sample data without assuming any particular type for the distribution. In [11], the authors used the Skorokhod metric for conformance checking for dynamical systems. The pseudo-metric d in this paper is designed to respect bounded bisimilarity between states.

Chapter 8
Conclusions and Future Work

8.1 Conclusions

We present a data-driven abstraction framework and experimental evidence demonstrating the practicality of our approach for program analysis. The data driven abstraction framework is a new approach to automatically create a sequence of abstractions that converge to an ideal abstraction through a particular domain refinement with minimal manual intervention. The resulting abstraction is able to describe concrete system behaviors with sufficient accuracy, and suitable for solving a program analysis problem.

The key is that we construct the abstract domain based on random samples dynamically drawn from the concrete program. We use probability density functions as symbolic representations of abstract states (clusters) which provides a method to determine the abstraction mapping, i.e., identify a corresponding abstract state for a given concrete state. This is a fresh idea to approximately represent an infinite set. Our method enables us to evaluate the accuracy of the abstraction in terms of probability. On the other hand, since it is based on dynamically randomly generated sample data drawn from the concrete system, possible incomplete program executions may affect the guarantee of the accuracy of the abstraction.

We have established a thorough formalization of the data-driven abstraction approach, have created a practically useful program analysis tool based on the formalization, that can verify, within an error probability, if a given program satisfies properties specified in the bounded CTL logic, and have actually made a quantification of this tool and tested its success on a set of standard C programs.

We believe that our framework is robust and flexible enough to handle realistic programs.

8.2 Future Work

We have restricted ourselves to program variables that can meaningfully be placed on an interval. This means our properties are defined in terms of numeric or boolean variables, and our traces and clustering are based on those variables as well. We assume the vast majority of programmers using, for example, the integer type, will consider 42 closer to 43, and farther from 9823. There are exceptions to this, such as when an integer might be used to store flag values concatenated with the bitwise `or` operation. While we initially stored string variables, and other collections of byte data without consistent frameworks, we found they are difficult to place on what should be a numeric execution similarity interval. Considering that we have demonstrated a methodology to verify properties based on variable values, we consider the following question. Can we extend that verification to build a classification system for these more difficult types of variables?

If we restrict ourselves first to string type variables, can we take the clustering algorithm we have developed, and some basic building blocks to automatically develop a string classification function that will generate a numeric value which will predict the effects that a particular string would have on program execution? We might consider a function that looks at the individual bytes in the string variable as distinct numeric values, discard locations irrelevant to the execution path, and reorder the remaining values to create a single numeric value for a string. How often would such an approach produce a numeric value that could be used to predict program execution? Would any generated functions have a general applicability outside of the program they were created on? If so, we could build a system to

help highlight general byte data that might be responsible for abnormal program execution, or classify strings into formatted common types.

What the trace/clustering model has shown is that there are some variables in the program (informally integers and bools in our examples), that are "cluster predictive". Part of the reason for this is that these variables have universal conventions on how to determine meaning and distance. Therefore, when traced values from those variables are placed on a number line interval, we can build meaningful clusters around those values to predict execution.

Another set of variables (informally strings, but also some structs and buffers) are not as useful, because they are comprised larger byte chunks and not easily mapped meaningfully to a small interval. We may use concepts like *edit distance*, but there are limitations of any of those approaches when applied generally. This doesn't mean that these variables don't alter program flow, however.

If we have a traced program (probably a step up in complexity from the current ones), with many inputs that has a mix of predictive and not predictive variables, could we use the predictive variables to assign a meaningful metric to the non-predictive ones?

So if we divided such a program's inputs into separate training and verification sets, could we use the values of nonpredictive variables, linked to the clustered values of the predictive variables at the same points, in the training set, to create some function f such that f would take as input an nonpredictive data chunk, and output a value on an interval that would be predictive? And could we then test the usefulness of that function f on the verification data? The risk here is that there is a real possibility that the answer might be "that wouldn't work since f would have to be too complex".

We are concerned that the "building blocks" to create a function like f might be a challenge. There are some approaches that might work (permutations of digit significance, finding the first null terminator to determine a length), but it might be a challenge. There would also be the question of how general or specific we would have to make f, would there be a function f that would be useful in several different programs (say ones that all had a string that was formatted as an address).

References

[1] Balarin, F., Sangiovanni-Vincentelli, A.L.: An iterative approach to language containment. In: Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings. pp. 29–40 (1993)

[2] Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008. pp. 3–14 (2008)

[3] Bielik, P., Raychev, V., Vechev, M.T.: PHOG: probabilistic model for code. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. pp. 2933–2942 (2016)

[4] Bielik, P., Raychev, V., Vechev, M.T.: Learning a static analyzer from data. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. pp. 233–253 (2017)

[5] Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the vapnik-chervonenkis dimension. J. ACM 36(4), 929–965 (1989)

[6] Chen, Y., Hsieh, C., Lengál, O., Lii, T., Tsai, M., Wang, B., Wang, F.: PAC learning-based verification and model synthesis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. pp. 714–724 (2016)

[7] Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)

[8] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)

[9] Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. pp. 169–193 (2012)

[10] Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order (2. ed.). Cambridge University Press (2002)

[11] Deshmukh, J.V., Majumdar, R., Prabhu, V.S.: Quantifying conformance using the skorokhod metric. Formal Methods in System Design 50(2-3), 168–206 (2017)

[12] Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 995–1072 (1990)

[13] Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. 2(3), 241–266 (1982)

[14] Gehr, T., Dimitrov, D., Vechev, M.T.: Learning commutativity specifications. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 307–323 (2015)

[15] Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. pp. 72–83 (1997)

[16] Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: TACAS. pp. 443–458 (2008)

[17] Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006. pp. 117–127 (2006)

[18] Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: APLAS. pp. 343–358 (2009)

[19] Liang, P., Tripp, O., Naik, M.: Learning minimal abstractions. In: POPL. pp. 31–42 (2011)

[20] Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP. pp. 5–20 (2005)

[21] Might, M.: Abstract interpreters for free. In: SAS. pp. 407–421 (2010)

[22] Monniaux, D.: Abstract interpretation of programs as markov decision processes. In: Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. pp. 237–254 (2003)

[23] Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The yogiproject: Software property checking via static analysis and testing. In: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on

Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. pp. 178–181 (2009)

[24] Ore, O.: Galois connexions. Transactions of the American Mathematical Society 55, 493–513 (1944)

[25] Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 42–56 (2016)

[26] Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. Log. Comput. 17(1), 157–197 (2007)

[27] Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5) (2007)

[28] Sankaranarayanan, S., Chaudhuri, S., Ivancic, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008. pp. 295–306 (2008)

[29] Sharma, R.: Data-driven verification. Ph.D. thesis, Stanford (2016)

[30] Sharma, R., Nori, A.V., Aiken, A.: Bias-variance tradeoffs in program analysis. In: POPL. pp. 127–138 (2014)

[31] Smith, M.J.A.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. Electr. Notes Theor. Comput. Sci. 220(3), 43–59 (2008)

[32] Wasserman, L.: All of Nonparametric Statistics (Springer Texts in Statistics). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)

[33] Zhang, X., Naik, M., Yang, H.: Finding optimum abstractions in parametric dataflow analysis. In: PLDI. pp. 365–376 (2013)

Appendix A
More Experimental Results

In this appendix section, we shall provide and summarize our experimental results obtained by applying the data-driven abstraction approach on a test suite of C programs from GNU coreutils, diffutils, and grep.

A.1 Properties of Programs

First, Table A.1 presents properties chosen to be verified on all observed programs (Table 6.1). These properties (in terms of CTL_k formulas) were checked within k steps. We will provide an explicit k value we use to verify these CTL_k formulas for each observed location in a program.

TABLE A.1: Properties chosen for fourteen observed programs

Program	base64
Property 1	$\mathbf{E}_k\mathbf{F}(\text{ignore\_garbage} = T \wedge \text{decode} = T)$
Property 2	$\mathbf{E}_k\mathbf{X}^k(\text{wrap\_column} = F)$
Property 3	$\mathbf{A}_k\mathbf{G}(\text{decode} = F)$
Program	cat
Property 1	$\mathbf{E}_k\mathbf{F}(\text{show\_ends} = T)$
Property 2	$\mathbf{E}_k\mathbf{X}^k(\text{number} = T \vee \text{squeeze\_blank} = T)$
Property 3	$\mathbf{A}_k\mathbf{G}\neg((\text{show\_nonprinting} = T) \wedge (\text{show\_tabs} = F))$

Program	cmp
Property 1	$\mathbf{E}_k\mathbf{X}^k(\text{opt\_print\_bytes} = T)$
Property 2	$\mathbf{E}_k\mathbf{F}((\text{opt\_print\_bytes} = T) \wedge (\text{ignore\_initial}[2] = F))$
Property 3	$\mathbf{A}_k\mathbf{G}((\text{opt\_print\_bytes} = T) \vee (\text{ignore\_initial}[2] = F))$
Program	comm
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{only\_file\_2} = T) \vee (\text{seen\_unpairable} = T))$
Property 2	$\mathbf{E}_k\mathbf{X}^k((\text{both} = T) \wedge (\text{only\_file\_1} = T))$
Property 3	$\mathbf{E}_k\mathbf{X}^k((\text{both} = T) \vee (\text{only\_file\_2} = F))$
Program	cp
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{make\_backups} = T) \vee (\text{remove\_trailing\_slashes} = T))$
Property 2	$\mathbf{E}_k\mathbf{X}^k((\text{no\_target\_directory} = T) \vee (\text{copy\_contents} = T))$
Property 3	$\mathbf{A}_k\mathbf{G}((\text{ok} = F) \vee (\text{copy\_contents} = T))$
Program	csplit
Property 1	$\mathbf{E}_k\mathbf{F}((\text{suppress\_count} = T) \vee (\text{elide\_empty\_files} = T))$
Property 2	$\mathbf{E}_k\mathbf{X}^k((\text{suppress\_matched} = T) \wedge (\text{remove\_files} = T))$
Property 3	$\mathbf{A}_k\mathbf{G}((\text{remove\_files} = T) \vee (\text{elide\_empty\_files} = T))$
Program	dd

Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{w\_partial} = F) \wedge (\text{w\_full} = F))$
Property 2	$\mathbf{A}_k\mathbf{G}(\text{r\_partial} = F)$
Property 3	$\mathbf{A}_k\mathbf{G}((\text{conversions\_mask} = F) \vee (\text{oc} = F))$
Program	diff
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{new\_file} = F) \wedge (\text{report\_identical\_files} = F))$
Property 2	$\mathbf{A}_k\mathbf{G}((\text{new\_file} = F) \wedge (\text{explicit\_context} = T))$
Property 3	$\mathbf{E}_k\mathbf{F}(\text{new\_file} = T)$
Program	du
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{apparent\_size} = T) \vee (\text{print\_grand\_total} = T))$
Property 2	$\mathbf{A}_k\mathbf{G}((\text{hash\_all} = F) \wedge (\text{human\_output\_opts} = F))$
Property 3	$\mathbf{E}_k\mathbf{F}((\text{opt\_nul\_terminate} = T) \vee (\text{opt\_inodes} = T))$
Program	fmt
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{crown} = T) \wedge (\text{tagged} = T))$
Property 2	$\mathbf{E}_k\mathbf{F}((\text{prefix\_full\_length} = F) \wedge (\text{split} = T))$
Property 3	$\mathbf{A}_k\mathbf{G}((\text{uniform} = T) \vee (\text{prefix\_length} = F))$
Program	fold
Property 1	$\mathbf{E}_k\mathbf{X}^k(\text{break\_spaces} = T)$

Property 2	$\mathbf{A}_k\mathbf{G}((\text{count\_bytes} = F) \wedge (\text{break\_spaces} = F))$
Property 3	$\mathbf{E}_k\mathbf{F}(\text{count\_bytes} = T)$
Program	grep
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{skip\_empty\_lines} = T) \vee (\text{match\_words} = T))$
Property 2	$\mathbf{E}_k\mathbf{F}((\text{out\_invert} = T) \wedge (\text{suppress\_errors} = F))$
Property 3	$\mathbf{A}_k\mathbf{G}((\text{out\_byte} = F) \vee (\text{match\_icase} = T))$
Program	truncate
Property 1	$\mathbf{A}_k\mathbf{G}\neg((\text{got\_size} = T) \wedge (\text{block\_mode} = T))$
Property 2	$\mathbf{E}_k\mathbf{F}(\text{no\_create} = T)$
Property 3	$\mathbf{E}_k\mathbf{F}((\text{got\_size} = T) \wedge (\text{errors} = F))$
Program	wc
Property 1	$\mathbf{E}_k\mathbf{X}^k((\text{ok} = T) \vee (\text{print\_words} = F))$
Property 2	$\mathbf{A}_k\mathbf{G}((\text{print\_chars} = F) \wedge (\text{print\_lines} = F))$
Property 3	$\mathbf{E}_k\mathbf{F}((\text{print\_linelength} = T) \wedge (\text{ok} = T))$

A.2 Statistical Results

In the following we present quantitative evaluations of the performance of data-driven abstraction process on fourteen observed programs, as well as relevant statistical results.

1. For each of observed programs, we provide a statistics table including the following terms:

- a chosen observed location (name)
- the number n of clusters
- the average m of sizes of cluster samples
- TPR and FPR for verifying a given property φ_k in Table A.1
- a fixed step number k associated to φ_k
- an optimal $\overline{k}$ value (If there are multiple optimal values for $\overline{k}$, we choose the smallest one)
- an optimal performance measure $\mathcal{S}$ with respect to that optimal $\overline{k}$

We name every meaningful code location in an instrumented program in the alphabetical order, for example, `main_A`, `main_AA`, `main_AB`, `main_AC`, and so on.

2. Based on the data-driven abstraction framework, we provide a program analyzer that can be used to verify the chosen properties. For each of observed programs, we present quantitative evaluations for the performance of the program analyzer on a validation data set for different levels of abstractions, by graphical plots (plotting the weighted sum $\mathcal{S}$ for different abstractions versus the (modified) hyperparameter $\log_2 \beta$). When plotting such terms, we took the parameter values

$$w_1 = w_2 = 1, w_3 = 0.005$$

for every observed case.

A.2.1 The Program *base64*

TABLE A.2: Statistical results for program *base64*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AM	50	56	0.53	0.08	14	33	0.4797
main_AO	3	238	1.0	0	15	11	0.9920
main_AW	3	356	1.0	0	12	8	0.9920
location	n	average m	φ_k: property 2 TPR	φ_k: property 2 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_BB	11	97	0.78	0.10	10	45	0.6768
location	n	average m	φ_k: property 3 TPR	φ_k: property 3 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AU	2	535	1.0	0	13	5	0.995

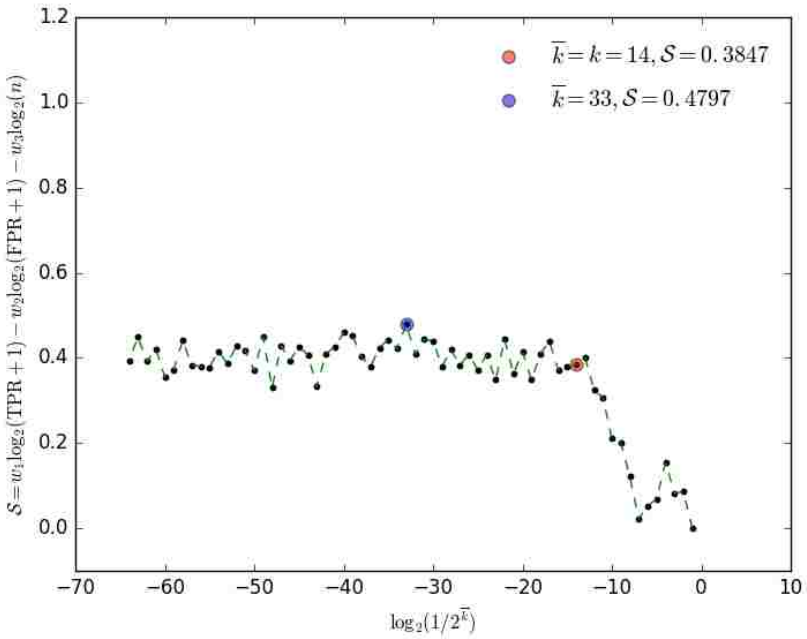


FIGURE A.1. Performance measure on *base64*, location main_AM, property 1

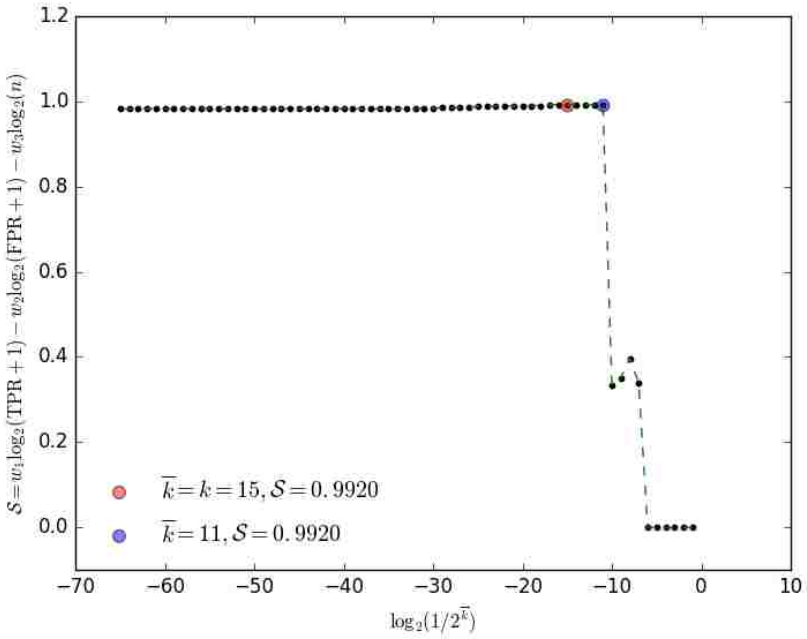


FIGURE A.2. Performance measure on *base64*, location main_AO, property 1

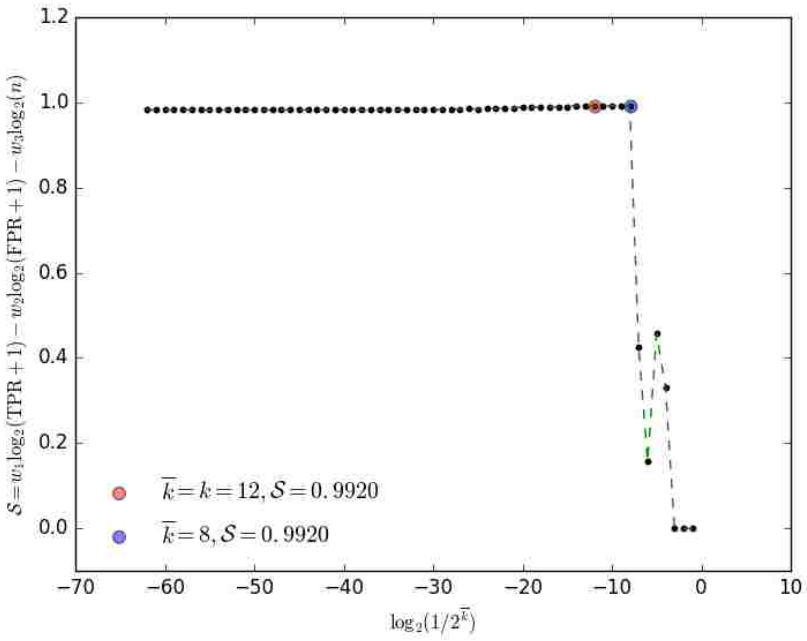


FIGURE A.3. Performance measure on *base64*, location main_AW, property 1

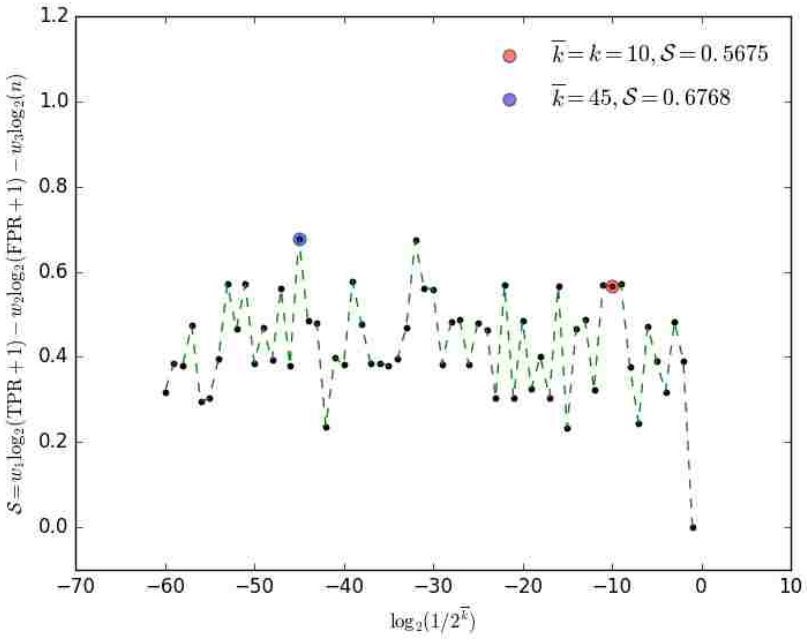


FIGURE A.4. Performance measure on *base64*, location main_BB, property 2

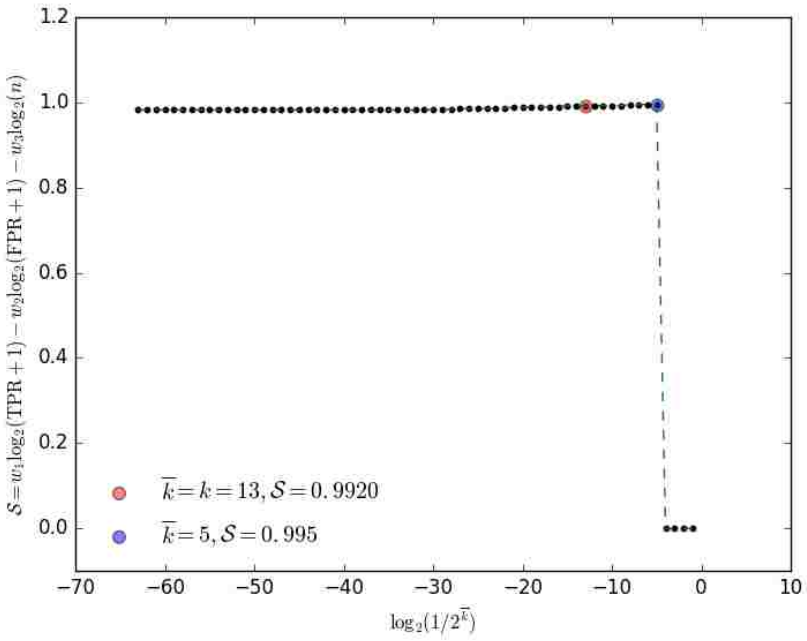


FIGURE A.5. Performance measure on *base64*, location main_AU, property 3

A.2.2 The Program *cat*

TABLE A.3: Statistical results for program *cat*

location	n	average m	φ_k: property 1 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_A	22	52	1.0	0	41	35	0.9778
main_AD	38	30	1.0	0	30	34	0.9737
main_AE	38	30	1.0	0	29	33	0.9737
main_AN	45	51	0.60	0	14	40	0.648
main_AP	23	38	1.0	0	27	29	0.9773
main_AQ	23	38	1.0	0	26	28	0.9774
main_AS	22	296	1.0	0	19	36	0.9778
main_B	14	82	1.0	0	40	31	0.981
main_BG	21	309	1.0	0	18	53	0.9779
main_CH	18	17	0.52	0	59	41	0.6465
main_D	39	29	1.0	0	38	31	0.9791
main_DG	21	67	1.0	0	14	44	0.9781
main_DT	20	79	1.0	0	9	38	0.9782
main_E	39	27	1.0	0	37	31	0.9776
main_EH	22	52	1.0	0	42	37	0.9778
location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_C	3	386	1.0	0	39	15	0.992
main_CF	81	16	0.86	0.10	57	79	0.7209
main_DI	44	44	0.998	0.02	5	53	0.9382
main_DW	7	229	1.0	0	8	21	0.9859

main_O	32	72	1.0	0	11	34	0.975
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_CF	81	16	0.93	0.18	57	89	0.6792
main_CM	3	89	1.0	0	60	3	0.992

