# An Empirical Study of Random Sampling Methods for Changing Discrete Distributions

by

Yunpeng Tang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

# Abstract

In this thesis, we focus on finding efficient practical random sampling methods for time-changing discrete distributions. We empirically study ten methods including existing algorithms, and two new ones: three level search and the flat method. We review the core ideas of existing methods including their runtime complexity and correctness. We study how algorithms for static distribution can be adopted to the dynamic case that we find in many scenarios. We also implement all methods in a software package UpdateRandom. All methods were tested in the test platform described in the thesis to make sure they are able to generate random numbers properly according to the distribution and their performance matches the analytical result. In order to measure their performance in practice, we used a linear regression model to determine the cost of reference implementations as a function of the number of random samples, the number of weight updates, and the size of the weight array. With the models and application example we provide, readers can easily find the most efficient methods for any usage scenario.

# Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Martin Müller, who does not only give me valuable guidance and constructive advice, but also detailed comments and feedback throughout the research and writing of this thesis.

I would like to thank my family and friends. It is their great support that helps me get through all these challenges and hard times.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Sampling from a Time-Changing Discrete Distribution

Given a discrete random variable $X$, the discrete distribution of $X$ describes the probability that $X = i$ for each possible discrete value $i$. In this thesis, we consider consecutive integer indices $i \in [0, 1, ..., n-1]$, and use a weight array $W$ containing $n$ non-negative real-valued weights $[w_0, \ldots, w_{n-1}]$ to describe a discrete distribution. During sampling, each index $i$ should be selected with a probability of $p_i = \frac{w_i}{S}$, where $S = \sum_{i=0}^{n-1} w_i$. We assume that $S > 0$ to make the problem meaningful.

A time-changing discrete distribution allows modifying individual weights $w_i$, such that we can gradually change the weight array $W$ over time. Once the size n is specified, we cannot have more than $n$ weights in $W$. However, some (but not all) weights might be zero, which can be used to model a distribution with fewer than $n$ events.

## 1.2 Research Motivation and Problem Setting

In recent years, with the increasing usage of sampling techniques in computer science, such as Monte Carlo methods and sampling in reinforcement learning, the demand of efficient random sampling methods is growing rapidly. For the simple case of random sampling from a fixed discrete distribution, the well known Alias Method [18] is both optimal in theory and efficient in practice.

However, the situation is less clear when it comes to random sampling from a time-changing discrete distribution where the weights in the distribution can be updated between samples. There exist methods [6, 12] with runtime complexity of O(1) for sampling, O(1) for updating and O(n) for constructing the initial distribution. This is optimal in theory. However, the constant factor overhead of these methods is prohibitively high, and they are not used in practice.

While previous work such as [6] used only asymptotic complexity to measure algorithm performance, our goal in this thesis is to find methods that are efficient in practice for both sampling and updating. In order to show the performance trade-offs between different approaches in an informative way, we use linear regression combined with runtime complexity analysis to build a runtime model as $time(s, u, n) = an + bs \cdot f(n) + cu \cdot g(n)$ for each method. In the expression, $n$ is the number of elements in the discrete distribution, $s$ is the number of random samples, $u$ is the number of weight updates, and $a$, $b$, $c$ are corresponding coefficients. The functions $f(n)$ and $g(n)$ are determined by runtime complexity analysis. By comparing these models, a reader can easily determine the most suitable methods for their application.

## 1.3 Machine Dependency

In this thesis, we measure the actual time cost of each method in different scenarios. We build models on the basis of the time we measure. The time cost depends on the hardware and software we use, so the models we build are machine dependent. However, readers with other hardware and software can still refer to our models and conclusions. Although the actual time cost depends on the machine, the major factors that determine the relative strength of a method in different scenarios are the same: the number of samples, the number of updates and the number of elements in the weight array $W$. We compare the methods based on these factors such that our models and conclusions can be generalized to different machines.

## 1.4 Contributions of this Thesis

This thesis contains the following contributions:

- We review the following algorithms for time-changing discrete distribution: linear search, binary search, the Alias Method, two level search, binary tree search, the rejection method, and the log* method.
- We review both the Boost implementation and the GNU Compiler implementation of the C++ *discrete_distribution* random number generator.
- We introduce the core ideas of these methods, as well as analyze their correctness and runtime complexity.
- We derive a practical Flat Method from Hagerup's theoretical work [6].
- We propose three, four and five level search methods based on the existing two level search.
- We implement all methods based on a unified framework in a software package UpdateRandom. The package is available on `https://github.com/yunpeng5/UpdateRandom`.
- We use statistical methods to check that all methods behave as expected.
- We use linear regression models to summarize the performance of each method in the experiments, such that we can compare them in practice.
- We apply our models to compare the practical performance of all methods in two representative scenarios: sampling from a static discrete distribution and sampling from a time-changing discrete distribution where we make one update after each sample.
- We demonstrate how we use the methods and our models to speedup the sampling process in an application to the game of NoGo.

The thesis is organized as follows: In Chapter 2, we review sampling methods, with their core ideas, runtime complexity, proof of correctness and implementation details. In Chapter 3, we introduce our two new methods. In

Chapter 4, we introduce our test platform and the $time(s, u, n)$ models we develop for each method. In Chapter 5, we apply our models to compare the practical performance of all methods and make conclusions. In Chapter 6, we introduce the game of NoGo and demonstrate how we use the sampling methods together with our models to speedup the sampling process. Chapter 7 concludes our work and gives potential future work.

# Chapter 2

# Literature Review

In this chapter, we review existing random sampling methods that are either commonly used or attractive in theory. We talk about the assumptions we make in Section 2.1. Then we introduce the general framework we use in Section 2.2, and review the methods in Sections 2.3 to 2.10.

## 2.1 Assumptions

All methods in the review satisfy our assumptions: they generate samples according to a time-changing weight array $W = [w_0, \ldots, w_{n-1}]$ containing $n$ non-negative real-valued weights. While $n$ is fixed such that we cannot have more than $n$ weights by dynamically extending $W$, we can still have fewer than $n$ weights by having $w_i = 0$ for some $i$. When analyzing the runtime complexity, we assume that all the following standard operations take constant time: arithmetic, logarithms and generating a single random number (either a real number or an integer) from a uniform distribution.

## 2.2 General Framework

To sample from a time-changing discrete distribution, each method needs at least three functions: a construct function that builds the data structures for the method from the given $W$, an $update(index, weight)$ function that updates $w_{index}$ to a new $weight$ and a $sample()$ function that generates a sample. In this chapter, we use a general framework to review each method:

1. We show the data structures that a method uses and its construction procedure.

2. We show how a method generates samples from a discrete distribution.

3. We show how a method updates a weight in a discrete distribution.

### 2.2.1 Utility Functions

Several utility functions are widely used among different sampling methods. We discuss these functions in this section, and we will refer to them in later sections.

The most commonly used utility functions are $uniform\_real(a, b)$ and $uniform\_int(a, b)$. We can use $uniform\_real(a, b)$ to generate a random real number $d \in [a, b)$ and $uniform\_int(a, b)$ to generate a random integer $i \in [a, b]$. These functions are usually specified by a programming language standard such as [8] and provided by a standard library.

Given an array $W = [w_0, w_1, ..., w_{n-1}]$, when we try to find the smallest $i$ satisfying $w_{startIndex} + w_{startIndex+1} + ... + w_i > d$ for a real number $d \in [0, \sum_{i=startIndex}^{n-1} w_i)$ and an integer $startIndex \in [0, n)$, we use the linear search function $linear\_search(W, d, startIndex)$ shown in Algorithm 1.

---

**Algorithm 1:** Linear Search

---

**Input:** Array $W = [w_0, w_1, ..., w_{n-1}]$, real number target $d$ and the $startIndex$ we start searching from

**Output:** The smallest integer $i$ satisfying $w_{startIndex} + ... + w_i > d$

1. $i = startIndex$
2. While $d \geq w_i$
   (a) $d = d - w_i$
   (b) $i = i + 1$
3. return $i$

---

When we try to remove an element from an array, directly erasing it could be time consuming as we need to rearrange all elements after it. If

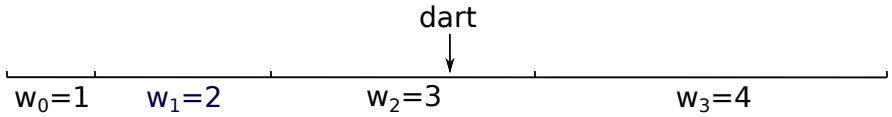



Figure 2.1: In this example, $W = [1, 2, 3, 4]$, and the dart lands in the part corresponding to $w_2 = 3$. The random sampling result is the selected index 2.

we do not care about the order of elements in the array, we can use function $swap\_and\_remove(W, index)$ shown in Algorithm 2 to remove it efficiently.

**Algorithm 2:** Element Removal from Array

**Input:** Array $W = [w_0, w_1, ..., w_{n-1}]$, and $index$ of the element we want to remove

1. $w_{index}, w_{n-1} = w_{n-1}, w_{index}$
2. $W.pop()$

## 2.3 Method 1: Linear Search

To illustrate sampling from a discrete distribution according to a weight array W, an intuitive way is to view W as a belt, on which each element i is expressed by a piece with a length proportional to $w_i$. We can then throw a dart at a random location on the belt and see which part it lands in. The example in Figure 2.1 illustrates this process.

To implement this idea, we need to store the weight array $W$ and calculate the sum of all weights S=$\sum_{i=0}^{n-1} w_i$. We show the initialization of the data structures in Algorithm 3.

To sample, we choose a real number $d \in [0, S)$ uniformly at random. Then we run a O(n) linear search from elements 0 to $n - 1$ to find the smallest $i$ satisfying $w_0 + w_1 + ... + w_i > d$, and return this $i$ as the sample. We show the pseudo code in Algorithm 4.

To update a weight, we only need to update $w_i$ in $W$ and $S$. We show the pseudo code in Algorithm 5.

**Algorithm 3:** Linear Search Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$
**Output:** Sum $S = \sum_{i=0}^{n-1} w_i$

1. $S = 0$
2. For each integer $i \in [0, n)$
   (a) S=S+$w_i$

**Algorithm 4:** Linear Search Sampling

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$ and sum $S = \sum_{i=0}^{n-1} w_i$
**Output:** Sampling result $i$

1. $d = \mathit{uniform\_real}(0, S)$
2. $i = \mathit{linear\_search}(W, d, 0)$
3. return $i$

**Algorithm 5:** Linear Search Updating

**Input:** A new weight $w_i'$

1. $S = S - w_i + w_i'$
2. $w_i = w_i'$

We refer to this method as the linear search method. The time complexity of construction and sampling is O(n), but it only takes constant $\Theta(1)$ time to update a weight.

## 2.4 Method 2: Binary Search

Besides linear search, we can also use binary search. Instead of W, we maintain the cumulative weight array $WS = [ws_0, ..., ws_{n-1}]$ with $ws_i = \sum_{j=0}^{i} w_j$ for $0 \leq i \leq n-1$. We show the initialization of the data structures in Algorithm 6.

**Algorithm 6:** Binary Search Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$
**Output:** Cumulative weight array $WS = [ws_0, ..., ws_{n-1}]$

1. $ws_0 = w_0$
2. For each integer $i \in [1, n)$
   (a) $ws_i = ws_{i-1} + w_i$

To sample, we choose $d \in [0, ws_{n-1})$ uniformly at random, and use binary search to find the smallest i satisfying $w_0 + w_1 + ... + w_i = ws_i > d$. We show the pseudo code in Algorithm 7.

To update a weight $w_i$, we need to update all $ws_j$ with $j \geq i$. We show the pseudo code in Algorithm 8.

By using binary search instead of linear search, we reduce the time complexity of sampling to O(log(n)). However, it takes linear time instead of constant time to update in the worst case.

## 2.5 Method 3: The Alias Method

Vose's Alias method [18] is the state of art method for sampling from a static discrete distribution. It is an implementation of the famous Alias method algorithm [19]. It takes $\Theta$(n) time to preprocess the weight array, but only $\Theta(1)$ time to generate a sample.

**Algorithm 7:** Binary Search Sampling

**Input:** Cumulative weight array $WS = [ws_0, ..., ws_{n-1}]$

**Output:** Sampling result $r$

1. $d = \mathit{uniform\_real}(0, ws_{n-1})$, $l = 0$, $r = n - 1$
2. While $l < r$
   (a) $m = \left\lfloor \frac{l+r}{2} \right\rfloor$
   (b) If $ws_m \leq d$
      (1). $l = m + 1$
   (c) Else
      (1). $r = m$
3. return $r$

**Algorithm 8:** Binary Search Updating

**Input:** A new weight $w'_i$

1. For each integer $j \in [i, n)$
   (a) $ws_j = ws_j - w_i + w'_i$
2. $w_i = w'_i$

We use the following example from Keith Schwarz's online publication [16] to illustrate table construction and sampling processes. Consider a weight array of size 4, such as $W = [6, 4, 1, 1]$. We first scale all weights $w_i$ to generate a new weight array $W' = [2, \frac{4}{3}, \frac{1}{3}, \frac{1}{3}]$, with sum $S' = \sum w'_i = n$, such that the probability of choosing each index $i$ remains the same. Intuitively, we can view each weight $w_i$ as a rectangle with width 1 and height $w_i$ (shown in Figure 2.2 on the left). We split each rectangle into pieces of height at most 1 (shown in Figure 2.2 on the right). Since the average weight $\overline{w} = 1$, we can cut rectangles

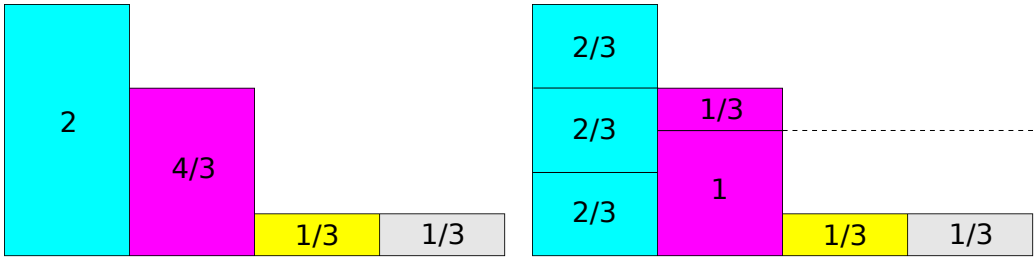



Figure 2.2: Viewing $w_i$ as rectangles and splitting them into small pieces.

and rearrange the pieces such that each of the four columns has a height of exactly 1(shown in Figure 2.3 left). Crucially, this rearrangement can be done

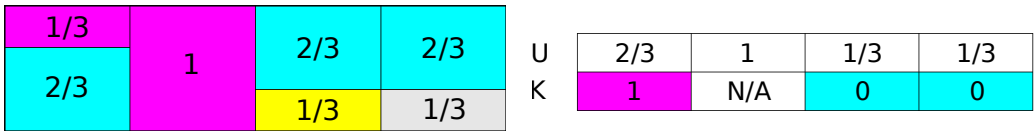



Figure 2.3: The rearrangement of the pieces and the corresponding tables.

in a way such that:

- There are at most two pieces in each column.
- At least one piece of the $i^{th}$ rectangle is placed in the $i^{th}$ column.

We denote the process above as the Alias Method Rearrangement. To sample, we first choose one of the four columns uniformly at random and then choose one piece in the column according to their proportion. Since the $i^{th}$ column contains at most one piece that is not from the $i^{th}$ rectangle, we use $k_i$ in the alias table K to record the original column of that piece. $k_i = -1$ if there is only one piece in the $i^{th}$ column. $u_i$ in table $U$ is the probability of choosing

$i$ when column $i$ is randomly selected, and $k_i$ is chosen with a probability of $1 - u_i$. In the example, the alias table $K = [1, -1, 0, 0]$ and the probability table $U = [\frac{2}{3}, 1, \frac{1}{3}, \frac{1}{3}]$, as shown in Figure 2.3 on the right. For example, if we choose the $0^{th}$ column by sampling, we will output 0 with probability $u_0 = \frac{2}{3}$ and 1 with probability $1 - u_0 = \frac{1}{3}$.

### 2.5.1 Correctness of the Alias Method

To prove that the Alias Method is correct, the key point is to prove that it is always possible to construct the alias and probability tables for an arbitrary weight array $W$. There always exists a way to split and rearrange the rectangles of weights such that the following conditions are met:

- There are at most two rectangles in each column.
- The height of each column is exactly 1.
- One piece of the $i^{th}$ rectangle is placed in the $i^{th}$ column.

We modify the proof procedure at [16] to formalize and prove the key point above.

**Lemma 1.** *For arbitrary weight array $W = [w_0, w_1, ..., w_{n-1}]$, we can scale the weights to form a new array $W' = [w'_0, w'_1, ..., w'_{n-1}]$, such that $S' = \sum_{i=0}^{n-1} w'_i = n$ and $p'_i = \frac{w'_i}{S'} = p_i = \frac{w_i}{S}$ for all $0 \leq i < n$.*

*Proof.* Let $w'_i = \frac{n}{S} w_i$ for all $0 \leq i < n$, then $S' = \sum_{i=0}^{n-1} w'_i = \sum_{i=0}^{n-1} \frac{n}{S} w_i = \frac{n}{S} \sum_{i=0}^{n-1} w_i = \frac{n}{S} S = n$. We also have $p_i = \frac{w'_i}{S'} = \frac{\frac{n}{S} w_i}{n} = \frac{w_i}{S}$ for all $0 \leq i < n$. □

**Lemma 2.** *Given $n$ weights $w_0, w_1, ..., w_{n-1}$ where $n \geq 1$ such that $\sum_{i=0}^{n-1} w_i = n$, there must be at least one weight $w_l \leq 1$ and one weight $w_g \geq 1$.*

*Proof.* By contradiction. Assume that there is no weight $w_l \leq 1$, which means $w_i > 1$ for all $0 \leq i < n$. Then $\sum_{i=0}^{n-1} w_i > n \cdot 1 = n$, which is a contradiction. Similarly, if we assume that there is no weight $w_g \geq 1$, the sum of all weights would be smaller than n, which is also a contradiction. □

Remark: in the proof above, it is possible that $l = g$.

**Theorem 1.** *Given an arbitrary weight array* $W = [w_0, w_1, ..., w_{n-1}]$*, it is always possible to construct the alias and probability tables.*

*Proof.* By Lemma 1, we can replace the weight array $W$ with $W' = [w'_0, w'_1, ..., w'_{n-1}]$, such that $\sum_{i=0}^{n-1} w'_i = n$ and the probability of choosing each index remains the same. Then proving that the theorem holds is equal to proving that the Alias Method Rearrangement is always doable.

We prove this by induction. In the base case $n = 1$, there is only one rectangle of height $w_0 = 1$. Each column has height 1, contains at most two different rectangles, and the $0^{th}$ column contains at least one piece of the $0^{th}$ rectangle.

For the induction step, assume that the theorem holds for $n$. For $n + 1$ width-one rectangles of heights $w_0, w_1, ..., w_n$ such that $\sum_{i=0}^{n} w_i = n + 1$, according to Lemma 2, there must be at least one rectangle with height $w_l \leq 1$ and one rectangle with height $w_g \geq 1$. We assume that the $l^{th}$ rectangle has a height $w_l \leq 1$ and the $g^{th}$ rectangle has a height $w_g \geq 1$. Then there are two possible cases.

Case 1: If $w_l = 1$, we just keep this rectangle in the $l^{th}$ column without splitting. This column satisfies all the conditions. The remaining $n$ rectangles have sum $n+1-1 = n$. By the induction hypothesis, we can assign them into n columns while satisfying the conditions.

Case 2: If $w_l < 1$, we can cut off a piece with height $1 - w_l$ from rectangle g and move it to fill the $l^{th}$ column up to a height of 1. Then the $l^{th}$ column satisfies all the conditions and there are $n$ rectangles left with a sum of $n + 1 - 1 = n$. Again, by the induction hypothesis, we can assign them into $n$ columns while satisfying the conditions. □

### 2.5.2 Implementation

We follow the pseudo code from [16] to implement Vose's Alias Method. The induction step in the proof of Theorem 1 gives the idea for building the probability and alias table. After scaling the weights $\sum_{i=0}^{n-1} w_i = n$, in each round, we select a combination of weights $w_l < 1$ and $w_g \geq 1$ to determine the $l^{th}$

column of the probability table *Prob* and alias table *Alias*. Note that any $w_g \geq 1$ can fill the gap in the $l^{th}$ column for a small weight $w_l < 1$. We use two queues, *Small* to keep track of weights smaller than 1, and *Large* for the weights greater than or equal to 1. In Algorithm 9 we show the construction procedure. After step 5, at least one of the queues is empty. The remaining weights should all be 1 in *Large*. Theoretically, there should not be $w_i < 1$ in *Small* at this time point. But weights can be slightly larger or smaller due to numerical error, so in practice we add steps 6 and 7 in Algorithm 9 to deal with these weights.

To sample an element according to the weight array $W$, we choose an integer $i \in [0, n)$ and a real number $d \in [0, 1)$ uniformly at random. The algorithm outputs i if $d < Prob_i$, and $Alias_i$ otherwise. We show the implementation of sampling in Algorithm 10.

### 2.5.3 Using the Alias Method for a Time-changing Distribution

Unfortunately, as far as we know, there is no efficient way to update the weights in K and U after a weight change. For example, if we update $w_0$ from 6 to 2 for $W = [6, 4, 1, 1]$, the new $W' = [2, 4, 1, 1]$, Figures 2.4 and 2.5 show that in the new tables, all entries except for the values in column 1 are different.

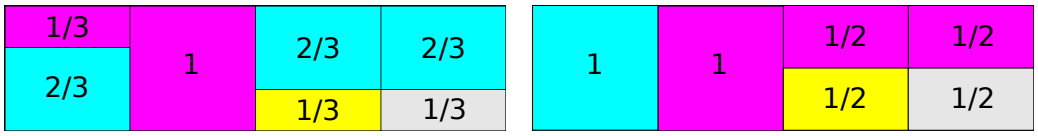



Figure 2.4: The rearrangement of the pieces for the new weight array $W' = [2, 4, 1, 1]$ on the right compared with the old $W = [6, 4, 1, 1]$ on the left.

| U | 2/3 | 1 | 1/3 | 1/3 |
|---|---|---|---|---|
| K | 1 | N/A | 0 | 0 |

| U | 1 | 1 | 1/2 | 1/2 |
|---|---|---|---|---|
| K | N/A | N/A | 1 | 1 |

Figure 2.5: The alias and probability tables for the new weight array $W = [2, 4, 1, 1]$ on the right compared with the old ones on the left.

Updating a weight can have $\Omega(n)$ runtime complexity as we need to completely rebuild the tables.

**Algorithm 9:** Alias Method Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$
**Output:** Probability table $Prob$ and alias table $Alias$

1. $Small = Queue()$, $Large = Queue()$
2. $S = \sum_{i=0}^{n-1} w_i$
3. For each integer $i \in [0, n)$
   (a) $w_i = \frac{w_i \cdot n}{S}$
   (b) If $w_i < 1$
      i. $Small.put(i)$
   (c) Else
      i. $Large.put(i)$
4. While not ($Small.empty()$ and $Large.empty()$)
   (a) $l = Small.get()$, $g = Large.get()$
   (b) $Prob[l] = w_l$, $Alias[l] = g$
   (c) $w_g = w_l + w_g - 1$
   (d) If $w_g < 1$
      i. $Small.put(g)$
   (e) Else
      i. $Large.put(g)$
5. While not $Large.empty()$
   (a) $g = Large.get()$
   (b) $Prob[g] = 1$
6. While not $Small.empty()$
   (a) $l = Small.get()$
   (b) $Prob[l] = 1$

**Algorithm 10:** Alias Method Sampling

**Input:** Probability table $Prob$ and alias table $Alias$
**Output:** Sampling result $i$ or $Alias_i$

1. $i = uniform\_int(0, n-1)$, $d = uniform\_real(0, 1)$
2. If $d < Prob_i$
   (a) return $i$
3. Else
   (a) return $Alias_i$

## 2.6 Method 4: C++11 discrete_distribution

Starting from C++11 [8], the programming language C++ provides a random number generator for discrete distributions. Using the $std :: discrete\_distribution$ template together with a random number engine such as the Mersenne Twister random number engine provided by C++ libraries, one can generate random numbers according to a given weight distribution. C++ standard requires a combination of generator and distribution to produce random numbers: while a generator produces a sequence of uniformly distributed random numbers, a distribution object transforms the sequence into numbers following a specific random variable distribution. Figure 2.6 shows an example of this process.

For time-changing discrete distributions, users must call the constructor each time the distribution is updated, as no updating function is provided in the class.

The C++11 standard does not specify the sampling runtime complexity, so we checked two major open source implementations, the Boost C++ Library [1] and the GNU Compiler Collection [15] (GCC), to determine their runtime complexity. Interestingly, they used different algorithms. The Boost C++ Library used Vose's Alias method [18], while GCC used binary search. The asymptotic runtime complexity of both methods is different, as shown in Table 2.1.

```cpp
#include <iostream>
#include <random>

int main()
{
  //create the random number engine and use random_device to seed it
  std::random_device rd;
  std::mt19937 generator(rd());

  //create the distribution object according to the weight array W={6,4,1,1}
  std::discrete_distribution<> distribution({6,4,1,1});

  //generate a random number according to the weight array
  std::cout<<distribution(generator)<<std::endl;

  return 0;
}
```

Figure 2.6: An example of generating random numbers according to weight array $W = [6, 4, 1, 1]$ using C++ libraries.

| | Sample | Update | Construction |
|---|---|---|---|
| Boost C++ Library | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| GNU Compiler Collection | $\Theta(\log(n))$ | $O(n)$ | $\Theta(n)$ |

Table 2.1: The sample, update and construction runtime complexity for the Boost C++ Library and the GNU Compiler Collection implementations of the class std::discrete_distribution in the C++11 Standard.

## 2.7 Method 5: Two Level Search

Two level search is a simple but effective sampling method that is well known in the Computer Go community. However, despite a thorough search, we could not identify a formal reference. The idea is to build a two level structure to achieve a better sampling runtime while keeping updating efficient. The first level maintains the original weight array $W$, while the second level maintains a partial sum array $WD = [wd_0, ..., wd_{\lceil \frac{n}{L} \rceil - 1}]$. Each entry in $WD$ maintains the partial sum of a group of $L < n$ weights $wd_i = \sum_{j=start(i)}^{end(i)} w_j$, with $start(i) = L \cdot i$ and $end(i) = min(L \cdot (i+1) - 1, n - 1)$. Figure 2.7 illustrates $W$ as a two-dimensional grid on the left. The array $WD$ on the right contains the sum of each row. We also maintain the global sum $S = \sum_{i=0}^{n-1} w_i$. We show the initialization of the data structures in Algorithm 11. We set $L = \lceil \sqrt{n} \rceil$, and will discuss this choice in Section 2.7.1.

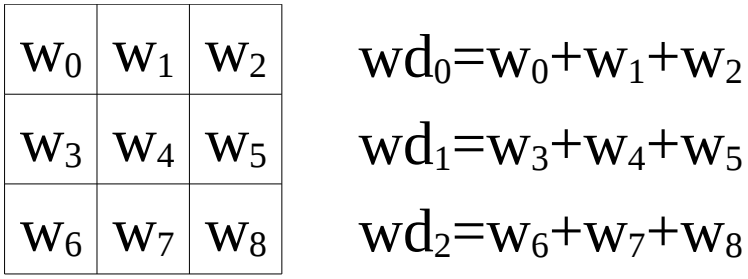



Figure 2.7: An example of two level search structure with $n = 9$ and $L = 3$.

**Algorithm 11:** Two Level Search Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$

**Output:** Array $WD$ storing the partial sums and the overall sum $S$

1. $S = 0$, $L = \lceil \sqrt{n} \rceil$, $WDn = \left\lceil \frac{n}{L} \right\rceil$
2. For each integer $i \in [0, WDn)$
   (a) $start(i) = L \cdot i$, $end(i) = min(L \cdot (i + 1) - 1, n - 1)$
   (b) $wd_i = \sum_{j=start(i)}^{end(i)} w_j$
   (c) $S = S + wd_i$

To sample from the distribution, we choose a real number $d \in [0, S)$ uniformly at random. We use linear search in $WD$ to find the corresponding group, i.e., we find the smallest $r$ satisfying $wd_0 + wd_1 + ... + wd_r > d$. Then we run another linear search from $start(r)$ to $end(r)$ in $W$ to find the smallest index $i$ satisfying $w_0 + w_1 + ... + w_i = \sum_{j=0}^{r-1} wd_j + w_{start(r)} + ... + w_i > d$. Index $i$ is the sampling result. We show the pseudo code in Algorithm 12.

**Algorithm 12:** Two Level Search Sampling

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, partial sum array $WD = [wd_0, wd_1, ..., wd_{WDn-1}]$ and sum $S = \sum_{i=0}^{n-1} w_i$

**Output:** Sampling result $i$

1. $d = uniform\_real(0, S)$
2. $i = linear\_search(WD, d, 0)$
3. $d = d - \sum_{j=0}^{i-1} wd_j$
4. $L = \lceil \sqrt{n} \rceil$, $i = i \cdot L$
5. $i = linear\_search(W, d, i)$
6. return $i$

To update a weight, we need to update three variables: $w_i$ in $W$, the corresponding $wd_r$ with $r = \left\lfloor \frac{i}{L} \right\rfloor$ in $WD$, and $S$. We show the pseudo code in Algorithm 13.

