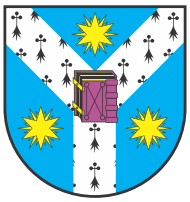
Heuristic Optimizations for Boolean Formulas with Applications in Attribute-Based Encryption

Iulian Oleniuc

Coord. Drd. Alexandru Ioniță



Submitted for the degree of Bachelor in Computer Science

Alexandru Ioan Cuza University of Iași

June 2023

Abstract

We present a method of optimizing monotone Boolean formulas by rewriting them in a simpler form (i.e., using fewer literals). It makes use of three operations that can be applied to the Abstract Syntax Tree associated with a given formula, namely factorization, absorption, and distribution. We combined this method with four different heuristics: Naïve, Hill Climbing, Simulated Annealing, and Custom Heuristic; their iterated versions were taken into account as well. Our main motivation was improving the performance of a specific Attribute-Based Encryption scheme. Therefore, we tested the performance of our heuristics inside ABE systems, but also as standalone optimizers for Boolean formulas. They managed to score improvements of up to 60% in real-world scenarios! Additionally, we describe the high-level idea of a technique that is completely different from the first one, for it operates directly on the Boolean circuit.

Contents

Chapter 1

Introduction

Cloud computing enables the on-demand provision of various resources under the umbrella of computation power and storage over the Internet. As a consequence, enterprises no longer need to manage the IT infrastructure on their own. Many people choose to rely on cloud services, for it offers great flexibility and state-of-the-art security, while also contributing to crucial operational savings [1].

Modern enterprise software relies more and more on cloud services for in-common file storage and collaborative access to data. These systems bring up a crucial privacy problem. Usually, the cloud service provider has access to all the sensitive data. As a matter of fact, even if it is part of the same organization, it is still a big issue that every user has access to all the data.

For instance, the organization may store data of three different levels of sensitivity, namely `public`, `private`, and `top-secret`. A user, by their hierarchical role, should be able to access either only the `public` data, only the `public` and `private` data, or any of the three types of data. A cryptographic framework for implementing this system is Role-Based Access Control (RBAC), where a set of roles is assigned to each user. Every role has different permissions (such as `read` and `write`) over files and every file can be accessed by the users that are assigned a certain role.

There are many real-world scenarios where RBAC is not expressive enough. In the paradigm of Attribute-Based Access Control (ABAC), on the other hand, attributes are assigned to both users and files. The authorization of a user to perform some action on a specific file is made by evaluating the attributes involved. Thus, ABAC lets us define complex fine-grained access policies, based on the relations between the values of different attributes (of either the user or the file).

1.1 Attribute-Based Encryption

Attribute-Based Encryption (ABE), introduced by Goyal et al. in 2006 [2], is a relatively new cryptographic technique that enforces ABAC in data encryption. It takes the forms of Ciphertext-Policy ABE (CP-ABE) and Key-Policy ABE (KP-ABE).

CP-ABE In the context of CP-ABE, the private key of a user is associated with a set of attributes S, and a ciphertext specifies an access policy φ over the universe of attributes $\mathcal{U}$ within the system. A user will be able to decrypt a ciphertext if and only if their attributes satisfy its policy. For example, let $\mathcal{U} = \{A, B, C, D\}$ and $\varphi = ((A \wedge B) \vee (C \wedge D))$. A user with $S = \{C, D\}$ will be able to decrypt a ciphertext with policy φ, whilst a user with $S = \{A, C\}$ will not.

KP-ABE KP-ABE is the dual of CP-ABE. In KP-ABE, the access policy φ is embedded into the secret key of the user, whilst the ciphertext is associated with a set of attributes S. For example, if a user has $\varphi = ((A \vee B) \wedge C)$, then they can decrypt a file having $S = \{A, C\}$, but not one having $S = \{A, B\}$.

In both cases of CP-ABE and KP-ABE, the policies can be defined using conjunctions, disjunctions, and (k, n)-threshold gates (they evaluate to `true` when at least k of the n input attributes are present).

This paper focuses on KP-ABE, so let us give a brief example of a scenario where it can be used. Suppose we are a medical institution and we store many documents related to our patients. They are associated with attributes such as *department* $\in \{\texttt{cardiology}, \texttt{pediatrics}, \texttt{radiology}\}$, *sensitivity* $\in \{\texttt{confidential}, \texttt{public}\}$, or *year* $\in \{1990, \ldots, 2023\}$. A decryption key issued by the central authority may be based on the policy $((\textit{department} = \texttt{cardiology} \vee \textit{department} = \texttt{radiology}) \wedge \textit{sensitivity} = \texttt{public} \wedge (2015 \leq \textit{year} \wedge \textit{year} \leq 2020))$. Using this key, someone can decrypt, for instance, a document having *department* = `radiology`, *sensitivity* = `public`, and *year* = 2017. Note that the attributes can refer to the user (e.g., *department*), but also to the document (e.g., *sensitivity* and *year*).

1.2 Boolean Trees and Boolean Circuits

A Boolean circuit is a Directed Acyclic Graph over a set of input wires, each one corresponding to an attribute in the ABE scheme, and concluding to a single output wire. Each of its internal nodes is either a logic gate (i.e., $\vee$ or $\wedge$) or a *fan-out* node. A logic gate can have any number of input wires, but only one output wire. In contrast, a fan-out node can have any number of output wires, but a single input wire. In other words, the purpose of the fan-out nodes is to send the output of the logic gates to other nodes. Moreover, an input node can have only one output wire and an output node can have only one input wire.

A Boolean tree is a Boolean circuit without fan-out nodes. Therefore, the difference between a Boolean tree and a Boolean circuit (regarding expressiveness) is evident. Unlike a Boolean tree, a Boolean circuit can model a policy whose logical expression form can contain more than one occurrence of the same attribute. Figure 1.1 shows an example of a Boolean circuit and one of a Boolean tree, respectively, alongside the policies they model.

1.3 KP-ABE for Boolean Circuits

The KP-ABE scheme for Boolean circuits described by Țiplea and Drăgan [3] uses *bilinear maps* as key components in the construction process. Consequently, the running time of each of the four phases (i.e., Setup, KeyGen, Encrypt, and Decrypt) depends strictly on the number of pairings (i.e., operations involving bilinear maps) computed, for they are the most expensive.

The KeyGen algorithm applies a *secret sharing* technique to the circuit in a top-down fashion, starting from the output node and ending in the input nodes. The number of shares each attribute will receive at the end of the construction equals the number of paths from its input node to the output node.

The problem we want to tackle in this paper is minimizing the number of pairings that need to be computed during the decryption phase of the Țiplea–Drăgan scheme [3] in order to improve its time performance. As shown above, this can be achieved by replacing the initial circuit with an equivalent one whose number of paths from the output node to the input nodes is as small as possible.

To emphasize the potential impact of this optimization, we show in Figure 1.1 how changing the structure of a simple — yet very natural — Boolean circuit can lead to an improvement of 25% in decryption time, since its number of paths changes from 4 to 3.

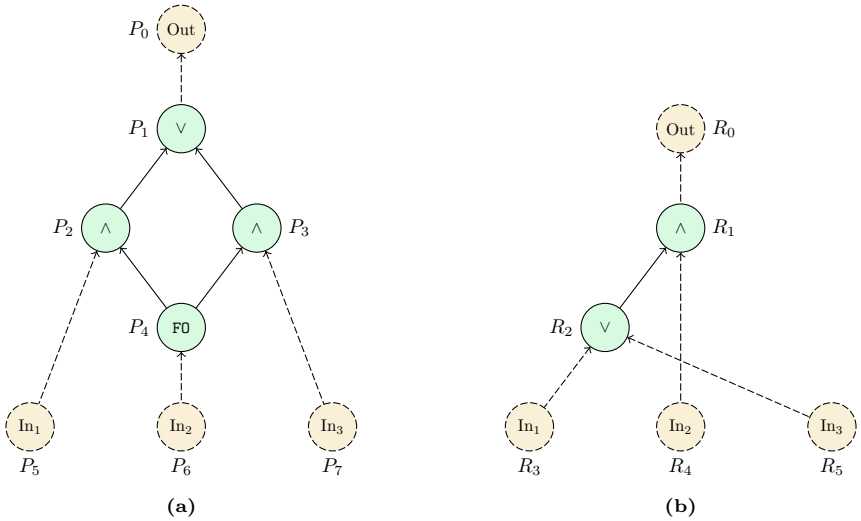


Figure 1.1: A Boolean circuit in **(a)** and its lower-cost equivalent version in **(b)**. The first circuit corresponds to formula $((\mathrm{In}_1 \wedge \mathrm{In}_2) \vee (\mathrm{In}_2 \wedge \mathrm{In}_3))$ and has cost 4, whilst the second one corresponds to formula $((\mathrm{In}_1 \vee \mathrm{In}_3) \wedge \mathrm{In}_2)$ and has cost 3. Moreover, the second circuit also happens to be a Boolean *tree*.

1.4 Our Contribution

We propose two approaches for optimizing Boolean circuits in the context of the Ţiplea–Drăgan scheme [3], with the final purpose of reducing the duration of the decryption phase. The first approach works on the Abstract Syntax Tree of the Boolean formula associated with the circuit, whilst the second approach works directly on the circuit. The latter is not yet fully developed, but we show it has great potential if employed with a specific secret sharing technique. On the other hand, the former solution already achieved huge time improvements, of up to 60%.

1.5 Related Work

Before trying our luck to optimize ABE systems, we did some research on the relevant topics. This section offers a brief report on the information we found about the ABE schemes using Boolean circuits, the problem of Boolean Formula Minimization, and lastly, the heuristic optimizations already applied in cryptography.

1.5.1 ABE for Boolean Circuits

The first ABE systems were introduced in two flavors, namely Key-Policy (KP-ABE) [2] and Ciphertext-Policy (CP-ABE) [4], both of them supporting Boolean trees as access structures. Meanwhile, the problem of designing access structures that are more expressive arose.

For instance, Boolean circuits cover a much wider range of use-cases. Compared to a Boolean tree, a Boolean *circuit* does not impose a limit of one on the fan-out of its gates. Thus, finding an efficient ABE scheme for Boolean circuit access structures is an open problem of great importance in cryptography.

The first such system was introduced by Garg et al. in 2013 [5]. However, it relies on multilinear maps and the Multilinear Decisional Diffie–Hellman (MDDH) assumption, for which there is no known mathematical construction yet [6, 7]. Other approaches [3, 8] provide constructions based on secure and efficient mathematical primitives, like bilinear maps. Nonetheless, the decryption key can expand exponentially for some particular circuits.

1.5.2 Boolean Formula Minimization

The problem of finding a way to rewrite Boolean circuits in a more efficient form (i.e., with fewer logic gates) is well-known in the literature as Boolean Formula Minimization.

