論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Improvements of PPSZ Algorithms for Unique 3-SAT Problem
著者(和文)	秦同
Author(English)	Tong Qin
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第12062号, 授与年月日:2021年9月24日, 学位の種別:課程博士, 審査員:渡辺 治,伊東 利哉,田中 圭介,山下 真,鹿島 亮
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第12062号, Conferred date:2021/9/24, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Improvements of PPSZ Algorithms for Unique 3-SAT Problem

Tong Qin

17D38061

Supervised by

Prof. Osamu Watanabe

Department of Mathematical and Computing Science

Tokyo Institute of Technology

Abstract

The Boolean Satisfiability Problem (SAT in short) is the decision problem of determining whether there exists an assignment of boolean values, 0 (*false*) or 1 (*true*), to the variables of a boolean formula such that the whole formula evaluates to 1 (then the assignment is called *a satisfying assignment*). The problem k-SAT is a special case of SAT, where the input formula is given in k-CNF (Conjunctive Normal Form), i.e., as a conjunction of clauses of at most k literals. 2-SAT can be solved in linear time. In contrast, 3-SAT is the first known NP-complete problem (k-SAT for any $k > 3$ is also NP-complete), as proven independently by Cook [2] and Levin [8]. A lot of problems have been proven to be NP-complete using reductions from 3-SAT, which is seen as a basic method to prove a problem NP-complete. Thus, finding better exponential-time algorithms for solving 3-SAT has been an active research target for decades.

Though we cannot expect a polynomial-time algorithm for 3-SAT (unless P = NP), it is still possible to obtain an exponential-time algorithm that solves 3-SAT significantly faster than the naive exponential-time algorithm that runs in $2^n \cdot \mathrm{poly}(n)$-time to solve a given 3-CNF formula with n variables. Many researchers have proposed better algorithms for solving 3-SAT. Among them, the randomized algorithm proposed by Paturi, Pudlàk, Saks, and Zane [12] (PPSZ in short) is simple but strong, and it has been studied as one of the standard templates of 3-SAT algorithms. The key idea of PPSZ is simple. Choose a random order π on the variables and fix them one by one in this order. Set each variable x to "your best knowledge" under the assumption that all previous choices can still be extended to a satisfying assignment. If you can somehow figure out

the correct assignment for variable x, then follow your assignment; otherwise, you guess its value uniformly ar random.

Usually, we discuss k-SAT in two different cases. Unique k-SAT is a problem where the input formula F has at most one satisfying assignment, and General k-SAT is a problem where the input formula F has more than one satisfying assignment. Because of the proved theorem that if we can improve Unique k-SAT, we can also get a similar (smaller) improvement for General k-SAT [17], most researchers focus on the study of Unique k-SAT. There are two main kinds of improvements over PPSZ for Unique 3-SAT. One is based on the idea from Hertli [4], and the other is biased-PPSZ [3]. In this thesis, we give two corresponding improvements for each algorithm.

In [4], Hertli studies the special cases of PPSZ in which PPSZ behaves better than usual. He emphasizes the importance of the critical clause, where there is only one true literal under a satisfying assignment. He shows that we can divide the input formulas into two groups, one has many variables that have more than one critical clause; and the other has few variables that have more than one critical clause. For the first kind of formula, PPSZ itself can give us a faster running time. For the second kind of formula, Hertli uses brute-force search to screen those variables having more than one critical clause and gives a careful case by case method combing with [19] to deal with the rest part. In chapter 3, we push his idea further. We give a smarter way to deal with the rest part, which returns a better result than Hertli's. Although the amount of improvement is insignificant, it at least shows that the improvement of the algorithm in question is far from over.

As for another method, biased-PPSZ in [3], Hansen, Kaplan, Zamir, and Zwick introduce a new idea of using "bias" and claimed that a relatively large improvement is possible by using "bias". There are two interesting points in their approach. Firstly, they propose a systematic way to use "bias" for predicting an assignment to a variable when the assignment cannot be inferred from the previous assignments, whereas an assignment is guessed uniformly at random in PPSZ. Since this idea is natural, we may be able to introduce several variations following this approach. In chapter 4, we

propose an additional way to use "bias" and define an algorithm QW_biased-PPSZ that would improve their algorithm further. Secondly, they introduce a numerical method for analyzing the computational complexity of their algorithm. The analysis is rigorous at least on grid points, and we may expect that the accuracy of the analysis increases by reducing the grid interval. We introduce in this thesis another way to justify the complexity bound that is obtained numerically for our algorithm, which is the best result for Unique 3-SAT so far.

Acknowledgements

First of all, I would like to thank my advisor Watanabe Osamu, for his help with my Ph.D. thesis and other research, as well as for his support in studying and living at Tokyo Tech. We had a lot of enlightening discussions about k-SAT and computational complexity. In particular, he has been patient in guiding me through my research during this recent year when things have not been going well.

Then I would like to thank my parents, Qin Changhong and He Ben, for supporting my study in Japan. I am very appreciative that they always trust me and give their love and care to me. I am sorry that I cannot spend much more time with them.

Last, I would like to thank my family in Japan. My girlfriend, Wang Ping, has been a source of joy and spiritual support in my life. Without her patient care and support, I would not have completed my Ph.D. thesis and started my new career life. And thanks to my pets, Pi Pi (the cat) and Mao Mao (the dog), for their companion in such difficult times.

Contents

Chapter 1

Introduction

1.1 Background

The Boolean Satisfiability Problem (SAT in short) is the decision problem of determining whether there exists an assignment of boolean values, 0 (*false*) or 1 (*true*), to the variables of a boolean formula such that the whole formula evaluates to 1 (then the assignment is called *a satisfying assignment*). The problem k-SAT is a special case of SAT, where the input formula is given in k-CNF (Conjunctive Normal Form), i.e., as a conjunction of clauses of at most k literals. 2-SAT can be solved in linear time. In contrast, 3-SAT is the first known NP-complete problem (k-SAT for any $k > 3$ is also NP-complete), as proven independently by Cook [2] and Levin [8]. A lot of problems have been proven to be NP-complete using reductions from 3-SAT, which is seen as a basic method to prove a problem NP-complete. Thus, finding better exponential-time algorithms for solving 3-SAT has been an active research target for decades. A naive brute-force search algorithm solves it in time $O(2^n \text{poly}(n))$[1], where n is the number of variables. The first non-trivial upper bound on the exponent, for any $k \geq 3$, was obtained by Monien and Speckenmeyer [9]. They used a deterministic branching algorithm to show that it runs in time $O(1.619^n)$ when $k = 3$. Improved deterministic algorithms were then obtained, culminating with the time bound $O(1.476^n)$ by Rodošek [16].

[1] From now on, we omit the poly(n) term for simplicity.

Further improved upper bounds came from randomized algorithms. Schöning [18] proposed a very simple randomized algorithm based on a simple *random walk*. It solves k-SAT in $O((2-\frac{2}{k})^n)$ time. In 1997, Paturi, Pudlák and Zane [11] found another randomized algorithm (which is commonly called PPZ) for k-SAT in $O(2^{(1-\frac{1}{k})n})$ time. Afterward, Paturi, Pudlák, Saks, and Zane [12] presented an improved version of the PPZ algorithm by using the *bounded resolution*, which is known as the PPSZ algorithm. It solves k-SAT in a faster running time, for example, $O(1.364^n)$ for 3-SAT and $O(1.308^n)$ for Unique 3-SAT, where Unique k-SAT means that there is only one satisfying assignment for the input formula. They showed that for $k \geq 5$, the PPSZ algorithm behaves the same in both unique and general cases. However, when $k = 3, 4$, their analysis showed that the PPSZ algorithm for the general case behaves worse than the unique case. Many researchers tried to close the gap between the unique case and the general case for several years. There were many improved results of the general case for $k = 3, 4$ but still not close to the time bounds of the unique case. Until Hertli [5] found a new method called *d-implication* instead of the bounded resolution to predict the values of variables, which closed the gap. Hertli [4] also showed that the PPSZ exponent for Unique 3-SAT can be improved by about 10^{-24}. He used an algorithm from Wahlström [19] that is faster than PPSZ on short formulas, that is, formulas with a relatively small number of clauses. I and Watanabe [13, 14] pushed the idea of Hertli a bit further and analyzed the algorithm more carefully, which reduced the exponent by about 10^{-19} and lead to the best result for Unique 3-SAT. A year later, Hansen, Kaplan, Zamir, and Zwick [3] presented a biased version of the PPSZ algorithm. Even with an incomplete analysis, they showed that the biased-PPSZ algorithm runs in time, for example, $O(1.306996^n)$ for Unique 3-SAT. It updates our result and has become the best for the Unique k-SAT problem now.

In 2017, Scheder and Steinberger [17] simplified Hertli's analysis [5] of the general case, and also showed that improvements over the PPSZ-type algorithms for the unique case imply small(er) improvements also for the general case. Thus, by using the theorem, the improvement obtained from biased-PPSZ [3] for Unique k-SAT also ensures an

improvement for General k-SAT, which is also the best result so far. The theorem is meaningful since we only need to focus on the Unique k-SAT problem, which is usually believed easier to improve than the General k-SAT problem.

1.2 Our contributions

There are two main contributions in this thesis.

First, we analyze Hertli's methods in [4] and push his idea further, which gives us a better time bound for Unique 3-SAT.

Second, we give a more efficient algorithm than biased-PPSZ from [3]. We present a *biased-Search* instead of the brute-force search that is used when the maximum disjoint set is small. We give a natural enhanced version of the biased-PPSZ algorithm, which is named by QW_biased-PPSZ and thereby has a faster running time $O(1.306984^n)$ for Unique 3-SAT. The algorithm and the proof of the improvement can be applied to Unique k-SAT for $k \geq 3$. However, the numerical part would be very complicated and has no guarantee that it is tight. Therefore we have to omit the computation for simplicity like [10] and give a careful analysis for Unique 3-SAT as an example.

Furthermore, as discussed previously, the improvement for Unique k-SAT can easily be extended to General k-SAT by the main theorem in [17], which gives the best results for both Unique and General 3-SAT so far. Table 3.1 shows the main results for Unique 3-SAT on the exponential coefficient ϵ, which is the exponent of base 2 of the running time.

1.3 Notation

A boolean formula in propositional is an expression built from boolean variables, the operators AND ($\wedge$), OR ($\vee$) and NOT ($\neg$ or an upper bar $^-$ over a variable). Let V be a finite set of boolean variables. n is the number of the variables in V. Variables are indexed as $x_1, \ldots, x_n$. But we can use x, y, z to denote variables in $\{x_1, \ldots, x_n\}$ for simplicity. A *literal* l_i over $x_i \in V$ is itself x_i or its negation denoted by $\bar{x}_i$. If $l_i = \bar{x}_i$,

Algorithm	ϵ	2^{ϵ}
Naive: ϵ_{naive}	1.0	2
PPSZ: ϵ_{PPSZ}	0.386295	1.307031
Qin, Watanabe[2018]: ϵ_{2018}	$0.386295 - 2.47 \times 10^{-19}$	
HKZZ_biased-PPSZ: ϵ_{HKZZ}	0.386254	1.306995
QW_biased-PPSZ: ϵ_{QW}	0.386241	1.306984

Table 1.1: Comparison of exponential coefficient ϵ

then $\bar{l}_i$ is defined as x_i. A *(CNF) clause* C over V is a finite disjunction of literals from V such as $x_1 \vee x_3 \vee \bar{x}_4$. A *(CNF) boolean formula* F is a finite conjunction of clauses over V. For a clause C (resp., a formula F), we write $\text{var}(C)$ (resp., $\text{var}(F)$) for the set of variables appearing in C (resp., F). We say that formula F is a k-CNF formula if every clause of F contains at most k literals. For any k-CNF formula F, let $|F|$ denote the *size* of F, which is the number of the clauses contained in F.

A *(truth) assignment* on V is a mapping $\alpha : V \mapsto \{0, 1\}$, which assigns a boolean value to each variable. An assignment α satisfies a formula F if F evaluates to 1 when substituting the variables with their values under α. Such an assignment is called a *satisfying assignment*. The set of all satisfying assignments of F is denoted by $\text{sat}(F)$. For a given formula F, *assigning* b *to* x, denoted by $x \leftarrow b$, means that substitute the value b to variable x in formula F and update it. We call b the single (satisfying) assignment of x, and the obtained formula is denoted by $F|_{x \leftarrow b}$. Similarly, the obtained formula, called *current formula*, after assigning a partial assignment α' to F is denoted by $F|_{\alpha_0}$. In this thesis, we only consider the case that the k-CNF formula F is satisfiable. And we use log to denote $\log_2$ for short.

1.4 Outline

In Chapter 2, we introduce the PPZ and PPSZ algorithm from top to bottom. In Chapter 3, we firstly explain Hertli's approach of improving PPSZ for Unique 3-SAT. Then we

introduce our improvement over him and give a more reliable analysis. In Chapter 4, we do the same thing again. First, we explain the biased-PPSZ algorithm and its analysis according to [3] and [10]. Then we introduce biased-Search and the QW-PPSZ algorithm for Unique 3-SAT. Last, we show the improvement over biased-PPSZ and give a careful numerical analysis for the success probability. Finally, in Chapter 5, we conclude this thesis and give some open problems for further study.

Chapter 2

The PPZ and PPSZ Algorithms

In this chapter, we introduce the PPZ [11], and PPSZ [12] algorithms (in short, PPZ and PPSZ) by following Hertli's modified version of PPSZ [5] and his doctor thesis [6]. We use the same notations as much as possible to avoid confusion. The differences from the original paper [12] will be mentioned if needed.

2.1 The PPZ algorithm

The idea behind PPZ (also PPSZ) is simple. Choose a random order π on the variables and assign them one by one in this order. If a single assignment to a variable is "obvious", that is, it can be implied from the previously assigned values and the current formula by some approach, then assign the variable by following the "obvious" single assignment and call the variable *forced*. Otherwise, assign the variable uniformly at random from $\{0, 1\}$ and call the variable *guessed*. Then, substitute the single assignment to the variable and update the current formula; that is, a clause containing a literal whose value becomes 1 is removed, and a literal whose value becomes 0 is removed from clauses containing this literal. If the "empty" clause, a clause with no literal, appears in the current formula, the algorithm stops with failure. Otherwise, if all clauses are removed (satisfied), then the algorithm outputs the obtained assignment as a satisfying assignment.

The critical point is how to decide whether the single assignment to each variable

is"obvious". The PPZ algorithm uses a special kind of clauses called *unit clause*.

Definition 2.1. A clause is called *unit clause* if it consists of only contains one literal.

If there is a unit clause of x in the current formula, it is straightforward to decide the single assignment of x. For example, if there is a clause $C = \bar{x}$, then x must be assigned 0 to satisfy C. The PPZ algorithm (in short, PPZ) is described below formally.

Algorithm 1 The PPZ algorithm

```
1: π := a random permutation on V
2: α' := an empty assignment
3: for x ∈ V in the order of π do
4:     if there is a unit clause of x then
5:         assign x the value that can satisfy the unit clause
6:     else
7:         assign x from {0, 1} uniformly randomly
8:     end if
9: end for
```

A little bit further, we have the following special clauses deeply related to unit clauses and essential in the PPSZ algorithm.

Definition 2.2. Let F be a CNF formula satisfied by α. We call a clause C *critical* for x (w.r.t. α) if there is only one literal, which is over x, evaluated to 1 under α. We also say that x is critical in clause C.

Consider any unique k-CNF formula F. Since it has only one satisfying assignment, each variable in F has at least one critical clause. Because otherwise, we can obtain another satisfying assignment by flipping the value of a variable with no critical clause. Since F is k-CNF formula, there are at most k literals in any clause $C \in F$. For a critical clause C_x of x, recall that only the literal over x is true under the unique satisfying assignment α. Therefore if the other (at most) $k-1$ variables in C_x appear before x in π *and* these variables are assigned consistent with α, then C_x should become

a unit clause of x since the other literals are all false under α. As the permutation is chosen randomly, we can hope for a "good" enough permutation to produce such a unit clause. The probability that x is forced is at least $\frac{1}{k}$, which proves the following bound.

Theorem 2.3. *Given a k-CNF formula F with a unique satisfying assignment, then the probability that PPZ returns the unique satisfying assignment is at least $2^{-(1-\frac{1}{k})n}$.*

The proof can be found in [11, Theorem 9].

2.2 The PPSZ algorithm

The PPSZ algorithm (in short, PPSZ) is an improved version of PPZ. In [12], the authors use *bounded resolution* to increase the probability that a variable is forced. Later, Hertli [5] uses a weaker approach called "d-implication" to deduce a single assignment for each variable and shows that it works as well as original bounded resolution for Unique k-SAT. Thus, we use his approach for explaining PPSZ. First, we define the notion of "d-imply" as follows.

Definition 2.4. For any d, we say that a CNF formula F *d-implies* a literal ℓ (or the variable x for a literal ℓ) if there exists a subformula $G \subseteq F$ with $|G| \leq d$ such that all satisfying assignments of G set ℓ to 1.

We use $F \vDash_d \ell$ (resp., $F \vDash_d [x \leftarrow b]$) to denote that F d-implies ℓ (resp., $x \leftarrow b$), where $b \in \{0, 1\}$.

Fact 2.5 ([6, Observation 3.2])**.** *For any k-CNF formula F over n variables and any literal ℓ, $F \vDash_d \ell$ can be decided in time $O(n^{kd} \cdot 2^{kd} \cdot \mathrm{poly}(n))$.*

The *d-implication* is to determine the single assignment to a variable x if $F \vDash_d [x \leftarrow b]$. We use $P_d(x)$ to denote a procedure implementing the d-implication. For any variable x, $P_d(x)$ returns b if $F \vDash_d [x \leftarrow b]$ for some $b \in \{0, 1\}$. Otherwise, $P_d(x)$ returns "?". From the above fact, we have a subexponential algorithm for P_d if $d = O(\log n)$. Our PPSZ algorithm stated below as **Algorithm 2** uses P_d with $d = \log n$. Note that the

"unit clause implication" of PPZ is nothing but 1-implication. In other words, PPZ is a special case of PPSZ in this sense.

Algorithm 2 The PPSZ algorithm

```
1: π := a random permutation on V
2: α' := an empty assignment
3: for x ∈ V in the order of π do
   (PPSZ process)
4:     if P_d(x) ∈ {0,1} then                          ▷ d = log n
5:         assign P_d(x) to x                          ▷ x is forced
6:     else
7:         assign x from {0,1} uniformly randomly      ▷ x is guessed
8:     end if
9: end for
```

2.3 The Analysis of PPSZ

In this section, we review the analysis of the PPSZ algorithm for Unique k-SAT. Here we introduce a way to determine a random permutation on V, which is quite helpful for the analysis of PPSZ. We assume that a random permutation of variables is defined according to their values under the *random placement* defined below.

Definition 2.6. A *random placement* π is a mapping from V to $[0,1]$ such that for each x, $\pi(x)$ is chosen independently and uniformly at random from $[0,1]$. In general, for any parameter $p \in [0,1)$, a *random* $(\geq p)$*-placement* π is a mapping from V to $[p,1]$ defined in the same way.

We use π to denote this mapping throughout this thesis and assume that variables are sorted according to their π-values (ascending). Note that we may ignore the case where some two variables have the same π-values. We consider any k-CNF formula F with a unique satisfying assignment and let it be fixed throughout this section. For convenience, we assume that $\alpha(x) = 1$ for all variables of F.

Note that for all forced variables, PPSZ always assigns correctly. So if all guessed variables are assigned correctly, PPSZ returns the unique satisfying assignment α. A variable is whether forced or guessed is based on the previous partial assignment. Notice that both the values and the order (or which variables are assigned previously) matter. Thus, the guessed variables during PPSZ are decided by formula F, satisfying assignment α and permutation π. Since the formula F is given and fixed, we are allowed to omit it for simplicity. Let $\text{Guessed}(\pi, \alpha)$ denote the set of guessed variables after a complete run of PPSZ in the order of π following the values from α. Let $G(\pi, \alpha) := |\text{Guessed}(\pi, \alpha)|$.

Fact 2.7 ([6, Observation 3.4])**.** *Given a k-CNF formula F and a satisfying assignment α. Then* PPSZ *returns α if and only if x is assigned (guessed) correctly (equal to $\alpha(x)$) for all $x \in \text{Guessed}(\pi, \alpha)$.*

By the above fact, the probability that PPSZ returns α is the following expectation,

$$\Pr\left[\text{PPSZ returns } \alpha\right] = \sum_{\pi} \frac{1}{n!} 2^{-G(\pi,\alpha)} = \mathbb{E}_{\pi}\left[2^{-G(\pi,\alpha)}\right] \tag{2.1}$$

By Jensen's inequality on the convex function $x \mapsto 2^{-x}$, we have:

$$\Pr\left[\text{PPSZ returns } \alpha\right] \geq 2^{-\mathbb{E}_{\pi}[G(\pi,\alpha)]} \tag{2.2}$$

Next, we need to bound the expected size of the guessed variables. By the definition, $\mathbb{E}_{\pi}[G(\pi, \alpha)] = \sum_{x \in V} \Pr_{\pi}[x \in \text{Guessed}(\pi, \alpha)]$. The task becomes finding a way to compute the probability $\Pr_{\pi}[x \in \text{Guessed}(\pi, \alpha)]$, the probability that any fixed variable x is guessed conditioned on the previous variables (in the order of π) assigned correctly. First, we define S_k that will be used to bound this probability.

Definition 2.8.

$$S_k := \int_0^1 \frac{p^{\frac{1}{k-1}} - p}{1 - p} dp$$

Theorem 2.9 ([5, Theorem 2.4])**.** *Given a k-CNF formula F with a unique satisfying assignment α, then*

$$\Pr_{\pi}\left[x \in \text{Guessed}(\pi, \alpha)\right] \leq S_k + \epsilon_k^d$$

where ϵ_k^d *goes to* 0 *for* $d \to \infty$ *for any fixed* k*.*

So far, one run of PPSZ spends subexponential time and finds the satisfying assignment with an exponentially small probability. To turn this into a randomized algorithm for Unique k-SAT, we need to repeat PPSZ, proportional to the inverse success probability. The runs are all independent; once a satisfying assignment is found, it is returned, and the algorithm stops. If no satisfying assignment has been found, then the formula is declared unsatisfiable.

Corollary 2.10 ([5, Theorem 2.5])**.** *Given a* k*-CNF formula* F *with a unique satisfying assignment, then* PPSZ *can be turned into a randomized algorithm that returns a satisfying assginment in time* $2^{(S_k+o(1))n}$*.*

2.4 Proof of Theorem 2.9

In this section, we prove the main theorem of PPSZ. Consider any k-CNF formula F with a unique satisfying assignment, and let it be fixed. For convenience, we assume that $\alpha(x) = 1$ for all $x \in V$.

For better understanding, we state the outline of this section here. First, in Subsection 2.4.1, we define *critical clause trees* that represent the structure of a uniquely satisfiable k-CNF formula. Then in Subsection 2.4.2, we show the relation between the critical clause trees and the probability that a variable x is forced during the PPSZ process by reducing the problem into another equivalent probability calculation. Third, in Subsection 2.4.3, we prove the main theorem and compute the alternative probability defined in Subsection 2.4.2 to help us to bound the target probability in (2.2). Last, in Subsection 2.4.4, we state some extra theorems needed in the previous three subsections. Note that the whole proof is kind of different from the one in [12].

2.4.1 Critical Clause Trees

We construct a collection of $(k-1)$-ary trees $\{T_x\}_{x \in V}$, with T_x called a *critical clause tree of* x, which in some way represents multiple ways that x can become d-implied

during a run of the PPSZ algorithm. For a tree T, we denote by $V(T)$ the set of nodes of T. The *depth* of a node is its distance to the root (where the root's depth is 0). The *length* of T, denoted by $l(T)$, is the maximum depth of a node of T.

T_x is a rooted $(k-1)$-ary tree in which every node has at most $k-1$ children. Every node $u \in V(T)$ is labeled both with a variable $x \in V$, which is denoted by var-label(u), and with a clause $C \in F$, denoted by clause-label(u). Here is how T_x is built for a fixed $x \in V$:

1. Start with T_x consisting of a single root. The root has var-label(x), and does not have any clause-label yet.

2. As long as there is a leaf $u \in V(T)$ that does not have any clause-label, execute as follows:

   (a) Define $W := \{\text{var-label}(v) | v \text{ is an ancestor of } u \text{ in } T\}$, where an *ancestor* of u is a node one the path from u to the root, including u itself.

   (b) Define the assignment β as

   $$\beta(z) = \begin{cases} 1 - \alpha(z) & \text{if } z \in W \\ \alpha(z) & \text{otherwise} \end{cases}$$

   (c) Let $C \in F$ be a clause not satisfied by β. Since $\beta \neq \alpha$ and α is the unique satisfying assignment, such a clause exists. Set clause-label(u):= C.

   (d) For each variable w in C whose literal in C is evaluated to false by α, create a new child of u with var-label(w) and non clause-label.

We denote the resulting tree by T_x. Note that every clause in F has at most $k-1$ literals evaluated to false by α, and thus in step (d) we append at most $k-1$ children. Suppose v is an ancestor of u with var-label(v) = y. Since $\beta(y) \neq \alpha(y)$, there is no literal over y that is evaluated to false by α contained in clause-label(u). Therefore, the following fact holds.

Fact 2.11 ([6, Obsercation 3.9])**.** *In T_x, no node has the same var-label with one of its ancestors. And $l(T_x) \leq n-1$.*

2.4.2 Relation to Forced Variables

The next step is to find the relation between critical clause trees and forced variables to compute the probability that a variable is forced (or equally, guessed).

Let $p \in [0,1]$ and T_x be the critical clause tree of fixed x. We call a node $u \in T_x$ *reachable at time p w.r.t. π* if the path $P := (\text{root} = v_0, v_1, \ldots, v_m = u)$ from root to u in T_x satisfies that $\pi(\text{var-label}(v_i)) \geq p$ for all $1 \leq i \leq m$. Let $\text{Reachable}(T_x, p, \pi)$ be the set of all nodes in T_x reachable at time p w.r.t. π.

Lemma 2.12 ([6, Lemma 3.10])**.** *If we have* $|\text{Reachable}(T_x, \pi(x), \pi)| \leq d$*, then* $x \in \text{Forced}(\pi, \alpha)$.

Proof. Let α' be the partial assignment to the variables which come before x and compatible with α, where $\alpha = (1, 1, \ldots, 1)$ by assumption. By **Definition 2.4**, x is forced if there is a subformula $F' \subseteq F|_{\alpha'}$ with $|F'| \leq d$ that implies x. Let G be a subformula of F consisting of the clause labels of $\text{Reachable}(T_x, \pi(x), \pi)$. Since $|\text{Reachable}(T_x, \pi(x), \pi)| \leq d$, $|G| \leq d$. If $G|_{\alpha'}$ implies x, the statement is true since $G|_{\alpha'} \subseteq F|_{\alpha'}$.