**Algorithm 13:** Two Level Search Update

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, partial sum array $WD = [wd_0, wd_1, ..., wd_{WDn-1}]$ , sum $S = \sum_{i=0}^{n-1} w_i$ and the new weight $w_i'$

1. $L = \lceil \sqrt{n} \rceil$, $r = \left\lfloor \frac{i}{L} \right\rfloor$
2. $wd_r = wd_r + w_i' - w_i$,$S = S + w_i' - w_i$
3. $w_i = w_i'$

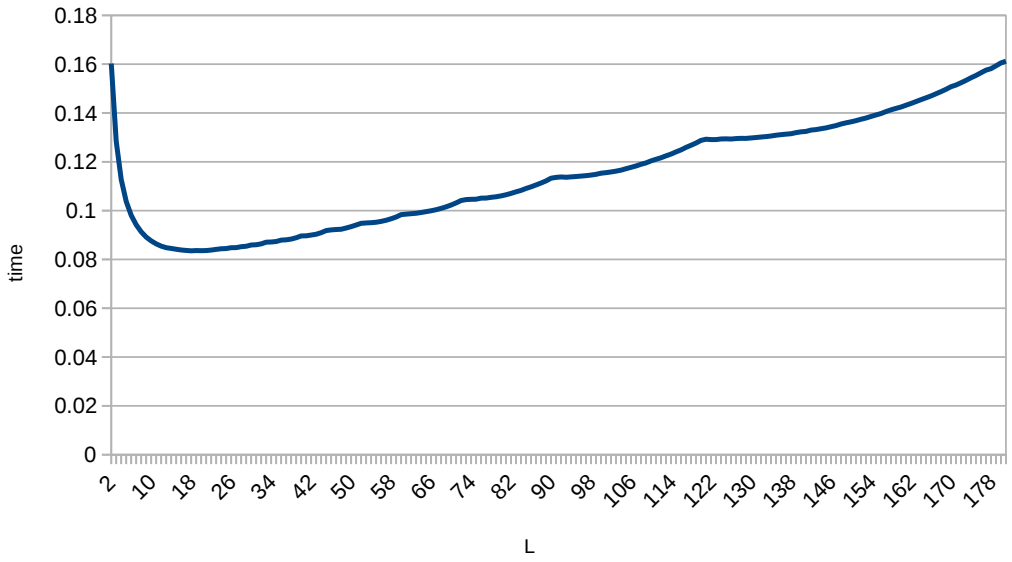



Figure 2.8: Two level search for $19 \times 19$ Go with $n = 361$ and varying $L$. The y-axis in the plot shows the average time spent to sample $10^5$ times with one update after each sample, plotted against $L$ on the x-axis. The time spent for each $L$ was averaged over 100 rounds. The minimum time spent is achieved with $L$ chosen around 19, which is the square root of 361.

### 2.7.1 Determining the Partial Sum Group Size $L$

The partial sum group size $L$ can be any number between 1 and $n$. Empirically, $L = \lceil \sqrt{n} \rceil$ leads to the best performance, as shown in Figure 2.8. This result matches our theoretical analysis: For a two level search, the time complexity of sampling is $\mathrm{O}(\lceil \frac{n}{L} \rceil + L) = \mathrm{O}(\frac{n}{L} + L)$. This expression achieves its minimum $\frac{n}{L} + L \geq 2\sqrt{\frac{n}{L} * L} = 2\sqrt{n}$ when $\frac{n}{L} = L$, i.e., $L = \sqrt{n}$.

The time complexity of method construction is $\Theta$(n) since we can calculate $WD$ and $S$ in a single pass as in Algorithm 11. With $L = \lceil \sqrt{n} \rceil$, the time complexity of sampling is $\mathrm{O}(\lceil \frac{n}{\lceil \sqrt{n} \rceil} \rceil + \lceil \sqrt{n} \rceil) = \mathrm{O}(\sqrt{n})$. The updating procedure shown in Algorithm 13 has a runtime complexity of $\Theta(1)$.

## 2.8 Method 6: Binary Tree Search

Instead of building only one extra level on top of the original weight array $W$, we can build a binary tree to help searching. The leaves of the tree are the weights $w_i$ in $W$. Each node in the tree stores the sum of weights

of its children. To optimize the space needed, we build the tree using a 1-based compact array $T = [t_1, ..., t_{2n-1}]$. Each element $t_{n+i}$ stores the original weight $w_i$ for $0 \leq i < n$, and each element $t_i$ stores the sum of its children $t_i = t_{2i} + t_{2i+1}$, for $1 \leq i < n$. The root $t_1$ contains the sum of all weights $\sum_{i=0}^{n-1} w_i$. We show the implementation of the data structure initialization in Algorithm 14.

**Algorithm 14:** Binary Tree Search Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$
**Output:** Array $T = [t_1, ..., t_{2n-1}]$ representing the binary tree

1. For each integer $i \in [0, n)$
   (a) $t_{n+i} = w_i$
2. $i = n - 1$
3. While $i \geq 1$
   (a) $t_i = t_{2i} + t_{2i+1}$
   (b) $i = i - 1$

To sample from the distribution, we choose a real number $d \in [0, t_1)$ uniformly at random. Starting from the root $t_1$, we search down the tree by repeating the following process: for each node $t_i$ we are on, if $d < t_{2i}$, we go to the left child $t_{2i}$, otherwise, we subtract $t_{2i}$ from $d$ and go to the right child $t_{2i+1}$. We repeat the process until we reach a leaf $t_i$ with $n \leq i < 2n$, and we output $i - n$ as the sampling result. We show the pseudo code in Algorithm 15.

To update a weight $w_i$, we first update the value in the leaf, then iteratively update the values of its ancestors. We show the pseudo code in Algorithm 16.

The time complexity of method construction is $\Theta$(n) since it takes $\Theta(1)$ time to fill each of the $2n - 1$ elements in $T$. The time complexity of both sampling and updating is $\Theta$(log(n)): we need to visit each level in the tree. There are $\lfloor \log(n) \rfloor + 1$ levels, and there is $\Theta(1)$ work on each level.

**Algorithm 15:** Binary Tree Search Sampling

**Input:** Array $T = [t_1, ..., t_{2n-1}]$ representing the binary tree
**Output:** Sampling result $i$

1. $d = \mathit{uniform\_real}(0, t_1)$
2. $i = 1$
3. While $i < n$
   (a) If $d < t_{2i}$
      i. $i = 2i$
   (b) Else
      i. $d = d - t_{2i}$
      ii. $i = 2i + 1$
4. return $i - n$

**Algorithm 16:** Binary Tree Search Update

**Input:** Array $T = [t_1, ..., t_{2n-1}]$ representing the binary tree and the new weight $w_i'$

1. $i = n + i$
2. $t_i = w_i'$
3. While $i > 1$
   (a) $\mathit{leftChild} = i$, $\mathit{rightChild} = i$
   (b) If $i\%2 == 0$
      i. $\mathit{rightChild} = \mathit{rightChild} + 1$
   (c) Else
      i. $\mathit{leftChild} = \mathit{leftChild} - 1$
   (d) $i = \left\lfloor \frac{i}{2} \right\rfloor$
   (e) $t_i = t_{\mathit{leftChild}} + t_{\mathit{rightChild}}$

## 2.9 Method 7: Rejection Method

Rejection sampling is a Monte Carlo method which generates samples from a probability distribution. We use an example to illustrate how it works. Figure 2.9 shows a probability density function $p(x) = \frac{1}{1.2113}(0.3e^{-(x-0.3)^2} + 0.7e^{-\frac{10}{3}(x-2)^2})$ on the left. It is not trivial to draw a sample according to this

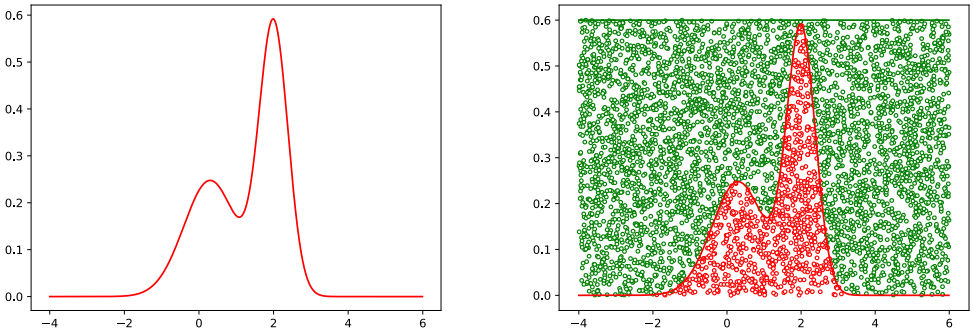



Figure 2.9: The plot on the left shows the probability density function $p(x) = \frac{1}{1.2113}(0.3e^{-(x-0.3)^2} + 0.7e^{-\frac{10}{3}(x-2)^2})$ in the interval $[-4, 6]$ while the plot on the right shows how we draw samples according to this function. The circles in the right plot are the samples we draw, and those colored red are the ones we accept.

probability density function directly. Instead, we can draw sample points $(x, y)$ in the rectangular area $x \in$ [-4, 6) and $y \in [0, 0.6)$ uniformly at random, and only accept the points under the curve (shown in Figure 2.9 on the right). It is obvious that the probability with which we accept a sample $x$ is proportional to $p(x)$. Given that $x$ and $y$ are chosen uniformly at random, we are sampling according to $p(x)$ inside the rectangle. To draw one sample, we keep drawing sample points $(x, y)$ until $y < p(x)$ occurs, and output $x$ as the result.

It is not hard to apply rejection sampling to discrete distributions. To construct the method, we need to store the weight array $W$ and the maximum weight $B = max(w_i)$. We show the initialization of the data structures in Algorithm 17.

To sample from a weight array W, we repeat the following process until a value is output: choose an integer i $\in$ [0, n) and a real number d $\in$ [0, 1) uniformly at random, then output i if $d < \frac{w_i}{B}$. As shown in Figure 2.10, this

**Algorithm 17:** Rejection Method Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$
**Output:** Bound $B$

1. $B = 0$
2. For each integer $i \in [0, n)$
   (a) $B = max(B, w_i)$

process is equivalent to drawing sample points uniformly from the rectangular area $x \in$ [-0.5, 3.5) and $y \in$ [0, 6), and accepting those in the gray area. We show the pseudo code in Algorithm 18.

**Algorithm 18:** Rejection Method Sampling

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$ and bound $B$
**Output:** Sampling result $i$

1. While true
   (a) $i = \textit{uniform\_int}(0, n)$, $d = \textit{uniform\_real}(0, 1)$
   (b) If $d < \frac{w_i}{B}$
      (1). return $i$

To update a weight, we need to modify $W$ and update $B$. In our implementation, we update the bound $B = \max(w_i)$ only when $\max w_i$ increases. We show the pseudo code in Algorithm 19.

**Algorithm 19:** Rejection Method Update

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, bound $B$ and new weight $w'_i$

1. $B = max(B, w'_i)$
2. $w_i = w'_i$

The time complexity of construction is $\Theta$(n). A single iteration leads to an output with probability $\frac{S}{nB}$, so the expected sampling time is $\Theta(\frac{nB}{S})$. Sampling remains efficient as long as $S$ is not much smaller than $nB$. An example of the

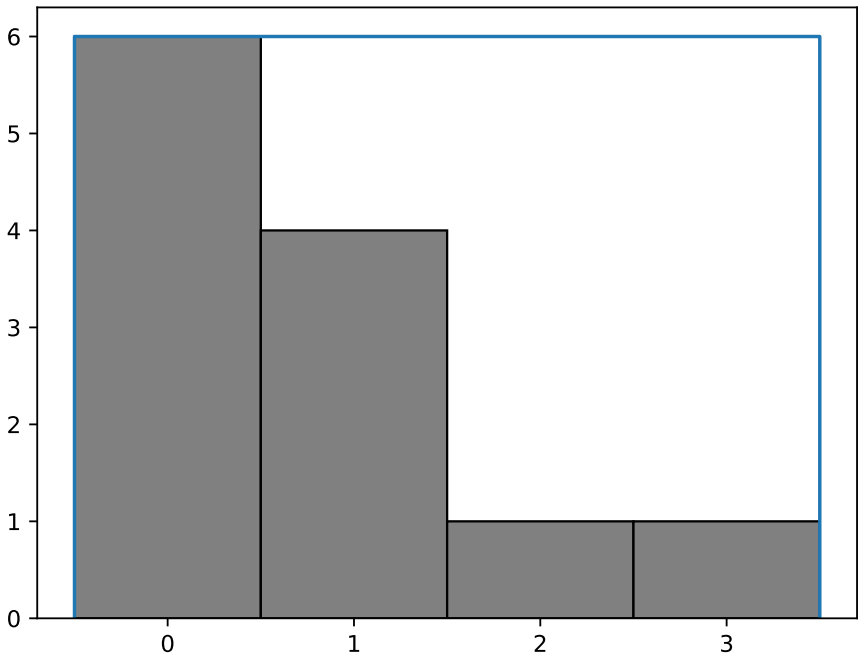



Figure 2.10: Drawing samples using the rejection method from the discrete distribution $W = [6, 4, 1, 1]$.

worst case is when we have $W = [w_0, 0, 0, ..., 0]$ with $w_0 \neq 0$: we can generate a sampling result only if we select $i = 0$, and the expected sampling time becomes $\Theta(\frac{nB}{w_0})$. If $B = w_0$, the expected sampling time is $\Theta(n)$. If $B \gg w_0$, the expected sampling time could be much worse. On the other hand, if we have a weight array with equal weights, the expected sampling time becomes $\Theta(\frac{nB}{nB}) = \Theta(1)$. The updating runtime complexity is $\Theta(1)$.

## 2.10 Method 8: The Log* Method

Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni built another sampling method [12] based on the rejection method. Their method, referred to as the log* method in this thesis, achieves $O(\log^* n)$ sampling and $O(2^{\log^* n})$ updating time complexity. With $\log^* n \leq 5$ and $2^{\log^* n} \leq 32$ for $n \leq 2^{65536}$, their method works in constant time for practical purposes.

The core idea of this method is to partition the weights into $[2^{j-1}, 2^j)$ groups and then treat the sum of each group as a new weight. The group weights form a new level, and groups containing two or more elements are used to recursively build the next level. This recursion builds a forest of trees, with the height of a tree being at most $\log^* n + 1$. Deciding the starting level and searching down from this level by applying the rejection method in each level requires visiting at most $O(\log^* n)$ levels, and using the rejection method in each group has a expected runtime complexity of $O(1)$. As a result, the time complexity of sampling is $O(\log^* n)$. The updating procedure is much more sophisticated and has $O(2^{\log^* n})$ time complexity. The authors also proved that the construction time is O(n).

We use an example to illustrate the construction process of the log* method. Consider a weight array $W = [\frac{1}{4}, \frac{1}{3}, \frac{1}{2}, 8, 15, 15, 16, 16, 268]$. We can partition the weights into five groups to form level 0 in the trees. We use $R_i^j$ to denote the group in level $i$ containing weights belonging to $[2^{j-1}, 2^j)$ and $S_i^j$ to denote the sum of weights in each group. At level 0, we have $R_1^{-1} = \{\frac{1}{3}, \frac{1}{4}\}$, $R_1^0 = \{\frac{1}{2}\}$, $R_1^4 = \{8, 15, 15\}$, $R_1^5 = \{16, 16\}$ and $R_1^9 = \{268\}$, with sums $S_1^{-1} = \frac{7}{12}$, $S_1^0 = \frac{1}{2}$, $S_1^4 = 38$, $S_1^5 = 32$ and $S_1^9 = 268$. We use a level table $T$ to record the groups

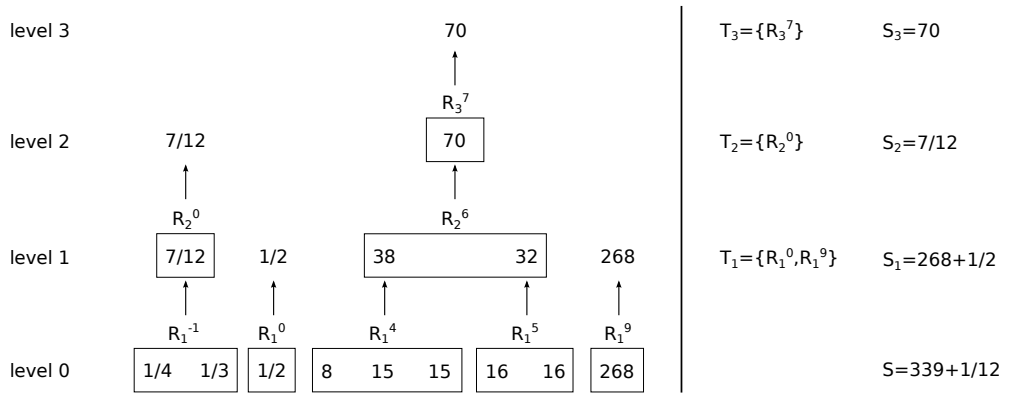



Figure 2.11: Example of the forest built by the log* method. There are three levels and four roots. The right side shows the level table and the level sums.

which contain only one weight (called roots), and treat the sum of groups which contain two or more weights, $S_1^{-1}$, $S_1^4$ and $S_1^5$, as new weights. We can again partition the new weights $\{\frac{7}{12}, 38, 32\}$ into two groups $R_2^0 = \{\frac{7}{12}\}$ and $R_2^6 = \{32, 38\}$ with sums $S_2^0 = \frac{7}{12}$ and $S_2^6 = 70$. By recursively partitioning, we construct a forest of trees shown in Figure 2.11. There are three levels in addition to the original weight level, and there are four roots, $R_1^0$, $R_1^9$, $R_2^0$ and $R_3^7$. To store these roots in the level table T, such that $T_l$ contains all the roots in level $l$, we have $T_1 = \{R_1^0, R_1^9\}$, $T_2 = \{R_2^0\}$, $T_3 = \{R_3^7\}$. Besides the level table, we also store the sums of weights in each level $S_1 = 268\frac{1}{2}$, $S_2 = \frac{7}{12}$, $S_3 = 70$ as well as the overall sum $S = 339\frac{1}{12}$.

To implement this method, each group $R_l^j$ needs to record its children in an array. To enable efficient removal, it also needs to record and maintain the array index of each child. In addition, we store the sum of weights of its children in $R_l^j.weight$. Each $T_l$ in the level table $T$ is implemented as a hashmap that maps $j$ to $R_l^j$. We maintain an integer $level\_root_l$ that keeps track of all roots in level $l$. Its $j^{th}$ bit is set iff the root $R_l^j$ exists. We show the pseudo code in Algorithm 20. We use a helper function *construct_level* shown in Algorithm 21 to demonstrate the level by level construction process recursively.

To sample from the discrete distribution, we need to determine the root to start from. We choose a real number d $\in$ [0, S) uniformly at random, and run

**Algorithm 20:** Log* Method Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$

**Output:** Level table $T$, level sum $level\_sum$, root tacking table $level\_root$ and the overall sum S

1. $Q = Queue()$, $S = set()$
2. For each integer $i \in [0, n)$
   (a) $j = \lfloor log(w_i) \rfloor + 1$
   (b) $R_1^j.children.append(i)$, $R_1^j.weight = R_1^j.weight + w_i$
   (c) If $R_1^j$ not in $S$
      i. $S.add(R_1^j)$
      ii. $Q.put(R_1^j)$
   (d) $S = S + w_i$
3. $construct\_level(1, Q)$

a linear search to find the smallest $i$ satisfying $S_{level} = S_1 + S_2 + ... + S_i > d$. Then we use another linear search in level $i$ to determine the root $R_i^j$ we start from. We start with the largest $j$ in level $i$, and keep adding $S_i^j$ to a temporary sum $TS$, until $S_{level} + TS > d$. The last $S_i^j$ we added corresponds to the root $R_i^j$ we start from. Starting from root $R_i^j$, we trace down the tree to find the index we want to output. For a group $R_i^j$ we are on, we use rejection sampling with bound $B = 2^j$ to choose one weight and go to the corresponding group. We repeat this procedure until we reach level 0, where we output the index chosen as the result. We show the pseudo code in Algorithm 22.

We need two helper functions to accomplish the update procedure:

1. A $remove(R_i^j, item)$ function that removes $item$ from group $R_i^j$. Based on the level $i$, the $item$ could be either a weight from $W$ or a group $R_{i-1}^{j'}$. We need to update the sum $S_i^j$, and we may need to change the parent of $R_i^j$ in the forest, which triggers recursive calls $remove(R_{i+1}^{j'}, R_i^j)$ and $add(R_{i+1}^{j''}, R_i^j)$.

2. An $add(R_i^j, item)$ function that adds $item$ to group $R_i^j$. Similar to $remove(R_i^j, item)$, the $item$ could be a weight or a group. We also need

**Algorithm 21:** Log* Method *construct_level*

**Input:** Level number $l$ and queue of groups $Q$

**Output:** Level table $T$, level sum $level\_sum$ and root tacking table $level\_root$

1. $more\_than\_one = false$
2. $Q' = Queue()$
3. While not $Q.empty()$
   (a) $R_l^i = Q.get()$
   (b) If $len(R_l^i.children) == 1$
      i. $level\_sum_l = level\_sum_l + R_l^i.weight$
      ii. $level\_root_l = level\_root_l + 2^i$
      iii. $T_l[i] = R_l^i$
   (c) Else
      i. $j = \lfloor log(R_l^i.weight) \rfloor + 1$
      ii. $R_{l+1}^j.children.append(R_l^i)$
      iii. $R_{l+1}^j.weight = R_{l+1}^j.weight + R_l^i.weight$
      iv. $Q'.put(R_{l+1}^j)$
      v. $more\_than\_one = true$
4. If $more\_than\_one$
   (a) $construct\_level(l + 1, Q')$

**Algorithm 22:** Log* Method Sampling

**Input:** Level table $T$, level sum $level\_sum$, root tacking table $level\_root$ and the overall sum S

**Output:** Sampling result $i$

1. $d = uniform\_real(0, S)$
2. $l = linear\_search(level\_sum, d, 1)$
3. $root = None$
4. While $root == None$
   (a) $j = \lfloor log(level\_root_l) \rfloor + 1$, $R_l^j = T_l[j]$
   (b) $level\_root_l = level\_root_l - 2^j$
   (c) If $d < R_l^j.weight$
      i. $root = R_l^j$
   (d) Else
      i. $d = d - R_l^j.weight$
5. While true
   (a) $B = 2^j$, $i = uniform\_int(0, len(root.children) - 1)$, $d = uniform\_real(0, 1)$
   (b) $R_l^j = root$
   (c) If $l == 1$
      i. If $d < \frac{w[root.children[i]]}{2^j}$
         A. return $root.children[i]$
   (d) Else
      i. If $d < \frac{root.children[i].weight}{2^j}$
         A. $root = root.children[i]$

to call $remove(R_{i+1}^{j'}, R_i^j)$ and $add(R_{i+1}^{j''}, R_i^j)$ recursively if the parent of $R_i^j$ changes because of the update of $S_i^j$.

It is worth noting that, since we change the number of children $R_i^j$ has in both functions, we also need to consider the case where $R_i^j$ switches between root, non-root and non-valid. To update a weight $w_i$ to $w_i'$, we just need to call $remove(R_i^j, w_i)$ and $add(R_i^{j'}, w_i')$, where $R_i^j$ is the original group $w_i$ belongs to, and $R_i^{j'}$ is the new group $w_i'$ belongs to. We show the pseudo code for the updating process, *add_weight* and *delete_weight* in Algorithms 23 - 25.

**Algorithm 23:** Log* Method Update

**Input:** Level table $T$, level sum *level_sum*, root tracking table *level_root*, the overall sum S and the new weight $w_i'$

1. $j = \lfloor \log(w_i) \rfloor + 1$
2. $S = S - w_i$
3. $delete\_weight(R_1^j, w_i)$
4. $S = S + w_i'$
5. $j' = \lfloor \log(w_i) \rfloor + 1$
6. $add\_weight(R_1^{j'}, w_i')$

The authors [12] introduced a way to modify this method to achieve O(log* n) updating time complexity. Inspired by T. Hagerup's paper [6], they also propose a way to make this method achieve O(1) time complexity for both sampling and updating. However, these modifications involve time consuming operations such as building a lookup table to handle all possible changes in the future. Hence, the authors still recommend the basic log* method for general use [12]. In our experiments, we found that the basic log* method already has too much overhead in practice. Considering our goal of finding suitable methods for general use, we chose not to introduce more overhead by building even more complex methods.

**Algorithm 24:** Log* Method *add_weight*

**Input:** Level table $T$, level sum *level_sum*, root tracking table *level_root*, target group $R_l^i$,and the new child *child*

1. $R_l^i.children.append(child)$
2. $R_l^i.weight = R_l^i.weight + child.weight$
3. If $len(R_l^i.children) == 1$
   (a) $level\_sum_l = level\_sum_l + R_l^i.weight$
   (b) $level\_root_l = level\_root_l + 2^i$
   (c) $T_l[i] = R_l^i$
4. Elif $len(R_l^i.children) == 2$
   (a) $level\_sum_l = level\_sum_l - (R_l^i.weight - child.weight)$
   (b) $level\_root_l = level\_root_l - 2^i$
   (c) $j = \lfloor \log(R_l^i.weight) \rfloor + 1$
   (d) $add\_weight(R_{l+1}^j, R_l^i)$
5. Else
   (a) $j' = \lfloor \log(R_l^i.weight - child.weight) \rfloor + 1$
   (b) $delete\_weight(R_{l+1}^{j'}, R_l^i)$
   (c) $j = \lfloor \log(R_l^i.weight) \rfloor + 1$
   (d) $add\_weight(R_{l+1}^j, R_l^i)$

**Algorithm 25:** Log* Method *delete_weight*

**Input:** Level table $T$, level sum $level\_sum$, root tracking table $level\_root$, group $R_l^i$, and the node to be removed $child$

1. $R_l^i.weight = R_l^i.weight - child.weight$
2. $index$=index of $child$ in $R_l^i.children$
3. $swap\_and\_remove(R_l^i.children, index)$
4. If $len(R_l^i.children) == 0$
   (a) $level\_sum_l = level\_sum_l - R_l^i.weight$
   (b) $level\_root_l = level\_root_l - 2^i$
   (c) $T_l.pop(i)$
5. Elif $len(R_l^i.children) == 1$
   (a) $level\_sum_l = level\_sum_l + R_l^i.weight$
   (b) $level\_root_l = level\_root_l + 2^i$
   (c) $T_l[i] = R_l^i$
   (d) $j' = \lfloor \log(R_l^i.weight + child.weight) \rfloor + 1$
   (e) $delete\_weight(R_{l+1}^{j'}, R_l^i)$
6. Else
   (a) $j' = \lfloor \log(R_l^i.weight + child.weight) \rfloor + 1$
   (b) $delete\_weight(R_{l+1}^{j'}, R_l^i)$
   (c) $j = \lfloor \log(R_l^i.weight) \rfloor + 1$
   (d) $add\_weight(R_{l+1}^{j}, R_l^i)$

| | Sampling time | Updating time | Constructing time |
|---|---|---|---|
| Alias Method | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| C++ discrete | unspecified[1] | unspecified[1] | $\Theta(n)$ |
| Linear search | $O(n)$ | $\Theta(1)$ | $\Theta(n)$ |
| Binary search | $O(\log(n))$ | $O(n)$ | $\Theta(n)$ |
| Two level search | $O(\sqrt{n})$ | $\Theta(1)$ | $\Theta(n)$ |
| Binary tree search | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$ |
| Rejection Method | $O(\frac{nB}{S})$ | $\Theta(1)$ | $\Theta(n)$ |
| Log* Method | $O(\log^* n)$ | $O(2^{\log^* n})$ | $\Theta(n)$ |

Table 2.2: The table summarizes the time complexity for all the sampling methods in this chapter. In the table, $n$ is the size of weight array $W$, $S$ is the sum of all weights, $B$ is the maximum weight in $W$, and $M$ represents the maximum possible weight.

## 2.11 Summary

To sample from discrete distributions, the methods in the literature can be partitioned into three groups, namely search-based methods, rejection-based methods and the Alias Method. Search-based methods include linear search, binary search, two level search, and binary tree search. They find the sample by searching, and use simple data structures to optimize the search procedure. Rejection-based methods include the rejection method and log* method. They could be more complex but their runtime depends less on $n$. In order to compare all methods, Table 2.2 shows all the known theoretical runtime complexity results.

[1]The sampling and updating time complexity of C++ discrete depends on the implementations. Please refer to Section 2.6 for details.

# Chapter 3

# New Sampling Methods for Time-Changing Discrete Distributions

By expanding on ideas in existing methods and algorithms, and based on practical considerations, we derive and propose two new methods: In Section 3.1, we give an introduction to multi-level search, with a focus on three level search. In Section 3.2, we give an introduction to flat method.

## 3.1 Multi-Level Search

Expanding the idea of the two level search of Section 2.7, we build multi-level structures, starting with three levels. The third level *WD2* maintains the partial sums of the second level *WD*. The data structures are identical to the ones we introduced in Section 2.7, plus an additional array $WD2 = [wd2_0, ..., wd2_{\lceil \frac{n}{L^2} \rceil - 1}]$. Each entry in *WD2* maintains the partial sum of a group of $L$ weights $wd2_i = \sum_{j=start(i)}^{end2(i)} wd_j$, with $start(i) = L \cdot i$ and $end2(i) = min(L \cdot (i+1) - 1, \left\lceil \frac{n}{L} \right\rceil - 1)$. We can view this three-level structure as a forest of trees, where each node is an element in $W$, *WD* or *WD2*. The roots are elements in *WD2*. Figure 3.1 demonstrates this idea with an example. We show the initialization of the data structures in Algorithm 26. We set $L = \lceil \sqrt[3]{n} \rceil$, and will discuss this choice at the end of this subsection.

To sample from a distribution, we choose a real number $d \in [0, S)$ uniformly at random. We run a linear search in *WD2* to find the smallest $r2$ satisfying

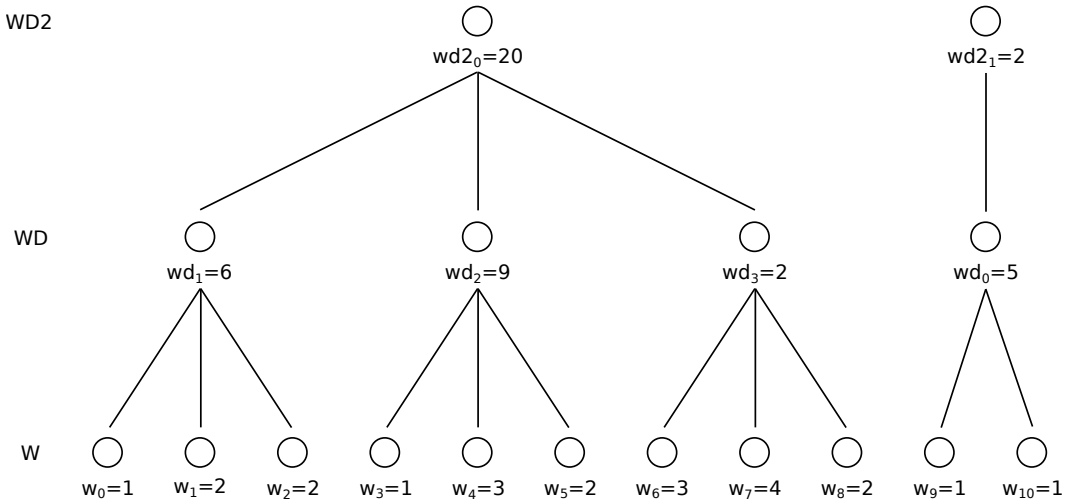



Figure 3.1: In this example, $W = [1, 2, 2, 1, 3, 2, 3, 4, 2, 1, 1]$ and $L = 3$. We construct $WD = [5, 6, 9, 2]$ and $WD2 = [20, 2]$ according to $W$.