One of the most popular algorithms for this problem is the Quine–McCluskey algorithm [9, 10]. However, it requires the entire truth table of the given Boolean formula in order to work. This is impractical for use in an ABE scenario since the Boolean formula grows exponentially with the actual size of the access structure. Computing the truth table for this formula, whose size is already exponential in the size of the formula, makes the approach even more unfeasible.

There is also another algorithm for this problem, called Espresso [11, 12]. Its key advantage is that, due to using various heuristics, it is far more efficient than Quine–McCluskey, hence being able to run on larger inputs. It operates under the context of multiple-valued logic. That is, there are more than two truth values used. In the case of this algorithm, they are `True`, `False`, and `Don't-Care`.

Other newer Boolean minimizers have been proposed, following similar ideas with [13, 14]. However, all these approaches differ from our scope, for their kind of optimization is not the one required by our ABE systems. Moreover, from our experiments, we saw that the above two mentioned algorithms do not have a relevant optimization potential for the specific type of formulas we have to deal with (i.e., *monotone* Boolean formulas).

1.5.3 Heuristic Optimizations in Cryptography

The idea of optimizing the time performance of cryptographic schemes has already been tried before. That being said, most of the time, the optimization needed to be a very peculiar one in order to comply with the chosen cryptosystem.

In this regard, Sasi and Sivanandam present a report [15] on such cryptographic scheme optimizers with applicability in Wireless Sensor Networks. Some of the heuristics mentioned there include genetic algorithms and nature-inspired methods like Particle Swarm and Ant Colony.

As far as we know, there is no previous work on optimizing the decryption phase of ABE based on heuristic approaches. The only work related to ABE systems we found is a very recent paper [16] that applies a heuristic for converting between different types of underlying bilinear map primitives. Not to mention this is a totally different type of optimization from ours, both in means and scope.

Chapter 2

Monotone Boolean Formula Optimization

In this chapter, we shall present the optimization problem we want to approach, in a way that has almost nothing to do with Attribute-Based Encryption anymore, but rather with pure propositional logic. We will formally define the primitives needed, such as the concept of *clean* formulas and the *trim* function. Afterward, we will present the operations that directly affect the cost of the formulas, and finally, we will describe how they are applied in our four proposed heuristics.

2.1 The Optimization Problem

We will first state the formal definition of the problem we want to solve, and afterward, we will go through the propositional logic background needed to fully understand it.

Problem 1 (Monotone Boolean Formula Optimization)**.** *For a given monotone Boolean formula φ, find an equivalent monotone Boolean formula ψ with a cost as low as possible compared to the one of φ. In other words, find some ψ such that $\psi \equiv \varphi$ and $c(\psi) \ll c(\varphi)$.*

2.1.1 Propositional Logic Background

Definition 1 (Truth Value)**.** *The set of* truth values *(or Boolean values) is $\Omega \triangleq \{0, 1\}$. Usually, 0 represents the value* `false` *and 1 represents the value* `true`*.*

Definition 2 (Propositional Variable)**.** *A* propositional variable *is any mathematical symbol denoting a variable that can take values from Ω. Let us denote the set of propositional variables by Σ.*

What makes a Boolean formula *monotone* is the fact that it cannot contain the negation operator (i.e., $\neg$). As a matter of fact, it does not contain implications or equivalences either. Therefore, instead of defining the general concept of Boolean formulas, we will directly refer to this particular type.

Definition 3 (Monotone Boolean Formula)**.** *A* monotone Boolean formula *can be defined as either the symbol associated with a variable in Σ or as an expression of the form $(\varphi_1 \vee \cdots \vee \varphi_n)$ or $(\varphi_1 \wedge \cdots \wedge \varphi_n)$, where $n \geq 2$ is some integer number and $\varphi_1, \ldots, \varphi_n$ are all monotone Boolean formulas as well.*

Example 1. *Some valid monotone Boolean formulas are* $(a \vee b)$, $(x_1 \wedge x_2 \wedge x_3)$, x, *and* $(p \wedge q \wedge (r \vee p))$. *Some expressions that are not Boolean formulas include* $a \wedge b$, $(x \wedge y \vee z)$, (p), $\neg(s \vee t)$, *and* $((m \wedge n) \vee p$.

Hereinafter, we shall refer to "monotone Boolean formulas" as just "formulas."

Definition 4 (Literal)**.** *A formula is called* literal *if it consists of just one symbol.*

Definition 5 (Equal Formulas)**.** *Two formulas* φ *and* ψ *are said to be* equal*, denoted as* $\varphi = \psi$*, if they are either the same literal or* $\langle \varphi_1, \ldots, \varphi_m \rangle \equiv \langle \psi_1, \ldots, \psi_n \rangle$,[1] *where* $(\varphi_1 \oplus \cdots \oplus \varphi_m) \triangleq \varphi$ *and* $(\psi_1 \otimes \cdots \otimes \psi_n) \triangleq \psi$ *such that* $\oplus, \otimes \in \{\vee, \wedge\}$.

Example 2. *The formulas* $\varphi \triangleq (a \vee b \vee a \vee (d \wedge c))$ *and* $\psi \triangleq ((c \wedge d) \vee a \vee a \vee b)$ *are equal. That is for* $\langle a, b, a, (d \wedge c) \rangle \equiv \langle (c \wedge d), a, a, b \rangle$ *and* $(d \wedge c) = (c \wedge d)$.

Definition 6 (Assignment)**.** *An* assignment *is any function* $\tau : \Sigma \to \Omega$. *In other words, an assignment* τ *maps every propositional variable to a Boolean value.*

Definition 7 (Evaluating a Formula)**.** *The* value *of a formula* φ *under an assignment* τ *is denoted by* $\hat{\tau}$. *If* φ *is a literal, then* $\hat{\tau}(\varphi) = \tau(\varphi)$. *If* $(\varphi_1 \vee \cdots \vee \varphi_n) \triangleq \varphi$, *then* $\hat{\tau}(\varphi) = 1$ *if and only if* $\hat{\tau}(\varphi_i) = 1$ *for at least one* i. *Finally, if* $(\varphi_1 \wedge \cdots \wedge \varphi_n) \triangleq \varphi$, *then* $\hat{\tau}(\varphi) = 1$ *if and only if* $\hat{\tau}(\varphi_i) = 1$ *for every* i.

Definition 8 (Equivalent Formulas)**.** *Two formulas* φ *and* ψ *are said to be* equivalent*, denoted as* $\varphi \equiv \psi$*, if* $\hat{\tau}(\varphi) = \hat{\tau}(\psi)$ *for every possible assignment* τ.

Example 3. *One can easily verify that* $x \equiv (x \vee x) \equiv (x \wedge x)$, $(a \vee b) \not\equiv (a \wedge b)$, *and* $((a \wedge b) \vee (a \wedge c)) \equiv (a \wedge (b \vee c))$.

When working with formulas (i.e., applying certain types of operations to them) it would be more intuitive to represent them graphically. That is why we need to introduce the concept of ASTs.

Definition 9 (Abstract Syntax Tree)**.** *The* Abstract Syntax Tree (AST) *associated with a formula* $\varphi \triangleq (\varphi_1 \oplus \cdots \oplus \varphi_n)$*, where* $\oplus \in \{\vee, \wedge\}$*, is a tree whose children are the ASTs of each* φ_i *and whose root is labeled as* φ *if* φ *is a literal, or as* $\oplus$ *otherwise. The nodes of the AST associated with literals are called* leaves.

Example 4. *The AST associated with* $(a \vee (b \wedge c \wedge d))$ *is depicted in Figure 2.1.*

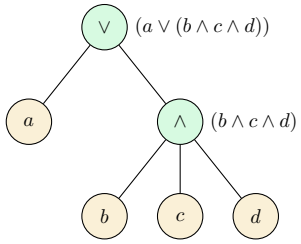


Figure 2.1: The AST associated with formula $(a \vee (b \wedge c \wedge d))$.

Definition 10 (Subtrees)**.** *A* subtree *of a tree* T *is some node of* T *alongside all its descendants. A* subtree* *of* T *is the union of some subtrees rooted in sibling nodes of* T. *A* subtree$^+$ *of* T *is a non-empty* subtree* *of* T.

[1] The symbols $\langle$ and $\rangle$ are used to delimit arrays (i.e., sequences of values). In the context of arrays, $A \equiv B$ means that the elements of A can be rearranged in such a way that the new array A will become the same as B. For example, $\langle 1, 3, 2 \rangle \equiv \langle 1, 2, 3 \rangle$, whilst $\langle 1, 2, 2 \rangle \not\equiv \langle 1, 2 \rangle$.

Subtrees are represented with *solid* triangles, subtrees* with *dashed* triangles, and subtrees$^+$ with *double* triangles. These triangles are labeled with letters denoting their associated formulas (or ϵ if they are empty).

Example 5. *Figure 2.2 shows a way of representing the AST in Figure 2.1 using every type of subtree.*

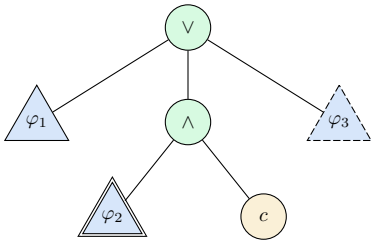


Figure 2.2: The AST in Figure 2.1 drawn using every type of subtree. The subformulas $\varphi_1 \triangleq a$, $\varphi_2 \triangleq b \wedge d$, and $\varphi_3 \triangleq \epsilon$ represent a subtree, a subtree$^+$, and a subtree*, respectively. Now it may become obvious why we defined the equality of formulas the way we did; it lets us choose φ_2 this way, since $(b \wedge c \wedge d) = (b \wedge d \wedge c)$. Moreover, note the lack of parentheses in φ_2. It shows that φ_2 is not a single subtree, but rather a union of subtrees.

2.1.2 Clean Formulas and Trimming Trees

We observed that it would be far easier to work with formulas when their structure respects three particular invariants, which we will present next. Let us call these formulas *clean*. We designed all our operations to work with formulas that are already clean and in a way that preserves these invariants.

