Quantitative Analysis of Systems Using Game-Theoretic Learning

Sanjit A. Seshia
Alexander Rakhlin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Acknowledgement

Quantitative Analysis of Systems Using Game-Theoretic Learning

Sanjit A. Seshia

University of California, Berkeley

sseshia@eecs.berkeley.edu

and

Alexander Rakhlin

University of Pennsylvania

rakhlin@wharton.upenn.edu

The analysis of quantitative properties, such as timing and power, is central to the design of reliable embedded software and systems. However, the verification of such properties on a program is made difficult by their heavy dependence on the program's environment, such as the processor it runs on. Modeling the environment by hand can be tedious, error-prone and time consuming. In this paper, we present a new, game-theoretic approach to analyzing quantitative properties that is based on performing systematic measurements to automatically learn a model of the environment. We model the problem as a game between our algorithm (player) and the environment of the program (adversary), where the player seeks to accurately predict the property of interest while the adversary sets environment states and parameters. To solve this problem, we employ a randomized strategy that repeatedly tests the program along a linear-sized set of program paths called basis paths, using the resulting measurements to infer a weighted-graph model of the environment, from which quantitative properties can be predicted. Test cases are automatically generated using satisfiability modulo theories (SMT) solving. We prove that our algorithm can, under certain assumptions and with arbitrarily high probability, accurately predict properties such as worst-case execution time or estimate the distribution of execution times. Experimental results for execution time analysis demonstrate that our approach is efficient, accurate, and highly portable.

1. INTRODUCTION

The main distinguishing characteristic of embedded computer systems is the tight integration of computation with the physical world. Consequently, the behavior of software controllers of such *cyber-physical* systems has a major effect on physical properties of such systems. These properties are quantitative, including constraints on resources, such as timing and power, and specifications involving physical parameters, such as position and velocity. The verification of such physical properties of embedded software systems requires modeling not only the software program but also the relevant aspects of the program's environment. However, only limited progress has been made on these verification problems. One of the biggest obstacles is the difficulty in creating an accurate model of a complex environment.

Consider, for example, the problem of estimating the execution time of a software task.

This problem plays a central role in the design of real-time embedded systems, for example, to provide timing guarantees, for use in scheduling algorithms, and also for use in co-simulating a software controller with a model of the continuous plant it controls.[1] In particular, the problem of estimating the worst-case execution time (WCET), has been the subject of significant research efforts over the last 20 years (e.g. [Li and Malik 1999; Reinhard Wilhelm et al. 2008]). However, much work remains to be done to completely solve this problem. The complexity arises from two dimensions of the problem: the *path problem*, which is to find the worst-case path through the task, and the *state problem*, which seeks to find the worst-case environment state to run the task from. The problem is particularly challenging because these two dimensions interact closely: the choice of path affects the state and vice-versa. Significant progress has been made on this problem, especially in the computation of bounds on loops in tasks, in modeling the dependencies amongst program fragments using (linear) constraints, and modeling some aspects of processor behavior. However, as pointed out in recent papers by Lee [2007] and Kirner and Puschner [2008], it is becoming increasingly difficult to precisely model the complexities of the underlying hardware platform (e.g., out-of-order processors with deep pipelines, branch prediction, caches, parallelism) as well as the software environment. This results in timing estimates that are either too pessimistic (due to conservative platform modeling) or too optimistic (due to unmodeled features of the platform). Industry practice typically involves making random, unguided measurements to obtain timing estimates. As Kirner and Puschner [2008] write, a major challenge for measurement-based techniques is the automatic and systematic generation of test data.

In this paper, we present a new *game-theoretic* approach to verifying physical properties of embedded software that operates by *systematically testing* the software in its target environment, and *learning an environment model*. The following salient features of our approach distinguish it from previous approaches in the literature:

- *Game-theoretic formulation:* We model the problem of estimating a physical quantity (such as time) as a multi-round game between our estimation algorithm (player) and the environment of the program (adversary). The physical quantity is modeled as the length of the particular execution path the program takes. In the game, the player seeks to estimate the length of any path through the program while the adversary sets environment parameters to thwart the player. Each round of the game constitutes one test. Over several rounds, our algorithm learns enough about the environment to be able to accurately predict path lengths with high probability, where the probability increases with the number of rounds (precise statement in Sec. 4). In particular, we show how our algorithm can be used to predict the longest path and thus predict properties such as worst-case execution time (WCET).

- *Learning an environment model:* A key component of our approach is the use of statistical learning to generate an environment model that is used to estimate the physical quantity of interest. The environment is viewed as an adversary that selects weights on edges of the program's control flow graph in a manner that can possibly depend on the choice of the path being tested. This path-dependency is modeled as a perturba-

---

[1]In "software-in-the-loop" simulation, the actual software implementation of a controller is simulated along with a model of the continuous plant it controls. However, for scalability, such simulation must be performed on a workstation, not on the embedded target. Consequently, the timing behavior of different execution paths in the program must be inferred a-priori on the target and then used during the workstation-based simulation.

tion of weights by the adversary. Our algorithm seeks to estimate path lengths in spite of such adversarial setting of weights. The weight-perturbation model can capture not only adversarial choices made by the environment, but also to errors in measurement.

- *Systematic and efficient testing and measurement:* Another central idea is to perform *systematic measurements* of the physical quantity, by sampling only so-called *basis paths* of the program. The intuition is that the length of any program path can be approximated as a linear combination of the observed lengths of the basis paths plus a perturbation. We use satisfiability modulo theories (SMT) solvers [Barrett et al. 2009] and integer programing to generate feasible basis paths and to generate test inputs to drive a program's execution down a basis path. This approach is efficient because the number of basis paths is linear in the size of the program. Additionally, the measurements are made *without instrumenting the program*, thereby avoiding any resultant skew in behavior.

A goal of this paper is to demonstrate that the above three concepts can be useful for quantitative analysis of software-controlled embedded systems. Additionally, although our focus is on software analysis, we believe that these concepts can also be useful for the analysis of physical properties of cyber-physical systems in general.

We present both theoretical and experimental results demonstrating the utility of our approach. On the theoretical side, we prove that, for any *adversarially-chosen sequence* of environment states to run the program from and given any $\delta > 0$, if we *run a number of tests that is polynomial* in the input size and $\ln \frac{1}{\delta}$, our algorithm accurately estimates the average length of any program path for that environment state sequence with probability $1 - \delta$ (formal statement in Section 4). Under certain assumptions, we can use this result to accurately find the the longest path, which, for timing, amounts to finding the input that yields the worst-case execution time (WCET). Moreover, for any ε, our algorithm can also be used to find a set of paths of length within ε of the longest.

We demonstrate our approach for the problem of execution time analysis of embedded software. Our approach is implemented in a tool called GAMETIME, which has the following features and applications:

- *Portability and ease of use:* GAMETIME is measurement-based and so can be more easily applied to complex, hard-to-model platforms, and during design space exploration;
- *WCET estimation:* GAMETIME generates test cases exhibiting lower bounds on the true WCET, which are tight, under certain assumptions, with arbitrarily high probability. We present experimental results comparing GAMETIME to existing state-of-the-art WCET estimation tools aiT [AbsInt Angewandte Informatik] and Chronos [Li et al. 2005]. Perhaps surprisingly, results indicate that GAMETIME can generate *even bigger* execution-time estimates than some of these tools;
- *Predicting execution times of arbitrary paths:* GAMETIME can be used to predict a set of ε-longest paths as well as the distribution of execution times of a program. These problems are relevant for soft real-time systems and for generating execution-time estimates to guide combined simulation of a software controller and its plant.

For concreteness, we focus the rest of the paper on execution time analysis. However, the theoretical formulation and results in Section 4 can apply to the estimation of *any physical quantity* of systems with graph-based models; we have therefore sought to present our theoretical results in a general manner as relating to the lengths of paths in a graph.

The outline of the paper is as follows. We begin with a survey of related work in Section 2, mainly focussed on execution time analysis. The basic formulation and an overview of our approach is given in Section 3. The algorithm and main theorems are given in Section 4, and experimental results in Section 5. We conclude in Section 6.

A preliminary version of this work appeared in [Seshia and Rakhlin 2008]. This extended version substantially expands on both theoretical and experimental results, and describes the theoretical model in far greater detail.

2. BACKGROUND AND RELATED WORK

We briefly review literature on estimating physical parameters of software and relevant results from learning theory and place our work in this context.

2.1 Estimating Execution Time and Other Physical Quantities

There is a vast literature on estimating execution time, especially WCET analysis, comprehensively surveyed by Li and Malik [Li and Malik 1999] and Wilhelm et al. [Wilhelm 2005; Reinhard Wilhelm et al. 2008]. For lack of space, we only include here a brief discussion of current approaches and do not cover all tools. References to current techniques can be found in a recent survey [Reinhard Wilhelm et al. 2008].

There are two parts to current WCET estimation methods: *program path analysis* (also called *control flow analysis*) and *processor behavior analysis*. In program path analysis, the tool tries to find the program path that exhibits worst-case execution time. In processor behavior analysis (PBA), one models the details of the platform that the program will execute on, so as to be able to predict environment behavior such as cache misses and branch mis-predictions. PBA is an extremely time-consuming process, with several man-months required to create a reliable timing model of even a simple processor design.

Current tools are broadly classified into those based on *static analysis* (e.g., aiT, Bounds-T, SWEET, Chronos) and those that are *measurement-based* (e.g., RapiTime, SymTA/P, Vienna M./P.). Static tools rely on abstract interpretation and dataflow analysis to compute facts at program points that identify dependencies between code fragments and generate loop bounds. Even static techniques use measurement for estimating the time for small program fragments, and measurement-based techniques rely on techniques such as model checking to guide path exploration. Static techniques also perform implicit path enumeration (termed "IPET"), usually based on integer linear programming. The state-of-the-art measurement-based techniques [Wenzel et al. 2008] are based on generating test data by a combination of program partitioning, random and heuristic test generation, and exhaustive path enumeration by model checking.

Our technique is *measurement-based*, it suffers no over-estimation, and it is easy to port to a new platform. It is distinct from existing measurement-based techniques due to the novel game-theoretic formulation, basis path-based test generation, and the use of online learning to infer an environment model. Our approach does rely on some static techniques, in deriving loop bounds and using symbolic execution and satisfiability solvers to compute inputs to drive the program down a specific path of interest. In particular, note that our approach completely avoids the difficulties of processor behavior analysis, instead directly executing the program on its target platform. Moreover our approach applies not just to WCET estimation, but also to estimating the distribution of execution times of a program.

While there have been several papers about quantitative verification of formal models of systems (e.g. [Chakrabarti et al. 2005]), these typically assume that the quantitative

parameters of primitive elements (such as execution time of software tasks) are given as input. There is relatively little work on directly verifying non-timing properties on software, with the exception of estimating the power used by software-controlled embedded systems [Tiwari et al. 1994].

Adversarial analysis has been employed for problems such as system-level dynamic power management [Irani et al. 2005], but to our knowledge, the adversarial model and analysis used in this paper is the first for timing analysis and for estimating quantitative parameters of software-based systems.

2.2 Learning Theory

Results of this paper build on the *game-theoretic prediction* literature in learning theory. This field has witnessed an increasing interest in sequential (or *online*) learning, whereby an agent discovers the world by repeatedly acting and receiving feedback. Of particular interest is the problem of learning in the presence of an adversary with a *complete absence of statistical assumptions* on the nature of the observed data.

The problem of sequentially choosing paths to minimize the *regret* (the difference between cumulative lengths of the paths chosen by our algorithm and the total length of the longest path after T rounds) is known as an instance of *bandit online linear optimization*. The "bandit" part of the name is due to the connection with the *multi-armed bandit* problem, where only the payoff of the chosen "arm" (path) is revealed. The basic "bandit" problem was put forth by Robbins [1952] and has been well-understood since then. The recent progress comes from the realization that well-performing algorithms can be found (a) for large decision spaces, such as paths in a graph, and (b) under adversarial conditions rather than the stochastic formulation of Robbins. We are the first to bring these results to bear on the problem of quantitative analysis of embedded software. In addition to this novel application, our results are of independent theoretical interest in the area of game-theoretic learning. Briefly, with the results herein, we now know that the best achievable regret for the stochastic multiarmed bandit problem is $O(\sqrt{T})$ without a *margin assumption* (Assumption 4.1 in Sec. 4.2) and $O(\log T)$ with the assumption. The $O(\sqrt{T})$ regret has been recently proven for the non-stochastic bandit without the margin assumption [Auer et al. 2003]. The new result of this paper is the $O(\log T)$ regret rate under the margin assumption for the non-stochastic bandit setting, completing the picture (see Corollary 4.1).