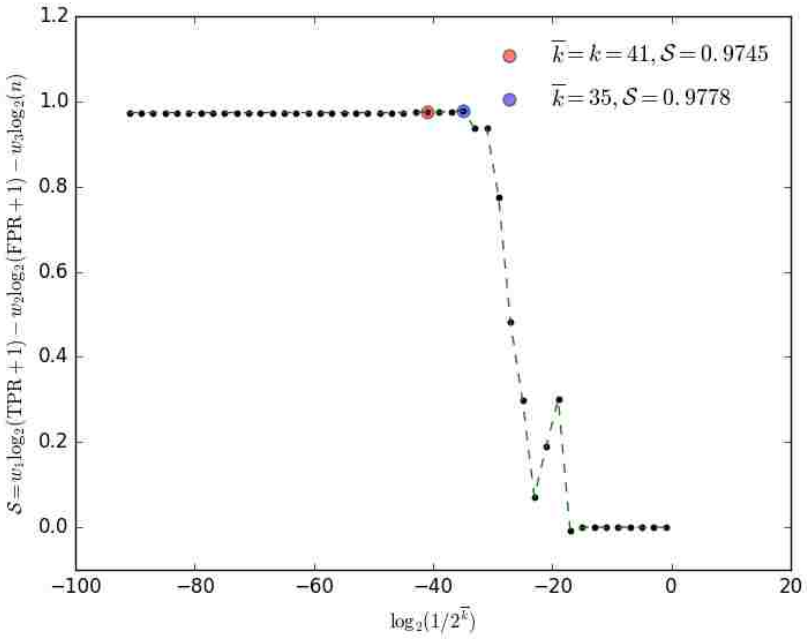


FIGURE A.6. Performance measure on *cat*, location main_A, property 1

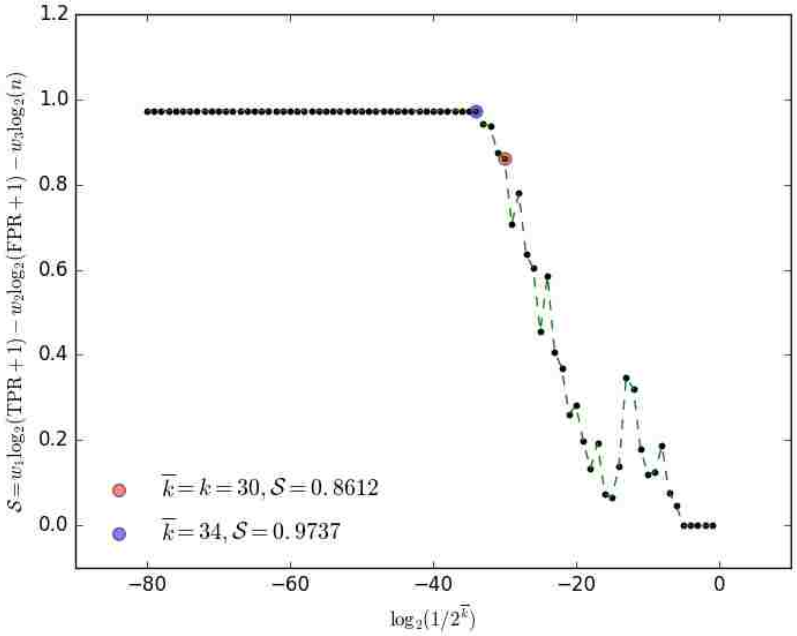


FIGURE A.7. Performance measure on *cat*, location main_AD, property 1

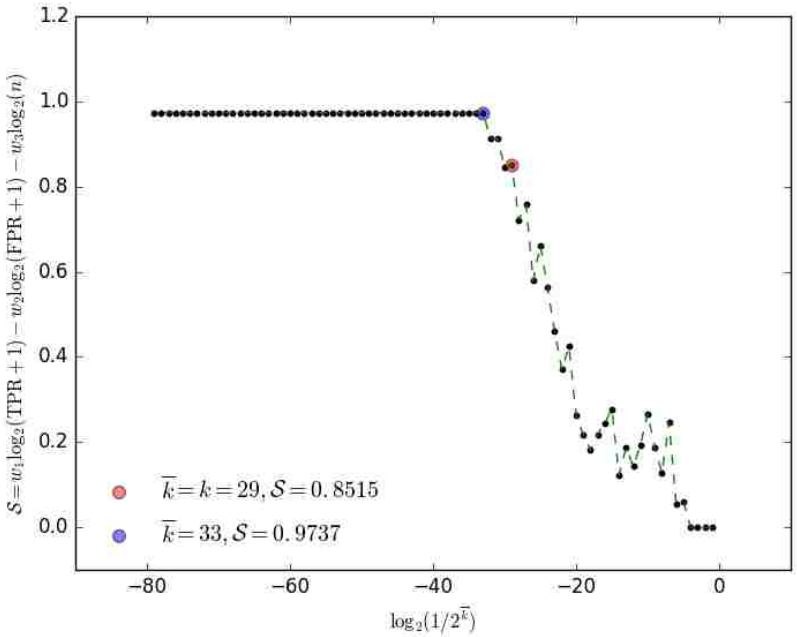


FIGURE A.8. Performance measure on *cat*, location main_AE, property 1

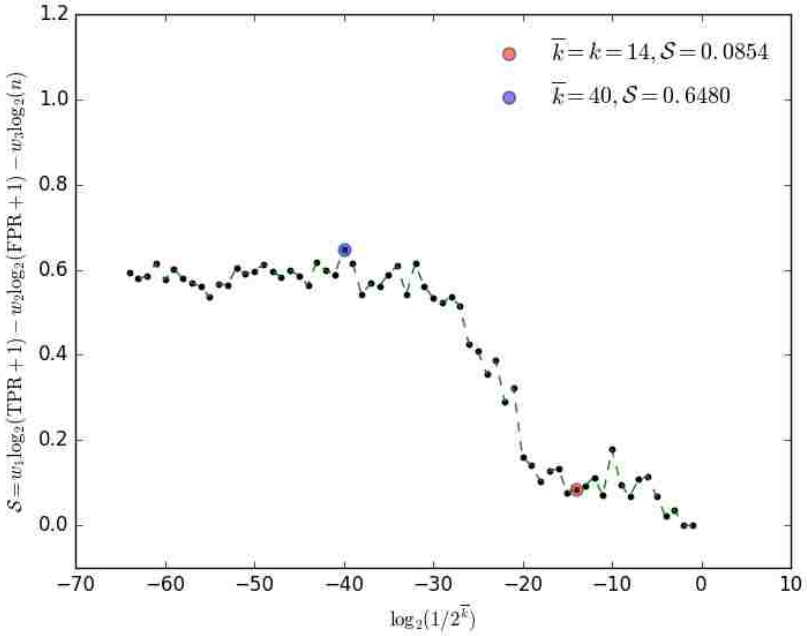


FIGURE A.9. Performance measure on *cat*, location main_AN, property 1

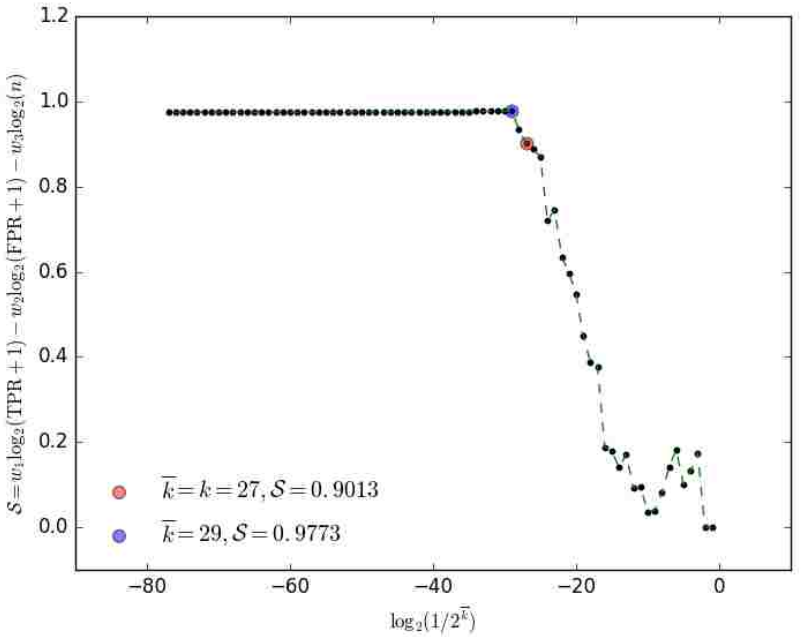


FIGURE A.10. Performance measure on *cat*, location main_AP, property 1

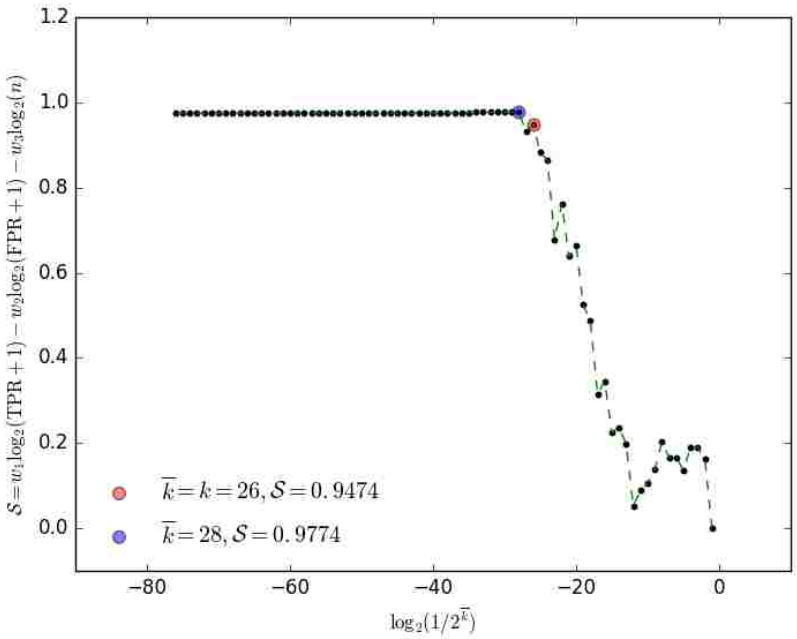


FIGURE A.11. Performance measure on *cat*, location main_AQ, property 1

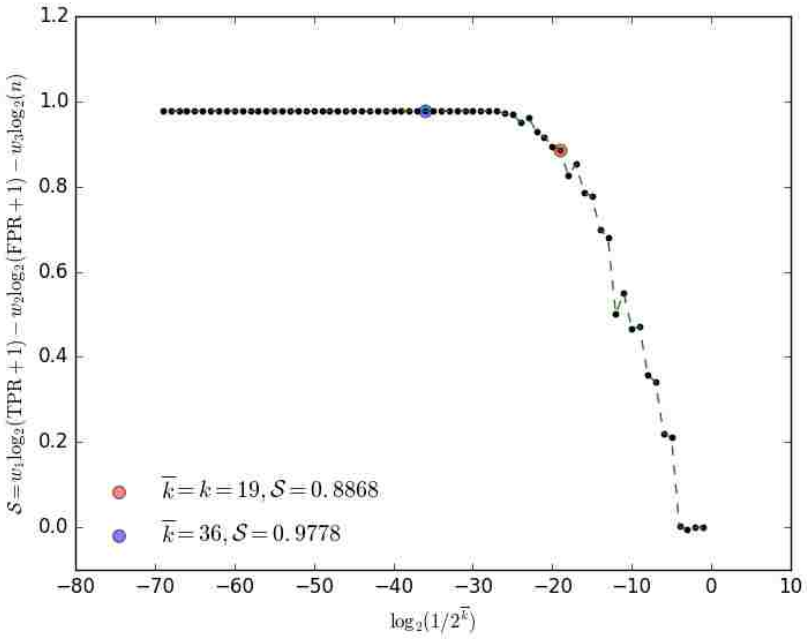


FIGURE A.12. Performance measure on *cat*, location main_AS, property 1

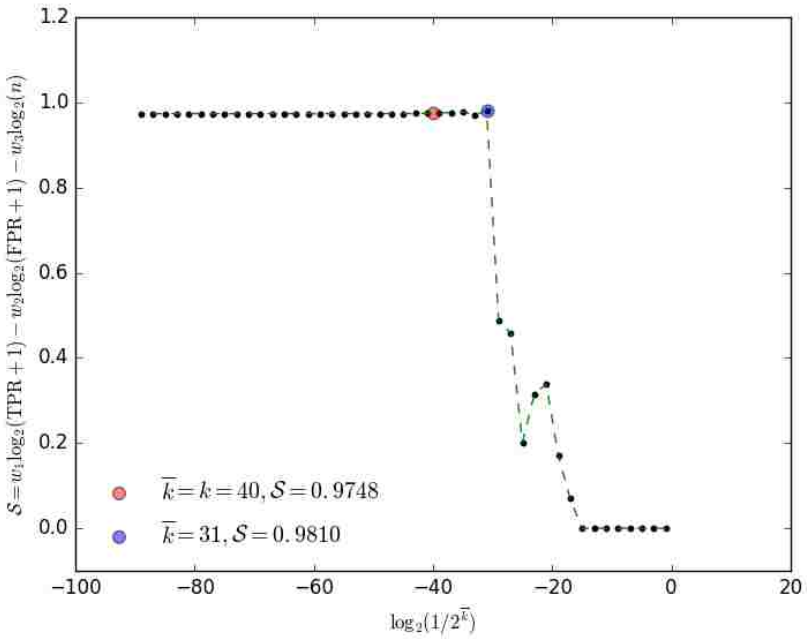


FIGURE A.13. Performance measure on *cat*, location main_B, property 1

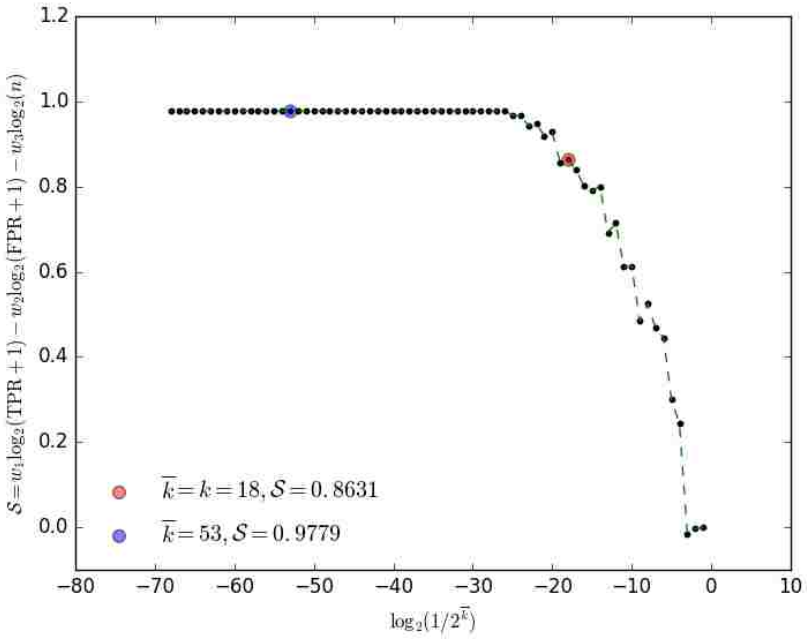


FIGURE A.14. Performance measure on *cat*, location main_BG, property 1

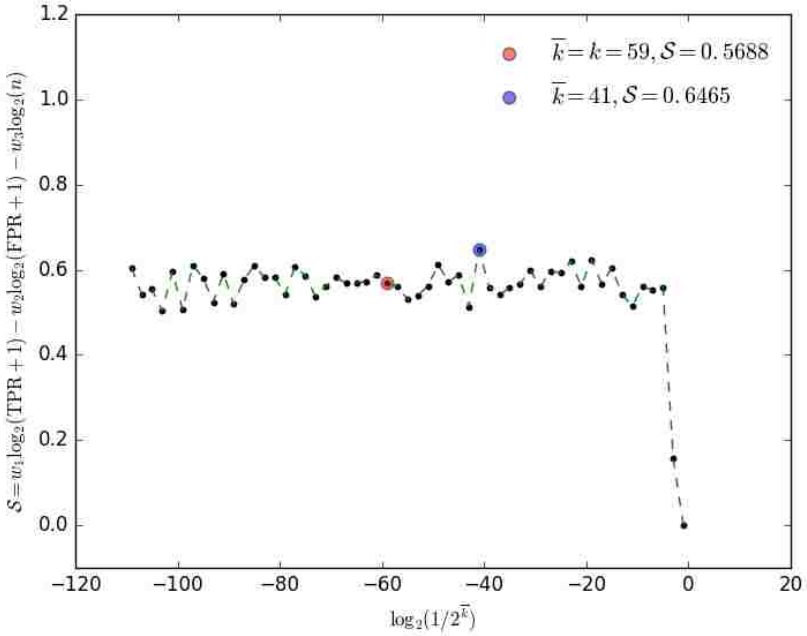


FIGURE A.15. Performance measure on *cat*, location main_CH, property 1

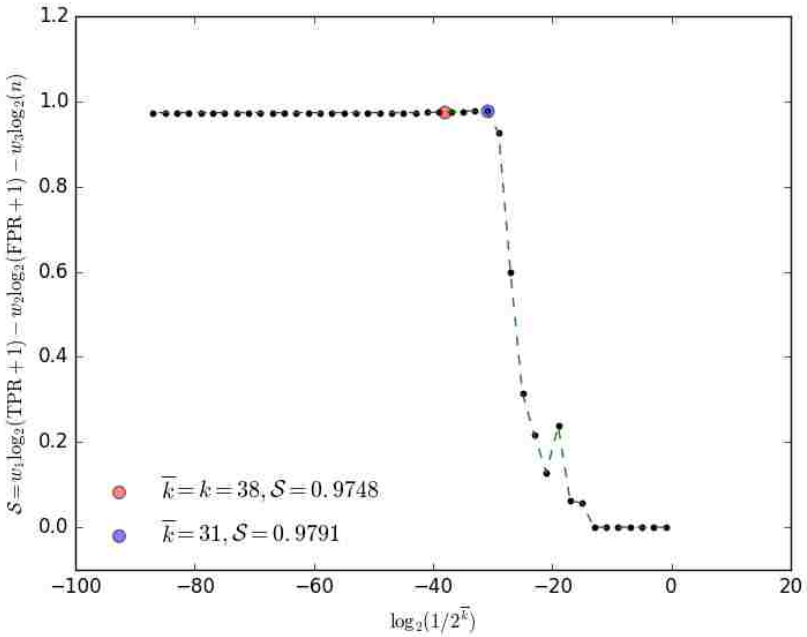


FIGURE A.16. Performance measure on *cat*, location main_D, property 1

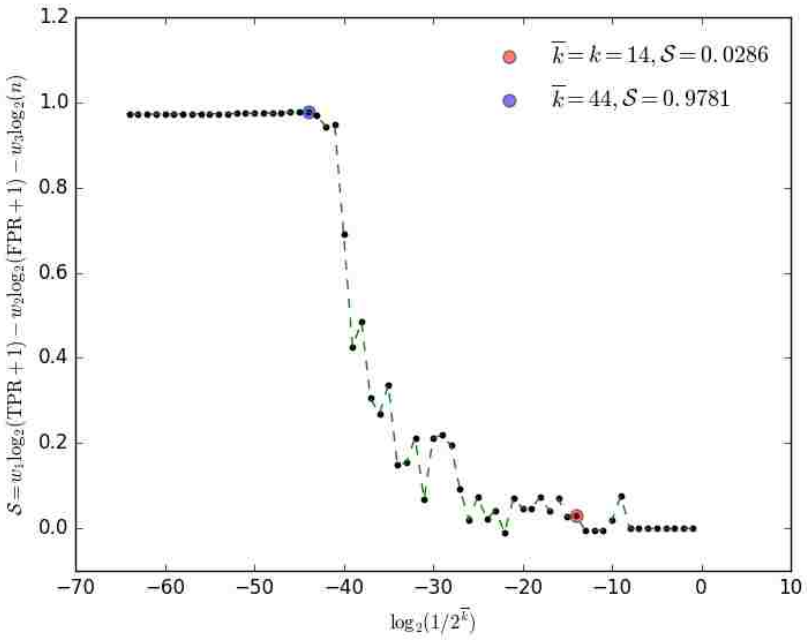


FIGURE A.17. Performance measure on *cat*, location main_DG, property 1

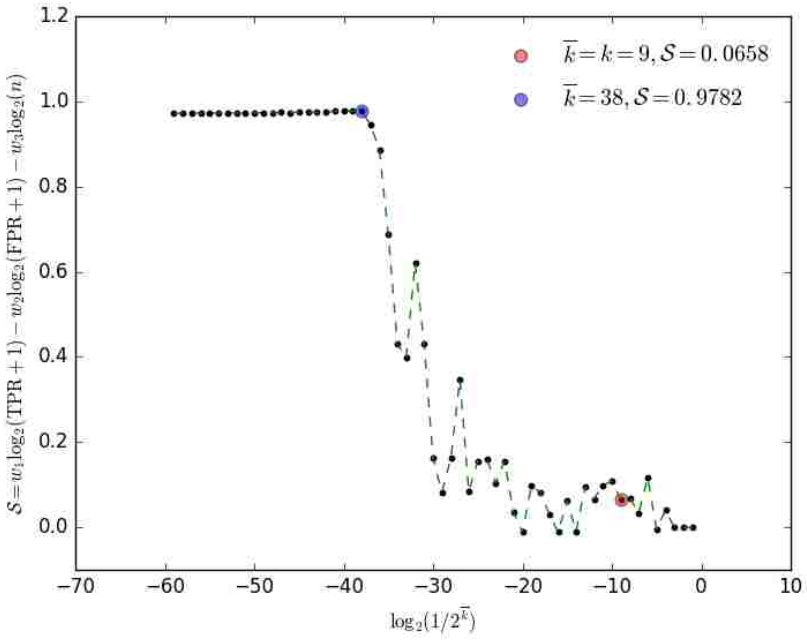


FIGURE A.18. Performance measure on *cat*, location main_DT, property 1

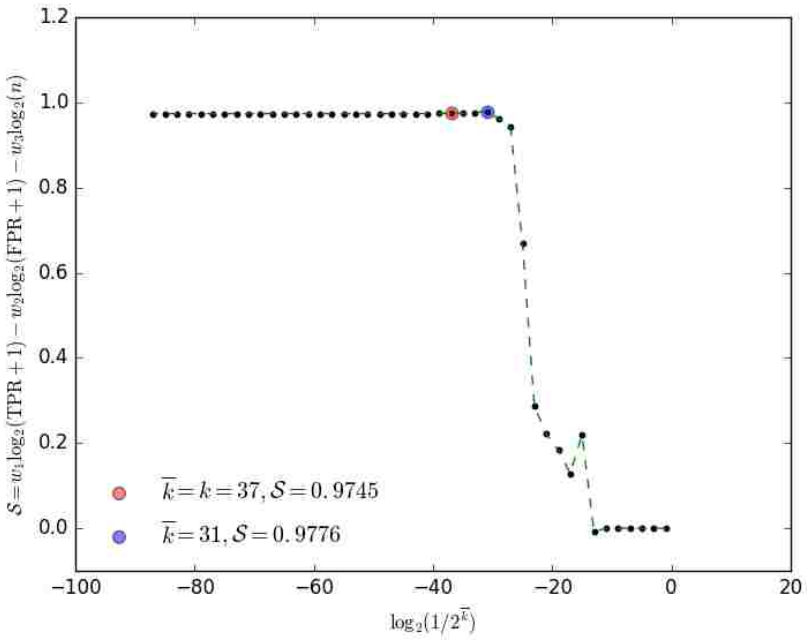


FIGURE A.19. Performance measure on *cat*, location main_E, property 1

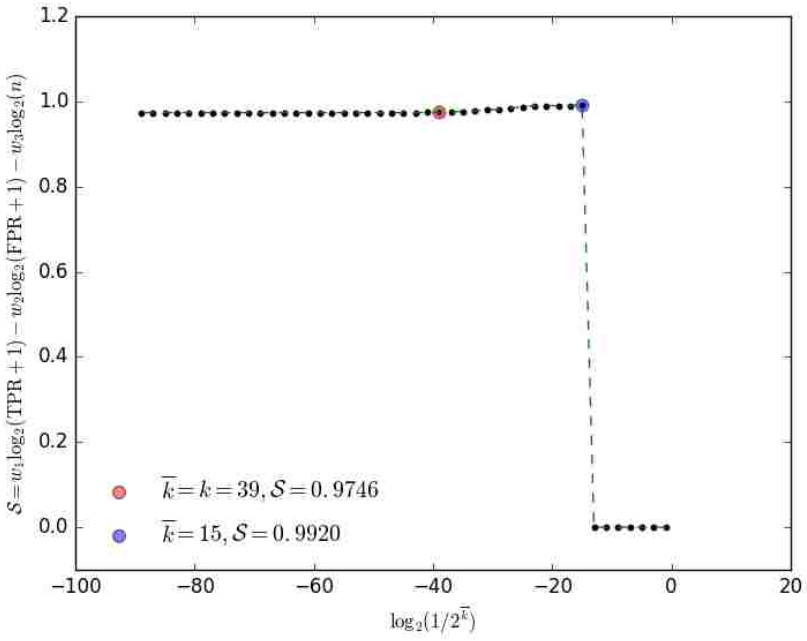


FIGURE A.20. Performance measure on *cat*, location main_C, property 2

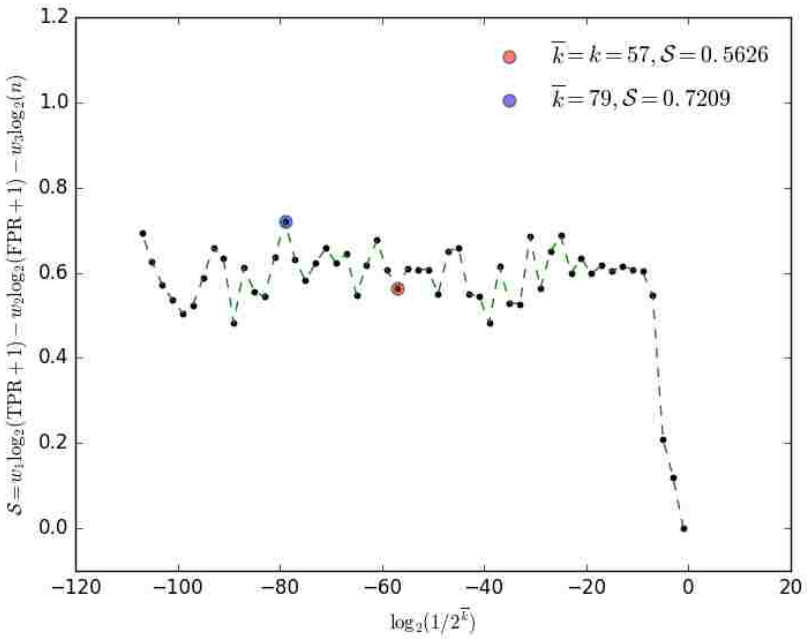


FIGURE A.21. Performance measure on *cat*, location main_CF, property 2

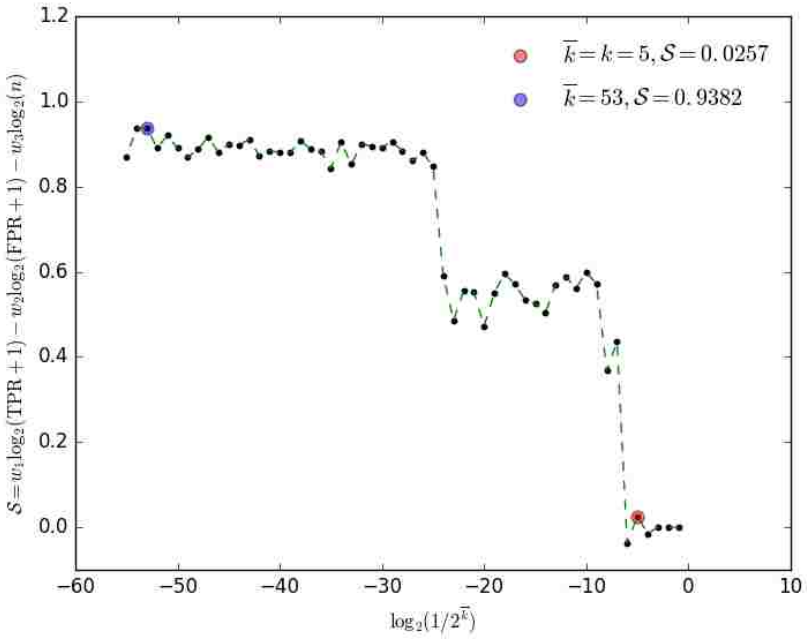


FIGURE A.22. Performance measure on *cat*, location main_DI, property 2

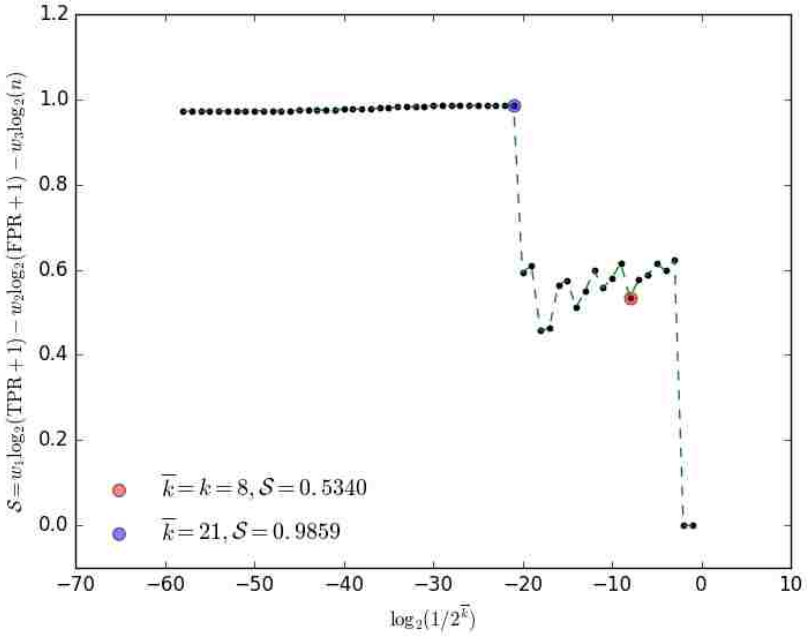


FIGURE A.23. Performance measure on *cat*, location main_DW, property 2

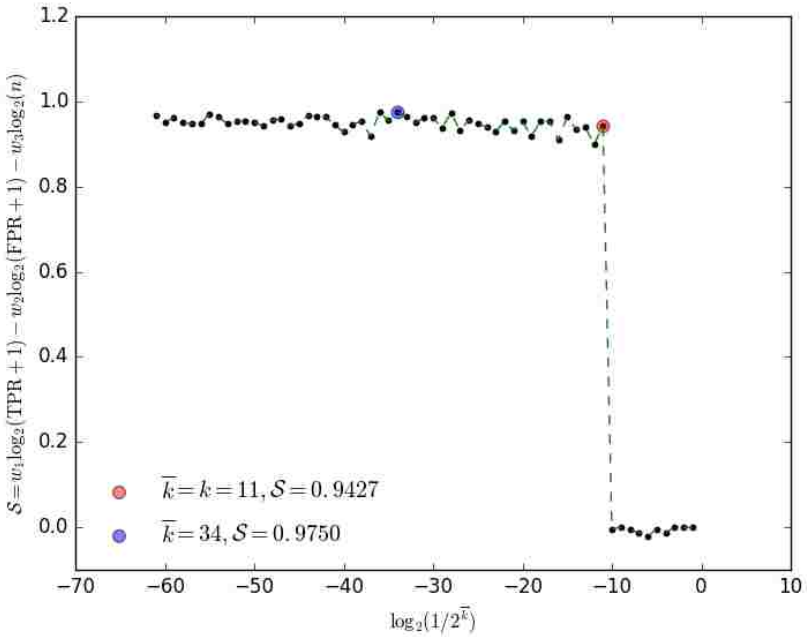


FIGURE A.24. Performance measure on *cat*, location main_O, property 2

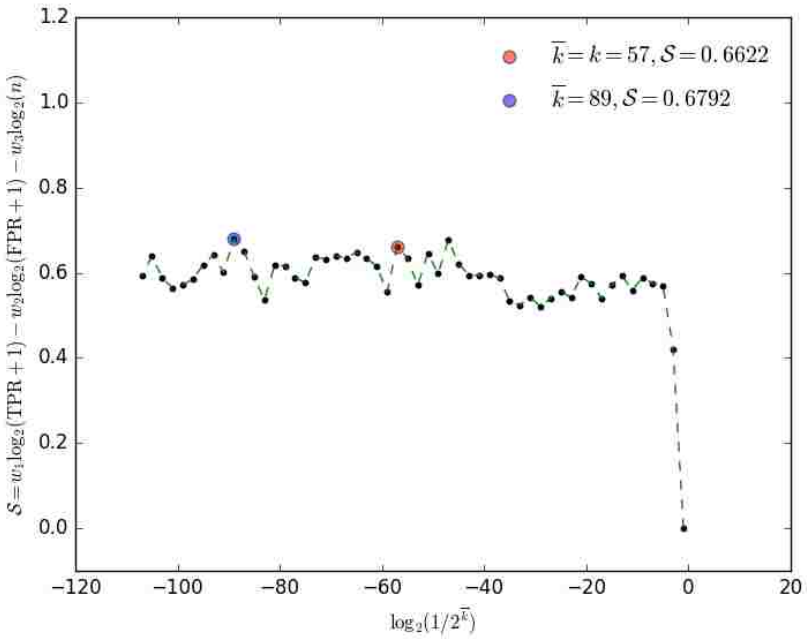


FIGURE A.25. Performance measure on *cat*, location main_CF, property 3

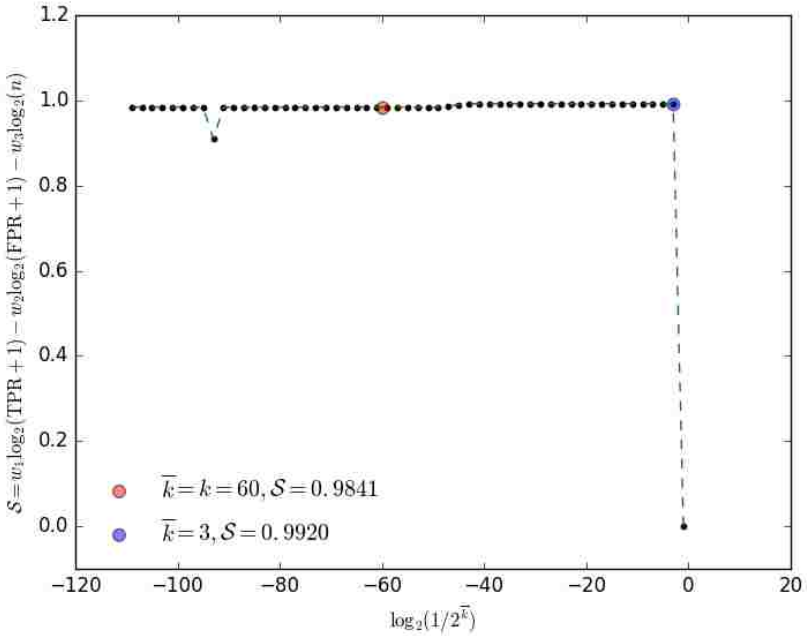


FIGURE A.26. Performance measure on *cat*, location main_CM, property 3

A.2.3 The Program *cmp*

TABLE A.4: Statistical results for program *cmp*

location	n	average m	φ_k: property 1 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AAN	9	301	0.49	0.19	12	31	0.2994
main_AAU	3	178	0.74	0.06	9	25	0.7062
main_AN	3	386	0.41	0.06	23	22	0.3984
main_J	5	125	0.81	0.03	18	50	0.8029
main_P	3	177	0.83	0.15	38	37	0.6659
location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_N	3	177	0.69	0.006	39	81	0.7361
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_J	3	208	0.92	0.34	18	3	0.5125

