Smoothed Analysis of 0-1 Quadratic Programming

Yunyue Chen
Department of Informatics
King's College London
yunyue.chen@kcl.ac.uk

Jan 2023

Abstract

In this paper, I present a smoothed analysis of 0-1 quadratic programming. By analysing the running time of algorithms under random perturbations of the inputs, smoothed analysis framework can characterise the performance of algorithms in practice (beyond the worst case). Based on the works of Vöcking *et.al.* [1, 2] in 0-1 and integer linear programming, I extend the analysis towards the smoothed complexity of 0-1 quadratic programming problems.

My main results show that: any 0-1 quadratic programming problem with a *pseudo-polynomial* deterministic algorithm has *polynomial smoothed complexity*; any 0-1 quadratic programming problem with *polynomial smoothed complexity* has a randomised (Las Vegas) algorithm with *expected pseudo-polynomial* running time.

Furthermore, this paper also offers a cornerstone for the further understanding of real-world performance of ℓ_0-minimisation algorithms (learning with sparsity), which is highly valued by Machine Learning and Statistics communities.

Keywords: Smoothed Analysis, 0-1 Quadratic Programming, Pseudo-polynomial, Randomised Algorithms

Contents

Nomenclature

ϕ	Perturbation density parameter
f_ϕ	Perturbation model with parameter ϕ
$\|\cdot\|_\infty$	Sup norm
$\|\cdot\|_p$	L^p-norm
$\mathbb{R}$	Real numbers
$T(\cdot)$	Running time of algorithms
max	Maximisation
$\mathbb{E}[\cdot]$	Expectation
$\Pr[\cdot]$	Probability
poly	Polynomial
$O(\cdot)$	Big O notation
$(\cdot)^T$	Transpose
$*$	Convolution
$\|\cdot\|$	Absolute value
$\int$	Integral
$\{0,1\}^n$	Set of all possible 0-1 vectors with length n
$t \sim f$	Random variable t follows the distribution f
Δ_{obj}	*Objective gap*
Δ_{win}	*Winner gap*
Δ_i	*i-th gap*
$\bigcup$	Union of sets/events
$\lfloor\cdot\rfloor$	Rounding down
$\lceil\cdot\rceil$	Rounding up
δ	Changes of the *objective gap* due to rounding
δ'	*Perturbation gap*
$Q_{ij}^{(ad)}, b_k^{(ad)}$	Adversarial part of coefficients
$\tilde{Q}_{ij}, \tilde{b}_k$	Stochastic part of coefficients
$\vee$	Or/Union
$\wedge$	And/Intersection

List of Figures

List of Algorithms

1 Introduction

To characterise the efficiency of an algorithm with theoretical underpins, worst-case time complexity analysis used to be a successful and promising framework. Such pessimistic measure of running time, however, has failed to explain the success of many remarkable algorithms in practice, for their worst-case complexity can be exponential or higher-order polynomial. Facing this issue, average-case analysis [3, 4, 5] offers some insight into algorithms' performance in "common cases", but it requires a priori of the distribution of inputs, and what kind of distribution can be called "natural" is still debatable.

Since worst-case analysis is too pessimistic and average-case analysis needs fixed input distribution assumption, a new analysis method called smoothed analysis [6, 7, 8] was introduced and developed in the past twenty years. Smoothed analysis not only considers the stochastic factor in practice but also keeps the adversarial structure in worst-case analysis. On the one hand, smoothed time complexity is still defined as the algorithm's running time on the worst input instance. On the other hand, it models real-world noise as a stochastic perturbation on the instance. Due to the perturbation, some worst instances may not show up anymore, so the upper bound of running time will decrease. Because this framework only requires the model of random noise, such as Gaussian, smoothed time complexity only reveals the properties of the algorithm and disentangles the impact of data-dependent factors.

As an initial work of smoothed analysis, Spielman and Teng [7] show that the smoothed complexity of simplex method is polynomial, although its worst-case complexity is exponential. They assume that all the input numbers (vertex locations for example) are perturbed slightly by a zero-mean Gaussian distribution, then they define the smoothed time complexity as the expected running time over such perturbation. As for other smoothed analysis in linear programming, [9] and [10] also show that both perceptron algorithms and interior point algorithms have polynomial smoothed complexity. In machine learning, [11] and [12] study the smoothed complexity of k-means, which is much lower than its worst-case complexity. For more smoothed complexity research in PAC-learning, multi-objective optimisation, and low-rank quasi-concave minimization, I refer the readers to [13], [14], and [15].

As for combinational optimisation problems, Vöcking *et.al.* [1, 2] extend smoothed analysis from continuous linear programming to binary and integer linear programming. Under their settings, every coefficient in objective and constraints can be randomly perturbed by a perturbation model. By comparing the changes of objective values due to perturbation and the sensitivity of the feasible solutions' order, they show a connection between polynomial smoothed complexity and pseudo-polynomial expected/worst-case complexity. Moreover, they characterise the perturbation distribution by its maximum density (sup norm) ϕ, so their analysis can cover various probability distributions, including, but not limited to, Gaussian perturbation.

1.1 Contributions

In this paper, I further extend the analysis toward 0-1 quadratic programming. Unlike the linear case, the estimation of the upper bound or lower bound of the objective values changes and perturbation sensitivity is not trivial due to the nature of quadratic form. My analysis not only adopts the methods and tools from combinational optimisation and concentration inequality but also introduces conclusions from real analysis. My

contributions are as follows:

- I generalise the isolation lemma to 0-1 quadratic programming, finding the probability bound of the objective value gap between the optimal solution and the second optimal solution, which we call *winner gap* (Lemma 3).
- I prove that under the perturbation, the optimal solution is still fixed with high probability if we only know the first b bits after the binary point of the coefficients (Lemma 4).
- My main result shows connections between pseudo-polynomial expected/worst-case complexity and polynomial smoothed complexity for 0-1 quadratic programming (Theorem 4).
  - If a 0-1 quadratic programming has polynomial smoothed complexity, then it has pseudo-polynomial expected complexity.
  - If a 0-1 quadratic programming has pseudo-polynomial worst-case complexity, then it has polynomial smoothed complexity.

1.2 Structure of the Paper

In Section 2, we look into backgrounds and preliminaries, ranging from the formal definition of smoothed analysis framework to Young's convolution inequality in real analysis. In Section 3, we extend Beier and Vöcking [1]'s conclusion on *winner gap* from linear case to quadratic case. Finally, we present and prove the main theorem of 0-1 quadratic programming. In Section 4, we apply the main result to three typical problems in complexity analysis, then we discuss a possible future research direction.

2 Preliminaries and Literature Review

2.1 Smoothed Complexity Analysis

The basic idea of smoothed analysis is that after random perturbation, some worst input instances will disappear, thus the worst-case running time of algorithms will also decrease. To understand this framework intuitively, we need to introduce the adversary framework in algorithm design.

2.1.1 Adversary Framework in Algorithm Design

Consider the process of solving problems as a game between an algorithm and an adversary. The aim of the adversary is to construct some extremely bad problem instances, while the aim of the algorithm is to solve the given problem efficiently. Although an algorithm is designed to solve a class of problems, its running time depends on the difficulty of the input instances allocated by the adversary.

In the traditional worst-case analysis, the adversary has the ability to construct arbitrarily bad instances, so an algorithm that can work fine in most cases may still have bad behaviour on such worst instances. In the design of a randomised algorithm [16], the aim of introducing randomisation is to avoid those worst cases in most of the runs and decrease the time complexity in expectation. As for average-case analysis [4, 5],

it does not deal with the adversary but changes the definition of time complexity as the average running time.

In smoothed analysis, except for the algorithm and its adversary, there is also an environment. After the adversary allocates the problem instance, the environment can always perturb it slightly, and the algorithm just needs to solve the perturbed problem (we call it a "smoothed problem"). Since the perturbation is free from the adversary's control, there is no guarantee that the deliberately constructed worst instance remains the worst when the algorithm solves it. For example, consider a binary search problem on a line segment, the adversary can specify a very weird position such that the algorithm needs a large number of iterations to locate. However, under smoothed setting, such position is perturbed randomly, so its position will be randomly distributed on a small interval rather than a single point. Therefore, if the left end or the right end of the search range falls within such an interval, the algorithm can return correctly without further iteration with a certain probability. Compute the expectation of running time over the perturbation distribution, it will be shorter than the worst-case running time without perturbation.

In the following texts, we will call the problem without perturbation "adversarial problem" and the perturbed problem "smoothed problem". The usage of these terms in our paper may not correspond to the jargon from other fields.

2.1.2 Perturbation Model

To characterise the random perturbation, we define our perturbation model as:

Definition 1 The perturbation model is a class of continuous bounded probability density functions $f_\phi : \mathbb{R} \to \mathbb{R}_{\geq 0}$ with sup norm $\|f_\phi\|_\infty = \max\{|f(t)|\} = \phi$ and finite expected absolute value

$$\int |t|\, f_\phi(t)\mathrm{d}t = \frac{E}{\phi}, \tag{1}$$

where E is a constant and ϕ is the perturbation density parameter.