Assume that $G|_{\alpha'}$ does not imply x. Then there is an assignment β that is compatible with α', where $\beta(x) = 0$ and β satisfies G. Choose a longest path in T_x, starting at the root and containing only nodes v such that $\beta(\text{var-label}(v)) \neq \alpha(\text{var-label}(v))$. Since $\beta(x) \neq \alpha(x)$, the path is not empty. Let u be its endpoint and u must be reachable. Otherwise there is an ancestor y on the path from root to u such that $\pi(y) < p$ and $\beta(\text{var-label}(y)) \neq \alpha(\text{var-label}(y))$, which is a contradiction to that β is compatible with α'.

Since the path from root to u is the longest in T_x such that $\beta(\text{var-label}(v)) \neq \alpha(\text{var-label}(v))$, any child w of u satisfies that $\beta(\text{var-label}(w)) = \alpha(\text{var-label}(w))$. And for any ancestor y of u, $\beta(\text{var-label}(y)) \neq \alpha(\text{var-label}(y))$. By the construction of T_x, all literals of clause-label(u) satisfied by α must be over var-labels of the nodes on the path from u to the root, while all literals of clause-label(u) unsatisfied by α must be over the var-labels of the children of u. Thus, clause-label(u) is unsatisfied by β. However,

clause-label(u) is reachable and in G. It should be satisfied by β. A contradiction.

□

The proof is different from the original PPSZ in [12]. However, the usage of the properties of critical clause trees is similar. And it is clear from the lemma above that

$$\Pr\left[x \in \text{Forced}(\pi, \alpha)\right] \geq \Pr\left[|\text{Reachable}(T_x, \pi(x), \pi)| \leq d\right] \tag{2.3}$$

This reduces the problem to a probabilistic calculation on $(k-1)$-ary trees, more specifically, the probability that T_x has at most d reachable nodes at time $\pi(x)$. It can be considered as another probability that T_x has at most d nodes left after deleting[1] all nodes whose var-labels come before x in the permutation induced by π. More generally, this alternative probability can be stated and bounded by the following theorem.

Theorem 2.13. *For any $\epsilon > 0$ there is a $d_k(\epsilon) \in \mathbb{N}$ such that the following holds. Let $Z_1, Z_2, \ldots, Z_r \in \{0,1\}$ be mutually independent binary random variables, each of which takes value one with probability p. Let T be any finite (not necessarily full) $(k-1)$-ary tree with a labeling $\sigma : V(T)\backslash\{root\} \to \{1, \ldots, r\}$ of the non-root nodes of T with indices with the property that on each path from the root to a leaf, σ is injetive. Draw $Z_1, \ldots, Z_r$ according to their distribution and delet all nodes u from T for which $Z_{\sigma(u)} = 1$. The resulting tree is T'. Then the probability that T' contains more than $d_k(\epsilon)$ nodes is at most $S_k + \epsilon_k^d$, where ϵ_k^d goes to 0 for $d \to \infty$ for any fixed k.*

With the above theorem, we can prove **Theorem 2.9** directly.

Proof of **Theorem 2.9**. By **Theorem 2.13**, the probability that T' contains more than $d_k(\epsilon)$ nodes is at most $S_k + \epsilon_k^d$ after the random deletion. Let $d := d_k(\epsilon)$, then assume the root of T has var-label x. Then, $\Pr\left[|\text{Reachable}(T_x, \pi(x), \pi)| \geq d\right] \leq S_k + \epsilon_k^d$. Hence we have the following inequalities by (2.3):

$$\begin{aligned}
\Pr\left[x \in \text{Forced}(\pi, \alpha)\right] &\geq \Pr\left[|\text{Reachable}(T_x, \pi(x), \pi)| \leq d\right] \\
&= 1 - \Pr\left[|\text{Reachable}(T_x, \pi(x), \pi)| \geq d\right] \\
&\geq 1 - S_k - \epsilon_k^d
\end{aligned} \tag{2.4}$$

[1]Here "deleting" a node means erasing the whole subtree rooted at this node.

Thus, $\Pr[x \in \text{Guessed}(\pi, \alpha)] = 1 - \Pr[x \in \text{Forced}(\pi, \alpha)] \leq S_k + \epsilon_k^d$. □

2.4.3 Probability Analysis

In this subsection, we prove **Theorem 2.13**. Note that this theorem is purely about probabilities in a combinatorial random experiment on $(k-1)$-ary trees while there is no reference to PPSZ.

At first, consider an easier case: an infinite $(k-1)$-ary tree T, where each node (with subtree) is deleted with a fixed probability p independently. We want to find out the probability, denoted by q, that the remaining tree T' is finite. We consider the probability that the length of the remaining tree T' is at most L instead. Somehow, we can bound this probability with some expression of p. Also, we observe that a $(k-1)$-ary tree with bounded length has a bounded number of nodes, which is related to the number of reachable nodes used in the analysis of PPSZ. Next, since we can embed any finite $(k-1)$-ary tree into the infinite one, so the only remaining problem are the dependencies between the nodes. We show that these dependencies do not destroy our proof of the independent case. In the end, we show what happens if the deletion probability p is also random; that is, if it is chosen uniformly randomly from $[0, 1]$.

Note that T' is finite if and only if for each child u of T's root, either ① has been deleted; or ② the subtree rooted at u is finite. Case ① happens with probability p, and case ② happens with probability q. Hence for one child u, the condition holds with probability $p + (1-p)q$, and by independence we have

$$q = (p + (1-p)q)^{k-1} \tag{2.5}$$

Let $R_k(p)$ denote the smallest non-negative solution q to (2.5). And we have the following fact.

Fact 2.14. *Let T_∞ be the infinite rooted full $k-1$-ary tree. For each non-root node from T_∞ is deleted independently with probability p. Then the probability that the remaining tree T' is finite is at least $R_k(p)$.*

$R_k(p)$ has the following lemma which is important to help us to compute the value of $R_k(p)$.

Lemma 2.15 ([6, Lemma 3.14])**.** *Define* $S_k(t) := \frac{t^{\frac{1}{k-1}} - t}{1-t}$ *for* $t \in [0,1)$*, and* $S_k(1) = \frac{k-2}{k-1}$*. Then*

1. *S_k and R_k are monotonically non-decreasing and continuous on $[0,1)$.*
2. *For $p \in [0, \frac{k-2}{k-1})$, $S_k(t)$ is the inverse function[2] of $R_k(p)$.*
3. *For $p \in [\frac{k-2}{k-1}, 1]$, $R_k(p) = 1$.*
4. *Especially,*

$$R_3(p) = \begin{cases} \frac{p^2}{(1-p)^2} & \text{if } p < \frac{1}{2} \\ 1 & \text{otherwise} \end{cases}$$

Proof. Recall that $R_k(p)$ is the smallest non-negative solution to (2.5). For $k = 3$, the equation is a quadratic polynomial with solutions 1 and $\left(\frac{p}{1-p}\right)^2$, and the statement 4 for the $k = 3$ case is true.

Since $t \in [0,1)$, we have

$$\sum_{i=0}^{k-2} t^{\frac{i}{k-1}} = \frac{1-t}{1-t^{\frac{1}{k-1}}} \tag{2.6}$$

Combine it with the definition of $S_k(t)$, we can rewrite

$$S_k(t) = 1 - \frac{1}{\sum_{i=0}^{k-2} t^{\frac{i}{k-1}}} \tag{2.7}$$

Thus, it is clear that $\lim_{t \to 1^+} S_k(t) = S_k(1)$, and that $S_k(t)$ is continuous and monotonically non-decreasing on $[0,1]$, which completes the half of the proof for the statement 1.

Then we show the statement 3. Since $q = 1$ is a solution to (2.5), $R_k(p) \leq 1$. Define $f_k(p,q) := (p + (1-p)q)^{k-1}$. Since f_k is increasing in p when $p \in \left[\frac{k-2}{k-1}, 1\right]$ and $q < 1$, f_k is minimized at $p = \frac{k-2}{k-1}$. Thus, we have

[2]We say function g is an *inverse function* of function f if and only if $g(f(x)) = x$.

$$
\begin{aligned}
f_k(p,q) &= (p+(1-p)q)^{k-1} \\
&\geq \left(\frac{k-2}{k-1} + \frac{1}{k-1}q\right)^{k-1} \\
&= \left(1 - \frac{1}{k-1}(1-q)\right)^{k-1} \\
&> 1 - (k-1)\cdot\frac{1-q}{k-1} = q
\end{aligned}
\tag{2.8}
$$

Since $f_k(p,q) > q$ when $q < 1$, $q = 1$ is the smallest non-negative solution to (2.5). Thus, for $p \in \left[\frac{k-2}{k-1}, 1\right], R_k(p) = 1$, and statement 3 is true.

Next, let's show the statement 2. For $q \in [0,1)$ satisfying (2.5), we have

$$
\begin{aligned}
(p+(1-p)q)^{k-1} &= q \\
p+(1-p)q &= q^{\frac{1}{k-1}} \\
p(1-q) &= q^{\frac{1}{k-1}} - q \\
p &= \frac{q^{\frac{1}{k-1}} - q}{(1-q)} = S_k(q)
\end{aligned}
\tag{2.9}
$$

Thus, if $R_k(p) < 1$, then $S_k(q)$ is the inverse function of $R_k(p)$. To avoid any confusion, we use notation t to replace q in $S_k(.)$ function.

For proving the statement 2, we still need to prove that for $p \in \left[0, \frac{k-2}{k-1}\right), R_k(p) < 1$. Note that for $p = 0$, the smallest $q \geq 0$ satisfying $q = (p+(1-p))q)^{k-1} = q^{k-1}$ is $q = 0$; hence $R_k(0) = 0 < 1$. For $0 < p < \frac{k-2}{k-1}$, by definition of $R_k(p)$, we have

$$
R_k(p) = (p+(1-p)R_k(p))^{k-1} \tag{2.10}
$$

Let $\delta = 1 - R_k(p)$, and it implies that:

$$
\begin{aligned}
(p+(1-p)(1-\delta))^{k-1} + \delta &= 1 \\
(1-(1-p)\delta)^{k-1} + \delta - 1 &= 0
\end{aligned}
$$

Define $g(p,\delta) := (1-(1-p)\delta)^{k-1} + \delta - 1$. For a fixed $0 \leq p \leq \frac{k-2}{k-1}$, g is a continuous function of δ. Moreover, $g'(p,0) = (k-1)(p-1) + 1$. Since $k \geq 3$, $g'(p,0) < 0$. We also have that $g(p,0) = 0$, and that $g(p,1) > 0$ for $p > 0$. So for all $0 < p < \frac{k-2}{k-1}$ there is $\delta' > 0$ such that $g(p,\delta') = 0$. Thus, $1 - \delta' < 1$ is a solution of (2.5), which implies

that $R_k(p) < 1$, and therefore that $S_k(t)$ is the inverse function of $R_k(p)$ on the interval $\left[0, \frac{k-2}{k-1}\right)$. Since $S_k(t)$ is continuous and monotonically non-decreasing function, so is $R_k(p)$. □

Next, we consider the relation between the probability that T' is finite after the random deletion and the probability that the length of T' is bounded. For any tree T, let $l(T)$ denote its length. If T is infinite, $l(T) = \infty$.

Lemma 2.16. *Let T_∞ be the infinite rooted full $(k-1)$-ary tree. Each non-root node from T_∞ is deleted independently with probability p, and let T' be the remaining tree. Then we have*

$$\lim_{L\to\infty} \Pr\left[l\left(T'\right) \leq L\right] = \Pr\left[T' \text{ is finite}\right]$$

Proof. Let B_i be the event that there is a path from the root to some node at depth i in T'. If this event B_i happens, then there exists a path from the root to some node at depth i in T' after the deletion. Consider the event $\bigcap_{i\geq 1} B_i$ that happens, which means that T' contains finite paths of arbitrary length. Thus, T' cannot be finite. And if T' is infinite, it is trivial that $\bigcap_{i\geq 1} B_i$ happens. From **Theorem 2.21**, we have[3]

$$\lim_{i\to\infty} \Pr\left[B_i\right] = \Pr\left[T' \text{ is infinite}\right]$$

Taking complements,

$$\begin{aligned}
&\lim_{L\to\infty} \Pr\left[l\left(T'\right) \leq L\right] = \lim_{L\to\infty} \overline{B_L} \\
&= 1 - \lim_{L\to\infty} B_L = 1 - \Pr\left[T' \text{ is infinite}\right] \\
&= \Pr\left[T' \text{ is finite}\right]
\end{aligned}$$

□

Then we have the following result by combining **Lemma 2.15** and **Lemma 2.16**.

Lemma 2.17. *For each $p \in [0,1]$, there is a sequence $\epsilon_k^1(p), \epsilon_k^2(p), \cdots \in \mathbb{R}_0^+$ with $\lim_{L\to\infty} \epsilon_k^L(p) = 0$ such that the following is true. Let T be any finite $(k-1)$-ary tree.*

[3] $\Pr[B_i]$ means event B_i happens.

Each non-root node from T is deleted independently with probability p, and let T' be the remaining tree. Then for any L, we have

$$\Pr\left[l\left(T'\right) \leq L\right] \geq R_k(p) - \epsilon_k^L(p)$$

Proof. The random deletion we are conducting on T can be coupled to a random deletion conducted on T_∞ in the following way: T is embeddable into T_∞ with the two roots coinciding and then if we delete a node from T_∞, the same deletion happens on T. Let T'' be the remaining tree from T_∞. Obviously, T' is a subtree of T''. Therefore we have $l(T') \leq l(T'')$ and hence $\Pr[l(T') \leq L] \geq \Pr[l(T'') \leq L]$. Recall that $R_k(p)$ is the probability that T'' is finite after the random deletion, which means the length of T'' can be bounded by some constant (for example L). Thus, we have $R_k(p) \geq \Pr[l(T'') \leq L]$ for all $L \geq 1$, and $\lim_{L\to\infty} \Pr\left[l\left(T''\right) \leq L\right] = \Pr\left[T'' \text{ is finite}\right] = R_k(p)$ by **Lemma 2.16**. Define

$$\epsilon_k^L(p) := \max\left\{R_k(p) - \Pr\left[l\left(T''\right) \leq L\right], 0\right\}$$

where the sequence $\epsilon_k^1(p), \epsilon_k^2(p), \cdots \in \mathbb{R}_0^+$ with $\lim\limits_{L\to\infty} \epsilon_k^L(p) = 0$. Thus,

$$\Pr\left[l(T') \leq L\right] \geq \Pr\left[l\left(T''\right) \leq L\right] = R_k(p) - \epsilon_k^L(p)$$

□

Simply speaking, **Lemma 2.17** implies that the random deletion on a finite tree T won't cause much difference comparing with the random deletion on a infinite tree T_∞; that is, the difference can be bounded by a "small" value $\epsilon_k^L(p)$. Thus, we have the following theorem.

Theorem 2.18 ([6, Theorem 3.11])**.** *For any $\epsilon > 0$ there is a $d_k(\epsilon) \in \mathbb{N}$ such that the following holds. Let $X_1, X_2, \ldots, X_r \in [0,1]$ be real random variables distributed uniformly and mutually independently. Let T be any finite $(k-1)$-ary tree and σ be a mapping (labeling) from nodes in T to $\{X_1, \ldots, X_r\}$ such that σ is injective on each path from the root to a leaf of T. Consider drawing $X_1, X_2, \ldots, X_r$ according to their distribution and then deleting all nodes u from T for which $X_{\sigma(u)} < X_{\sigma(root)}$. Let T' be the remaining*

tree after the deletion. Then the probability that T' contains more than $d_k(\epsilon)$ nodes is at most $S_k + \epsilon_k^d$.

In the random deletion studied so far, each node was deleted independently of all other nodes. However, this is not the case in the PPSZ analysis (critical clause tree), where nodes of the tree are linked to one another and if one of these linked nodes is deleted, all of them are deleted. The following lemma shows that these dependencies can only improve the probability to have a smaller remaining tree, which implies that the above theorem still works for these dependencies.

Lemma 2.19. *For any $\epsilon > 0$ there is a $d_k(\epsilon) \in \mathbb{N}$ such that the following holds. Let $Z_1, Z_2, \ldots, Z_r \in \{0,1\}$ be mutually independent binary random variables, each of which takes value one with probability p. Let T be any finite (not necessarily full) $(k-1)$-ary tree with a labeling $\sigma : V(T)\backslash\{root\} \to \{1,\ldots,r\}$ of the non-root nodes of T with indices with the property that on each path from the root to a leaf, σ is injetive. Draw $Z_1, \ldots, Z_r$ according to their distribution and delet all nodes u from T for which $Z_{\sigma(u)} = 1$. The resulting tree is T'. Juxtapose the deletion in T where every non-root node is deleted independently from all other nodes with probability p. The resulting tree is T''. Then for any L,*

$$\Pr\left[l(T') \leq L\right] \geq \Pr\left[l(T'') \leq L\right]$$

Proof. If σ is globally injective, the statement is trivial. We now use the FKG inequality (**Theorem 2.22**) to demonstrate that correlations arising from duplicate labels cannot decrease this probability if none of these duplicates are in an ancestor-descendant relation.

Consider an induction on L. For $L = 0$ the statement is trivial.

For the induction step, let $L > 0$. If the root of T has no child, the statement is trivial. Suppose the root has $0 < j < k-1$ children $u_1, u_2, \ldots, u_j$. Let $i \in \{1, 2, \ldots, j\}$ and let us look at each of the root's subtrees separately. Let T_i be the subtree rooted at u_i and let $Z_i = \sigma(u_i)$. Let T_i' denote the subtree of T' rooted at u_i and T_i'' the subtree of T_i'' rooted at u_i (empty trees if u_i is deleted). The hypothesis on σ entails that no

other node in T_i is labeled with Z_i, so whatever happens in the non-root nodes of T_i is independent of whether u_i itself is being deleted or note. Let

$$E_i := \left\{ Z_i = 1 \vee \left(Z_i = 0 \wedge l\left(T_i'\right) \leq L - 1 \right) \right\}$$

We have that

$$\left\{ l\left(T'\right) \leq L \right\} = \bigwedge_{i=1}^{j} E_i$$

Moreover, the E_i are events which are determined by $\{Z_1, \ldots, Z_r\}$ and as one can easily check, monotonically increasing in these. Therefore we can use the FKG inequality to obtain that

$$\Pr\left(l\left(T'\right) \leq L\right) = \Pr\left(\bigwedge_{i=1}^{j} E_i\right) \geq \prod_{i=1}^{j} \Pr\left(E_i\right)$$

For these separate events, we have, due to the independence of T_i from Z_i

$$\Pr\left(E_i\right) = p + (1-p)\Pr\left(l\left(T_i'\right) \leq L - 1\right) \geq p + (1-p)\Pr\left(l\left(T_i''\right) \leq L - 1\right)$$

where we have used the induction hypothesis for the inequalities. Combining the last two inequalities, we obtain

$$\begin{aligned} \Pr\left(l\left(T'\right) \leq L\right) &\geq \prod_{i=1}^{j} \left(p + (1-p)\Pr\left(l\left(T_i''\right) \leq L - 1\right)\right) \\ &= \Pr\left(l\left(T''\right) \leq L\right) \end{aligned}$$

□

Then, we are ready to prove **Theorem 2.13** with the previous results. Recall it here.

Theorem 2.13. *For any $\epsilon > 0$ there is a $d_k(\epsilon) \in \mathbb{N}$ such that the following holds. Let $Z_1, Z_2, \ldots, Z_r \in \{0, 1\}$ be mutually independent binary random variables, each of which takes value one with probability p. Let T be any finite (not necessarily full) $(k-1)$-ary tree with a labeling $\sigma : V(T) \backslash \{root\} \to \{1, \ldots, r\}$ of the non-root nodes of T with indices with the property that on each path from the root to a leaf, σ is injetive. Draw $Z_1, \ldots, Z_r$ according to their distribution and delet all nodes u from T for which $Z_{\sigma(u)} = 1$. The*

resulting tree is T'. Then the probability that T' contains more than $d_k(\epsilon)$ nodes is at most $S_k + \epsilon_k^d$, where ϵ_k^d goes to 0 for $d \to \infty$ for any fixed k.

Proof. For a fixed ϵ, consider an arbitrary tree T is labeled with real-valued random variables as $\sigma : V(T) \to \{X_1, X_2, \ldots, X_r\}$ in such a way that σ is injective on any path from the root to a leaf of T. Now each X_i is drawn independently and uniformly from $[0, 1]$. Then all nodes are deleted (along with their subtrees) whose corresponding random variable is smaller than that of the root, producing the resultant random tree T'.

Without loss of generality, suppose $\sigma(\text{root}) = X_r$. Note that this value is independent from all other values used because the root must be contained in all paths and σ is injective on each path. Let $X_r = p_0$ for some fixed value $p_0 \in [0, 1]$, and consider for $1 \leq i \leq r-1$ the binary random variables:

$$Z_i := \begin{cases} 1 & \text{if } X_i < p_0 \\ 0 & \text{otherwise} \end{cases}$$

The $\{Z_1, Z_2, \ldots, Z_{r-1}\}$ are independent binary random variables, each of which takes value 1 with probability p_0. By **Theorem 2.18** and **Lemma 2.19**, we have

$$\begin{aligned} &\Pr\left[l\left(T'\right) \leq L \mid X_r = p_0\right] \geq R_k(p_0) - \epsilon_k^L(p_0) \\ &\text{and} \\ &\lim_{L\to\infty} \Pr\left[l\left(T'\right) \leq L \mid X_r = p_0\right] \geq R_k(p_0) \end{aligned} \tag{2.11}$$

That is, the probability that the length of T' conditioned on $X_r = p_0$ is at most L is at least $R_k(p_0)$.

Since X_r is uniformly distributed, we can convert the conditional into an unconditional probability, that is,

$$\Pr\left[l(T') \leq L\right] = \int_0^1 \Pr\left[l(T') \leq L | X_r = p_0\right] dp_0 \tag{2.12}$$

Since the probability above and $R_k(p_0)$ are monotonically non-decreasing, we have

$$\begin{aligned} \lim_{L\to\infty} \Pr\left[l\left(T'\right) \leq L\right] &= \lim_{L\to\infty} \int_0^1 \Pr\left[l\left(T'\right) \leq L \mid X_r = p_0\right] dp_0 \\ &\geq \int_0^1 R_k(p_0) dp_0 \end{aligned}$$

By **Lemma 2.15**, since $R_k(p)$ is the inverse of $S_k(t)$, we have

$$\begin{aligned} 1 - S_k &= 1 - 1 \cdot S_k(1) + 0 \cdot S_k(0) + \int_0^1 S_k'(t)dt \\ &= \frac{1}{k-1} + \int_0^1 R_k\left(S_k(t)\right) S_k'(t)dt \\ &= \frac{1}{k-1} + \int_0^{S_k(1)} R_k(p)dp \\ &= \int_{\frac{k-2}{k-1}}^1 R_k(p)dp + \int_0^{\frac{k-2}{k-1}} R_k(p)dp \\ &= \int_0^1 R_k(p)dp \end{aligned} \tag{2.13}$$

Thus, the probability that T' contains more than $d_k(\epsilon)$ nodes is at most

$$1 - \int_0^1 \left(R_k(p) - \epsilon_k^L(p)\right) dp = S_k + \epsilon_k^d$$

Especially, for $k = 3$, since

$$\int_0^1 R_3(p)dp = 2 - 2\ln 2$$

then $S_3 = 2\ln 2 - 1$.

□

Thus, by (2.3), we complete the proof of **Theorem 2.9**. Particularly, together with (2.2), we have

Lemma 2.20. *Given a 3-CNF formula F with a unique satisfying assignment, then*

$$\Pr\left[\text{PPSZ } \textit{returns } \alpha\right] \geq 2^{-(S_3+o(1))n} \geq 2^{-(2\ln 2-1+o(1))n}$$

2.4.4 Extra Omitted Proofs

In this subsection, we state below several mathematical theorems that are used in the previous proofs. We omit the proofs of these theorems here, which can be found in [12] and [6].

Theorem 2.21. *[6, Theorem A.4] Let $B_1, B_2, \ldots$ be an infinite sequence of events such that $B_i \supseteq B_{i+1}$ for all $i \geq 1$. Then we have*

$$\lim_{n\to\infty} \Pr(B_n) = \Pr\left(\bigcap_{i=1}^{\infty} B_i\right)$$

Theorem 2.22 (FKG Inequality)**.** *[6, Theorem A.7] Let $\mathcal{A} = \{A_1, A_2, \ldots, A_r\}$ be a collection of mutually independent binary random variables and E_1 and E_2 events which are determined by $\mathcal{A}$ and monotonically increasing in $\mathcal{A}$. Then*

$$\Pr(E_1 \wedge E_2) \geq \Pr(E_1) \cdot \Pr(E_2)$$

Furthermore, for m, the events $E_1, E_2, \ldots, E_m$

$$\Pr\left(\bigwedge_{i=1}^{m} E_i\right) \geq \prod_{i=1}^{m} \Pr(E_i)$$

Chapter 3

The Hertli's Algorithm and its Improvements

In this chapter, we recall Hertli's algorithm and some of the facts from [4] necessary for our discussion firstly. We follow [4] and use the same algorithms and lemmas including the usage of symbols as much as possible (except for correcting some minor errors and keep the same notation used in the last chapter). Note that we only consider Unique 3-SAT in this chapter.

3.1 Hertli's Algorithm

Hertli's algorithm uses PPSZ [12] and Wahlström's algorithm [19]. First consider PPSZ and discuss two minor changes on PPSZ introduced in [4] for simplifying analysis. We start with some notions.

We prepare some of the key notions for our discussion. For any $k \geq 1$, a k-clause is a clause having exactly k literals. For simplifying our statement, unless otherwise stated explicitly, we use F to denote any satisfiable 3-CNF formula and α to denote its unique satisfying assignment regarded as a target assignment. We may assume that F also satisfies a certain condition given in each context. For any algorithm $\mathbf{A}$, let $\mathrm{E_A}$ denote the event that $\mathrm{A}(F)$ yields α. Below let $\mathrm{PPSZ}(F)$ denote the execution of PPSZ on

F. Similar to Chapter 2, let $G(\pi, \alpha)$ be the number of the guessed variables during the execution of PPSZ. Since there is only one satisfying assignemnt α in the unique case, we can omit it and write $G(\pi)$ instead for simlicity. We restate the following two lemmas and PPSZ below. Note that although they are somewhat different in their description from those in Chapter 2, they have the same meaning and can be easily verified using (2.13) and **Lemma 2.15**.