Definition 11 (Clean Formula)**.** *We say that a formula is* clean *if its associated AST does not contain any of the following structural design flaws:*

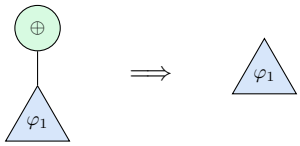
1. *any node with only one child;*
2. *any node with the same operator as its parent;*
3. *any node with two or more children of the same formula.*

Example 6. *Formula* $((a \wedge b))$ *breaks invariant (1), formula* $(a \vee (b \vee c))$ *breaks invariant (2), and formula* $(a \wedge a \wedge b)$ *breaks invariant (3). Some equivalent clean versions of these formulas are* $(a \wedge b)$*,* $(a \vee b \vee c)$*, and* $(a \wedge b)$*, respectively.*

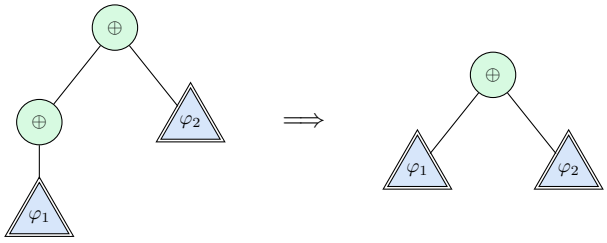
Figure 2.3 shows the general subtrees that violate each of the above invariants and how to correct them. However, the order in which they should be corrected to obtain a genuinely valid formula is not immediately evident, for a correction often leads to another violation.

Remark 1. *One may wonder what is the purpose of invariant (1), since it does not portray an AST associated with a* valid *formula. Well, we might obtain it while trying to correct another invariant violation. For example,* $(a \wedge a)$ *gets transformed into* (a)*, which must be corrected as* a*.*

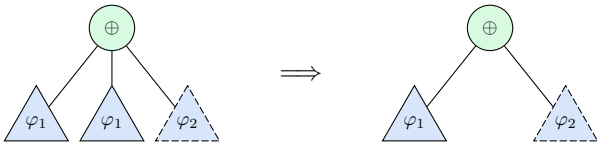
As said before, we should elaborate on the process of transforming a given formula into a clean one. We wrote a special function for that [17], called *trim*. Since it is simple enough, we described it in Algorithm 1.



(a) (φ_1) becomes φ_1.



(b) $((\varphi_1) \oplus \varphi_2)$ becomes $(\varphi_1 \oplus \varphi_2)$.



(c) $(\varphi_1 \oplus \varphi_1 \oplus \varphi_2)$ becomes $(\varphi_1 \oplus \varphi_2)$.

Figure 2.3: The ith subfigure illustrates an AST (on the left) violating invariant (i) and a way of rewriting it (on the right) to get rid of that problem. Here, $\oplus \in \{\vee, \wedge\}$.

Algorithm 1 Trim

Require: The AST root u.
Ensure: The formula of u is clean.

```
 1 let C be an empty array
 2 for each child v of u do
 3     trim v recursively
 4     if u and v have the same operator then                ▷ invariant (2)
 5         for each child w of v do
 6             if the formula of w was not seen before then
 7                 add w to C
 8             end if
 9         end for
10     else if the formula of v was not seen before then
11         add v to C
12     end if
13 end for
14 make C the new children array of u                        ▷ invariant (3)
15 if u has only one child v then                            ▷ invariant (1)
16     u ← v
17 end if
```

Remark 2. *One may wonder how the formula associated with any node of the tree can be obtained fast enough at any given moment. The answer is that we store it in the node itself all the time. Hence, it needs to be updated when the children of the node change (i.e., on line 14). After the trim, this update also needs to be propagated to every ancestor of this node, until reaching the root of the tree.*

Remark 3. *In order to easily check if two formulas are equal, we need to store the formulas of the AST in a particular way. When updating some formula φ, if the formulas of its children are $\varphi_1, \ldots, \varphi_n$, we do* not *compute φ as $(\varphi_1 \oplus \cdots \oplus \varphi_n)$, where $\oplus \in \{\vee, \wedge\}$ is the operator of the node of φ, but rather as $(\psi_1 \oplus \cdots \oplus \psi_n)$, where $\langle \psi_1, \ldots, \psi_n \rangle$ is the sorted version of the array $\langle \varphi_1, \ldots, \varphi_n \rangle$. The formulas are compared (and thus, sorted) as character strings.*

Example 7. *We broadly show below the process of running Algorithm 1 on the AST of some formula φ. Next to each line, we mentioned the one invariant violated by the current formula that gets corrected on the next line.*

$$
\begin{aligned}
\varphi &\triangleq (((a \vee b) \wedge (a \vee b)) \vee a \vee (b \vee b) \vee (((c)))) && \textit{invariant (1)} \\
&\equiv (((a \vee b) \wedge (a \vee b)) \vee a \vee (b \vee b) \vee ((c))) && \textit{invariant (1)} \\
&\equiv (((a \vee b) \wedge (a \vee b)) \vee a \vee (b \vee b) \vee (c)) && \textit{invariant (1)} \\
&\equiv (((a \vee b) \wedge (a \vee b)) \vee a \vee (b \vee b) \vee c) && \textit{invariant (3)} \\
&\equiv (((a \vee b) \wedge (a \vee b)) \vee a \vee (b) \vee c) && \textit{invariant (1)} \\
&\equiv (((a \vee b) \wedge (a \vee b)) \vee a \vee b \vee c) && \textit{invariant (3)} \\
&\equiv (((a \vee b)) \vee a \vee b \vee c) && \textit{invariant (1)} \\
&\equiv ((a \vee b) \vee a \vee b \vee c) && \textit{invariant (2)} \\
&\equiv (a \vee b \vee a \vee b \vee c) && \textit{invariant (3)} \\
&\equiv (a \vee b \vee c)
\end{aligned}
$$

2.2 Operating on the Abstract Syntax Tree

Remember that our goal is to lower the cost $c(\varphi)$ of some formula φ, but we did not define the cost function yet.

Definition 12 (Formula Cost)**.** *The* cost *of a formula φ, denoted by $c(\varphi)$, is the number of literals in φ. For the sake of brevity, for a tree T, we will also denote by $c(T)$ the cost of the formula associated with T (i.e., the number of leaves of T).*

The reason behind this definition is that the number of literals of a formula is equal to the number of paths from the output node to the input nodes in the Boolean circuit associated with it — the value we stated in the first chapter that we want to optimize.

Example 8. *For $\varphi \triangleq (a \vee (b \wedge (c \vee a) \wedge b))$, we have that $c(\varphi) = 5$.*

Remark 4. *One should pay attention not to confuse the cost of φ with the number of variables involved in φ, which in Example 8 would lead to $c(\varphi) = 3$.*

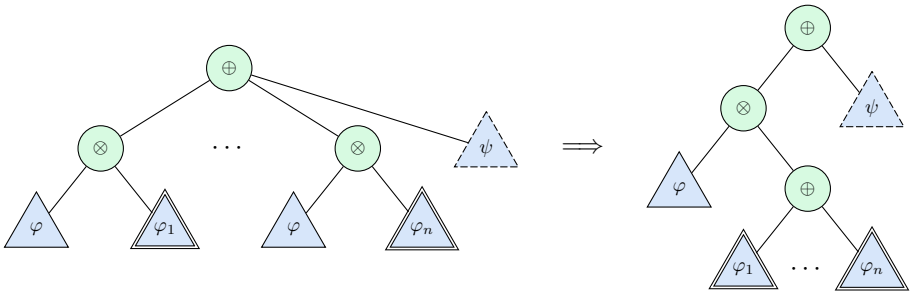
The nature of the heuristics we used to optimize the cost of a given formula φ requires two kinds of functions to be defined, namely one that decreases $c(\varphi)$, but also one that *increases* $c(\varphi)$. The rest of this section will focus on defining three AST operations we designed to help us in this regard. That is, two for decreasing the cost and one for increasing it.

Factorization The first operation is called *factorization*. It was inspired by the fact that $((a \wedge b) \vee (a \wedge c)) \equiv (a \wedge (b \vee c))$. The intuitive explanation is that for one of the clauses $(a \wedge b)$ and $(a \wedge c)$ to be true, a definitely needs to be true, since it is a term of both; moreover, the other term of one of the clauses has to be true as well. That is, $(b \vee c)$ must be true. Figure 2.4a shows the general structure of a factorization.

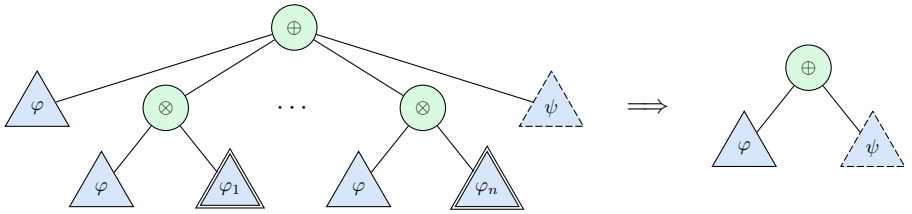
Absorption The second operation is called *absorption*. It treats a special case that could not be covered by factorization. For example, in $(a \vee (a \wedge b))$, we may want to factorize a, but the first a is not the term of any conjunctive clause. In other words, informally, we would like to obtain something like $(a \wedge (1 \vee b))$, which is not a formula. Instead, we will explain why $(a \vee (a \wedge b)) \equiv a$ without mentioning the idea of factorization. Indeed, the intuition is that the value of b does not matter, considering that it only influences the clause $(a \wedge b)$, which requires a to be true anyway. Figure 2.4b shows the general structure of an absorption.

Distribution The third and final operation is called *distribution*. In [18] it was initially called *defactorization* (for it is roughly the opposite operation of factorization), but meanwhile we found a better name. It is based on the fact that, for instance, $(a \wedge (b \vee c)) \equiv ((a \wedge b) \vee (a \wedge c))$, and it is depicted in Figure 2.4c.

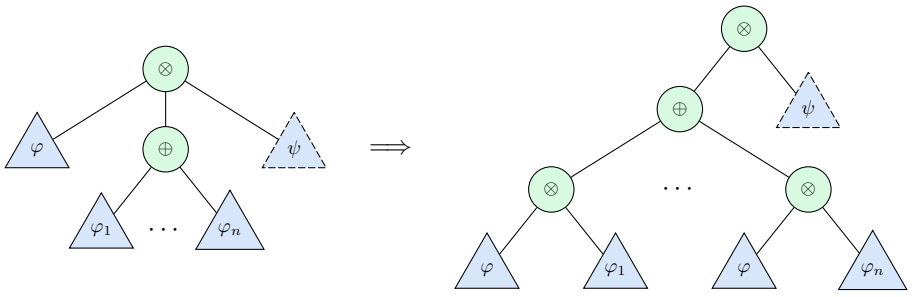
Note that all the operations are symmetric regarding the node operators involved. For example, $((a \wedge b) \vee (a \wedge c)) \equiv (a \wedge (b \vee c))$ is true, and $((a \vee b) \wedge (a \vee c)) \equiv (a \vee (b \wedge c))$ is true as well. For factorizations and distributions, the reasoning is similar to the explanations given earlier.



(a) Factorization: $(\varphi \otimes \varphi_1) \oplus \cdots \oplus (\varphi \otimes \varphi_n)$ becomes $(\varphi \otimes (\varphi_1 \oplus \cdots \oplus \varphi_n))$.