Following the same idea of Vöcking *et.al.* [1, 2], we do not restrict our noise distribution to be Gaussian. Thus our result can generalise well to many other possible perturbation distributions. Intuitively, the larger ϕ, the closer we are to the adversarial problems for some extreme cases have a very large probability. However, if ϕ is small enough, the perturbation distribution tends to be a uniform distribution.

Example 1 (Gaussian perturbation) Consider a Gaussian perturbation model, the distribution density function is

$$f_{\phi_G}(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2}, \tag{2}$$

where μ is the mean and σ^2 is the variance. According to our abstract definition of perturbation model, its density parameter is

$$\phi_G = \frac{1}{\sigma\sqrt{2\pi}}, \tag{3}$$

which is closely related to its variance. If such a perturbation applies on a fixed coefficient x, then the smoothed version of x will be a random variable $x + t, t \sim f_{\phi_G}$.

2.1.3 Smoothed Complexity

Based on the definition of the perturbation model, we can define smoothed time complexity as the expected running time over the perturbation distribution.

Definition 2 (Smoothed complexity [1, 8]) Given a perturbation model f_ϕ, the adversarial part of problem instance I, and a constant $\alpha > 0$, the smoothed complexity of an algorithm A is defined as a function of its input size n and the perturbation parameter ϕ:

$$C_{smoothed}(n, \phi) = \max_{I} \left\{ \mathbb{E}_{f_\phi} \left[(T(A(I + f_\phi)))^\alpha \right] \right\}. \tag{4}$$

The constant α here is for the robustness of the definition, we will set $\alpha = 1$ for simplicity in the rest of this paper. In comparison to smoothed complexity, we state the definition of average-case complexity here.

Definition 3 (Average-case complexity [17]) Suppose the input instance I have a distribution $\mathcal{D}_I$, we define average-case complexity as the expected running time over the distribution of inputs, which is

$$C_{ave}(n) = \mathbb{E}_{\mathcal{D}_I} \left[T(A(I)) \right]. \tag{5}$$

In average-case analysis, the expectation is computed over the distribution of inputs, while the expectation in smoothed complexity is computed over the distribution of perturbation noise. Thus, unlike average-case complexity, smoothed complexity is still a worst-case measure of the running time. To understand smoothed analysis intuitively, here is a toy example.

Example 2 (Worst-case analysis, average-case analysis, and smoothed analysis) Consider a toy case where we use Bubble sort algorithm to sort all the possible arrays containing numbers 0 to 5 in ascending order. It is obvious that it will take a longer time to sort $(5, 4, 3, 2, 1, 0)$ than sorting $(0, 1, 2, 3, 4, 5)$. If we list all the permutations of these six numbers and run Bubble sort on each instance, we can get an instances-running time diagram (see Figure 1).

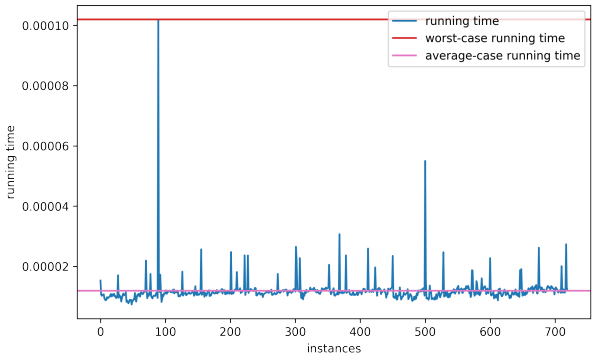

Figure 1: Worst/Average-case complexity

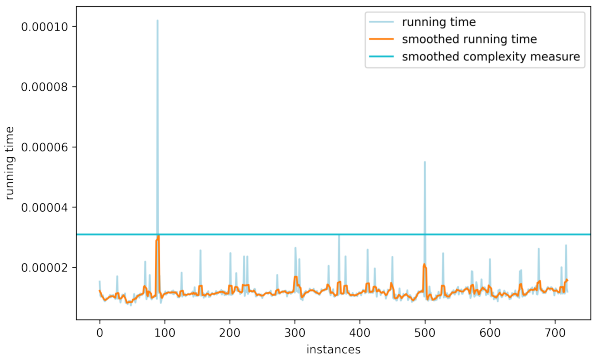

Figure 2: Smoothed complexity

In Figure 1, although the running time in most cases is relatively low, the worst-case running time (red line) is extremely high. Since such an extreme case is not common, worst-case complexity is too pessimistic to measure the real-world performance of an algorithm. Suppose all the instances show up with uniform distribution, then the average-case

running time (purple line) will be far lower than the worst-case running time, although it is too optimistic.

As for smoothed analysis (Figure 2), suppose every instance can be perturbed to be its first neighbours (move one step left or right along the "instance" axis) or second neighbours (move two steps left or right along the "instance" axis), the smoothed running time (orange line) is a convolution between the running time (blue line) and the perturbation model (uniform distribution). Due to the smoothing process, the peaks (bad cases) in running time (blue line) are suppressed, and the smoothed complexity (cyan line) is lower than the worst-case running time but not as optimistic as the average-case running time.

Now, we can define polynomial smoothed complexity as follows.

Definition 4 (Polynomial smoothed complexity [8]) Given the perturbation model f_ϕ, the adversarial problem instance I with input size n, and a constant $\alpha > 0$, an algorithm A has polynomial smoothed complexity if there exists a polynomial $\text{poly}(n, \phi)$ such that

$$C_{smoothed}(n, \phi) = \max_I \left\{ \mathbb{E}_{f_\phi} \left[\left(T \left(A \left(I + f_\phi \right) \right) \right)^\alpha \right] \right\} = O \left(\text{poly} \left(n, \phi \right) \right). \tag{6}$$

Without loss of clarity, we will rewrite Equation (6) as

$$\mathbb{E}_{f_\phi} \left[T(A) \right] = O \left(\text{poly}(n, \phi) \right) \tag{7}$$

for simplicity in the rest of this paper. Despite this definition being based on expectation, we can also define polynomial smoothed complexity by probability bounds.

Definition 5 (An equivalent definition of polynomial smoothed complexity [2]) Given the perturbation model f_ϕ and the adversarial instance I with input size n, the running time of a polynomial smoothed algorithm A satisfies

$$\Pr \left[T(A(I + f_\phi)) > \text{poly} \left(n, \phi, \frac{1}{\epsilon} \right) \right] \leq \epsilon, \tag{8}$$

where $\epsilon \in (0, 1)$.

For simplicity, we will also leave out the notion of I and f_ϕ, rewriting Equation (8) as

$$\Pr \left[T(A) > \text{poly} \left(n, \phi, \frac{1}{\epsilon} \right) \right] \leq \epsilon \tag{9}$$

in our analysis, if the context is clear. In Section 3.4.2, we are going to use Equation (9) to prove that the algorithm we constructed has polynomial smoothed complexity.

2.1.4 Smoothed Analysis of 0-1 Linear Programming

Before we start the analysis of 0-1 quadratic programming, it is necessary to review the work of Beier and Vöcking [1]. They establish a connection between pseudo-polynomial complexity and polynomial smoothed complexity for 0-1 linear programming.

Theorem 1 (Smoothed complexity of 0-1 linear programming [1]) *A 0-1 linear programming problem has smoothed polynomial complexity if and only if it has (expected) pseudo-polynomial complexity.*

In general, they prove this theorem in two directions: using a pseudo-polynomial deterministic algorithm to construct a polynomial smoothed algorithm; using a polynomial smoothed algorithm to construct a pseudo-polynomial expected complexity randomised algorithm.

For the first direction, since the coefficients are unfixed stochastic values in smoothed problems, there is no way to directly apply adversarial algorithms designed for fixed coefficients. To tackle this issue, they introduce a method called "adaptive rounding procedure". Suppose all the coefficients are represented in binary code, if we reveal the first b bits after the binary point and leave the rest part of the coefficients to be still unknown, we have a b-bit rounded version of the smoothed problem. If the rounding error does not affect the optimality of the best solution, then the optimal solution for the rounded problem is also the optimal solution for the original smoothed problem. To characterise the sensitivity of optimality, they have the following lemma.

Lemma 1 (Optimality in 0-1 linear programming [1]) *Consider the perturbation on the objective function $\boldsymbol{c}^T\boldsymbol{x}$ with parameter ϕ, if we reveal b bits after the binary point of all coefficients, then with a probability of at least $1-\frac{2n^2\phi}{2^b}$, the optimal solution of the rounded problem is also the optimal solution of the original smoothed problem.*

With the help of Lemma 1,they reveal the coefficients bit by bit until the adversarial algorithm outputs the correct optimal solution. By analysing the running time of the constructed algorithm, they prove that its smoothed complexity is polynomial according to Equation (9).