Lemma 3.1. *Define* $S_{3,t}$ *by*

$$S_{3,t} = \int_t^1 \left(1 - \min\left\{1, \frac{p^2}{(1-p)^2}\right\}\right) dp$$

For any $t \in [0, 1/2]$, *let* $G_t(\pi)$ *be the number of variables with placement* $> t$ *that are guessed during the execution* $\mathrm{PPSZ}(F|\pi, \alpha)$. *Then we have*

(1) for any $t \in [0, 1/2]$, $\mathbb{E}_\pi[G_t(\pi)] = (S_{3,t} + o(1))n$,

(2) $S_{3,t} = S_3 - t + I(t)$, *where* $I(t) := \int_0^t \frac{p^2}{(1-p)^2} dp$ *and*

$$S_3 = \int_0^1 \left(1 - \min\left\{1, \frac{p^2}{(1-p)^2}\right\}\right) dp = 2\ln 2 - 1 = 0.386295\cdots$$

From these lemmas, we have the following bounds.

Lemma 3.2. *Let* F *be any satisfiable* 3*-CNF that has a unique satisfying assignment* α. *Then* $\Pr[\mathrm{E}_{\mathrm{PPSZ}}]$ *is at least* $2^{-(S_3+o(1))n}$. *Furthermore, suppose that we pick every variable of* F *with probability* t *independently, and let* V_t *be the resulting set. Let* $\mathrm{E}_{\mathrm{PPSZ},V_t}$ *denote the event that* $\mathrm{PPSZ}(F|_{\alpha|_{V_t}})$ *returns* $\alpha|_{V\setminus V_t}$. *Then we have* $\mathbb{E}_{V_t}[\log \Pr[\mathrm{E}_{\mathrm{PPSZ},V_t}]] \geq \mathbb{E}_\pi[-G_t(\pi)] = -(S_{3,t} + o(1))n$.

We consider the above bound $2^{-(S_3+o(1))n}$ as a target, and we propose algorithms for certain types of input formulas with some "efficiency improvements" that is, algorithms that have success probabilities larger than $2^{-(S_3-\varepsilon+o(1))n}$ for some $\varepsilon > 0$. The first such example is PPSZ itself; PPSZ performs better if a given formula has many variables with more than one critical clauses [4, 10].

Lemma 3.3. *Let F be any satisfiable 3-CNF that has a unique satisfying assignment α. If Δn variables of F have more than one critical clause, then*

$$\Pr[\mathrm{E}_{\mathrm{PPSZ}}] \geq 2^{-(S_3 - 0.00145\cdots\Delta + o(1))n}$$

Furthermore, if F has more than Δn variables that have a critical 2-clause, then $\Pr[\mathrm{E}_{\mathrm{PPSZ}}] \geq 2^{-(S_3 - 0.0353\Delta + o(1))n}$.

Wahlström [19] proposed a deterministic algorithm for solving the CNF-SAT problem. Here we denote by WAHLSTROM the following procedure based on Wahlström's algorithm.

Remark. Wahlström's algorithm is valid for any k-CNF-SAT problem. However, in this thesis, we only use it for 3-SAT.

Definition 3.4. For a CNF formula F and a variable x, the *degree* of x in F is defined to be the number of clauses in F that contain the variable x, which is denoted by $\deg(F, x)$. The *average degree* of any CNF formula F is the average degree among all variables in F.

Lemma 3.5. *For any CNF formula F with no unit clause that has average degree at most d, $4 < d \leq 5$,* WAHLSTROM*(F) computes one of its satisfying assignments in time $O(2^{0.115707\cdots(d-1)n})$.*

Note that the time complexity of WAHLSTROM is not $2^{o(n)}$. Thus, we consider the following randomized procedure W_rand: For a given input CNF formula F over n variables with average degree $\leq d$, W_rand(d, F) executes the above procedure with probability $2^{-0.115707(d-1)n}$, and otherwise it stops the computation immediately with failure. We remark here that WAHLSTROM halts in time $O(2^{0.115707\cdots(d-1)n})$ even for an input that does not satisfy the condition of the above lemma. Thus, the expected running time of W_rand, which executes WAHLSTROM with probability $2^{-0.115707(d-1)n}$, is $2^{o(n)}$ for any input (including the case where the input does not satisfy the condition of the lemma).

Now we consider Hertil's algorithm HERTLI stated as **Algorithm 3**. First we remark on our way to state algorithms by pseudo codes. In the following algorithms such as **Algorithm 3**, we consider several cases on a given formula, and execute a sat. algorithm on the formula or its subformula that works efficiently for each case. Though it is not stated explicitly, by, e.g., "Execute PPSZ(F')" we mean to (i) execute the procedure PPSZ on F', (ii) compute a satisfying assignment of the input formula of the procedure based on the obtained satisfying assignment, and then (iii) terminate the computation by yielding the computed satisfying assignment. Clearly, if the execution at the step (i) fails, then the computation is terminated reporting "failure". Note also that it may not be easy to determine which case actually holds for a given formula. Therefore, we consider all the cases *in parallel*. By "**assume** Φ **then** $\cdots$" in our algorithm descriptions, we mean to execute the "$\cdots$" part in parallel with the other cases assuming that Φ holds.

Algorithm 3 HERTLI

input: a 3-CNF formula F, **parameter:** $\Delta_1, \Delta_2, \delta_3, \delta_4, t$

assume F has more than $\Delta_1 n$ var.s with more than one critical clause **then**
  Execute PPSZ(F)
assume otherwise **then**
  Choose u.a.r. a size $\Delta_1 n$ subset W_1 of V and an assignment α_1 on W_1
    ▷ assume below that W_1 and α_1 are correctly chosen
  $F' \leftarrow F|_{\alpha_1}$; $V' \leftarrow \mathrm{var}(F')$; $n' \leftarrow |V'|$
  assume F' is Δ_2-dense **then** ▷ what follows is the dense-case algorithm
    $F_2 \leftarrow$ GetInd2Clauses(F')
    Execute DensePPSZ$_t$(F', F_2)
  assume otherwise **then** ▷ what follows is the sparse-case algorithm
    Choose u.a.r. a size $\Delta_2 n'$ subset W_2 of V' and an assignment α_2 on W_2
      ▷ assume below that W_2 and α_2 are correctly chosen
    $F'' \leftarrow F'|_{\alpha_2}$
    Execute SparsePPSZ(F'')

Algorithm 4 GetInd2Clauses

input: a 3-CNF formula F', **parameter:** Δ_2

$V' \leftarrow \mathrm{var}(F')$; $n' \leftarrow |V'|$

$F_3 \leftarrow \{C \in F : |C| = 3\}$; $F_2 \leftarrow \emptyset$; $W_2' \leftarrow \emptyset$

for $T_2(n)$ times **do** ▷ Define $T_2(n) = \lceil \Delta_2 n/2 \rceil$

  $x \leftarrow$ a variable of F_3 with $\deg_3(F_3, x) \geq 5$

  ▷ *failure stop* if no such variable exists

  Choose C u.a.r. from all clauses of F_3 with $\ell_x \in C$

  $C_2 \leftarrow C \setminus \{\ell_x\}$

  $F_2 \leftarrow F_2 \cup \{C_2\}$; $W_2' \leftarrow W_2' \cup \mathrm{var}(C_2)$

  $F_3 \leftarrow \{C \in F_3 : \mathrm{var}(C) \cap \mathrm{var}(C_2) = \emptyset\}$

end for

return F_2

Subprocedures GetInd2Clauses, DensePPSZ$_p$, and SparsePPSZ that are used in HERTLI are stated as **Algorithms 4**, **5**, and **6**.

Based on [4, 6][1] we can show that the following efficiency improvement is possible by HERTLI on uniquely satisfiable 3CNF formulas.

Theorem 3.6. *Use values given in the "value of [6]" column of Table 3.1 for the parameters of the procedure* HERTLI *and its subprocedures, and also for* ε_0*. For any uniquely satisfiable 3CNF formula* F*, let* $\mathrm{E}_{\mathrm{HERTLI}}$ *denote the event that* HERTLI(F) *yields the unique satisfying assignment of* F*. Then we have* $\log \Pr[\mathrm{E}_{\mathrm{HERTLI}}] \geq -(S_3 - \varepsilon_0 + o(1))n$.

We explain the proof of the theorem by showing that each procedure achieves its required task with desired probability. Since our target F is any 3-CNF formula with a unique satisfying assignment α. Thus, by "success probability" we mean the probability that α is obtained. We assume that variables in the procedures with the same name are given these values and that parameters used in the procedures are set values given

[1] Due to some minor error in [4], the choice of parameters in [4] is not appropriate, which has been corrected in [6]. Here we use this corrected version.

Algorithm 5 DensePPSZ$_t$

input: a 3-CNF formula F' and a 2CNF formula F_2, **parameter:** $t \in (0,1)$

$V' \leftarrow \mathrm{var}(F')$; $n' \leftarrow |V'|$

$V'_t \leftarrow$ pick each $x \in V'$ with probability t

Let α'_2 be a partial assignment on V' initially undefined for all $x \in V'$

for $C_2 \in F_2$ **do**

  if $\mathrm{var}(C_2) \subseteq V_p$ **then** ▷ let u and v are two literals of C_2

$$(\alpha'_2(u), \alpha'_2(v)) \leftarrow \begin{cases} (0,0) & \text{with probability } 1/5\ (= 3/15), \text{ and} \\ (0,1),\ (1,0), \text{ or } (1,1) & \text{with probability } 4/15 \text{ for each} \end{cases}$$

  end if

end for

for $x \in V'_p$ **do**

  if $\alpha'_2(x)$ is undefined **then**

    $\alpha'_2(x) \leftarrow_{\text{u.a.r.}} \{0,1\}$

  end if

end for

execute PPSZ($F'|_{\alpha'_2}$)

Algorithm 6 SparsePPSZ

input: a 3-CNF formula F'', **parameter:** δ_3, δ_4

Let α'' be a partial assignment on F'' initially undefined for all $x \in \mathrm{var}(F'')$
$\widetilde{F} \leftarrow F''$;
if $\widetilde{F}$ has a unit clause **then**
    Extend α'' to satisfy all unit clauses and simplify $\widetilde{F}$
        ▷ if an unsat. clause is derived in $\widetilde{F}$ during this step, then stop with "failure"
end if
while $\widetilde{F}$ has some clause **do**
    $\widetilde{V} \leftarrow \mathrm{var}(\widetilde{F})$; $\widetilde{n} \leftarrow |\widetilde{V}|$
    $F_2 \leftarrow \{C \in \widetilde{F} : |C| = 2\}$
    if $|F_2| \leq \delta_3 \widetilde{n}$ **then**
        Execute W_rand$(2\delta_3 + 4, \widetilde{F})$
    else
        assume $\widetilde{F}$ has $\delta_4 \widetilde{n}$ critical 2-clauses **then** Execute PPSZ$(\widetilde{F})$
        assume otherwise **then**
                ▷ that is, more than $1 - \delta_4/\delta_3$ of 2-clauses of F_2 are noncritical
            Choose C u.a.r. from F_2
            Extend α'' to satisfy all literals in C and simplify $\widetilde{F}$
    end if
end while

name	ref.	value in [6]	new value	note
ε_0	Thm. 3.6	10^{-25}	2.47×10^{-19}	the efficiency improvement totally
ε_1	Lem. 3.9	10^{-3}	$(2.32\cdots) \times 10^{-3}$	the efficiency improvement for (H3)
ε_2	Lem. 3.10	10^{-20}	$(9.23\cdots) \times 10^{-15}$	the efficiency improvement for (H2)
Δ_1		10^{-22}	1.6595×10^{-16}	the fraction of W_1
Δ_2		5×10^{-5}	3.7736×10^{-3}	the fraction of W_2
δ_3		$1/10$	0.159227	
δ_4		$1/30$	0.06572	
t	t_*	5×10^{-7}	$(3.82\cdots) \times 10^{-5}$	
q		–	$(10^5 \Delta_2 n)^{-1}$	

Table 3.1: Parameters used in Hertli's algorithm and our improvements

in the "value in [6]" column of Table 3.1. Values of this column are also used efficiency improvements ε_0, ε_1, and ε_2. We also assume that n is quite large so that our choice of parameters would make sense.

First consider the outline of HERTLI. From **Algorithm 3** we see that HERTLI uses three algorithms for the following three cases:

(H1) := [F has more than $\Delta_1 n$ variables with more than one critical clause]

(H2) := [$\neg$ (H1) $\wedge$ F'_* is Δ_2-dense]

(H3) := [$\neg$ (H1) $\wedge$ F'_* is Δ_2-sparse]

That is, PPSZ(F), DensePPSZ$_t(F'_*, F_{2,*})$, and SparsePPSZ(F''_*) are executed when (H1), (H2), and (H3) holds.

Definition 3.7. The *k-clauses degree* (or degree$_k$) of x in F is defined to be the number of k-clauses in F that contain the variable x, which is denoted by $\deg_k(F, x)$.

Note we say that a formula is b-degree$_k$ *bounded* if the largest k-clauses degree of its variables is at most b.

Here we consider the situation where the values of the variables W_1, α_1, W_2, and α_2 of HERTLI are guessed "appropriately" and take the following values:

$$\begin{aligned} W_{1,*} &= \text{the set of variables with more than one critical clause,} \\ W_{2,*} &= \text{a set of varriables of size } \leq 2T_2(n) \text{ s.t. } F \setminus W_{2,*} \text{ is 4-degree}_3 \text{ bounded,} \\ \alpha_{1,*} &= \alpha_*(W_{1,*}), \text{ and } \alpha_{2,*} = \alpha_*(W_{2,*}) \end{aligned}$$

Then the variables F', F'', F_2 are set the following values:

$$F'_* = F|_{\alpha_{1,*}}, \ F''_* = F'_*|_{\alpha_{1,*}}, \text{ and } F_{2,*} = \text{GetInd2Clauses}(F'_*),$$

where the last one is for the case that GetInd2Clauses(F'_*) successfully returns a result. We also use V'_*, n'_*, V''_*, and n''_* for the corresponding values, i.e., $\text{var}(F'_*)$, $|\text{var}(F'_*)|$, $\text{var}(F''_*)$, and $|\text{var}(F''_*)|$.

The case where (H1) holds is simple; in fact, we have already prepared **Lemma 3.3** for this case, which gives the following success probability bound.

Lemma 3.8. *Suppose that (H1) holds for F. Then we have* $\log \Pr[\mathrm{E}_{\mathrm{PPSZ}}] \geq -(S_3 - 0.00145 \cdots \Delta_1 + o(1))n$. *That is, the log of the success probability of the line* $2-3$ *of* HERTLI *is at least* $-(S_3 - 0.00145 \cdots \Delta_1 + o(1))n$, *which is larger than* $-(S_3 - \varepsilon_0 + o(1))n$

Thus, for proving the theorem, it suffices to guarantee the improvement over exponential coefficient, which is called as efficiency imrpovement shortly, ε_0 for the other cases.

For the case where (H3) holds, the procedure SparsePPSZ is used. Its task is simply to get a satisfying assignment for F''_* given in this case. For its success probability, we have the following lemma[2], which is proved in [4, Lemma 6]. Below we use $\mathrm{H}(r)$ to denote the binary entropy, and use the well-known bound $\log \binom{n}{rn} \leq \mathrm{H}(r)n$ that holds for any $r \in [0,1]$ such that rn is an integer.

[2]The success probability bound can be improved by analyzing it more carefully using $n''_* = (1 - \Delta_1)(1 - \Delta_2)n$. Here we follow [4] and omit this consideration in this and the next lemmas; we will discuss such detail improvements in the next section.

Lemma 3.9. *Suppose that (H3) holds for F. Let* $\mathrm{E}_{\mathrm{Sparse}}$ *denote the event that* SparsePPSZ(F''_*) *returns* $\alpha_*|_{V''_*}$. *Then we have* $\log \Pr[\mathrm{E}_{\mathrm{Sparse}}] \geq -(S - \varepsilon_1 + o(1))n''_*$. *That is, the efficiency improvement of* SparsePPSZ *on* F''_* *is at least* ε_1. *Furthermore, including the probability of guessing* $W_{1,*}$, $\alpha_{1,*}$, $W_{2,*}$, *and* $\alpha_{2,*}$, *the log of the success probability of the lines from 9 to 12 of* HERTLI *is at least* $-(S_3 + \Delta_1 + \mathrm{H}(\Delta_1) + \Delta_2 + \mathrm{H}(\Delta_2) - \varepsilon_1)n$, *which is larger than* $-(S_3 - \varepsilon_0 + o(1))n$.

Finally consider the case where (H2) holds. In this case, HERTLI first executes GetInd2Clauses(F'_*) to get a set $F_{2,*}$ of $T_2(n)$ independent 2-clauses, and then executes DensePPSZ$_t(F'_*, F_{2,*})$ to get a satisfying assignment for F'_*. Since (H2) holds, it is easy to see that the execution GetInd2Clauses(F'_*) always terminates successfully. Then 2-clauses of $F_{2,*}$ are obtained from 3-clauses of F'_*; furthermore, it follows from (H2) that on average (w.r.t. the randomness of GetInd2Clauses) at least 4/5 2-clauses in $F_{2,*}$ are from noncritical 3-clauses of F'_*. This is a key to derive the following lower bound on the success probability of the execution DensePPSZ$_t(F'_*, F_{2,*})$.

Lemma 3.10. *Suppose that (H2) holds for F. Then* GetInd2Clauses *successfully returns a set of* $T_2(n)$ *independent 2-clauses. Let* $\mathrm{E}_{\mathrm{Dense}_t}$ *denote the event that* DensePPSZ$_t(F'_*, F_{2,*})$ *returns* $\alpha_*|_{V'_*}$. *Then we have* $\log \Pr[\mathrm{E}_{\mathrm{Dense}_t}] \geq -(S_3 + I(t) - a_0\Delta_2 t^2 + o(1))n'_*$, *where* $a_0 = 0.00505\cdots$. *That is, the efficiency improvement of* DensePPSZ$_t$ *on* $(F'_*, F_{2,*})$ *is at least* $\varepsilon_2 := \max_t(-I(t) + a_0\Delta_2 t^2)$. *Thus, including the probability of guessing* $W_{1,*}$ *and* $\alpha_{1,*}$, *the log of the success probability of the lines from 6 to 8 of* HERTLI *is at least* $-(S_3 + \Delta_1 + \mathrm{H}(\Delta_1) - \varepsilon_2 + o(1))n$, *which is larger than* $-(S_3 - \varepsilon_0 + o(1))n$.

(Since this lemma is related to our improvement, we state the outline of the proof from [4].)

Proof Outline. We give an outline of the analysis of $\Pr[\mathrm{E}_{\mathrm{Dense}_t}]$. Consider **Algorithm 5**, i.e., the procedure DensePPSZ. We borrow the symbols V'_t and α'_2 there and use them also as random variables to denote respectively a set selected randomly at the line 2 and an assignment on V'_t defined randomly at the line 3. Let $\mathrm{E}_{\mathrm{guess}}$ denote the event that α'_2

$= \alpha_*(V'_p)$. Note that $\mathrm{E}_{\mathrm{guess}}$ also depends on the execution of GetInd2Clauses(F'_*). Then as shown in [4, Lemma 5], we have

$$\begin{aligned}\log \Pr[\mathrm{E}_{\mathrm{Dense}_t}] &\geq \mathbb{E}_{V'_t}\left[\log \Pr[\mathrm{E}_{\mathrm{guess}}|V'_t] + \log \Pr[\mathrm{E}_{\mathrm{PPSZ},V'_t}]\right] \\ &= \mathbb{E}_{V'_t}\left[\log \Pr[\mathrm{E}_{\mathrm{guess}}|V'_t]\right] + \mathbb{E}_{V'_t}\left[\log \Pr[\mathrm{E}_{\mathrm{PPSZ},V'_t}]\right]\end{aligned}$$

Note that **Lemma 3.2** gives a bound for the second term. That is, we have

$$\mathbb{E}_{V'_t}\left[\log \Pr[\mathrm{E}_{\mathrm{PPSZ},V'_t}]\right] \geq -(S_{3,t} + o(1))n = -(S_3 + I(t) - t)n.$$

Hence, what we need is to bound $\mathbb{E}_{V'_t}\left[\log \Pr[\mathrm{E}_{\mathrm{guess}}|V'_t]\right]$. That is, our task is to analyze the probability that α'_2 is consistent with α_* on V'_t.

We introduce useful notions. Consider any 2-clause $C \in F_{2,*}$. Recall that it is obtained from some 3-clause of F'_* by removing a literal ℓ in the execution GetInd2Clauses(F'_*). We say that C is *critical clause origin* (resp., *noncritical clause origin*) if C is obtained from a 3-clause that is critical for the variable x.

Let us first see how to assign variables in V'_t. The assignment is determined at the lines from 4 to 13 of DensePPSZ$_t$. In particular, for a randomly chosen set V'_t of variables of F'_*, if it has a pair of literals appearing in some $C \in F_{2,*}$, then the assignment on these literals is determined as stated at the line 6. It can be shown that this is an optimal way to obtain an assignment consistent with α_* when we know that the probability that $C \in F_{2,*}$ is critical clause origin is at most $1/5$ (hence, the probability that its two literals are assigned $(0,0)$ by α_* is at most $1/5$).

We analyze this method and give a lower bound for the probability that α'_2 is consistent with α_* on V'_t. We consider two random variables m_0 and m_1. Let m_0 (resp., m_1) denote the number of 2-clauses C of $F_{2,*}$ such that $\mathrm{var}(C) \subseteq V'_t$ and it is critical (resp., noncritical) clause origin. Note that m_0 and m_1 are random variables depending on both V'_t and the randomness in the execution GetInd2Clauses(F'_*). Thus, by, e.g., "$\mathbb{E}[m_0]$" we mean the expectation of m_0 over the random variable V'_t and the randomness in GetInd2Clauses(F'_*). Note, however, $m_0 + m_1$ depends only on V'_t; in fact, we have $\mathbb{E}[m_0 + m_1] = t^2 T_2(n)$, which is independent from GetInd2Clauses(F'_*). On the other hand, we have $\mathbb{E}[m_0] \leq t^2(1/5)T_2(n)$ because, on average, at most $1/5$ clauses of

$F_{2,*}$ are critical clause origin. The probability that α'_2 is consistent with α_* on V'_t is $(1/2)^{|V'_t|-2(m_0+m_1)}(1/5)^{m_0}(4/15)^{m_1}$.

Based on this observation, the following bound is shown in [4].

$$\begin{aligned}
&\mathbb{E}_{V'_t} \quad \left[\log \Pr[\mathrm{E}_{\mathrm{guess}}|V'_t]\right] \\
&= \quad -\mathbb{E}[|V'_t| - 2(m_0+m_1)] + \mathbb{E}[m_0]\log\left(\frac{1}{5}\right) + \mathbb{E}[m_1]\log\left(\frac{4}{15}\right) \\
&= \quad -pn + \left(2 + \log\left(\frac{4}{15}\right)\right)\mathbb{E}[m_0+m_1] - \mathbb{E}[m_0]\log\left(\frac{4}{3}\right) \\
&= \quad -tn + t^2 T_2(n)\left(2 + \log\left(\frac{4}{15}\right)\right) - \mathbb{E}[m_0]\log\left(\frac{4}{3}\right) \qquad (3.1) \\
&\geq \quad -tn + \frac{t^2\Delta_2 n}{2}\left(2 + \log\left(\frac{4}{15}\right) - \frac{1}{5}\log\left(\frac{4}{3}\right)\right) \\
&= \quad -(t - a_0\Delta_2 t^2)n,
\end{aligned}$$

where $a_0 = (2 + \log(4/15) - (1/5)\log(4/3))/2 = 0.00505\cdots$. This is sufficient for the desired bound.

3.2 Our Improvements

The key idea of our main improvement is to use GetInd2Clauses for obtaining W_2 instead of guessing it randomly in the straightforward way, thereby removing the $-\mathrm{H}(\Delta_2)$ term from the efficiency improvement of the sparse case (**Lemma 3.9**). In order to give a condition that this idea works, we introduce a "soft" version of the Δ-sparseness/-denseness.

The new algorithm is given as **Algorithm 7**. It uses a procedure $\mathrm{GetInd2Clauses}^+_t$ for computing a set W_2 corresponding to the one guessed in the original Hertli's algorithm. This procedure is obtained by modifying the procedure GetInd2Clauses on two points. For a given formula F', instead of computing a set F_2 of independent 2-clauses, $\mathrm{GetInd2Clauses}^+_t$ aims to compute a set W_2 such that $F' \setminus W_2$ becomes 4-degree$_3$ bounded. Thus, the line 4 of **Algorithm 4** is modified so that if x cannot be found, then the algorithm stops *successfully* by reporting W_2. On the other hand, the termination of

Algorithm 7 newPPSZ

input: a 3-CNF formula F, **parameter:** $\Delta_1, \Delta_2, \delta_3, \delta_4, t, q$

1: **assume** F has more than $\Delta_1 n$ variables with more than one critical clause **then**
2: Execute PPSZ(F)
3: **assume** otherwise **then**
4: Choose u.a.r. a size $\Delta_1 n$ subset W_1 of V and an assignment α_1 on W_1;
5: $F' \leftarrow F|_{\alpha_1}$; $V' \leftarrow \mathrm{var}(F')$; $n' \leftarrow |V'|$
6: **assume** F' is (Δ_2, q)-dense **then**
7: $F_2 \leftarrow$ GetInd2Clauses(F')
8: Execute DensePPSZ$_t(F', F_2)$
9: **assume** otherwise **then**
10: $W_2 \leftarrow \mathrm{GetInd2Clauses}_t^+(F')$
11: Choose α_2 u.a.r. from all assignments on W_2;
12: $F'' \leftarrow F'|_{\alpha_2}$
13: Execute SparsePPSZ(F'')

its main for-loop, i.e., the lines from 3 to 9 of **Algorithm 4**, is regarded as an undesired situation. GetInd2Clauses$_t^+$ tries this part for $1/t$ times for a given parameter t and stops with "failure" if a desired W_2 is not obtained by all trials. We state this procedure as **Algorithm 8** below.

Our new condition is to determine which of GetInd2Clauses and GetInd2Clauses$_t^+$ is likely to succeed. Consider the execution GetInd2Clauses(F'). We regard it as a random process of collecting an independent 2-clause from $F' \setminus W_2$ to F_2 while choosing a variables x with $\deg_3(x) \geq 5$ randomly. For each $j \geq 1$, let N_j denote the event that there exists a degree$_3 \geq 5$ variable in F_3 at beginning of the jth iteration of the main for-loop and hence the algorithm executes the jth iteration. We define $N_{\leq j} \iff \bigwedge_{1 \leq i \leq j} N_i$ and $q_j = \Pr[\neg N_j \mid N_{\leq j-1}]$.

Definition 3.11. For any $q \in [0,1]$ and $\Delta > 0$, a 3-CNF formula F is (Δ, q)-*dense* if $q_j \leq q$ for all j, $1 \leq j \leq \lceil \Delta n/2 \rceil$; otherwise, F is (Δ, q)-*sparse*.