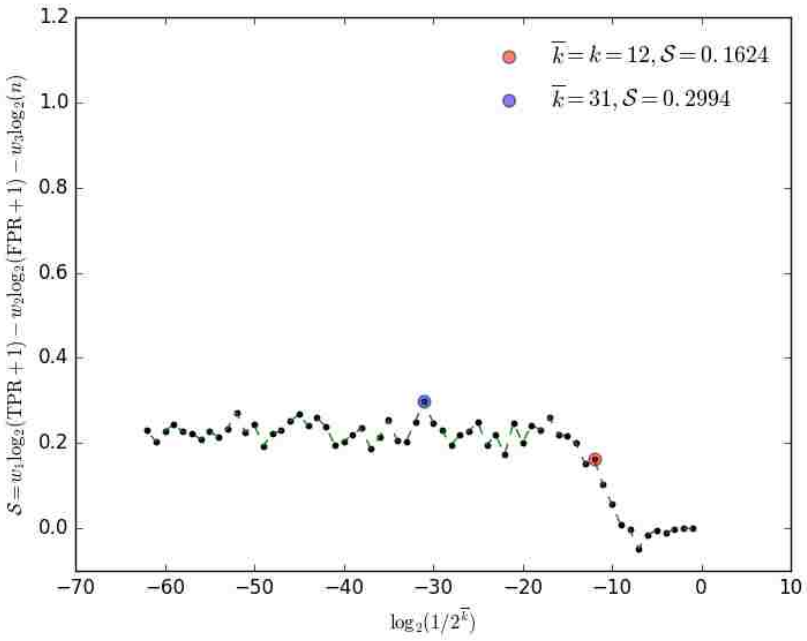


FIGURE A.27. Performance measure on *cmp*, location main_AAN, property 1

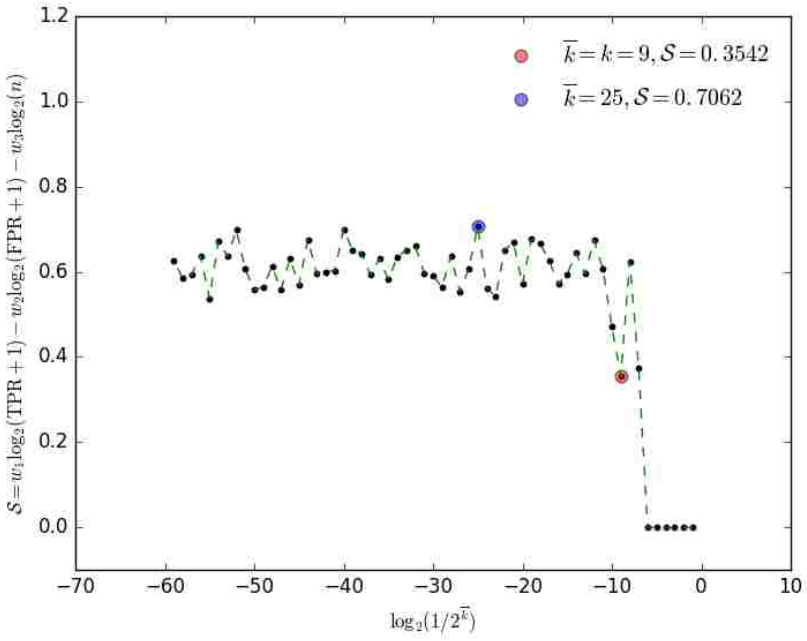


FIGURE A.28. Performance measure on *cmp*, location main_AAU, property 1

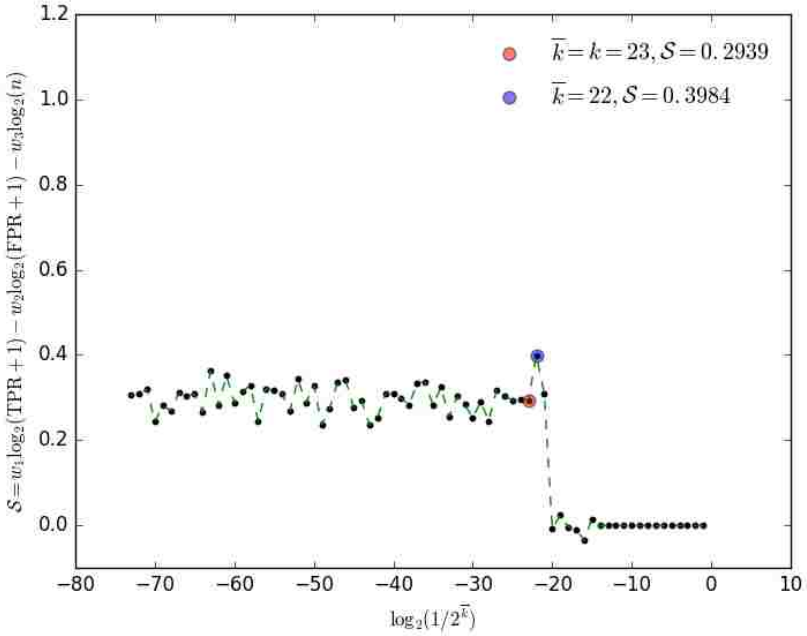


FIGURE A.29. Performance measure on *cmp*, location main_AN, property 1

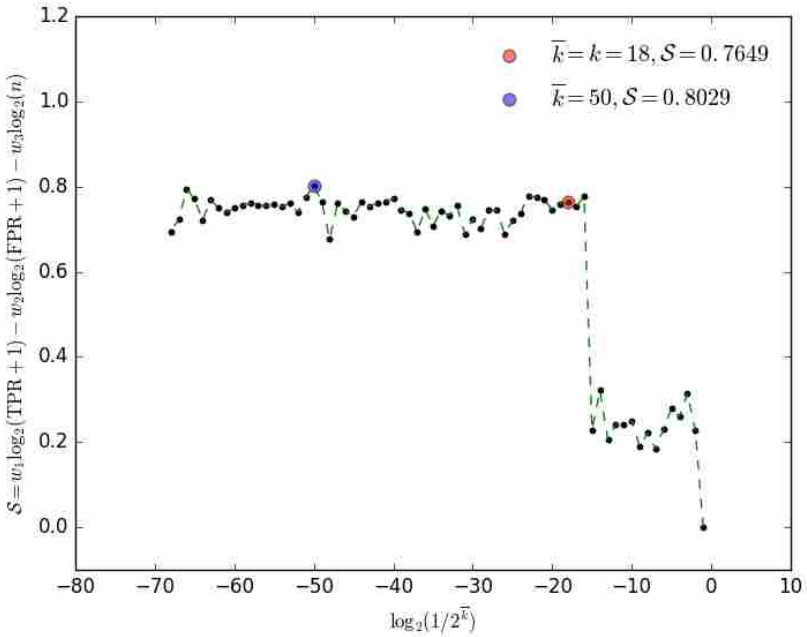


FIGURE A.30. Performance measure on *cmp*, location main_J, property 1

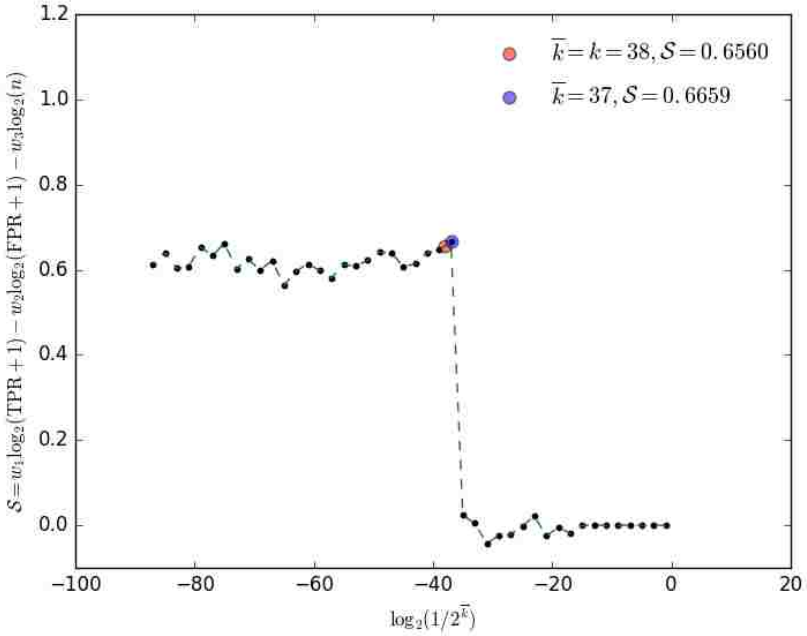


FIGURE A.31. Performance measure on *cmp*, location main_P, property 1

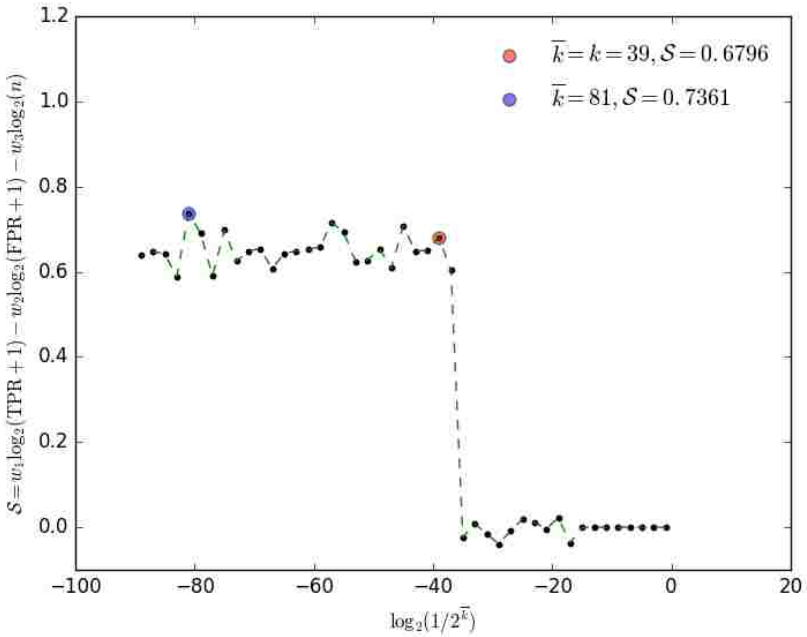


FIGURE A.32. Performance measure on *cmp*, location main_N, property 2

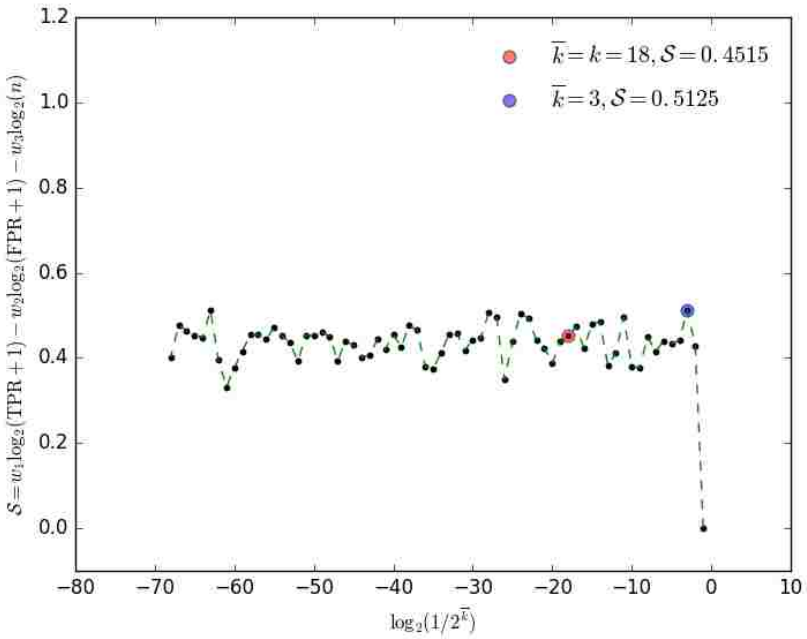


FIGURE A.33. Performance measure on *cmp*, location main_J, property 3

A.2.4 The Program *comm*

TABLE A.5: Statistical results for program *comm*

location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AE	14	89	0.74	0.14	50	49	0.5847
main_BN	4	287	0.71	0.36	19	34	0.3173
main_CH	6	772	0.42	0.19	6	7	0.2359
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AE	14	89	0.90	0.16	50	43	0.6937
main_BE	27	169	0.82	0.36	40	35	0.3995
main_CF	8	570	0.75	0.45	8	8	0.2533
main_R	18	54	1.0	0	52	23	0.9791
main_T	14	64	0.94	0	57	43	0.9611

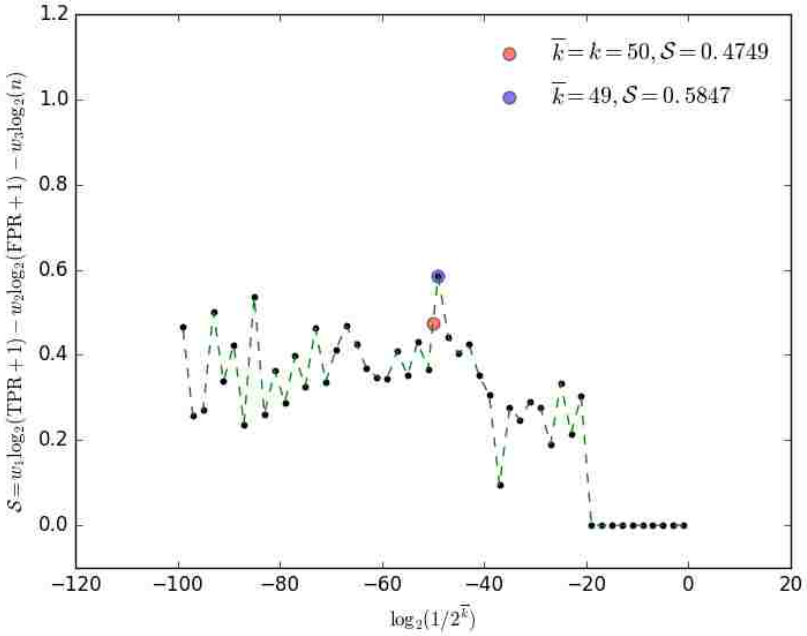


FIGURE A.34. Performance measure on *comm*, location main_AE, property 2

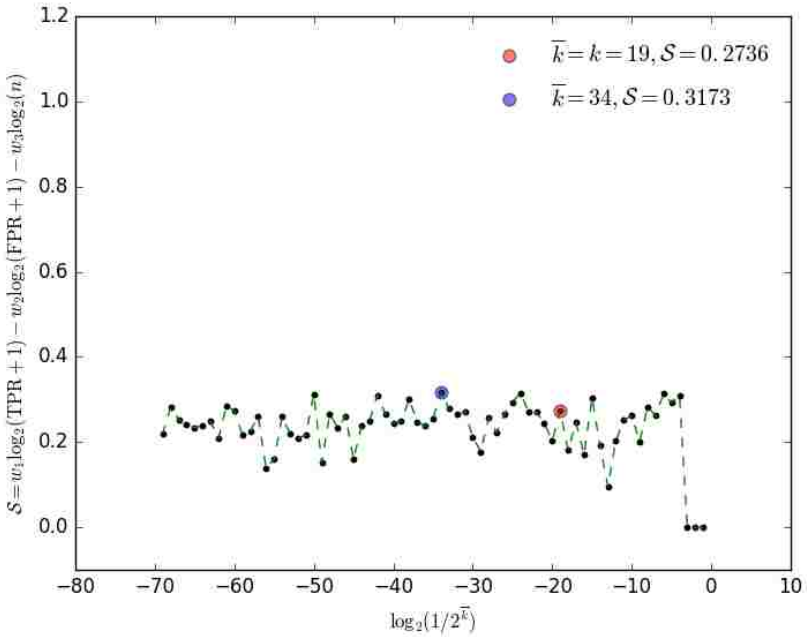


FIGURE A.35. Performance measure on *comm*, location main_BN, property 2

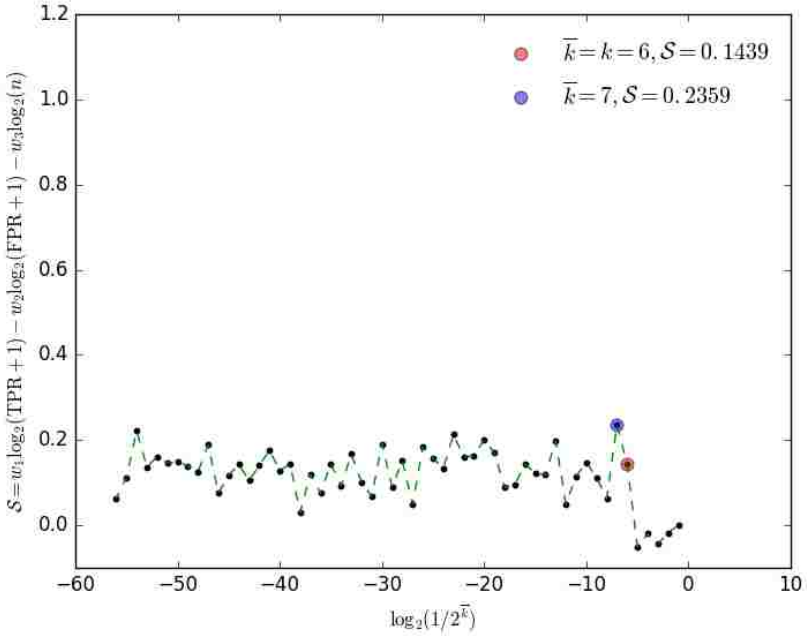


FIGURE A.36. Performance measure on *comm*, location main_CH, property 2

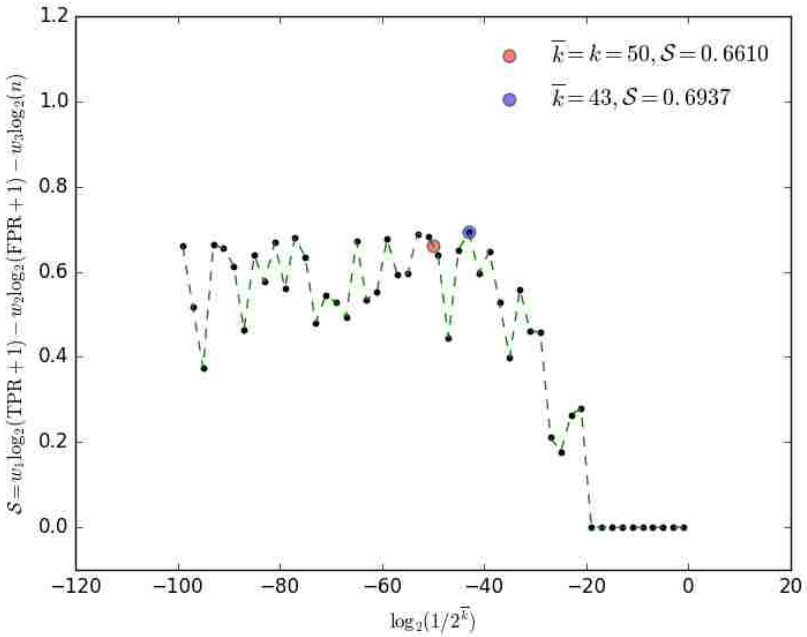


FIGURE A.37. Performance measure on *comm*, location main_AE, property 3

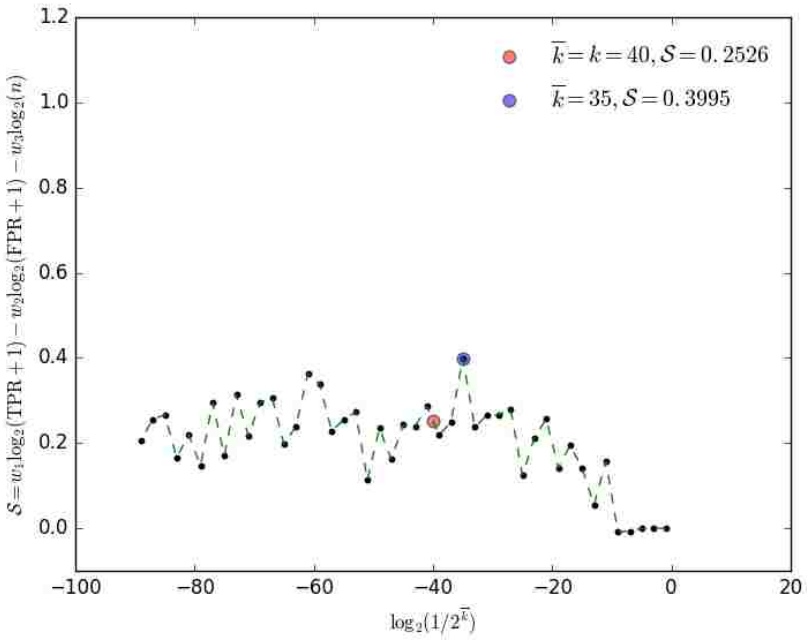


FIGURE A.38. Performance measure on *comm*, location main_BE, property 3

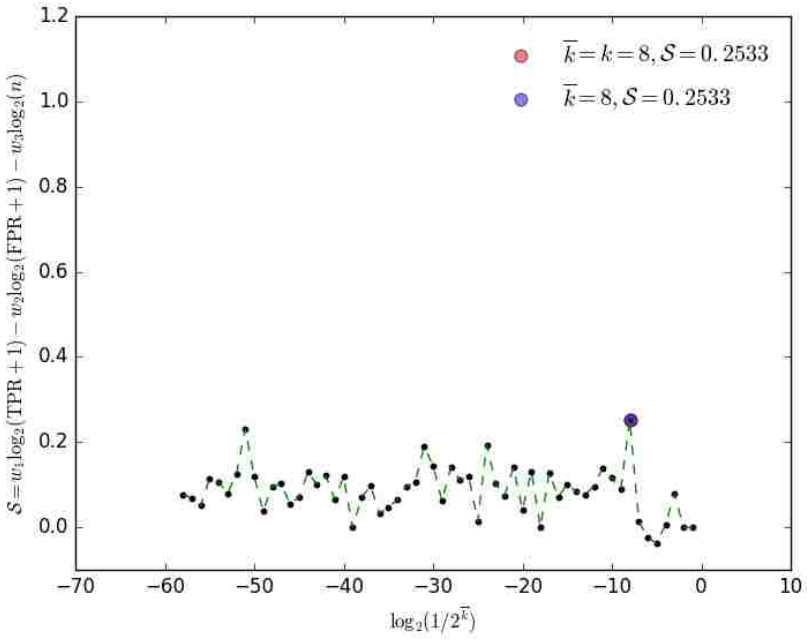


FIGURE A.39. Performance measure on *comm*, location main_CF, property 3

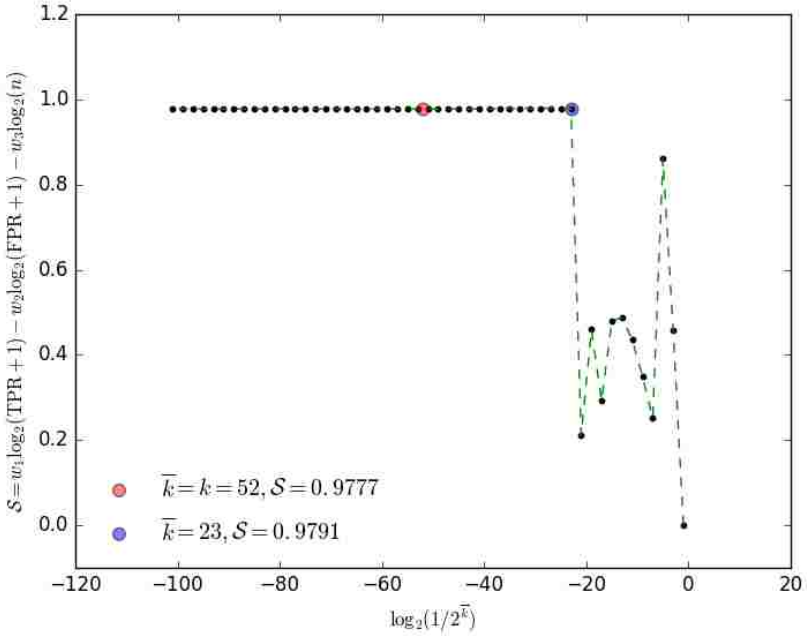


FIGURE A.40. Performance measure on *comm*, location main_R, property 3

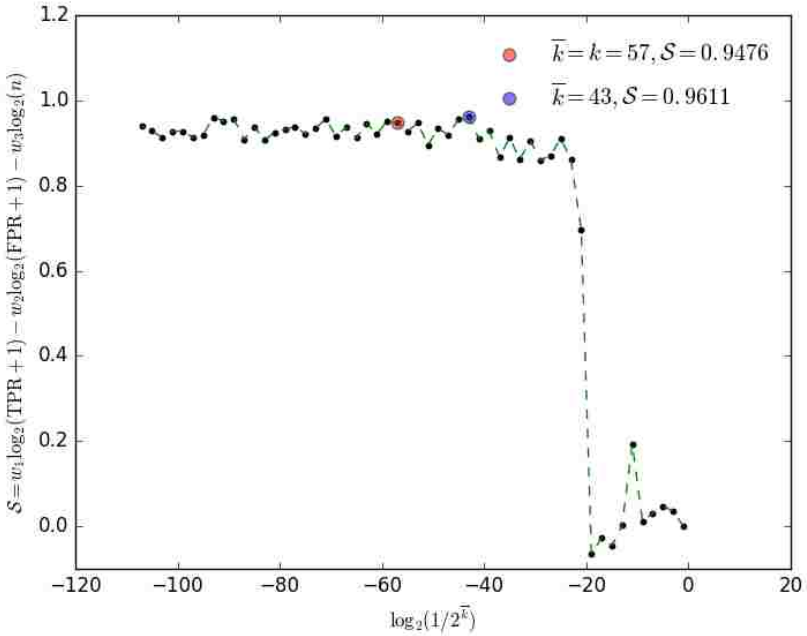


FIGURE A.41. Performance measure on *comm*, location main_T, property 3

A.2.5 The Program *cp*

TABLE A.6: Statistical results for program *cp*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AA	7	102	1.0	0	3	23	0.757
main_AI	7	102	0.72	0.1	3	12	0.627
location	n	average m	φ_k: property 2 TPR	φ_k: property 2 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AI	5	143	1.0	0.29	3	3	0.6201
main_BB	7	115	1.0	0.24	4	15	0.6764
main_BD	10	143	1.0	0.06	4	3	0.9001
main_BS	10	303	1.0	0.05	2	7	0.9081
location	n	average m	φ_k: property 3 TPR	φ_k: property 3 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AI	5	143	1.0	0.13	3	3	0.8106
main_AU	4	156	0.71	0.01	3	16	0.7435
main_BB	3	267	1.0	0	4	2	0.992