In this paper, we extend Lemma 1 to 0-1 quadratic programming in Section 3.2 and 3.3, then we prove how to construct a polynomial smoothed complexity algorithm from a pseudo-polynomial adversarial algorithm for 0-1 quadratic programming in Section 3.4.2.

As for the second direction, they construct a randomised algorithm that can perturb the coefficients actively, calling the smoothed algorithm to solve the perturbed problem. By controlling the perturbation magnitudes, the order of solutions will not be changed, so the optimal solution for the perturbed problem is still the optimal solution for the original adversarial problem. After analysing the success probability and expected running time, they show that such a randomised algorithm has pseudo-polynomial expected complexity.

Following a similar idea, in Section 3.4.1, we elaborate on how to construct a randomised algorithm with pseudo-polynomial expected running time by an algorithm with polynomial smoothed complexity for 0-1 quadratic programming.

2.2 Other Background Theories

In this section, we are going to list some concepts and tools that are needed for our analysis.

2.2.1 Pseudo-polynomial Complexity

Definition 6 (Pseudo-polynomial time [18]) Let M denote the largest value of the input numbers, if the running time of an algorithm A satisfies

$$T(A) = O\left(\text{poly}\left(n, M\right)\right), \tag{10}$$

the time complexity of A is pseudo-polynomial.

Many NP-complete/hard problems have pseudo-polynomial complexity, and they are called weakly NP-complete/hard. As for those problems that have no pseudo-polynomial algorithms, they are called strongly NP-complete or strongly NP-hard.

According to our main result (Theorem 4), if a 0-1 quadratic programming is strongly NP-hard, then it does not have a polynomial smoothed complexity.

2.2.2 Randomised Algorithm

In Section 3.4.1, we will construct a randomised algorithm by a smoothed algorithm and analysis its expected running time. Now we are going to introduce some basic knowledge in randomised computing. We start from a simple definition of randomised algorithms, for more systematic definitions based on probabilistic Turing machine, we refer the readers to [17, 19].

Definition 7 A randomised algorithm can make randomised choices during its running with a probability distribution.

Since the execution involves probabilistic decisions, a typical randomised algorithm always does not guarantee the correctness of its output. To measure the "reliability" of randomized algorithms, we can define the success rate/probability.

Definition 8 Let $I(x)$ denotes the problem mapping with input x, an algorithm A has a correct output if $A(x) = I(x)$. The probability

$$\Pr\left[A(x) = I(x)\right] \tag{11}$$

is called algorithm A's success probability.

Although a randomised algorithm cannot always return the correct output, if its success probability is above $\frac{1}{2}$, we can run the algorithm repeatedly, and the probability that we finally have the correct output will approach 1 quickly.

2.2.3 Worst-case Complexity and Expected Complexity

We have two feasible measures for the efficiency of randomised algorithms: worst-case running time and expected running time. Similar to the analysis of deterministic algorithms, the worst-case complexity of randomised algorithms is the maximum running time regardless of the distribution of random choices.

Definition 9 (Worst-case complexity of randomised algorithm [19]) Given problem instances I, the worst-case complexity of a randomised algorithm A is defined as

$$C\left(n\right) = \max_{I}\left\{T\left(A\left(I\right)\right)\right\}, \tag{12}$$

where n is the input size.

As for expected complexity, it measures the maximum expected running time over the distribution of random choices.

Definition 10 (Expected complexity of randomised algorithm [19]) Given a problem instance I and the distribution of random choices $\mathcal{D}$, the expected complexity of a randomised algorithm A is defined as

$$C_{exp}(n) = \max_{I}\left\{T_{exp}(A(I))\right\} = \max_{I}\left\{\mathbb{E}_{\mathcal{D}}\left[T(A(I))\right]\right\}, \tag{13}$$

where n is the input size.

2.2.4 Las Vegas Algorithm

In Section 3.4.1, the randomised algorithm we constructed is a Las Vegas algorithm, whose worst-case complexity and expected complexity are closely related. Las Vegas algorithms have two equivalent definitions:

Definition 11 (First definition of Las Vegas algorithm [16]) A randomised algorithm A computing a function $I(x)$ is a Las Vegas algorithm if its success probability

$$\Pr\left[A(x) = I(x)\right] = 1 \tag{14}$$

for any input x.

The first definition characterises Las Vegas algorithms as "zero error" algorithms, and we always study its expected running time rather than the worst-case running time. As for the second definition of Las Vegas algorithms, we still do not allow the algorithm to make mistakes, but we allow the algorithm to return *"fail"*.

Definition 12 (Second definition of Las Vegas algorithm [16]) A Las Vegas algorithm A that computes function $I(x)$ can output *"fail"* or the correct answer for each input x, such that

$$\Pr\left[A(x) = I(x)\right] \geq \frac{1}{2}, \text{ and } \Pr\left[A(x) = \textit{"fail"}\right] = 1 - \Pr\left[A(x) = I(x)\right] \leq \frac{1}{2}. \tag{15}$$

Given an algorithm that satisfies the first definition, we can run the algorithm and force it to return *"fail"* if its running time exceeds a time bound. By setting the time bound carefully, we can make sure the altered algorithm's success probability is above $\frac{1}{2}$ and satisfy the second definition. As for an algorithm that follows the second definition, we can run such an algorithm repeatedly until it returns the correct answer, so it will correspond to the first definition of Las Vegas algorithms.

As for the relationship between the expected running time in the first definition and the worst-case running time in the second definition, we have such a lemma from [16].

Lemma 2 *Suppose A_1 is the Las Vegas algorithm that follows the first definition, and A_2 is the Las Vegas algorithm that follows the second definition, we have*

$$T_{exp}(A_1) = O\left(T(A_2)\right). \tag{16}$$

At the end of Section 3.4.1, we prove the running time of the constructed Las Vegas algorithm (which follows the second definition) is pseudo-polynomial. Then we claim that such an algorithm also has pseudo-polynomial expected running time according to Lemma 2.

2.2.5 Convolution formula and Young's convolution inequality

In Section 3.2, we need to find an upper bound for the probability with the form

$$\Pr\left[\left|\sum_i x_i\right| < c\right], \tag{17}$$

in which c is a constant, and $\sum_i x_i$ is a sum of independent random variables. If we know that the sup norm of the probability density function of $\sum_i x_i$ is ϕ, then we can have the upper bound

$$\Pr\left[\left|\sum_i x_i\right| < c\right] \leq 2c\phi. \tag{18}$$

Therefore, we need to compute the probability density function of a sum of independent random variables and find its sup norm, which requires the convolution formula for density functions and Young's convolution inequality.

Theorem 2 (Convolution formula for density functions [20]) *Let X and Y be independent random variables, then $X + Y$ has a probability density function*

$$f_{X+Y} = f_X * f_Y, \tag{19}$$

which is the convolution of f_X and f_Y.

Suppose we have a sum of n independent and identically distributed (*i.i.d.*) random variables, the probability density function is

$$f_{\sum_i^n x_i} = (f_x)^{*n}, \tag{20}$$

which is called "n-fold self-convolution" of f_x.

Now we can use Young's convolution inequality to estimate an upper bound for its sup norm.

Theorem 3 (Young's convolution inequality [21]) *Let $f \in L^p$, $g \in L^q$, and $\frac{1}{p} + \frac{1}{q} = \frac{1}{r} + 1$ with $1 \geq p, q, r \leq \infty$, then $f * g \in L^r$ and*

$$\|f * g\|_r \leq \|f\|_p \|g\|_q, \tag{21}$$

where L^p, L^q, L^r are so-called Lebesgue spaces[1], and

$$\|f\|_p = \left(\int |f(x)|^p \, \mathrm{d}x\right)^{1/p} \tag{22}$$

is the L^p norm of f.

In Section 3.2, we set $p = \infty, q = 1, r = \infty$ so that for the self-convolution of f_x, we have

$$\|f_x * f_x\|_\infty \leq \|f_x\|_\infty \|f_x\|_1. \tag{23}$$

As a probability density function, $f_x : \mathbb{R} \to \mathbb{R} \geq 0$ has

$$\|f_x\|_1 = \int |f_x(x)| \mathrm{d}x = 1, \tag{24}$$

so we have

$$\|f_x * f_x\|_\infty \leq \|f_x\|_\infty = \phi, \tag{25}$$

if the sup norm of f_x is ϕ.

For a more comprehensive view of convolution of probability density functions and their sup norm, I refer the readers to [20], [21], and [22].

[1] *Lebesgue spaces* are function spaces with generalised p-norm defined. In Section 3.2, we will adopt Young's convolution inequality as a tool, and the knowledge related to real analysis is not required.