Algorithm 8 GetInd2Clauses$_t^+$

input: a 3-CNF formula F'

$V' \leftarrow \mathrm{var}(F')$; $n' \leftarrow |V'|$;

$F_3 \leftarrow \{C \in F : |C| = 3\}$; $F_2 \leftarrow \emptyset$; $W_2' \leftarrow \emptyset$

for $1/q$ times **do**

  if there is no variable of F_3 with $\deg_3(F_3, x) \geq 5$ **then**

    return W_2

    break

  else

    $x \leftarrow$ a variable of F_3 with $\deg_3(F_3, x) \geq 5$

    Choose C u.a.r. from all clauses of F_3 with $\ell_x \in C$

      $C_2 \leftarrow C \setminus \{\ell_x\}$

    $F_2 \leftarrow F_2 \cup \{C_2\}$; $W_2' \leftarrow W_2' \cup \mathrm{var}(C_2)$

    $F_3 \leftarrow \{C \in F_3 : \mathrm{var}(C) \cap \mathrm{var}(C_2) = \emptyset\}$

  end if

end for

return failure

This is the new condition used in **Algorithm 7**. Although not exactly the same, the $(\Delta, 0)$-dense/sparse condition is practically the same as the Δ-dense/sparse condition w.r.t. the execution of GetInd2Clauses.

Now our task is to show that the success probability of the new sparse case is in fact improved and that the success probability of the new dense case is not so affected. Similar to the previous discussion, we consider the execution of the procedure newPPSZ when some sufficiently large and uniquely satisfiable 3-CNF formula F is given as an input; the symbols such as F'_*, etc. are used in the same way as before while we leave the choice of parameter values for a later discussion. For a given parameter q, we define $(\mathrm{H2})_t^+$ and $(\mathrm{H3})_t^+$ by

$(\mathrm{H2})_t^+ := [\, \neg(\mathrm{H1}) \wedge F'_* \text{ is } (\Delta_2, q)\text{-dense} \,]$

$(\mathrm{H3})_t^+ := [\,\neg(\mathrm{H1}) \wedge F'_*$ is (Δ_2, q)-sparse$\,]$

We first show that the success probability is improved for the (Δ_2, q)-sparse case. In the following, let T_2 denote $T_2(n)$ and N denote the event $\bigwedge_{1\le i\le T_2} N_i$.

Lemma 3.12. *Suppose that $(H3)_t^+$ holds for F. Then with $\Omega(1)$ probability W_2 is successfully computed by* $\mathrm{GetInd2Clauses}_t^+(F'_*)$. *Thus, including the probability of guessing $W_{1,*}$, $\alpha_{1,*}$, and $\alpha_{2,*}$, the log of the success probability of the lines from 9 to 13 of* newPPSZ *is at least $-(S + \Delta_1 + \mathrm{H}(\Delta_1) + \Delta_2 - \varepsilon_1)n$, where ε_1 is a lower bound for the efficiency improvement of* SparsePPSZ *on F''_*.*

Proof. Suppose that $(\mathrm{H3})_t^+$ holds for F. That is, there is some i_0 such that $q_{i_0} \ge q$. Then we have

$$\begin{aligned}\Pr[N] &= \Pr\left[\bigwedge_{1\le i\le T_2} N_i\right] = \prod_{1\le i\le T_2} \Pr[\,N_i \,|\, N_{\le i-1}\,] \\ &= \prod_{1\le i\le T_2} (1-q_i) \;\le\; 1-q_{i_0} \;<\; 1-q.\end{aligned}$$

Note that $\Pr[N]$ is the probability that W_2 is not obtained at one execution of the main for-loop of $\mathrm{GetInd2Clauses}_t^+(F'_*)$. Since $\mathrm{GetInd2Clauses}_t^+$ tries the main for-loop for $1/q$ times, the probability that W_2 cannot be obtained by all these trials is at most $(1-q)^{1/q} \le \mathrm{e}^{-1}$, proving the first claim of the lemma.

Let $W_{2,*}$ be the set obtained by $\mathrm{GetInd2Clauses}_t^+(F'_*)$. Then $F''_* := F \setminus W_{2,*}$ is 4-degree$_3$ bounded, and for such F''_* it is easy to see that the efficiency improvement of $\mathrm{SparsePPSZ}(F''_*)$ is the same as the one given before by **Lemma 3.9**, which is sufficient for proving the rest of the lemma. □

Then by following lemma we can say that DensePPSZ works as well even under the condition $(\mathrm{H2})_t^+$.

Lemma 3.13. *Suppose that $(H2)_t^+$ holds for F. Then with probability at least $1 - T_2 q$, $F_{2,*}$ is successfully computed by* $\mathrm{GetInd2Clauses}(F'_*)$. *Recall that $\mathrm{E}_{\mathrm{Dense}_t}$ denote the event that* $\mathrm{DensePPSZ}_t(F'_*, F_{2,*})$ *returns $\alpha_*(V'_*)$. We have $\log \Pr[\mathrm{E}_{\mathrm{Dense}_t}] \ge -(S_3 +$*

$I(t) - a_q\Delta_2 t^2 + o(1))n$, *where*

$$a_q \;=\; \frac{1}{2}\left(2 + \log\left(\frac{4}{15}\right) - \frac{1}{5(1-T_2q)}\log\left(\frac{4}{3}\right)\right),$$

which is close to a_0 *of* **Lemma 3.10** *by setting* $q = (10^5 T_2)^{-1} = (10^5\Delta_2 n)^{-1}$.

Proof. Here again we analyze $\Pr[N]$ as the probability that $F_{2,*}$ is obtained successfully in the execution of GetInd2Clauses(F'_*).

$$\begin{aligned}
\Pr[N] &= 1 - \Pr\left[\bigvee_{1\le i\le T_2} \neg N_i \wedge N_{\le i-1}\right] \;=\; 1 - \sum_{1\le i\le T_2} \Pr[\,\neg N_i \wedge N_{\le i-1}\,] \\
&= 1 - \sum_{1\le i\le T_2} \Pr[\,\neg N_i \,|\, N_{\le i-1}\,]\cdot \Pr[N_{\le i-1}] \\
&\ge 1 - \sum_{1\le i\le T_2} \Pr[\,\neg N_i \,|\, N_{\le i-1}\,] \;=\; 1 - \sum_{1\le i\le T_2} q_i \;\ge\; 1 - T_2 q.
\end{aligned}$$

This proves the first claim of the lemma.

The log of the success probability of DensePPSZ$_t$, i.e., $\log \Pr[\mathrm{E}_{\mathrm{Dense}_t}]$ can be analyzed in the same way as the proof of **Lemma 3.10**. In fact, we can start from the equation (3.1), and for the analysis, it suffices to estimate $\mathbb{E}[m_0]$. Recall that the random variable m_0 denotes the number of "critical clause origin" 2-clauses C of $F_{2,*}$ such that $\mathrm{var}(C) \subseteq V'_p$. Since variables are chosen to V'_p uniformly at random, $\mathbb{E}[m_0]$ is determined by $\mathbb{E}[m'_0|N]$, where m'_0 is the number of "critical clause origin" 2-clauses selected for $F_{2,*}$ in the execution of GetInd2Clauses(F'_*) and the expectation is over the randomness of the execution of GetInd2Clauses(F'_*) (under the condition that the execution is terminated successfully). For each i, $1 \le i \le T_2$, let I_i denote a random variable that takes 1 if the 2-clause selected at the i iteration is critical clause origin, and takes 0 otherwise. Let $I_{i,1}$ denote the event that $I_i = 1$. Note that $\Pr[I_{i,1}|N_{\le i-1}] \le 1/5$ since the variable x selected at the ith iteration appears in at least five 3-clauses and each variable has at

most one critical 3-clause. Then we have

$$\begin{aligned}
\mathbb{E}[\, m_0' \,|\, N\,] &= \mathbb{E}\left[\, \sum_{1\le t\le T_2} I_t \,\middle|\, N \,\right] = \sum_{1\le i\le T_2} \Pr[\, I_{i,1} \,|\, N\,] \\
&= \sum_{1\le i\le T_2} \frac{\Pr[\, I_{i,1}\wedge N\,]}{\Pr[N]} \le \frac{1}{\Pr[N]} \sum_{1\le i\le T_2} \Pr[\, I_{i,1}\wedge N_{\le i-1}\,] \\
&= \frac{1}{\Pr[N]} \sum_{1\le i\le T_2} \Pr[\, I_{i,1} \,|\, N_{\le i-1}\,]\cdot \Pr[N_{\le i-1}] \\
&\le \frac{1}{\Pr[N]} \sum_{1\le i\le T_2} \Pr[\, I_{i,1} \,|\, N_{\le i-1}\,] \le \frac{1}{\Pr[N]} \sum_{1\le i\le T_2} \frac{1}{5} \le \frac{T_2}{5(1-T_2q)}.
\end{aligned}$$

Hence, we have $\mathbb{E}[m_0] \le t^2T_2/(5(1-T_2q))$. By substituting this to (3.1), we have the bound of the lemma. □

As stated in **Lemma 3.10**, a lower bound for the efficiency improvement of $\mathrm{DensePPSZ}_t(F_*', F_{2,*})$ is calculated as $\varepsilon_2 := \max_t(-I(t) + a_q\Delta_2 t^2)$. Then including the probability of guessing $W_{1,*}$, $\alpha_{1,*}$, and $\alpha_{2,*}$, the log of the success probability of the lines from 6 to 8 of newPPSZ is at least $-\{\Delta_1 + \mathrm{H}(\Delta_1) + (S_3-\varepsilon_2)(1-\Delta_1)\}n$.

3.3 Detail Efficiency Improvement Analysis and Our Choice of Parameters

Now by setting the parameters used in the algorithms appropriately, we can show the following efficiency improvement.

Theorem 3.14. *For the procedure* newPPSZ*, use values given in the "new value" column of Table 3.1 for its parameters and also for ε_0. For any uniquely satisfiable 3-CNF formula, the log of its success probability is at least $-(S_3 - \varepsilon_0 + o(1))n$.*

We explain how to set the parameters to obtain the efficiency improvement in **Theorem 3.14**.

We choose the parameter values so that the final efficiency improvement, i.e., ε_0, is optimal up to three significant figures. Values stated as, e.g., $(2.32\cdots)\cdot 10^{-3}$, are

estimated to enough significant figures, while values stated as, e.g., $1.71527 \cdot 10^{-16}$ are used ones by confirming that they are precise enough to guarantee that ε_0 is optimal up to three significant figures.

In order to achieve the optimal efficiency improvement, we need to be a bit more careful than [4, 6]. First consider the procedure SparsePPSZ. See **Algorithm 6**; we use symbols defined there. In this procedure, three methods are used to compute a satisfying assignment depending on the condition of a given formula F''. Let γ_3 and γ_4 denote respectively the proportion of 2-clauses and critical 2-clauses of F''. If γ_3 is small enough, Walhström's algorithm (more precisely, W_rand) works better. If γ_3 is large and γ_4 is also large, then PPSZ works better. Otherwise, we had better reduce the current formula by assigning values to two variables to satisfy all two literals of a randomly chosen clause in F_2. Considering the worst case (which is the case where all three methods are equally good), we compute thresholds for γ_3 and γ_4 that are used as parameters δ_3 and δ_4. The efficiency improvement ε_1 of this procedure is also estimated from the worst case success probability. Note that while γ_3 is easy to compute, γ_4 may not be computable easily. Thus, in the procedure SparsePPSZ, the procedure W_rand is simply used when $\gamma_3 \leq \delta_3$; on the other hand, both PPSZ and the variable number reduction are executed in parallel.

Once ε_1 is estimated, we need to choose Δ_2 so that the success probability is large enough compared with ε_0. In **Lemma 3.9** and **Lemma 3.12**, we stated a slightly weak bound for comparison. In fact, there should be two more precise bounds and we use these instead for better accuracy. For **Lemma 3.9**, the log of the success probability of the lines from 9 to 12 of HERTLI is at least $-\{\Delta_1 + \mathrm{H}(\Delta_1) + \Delta_2 + \mathrm{H}(\Delta_2) + (S_3 - \varepsilon_1)(1-\Delta_1-\Delta_2)\}n$; for **Lemma 3.12**, the log of the success probability of the lines from 6 to 8 of newPPSZ is at least $-\{\Delta_1 + \mathrm{H}(\Delta_1) + \Delta_2 + (S_3 - \varepsilon_1)(1 - \Delta_1 - \Delta_2)\}n$. Since $\Delta_1 \ll \Delta_2$, we may be able to ignore Δ_1 for determining Δ_2 and check whether this gives an enough room later after determining Δ_1. The rest of the calculation for parameter values is almost the same as [4, 6]. We compute the optimal value for p following **Lemma 3.10**. As mentioned in **Lemma 3.13**, we use a_q with $q = (10^5\Delta_2 n)^{-1}$, which

gives a success probability bound ε_1 close enough to the one calculated using a_0 for obtaining the optimal ε_0. From ε_1, we determine Δ_1 so that the success probability given in the explanation below **Lemma 3.13** is large enough compared with ε_0. Finally, we use **Lemma 3.8** to estimate ε_0 and obtain the results of our improvements shown in Table 3.1.

Chapter 4

Biased-PPSZ Algorithms and Our Improvements

In 2019, Hansen, Kaplan, Zamir, and Zwick [3] claimed that a relatively[1] large improvement is possible by introducing "bias", and they in fact gave the complexity upper bound of their concrete algorithm (we call it HKZZ_biased-PPSZ in short) for Unique 3-SAT [10]. There are two interesting points in their approach. Firstly, they propose a systematic way to use "bias" for predicting a single assignment to a variable (when the assignment cannot be inferred from the previous assignments), whereas the assignment is guessed uniformly at random in PPSZ. Since this idea is natural, we may be able to introduce several variations following this approach. In fact, we propose here an additional way to use "bias" and define an algorithm QW_biased-PPSZ that would improve their algorithm further. Secondly, they introduced in [10] (not in [3]) a numerical method for analyzing the computational complexity of their algorithm. The analysis is rigorous, at least on examined grid points, and we may expect that the accuracy of the analysis increases by reducing the grid interval. Unfortunately, we could not understand how they justify the precision of their obtained complexity bound; on the other hand, we introduce here another way to justify (under a certain technical assumption)

[1] Compared with the improvements we discussed in Chapter 3.

the complexity bound that is obtained numerically for our algorithm. In this chapter, we explain first the idea of a general biased-PPSZ algorithm and a more concrete one proposed by Hansen et al. in [3]. We then explain our new idea of using "bias" and the algorithm QW_biased-PPSZ implementing this idea. Finally, we explain the numerical method of Zamir [10] and give our new detail analysis deriving the success probability lower bound (more specifically, the exponential constant $\varepsilon_{\text{QW-PPSZ}}$ of Table 1.1 given in Introduction).

Though biased-PPSZ algorithms could be defined for any k-SAT, we focus on 3-SAT in this chapter, and we define algorithms for 3-SAT, and analyze their success probability on Unique 3-SAT instances. Thus, throughout this chapter, we use F to denote any uniquely satisfiable 3-CNF formula, and let α denote its unique sat. assignment. For simplicity, we only discuss the *exact* Unique 3-SAT problem in this chapter that is, all clauses in the input formula F have exactly 3 literals. Note that all the algorithms stated in this chapter are also valid for $(\leq k)$-CNF formula with a similar analysis like Chapter 2 and Chapter 3.

4.1 The Generic Biased-PPSZ Algorithm

As we explained in Chapter 2, PPSZ deduces the value of a variable from single assignments set so far by some proof heuristic such as d-implication. If this fails, it guesses the value of the variable uniformly at random. However, even if the proof heuristic cannot determine the current single assignment, it may be possible to infer, e.g., $x = 0$ is more likely than $x = 1$, which we call in general "bias." Hansen, Kaplan, Zamir, and Zwick [3] introduced a systematic way to use such "bias", which we refer to as biased-PPSZ algorithms. Here we first explain *the generic biased-PPSZ algorithm* as a template of these biased-PPSZ algorithms, giving an idea as well as properties common to all biased-PPSZ algorithms.

In the execution of a biased-PPSZ algorithm in general, when a variable x needs to be guessed, it is assigned 0 with probability β_T and 1 with probability $1-\beta_T$, where β_T is

the "bias" of x determined based on its "type." Thus, roughly speaking, a biased-PPSZ algorithm is defined once we determine a way to give a type T to a guessed variable and a way to define its bias β_T. For example, PPSZ is regarded as a biased-PPSZ algorithm, where there is only one type and its bias is the trivial one, $1/2$.

Now we explain the generic biased-PPSZ algorithm that is regarded as a general template of all biased-PPSZ algorithms. **Algorithm 9** is its description. Let us first ignore the first for-loop, and consider the body of this loop starting from line 5. This part corresponds to the standard PPSZ process, which we call the *biased-PPSZ process*. At each iteration of this biased-PPSZ process, if the algorithm needs to guess the single assignment of a variable x, it determines a random value following a bias β_T that is given based on the "type" T of x. Here the key is the choice of β_T for each type T. Below we will drive the definition of the "best" biases $\{\beta_T^*\}_{T\in\mathcal{T}}$, where $\mathcal{T}$ is the set of all types used in the algorithm. As we will see, they are defined based on the statistical parameters of an input formula and its target satisfying assignment, and it is quite likely that some of such parameters are hard to compute. Thus, instead of computing these biases, we "guess" them with reasonable precision. In its actual execution, the algorithm tries all possible values of β_T from the set $\{1/e(n), \ldots, (e(n)-1)/e(n)\}$ for each type T. This is the role of the first for-loop. We need to execute the biased-PPSZ process for $e(n)^{|\mathcal{T}|}$ times, which is subexponential if $|\mathcal{T}| \log e(n) = o(n)$. Note that for the ideal bias value β_T^*, there is some choice of β_T that is $1/e(n)$-close to it, that is, $|\beta_T^* - \beta_T| \leq 1/e(n)$ holds. Thus, there is at least one execution of the biased-PPSZ process where β_T's are $1/e(n)$-close to β_T^*'s for all $T \in \mathcal{T}$. We will show that a similar success probability lower bound holds in this case provided that $e(n) = \omega(1)$. Note that even with wrong choices of biases, the biased-PPSZ process never outputs a wrong answer, i.e., an unsatisfying assignment.

To derive the ideal bias values, we analyze the success probability of one execution of the biased-PPSZ process by using biases $\{\beta_T\}_{T\in\mathcal{T}}$. The analysis of PPSZ extends naturally to this case. Recall (2.2). For each random permutation π, let $G(\pi, \alpha)$ denote

Algorithm 9 The generic biased-PPSZ algorithm

$\pi :=$ a random permutation on V
$\alpha' :=$ an empty assignment
for every choice of values $\beta_T \in \{\frac{1}{e(n)}, \frac{2}{e(n)}, \ldots, \frac{e(n)-1}{e(n)}\}$ for all $T \in \mathcal{T}$ **do**
    (*biased-PPSZ process*)
    for $x \in V$ in the order of π **do**
        if $P(x) \neq ?$ **then**
            assign $P(x)$ to x
        else
            identify the type T of x
            assign 0 to x with probability β_T, and
            assign 1 to x with probability $1 - \beta_T$
        end if
    end for
end for

the number of guessed variables in the PPSZ process following[2] π and α. Then the probability that α is produced by the PPSZ process following π is $2^{-G(\pi,\alpha)}$; this is because the probability that each guessed variable x is assigned $\alpha(x)$ is $1/2$. Then its expectation over π, i.e., $\mathbb{E}_\pi[2^{-G(\pi,\alpha)}]$, is the probability that the PPSZ returns α as stated (2.1). Here for each type $T \in \mathcal{T}$, let[3] $G_T(\pi)$ be the number of guessed variables of type T during the biased-PPSZ process following π and α. Additionally, let $G_T^{(0)}(\pi)$ and $G_T^{(1)}(\pi)$ be the number of guessed variables of type T whose values by α is 0 and 1, respectively. Note that $G_T(\pi) = G_T^{(0)}(\pi) + G_T^{(1)}(\pi)$. Then it is easy to see that $2^{-G(\pi,\alpha)}$ is now replaced with

$$\prod_{T \in \mathcal{T}} \beta_T^{G_T^{(0)}(\pi)} (1 - \beta_T)^{G_T^{(1)}(\pi)}.$$

[2]Recall that by "following α" we mean the PPSZ process where each guessed variable x is assigned $\alpha(x)$.

[3]Since α has been fixed throughout our analysis, we will omit specifying α.

Thus, by taking its expectation, we get the following generalization of (2.1) and (2.2).

$$\begin{aligned}\Pr[\text{the biased-PPSZ process yields } \alpha] &= \mathbb{E}_\pi\left[\prod_{T\in\mathcal{T}} \beta_T^{G_T^{(0)}(\pi)}\left(1-\beta_T\right)^{G_T^{(1)}(\pi)}\right] \\ &\geq \prod_{T\in\mathcal{T}} \beta_T^{\mathbb{E}_\pi\left[G_T^{(0)}(\pi)\right]}\left(1-\beta_T\right)^{\mathbb{E}_\pi\left[G_T^{(1)}(\pi)\right]} \\ &= 2^{\left(\sum_{T\in\mathcal{T}} \mathbb{E}_\pi\left[G_T^{(0)}(\pi)\right]\log(\beta_T)+\mathbb{E}_\pi\left[G_T^{(1)}(\pi)\right]\log(1-\beta_T)\right)}\end{aligned}$$

Let $G_T := \mathbb{E}_\pi[G_T(\pi)]$ and

$$\beta_T^* := \mathbb{E}_\pi\left[G_T^{(0)}(\pi)\right]/G_T. \tag{4.1}$$

Then the exponent of the above equation can be rewritten as

$$\sum_{T\in\mathcal{T}} G_T\cdot\left(\beta_T^*\log\left(\beta_T\right)+\left(1-\beta_T^*\right)\log\left(1-\beta_T\right)\right) \tag{4.2}$$

Let $f(x) := \beta_T^*\log(x)+(1-\beta_T^*)\log(1-x)$. Then it is easy to see that $f(x)$ is maximized at $x=\beta_T^*$. Hence, the best choice of β_T for the algorithm is $\beta_T=\beta_T^*$. That is, (4.1) is the ideal bias value. Now suppose that all β_T's are $1/e(n)$-close to β_T^*'s, and estimate how the exponent changes from the ideal case. By using the Talor series of $f(x)$ at β_T^*, we have

$$\begin{aligned}\beta_T^*\log\left(\beta_T\right)+\left(1-\beta_T^*\right)\log\left(1-\beta_T\right) &\geq \beta_T^*\log\left(\beta_T^*\right)+\left(1-\beta_T^*\right)\log\left(1-\beta_T^*\right)-O\left(\frac{1}{e(n)}\right) \\ &= -\mathrm{H}\left(\beta_T^*\right)-O\left(\frac{1}{e(n)}\right),\end{aligned}$$

where we use the binary entropy function $\mathrm{H}(x) := -x\log x-(1-x)\log(1-x)$ to simplify the expression. From this, the exponent (4.2) is evaluated as follows.

$$\begin{aligned}\sum_{T\in\mathcal{T}} G_T\cdot\left(\beta_T^*\log\left(\beta_T\right)+\left(1-\beta_T^*\right)\log\left(1-\beta_T\right)\right) &\geq -\sum_{T\in\mathcal{T}} G_T\cdot\left(\mathrm{H}\left(\beta_T^*\right)+O\left(\frac{1}{e(n)}\right)\right) \\ &\geq -\sum_{T\in\mathcal{T}} G_T\cdot\mathrm{H}\left(\beta_T^*\right)-O\left(\frac{n}{e(n)}\right).\end{aligned}$$

The last bound follows from the fact that $\sum_T G_T\leq n$. Therefore, if we can let $e(n)=\omega(1)$, the above error term can be bounded by $o(n)$. From this analysis, we have the following lemma.

Lemma 4.1. *Let $e(n)$ be any function such that $e(n) = \omega(1)$. For some choice of $\beta_T \in \{\frac{1}{e(n)}, \frac{2}{e(n)}, \ldots, \frac{e(n)-1}{e(n)}\}$ for all $T \in \mathcal{T}$, the biased-PPSZ process of the generic biased-PPSZ algorithm on F returns the unique satisfying assignment α with probability at least $2^{-(\sum_{T\in\mathcal{T}} G_T \cdot \mathrm{H}(\beta_T^*))-o(n)} = 2^{-((\sum_{T\in\mathcal{T}} G_T \cdot \mathrm{H}(\beta_T^*))+o(n))}$, where $\{\beta_T^*\}_{T\in\mathcal{T}}$ are the best biases defined by (4.1).*

Note that $\mathrm{H}(\beta_T^*) \leq 1$ and $\sum_{T\in\mathcal{T}} G_T = \mathbb{E}_\pi[G(\pi,\alpha)] \geq S_3 n + o(n)$. Thus, the advantage of a biased-PPSZ algorithm over PPSZ is determined by the amount that we can reduce $\sum_{T\in\mathcal{T}} G_T \cdot \mathrm{H}(\beta_T^*)$ from $S_3 n$.

4.2 The HKZZ_biased-PPSZ Algorithm

We explain the biased-PPSZ algorithm that Hansen et al. proposed in [3]. Below we refer to their algorithm as HKZZ_biased-PPSZ and their way of defining biases as HKZZ_bias. They show that the exponential coefficient can be improved by HKZZ_biased-PPSZ in general for any k-SAT. Here we omit explaining their analysis since we will give an improved result in Section 4.4 (though for Unique 3-SAT).

See **Algorithm 10** for the description of HKZZ_biased-PPSZ. A key idea of this algorithm is to use a "maximal set of disjoint clauses." We say that two clauses are *disjoint* if they have no variable in common. HKZZ_biased-PPSZ first computes a maximal set D of disjoint clauses of F; it simply collects clauses to D until no disjoint clause exists in $F\backslash D$. Let V_D denote the set of variables appearing in D, where $|V_D| = 3|D|$.

An important property here is that every clause in $F\backslash D$ has at least one variable in V_D. Using this property, we can design an efficient algorithm if $|D|$ is small, say, less than γn for some small constant γ. First, note that there are $7^{|D|}$ assignments[4] to V_D that satisfy all clauses of D, and it is easy to enumerate them. Hence, for each such assignment α', we can check whether it can be extended to satisfy remaining clauses $F\backslash D$, in other

[4]Baumer and Schuler [1] gave a better way to select a disjoint clause set, by which the number of possible assignments can be bounded by $6^{|D|}$, which immediately gives a better exponential coefficient $0.386249\cdots$ when used with the biased-PPSZ part of HKZZ_biased-PPSZ. Note, on the other hand, that our improvement introduced next section gives a better exponential constant.