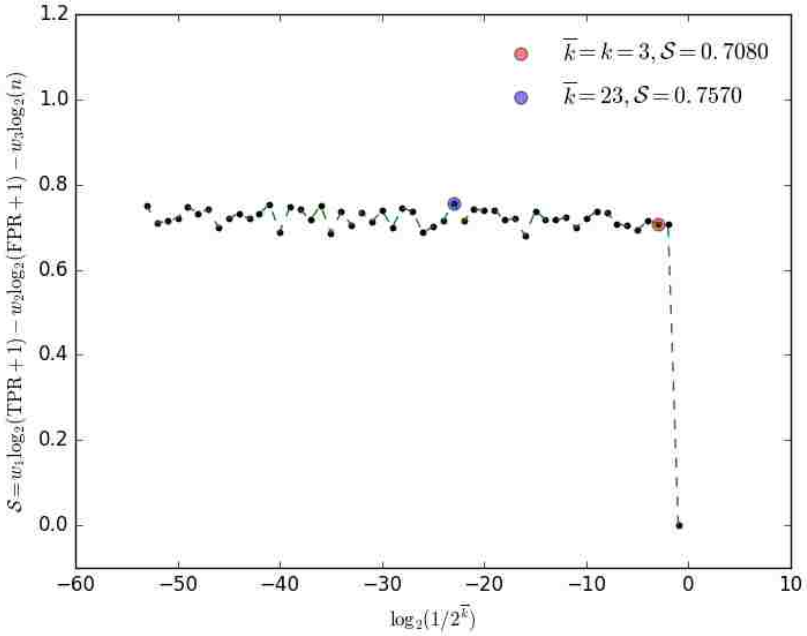


FIGURE A.42. Performance measure on *cp*, location main_AA, property 1

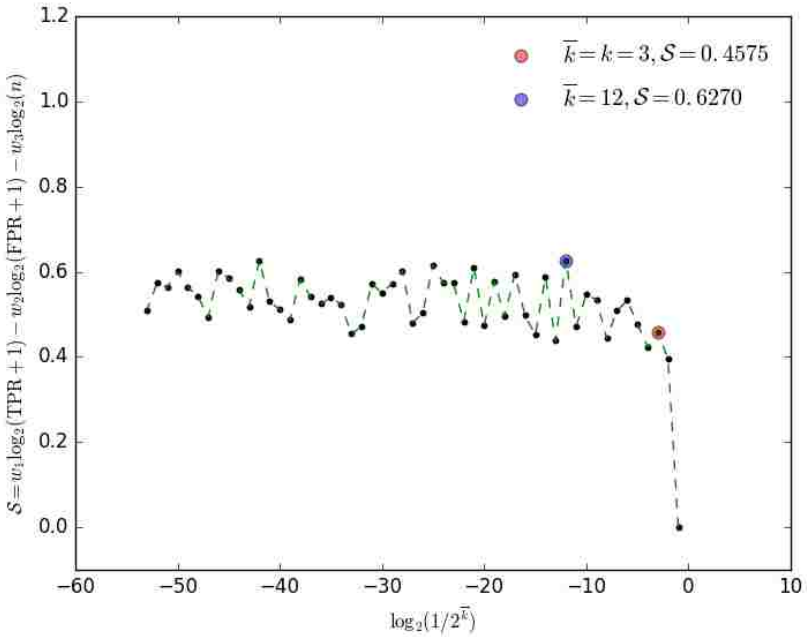


FIGURE A.43. Performance measure on *cp*, location main_AI, property 1

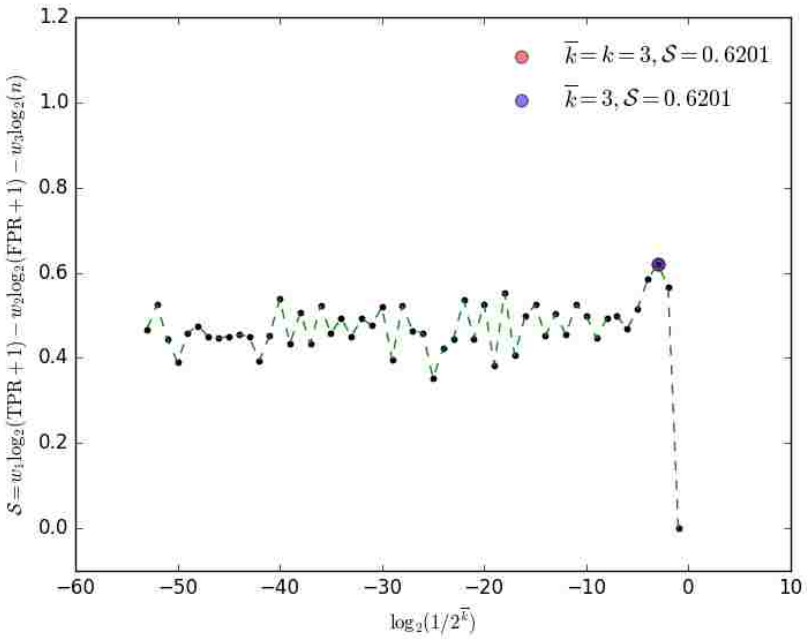


FIGURE A.44. Performance measure on *cp*, location main_AI, property 2

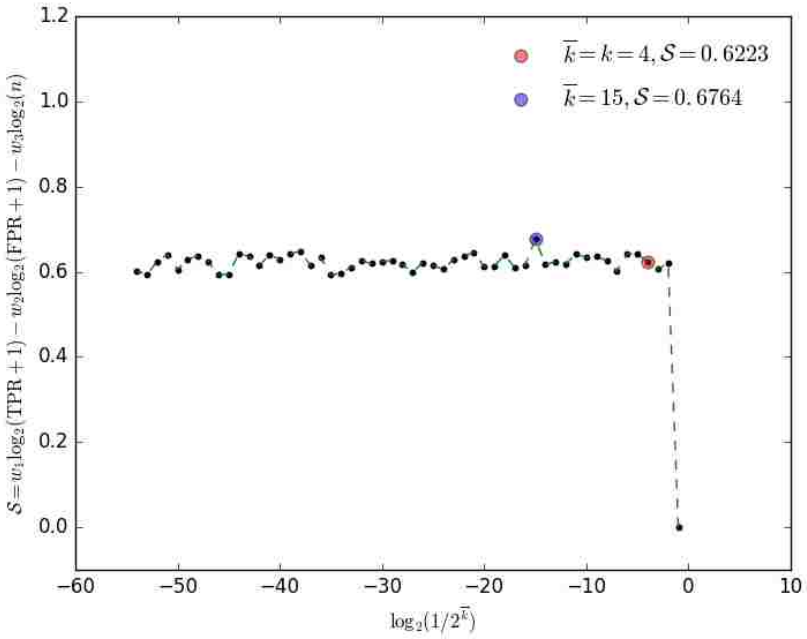


FIGURE A.45. Performance measure on *cp*, location main_BB, property 2

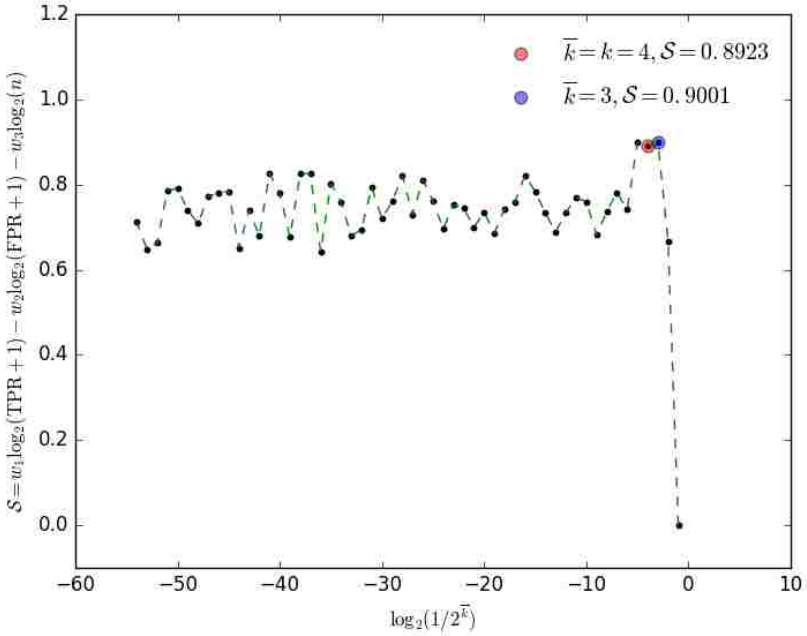


FIGURE A.46. Performance measure on *cp*, location main_BD, property 2

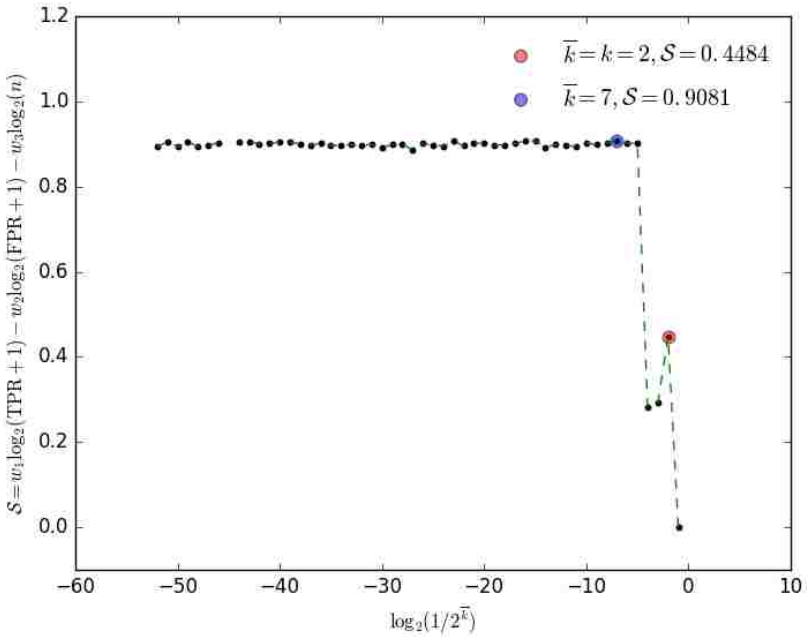


FIGURE A.47. Performance measure on *cp*, location main_BS, property 2

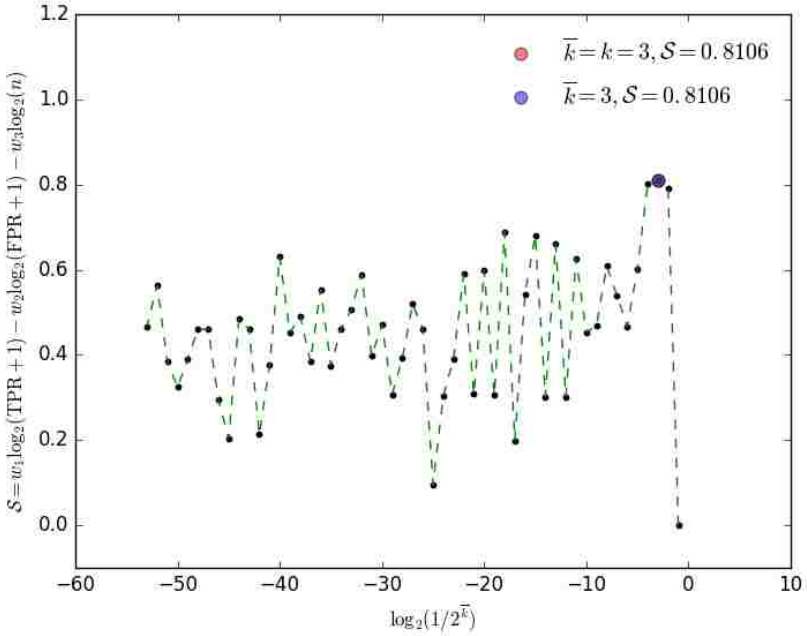


FIGURE A.48. Performance measure on *cp*, location main_AI, property 3

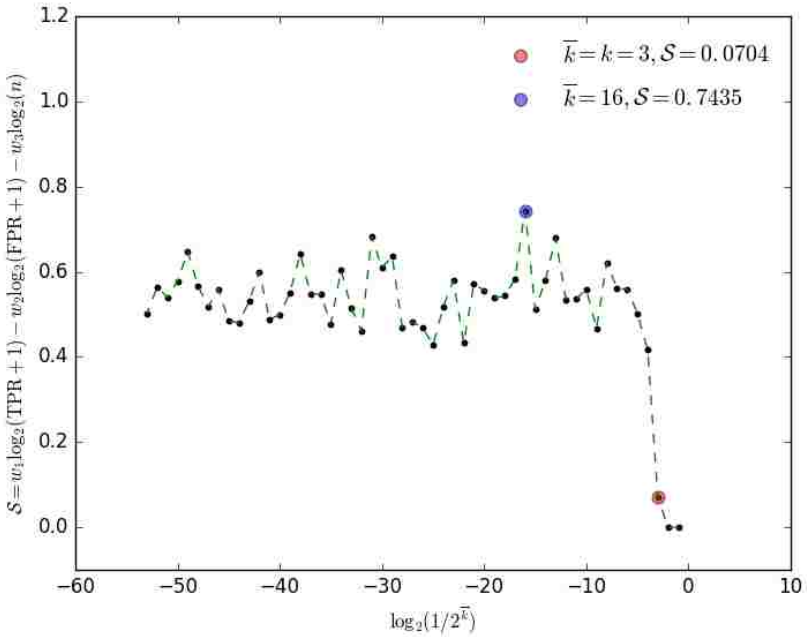


FIGURE A.49. Performance measure on *cp*, location main_AU, property 3

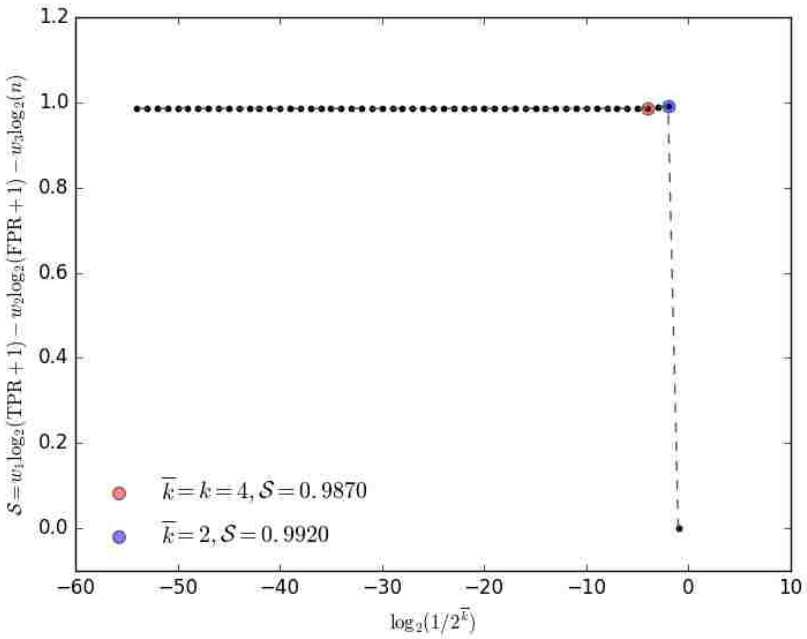


FIGURE A.50. Performance measure on *cp*, location main_BB, property 3

A.2.6 The Program *csplit*

TABLE A.7: Statistical results for program *csplit*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AAB	3	624	0.44	0.08	23	23	0.4043
main_AC	3	536	0.43	0.1	86	97	0.3584
main_AQ	33	616	0.31	0.05	33	75	0.3045
main_W	3	89	1.0	0	92	83	0.992
location	n	average m	φ_k: property 2 TPR	φ_k: property 2 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_CAA	6	446	0.41	0.03	3	42	0.4388
main_FAC	2	268	1.0	0.25	2	38	0.6745
main_W	2	134	0.59	0	92	33	0.6658
location	n	average m	φ_k: property 3 TPR	φ_k: property 3 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_Q	4	112	0.8	0	88	89	0.8379

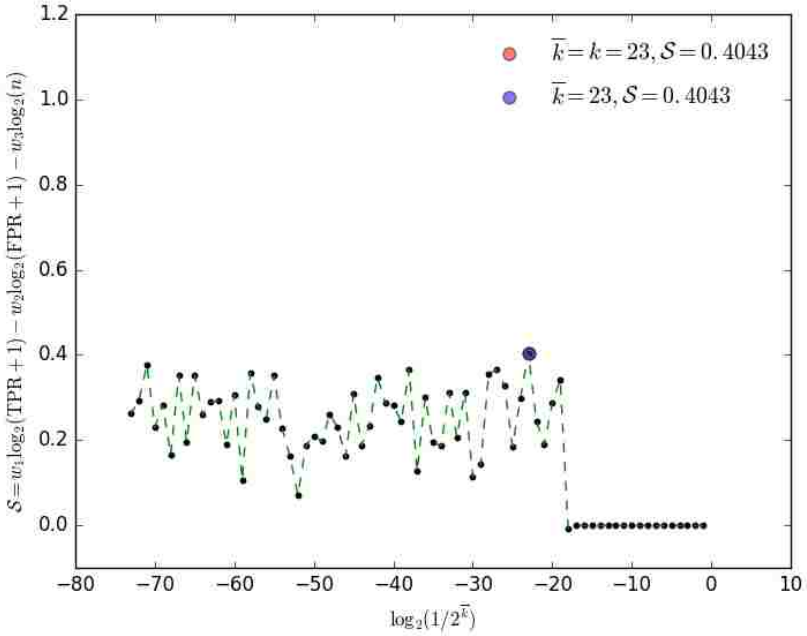


FIGURE A.51. Performance measure on *csplit*, location main_AAB, property 1

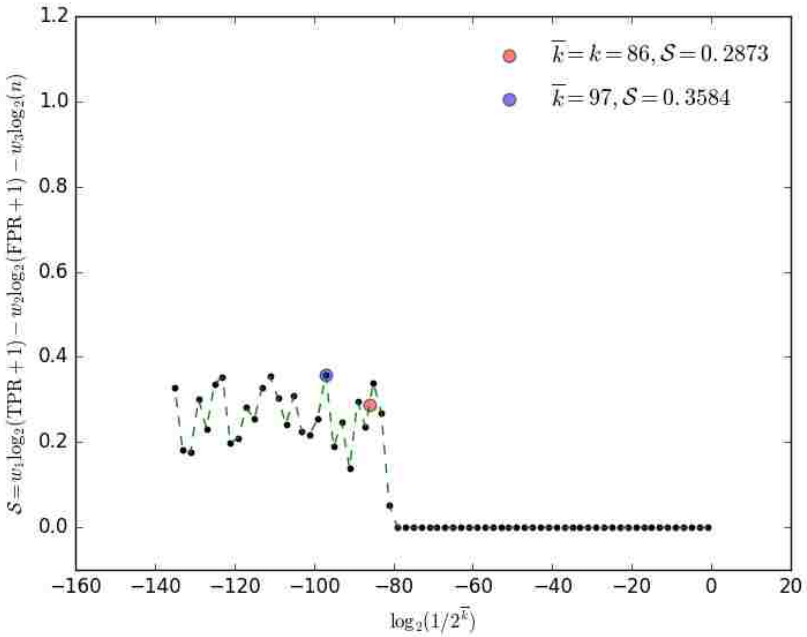


FIGURE A.52. Performance measure on *csplit*, location main_AC, property 1

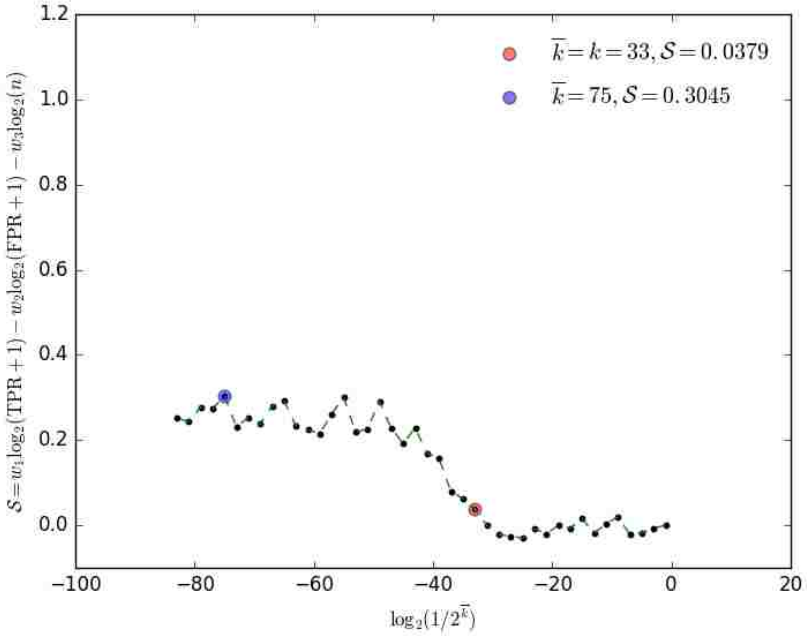


FIGURE A.53. Performance measure on *csplit*, location main_AQ, property 1

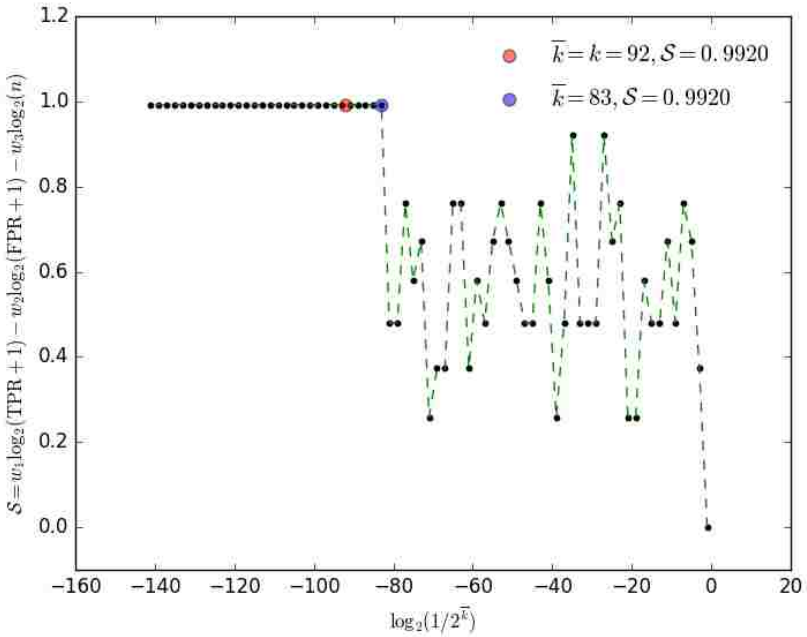


FIGURE A.54. Performance measure on *csplit*, location main_W, property 1

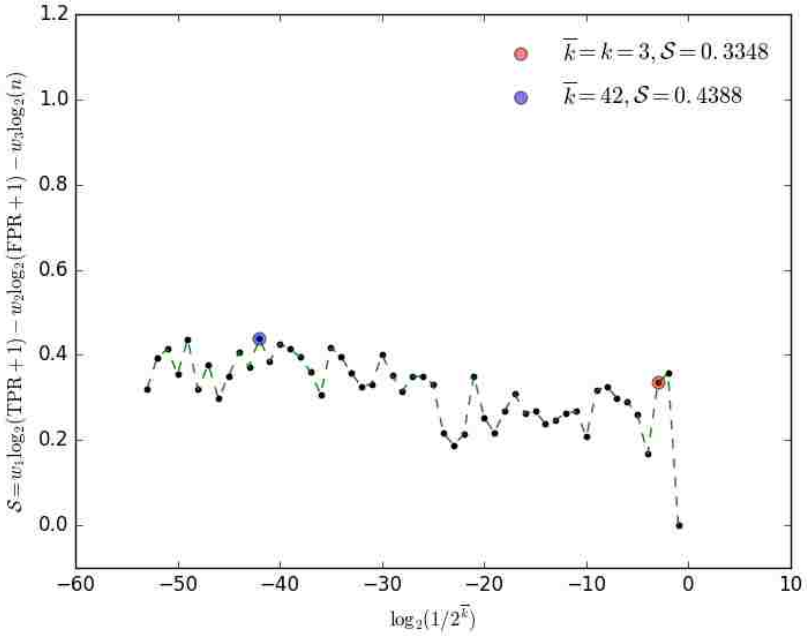


FIGURE A.55. Performance measure on *csplit*, location main_CAA, property 2

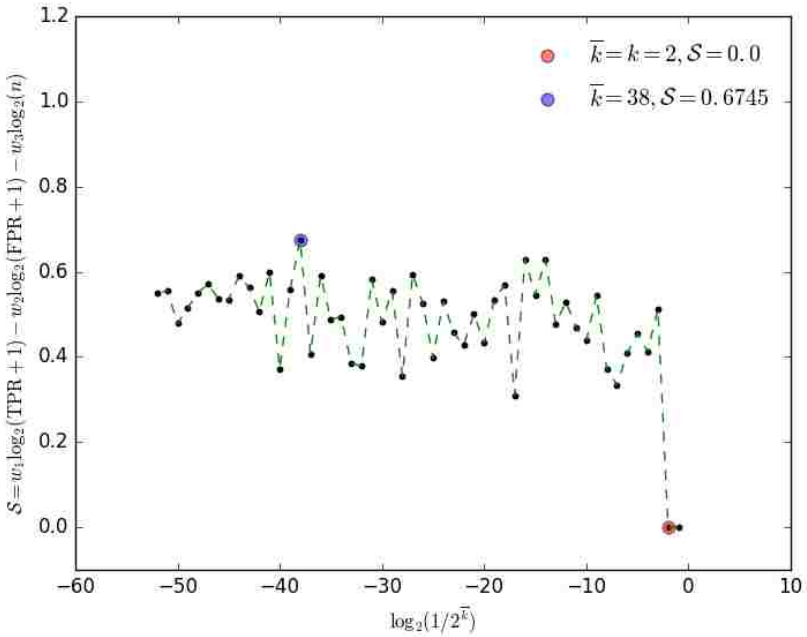


FIGURE A.56. Performance measure on *csplit*, location main_FAC, property 2

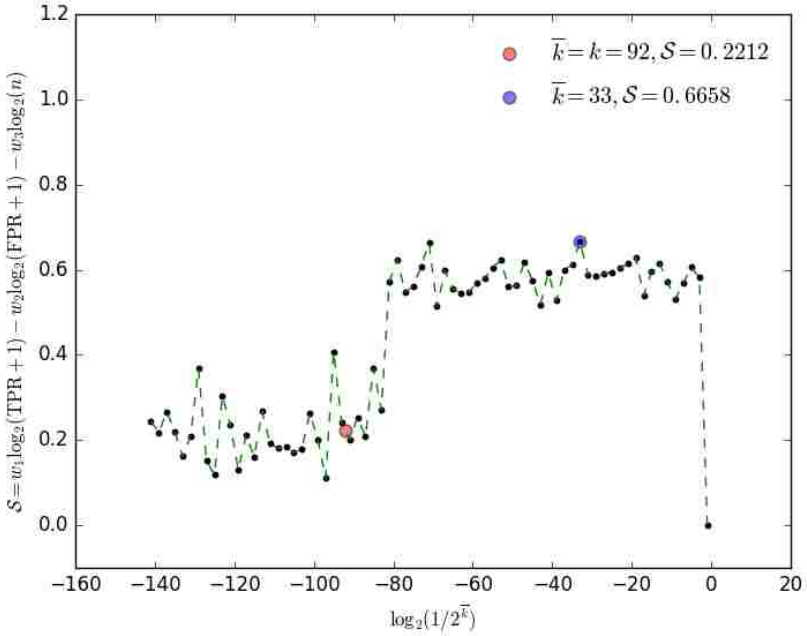


FIGURE A.57. Performance measure on *csplit*, location main_W, property 2

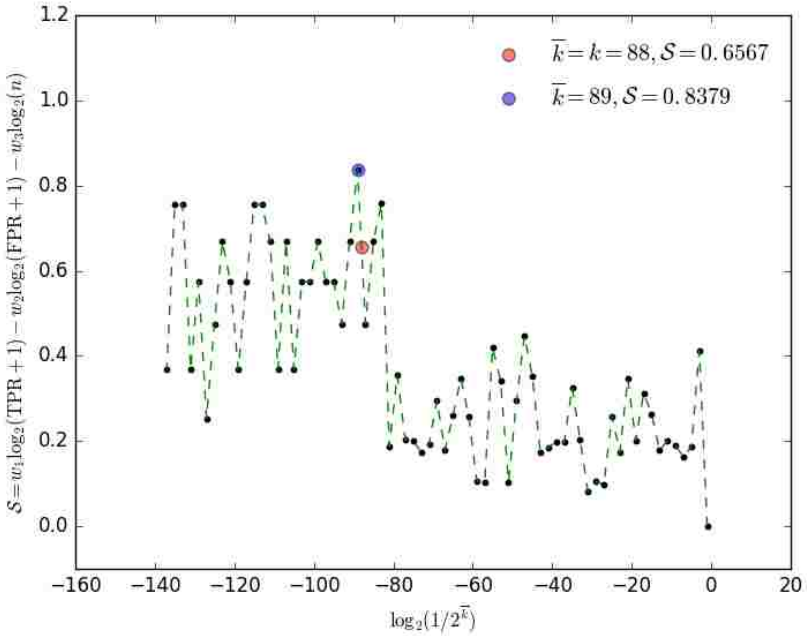


FIGURE A.58. Performance measure on *csplit*, location main_Q, property 3

A.2.7 The Program *dd*

TABLE A.8: Statistical results for program *dd*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AAR	22	252	0.88	0.22	2	40	0.6009
main_ABF	18	204	1.0	0.07	8	34	0.8848
main_ABG	16	229	1.0	0.06	7	28	0.8899
main_ABI	3	445	1.0	0.28	5	16	0.6353
location	n	average m	φ_k: property 2 TPR	φ_k: property 2 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_ABG	18	204	0.91	0.13	7	32	0.733
main_NAA	6	312	1.0	0	14	13	0.987
main_X	4	267	1.0	0	13	12	0.99
location	n	average m	φ_k: property 3 TPR	φ_k: property 3 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_ABF	4	916	0.91	0	8	8	0.9262
main_ABG	4	916	0.91	0	7	6	0.9239
main_ABI	3	445	0.88	0.06	5	16	0.812