We refer the reader to a recent book [Cesa-Bianchi and Lugosi 2006] for a comprehensive treatment of sequential prediction. Some relevant results can be found in [McMahan and Blum 2004; György et al. 2007; Awerbuch and Kleinberg 2004].

2.3 Miscellaneous

Our algorithm uses the concept of *basis paths* of a program, which has been explored for computing the *cyclomatic complexity* of a program [McCabe 1976]; however, we give theoretical results by extracting a special basis called a *barycentric spanner* [Awerbuch and Kleinberg 2004]. For input test generation, our approach heavily relies on advances in SMT solving; these techniques are surveyed in a recent book chapter [Barrett et al. 2009].

3. THEORETICAL FORMULATION AND OVERVIEW

We are concerned with estimating a physical property of a software task (program) executing in its target platform (environment). The property is a function of a physical quantity of interest, denoted by q; e.g., it can be the maximum value that q takes. In general, q is

a function of three entities: the program code, the parameters of its environment, and the input to the program. More concisely, we can express q as the following function

$$q = f_P(x, w)$$

where x denotes the input to the program P (such as data read from sensors or received over the network), w denotes the state of the environment (such as the initial hardware state including, for example, the contents of the cache), and f_P denotes the program-specific function that maps x and w to a value of the physical quantity.

In general, x, w, and q vary over time; we make the variation explicit with a subscript t:

$$q_t = f_P(x_t, w_t)$$

Some sample problems of interest are listed below. In each problem, we are not only interested in estimating a physical property, but also in computing an input x that serves as a witness or counterexample for that property.

(1) *Global worst-case estimation:* In this case, we want to estimate the largest value of the quantity q for all values of x and w, namely, $\max_{x,w} f_P(x, w)$. Equivalently, given *any* infinite sequence of x_t and w_t values, we wish to estimate the following quantity:

$$\max_{t=1..\infty} \max_{x_t,w_t} f_P(x_t, w_t) \tag{1}$$

The arbitrary infinite sequence of x_t and w_t values represents an arbitrary sequence of inputs and environment parameters encountered over the lifetime of program P.

(2) *Worst-case estimation over a time horizon τ:* In this case, the worst case is to be computed over a finite time horizon τ, formally specified as follows:

$$\max_{t=1..\tau} \max_{x_t,w_t} f_P(x_t, w_t) \tag{2}$$

(3) *Estimating sample average over time horizon τ against an adversarial environment:* In this case, we want to estimate, for a time horizon τ and for *any* sequence of environment parameters $w_1, w_2, \ldots, w_\tau$, the following quantity:

$$\max_{x_t} \frac{1}{\tau} \sum_{t=1}^{\tau} f_P(x_t, w_t) \tag{3}$$

(4) *Can the system consume R resources at any point over a time horizon of τ:* The question we ask here is whether there are values x_t and w_t, $t = 1, 2, \ldots, \tau$, such that q_t exceeds R. For example, a concrete instance of this problem is to ask whether a software task can take more than R seconds to execute.

In this paper, the program P is assumed to have the following properties: (i) P is known to terminate (so we are not attempting to solve the halting problem); (ii) there are statically-known upper bounds on all loops and on the depth of all recursive function calls in P (we are not solving the loop bound inference problem); and (iii) P is single-threaded and runs uninterrupted.

Under these assumptions, we give a randomized algorithm to solve Problem 3 which, under certain assumptions (formally stated in Sec. 4), can also be used to perform worst-case estimation (Problems 1 and 2) as well as answer the resource-bound consumption problem listed above (Problem 4). We note that previous work on WCET has sought to

address the global WCET estimation problem (Problem 1), and therefore our experimental comparison is with these techniques for the global WCET estimation problem.

For concreteness, in the remainder of this section, we will focus on a single quantity, *execution time*, and on a single representative problem, namely, the *worst-case execution time* (WCET) estimation problem. As noted earlier, our theoretical formulation and algorithms can carry over to estimating any physical quantity and to problems other than worst-case analysis.

The rest of this section is organized as follows. We begin with a general discussion of how the estimation problem can be formulated as a game (Section 3.1). Next, we formalize our problem definition and introduce our theoretical model of the environment (Section 3.2). Finally, we give an overview of our solution approach using a small example (Section 3.3).

3.1 Intuition for the Game-Theoretic Formulation

Consider the finite-horizon WCET estimation problem (Problem 2). For brevity, in what follows, we will refer to this problem as simply WCET estimation.

Game-theoretic formulation: We model the estimation problem as a game between the estimation tool $\mathcal{T}$ and the environment $\mathcal{E}$ of program P.

The game proceeds over multiple rounds, $t = 1, 2, 3, \ldots$. In each round, $\mathcal{T}$ picks the input x_t to P; the execution path taken through P is determined by x_t. $\mathcal{E}$ picks, in a possibly adversarial fashion, environment parameters (state) w_t. Note that w_t *can depend* on x_t. P is then run on x_t for parameters w_t.

At the end of each round t, $\mathcal{T}$ receives as feedback (only) the execution time l_t of P for the chosen input x_t under the parameters w_t selected by $\mathcal{E}$. Note that $\mathcal{T}$ does not observe w_t. Note also that $\mathcal{T}$ *only receives the overall execution time* of the task P, not a more fine-grained measurement of (say) each basic block in the task along the chosen path. This enables us to minimize any skew from instrumentation inserted to measure time. Based on the feedback l_t, $\mathcal{T}$ can modify its input-selection strategy.

After some number of rounds τ, we stop: $\mathcal{T}$ must output its prediction x_τ^* of the input x^* that maximizes the quantity defined in Equation 2 above (for the choice of w_t's made by $\mathcal{E}$). $\mathcal{T}$ wins the game if its prediction is correct (i.e., $x_\tau^* = x^*$); otherwise, $\mathcal{E}$ wins. In addition to generating the prediction x_τ^*, $\mathcal{T}$ must also output an estimate of the quantity in Equation 2.

The goal of $\mathcal{T}$ is thus to select a sequence of inputs $x_1, x_2, \ldots, x_\tau$ so that it can identify (at least with high probability) the longest execution time of P during $t = 1, 2, \ldots, \tau$.

Note that this longest execution time need not be due to inputs that have been already tried out by $\mathcal{T}$.

By permitting $\mathcal{E}$ to select environment parameters based on $\mathcal{T}$'s choice of input, we can model input-dependent perturbation in execution time of basic blocks as well as perturbation in execution time on a single input due to variation in environmental conditions or measurement error. However, in practice, such perturbation by $\mathcal{E}$ cannot be arbitrary, otherwise, it will be impossible to accurately predict execution time and compute worst-case inputs. Intuitively, the perturbation corresponds to the timing predictability of the platform. If a platform has predictable timing, such as the PRET processor proposed by Edwards and Lee [2007], it would mean that the perturbation is small.

In practice, as in any technique involving game-solving (e.g., [Chakrabarti et al. 2005]), it is necessary to suitably explore the choices of w_t's by $\mathcal{E}$ during the estimation process.

For example, to estimate the global worst-case execution time (Problem 1), we require that $\mathcal{E}$ selects the worst-case environment state at some point during $t = 1, 2, \ldots, \tau$. If there are a finite number of environment states, one would need to exhaustively enumerate these during $t = 1, 2, \ldots, \tau$ in order to perform worst-case analysis. The power of the approach we present in this paper is that we do not require to exhaustively explore the space of inputs x_t for each choice of w_t.

Formulation as a graph problem: An additional aspect of our model is that the game can be viewed to operate on the control-flow graph G_P of the task P, with $\mathcal{T}$ selecting inputs that drive P's execution down selected paths, while $\mathcal{E}$ selects environment parameters w that determine path lengths. We elaborate below.

3.2 Theoretical Formulation

Consider a directed acyclic graph $G = (V, E)$ derived from the control-flow graph of the task with all loops unrolled to a safe upper bound and all function calls inlined. We will assume that there is a single source node u and single sink node v in G; if not, then dummy source and sink nodes can be added without loss of generality.

Let $\mathcal{P}$ denote the set of all paths in G from source u to sink v. We can associate each of the paths with a binary vector with $m = |E|$ components, depending on whether the edge is present or not. In other words, each source-sink path is a vector x in $\{0, 1\}^m$, where the ith entry of the vector for a path x corresponds to edge i of G, and is 1 if edge i is in x and 0 otherwise. The set $\mathcal{P}$ is thus a subset of $\{0, 1\}^m$.

The path prediction interaction is modeled as a repeated game between our algorithm ($\mathcal{T}$) and the program environment ($\mathcal{E}$). On each round t, $\mathcal{T}$ chooses a path $x_t \in \mathcal{P}$ between u and v. Also, the adversary $\mathcal{E}$ picks a table of non-negative path lengths given by the function $\mathcal{L}_t : \mathcal{P} \to \mathbb{R}^{\geq 0}$. Then, the total length l_t of the chosen path x_t is revealed, where $l_t = \mathcal{L}_t(x_t)$. The game proceeds for some number of rounds $t = 1, 2, \ldots, \tau$.

At the end of round τ, $\mathcal{T}$ wins iff it correctly estimates a path x^* that generates the worst-case execution time due to environment states in rounds $t = 1, 2, \ldots, \tau$. $\mathcal{T}$ must also output an estimate of the corresponding WCET, which is expressed as the following quantity:

$$\mathcal{L}_{\max} = \max_{x \in \mathcal{P}} \max_{t = 1, 2, \ldots, \tau} \mathcal{L}_t(x) \tag{4}$$

The worst-case path is an element of $\mathcal{P}$ at which $\mathcal{L}_{\max}$ is attained:

$$x^* = \text{argmax}_{x \in \mathcal{P}} \max_{t = 1, 2, \ldots, \tau} \mathcal{L}_t(x) \tag{5}$$

We make a few remarks on the above theoretical model.

First, we stress that, in the above formulation, the goal is to find the WCET *due to environment states in rounds $t = 1, 2, \ldots, \tau$*. In order to find the true WCET, for all possible environment states, we need to assume (or ensure, through systematic exploration) that the worst-case state occurs at some time between $t = 1$ and $t = \tau$. We contend that this formulation is useful in spite of this assumption. For whole program WCET analysis, the starting environment state is typically assumed to be known. Even if one needs to enumerate all possible starting environment states, it is computationally very expensive to also enumerate all possible paths for each such state. With our formulation, we seek to demonstrate that one can accurately estimate the WCET even if we do not sample the worst-case path when the worst-case state occurred.

Second, the definition of our estimation target $L_{\max}$ assumes that the timing of a program depends only on the control flow through that program. In general, the timing can also depend on characteristics of input data that do not influence control flow. We believe that the basic game framework we describe can also apply to the case of data-dependent timing, and leave an exploration of this aspect to future work.

Overall, we believe that decoupling the path problem from the state problem in a manner that can be applied easily to any platform is in itself a significant challenge. This paper mainly focuses on addressing this challenge. In future work, we plan to address the limitations of the model as identified above.

The third and final remark we make is about the "size" of the theoretical model. Since a DAG can have exponentially-many paths in the number of nodes and edges, the domain of the function L_t is potentially exponential, and can change at each round t. In the worst case, the strategy sets of both $\mathcal{T}$ and $\mathcal{E}$ in this model are exponential-sized, and it is impossible to exactly learn L_t for every t without sampling all paths. Hence, we need to approximate the above model with another model that, while being more compact, is equally expressive and generates useful results in practice.

Below, we present a more compact model, which our algorithm is then based upon. We first present our general model, and then describe a simplified model that will be used in Sec. 4 to introduce more easily some key ideas of our algorithm.

3.2.1 *Modeling with Weights and Perturbation.* We model the selection of the table of lengths L_t by the environment $\mathcal{E}$ as a two-step procedure.

(i) First, concurrent with the choice of x_t by $\mathcal{T}$, $\mathcal{E}$ chooses a vector of non-negative *edge weights*, $w_t \in \mathbb{R}^m$, for G. These weights represent *path-independent* delays of basic blocks in the program.