3 Main Results

3.1 Assumptions and Preliminary Settings

In this section, we are going to analyse the connections between pseudo-polynomial time complexity and polynomial smoothed time complexity in 0-1 quadratic programming. At the beginning, we need to list a few statements about our analysis framework.

- Without loss of generality, we consider the smoothed maximisation problem of 0-1 quadratic programming with the form

$$\max F_q(\boldsymbol{x}) = \max \left\{ \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} \right\} \text{ s.t. } \boldsymbol{x} \in \mathcal{S}, \tag{26}$$

  in which $\boldsymbol{x} \in \{0,1\}^n$, $\boldsymbol{Q}$ is a $n \times n$ symmetric matrix, $\boldsymbol{b}$ is a n-dimensional vector, and $\mathcal{S}$ is the feasible region.

- For simplicity, we only consider the perturbed objective function, while the constraints (feasible region) are fixed.

- In the perturbed 0-1 quadratic programming, the coefficients of $\boldsymbol{Q}$ and $\boldsymbol{b}$ will be perturbed by adding *i.i.d.* random variables $t \sim f_\phi$. We can decompose each coefficient into adversarial part and stochastic part

$$Q_{ij} = Q_{ij}^{(ad)} + t_{ij}, \text{ and } b_k = b_k^{(ad)} + t_k, \tag{27}$$

  where t is the added perturbation random variable (stochastic part), while $Q_{ij}^{(ad)}$ and $b_k^{(ad)}$ are the coefficients before the perturbation (adversarial part). The structure of quadratic programming requires symmetric matrix $\boldsymbol{Q}$, so we have $t_{ij} = t_{ji}$.

- The adversarial version (unperturbed version) of the problem can be written as

$$\max F_q^{(ad)}(\boldsymbol{x}) = \max \left\{ \boldsymbol{x}^T \boldsymbol{Q}^{(ad)} \boldsymbol{x} + \boldsymbol{b}^{(ad)^T} \boldsymbol{x} \right\} \text{ s.t. } \boldsymbol{x} \in \mathcal{S}. \tag{28}$$

  We suppose that all the coefficients fall into the interval $[-1, 1]$ before the perturbation. This condition can be satisfied by scaling.

With respect to the perturbation setting, we can let different coefficients have different perturbation models ($t_{ij} \sim f_{\phi_{ij}}$ and $\phi_{ij} \neq \phi_{i'j'}$ if $i \neq i' \vee j \neq j'$), but in practise we often assume the noise to be independent and identically distributed. Therefore we only consider that all the coefficients are perturbed by the same perturbation model in the following analysis.

3.2 The Properties of Winner Gap

To extend the conclusion of Vöcking *et.al.* [1, 2] to quadratic objective, we need to generalise the isolation lemma to quadratic programming, which reveals the property of winner gap. We start from the definition of objective gap and winner gap. Suppose the size of feasible region is large enough.

Definition 13 (*Objective gap*) For any $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{S}, F_q(\boldsymbol{x}') \neq F_q(\boldsymbol{x})$, the *objective gap* is

$$\Delta_{obj} = |F_q(\boldsymbol{x}) - F_q(\boldsymbol{x}')| = \left| \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} - \boldsymbol{x}'^T \boldsymbol{Q} \boldsymbol{x}' - \boldsymbol{b}^T \boldsymbol{x}' \right|. \tag{29}$$

If two feasible solutions $\boldsymbol{x}$ and $\boldsymbol{x}'$ have the same objective value ($\Delta_{obj} = 0$), we do not distinguish them. Based on the definition of *objective gap*, we can define *winner gap*, which is the *objective gap* between the optimal solution $\boldsymbol{x}^*$ and the second optimal solution $\boldsymbol{x}^{**}$.

Definition 14 (*Winner gap*) For the optimal solution $\boldsymbol{x}^*$ and second optimal solution $\boldsymbol{x}^{**}$, we define the winner gap as

$$\Delta_{win} = F_q(\boldsymbol{x}^*) - F_q(\boldsymbol{x}^{**}) = \boldsymbol{x}^{*T}\boldsymbol{Q}\boldsymbol{x}^* + \boldsymbol{b}^T\boldsymbol{x}^* - \boldsymbol{x}^{**T}\boldsymbol{Q}\boldsymbol{x}^{**} - \boldsymbol{b}^T\boldsymbol{x}^{**} > 0. \tag{30}$$

In smoothed problems, all the coefficients are random variables, so we cannot directly apply any deterministic algorithms to it. However, we can use a rounding procedure (see Section 3.3) to obtain a rounded problem with fixed coefficients. If the *winner gap* is large enough such that the optimal solution is securely isolated from all the sub-optimal solutions, then the optimal solution for the rounded problem is also the optimal solution for the original smoothed problem. Such properties imply the relationship between worst-case analysis and smoothed analysis.

Based on this idea, we continue to find the upper/lower bounds for the *winner gap*.

Lemma 3 (Generalised isolating lemma for 0-1 quadratic programming) *Let ϕ be the density parameter. For any $\epsilon \geq 0$*

$$\Pr[\Delta_{win} < \epsilon] \leq \epsilon\phi n. \tag{31}$$

Proof. Suppose the feasible region $\mathcal{S}$ is large enough so that we can define ith gap

$$\Delta_i = F_q(\boldsymbol{x}^*) - F_q(\neg_i\boldsymbol{x}^*) > 0 \tag{32}$$

for each $i \in [n]$, where $\neg_i\boldsymbol{x}^* = \arg\max\{f_q(\boldsymbol{x})|\boldsymbol{x} \in \mathcal{S}, x_i \neq x_i^*\}$, the optimal solution if we restrict x_i to be not equal to x_i^*. All of the $\neg_i\boldsymbol{x}^*$ form a set of "sub-optimal solutions" $\neg\mathcal{S}$.

For 0-1 quadratic programming, the optimal solution $\boldsymbol{x}^*$ and second optimal solution $\boldsymbol{x}^{**}$ must be different at least for 1 bit, which means $\exists i \in [n], x_i^* \neq x_i^{**}$. Suppose the difference is only 1 bit, then $\boldsymbol{x}^{**}$ must be an element of $\mathcal{S}$. If the difference is larger then one bit, then $\boldsymbol{x}^{**}$ is still within on of the sets $\{\boldsymbol{x} \in \mathcal{S}|x_i \neq x_i^*\}$. According to the definition of $\neg_i\boldsymbol{x}^*$, we still have $\boldsymbol{x}^{**} = \neg_i\boldsymbol{x}^* \in \mathcal{S}$. Therefore, there exists an index i such that $\Delta_{win} = \Delta_i$ and

$$\Pr\left[\Delta_{win} < \epsilon\right] = \Pr\left[\exists i, \Delta_i < \epsilon\right] \leq \Pr\left[\bigcup_{i=1}^{n} \Delta_i < \epsilon\right] \leq \sum_{i=1}^{n} \Pr\left[\Delta_i < \epsilon\right]. \tag{33}$$

Since we know the sup norm of the perturbation probability distribution f_ϕ, we can find a upper bound for $\Pr\left[\Delta_i < \epsilon\right]$.

Now for a fixed i, we can divide the feasible region $\mathcal{S}$ into two subsets

$$\mathcal{S}^{(0)} = \{\boldsymbol{x} \in \mathcal{S} \mid x_i = 0\} \text{ and } \mathcal{S}^{(1)} = \{\boldsymbol{x} \in \mathcal{S} \mid x_i = 1\}. \tag{34}$$

Because $\exists i \in [n], x_i^* \neq x_i^{**}$, the optimal solution and the second optimal solution cannot fall within the same subset $\mathcal{S}^{(0)}$ or $\mathcal{S}^{(1)}$. We can rewrite the *i-th gap* as

$$\Delta_i = \left|F_q(\boldsymbol{x}^{(1)}) - F_q(\boldsymbol{x}^{(0)})\right|, \tag{35}$$

where $\boldsymbol{x}^{(1)} = \arg\max_{\boldsymbol{x}\in\mathcal{S}^{(1)}} \{F_q(\boldsymbol{x})\}$ and $\boldsymbol{x}^{(0)} = \arg\max_{\boldsymbol{x}\in\mathcal{S}^{(0)}} \{F_q(\boldsymbol{x})\}$. Then we have

$$\begin{aligned}\Delta_i &= \left|\left(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}\right)^T \boldsymbol{Q}\left(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}\right) + \boldsymbol{b}^T\left(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}\right)\right| \\ &= \left|\left(\boldsymbol{x}^{(diff)} + \boldsymbol{e}_i\right)^T \boldsymbol{Q}\left(\boldsymbol{x}^{(diff)} + \boldsymbol{e}_i\right) + \boldsymbol{b}^T\left(\boldsymbol{x}^{(diff)} + \boldsymbol{e}_i\right)\right|,\end{aligned} \tag{36}$$

where $\boldsymbol{e_i}$ is the standard basis vector that only $e_i = 1$ and $\boldsymbol{x}^{(diff)}$ is the rest difference between $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(0)}$.