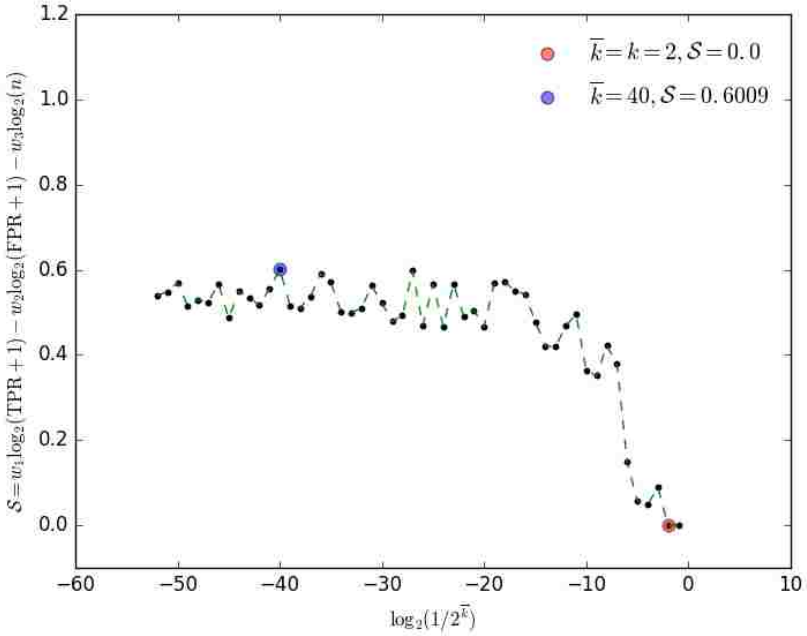


FIGURE A.59. Performance measure on *dd*, location main_AAR, property 1

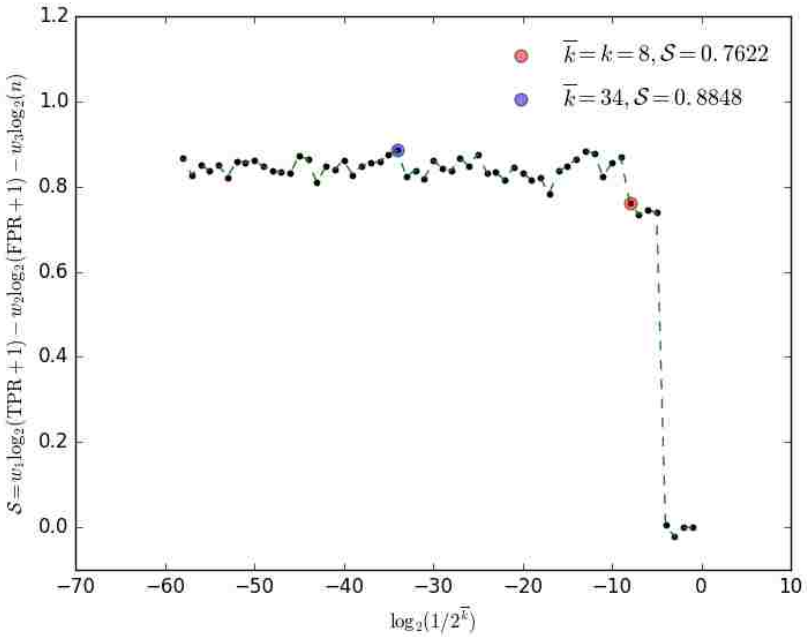


FIGURE A.60. Performance measure on *dd*, location main_ABF, property 1

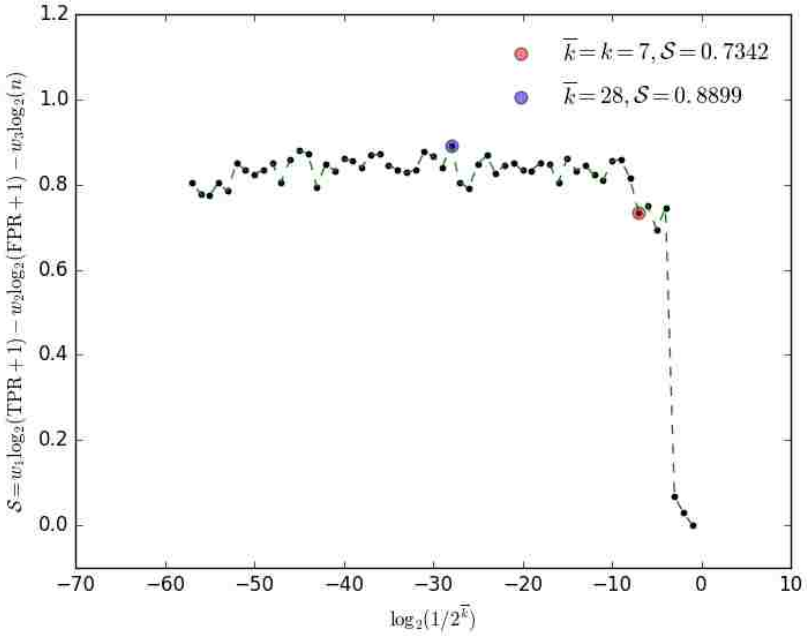


FIGURE A.61. Performance measure on *dd*, location main_ABG, property 1

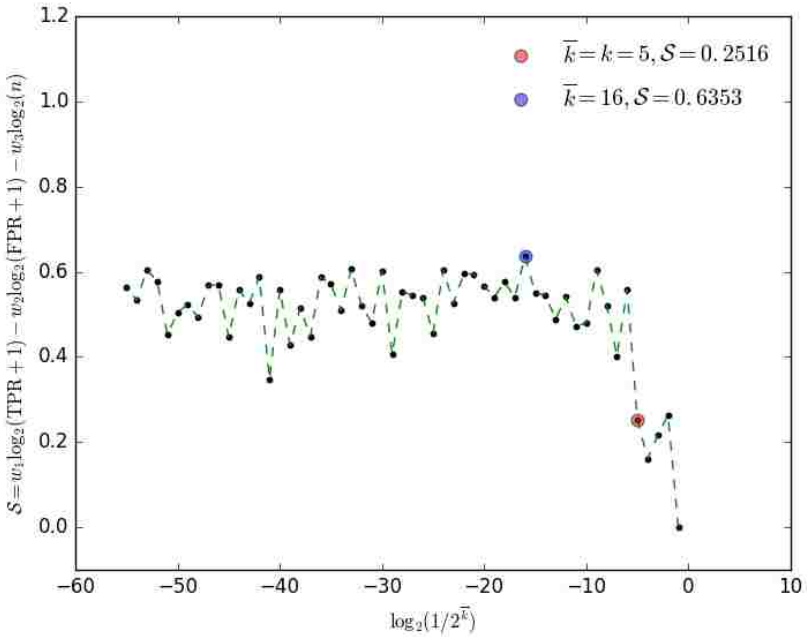


FIGURE A.62. Performance measure on *dd*, location main_ABI, property 1

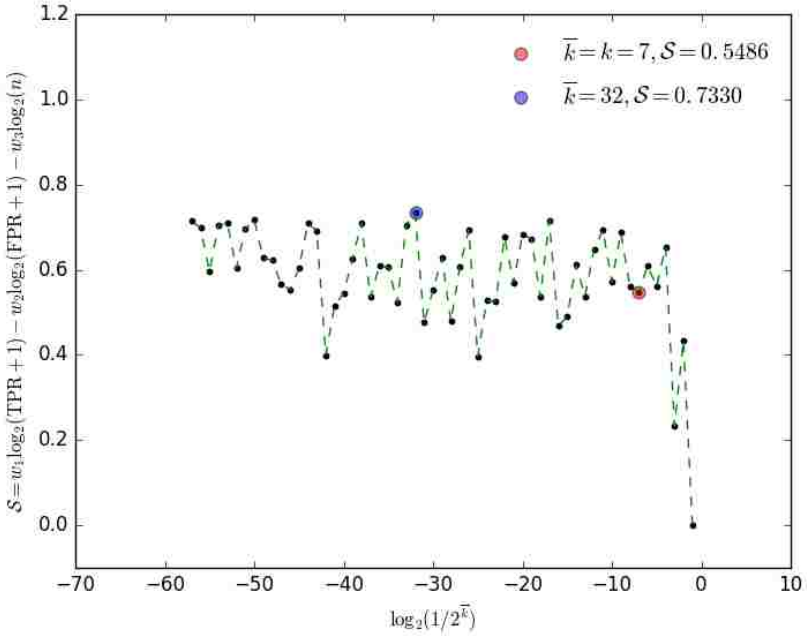


FIGURE A.63. Performance measure on *dd*, location main_ABG, property 2

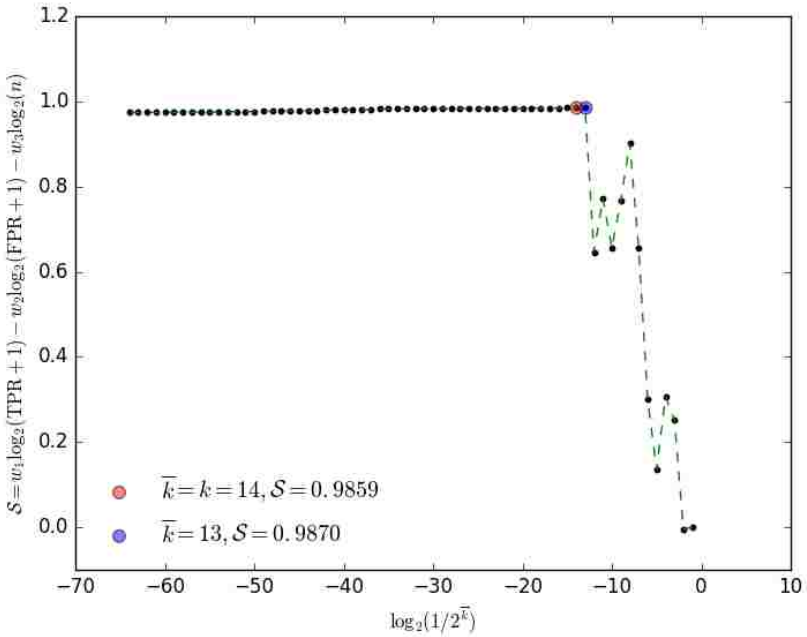


FIGURE A.64. Performance measure on *dd*, location main_NAA, property 2

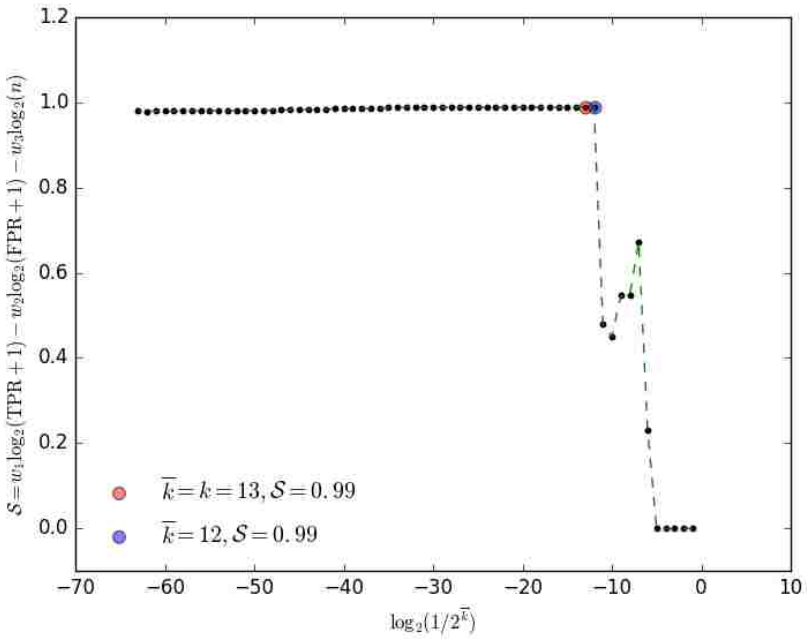


FIGURE A.65. Performance measure on *dd*, location main_X, property 2

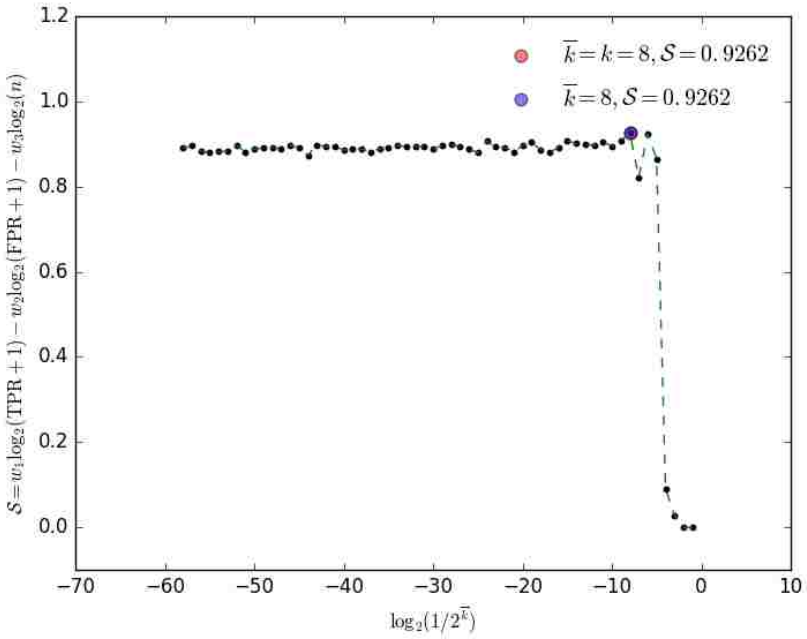


FIGURE A.66. Performance measure on *dd*, location main_ABF, property 3

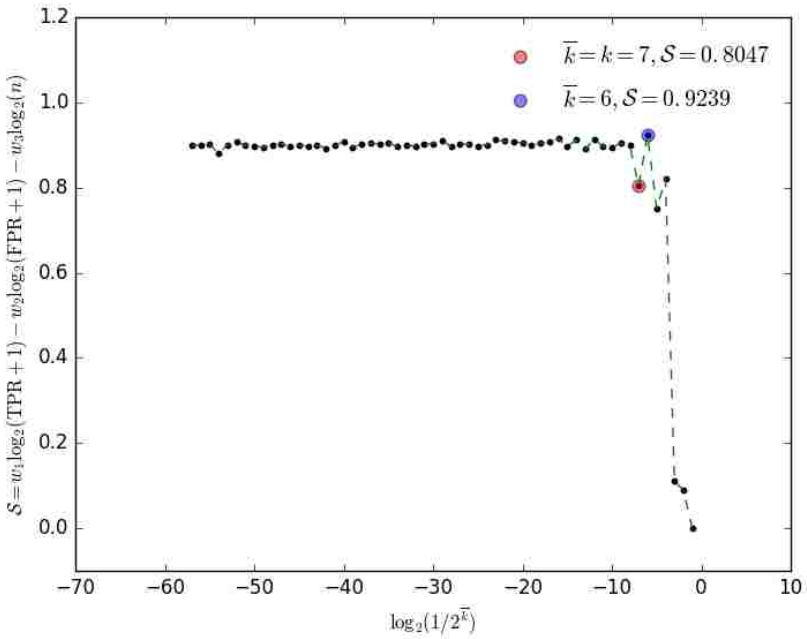


FIGURE A.67. Performance measure on *dd*, location main_ABG, property 3

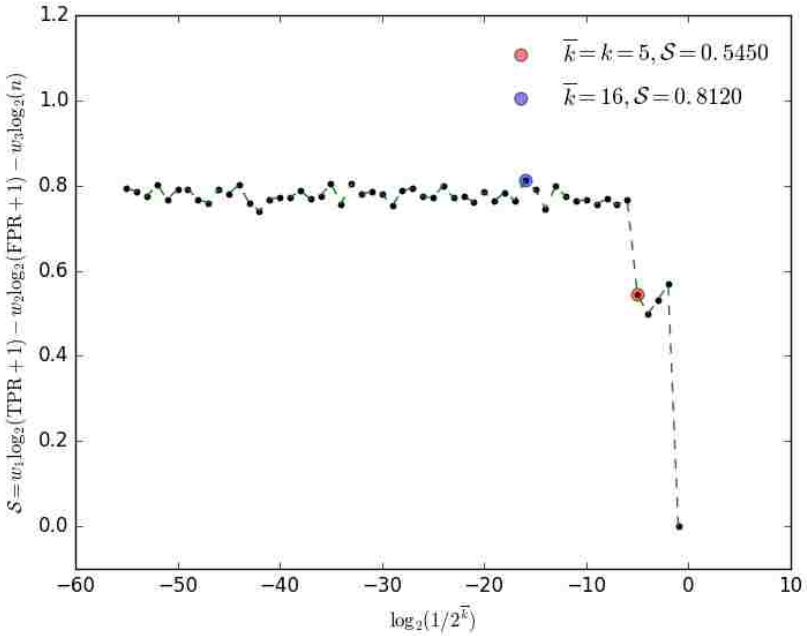


FIGURE A.68. Performance measure on *dd*, location main_ABI, property 3

A.2.8 The Program *diff*

TABLE A.9: Statistical results for program *diff*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AAA	2	936	0.9	0.66	34	29	0.1851
main_DL	10	241	0.7	0.33	53	49	0.3374
main_DR	3	446	0.66	0.42	52	49	0.2132
main_DS	4	468	0.75	0.23	51	63	0.5014
main_ED	3	624	0.68	0.23	42	41	0.4383
location	n	average m	φ_k: property 3 TPR	φ_k: property 3 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_BG	2	223	1.0	0	54	3	0.995
main_DL	11	219	0.79	0.21	53	101	0.5492

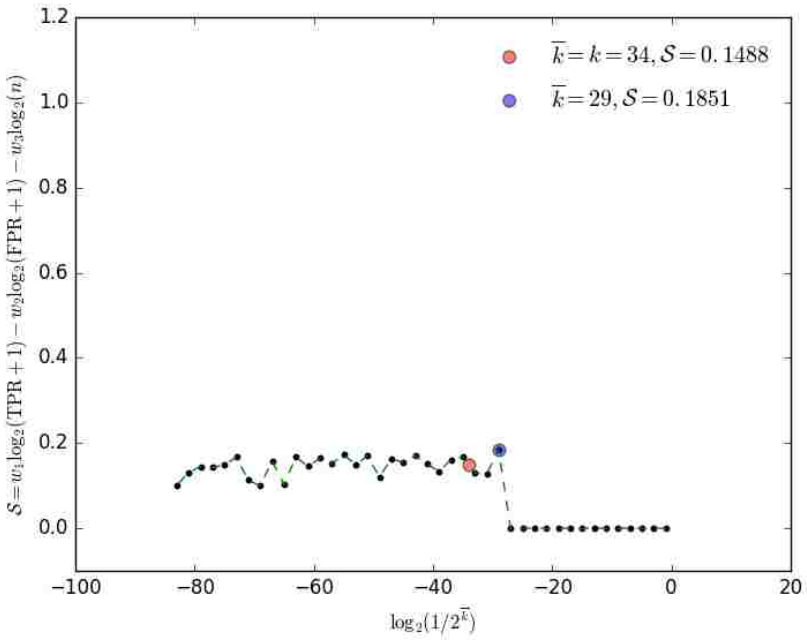


FIGURE A.69. Performance measure on *diff*, location main_AAA, property 1

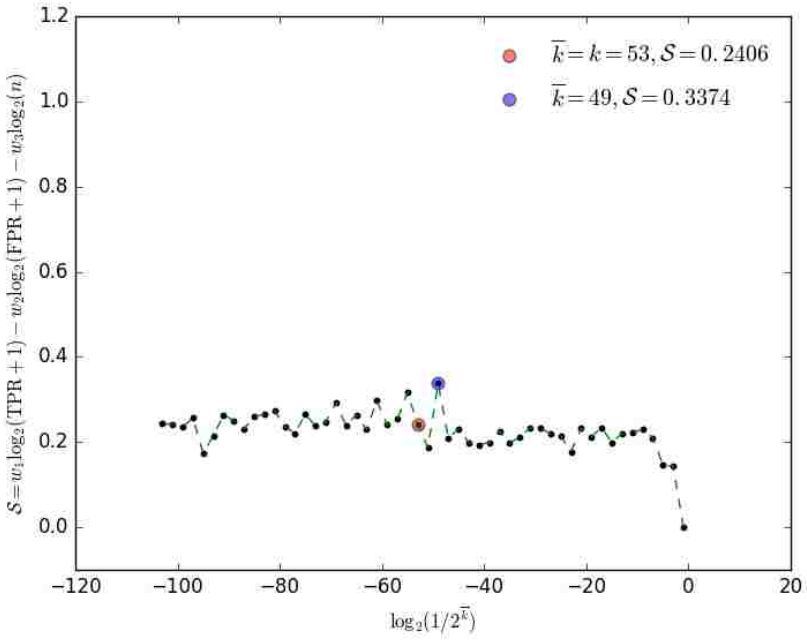


FIGURE A.70. Performance measure on *diff*, location main_DL, property 1

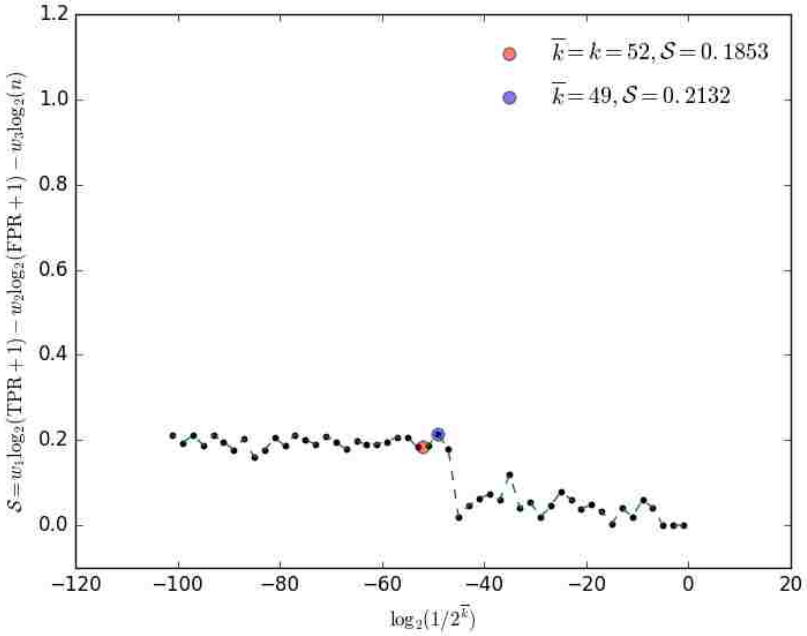


FIGURE A.71. Performance measure on *diff*, location main_DR, property 1

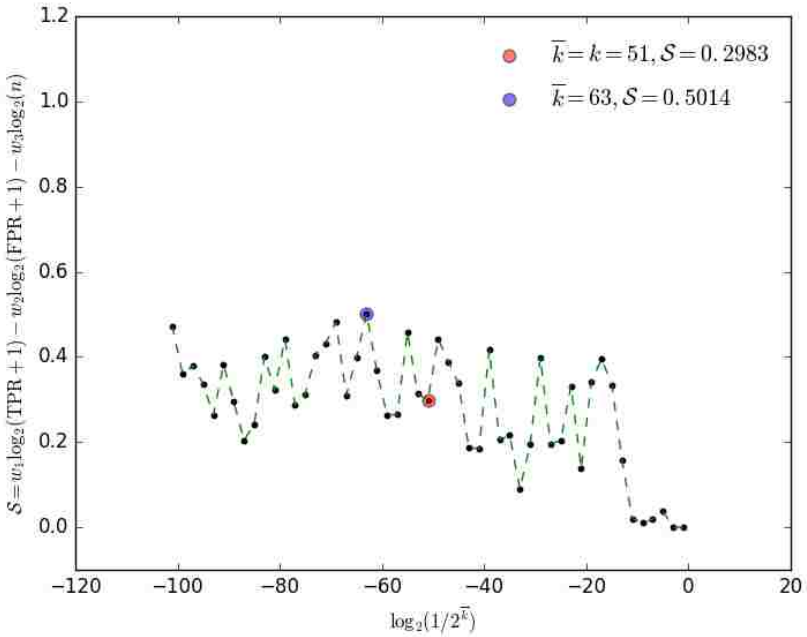


FIGURE A.72. Performance measure on *diff*, location main_DS, property 1

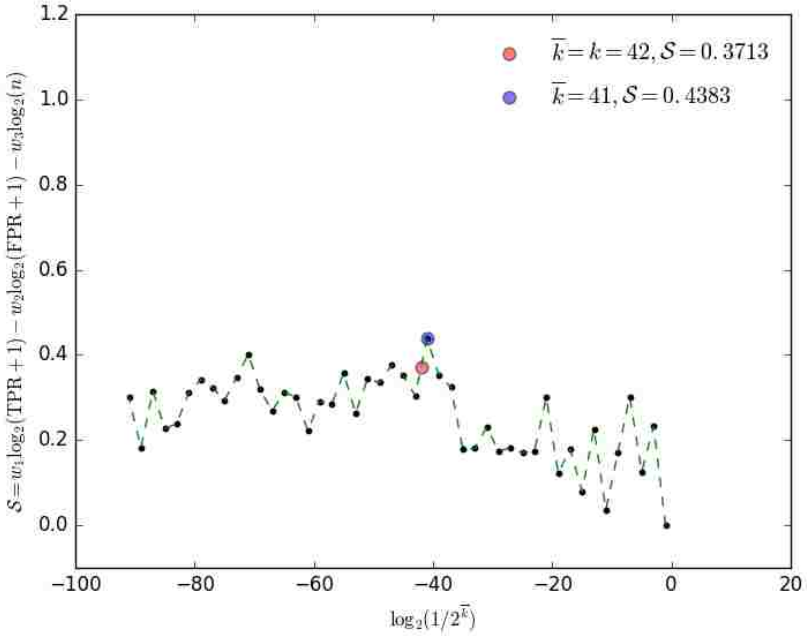


FIGURE A.73. Performance measure on *diff*, location main_ED, property 1

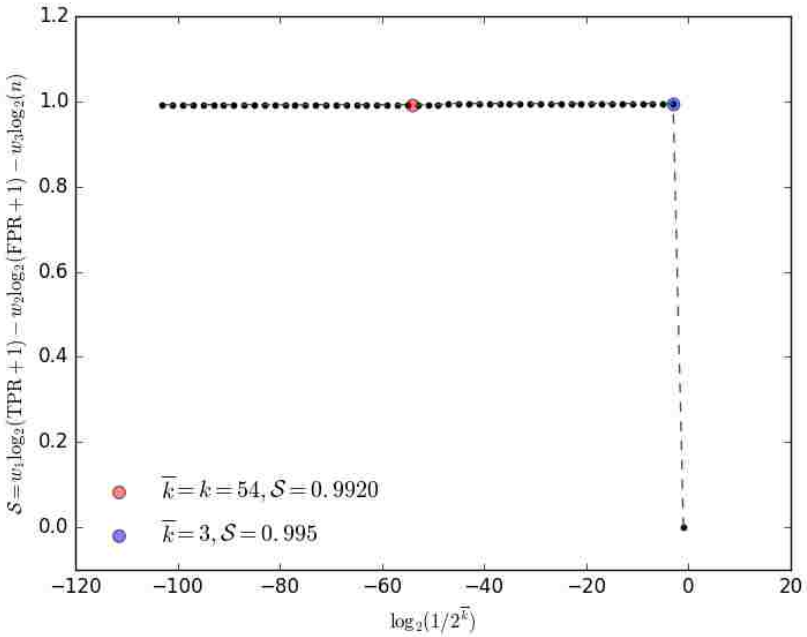


FIGURE A.74. Performance measure on *diff*, location main_BG, property 3

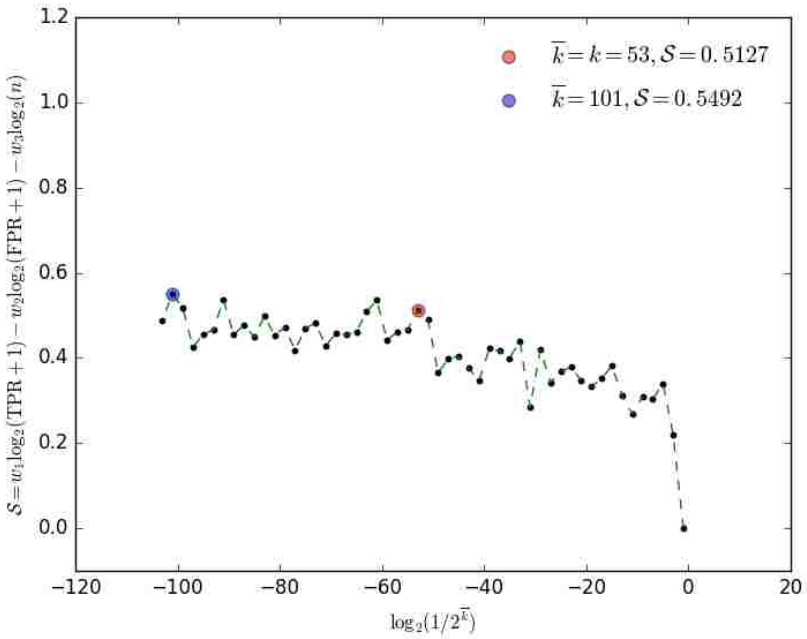


FIGURE A.75. Performance measure on *diff*, location main_DL, property 3

A.2.9 The Program *du*

TABLE A.10: Statistical results for program *du*

location	n	average m	φ_k: property 1 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AB	3	475	0.71	0.22	8	31	0.3614
main_R	45	226	0.78	0.2	3	48	0.5327
main_S	45	225	0.77	0.7	2	44	0.5267
location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AP	2	223	0.7	0.34	4	28	0.3388
main_R	45	226	0.97	0.14	3	50	0.76
main_S	45	225	0.98	0.12	2	44	0.7906
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AA	4	357	0.45	0.2	7	32	0.2625
main_R	44	233	0.81	0.05	3	27	0.7663
main_S	45	225	0.86	0.03	2	42	0.8289

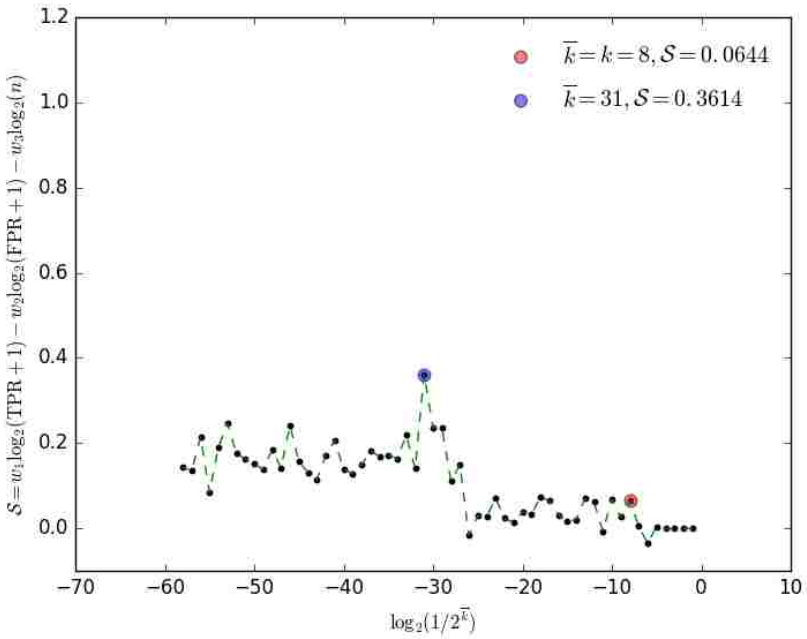


FIGURE A.76. Performance measure on *du*, location main_AB, property 1

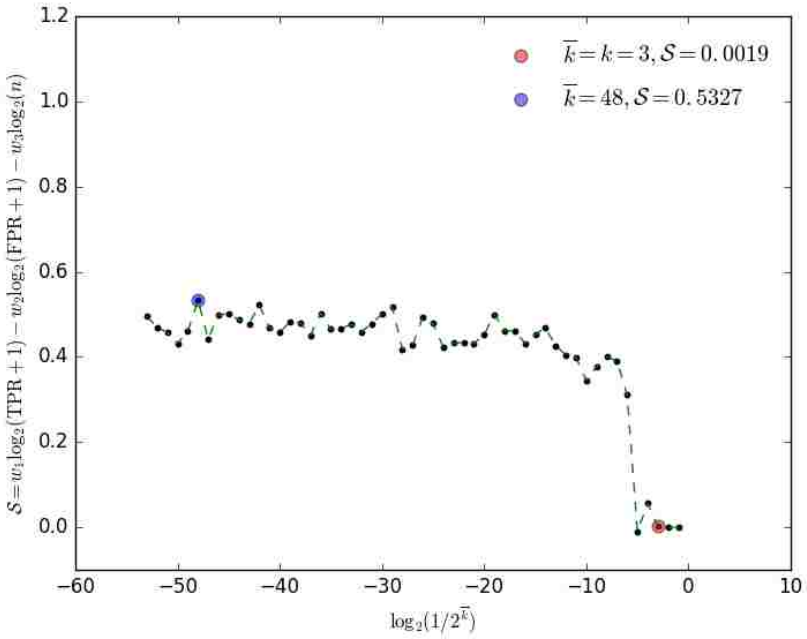


FIGURE A.77. Performance measure on *du*, location main_R, property 1

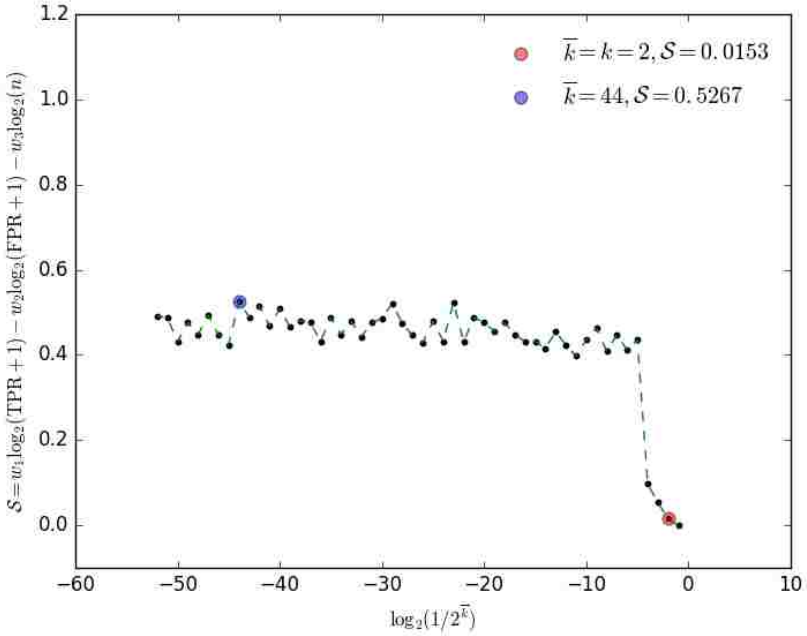


FIGURE A.78. Performance measure on *du*, location main_S, property 1

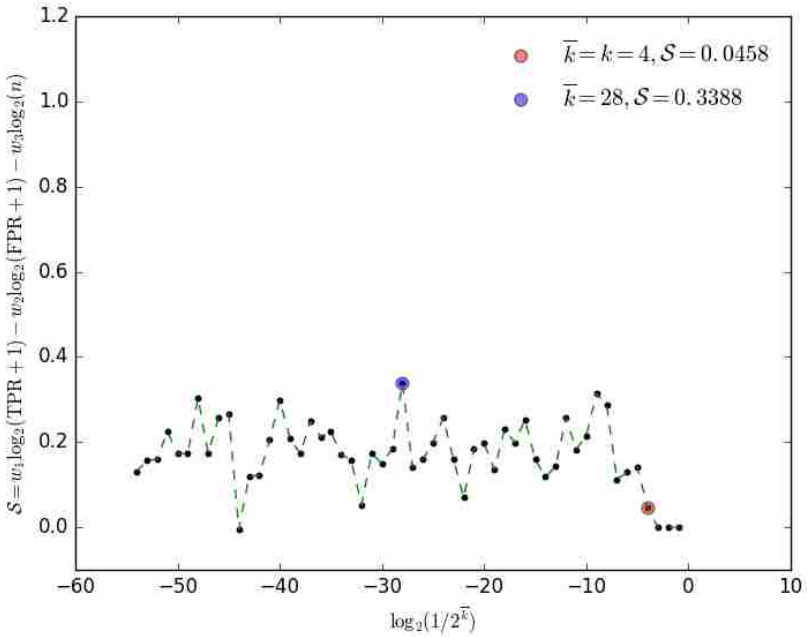


FIGURE A.79. Performance measure on *du*, location main_AP, property 2

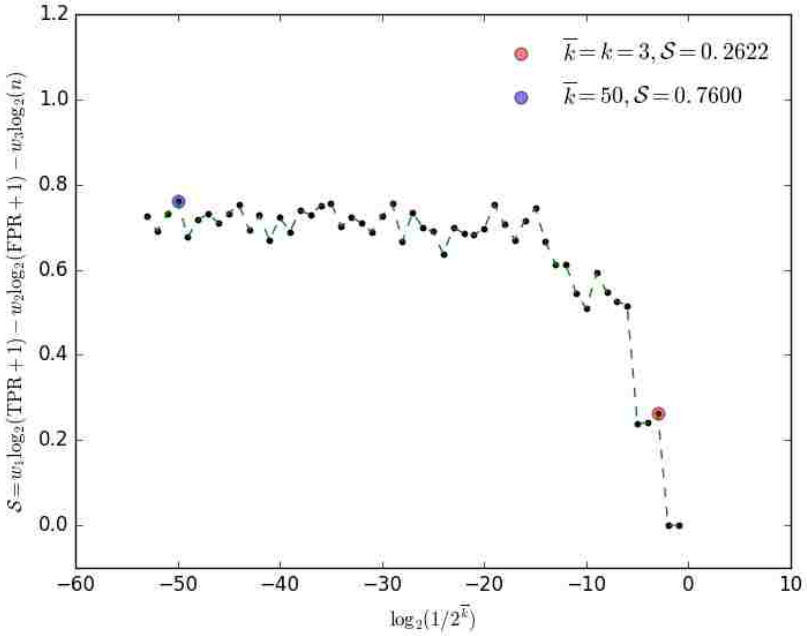


FIGURE A.80. Performance measure on *du*, location main_R, property 2

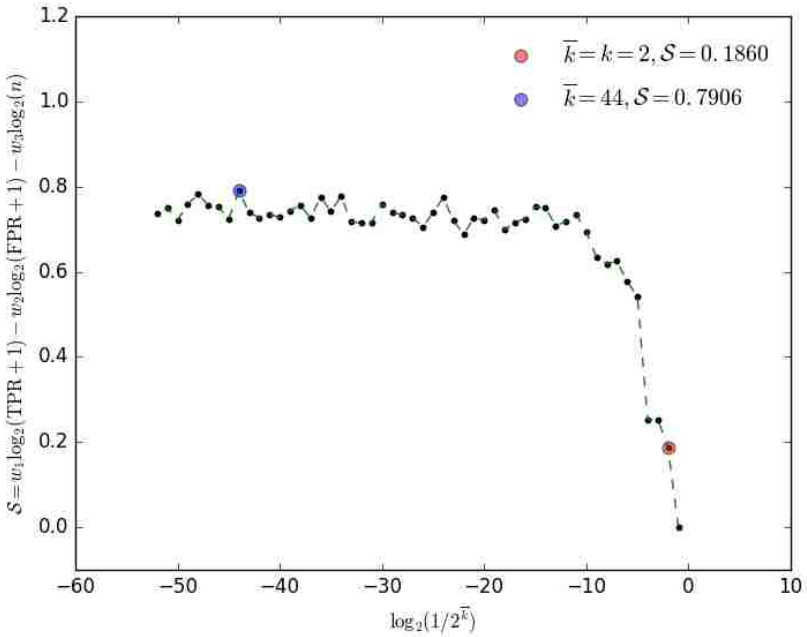


FIGURE A.81. Performance measure on *du*, location main_S, property 2

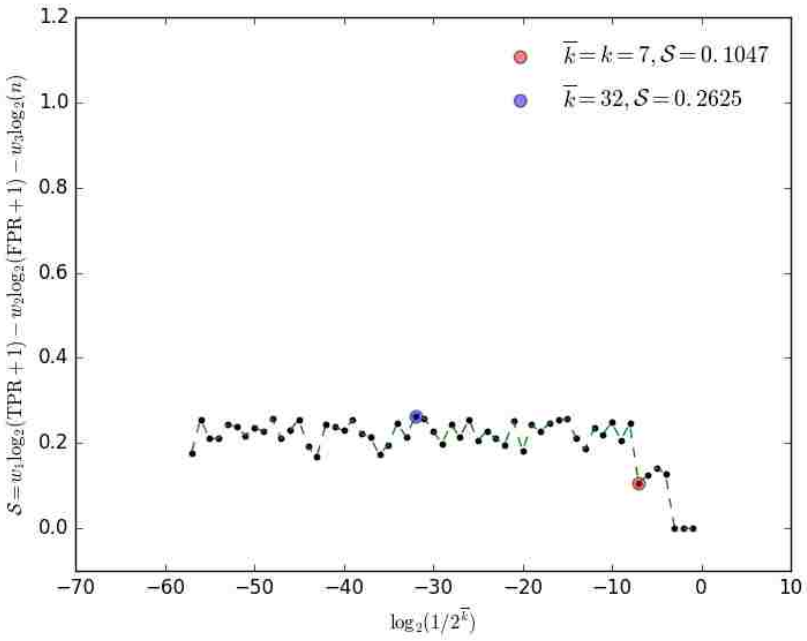


FIGURE A.82. Performance measure on *du*, location main_AA, property 3

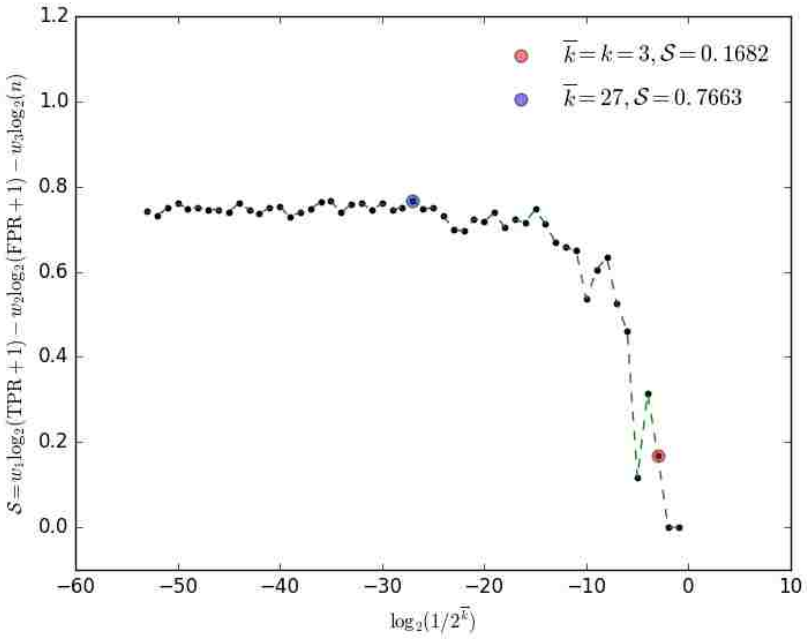


FIGURE A.83. Performance measure on *du*, location main_R, property 3

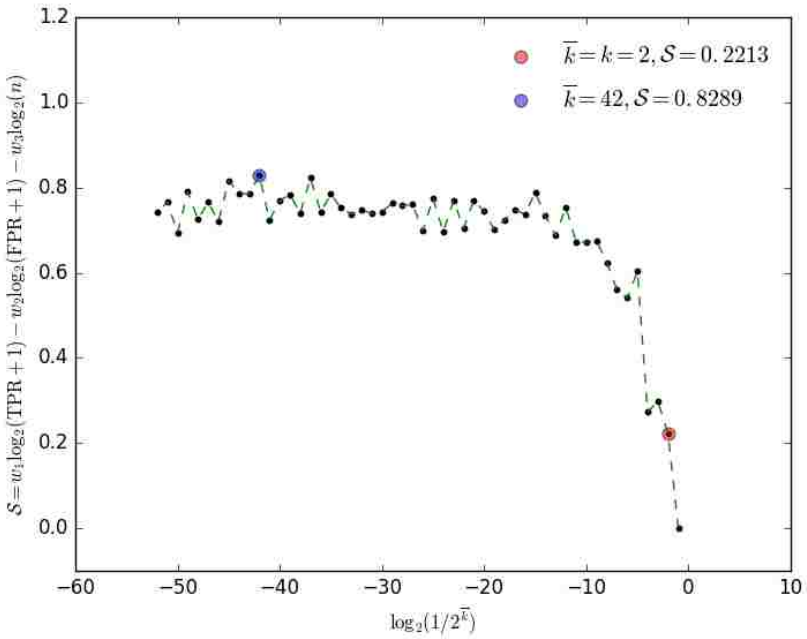


FIGURE A.84. Performance measure on *du*, location main_S, property 3

A.2.10 The Program *fmt*

TABLE A.11: Statistical results for program *fmt*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_R	9	158	1.0	0	4	4	0.9841
main_T	3	208	0.5	0.14	9	15	0.3886
main_U	4	156	0.54	0	2	12	0.6135
location	n	average m	φ_k: property 2 TPR	φ_k: property 2 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AB	2	268	0.9	0.24	6	48	0.6093
main_U	4	156	1.0	0.22	2	15	0.7032
main_Z	3	208	1.0	0.23	1	4	0.6927
location	n	average m	φ_k: property 3 TPR	φ_k: property 3 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_V	3	178	1.0	0.31	7	43	0.6009