---

**Algorithm 26:** Three Level Search Construction

---

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$

**Output:** Array $WD$, $WD2$ storing the partial sums and the overall sum $S$

1. $S = 0$, $L = \lceil \sqrt[3]{n} \rceil$
2. For each integer $i \in [0, \left\lceil \frac{n}{L} \right\rceil)$
   (a) $start(i) = L \cdot i$, $end(i) = min(L \cdot (i + 1) - 1, n - 1)$
   (b) $wd_i = \sum_{j=start(i)}^{end(i)} w_j$
   (c) $S = S + wd_i$
3. For each integer $i \in [0, \left\lceil \frac{n}{L^2} \right\rceil)$
   (a) $start(i) = L \cdot i$, $end2(i) = min(L \cdot (i + 1) - 1, \left\lceil \frac{n}{L} \right\rceil - 1)$
   (b) $wd2_i = \sum_{j=start(i)}^{end2(i)} wd_j$

---

$wd2_0 + wd2_1 + ... + wd2_{r2} > d$. Each tree with root being an element of *WD2* can then be viewed as a two level search problem. So we can subtract $\sum_{j=0}^{r2-1} wd2_j$ from $d$, and follow the sampling process we introduced in Section 2.7 to generate the result from the two-level subproblem. The only difference is that when we run linear search in $WD$, we start from $start(r2)$ instead of 0 as we have already subtracted some values from $d$. We show the pseudo code in Algorithm 27.

**Algorithm 27:** Three Level Search Sampling

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, partial sum array $WD = [wd_0, wd_1, ..., wd_{WDn-1}]$ and sum $S = \sum_{i=0}^{n-1} w_i$

**Output:** Sampling result $i$

1. $d = uniform\_real(0, S)$
2. $i = linear\_search(WD2, d, 0)$
3. $d = d - \sum_{j=0}^{i-1} wd2_j$
4. $L = \lceil \sqrt[3]{n} \rceil$, $i = i * L$
5. $i' = linear\_search(WD, d, i)$
6. $d = d - \sum_{j=i}^{i'-1} wd_j$
7. $i' = i' * L$
8. $i' = linear\_search(W, d, i')$
9. return $i'$

The updating process is the same as the one we introduced in Section 2.7, except that we also need to update $wd2_{r2}$ with $r2 = \left\lfloor \frac{i}{L^2} \right\rfloor$ in *WD2*. We show the pseudo code in Algorithm 28.

The time complexity of construction remains $\Theta(n)$. The updating procedure has a runtime complexity of $\Theta(1)$. The time complexity of sampling is $O(2L + \frac{n}{L^2})$. Let $f(L) = 2L + \frac{n}{L^2}$, where $L > 0$. The function $f$ reaches its minimum when its derivative $f'(L) = 2 - 2nL^{-3} = 0$, i.e., $L = \sqrt[3]{n}$. In practice we set $L = \lceil \sqrt[3]{n} \rceil$, and the time complexity of sampling is $O(2 \lceil \sqrt[3]{n} \rceil + \frac{n}{\lceil \sqrt[3]{n} \rceil^2})$ $= O(\sqrt[3]{n})$.

**Algorithm 28:** Three Level Search Update

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, partial sum array $WD = [wd_0, wd_1, ..., wd_{WDn-1}]$ , sum $S = \sum_{i=0}^{n-1} w_i$ and the new weight $w'_i$

1. $L = \lceil \sqrt[3]{n} \rceil$, $r = \left\lfloor \frac{i}{L} \right\rfloor$, $r2 = \left\lfloor \frac{i}{L^2} \right\rfloor$
2. $\Delta = w'_i - w_i$
3. $wd_r = wd_r + \Delta$, $wd2_{r2} = wd2_{r2} + \Delta$, $S = S + \Delta$
4. $w_i = w'_i$

### 3.1.1 Adding More Levels

Compared with the two level search, three level search decreases the sampling time to $O(\sqrt[3]{n})$ while keeping updating time $\Theta(1)$. If we add more levels, the sampling time decreases further to $O(\sqrt[4]{n})$, $O(\sqrt[5]{n})$, ... while the updating time remains $\Theta(1)$. However, it is obvious that by adding levels, we need to go through more levels in both sampling and updating, which involves more operations. So the actual running time can increase. In order to make it fast in practice, we need to consider the trade off between the search time in each level and the number of levels, which becomes a practical problem. We will discuss it further in Section 5.2.

Note that since $L \geq 2$, if we keep adding levels, the method will eventually become a binary tree search with both sampling time and the updating time being $O(\log(n))$. Please refer to Section 2.8 for a space optimized binary tree search method.

## 3.2 Flat Method

On the base of the Rejection method, T. Hagerup, K. Mehlhorn and I. Munro proposed a sampling method that achieves both O(1) sampling and O(1) updating time complexity [6]. The method requires that the maximum weight is bounded by a polynomial in n [12], which is a strong assumption. The authors made the assumption because they need to "compress" the weight arrays to

certain sizes and use a finite set of integers to encode them. While we cannot use this method as we do not make the same assumption, we use their building blocks to compress $W$: we partition the weights into different groups such that applying the rejection method in each group becomes efficient. But instead of integer encodings and lookup tables, we use a linear search to select the proper group in the sampling. We implemented this method denoted as flat method and tested it in practice.

To compress $W$, we partition the weights $w_i$ into different groups, such that group $G_j$ only contains weights $w_i \in [2^{j-1}, 2^j)$. After setting a bound $B_j = 2^j$ for group $G_j$, the expected sampling time for the rejection method within each group becomes $\mathrm{O}(\frac{n*2^j}{S}) = \mathrm{O}(\frac{n*2^j}{2^{j-1}*n}) = \mathrm{O}(1)$. In our implementation, we put small weights with $w_i < 1$ into a single group $G_0$ with bound B=1. The expected sampling time within $G_0$ is $\mathrm{O}(\frac{n_0}{S_0})$, where $n_0$ is the number of weights and $S_0$ is the sum of weights in $G_0$. Considering the possibility of selecting $G_0$ during sampling, the expected time becomes $\frac{S_0}{S} * O(\frac{n_0}{S_0}) = O(\frac{n_0}{S}) \leq O(\frac{n}{S}) = O(\frac{1}{\bar{w}})$, where $S$ is the sum of all weights and $\bar{w}$ is the average weight.

To construct the method, we need to manually set the maximum possible weight $M$. We maintain a group table $G$ to keep track of the weights partitioned into different groups. We have groups $G_0, G_1, ..., G_{\lfloor log(M) \rfloor + 1}$, where group $G_j$ with $j \geq 1$ contains all indices $i$ with weights $w_i \in [2^{j-1}, 2^j)$. $G_0$ contains all indices $i$ with weights $w_i < 1$. We use another table to record and maintain the position of $w_i$, including the group $w_i$ belongs to, and the index of $w_i$ in the group. In addition, we use $S$ to store the overall sum and $GS_j$ to keep track of the sum of all weights in $G_j$. We show the initialization of the data structures in Algorithm 29.

To sample from the distribution, we choose a real number d $\in$ [0, S) uniformly at random. We use a linear search in $GS$ to find the corresponding group, i.e., the smallest $j$ satisfying $gs_0 + gs_1 + ... + gs_j > d$. Then we run the rejection method with bound $B_j = 2^j$ in group $G_j$ to generate the result. We show the pseudo code in Algorithm 30.

To update a weight $w_i$ to $w_i'$, there are two possible cases:

**Algorithm 29:** Flat Method Construction

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$

**Output:** Group table $G$, group sum table $GS$ and the overall sum $S$

1. For each integer $i \in [0, n)$
   (a) $j = \lfloor log(w_i) \rfloor + 1$
   (b) $G_j.append(i)$
   (c) $S = S + w_i$, $GS_j = GS_j + w_i$

**Algorithm 30:** Flat Method Sampling

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, group table $G$, group sum table $GS$ and the overall sum $S$

**Output:** Sampling result $i$

1. $d = uniform\_real(0, S)$
2. $j = linear\_search(GS, d, 0)$
3. While true
   (a) $i = uniform\_int(0, len(G_j) - 1)$, $d = uniform\_real(0, 1)$
   (b) If $d < \frac{w[G_j[i]]}{2^j}$
      i. return $G_j[i]$

1. If $w_i$ and $w_i'$ belong to the same group $G_j$, we only need to update $W$, $GS_j$ and $S$.

2. If $w_i$ belongs to $G_j$ while $w_i'$ belongs to $G_{j'}$, where $j \neq j'$, we need to erase $i$ from $G_j$ and append it to the back of $G_{j'}$. Then we update $W$, $GS_j$, $GS_{j'}$ and $S$ accordingly.

We show the pseudo code in Algorithm 31.

---

**Algorithm 31:** Flat Method Update

---

**Input:** Weight array $W = [w_0, w_1, ..., w_{n-1}]$, group table $G$, group sum table $GS$, the overall sum $S$ and the new weight $w_i'$

1. $j' = \lfloor log(w_i') \rfloor + 1$
2. $j = \lfloor log(w_i) \rfloor + 1$, $index =$ index of $w_i$ in $G_j$
3. If $j = j'$
   (a) $S = S - w_i + w_i'$, $GS_j = GS_j - w_i + w_i'$
4. Else
   (a) $S = S - w_i$, $GS_j = GS_j - w_i$
   (b) $swap\_and\_remove(G_j, index)$
   (c) $S = S + w_i'$, $GS_{j'} = GS_{j'} + w_i'$
   (d) $GS_{j'}.append(i)$
5. $w_i = w_i'$

---

The time complexity of construction is $\Theta(n)$ as we only need to loop through all weights once. During sampling, it takes $O(\log(M))$ time to run the linear search in $GS$. The expected sampling time within each group is $O(1)$, except for group $G_0$, where $O(\frac{1}{\bar{w}})$ sampling time is needed. As a result, the overall expected sampling time is $O(\log(M) + \frac{1}{\bar{w}})$. If we assume that $\bar{w} \geq c$ for constant $c > 0$, the expected sampling time becomes $O(\log(M))$. The time complexity of updating is $\Theta(1)$.

# Chapter 4

# Experiments

In this chapter, we describe our experiments. They are divided into two major parts:

- We designed and built a test platform for all sampling methods to check if our implementations work as expected.
- To compare the practical performance of all sampling methods, we collected actual runtime data and built an empirical runtime model, using linear regression to fit the model to the data.

We show our test environment in Section 4.1, introduce the test platform in Section 4.2, and the empirical runtime model in Section 4.3.

## 4.1 Test Environment

Our test environment contains an 8 core processor Intel i7-6700HQ (2.60 GHz) with 16 GB memory. Our program runs on Ubuntu 16.04 LTS and is compiled with O2 optimization under gcc 5.4.0. All methods are implemented in C++11 [8].

## 4.2 Test Platform

The test platform contains three parts: unit test, assertion test and statistical test. We used the factory method pattern [5] to create the objects for all methods. The C++ implementations of all classes share a base class *baseDiscrete*

with a constructor, a sampling function *sample*() and a updating function *update*($index, weight$). By using factory classes, we can build a unified test procedure to test all methods without writing code for each case separately.

### 4.2.1 Unit Tests and Assertions

For each method, we define unit tests to check if the values in the data structures are consistent with our expectation. For methods with simple data structures such as binary search and two level search, we directly check if the values are correct after a series of calls to *update*($index, weight$) and *sample*(). For methods with complex data structures such as the log* method, we check the internal consistency of the values in the data structures after each *update*($index, weight$) and each *sample*() call.

In addition to the unit tests, we developed simple assertion based tests to check the basic correctness of all the methods: For each method, we construct a random discrete distribution of size $n \in [10, 30000]$. Each weight $w_i \in [0, 10000)$ is chosen uniformly at random. We randomly select $\left\lfloor \frac{n}{2} \right\rfloor$ indices $i$ and update the corresponding weights $w_i$ with new weights also chosen from $[0, 10000)$. Then we generate $n$ samples and assert that no sampling results are invalid, i.e., out of the range of $[0, n)$. We repeat this test for 10 times for each method.

### 4.2.2 Statistical Tests

The unit and unified tests only check the basic correctness of the methods. Sampling methods also need to sample properly according to the specified underlying distribution. For each method, we used Pearson's chi-squared goodness of fit test [14] to establish whether the observed frequency distribution of our sampling result differs significantly from the true distribution. In the test, we build each method according to a weight array $W$ with size $n = 1000$. The chi-squared test requires that each index $i$ should have at least 5 expected counts in sampling reference, so we select weights $w_i \in [10, 2000]$ uniformly at random. We construct a sampling method for these weights. During sampling, we generate $10^7$ samples, such that the least expected count for each index $i$ is

| | $\chi^2$ | P-value |
|---|---|---|
| Alias Method | 959.1994 | 0.8126 |
| C++ discrete | 962.9184 | 0.7888 |
| Linear search | 991.1013 | 0.5644 |
| Binary search | 977.3959 | 0.6815 |
| Two level search | 1019.5578 | 0.3186 |
| Three level search | 1000.8893 | 0.4772 |
| Binary tree search | 951.8387 | 0.8548 |
| Rejection Method | 973.7842 | 0.7102 |
| Flat Method | 932.7314 | 0.9334 |
| Log* Method | 977.5903 | 0.6799 |

Table 4.1: The $\chi^2$ values and the corresponding p-values in Pearson's chi-squared goodness of fit tests for all methods.

$10^7 \times \frac{10}{2000 \times 1000} = 50 > 5$. To make sure that the updating also works properly, before we start sampling, we randomly select 500 indices $i$ and update the corresponding weights $w_i$ with new weights also chosen uniformly at random from [10,2000].

For the chi-squared test, our null hypothesis is $H_0$: The probability of choosing an index $i$ as the sampling result is equal to $p_i = \frac{w_i}{S}$ for $0 \leq i < n$, where $S = \sum_{i=0}^{n-1} w_i$. Our alternative hypothesis is $H_a$: At least one probability is not equal to $p_i$. In each test, the degree of freedom is $n - 1 = 999$. After $10^7$ samples, the expected count for each index $i$ is $expected_i = 10^7 \frac{w_i}{\sum_{j=0}^{n-1} w_j}$. While the observed count for each index $i$ is $observed_i$, we calculate $\chi^2 = \sum_{i=0}^{n-1} \frac{(observed_i - expected_i)^2}{expected_i}$. Referring to the $\chi^2$ table, we can get the P-value representing the probability of observing a sample statistic as extreme as the test statistic.

Table 4.1 shows our test results. As all the p-values are greater than 0.05, we can conclude that, at the significance level of $\alpha = 0.05$, there is not enough evidence to reject $H_0$, which is evidence that our methods generate samples properly according to the distribution.

## 4.3 Performance Measurement

In this thesis, our goal is to find efficient sampling methods in practice. To compare the practical performance of all methods in different scenarios, we need to clarify the factors that influence the performance, such that we can collect data points and build our model based on these factors. We introduce these factors in Section 4.3.1. Then we show how we collect data points and build the model in Sections 4.3.2 and 4.3.3. At last, we show the experimental results in Section 4.3.4.

### 4.3.1 Factors Influencing the Performance of Random Sampling

The practical performance of each method is affected by multiple factors. As far as we understand, the major factors are the number of samples generated, the number of updates, the size of the weight array, and the distribution from which weights $w_i$ are drawn. Based on these factors, we are able to develop a model $time(s, u, n)$ to predict the runtime for each method. In $time(s, u, n)$, $s$ is the number of random samples, $u$ is the number of weight updates, $n$ is the size of weight array $W$.

### 4.3.2 Determining time(s, u) for fixed n

To determine what $time(s, u, n)$ should look like, we first fix some values of $n$ and derive a linear model $time(s, u) = a + bs + cu$. In $time(s, u)$, $s$ is the number of samples and $u$ is the number of updates. $b$ and $c$ represent the average runtime for one sample and one update respectively. $a$ represents the construction time. The correctness of $time(s, u)$ is obvious: with larger number of samples and updates, the actual runtime should increase linearly.

We need to choose the set of fixed $n$ values $N = [n_0, n_1, ...]$ carefully, such that it will be helpful when we fit a $time(s, u, n)$ model to a set of $time(s, u)$ models. Starting from $n_0 = 10$, we set $n_{i+1} = \lfloor 1.1 \cdot n_i \rfloor$ for $0 \leq i < 84$, with the maximum $n_{84} = 29991$.

We collected data points for each fixed $n$ and for each method. With

weights $w_i$ drawn from a uniform distribution U(0, $10^4$) for $0 \leq i < n$, we gathered data points $(s, u, t)$, where $t$ is the actual time spent in our experiment for $s$ samples and $u$ updates. We recorded $t$ for the following samples:

1. $s = 0, 10000, 20000, ..., 10^6$ and $u = 0$.

2. $s = 0, 10000, 20000, ..., 10^6$ and $u = s$, where we make one update right before each sample.

In each data point $t$ is averaged over three runs.

Given data points $(s, u, t)$, we used an ordinary least squares (OLS) method [4] to derive the coefficients in $time(s, u)$. To estimate each coefficient more accurately, we use the following steps to estimate them one by one:

1. Based on all data points $(s, u, t)$ with $s = 0$, $u = 0$, we use OLS to estimate coefficient $a$.

2. Based on estimated $a$ and all data points $(s, u, t)$ with $s \neq 0$, $u = 0$, we use OLS to estimate coefficient $b$.

3. Based on estimated $a, b$ and all data points $(s, u, t)$ with $s \neq 0$, $u \neq 0$, we use OLS to estimate coefficient $c$.

For 10 sampling methods and 85 values of $n$, we derived $10 \times 85 = 850$ $time(s, u)$ models with different fixed $n$. We list all these models in Appendix A.

### 4.3.3 Determining time(s, u, n)

In order to make the model useful in practice, we need to study its dependence on $n$. For each method, we use one $time(s, u, n)$ model to summarize all 85 $time(s, u)$ models with different $n$. Theoretically, along with the increase of $n$, the increase of actual construction, sampling and updating time should fit the runtime complexity analysis. While the construction time of all methods is linearly dependent on $n$, for different methods, the factor $n$ has different influences on sampling and updating time. Thus we can write $time(s, u, n) = an + bs \cdot f(n) + cu \cdot g(n)$, where $s$ is the number of samples, $u$ is the number

| Method | f(n) | g(n) |
|---|---|---|
| Alias Method | 1 | $n$ |
| C++ discrete | $\log(n)$ | $n$ |
| Linear search | $n$ | 1 |
| Binary search | $\log(n)$ | $n$ |
| Two level search | $\sqrt{n}$ | 1 |
| Three level search | $\sqrt[3]{n}$ | 1 |
| Binary tree search | $\log(n)$ | $\log(n)$ |
| Rejection Method | 1 | 1 |
| Flat Method | 1 | 1 |
| Log* Method | 1 | 1 |

Table 4.2: The terms $f(n)$ and $g(n)$ in $time(s, u, n)$ for all methods.

of updates and $n$ is the size of the weight array. Note that the meaning of $a$, $b$ and $c$ is different from that in $time(s, u)$ models.

To train the $time(s, u, n)$ models using the data points $(s, u, t)$ we gathered in Section 4.3.2, we can fit the model to the data points directly and use linear regression to minimize the sum of mean squared loss [10]. However, this approach could lead to overfitting [11] that reduces the generalizability of our models outside the original dataset: When we used this approach to derive the model, some of the coefficients $a$, $b$ or $c$ in $time(s, u, n)$ were determined as negative numbers.

To avoid this issue, we used linear regression to fit $an$, $b \cdot f(n)$ and $c \cdot g(n)$ in $time(s, u, n)$ to all $a$s, $b$s and $c$s in 85 $time(s, u)$ models. Given $n$, we can use the $time(s, u, n)$ model to predict the $a$, $b$ and $c$ in the $time(s, u)$ model first, and then use $time(s, u)$ model to predict the runtime. The terms $f(n)$ and $g(n)$ for each method are shown in Table 4.2. Note that the sampling and updating runtime complexity of the rejection method, flat method and log* method is not $\Theta(1)$. But since $n$ has very little influence on their runtime in practice, we simply set $f(n) = 1$ and $g(n) = 1$ in their $time(s, u, n)$ models.

While our $time(s, u, n)$ models make sense according to theoretical runtime complexity analysis, we use Appendix B to demonstrate the reasonability of this approach in practice: for each method, we use three plots to show all $a$s, $b$s and $c$s with different $n$ in 85 $time(s, u)$ models as data points. Then we

draw the curves according to the theoretical runtime complexity to show how well our models fit these data points.

In order to illustrate how the distribution from which weights $w_i$ are drawn influences the practical performance, we repeated the whole experiment with weights drawn from an exponential distribution with $\lambda = 10^{-3}$. In order to avoid potential numerical error, we still use $10^4$ as the maximum weight.

### 4.3.4 Experimental Results

Our $time(s, u, n)$ models for all 10 methods are shown in Table 4.3. Comparing the two $time(s, u, n)$ models for each method, the performance of most methods is not affected by the distribution from which weights $w_i$ are drawn. However, the distribution had a significant influence on the rejection method. This result is consistent with the time complexity analysis: For the time complexity of $O(\frac{nB}{S})$, the performance will get worse if the sum of weights $S$ decreases while the bound $B$ remains the same. We calculated the average $\frac{nB}{S}$ values for both models when we collected the data for $n = 29991$. When the weights are drawn from the uniform distribution, the average $\frac{nB}{S} = 2.0007$. When the weights are drawn from the exponential distribution, the average $\frac{nB}{S} = 10.0089$.

For linear search, we also notice a big gap between coefficient $c$s in the two models: with all other coefficients being similar, we got $c = 0.0037$ for weights drawn from the uniform distribution, while we got $c = 0.0007$ for weights drawn from the exponential distribution. However, the time spent by linear search for time-changing discrete distributions is dominated by the time needed to generate samples. The big gap between coefficient $c$s does not make a big difference: Figure 4.1 compares the time spent for linear search to generate samples for two distributions. Even with a large number of 30 updates after each sample, the performance of linear search for both distributions is still very close.

| Method | Distribution | $time(s, u, n)$ |
|---|---|---|
| Alias Method | Uniform | $0.0246n + 0.0400s + 0.0225un$ |
| | Exponential | $0.0211n + 0.0404s + 0.0209un$ |
| C++ discrete | Uniform | $0.0144n + 0.0064s\log(n)$ $+0.0057un$ |
| | Exponential | $0.0112n + 0.0066s\log(n)$ $+0.0057un$ |
| Linear search | Uniform | $0.0014n + 0.0006sn + 0.0037u$ |
| | Exponential | $0.0013n + 0.0006sn + 0.0007u$ |
| Binary search | Uniform | $0.0048n + 0.0063s\log(n)$ $+0.0026un$ |
| | Exponential | $0.0037n + 0.0065s\log(n)$ $+0.0026un$ |
| Two level search | Uniform | $0.0029n + 0.0012s\sqrt{n} + 0.0050u$ |
| | Exponential | $0.0025n + 0.0012s\sqrt{n} + 0.0063u$ |
| Three level search | Uniform | $0.0029n + 0.0024s\sqrt[3]{n} + 0.0108u$ |
| | Exponential | $0.0025n + 0.0025s\sqrt[3]{n} + 0.0122u$ |
| Binary tree search | Uniform | $0.0012n + 0.0065s\log(n)$ $+0.0038u\log(n)$ |
| | Exponential | $0.0009n + 0.0068s\log(n)$ $+0.0037u\log(n)$ |
| Rejection Method | Uniform | $0.0009n + 0.0787s + 0.0012u$ |
| | Exponential | $0.0008n + 0.2192s + 0.0977u$ |
| Flat Method | Uniform | $0.0720n + 0.0886s + 0.0634u$ |
| | Exponential | $0.0660n + 0.0871s + 0.0637u$ |
| Log* Method | Uniform | $0.0443n + 0.2080s + 0.4466u$ |
| | Exponential | $0.0430n + 0.2165s + 0.4989u$ |

Table 4.3: The $time(s, u, n) = an+bs\cdot f(n)+cu\cdot g(n)$ models for all 10 sampling methods. For each method, we show the two models with weights $w_i$ drawn from the uniform distribution U(0, $10^4$) and the exponential distribution ($\lambda = 10^{-3}$) separately. The time was measured in microseconds. All coefficients were rounded to four decimal places.

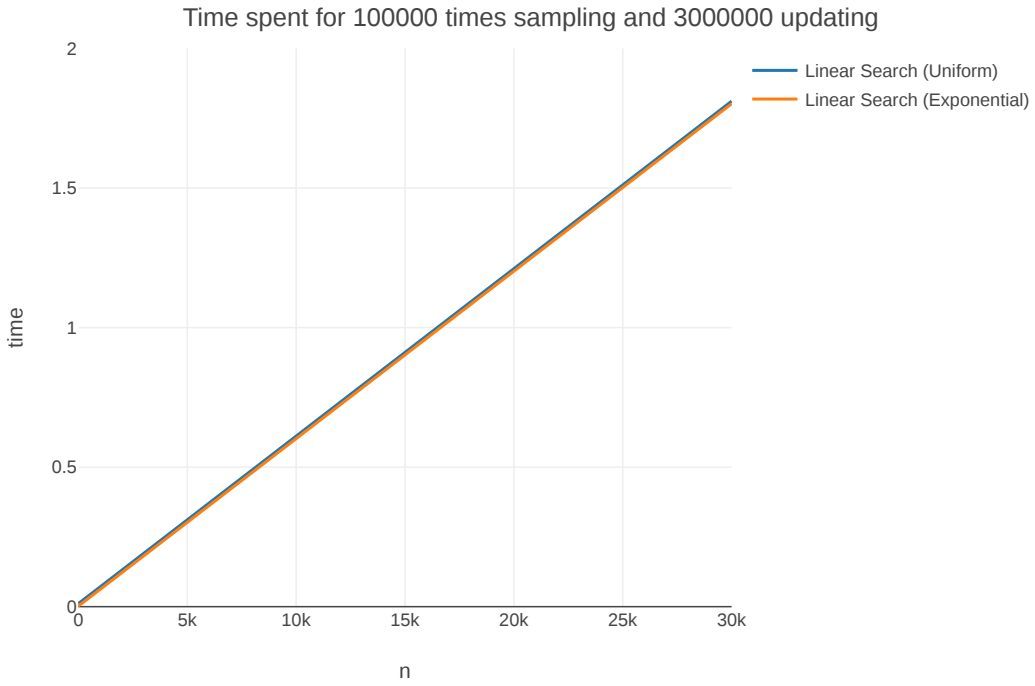



Figure 4.1: The plot shows time spent for linear search to generate $10^5$ samples with 30 weight updates after each sample. We used both models derived for the uniform distribution and the exponential distribution. The time was measured in seconds.

# Chapter 5

# Analysis of Experimental Results

In this chapter, we analyze the $time(s, u, n)$ models we derived in Chapter 4. In Section 5.1, we give examples of how we apply the models in the scenarios we designed to find suitable methods. In Section 5.2 and 5.3, we analyze and summarize the models to make conclusions about their practical performance. We focus on two groups of methods, namely leveling and rejection-based methods, and develop criteria to choose the best method for a given application.

## 5.1 Applying time(s,u,n) Models

Using the $time(s, u, n)$ models, users are able to find the most suitable methods for their application scenarios. In the following plots, we show two representative scenarios using $time(s, u, n)$ models shown in Table 4.3 with weights drawn from the exponential distribution: Figure 5.1 compares the time spent for all methods to generate $10^5$ samples without updating, while Figure 5.2 compares the time spent to generate $10^5$ samples, with one update after each sample.

In Figure 5.1, the best methods for different values of $n$ are: two level search for $n \in [10, 82)$, three level search for $n \in [82, 4488)$, and Alias method for $n \in [4488, 30000]$. Looking back at the raw data we collected in Section 4.3.2, the best methods for different values of $n$ are: two level search for most $n \in [10, 67)$, and Alias method for $n \in [67, 29991]$. The Alias Method is the

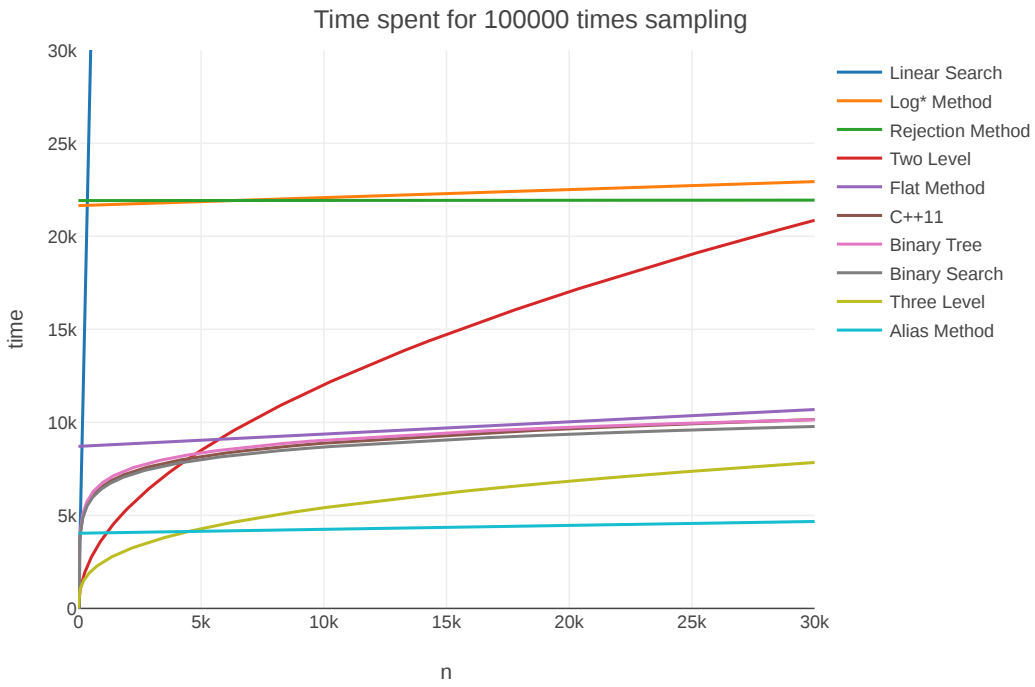



Figure 5.1: The plot shows time spent for all methods to generate $10^5$ samples without updating. The weights $w_i$ are drawn from the exponential distributions with $\lambda = 10^{-3}$ and the time were measured in microseconds.

best choice for most $n$ both in the prediction of our model and in the raw data. We can conclude that the traditional Alias Method is still the overall best choice for sampling from a large static discrete distribution.

In Figure 5.2, with updates, the best methods for different values of $n$ are: linear search for $n \in [10, 18)$, two level search for $n \in [18, 423)$, and three level search for $n \in [423, 30000]$. Looking back at the raw data we collected in Section 4.3.2, the best methods for different values of $n$ are: linear search for $n \in [10, 24)$, two level search for most $n \in [24, 548)$, and three level search for $n \in [548, 29991]$. The best methods in the raw data match our predictions in general, which is strong evidence that our models are capable of describing the data and making useful predictions.