Algorithm 10 HKZZ_biased-PPSZ

1: compute a maximal set D of disjoint clauses of F
2: **if** $|D| \leq \gamma n$ **then** (*brute force search part*)
3: **for** all possible satisfying assignments α' of D **do**
4: search for a sat. assignment α'' of $F|_{\alpha'}$ and return $\alpha' \cup \alpha''$ if α'' is obtained
5: **end for**
6: **else** (*biased-PPSZ part*)
7: $\pi :=$ a random permutation on V
8: $\alpha' :=$ an empty assignment
9: **for** every weight vector (W_1, W_2, W_3),
10: every threshold time $t \in (0, 1)$, and
11: every choice of values for $\{\beta_P\}_{P \in \mathcal{P}}$ **do**
12: execute the biased-PPSZ process of the generic biased-PPSZ
13: **end for**
14: **end if**

words, whether $(F \backslash D)|_{\alpha'}$ is satisfiable. Since D is a set of disjoint clauses in F, the remaining clauses in $F \backslash D$ are regarded as a 2-CNF formula. Note that since 2-SAT is polynomial-time solvable, checking whether $(F \backslash D)|_{\alpha'}$ is satisfiable (and computing its satisfying assignment) can be done in polynomial-time. Thus, the total running time is of this algorithm is bounded by $7^{\gamma n} \cdot \mathrm{poly}(n)$, and we can easily convert this deterministic algorithm to a randomized algorithm whose success probability is at least $7^{-\gamma n}$. Hence, if $\gamma < 0.1$ for example, this approach is much better than the existing algorithms.

We use the generic biased-PPSZ with HKZZ_bias for the case when $|D| > \gamma n$, which we refer *the biased-PPSZ part* starting from line 9 of **Algorithm 10**. Before executing this part (for simplifying the following discussion), we assume that the algorithm flips negated variables appearing in D so that all clauses of D consist of only nonnegative variables (which is possible since all clauses in D are disjoint).

Now we explain how to define HKZZ_bias. We consider only variables in V_D. For the other variables, we use the trivial bias 1/2; that is, these variables are assigned 0

or 1 uniformly at random if they have to be guessed. As we discussed for the generic biased-PPSZ algorithm, a bias value is determined depending on the variable's "type." Suppose that a variable $x \in V_D$ is chosen as the next and guessed variable. At this point, we need to determine its type. Note that x appears in exactly one clause C of D because all clauses of D are disjoint. Let $C = (x \vee y \vee z)$ be this clause. At the point when x is chosen, some of the other variables in C, i.e., y and z, might have been already assigned values[5]. We use the pattern of this assignment as a type, which we call the "prefix" of x. More specifically, the *prefix* of x is the pattern of fixed literal values in the order of π. For example, if $\alpha(y) = 0$ and $\alpha(z) = 1$, and if $\pi(y) < \pi(z) < \pi(x)$, then the prefix P of x (when x is chosen) is 01; and if $\pi(z) < \pi(x) < \pi(y)$, then $P = 1$. The domain of prefixes is $\mathcal{P} := \{\emptyset, 0, 1, 00, 01, 10, 11\}$; $\emptyset$ denotes the case where x is the first variable in C chosen in biased-PPSZ process (i.e., $\pi(x) < \pi(y), \pi(z)$).

Now that we have defined our types, the best biases $\{\beta^*_P\}_{P\in\mathcal{P}}$ are defined by (4.1) as we discussed for the generic biased-PPSZ algorithm. But some additional factors are considered in HKZZ_bias. The first one is a "weight distribution" on clauses in D. The *weight* of a clause is the number of literals evaluated 1 under the unique satisfying assignment α. For a given set D of disjoint clauses, its *weight vector* is a triple (W_1, W_2, W_3), where for each $i \in \{1, 2, 3\}$, W_i is the number of clauses of D with weight i. Since all clauses of D have a positive weight, we have $W_1 + W_2 + W_3 = |D|$; hence, the vector is determined by two parameters such as W_1 and W_2. A weight vector is a statistical property of a given formula; in HKZZ_bias, we consider that the best bias values are chosen depending on[6] each weight vector. The second factor is the "time" that a variable x is chosen. For a given parameter t (which we call a *time threshold*) that is fixed "appropriately" from $(0, 1)$, we determine the bias of x depending on whether

[5]We may assume that these values are correct since we consider only the execution where all (so far) guessed assignments are assigned consistent with the unique satisfying assignment α.

[6]This interpretation is in fact for the sake of our analysis because the best biases (more precisely, their $1/e(n)$-close approximation) are chosen (based on the brute force search) for each given input formula F anyway, and there is no need to choose them depending on F's weight vector (which again is chosen based on the brute force search as explained below).

$\pi(x) \leq t$ or not. In HKZZ_bias, biases $\{\beta_P\}_{P\in\mathcal{P}}$ are considered only variables x such that $\pi(x) \leq t$. For variables that are chosen later are given again the trivial bias 1/2.

We need to explain a way to determine a weight vector and a threshold t. Formally speaking, they need to be chosen (and fixed) before determining biases. Clearly, it is hard to determine F's weight vector correctly and choose a time threshold appropriately. Thus, here again, we "guess" them. For a weight vector, there are at most n^2 possibilities, we can try all possible values, and one of them (and exactly one of them) must be the correct one. On the other hand, for a time threshold, we simply try all possible values from some finite set TH, where TH $:= \{\tau, 2\tau, \ldots, \tau\lfloor 0.5/\tau \rfloor\}$ for some constant τ. (We do not have to consider a time threshold greater than 0.5 since it is known that any variable x with $\pi(x) > 0.5$ is forced with probability 1 by **Lemma 2.15**.) While we may expect a better result by choosing smaller τ, from our numerical analysis, we can get a reasonable improvement by using $\tau = 1.0 \times 10^{-2}$.

Here is an intuitive idea of these types. Suppose that W_1 is large for the input formula F; that is, it has many clauses in D that have only one variable that is assigned 1 under α. (Recall that all clauses of D consists of non-negated literals.) Suppose that the prefix of a variable x when it is chosen in the biased-PPSZ process is 1. Then it is more likely that $\alpha(x) = 0$. More specifically, it would be better to determine the assignment of x proportional to the fraction of variables that should be assigned 0 (resp., 1) under α among all such variables with prefix 1 (which idea is indeed the fact that β^*_T is the best choice). Of course, this fraction would vary depending on the timing when x is considered in the biased-PPSZ process. For this, HKZZ_bias uses the time threshold t and classifies a variable x whether it appears in the biased-PPSZ process early (i.e., $\pi(x) \leq t$) or not. Intuitively, a variable appearing later is likely forced. Thus, the trivial bias is used (for simplifying our analysis) by HKZZ_bias if $\pi(x) > t$.

For our later discussion, we introduce type $\overline{D}$ and $\perp$ for variables $x \notin V_D$ and for variables x in V_D with $\pi(x) > t$ respectively. Formally speaking, the set of types is $\mathcal{P} \cup \{\perp, \overline{D}\}$; but we will keep using $\mathcal{P}$ as the domain of types for which nontrivial biases are used. We point out here that $|\mathcal{P}|$ is constant (hence, we can choose $e(n) = \omega(1)$),

and that it is easy to determine the type of x when it is chosen in the biased-PPSZ process.

In [3], Hansen et al. prove that the algorithm HKZZ_biased-PPSZ gives an exponential coefficient better than PPSZ for Unique k-SAT. In particular, by numerical analysis, they show [10] that the algorithm guarantees the exponential coefficient $\epsilon_{\mathrm{HKZZ}} := 0.386254$ for Unique 3-SAT (as stated in Table 1.1).

4.3 The biased-Search and QW_biased-PPSZ Algorithm

We present a new method called "biased-Search" as an additional way to improve HKZZ_biased-PPSZ. The idea is simple. In HKZZ_biased-PPSZ, in the case where the obtained set D of disjoint clauses is small, a satisfying assignment is searched in a brute force way. We propose an algorithm for conducting this search more efficiently when there is some "bias." Let us call this new search algorithm the *biased-Search*.

We should mention here that the idea similar to this has been proposed by Hofmeister, Schöning, Schuler, and Watanabe in [7], where they proposed to use a search algorithm that is essentially the same as our biased-Search with the random walk type algorithm to improve its success probability. Here we applied the same idea for improving HKZZ_Biased-PPSZ.

See **Algorithm 11** for the description of the biased-Search. The algorithm chooses an assignment to variables in V_D uniformly at random from all possible choices of D_1, D_2, and D_3 and all possible choices of literals evaluated as 1 (resp., 0) in clauses of D_1 (resp., D_2). After this random selection of an assignment α' satisfying clauses in D, the algorithm simply executes a 2-SAT solver for computing the sat. assignment α'' of the remaining clauses, i.e., $F|_{\alpha'}$, and it returns $\alpha' \cup \alpha''$ if α'' is obtained.

Consider the success probability of the biased-Search. This is nothing but the probability that all variables in V_D are assigned correctly. Note that the correct assignment to V_D is exactly one of the above random choices. Letting $M := |D| = W_1 + W_2 + W_3$,

Algorithm 11 The Biased-Search

(Assume that (W_1, W_2, W_3) is the correct weight vector)

$D_1 := W_1$ clauses randomly chosen from D

determine randomly single assignments to all variables in D_1

so that each clause has exactly one literal evaluated 1

$D_2 := W_2$ clauses randomly chosen from $D \setminus W_1$

determine randomly single assignments to all variables in D_2

so that each clause has exactly one literal evaluated 0

$D_3 := D \setminus (W_1 \cup W_2)$

determine single assignments to all variables in D_3

so that all literals of each clause are evaluated 1

$\alpha' :=$ the obtained assignment to all variables in V_D

search for a sat. assignment α'' of $F|_{\alpha'}$ and return $\alpha' \cup \alpha''$ if α'' is obtained

this probability is expressed by

$$\left(\frac{M!}{W_1!W_2!W_3!} \cdot 3^{W_1} \cdot 3^{W_2}\right)^{-1}.$$

This is approximated (by Stirling's approximation) and bounded as follows, ignoring some $(1 - \mathrm{poly}(n)^{-1})$-factor that is negligible for evaluating the exponential coefficient.

$$\begin{aligned}\left(\frac{M!}{W_1!W_2!W_3!} \cdot 3^{W_1} \cdot 3^{W_2}\right)^{-1} &\approx \left(\frac{e}{M}\right)^M \left(\frac{W_1}{3e}\right)^{W_1} \left(\frac{W_2}{3e}\right)^{W_2} \left(\frac{W_3}{e}\right)^{W_3} \\ &\geq \left(\frac{e}{M}\right)^M \left(\frac{W_1 + W_2 + W_3}{W_1 \frac{3e}{W_1} + W_2 \frac{3e}{W_2} + W_3 \frac{e}{W_3}}\right)^{(W_1+W_2+W_3)} \\ &= \left(\frac{1}{3+3+1}\right)^M = 7^{-M}.\end{aligned} \tag{4.3}$$

The second inequality is from the relation between arithmetic and geometric means, and the equality (i.e., the worst-case) holds if $(w_1, w_2, w_3) = (3/7, 3/7, 1/7)$. It is easy to see that the success probability of biased-Search improves exponentially by any weight vector deviate from the above worst-case by a small constant. On the other hand, the numerical analysis of Section 4.4 shows that this worst-case weight vector is preferable for

(the biased-PPSZ part of) HKZZ_biased-PPSZ, and its success probability gets improved exponentially. Therefore, we can conclude that an exponential coefficient better than HKZZ_biased-PPSZ is obtained by this additional improvement (at least for Unique 3-SAT).

Next, consider an algorithm including this biased-Search, which we call QW_biased-PPSZ. See **Algorithm 12** for the description of QW_biased-PPSZ. Here we simply execute both the biased-Search and the biased-PPSZ part of HKZZ_biased-PPSZ by omitting the decision whether D is small or not in order to avoid estimating the parameter γ used in HKZZ_biased-PPSZ[7].

Algorithm 12 The QW_biased-PPSZ Algorithm

compute a maximal set D of disjoint clauses of F

for every weight vector (W_1, W_2, W_3) **do**

execute the biased-Search (and return the sat. assignment if it finds)

execute the biased-PPSZ part of HKZZ_biased-PPSZ

using the weight vector (W_1, W_2, W_3) (and return the sat. assignment if it finds)

end for

Now we analyze the success probability of this QW_biased-PPSZ, that is, the probability that QW_biased-PPSZ outputs α.

We consider the situation where the weight vector (W_1, W_2, W_3) is correct and for some $e(n) = \omega(1)$, the biases $\{\beta_P\}_{P\in\mathcal{P}}$ are all $1/e(n)$-close to the ideal ones $\{\beta^*_P\}_{P\in\mathcal{P}}$ w.r.t. (W_1, W_2, W_3) and t. Since the difference from the ideal ones can be ignored as shown in Section 4.1, we simply assume here that $\{\beta_P\}_{P\in\mathcal{P}}$ are the ideal ones defined by (4.1). We express various parameters used in the analysis in a fractional way. Let $\boldsymbol{w} = (w_1, w_2, w_3)$ is the fractional version of (W_1, W_2, W_3); that is, $w_i = W_i/|D|$. Let $\gamma := |D|/n$ so that the size of D is expressed as γn. Recall that in the analysis of the success probability of the biased-PPSZ process, we used variables, e.g., $G^{(0)}_T(\pi)$, the number of guessed variables of type T whose values by α is 0. Here we consider their

[7]In fact, we could have stated HKZZ_biased-PPSZ in the same way; but we followed the original explanation of [3] for the sake of intuitively clear explanation.

fractional version. For any $P \in \mathcal{P}$ and $b \in \{0,1\}$, let $G_P^{(b)}(\pi)$ be the number of guessed variables x of type P in V_D such that $\pi(x) \le t$ and $\alpha(x) = b$ in the biased-PPSZ process following π. Also let $G_\perp(\pi)$ (resp., $G_{\overline{D}}(\pi)$) the number of guessed variables x such that $x \in V_D$ and $\pi(x) > t$ (resp., $x \notin V_D$). Then define the following parameters.

$$g_P^b(\pi) := G_P^{(b)}(\pi)/|V_D|, \quad g_P^b := \mathbb{E}_\pi\left[g_P^b(\pi)\right], \quad g_P := g_P^0 + g_P^1,$$
$$g_\perp := \mathbb{E}_\pi\left[G_\perp(\pi)/|V_D|\right], \quad \text{and} \quad g_{\overline{D}} := \mathbb{E}_\pi\left[G_{\overline{D}}(\pi)/|V \setminus V_D|\right].$$

Then for the ideal bias (which is now denoted by β_P), we have

$$\beta_P = \frac{g_P^0}{g_P} = \frac{g_P^0}{g_P^0 + g_P^1}. \tag{4.4}$$

Let us analyze the success probability of each part; more specifically, its exponent w.r.t. base 2. First for the biased-Search, by taking the log of (4.3), the success probability exponent (ignoring the $o(n)$ term) can be expressed by

$$\begin{aligned}
&\log\left(\frac{e}{M}\right)^M \left(\frac{W_1}{3e}\right)^{W_1} \left(\frac{W_2}{3e}\right)^{W_2} \left(\frac{W_3}{e}\right)^{W_3} \\
&\quad = -|D|\log|D| + \sum_{i=1}^{3} W_i \log W_i - (W_1 + W_2)\log 3 \\
&\quad = -|D|\left(-\sum_{i=1}^{3} w_i \log w_i + (w_1 + w_2)\log 3\right) = -\gamma s_{\boldsymbol{w}} n,
\end{aligned} \tag{4.5}$$

where

$$s_{\boldsymbol{w}} := (w_1 + w_2)\log 3 - \sum_{i=1}^{3} w_i \log w_i. \tag{4.6}$$

Secondly, for the biased-PPSZ part, we follow the analysis of the genetic biased-PPSZ and use the fact that the exponent of the success probability is at least $-\left(\sum_{T\in\mathcal{T}} G_T \cdot \mathrm{H}(\beta_T^*) + o(n)\right)$ (**Lemma 4.1**). This bound (ignoring the minor $o(n)$ term)

is restated by using the current notation as follows.

$$
\begin{aligned}
-\sum_{T\in\mathcal{T}} G_T \cdot \mathrm{H}\left(\beta_T^*\right) &= -\sum_{P\in\mathcal{P}} g_P\left|V_D\right|\cdot \mathrm{H}(\beta_P) - g_\perp\left|V_D\right| - g_{\overline{D}}|V\setminus V_D| \\
&\geq -\sum_{P\in\mathcal{P}} g_P\left|V_D\right|\cdot \mathrm{H}(\beta_P) - g_\perp\left|V_D\right| - (S_3+o(1))|V\setminus V_D| \qquad (4.7)\\
&= -3\gamma n\left(\sum_{P\in\mathcal{P}} g_P \mathrm{H}(\beta_P) + g_\perp\right) - (n-3\gamma n)(S_3+o(1)) \\
&= -\left(S_3 - 3\gamma\left(S_3 - \sum_{P\in\mathcal{P}} g_P\mathrm{H}(\beta_P) - g_\perp\right)\right)n - o(n). \qquad (4.8)
\end{aligned}
$$

Note that $g_{\overline{D}}|V\setminus V_D| = \mathbb{E}_\pi\left[G_{\overline{D}}(\pi)\right]$ is simply the expected number of guessed variables in $V\setminus V_D$, which can be bounded by $(S_3+o(1))|V\setminus V_D|$ in the same way as proving **Theorem 2.9**. The bound of (4.7) follows from this observation.

Define $\Delta_{\boldsymbol{w},t}$ by

$$\Delta_{\boldsymbol{w},t} := S_3 - \sum_{P\in\mathcal{P}} g_P\mathrm{H}(\beta_P) - g_\perp. \qquad (4.9)$$

as the variable-wise advantage of using HKZZ_bias under the time threshold t for instances with weight vector $\boldsymbol{w}$. We may assume that the algorithm uses the best threshold in TH; hence,

$$\Delta_{\boldsymbol{w}} := \max_{t\in\mathrm{TH}} \Delta_{\boldsymbol{w},t} \qquad (4.10)$$

is the variable-wise advantage of using HKZZ_bias for instances with weight vector $\boldsymbol{w}$, and with this we express the success probability exponent of the biased-PPSZ part as follows.

$$(4.8) = -\left(S_3 - 3\gamma\Delta_{\boldsymbol{w}} + o(1)\right)n. \qquad (4.11)$$

Since the algorithm executes both parts, the total success probability lower bound is the largest one of $2^{(4.5)}$ and $2^{(4.11)}$. The worst situation is the case when they are balanced; more specifically, when the size of D (or, its size parameter γ) satisfies

$$\gamma s_{\boldsymbol{w}} = S_3 - 3\gamma\Delta_{\boldsymbol{w}}.$$

Let $\gamma_{\boldsymbol{w}}$ denote γ satisfying the above; that is, it is defined by

$$\gamma_{\boldsymbol{w}} := \frac{S_3}{s_{\boldsymbol{w}} + 3\Delta_{\boldsymbol{w}}}. \tag{4.12}$$

From the viewpoint of instances, an instance F with a disjoint clause set D of size $\gamma_{\boldsymbol{w}} n$ (if it exists) is the worst-case instance among those with weight vector $\boldsymbol{w}$.

What we show here is that for any weight vector $\boldsymbol{w}$, the success probability of QW_biased-PPSZ is at least

$$2^{-(\gamma_{\boldsymbol{w}} s_{\boldsymbol{w}} + o(1))n} = 2^{-(S_3 - 3\gamma_{\boldsymbol{w}}\Delta_{\boldsymbol{w}} + o(1))n}.$$

From this, we define $\boldsymbol{w}_{\mathrm{wst}}$ by

$$\boldsymbol{w}_{\mathrm{wst}} := \mathrm{argmin}_{\boldsymbol{w}} 3\gamma_{\boldsymbol{w}}\Delta_{\boldsymbol{w}}, \tag{4.13}$$

and claim that

$$S_3 - 3\gamma_{\boldsymbol{w}_{\mathrm{wst}}}\Delta_{\boldsymbol{w}_{\mathrm{wst}}}$$

is an exponential coefficient, that is, a lower bound of the coefficient of n in the base 2 exponent of the success probability of QW_biased-PPSZ on Unique 3-SAT instances. We state this as the following theorem.

Theorem 4.2. *Use symbols defined by (4.10), (4.12), and (4.13). For any instance of Unique 3-SAT, the algorithm QW_biased-PPSZ outputs its satisfying assignment (if it exists) with probability at least* $2^{-(S_3 - 3\gamma_{\boldsymbol{w}_{\mathrm{wst}}}\Delta_{\boldsymbol{w}_{\mathrm{wst}}} + o(1))n}$. *Thus,* $S_3 - 3\gamma_{\boldsymbol{w}_{\mathrm{wst}}}\Delta_{\boldsymbol{w}_{\mathrm{wst}}}$ *can be regarded as an exponential coefficient of QW_biased-PPSZ.*

We may regard $3\gamma_{\boldsymbol{w}_{\mathrm{wst}}}\Delta_{\boldsymbol{w}_{\mathrm{wst}}}$ as the improvement of QW_biased-PPSZ. In the next section, we obtain the actual value of the above exponential coefficient by evaluating $\Delta_{\boldsymbol{w},t}$ (defined by (4.9)) for each t and $\boldsymbol{w}$ by some numerical method.

4.4 Numerical Analysis for an Exponential Coefiicient of QW_biased-PPSZ

Hansen et al. [3] claimed the exponential coefficient ϵ_{HKZZ} ($= 0.386254$) of their biased PPSZ algorithm (more specifically, HKZZ_biased-PPSZ) for Unique 3-SAT. This bound

is obtained by numerical analysis given by Zamir in his doctoral thesis [10]. Here in order to show the actual effect of our improvement, we follow Zamir's thesis [10] to derive an exponential coefficient ϵ_{QW} of our algorithm QW_biased-PPSZ for Unique 3-SAT.

4.4.1 Zamir's numerical analysis

We explain Zamir's numerical analysis applied to our algorithm, in particular, its biased-PPSZ part.

First, we prepare some notations and recall some key facts for our discussion. Throughout this section, we fix the domain of type P to $\mathcal{P}$, and we omit to specify it. Also, the usage of the symbol b is fixed to denote a Boolean value, i.e., 0 or 1; thus, we omit to specify its domain. Let t be any time threshold in the set TH; recall that $t \leq 0.5$. We use Guessed(π) and Forced(π) to denote the sets of variables that are guessed and forced respectively in the biased-PPSZ process following π and α.

The following lemma can be proved as a generalization of **Lemma 2.15**.

Lemma 4.3. *For any variable x, we have*

$$\Pr[x \in \mathrm{Forced}(\pi) | \pi(x) = p] \geq \left(\frac{p}{1-p}\right)^2 - o(1), \quad \text{and}$$
$$\Pr[x \in \mathrm{Guessed}(\pi) | \pi(x) = p] \leq 1 - \left(\frac{p}{1-p}\right)^2 + o(1).$$

Then it follows that for any variable x, the probability that x appears as a forced variable before the time threshold t is bounded as follows.

$$\Pr[x \text{ appears as a forced variable before time } t] \geq \int_0^t \left(\frac{p}{1-p}\right)^2 dp - o(1). \tag{4.14}$$

Now consider the difference of the success probability exponent between HKZZ_bias and PPSZ; more precisely, the variable-wise advantage on the variables in V_D, that is, $\Delta_{\boldsymbol{w},t}$ defined by (4.9). We can restate it as follows.

$$\begin{aligned}\Delta_{\boldsymbol{w},t} &= S_3 - \left(\sum_P g_P \mathrm{H}\left(\beta_P\right) + g_\perp\right) \\ &\geq S_3 - \left(\sum_P g_P \mathrm{H}\left(\beta_P\right) + S_3 - t + \int_0^t \left(\frac{p}{1-p}\right)^2 dp\right) \\ &= t - \int_0^t \left(\frac{p}{1-p}\right)^2 dp - \sum_P g_P \mathrm{H}\left(\beta_P\right). \end{aligned} \tag{4.15}$$

Here we use the above lemma and **Lemma 3.1** for bounding $g_\perp$ by

$$\begin{aligned} g_\perp &= \int_t^1 \Pr[x \text{ is guessed} | \pi(x) = p] dp \leq \int_t^1 \left(1 - \min\left(\left(\frac{p}{1-p}\right)^2, 1\right)\right) dp + o(1) \\ &= \int_0^1 \left(1 - \min\left(\left(\frac{p}{1-p}\right)^2, 1\right)\right) dp - \int_0^t \left(1 - \min\left(\left(\frac{p}{1-p}\right)^2, 1\right)\right) dp + o(1) \\ &= S_3 - t + \int_0^t \left(\frac{p}{1-p}\right)^2 dp + o(1), \end{aligned}$$

where the last equality holds since we assume that $t \leq 1/2$. Recall that by "$o(1)$" we mean a term that goes to 0 when n gets large. Here and in the following discussion, we may assume that n is large enough so that $o(1)$ is negligible, or more precisely, this term is within the margin introduced by rounding up the final bound.

Our plan is to give a lower bound of $\Delta_{\boldsymbol{w},t}$ in terms of only (besides t) the weight vector $\boldsymbol{w} := (w_1, w_2, w_3)$, i.e., the fractions of clauses of D with weights $1 \sim 3$. We focus on $\sum_P g_P \mathrm{H}(\beta_P) = -\sum_P \left(g_P^0 \log\left(\frac{g_P^0}{g_P}\right) + g_P^1 \log\left(\frac{g_P^1}{g_P}\right)\right)$. Consider any g_P^b. Note that it is the expected fraction of $x \in V_D$ such that (i) $\pi(x) \leq t$, (ii) $\alpha(x) = b$, and (iii) x appears as a guessed variable with prefix P in the biased-PPSZ process. While this g_P^b depends on the input formula F, we can express a similar fraction by only using $\boldsymbol{w}$. Define v_P^b be the expected fraction of $x \in V_D$ such that (i) $\pi(x) \leq t$, (ii) $\alpha(x) = b$, and

(iii)′ x appears with prefix P. We can express these v_P^b's as follows.

$$v_\emptyset^0 = \left(\frac{2}{3}w_1 + \frac{1}{3}w_2\right)\int_0^t (1-p)^2 dp \qquad v_\emptyset^1 = \left(\frac{1}{3}w_1 + \frac{2}{3}w_2 + w_3\right)\int_0^t (1-p)^2 dp$$

$$v_0^0 = \frac{1}{3}w_1 \int_0^t 2p(1-p)dp \qquad v_0^1 = \left(\frac{1}{3}w_1 + \frac{1}{3}w_2\right)\int_0^t 2p(1-p)dp$$

$$v_1^0 = \left(\frac{1}{3}w_1 + \frac{1}{3}w_2\right)\int_0^t 2p(1-p)dp \qquad v_1^1 = \left(\frac{1}{3}w_2 + w_3\right)\int_0^t 2p(1-p)dp$$

$$v_{00}^1 = \frac{1}{3}w_1 \int_0^t p^2 dp$$

$$v_{01}^0 = \frac{1}{3}w_1 \int_0^t p^2 dp \qquad v_{01}^1 = \frac{1}{3}w_2 \int_0^t p^2 dp$$

$$v_{10}^0 = \frac{1}{3}w_1 \int_0^t p^2 dp \qquad v_{10}^1 = \frac{1}{3}w_2 \int_0^t p^2 dp$$

$$v_{11}^0 = \frac{1}{3}w_2 \int_0^t p^2 dp \qquad v_{11}^1 = w_3 \int_0^t p^2 dp$$

We explain a reason, e.g., for the estimate of v_1^0. The factor $\int_0^t 2p(1-p)dp$ is the probability, for any given $x \in V_D$ (letting C be the clause of D that contains x), that $\pi(x) = p \leq t$ and exactly one of the other two variables of C has appeared before time p. On the other hand, $(w_1/3 + w_2/3)$ is the expected fraction of clauses in D with assignment pattern 10 under the condition that two of its variables are assigned. Hence, by multiplying them, we get the expected fraction v_P^b of $x \in V_D$ that satisfies the conditions (i), (ii), and (iii') for $P = 1$ and $b = 0$.