(b) Absorption: $\varphi \oplus (\varphi \otimes \varphi_1) \oplus \cdots \oplus (\varphi \otimes \varphi_n)$ becomes φ.



(c) Distribution: $\varphi \otimes (\varphi_1 \oplus \cdots \oplus \varphi_n)$ becomes $((\varphi \otimes \varphi_1) \oplus \cdots \oplus (\varphi \otimes \varphi_n))$.

Figure 2.4: The operations of factorization, absorption, and distribution, respectively. The initial AST is on the left side of each subfigure, whilst the right side shows the AST after applying the operation. Here, $n \geq 2$ and $(\oplus, \otimes) \in \{(\vee, \wedge), (\wedge, \vee)\}$.

However, it is not the case for absorption. The informal way of seeing $(a \vee (a \wedge b))$ as the factorization $(a \wedge (1 \vee b))$ misleads us into thinking that $(a \wedge (a \vee b)) \equiv (a \vee (1 \wedge b)) \equiv (a \vee b)$. Regardless, it is obvious that $(a \wedge (a \vee b)) \not\equiv (a \vee b)$; take for instance $\tau = \{(a, 0), (b, 1), \ldots\}$. This is a good example that illustrates why it is not a good idea to deviate too much from the formal way of reasoning (i.e., using the constant 1 in a formula).

For brevity, we will not cover here the actual implementations of these three functions, as they only consist of pointer operations and do not present any interesting particularities. However, they can be consulted in [17].

Remark 5. *A very important note is that, after applying any of these operations on an AST, it needs to be trimmed to preserve the invariants, as Example 9 shows.*

Example 9. *Below are listed three examples of formulas that need to be trimmed after applying factorization, absorption, and distribution, respectively.*

- $((a \wedge b) \vee (a \wedge c)) \equiv ((a \wedge (b \vee c))) \equiv (a \wedge (b \vee c))$
- $(a \vee (a \wedge b)) \equiv (a) \equiv a$
- $((a \wedge b) \vee (a \wedge (b \vee c))) \equiv ((a \wedge b) \vee (((a \wedge b) \vee (a \wedge c)))) \equiv ((a \wedge b) \vee ((a \wedge b) \vee (a \wedge c))) \equiv ((a \wedge b) \vee (a \wedge b) \vee (a \wedge c)) \equiv ((a \wedge b) \vee (a \wedge c))$

One may assume that factorization and absorption always decrease the cost of the formula, whilst distribution always increases it. Nevertheless, these properties should be studied more thoroughly. We do this in the following four theorems. As a result of the last one, the assumption regarding distribution proves to be wrong.

Theorem 1. *Transforming a tree T into T' by trimming assures that $c(T') \leq c(T)$.*

Proof. It is clear that any trimming of T consists of successively applying transformations from Figure 2.3 to various nodes of T. Thus, we only need to show that no such transformation can *increase* the cost of T on its own.

If we apply the first one, from (φ_1) to φ_1, then $c(T') = c(T) = c(\varphi_1)$. If we apply the second one, from $((\varphi_1) \oplus \varphi_2)$ to $(\varphi_1 \oplus \varphi_2)$, then $c(T') = c(T) = c(\varphi_1) + c(\varphi_2)$. Lastly, if we apply the third transformation, from $(\varphi_1 \oplus \varphi_1 \oplus \varphi_2)$ to $(\varphi_1 \oplus \varphi_2)$, then $c(\varphi_1) + c(\varphi_2) = c(T') < c(T) = 2 \cdot c(\varphi_1) + c(\varphi_2)$. □

Theorem 2. *Applying a factorization to a tree T, obtaining T', and then transforming it into T'' by trimming, assures that $c(T'') < c(T)$.*

Proof. Let $(\varphi \otimes \varphi_1) \oplus \cdots \oplus (\varphi \otimes \varphi_n)$ be the formula of the subtree$^+$ where the factorization was applied. It got replaced with a subtree having formula $\psi \triangleq (\varphi \otimes (\varphi_1 \oplus \cdots \oplus \varphi_n))$. Thus,

$$\begin{aligned} c(T') - c(T) &= c(\psi) - c(\varphi) \\ &= (c(\varphi) + (c(\varphi_1) + \cdots + c(\varphi_n))) \\ &- ((c(\varphi) + c(\varphi_1)) + \cdots + (c(\varphi) + c(\varphi_n))) \\ &= (1 - n) \cdot c(\varphi) \\ &\leq -c(\varphi) \\ &< 0. \end{aligned}$$

Combining this with Theorem 1, we get that $c(T'') \leq c(T') < c(T)$. □

Theorem 3. *Applying an absorption to a tree T, obtaining T', and then transforming it into T'' by trimming, assures that $c(T'') < c(T)$.*

Proof. Let $\varphi \oplus (\varphi \otimes \varphi_1) \oplus \cdots \oplus (\varphi \otimes \varphi_n)$ be the formula of the subtree$^+$ where the absorption was applied. It got replaced with a subtree having formula φ. Thus,

$$\begin{aligned} c(T') - c(T) &= c(\psi) - c(\varphi) \\ &= (c(\varphi)) - (c(\varphi) + (c(\varphi) + c(\varphi_1)) + \cdots + (c(\varphi) + c(\varphi_n))) \\ &= -n \cdot c(\varphi) - c(\varphi_1) - \cdots - c(\varphi_n) \\ &< 0. \end{aligned}$$

Combining this with Theorem 1, we get that $c(T'') \leq c(T') < c(T)$. □

Theorem 4. *Applying a distribution to a tree T, obtaining T', and then transforming it into T'' by trimming, does not assure that $c(T'') > c(T)$.*

Proof. The intuition is that $c(T') > c(T)$ (which is indeed true), but the final trimming reduces the cost by more than $c(T') - c(T)$, making $c(T'') < c(T)$. An example that illustrates this is

$$\begin{array}{rclcrcl} \varphi & = & ((a \wedge b) \vee (a \wedge (b \vee c))) & \Longrightarrow & c(\varphi) & = & 5 \\ \varphi' & = & ((a \wedge b) \vee (((a \wedge b) \vee (a \wedge c)))) & \Longrightarrow & c(\varphi') & = & 6 \\ \varphi'' & = & ((a \wedge b) \vee (a \wedge c)) & \Longrightarrow & c(\varphi'') & = & 4, \end{array}$$

where φ, φ', and φ'' are the formulas of T, T', and T'', respectively. □

2.3 The Proposed Heuristics

This section offers a summary of the four heuristics proposed. Beforehand, it is worth explaining that "trying to increase (or decrease) $c(T)$" means searching in T for every context[2] where a factorization or an absorption (or a distribution) can be applied, and then randomly choosing one such context and applying the operation to it. This process is considered to fail only when there were no contexts found.

2.3.1 Naïve

The Naïve approach — it cannot really be called a heuristic — randomly tries to apply factorizations and absorptions until there are no available contexts anymore. It does not care how good the improvement of a given transformation is, nor what future optimization potential it offers. The pseudocode for this approach is available in Algorithm 2.

Algorithm 2 Naïve

Require: The AST T.

```
1 while trying to decrease c(T) succeeded do
2     nothing
3 end while
```

[2] The context needed to perform, for instance, a factorization, is the set of nodes having φ in $(\varphi \otimes \varphi_1) \oplus (\varphi \otimes \varphi_2) \oplus \cdots \oplus (\varphi \otimes \varphi_n)$, since — generally having access to the parent of any given node — they are enough to let us perform the required operation.

2.3.2 Hill Climbing

In its general form, the Hill Climbing heuristic views the optimization problem as a directed graph whose nodes are problem states (in our case, ASTs) and whose edges are transitions from one state to another (in our case, factorizations and absorptions). The algorithm starts from the initial state (in our case, the given AST T) and moves to a *better* neighboring state (in our case, some AST T' such that $c(T') < c(T)$). The process continues until a terminal state is reached (i.e., one with no more neighbors).

Our algorithm does not explore every single neighbor of the current state, since their number can be very large. Instead, it randomly chooses only k of them (i.e., k random contexts where the operation can be applied) and picks up one having the minimum cost. The pseudocode for this approach is available in Algorithm 3.

Algorithm 3 Hill Climbing

Require: The AST T.

```
1  while true do
2      c_min ← ∞
3      T_min ← null
4      for i = 1, k do
5          let T' be a clone of T
6          if trying to decrease c(T') failed then        ▷ no more neighbors of T
7              exit
8          end if
9          c ← c(T')
10         if c < c_min then
11             c_min ← c
12             T_min ← T'
13         end if
14     end for
15     T ← T_min
16 end while
```

2.3.3 Simulated Annealing

One downside of Hill Climbing is that, since it greedily chooses the best transition at each step, it can get stuck in a local minimum. That is where Simulated Annealing [19] comes in handy, for it sometimes chooses a worse state than the current one. In other words, in our case, it also uses distributions.

Simulated Annealing is inspired by the real process of physical annealing. The latter consists of heating up a material (usually a metal) until it reaches a specific temperature, after which it will be cooled down slowly, with the goal of changing its structure to a desired one. When the material is hot, its molecular structure is weaker, thus being more susceptible to change. However, when it cools down, its structure becomes harder and more resistant to external forces.

Simulated Annealing requires the following parameters:

- the initial temperature $t_{\max}$;
- the target temperature $t_{\min}$;
- the cooling rate β;
- the probability of choosing a worse state than the current one p;
- the number of iterations of each step n.

After each temperature change, we repeat n times the process of generating a neighbor T' of T, the transition being chosen according to p. If $c(T') < c(T)$, then T' is accepted (i.e., we move into its state). Otherwise, T' is accepted with probability $e^{-\Delta/t}$, where $\Delta \triangleq c(T') - c(T)$ and t is the current temperature. The way temperature changes from one step to the next is $t \mapsto t/(1 + \beta \cdot t)$. Figure 2.5 illustrates its decrease over time.

At the end of the algorithm, we apply all the remaining factorizations and absorptions. Our implementation of Simulated Annealing is shown in Algorithm 4.

Algorithm 4 Simulated Annealing

Require: The AST T.

```
t ← t_max
while t > t_min do
    for i ∈ 1, n do
        let T' be a clone of T
        c_1 ← c(T')
        if rand(0, 1) < p then
            try to increase c(T')
        else
            try to decrease c(T')
        end if
        c_2 ← c(T')
        Δ ← (c_2 − c_1)/c_1                          ▷ Δ as percentage
        if Δ < 0 or rand(0, 1) < e^{−Δ/t} then
            T ← T'
        end if
    end for
    t ← t/(1 + β · t)                                ▷ cooling t
end while
while trying to decrease c(T) succeeded do
    nothing
end while
```

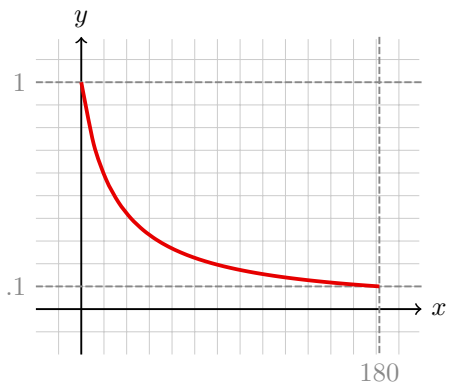


Figure 2.5: Plot of the temperature function for $t_{\min} = .1$, $t_{\max} = 1$, and $\beta = .05$. The temperature needs 180 steps to reach $t_{\min}$. It can be proven that the function that maps every step $i \geq 0$ to its corresponding temperature is $i \mapsto t_{\max}/(1 + i \cdot \beta \cdot t_{\max})$.