For a changing distribution, all methods with O(n) sampling or updating time, namely the Alias Method, std::discrete_distribution provided by C++11, binary search and linear search, perform poorly for large $n$. They are barely visible on the left side of Figure 5.2 since their runtime is not competitive at all. We can conclude that these methods are not suitable for sampling from a

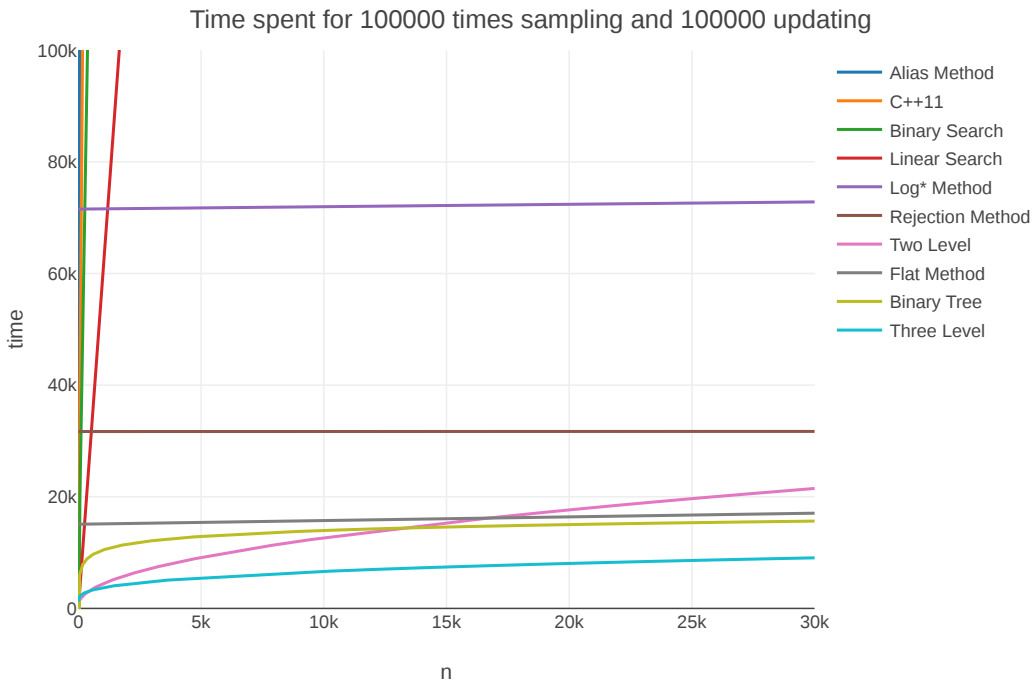



Figure 5.2: The plot shows time spent for all methods to generate $10^5$ samples with one weight update after each sample.

large time-changing discrete distribution.

## 5.2 Summary of Performance for Leveling Methods

We call two level search, three level search and the binary tree search the leveling methods. These methods use simple level structures to keep both sampling and updating efficient. In the scenario of Figure 5.2, these methods have superior performance. Among leveling methods, two level search dominates the others for $n < 423$. When n grows larger, three level search gradually takes over and performs the best. Theoretically, binary tree search with its O(log(n)) runtime complexity should eventually outperform all of the other leveling methods with their polynomial runtime complexity, if n grows large enough. However, our experiment shows that for $n \leq 30000$, three level search remains much faster.

A related question for the leveling methods is: as n grows, is it worthwhile to continually add more levels to the search? To answer this question, we

| Method | Distribution | $time(s, u, n)$ |
|---|---|---|
| Four level search | Uniform | $0.0033n + 0.0049s\sqrt[4]{n} + 0.0143u$ |
| | Exponential | $0.0027n + 0.0050s\sqrt[4]{n} + 0.0151u$ |
| Five level search | Uniform | $0.0031n + 0.0102s\sqrt[5]{n} + 0.0149u$ |
| | Exponential | $0.0027n + 0.0111s\sqrt[5]{n} + 0.0161u$ |

Table 5.1: The $time(s, u, n) = an + bs \cdot f(n) + cu \cdot g(n)$ models for four and five level search. For each method, we show the two models with weights $w_i$ drawn from the uniform distribution U(0, $10^4$) and the exponential distribution ($\lambda = 10^{-3}$) separately. The time was measured in microseconds. All coefficients were rounded to four decimal places.

implemented two additional methods and repeated the experiments in Section 4.3 to derive the $time(s, u, n)$ models for them:

1. A four level search with O($\sqrt[4]{n}$) sampling and O(1) updating runtime complexity.

2. A five level search with O($\sqrt[5]{n}$) sampling and O(1) updating runtime complexity.

Table 5.1 shows the experimental results, and Figure 5.3 compares the performance of all leveling methods using the models derived for exponential distributions. The best methods for different values of $n$ are: two level search for $n \in [10, 423)$, three level search for $n \in [423, 8340)$, and four level search for $n \in [8340, 30000]$. By adding an extra level, four level search is 10% faster than three level search for $n = 3 \times 10^4$. For five level search, we calculated its estimated time spent for $n = 10^6$ and compared it with other methods: While it is still slower than four level search, it also becomes slower than binary tree search. As a result, we may conclude that having five or more levels in search is never the best for these scenarios.

For the case of multiple updates after each sample, search with more levels needs more time to update the partial sum on each level. In practice, the increased overhead from extra levels makes four level search a good compromise for large $n$. For small $n$, two and three level search are good choices.

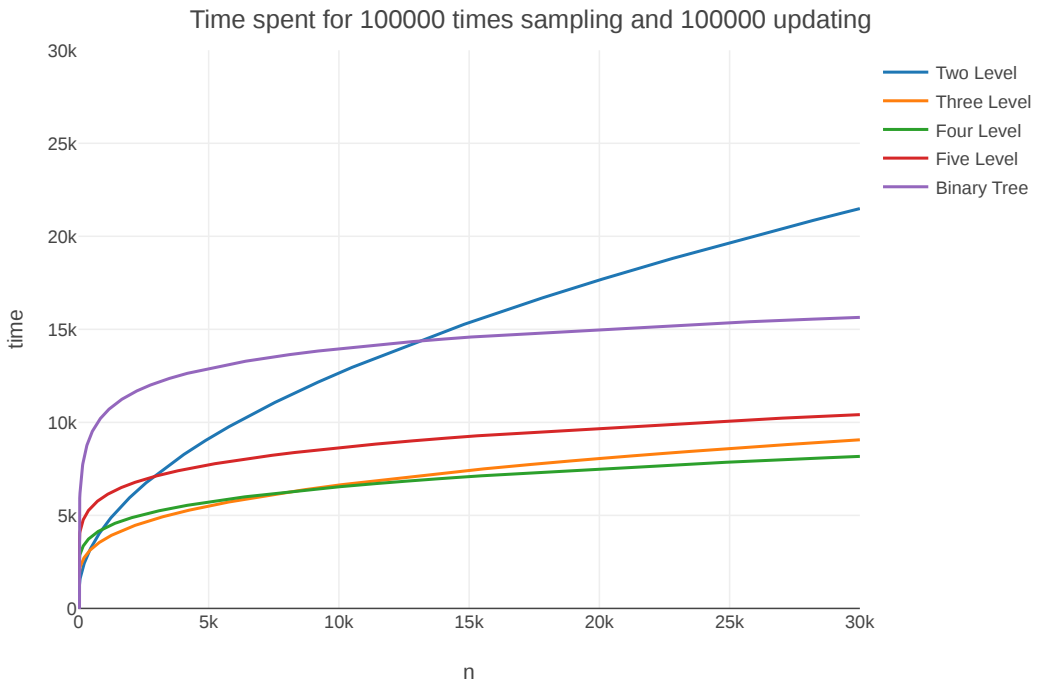



Figure 5.3: Time for two, three, four, and five level search, and binary tree search to generate $10^5$ samples with one weight update after each sample.

## 5.3 Summary of Performance for Rejection-Based Methods

The group of rejection-based methods contains the rejection method, the flat method and the log* method. These methods are built based on rejection sampling, and their performance is not or only slightly affected by n. Figure 5.4 and Figure 5.5 compares the performance of all rejection-based methods for weights drawn from different distributions. The performance of the rejection method was much worse when the weights were drawn from an exponential distribution instead of a uniform distribution. If a discrete distribution has few large weights and many small weights, we expect its performance to be bad.

For the flat method, its performance is affected by the maximum possible weight $M$. We adjusted the maximum possible weight $M$ from 10000 to 100, and repeated the experiments in Section 4.3 to derive a $time(s, u, n)$ model for the flat method. In the experiments, the weights $w_i$ are drawn from an

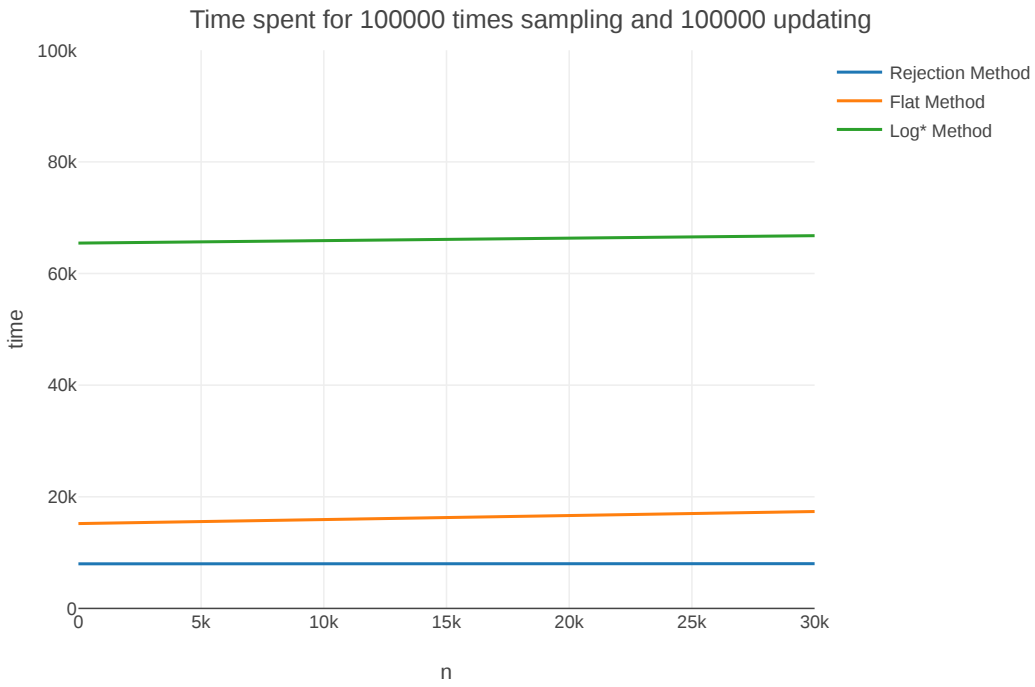



Figure 5.4: Time for the rejection method, the flat method and the log* method to generate $10^5$ samples with one weight update after each sample. The plot is generated using the models derived for weights drawn from uniform distributions.

exponential distribution with $\lambda = 10^{-3}$. Using this model, Figure 5.6 compares the performance of all rejection-based methods again. Compared with Figure 5.5, the performance of the flat method is slightly better in this experiment: for $n = 3 \times 10^4$, it's 24% faster. This result indicates that the flat method can be considered as suitable if the ratio of maximum weight to average weight is small.

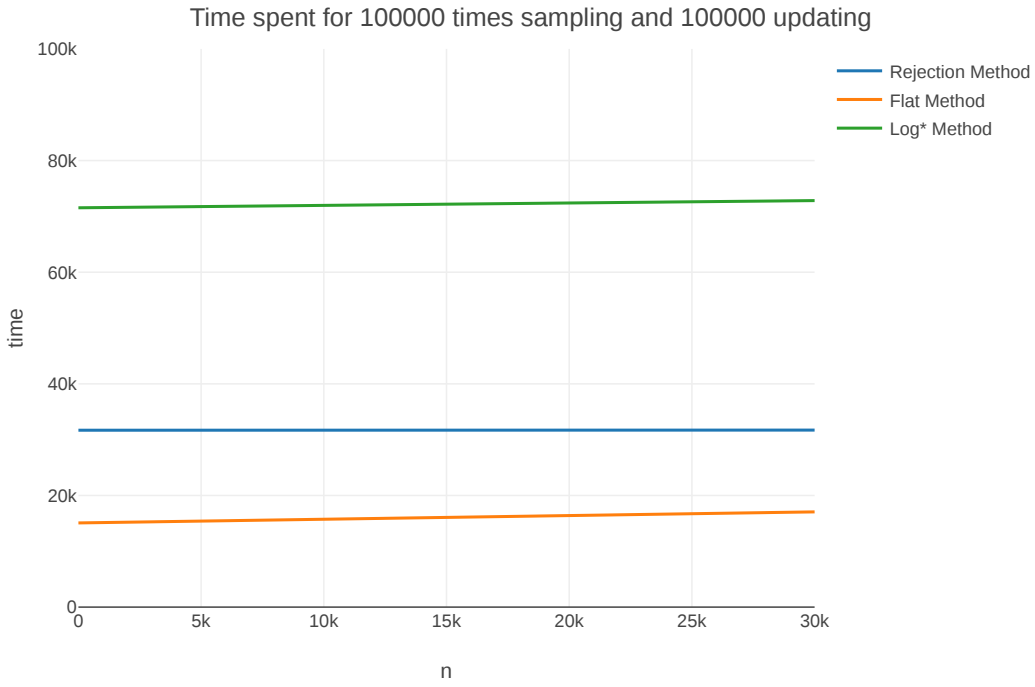



Figure 5.5: Time for the rejection method, the flat method and the log* method to generate $10^5$ samples with one weight update after each sample. The plot is generated using the models derived for weights drawn from exponential distributions.

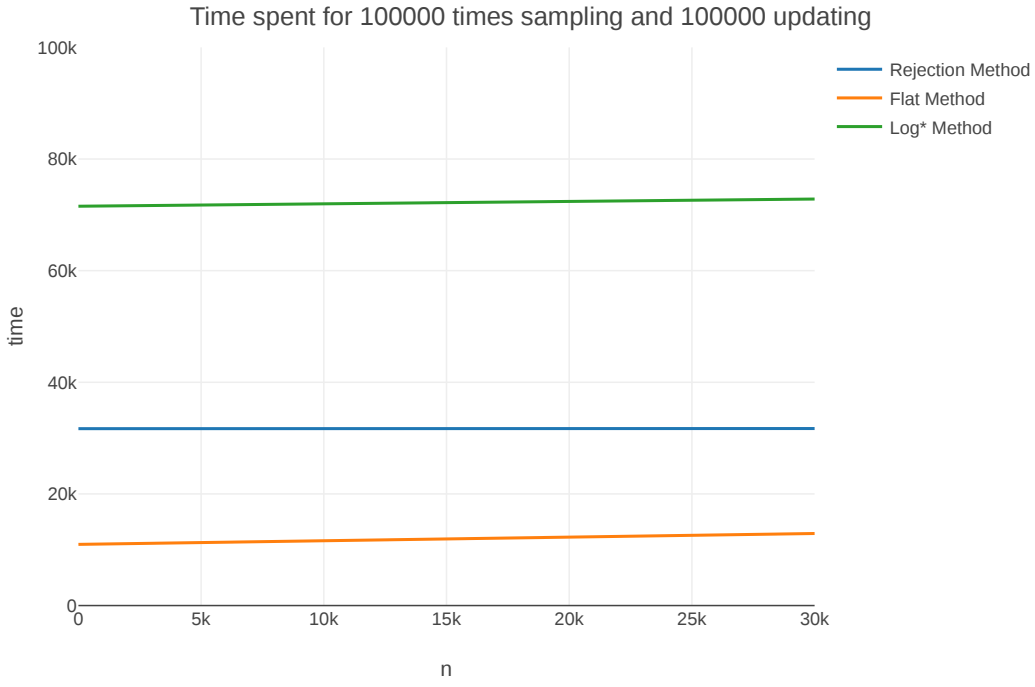



Figure 5.6: Time spent to generate $10^5$ samples with one update after each sample. The plot is generated using the models derived for weights drawn from exponential distributions.

# Chapter 6

# Application: Fast Sampling for Monte Carlo Playouts in NoGo

In this chapter, we use the game of No Capture Go (NoGo) as our application to demonstrate how we use our methods to optimize the sampling process. We speed up the randomized playout process in a Monte Carlo Tree Search algorithm [2] for playing NoGo. Faster playouts allow an engine to search more states in the same time, making it stronger.

## 6.1 The Game of NoGo

NoGo is a variant of the famous ancient Asian game of Go. It is first mentioned by John Moore in 2005 [13]. Similar to Go, the game is played by two players Black and White on a square $n \times n$ grid, called the board. The intersections of the grid are called points. In each game, the two players take turns to place a single stone of their own color on an empty point until the end. The connected components of stones of the same color are called blocks. The empty points adjacent to a block are called its liberties. Each block must have at least one liberty. Different from Go, both capture and suicide are illegal in NoGo. Players are not allowed to pass, and a player whose turn it is loses the game if there is no legal move. Figure 6.1 shows an example of the end of the game. The game was played on the standard $9 \times 9$ board. The last stone was played by Black on H6. There is no legal move for White to play, since all moves on empty points are either suicide or capture. For example, *A2* is suicide since

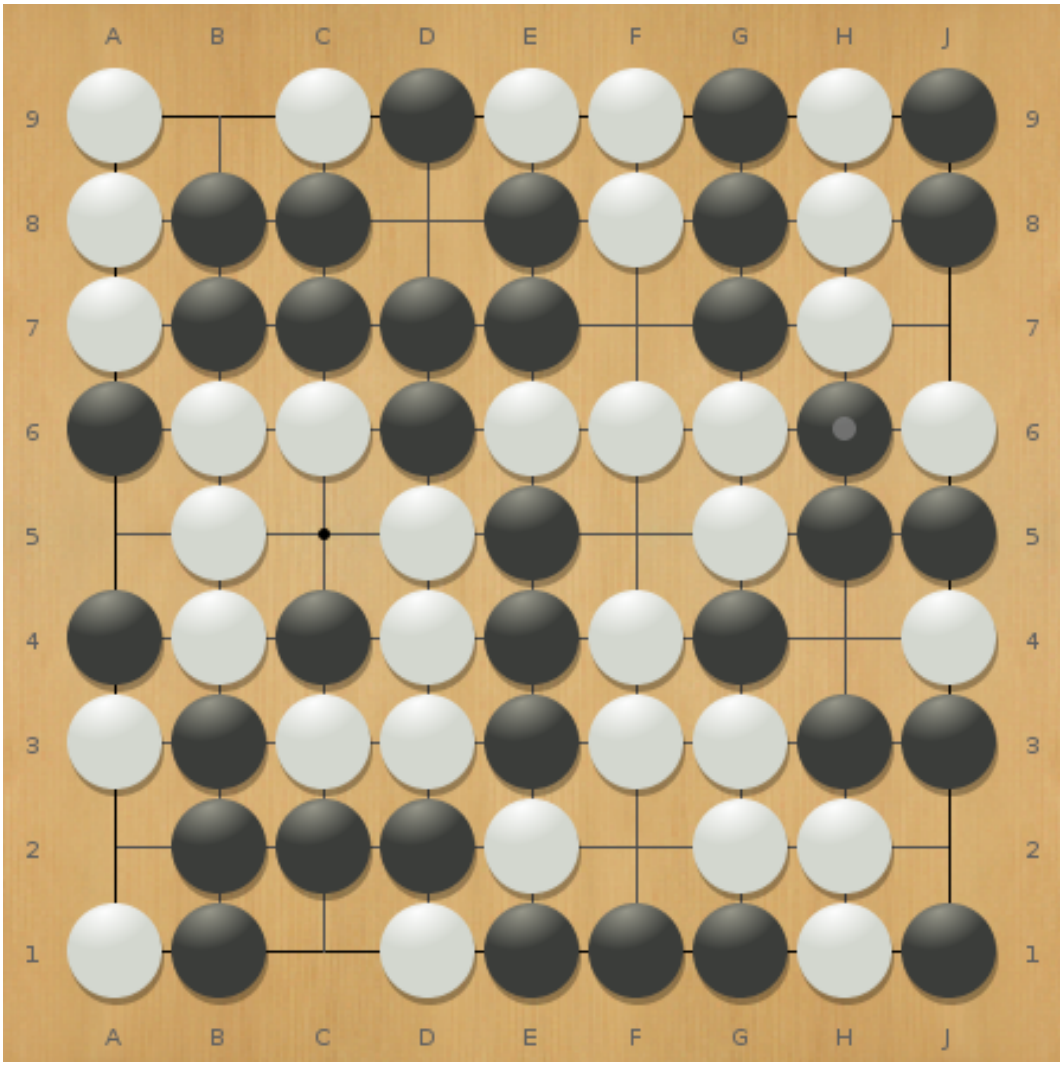



Figure 6.1: An example of the end of a NoGo game

it is the last liberty of blocks *A1* and *A3*. *F2* is illegal since it would capture three black stones on *E1*, *F1* and *G1*. White has no legal move and loses the game.

## 6.2 Playouts in NoGo

Computer programs that play board games such as NoGo often use playouts to evaluate board positions. In each playout, we play the game from a given board position until the end, with moves of both players selected in a randomized fashion. Playouts provide useful win/loss statistics, which are then used as a

heuristic to determine how likely one player is to win the game.

To improve the quality of these win/loss statistics, we need to generate high quality moves instead of purely random moves in the playouts. A common way is to use machine learning to imitate the moves played by human experts, as was done in early versions of AlphaGo [17]. Since there are no human experts playing NoGo, we used supervised learning to learn moves generated by the strong program BobNoGo [7]. Each legal move is evaluated by a single $3 \times 3$ feature that records the color of the eight neighbors surrounding this move. Each neighbor can be empty, an opponent's stone, the player's stone, or off the board, so there are up to $4^8 = 65536$ possible patterns. After we learn the weight of each pattern, the whole board can be viewed as a weight array $W$ of size $9 \times 9 = 81$, and we can sample a move with probability proportional to $W$ to decide the next move in a playout. After each move, only the weights of that move and its up to 8 neighbors need to be updated, since only their $3 \times 3$ features change.

We used Remi Coulom's Minorization-Maximization supervised learning technique [3] to learn the weights of all features. Coulom used a training set containing 131939 moves for Go. To achieve a similar size of the training set, we let BobNoGo play against itself for 2772 games, which generates 168600 moves. We extracted 140560 moves from 2310 games as the training set, and 28040 moves from the remaining 462 games as the test set. After learning, we achieved an accuracy of 20.1% on the test set. Although it is less than Remi Coulom's accuracy of 34.9% in Go (with a larger Go-specific feature set), we consider it as sufficient for improving our playouts' quality.

## 6.3 Implementation

We simulate the playout process with the learned feature weights in NoGo. In each playout, we maintain a set of legal moves and the weight array $W$ of size 81 for each player. At the beginning of each turn, we sample according to the weight array to select the move. After each move, we go through the legal moves for both players to see if they are still legal, and update the weight

array if their weights changed.

We use Figure 6.2 to demonstrate the sampling and updating process. On

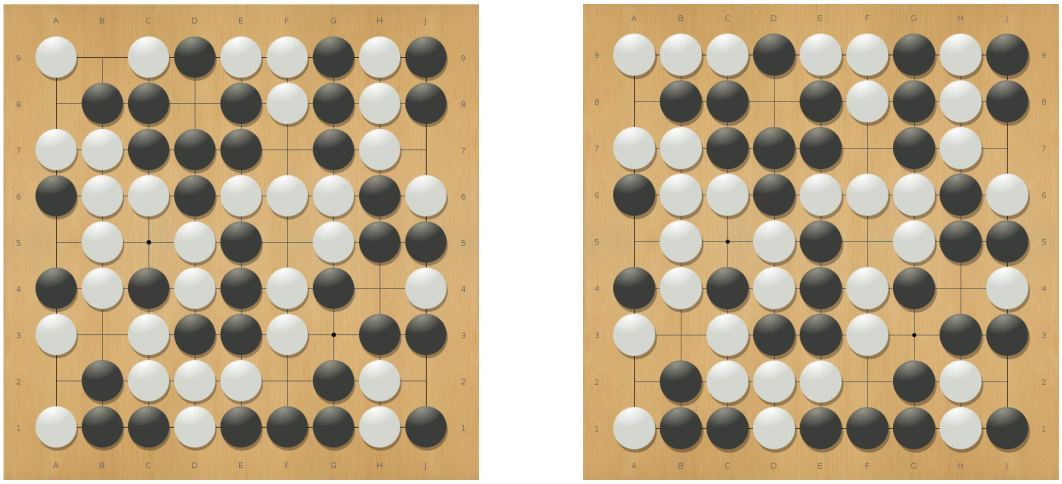


Figure 6.2: Boards before and after White makes a move on $B9$.

the board shown in Figure 6.2 on the left, it is White's turn to make a move. The legal moves for White are $B9, A8, B3, G3, A2, F2$, and the corresponding weights in the weight array $W$ are $w_{B9} = 1.541, w_{A8} = 0.797, w_{B3} = 0.4814, w_{G3} = 0.7574, w_{A2} = 1.541, w_{F2} = 1.6737$. All other weights in $W$ are 0. Similarly, the non-zero weights $w'_i$ for Black are $w'_{A8} = 0.9957, w'_{D8} = 0.3139, w'_{B3} = 0.5388, w'_{G3} = 0.4662, w'_{F2} = 0.4667$. We sample according to $W$ to decide a move for White. The plot in Figure 6.2 on the right shows the board after we choose $B9$ as the move. On this board, all non-zero weights for White are $w_{A8} = 1.6391, w_{B3} = 0.4814, w_{G3} = 0.7574, w_{A2} = 1.541, w_{F2} = 1.6737$, and all non-zero weights for Black are $w'_{D8} = 0.3139, w'_{B3} = 0.5388, w'_{G3} = 0.4662, w'_{F2} = 0.4667$. Only the weights of positions around $B9$, namely $w_{B9}, w_{A8}$ and $w'_{A8}$, are changed.

To sample from the weight array, we compare two methods: the most ordinary way is to use the C++ discrete_distribution class. We need to construct a discrete_distribution object with the current weight array each time before sampling. A more efficient way is to use our sampling methods. After constructing the data structures once at the beginning of the playout, we just need to make updates before sampling.

Based on our experiments, each game has 72 moves on average, and each

| Method | Actual runtime | Ranking | Predicted runtime | Ranking |
|---|---|---|---|---|
| Linear search | 0.0871 | **1** | 0.0122 | **1** |
| Two level search | 0.1012 | **2** | 0.0152 | **2** |
| Three level search | 0.1185 | **3** | 0.0267 | **3** |
| Binary tree search | 0.1445 | 5 | 0.0551 | 4 |
| Flat Method | 0.2325 | 6 | 0.1587 | 5 |
| Rejection Method | 2.0951 | 9 | 0.2375 | 6 |
| Binary search | 0.1244 | 4 | 0.4192 | 7 |
| Log* Method | 1.1752 | 8 | 1.0271 | 8 |
| Alias Method | 0.4545 | 7 | 3.3049 | 9 |

Table 6.1: The actual runtime and expected runtime with associated rankings for all methods to construct, sample and update in 3000 playouts. Runtime is measured in microseconds and rounded to four decimal places. The methods are sorted according to their predicted runtime.

move leads to 9 updates on average. As a result we have $s = 72$, $u = 648$ and $n = 81$ in our $time(s, u, n)$ models. For most turns, we expect there to be few good moves, lots of bad moves and a large number of illegal moves with weights equal to 0. So compared with weights drawn from a uniform distribution, weights drawn from an exponential distribution can better describe the weight array $W$ we have. According to Table 4.3, we found that linear search is the best method for our case.

To compare the sampling speed, we write two programs to simulate the playouts in NoGo. One program uses the ordinary sampling while another uses our linear search method. The two programs are exactly the same except for the sampling and weight updates parts.

## 6.4 Optimization Result

We use "g++ -std=c++11 -O2" to compile both programs. The ordinary sampling program spends 0.1462 seconds on sampling while the linear search program spends only 0.0871 seconds. By using the linear search method instead of ordinary sampling, we achieved 40% speedup in the sampling process.

We tested all our implemented sampling methods in this application for $9 \times 9$ board. Table 6.1 shows their actual runtime and expected runtime calculated using our models from Table 4.3. Although the actual runtime

| | Sampling runtime | Overall runtime |
|---|---|---|
| Ordinary sampling | 0.6792 | 4.6740 |
| Two level search | 0.1431 | 3.8702 |

Table 6.2: Sampling runtime and overall runtime for ordinary sampling and two level search to run 3000 playouts on a $19 \times 19$ board. Runtime is measured in microseconds and rounded to four decimal places.

and the predicted runtime for all methods are different as our model cannot perfectly describe the scenario, with Kendall rank correlation coefficient $\tau = 0.67$ [9], we may conclude that the two rankings are similar. In addition, the top 3 methods in both rankings are the same. This is strong evidence that our models are useful for selecting the best sampling methods in practice.

To illustrate how optimization in sampling process can affect the overall runtime, we also tested our sampling method against the ordinary way on a $19 \times 19$ board. With $n = 19 \times 19 = 361$, we chose two level search as the best method for this case according to Table 4.3. Table 6.2 shows the results. By using the two level search method instead of ordinary sampling, we achieved 79% speedup in the sampling process, which leads to a 17% speedup in the entire program.

# Chapter 7

# Conclusions and Future Work

In this thesis, we perform an empirical study of random sampling methods for time-changing discrete distributions: We review eight sampling methods and propose two new methods. We implemented all methods based on a unified framework, designed experiments to evaluate them, checked their correctness, and derived a series of models to evaluate their practical performance. We applied these models in an application to the game of NoGo to speed up the sampling process.

Future work includes:

1. Under the condition where multiple updates are required after each sample, study how batch update can be used to speedup the updating process.

2. Study how parallel computing can be used to speedup the sampling and updating processes. Measure and compare the practical performance of sampling methods running in parallel.

3. Develop an automated tool that can monitor the sampling requirements of an application and automate the process of selecting the best sampling method.

# Bibliography


[1] Boost. Boost C++ libraries version 1.68.0, 2018.

[2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[3] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. In *Computer games workshop*, 2007.

[4] Francis Galton. Regression towards mediocrity in hereditary stature. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 15:246–263, 1886.

[5] Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Pearson Education India, 1995.

[6] Torben Hagerup, Kurt Mehlhorn, and James Ian Munro. Optimal algorithms for generating discrete random variables with changing distributions. In *Proceedings, 20th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci*, volume 700, pages 253–264, 1993.