Here we consider this estimate more carefully for later reference. Again we focus on v_1^0. Consider any $x \in V_D$, and let $C = (x \vee y \vee z)$ be the clause of D that contains x. Suppose that $\alpha(x) = 0$, $\alpha(y) = 0$, and $\alpha(z) = 1$; hence, C's weight is 1. Then x is counted v_1^0 if and only if (i) $\pi(x) \leq t$ (which is a probabilistic event), (ii) $\alpha(x) = 0$ (which holds from our assumption), and (iii') its appears with prefix 1 (which is the probabilistic event that $\pi(z) < \pi(x) < \pi(y)$). For any $p \leq t$, under the condition $\pi(x) = p$, the probability that one of the variables y and z appears earlier than x is $2p(1-p)$. Hence, $\int_0^t 2p(1-p)dp$ is the probability that $\pi(x) \leq t$ and one of the variables y and z appears earlier than x. Then under this condition, the probability that z is the variable that comes faster than x (which is equivalent to the event $\pi(z) < \pi(y)$ occurs) is

1/2. Thus, for this x, the probability that (i), (ii), and (iii') hold is $(1/2)\int_0^t 2p(1-p)dp$. Note that there are $2W_1$ variables that are assiged 0 under α in weight one clauses of D. Hence, the expected number of variables in weight one clauses of D satisfying (i), (ii), and (iii') is

$$2W_1 \cdot \frac{1}{2}\int_0^t 2p(1-p)dp = W_1 \int_0^t 2p(1-p)dp.$$

Then dividing this by $|V_D| = 3(W_1 + W_2 + W_3)$, we get $(w_1/3)\int_0^t 2p(1-p)dp$ as the expected fraction of variables in weight one clauses satisfying (i), (ii), and (iii'). The other term $(w_2/3)\int_0^t 2p(1-p)dp$ is derived similarly, and this is how v_1^0 is estimated as the above.

Clearly, the difference between g_P^b and v_P^b is due to the conditions (iii) and (iii)$'$; then we can see that $g_P^b = v_P^b - f_P^b$, where f_P^b is the expected fraction of variable $x \in V_D$ satisfying (i), (ii), (iii)$'$, and *forced*. We use a vector notation to express the values of $\{f_P^b\}_{P\in\mathcal{P},b\in\{0,1\}}$; more concretely, we express them by $\boldsymbol{f} = (f_\emptyset^0, f_\emptyset^1, f_0^0, f_0^1, f_1^0, f_1^1, f_{00}^0, f_{00}^1, f_{01}^0, f_{01}^1, f_{10}^0, f_{10}^1, f_{11}^0, f_{11}^1)$, and, for example, the forth component of $\boldsymbol{f}$ is called the $(0,1)$th component. Then we have

$$\begin{aligned}\sum_P g_P \mathrm{H}(\beta_P) &= -\sum_P \left(g_P^0 \log\left(\frac{g_P^0}{g_P}\right) + g_P^1 \log\left(\frac{g_P^1}{g_P}\right)\right) \\ &= \sum_P \mathrm{h}\left(g_P^0, g_P^1\right) \\ &= \sum_P \mathrm{h}\left(v_P^0 - f_P^0, v_P^1 - f_P^1\right), \end{aligned} \tag{4.16}$$

where we define $\mathrm{h}(x,y) := -x\log(x/(x+y)) - y\log(y/(x+y))$.

We say that a vector $\boldsymbol{f}$ is *realizable* if there is some (F,D) with weight $\boldsymbol{w}$ such that each component f_P^b is the expected fraction of variables x such that (i) $\pi(x) \le t$, (ii) $\alpha(x) = b$, and (iii) x appears as a forced variable of type P. Among all realizable $\boldsymbol{f}$, define $\boldsymbol{f}_{\mathrm{wst}}$ be the one that maximizes $\sum_P g_P \mathrm{H}(\beta_P) = \sum_P \mathrm{h}(v_P^0 - f_P^0, v_P^1 - f_P^1)$. Then this, i.e., $\sum_P \mathrm{h}(v_P^0 - f_{\mathrm{opt,P}}^0, v_P^1 - f_{\mathrm{opt,P}}^1)$, gives a lower bound of $\Delta_{\boldsymbol{w},t}$ by (4.15). In the following, we let

$$\mathrm{h}(\boldsymbol{f}) := \sum_P \mathrm{h}(v_P^0 - f_P^0, v_P^1 - f_P^1),$$

and call it an *objective function*[8]. It seems, however, difficult to compute $\boldsymbol{f}_{\mathrm{wst}}$; thus, we aim to compute $\boldsymbol{f}_{\mathrm{opt}}$ such that $\mathrm{h}(\boldsymbol{f}_{\mathrm{opt}}) \geq \mathrm{h}(\boldsymbol{f}_{\mathrm{wst}})$ and $\mathrm{h}(\boldsymbol{f}_{\mathrm{opt}}) - \mathrm{h}(\boldsymbol{f}_{\mathrm{wst}})$ is small enough, which can be used as a good upper bound of $\mathrm{h}(\boldsymbol{f})$ that can be used to derive a lower bound of $\Delta_{\boldsymbol{w},t}$.

We give below a sweep-line algorithm for computing $\boldsymbol{f}_{\mathrm{opt}}$. First, we show some facts that are useful for discussing our sweep-line algorithm, and define $\boldsymbol{f}_{\mathrm{opt}}$ precisely. Consider any realizable $\boldsymbol{f}$. By definition (i.e., from the meaning of $\boldsymbol{f}$), we have $0 \leq f_P^b \leq v_P^b$ for any P and b. These bounds are "trivial constraints" for $\boldsymbol{f}$. On the other hand, we have some nontrivial constraints for $\boldsymbol{f}$. Recall that $\left(\sum_{P,b} f_P^b\right) |V_D|$ is the expected number of forced variables x such that $\pi(x) \leq t$. Then it is at least $\left(\int_0^t p^2/(1-p)^2 dp - o(1)\right) |V_D|$ because the probability that x appears as a forced variable before time t is at least $\int_0^t p^2/(1-p)^2 dp - o(1)$ for every variable x from (4.14). Hence, we have $\sum_{P,b} f_P^b \geq \int_0^t p^2/(1-p)^2 dp$ ignoring the $o(1)$ term. On the other hand, from the analysis of the objective function $\mathrm{h}(\boldsymbol{f})$ we will show later (Fact 4.13), $\mathrm{h}(\boldsymbol{f})$ gets larger by $\boldsymbol{f}$ with smaller $\sum_{P,b} f_P^b$. Thus, we use

$$\sum_{P,b} f_P^b = \int_0^t \left(\frac{p}{1-p}\right)^2 dp. \tag{4.17}$$

as a key constraint for $\boldsymbol{f}_{\mathrm{opt}}$.

Remark 4.1. While $\boldsymbol{f}_{\mathrm{wst}}$ satisfies $\sum_{P,b} f_P^b \geq \int_0^t p^2/(1-p)^2 dp + o(1)$ (and hence, we have $\mathrm{h}(\boldsymbol{f}_{\mathrm{opt}}) \geq \mathrm{h}(\boldsymbol{f}_{\mathrm{wst}})$ by $\boldsymbol{f}_{\mathrm{opt}}$ satisfying (4.17) from Fact 4.13), it is quite likely that (4.17) does not hold for $\boldsymbol{f}_{\mathrm{wst}}$. In general we introduce constraints to make $\mathrm{h}(\boldsymbol{f}_{\mathrm{opt}})$ closer to $\mathrm{h}(\boldsymbol{f}_{\mathrm{wst}})$ while keeping $\mathrm{h}(\boldsymbol{f}_{\mathrm{opt}}) \geq \mathrm{h}(\boldsymbol{f}_{\mathrm{wst}})$. Furthermore, we introduce only those easy to use. (Note that $h(\boldsymbol{f})$ can take a larger value under weaker constraints.) For example, we know that for every variable, the probability that it appears as a forced variable before time t is at least $\int_0^t p^2/(1-p)^2 dp - o(1)$; hence, we could introduce this as a constraint. We consider, however, the above constraint only for $\sum_{P,b} f_P^b$.

The next constraints are from some lemmas in [3, 10]. We first consider the simplest

[8]When $\mathrm{h}(\boldsymbol{f})$ is regarded as an objective function, the components of $\boldsymbol{f} = (f_\emptyset^0, \ldots, f_{11}^1)$ are regarded as variables.

case. Here consider any clause C of D, and let x, y, and z be three variables appearing in C; also let p be any number in $(0,1)$. Let us use the following abbreviations.

$$\Pr[\mathrm{G}x,3\mathrm{rd}|p] := \Pr\big[(x \in \mathrm{Guessed}(\pi)) \wedge (\pi(y) < p) \wedge (\pi(z) < p)\big|\pi(x) = p\big], \quad \text{and}$$

$$\Pr[\mathrm{F}x|p] := \Pr\big[x \in \mathrm{Forced}(\pi)\big|\pi(x) = p\big].$$

Lemma 4.4. *[10, Lemma 2.10.4] For at least two variables of C (denoting this variable by x and the other variables of C by y and z), the following holds.*

$$\Pr[\mathrm{G}x,3\mathrm{rd}|p] + \Pr[\mathrm{F}x|p] \geq 2p^2 - p^3. \tag{4.18}$$

From these lemmas, we can show the following bound.

Lemma 4.5. *For any realizable f^1_{11}, we have*

$$\begin{aligned} f^1_{11} &\leq v^1_{11} - |V_D|^{-1} \sum_{x \in V_{3,1,\mathrm{ok}}} \int_0^t \big(2p^2 - p^3 - \Pr[\mathrm{F}x|p]\big)\, dp \\ &\leq v^1_{11} - \frac{2}{3} w_3 \int_0^t \big(2p^2 - p^3\big)\, dp + |V_D|^{-1} \sum_{x \in V_{3,1,\mathrm{ok}}} \int_0^t \Pr[\mathrm{F}x|p] dp, \end{aligned}$$

where we use, for any $i \in \{1,2,3\}$ and $b \in \{0,1\}$, $V_{i,b,\mathrm{ok}}$ to denote the set of variables x of some weight i clause in D such that $\alpha(x) = b$, and the bound (4.18) of the above lemma holds.

Proof. Reall that $g^1_{11} = v^1_{11} - f^1_{11}$; hence, we derive the above upper bound from a lower bound of g^1_{11}. Recall also that g^1_{11} is the expected fraction of variables x in V_D such that such that (i) $\pi(x) \leq t$, (ii) $\alpha(x) = 1$, and (iii) x appears as a guessed variable with prefix 11. Candidates of such variables x are those in weight three clauses such that $\alpha(x) = 1$. Among them, we consider only variables satisfying the bound (4.18). $V_{3,1,\mathrm{ok}}$ is the set of such variables. Consider any variable x in $V_{3,1,\mathrm{ok}}$, and let C be the clause containing x. For each $p \leq t$, reall that $\Pr[\mathrm{G}x,3\mathrm{rd}|p]$ is the probability that x is the last and guessed variable of C appearing in the PPSZ process under the condition $\pi(x) = p$. Since this event is equivalent to the event that (i), (ii), and (iii) hold, the probability that x satisfies the above (i), (ii), and (iii) is

$$\int_0^t \Pr[\mathrm{G}x,3\mathrm{rd}|p] dp \geq \int_0^t \big(2p^2 - p^3 - \Pr[\mathrm{F}x|p]\big)\, dp,$$

from the bound (4.18). Then summing up this bound for all variables in $V_{3,1,\mathrm{ok}}$ and dividing by $|V_D|$, we obtain a lower bound for g^1_{11}, from which the first upper bound of the lemma follows.

For showing the second bound, we use **Lemma 4.4**. From the lemma, it follows that each weight three clause has at least two variables satisfying the bound (4.18), both of which are clearly assigned 1 by α. Hence, $|V_{3,1,\mathrm{ok}}| \geq 2W_3$; thus, deviding this by $|V_D| = 3(W_1 + W_2 + W_3)$, we have $2w_3/3$ because $w_3 = W_3/(W_1 + W_2 + W_3)$. Then the second bound follows. □

Now consider the term $|V_D|^{-1} \sum_{x \in V_{3,1,\mathrm{ok}}} \int_0^t \Pr[\mathrm{F}x|p]dp$ in the upper bound of the above lemma. Note again that $\int_0^t \Pr[\mathrm{F}x|p]dp$, the probability that x appears as a forced variable before time t, is at least $\int_0^t p^2/(1-p)^2 dp - o(1)$ for every variable x. On the other hand, it is clear that the success probability of the biased-PPSZ part gets worse if this forced probability gets smaller. In fact, under the assumption that $\int_0^t \Pr[\mathrm{F}x|p]dp \geq \int_0^t p^2/(1-p)^2 dp$ for every variable x, we can show (by the argument proving Fact 4.13) that $\mathrm{h}(\boldsymbol{f})$ becomes largest if $\int_0^t \Pr[\mathrm{F}x|p]dp = \int_0^t p^2/(1-p)^2 dp$ for every x. Thus, under this senario we have

$$|V_D|^{-1} \sum_{x \in V_{3,1,\mathrm{ok}}} \int_0^t \Pr[\mathrm{F}x|p]dp = |V_D|^{-1} \sum_{x \in V_{3,1,\mathrm{ok}}} \int_0^t \left(\frac{p}{1-p}\right)^2 dp = \frac{2}{3} w_3 \int_0^t \left(\frac{p}{1-p}\right)^2 dp$$

From this consideration, we derive the following constraint.

$$f^1_{11} \leq \widehat{f}^1_{11} := v^1_{11} - \frac{2}{3} w_3 \int_0^t \left(2p^2 - p^3 - \left(\frac{p}{1-p}\right)^2\right) dp. \tag{4.19}$$

Remark 4.2. Note that we use the condition $\int_0^t \Pr[\mathrm{F}x|p]dp = \int_0^t p^2/(1-p)^2 dp$ for every x only for deriving our bounds for f^b_P, and we would not use it as our constraint.

Next consider constraints for f^0_{01} and f^0_{10}.

Lemma 4.6. *For any realizable f^0_{01} and f^0_{10}, we have*

$$f^0_{01} \leq v^0_{01} - \frac{1}{2|V_D|} \sum_{x \in V_{2,0,\mathrm{ok}}} \int_0^t \left(2p^2 - p^3 - \Pr[\mathrm{F}x|p]\right) dp$$

$$f^0_{10} \leq v^0_{10} - \frac{1}{2|V_D|} \sum_{x \in V_{2,0,\mathrm{ok}}} \int_0^t \left(2p^2 - p^3 - \Pr[\mathrm{F}x|p]\right) dp$$

Proof. The proof goes in the same way as that of **Lemma 4.5** except that the probability that (i) $\pi(x) \leq t$, (ii) $\alpha(x) = 1$, and (iii) x appears as a guessed variable with prefix 01 (resp., 10) is the half of $\Pr[\mathrm{G}x, 3\mathrm{rd}|p]$ due to the order of the other two variables by π. The rest is the same and it is omitted here. □

Unfortunately, a bound derived from this lemma is weaker than the previous one. The problem is that we cannot guarantee $|V_{1,0,\mathrm{ok}}| \geq 2W_1$. There are certainly at least $2W_1$ variables in weight one clauses of D that satisfy the bound (4.18); however, there may be some weight one clause in which a variable satisfying (4.18) is the one assigned 1 and there is only one variable that is assigned 0 and that satisfies (4.18). In other words, while we have $|V_{1,0,\mathrm{ok}}|+|V_{1,1,\mathrm{ok}}| \geq 2W_1$, it does not guarantee that $|V_{1,0,\mathrm{ok}}| \geq 2W_1$ because it may be the case $V_{1,1,\mathrm{ok}} \neq \emptyset$. We show below that we may expect a larger variable-wise advantage in this case, which guarantees us to ignore this case for estimating a lower bound of $\Delta_{\boldsymbol{w},t}$.

We prepare some lemmas. Recall that a clause C is called a critical clause of x if x is the variable of only one literal evaluated 1 in C by (the target sat. assignment) α. Let $C(x; y, z)$ be the clause composed of literals of the variables x, y, z such that the clause is a critical clause of x. Note that the clause $C(x; y, z)$ may or may not appear in the input formula F.

Lemma 4.7. *[10, Lemma 2.8.5.][3, Lemma 7.5] If the clause $C(x; y, z)$ does not appear in the input formula F, then x satisfies (4.18).*

Lemma 4.8. *[10, Lemma 2.7.1.][3, Lemma 6.1.] Let x be a variable with more than one critical clause, then for any $p < 1/2$, we have $\Pr[x \in \mathrm{Forced}(\pi)|\pi(x) = p] \geq p^2/(1-p) + p^2 - p^3$.*

Lemma 4.9. *[10, Lemma 2.7.3.][3, Lemma 6.2.] Let x, y, z_1, z_2 be some variables such that x has a critical clause containing literals of y and z_1, and y has a critical clause containing literals of x and z_2. Then, for any $p < 1/2$, either x or y satisfies*

$$\Pr[x \in \mathrm{Forced}(\pi)|\pi(x) = p] \geq p^2/(1-p) + p^2 - p^3$$

$$\Pr[y \in \text{Forced}(\pi) | \pi(y) = p] \geq p^2/(1-p) + p^2 - p^3$$

Corollary 4.10. *Let x, y, z_1, z_2 be some variables such that x has a critical clause containing literals of y and z_1, and y has a critical clause containing literals of x and z_2. If both x and y has only one critical clause, then for any $p < 1/2$, both x and y satisfy*

$$\Pr[x \in \text{Forced}(\pi) | \pi(x) = p] \geq p^2/(1-p) + p^2 - p^3$$

$$\Pr[y \in \text{Forced}(\pi) | \pi(y) = p] \geq p^2/(1-p) + p^2 - p^3$$

Proof. First, consider the construction of the critical clause tree of x. As we assume that x has only one critical clause, the var-labels of the children of the root must be y and z_1. When we extend the tree further by adding children to the node whose var-label is y, we flip the values of both x and y in the unique satisfying assignment and look for an unsatisfied clause. If such clause does not contain neither x nor y, then the assignment is not satisfying assignment to begin with. If it contains only one of them, then this variable has an additional critical clause, which contrasts to our assumption. Thus, each such unsatisfied clause must contain both x and y. It implies that the node whose var-label is y has at most one child. Therefore, by a similar analysis to (2.5) and the proof of **Theorem 2.13**, y's subtree in x's critical clause tree has a probability of at least $p + (1-p) \cdot \frac{p}{1-p} = 2p$ to contain more than $d_3(\epsilon)$ nodes at time p after the random deletion. Thus the total probability of the entir critical clause tree to contain more than $d_3(\epsilon)$ nodes at time p after the random deletion is at least $2p \cdot \frac{p}{1-p} = \frac{2p^2}{1-p}$. The same holds for y's critical clause tree. Thus, this corollary holds with $\frac{2p^2}{1-p} \geq \frac{p^2}{1-p} + p^2 - p^3$. □

Thus, we have the following lemma.

Lemma 4.11. *Consider any weight one clause $C = (x \vee y \vee z)$ in D, and assume that $\alpha(x) = 1$ and $\alpha(y) = \alpha(z) = 0$. Then if either y or z does not satisfy (4.18) for some $p < 1/2$, then, for any $p < 1/2$, we have*

$$\Pr[x \in \text{Forced}(\pi) | \pi(x) = p] \geq \max\left(\frac{p^2}{1-p} + p^2 - p^3, \left(\frac{p}{1-p}\right)^2\right) \tag{4.20}$$

Proof. Without loss of generality, assume that y does not satisfy (4.18) and z does. Then, we have $C(y; x, z) \in F$, because, otherwise, y satisfies (4.18) by **Lemma 4.7**. Consider the number of critical clauses of y in F. If y has more than one critical clause, then y satisfies (4.18) by **Lemma 4.8**. So y must have only one critical clause. Next, if x have only one critical clause (which is $C(x; y, z)$) in F, then y satisfies (4.18) by **Corollary 4.10**. Thus, x must have more than one critical clause in F, and we can say x satisfies (4.20) by **Lemma 4.8**.

□

Let D_1 and V_1 denote respectively the set of weight one clauses of D and the set of variables in such weight one clauses. We focus on weight one clauses of D_1 fulfilling the condition of the above lemma; let $D_{1,\mathrm{ng}}$ denote the set of such weight one clauses. For such a clause $C \in D_{1,\mathrm{ng}}$, we symbolically use here (only here) "x" to denote a variable assigned 1 by α, "y" to denote a variable with $\alpha(y) = 0$ for which (4.18) does not hold, and "z" to denote the third one in the clause. Then let $V_{1,\mathrm{ngx}}$, $V_{1,\mathrm{ngy}}$, and $V_{1,\mathrm{ngz}}$ denote respectively the set of variables "x", "y", and "z" defined above. Finally, let $V_{1,\mathrm{ng}} := V_{1,\mathrm{ngx}} \cup V_{1,\mathrm{ngy}} \cup V_{1,\mathrm{ngz}}$; note that $V_{1,\mathrm{ng}}$ is the set of all variables appearing in clauses of $D_{1,\mathrm{ng}}$.

We discuss below a lower bound of the variable-wise advantage of variables in $V_{1,\mathrm{ng}}$. Recall (4.15); this is the definition of $\Delta_{\boldsymbol{w},t}$, the variable-wise advantage of variables in V_D on average. We estimate this value for variables in $V_{1,\mathrm{ng}}$. More specifically, for any variable $x \in V_D$, define $\Delta_{\boldsymbol{w},t}(x)$ by

$$\Delta_{\boldsymbol{w},t}(x) := S_3 - \left(\sum_P \Pr[\mathrm{G}x, \leq t, Px] \cdot \mathrm{H}(\beta_P) + \Pr[\mathrm{G}x, > t] \right),$$

where we use "$\mathrm{G}x, \leq t, Px$" and "$\mathrm{G}x, > t$" as abbreviations of "x appears before time t as a guessed variable with prefix P" and "x appears after time t as a guessed variable." It is easy to see that $\Delta_{\boldsymbol{w},t} = (\sum_{x \in V_D} \Delta_{\boldsymbol{w},t}(x))/|V_D|$. We estimate $\Delta_{\boldsymbol{w},t}(x)$ for $x \in V_{1,\mathrm{ng}}$, in particular, $x \in V_{1,\mathrm{ngx}}$.

Lemma 4.12. *For any $x \in V_{1,\mathrm{ngx}}$, we have*

$$\Delta_{\boldsymbol{w},t}(x) \geq \int_0^{t_0} \left(\frac{p^2}{1-p} + p^2 - p^3 - \left(\frac{p}{1-p} \right)^2 \right) dp \geq 0.0035.$$

Here t_0 is the point from where $(p/(1-p))^2$ is larger than $p^2/(1-p) + p^2 - p^3$; that is, $(p/(1-p))^2 < p^2/(1-p) + p^2 - p^3$ for $p \in [0, t_0)$ and $(p/(1-p))^2 > p^2/(1-p) + p^2 - p^3$ for $p \in (t_0, 1)$. (Note that $t_0 = 0.31767\cdots$.)

Proof. Consider any $x \in V_{1,\mathrm{ngx}}$, and let $C = (x \vee y \vee z)$ be the weight one clause of D containing x. Since $\alpha(x) = 1$ and $\alpha(y) = \alpha(z) = 0$, the prefix x is $\emptyset$, 0, or 00. More specifically, for example, the event of "Gx, $\pi(x) \leq t$, and its prefix is $\emptyset$" is equivalent to that of "Gx, $\pi(x) \leq t$, and 2nd," where by "2nd" we mean the condition that x appears second among x, y, and z. Then we have

$$\begin{aligned}
\Delta_{\boldsymbol{w},t}(x) &= S_3 - \left(\sum_P \Pr[\mathrm{G}x, \leq t, Px] \cdot \mathrm{H}(\beta_P) + \Pr[\mathrm{G}x, > t] \right) \\
&= S_3 - \big(\Pr[\mathrm{G}x, \leq t, 1\mathrm{st}] \cdot \mathrm{H}(\beta_\emptyset) + \Pr[\mathrm{G}x, \leq t, 2\mathrm{nd}] \cdot \mathrm{H}(\beta_0) \\
&\qquad + \Pr[\mathrm{G}x, \leq t, 3\mathrm{rd}] \cdot \mathrm{H}(\beta_{00}) + \Pr[\mathrm{G}x, > t] \big)
\end{aligned}$$

Noting that $\mathrm{H}(\beta_P) \leq 1$, we further have

$$\begin{aligned}
&\geq S_3 - \big(\Pr[\mathrm{G}x, \leq t, 1\mathrm{st}] + \Pr[\mathrm{G}x, \leq t, 2\mathrm{nd}] + \Pr[\mathrm{G}x, \leq t, 3\mathrm{rd}] + \Pr[\mathrm{G}x, > t] \big) \\
&= S_3 - \Pr[\mathrm{G}x] = S_3 - \big(1 - \Pr[\mathrm{F}x] \big) \\
&\geq S_3 - \left(1 - \left(\int_0^{t_0} \left(\frac{p^2}{1-p} + p^2 - p^3 \right) dp + \int_{t_0}^1 \min \left(\left(\frac{p}{1-p} \right)^2, 1 \right) dp \right) \right) \\
&= 1 - \int_0^t \min \left(\left(\frac{p}{1-p} \right)^2, 1 \right) dp \\
&\qquad - \left(1 - \left(\int_0^{t_0} \left(\frac{p^2}{1-p} + p^2 - p^3 \right) dp + \int_{t_0}^1 \min \left(\left(\frac{p}{1-p} \right)^2, 1 \right) dp \right) \right) \\
&= \int_0^{t_0} \left(\frac{p^2}{1-p} + p^2 - p^3 - \left(\frac{p}{1-p} \right)^2 \right) dp,
\end{aligned}$$

proving the desired bound. Here we use the bound (4.20) and the fact that $t_0 < 1/2$ (and hence, $(p/(1-p))^2 < 1$ for any p, $0 \leq p \leq t_0$). □

Clearly, $\Delta_{\boldsymbol{w},t}(x) \geq 0$ for any x. Thus, for the variable-wise advantage on $V_{1,\mathrm{ng}}$, we have

$$\frac{\sum_{x \in V_{1,\mathrm{ng}}} \Delta_{\boldsymbol{w},t}(x)}{|V_{1,\mathrm{ng}}|} \geq \frac{0.0035}{3} \geq 1.1 \times 10^{-3}.$$

As we will see in the next subsection, we estimate that $\Delta_{\boldsymbol{w}}$ at the worst-case weight vector $\boldsymbol{w}_{\mathrm{wst}}$ (with $t = 0.17$) is 1.188×10^{-4}, which is much smaller than the above. Thus, for searching such worst-case (i.e., the smallest $\Delta_{\boldsymbol{w}}$), we should assume[9] that $D_{1,\mathrm{ng}} = \emptyset$, that is, there is no weight one clause satisfying the condition of **Lemma 4.11**; that is, $V_{1,0,\mathrm{ng}} = \emptyset$. Then we may assume that $|V_{1,0,\mathrm{ok}}| \geq 2W_2$, from which we can derive the following constraint.

$$f^0_{01} \leq \widehat{f}^0_{01} := v^0_{01} - \frac{1}{3} w_1 \int_0^t \left(2p^2 - p^3 - \left(\frac{p}{1-p} \right)^2 \right) dp, \quad \text{and} \tag{4.21}$$

$$f^0_{10} \leq \widehat{f}^0_{10} := v^0_{10} - \frac{1}{3} w_1 \int_0^t \left(2p^2 - p^3 - \left(\frac{p}{1-p} \right)^2 \right) dp. \tag{4.22}$$

For the other components of $\boldsymbol{f}$, we simply use the trivial constraints, that is, $\widehat{f}^0_{11} := v^0_{11}$ for example. This is because a nontrivial constraint like $\widehat{f}^1_{11}$ is not necessary for the other components from the actual execution of our numerical analysis (see the next subsection). Thus, theoretically, we may consider that the trivial constraints are used for these other components.