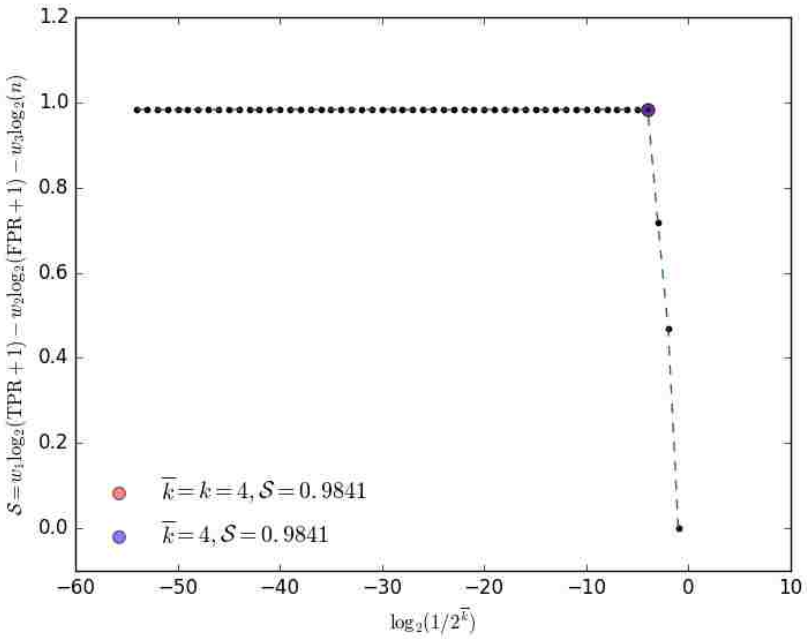


FIGURE A.85. Performance measure on *fmt*, location main_R, property 1

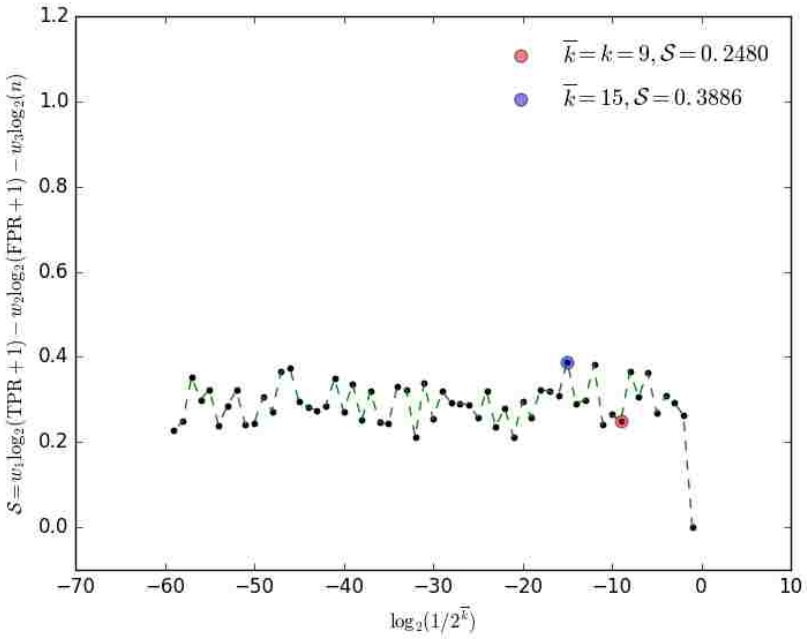


FIGURE A.86. Performance measure on *fmt*, location main_T, property 1

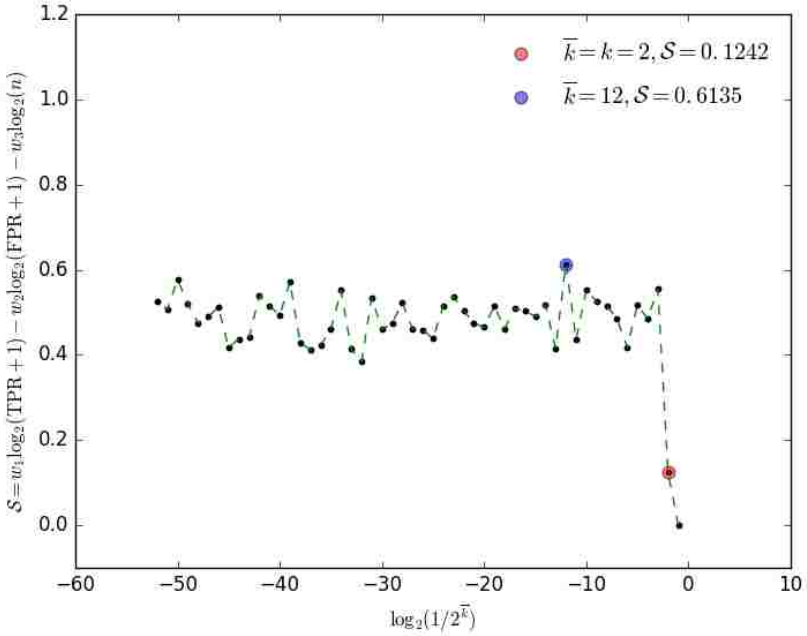


FIGURE A.87. Performance measure on *fmt*, location main_U, property 1

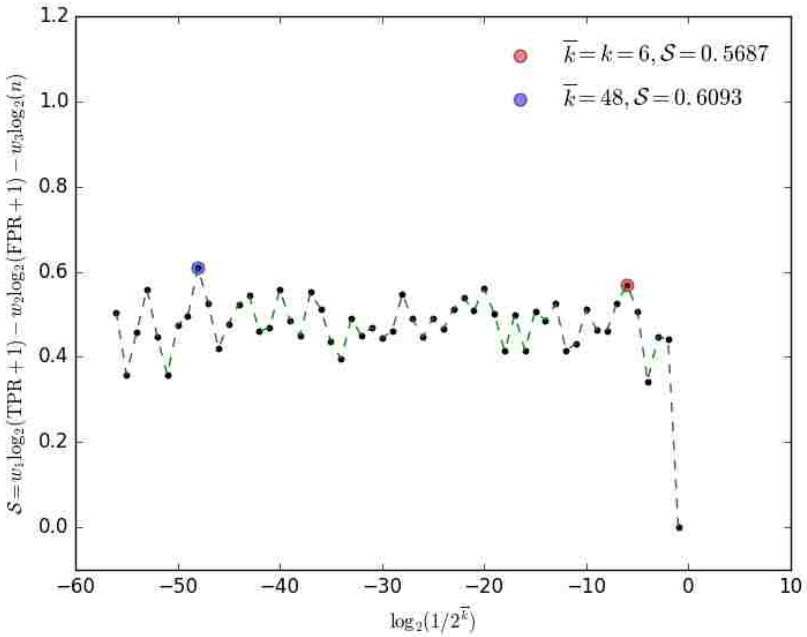


FIGURE A.88. Performance measure on *fmt*, location main_AB, property 2

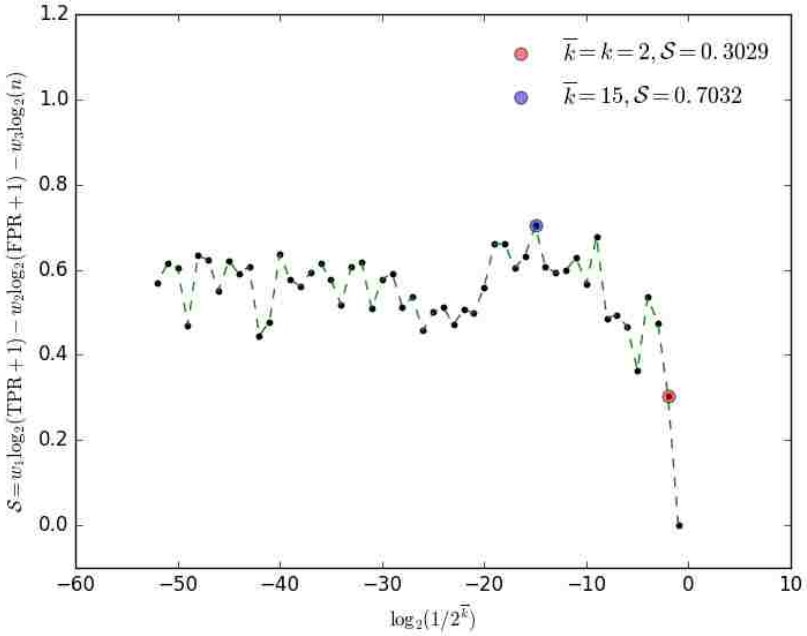


FIGURE A.89. Performance measure on *fmt*, location main_U, property 2

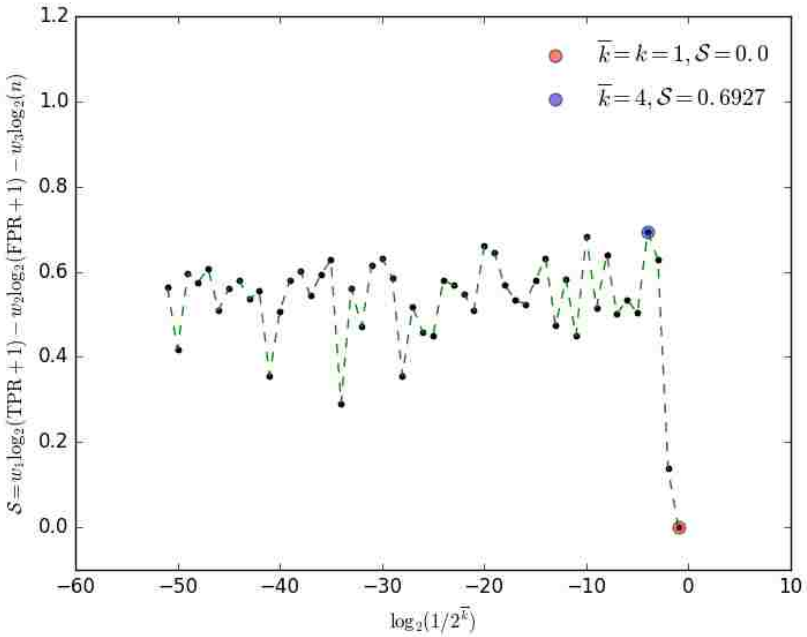


FIGURE A.90. Performance measure on *fmt*, location main_Z, property 2

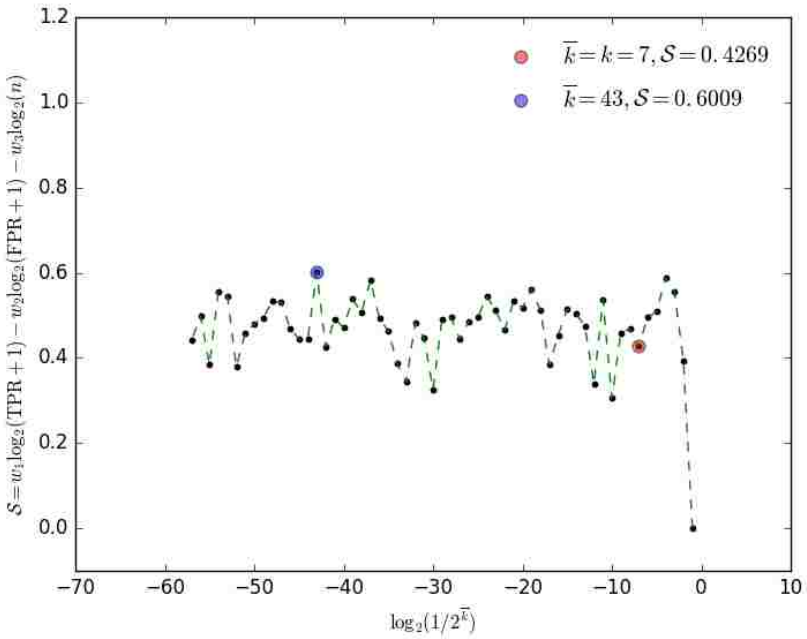


FIGURE A.91. Performance measure on *fmt*, location main_V, property 3

A.2.11 The Program *fold*

TABLE A.12: Statistical results for program *fold*

location	n	average m	φ_k: property 1 TPR	φ_k: property 1 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_A	7	204	0.91	0.07	26	28	0.8187
main_AB	11	229	0.91	0.04	13	41	0.8676
main_BB	5	214	0.81	0.08	29	25	0.7328
main_F	4	357	0.78	0.02	24	17	0.8008
main_I	3	475	0.86	0.09	21	8	0.7671
main_M	68	202	0.91	0.12	11	59	0.7341
location	n	average m	φ_k: property 2 TPR	φ_k: property 2 FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AA	8	312	0.65	0.21	14	34	0.4317
main_H	57	222	0.58	0.23	9	39	0.3289

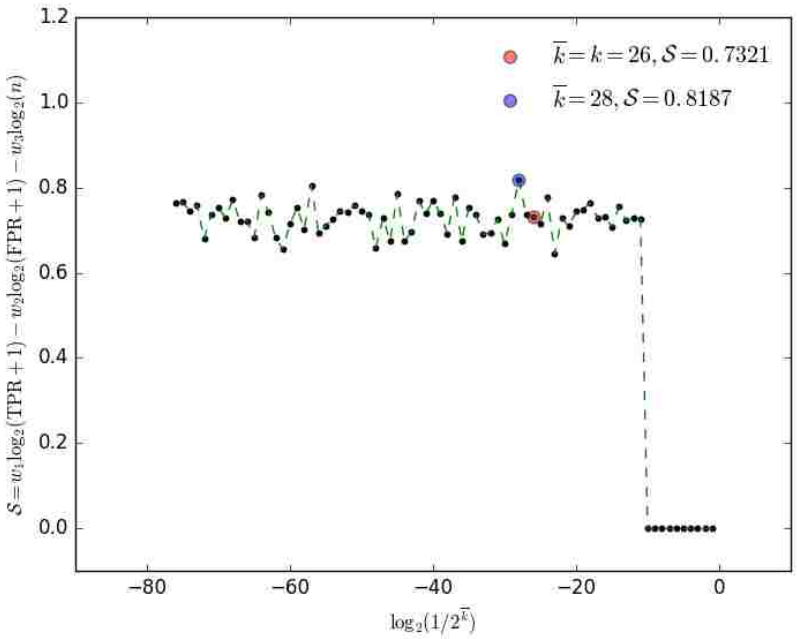


FIGURE A.92. Performance measure on *fold*, location main_A, property 1

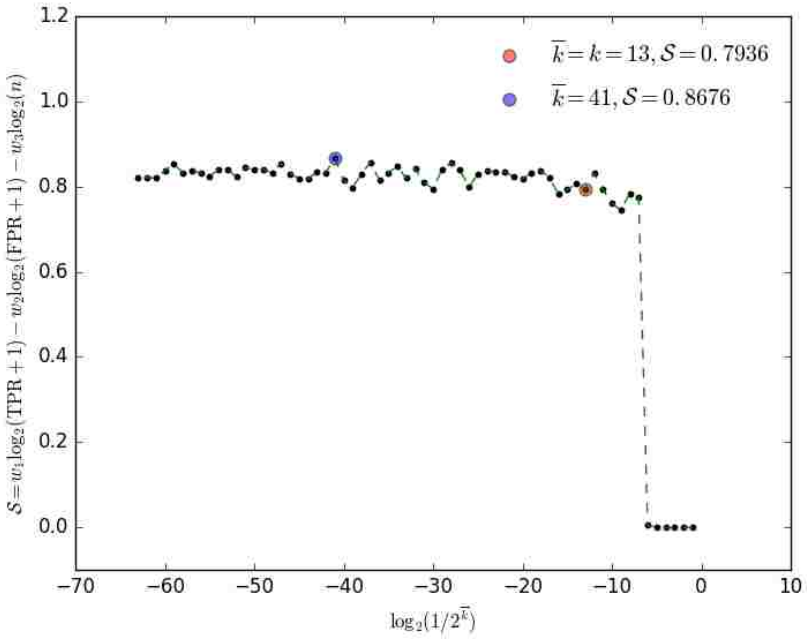


FIGURE A.93. Performance measure on *fold*, location main_AB, property 1

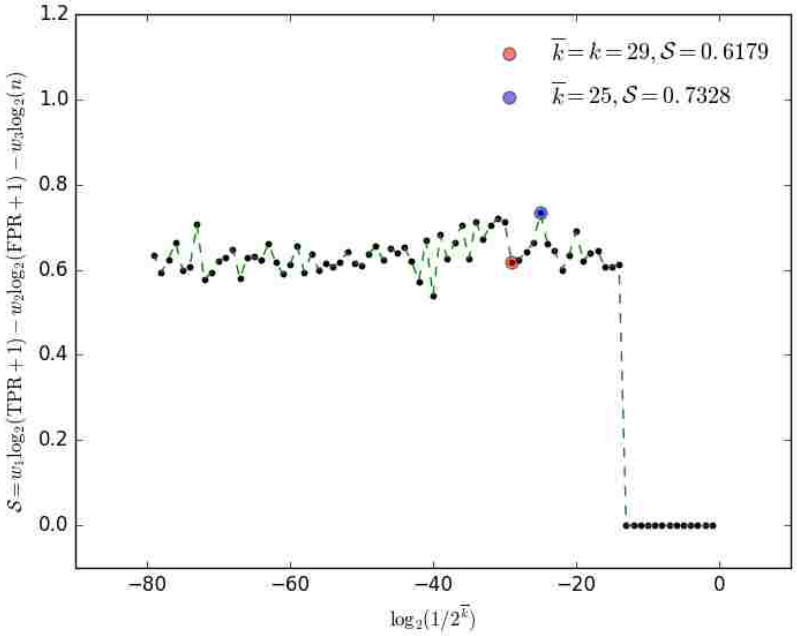


FIGURE A.94. Performance measure on *fold*, location main_BB, property 1

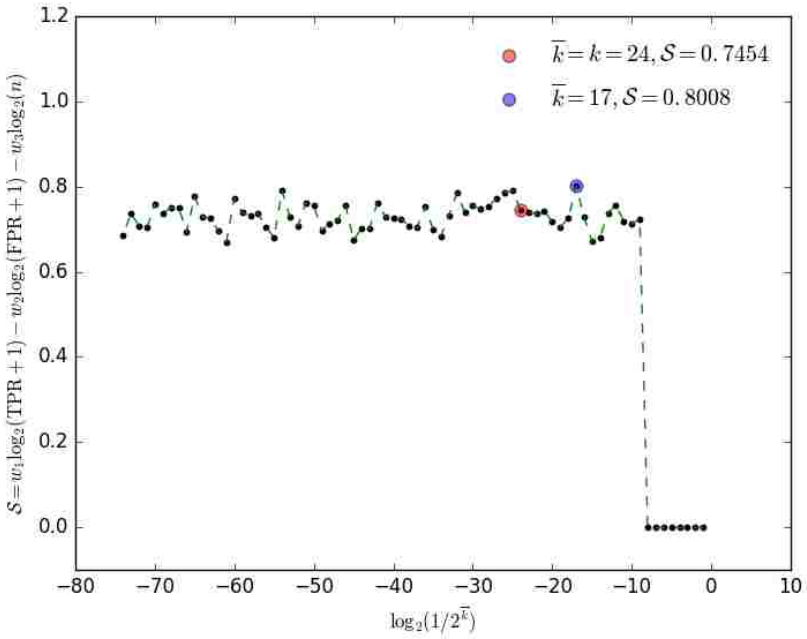


FIGURE A.95. Performance measure on *fold*, location main_F, property 1

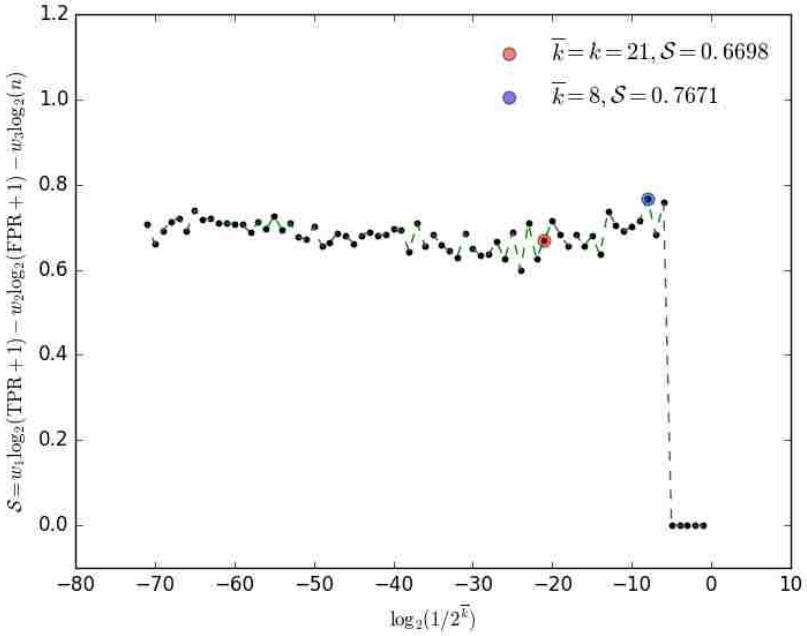


FIGURE A.96. Performance measure on *fold*, location main_I, property 1

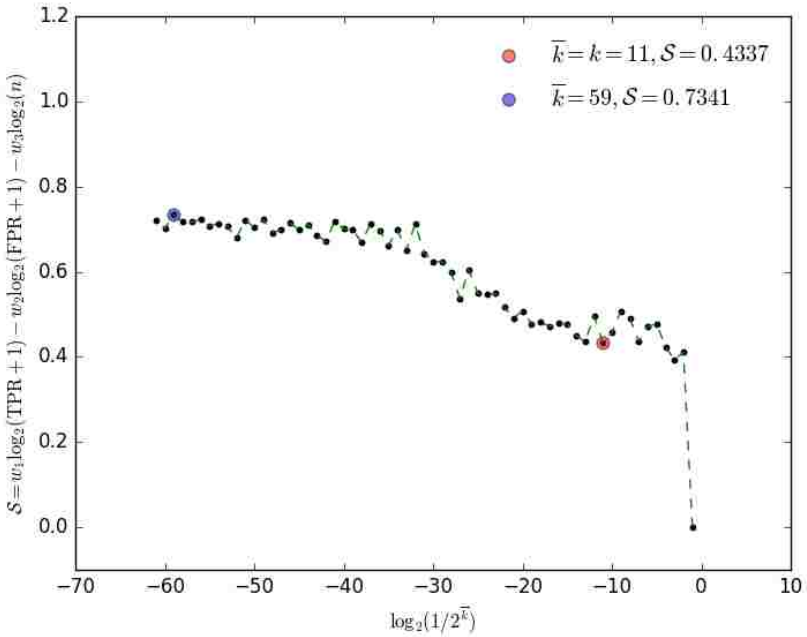


FIGURE A.97. Performance measure on *fold*, location main_M, property 1

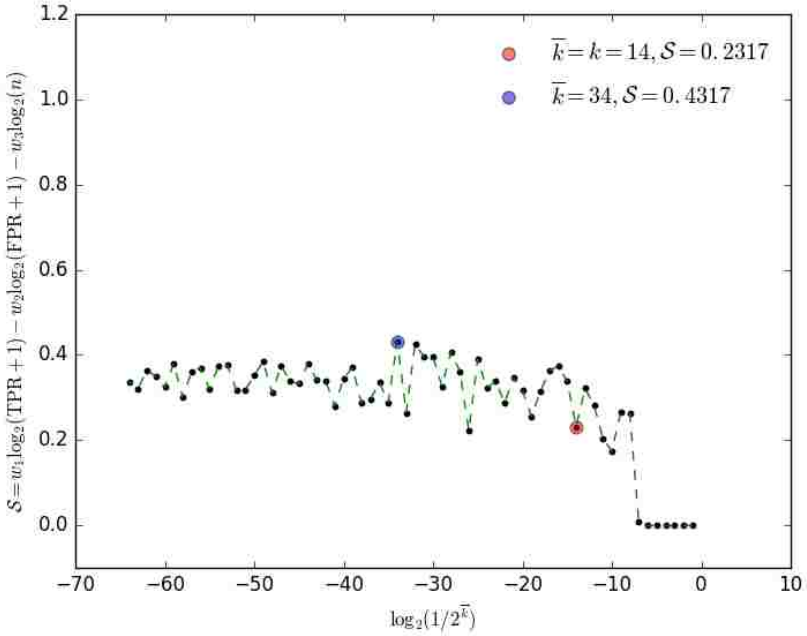


FIGURE A.98. Performance measure on *fold*, location main_AA, property 2

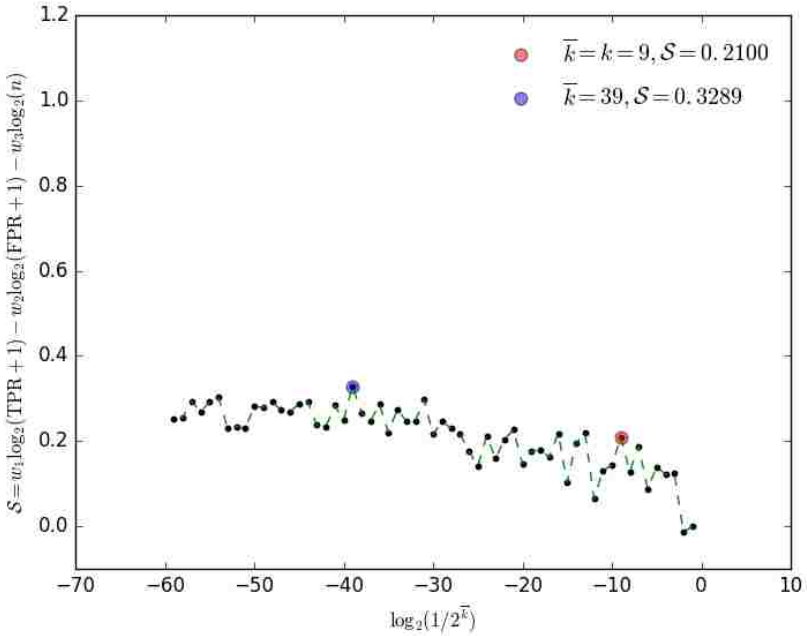


FIGURE A.99. Performance measure on *fold*, location main_H, property 2

A.2.12 The Program *grep*

TABLE A.13: Statistical results for program *grep*

location	n	average m	φ_k: property 1 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_CN	2	223	0.47	0.17	63	37	0.3241
main_CQ	3	178	1.0	0	62	61	0.992
location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_CO	5	178	1.0	0	63	63	0.9883
main_CP	5	178	1.0	0	62	61	0.9883
main_CQ	3	178	1.0	0	62	61	0.992
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AAO	4	646	0.86	0.37	19	22	0.4265
main_ABH	5	517	0.81	0.32	10	23	0.4476
main_CP	7	127	0.88	0.08	62	67	0.7794
main_GE	5	517	0.81	0.36	29	33	0.403