2.3.4 Custom Heuristic

We also developed a Custom Heuristic — some sort of a lighter version of Simulated Annealing. The idea is still to apply distributions more and more rarely. However, it keeps track of the moments when no more factorizations or absorptions could be made, in order to forcefully make a distribution on the next step. Custom Heuristic makes a total of n steps and uses a special constant α to adjust the probability of applying a distribution. Its implementation can be consulted in Algorithm 5.

Algorithm 5 Custom Heuristic

Require: The AST T.

```
1  factorizable ← true
2  for i ∈ 1, n do
3      if not factorizable or rand(1, α · n) < n − i then
4          try to increase c(T)
5          factorizable ← true
6      else if trying to decrease c(T) failed then
7          factorizable ← false
8      end if
9  end for
10 while trying to decrease c(T) succeeded do
11     nothing
12 end while
```

Chapter 3

Testing Our Main Approach

In this chapter, we shall present the results of our work. Specifically, we will talk about how we generated the test data, how we tested and fine-tuned our proposed heuristics, and what conclusions we can draw from these results. Finally, we will describe our open-source Python library for creating datasets and testing our heuristics — and those proposed by anybody else! — on them.

3.1 Generating Test Data

We generated two kinds of datasets. Compared to the second one, the first kind is more relevant to the general problem of optimizing monotone Boolean formulas than it is to improving ABE systems.

The first kind consists of three datasets, namely `small`, `medium`, and `large`, named according to the average size of the formulas they contain. The second kind consists of just one dataset, written manually, containing formulas that model a specific access policy that arises in practical ABE systems.

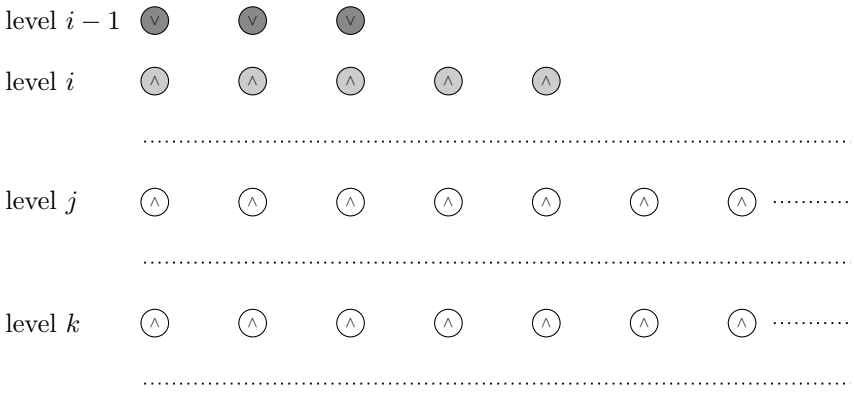
Table 3.1 lists the parameters used in generating the datasets. Let us denote the variable count by V, the maximum degree by D, and the literal count range by $[L_1, L_2]$. Note that instead of generating the formulas directly, we focused on their ASTs, which we later converted to Boolean formulas.

3.1.1 Monotone Boolean Formulas

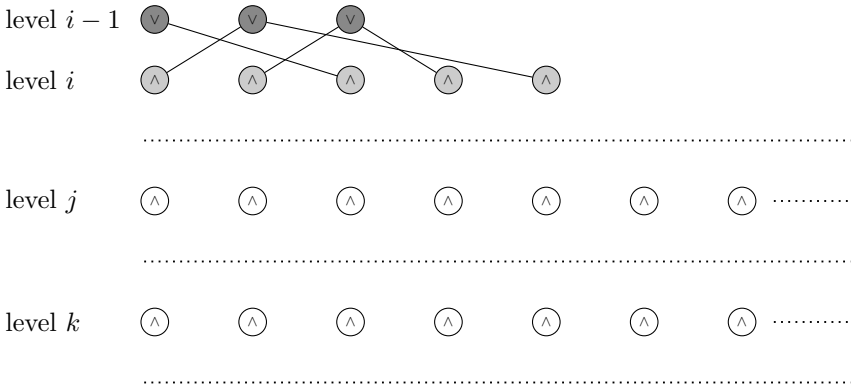
We start by creating the last level of the tree. Initially, it consists of one node for each of the V required variables; let them be a, b, c, etc. We then build the tree one level at a time. We stop when the last generated level has only one node. This node will become the root of the AST.

Dataset	Variable Count	Max. Degree	Literal Count	Formula Count
`small`	20–25	2–5	50–75	25
`medium`	26–30	2–10	100–150	25
`large`	31–40	5–10	200–300	25
`real`	14–24	4–7	42–120	7

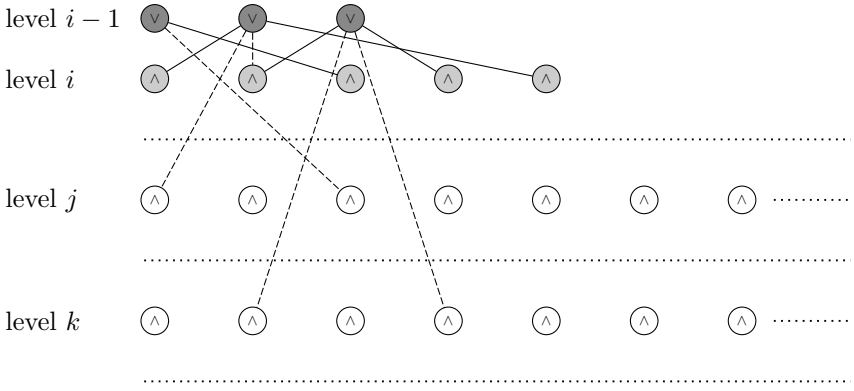
Table 3.1: Parameters used in generating the datasets: the number of variables in the formula, the maximum degree of the AST (i.e., the most children a node is allowed to have), the number of literals in the formula (i.e., the number of leaves in the AST, which is equivalent to the cost of the formula), and the number of formulas to generate, respectively.



(a) Creating $\lceil 5/2 \rceil = 3$ nodes on level $i-1$.



(b) Choosing some node from level $i-1$ as parent for each node of level i.



(c) Connecting nodes on level $i-1$ with fresh copies of random nodes from previous i, j, and k levels, where $i < j < k$ have the same parity.

Figure 3.1: Constructing level $i-1$ of the randomly generated AST in three steps, based on the previously built levels, while making sure that no node will have more children than required (let it be 4 in this case). The $\vee$ and $\wedge$ operators are to be used interchangeably.

Let i be the previous level. Thus, the current level (i.e., the one above it) is $i-1$. When created, the nodes on level $i-1$ should be assigned the opposite operator to the one on level i (or a random operator if this is the first step). If level i contains n nodes, then level $i-1$ should initially consist of $\lceil n/2 \rceil$ nodes (see Figure 3.1a). Their role is to become parents of the nodes on level i.

Indeed, the next step is to iterate over each node v of level i, choose a random node u on level $i-1$, and make u the parent of v (see Figure 3.1b). When choosing u, we should pay attention not to choose a node that already has D children, in order not to break the maximum degree requirement. The most restrictive case regarding how many children some node is able to have is $D = 2$, hence our $\lceil n/2 \rceil$ choice for the initial size of level $i-1$.

After that, we iterate over each node u of level $i-1$ and choose some random nodes from previous levels to add to its children, while making sure again they will not count more than D (see Figure 3.1c). We can choose nodes from the lowest level and those of the same parity as i. Thus, no child of u will have the same operator as u. Since these new nodes already have a parent, we do not connect u directly to themselves, but rather to *clones* of them. That is, we create a deep copy of their entire subtrees and make u the parent of their roots.

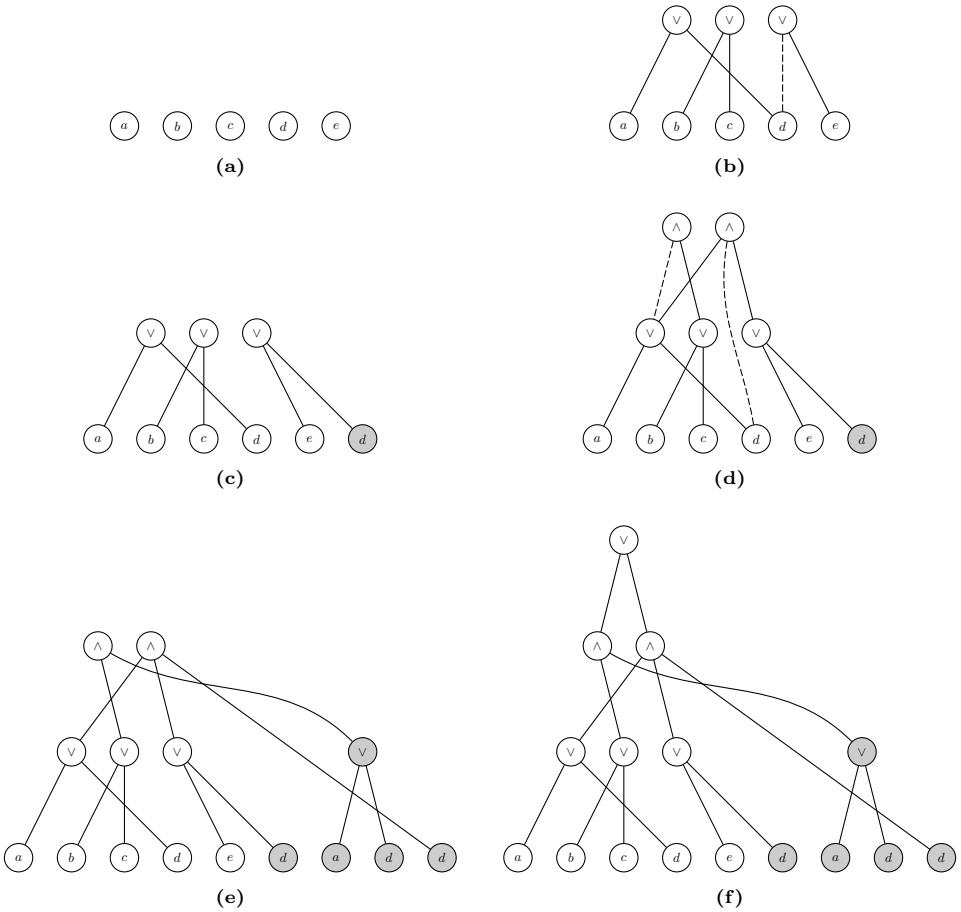


Figure 3.2: A concrete example of randomly generating an AST, leading to the formula $\varphi = (((b \vee c) \wedge (a \vee d)) \vee ((a \vee d) \wedge (e \vee d) \wedge d))$. In subfigures **(b)**, **(d)**, and **(f)** a new level is created, its nodes are connected to those on the previous level, and new conceptual edges (represented with dashed lines) are drawn to nodes that already had a parent. In subfigures **(c)** and **(e)** the latter edges materialize into real ones by cloning their bottom extremities (along with their entire subtrees). The cloned nodes are shown in gray.