[7] Bob Hearn and Martin Müller. The BobNoGo program. `https://webdocs.cs.ualberta.ca/~mmueller/nogo/BobNoGo.html/`, 2011. [Online; posted 2011].

[8] ISO/IEC 14882:2011. *International Standard Programming Language C++*, Third edition, 2011.

[9] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[10] Erich L Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[11] David J Leinweber. Stupid data miner tricks: overfitting the s&p 500. *The Journal of Investing*, 16(1):15–22, 2007.

[12] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory of Computing Systems*, 36(4):329–358, 2003.

[13] John Moore. Anti atari go (version 1), April 2005. [Online; posted April-2005].

[14] Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[15] GNU Project. GCC 4.8.4. `https://gcc.gnu.org/`, December 2014.

[16] Keith Schwarz. Darts, dice, and coins: Sampling from a discrete distribution, December 2011. [Online; posted December-2011].

[17] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[18] Michael D Vose. A linear algorithm for generating random numbers with a given distribution. *Software Engineering, IEEE Transactions on*, 17(9):972–975, 1991.

[19] Alastair J Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256, 1977.

# Appendix A

# time(s,u) Models

$time(s,u) = a + bs + cu$ models for all 10 methods with 85 different $n$. We use a table to show all the coefficients in the models.

| Alias Method | | | | | | | |
|---|---|---|---|---|---|---|---|
| n | a | b | c | n | a | b | c |
| 10 | 5.6667 | 0.0386 | 0.4046 | 11 | 5.0 | 0.0398 | 0.4397 |
| 12 | 5.0 | 0.0394 | 0.474 | 13 | 4.3333 | 0.038 | 0.4905 |
| 15 | 10.3333 | 0.0383 | 0.4984 | 16 | 4.6667 | 0.0384 | 0.5119 |
| 18 | 4.3333 | 0.0388 | 0.5399 | 19 | 4.0 | 0.0405 | 0.5523 |
| 21 | 16.0 | 0.0384 | 0.5819 | 24 | 4.6667 | 0.04 | 0.6207 |
| 26 | 10.6667 | 0.04 | 0.6402 | 29 | 5.0 | 0.0376 | 0.6931 |
| 31 | 9.3333 | 0.0386 | 0.787 | 35 | 10.3333 | 0.0413 | 0.9611 |
| 38 | 11.0 | 0.045 | 0.8429 | 42 | 11.3333 | 0.0401 | 1.1198 |
| 46 | 5.0 | 0.0393 | 0.9375 | 51 | 11.0 | 0.0399 | 0.9527 |
| 56 | 5.3333 | 0.0385 | 1.2791 | 61 | 5.6667 | 0.0442 | 1.1288 |
| 67 | 12.0 | 0.0392 | 1.431 | 74 | 18.0 | 0.0391 | 1.252 |
| 81 | 5.3333 | 0.0397 | 1.5781 | 90 | 6.6667 | 0.0387 | 1.4492 |
| 98 | 6.3333 | 0.0393 | 1.5297 | 108 | 10.6667 | 0.0392 | 1.6557 |
| 119 | 20.3333 | 0.0395 | 2.1394 | 131 | 21.0 | 0.0395 | 2.0339 |
| 144 | 20.6667 | 0.0387 | 2.2328 | 159 | 20.6667 | 0.0401 | 2.4066 |
| 174 | 33.6667 | 0.0393 | 2.7645 | 192 | 22.6667 | 0.0394 | 2.9577 |
| 211 | 22.6667 | 0.0399 | 3.008 | 232 | 29.0 | 0.0397 | 3.5782 |
| 255 | 30.6667 | 0.0401 | 3.5893 | 281 | 24.0 | 0.0386 | 4.0541 |
| 309 | 35.0 | 0.0397 | 4.4534 | 340 | 31.3333 | 0.0394 | 4.6686 |
| 374 | 36.6667 | 0.0395 | 5.1573 | 411 | 42.6667 | 0.0402 | 5.6482 |
| 453 | 33.6667 | 0.0395 | 6.1465 | 498 | 34.6667 | 0.0404 | 6.7275 |
| 548 | 42.6667 | 0.0405 | 7.5715 | 602 | 44.0 | 0.0396 | 8.3159 |
| 663 | 39.6667 | 0.0412 | 9.1818 | 729 | 34.3333 | 0.0403 | 10.0214 |
| 802 | 62.3333 | 0.0399 | 11.2392 | 882 | 65.0 | 0.0402 | 12.4735 |
| 970 | 67.3333 | 0.0397 | 13.6083 | 1067 | 63.6667 | 0.0401 | 15.0695 |
| 1174 | 71.6667 | 0.0411 | 16.793 | 1291 | 64.6667 | 0.0397 | 18.6851 |

| 1420 | 86.0 | 0.0402 | 20.6082 | 1562 | 72.0 | 0.0399 | 23.4612 |
|---|---|---|---|---|---|---|---|
| 1719 | 104.0 | 0.0411 | 25.9226 | 1891 | 124.3333 | 0.0396 | 29.0787 |
| 2080 | 99.0 | 0.0401 | 32.2493 | 2288 | 113.3333 | 0.0406 | 36.1112 |
| 2516 | 97.0 | 0.0399 | 41.3022 | 2768 | 109.0 | 0.041 | 46.6288 |
| 3045 | 110.6667 | 0.0417 | 52.2981 | 3349 | 131.3333 | 0.0408 | 60.4006 |
| 3684 | 109.0 | 0.0404 | 68.7662 | 4053 | 150.6667 | 0.0406 | 78.7969 |
| 4458 | 158.0 | 0.0402 | 89.2243 | 4904 | 171.6667 | 0.0403 | 100.4285 |
| 5394 | 173.0 | 0.0404 | 113.382 | 5933 | 192.3333 | 0.0406 | 125.8351 |
| 6527 | 206.3333 | 0.0409 | 139.4262 | 7180 | 234.3333 | 0.0406 | 155.8754 |
| 7897 | 256.6667 | 0.0407 | 171.647 | 8687 | 248.3333 | 0.0409 | 190.3503 |
| 9556 | 283.0 | 0.0403 | 210.5244 | 10512 | 309.3333 | 0.0405 | 232.3548 |
| 11563 | 316.3333 | 0.0399 | 256.7557 | 12719 | 333.3333 | 0.0413 | 282.2922 |
| 13991 | 370.6667 | 0.0397 | 310.671 | 15390 | 417.0 | 0.0399 | 342.5056 |
| 16929 | 449.0 | 0.04 | 377.6489 | 18622 | 482.3333 | 0.0407 | 415.5129 |
| 20484 | 516.3333 | 0.0413 | 458.381 | 22532 | 544.6667 | 0.0408 | 503.3267 |
| 24786 | 625.6667 | 0.04 | 555.7166 | 27264 | 674.6667 | 0.0411 | 611.2315 |
| 29991 | 732.3333 | 0.0406 | 673.6673 | | | | |
| C++ discrete | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 10.6667 | 0.0306 | 0.0904 | 11 | 10.0 | 0.0319 | 0.0933 |
| 12 | 10.6667 | 0.032 | 0.1 | 13 | 4.0 | 0.0357 | 0.1059 |
| 15 | 3.6667 | 0.0346 | 0.1471 | 16 | 3.3333 | 0.0352 | 0.1557 |
| 18 | 3.3333 | 0.0358 | 0.1644 | 19 | 3.6667 | 0.0361 | 0.1706 |
| 21 | 3.3333 | 0.0366 | 0.1796 | 24 | 3.3333 | 0.0378 | 0.1942 |
| 26 | 3.3333 | 0.0391 | 0.2055 | 29 | 3.3333 | 0.039 | 0.2193 |
| 31 | 3.6667 | 0.0424 | 0.2418 | 35 | 9.0 | 0.0425 | 0.2559 |
| 38 | 3.6667 | 0.0432 | 0.276 | 42 | 7.6667 | 0.0437 | 0.2992 |
| 46 | 10.6667 | 0.0434 | 0.3099 | 51 | 3.3333 | 0.0441 | 0.3454 |
| 56 | 3.6667 | 0.0451 | 0.3691 | 61 | 8.6667 | 0.045 | 0.3942 |
| 67 | 9.3333 | 0.0456 | 0.4224 | 74 | 3.6667 | 0.0472 | 0.4525 |
| 81 | 4.0 | 0.0477 | 0.4888 | 90 | 3.3333 | 0.0483 | 0.5365 |
| 98 | 9.3333 | 0.0489 | 0.5772 | 108 | 4.3333 | 0.0499 | 0.6295 |
| 119 | 9.0 | 0.0512 | 0.688 | 131 | 11.3333 | 0.0516 | 0.7519 |
| 144 | 10.3333 | 0.0525 | 0.8214 | 159 | 11.0 | 0.0531 | 0.899 |
| 174 | 5.3333 | 0.0538 | 0.974 | 192 | 5.0 | 0.0556 | 1.0734 |
| 211 | 18.0 | 0.0568 | 1.1559 | 232 | 10.6667 | 0.0582 | 1.304 |
| 255 | 16.0 | 0.0617 | 1.421 | 281 | 11.3333 | 0.0618 | 1.6677 |
| 309 | 10.3333 | 0.0608 | 1.7598 | 340 | 5.3333 | 0.0614 | 1.8797 |
| 374 | 5.3333 | 0.0627 | 2.074 | 411 | 5.6667 | 0.0667 | 2.2317 |
| 453 | 18.3333 | 0.0665 | 2.4824 | 498 | 16.6667 | 0.0666 | 2.6876 |
| 548 | 23.3333 | 0.0679 | 2.9921 | 602 | 24.6667 | 0.0677 | 3.3105 |
| 663 | 20.0 | 0.0685 | 3.5344 | 729 | 24.6667 | 0.0696 | 3.9102 |

| 802 | 26.6667 | 0.0706 | 4.2571 | 882 | 32.3333 | 0.072 | 4.7628 |
|---|---|---|---|---|---|---|---|
| 970 | 22.6667 | 0.0729 | 5.2215 | 1067 | 26.6667 | 0.0737 | 5.8836 |
| 1174 | 29.6667 | 0.0743 | 6.3998 | 1291 | 22.0 | 0.0755 | 7.1025 |
| 1420 | 30.3333 | 0.0751 | 7.903 | 1562 | 24.3333 | 0.0765 | 8.6302 |
| 1719 | 26.3333 | 0.0775 | 9.4466 | 1891 | 22.3333 | 0.0778 | 10.401 |
| 2080 | 40.6667 | 0.0794 | 11.524 | 2288 | 41.0 | 0.0794 | 12.6318 |
| 2516 | 36.3333 | 0.0798 | 13.9665 | 2768 | 39.3333 | 0.08 | 15.2874 |
| 3045 | 59.0 | 0.0813 | 16.9871 | 3349 | 68.0 | 0.0822 | 18.6043 |
| 3684 | 69.0 | 0.0837 | 20.4679 | 4053 | 74.0 | 0.0845 | 22.5784 |
| 4458 | 71.6667 | 0.0856 | 24.5494 | 4904 | 92.6667 | 0.0862 | 27.3079 |
| 5394 | 89.6667 | 0.0877 | 29.7232 | 5933 | 105.0 | 0.0883 | 33.1412 |
| 6527 | 60.6667 | 0.0897 | 35.9021 | 7180 | 93.3333 | 0.0908 | 39.9829 |
| 7897 | 104.6667 | 0.091 | 44.1926 | 8687 | 151.6667 | 0.0921 | 48.5042 |
| 9556 | 275.3333 | 0.0924 | 54.3388 | 10512 | 171.0 | 0.0938 | 59.395 |
| 11563 | 174.0 | 0.095 | 65.1993 | 12719 | 138.0 | 0.0958 | 71.5595 |
| 13991 | 142.0 | 0.0959 | 79.299 | 15390 | 127.3333 | 0.0966 | 87.4697 |
| 16929 | 265.0 | 0.0981 | 96.861 | 18622 | 280.6667 | 0.0994 | 106.4459 |
| 20484 | 321.0 | 0.1016 | 118.334 | 22532 | 326.3333 | 0.1011 | 130.6239 |
| 24786 | 363.0 | 0.1043 | 141.9038 | 27264 | 419.3333 | 0.1033 | 157.4953 |
| 29991 | 467.0 | 0.1055 | 172.5918 | | | | |
| Linear Search | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 8.0 | 0.0311 | 0.0032 | 11 | 3.0 | 0.0321 | 0.005 |
| 12 | 2.0 | 0.0326 | 0.0027 | 13 | 2.3333 | 0.0322 | 0.0033 |
| 15 | 6.6667 | 0.0341 | 0.0027 | 16 | 2.0 | 0.0383 | 0.0028 |
| 18 | 7.6667 | 0.0403 | 0.0026 | 19 | 2.3333 | 0.0401 | 0.0033 |
| 21 | 7.3333 | 0.0412 | 0.0029 | 24 | 8.3333 | 0.0439 | 0.0014 |
| 26 | 2.0 | 0.0425 | 0.0021 | 29 | 2.3333 | 0.0423 | 0.0037 |
| 31 | 7.3333 | 0.0439 | 0.0037 | 35 | 2.3333 | 0.0486 | 0.002 |
| 38 | 2.0 | 0.0498 | 0.003 | 42 | 2.0 | 0.0527 | 0.0019 |
| 46 | 7.3333 | 0.0564 | 0.0011 | 51 | 2.6667 | 0.0596 | 0.0023 |
| 56 | 2.3333 | 0.0646 | 0.0017 | 61 | 2.0 | 0.0647 | 0.0013 |
| 67 | 7.3333 | 0.068 | 0.0018 | 74 | 2.3333 | 0.0736 | 0.0003 |
| 81 | 2.3333 | 0.0757 | 0.002 | 90 | 2.0 | 0.0817 | 0.0014 |
| 98 | 2.3333 | 0.0867 | 0.001 | 108 | 8.0 | 0.0928 | 0.001 |
| 119 | 2.3333 | 0.0979 | 0.0024 | 131 | 7.6667 | 0.1052 | 0.0022 |
| 144 | 2.3333 | 0.1141 | 0.0006 | 159 | 6.6667 | 0.1203 | 0.0031 |
| 174 | 2.3333 | 0.1294 | 0.0029 | 192 | 2.3333 | 0.1413 | 0.0015 |
| 211 | 2.3333 | 0.1528 | 0.0011 | 232 | 2.6667 | 0.1633 | 0.0029 |
| 255 | 2.3333 | 0.1781 | 0.0014 | 281 | 3.0 | 0.1942 | 0.0021 |
| 309 | 3.6667 | 0.2062 | 0.0042 | 340 | 3.3333 | 0.2259 | 0.0031 |
| 374 | 7.0 | 0.2504 | -0.0014 | 411 | 8.0 | 0.2649 | 0.0053 |

| 453 | 3.6667 | 0.2912 | 0.0034 | 498 | 14.3333 | 0.3201 | 0.001 |
|---|---|---|---|---|---|---|---|
| 548 | 10.0 | 0.3488 | 0.0014 | 602 | 12.0 | 0.3791 | 0.0025 |
| 663 | 17.6667 | 0.4123 | 0.0046 | 729 | 16.6667 | 0.4502 | 0.0061 |
| 802 | 4.0 | 0.4938 | 0.0044 | 882 | 8.3333 | 0.5397 | 0.0053 |
| 970 | 10.3333 | 0.591 | 0.0065 | 1067 | 22.0 | 0.6497 | 0.0076 |
| 1174 | 17.6667 | 0.7115 | 0.0038 | 1291 | 17.6667 | 0.7814 | 0.0021 |
| 1420 | 17.6667 | 0.8577 | 0.0011 | 1562 | 17.6667 | 0.9345 | 0.0087 |
| 1719 | 7.0 | 1.0354 | -0.0021 | 1891 | 6.0 | 1.1331 | -0.001 |
| 2080 | 6.0 | 1.2474 | -0.0044 | 2288 | 12.3333 | 1.3626 | 0.0028 |
| 2516 | 6.6667 | 1.4896 | 0.008 | 2768 | 9.0 | 1.6539 | -0.0075 |
| 3045 | 7.6667 | 1.8142 | -0.0106 | 3349 | 9.0 | 1.97 | 0.0128 |
| 3684 | 15.3333 | 2.1813 | 0.0014 | 4053 | 26.6667 | 2.3942 | 0.0235 |
| 4458 | 14.3333 | 2.6211 | 0.0099 | 4904 | 26.0 | 2.8865 | 0.0075 |
| 5394 | 16.0 | 3.181 | 0.0125 | 5933 | 12.6667 | 3.5069 | -0.0131 |
| 6527 | 12.6667 | 3.8446 | -0.0014 | 7180 | 14.0 | 4.2285 | -0.0082 |
| 7897 | 20.3333 | 4.6382 | -0.0034 | 8687 | 19.6667 | 5.1097 | -0.0093 |
| 9556 | 31.3333 | 5.6511 | -0.0291 | 10512 | 22.6667 | 6.18 | -0.0165 |
| 11563 | 20.0 | 6.797 | 0.017 | 12719 | 21.0 | 7.4371 | 0.0096 |
| 13991 | 28.6667 | 8.1733 | 0.0167 | 15390 | 24.3333 | 9.0222 | 0.0098 |
| 16929 | 39.6667 | 9.8883 | -0.0074 | 18622 | 45.3333 | 10.8748 | 0.0681 |
| 20484 | 30.6666 | 11.9862 | -0.0039 | 22532 | 33.6666 | 13.2087 | -0.0154 |
| 24786 | 39.3334 | 14.5315 | 0.0117 | 27264 | 40.3333 | 15.9592 | 0.0244 |
| 29991 | 44.6667 | 17.5518 | 0.0582 | | | | |
| Binary Search | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 3.0 | 0.0326 | 0.022 | 11 | 3.0 | 0.0342 | 0.0253 |
| 12 | 2.6667 | 0.0331 | 0.0301 | 13 | 2.6667 | 0.0354 | 0.0308 |
| 15 | 7.0 | 0.0378 | 0.0359 | 16 | 2.3333 | 0.0361 | 0.0408 |
| 18 | 2.6667 | 0.0368 | 0.0457 | 19 | 2.6667 | 0.0378 | 0.047 |
| 21 | 2.3333 | 0.0393 | 0.0514 | 24 | 7.6667 | 0.0398 | 0.0601 |
| 26 | 8.6667 | 0.0402 | 0.066 | 29 | 8.6667 | 0.0417 | 0.0737 |
| 31 | 3.0 | 0.0416 | 0.0802 | 35 | 6.6667 | 0.0428 | 0.0962 |
| 38 | 2.6667 | 0.0435 | 0.1004 | 42 | 2.3333 | 0.045 | 0.1099 |
| 46 | 2.3333 | 0.0448 | 0.1187 | 51 | 7.6667 | 0.0454 | 0.1306 |
| 56 | 2.6667 | 0.0463 | 0.143 | 61 | 3.0 | 0.0471 | 0.1621 |
| 67 | 2.6667 | 0.0467 | 0.1734 | 74 | 2.3333 | 0.0481 | 0.191 |
| 81 | 8.6667 | 0.049 | 0.2086 | 90 | 3.3333 | 0.0498 | 0.2318 |
| 98 | 3.0 | 0.0511 | 0.2541 | 108 | 3.0 | 0.0518 | 0.2796 |
| 119 | 4.0 | 0.0528 | 0.3073 | 131 | 4.0 | 0.0536 | 0.3371 |
| 144 | 4.0 | 0.0542 | 0.3715 | 159 | 4.0 | 0.0556 | 0.4102 |
| 174 | 4.0 | 0.0561 | 0.4484 | 192 | 3.6667 | 0.0574 | 0.4961 |
| 211 | 11.0 | 0.0589 | 0.5629 | 232 | 9.3333 | 0.0599 | 0.5987 |

| 255 | 9.3333 | 0.0615 | 0.6576 | 281 | 3.3333 | 0.0619 | 0.7262 |
|---|---|---|---|---|---|---|---|
| 309 | 3.3333 | 0.0619 | 0.7986 | 340 | 3.6667 | 0.063 | 0.8777 |
| 374 | 16.0 | 0.0647 | 1.0052 | 411 | 8.0 | 0.0692 | 1.0717 |
| 453 | 10.0 | 0.067 | 1.1683 | 498 | 10.3333 | 0.0685 | 1.2831 |
| 548 | 9.3333 | 0.0693 | 1.414 | 602 | 4.3333 | 0.07 | 1.556 |
| 663 | 10.6667 | 0.0704 | 1.7108 | 729 | 5.0 | 0.0713 | 1.8815 |
| 802 | 23.6667 | 0.072 | 2.0707 | 882 | 23.3333 | 0.073 | 2.3191 |
| 970 | 18.0 | 0.0745 | 2.5114 | 1067 | 16.3333 | 0.0751 | 2.7562 |
| 1174 | 10.6667 | 0.0745 | 3.0861 | 1291 | 14.0 | 0.0762 | 3.3641 |
| 1420 | 13.6667 | 0.0772 | 3.6799 | 1562 | 26.3333 | 0.0778 | 4.0358 |
| 1719 | 19.6667 | 0.0781 | 4.4469 | 1891 | 15.0 | 0.08 | 4.8934 |
| 2080 | 23.0 | 0.08 | 5.3911 | 2288 | 21.3333 | 0.0804 | 5.9817 |
| 2516 | 30.0 | 0.0809 | 6.5715 | 2768 | 22.0 | 0.0817 | 7.2669 |
| 3045 | 18.0 | 0.0834 | 7.9814 | 3349 | 26.3333 | 0.0846 | 8.7572 |
| 3684 | 27.6667 | 0.0852 | 9.6298 | 4053 | 33.0 | 0.0851 | 10.6159 |
| 4458 | 33.3333 | 0.0863 | 11.7553 | 4904 | 25.0 | 0.0869 | 12.8722 |
| 5394 | 37.3333 | 0.0877 | 14.1663 | 5933 | 37.0 | 0.0885 | 15.6213 |
| 6527 | 30.0 | 0.0897 | 17.112 | 7180 | 35.3333 | 0.0914 | 18.8577 |
| 7897 | 29.3333 | 0.0916 | 20.724 | 8687 | 47.0 | 0.0931 | 22.8142 |
| 9556 | 56.3333 | 0.0937 | 25.0116 | 10512 | 52.0 | 0.0944 | 27.561 |
| 11563 | 59.3333 | 0.0959 | 30.2711 | 12719 | 68.6667 | 0.0965 | 33.3798 |
| 13991 | 60.3333 | 0.0969 | 36.6973 | 15390 | 58.0 | 0.0981 | 40.3265 |
| 16929 | 119.3333 | 0.0991 | 44.4831 | 18622 | 89.6667 | 0.1002 | 48.9436 |
| 20484 | 102.0 | 0.1016 | 53.9344 | 22532 | 122.6667 | 0.1026 | 59.1902 |
| 24786 | 115.3333 | 0.1036 | 65.0318 | 27264 | 126.6667 | 0.1046 | 71.876 |
| 29991 | 168.6667 | 0.1066 | 78.7928 | | | | |
| Two Level Search | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 3.0 | 0.0317 | 0.0061 | 11 | 3.3333 | 0.0319 | 0.0078 |
| 12 | 2.3333 | 0.0335 | 0.0081 | 13 | 3.0 | 0.033 | 0.007 |
| 15 | 2.3333 | 0.034 | 0.0072 | 16 | 10.3333 | 0.0351 | 0.0067 |
| 18 | 2.6667 | 0.0359 | 0.0062 | 19 | 3.0 | 0.0362 | 0.0063 |
| 21 | 9.0 | 0.0365 | 0.0064 | 24 | 2.6667 | 0.0374 | 0.0066 |
| 26 | 2.3333 | 0.0384 | 0.0059 | 29 | 9.0 | 0.0391 | 0.0062 |
| 31 | 8.3333 | 0.039 | 0.0061 | 35 | 2.6667 | 0.0406 | 0.0068 |
| 38 | 2.6667 | 0.0417 | 0.006 | 42 | 2.3333 | 0.0414 | 0.0057 |
| 46 | 3.0 | 0.042 | 0.0055 | 51 | 8.6667 | 0.0419 | 0.0056 |
| 56 | 8.6667 | 0.0427 | 0.0056 | 61 | 3.0 | 0.0434 | 0.0055 |
| 67 | 7.0 | 0.0434 | 0.0064 | 74 | 7.3333 | 0.045 | 0.0053 |
| 81 | 9.3333 | 0.0459 | 0.0049 | 90 | 3.0 | 0.0461 | 0.0058 |
| 98 | 3.3333 | 0.0471 | 0.0057 | 108 | 14.6667 | 0.0479 | 0.0053 |
| 119 | 10.0 | 0.0482 | 0.0059 | 131 | 4.0 | 0.049 | 0.0049 |

| 144 | 10.0 | 0.0499 | 0.0054 | 159 | 10.0 | 0.0507 | 0.0051 |
|---|---|---|---|---|---|---|---|
| 174 | 4.0 | 0.0513 | 0.0058 | 192 | 4.0 | 0.0518 | 0.0058 |
| 211 | 4.0 | 0.0533 | 0.0052 | 232 | 9.6667 | 0.0547 | 0.0046 |
| 255 | 13.6667 | 0.0554 | 0.0048 | 281 | 10.0 | 0.0562 | 0.0047 |
| 309 | 3.3333 | 0.0575 | 0.0048 | 340 | 8.6667 | 0.0582 | 0.0053 |
| 374 | 3.6667 | 0.0609 | 0.0043 | 411 | 3.3333 | 0.0646 | 0.0079 |
| 453 | 4.0 | 0.0626 | 0.0047 | 498 | 4.0 | 0.0638 | 0.0048 |
| 548 | 16.6667 | 0.0656 | 0.0042 | 602 | 15.3333 | 0.0664 | 0.0055 |
| 663 | 4.3333 | 0.0679 | 0.0056 | 729 | 5.0 | 0.0701 | 0.0042 |
| 802 | 16.3333 | 0.0705 | 0.0058 | 882 | 17.3333 | 0.0728 | 0.0055 |
| 970 | 5.3333 | 0.0752 | 0.0043 | 1067 | 18.6667 | 0.077 | 0.0048 |
| 1174 | 12.6667 | 0.0793 | 0.0051 | 1291 | 24.0 | 0.081 | 0.0049 |
| 1420 | 24.6667 | 0.0825 | 0.0057 | 1562 | 23.6667 | 0.0852 | 0.0052 |
| 1719 | 18.3333 | 0.0877 | 0.0054 | 1891 | 14.3333 | 0.0901 | 0.0052 |
| 2080 | 20.0 | 0.0919 | 0.0051 | 2288 | 15.3333 | 0.0948 | 0.0054 |
| 2516 | 21.3333 | 0.0978 | 0.0041 | 2768 | 15.6667 | 0.1008 | 0.0045 |
| 3045 | 17.6667 | 0.1039 | 0.0042 | 3349 | 18.3333 | 0.1069 | 0.0048 |
| 3684 | 25.0 | 0.1092 | 0.006 | 4053 | 24.6667 | 0.1139 | 0.0052 |
| 4458 | 31.6667 | 0.1176 | 0.0047 | 4904 | 44.3333 | 0.1211 | 0.0046 |
| 5394 | 22.3333 | 0.1251 | 0.0051 | 5933 | 31.6667 | 0.1312 | 0.0037 |
| 6527 | 37.0 | 0.1343 | 0.0042 | 7180 | 38.6667 | 0.1389 | 0.0059 |
| 7897 | 35.6667 | 0.1431 | 0.0049 | 8687 | 33.0 | 0.1485 | 0.0045 |
| 9556 | 42.6667 | 0.1545 | 0.0042 | 10512 | 45.0 | 0.1602 | 0.0061 |
| 11563 | 43.0 | 0.1655 | 0.0048 | 12719 | 42.3333 | 0.1721 | 0.0034 |
| 13991 | 46.0 | 0.1785 | 0.0038 | 15390 | 56.0 | 0.1858 | 0.0032 |
| 16929 | 64.3333 | 0.1924 | 0.0032 | 18622 | 67.3333 | 0.203 | 0.0033 |
| 20484 | 60.3333 | 0.2159 | -0.0038 | 22532 | 65.0 | 0.2207 | -0.0004 |
| 24786 | 77.3333 | 0.2254 | 0.0029 | 27264 | 87.0 | 0.2336 | 0.0045 |
| 29991 | 95.6667 | 0.2449 | 0.0024 | | | | |
| Three Level Search | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 8.6667 | 0.0295 | 0.0097 | 11 | 3.6667 | 0.0313 | 0.0088 |
| 12 | 3.3333 | 0.0337 | 0.0083 | 13 | 3.3333 | 0.0326 | 0.0091 |
| 15 | 3.0 | 0.0342 | 0.0112 | 16 | 9.0 | 0.0359 | 0.0105 |
| 18 | 3.0 | 0.0371 | 0.0122 | 19 | 3.0 | 0.0361 | 0.0123 |
| 21 | 10.0 | 0.0372 | 0.0122 | 24 | 8.3333 | 0.0396 | 0.0111 |
| 26 | 8.3333 | 0.0393 | 0.0121 | 29 | 3.0 | 0.0402 | 0.0097 |
| 31 | 3.0 | 0.0404 | 0.0105 | 35 | 15.3333 | 0.0418 | 0.0106 |
| 38 | 3.0 | 0.0422 | 0.012 | 42 | 9.3333 | 0.0422 | 0.0117 |
| 46 | 11.0 | 0.043 | 0.0119 | 51 | 3.3333 | 0.0428 | 0.0116 |
| 56 | 3.0 | 0.0437 | 0.012 | 61 | 3.0 | 0.0446 | 0.0114 |
| 67 | 3.0 | 0.0448 | 0.0117 | 74 | 8.3333 | 0.0456 | 0.0122 |