Now we formally define our $\boldsymbol{f}_{\mathrm{opt}}$ as a vector $\boldsymbol{f}$ that maximizes the objective function $\mathrm{h}(\boldsymbol{f})$ among vectors satisfying the above-defined constraints. This is the target of our numerical analysis for a given $\boldsymbol{w}$. Here we remark again that it is quite likely that the obtained vector is not realizable, that is, it has no corresponding instance formula F. This is why $\boldsymbol{f}_{\mathrm{opt}}$ may not be the same as $\boldsymbol{f}_{\mathrm{wst}}$, but at least we have $\boldsymbol{f}_{\mathrm{opt}} \geq \boldsymbol{f}_{\mathrm{wst}}$.

[9]Formally speaking, when evaluating the average variable-wise gain $\Delta_{\boldsymbol{w},t}$, we consider variables in weight one clauses of $D_{1,\mathrm{ng}}$ separately to show the above lower bound while variables in the other clauses of D are evaluated by our numerical method. Then we confirm that the obtained lower bound is smaller by assuming $D_{1,\mathrm{ng}} = \emptyset$. Here we skip this process and assume that $D_{1,\mathrm{ng}} = \emptyset$ for simplifying our discussion.

We show some basic properties that give useful suggestions for designing the search algorithm. (Below we consider $\mathrm{h}(\boldsymbol{f})$ on vectors $\boldsymbol{f}$ satisfying the trivial constraints, i.e., $0 \leq f_P^b \leq v_P^b$ for any P and b.)

Fact 4.13. $\mathrm{h}(\boldsymbol{f})$ *is component-wise monotonistically decreasing.*

Proof. The fact follows from the following partial derivative of the objective function with respect to each component f_P^b.

$$\frac{\partial}{\partial f_P^b}\left(\mathrm{h}\left(v_P^0 - f_P^0, v_P^1 - f_P^1\right)\right) = \log\left(\frac{v_P^b - f_P^b}{v_P^b - f_P^b + v_P^{1-b} - f_P^{1-b}}\right) < 0$$

□

From this property, we see that $\mathrm{h}(\boldsymbol{f})$ takes the largest value at the zero vector $\mathbf{0} = (0, 0, \ldots, 0)$. But clearly, the zero vector does not satisfy our key constraint (4.17). Thus, starting from $\boldsymbol{f} = \mathbf{0}$, we increase some component of $\boldsymbol{f}$ bit by bit until it satisfies the constraint (4.17). This is our basic strategy. Then we use the following facts to determine the order of components to be increased their values. In the following, we use θ to denote any positive value that is small enough in each context. We use $\boldsymbol{f} +_{(P,b)} \theta$ to denote a vector with the same components as $\boldsymbol{f}$ except that the (P, b)th component is not f_P^b but $f_P^b + \theta$. For any type P, let $\beta_P := (v_P^0 - f_P^0)/(v_P^0 - f_P^0 + v_P^1 - f_P^1)$ (which is the same definition as the bias for type P).

Fact 4.14. *If* $v_P^b - f_P^b > v_P^{1-b} - f_P^{1-b}$, *then* $\mathrm{h}(\boldsymbol{f} +_{P,b} \theta) > \mathrm{h}(\boldsymbol{f} +_{(P,1-b)} \theta)$.

Proof. Note that

$$\begin{aligned}
&\frac{\partial}{\partial x}\ \left(\mathrm{h}\left(v_P^b - (f_P^b + (\theta - x)), v_P^{1-b} - (f_P^{1-b} + x)\right)\right) \\
&= -\log\left(\frac{v_P^b - (f_P^b + \theta - x)}{v_P^{1-b} - (f_P^{1-b} + x)}\right) = -\log\left(\frac{v_P^b - f_P^b - \theta + x)}{v_P^{1-b} - f_P^{1-b} - x}\right).
\end{aligned}$$

This value is positive for small θ and $0 \leq x \leq \theta$. Hence, $\mathrm{h}(\boldsymbol{f} +_{(P,b)} (\theta - x))$ is largest when $x = 0$, which implies the fact. □

This fact suggests that if we need to increase the components corresponding to some type P, we should increase only the (P,b)th component with larger $v_P^b - f_P^b$. Note that $v_P^0 - f_P^0 > v_P^1 - f_P^1$ (resp., $v_P^1 - f_P^1 > v_P^0 - f_P^0$) if and only if $\beta_P > 1/2$ (resp., $\beta_P < 1/2$). Thus, while $\beta_P > 1/2$ (resp., $\beta_P < 1/2$), we need to increase only the $(P,0)$th (resp., the $(P,1)$th) component. Let us use $b(P)$ to indicate whether $\beta_P > 1/2$ or $\beta_P < 1/2$; that is, $b(P) = 0$ if $\beta_P > 1/2$, and $b(P) = 1$ if $\beta_P < 1/2$. Then we can restate the above strategy as follows: While $|\beta_P - 1/2| > 0$, we need to increase only the $(P, b(P))$th component, reducing $|\beta_P - 1/2|$.

This fact also suggests a way to increase some components of $\boldsymbol{f}$ when $\beta_P = 1/2$ (i.e., $v_P^0 - f_P^0 = v_P^1 - f_P^1$) for all types P. From this fact, we had better keep $\beta_P = 1/2$ when increasing some components of $\boldsymbol{f}$, which is, in fact, possible by increasing both f_P^0 and f_P^1 by the same amount x. Thus, if we need to increase some components by θ, we should increase both f_P^0 and f_P^1 by $\theta/2$ at some P. (Let us denote this type of update by $\boldsymbol{f} +_P \theta/2$.) The following fact shows that P can be chosen arbitrarily.

Fact 4.15. *For any P_1 and P_2 such that $\beta_{P_1} = \beta_{P_2} = \frac{1}{2}$, then we have* $\mathrm{h}(\boldsymbol{f} +_P \theta) = \mathrm{h}(\boldsymbol{f} +_P \theta)$.

Proof. We have

$$\begin{aligned}
&\frac{\partial}{\partial x}\left(\mathrm{h}\left(v_{P_1}^0 - (f_{P_1}^0 + x), v_{P_1}^1 - (f_{P_1}^1 + x)\right)\right) - \frac{\partial}{\partial x}\left(\mathrm{h}\left(v_{P_2}^0 - (f_{P_2}^0 + x), v_{P_2}^1 - (f_{P_2}^1 + x)\right)\right) \\
&= \log\left(\frac{v_{P_1}^0 - (f_{P_1}^0 + x)}{v_{P_1}^0 - (f_{P_1}^0 + x) + v_{P_1}^1 - (f_{P_1}^1 + x)}\right) - \log\left(\frac{v_{P_2}^0 - (f_{P_2}^0 + x)}{v_{P_2}^0 - (f_{P_2}^0 + x) + v_{P_2}^1 - (f_{P_2}^1 + x)}\right) \\
&= \log\frac{1}{2} - \log\frac{1}{2} = 0.
\end{aligned}$$

The fact follows from this. □

On the other hand, when there are some types such that $\beta_P \neq 1/2$, the following fact suggests that we should increase the component corresponding to type P with the largest $\beta_P > 1/2$ (resp., the smallest $\beta_P < 1/2$).

Fact 4.16. *Consider the case where both β_{P_1} and β_{P_2} are larger than $1/2$ (hence, $v_{P_i}^0 \geq v_{P_i}^1$ for $i = 1, 2$). Then if $\beta_{P_1} > \beta_{P_2}$, then* $\mathrm{h}(\boldsymbol{f} +_{(P_1,0)} \theta) > \mathrm{h}(\boldsymbol{f} +_{(P_2,0)} \theta)$. *In general, if*

$|\beta_{P_1} - 1/2| > |\beta_{P_2} - 1/2|$, *then* $\mathrm{h}(\boldsymbol{f} +_{(P_1,b_1)} \theta) > \mathrm{h}(\boldsymbol{f} +_{(P_2,b_2)} \theta)$, *where* $v_{P_i}^{b_i} \geq v_{P_i}^{1-b_i}$ *for* $i = 1, 2$.

Proof. Consider the case where both β_{P_1} and β_{P_2} are larger than $1/2$. We have

$$\begin{aligned}
&\frac{\partial}{\partial x}\left(\mathrm{h}\left(v_{P_1}^0 - (f_{P_1}^0 + x), v_{P_1}^1 - f_{P_1}\right)\right) - \frac{\partial}{\partial x}\left(\mathrm{h}\left(v_{P_2}^0 - (f_{P_2}^0 + x), v_{P_2}^1 - f_{P_2}^1\right)\right) \\
&= \log\left(\frac{v_{P_1}^0 - (f_{P_1}^0 + x)}{v_{P_1}^0 - (f_{P_1}^0 + x) + v_{P_1}^1 - f_{P_1}^1}\right) - \log\left(\frac{v_{P_2}^0 - (f_{P_2}^0 + x)}{v_{P_2}^0 - (f_{P_2}^0 + x) + v_{P_2}^1 - f_{P_2}^1}\right).
\end{aligned}$$

This is positive for small x since $\beta_{P_1} > \beta_{P_2} > 1/2$, and the fact follows from this. The general case can be shown by considering each case similarly. □

From these facts, we give an algorithm for computing an approximation of $\boldsymbol{f}_{\mathrm{wst}}$. **Algorithm 13** states its outline[10] Here we explain some key ideas and the meaning of some notions and notations used in the algorithm.

As discussed above, our basic strategy is to increase some components of a vector $\boldsymbol{f}$ starting from the zero vector until the target constraint (4.17) is satisfied. Let $\delta(\boldsymbol{f})$ denote the amount we need to increase until this goal; that is,

$$\delta(\boldsymbol{f}) := \int_0^t p^2/(1-p)^2 dp - \sum_{P,b} f_P^b.$$

On the other hand, there is a limit of an increment at each component. Firstly, for each pair of P and b, there is a limit of increasing the (P, b)th component from our constraints, which we denote by $\delta_{P,b}(\boldsymbol{f})$. For example, since we have $\widehat{f}_{11}^1$, $\delta_{11,1}(\boldsymbol{f}) := \widehat{f}_{11}^1 - f_{11}^1$. On the other hand, by the trivial constraint, we define $\delta_{\emptyset,0}(\boldsymbol{f}) := v_\emptyset^0 - f_\emptyset^0$. Secondly, there is a limit for using the same updating strategy. For example, suppose that we need to increase the $(P, 0)$th component of $\boldsymbol{f}$ because $|\beta_P - 1/2|$ is the largest among all types and $\beta_P > 1/2$. Then let $\theta_P(\boldsymbol{f})$ denote the lmit of increasing the $(P, b(P))$th component until this situation gets changed; more specifically, $\theta_P(\boldsymbol{f}) := \min(\theta_1, \theta_2)$, where θ_1 and θ_2 are thresholds such that $|\beta_P - 1/2|$ becomes the same as the second largest one in $\boldsymbol{f} +_{(P,b(P))} \theta_1$, and $\beta_P = 1/2$ in $\boldsymbol{f} +_{(P,b(P))} \theta_2$ respectively.

[10]This version is simpler than the one explained in our IEICE paper (Appendix A of [15]) because we do not use a fixed update value by which we no longer need to consider $\boldsymbol{f}_{\mathrm{last}}$.

In the algorithm, we use a variable $\mathcal{B}$ to keep the set of types P such that $|\beta_P - 1/2| > 0$. A variable $\mathcal{A}$ is used to keep the set of "active" types P in the sense $\delta_{P,b(P)} > 0$ if there is some $P \in \mathcal{B}$ and $\delta_{P,0}, \delta_{P,1} > 0$ if $\beta_P = 1/2$ for all types P.

Algorithm 13 The Sweep-line Search

$\boldsymbol{f} := \mathbf{0}$, a vector whose (P,b)th component is 0 for all $P \in \mathcal{P}$ and $b \in \{0,1\}$

$\mathcal{A} :=$ the set of all "active" types

$\mathcal{B} :=$ the set of all types P such that $|\beta_P - 1/2| > 0$

while $\mathcal{A} \cap \mathcal{B} \neq \emptyset$ **do**

$\mathcal{C} :=$ the set of all types $P \in \mathcal{A} \cap \mathcal{B}$ with the largest $|\beta_P - 1/2|$; $\quad m := |\mathcal{C}|$

$\delta := \min\big(\delta(\boldsymbol{f})/m,\ \min_{P\in\mathcal{C}} \delta_{P,b(P)}(\boldsymbol{f}),\ \min_{P\in\mathcal{C}} \theta_P(\boldsymbol{f})\big)$

$\boldsymbol{f} := \boldsymbol{f} +_{(P,b(P))} \delta$ for each $P \in \mathcal{C}$

if (4.17) is satisfied **then** return $\boldsymbol{f}$ as $\boldsymbol{f}_{\text{opt}}$

update $\mathcal{A}$ and $\mathcal{B}$

end while

(The case where $\mathcal{B} = \emptyset$ and still $\delta(\boldsymbol{f}) > 0$)

$\mathcal{A} :=$ the set of all "active" types (under this situation)

while true **do** (the loop must be terminated unless $\boldsymbol{w}$ is not realizable)

$\delta := \min\left(\delta(\boldsymbol{f})/2,\ \max_{P\in\mathcal{A}} \min(\delta_{P,0}(\boldsymbol{f}), \delta_{P,1}(\boldsymbol{f}))\right)$

$\boldsymbol{f} := \boldsymbol{f} +_P \delta$, where $P = \text{argmax}_{P\in\mathcal{A}} \min(\delta_{P,0}(\boldsymbol{f}), \delta_{P,1}(\boldsymbol{f}))$

if (4.17) is satisfied **then** return $\boldsymbol{f}$ as $\boldsymbol{f}_{\text{opt}}$

update $\mathcal{A}$

end while

Remark. The program in Appendix 2, which is used for searching the worst case numerically, is a little different from **Algorithm 13**. In order to make the program simpler and more readable, we have changed some algorithmic details. Please refer to the appendix for more information.

4.4.2 Derivation of our exponential coefficient ϵ_{QW}

For a given weight vector $\boldsymbol{w}$ (a fractional version) and a time threshold t, we can use the numerical algorithm to compute $\boldsymbol{f}_{\mathrm{opt}}$ and a lower bound for $\Delta_{\boldsymbol{w},t}$, which we simply regard as $\Delta_{\boldsymbol{w},t}$. Then taking a maximum over all $t \in \mathrm{TH}$, we get $\Delta_{\boldsymbol{w}}$. Then we have

$$\varepsilon_{\mathrm{QW}}(\boldsymbol{w}) := S_3 - 3\gamma_{\boldsymbol{w}}\Delta_{\boldsymbol{w}} \tag{4.23}$$

as a candidate of the exponential coefficient based on the weight vector $\boldsymbol{w}$. Then to estimate the success probability lower bound, we need to find the worst weight vector that maximizes this value. It is, however, impossible to compute $\Delta_{\boldsymbol{w}}$ for all weight vectors, and we introduce some approximation approaches.

We consider weight vectors whose components are discrete values in $[0, 1]$ at interval $\delta_{\mathrm{wstep}} := 1.0 \times 10^{-3}$; that is, those in a set VW defined by

$$\mathrm{VW} := \{(x, y, 1-(x+y)) | x = k_1\delta_{\mathrm{wstep}} \text{ and } y = k_2\delta_{\mathrm{wstep}} \text{ so that } x + y \leq 1\}.$$

In our numerical analysis, we computed $\Delta_{\boldsymbol{w}}$ for all $\boldsymbol{w} \in \mathrm{VW}$ by the method of the previous section, and obtained the worst one $\boldsymbol{w}_{\max} := (0.663, 0.160, 0.177)$ giving the smallest $\Delta_{\boldsymbol{w}}$ in this set, which gives the value of $\varepsilon_{\mathrm{QW}}(\boldsymbol{w}_{\max})$ as $0.38624062\ldots$ with $t = 0.17$. While we can expect that this is close to the largest, the worst weight vector likely exists that gives a yet larger estimate of $\varepsilon_{\mathrm{QW}}(\boldsymbol{w})$.

In order to check this point, we want to express $\boldsymbol{f}_{\mathrm{opt}}$ as a relatively simple formula. Fortunately, we found that this is possible by focusing on weight vectors that are close to this $\boldsymbol{w}_{\max}$ (w.r.t. the fixed time threshold $t = 0.17$), and we could give a formula expressing $\boldsymbol{f}_{\mathrm{opt}}$. Note that in our numerical method, i.e., in **Algorithm 13**, the essential part of the computation is to update the component(s) of $\boldsymbol{f}$ corresponding to the "active" type(s) P with the largest $|\beta_P - 1/2|$, among *only six* possible types. Here we consider weight vectors in a small region around $\boldsymbol{w}_{\max} = (w_{\max,1}, w_{\max,2}, w_{\max,3})$; more concretely, weight vectors $\boldsymbol{w}(x, y)$ in the set $N := \{(w_{\max,1} + x, w_{\max,2} + y, w_{\max,3} - (x + y)) | x, y \in [-5\delta_{\mathrm{wstep}}, 5\delta_{\mathrm{wstep}}]\}$. Then we can confirm that the process of computing $\boldsymbol{f}_{\mathrm{opt}}$ runs as follows: first, (i) both f^0_{01} and f^1_{10} are increased up to $\widehat{f}^0_{01}$ and $\widehat{f}^1_{10}$; next, (ii) f^1_{11} is increased up to $\widehat{f}^1_{11}$; then, (iii) f^1_0 and f^0_1 are increased while keeping $\beta_0 = \beta_1$ until the key

constraint (4.17) is satisfied. The other components are not changed. Based on this observation, we can give a relatively simple formula expressing $\boldsymbol{f}_{\text{opt}}$ and then $\varepsilon_{\text{QW}}(\boldsymbol{w}(x,y))$ in terms x and y for $\boldsymbol{w}(x,y) = (w_{\max,1}+x, w_{\max,2}+y, w_{\max,3}-(x+y)) \in N$. Then analyzing $\varepsilon_{\text{QW}}(\boldsymbol{w}(x,y))$ by *Mathematica*[11], we derived that $\epsilon_{\text{QW}}(\boldsymbol{w}(x,y))$ takes the maximum $0.38624055\ldots$ at $(x,y) = (-0.00030637, 0.00015844)$. (We also confirmed that $t = 0.17$ is the best in TH for weight vectors in N.) Note that $\boldsymbol{w}(-0.00030637, 0.00015844)$ is component-wise δ_{wstep}-close to $\boldsymbol{w}_{\max}$; hence, this result can be regarded as evidence that our interval δ_{wstep} grid is enough to find the worst-case weight vector. Therefore under the assumption that the real worst-case weight vector is component-wise δ_{wstep}-close to $\boldsymbol{w}_{\max}$, we can claim the following theorem.

Theorem 4.17. *For any instance of Unique 3-SAT, the algorithm QW_biased-PPSZ outputs its satisfying assignment (if it exists) with a probability of at least* $2^{-(\varepsilon_{QW}+o(1))n}$, *with the exponential coefficient* $\varepsilon_{QW} = 0.386241$ *under the following technical condition* $(*)$.

Technical Condition $(*)$ *We numerically confirmed that* $\varepsilon_{QW}(\boldsymbol{w})$ *of (4.23), a candidate of exponential coefficient of QW_biased-PPSZ, takes the largest value at* $\boldsymbol{w}_{\max}$ *with* $t = 0.17$ *among all* $\boldsymbol{w}$ *in the* $\delta_{\text{wstep}} = 1.0 \times 10^{-3}$*-interval grid* VW. *Our technical condition is that the real worst-case weight vector (for the fixed* $t = 0.17$*) exists within component-wise* δ_{wstep}*-close to this* $\boldsymbol{w}_{\max}$.

[11] See Appendix 1 for details.

Chapter 5

Conclusion

In this thesis, we first introduced the k-SAT problem and the PPSZ algorithms. Then we explained two different kinds of improvements over PPSZ, one is from Hertli [4] and the other is biased-PPSZ [3]. For Hertli's improvement, we pushed his idea further by refining GetInd2Clauses, which returned an extremely small improvement. For biased-PPSZ, we used biased-Search to replace brute-force search when the maximal disjoint set D is small enough. We also gave a more careful numerical analysis of the worst case of the algorithm. Finally, we obtained a better result comparing to [3], which becomes the best result for Unique 3-SAT so far.

There are still many open problems for the k-SAT problem. First, although we only considered biased-Search for Unique 3-SAT, the idea can be applied to Unique k-SAT easily. However, the numerical analysis for Unique 3-SAT cannot be applied to Unique k-SAT trivially. There may be a more clever way to search the worst case, which should not be based on program computing and can be easily applied to Unique k-SAT. Second, we believe that there should be some better way to define types and biases. The maximal disjoint set maybe is not necessary. Third, the analysis of the success probability of PPSZ is not tight. A better analysis may improve its result without any changes to the algorithm. What's more, we discussed the lower bound of the success probability of PPSZ in this thesis. However, the research of the upper bound can be also interesting.

Bibliography

[1] Sven Baumer and Rainer Schuler. "Improving a Probabilistic 3-SAT Algorithm by Dynamic Search and Independent Clause Pairs". In: *Theory and Applications of Satisfiability Testing*. Ed. by Gerhard Goos et al. Vol. 2919. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 150–161. ISBN: 978-3-540-20851-8 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_12. URL: http://link.springer.com/10.1007/978-3-540-24605-3_12 (visited on 05/23/2021).

[2] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[3] Thomas Dueholm Hansen, Haim Kaplan, Or Zamir, and Uri Zwick. "Faster k -SAT algorithms using biased-PPSZ". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing - STOC 2019*. Phoenix, AZ, USA: ACM Press, 2019, pp. 578–589. ISBN: 978-1-4503-6705-9. DOI: 10.1145/3313276.3316359. URL: http://dl.acm.org/citation.cfm?doid=3313276.3316359 (visited on 08/25/2020).

[4] Timon Hertli. "Breaking the PPSZ Barrier for Unique 3-SAT". In: *arXiv:1311.2513 [cs]* (Nov. 2013). URL: http://arxiv.org/abs/1311.2513 (visited on 10/05/2018).

[5] Timon Hertli. "3-SAT Faster and Simpler—Unique-SAT Bounds for PPSZ Hold in General". In: *SIAM Journal on Computing* 43.2 (Jan. 2014), pp. 718–729. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/120868177. URL: http://epubs.siam.org/doi/10.1137/120868177 (visited on 10/05/2018).

[6] Timon Hertli. "Improved Exponential Algorithms for SAT and ClSP". PhD thesis. ETH Zurich, 2015. URL: http://hdl.handle.net/20.500.11850/103982 (visited on 10/05/2018).

[7] Thomas Hofmeister, Uwe Schoning, Rainer Schuler, and Osamu Watanabe. "Randomized Algorithms for 3-SAT". In: *Theory of Computing Systems* 40.3 (2007), pp. 249–262. ISSN: 1433-0490. DOI: 10.1007/s00224-005-1275-6. URL: https://doi.org/10.1007/s00224-005-1275-6.

[8] L. A. Levin. "Universal Sequential Search Problems". In: *Probl. Peredachi Inf.* 9.3 (1973), pp. 115–116.

[9] B. Monien and E. Speckenmeyer. "Solving satisfiability in less than 2n steps". In: *Discrete Applied Mathematics* 10.3 (Mar. 1985), pp. 287–295. ISSN: 0166218X. DOI: 10.1016/0166-218X(85)90050-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0166218X85900502 (visited on 08/18/2020).

[10] Zamir Or. "Breaking Barriers for the Satisfiability and Coloring Problems". In: Tel Aviv University, 2020. (Visited on 10/2020).

[11] R. Paturi, P. Pudlak, and F. Zane. "Satisfiability Coding Lemma". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. Oct. 1997, pp. 566–574. DOI: 10.1109/SFCS.1997.646146.

[12] R. Paturi, P. Pudlik, M. E. Saks, and F. Zane. "An improved exponential-time algorithm for k-SAT". In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*. Nov. 1998, pp. 628–637. DOI: 10.1109/SFCS.1998.743513.

[13] Tong Qin and Osamu Watanabe. “An Improvement of the Algorithm of Hertli for the Unique 3SAT Problem”. In: *WALCOM: Algorithms and Computation*. Ed. by M. Sohel Rahman, Wing-Kin Sung, and Ryuhei Uehara. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 93–105. ISBN: 978-3-319-75172-6.