Now that we described the general process of building a random AST, let us look at a concrete example, depicted in Figure 3.2. We start by filling the lowest level with 5 nodes. Then, we create 3 nodes on the next one and randomly assign them the $\vee$ operator. We appoint the parents of the nodes on the previous level (by drawing solid edges) and randomly add d as a new child to the third node on the current level. The next subfigure shows the cloning of node d, leading to the previous edge becoming solid and pointing to this new node. Afterward, we create a new level in a similar way — note that, this time, we had the chance to clone an entire subtree — and finally, we create the root of the AST.

At the very end, the AST gets trimmed (since some nodes may have only one child), and if the number of its leaves is not within $[L_1, L_2]$, then we start the generating process again.

3.1.2 Comparison Query Formulas

In designing real-world ABE systems, there is a particular type of access policy that arises very often. Let us take for example this policy that confers access to Gen X and Gen Z people born during summer:

$$(((1965 \leq y \wedge y \leq 1980) \vee y \geq 1997) \wedge (6 \leq m \wedge m \leq 8)).$$

As one may notice, a pattern always occurring in this kind of policy is the comparison between two integers, one known a priori and the other being an input.

Therefore, we should discuss how an AST for comparing an input x to a fixed value can be constructed. Such an example is illustrated in Figure 3.3, where the comparison worked on is $x \geq 45$. To better understand the structure of the AST, one can evaluate the Boolean formula on an arbitrary input. For instance, a value of $x = 41 = (00101001)_2$ would lead to the following process:

1. Is any of $x_7 = 1$ and $x_6 = 1$ true (meaning that even if $x_5 = \cdots = x_0 = 0$, we would still have $x > 45$)? No, but now we know that $x_7 = x_6 = 0$.
2. Is $x_5 = 0$ true (meaning that even if $x_4 = \cdots = x_0 = 1$, we would still have $x < 45$)? No, but now we know that $x_5 = 1$.
3. Is $x_4 = 1$ true (meaning that even if $x_3 = \cdots = x_0 = 0$, we would still have $x > 45$)? No, but now we know that $x_4 = 0$.
4. Is $x_3 = 0$ true (meaning that even if $x_2 = \cdots = x_0 = 1$, we would still have $x < 45$)? No, but now we know that $x_3 = 1$.
5. Is $x_2 = 0$ true (meaning that even if $x_1 = \cdots = x_0 = 1$, we would still have $x < 45$)? Yes, so the current conjunction is evaluated as false. Going up the AST, the entire formula becomes false, so $x \geq 45$ is false indeed.

We manually wrote the Boolean formulas of 7 random access policies that combine many comparison queries similar to the one discussed previously. Since this type of policy is so practical, we put these formulas in the `real` dataset.

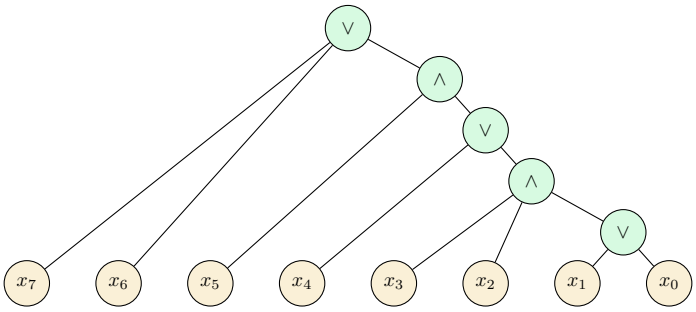


Figure 3.3: The AST corresponding to the comparison query $x \geq 45$, assuming that the input integers x are represented on 8 bits. The construction is based on the fact that $45 = (00101101)_2$. Furthermore, note that $(x_7 x_6 \cdots x_0)_2$ is the base 2 representation of the input x.

3.2 Testing Methodology

For some fixed formula φ and heuristic $\mathcal{H}$, we ran $\mathcal{H}(\varphi)$ a number of $n = 5$ times. We then computed the average and maximum values of the costs of the resulting ASTs, as well as the average running time of one iteration of $\mathcal{H}(\varphi)$. Letting $\mathcal{H}$ fixed, these values were finally averaged over all the formulas in each dataset, leading to the results in Table 3.2.

An important remark is the actual role of the "Max. Score" column. For the way it is computed, it tells us a way of deriving a new heuristic $\mathcal{H}^*$ from $\mathcal{H}$; let us call it "Iterated $\mathcal{H}$." Indeed, $\mathcal{H}^*$ just makes n calls to $\mathcal{H}$ and takes the best AST obtained. Thus, the only missing information in Table 3.2 is the average running time of $\mathcal{H}^*$, but it can easily be computed by multiplying the value in the "Avg. Time" column by $n = 5$.

It is also relevant to show the hyperparameters used during the testing process. For Hill Climbing we used $k = 10$. For Simulated Annealing we used $t_{\min} = .1$, $t_{\max} = 1$, $\beta = .05$, $p = .2$, and $n = 10$. Finally, for Custom Heuristic we used $\alpha = 5$ and $n = 150$. Needless to say, the Naïve approach does not use any hyperparameters.

3.3 Analyzing the Results

One can easily see from Table 3.2 that, when iterated, Simulated Annealing beats every other heuristic. However, this comes at a cost; that is, its running time. Nevertheless, we experimented quite a bit with its hyperparameters to obtain the best improvement over time ratio. During this experiments, we even got a 90% score for the `large` dataset and 70% for `real`, but they were not considered relevant since it took way too much time for just one iteration to complete; that is, roughly 10 seconds.

From now on, we shall split our discussion in two, namely analyzing the last dataset separately from the other three. Considering how the datasets were generated, the results obtained for the first three of them are more relevant to the general problem of optimizing monotone Boolean formulas than to improving ABE systems.

For the first datasets, Iterated Simulated Annealing is still the best choice, but again its running time is very large, and it also grows rapidly with the size of the formula. Leaving aside the iterated heuristics, we can conclude that Hill Climbing is the best choice, even if for the `small` dataset it performs slightly worse than Simulated Annealing (but at a much lower cost timewise). If we care even more about the running time, then the Naïve algorithm is a solid choice, being 8.5 times faster than Hill Climbing for the `large` dataset.

	Heuristic	Avg. Score (%)	Max. Score (%)	Avg. Time (s)
small	Naïve	25.65	27.21	0.00
	Hill Climbing	27.16	27.54	0.02
	Simulated Annealing	28.77	38.51	0.57
	Custom Heuristic	19.61	35.46	0.07
medium	Naïve	33.15	35.92	0.01
	Hill Climbing	35.89	36.66	0.09
	Simulated Annealing	22.55	40.73	1.32
	Custom Heuristic	19.11	37.07	0.15
large	Naïve	46.81	50.56	0.04
	Hill Climbing	53.40	54.19	0.34
	Simulated Annealing	38.31	58.42	2.60
	Custom Heuristic	25.32	51.30	0.41
real	Naïve	3.78	5.31	0.00
	Hill Climbing	5.31	5.31	0.01
	Simulated Annealing	39.73	60.65	0.89
	Custom Heuristic	13.38	32.58	0.11

Table 3.2: The results for each dataset and each heuristic when iterating it 5 times over each formula: the average and maximum cost improvements (in percentages) over all iterations and the average running time (in seconds) of one iteration. The best values for each dataset and each score type are colored red.

For the last dataset, Simulated Annealing is by far the best heuristic, both iterated and non-iterated. Furthermore, its non-iterated version has a decently reasonable running time of less than a second, making it a good enough choice. If this amount of time is too large, then Iterated Custom Heuristic comes into play, for it provides approximately 30% improvement for only half a second. What is more, `real` is the only dataset where the Naïve and Hill Climbing approaches are almost useless, regardless of their version.

In conclusion, Iterated Simulated Annealing always yields the best result, but at a very high cost, whilst Hill Climbing and Iterated Custom Heuristic — depending on the purpose of the input formula — ensure some very good improvements for a way lower running time. However, for a fixed set of formulas, one is advised to experiment with different hyperparameter values to achieve the best results for that particular situation. In this regard, we can state that Simulated Annealing is the most flexible algorithm.

3.4 Open-Source Python Library

As a result of our work on optimizing monotone Boolean formulas, a library for dataset generating and testing naturally emerged. Its first version was written in C++ and is available at [20]. Over time, we realized it lacked some particular features, and therefore we created a second version, available at [17].

Compared to the initial one, the new version is written in Python and supports generalized AST operations (they were designed to work only with $\vee$ as a top-level operator and with only two subformulas[1] at a time), the code is clean and well-documented, the logic of the formula generator is more natural, and the provided Command-Line Interface is way more flexible.

3.4.1 Project Structure

The most important file in the Python project is `tree.py`, defining methods that operate on a `Tree` instance, such as `trim`, `clone`, and `cost`, but also static methods related to ASTs: `parse` (for converting a formula given as a string to an AST), `random` (for generating a random AST according to given parameters), and `probably_equivalent` (for probabilistically testing[2] whether two ASTs represent equivalent Boolean formulas; it was used during the unit testing process).

The next most important file is `operations.py`, which for each operation $\mathcal{F}$ (either factorization, absorption, or distribution) contains two functions: one that finds every context of the AST where $\mathcal{F}$ can be applied and one that applies $\mathcal{F}$ on the provided context. Also, there is a function that tries to decrease the cost of the AST (by randomly applying a factorization or an absorption), as well as one that tries to increase it (by randomly applying a distribution).

This file is followed by `heuristics.py`, which defines the four heuristics, as well as a function that transforms a given heuristic into its iterated version. Note that everyone who uses this library can define in this file their own heuristic without needing to do any other setup.

The last relevant files are `main.py` and `generator.py`, used to implement the core features of the CLI, and lastly, `tests.py`, used for running unit tests on the entire project. As a side note, the CLI was made using the Rich [21] and Questionary [22] libraries.