| 81 | 8.3333 | 0.0461 | 0.0112 | 90 | 9.6667 | 0.0472 | 0.0115 |
|---|---|---|---|---|---|---|---|
| 98 | 10.6667 | 0.0475 | 0.0118 | 108 | 15.3333 | 0.0474 | 0.0122 |
| 119 | 4.0 | 0.0481 | 0.0121 | 131 | 10.3333 | 0.0483 | 0.0138 |
| 144 | 4.0 | 0.049 | 0.0122 | 159 | 4.6667 | 0.0502 | 0.0113 |
| 174 | 10.3333 | 0.0508 | 0.0112 | 192 | 4.3333 | 0.051 | 0.0118 |
| 211 | 4.3333 | 0.0518 | 0.0116 | 232 | 3.3333 | 0.0525 | 0.0116 |
| 255 | 9.3333 | 0.0533 | 0.0112 | 281 | 9.3333 | 0.0533 | 0.0118 |
| 309 | 3.6667 | 0.054 | 0.0116 | 340 | 3.6667 | 0.0546 | 0.0116 |
| 374 | 8.6667 | 0.056 | 0.0114 | 411 | 4.3333 | 0.0611 | 0.0129 |
| 453 | 4.3333 | 0.057 | 0.0109 | 498 | 4.3333 | 0.0586 | 0.0101 |
| 548 | 10.6667 | 0.0587 | 0.0114 | 602 | 4.3333 | 0.0594 | 0.011 |
| 663 | 17.0 | 0.0592 | 0.0113 | 729 | 5.0 | 0.0613 | 0.0095 |
| 802 | 16.6667 | 0.0612 | 0.0107 | 882 | 17.3333 | 0.0617 | 0.0121 |
| 970 | 17.6667 | 0.0626 | 0.0105 | 1067 | 18.0 | 0.0637 | 0.01 |
| 1174 | 18.3333 | 0.0645 | 0.0102 | 1291 | 24.0 | 0.0646 | 0.0107 |
| 1420 | 19.0 | 0.0653 | 0.0113 | 1562 | 25.0 | 0.0661 | 0.0099 |
| 1719 | 25.0 | 0.0658 | 0.0115 | 1891 | 26.3333 | 0.0684 | 0.01 |
| 2080 | 15.3333 | 0.0679 | 0.0106 | 2288 | 37.6667 | 0.0694 | 0.0106 |
| 2516 | 15.0 | 0.0702 | 0.0102 | 2768 | 16.6667 | 0.0715 | 0.0106 |
| 3045 | 16.0 | 0.0729 | 0.01 | 3349 | 35.6667 | 0.0744 | 0.0097 |
| 3684 | 29.3333 | 0.0746 | 0.0105 | 4053 | 30.0 | 0.0754 | 0.0103 |
| 4458 | 21.6667 | 0.0768 | 0.01 | 4904 | 29.0 | 0.0778 | 0.0095 |
| 5394 | 23.0 | 0.0781 | 0.0097 | 5933 | 35.3333 | 0.0804 | 0.0095 |
| 6527 | 33.3333 | 0.0802 | 0.0097 | 7180 | 23.6667 | 0.0823 | 0.01 |
| 7897 | 31.0 | 0.0822 | 0.0159 | 8687 | 41.3333 | 0.0838 | 0.0095 |
| 9556 | 38.6667 | 0.0862 | 0.0099 | 10512 | 46.3333 | 0.0879 | 0.0101 |
| 11563 | 45.6667 | 0.0883 | 0.0092 | 12719 | 38.3333 | 0.0901 | 0.0094 |
| 13991 | 47.6667 | 0.0905 | 0.0098 | 15390 | 46.0 | 0.092 | 0.0097 |
| 16929 | 49.0 | 0.0943 | 0.0094 | 18622 | 57.3333 | 0.0965 | 0.0088 |
| 20484 | 68.0 | 0.0979 | 0.0099 | 22532 | 81.0 | 0.1004 | 0.008 |
| 24786 | 82.0 | 0.1011 | 0.0101 | 27264 | 87.3333 | 0.1024 | 0.01 |
| 29991 | 95.6667 | 0.1049 | 0.01 | | | | |
| Binary Tree Search | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 3.0 | 0.0329 | 0.0167 | 11 | 3.0 | 0.0324 | 0.0185 |
| 12 | 2.6667 | 0.0348 | 0.019 | 13 | 9.3333 | 0.0356 | 0.0175 |
| 15 | 3.0 | 0.0367 | 0.0183 | 16 | 2.3333 | 0.037 | 0.0195 |
| 18 | 2.3333 | 0.0376 | 0.0197 | 19 | 2.3333 | 0.0385 | 0.0202 |
| 21 | 8.3333 | 0.039 | 0.0199 | 24 | 2.3333 | 0.0398 | 0.0205 |
| 26 | 2.6667 | 0.0414 | 0.02 | 29 | 2.3333 | 0.0413 | 0.0221 |
| 31 | 7.6667 | 0.0438 | 0.0244 | 35 | 2.6667 | 0.0453 | 0.0242 |
| 38 | 2.6667 | 0.0452 | 0.0252 | 42 | 2.3333 | 0.0447 | 0.023 |

| 46 | 8.3333 | 0.0451 | 0.0245 | 51 | 7.3333 | 0.0457 | 0.0255 |
|---|---|---|---|---|---|---|---|
| 56 | 2.3333 | 0.0472 | 0.0269 | 61 | 3.0 | 0.0494 | 0.0279 |
| 67 | 2.3333 | 0.0495 | 0.0282 | 74 | 2.3333 | 0.0503 | 0.0272 |
| 81 | 3.0 | 0.0529 | 0.0256 | 90 | 2.6667 | 0.0515 | 0.0294 |
| 98 | 3.0 | 0.0525 | 0.0296 | 108 | 3.0 | 0.0543 | 0.0301 |
| 119 | 4.0 | 0.0554 | 0.0306 | 131 | 9.3333 | 0.0561 | 0.0327 |
| 144 | 15.6667 | 0.0578 | 0.0298 | 159 | 9.6667 | 0.0588 | 0.0343 |
| 174 | 9.0 | 0.058 | 0.0329 | 192 | 3.6667 | 0.0588 | 0.034 |
| 211 | 3.3333 | 0.0601 | 0.0347 | 232 | 7.0 | 0.0616 | 0.0357 |
| 255 | 8.6667 | 0.0632 | 0.0365 | 281 | 15.0 | 0.0637 | 0.0354 |
| 309 | 2.3333 | 0.0637 | 0.0363 | 340 | 3.0 | 0.0643 | 0.0364 |
| 374 | 8.6667 | 0.0661 | 0.037 | 411 | 2.6667 | 0.0701 | 0.037 |
| 453 | 3.0 | 0.0682 | 0.0374 | 498 | 3.0 | 0.0693 | 0.0388 |
| 548 | 9.0 | 0.0711 | 0.039 | 602 | 2.6667 | 0.0716 | 0.0393 |
| 663 | 3.0 | 0.0713 | 0.0398 | 729 | 7.6667 | 0.0723 | 0.0405 |
| 802 | 3.0 | 0.0722 | 0.0419 | 882 | 21.0 | 0.0744 | 0.0425 |
| 970 | 9.3333 | 0.0751 | 0.0423 | 1067 | 21.3333 | 0.0757 | 0.043 |
| 1174 | 16.0 | 0.0765 | 0.043 | 1291 | 9.3333 | 0.0768 | 0.0443 |
| 1420 | 21.3333 | 0.0782 | 0.0443 | 1562 | 16.0 | 0.0786 | 0.051 |
| 1719 | 9.6667 | 0.0794 | 0.0452 | 1891 | 17.3333 | 0.0803 | 0.0472 |
| 2080 | 10.6667 | 0.082 | 0.0461 | 2288 | 5.0 | 0.0828 | 0.0469 |
| 2516 | 5.3333 | 0.0826 | 0.049 | 2768 | 10.0 | 0.0841 | 0.0474 |
| 3045 | 10.3333 | 0.0848 | 0.0484 | 3349 | 6.0 | 0.0859 | 0.0497 |
| 3684 | 17.0 | 0.0873 | 0.0492 | 4053 | 7.0 | 0.0881 | 0.05 |
| 4458 | 13.3333 | 0.0888 | 0.0501 | 4904 | 18.0 | 0.0892 | 0.0501 |
| 5394 | 13.3333 | 0.0901 | 0.0547 | 5933 | 9.6667 | 0.0911 | 0.0567 |
| 6527 | 10.0 | 0.0924 | 0.0517 | 7180 | 21.0 | 0.0925 | 0.053 |
| 7897 | 13.6667 | 0.0988 | 0.0499 | 8687 | 23.3333 | 0.0939 | 0.0547 |
| 9556 | 24.6667 | 0.0956 | 0.0548 | 10512 | 19.3333 | 0.0965 | 0.0546 |
| 11563 | 17.0 | 0.0965 | 0.0559 | 12719 | 17.3333 | 0.0983 | 0.0596 |
| 13991 | 18.6667 | 0.0988 | 0.0561 | 15390 | 26.0 | 0.1002 | 0.0577 |
| 16929 | 22.6667 | 0.1023 | 0.0574 | 18622 | 23.3333 | 0.1035 | 0.0584 |
| 20484 | 25.6667 | 0.1054 | 0.0594 | 22532 | 39.6667 | 0.1045 | 0.0587 |
| 24786 | 43.3333 | 0.1071 | 0.0586 | 27264 | 34.3333 | 0.1081 | 0.0594 |
| 29991 | 40.6667 | 0.1107 | 0.0593 | | | | |
| Rejection Method | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 3.3333 | 0.0661 | 0.0152 | 11 | 3.0 | 0.0821 | -0.0009 |
| 12 | 2.3333 | 0.0795 | 0.0019 | 13 | 2.3333 | 0.0676 | 0.0137 |
| 15 | 2.3333 | 0.0749 | 0.0061 | 16 | 2.3333 | 0.077 | 0.0038 |
| 18 | 2.3333 | 0.0817 | -0.0004 | 19 | 8.3333 | 0.0705 | 0.0102 |
| 21 | 3.0 | 0.0779 | 0.0021 | 24 | 8.3333 | 0.0813 | -0.0012 |

| 26 | 2.6667 | 0.0757 | 0.0041 | 29 | 2.3333 | 0.0837 | -0.0037 |
|---|---|---|---|---|---|---|---|
| 31 | 2.6667 | 0.0888 | -0.0035 | 35 | 7.3333 | 0.0797 | 0.0018 |
| 38 | 2.3333 | 0.0746 | 0.0071 | 42 | 2.3333 | 0.0741 | 0.0054 |
| 46 | 2.6667 | 0.0776 | 0.002 | 51 | 9.0 | 0.0847 | -0.0043 |
| 56 | 2.6667 | 0.0781 | 0.0007 | 61 | 2.3333 | 0.0755 | 0.0036 |
| 67 | 7.3333 | 0.0773 | 0.0021 | 74 | 8.0 | 0.0776 | 0.0022 |
| 81 | 2.3333 | 0.074 | 0.0053 | 90 | 2.6667 | 0.0734 | 0.0054 |
| 98 | 7.3333 | 0.0793 | 0.0001 | 108 | 2.6667 | 0.08 | -0.0008 |
| 119 | 8.3333 | 0.0779 | 0.0011 | 131 | 3.6667 | 0.0752 | 0.0037 |
| 144 | 3.3333 | 0.0792 | 0.0003 | 159 | 9.6667 | 0.0764 | 0.0027 |
| 174 | 3.0 | 0.0806 | -0.0016 | 192 | 3.3333 | 0.0798 | -0.0011 |
| 211 | 3.6667 | 0.0779 | 0.001 | 232 | 9.0 | 0.0766 | 0.0023 |
| 255 | 9.0 | 0.0793 | -0.0004 | 281 | 7.3333 | 0.0801 | -0.0011 |
| 309 | 2.3333 | 0.0795 | -0.0007 | 340 | 9.0 | 0.0808 | -0.0017 |
| 374 | 3.0 | 0.0794 | -0.0005 | 411 | 2.6667 | 0.0815 | 0.0006 |
| 453 | 3.0 | 0.0784 | 0.0005 | 498 | 9.3333 | 0.0777 | 0.0014 |
| 548 | 3.0 | 0.078 | 0.0014 | 602 | 9.0 | 0.0775 | 0.0014 |
| 663 | 9.0 | 0.0791 | -0.0 | 729 | 3.0 | 0.0785 | 0.0003 |
| 802 | 10.3333 | 0.0805 | -0.0012 | 882 | 3.3333 | 0.0798 | 0.0002 |
| 970 | 9.6667 | 0.0786 | 0.0004 | 1067 | 9.6667 | 0.0798 | -0.0012 |
| 1174 | 10.0 | 0.08 | -0.0002 | 1291 | 10.0 | 0.0784 | 0.0003 |
| 1420 | 11.0 | 0.079 | 0.0 | 1562 | 4.0 | 0.0812 | -0.0022 |
| 1719 | 11.3333 | 0.0778 | 0.001 | 1891 | 12.3333 | 0.0801 | -0.0003 |
| 2080 | 11.6667 | 0.0802 | -0.0002 | 2288 | 16.6667 | 0.0794 | -0.0007 |
| 2516 | 5.0 | 0.0789 | 0.0025 | 2768 | 5.3333 | 0.0791 | 0.001 |
| 3045 | 15.6667 | 0.0786 | 0.0002 | 3349 | 13.0 | 0.0799 | -0.0001 |
| 3684 | 12.0 | 0.0795 | 0.0005 | 4053 | 16.6667 | 0.0786 | 0.0001 |
| 4458 | 13.0 | 0.0796 | -0.0009 | 4904 | 13.3333 | 0.0792 | 0.0014 |
| 5394 | 12.0 | 0.0818 | -0.0011 | 5933 | 8.0 | 0.0786 | 0.0001 |
| 6527 | 25.3333 | 0.0786 | 0.0003 | 7180 | 9.6667 | 0.0796 | 0.0007 |
| 7897 | 17.6667 | 0.0793 | 0.0007 | 8687 | 16.0 | 0.079 | -0.0002 |
| 9556 | 11.6667 | 0.0794 | 0.0006 | 10512 | 12.6667 | 0.0787 | 0.0 |
| 11563 | 13.3333 | 0.0784 | 0.0008 | 12719 | 24.3333 | 0.0792 | 0.0001 |
| 13991 | 15.3333 | 0.0792 | 0.0001 | 15390 | 22.3333 | 0.0794 | 0.0002 |
| 16929 | 22.0 | 0.0802 | 0.0011 | 18622 | 23.6667 | 0.0795 | 0.001 |
| 20484 | 25.0 | 0.0801 | 0.0002 | 22532 | 23.0 | 0.0795 | 0.0008 |
| 24786 | 25.0 | 0.0798 | 0.0013 | 27264 | 32.0 | 0.0795 | 0.0012 |
| 29991 | 30.6667 | 0.0822 | 0.0026 | | | | |
| Flat Method | | | | | | | |
| n | a | b | c | n | a | b | c |
| 10 | 13.3333 | 0.0821 | 0.0728 | 11 | 20.3333 | 0.0869 | 0.0672 |
| 12 | 12.3333 | 0.0905 | 0.0639 | 13 | 19.3333 | 0.085 | 0.0694 |

| 15 | 7.6667 | 0.0882 | 0.0657 | 16 | 21.0 | 0.0855 | 0.0674 |
|---|---|---|---|---|---|---|---|
| 18 | 7.6667 | 0.0861 | 0.0671 | 19 | 22.0 | 0.0926 | 0.0607 |
| 21 | 22.3333 | 0.0821 | 0.0705 | 24 | 20.3333 | 0.0906 | 0.0617 |
| 26 | 8.6667 | 0.0814 | 0.07 | 29 | 21.0 | 0.0847 | 0.0668 |
| 31 | 22.3333 | 0.0961 | 0.0601 | 35 | 20.3333 | 0.0904 | 0.0671 |
| 38 | 29.3333 | 0.0917 | 0.0627 | 42 | 14.6667 | 0.0868 | 0.064 |
| 46 | 22.0 | 0.0889 | 0.0622 | 51 | 21.6667 | 0.0838 | 0.0684 |
| 56 | 31.3333 | 0.0813 | 0.0701 | 61 | 23.6667 | 0.0875 | 0.0829 |
| 67 | 35.3333 | 0.0839 | 0.0661 | 74 | 24.6667 | 0.0852 | 0.0641 |
| 81 | 31.6667 | 0.0869 | 0.0613 | 90 | 33.0 | 0.0861 | 0.0625 |
| 98 | 32.3333 | 0.088 | 0.0604 | 108 | 38.6667 | 0.087 | 0.0615 |
| 119 | 46.6667 | 0.0893 | 0.0594 | 131 | 49.6667 | 0.086 | 0.0626 |
| 144 | 48.6667 | 0.0884 | 0.0602 | 159 | 53.0 | 0.0864 | 0.0616 |
| 174 | 46.3333 | 0.0863 | 0.0619 | 192 | 57.0 | 0.0887 | 0.0601 |
| 211 | 47.6667 | 0.0874 | 0.0617 | 232 | 75.3333 | 0.0884 | 0.0623 |
| 255 | 36.0 | 0.0873 | 0.0624 | 281 | 62.6667 | 0.0881 | 0.0594 |
| 309 | 56.0 | 0.0871 | 0.0604 | 340 | 65.6667 | 0.0871 | 0.0613 |
| 374 | 61.3333 | 0.0866 | 0.0624 | 411 | 109.6667 | 0.092 | 0.0589 |
| 453 | 77.3333 | 0.0882 | 0.0608 | 498 | 92.0 | 0.0871 | 0.0623 |
| 548 | 102.6667 | 0.0896 | 0.0593 | 602 | 70.3333 | 0.0897 | 0.059 |
| 663 | 93.3333 | 0.088 | 0.0605 | 729 | 125.6667 | 0.0905 | 0.0624 |
| 802 | 112.0 | 0.0879 | 0.0605 | 882 | 100.3333 | 0.0891 | 0.0621 |
| 970 | 121.0 | 0.0884 | 0.0607 | 1067 | 146.0 | 0.0886 | 0.063 |
| 1174 | 146.6667 | 0.0884 | 0.0628 | 1291 | 159.3333 | 0.0892 | 0.0597 |
| 1420 | 172.6667 | 0.0886 | 0.0609 | 1562 | 155.6667 | 0.0901 | 0.0591 |
| 1719 | 205.0 | 0.0886 | 0.0604 | 1891 | 200.3333 | 0.0896 | 0.0596 |
| 2080 | 234.6667 | 0.0894 | 0.0601 | 2288 | 228.3333 | 0.0888 | 0.0604 |
| 2516 | 218.3333 | 0.0913 | 0.0587 | 2768 | 255.0 | 0.0894 | 0.06 |
| 3045 | 306.3333 | 0.0884 | 0.0598 | 3349 | 287.0 | 0.0886 | 0.0606 |
| 3684 | 306.0 | 0.089 | 0.0612 | 4053 | 349.6667 | 0.0893 | 0.0607 |
| 4458 | 393.3333 | 0.0881 | 0.0618 | 4904 | 394.6667 | 0.0887 | 0.0615 |
| 5394 | 463.6667 | 0.0894 | 0.0611 | 5933 | 471.0 | 0.0904 | 0.065 |
| 6527 | 497.3333 | 0.0895 | 0.0626 | 7180 | 605.6667 | 0.0896 | 0.0607 |
| 7897 | 585.0 | 0.0902 | 0.0637 | 8687 | 646.3333 | 0.0896 | 0.0629 |
| 9556 | 745.3333 | 0.0921 | 0.0672 | 10512 | 774.6667 | 0.0895 | 0.0655 |
| 11563 | 838.3333 | 0.0905 | 0.063 | 12719 | 964.3333 | 0.0897 | 0.0636 |
| 13991 | 963.3333 | 0.0896 | 0.0647 | 15390 | 1108.0 | 0.09 | 0.0645 |
| 16929 | 1280.6667 | 0.09 | 0.0656 | 18622 | 1279.6667 | 0.0906 | 0.066 |
| 20484 | 1543.0 | 0.0908 | 0.0654 | 22532 | 1574.3333 | 0.0908 | 0.0665 |
| 24786 | 2310.0 | 0.1071 | 0.0759 | 27264 | 1889.0 | 0.0919 | 0.0679 |
| 29991 | 2021.3333 | 0.0925 | 0.067 | | | | |
| Log* Method | | | | | | | |

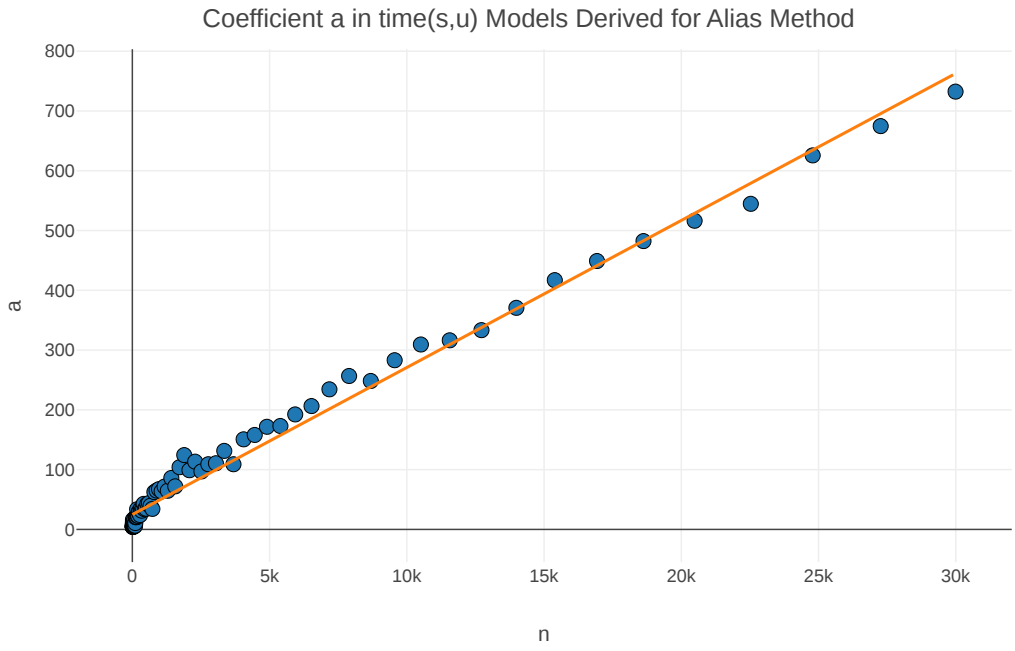
| n | a | b | c | n | a | b | c |
|---|---|---|---|---|---|---|---|
| 10 | 21.3333 | 0.1951 | 0.4009 | 11 | 22.6667 | 0.1718 | 0.4296 |
| 12 | 22.6667 | 0.2071 | 0.3991 | 13 | 32.3333 | 0.2125 | 0.4004 |
| 15 | 28.0 | 0.2092 | 0.4186 | 16 | 27.6667 | 0.1924 | 0.4418 |
| 18 | 23.6667 | 0.1894 | 0.452 | 19 | 28.0 | 0.1973 | 0.4469 |
| 21 | 35.3333 | 0.2049 | 0.4418 | 24 | 18.6667 | 0.2017 | 0.4459 |
| 26 | 24.6667 | 0.2085 | 0.4382 | 29 | 34.3333 | 0.1948 | 0.4507 |
| 31 | 25.0 | 0.2093 | 0.4623 | 35 | 33.0 | 0.2264 | 0.5015 |
| 38 | 30.3333 | 0.1986 | 0.5771 | 42 | 31.0 | 0.2154 | 0.4431 |
| 46 | 39.0 | 0.1966 | 0.4635 | 51 | 40.3333 | 0.2006 | 0.456 |
| 56 | 40.0 | 0.2009 | 0.4858 | 61 | 35.3333 | 0.2492 | 0.434 |
| 67 | 40.0 | 0.2109 | 0.4343 | 74 | 40.0 | 0.2113 | 0.4412 |
| 81 | 34.3333 | 0.2024 | 0.4562 | 90 | 43.3333 | 0.1942 | 0.4709 |
| 98 | 44.3333 | 0.194 | 0.4701 | 108 | 42.6667 | 0.2025 | 0.45 |
| 119 | 44.3333 | 0.2191 | 0.4205 | 131 | 46.6667 | 0.2196 | 0.4194 |
| 144 | 53.6667 | 0.2094 | 0.436 | 159 | 50.6667 | 0.2054 | 0.4868 |
| 174 | 54.0 | 0.1931 | 0.4743 | 192 | 56.0 | 0.2063 | 0.4645 |
| 211 | 66.6667 | 0.2057 | 0.4532 | 232 | 56.3333 | 0.2104 | 0.4279 |
| 255 | 42.0 | 0.2035 | 0.4289 | 281 | 59.0 | 0.2097 | 0.4223 |
| 309 | 54.3333 | 0.2049 | 0.4448 | 340 | 75.0 | 0.2071 | 0.46 |
| 374 | 80.6667 | 0.193 | 0.481 | 411 | 83.3333 | 0.2153 | 0.4868 |
| 453 | 79.3333 | 0.1956 | 0.4446 | 498 | 88.6667 | 0.215 | 0.4091 |
| 548 | 99.3333 | 0.2134 | 0.4081 | 602 | 87.6667 | 0.208 | 0.4273 |
| 663 | 94.3333 | 0.2035 | 0.4746 | 729 | 94.0 | 0.1941 | 0.5137 |
| 802 | 115.3333 | 0.2024 | 0.4884 | 882 | 117.6667 | 0.221 | 0.4169 |
| 970 | 108.0 | 0.2202 | 0.4068 | 1067 | 130.3333 | 0.2162 | 0.4305 |
| 1174 | 148.3333 | 0.2081 | 0.4163 | 1291 | 145.6667 | 0.2052 | 0.4557 |
| 1420 | 139.6667 | 0.206 | 0.474 | 1562 | 140.0 | 0.1901 | 0.4833 |
| 1719 | 175.0 | 0.2277 | 0.4123 | 1891 | 166.0 | 0.2219 | 0.4078 |
| 2080 | 153.6667 | 0.2149 | 0.409 | 2288 | 194.3333 | 0.2092 | 0.4154 |
| 2516 | 195.0 | 0.2051 | 0.4391 | 2768 | 216.0 | 0.2019 | 0.4821 |
| 3045 | 212.0 | 0.1922 | 0.4849 | 3349 | 222.0 | 0.2133 | 0.438 |
| 3684 | 242.0 | 0.223 | 0.4128 | 4053 | 257.0 | 0.2191 | 0.4078 |
| 4458 | 282.3333 | 0.2111 | 0.409 | 4904 | 303.0 | 0.2048 | 0.4184 |
| 5394 | 308.0 | 0.206 | 0.4835 | 5933 | 342.6667 | 0.203 | 0.4777 |
| 6527 | 386.0 | 0.1905 | 0.4821 | 7180 | 378.0 | 0.2309 | 0.4182 |
| 7897 | 449.6667 | 0.2198 | 0.4121 | 8687 | 464.0 | 0.2127 | 0.4141 |
| 9556 | 484.3333 | 0.2064 | 0.416 | 10512 | 521.6667 | 0.2149 | 0.4752 |
| 11563 | 591.6667 | 0.2058 | 0.482 | 12719 | 684.6667 | 0.1862 | 0.4918 |
| 13991 | 646.6667 | 0.2301 | 0.4306 | 15390 | 728.3333 | 0.2234 | 0.4193 |
| 16929 | 774.3333 | 0.2169 | 0.4218 | 18622 | 887.3333 | 0.2131 | 0.4234 |
| 20484 | 930.0 | 0.2074 | 0.4653 | 22532 | 1021.3333 | 0.2084 | 0.492 |

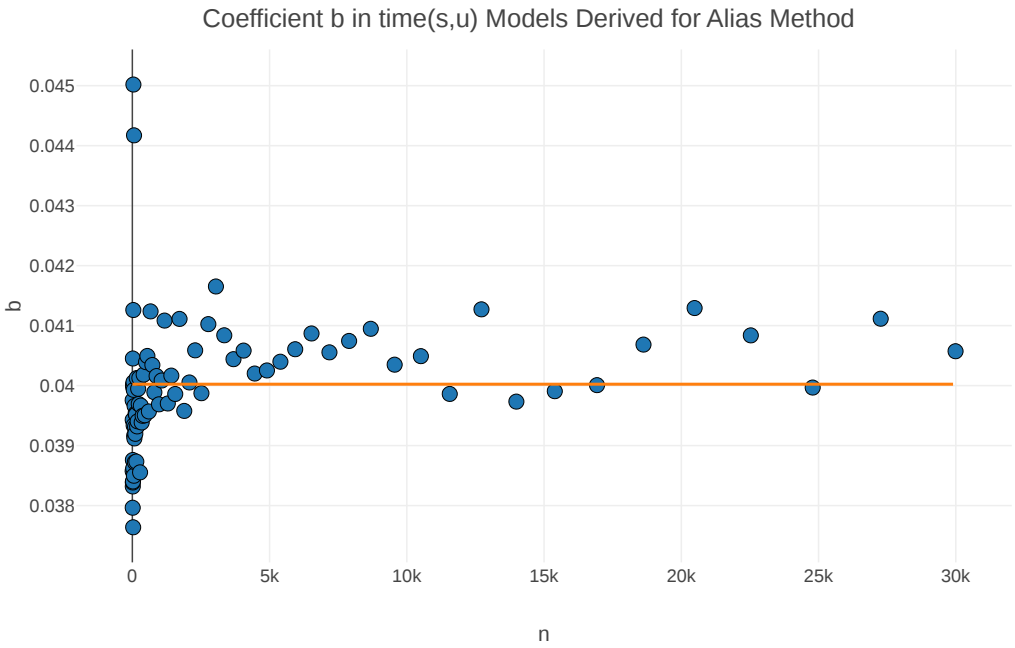
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 24786 | 1130.6667 | 0.192 | 0.4958 | 27264 | 1243.3333 | 0.2333 | 0.4312 |
| 29991 | 1376.3333 | 0.2283 | 0.4258 | | | | |

# Appendix B

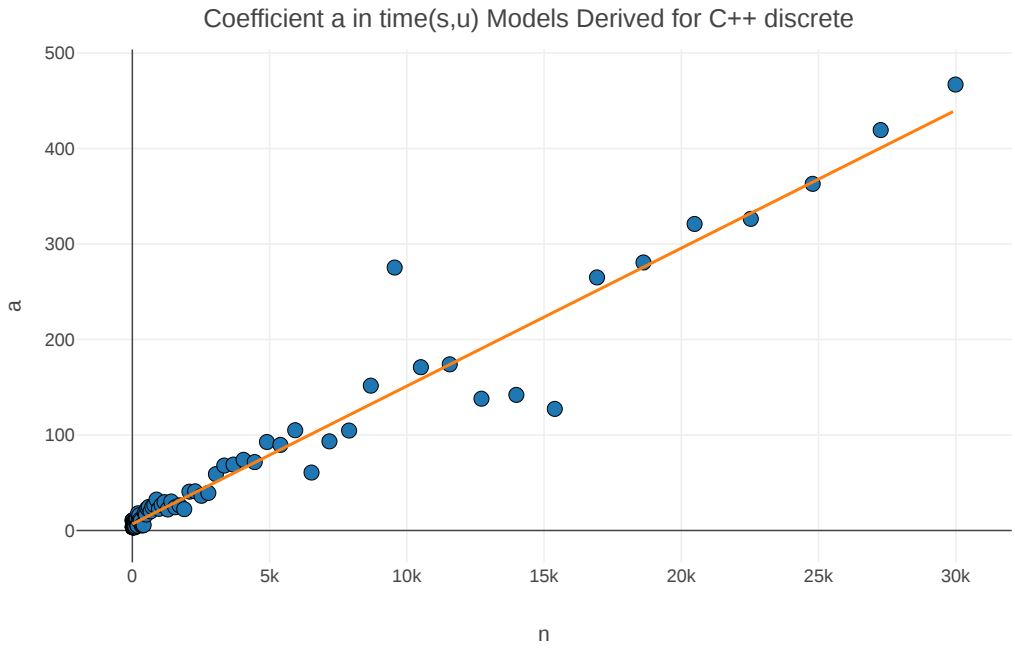
# Model Fitting

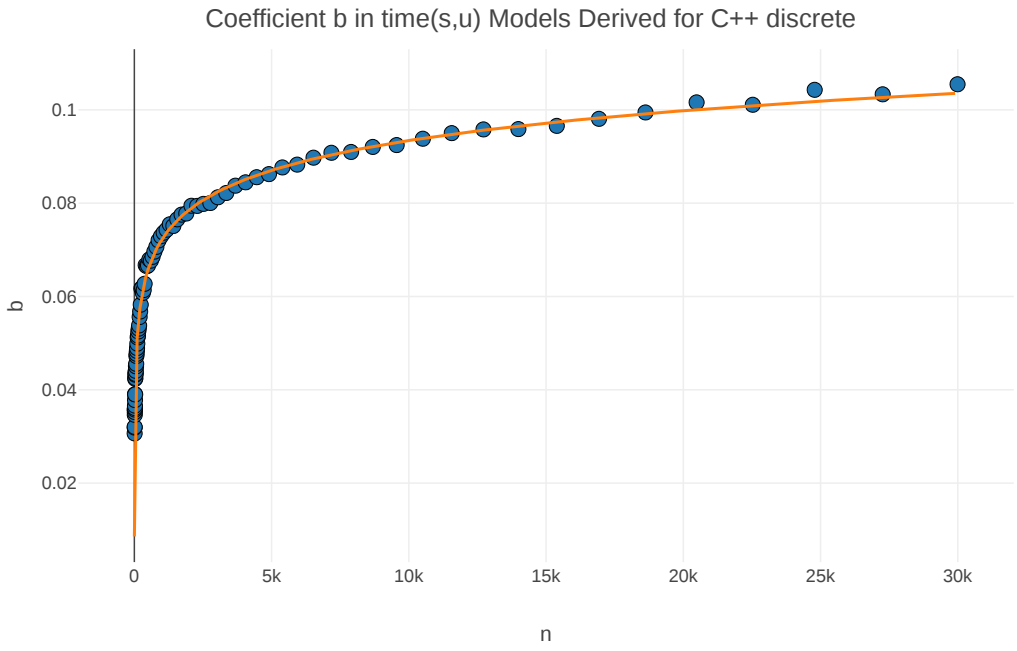
In each plot, we show 85 $a$s, $b$s or $c$s in $time(x, y)$ with a curve corresponding to the theoretical runtime complexity.