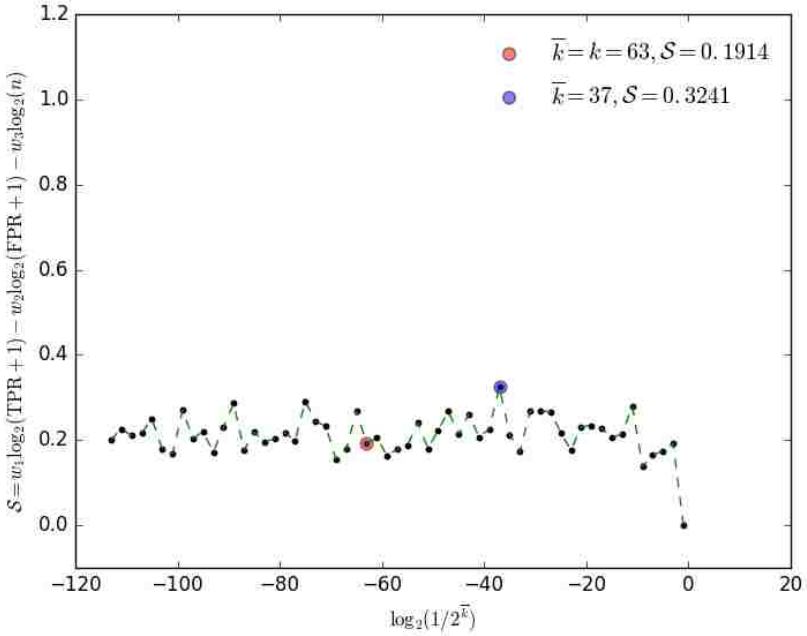


FIGURE A.100. Performance measure on *grep*, location main_CN, property 1

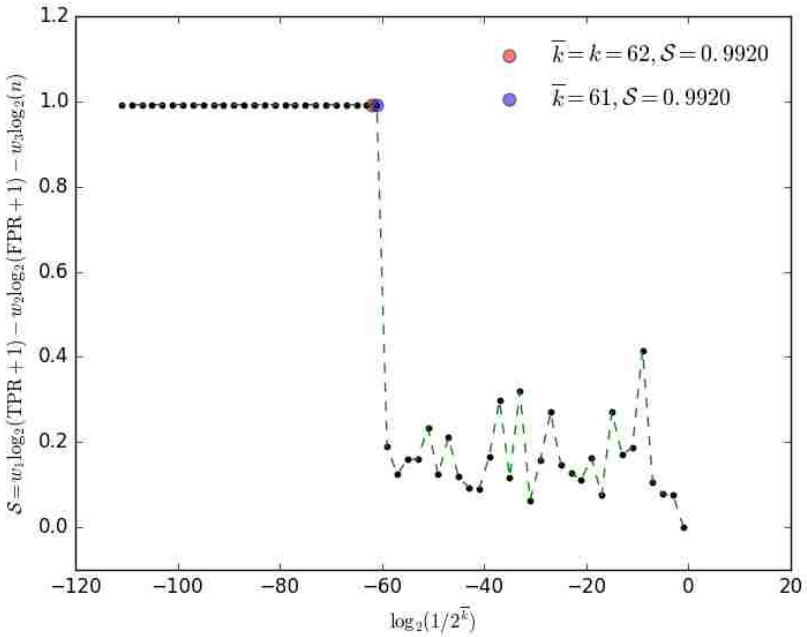


FIGURE A.101. Performance measure on *grep*, location main_CQ, property 1

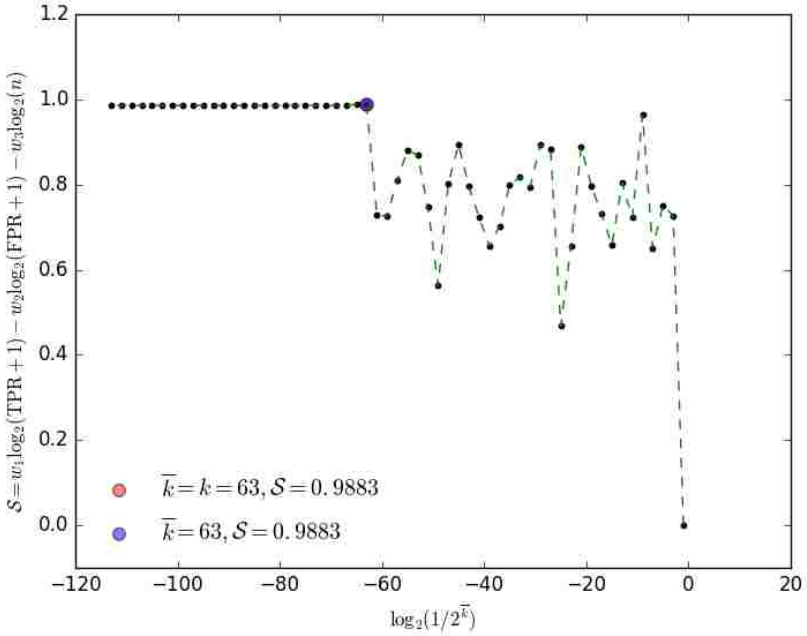


FIGURE A.102. Performance measure on *grep*, location main_CO, property 2

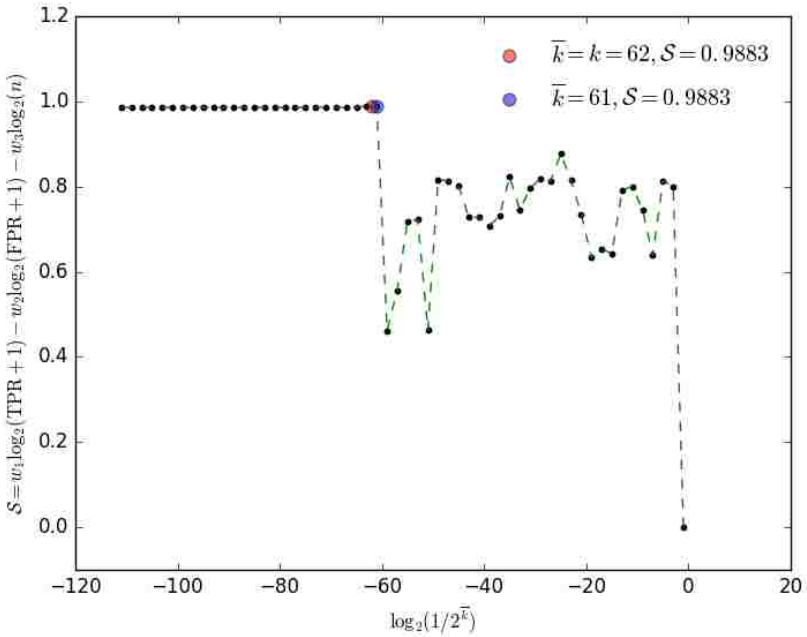


FIGURE A.103. Performance measure on *grep*, location main_CP, property 2

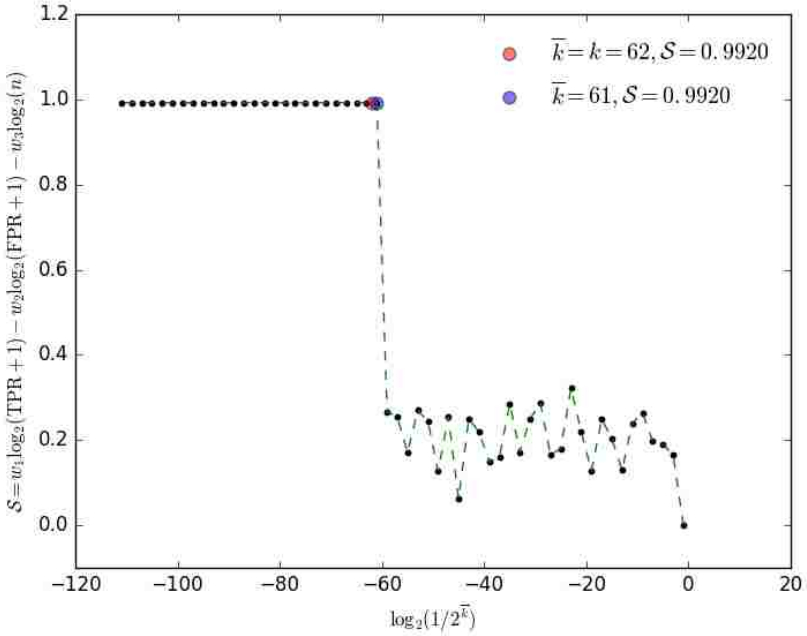


FIGURE A.104. Performance measure on *grep*, location main_CQ, property 2

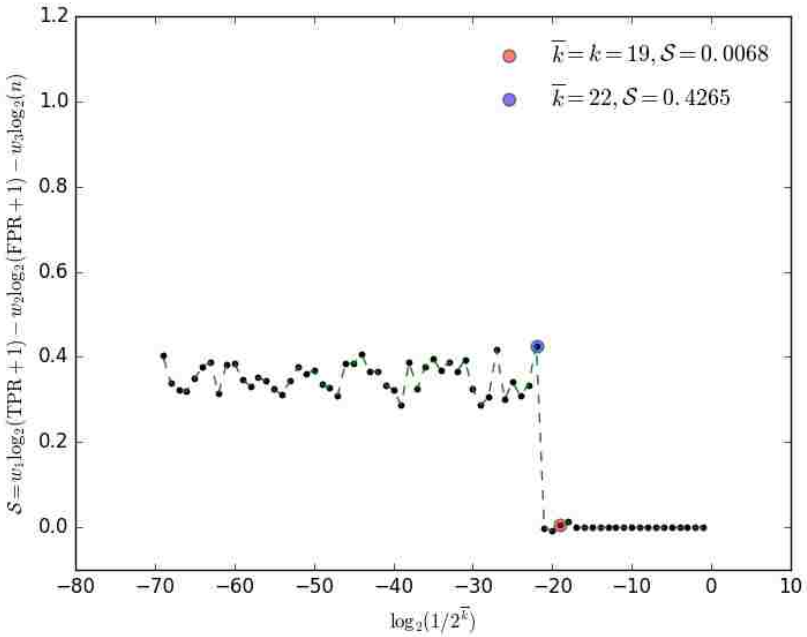


FIGURE A.105. Performance measure on *grep*, location main_AAO, property 3

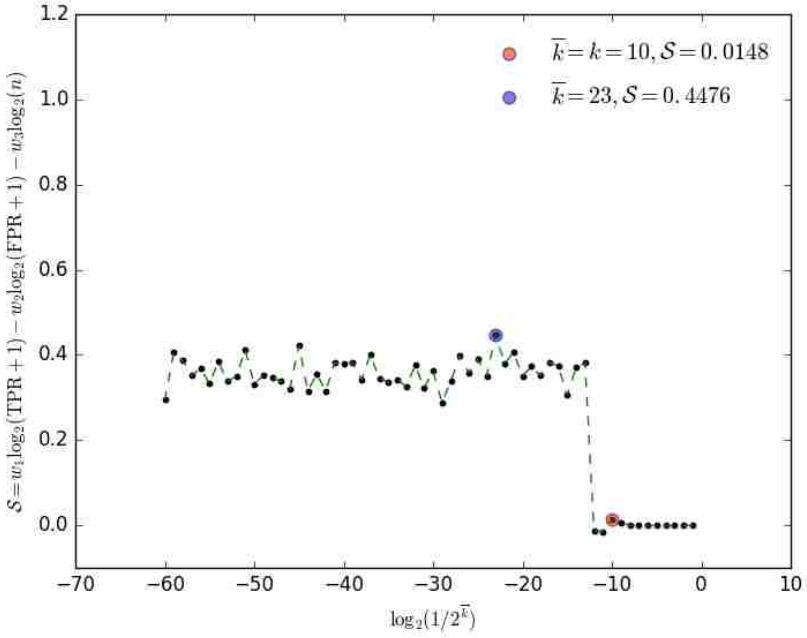


FIGURE A.106. Performance measure on *grep*, location main_ABH, property 3

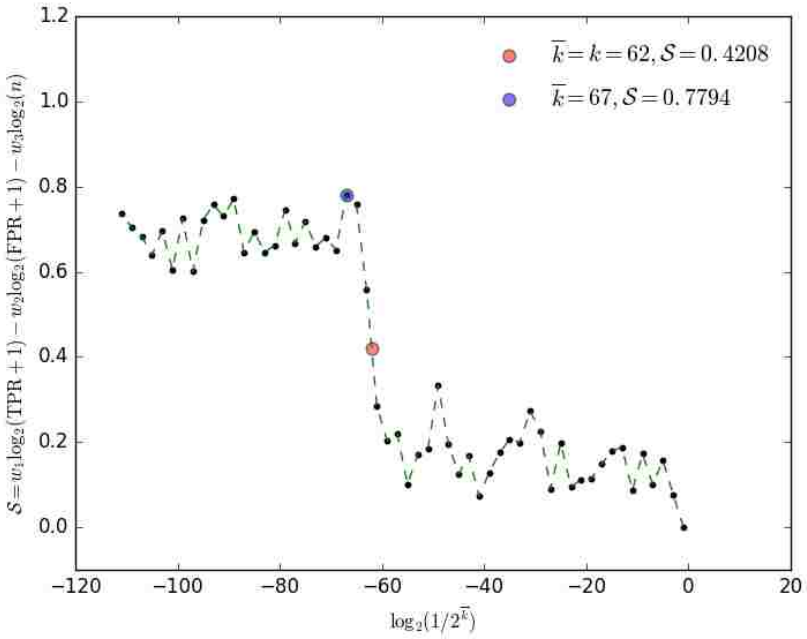


FIGURE A.107. Performance measure on *grep*, location main_CP, property 3

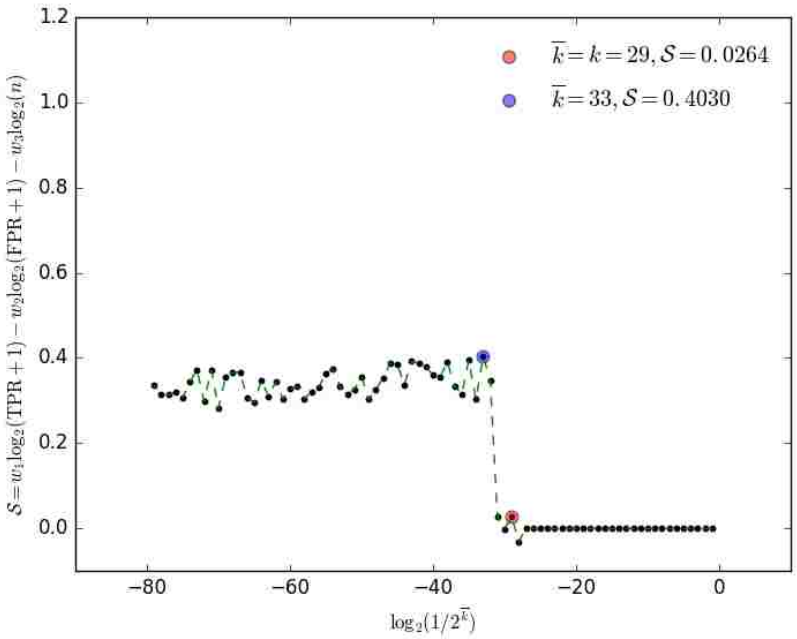


FIGURE A.108. Performance measure on *grep*, location main_GE, property 3

A.2.13 The Program *truncate*

TABLE A.14: Statistical results for program *truncate*

location	n	average m	φ_k: property 1 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_BA	2	535	1.0	0	10	9	0.995
main_CE	2	624	1.0	0	5	4	0.995
location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_AG	10	276	0.98	0.27	5	6	0.6262
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_BF	4	379	1.0	0	3	3	0.99

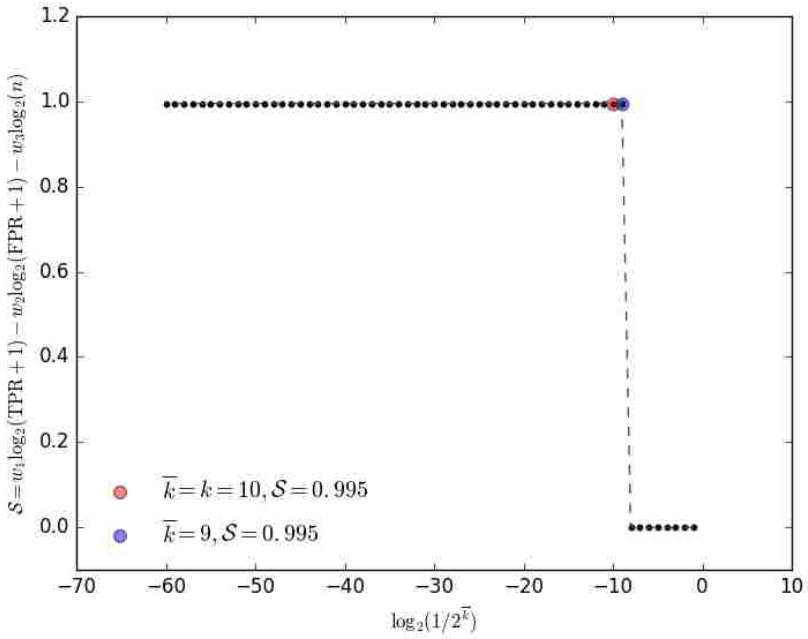


FIGURE A.109. Performance measure on *truncate*, location main_BA, property 1

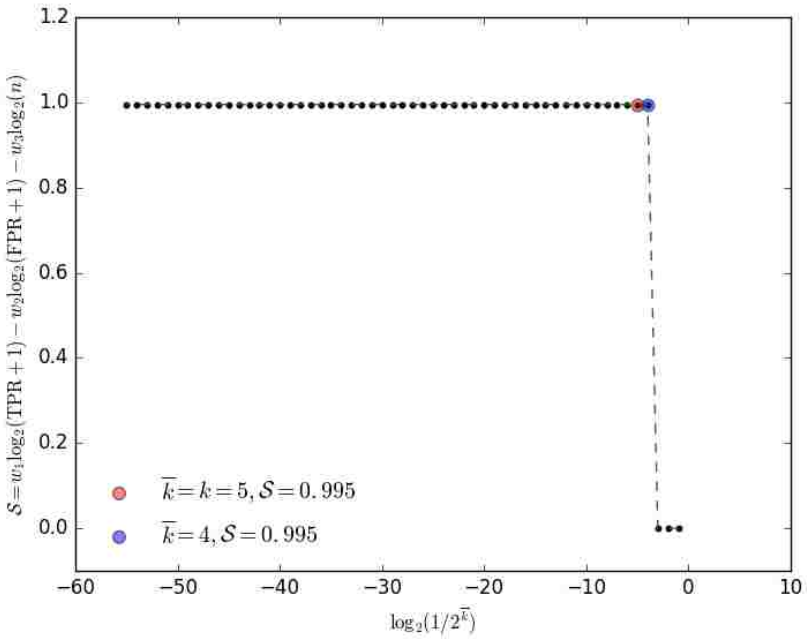


FIGURE A.110. Performance measure on *truncate*, location main_CE, property 1

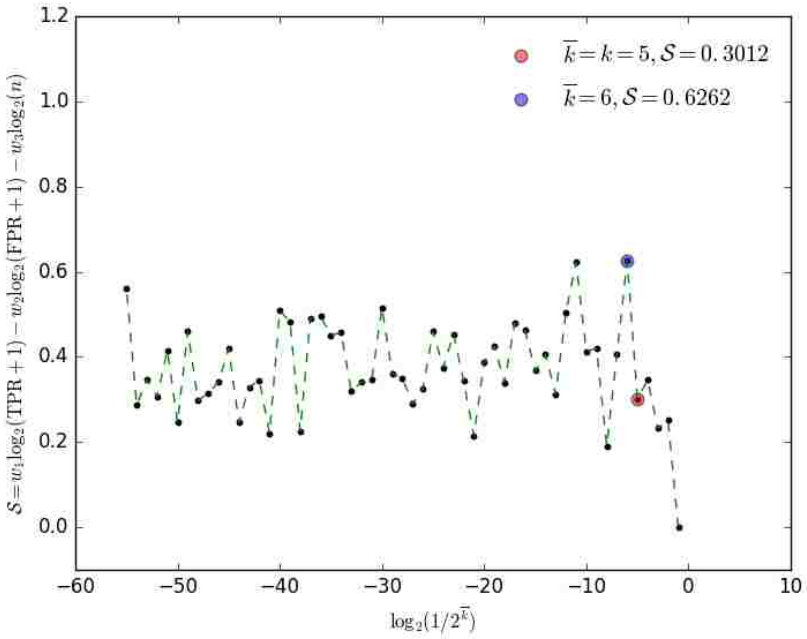


FIGURE A.111. Performance measure on *truncate*, location main_AG, property 2

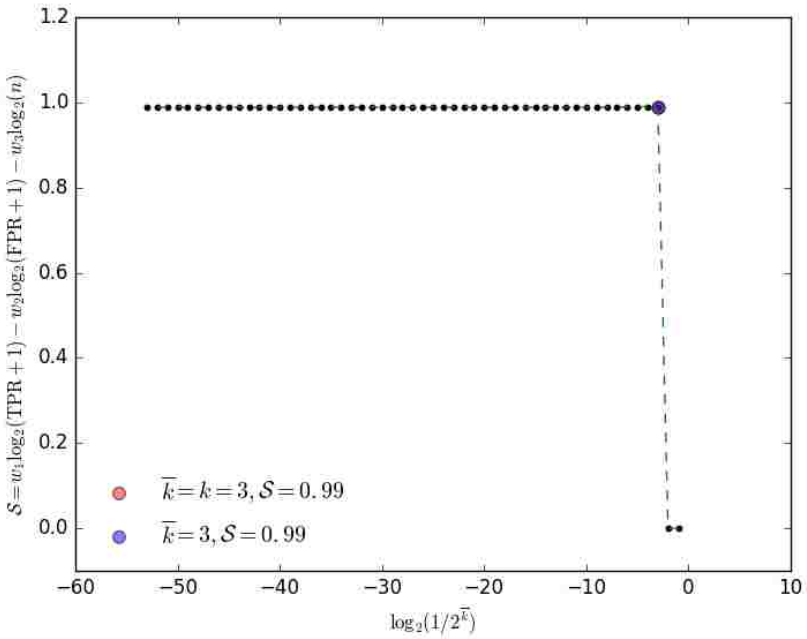


FIGURE A.112. Performance measure on *truncate*, location main_BF, property 3

A.2.14 The Program *wc*

TABLE A.15: Statistical results for program *wc*

location	n	average m	φ_k: property 2 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_B	4	1248	0.66	0	16	15	0.7191
main_C	4	1248	0.67	0	15	12	0.7247
main_CD	8	624	0.62	0.12	8	16	0.5223
main_DN	8	468	0.55	0	31	27	0.6168
main_EU	4	1248	0.66	0	19	20	0.7217
main_FD	6	816	0.65	0	18	28	0.7120
location	n	average m	φ_k: property 3 TPR	FPR	k	optimal $\overline{k}$	optimal $\mathcal{S}$
main_DL	12	468	0.58	0.17	32	47	0.4221
main_DO	6	253	0.67	0.26	33	53	0.3841

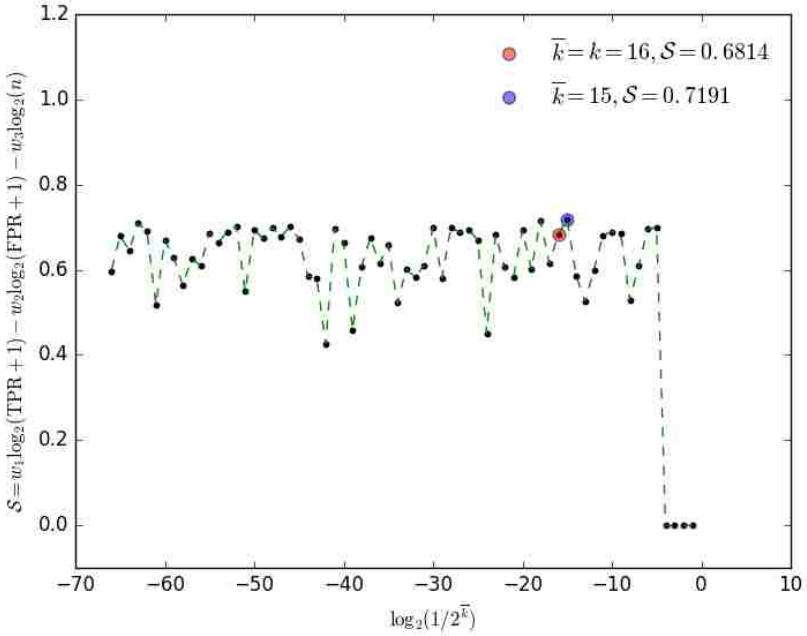


FIGURE A.113. Performance measure on *wc*, location main_B, property 2

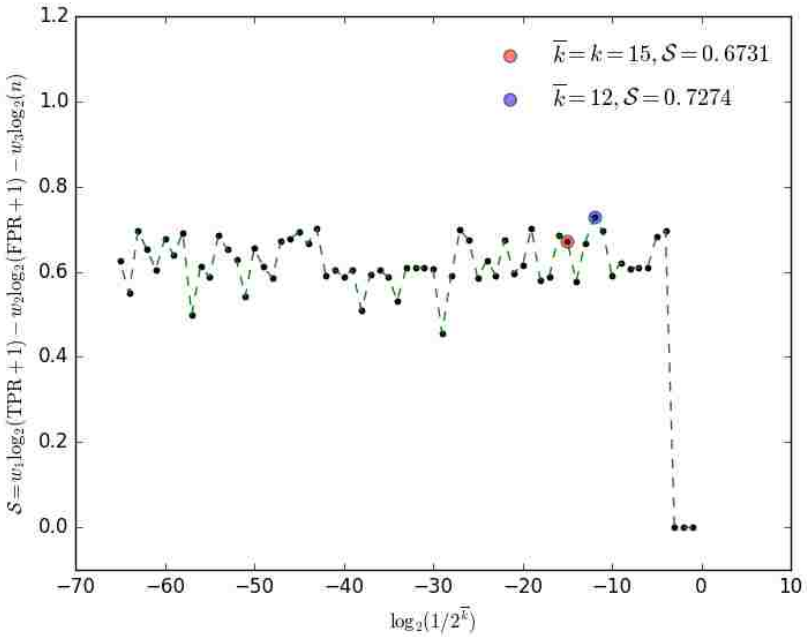


FIGURE A.114. Performance measure on *wc*, location main_C, property 2

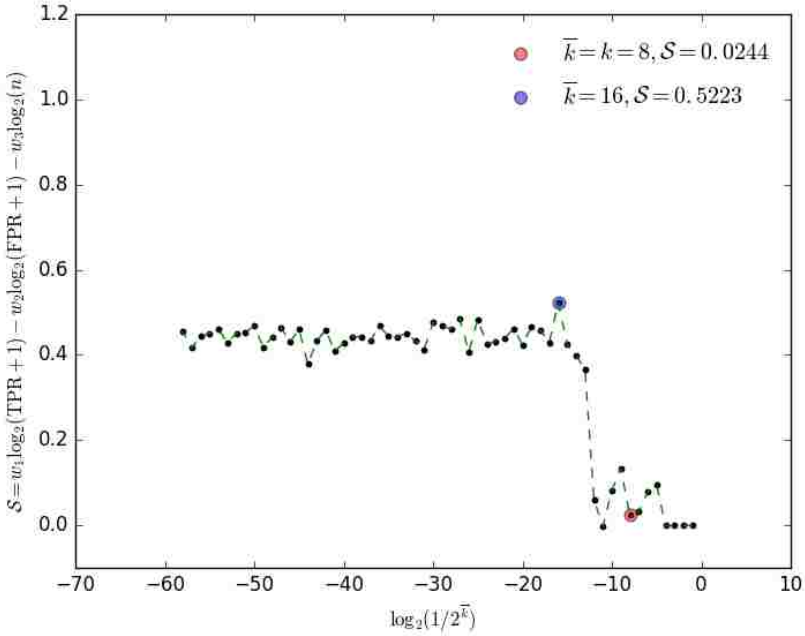


FIGURE A.115. Performance measure on *wc*, location main_CD, property 2

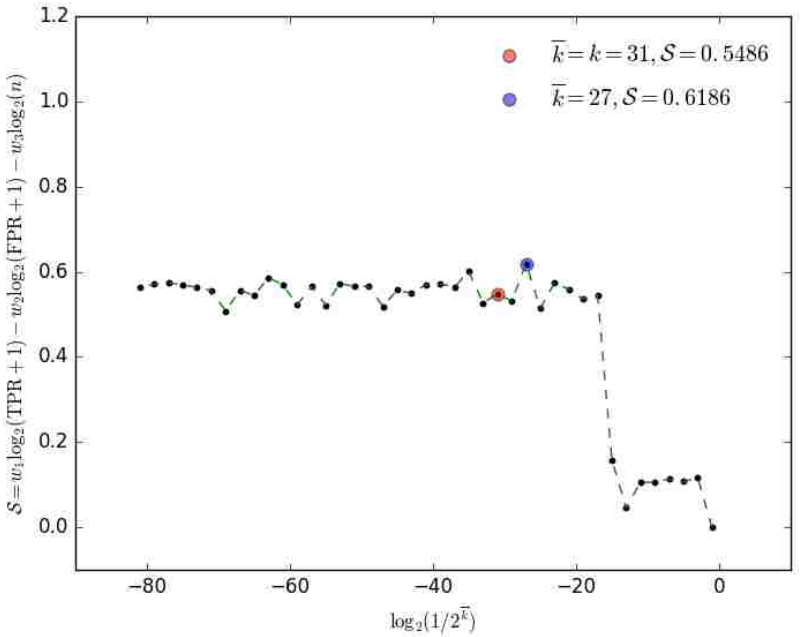


FIGURE A.116. Performance measure on *wc*, location main_DN, property 2

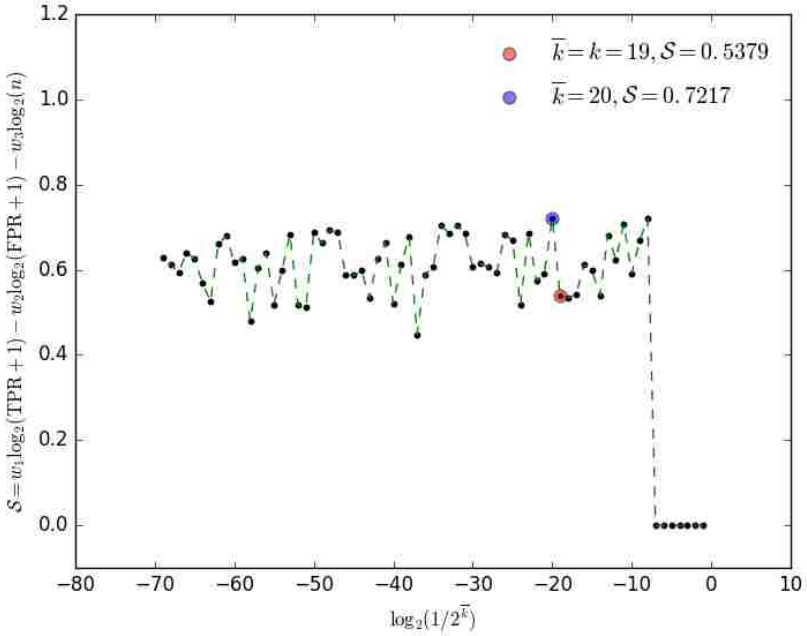


FIGURE A.117. Performance measure on *wc*, location main_EU, property 2

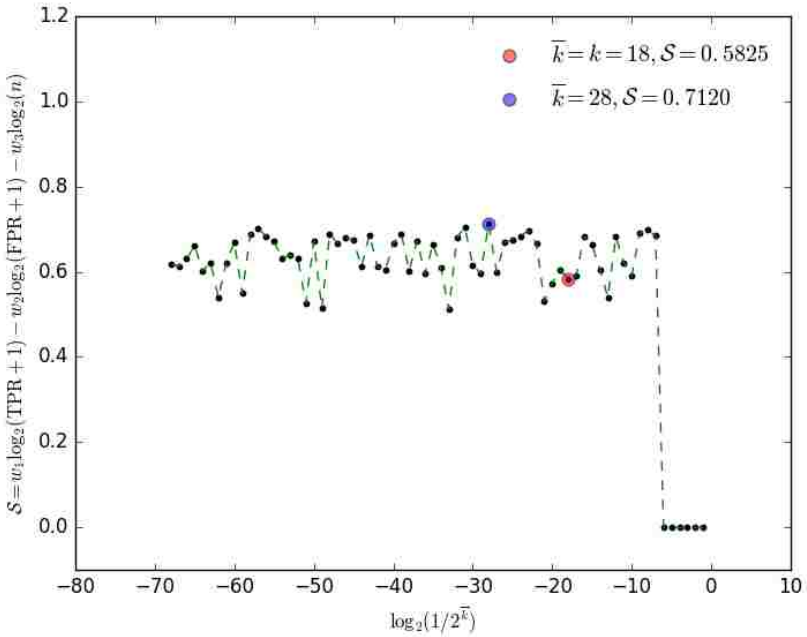


FIGURE A.118. Performance measure on *wc*, location main_FD, property 2

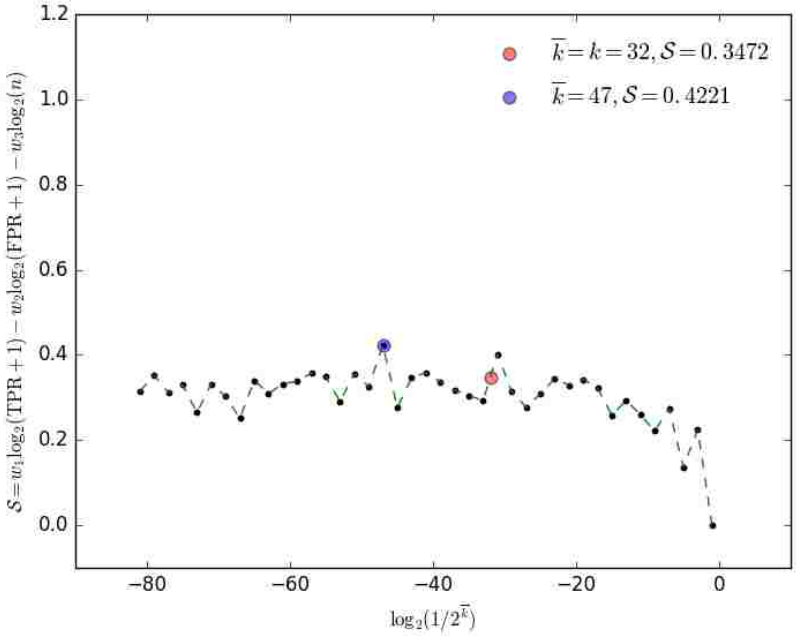


FIGURE A.119. Performance measure on *wc*, location main_DL, property 3

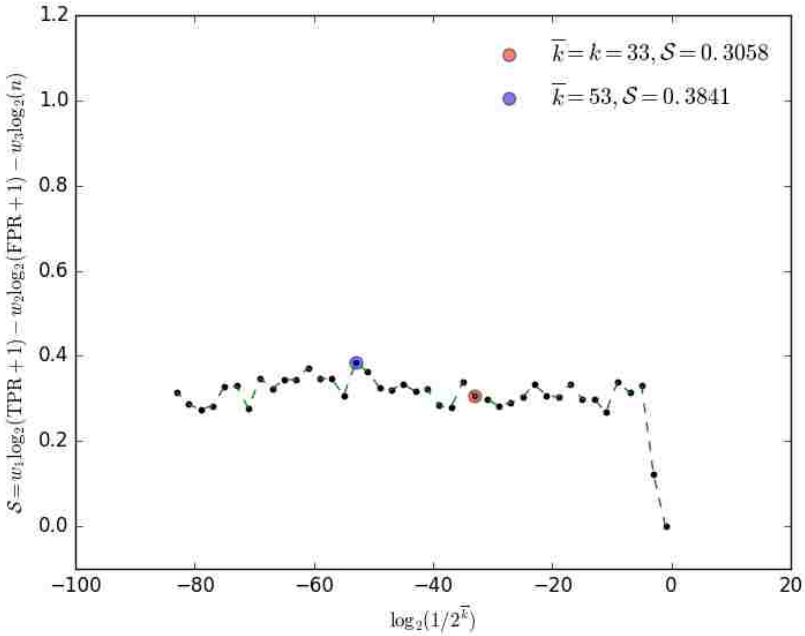


FIGURE A.120. Performance measure on *wc*, location main_DO, property 3

A.3 Plots of Run Time

To close the entire work, we present here a summary about the run time of data-driven abstraction approach on the chosen C programs.