For generality, we deliberately leave the exact specification of w_t unspecified, but we give a few possibilities here. For example, one could view w_t as the delay of a basic block if it were executed in the starting hardware state. As another example, one could associated with every basic block a default path, and consider w_t to be the delay of that basic block when the default path is executed.

(ii) Then, after observing the path x_t selected by $\mathcal{T}$, $\mathcal{E}$ picks a distribution from which it draws a perturbation vector $\pi_t(x_t)$. The functional notation indicates that the distribution is a function of x_t.

The vector $\pi_t(x_t)$ models the path-specific changes that $\mathcal{E}$ applies to its original choice w_t. For example, when $w_t(e)$ represents the delay of basic block e in the starting hardware state, the perturbation $\pi_t(x_t)(e)$ is the change in delay due to changes in hardware state along the path x_t. We will abbreviate both $\pi_t(x_t)$ and $\pi_t(x)$ by π^x_t when it is unnecessary to draw a distinction between those terms; otherwise, for x that could be different from x_t, we will explicitly write $\pi^x_t(x)$ or $\pi_t(x)$.

The only restriction we place on $\pi_t(x)$, for any x, is that $\|\pi_t(x)\|_1 \leq N$, for some finite N. The parameter N is arbitrary, but places the (realistic) constraint that the perturbation of any path length cannot be unbounded.

Thus, the overall path length observed by $\mathcal{T}$ is

$$l_t = x_t \cdot (w_t + \pi^x_t) = x_t^\mathsf{T}(w_t + \pi^x_t)$$

Now let us consider how this model relates to the original formulation we started with.

First, note that, in the original model, $\mathcal{E}$ picks the function $\mathcal{L}_t$ that defines the lengths of all paths. To relate to that model, here we can assume, without loss of generality, that $\mathcal{E}$ draws a-priori the perturbation vectors $\pi^x_t(x)$ for all $x \in \mathcal{P}$, but only $\pi^x_t(x_t)$ plays a role in determining l_t.

Second, equating the observed lengths, we see that

$$\mathcal{L}_t(x_t) = x_t^\top (w_t + \pi^x_t)$$

The main constraint on this equation is the requirement that $\|\pi^x_t\|_1 \leq N$, which implies that $|x_t^\top \pi^x_t| \leq N$. In effect, by using this model we require that $\mathcal{E}$ pick $\mathcal{L}_t$ by first selecting path-independent weights w_t and then, for each source-sink path, modifying its length by a bounded perturbation (of at most $\pm N$). Note, however, that the model places absolutely no restrictions on the value of w_t or how it changes with t (from round to round).

The goal for $\mathcal{T}$ in this model is to estimate the following quantity

$$\mathcal{L}_{\max} = \max_{x \in \mathcal{P}} \max_{t=1,2,\ldots,\tau} x^\top (w_t + \pi^x_t(x)) \tag{6}$$

Moreover, we would also like $\mathcal{T}$ to identify the worst-case path given by

$$x^* = \operatorname{argmax}_{x \in \mathcal{P}} \max_{t=1,2,\ldots,\tau} x^\top (w_t + \pi^x_t(x)) \tag{7}$$

3.2.2 *Simplified Model without Perturbation.* To more easily introduce the key concepts in our algorithm, we will initially assume that the perturbation vectors at all time points are identically 0, viz., $\pi^x_t(x) = 0$ for all t and x.

Clearly, this is an unrealistic idealization in practice, since in this model the length of an edge is independent of the path it lies on. We stress that our main theoretical results, stated in Sec. 4.3, are for the more realistic model defined above in Section 3.2.1.

We next give an overview of our approach in the context of a small example.

3.3 Overview of Our Approach

We describe the working of our approach using a small program from an actual real-time embedded system, the Paparazzi unmanned aerial vehicle (UAV) [Nemer et al. 2006]. Figure 2 shows the C source code for the `altitude_control_task` in the Paparazzi code, which is publicly available open source.

Starting with the source code for a task, and all the libraries and other definitions it relies on, we run the task through a C pre-processor and the CIL front-end [George Necula et al.] and extract the control-flow graph (CFG). In this graph, each node corresponds to the start of a basic block and edges are labeled with the basic block code or conditional statements that govern control flow (conditionals are replicated along both if and else branches). Note that we assume that code termi-

$x1 = (1,1,1,0,0,1,1,0,0,1)$

$x2 = (1,0,0,1,1,0,0,1,1,1)$

$x3 = (1,1,1,0,0,0,0,1,1,1)$

$x4 = (1,0,0,1,1,1,1,0,0,1)$

$x4 = x1 + x2 - x3$

Fig. 1. **Illustration of Basis Paths.** An edge label indicates the position for that edge in the vector representation of a path.

nates, and bounds are known on all loops. Thus, we start with code with all loops (if any) unrolled, and the CFG is thus a directed acyclic graph (DAG). We also pre-process the
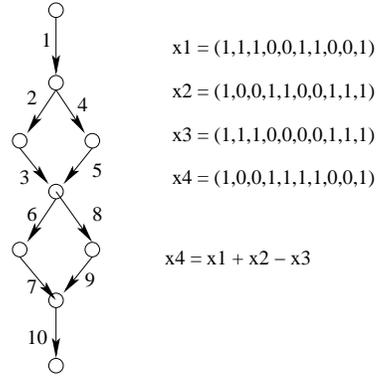
```
#define PPRZ_MODE_AUTO2 2
#define PPRZ_MODE_HOME  3
#define VERTICAL_MODE_AUTO_ALT 3
#define CLIMB_MAX 1.0
. . .
void altitude_control_task() {
  if (pprz_mode == PPRZ_MODE_AUTO2
     || pprz_mode == PPRZ_MODE_HOME) {
    if (vertical_mode == VERTICAL_MODE_AUTO_ALT) {
    /* inlined below: altitude_pid_run() */
      float err = estimator_z - desired_altitude;
      desired_climb = pre_climb + altitude_pgain * err;
      if (desired_climb < -CLIMB_MAX)
        desired_climb = -CLIMB_MAX;
      if (desired_climb > CLIMB_MAX)
        desired_climb = CLIMB_MAX;
}}}
```
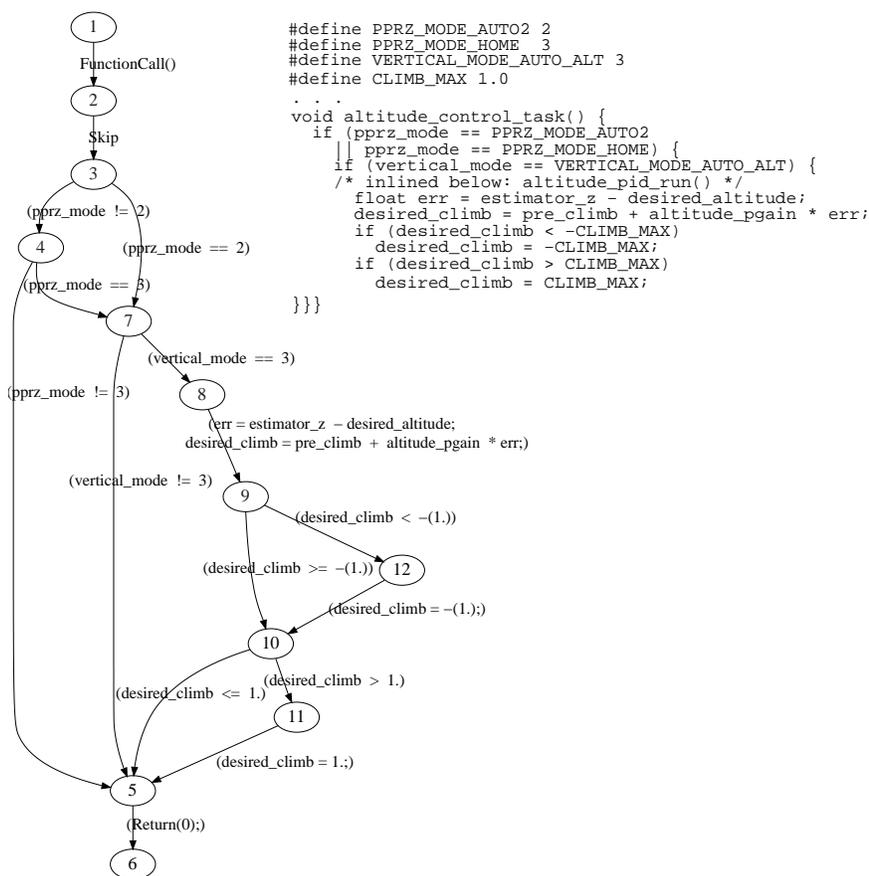
Fig. 2.   **Control-flow graph and code for `altitude_control_task`**

CFG so that it has exactly one source and one sink. Each execution through the program is a source-to-sink path in the CFG.

An exhaustive approach to program path analysis will need to enumerate all paths in this DAG. However, it is possible for a DAG to have exponentially many paths (in the number of vertices/edges). Thus, a brute-force enumeration of paths is not efficient.

Our approach is to sample a set of *basis paths*. Recall that each source-sink path can be viewed as a vector in $\{0,1\}^m$, where m is the number of edges in the unrolled CFG. The set of all source-sink paths thus forms a subset $\mathcal{P}$ of $\{0,1\}^m$. We compute the basis for $\mathcal{P}$ in which each element of the basis is also a source-sink path.

Figure 1 illustrates the ideas using a simple "2-diamond" example of a CFG. In this example, paths x_1, x_2 and x_3 form a basis and x_4 can be expressed as the linear combination $x_1 + x_2 - x_3$.

Our algorithm, described in detail in Section 4, randomly samples basis paths of the CFG and drives program execution down those paths by generating tests using SMT solving. From the observed lengths of those paths, we estimate edge weights on the entire graph. This estimate, accumulated over several rounds of the game, is then used to predict the

longest source-sink path in the CFG. Theoretical guarantees on performance are proved in Section 4 and experimental evidence for its utility is given in Section 5.

4. ALGORITHM AND THEORETICAL RESULTS

Recall that, in the model introduced in the previous section, the path prediction interaction is modeled as a repeated game between our algorithm (Player) and the program environment (Adversary) on the unrolled control-flow graph $G = (V, E)$. On each round t, we choose a source-sink path $x_t \in \mathcal{P} \subseteq \{0, 1\}^m$, where $m = |E|$. The adversary chooses the lengths of paths in the graph. We assume that this choice is made by the following two stage process: first, the adversary chooses the worst-case *weights*, $w_t \in \mathbb{R}^m$, on the edges of G independently of our choice x_t, and then skews these weights by adding a random perturbation $\pi^x{}_t$, whose distribution depends on x_t. (We will also refer to edge weights and path lengths as "delays", to make concrete the link to timing analysis.)

In the simplified model, which we consider first, we suppose that the perturbation is zero; thus, we observe the overall path length $l_t = x_t^\mathsf{T} w_t$. In the general model, only $l_t = x_t^\mathsf{T}(w_t + \pi^x{}_t)$ is observed. No other information is provided to us; not only do we not know the lengths of the paths not chosen, we do not even know the contributions of particular edges on the chosen path. It is important to emphasize that in the general model we assume that the adversary is *adaptive* in that w_t and $\pi^x{}_t$ can depend on the past history of choices by the player and the adversary.

Suppose that there is a single fixed path x^* which is the longest path on each round. One possible objective is to find x^*. In the following, we exhibit an efficient randomized algorithm which allows us to find it correctly with high probability. In fact, our results are more general: if no single longest path exists, we can provably find a batch of longest paths. We describe later how our theoretical approach paves the way for quantitative (timing) analysis given a range of assumptions at hand.

Before diving into the details of the algorithm, let us sketch how it works:

- First, compute a representative set of basis paths, called a *barycentric spanner*;
- For a specified number of iterations τ, do the following:
  - ⋆ pick a path from the representative set;
  - ⋆ observe its length;
  - ⋆ construct an estimate of edge weights on the whole graph from the observed length;
- Find the longest path or a set of longest paths based on the estimates over τ iterations.

It might seem mysterious that we can re-construct edge weights (delays, for the case of timing analysis) on the whole graph based a single number, which is the total length of the path we chose. To achieve this, our method exploits the power of randomization and a careful choice of a representative set of paths. The latter choice is discussed next.