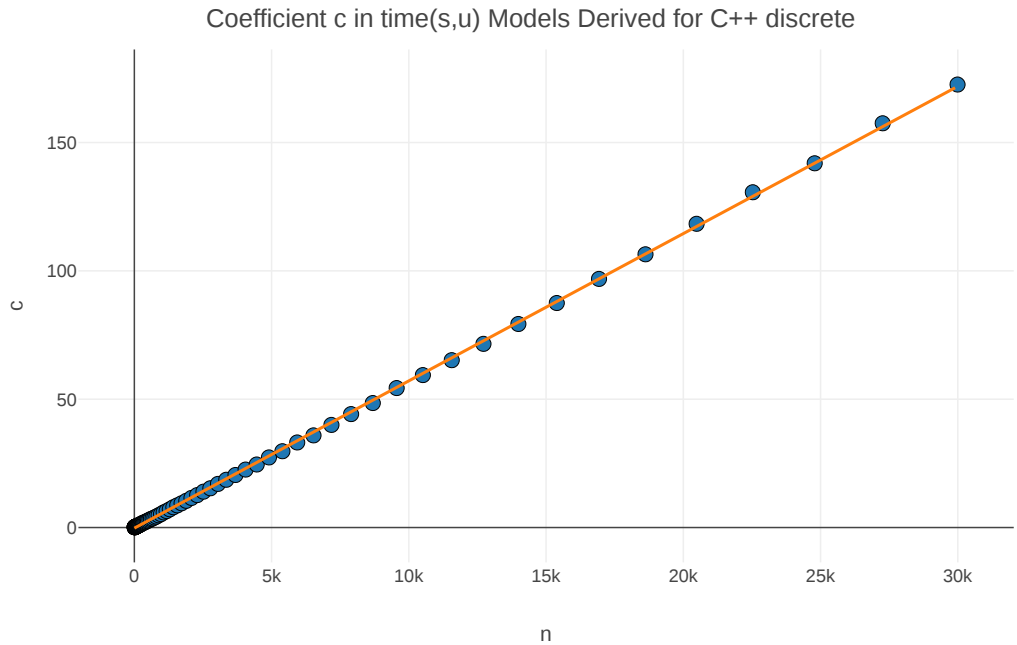

Coefficient a in time(s,u) Models Derived for Alias Method
800
700
600
500
400
300
200
100
0
a
0
5k
10k
15k
20k
25k
30k
n

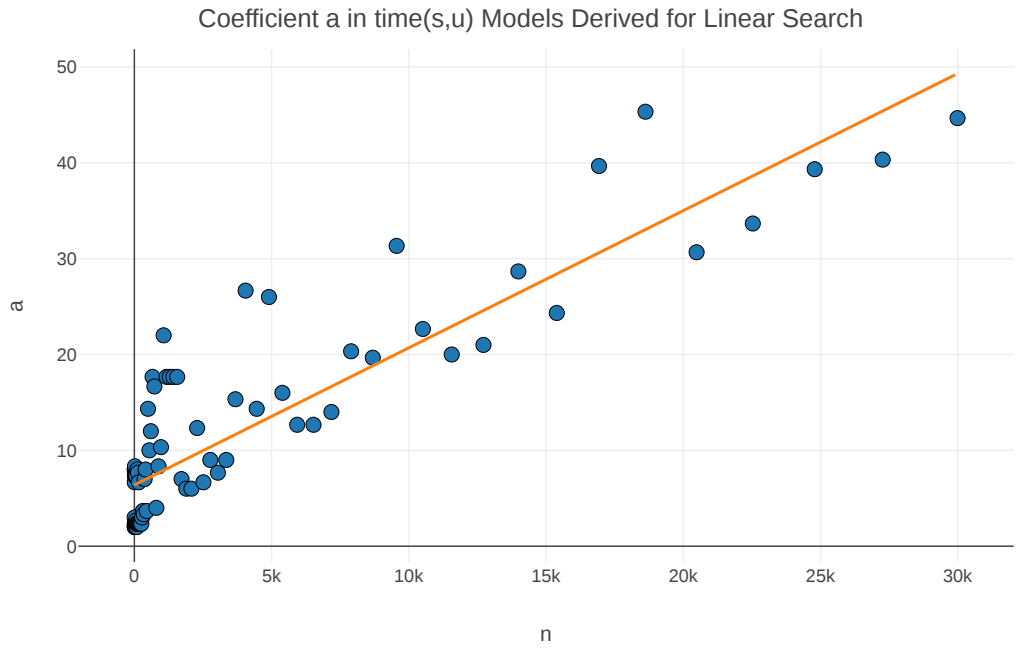

Coefficient b in time(s,u) Models Derived for Alias Method
0.045
0.044
0.043
0.042
0.041
0.04
0.039
0.038
b
0
5k
10k
15k
20k
25k
30k
n

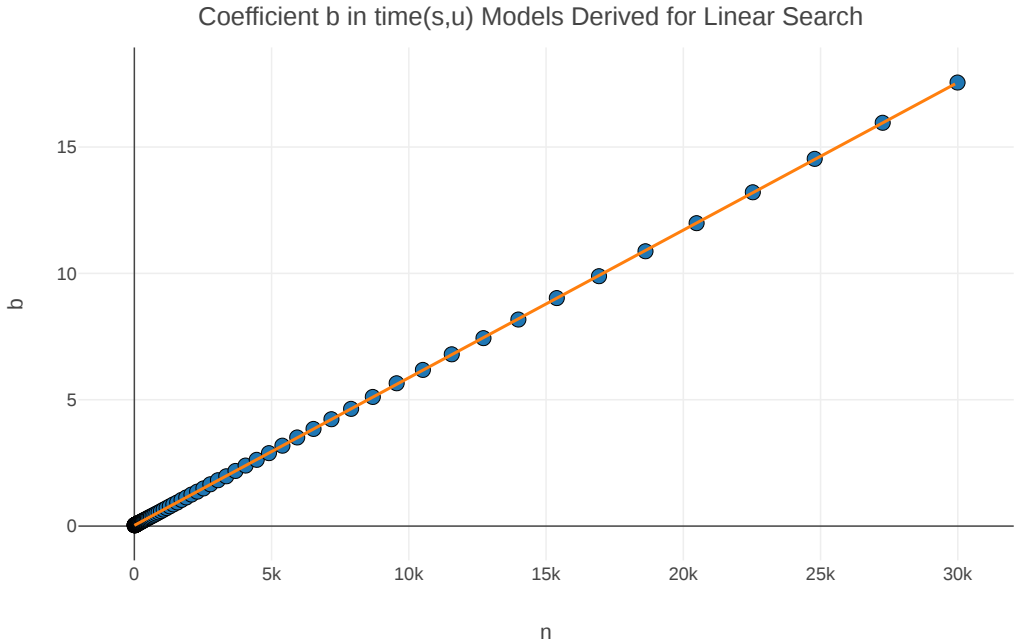

Coefficient c in time(s,u) Models Derived for Alias Method
c
n
0
100
200
300
400
500
600
700
0
5k
10k
15k
20k
25k
30k

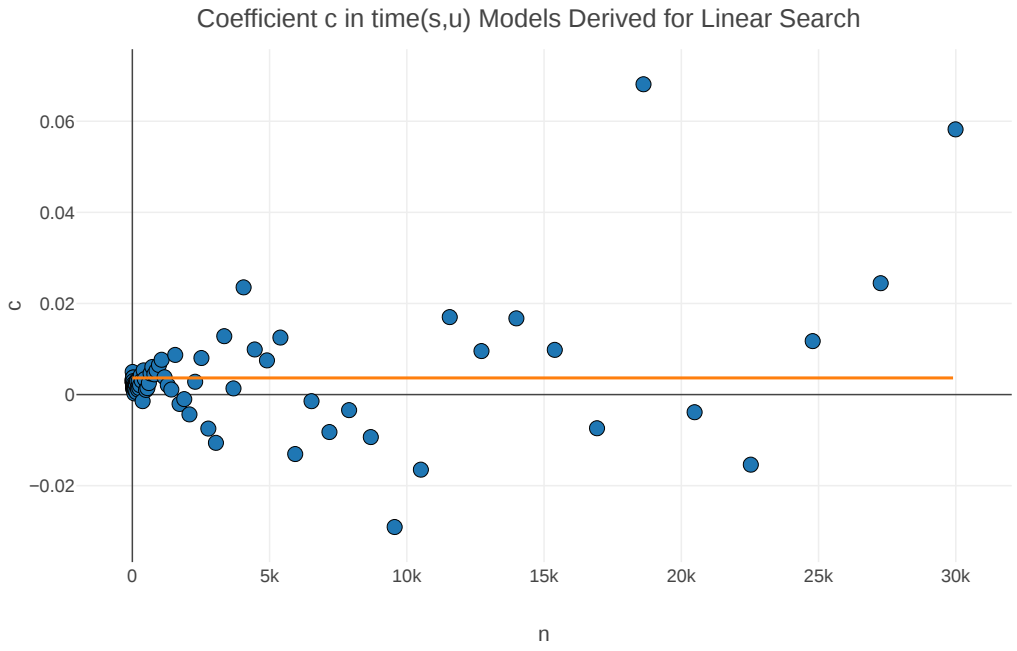

Coefficient a in time(s,u) Models Derived for C++ discrete
a
n
0
100
200
300
400
500
0
5k
10k
15k
20k
25k
30k

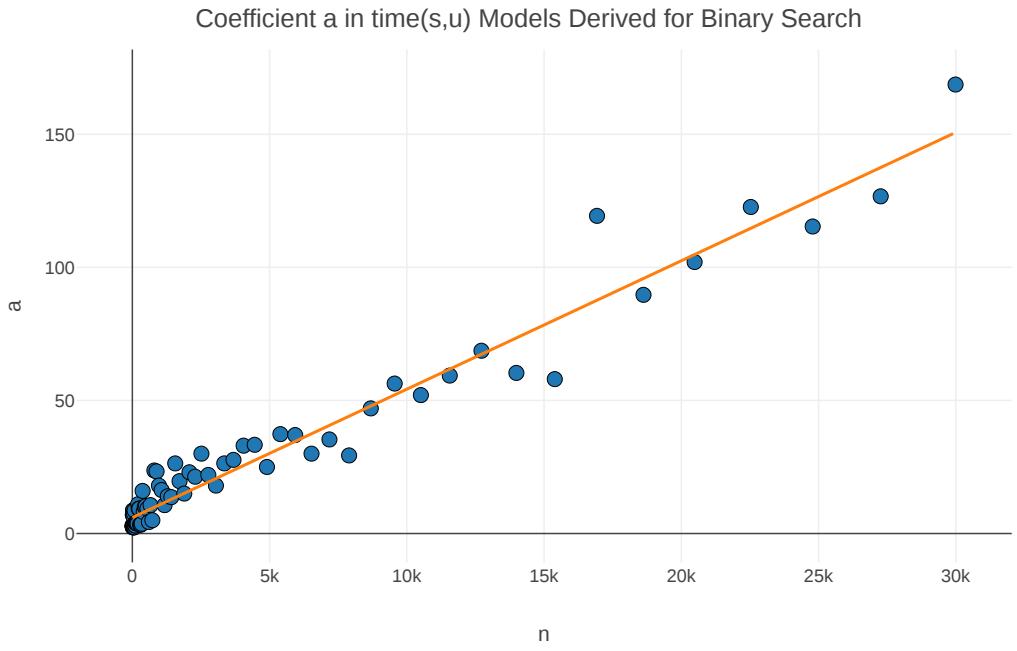

Coefficient b in time(s,u) Models Derived for C++ discrete
b
0.1
0.08
0.06
0.04
0.02
0
5k
10k
15k
20k
25k
30k
n

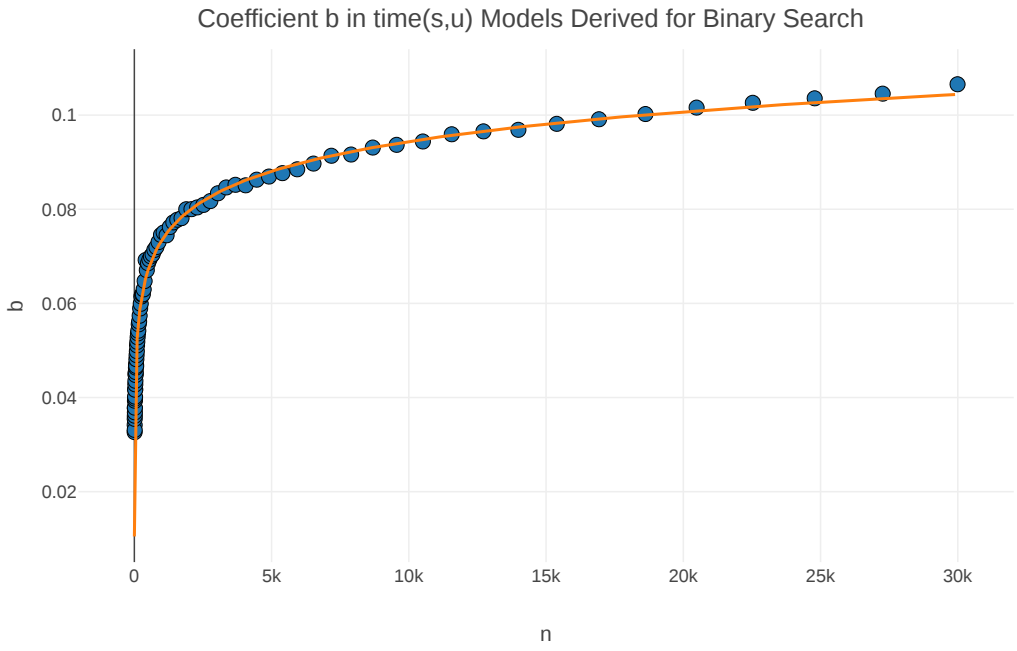

Coefficient c in time(s,u) Models Derived for C++ discrete
c
150
100
50
0
0
5k
10k
15k
20k
25k
30k
n

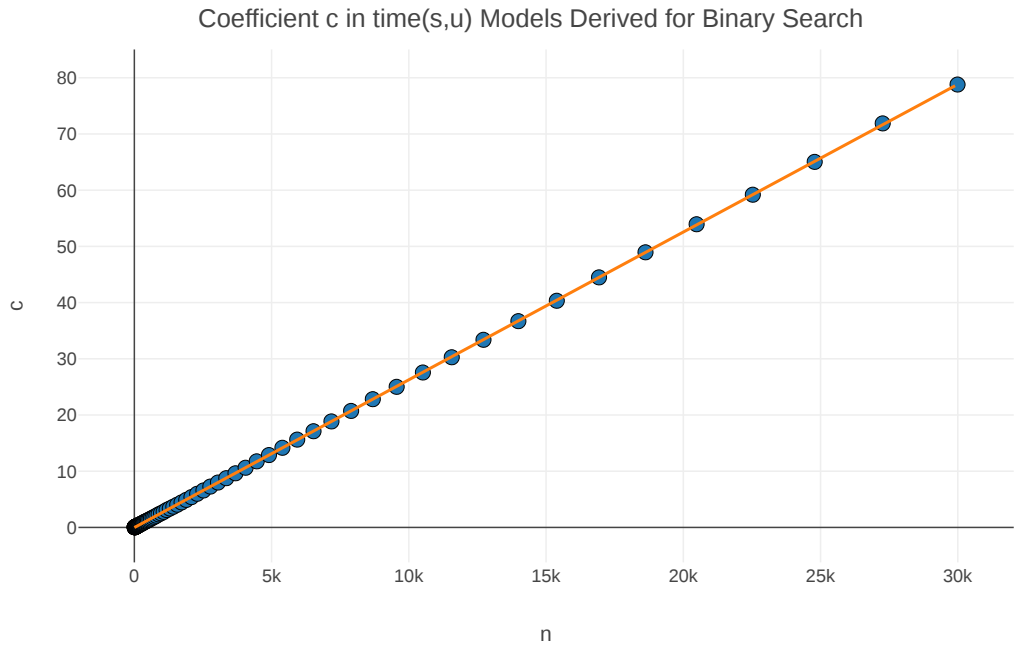

Coefficient a in time(s,u) Models Derived for Linear Search
a
n
0
10
20
30
40
50
0
5k
10k
15k
20k
25k
30k

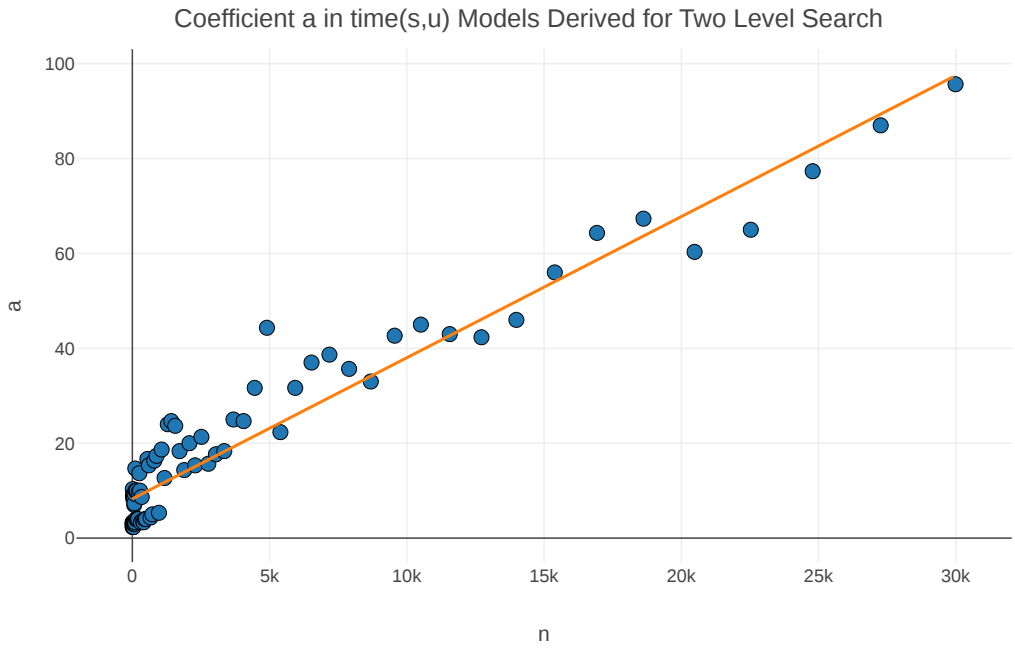

Coefficient b in time(s,u) Models Derived for Linear Search
b
n
0
5
10
15
0
5k
10k
15k
20k
25k
30k

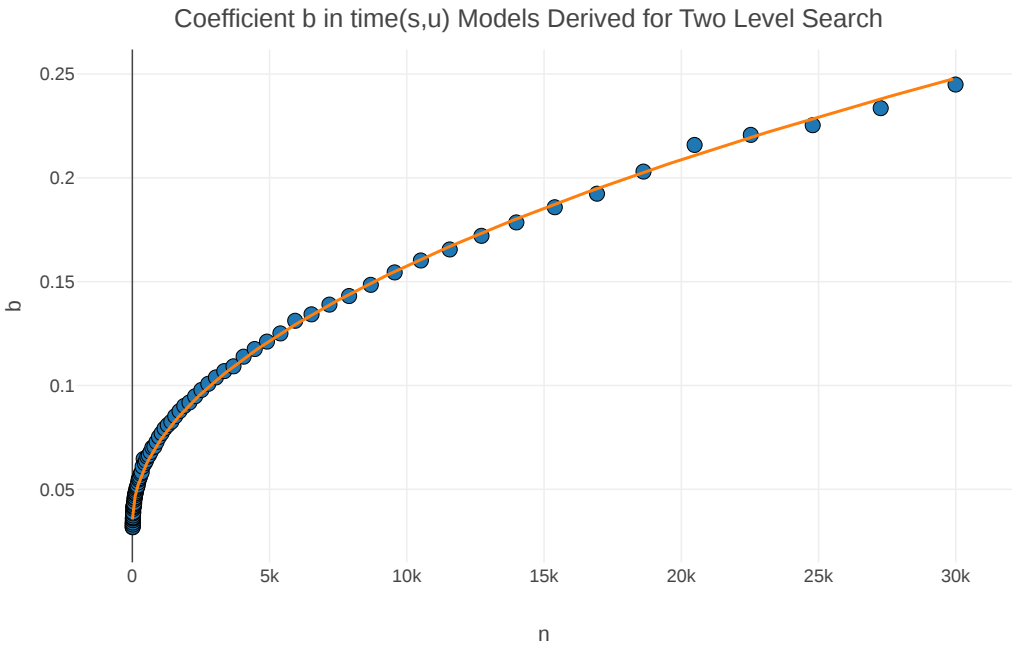

Coefficient c in time(s,u) Models Derived for Linear Search
0.06
0.04
0.02
0
−0.02
c
0
5k
10k
15k
20k
25k
30k
n

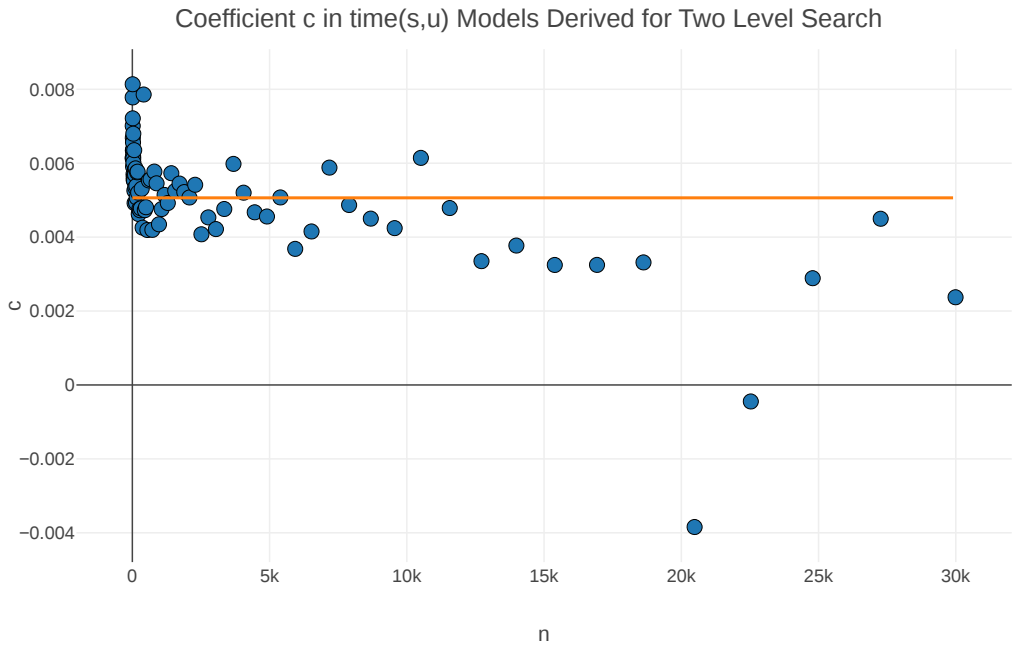

Coefficient a in time(s,u) Models Derived for Binary Search
150
100
50
0
a
0
5k
10k
15k
20k
25k
30k
n

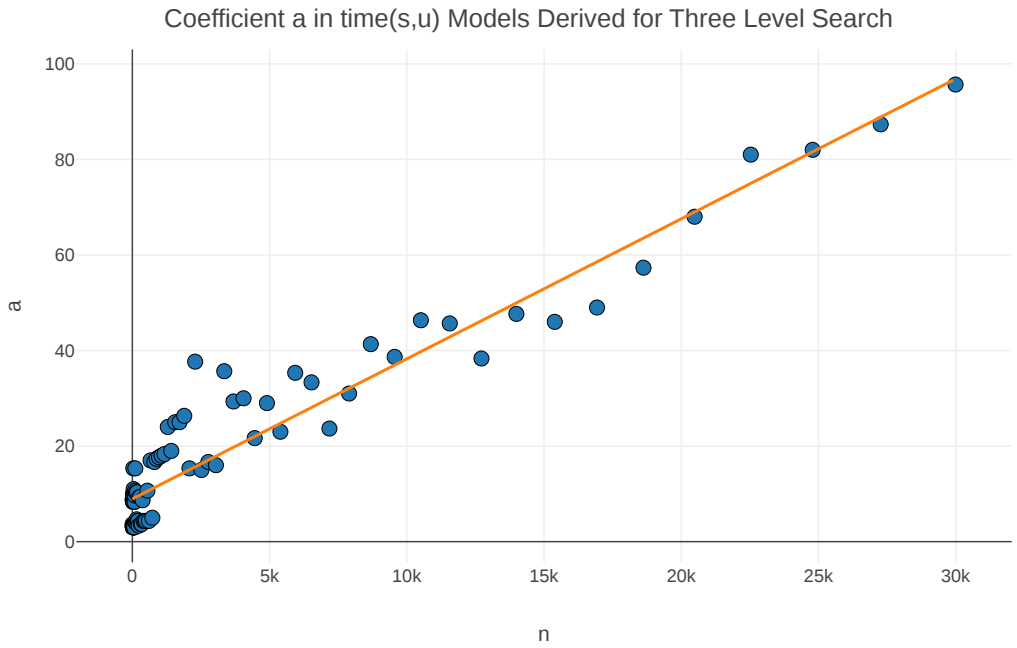

Coefficient b in time(s,u) Models Derived for Binary Search
b
0.1
0.08
0.06
0.04
0.02
0
5k
10k
15k
20k
25k
30k
n

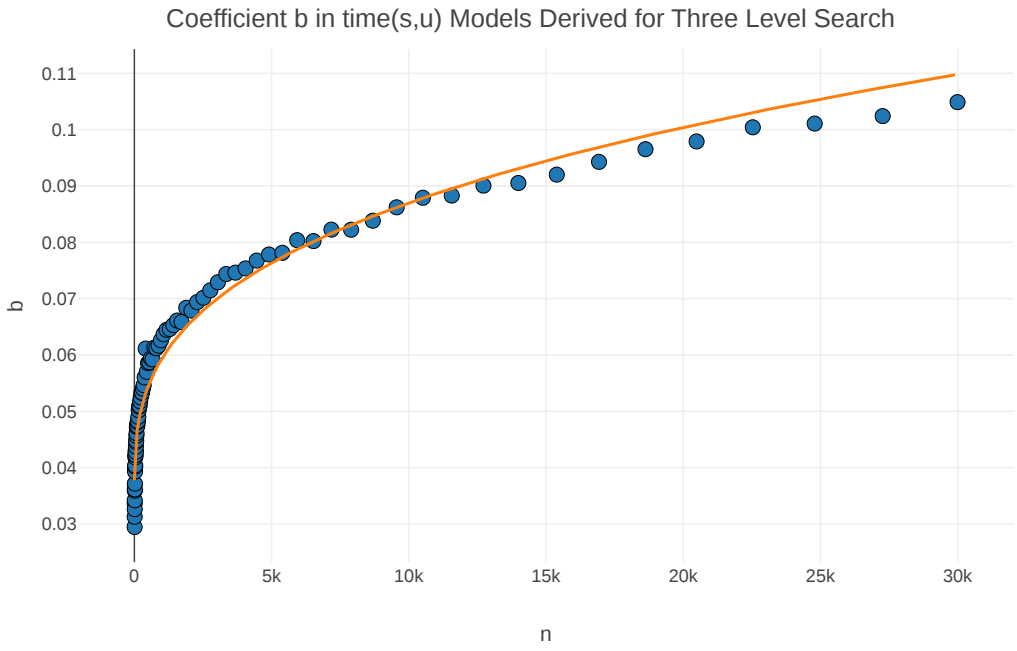

Coefficient c in time(s,u) Models Derived for Binary Search
c
80
70
60
50
40
30
20
10
0
0
5k
10k
15k
20k
25k
30k
n