We provide line plots of run time versus each observed point/location in one program. Precisely, we sorted run time for applying data-driven abstraction (DDA) approach on sample data drawn in each observed location from low to high, and plotted them in as y-values. The x-values then correspond to the indices of code locations. These plots give us a general map of how time complexity of DDA varied over sample data of different sizes randomly drawn from a program. For example, in the program *wc.c*, the traces traveled along loops multiple times (more frequently than in other programs), that resulted in more states drawn at some locations. Therefore, the Run time for applying DDA approach over those locations was not surprisingly longer than others, as we can see from the plots shown below.

Moreover, we calculated a 95% confidence interval of run time from sample data. That means in the future when we run the same clustering algorithm with the same input setting, it is 95% chances to take the same time as shown in the graphs. We can freely change this factor in order to meet different needs.

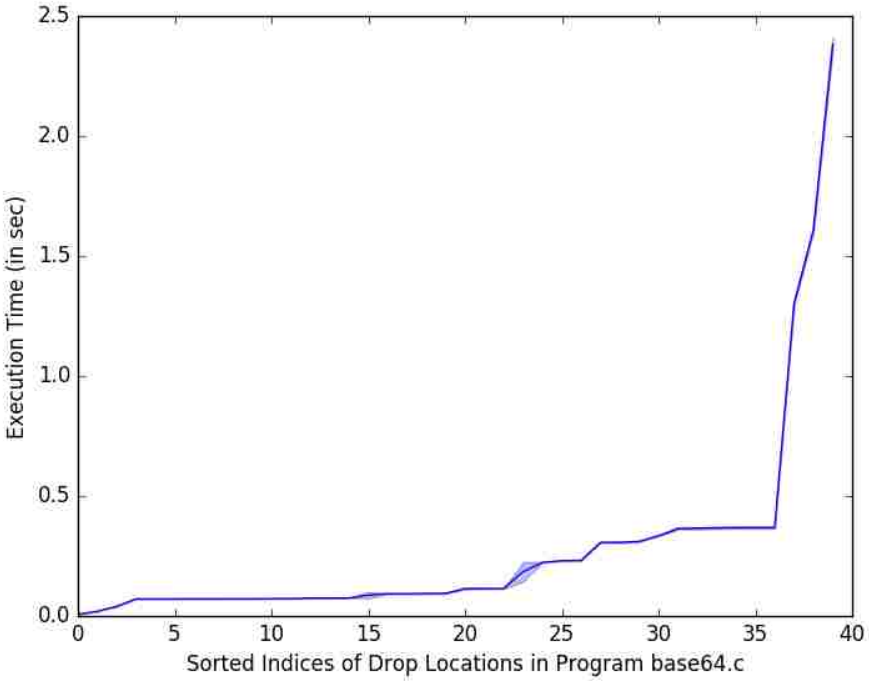


FIGURE A.121. Execution time of DDA on *base64*

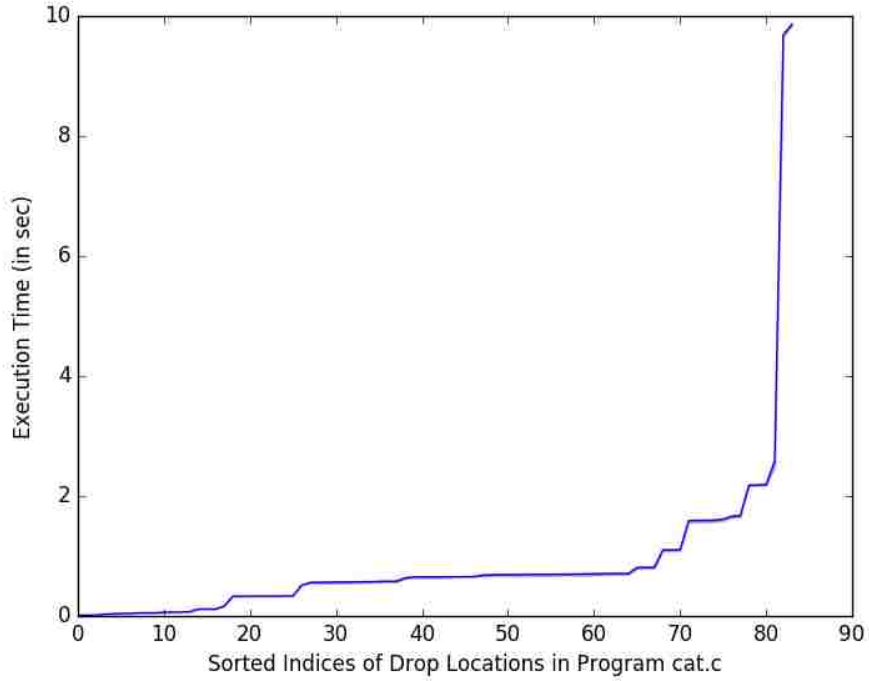


FIGURE A.122. Execution time of DDA on *cat*

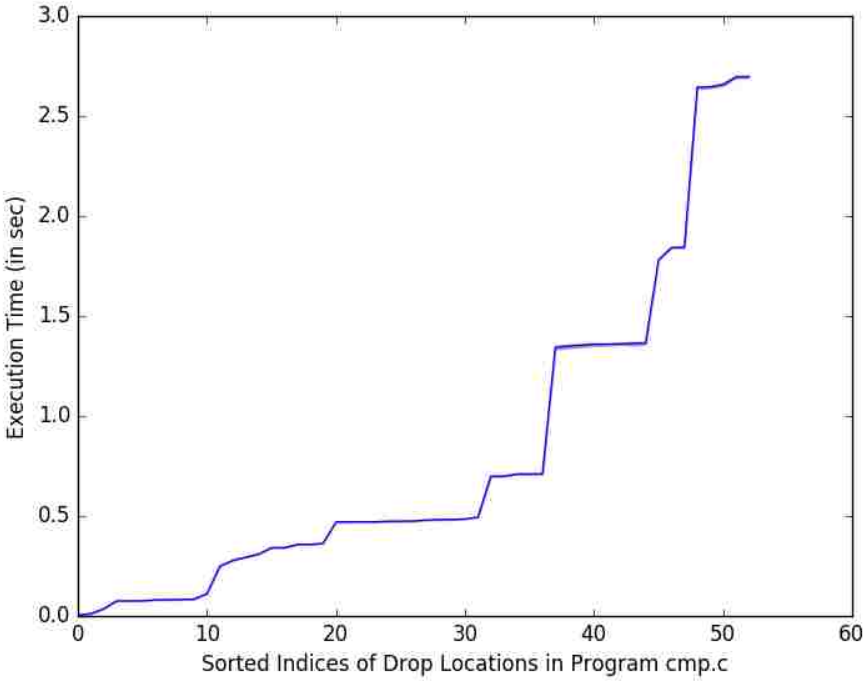


FIGURE A.123. Execution time of DDA on *cmp*

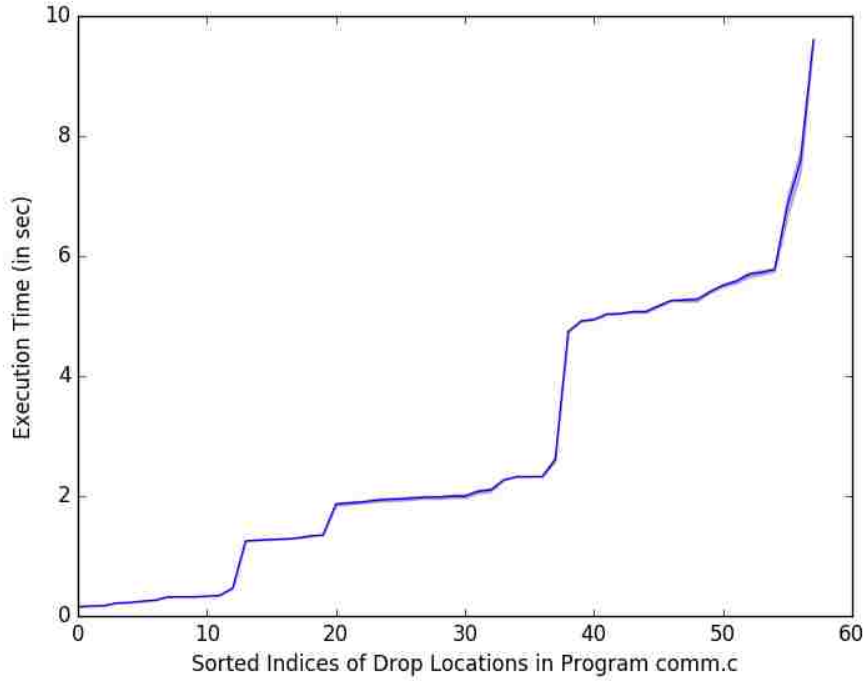


FIGURE A.124. Execution time of DDA on *comm*

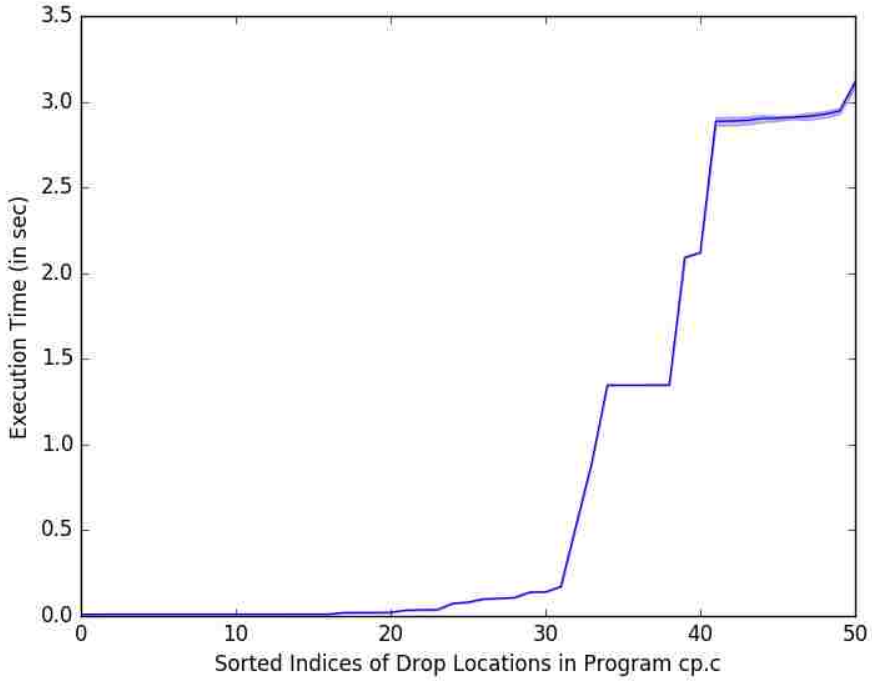


FIGURE A.125. Execution time of DDA on *cp*

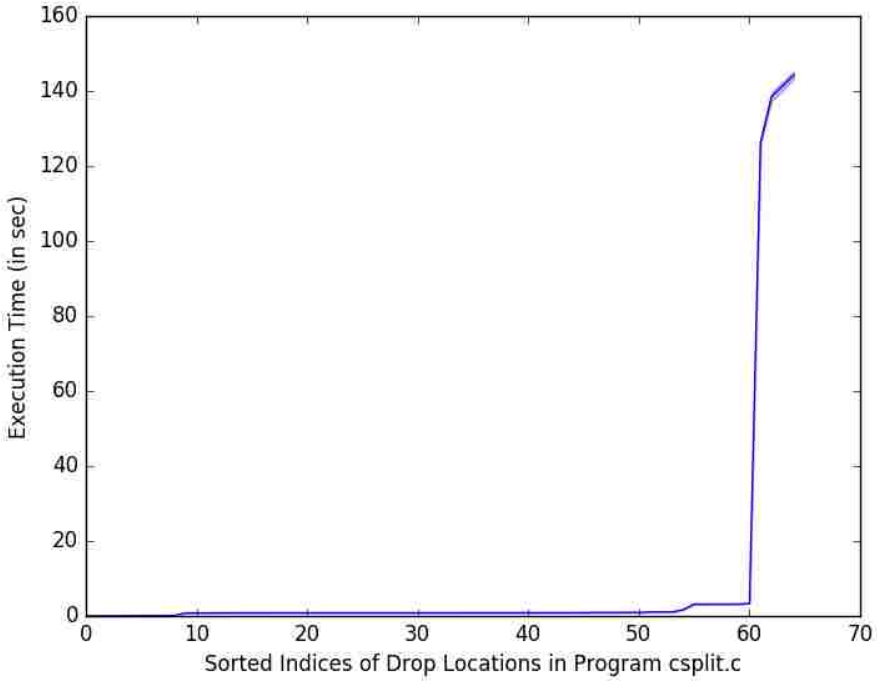


FIGURE A.126. Execution time of DDA on *csplit*

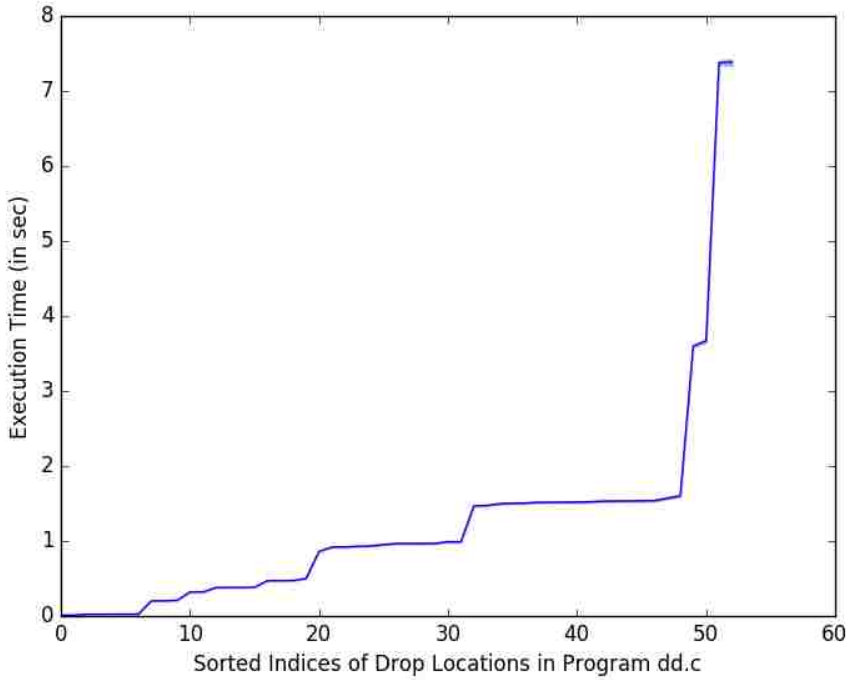


FIGURE A.127. Execution time of DDA on *dd*

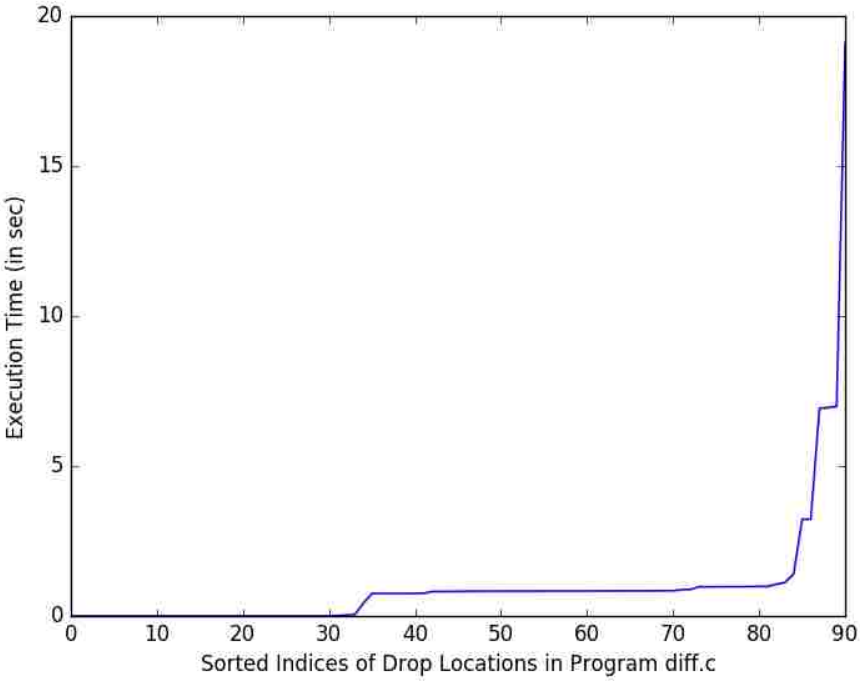


FIGURE A.128. Execution time of DDA on *diff*

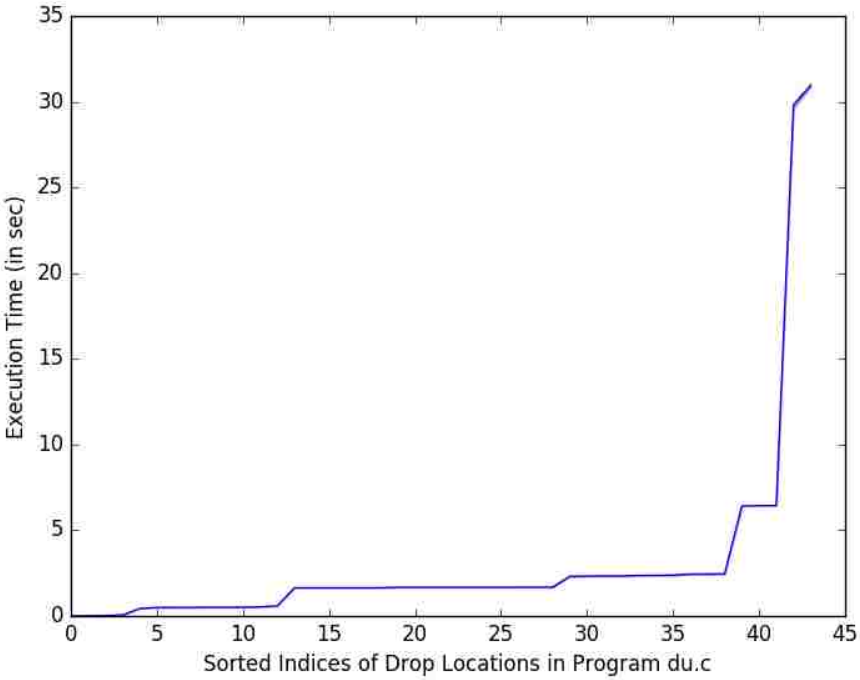


FIGURE A.129. Execution time of DDA on *du*

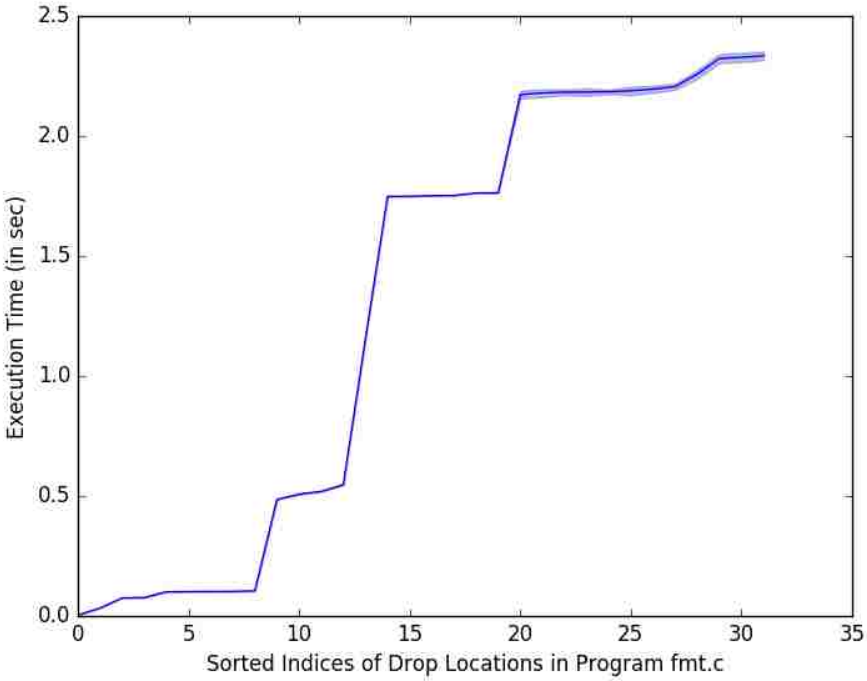


FIGURE A.130. Execution time of DDA on *fmt*

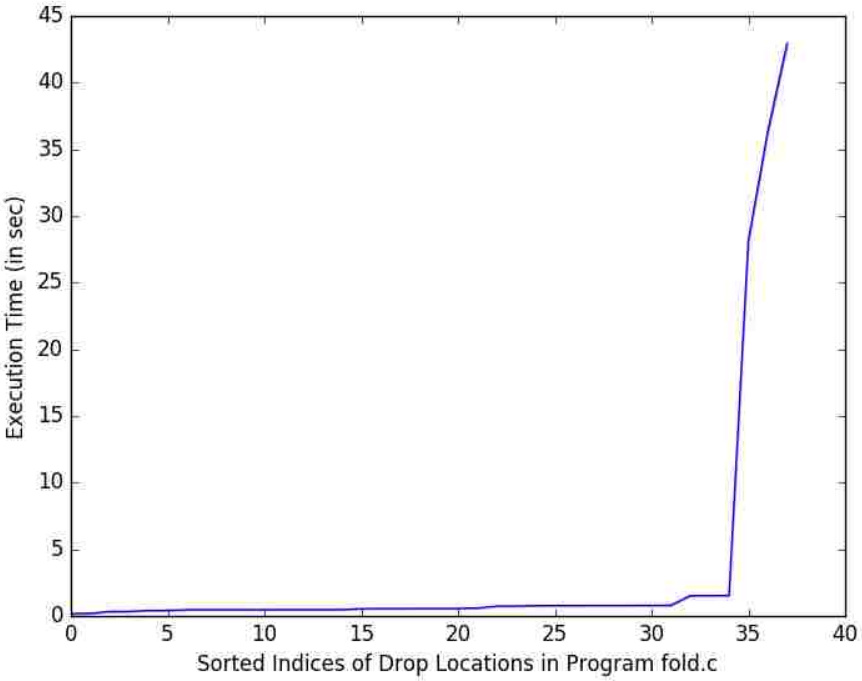


FIGURE A.131. Execution time of DDA on *fold*

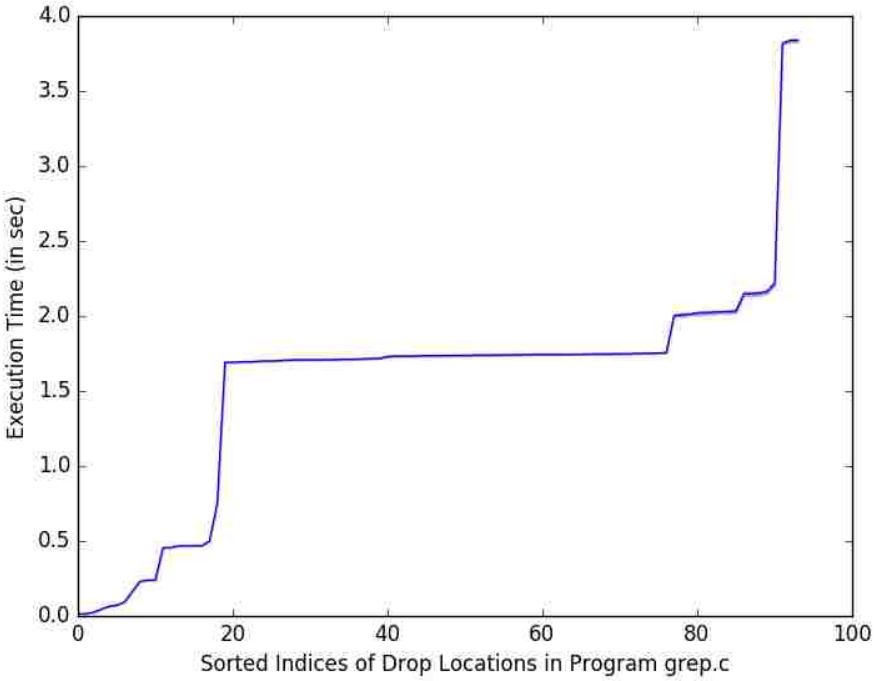


FIGURE A.132. Execution time of DDA on *grep*

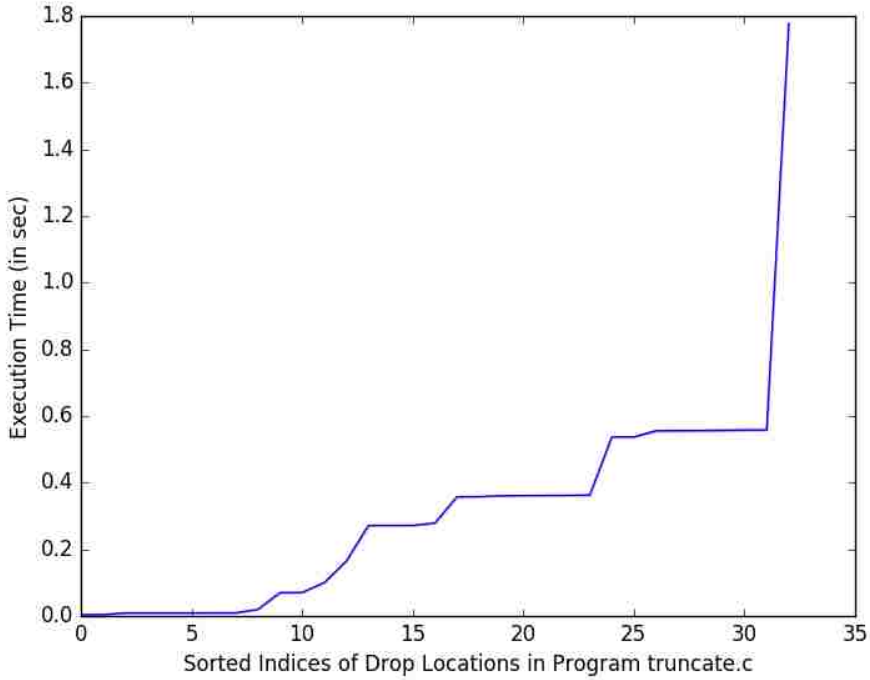


FIGURE A.133. Execution time of DDA on *truncate*

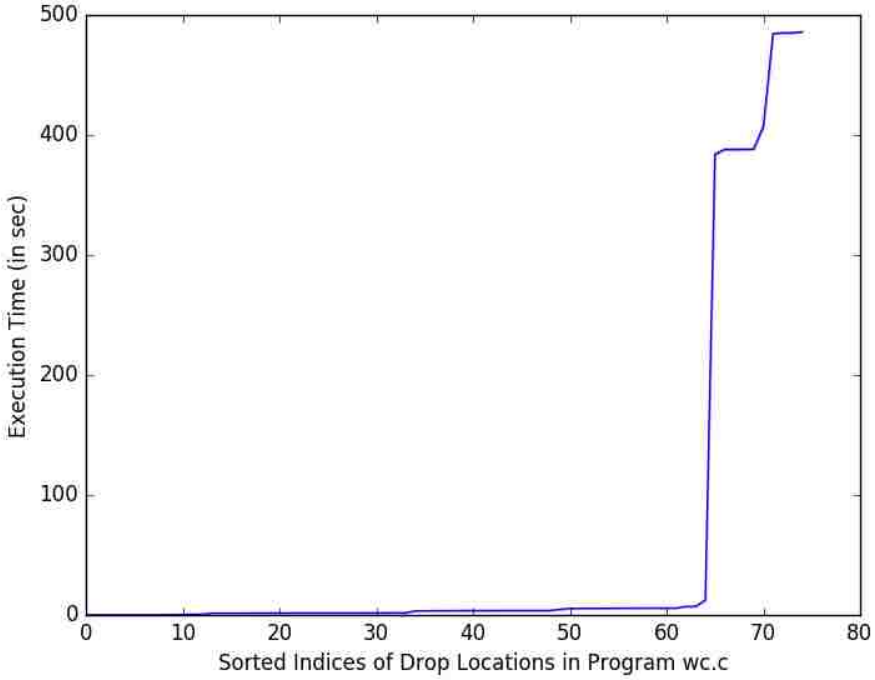


FIGURE A.134. Execution time of DDA on *wc*

Vita

Vivian M. Ho passed the final exam (final defense) on June 2017 and is currently a candidate for the degree of Master in Computer Science.