4.1 Focusing on a Barycentric Spanner

It is well-known in the game-theoretic study of path prediction that any deterministic strategy against an adaptive adversary will fail [Cesa-Bianchi and Lugosi 2006]. Therefore, the algorithm we present below is randomized. As we only observe the entire length of the path we choose, we must select a set of paths *covering* all (reachable) edges of the graph or else we risk missing a highly time-consuming edge. However, simply covering the graph – which corresponds to statement coverage in a program – is not enough, since timing, in general, is a *path property* requiring covering all ways of reaching a statement. Indeed, a

---

Algorithm 1 Finding a 2-Barycentric Spanner

1: $(b_1, \ldots, b_m) \leftarrow (\mathbf{e}_1, \ldots, \mathbf{e}_m)$.
2: **for** $i = 1$ to m **do** {{Compute a basis of $\mathcal{P}$}}
3:     $b_i \leftarrow \arg\max_{x \in \mathcal{P}} |\det(B_{x,i})|$
4: **end for**
5: **while** $\exists x \in \mathcal{P}$, $i \in \{1, \ldots, m\}$ satisfying
    $|\det(B_{x,i})| > 2|\det(B)|$ **do** {{Transform B into a 2-barycentric spanner}}
6:     $b_i \leftarrow x$
7: **end while**

---

key feature of the algorithm we propose is the ability to exploit correlations between paths to guarantee that we find the longest. Hence, we need a *barycentric spanner* (introduced by Awerbuch and Kleinberg [2004]), a set of up to m paths with two valuable properties: (i) any path in the graph can be written as a linear combination of the paths in the spanner, and (ii) the coefficients in this linear combination are bounded in absolute value. The first requirement says that the spanner is a good representation for the exponentially-large set of possible paths; the second says that lengths of some of the paths in the spanner will be of the same order of magnitude as the length of the longest path. These properties enable us to repeatedly sample from the barycentric spanner and reconstruct delays on the whole graph. We then employ concentration inequalities[2] to prove that these reconstructions, on average, converge to the true path lengths. Once we have a good statistical estimate of the true weights on all the edges, it only remains to run a longest-path algorithm for weighted directed acyclic graphs (LONGEST-PATH), subject to path feasibility constraints.

A set of b paths $\{b_1, \ldots, b_b\} \subseteq \mathcal{P}$ is called 2-barycentric if any path $x \in \mathcal{P}$ can be written as $x = \sum_{i=1}^{b} \alpha_i b_i$ with $|\alpha_i| \leq 2$. Awerbuch and Kleinberg [2004] provide a procedure to find a 2-barycentric spanner set whenever the set of paths $\mathcal{P}$ spans m dimensions (see also [McMahan and Blum 2004]). We exhibit the algorithm (see Algorithm 1) for completeness, as this is a necessary pre-processing step for our main algorithm. It is shown [Awerbuch and Kleinberg 2004] that the running time of Algorithm 1 is quadratic in m.

In Algorithm 1, $B = [b_1, \ldots, b_m]^\mathsf{T}$ and $B_{x,i} = [b_1, \ldots, b_{i-1}, x, b_{i+1}, \ldots, b_m]^\mathsf{T}$. B is initialized so that its ith row is e_i, the standard basis vector with 1 in the ith position and 0s elsewhere. The output of the algorithm is the final value of B, a 2-barycentric spanner. The ith iteration of the for-loop in lines 2-4 attempts to replace the ith element of the standard basis with a path that is linearly independent of the previous $i - 1$ paths identified so far and with all remaining standard basis vectors.[3] Line 3 of the algorithm corresponds to maximizing a linear function over the set $\mathcal{P}$, and can be solved using LONGEST-PATH.[4] At the end of the for-loop, we are left with a basis of $\mathcal{P}$ that is not necessarily a 2-barycentric

---

[2]Concentration inequalities are sharp probabilistic guarantees on the deviation of a function of random variables from its mean.
[3]Linear independence is maintained because we maximize the determinant, and because the determinant starts out non-zero.
[4]The LONGEST-PATH algorithm is the standard longest-path algorithm on a weighted DAG, which runs in linear time. In practice, to compute feasible basis paths one must add constraints that rule out infeasible paths, as is standard in integer programming formulations for timing analysis [Li and Malik 1999]; in this case, the longest-path computation is solved as an integer linear program and SMT solving is used to check feasibility of each generated basis path.

spanner. Lines 5-7 of the algorithm refine this basis into a 2-barycentric spanner using the same LONGEST-PATH optimization oracle that is used in the for-loop. One can intuitively view the determinant computation as computing the volume of the corresponding polytope. Maximizing the determinant amounts to spreading the vertices of the polytope as far as possible in order to obtain a "diverse" set of basis paths.

In general, the number of basis paths b is less than m. Gyorgy et al. [2007] extend the above procedure to the case where the set of paths spans only a b-dimensional subspace of $\mathbb{R}^m$. It is also possible to use a slightly different path representation where instead of introducing a 0-1 variable to represent each edge, one only introduces variables to encode branch points; in this case, it is possible to use Algorithm 1 directly on this representation, replacing m everywhere in the algorithm description with b. In this case, the final B is a $b \times m$ binary matrix. We define the Moore-Penrose pseudo-inverse of B as $B^+ = B^\mathsf{T}(BB^\mathsf{T})^{-1}$. It holds that $BB^+ = I_b$.

We also introduce some useful notation. For theoretical analysis, let M be any upper bound on the length of any basis path. (Note that M can be a very loose upper bound.) Since we have assumed an adaptive adversary that produces w_t based on our previous choices $x_1 \ldots x_{t-1}$ as well as the random factors $\pi^x_1 \ldots \pi^x_{t-1}$, we should take care in dealing with expectations. To this end, we denote as $\mathbb{E}_t[A]$ the conditional expectation $\mathbb{E}[A|i_1, \ldots, i_{t-1}, \pi^x_1, \ldots, \pi^x_{t-1}]$, keeping in mind that randomness at time t in the general model stems from our random choice i_t of the basis path *and* the adversary's random choice π^x_t given i_t. In the simplified model, all randomness is due to our choice of the basis path, and this makes the analysis easier to present. In the general model, the adversary can vary the distribution of π^x_t according to the path chosen by the Player.

4.2 Analysis under the Simplified Model

We now present the GAMETIME algorithm and analyze its effectiveness under the simplified model presented in Section 3.2.2. Recall that in this model the perturbation vectors at all time points are identically 0, viz., $\pi^x_t(x) = 0$ for all t and x. The practical interpretation of this model is that the execution times of each basic block (edge) is completely determined by the starting environment (hardware) state.

Consider Algorithm 2. Lines 4-6 describe the game interaction that we outlined earlier in this section, so we focus on the remainder of the algorithm. Line 7 involves assembling a vector of path lengths from a single path length, where the i_tth element of $\tilde{v}_t$ is $b\ell_t$, representing the length of path b_{i_t}, while all other basis path lengths are approximated as 0 (since they were not executed). Line 8 then uses $\tilde{v}_t$ to estimate edge weights that could have generated the path length ℓ_t. In Line 10, we provide a procedure to predict the longest path from the measurements taken over τ rounds; our theoretical guarantees in Theorem 4.2 are for this procedure.

EXAMPLE 4.1. *We illustrate the inner loop of Algorithm 2 with an example. Consider the graph in Fig. 1. Algorithm 2 will sample uniformly at random from the set $\{x_1, x_2, x_3\}$. Suppose, for simplicity, that the environment picks the same w_t at each round t. Thus, suppose that when we sample x_1, we observe length $\ell_1 = 15$, for x_2 we get $\ell_2 = 5$ and for x_3 we get $\ell_3 = 10$. Then, in Line 10 of the algorithm, the cumulative estimated weight vector is generated such that $\tilde{w}_{avg} = \frac{1}{\tau}\sum_{t=1}^{\tau}\tilde{w}_t$ where $\tilde{w}_{avg}(i) = 2.5$ for $i \in \{1,6,7,10\}$ and $\tilde{w}_{avg}(i) = 0$ for other edge indices i. Using this estimated weight vector, we compute the weight of path x_4 to be 10. This is as expected, since $x_4 = x_1 + x_2 - x_3$ and w_t is constant,*

Algorithm 2 GAMETIME with simplified environment model

1: Input $\tau \in \mathbb{N}$
2: Compute a 2-barycentric spanner $\{b_1, \ldots, b_b\}$
3: **for** $t = 1$ to τ **do**
4:     Environment chooses w_t.
5:     We choose $i_t \in \{1, \ldots, b\}$ uniformly at random.
6:     We predict the path $x_t = b_{i_t}$ and observe the path length $\ell_t = b_{i_t}^\mathsf{T} w_t$
7:     Estimate $\tilde{v}_t \in \mathbb{R}^b$ as $\tilde{v}_t = b\ell_t \cdot \mathbf{e}_{i_t}$, where $\{\mathbf{e}_i\}$ denotes the standard basis ($\mathbf{e}_i$ has 1 in the ith position and 0s elsewhere).
8:     Compute estimated weights $\tilde{w}_t = B^+ \tilde{v}_t$
9: **end for**
10: Use the obtained sequence $\tilde{w}_1 \ldots \tilde{w}_\tau$ to find a longest path(s). For example, for Theorem 4.2, we compute $x_\tau^* := \arg\max_{x \in \mathcal{P}} x^\mathsf{T} \sum_{t=1}^\tau \tilde{w}_t$.

---

yielding $x_4^\mathsf{T} w_t = \ell_1 + \ell_2 - \ell_3$.

We begin by proving some key properties of the algorithm.

<u>Preliminaries</u>

The following Lemma is key to proving that Algorithm 2 performs well. It quantifies the deviations of our estimates of the delays on the whole graph, $\tilde{w}_t$, from the true delays w_t, which we cannot observe.

LEMMA 4.1. *For any $\delta > 0$, with probability at least $1 - \delta$, for all $x \in \mathcal{P}$,*

$$\left| \frac{1}{\tau} \sum_{t=1}^\tau (\tilde{w}_t - w_t)^\mathsf{T} x \right| \leq \tau^{-1/2} c \sqrt{2b + 2\ln(2\delta^{-1})}, \tag{8}$$

where $c = 4bM$.

PROOF. We will show that $\mathbb{E}_t \tilde{w}_t x = w_t x$ for any $x \in \mathcal{P}$, i.e. the estimates are unbiased[5] *on the subspace spanned by* $\{b_1, \ldots, b_b\}$. By working directly in the subspace, we obtain the required probabilistic statement and will have the dimensionality of the subspace b, not m, entering the bounds.