[14] Tong Qin and Osamu Watanabe. “An improvement of the algorithm of Hertli for the unique 3SAT problem”. In: *Theoretical Computer Science* 806 (Feb. 2020), pp. 70–80. ISSN: 03043975. DOI: 10.1016/j.tcs.2018.11.023. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397518307138.

[15] Tong Qin and Osamu Watanabe. “An Improvement of the Biased-PPSZ Algorithm for the 3SAT Problem”. In: IEICE Trans. Foundations of Comput. Sci. (2021). to appear.

[16] Robert Rodošek. “A new approach on solving 3-satisfiability”. In: *Artificial Intelligence and Symbolic Mathematical Computation*. Ed. by G. Goos et al. Vol. 1138. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 197–212. ISBN: 978-3-540-61732-7 978-3-540-70740-0. DOI: 10.1007/3-540-61732-9_59. URL: http://link.springer.com/10.1007/3-540-61732-9_59 (visited on 08/25/2020).

[17] Dominik Scheder and John P. Steinberger. “PPSZ for General k-SAT - Making Hertli’s Analysis Simpler and 3-SAT Faster”. In: (2017). Ed. by Marc Herbstritt. DOI: 10.4230/lipics.ccc.2017.9.

[18] T. Schoning. “A probabilistic algorithm for k-SAT and constraint satisfaction problems”. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. New York City, NY, USA: IEEE Comput. Soc, 1999, pp. 410–414. ISBN: 978-0-7695-0409-4. DOI: 10.1109/SFFCS.1999.814612. URL: http://ieeexplore.ieee.org/document/814612/ (visited on 10/05/2018).

[19] Magnus Wahlström. “An Algorithm for the SAT Problem for Formulae of Linear Length”. In: *Algorithms – ESA 2005*. Ed. by David Hutchison et al. Vol. 3669.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 107–118. ISBN: 978-3-540-29118-3 978-3-540-31951-1. URL: http://link.springer.com/10.1007/11561071_12 (visited on 10/05/2018).

Appendix 1

This is a script of Mathematica for searching the worst case around (0.663, 0,160, 0.177). The referenced part comes from pages 75-76 in Chapter 4.

In[•]:=

```
Clear[t];
t = 0.17;
S3 = 2 * Log[2] - 1;
int0 = t + 1 / (1 - t) - 1 + 2 * Log[1 - t];
int1 = (1 / 3) * t^3 - t^2 + t;
int2 = t^2 - (2 / 3) * t^3;
int3 = (1 / 3) * t^3;
inth = (2 / 3) * t^3 - (1 / 4) * t^4 - int0;
w1[x_, y_] := 0.663 + x
w2[x_, y_] := 0.160 + y
w3[x_, y_] := 0.177 - (x + y)
ve0[x_, y_] := ((2 / 3) * w1[x, y] + (1 / 3) * w2[x, y]) * int1
ve1[x_, y_] := ((1 / 3) * w1[x, y] + (2 / 3) * w2[x, y] + w3[x, y]) * int1
v00[x_, y_] := (1 / 3) * w1[x, y] * int2
v01[x_, y_] := ((1. / 3) * w1[x, y] + (1. / 3) * w2[x, y]) * int2
v10[x_, y_] := ((1. / 3) * w1[x, y] + (1. / 3) * w2[x, y]) * int2
v11[x_, y_] := ((1. / 3) * w2[x, y] + w3[x, y]) * int2
v001[x_, y_] := (1. / 3) * w1[x, y] * int3
v010[x_, y_] := (1. / 3) * w1[x, y] * int3
v011[x_, y_] := (1. / 3) * w2[x, y] * int3
v100[x_, y_] := (1. / 3) * w1[x, y] * int3
v101[x_, y_] := (1. / 3) * w2[x, y] * int3
v110[x_, y_] := (1. / 3) * w2[x, y] * int3
v111[x_, y_] := w3[x, y] * int3
ge0[x_, y_] := ve0[x, y]
ge1[x_, y_] := ve1[x, y]
g010[x_, y_] := (1 / 3) * w1[x, y] * inth
g011[x_, y_] := v011[x, y]
g100[x_, y_] := (1 / 3) * w1[x, y] * inth
g101[x_, y_] := v101[x, y]
g110[x_, y_] := v110[x, y]
g111[x_, y_] := (2 / 3) * w3[x, y] * inth
fremain[x_, y_] := int0 - v001[x, y] - (v010[x, y] - g010[x, y]) -
  (v100[x, y] - g100[x, y]) - (v111[x, y] - g111[x, y])
f01[x_, y_] := (v00[x, y] * fremain[x, y] + v01[x, y] * v11[x, y] -
    v00[x, y] * v10[x, y]) / (v00[x, y] + v11[x, y])
f10[x_, y_] := fremain[x, y] - f01[x, y]
g00[x_, y_] := v00[x, y]
```

```
g01[x_, y_] := v01[x, y] - f01[x, y]
g10[x_, y_] := v10[x, y] - f10[x, y]
g11[x_, y_] := v11[x, y]
h[x_, y_] := -x * Log[x / (x + y)] / Log[2] - y * Log[y / (x + y)] / Log[2]
he[x_, y_] := h[ve0[x, y], ve1[x, y]]
h0[x_, y_] := h[g00[x, y], g01[x, y]]
h1[x_, y_] := h[g10[x, y], g11[x, y]]
h01[x_, y_] := h[g010[x, y], g011[x, y]]
h10[x_, y_] := h[g100[x, y], g101[x, y]]
h11[x_, y_] := h[g110[x, y], g111[x, y]]
sumh[x_, y_] := he[x, y] + h0[x, y] + h1[x, y] + h01[x, y] + h10[x, y] + h11[x, y]
d[x_, y_] := t - int0 - sumh[x, y]
s[x_, y_] :=
 (w1[x, y] + w2[x, y]) * Log[3] / Log[2] - (w1[x, y] * Log[w1[x, y]] / Log[2] +
    w2[x, y] * Log[w2[x, y]] / Log[2] + w3[x, y] * Log[w3[x, y]] / Log[2])
gam[x_, y_] := S3 / (s[x, y] + 3 * d[x, y])
success[x_, y_] := S3 - 3 * gam[x, y] * d[x, y]
```

In[ ]:=

```
Clear[x, y]
FindMaximum[{success[x, y], -0.005 < x < 0.005 && -0.005 < y < 0.005},
 {x, 0}, {y, 0}, WorkingPrecision → 50, MaxIterations → 100]
```

Out[ ]=

```
{0.38624055093686612855267743386619647116907958108192,
 {x → -0.00030637621113654576934098860974131639522965997457504,
  y → 0.00015844276446973223969154342949394731476786546409130}}
```

Appendix 2

This is a program of Python in Jupyter Notebook format for computing the worst case. Please refer Algorithm 13 in page 74.

1 Initial Setting

We mainly use `sympy`, a libarary of Python, to do the symbolic computation.

`w1`, `w2` and `w3` are the symbols to denote w_1, w_2, w_3 in Chapter 4. `t` denotes the time threshold parameter t in the thesis. `p` denotes the value of placement function $\pi(x)$. `x` is used for solving some equations later.

[ ]:
```
from sympy import *
import copy

w1, w2, w3 = symbols('w1 w2 w3')    # weight vector
t = symbols('t')    # time
x = symbols('x')    # variable in some equations
p = symbols('p', )    # pi(x)
recordW = {}    # list for saving the results over w
recordT = {}    # list for saving the results over t

# output the results
# file = open('/Users/scorpioares/Dropbox/CS PhD/Doctor Thesis/EX-result_v2/
 ↪original.txt', 'a')



# initial X_P^b
dict_ini_f0 = {'empty': 0, '0': 0, '1': 0, '01': 0, '10': 0, '11': 0}
dict_ini_f1 = {'empty': 0, '0': 0, '1': 0, '01': 0, '10': 0, '11': 0}

dict_ini_prefix = {'empty': 1, '0': 1, '1': 1, '01': 1, '10': 1, '11': 1}
```

2 Searching space

As stated in Chapter 4, we check all weight vectors in VW. Firstly, we fix a value of w_1, and then fix a value of w_3. Obviously, the value of w_2 is decided by $1 - w_1 - w_3$. Then for t from 0.01 to 0.5, we run the program to search the *optimal* values of $\{f_P^b\}_{P,b}$ and record the values obtained. Finally, the program returns the *worst* value. For every given weight vector (w_1, w_2, w_3), we compute the

minimal value among diffrent t and record it as `minT`. If there is a value returned in a loop in the program that is smaller than `minT`, we are able to skip this weight vector and move to the next one because that we only need a worse result comparing the current worst one.

[ ]:
```
# initial parameters
startwit1 = 0.672
endwit1 = 0.290
startwit3 = 0.184
starttime = 0.01
stepT = 0.01    # increment of t after each loop
stepB = 0.001   # decrement of beta
stepW = 0.001   # increment or decrement of weight after each loop
minT = 0.386156    # for skipping non-worst case

wit1 = startwit1    # w1
wit3 = startwit3    # w3
wit2 = 1 - wit1 - wit3  # w2
time = starttime    # t
```

3 Formulas

Here, we define some formulas we need for later computation. Note that all these formulas have been metioned in Chapter 4. Note that we use $\hat{f}^0_{01}, \hat{f}^0_{10}, \hat{f}^1_{11}$ defined in (4.19, 4.21, 4.22), and the rest upper bound of f^b_P is given the naive upper bound v^b_P.

[ ]:
```
# integrals: the case how many vars occur before x under permutation pi
dict_integral = {'none': (t - t ** 2 + t ** 3 / 3), 'one': (t ** 2 - 2 * (t **␣
 ↪3) / 3), 'two': (t ** 3 / 3)}

# definitions of V_P^b
dict_v0 = {'empty': (2/3 * w1 + 1/3 * w2) * dict_integral['none'], '0': (1/3 *␣
 ↪w1) * dict_integral['one'], '1': (1/3 * w1 + 1/3 * w2) *␣
 ↪dict_integral['one'], '01': 1/3 * w1 * dict_integral['two'], '10': 1/3 * w1␣
 ↪* dict_integral['two'], '11': 1/3 * w2 * dict_integral['two']}

dict_v1 = {'empty': (1/3 * w1 + 2/3 * w2 + w3) * dict_integral['none'], '0': (1/
 ↪3 * w1 + 1/3 * w2) * dict_integral['one'], '1': (w3 + 1/3 * w2) *␣
 ↪dict_integral['one'], '01': 1/3 * w2 * dict_integral['two'], '10': 1/3 * w2␣
 ↪* dict_integral['two'], '11': w3 * dict_integral['two']}

v001 = 1/3 * w1 * dict_integral['two']


var_forced = t * (t-2) / (t-1) + 2 * log(1-t)   # force probability of a␣
 ↪variable in PPSZ from 0 to t
var_ppsz = t - (t * (t-2) / (t-1) + 2 * log(1-t))         # PPSZ success␣
 ↪probability until t
```

```
# upper bound of forced vars (hat(f)) where t <= 0.24

dict_small_hat_f0 = {'empty': (2/3 * w1 + 1/3 * w2) * dict_integral['none'],␣
 ↪'0': (1/3 * w1) * dict_integral['one'], '1': (1/3 * w1 + 1/3 * w2) *␣
 ↪dict_integral['one'], '01': dict_v0['01'] - 1/3 * w1 * (- t**4 /4 + 2/3 *␣
 ↪t**3 - var_forced), '10': dict_v0['10'] - 1/3 * w1 * (- t**4 /4 + 2/3 * t**3␣
 ↪- var_forced), '11': dict_v0['11']}

dict_small_hat_f1 = {'empty': (1/3 * w1 + 2/3 * w2 + w3) *␣
 ↪dict_integral['none'], '0': (1/3 * w1 + 1/3 * w2) * dict_integral['one'],␣
 ↪'1': (w3 + 1/3 * w2) * dict_integral['one'], '01': dict_v1['01'], '10':␣
 ↪dict_v1['10'], '11': dict_v1['11'] - 2/3 * w3 * (- t**4 /4 + 2/3 * t**3 -␣
 ↪var_forced)}


# upper bound of forced vars where t > 0.24

dict_big_hat_f0 = {'01': dict_v0['01'] - 1/3 * w1 * (- t**4 /4 + 2/3 * t**3 - 0.
 ↪00147078), '10': dict_v0['10'] - 1/3 * w1 * (- t**4 /4 + 2/3 * t**3 - 0.
 ↪00147078), '11': dict_v0['11']}

dict_big_hat_f1 = {'01': dict_v1['01'], '10': dict_v1['10'], '11':␣
 ↪dict_v1['11'] - 2/3 * w3 * (- t**4 /4 + 2/3 * t**3 - 0.00147078)}

# exponent of running time of biased-search (sw)
var_bs = (w1 * log(3/w1, 2) + w2 * log(3/w2, 2) + w3 * log(1/w3, 2))
```

4 Definitons of Functions (Methods)

Remark: in this program, we decrease the largest current β a little bit (`StepB`) everytime, which induces a small enough incresement of f_P^b. It is different from **Algorithm 13** in Chapter 4, and it may cause an issue that we may have a returned set $\{f_P^b\}$ whose sum is larger than the f_{opt}. Thus, we will go back to the previous result (a smaller $\{f_P^b\}$) during the program if the *final* one is larger than f_{opt}. Although the result may be worse than the optimal one, which means that we will lose some advantages about the final improvement, we can ensure no mistakes of finding the worst case.

4.1 `update_one(prefix, beta, f0, f1, yn)`

When the current β (the one decreased by `stepB`) is larger than 0.5, we update the f_P^b by solving the equation $\frac{v_P^b - x}{v_P^b + v_P^{1-b} - f_P^{1-b} - x} = \beta$, where x is the variable to be solved. `yn` is a Boolean variable to decide whether $v_P^0 > v_P^1$ at the initial state.

[3]:
```
def update_one(prefix: str, beta: float, f0: dict, f1: dict, yn: bool):
    """function to update forced vars when beta > 0.5"""
    if yn == True:
```

```
        return solve((dict_v0[prefix] - x) / (dict_v0[prefix] + dict_v1[prefix]␣
↪- x - f1[prefix]) - beta, x)[0].evalf(subs={w1:wit1, w2:wit2, w3:wit3, t:
↪time})
    else:
        return solve((dict_v1[prefix] - x) / (dict_v0[prefix] + dict_v1[prefix]␣
↪- x - f0[prefix]) - beta, x)[0].evalf(subs={w1:wit1, w2:wit2, w3:wit3, t:
↪time})
```

4.2 update_both(prefix, f0, f1)

When the current β becomes 0.5, we no longer increase one of $\{f_P^0, f_P^1\}$ but both of them to keep $\frac{v_P^b - f_P^b}{v_P^b + v_P^{1-b} - f_P^{1-b} - f_P^b} = 0.5$. However, there are many restrictions to the maximum value of f_P^b that we can increase. We need to consider the following cases:

- If we increase f_P^b or f_P^{1-b} too much, for example we increased x, it may cause $f_P^b + x > \hat{f}_P^b$ or $f_P^{1-b} + x > \hat{f}_P^{1-b}$. Thus, the maximum value of x should be bounded by $\min\{\hat{f}_P^b - f_P^b, \hat{f}_P^{1-b} - f_P^{1-b}\}$
- Similarly, x may be too large to let $\sum_{P,b} f_P^b$ is larger than $\int_0^t \left(\frac{p}{1-p}\right)^2$, we need to bound x by this constraint too.

[ ]:
```
def update_both(prefix: str, f0: dict, f1: dict):
    """function to update forced vars when beta = 0.5"""

    var_sum0 = (f0['empty'] + f0['0'] + f0['1'] + f0['01'] + f0['10'] +␣
↪f0['11'])
    var_sum1 = (f1['empty'] + f1['0'] + f1['1'] + f1['01'] + f1['10'] +␣
↪f1['11'] + v001.evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time}))

    # difference from the opt
    var_sa = var_forced.evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time}) - var_sum0␣
↪- var_sum1

    if time < 0.24:
        # increase f0 and f1 until reaching the forced condition (stop)
        if (f0[prefix] + var_sa/2) <= dict_small_hat_f0[prefix].evalf(subs={w1:
↪wit1,w2:wit2,w3:wit3,t:time}) and (f1[prefix] + var_sa/2) <=␣
↪dict_small_hat_f1[prefix].evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time}):
            f0[prefix] += var_sa / 2
            f1[prefix] += var_sa / 2
        else:   # increase f0 and f1 until one of them reached hat_f
            if dict_small_hat_f0[prefix].evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:
↪time}) - f0[prefix] < dict_small_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}) - f1[prefix]:
                var_inc = dict_small_hat_f0[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}) - f0[prefix]
                f0[prefix] = dict_small_hat_f0[prefix]
```

```
                f1[prefix] += var_inc
            else:
                var_inc = dict_small_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}) - f1[prefix]
                f1[prefix] = dict_small_hat_f1[prefix]
                f0[prefix] += var_inc
    else:
        if (f0[prefix] + var_sa/2) <= dict_big_hat_f0[prefix].evalf(subs={w1:
↪wit1,w2:wit2,w3:wit3,t:time}) and (f1[prefix] + var_sa/2) <=␣
↪dict_big_hat_f1[prefix].evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time}):
            f0[prefix] += var_sa / 2
            f1[prefix] += var_sa / 2
        else:   # increase f0 and f1 until one of them reached hat_f
            if dict_big_hat_f0[prefix].evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:
↪time}) - f0[prefix] < dict_big_hat_f1[prefix].evalf(subs={w1:wit1,w2:wit2,w3:
↪wit3,t:time}) - f1[prefix]:
                var_inc = dict_big_hat_f0[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}) - f0[prefix]
                f0[prefix] = dict_big_hat_f0[prefix]
                f1[prefix] += var_inc
            else:
                var_inc = dict_big_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}) - f1[prefix]
                f1[prefix] = dict_big_hat_f1[prefix]
                f0[prefix] += var_inc
```

4.3 check_sum(f0, f1) and check_hat(prefix, f0, f1, dictA)

The first one is to check whether we already have the $\{f_P^b\}_{P,b}$ we want. The second function is to check whether the current f_P^b (obtained by the function `update_one`) is larger than $\hat{f}_P^b$. If so, make f_P^b equal to its upper bound.

[ ]:
```
def check_sum(f0: dict, f1: dict):
    """function to check if the result is opt"""

    sum1 = (f0['empty'] + f0['0'] + f0['1'] + f0['01'] + f0['10'] + f0['11'])
    sum2 = (f1['empty'] + f1['0'] + f1['1'] + f1['01'] + f1['10'] + f1['11'] +␣
↪v001.evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time}))
    if sum1+sum2 >= var_forced.evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time}):
        return True
    else:
        return False


def check_hat(prefix: str, f0: dict, f1: dict, dictA: dict):
    """function to check if f reaches the upper bound hat(f)"""
```

```
    if time < 0.24:
        if dictA[prefix] == 1:
            if f0[prefix] >= dict_small_hat_f0[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}):
                dictA[prefix] = 2
                f0[prefix] = dict_small_hat_f0[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time})
            if f1[prefix] >= dict_small_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}):
                dictA[prefix] = 2
                f1[prefix] = dict_small_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time})
    else:
        if dictA[prefix] == 1:
            if f0[prefix] >= dict_big_hat_f0[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}):
                dictA[prefix] = 2
                f0[prefix] = dict_big_hat_f0[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time})
            if f1[prefix] >= dict_big_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time}):
                dictA[prefix] = 2
                f1[prefix] = dict_big_hat_f1[prefix].evalf(subs={w1:wit1,w2:
↪wit2,w3:wit3,t:time})


def entropy(var1: str, var2: str):
    """function that compute the value of h function (not real entropy) """

    return (-var1 * log(var1 / (var1 + var2), 2) - var2 * log(var2 / (var1 +␣
↪var2), 2))
```

5 Searching Process

We first fix a weight (w_1, w_2, w_3), and then search the results for different t to find the best (fastest running time) choice of t.

[ ]:
```
# begin "while wit1 > endwit1" here
while wit1 > endwit1:
    wit3 = startwit3

    # begin "while wit3 < 1 - wit1" here
    while wit3 < 1 - wit1:
        wit2 = 1 - wit1 - wit3

        time = starttime
```

```
        breaktrigger = False

        dict_record_fixtime = {}

        # begin "while time <= 0.5" here
        while time <= 0.5:
            dict_act_prefix = copy.deepcopy(dict_ini_prefix)

            opttrigger = False


            dict_f0 = copy.deepcopy(dict_ini_f0)
            dict_f1 = copy.deepcopy(dict_ini_f1)
            dict_old_f0 = copy.deepcopy(dict_ini_f0)
            dict_old_f1 = copy.deepcopy(dict_ini_f1)

            # dict to store initial beta
            dict_ini_beta = {'empty': False, '0': False, '1': False, '01':␣
↪False, '10': False, '11': False}

            # a trigger to tell whether v_P^0 is larger than v_P^1
            dict_is0larger = {'empty': False, '0': False, '1': False, '01':␣
↪False, '10': False, '11': False}

            for prefix in dict_act_prefix:
                if (dict_v0[prefix] / (dict_v0[prefix] + dict_v1[prefix])).
↪evalf(subs={w1:wit1, w2:wit2, w3:wit3, t:time}) >= (dict_v1[prefix] /␣
↪(dict_v0[prefix] + dict_v1[prefix])).evalf(subs={w1:wit1, w2:wit2, w3:wit3,␣
↪t:time}):
                    dict_is0larger[prefix] = True
                    dict_ini_beta[prefix] = (dict_v0[prefix] / (dict_v0[prefix]␣
↪+ dict_v1[prefix])).evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time})
                else:
                    dict_ini_beta[prefix] = (dict_v1[prefix] / (dict_v0[prefix]␣
↪+ dict_v1[prefix])).evalf(subs={w1:wit1,w2:wit2,w3:wit3,t:time})


            dict_temp_beta = copy.deepcopy(dict_ini_beta)


            var_beta = 1

            # print("ok")


            # if some f reaches its hat(f), then keep it fixed
```

```
            while max(dict_temp_beta.values()) > 0.5:
                for prefix in dict_act_prefix:
                    if dict_act_prefix[prefix] == 2:
                        dict_temp_beta.pop(prefix)
                        dict_act_prefix[prefix] = 0

                # the key beta, which is the max of all temp beta
                var_beta = max(dict_temp_beta.values())

                dict_inverse_beta = {v : k for k, v in dict_temp_beta.items()}
                var_max_prefix = dict_inverse_beta[var_beta]

                # every step, decrease the key beta a little
                var_beta -= stepB

                # if the decreased key beta is smaller than 0.5, then make it 0.
↪5 and update the value of f
                if var_beta < 0.5 and dict_act_prefix[var_max_prefix] == 1:
                    if dict_is0larger[var_max_prefix] == True:
                        dict_old_f0[var_max_prefix] = copy.
↪deepcopy(dict_f0[var_max_prefix])
                        dict_f0[var_max_prefix] = update_one(var_max_prefix, 0.
↪5, dict_f0, dict_f1, True)
                    else:
                        dict_old_f1[var_max_prefix] = copy.
↪deepcopy(dict_f1[var_max_prefix])
                        dict_f1[var_max_prefix] = update_one(var_max_prefix, 0.
↪5, dict_f0, dict_f1, False)
                    check_hat(var_max_prefix, dict_f0, dict_f1, dict_act_prefix)
                    if check_sum(dict_f0, dict_f1):
                        if dict_is0larger[var_max_prefix] == True:
                            dict_f0[var_max_prefix] =␣
↪dict_old_f0[var_max_prefix]
                        else:
                            dict_f1[var_max_prefix] =␣
↪dict_old_f1[var_max_prefix]
                        breaktrigger = True
                    elif dict_act_prefix[var_max_prefix] == 1:
                        dict_temp_beta[var_max_prefix] = 0.5

                # print(dict_temp_beta)

                # breaktrigger can tell whether we already have the opt result
                if breaktrigger:
                    break
```

```
                # update the value of f after decreasing the key beta
                if dict_act_prefix[var_max_prefix] == 1 and var_beta >= 0.5:
                    if dict_is0larger[var_max_prefix] == True:
                        dict_old_f0[var_max_prefix] = copy.
↪deepcopy(dict_f0[var_max_prefix])
                        dict_f0[var_max_prefix] = update_one(var_max_prefix,␣
↪var_beta, dict_f0, dict_f1, True)
                    else:
                        dict_old_f1[var_max_prefix] = copy.
↪deepcopy(dict_f1[var_max_prefix])
                        dict_f1[var_max_prefix] = update_one(var_max_prefix,␣
↪var_beta, dict_f0, dict_f1, False)
                    check_hat(var_max_prefix, dict_f0, dict_f1, dict_act_prefix)
                    if check_sum(dict_f0, dict_f1):
                        if dict_is0larger[var_max_prefix] == True:
                            dict_f0[var_max_prefix] =␣
↪dict_old_f0[var_max_prefix]
                        else:
                            dict_f1[var_max_prefix] =␣
↪dict_old_f1[var_max_prefix]
                        breaktrigger = True
                    elif dict_act_prefix[var_max_prefix] == 1:
                        dict_temp_beta[var_max_prefix] = var_beta

                if breaktrigger:
                    break

                # end "while time <= 0.5" here


            # beta = 0.5 and no stop
            # print("ok")

            # if all temp beta = 0.5 and opt has not been found
            if not breaktrigger:
                for prefix in dict_act_prefix:
                    if dict_act_prefix[prefix] == 1:
                            update_both(prefix, dict_f0, dict_f1)
                            dict_act_prefix[prefix] = 2

                    if check_sum(dict_f0, dict_f1):


                            breaktrigger = True
                            break
```

```
            if breaktrigger == False:
                print("there is no break when", wit1, wit3, time)

            delta = var_ppsz
            for prefix in dict_ini_prefix:
                delta -= entropy((dict_v0[prefix] - dict_f0[prefix]),␣
↪(dict_v1[prefix] - dict_f1[prefix]))

            # exponent part
            result_expo = ((2 * log(2) - 1) / (1 + (3 * delta / var_bs))).
↪evalf(subs={w1:wit1, w2:wit2, w3:wit3, t:time})

            print(result_expo)

            # if there is some result better than the one which is so far the␣
↪best (worst case), then the one (weight) we are checking must not be the␣
↪worst case, and we can skip it.

            if result_expo < minT:
                breaktrigger = True
                break

            # print(dict_ini_beta)


            # record the result and change to a new t
            dict_record_fixtime[(wit1, wit3, time)] = result_expo
            time += stepT

        # record the result for each weight
        if breaktrigger == False:
            minT = min(dict_record_fixtime.values())
            new_dict = {v : k for k, v in dict_record_fixtime.items()}
            recordW[new_dict[minT]] = minT
            print(new_dict[minT], minT, file=file, flush=True)


        # end "while wit3 < 1 - wit1" here
        wit3 += stepW
    # end "while wit1 > endwit1" here
    wit1 -= stepW
```