For any 0-1 vector $\boldsymbol{v}$, let $\text{supp}(\boldsymbol{v})$ represent the set of indices with non-zero values (support set), and we have

$$\boldsymbol{v}^T\boldsymbol{Q}\boldsymbol{v} = \sum_{\substack{i\in\text{supp}(\boldsymbol{v}) \\ j\in\text{supp}(\boldsymbol{v})}} Q_{ij} \text{ and } \boldsymbol{b}^T\boldsymbol{v} = \sum_{k\in\text{supp}(\boldsymbol{v})} b_k. \tag{37}$$

Based on such fact, we observe that the coefficient b_i is special since $\boldsymbol{b}^T\boldsymbol{e}_i = b_i$, and b_i is not within $\boldsymbol{b}^T\boldsymbol{x}^{(diff)}$ because $x_i^{(diff)} = 0$. For the same reason, Q_{ab}, where $(a = i, b \in \text{supp}(\boldsymbol{x}))$ or $(a \in \text{supp}(\boldsymbol{x}), b = i)$ are also not included in ${\boldsymbol{x}^{(diff)}}^T\boldsymbol{Q}\boldsymbol{x}^{(diff)}$, but

$${\boldsymbol{x}^{(diff)}}^T\boldsymbol{Q}\boldsymbol{e}_i + \boldsymbol{e}_i^T\boldsymbol{Q}\boldsymbol{x}^{(diff)} + {\boldsymbol{e}_i}^T\boldsymbol{Q}\boldsymbol{e}_i = \sum_{a,b} Q_{ab}. \tag{38}$$

Suppose that all the rest coefficients are fixed arbitrarily, we have

$$\begin{aligned}\Pr\left[\Delta_i < \epsilon \mid \textit{Config}\right] &= \Pr\left[\left|{\boldsymbol{e}_i}^T\boldsymbol{Q}\boldsymbol{e}_i + \boldsymbol{b}^T\boldsymbol{e}_i\right| < \epsilon \;\middle|\; \textit{Config}\right] \\ &= \Pr\left[\left|\sum_{a,b} Q_{ab} + b_i\right| < \epsilon \;\middle|\; \textit{Config}\right],\end{aligned} \tag{39}$$

where *Config* denotes the fixed configuration of the rest coefficients and

$$(a,b) \in \{(a,b) \mid (a = i, b \in \text{supp}(\boldsymbol{x})) \vee (a \in \text{supp}(\boldsymbol{x}), b = i)\}. \tag{40}$$

If we know the density parameter ϕ_q for $\sum_{a,b} Q_{ab} + b_i$, then we can easily draw the conclusion that $\Pr\left[\Delta_i < \epsilon\right] \leq 2\epsilon\phi_q$. Therefore, the last part of the proof is to compute the density upper bound for the distribution of $\sum_{a,b} Q_{ab} + b_i$.

Recall that we have $Q_{ij} = Q_{ij}^{(ad)} + t_{ij}$, $b_k = b_k^{(ad)} + t_k$, and $t_{ij} = t_{ji}$ for the symmetric property. We can further simplify Equation (39) as

$$\Pr\left[\Delta_i < \epsilon \mid \textit{Config}\right] = \Pr\left[\left|2\sum_j t_{ij} + t_{ii} + t_i\right| < \epsilon \;\middle|\; \textit{Config}\right], \tag{41}$$

where $j \in \text{supp}(\boldsymbol{x}) - \{i\}$. Since the probability distribution function of a sum of random variables equals to the convolution of their probability distribution functions (Theorem 2), let $\alpha = 2\sum_j t_{ij}$, its distribution density function is

$$f_{\phi_\alpha} = \frac{1}{2} f^{*(|\text{supp}(x)|-1)}. \tag{42}$$

By Young's convolution inequality (Theorem 3), set $r = \infty, p = \infty, q = 1$, we have

$$\|f_\phi * f_\phi\|_\infty \leq \|f_\phi\|_\infty \|f_\phi\|_1 = \|f_\phi\|_\infty = \phi, \tag{43}$$

for $\|f_\phi\|_1 = \int_{-\infty}^{\infty} |f_\phi(t)| \, \mathrm{d}t = 1$. Thus, the density parameter for α has the upper bound

$$\phi_\alpha = \|f_{\phi_\alpha}\|_\infty = \frac{1}{2}\|f^{*(|\mathrm{supp}\, x|-1)}\|_\infty \leq \frac{1}{2}\phi. \tag{44}$$

Following the same idea, let $\beta = \alpha + t_{ii} + t_i$, the upper bound estimation for the density parameter of β is

$$\phi_\beta = \|f_{\phi_\alpha} * f_\phi * f_\phi\|_\infty \leq \|f_{\phi_\alpha}\|_\infty = \frac{1}{2}\phi, \tag{45}$$

so

$$\Pr\left[\Delta_i < \epsilon \mid \textit{Config}\right] = \Pr\left[|\beta| < \epsilon \mid \textit{Config}\right] \leq 2\epsilon \cdot \frac{1}{2}\phi = \epsilon\phi. \tag{46}$$

Because the configuration of other coefficients is independent, we have

$$\Pr\left[\Delta_i < \epsilon\right] = \Pr\left[\Delta_i < \epsilon \mid \textit{Config}\right] \leq \epsilon\phi. \tag{47}$$

Combining Equation (33) and (47), we finally have

$$\Pr\left[\Delta_{win} < \epsilon\right] \leq \sum_{i=1}^{n} \Pr\left[\Delta_i < \epsilon\right] \leq \epsilon\phi n. \tag{48}$$

In conclusion, we characterise the *winner gap* by investigating the property of *i-th gap*, then we estimate its upper bound in probability. Such an upper bound implies that if we set ϵ correctly, the *winner gap* is large enough with high probability. □

3.3 Rounding Procedure and Optimality

To build a connection from smoothed complexity to worst-case complexity, we need to elaborate on how to use an algorithm for adversarial problems to solve the corresponding smoothed problem. As all the coefficients in smoothed problems are stochastic real values, we can round down them to b bits after the binary point (suppose we use binary code), and the rounding error of each coefficient will be no larger than 2^{-b}. According to Lemma 3, if the rounding error is not too large, we can still compute the optimal solution for the smoothed problem by solving the rounded one, as the winner gap is large enough to separate the optimal solution from other solutions.

Lemma 4 *For a smoothed problem with the perturbation model f_ϕ, with a probability of at least $1 - (n+1)n^2\phi/2^b$, the optimal solution of the b-bits rounded problem is also the optimal solution of the original smoothed problem.*

Proof. After revealing b bits, the rounding error would be

$$\forall i, j, k, |Q_{ij} - \lfloor Q_{ij} \rfloor| \leq 2^{-b}, |b_k - \lfloor b_k \rfloor| \leq 2^{-b}, \tag{49}$$

where $\lfloor Q_{ij} \rfloor$ and $\lfloor b_k \rfloor$ are the rounded coefficients. For any pair of different solutions $\boldsymbol{x}$ and $\boldsymbol{x}'$, their objective gap is

$$\Delta_{obj} = \left| \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} - \left(\boldsymbol{x}'^T \boldsymbol{Q} \boldsymbol{x}' + \boldsymbol{b}^T \boldsymbol{x}' \right) \right|. \tag{50}$$

This gap may change due to the rounding procedure, so we have

$$\begin{aligned}\delta =& \left|\left|(\boldsymbol{x}-\boldsymbol{x}')^T \boldsymbol{Q} (\boldsymbol{x}-\boldsymbol{x}') + \boldsymbol{b}^T (\boldsymbol{x}-\boldsymbol{x}')\right| - \left|(\boldsymbol{x}-\boldsymbol{x}')^T \lfloor \boldsymbol{Q} \rfloor (\boldsymbol{x}-\boldsymbol{x}') + \lfloor \boldsymbol{b} \rfloor^T (\boldsymbol{x}-\boldsymbol{x}')\right|\right| \\ \leq& \left|(\boldsymbol{x}-\boldsymbol{x}')^T (\boldsymbol{Q} - \lfloor \boldsymbol{Q} \rfloor) (\boldsymbol{x}-\boldsymbol{x}') + (\boldsymbol{b} - \lfloor \boldsymbol{b} \rfloor)^T (\boldsymbol{x}-\boldsymbol{x}')\right| \\ \leq& \left|(\boldsymbol{x}-\boldsymbol{x}')^T (\boldsymbol{Q} - \lfloor \boldsymbol{Q} \rfloor) (\boldsymbol{x}-\boldsymbol{x}')\right| + \left|(\boldsymbol{b} - \lfloor \boldsymbol{b} \rfloor)^T (\boldsymbol{x}-\boldsymbol{x}')\right| \\ \leq& n^2 2^{-b} + n 2^{-b} \\ =& (n+n^2) 2^{-b}.\end{aligned} \tag{51}$$

According to Lemma 3, if we set $\epsilon = (n+n^2)2^{-b}$, we have

$$\Pr\left[\Delta_{win} < \left(n+n^2\right) 2^{-b}\right] \leq \left(n+n^2\right) 2^{-b} \cdot \phi n = \frac{(n+1)n^2\phi}{2^b} \tag{52}$$

which is the probability that such b-bit rounding will affect the optimal solution. Therefore we state that with a probability of at least $1 - \frac{(n+1)n^2\phi}{2^b}$, the winner gap is large enough so that the change of objective values will not influence the optimality of the best solution. In other words, the optimal solution of the rounded problem is still the optimal solution of the original smoothed problem. □

3.4 Main Theorem

In this section, we state our main theorem and prove it in detail.