Define $v_t = B w_t$ just as $\tilde{v}_t = B \tilde{w}_t$. Taking expectations with respect to i_t, conditioned on $i_1, \ldots, i_{t-1}$,

$$\mathbb{E}_t \tilde{v}_t = \mathbb{E}_t \left[b(b_{i_t}^\mathsf{T} w_t) \cdot \mathbf{e}_{i_t} \right] = \frac{1}{b} \sum_{i=1}^b b(b_i^\mathsf{T} w_t) \cdot \mathbf{e}_i = B w_t = v_t.$$

Fix any $\alpha \in \{-2, 2\}^b$. We claim that the sequence $Z_1, \ldots, Z_\tau$, where $Z_t = \alpha^\mathsf{T}(\tilde{v}_t - v_t)$ is a bounded martingale difference sequence. Indeed, $\mathbb{E}_t Z_t = 0$ by the previous argument. A bound on the range of the random variables Z_t can be computed by observing

$$|\alpha^\mathsf{T} \tilde{v}_t| = |\alpha^\mathsf{T}[b(b_{i_t}^\mathsf{T} w_t)\mathbf{e}_{i_t}]| \leq 2b|b_{i_t}^\mathsf{T} w_t| \leq 2bM \qquad \text{and} \qquad |\alpha^\mathsf{T} v_t| \leq 2bM,$$

implying $|Z_t| \leq 4bM \doteq c$. An application of Azuma-Hoeffding inequality (see Appendix)

---

[5]For random variables X and $\tilde{X}$, $\tilde{X}$ is said to be an unbiased estimate of X if $\mathbb{E}[X - \tilde{X}] = 0$.

for a martingale difference sequence yields, for the fixed α,

$$\Pr\left(\left|\sum_{t=1}^{\tau} Z_t\right| > c\sqrt{2\tau\ln(2(2^b)\delta^{-1})}\right) \leq \delta/2^b.$$

Having proved a statement for a fixed α, we would like to apply the union bound[6] to arrive at the corresponding statement for any $\alpha \in [-2,2]^b$. This is implausible as the set is uncountable. However, applying a union bound over the *vertices* of the hypercube $\{-2,2\}^b$ is enough. Indeed, if $|\sum_{t=1}^{\tau} Z_t| = |\alpha^{\mathsf{T}}\sum_{t=1}^{\tau}(\tilde{v}_t - v_t)| \leq \xi$ for all vertices of $\{-2,2\}^b$, then immediately $|\sum_{t=1}^{\tau} Z_t| \leq \xi$ for any $\alpha \in [-2,2]^b$ by linearity. Thus, by union bound,

$$\Pr\left(\forall \alpha \in [-2,2]^b, \ \left|\sum_{t=1}^{\tau} \alpha^{\mathsf{T}}(\tilde{v}_t - v_t)\right| \leq c\sqrt{2\tau b + 2\tau\ln(2\delta^{-1})}\right) \geq 1 - \delta.$$

Any path x can be written as $x^{\mathsf{T}} = \alpha^{\mathsf{T}}B$ for some $\alpha \in [-2,2]^b$. Furthermore, $\tilde{w}_t = B^+\tilde{v}_t$ implies that $x^{\mathsf{T}}\tilde{w}_t = \alpha^{\mathsf{T}}BB^+\tilde{v}_t = \alpha^{\mathsf{T}}\tilde{v}_t$ and $x^{\mathsf{T}}w_t = \alpha^{\mathsf{T}}v_t$. We conclude that

$$\Pr\left(\forall x \in \mathcal{P}, \ \left|\sum_{t=1}^{\tau} (\tilde{w}_t - w_t)^{\mathsf{T}}x\right| \leq c\sqrt{2\tau b + 2\tau\ln(2\delta^{-1})}\right) \geq 1 - \delta$$

and the statement follows by dividing by τ. □

Estimating the Set of Longest Paths

Consider the quantity $\frac{1}{\tau}\sum_{t=1}^{\tau} w_t$ which is a vector of weights for edges in the graph, averaged over the true weight vectors selected by the environment in rounds $t = 1, 2, \ldots, \tau$. The quantity $\frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t$ represents the average estimated weight vector over rounds $t = 1, 2, \ldots, \tau$.

With the help of Lemma 4.1, we can now analyze how the longest (or almost-longest) paths with respect to the averaged estimated weights ($\tilde{w}_t$'s), compare to the true averaged longest paths.

Note that in this discussion we are performing worst-case analysis with respect to paths, while considering the sample average for an arbitrary sequence of environment choices. We will later discuss how to extend this result to worst-case analysis over both paths and environment choices.

DEFINITION 4.1. *Define the set of ε-longest paths with respect to the actual delays*

$$S_\tau^\varepsilon = \left\{x \in \mathcal{P} : \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^{\mathsf{T}}x \geq \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^{\mathsf{T}}x' - \varepsilon\right\}$$

and with respect to the estimated delays

$$\tilde{S}_\tau^\varepsilon = \left\{x \in \mathcal{P} : \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^{\mathsf{T}}x \geq \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^{\mathsf{T}}x' - \varepsilon\right\}.$$

In particular, S_τ^0 is the set of longest paths.

---

[6]Also known as Boole's inequality, the union bound says that the probability that at least one of the countable set of events happens is at most the sum of the probabilities of the events, e.g. $\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$.
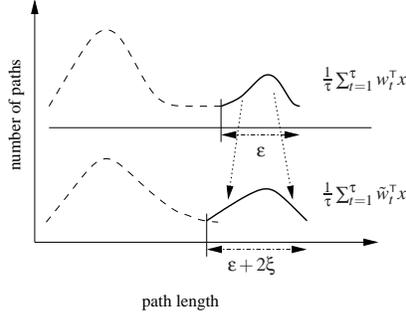
Fig. 3. **Illustration of the second inclusion in Lemma 4.2.** The top curve represents the true path length distribution, while the bottom curve represents the GAMETIME estimate of the distribution. The figure indicates that the true set of ε-longest paths is contained in the set of $(\varepsilon+2\xi)$-longest paths w.r.t. to the sequence $\tilde{w}_1, \ldots, \tilde{w}_\tau$. Under a margin assumption, equality between the two sets can be shown, as exhibited by Theorem 4.2.

The following Lemma makes our intuition precise: with enough trials τ, the set of longest paths, which we can calculate after running Algorithm 2, becomes almost identical to the true set of longest paths. We illustrate this point graphically in Figure 3: In a histogram of average path lengths, the set of longest paths (the right "bump") is somewhat smoothed when considering the path lengths under the estimated $\tilde{w}_t$'s. In other words, paths might have a slightly different average path length under the estimated and actual weights. However, we can still guarantee that this smoothing becomes negligible for large enough τ, enabling us to locate the longest paths.

LEMMA 4.2. *For any $\varepsilon \geq 0$ and $\delta > 0$, and for $\xi = \tau^{-1/2} 4bM\sqrt{2b + 2\ln(2\delta^{-1})}$,*

$$\tilde{S}_\tau^\varepsilon \subseteq S_\tau^{\varepsilon+2\xi} \quad and \quad S_\tau^\varepsilon \subseteq \tilde{S}_\tau^{\varepsilon+2\xi}$$

with probability at least $1 - \delta$.

PROOF. Let $x \in \tilde{S}_\tau^\varepsilon$ and $y \in S_\tau^0$. Suppose that we are in the $(1-\delta)$-probability event of Lemma 4.1. Then

$$\frac{1}{\tau}\sum_{t=1}^\tau w_t^\mathsf{T} x \geq \frac{1}{\tau}\sum_{t=1}^\tau \tilde{w}_t^\mathsf{T} x - \xi \geq \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^\tau \tilde{w}_t^\mathsf{T} x' - \varepsilon - \xi$$

$$\geq \frac{1}{\tau}\sum_{t=1}^\tau \tilde{w}_t^\mathsf{T} y - \varepsilon - \xi \geq \frac{1}{\tau}\sum_{t=1}^\tau w_t^\mathsf{T} y - \varepsilon - 2\xi$$

$$= \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^\tau w_t^\mathsf{T} x' - \varepsilon - 2\xi,$$

where the first and fourth inequalities follow by Lemma 4.1, the third inequality is by definition of maximum, and the second and fifth are by definitions of $\tilde{S}_\tau^\varepsilon$ and S_τ^0, resp. Since the sequence of inequalities holds for any $x \in \tilde{S}_\tau^\varepsilon$, we conclude that $\tilde{S}_\tau^\varepsilon \subseteq S_\tau^{\varepsilon+2\xi}$. The other direction of inclusion is proved analogously. □

Note that $\xi \to 0$ as $\tau \to \infty$. To compute the set S_τ^0, we can instead compute the set $\tilde{S}_\tau^{2\xi}$ that contains it. If $|\tilde{S}_\tau^{2\xi}| \leq k$, for some parameter k, then we can use an algorithm that

computes the k longest paths (see, e.g., [Eppstein 1998]) to find this set.

The Unique Longest Path Assumption: Worst-Case Analysis For Environment Choices

While Lemma 4.2 is very general, it holds for average-case analysis over environment choices. In order to perform worst-case analysis for environment choices, we consider the implication for finding a longest path under the following assumption.

ASSUMPTION 4.1. *There exists a single path x^* that is the longest path on any round with a certain (known) margin ρ:*

$$\forall x \in P, \ x \neq x^*, \ \forall t, \ (x^* - x)^\mathsf{T} w_t > \rho$$

Note that if there is a unique longest path (for any margin $\rho \geq 0$), then we can see that

$$x^* = \arg\max_{x \in \mathcal{P}} \frac{1}{\tau} \sum_{t=1}^{\tau} w_t^\mathsf{T} x = \arg\max_{x \in \mathcal{P}} \max_{t=1..\tau} w_t^\mathsf{T} x$$

In other words, the longest path with respect to averaged environment choices coincides with that with respect to the worst-case environment choice.

Under the above margin assumption, we can, in fact, recover the longest path, as shown in the next Theorem.

Theorem 4.2. *Suppose Assumption 4.1 holds with $\rho > 0$, and we run Algorithm 2 for $\tau = (8bM)^2 \rho^{-2}(2b + 2\ln(2\delta^{-1}))$ iterations. Then the output*

$$x_\tau^* := \arg\max_{x \in \mathcal{P}} x^\mathsf{T} \sum_{t=1}^{\tau} \tilde{w}_t$$

of Algorithm 2 is equal to x^* with probability at least $1 - \delta$.

PROOF. Let $x_\tau^* = \arg\max_{x \in \mathcal{P}} x^\mathsf{T} \sum_{t=1}^{\tau} \tilde{w}_t$. We claim that, with probability $1 - \delta$ it is equal to x^*. Indeed, suppose $x_\tau^* \neq x^*$. By Lemma 4.2, $x_\tau^* \in \tilde{\mathcal{S}}_\tau^0 \subseteq \mathcal{S}_\tau^{2\xi}$ with probability at least $1 - \delta$. Thus,

$$\frac{1}{\tau} \sum_{t=1}^{\tau} w_t^\mathsf{T} x_\tau^* \geq \frac{1}{\tau} \sum_{t=1}^{\tau} w_t^\mathsf{T} x^* - 2\xi.$$

Assumption 4.1, however, implies that

$$\frac{1}{\tau} \sum_{t=1}^{\tau} w_t^\mathsf{T} x_\tau^* < \frac{1}{\tau} \sum_{t=1}^{\tau} w_t^\mathsf{T} x^* - \rho$$

leading to a contradiction whenever $\rho \geq 2\xi = \tau^{-1/2} 8bM \sqrt{2b + 2\ln(2\delta^{-1})}$. Rearranging the terms, we arrive at $\tau \geq (8bM)^2 \rho^{-2}(2b + 2\ln(2\delta^{-1}))$, as assumed. We conclude that with probability at least $1 - \delta$, $x_\tau^* = x^*$ and $\{x^*\} = \tilde{\mathcal{S}}_\tau^0 = \mathcal{S}_\tau^{2\xi}$. □

The following weaker assumption also has interesting implications.

ASSUMPTION 4.2. *There exists a path $x^* \in \mathcal{P}$ such that it is the longest path on any round*

$$\forall x \in P, \ \forall t, \ (x^* - x)^\mathsf{T} w_t \geq 0$$

If, after running Algorithm 2 for enough iterations, we find all 2ξ-longest paths (the set $\tilde{S}_\tau^{2\xi}$), Lemma 4.2 guarantees that, under Assumption 4.2, the longest path x^*, which lies in S_τ^0, is one of them with high probability. As discussed earlier, we can use an efficient k-longest paths computation to find a set containing S_τ^0. We can then use this information to repeatedly test the candidate paths in this set to find the worst-case path x^* and estimate its length.

Even if Assumption 4.2 does not hold, one can still find the longest path x^* with respect to worst-case environment choices over $t = 1, 2, \ldots, \tau$, as long as x^* lies in S_τ^ϵ for some ϵ, and the size of S_τ^ϵ is a small number k (e.g. linear in the size of the program); the approach discussed above of computing the k-longest paths will work with high probability. In other words, if the longest path x^* with respect to the worst-case environment choice is *also* amongst the longest paths with respect to averaged environment choices, then we can efficiently find x^*. Preliminary experimental evidence suggests that this situation does hold often in practice.

4.3 Analysis under General Weights-Perturbation Model

We now present an analysis of GAMETIME under the general weight-perturbation model given in Sec. 3. For easy reference, we give the GAMETIME algorithm again below as Algorithm 3 with the new environment model.

As before, let M be any upper bound on the length of any basis path (where now the length includes the perturbation).

In the general model, the environment $\mathcal{E}$ picks a distribution with mean $\mu^x{}_t \in \mathbb{R}^m$, which depends on the algorithm's chosen path x. From this distribution, $\mathcal{E}$ draws a vector of perturbations $\pi^x{}_t \in \mathbb{R}^m$. The vector $\pi^x{}_t$ satisfies the following assumptions:

—*Bounded perturbation:*
  $\|\pi^x{}_t\|_1 \leq N$, where N is a parameter.