3.4.2 Usage and Public Datasets

Instructions on how to install and use the library are available in the `README.md` file in the repository at [17]. However, we mention here what the CLI can accomplish. Indeed, `main.py` will request the user to prompt it with the group of datasets they want to run tests on, followed by requiring them to select what heuristics they want to be used. Meanwhile, `generator.py` provides an interface for generating new datasets based on the hyperparameters provided by the user.

One of the most important things about this library is that the `inputs/new` directory provides the datasets we used in testing our heuristics. Therefore, anybody can compare the efficiency of their approach to ours. The `inputs` directory additionally contains an `old` subdirectory with the datasets used in [20]. The reason we needed to create new datasets is that, when using the new generator with the old hyperparameters, we got worse results than when we used the old generator, hence the latter, alongside the tests it created, had some structural flaws.

[1] Working with only two subformulas means that when talking about, for instance, factorization, $(x \wedge \varphi_1) \vee (x \wedge \varphi_2) \vee (x \wedge \varphi_3)$ can be replaced by $(x \wedge (\varphi_1 \vee \varphi_2)) \vee (x \wedge \varphi_3)$, but not by $(x \wedge (\varphi_1 \vee \varphi_2 \vee \varphi_3))$.

[2] The test consists of assigning random values to the variables involved in the formulas associated with the ASTs and evaluating the latter on these inputs. The described algorithm is iterated at most 1 000 times. If, at any step, it encounters a mismatch between the two resulting truth values, it concludes that the formulas are certainly not equivalent. Otherwise, they probably are.

Chapter 4

Subcircuit Replacement Approach

In this chapter, we describe a new approach for optimizing Boolean formulas, which, unlike the previous one, does not operate on the AST associated with the formula, but rather on its corresponding Boolean circuit (i.e., the actual representation of the ABE policy). Briefly, this technique is about searching for certain subcircuit patterns inside the given circuit and replacing them with lower-cost equivalent subcircuits.

Let $\mathcal{C}$ be the Boolean circuit associated with the given Boolean formula φ. Let ξ be the set of "pattern-replacement" pairs. Namely, $\xi \triangleq \{(\mathcal{P}_i, \mathcal{R}_i) \mid 1 \leq i \leq n\}$, where each $\mathcal{P}_i$ is a subcircuit to be searched in $\mathcal{C}$ and $\mathcal{R}_i$ is a subcircuit for each $\mathcal{P}_i$ occurrence to be replaced with.

4.1 Circuit Structure

For the beginning, let us formally define the circuit $\mathcal{C}$ as a Directed Acyclic Graph (V, E), where V is the set of *nodes* and $E \subseteq V \times V$ is the set of *edges*. Also, let $N_u \triangleq \{v \mid \exists (v, u) \in E\}$ be the children set of u.

Intuitively, each node $u \in V$ is associated with a subformula $\gamma(u)$ of φ. In particular, the root corresponds to the entire φ. The values of γ can be defined recursively in the following manner:

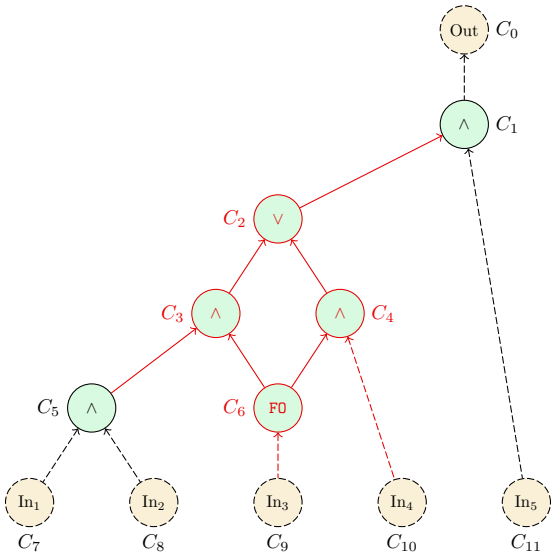
$$\gamma(u) \triangleq \begin{cases} \bigwedge_{v \in N_u} \gamma(v) & \text{for } \text{type}(u) = \texttt{AND} \\ \bigvee_{v \in N_u} \gamma(v) & \text{for } \text{type}(u) = \texttt{OR} \\ \gamma(v) : v \in N_u & \text{for } \text{type}(u) \in \{\texttt{FO}, \texttt{OUTPUT}\} \\ \text{variable}(u) & \text{for } \text{type}(u) = \texttt{INPUT}. \end{cases}$$

Of course, the base case for the recursion is the last one, namely $\text{type}(u) = \texttt{INPUT}$, where u corresponds to a specific variable of φ.

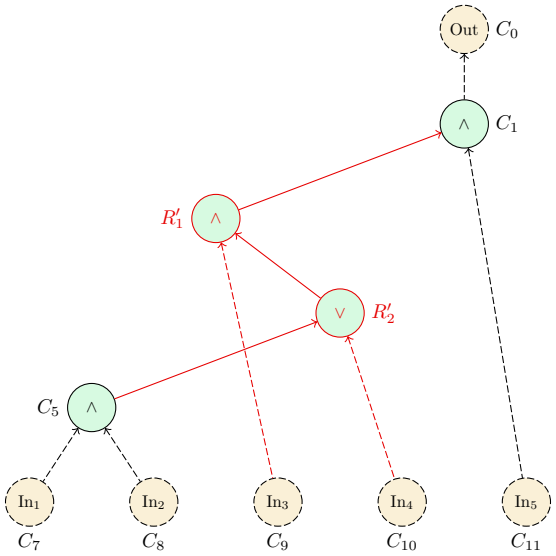
As one can see, any non-terminal node u is either of type `AND`/`OR` or `FO`. If its type is `AND`/`OR`, then u is limited to be the child of exactly one node, while if its type is `FO`, then u is limited to have exactly one child. This duality guarantees that there can be no two different nodes $u, v \in V$ such that $\gamma(u) = \gamma(v)$.

Figure 4.1a shows an example of a Boolean circuit. The subformulas of φ corresponding to each internal node are:

$$\gamma(C_5) = (\mathrm{In}_1 \wedge \mathrm{In}_2)$$
$$\gamma(C_6) = \mathrm{In}_3$$
$$\gamma(C_3) = ((\mathrm{In}_1 \wedge \mathrm{In}_2) \wedge \mathrm{In}_3)$$
$$\gamma(C_4) = (\mathrm{In}_3 \wedge \mathrm{In}_4)$$
$$\gamma(C_2) = (((\mathrm{In}_1 \wedge \mathrm{In}_2) \wedge \mathrm{In}_3) \vee (\mathrm{In}_3 \wedge \mathrm{In}_4))$$
$$\gamma(C_1) = ((((\mathrm{In}_1 \wedge \mathrm{In}_2) \wedge \mathrm{In}_3) \vee (\mathrm{In}_3 \wedge \mathrm{In}_4)) \wedge \mathrm{In}_5).$$



(a) A circuit $\mathcal{C}$ and the occurrence $\mathcal{P}'$ of $\mathcal{P}$ in it (colored red), where $\mathcal{P}$ is the subcircuit in Figure 1.1a. The formula corresponding to $\mathcal{C}$ is $((((\mathrm{In}_1 \wedge \mathrm{In}_2) \wedge \mathrm{In}_3) \vee (\mathrm{In}_3 \wedge \mathrm{In}_4)) \wedge \mathrm{In}_5)$.



(b) Circuit $\mathcal{C}$ after replacing $\mathcal{P}'$ with a copy $\mathcal{R}'$ of $\mathcal{R}$ (colored red), where $\mathcal{R}$ is the subcircuit in Figure 1.1b. The formula corresponding to $\mathcal{C}$ becomes $((((\mathrm{In}_1 \wedge \mathrm{In}_2) \vee \mathrm{In}_4) \wedge \mathrm{In}_3) \wedge \mathrm{In}_5)$.

Figure 4.1: The process of replacing the occurrence of $\mathcal{P}$ in $\mathcal{C}$ with a copy of $\mathcal{R}$.

4.2 Main Idea

We randomly select a pair $(\mathcal{P}, \mathcal{R}) \in \xi$ and we search for an occurrence $\mathcal{P}'$ of $\mathcal{P}$ in $\mathcal{C}$. We then replace $\mathcal{P}'$ with a copy of $\mathcal{R}$. We repeat this process until we have not found the wanted pattern $k \triangleq 2 \cdot |\xi|$ times in a row. As an example, let $\mathcal{C}$ be the circuit in Figure 4.1a and $\mathcal{P}$ and $\mathcal{R}$ be the subcircuits in Figure 1.1 (without the terminal nodes, but alongside their corresponding wires, conceptually).

Now it is time to define what a *subcircuit* actually is. Formally, we can define a subcircuit $\mathcal{S}$ as a triple $(\mathcal{C}, L, U)$, where $\mathcal{C}$ is the *base circuit*, L is the array of *lower nodes* and U is the array of *upper nodes*. These nodes inside L and U are places where we can attach other nodes in order to form a proper circuit. We will denote the nodes that get attached to each lower node u_i by $\lambda_L^{\mathcal{S}}(i)$. Likewise, each upper node v_i gets linked to $\lambda_U^{\mathcal{S}}(i)$. Note that the nodes $\lambda_L^{\mathcal{S}}(i)$ do not need to be all distinct. Also, any node $u \in L$ can be linked to multiple nodes; that is, one for each occurrence of u in L. The same goes for the nodes in U, even though most of the time $|U| = 1$.

Going back to our example in Figure 1.1, we see that $\mathcal{P} = (\mathcal{C}^{\mathcal{P}}, \langle P_2, P_4, P_3 \rangle, \langle P_1 \rangle)$ and $\mathcal{R} = (\mathcal{C}^{\mathcal{R}}, \langle R_2, R_1, R_2 \rangle, \langle R_1 \rangle)$. If we create three new nodes X, Y, and Z and attach them to the lower nodes of $\mathcal{P}$ (i.e., we set $\lambda_L^{\mathcal{P}} = \{1 \to X, 2 \to Y, 3 \to Z\}$) we obtain $\gamma(P_1) = ((\gamma(X) \wedge \gamma(Y)) \vee (\gamma(Y) \wedge \gamma(Z)))$. Similarly, if we were to attach X, Y, and Z to the lower nodes of $\mathcal{R}$ instead (i.e., setting $\lambda_L^{\mathcal{R}} = \{1 \to X, 2 \to Y, 3 \to Z\}$) we would have obtained $\gamma(R_1) = ((\gamma(X) \vee \gamma(Z)) \wedge \gamma(Y))$.