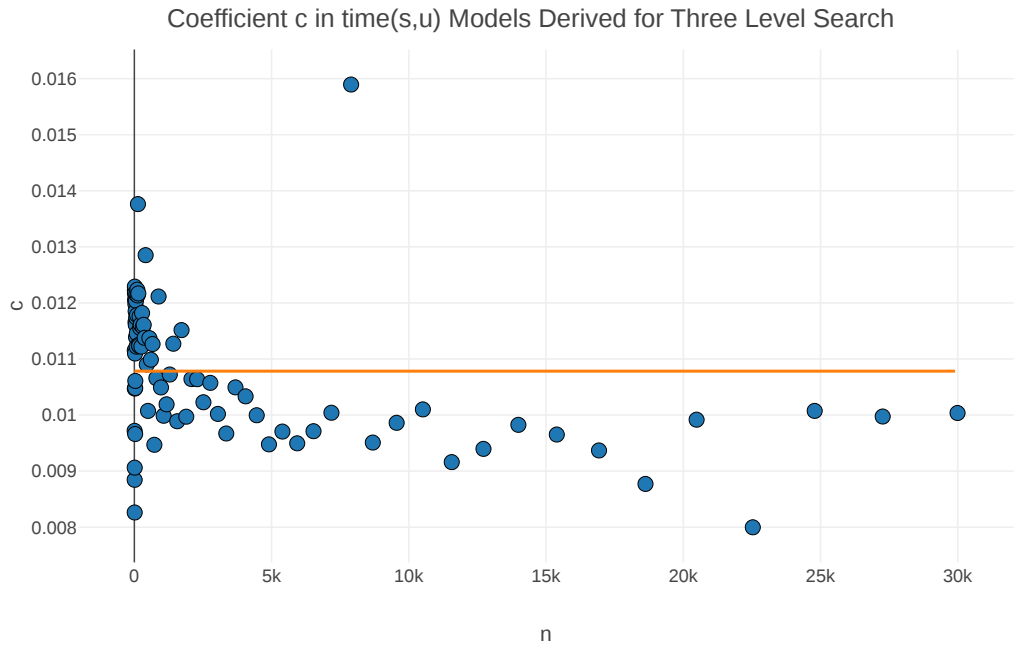

Coefficient a in time(s,u) Models Derived for Two Level Search
100
80
60
40
20
0
a
0
5k
10k
15k
20k
25k
30k
n

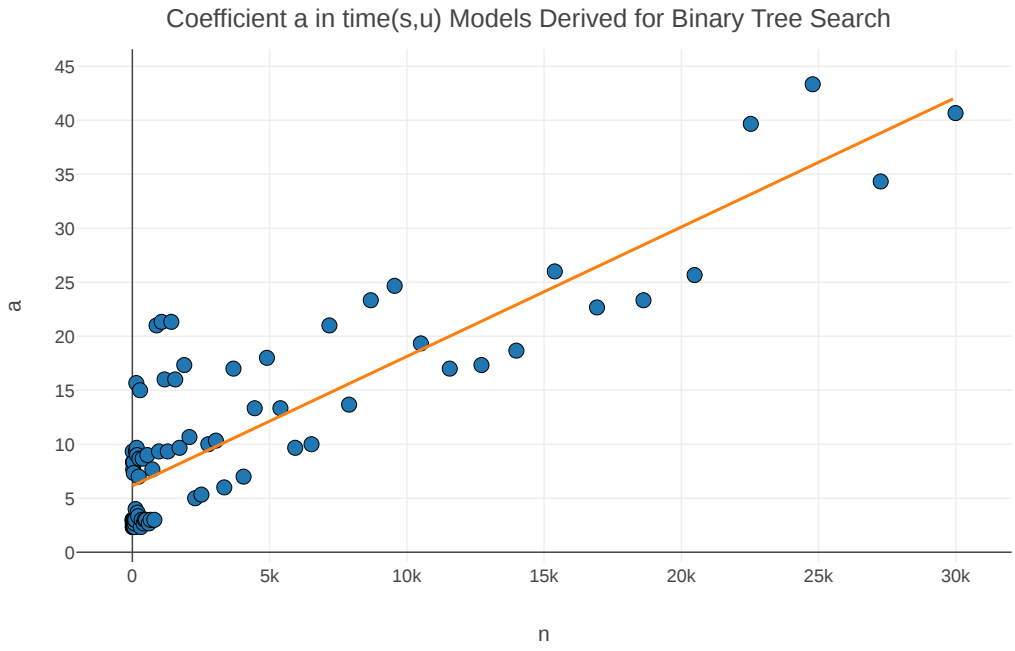

Coefficient b in time(s,u) Models Derived for Two Level Search
0.25
0.2
0.15
0.1
0.05
b
0
5k
10k
15k
20k
25k
30k
n

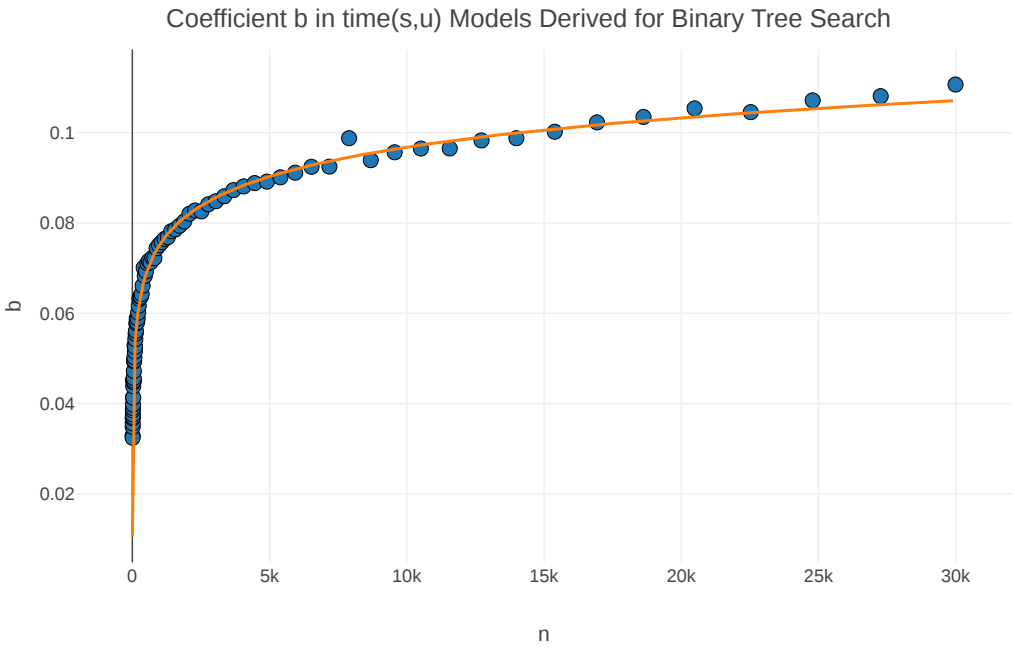

Coefficient c in time(s,u) Models Derived for Two Level Search
0.008
0.006
0.004
0.002
0
−0.002
−0.004
c
0
5k
10k
15k
20k
25k
30k
n

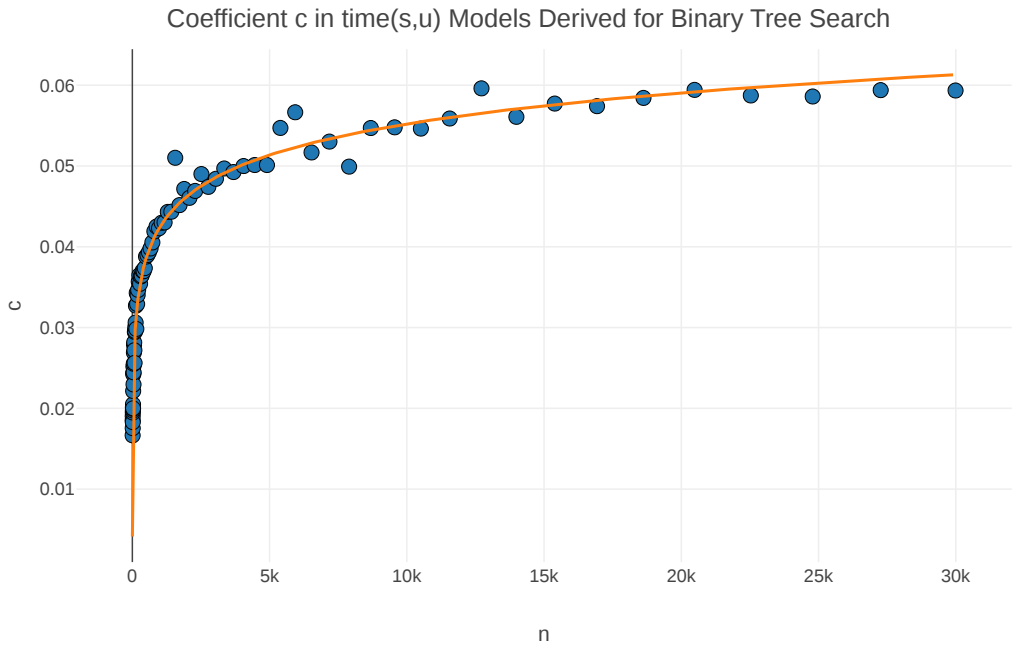

Coefficient a in time(s,u) Models Derived for Three Level Search
100
80
60
40
20
0
a
0
5k
10k
15k
20k
25k
30k
n

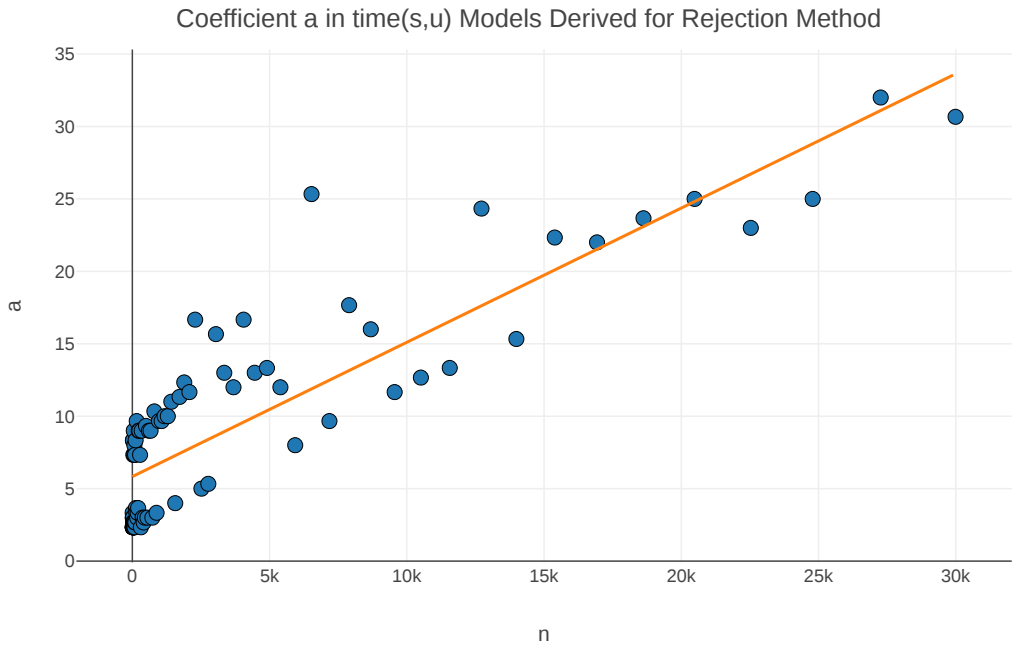

Coefficient b in time(s,u) Models Derived for Three Level Search
b
0.11
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0
5k
10k
15k
20k
25k
30k
n

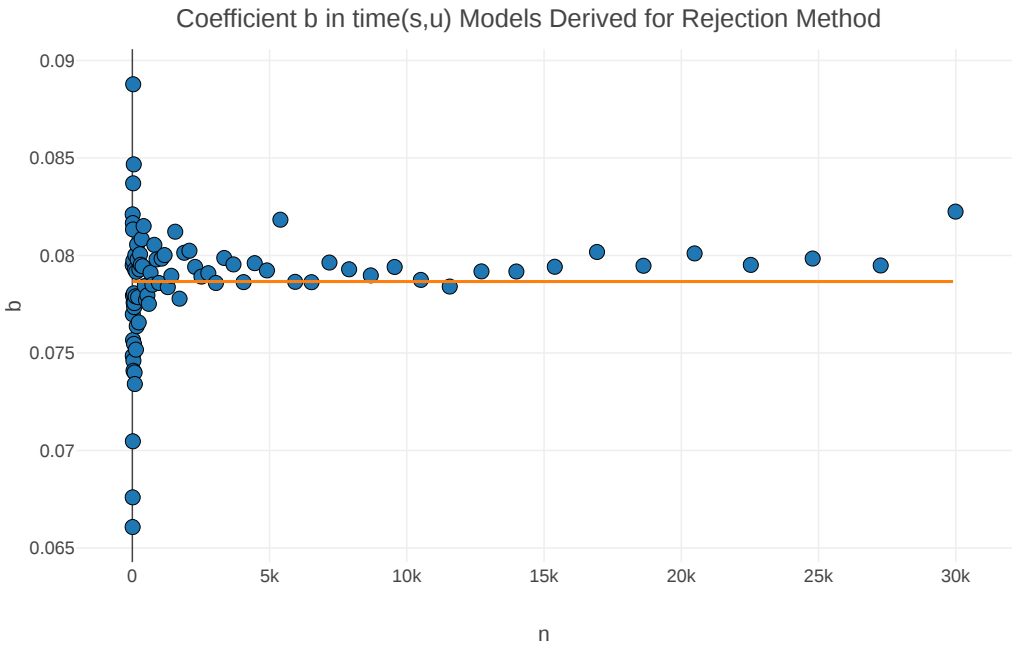

Coefficient c in time(s,u) Models Derived for Three Level Search
c
0.016
0.015
0.014
0.013
0.012
0.011
0.01
0.009
0.008
0
5k
10k
15k
20k
25k
30k
n

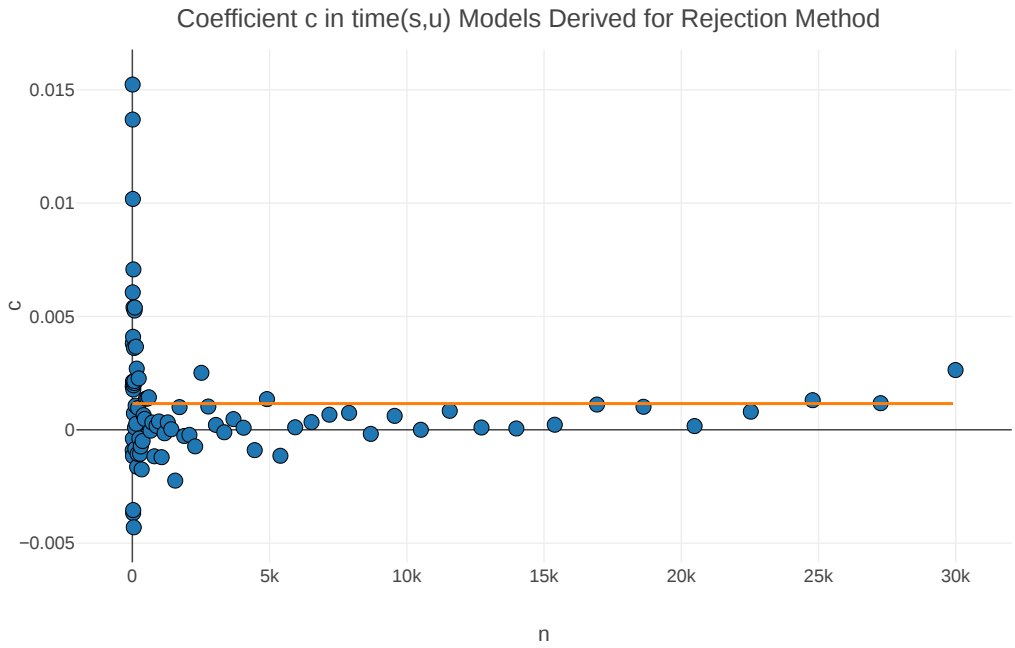

Coefficient a in time(s,u) Models Derived for Binary Tree Search
45
40
35
30
25
20
15
10
5
0
a
0
5k
10k
15k
20k
25k
30k
n

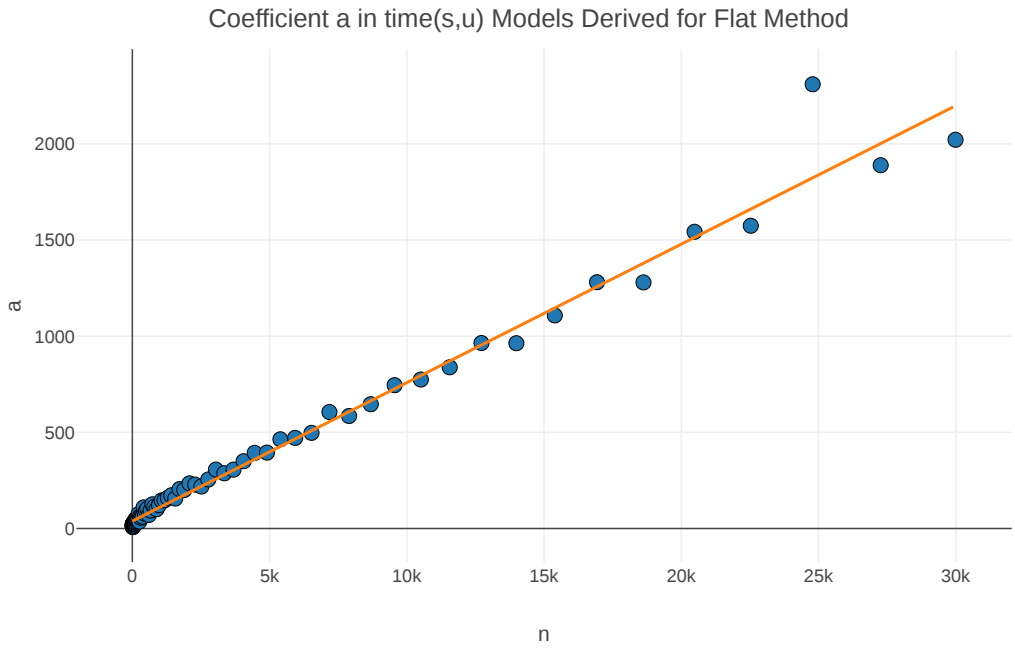

Coefficient b in time(s,u) Models Derived for Binary Tree Search
0.1
0.08
0.06
0.04
0.02
b
0
5k
10k
15k
20k
25k
30k
n

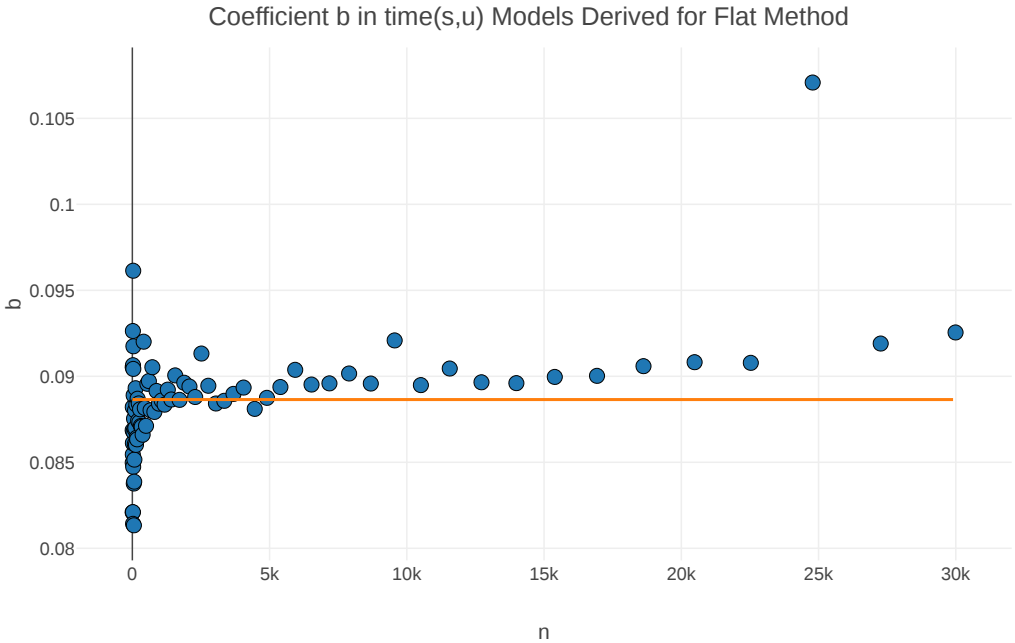

Coefficient c in time(s,u) Models Derived for Binary Tree Search
c
0.06
0.05
0.04
0.03
0.02
0.01
0
5k
10k
15k
20k
25k
30k
n

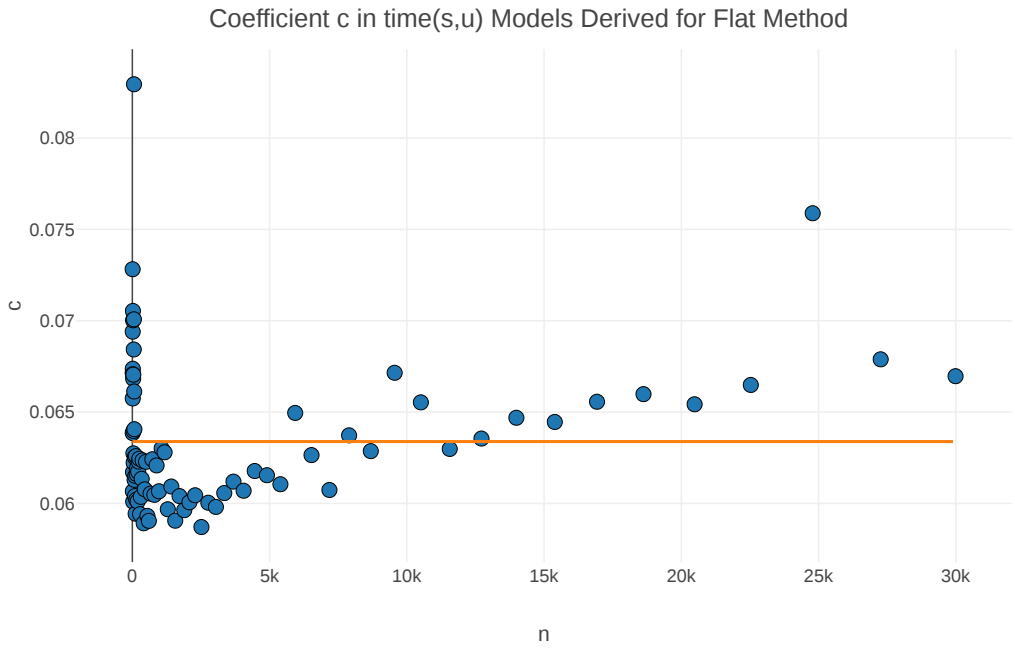

Coefficient a in time(s,u) Models Derived for Rejection Method
a
35
30
25
20
15
10
5
0
0
5k
10k
15k
20k
25k
30k
n

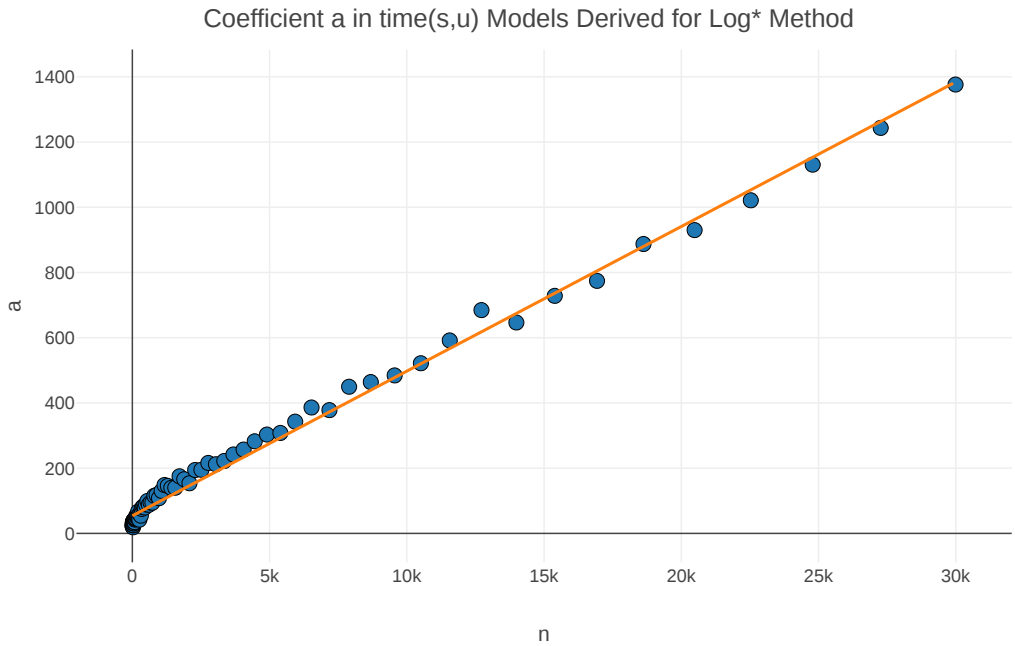

Coefficient b in time(s,u) Models Derived for Rejection Method
0.09
0.085
0.08
0.075
0.07
0.065
b
0
5k
10k
15k
20k
25k
30k
n

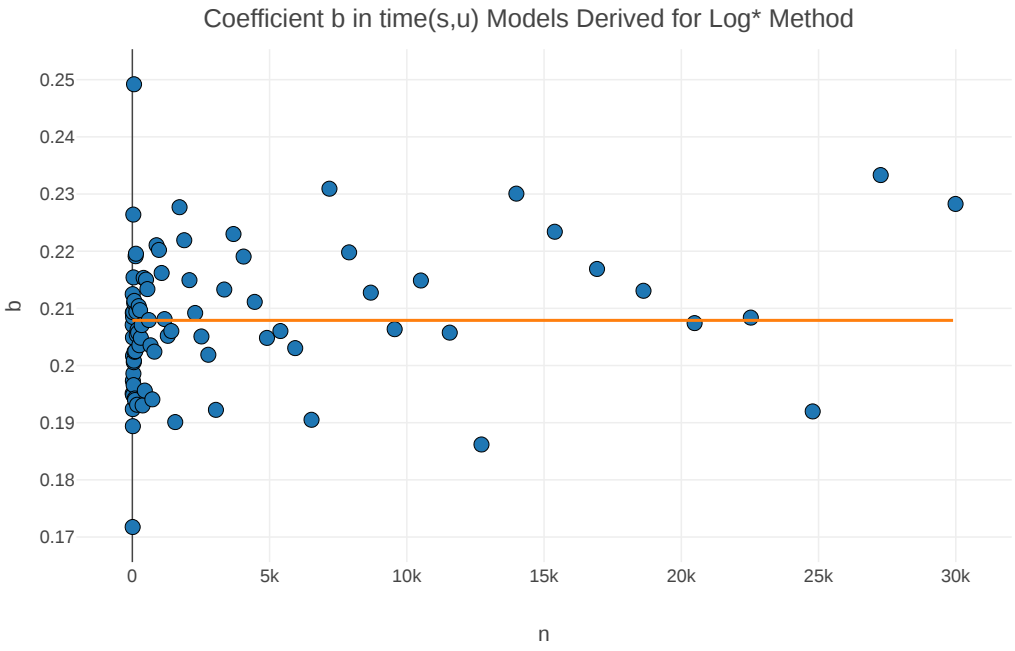

Coefficient c in time(s,u) Models Derived for Rejection Method
0.015
0.01
0.005
0
−0.005
c
0
5k
10k
15k
20k
25k
30k
n

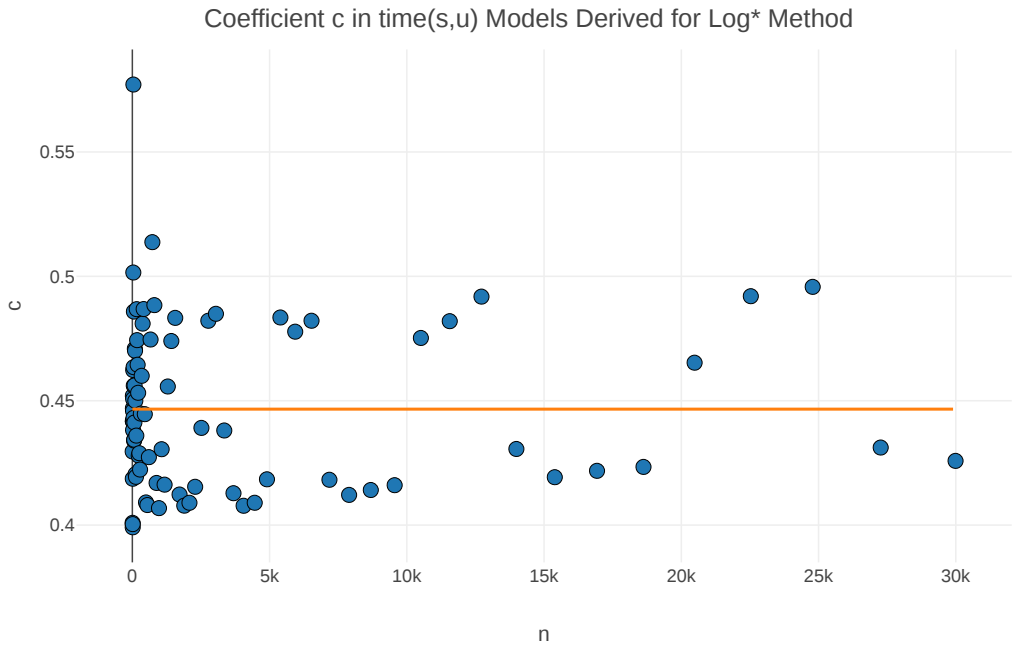

Coefficient a in time(s,u) Models Derived for Flat Method
2000
1500
1000
500
0
a
0
5k
10k
15k
20k
25k
30k
n


Coefficient b in time(s,u) Models Derived for Flat Method
0.105
0.1
0.095
0.09
0.085
0.08
b
0
5k
10k
15k
20k
25k
30k
n


Coefficient c in time(s,u) Models Derived for Flat Method
0.08
0.075
0.07
0.065
0.06
c
0
5k
10k
15k
20k
25k
30k
n


Coefficient a in time(s,u) Models Derived for Log* Method
1400
1200
1000
800
600
400
200
0
a
0
5k
10k
15k
20k
25k
30k
n


Coefficient b in time(s,u) Models Derived for Log* Method
b
0.25
0.24
0.23
0.22
0.21
0.2
0.19
0.18
0.17
0
5k
10k
15k
20k
25k
30k
n


Coefficient c in time(s,u) Models Derived for Log* Method
c
0.55
0.5
0.45
0.4
0
5k
10k
15k
20k
25k
30k
n