Theorem 4 *Consider any 0-1 quadratic programming problem with the form*

$$\max\left\{\boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x}\right\} \text{ s.t. } \boldsymbol{x} \in \mathcal{S}.$$

If we consider the perturbation on the objective function, then we have the following two results.

1. *If such a problem has polynomial smoothed complexity, then it has a randomised algorithm with pseudo-polynomial expected running time.*
2. *If there is a pseudo-polynomial algorithm for it, then we can also construct an algorithm with polynomial smoothed complexity.*

Theorem 4 also applies to minimisation problems, but we will focus on the maximisation problem in the following proofs without loss of generality. For clarity, we will decompose Theorem 4 into two lemmas (Lemma 5 and 6) and prove them respectively in the next two subsections. Once we finish the proof of Lemma 5 and Lemma 6, the main theorem is a direct consequence of them.

3.4.1 From Smoothed Polynomial to Expected Pseudo-polynomial

Lemma 5 *For any 0-1 quadratic programming problem, if we have a polynomial smoothed complexity algorithm A_s, we can construct a randomised Las Vegas algorithm A_r with pseudo-polynomial expected running time.*

Algorithm 1 Pseudo-polynomial running time randomised algorithm A_r

Require: Polynomial smoothed complexity algorithm A_s

For the input adversarial problem, perturbed the coefficients by *i.i.d.* random variables $t \sim f_\phi$ with $\phi = 6n^2(n+1)^2EM$
Check if the perturbation is *sufficient proper* in polynomial time
if The perturbation is *sufficient proper* **then**
    Run A_s on the perturbed problem
    if A_s halt within poly(n, ϕ) time **then**
        return the optimal solution
    else
        return *"fail"*
    end if
else
    return *"fail"*
end if

Proof. Suppose we have an algorithm A_s with polynomial smoothed complexity, this algorithm cannot be directly applied to the unperturbed problem since the coefficients are not stochastic. However, we can construct an algorithm A_r that actively perturbs the problem and call A_s as a subroutine to solve it (see Algorithm 1).

To prove that the Algorithm 1's running time is pseudo-polynomial, we need to define what is a *proper* perturbation first. Since all the coefficients in adversarial problems have finite length, without the loss of generality, we can scale them up to integers and let the largest absolute value of coefficients be

$$M = \max_{\forall i,j,k} \left\{ Q_{ij}^{(ad)}, b_k^{(ad)} \right\}. \tag{53}$$

Then we divide all the coefficients by M, downscaling them to the interval $[-1, 1]$. Based on the fact in Equation (37), the lower bound of the objective gap is

$$\Delta_{obj} = \left| \boldsymbol{x}^T \boldsymbol{Q}^{(ad)} \boldsymbol{x} + \boldsymbol{b}^{(ad)^T} \boldsymbol{x} - \left(\boldsymbol{x}'^T \boldsymbol{Q}^{(ad)} \boldsymbol{x}' + \boldsymbol{b}^{(ad)^T} \boldsymbol{x}' \right) \right| \geq \frac{1}{M}, \tag{54}$$

because the minimum absolute value of coefficients is $1/M$. If the perturbation on the objective gap is not larger than $1/2M$, the order of solutions will not be changed, and we can call the smoothed algorithm A_s to compute the correct optimal solution. Therefore, we have such a definition of *proper* perturbation:

Definition 15 (*Proper* perturbation) If the perturbation gap

$$\delta' = \left| \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} - \left(\boldsymbol{x}^T \boldsymbol{Q}^{(ad)} \boldsymbol{x} + \boldsymbol{b}^{(ad)^T} \boldsymbol{x} \right) \right| \leq \frac{1}{2M}, \tag{55}$$

then we call such a perturbation *proper*.

For some distributions like uniform distribution, the value of random variables is bounded, so the perturbation will never exceed its upper bound. However, for many other distributions with infinite domain, such as Gaussian distribution, we need to find an upper bound of the probability that the perturbation is *proper*.

Let $\tilde{\boldsymbol{Q}}$ and $\tilde{\boldsymbol{b}}$ represent the stochastic part of coefficients, using Markov inequality, we have

$$\begin{aligned}
\Pr\left[\delta' \geq \frac{1}{2M}\right] =& \Pr\left[\left|\boldsymbol{x}^T\tilde{\boldsymbol{Q}}\boldsymbol{x} + \tilde{\boldsymbol{b}}^T\boldsymbol{x}\right| \geq \frac{1}{2M}\right] \\
\leq& \mathbb{E}\left[\left|\boldsymbol{x}^T\tilde{\boldsymbol{Q}}\boldsymbol{x} + \tilde{\boldsymbol{b}}^T\boldsymbol{x}\right|\right] \cdot 2M \\
=& \mathbb{E}\left[\left|\sum_{i\in\mathrm{supp}(\boldsymbol{x})}\sum_{j\in\mathrm{supp}(\boldsymbol{x})}\tilde{Q}_{ij} + \sum_{k\in\mathrm{supp}(\boldsymbol{x})}\tilde{b}_k\right|\right] \cdot 2M \\
\leq& \mathbb{E}\left[n^2\left|t\right| + n\left|t\right|\right] \cdot 2M \\
=& \left(n^2\mathbb{E}\left[|t|\right] + n\mathbb{E}\left[|t|\right]\right) 2M \\
=& \frac{2n(n+1)EM}{\phi},
\end{aligned} \tag{56}$$

in which $t \sim f_\phi$. If we set $\phi = 6n(n+1)EM$, then the probability that f_ϕ is *proper* is

$$\Pr\left[\delta' < \frac{1}{2M}\right] \geq 1 - \frac{1}{3} = \frac{2}{3}. \tag{57}$$

However, in the next step of Algorithm 1 we need to check whether the current perturbation is *proper* or not. If we check the values of all possible δ', then we have to consider not only the perturbation random variables but also all the feasible solutions. Therefore, we adopt a simpler approach to check it by introducing a more restricted definition of *proper*.

Definition 16 (*Sufficient proper* perturbation) For all perturbation random variables $\tilde{Q}_{ij}$ and $\tilde{b}_k$, if

$$\forall i,j,k, \left|\tilde{Q}_{ij}\right| < \frac{1}{2n(n+1)M}, \text{and } \left|\tilde{b}_k\right| < \frac{1}{2n(n+1)M}, \tag{58}$$

we call the perturbation *sufficient proper*.

It is easy to observe that *sufficient proper* is a sufficient condition to the previous definition of *proper*. Then, the probability that a perturbation is not *sufficient proper* is

$$\begin{aligned}
&\Pr\left[\bigcup_{ij}\left|\tilde{Q}_{ij}\right| \geq \frac{1}{2n(n+1)M} \vee \bigcup_{k}\left|\tilde{b}_k\right| \geq \frac{1}{2n(n+1)M}\right] \\
\leq& \sum_{i,j}\Pr\left[\left|\tilde{Q}_{ij}\right| \geq \frac{1}{2n(n+1)M}\right] + \sum_{k}\Pr\left[\left|\tilde{b}_k\right| \geq \frac{1}{2n(n+1)M}\right] \\
\leq& \sum_{i,j}\mathbb{E}\left[t\right]2n(n+1)M + \sum_{k}\mathbb{E}\left[t\right]2n(n+1)M \\
=& \frac{2n^2(n+1)^2ME}{\phi}
\end{aligned} \tag{59}$$

by Markov inequality and Union bound, where $t \sim f_\phi$. Recall that in Algorithm 1 we set $\phi = 6n^2(n+1)^2ME$, thus

$$\Pr\left[\textit{sufficient proper}\right] \geq 1 - \frac{1}{3} = \frac{2}{3}, \tag{60}$$

which is the probability that A_r will call A_s. Although A_s can always return the correct solution, its running time will still exceed $\text{poly}(n, \phi)$ with an arbitrarily small probability ϵ, and formally we have

$$\exists \epsilon \in (0,1), \Pr\left[T(A_s) \geq \text{poly}(n, \phi, \frac{1}{\epsilon})\right] \leq \epsilon. \tag{61}$$

Thus, if we set ϵ properly, the probability that Algorithm A_r can return the correct solution is

$$\Pr\left[\textit{sufficient proper} \wedge T(A_s) \leq \text{poly}(n, \phi, \frac{1}{\epsilon})\right] \geq \frac{2}{3} + (1-\epsilon) - 1 = \frac{2}{3} - \epsilon > \frac{1}{2}, \tag{62}$$

and A_r is a Las Vegas algorithm. In addition, according to the definition of *sufficient proper*, we just need to check the absolute values of coefficients, so the time complexity of checking *sufficient proper* is only $O(n^2)$.