We can easily see that the Boolean formulas of $\mathcal{P}$ and $\mathcal{R}$ are equivalent. Thus, we can replace occurrences of $\mathcal{P}$ in $\mathcal{C}$ with copies of $\mathcal{R}$. We shall detail in the next two sections what this process looks like and how it behaves in our example.

4.3 Pattern Searching

Formally, searching for subcircuit $\mathcal{P}$ in circuit $\mathcal{C}$ means finding, firstly, a circuit $\mathcal{C}^{\mathcal{P}'}$ such that $\mathcal{C}^{\mathcal{P}'}$ is a subgraph of $\mathcal{C}$ and is isomorphic to $\mathcal{C}^{\mathcal{P}}$.

Let $f(u)$ be the correspondent of node $u \in \mathcal{C}^{\mathcal{P}}$ in $\mathcal{C}^{\mathcal{P}'}$. In this context, *isomorphic* extends its classical graph definition with the fact that $\text{type}(u) = \text{type}(f(u))$ must hold for every node $u \in \mathcal{C}^{\mathcal{P}}$.

The next thing that needs to be found is the function $\lambda_L^{\mathcal{P}'}$ that, for every lower node u of $\mathcal{P}$, links $f(u)$ to a node in $V(\mathcal{C}) \setminus V(\mathcal{C}^{\mathcal{P}'})$. Likewise, we also need to find the function $\lambda_U^{\mathcal{P}'}$ for the upper nodes.

Now that we have $\lambda_L^{\mathcal{P}'}$ and $\lambda_U^{\mathcal{P}'}$ defined, we can state the last condition for $\mathcal{P}'$ to be a match of $\mathcal{P}$. Indeed, no node $u \in \mathcal{C}^{\mathcal{P}'}$ should have an edge to/from any node $v \notin V(\mathcal{C}^{\mathcal{P}'}) \cup \{\lambda_L^{\mathcal{P}'}(f(x)) \mid x \in L(\mathcal{P})\} \cup \{\lambda_U^{\mathcal{P}'}(f(x)) \mid x \in U(\mathcal{P})\}$.

The match $\mathcal{P}' \triangleq (\mathcal{C}^{\mathcal{P}'}, L, U)$ of $\mathcal{P}$ in $\mathcal{C}$ is colored red in Figure 4.1a. Therefore, one can see that $\mathcal{C}^{\mathcal{P}'}$ is the subgraph induced by $\{f(P_1), f(P_2), f(P_3), f(P_4)\} = \{C_2, C_3, C_4, C_6\}$ in $\mathcal{C}$, $L = \langle C_3, C_6, C_4 \rangle$, $U = \langle C_2 \rangle$, $\lambda_L^{\mathcal{P}'} = \{1 \to C_5, 2 \to C_9, 3 \to C_{10}\}$, and $\lambda_U^{\mathcal{P}'} = \{1 \to C_1\}$.

The process itself (i.e., of finding a match for $\mathcal{P}$ in $\mathcal{C}$) is quite straightforward. It is a backtracking algorithm that involves generating arrangements of length $|V(\mathcal{C}^{\mathcal{P}})|$ for $V(\mathcal{C})$. We immediately discard nodes that do not satisfy the above criteria, by looking at their type and in/out degree.

4.4 Subcircuit Replacement

Conceptually, the process of replacing $\mathcal{P}'$ with a copy $\mathcal{R}'$ of $\mathcal{R}$ in $\mathcal{C}$ involves setting the entire functions $\lambda_L^{\mathcal{R}'}$ and $\lambda_U^{\mathcal{R}'}$ to $\lambda_L^{\mathcal{P}'}$ and $\lambda_U^{\mathcal{P}'}$, respectively. This way, we link the lower and the upper nodes of $\mathcal{R}'$ to the appropriate nodes in $\mathcal{C}$ (i.e., the ones that were linked to $\mathcal{P}'$ accordingly). Of course, after this operation, we should also delete the nodes of $\mathcal{C}^{\mathcal{P}'}$, along with their edges. Figure 4.1b shows $\mathcal{C}$ at the end of the replacement described in our example.

You may notice that the order we link the lower nodes of $\mathcal{R}'$ to actual nodes of $\mathcal{C}$ is crucial. That is, the arrays $L(\mathcal{P})$ and $L(\mathcal{R})$ should be given in such orders that, after a replacement, the old formula $\gamma(\mathcal{C})$ remains equivalent to the new one. For instance, if we were given $L(\mathcal{R}) = \langle R_1, R_2, R_2 \rangle$, then we would have obtained $\gamma(R_1') = (\gamma(\lambda_L^{\mathcal{P}'}(1)) \wedge (\gamma(\lambda_L^{\mathcal{P}'}(2)) \vee \gamma(\lambda_L^{\mathcal{P}'}(3)))) = (\gamma(C_5) \wedge (\gamma(C_9) \vee \gamma(C_{10})))$, which is of course very different from $(\gamma(C_9) \wedge (\gamma(C_5) \vee \gamma(C_{10})))$.

4.5 Method Drawback

One big problem of this approach is the fact that searching for subcircuits (i.e., finding subgraphs isomorphic to some other graph in a larger graph) is a computationally hard problem. Therefore, this solution was not suitable for the heuristics proposed in the second chapter.

Chapter 5

Conclusions

We proposed multiple heuristics for optimizing monotone Boolean formulas. As the tests we ran reveal, they provide a considerable improvement in circuit size for the KP-ABE scheme described by Țiplea and Drăgan [3]. These optimizers can be called at the beginning of the *key generation* phase of the encryption scheme, in order to replace the circuit for which the keys are generated with a better one. This leads to a little higher setup time, but, more importantly, to smaller decryption keys and decryption times. Our optimization strategy can have a significant impact on cloud systems that implement fine-grained cryptographic access control over data through ABE schemes. For example, for the price of less than a second overhead in the key generation phase, Simulated Annealing cuts the decryption time in half.

We wrote an open-source Python library [17] which is publicly available for anyone to use. Its purpose is the testing of various heuristics and comparing the results on the same datasets as we used. What is more, it provides a CLI that can be used directly to generate lower-cost equivalent formulas for those in the input.

We already sent an article [18] on this research to the KES conference, which was accepted, but this paper coveres the subject in much more detail. Also, as shown two chapters ago, the new library presents some relevant improvements over the C++ one [20].

5.1 Further Work

There is still room for improvement, especially for the second approach, which yielded zero improvements for any non-trivial circuit. Nevertheless, we believe it can be adjusted to obtain good results if we use cryptographic strategies to improve the secret sharing of subcircuits, instead of just relying on the logical equivalence of their formulas.

Specifically, we can replace certain subcircuits with structures containing Compartmented Access Structure (CAS) nodes, as shown by Ioniță [23]. Broadly, a CAS node is a node divided into k disjoint compartments, each of them having its own threshold $0 \leq t_i \leq n_i$, such that $t_1 + \cdots + t_k \leq t \leq n = n_1 + \cdots + n_k$, where t is the threshold of the CAS node itself, n is its number of children, and n_i is the number of children of compartment i. The CAS nodes can be used in scenarios where the circuit contains two sibling nodes such that their parent is an `AND` gate and the children set of one is a subset of the children set of the other.

This remains an interesting problem we will study further.

Bibliography

[1] Y. Zhang, R.H. Deng, S. Xu, J. Sun, and Q. Li. Attribute-Based Encryption for Cloud Computing Access Control: A Survey. Research Collection School Of Computing and Information Systems, 2020.

[2] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 89–98, 2006.

[3] F.L. Țiplea and C.C. Drăgan. Key-Policy Attribute-Based Encryption for Boolean Circuits From Bilinear Maps. In *International Conference on Cryptography and Information Security in the Balkans*, pages 175–193. Springer, 2014.

[4] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-Policy Attribute-Based Encryption. In *2007 IEEE Symposium on Security and Privacy*, pages 321–334. IEEE, 2007.

[5] S. Garg, C. Gentry, S. Halevi, A. Sahai, and B. Waters. Attribute-Based Encryption for Circuits From Multilinear Maps. In *Annual Cryptology Conference*, pages 479–499. Springer, 2013.

[6] M. Albrecht and A. Davidson. Are Graded Encoding Schemes Broken Yet?, 2017.

[7] F.L. Țiplea. Multilinear Maps in Cryptography. In *Conference on Mathematical Foundations of Informatics*, pages 241–258, 2018.

[8] P. Hu and H. Gao. A Key-Policy Attribute-Based Encryption Scheme for General Circuit From Bilinear Maps. *IJ Network Security*, 19(5):704–710, 2017.

[9] W.V. Quine. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[10] E.J. McCluskey. Minimization of Boolean Functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[11] R.K. Brayton, G.D. Hachtel, L.A. Hemachandra, A.R. Newton, and A.L.M. Sangiovanni-Vincentelli. A Comparison of Logic Minimization Strategies Using Espresso: An APL Program Package for Partitioned Logic Minimization. In *Proceedings of the International Symposium on Circuits and Systems*, pages 42–48, 1982.

[12] R.K. Brayton, G.D. Hachtel, C. McMullen, and A.L.M. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*, volume 2. Springer Science & Business Media, 1984.

[13] S. Sapra, M. Theobald, and E. Clarke. SAT-Based Algorithms for Logic Minimization. In *Proceedings of the 21st International Conference on Computer Design*, pages 510–517. IEEE, 2003.

[14] O. Coudert, J.C. Madre, and H. Fraisse. A New Viewpoint on Two-Level Logic Minimization. In *Proceedings of the 30th International Design Automation Conference*, pages 625–630, 1993.

[15] S.B. Sasi and N. Sivanandam. A Survey on Cryptography Using Optimization Algorithms in WSNs. *Indian Journal of Science and Technology*, 8(3):216, 2015.

[16] A. de la Piedra, M. Venema, and G. Alpár. ABE Squared: Accurately Benchmarking Efficiency of Attribute-Based Encryption. *Cryptology ePrint Archive*, 2022.

[17] I. Oleniuc. New Monotone Boolean Formula Minimizer. `https://github.com/gareth618/abepy`, 2023.

[18] A. Ioniță, D. Banu, and I. Oleniuc. Heuristic Optimizations of Boolean Circuits with Application in Attribute-Based Encryption. In *27th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*, 2023.

[19] S. Kirkpatrick and M.P. Vecchi C.D. Gelatt Jr. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[20] A. Ioniță, D. Banu, and I. Oleniuc. Monotone Boolean Formula Minimizer. `https://github.com/Juve45/Boolean-Circuit-Minimizer-for-ABE`, 2023.

[21] Rich Documentation. `https://rich.readthedocs.io`.

[22] Questionary Documentation. `https://questionary.readthedocs.io`.

[23] A. Ioniță. Optimizing Attribute-Based Encryption for Circuits using Compartmented Access Structures. Cryptology ePrint Archive, Paper 2023/712, 2023.