—*Bounded mean perturbation of path length:*
  For any path $x \in \mathcal{P}$, $|x^\mathsf{T}\mu^x{}_t| \leq \mu_{\max}$

Note that $\mu^x{}_t$ is a function of the chosen path, and that $\pi^x{}_t$ depends on $\mu^x{}_t$.

Algorithm 3 GAMETIME with general environment model

1: Input $\tau \in \mathbb{N}$
2: Compute a 2-barycentric spanner $\{b_1, \ldots, b_b\}$
3: **for** $t = 1$ to τ **do**
4:     Environment chooses w_t.
5:     We choose $i_t \in \{1, \ldots, b\}$ uniformly at random.
6:     Environment chooses a distribution from which to draw $\pi^x{}_t$, where the mean $\mu^x{}_t$ and support of the distribution satisfies the assumptions given above.
7:     We predict the path $x_t = b_{i_t}$ and observe the path length $\ell_t = b_{i_t}^\mathsf{T}(w_t + \pi^x{}_t)$
8:     Estimate $\tilde{v}_t \in \mathbb{R}^b$ as $\tilde{v}_t = b\ell_t \cdot \mathbf{e}_{i_t}$, where $\{\mathbf{e}_i\}$ denotes the standard basis.
9:     Compute estimated weights $\tilde{w}_t = B^+ \tilde{v}_t$
10: **end for**
11: Use the obtained sequence $\tilde{w}_1 \ldots \tilde{w}_\tau$ to find a longest path(s). For example, for Theorem 4.4, we compute $x_\tau^* := \arg\max_{x \in \mathcal{P}} x^\mathsf{T} \sum_{t=1}^\tau \tilde{w}_t$.

We now state the main lemma for our general model. In this case, we calculate $\tilde{w}_t$ as an estimate of the sum $w_t + \pi^x{}_t$.

LEMMA 4.3. *For any $\delta > 0$, with probability at least $1 - \delta$, for all $x \in \mathcal{P}$,*

$$\left| \frac{1}{\tau} \sum_{t=1}^{\tau} (\tilde{w}_t - w_t - \pi^x{}_t)^\mathsf{T} x \right| \leq (2b+1)\mu_{\max} + \frac{1}{\sqrt{\tau}} \left[c\sqrt{2b + 2\ln(4\delta^{-1})} + d\sqrt{2m + 2\ln(4\delta^{-1})} \right] \tag{9}$$

where $c = 2b(2M + \mu_{\max})$ and $d = N + \mu_{\max}$.

PROOF. The proof is similar to that of Lemma 4.1, so we only highlight the differences here.

$$\begin{aligned}
\mathbb{E}_t \tilde{v}_t &= \mathbb{E}_{i_t} \left\{ \mathbb{E}_{\pi^x{}_t} \left[b(b_{i_t}^\mathsf{T} w_t + b_{i_t}^\mathsf{T} \pi^x{}_t(x_t)) \cdot \mathbf{e}_{i_t} | i_t \right] \right\} \\
&= \frac{1}{b} \sum_{i=1}^{b} b(b_i^\mathsf{T} w_t) \cdot \mathbf{e}_i + \frac{1}{b} \sum_{i=1}^{b} b(b_i^\mathsf{T} \mu_t^{b_i}) \mathbf{e}_i \\
&= B w_t + \mu_t^{\text{basis}}
\end{aligned}$$

where μ_t^{basis} denotes the $b \times 1$ vector of means in which the ith element is $b_i^\mathsf{T} \mu_t^{b_i}$ and each entry is bounded in absolute value by $\mu_{\max}$.

Fix any $\alpha \in \{-2, 2\}^b$. As before, the sequence $Z_1, \ldots, Z_\tau$, where $Z_t = \alpha^\mathsf{T}(\tilde{v}_t - v_t - \mu_t^{\text{basis}})$ is a bounded martingale difference sequence. A bound on the range of the random variables can be computed by observing

$$|\alpha^\mathsf{T} \tilde{v}_t| = |\alpha^\mathsf{T} [b(b_{i_t}^\mathsf{T}(w_t + \pi^x{}_t))\mathbf{e}_{i_t}]| \leq 2b|b_{i_t}^\mathsf{T}(w_t + \pi^x{}_t)| \leq 2bM$$

and

$$|\alpha^\mathsf{T} \mu_t^{\text{basis}}| \leq 2b\mu_{\max}, \quad |\alpha^\mathsf{T} v_t| \leq 2bM$$

implying

$$|Z_t| \leq 2b(2M + \mu_{\max}) \doteq c.$$

Thus, using Azuma-Hoeffding inequality, we can conclude that for any $\delta_1 > 0$, and for fixed α,

$$\Pr\left(\left| \sum_{t=1}^{\tau} Z_t \right| > c\sqrt{2\tau \ln(2(2^b)\delta_1^{-1})} \right) \leq \delta_1/2^b$$

and (skipping a few intermediate steps involving the union bound as before), we finally get

$$\Pr\left(\forall x \in \mathcal{P}, \left| \sum_{t=1}^{\tau} (\tilde{w}_t - w_t)^\mathsf{T} x \right| \leq 2b\tau\mu_{\max} + c\sqrt{2\tau b + 2\tau \ln(2\delta_1^{-1})} \right) \geq 1 - \delta_1. \tag{10}$$

Now consider any fixed $x \in \mathcal{P}$. We claim that the sequence $Y_1, \ldots, Y_\tau$, where $Y_t = x^\mathsf{T} \pi^x{}_t(x) - x^\mathsf{T} \mu^x{}_t$ is also a bounded martingale difference sequence. Clearly, since $\mathbb{E}_t[\pi^x{}_t(x)] = \mu^x{}_t$, $\mathbb{E}_t[Y_t] = 0$. Further, a bound on the range of the random variables can be computed by observing

$$|x^\mathsf{T} \pi^x{}_t(x)| \leq N \quad \text{and} \quad |x^\mathsf{T} \mu^x{}_t| \leq \mu_{\max}.$$

Thus,

$$|Y_t| \le N + \mu_{\max} =: d.$$

An application of Azuma-Hoeffding inequality for the fixed x and for any $\delta_2 > 0$ yields,

$$\Pr\left(\left|\sum_{t=1}^{\tau} Y_t\right| > d\sqrt{2\tau \ln(2(2^m)\delta_2^{-1})}\right) \le \delta_2/2^m.$$

Taking the union bound over all $x \in \mathcal{P}$,

$$\Pr\left(\forall x \in \mathcal{P}, \ \left|\sum_{t=1}^{\tau} x^{\mathsf{T}}(\pi^x_t(x) - \mu^x_t)\right| \le d\sqrt{2\tau m + 2\tau \ln(2\delta_2^{-1})}\right) \ge 1 - \delta_2.$$

Thus, we get

$$\Pr\left(\forall x \in \mathcal{P}, \ \left|\sum_{t=1}^{\tau} x^{\mathsf{T}}\pi^x_t(x)\right| \le \left|\sum_{t=1}^{\tau} x^{\mathsf{T}}\mu^x_t\right| + d\sqrt{2\tau m + 2\tau \ln(2\delta_2^{-1})}\right) \ge 1 - \delta_2$$

and finally

$$\Pr\left(\forall x \in \mathcal{P}, \ \left|\sum_{t=1}^{\tau} x^{\mathsf{T}}\pi^x_t(x)\right| \le \tau\mu_{\max} + d\sqrt{2\tau m + 2\tau \ln(2\delta_2^{-1})}\right) \ge 1 - \delta_2. \quad (11)$$

Setting $\delta_1 = \delta_2 = \frac{\delta}{2}$ in Relations 10 and 11 above and dividing them throughout by τ, we get that, for all $x \in \mathcal{P}$, each of the following two inequalities hold with probability at most $\frac{\delta}{2}$:

$$\frac{1}{\tau}\left|\sum_{t=1}^{\tau} x^{\mathsf{T}}(\tilde{w}_t - w_t)\right| > 2b\mu_{\max} + \tau^{-1/2}c\sqrt{2b + 2\ln(4\delta^{-1})}$$

$$\frac{1}{\tau}\left|\sum_{t=1}^{\tau} x^{\mathsf{T}}\pi^x_t(x)\right| > \mu_{\max} + \tau^{-1/2}d\sqrt{2m + 2\ln(4\delta^{-1})}$$

From the above relations, we can conclude that, for all $x \in \mathcal{P}$, the following inequality holds with probability at least $1 - \delta$

$$\frac{1}{\tau}\left|\sum_{t=1}^{\tau} x^{\mathsf{T}}(\tilde{w}_t - w_t - \pi^x_t(x))\right| \le (2b+1)\mu_{\max} +$$

$$\tau^{-1/2}\left(c\sqrt{2b + 2\ln(4\delta^{-1})} + d\sqrt{2m + 2\ln(4\delta^{-1})}\right) \quad (12)$$

which yields the desired lemma.

□

Estimating Longest Paths for Sample Average over Environment Choices

From Lemma 4.3, we can derive results on estimating the ε-longest paths and the longest path in a manner similar to that employed in Section 4.2. The main difference is that now we view $\frac{1}{\tau}\sum_{t=1}^{\tau}\tilde{w}_t$ as an estimate of $\frac{1}{\tau}\sum_{t=1}^{\tau}(w_t + \pi^x_t)$ rather than of simply $\frac{1}{\tau}\sum_{t=1}^{\tau}w_t$.

Thus, we now define the set $\mathcal{S}_{\tau}^{\varepsilon}$ as follows:

DEFINITION 4.3. *The set of ε-longest paths with respect to the actual delays is*

$$S_\tau^\varepsilon = \left\{ x \in \mathcal{P} : \frac{1}{\tau} \sum_{t=1}^{\tau} (w_t + \pi^x_t)^\mathsf{T} x \geq \max_{x' \in \mathcal{P}} \frac{1}{\tau} \sum_{t=1}^{\tau} (w_t + \pi^x_t(x'))^\mathsf{T} x' - \varepsilon \right\}$$

The definition of the set $\tilde{S}_\tau^\varepsilon$ stays unchanged. The lemma on approximating the sets S by $\tilde{S}$ now becomes the following:

LEMMA 4.4. *For any* $\varepsilon \geq 0$, $\delta > 0$, *and for*

$$\xi = (2b+1)\mu_{\max} + \tau^{-1/2} \left[c\sqrt{2b + 2\ln(4\delta^{-1})} + d\sqrt{2m + 2\ln(4\delta^{-1})} \right],$$

with probability at least $1 - \delta$ *it holds that* $\tilde{S}_\tau^\varepsilon \subseteq S_\tau^{\varepsilon+2\xi}$ *and* $S_\tau^\varepsilon \subseteq \tilde{S}_\tau^{\varepsilon+2\xi}$.

Worst-Case Analysis for Environment Choices

Consider the longest path x^* with respect to worst-case environment choices

$$x^* = \arg\max_{x \in \mathcal{P}} \max_{t=1..\tau} (w_t + \pi^x_t(x))^\mathsf{T} x$$

Under the margin assumption (Assumption 4.1), we can recover the longest path in the general weight-perturbation model with respect to worst-case environment choices, using an identical reasoning as before.

Theorem 4.4. *Suppose Assumption 4.1 holds with* $\rho > (4b+2)\mu_{\max}$, *and we run Algorithm 3 for* $\tau = 8(\rho - (4b+2)\mu_{\max})^{-2} \left(c^2(b + \ln(4\delta^{-1})) + d^2(m + \ln(4\delta^{-1})) \right)$ *iterations. Then the output*

$$x_\tau^* := \arg\max_{x \in \mathcal{P}} x^\mathsf{T} \sum_{t=1}^{\tau} \tilde{w}_t$$

of Algorithm 3 is equal to x^* *with probability at least* $1 - \delta$.

The proofs of Lemma 4.4 and Theorem 4.4 are virtually identical to the corresponding results in Section 4.2, so we omit them here. Also, as in that section, we can also identify the longest path under the weaker Assumption 4.2 by finding the set $\tilde{S}_\tau^{(4b+2)\mu_{\max}}$ containing S_τ^0 and enumerating the paths in it.

Finally, as in the case of the simplified model, if the longest path x^* with respect to worst-case environment choices is *also* amongst the longest paths with respect to average-case analysis of environment choices, then we can efficiently find x^*.