Now, we can prove that the running time of A_r is pseudo-polynomial. The first two steps in Algorithm 1 take $O(n^2)$ time because the numbers of coefficients that need perturbation and checking are both $n^2 + n$. Its total running time will be $\text{poly}(n)$ if the perturbation is not *sufficient proper*, while it will be $\text{poly}(n, \phi)$ if the perturbation is *sufficient proper*. As a result, the total running time is pseudo-polynomial since we have $\phi = \text{poly}(n, M)$.

Because Las Vegas algorithms can be characterised in two equivalent ways, we can convert A_r into another algorithm A'_r that always outputs the correct optimal solutions.

Algorithm 2 Pseudo-polynomial expected running time randomised algorithm A'_r

Require: Pseudo-polynomial randomised algorithm A_r

```
Run A_r
while A_r outputs "fail" do
    Run A_r
end while
Return the optimal solution
```

Based on the connection between the expected complexity and worst-case complexity of Las Vegas algorithms (Lemma 2), the expected running time of A'_r is

$$T_{exp}(A'_r) \in O(T(A_r)) = \text{poly}(n, \phi). \tag{63}$$

Therefore, we have proved that we can construct a pseudo-polynomial expected complexity algorithm from a polynomial smoothed algorithm. □

3.4.2 From Pseudo-polynomial to Smoothed Polynomial

Lemma 6 *For any 0-1 quadratic programming problem, if we have a deterministic pseudo-polynomial complexity algorithm A_d, we can construct a polynomial smoothed complexity algorithm A_s.*

Proof. Suppose we have a pseudo-polynomial deterministic algorithm A_d that can deal with integer coefficients (finite length), we need to use A_d as a subroutine to solve a smoothed problem, whose coefficients are stochastic values. Using the adaptive rounding procedure, we reveal the first b bits of coefficients after the binary point. Then, we upscale the coefficients by 2^b and run A_d to solve the rounded problem. If the rounding error does not affect the optimality of the best solution, A_d can output the correct optimal solution for the original smoothed 0-1 quadratic programming problem.

Algorithm 3 Polynomial smoothed complexity algorithm A_s

Require: Pseudo-polynomial algorithm A_d

1: Round down all the coefficients to $b = \log\left(\frac{2(n+1)n^2\phi}{\epsilon}\right)$ bits after the binary point
2: Run A_d on the rounded 0-1 quadratic programming
3: Check the optimality of A_d's output
4: **while** A_d's output is not optimal **do**
5: Reveal one more bit
6: Run A_d on the rounded 0-1 quadratic programming
7: Check the optimality of A_d's output
8: **end while**
9: Return the output of A_d

Listed in Algorithm 3, A_s will round the coefficients adaptively and run A_d repeatedly until the optimality of the output is guaranteed. To prove that A_s has polynomial smoothed complexity, we need to define how to check the optimality first.

After rounding to b bits, A_d returns the optimal solution $\boldsymbol{x}'$ for the rounded problem such that

$$\boldsymbol{x}' = \arg\max_{\boldsymbol{x}} \left\{ \boldsymbol{x}^T \lfloor \boldsymbol{Q} \rfloor \boldsymbol{x} + \lfloor \boldsymbol{b} \rfloor^T \boldsymbol{x} \right\}. \tag{64}$$

To check the optimality of $\boldsymbol{x}'$, we define $\bar{\boldsymbol{Q}}$ and $\bar{\boldsymbol{b}}$ as

$$\bar{Q}_{ij} = \begin{cases} \lfloor Q_{ij} \rfloor, \text{if } x'_i = 1 \wedge x'_j = 1 \\ \lceil Q_{ij} \rceil = \lfloor Q_{ij} \rfloor + 2^{-b}, \text{if } x'_i = 0 \vee x'_j = 0 \end{cases}, \bar{b}_k = \begin{cases} \lfloor b_k \rfloor, \text{if } x'_i = 1 \\ \lceil b_k \rceil = \lfloor b_k \rfloor + 2^{-b}, \text{if } x'_i = 0 \end{cases}. \tag{65}$$

The corresponding optimal solution is

$$\boldsymbol{x}'' = \arg\max_{\boldsymbol{x}} \left\{ \boldsymbol{x}^T \bar{\boldsymbol{Q}} \boldsymbol{x} + \bar{\boldsymbol{b}}^T \boldsymbol{x} \right\}. \tag{66}$$

Because for all $\bar{Q}_{ij}$ and $\bar{b}_k$ that $x'_i = x'_j = x'_k = 0$, we have $\bar{Q}_{ij} \geq Q_{ij}$ and $\bar{b}_k \geq b_k$. Thus,

$$\boldsymbol{x}' = \arg\max_{\boldsymbol{x}} \delta''(\boldsymbol{x}) = \arg\max_{\boldsymbol{x}} \left\{ \left(\boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} \right) - \left(\boldsymbol{x}^T \bar{\boldsymbol{Q}} \boldsymbol{x} + \bar{\boldsymbol{b}}^T \boldsymbol{x} \right) \right\}. \tag{67}$$

If $\boldsymbol{x}' = \boldsymbol{x}''$, then $\boldsymbol{x}'$ both maximise $\boldsymbol{x}^T \bar{\boldsymbol{Q}} \boldsymbol{x} + \bar{\boldsymbol{b}}^T \boldsymbol{x}$ and $\left(\boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} \right) - \left(\boldsymbol{x}^T \bar{\boldsymbol{Q}} \boldsymbol{x} + \bar{\boldsymbol{b}}^T \boldsymbol{x} \right)$, so we can guarantee the optimality of $\boldsymbol{x}'$ that

$$\boldsymbol{x}' = \arg\max_{\boldsymbol{x}} \left\{ \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{x} \right\}. \tag{68}$$

Since we need to run A_d again to compute $\boldsymbol{x}''$, certifying optimality needs pseudo-polynomial time.

Now we can analyse the running time of A_s. We have already known that the running time of A_d and checking optimality is $2 \cdot \text{poly}(n, M) = \text{poly}(n, M)$, where M is the largest absolute value of integer coefficients. We can decompose M into

$$M = M_1 M_2, \tag{69}$$

in which M_1 is the largest absolute value of coefficients before scaling, and $M_2 \geq 2^b$ (depends on the optimality of $\boldsymbol{x}'$). Under our preliminary settings, all the coefficients before perturbation fall within the interval $[-1, 1]$, so we have

$$M_1 \leq \max_{i,j,k} \{|t_i j|, |t_k|\} + 1. \tag{70}$$

By Markov inequality, for every constant $c \geq 1$, let $t \sim f_\phi$,

$$\begin{aligned} \Pr\left[\max_{i,j,k} \{|t_{ij}|, |t_k|\} > c\right] \leq & \sum_{i,j} \Pr\left[|t_{ij}| > c\right] + \sum_k \Pr\left[|t_k| > c\right] \\ \leq & n(n+1)\frac{\mathbb{E}[|t|]}{c} \\ = & \frac{n(n+1)E}{c\phi}, \end{aligned} \tag{71}$$

so we can get

$$\Pr\left[M_1 > c + 1\right] \leq \frac{n(n+1)E}{c\phi}. \tag{72}$$

As for M_2, according to Algorithm 3, it will exceed 2^b if the $\boldsymbol{x}'$ is not optimal after the first run of A_d, so the probability that $M_2 > 2^b$ is related to Lemma 4, which is

$$\Pr\left[M_2 > 2^b\right] = \Pr\left[\Delta_{win} < \left(n + n^2\right) 2^{-b}\right] \leq \frac{(n+1)n^2\phi}{2^b}. \tag{73}$$

From Equation (72) and (73), the value of M is closely related to the value of ϕ and n, then we have

$$\Pr\left[M_1 > c + 1 \vee M_2 > 2^b\right] \leq \frac{n(n+1)E}{c\phi} + \frac{(n+1)n^2\phi}{2^b}. \tag{74}$$