4.4 Logarithmic Regret under Margin Assumption

As a corollary of Theorem 4.2, we obtain a new result in the field of game-theoretic learning, which has to do with the notion of *regret* – the difference between cumulative length of the longest path and that of the paths selected at each t. Formally, the *regret* over T rounds is defined as $R_T = \max_{x^* \in \mathcal{P}} \sum_{t=1}^{T} w_t^\mathsf{T} x^* - \sum_{t=1}^{T} w_t^\mathsf{T} x_t$ where x_t is the path selected at round t in a game that lasts for T rounds [Cesa-Bianchi and Lugosi 2006].

COROLLARY 4.1. *Under the assumptions of Theorem 4.2 and for large enough* T, *if we run Algorithm 2 with* $\delta = T^{-1} \log T$ *for* $\tau(\delta)$ *steps and then use* x_τ^* *for prediction in the remaining* $(T - \tau)$ *rounds, the expected regret is* $O(\log T)$.

PROOF. Algorithm 2 makes a mistake after $\tau(\delta)$ rounds with probability at most δ. In this case, the regret is at most $2bM \cdot T$. With probability at least $1 - \delta$, the algorithm produces the optimal path, resulting in at most $M \cdot \tau$ regret. Thus, the expected regret is bounded by $\mathbb{E}R_T \leq \tau M + 2bM\delta T = O(\log T)$.   □

The significance of this result in learning theory, as noted in Sec. 2, is the reduction of regret in the adversarial "bandit" setting from $O(\sqrt{T})$ to $O(\log T)$ under the addition of Assumption 4.1.

5.   EXPERIMENTAL RESULTS

We have implemented and evaluated our approach for problems in execution time analysis. Our analysis tool, called GAMETIME, can generate an estimate of the execution time profile of the program as well as a worst-case execution time estimate. This section details our implementation and results.

5.1   Implementation

GAMETIME operates in four stages, as described below.

1. Extract CFG. GAMETIME begins by extracting the control-flow graph (CFG) of the real-time task whose WCET must be estimated. This part of GAMETIME is built on top of the CIL front end for C [George Necula et al.]. Our CFG parameters (numbers of nodes, edges, etc.) is thus specific to the CFG representations constructed by CIL. The computed CFG is slightly different from the standard representation in that nodes correspond to the start of basic blocks of the program and edges indicate flow of control, with edges labeled by a conditional or basic block. In our experience, this phase is usually very fast, taking no more than a minute for any of our benchmarks.

2. Compute basis paths. The next step for GAMETIME is to compute the set of basis paths and the B^+ matrix. The basis paths are generated as described in Section 4.1. As noted there, it is also possible to ensure the feasibility of basis paths by the use of integer programming and SMT solving. This phase can be somewhat time-consuming; in our experiments, the basis computation for the largest benchmark (statemate) took about 15 minutes. The implementation of this phase is currently an unoptimized Matlab implementation, and further performance tuning appears possible.

3. Generate program inputs. Given the set of basis paths for the graph, GAMETIME then has to generate inputs to the program that will drive the program's execution down that path. It does this using *constraint-based test generation*, by generating a constraint satisfaction problem characterizing each basis path, and then using a constraint solver based on Boolean satisfiability (SAT). This phase uses the UCLID SMT solver [Bryant et al. 2007] to generate inputs for each path and creates one copy of the program for each path, with the different copies only differing in their initialization functions. For lack of space, we are unable to describe the test generation in detail; we only mention here that it is standard: the code for each path is translated to *static single assignment* (SSA) form, from which each statement is transformed into a corresponding logical formula (e.g., assignments become equalities), and the resulting formula is handed off to an SMT solver. For our experiments, this constraint-based test generation phase was very fast, taking less than a minute for each benchmark.

4. Predict estimated weight vector or longest path. Finally, Algorithm 3 is run with the set of basis paths and their corresponding programs, along with the B^+ matrix. The

number of iterations in the algorithm, τ, depends on the mode of usage of the tool. In the experiments reported below, we used a deterministic cycle-accurate processor simulator from a fixed initial hardware state, and hence τ was set equal to b, since we perform one simulation per basis path. In general, one can perform an exhaustive exploration of starting hardware states (e.g., for hard real-time systems) or sample starting hardware states from a given distribution (e.g., for soft real-time systems). In this situation, one can either pre-compute τ, given a δ, as described in Section 4 or it can be increased gradually while searching for convergence to a single longest path.

The run-time for this phase depends on the execution time of the program, the speed of the hardware simulator (if one is used), and the number of iterations of the loop in Algorithm 3. For our experiments, this run-time was under a minute for all benchmarks.

Given the estimated weights computed at each round, $\tilde{w}_1, \tilde{w}_2, \ldots, \tilde{w}_\tau$, we can compute the cumulative estimated weight vector $\frac{1}{\tau} \sum_{t=1}^{\tau} \tilde{w}_t$, and use this to predict the length of any path in the program. In particular, we can predict the longest path, and its corresponding length. Given the predicted longest path x_τ^*, we check its feasibility with an SMT solver; if it is feasible, we compute a test case for that path, but if it is not feasible, we generate the next longest path. The process is repeated until a feasible longest path is predicted. The predicted longest path can be executed (or simulated) several times to calculate the desired timing estimate.

5.2   Benchmarks

Our benchmarks were drawn from those used in the *WCET Challenge 2006* [Tan 2006], which were drawn from the Mälardalen benchmark suite [Mälardalen WCET Research Group] and the PapaBench suite [Nemer et al. 2006]. In particular, we used benchmarks that came from real embedded software (as opposed to toy programs), had non-trivial control flow, and did not require automatic estimation of loop bounds. The latter criterion ruled out, for example, benchmarks that compute a discrete cosine transform or perform data compression, because there is usually just one path through those programs (going through several iterations of a loop), and variability in run-time usually only comes from characteristics of the data. Most benchmarks in the Mälardalen suite are of this nature.

The main characteristics of the chosen benchmarks is shown in Table I. The first three benchmarks, altitude, stabilisation, and climb_control, are tasks in the open source PapaBench software for an unmanned aerial vehicle (UAV) [Nemer et al. 2006]. The last benchmark, statemate, is code generated from a STATEMATE Statecharts model for an automotive window control system. Note in particular, how the number of basis paths b is far less than the total number of source-sink paths in the CFG. (We are able to efficiently count the number of paths as the CFG is a DAG.) We also indicate the number of lines of code for each task; however, note that this is an imprecise metric as it includes declarations, comment lines, and blank lines – the CFG size is a more accurate representation of size.

5.3   Worst-Case Execution Time Analysis

We have compared GAMETIME with two leading tools for WCET analysis: Chronos [Li et al. 2005] and aiT [AbsInt Angewandte Informatik]. These tools are based on models crafted for particular architectures, and are designed to generate conservative (over-approximate) WCET bounds. Although GAMETIME is not guaranteed to generate an upper bound on the WCET, we have found that it is possible for GAMETIME to produce larger estimates. We also show that GAMETIME does significantly better than simply testing the

Name	LOC	Size of CFG		Total Num.	Num. of basis
		n	m	of paths	paths b
altitude	12	12	16	11	6
stabilisation	48	31	39	216	10
climb_control	43	40	56	657	18
statemate	916	290	471	7×10^{16}	183

Table I. **Characteristics of Benchmarks.** "LOC" indicates number of lines of C code for the task. The Control-Flow Graph (CFG) is constructed using the CIL front end, n is the number of nodes, m is the number of edges.

programs with inputs generated uniformly at random.

5.3.1 *Comparison with Chronos and Random Testing.* We performed experiments to compare GAMETIME against Chronos [Li et al. 2005] as well as against testing the programs on randomly-generated inputs. WCET estimates are output in terms of the number of CPU cycles taken by the task to complete in the worst case.

Chronos is built upon SimpleScalar [Todd Austin et al.], a widely-used tool for processor simulation and performance analysis. Chronos extracts a CFG from the binary of the program (compiled for MIPS using modified SimpleScalar tools), and uses a combination of dataflow analysis, integer programming, and manually constructed processor behavior models to estimate the WCET of the task.

To compare GAMETIME against Chronos, we used SimpleScalar to simulate, for each task, each of the extracted basis paths. We used the same SimpleScalar processor configuration as was used with Chronos, viz., Chronos' default SimpleScalar configuration. This configuration is given below:

```
-cache:il1 il1:16:32:2:l -mem:lat 30 2 -bpred 2lev -bpred:2lev 1 128 2 1 -decode:width
1 -issue:width 1 -commit:width 1 -fetch:ifqsize 4 -ruu:size 8
```

The parameters listed above use SimpleScalar's syntax for specifying processor parameters. For example, `-bpred 2lev` corresponds to 2-level branch prediction, `-issue:width 1` corresponds to an instruction issue width of 1, and `-fetch:ifqsize 4` corresponds to an instruction fetch queue of size 4. Further details are available in the SimpleScalar documentation.

Since SimpleScalar's execution is deterministic for a fixed processor configuration, we did not run Algorithm 3 in its entirety. Instead, we simulated each of the basis paths exactly once to obtain the execution time estimate for that path and then predicted a feasible longest path as described in Section 5.1. A test case was generated for the predicted longest path. The predicted longest path was then simulated once and its execution time is reported as GAMETIME's WCET estimate. Note that the availability of a test case implies that this is an actual execution time that the program can exhibit on the target platform.

Random testing was done by generating initial values for each program input variable uniformly at random from its domain. For each benchmark, we generated 500 such random initializations; note that GAMETIME performs significantly fewer simulations (no more than the number of basis paths, for a maximum of 183 for the statemate benchmark).

Our results are reported in Table II. We note that the estimate of GAMETIME T_g is lower than the WCET T_c reported by Chronos for three out of the four benchmarks. Interestingly, $T_g > T_c$ for the stabilisation benchmark; on closer inspection, we found that this occurred mainly because the number of misses in the instruction cache was significantly underestimated by Chronos. The over-estimation by Chronos for statemate is very large,

*   S. A. Seshia and A. Rakhlin

Name of Benchmark	Chronos WCET T_c	Random testing T_r	GAMETIME estimate T_g	$\frac{T_c - T_g}{T_g}$ (%)	Basis path times Max	Min
altitude	567	175	348	62.9	343	167
stabilisation	1379	1435	1513	−8.9	1513	1271
climb_control	1254	646	952	31.7	945	167
statemate	8584	4249	4575	87.6	3735	3235

Table II. **Comparison with Chronos and random testing.** Execution time estimates are in number of cycles reported by SimpleScalar. For random testing, the maximum cycle count over 500 runs is reported. The fifth column indicates the percentage over-estimation by Chronos over GAMETIME, and the last two columns indicate the maximum and minimum cycle counts for basis paths generated by GAMETIME.

Name of Benchmark	aiT WCET bound	GAMETIME estimate
altitude	251	126
stabilisation	926	583
statemate	4063	2555

Table III. **Comparison with aiT.** Execution time estimates are in number of cycles for the ARM7TDMI processor.

much larger than for altitude and climb_control. This appears to arise from the fact that the number of branch mis-predictions estimated by Chronos is significantly larger than that actually occurring: 106 by Chronos versus 19 mis-predictions on the longest path simulated by GAMETIME in SimpleScalar. In fact, the number of branches performed in a single loop of the statemate code is bounded by approximately 40.

We also note that GAMETIME's estimates can be significantly higher than those generated by random testing. Moreover, GAMETIME's predicted WCET is higher than the execution time of any of the basis paths, indicating that the basis paths taken together provide more longest path information than available from them individually.

5.3.2 *Comparison with aiT.* We have also run experiments with aiT [AbsInt Angewandte Informatik], one of the leading industrial tools for WCET analysis. Separate versions are available for certain processors and compilers. Our experiments used the version of aiT for the ARM7TDMI processor for use with binaries generated by both ARM and GNU compilers. For the experiments reported in this section, we used the GNU C compiler provided by the SimpleScalar toolset [Todd Austin et al.] to generate ARM binaries. Measurements for GAMETIME's analysis were generated using a version of the SimpleScalar-ARM simulator configured for the ARM7TDMI. The results are shown in Table III. Since the ARM7TDMI does not have a floating-point unit, we had to modify the altitude and stabilisation programs to use `int` instead of `float` types; such a modification was not possible for the climb_control task without substantially changing the code, so we omitted this benchmark from the comparison. We observe that aiT generates larger bounds. Since aiT is a closed industrial tool, unlike in the case of Chronos, we do not have a clear explanation of the results and an analysis of potential reasons for over-estimation. We note also that the execution time measurements used in aiT and GAMETIME are made using different methods, which could influence the relative comparison. A more detailed experimental comparison is left to future work.

ACM Transactions in Embedded Computing Systems, Vol. v, No. n, mm 20yy.

5.3.3 *Discussion.* As noted earlier, our theoretical guarantees are only for the finite-horizon WCET problem. Moreover, they are probabilistic guarantees, rather than absolute statements. Tools such as aiT and Chronos are designed to solve the global WCET problem for specific platforms. Naturally, the reader might wonder about the suitability of GAME-TIME versus tools that are designed to obtain a conservative upper bound for a specific platform, and for hard real-time systems versus soft real-time systems.

Consider first the case of hard real-time systems. If a conservative upper bound is required *and* an accurate model is available for the hardware platform, then tools that are formally proved to generate such an upper bound for that model are the right tools for the problem. However, it is often the case in the design of even hard real-time embedded systems that there is no conservative tool that supports exactly the hardware platform that the designers want to use. In such a case, GAMETIME provides a suitable alternative, especially in early-stage design space exploration, and when designers have complete knowledge of the environment states from which the program is to be executed. As we have observed in our experiments, it is possible for GAMETIME to generate larger execution times in some cases than the bounds predicted by a conservative tool. Inaccuracies or slight mismatches between the platform model and the actual platform are likely to be the reason. Importantly, GAMETIME is *easy to port* to new platforms and seeks to avoid modeling errors by directly executing on the target platform. Moreover, even when a conservative WCET tool is available for the platform, GAMETIME is useful for evaluating the tightness of the generated bound.

For soft real-time systems, GAMETIME is very suitable, since it generates probabilistic guarantees and (as we demonstrate in Section 5.4), can efficiently estimate the distribution of execution times exhibited by a task. Additionally, GAMETIME's ability to generate test cases for basis paths is a useful capability for both hard and soft real-time systems.

A very interesting direction would be to combine the capabilities of GAMETIME with that of a conservative WCET tool such as aiT, when an accurate platform model is available, so as to reduce the over-estimation of WCET. GAMETIME could be used to generate inputs corresponding to basis paths. These inputs can be used by a conservative tool to estimate the WCET for those specific program paths. GAMETIME can then use the resulting estimates to predict the true longest path (or set of paths), from which the conservative tool can generate a final WCET estimate. A further benefit of such estimation would also be to extend GAMETIME's capabilities for reasoning about data-dependent timing.

5.4   Estimating the Full Timing Profile of a Program

One of the unique aspects of GAMETIME is the ability to predict the *execution time profile of a program* — the distribution of execution times over program paths — by only measuring times for a linear number of basis paths, as formalized in Lemma 4.3.

To experimentally validate this ability, we performed experiments with a complex processor architecture – the StrongARM-1100 – which implements the ARM instruction set with a complex pipeline and both data and instruction caches. The SimIt-ARM cycle-accurate simulator [Qin and Malik] was used in these experiments.

In our experiments, we first executed each basis path generated by GameTime on the SimIt-ARM simulator and generated the averaged estimated weight vector $\tilde{w}_{avg} = \frac{1}{b} \sum_{t=1}^{b} \tilde{w}_t$. Using this estimated weight vector as the weights on edges in the CFG, we then efficiently computed the estimated length of each path x in the CFG as $x \cdot \tilde{w}_{avg}$ using dynamic pro-

gramming. [7] We also exhaustively enumerated all program paths for the small programs in our benchmark set, and simulated each of these paths to compute its execution time.

For the altitude program, the histogram of execution times generated by GAMETIME perfectly matched the true histogram generated by exhaustively enumerating program paths.

For the climb_control task, the GAMETIME histogram is a close match to the true histogram, as can be seen in Figure 4. Out of a total of 657 paths, 129 were found to be feasible; of these, GAMETIME's prediction differs from the true execution time on only 12 paths, but the prediction is never off by more than 40 cycles, and is always an over-approximation.
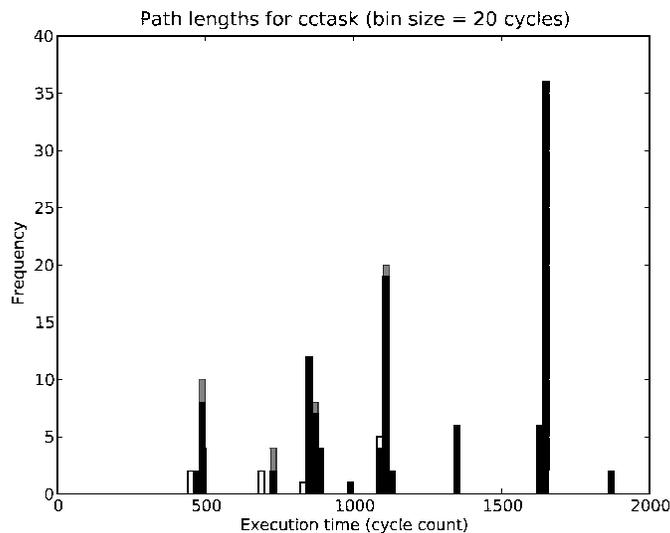

Fig. 4. **Estimating the distribution of execution times with** GAMETIME**.** The true execution times are indicated by white bars, the predicted execution times by gray bars, and the cases where the two coincide are colored black.

In summary, we have found GAMETIME to be an adequate technique to estimate not just the WCET, but also the distribution of execution times of a program, for even complex microprocessor platforms. A key aspect of GAMETIME's effectiveness has been the generation of tests for basis paths. We have also experimented with other coverage metrics such as statement or branch coverage, but these do not yield the same level of accuracy as does basis path coverage. Full path coverage is very difficult to achieve for programs that exhibit path explosion (e.g., statemate), whereas basis path coverage remains tractable.

6.   CONCLUSIONS

We have presented a new, game-theoretic approach to estimating quantitative properties of a software task, such as its execution time profile and worst-case execution time (WCET).

---

[7]The histogram of path lengths in a weighted DAG can be generated by traversing the DAG "bottom-up" from the sink, to compute, for each node, the lengths of all paths from that node to the sink. Such computation can be efficiently performed in practice using dynamic programming.

Our tool, GAMETIME, is measurement-based, making it easy to use on many different platforms without the need for tedious processor behavior analysis. We have presented both theoretical and experimental evidence for the utility of the GAMETIME approach for quantitative analysis, in particular for timing estimation.

We note that our algorithm and results of Section 4 are general, in that they apply to estimating longest paths in DAGs by learning an environment model, not just to timing estimation for embedded software. In particular, one could apply the algorithms presented in this paper to power estimation of software systems and to quantitative analysis of many systems that can be modeled as a DAG, such as combinational circuits and distributed embedded and control systems.

Acknowledgments

REFERENCES

ABSINT ANGEWANDTE INFORMATIK. aiT worst-case execution time analyzers. `http://www.absint.com/ait/`.

AUER, P., CESA-BIANCHI, N., FREUND, Y., AND SCHAPIRE, R. E. 2003. The nonstochastic multiarmed bandit problem. *SIAM J. Comput. 32,* 1, 48–77.

AWERBUCH, B. AND KLEINBERG, R. D. 2004. Adaptive routing with end-to-end feedback: distributed learning and geometric approaches. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 45–53.

BARRETT, C., SEBASTIANI, R., SESHIA, S. A., AND TINELLI, C. 2009. Satisfiability modulo theories. In *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. Vol. 4. IOS Press, Chapter 8.

BRYANT, R. E., KROENING, D., OUAKNINE, J., SESHIA, S. A., STRICHMAN, O., AND BRADY, B. 2007. Deciding bit-vector arithmetic with abstraction. In *TACAS*. LNCS, vol. 4424. 358–372.

CESA-BIANCHI, N. AND LUGOSI, G. 2006. *Prediction, Learning, and Games*. Cambridge University Press.

CHAKRABARTI, A., CHATTERJEE, K., HENZINGER, T. A., KUPFERMAN, O., AND MAJUMDAR, R. 2005. Verifying quantitative properties using bound functions. In *Proc. Correct Hardware Design and Verification Methods*. 50–64.

EDWARDS, S. A. AND LEE, E. A. 2007. The case for the precision timed (PRET) machine. In *Design Automaton Conference (DAC)*. 264–265.

EPPSTEIN, D. 1998. Finding the k shortest paths. *SIAM Journal on Computing 28,* 2, 652–673.

GEORGE NECULA ET AL. CIL - infrastructure for C program analysis and transformation. `http://manju.cs.berkeley.edu/cil/`.

GYÖRGY, A., LINDER, T., LUGOSI, G., AND OTTUCSÁK, G. 2007. The on-line shortest path problem under partial monitoring. *J. Mach. Learn. Res. 8*, 2369–2403.

IRANI, S., SINGH, G., SHUKLA, S., AND GUPTA, R. 2005. An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *IEEE Trans. VLSI 13,* 12 (Dec), 1349–1361.

KIRNER, R. AND PUSCHNER, P. 2008. Obstacles in worst-case execution time analysis. In *ISORC*. 333–339.

LEE, E. A. 2007. Computing foundations and practice for cyber-physical systems: A preliminary report. Tech. Rep. UCB/EECS-2007-72, University of California at Berkeley. May.

LI, X., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. 2005. Chronos: A timing analyzer for embedded software. Technical report, National University of Singapore. `http://www.comp.nus.edu.sg/~rpembed/chronos/chronos_tool.pdf`.

LI, Y.-T. S. AND MALIK, S. 1999. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic.

MÄLARDALEN WCET RESEARCH GROUP. The Mälardalen benchmark suite. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

MCCABE, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering 2,* 4, 308–320.

MCMAHAN, H. B. AND BLUM, A. 2004. Online geometric optimization in the bandit setting against an adaptive adversary. In *COLT'04*. 109–123.

NEMER, F., CASS, H., SAINRAT, P., BAHSOUN, J.-P., AND MICHIEL, M. D. 2006. Papabench: A free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*.

QIN, W. AND MALIK, S. Simit-ARM: A series of free instruction-set simulators and micro-architecture simulators. `http://embedded.eecs.berkeley.edu/mescal/forum/2.html`.

REINHARD WILHELM ET AL. 2008. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS) 7,* 3.

ROBBINS, H. 1952. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc. 58,* 5, 527–535.

SESHIA, S. A. AND RAKHLIN, A. 2008. Game-theoretic timing analysis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 575–582.

TAN, L. 2006. The Worst Case Execution Time Tool Challenge 2006: Technical Report for the External Test. Technical Reports of WCET Tool Challenge 1, Uni-DUE. December.

TIWARI, V., MALIK, S., AND WOLFE, A. 1994. Power analysis of embedded software: a first step towards software power minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 384–390.

TODD AUSTIN ET AL. The SimpleScalar tool set. `http://www.simplescalar.com`.

WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. 2008. Measurement-based timing analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*.

WILHELM, R. 2005. Determining Bounds on Execution Times. In *Handbook on Embedded Systems*, R. Zurawski, Ed. CRC Press.

A. AZUMA-HOEFFDING INEQUALITY

The Azuma-Hoeffding inequality is a very useful concentration inequality. A version of this inequality with a slightly better constant is given as Lemma A.7 in [Cesa-Bianchi and Lugosi 2006].

LEMMA A.1. *Let $Y_1, \ldots, Y_T$ be a martingale difference sequence. Suppose that $|Y_t| \leq c$ almost surely for all $t \in \{1, \ldots, \tau\}$. Then for any $\delta > 0$,*

$$\Pr\left(\left|\sum_{t=1}^{\tau} Y_t\right| > \sqrt{2\tau c^2 \ln(2/\delta)}\right) \leq \delta$$

One-sided inequalities for $\sum_{t=1}^{\tau} Y_t$ also hold by replacing $2/\delta$ with $1/\delta$ in the logarithm. The inequality is an instance of the so-called *concentration of measure inequalities*. Roughly speaking, it says that if each random variable fluctuates within the bounds $[-c, c]$, then the sum of these variables fluctuates, with high probability, within $[-c\sqrt{\tau}, c\sqrt{\tau}]$.