Let $\forall \epsilon \in (0, 1)$, if we set

$$c = \frac{2n(n+1)E}{\epsilon\phi}, 2^b = \frac{2(n+1)n^2\phi}{\epsilon}, \tag{75}$$

we have such bound

$$\begin{aligned} \Pr\left[M > \text{poly}\left(n, \phi, \frac{1}{\epsilon}\right)\right] = & \Pr\left[M = M_1 M_2 > \left(\frac{2n(n+1)E}{\epsilon\phi} + 1\right)\frac{2(n+1)n^2\phi}{\epsilon}\right] \\ \leq & \Pr\left[M_1 > c + 1 \vee M_2 > 2^b\right] \\ \leq & \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon, \end{aligned} \tag{76}$$

which implies

$$\Pr\left[\text{poly}(n, M) > \text{poly}\left(n, \phi, \frac{1}{\epsilon}\right)\right] \leq \epsilon. \tag{77}$$

Consider a "lucky" senario where A_d is only called twice by A_s (for computing $\boldsymbol{x}'$ and $\boldsymbol{x}''$), the running time will be

$$O(n^2) + 2 \cdot \text{poly}(n, M), \tag{78}$$

as rounding procedure needs $O(n^2)$ time. However, if $M_1 > c + 1$, the running time will still exceed the polynomial of n. Therefore, we can bound the running time of A_s as

$$\begin{aligned}
&\Pr\left[T(A_s) > \text{poly}\left(n, \phi, \frac{1}{\epsilon}\right)\right] \\
=&\Pr\left[T(A_s) > O(n^2) + 2 \cdot \text{poly}(n, M) \vee \text{poly}(n, M) > \text{poly}\left(n, \phi, \frac{1}{\epsilon}\right)\right] \\
\leq&\Pr\left[M_2 > 2^b \vee M_1 > c + 1 \vee M_2 > 2^b\right] \\
=&\Pr\left[M_1 > c + 1 \vee M_2 > 2^b\right] \leq \epsilon.
\end{aligned} \tag{79}$$

As a result, we have proved that using the adaptive rounding procedure, the smoothed complexity of the algorithm A_s we constructed is polynomial. □

3.5 Zero Coefficients

One problem of our smoothed analysis in 0-1 quadratic programming is that the perturbation will destroy the "zero structure". If the adversary assigns some coefficients to be zero, then the perturbation will turn such zero values into small non-zero values. This issue is discussed in [7] and [1], and our analysis is also robust to this problem.

If we need to maintain the "zero structure", we can let the environment avoid perturbing all the zero coefficients allocated by the adversary. Formally we have

$$\exists i, j, k, \text{s.t. } Q_{ij} = Q_{ij}^{(ad)} = 0, b_k = b_k^{(ad)} = 0, \tag{80}$$

and we can define a set of indices of zero coefficients as

$$\mathcal{Z} = \{i \mid (\exists j, Q_{ij} = 0 \vee Q_{ji} = 0) \vee (b_i = 0)\}. \tag{81}$$

Recall that we only distinguish the solutions with different objective values, if there are two solutions $\boldsymbol{x}$ and $\boldsymbol{x}'$ such that

$$x_i \neq x_i' \to i \in \mathcal{Z}, \tag{82}$$

then they are equivalent. Therefore, the issue of "zero structure preserving" will not affect our results for *winner gap*, *objective gap* and *perturbation gap* in Section 3.2 and 3.4.1.

4 Discussion

Now we have known that for any 0-1 quadratic programming problem, if such a problem is weakly NP-hard/NP-complete (pseudo-polynomial), then it has polynomial smoothed complexity. My analysis does not focus on a specific method but on the general structure of 0-1 quadratic programming, so it can be generalised to a wide range of noise distribution. In this section, first, we discuss three applications of the main theorem, and then we discuss a possible direction for future research: the smoothed analysis of ℓ_0-minimisation.

4.1 Applications of Our Results

4.1.1 0-1 Quadratic Knapsack Problem

Quadratic knapsack problem [23] is a well-studied classic 0-1 quadratic programming problem, which has numerous applications ranging from scheduling problems [24] to Very Large Scale Integration [25]. Although quadratic knapsack problems are strongly NP-hard [26], there is a pseudo-polynomial dynamic programming heuristic algorithm [27]. Therefore we claim that:

Corollary 1 *Quadratic knapsack problems have no polynomial smoothed complexity, but the dynamic programming heuristic algorithm can halt within polynomial time with high probability in practice.*

4.1.2 Max-cut Problem

Boros and Hammer [28] show the equivalence between maximum cut problems and unconstrained 0-1 quadratic programming problems. Because maximum cut problems are strongly NP-hard [29], combined with Theorem 4, we have such a negative result:

Corollary 2 *Maximum cut problems have no polynomial smoothed complexity.*

4.1.3 Partition Problem

Another important problem that can be reformulated to 0-1 quadratic programming is the set partitioning problem (2-partition) [30]. Because it has pseudo-polynomial time algorithms [18], it also has polynomial smoothed complexity.

Corollary 3 *Set partition problems have polynomial smoothed complexity.*

4.2 Towards the Smoothed Analysis of ℓ_0-Minimisation

At the end of this paper, we briefly discuss a possible approach for analysing the smoothed complexity of ℓ_0-minimisation problems, which is left for future research.

In machine learning and compressed sensing, learning with sparsity is always one of the most important problems [31]. For linear regression with sparsity, one way to define the ℓ_0-minimisation problem is:

Definition 17 (ℓ_0-minimisation) For a given integer K, a matrix $\boldsymbol{A}$, and a vector $\boldsymbol{b}$, find the best $\boldsymbol{x}$ that

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \left\{ \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2 \right\}, \text{s.t. } \|\boldsymbol{x}\|_0 \leq K. \tag{83}$$

Since ℓ_0-minimisation problems are highly non-convex and strongly NP-hard [32], two alternative approaches, Ridge regression [33] and LASSO [34], are widely applied in industries. However, it is still unknown if there exists an efficient algorithm for ℓ_0-minimisation problems in practice from the theoretical perspective.

To analyse the smoothed complexity of a ℓ_0-minisation problem, we can reformulate it to a mixed 0-1 quadratic programming problem.

Definition 18 For a given integer K, a matrix $\boldsymbol{A}$, and a vector $\boldsymbol{b}$, find the best $\boldsymbol{x}$ and $\boldsymbol{z} \in \{0,1\}^n$ that

$$\min_{\boldsymbol{x},\boldsymbol{z}} \left\{ (\boldsymbol{A}\text{diag}(\boldsymbol{z})\boldsymbol{x} - \boldsymbol{b})^T (\boldsymbol{A}\text{diag}(\boldsymbol{z})\boldsymbol{x} - \boldsymbol{b}) \right\}, \text{s.t. } \sum_{i=1}^{n} z_i \leq K, \tag{84}$$

where n is the dimension and $\text{diag}(\boldsymbol{z})$ is the diagonal matrix of 0-1 vector $\boldsymbol{z}$.

Under such reformulation, the 0-1 vector $\boldsymbol{z}$ is for feature selecting only. For any fixed $\boldsymbol{z}$ the optimal real-valued vector $\boldsymbol{x}$ can be obtained by gradient descent algorithms. Based on the nature of ℓ_0-minisation, we only need to consider the perturbation on $\boldsymbol{A}$ and $\boldsymbol{b}$, and the possible challenges of the analysis are twofold:

- As a mixed 0-1 quadratic programming, we need to deal with both $\boldsymbol{x}$ and $\boldsymbol{z}$, and we cannot directly apply ordinary least squares under high-dimensional settings.
- Let the stochastic part of $\boldsymbol{A}$ and $\boldsymbol{b}$ be $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{b}}$ respectively, we will need to estimate the upper bound for $\tilde{\boldsymbol{A}}^T\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{b}}^T\tilde{\boldsymbol{b}}$. Thus it requires the upper bound estimation for the variance of perturbation random variables.

In a word, I believe that this paper offers a cornerstone for the further understanding of the hardness of ℓ_0-minimisation under the smoothed setting.

References

[1] Rene Beier and Berthold Vöcking. Typical properties of winners and losers in discrete optimization. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 343–352, New York, NY, USA, 2004. Association for Computing Machinery.

[2] Heiko Röglin and Berthold Vöcking. Smoothed analysis of integer programming. In *Proceedings of the 11th International Conference on Integer Programming and Combinatorial Optimization*, IPCO '05, pages 276–290, Berlin, Heidelberg, 2005. Springer.

[3] Andrej Bogdanov and Luca Trevisan. Average-case complexity. *Foundations and Trends® in Theoretical Computer Science*, 2(1):1–106, 2006.

[4] Leonid A. Levin. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286, 1986.

[5] Shai Ben-David, Benny Chor, Oded Goldreich, and Michel Luby. On the theory of average case complexity. *Journal of Computer and System Sciences*, 44(2):193–219, 1992.

[6] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis: An attempt to explain the behavior of algorithms in practice. *Communications of the ACM*, 52(10):76–84, 2009.

[7] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

[8] Markus Bläser and Bodo Manthey. Smoothed complexity theory. *ACM Transactions on Computation Theory*, 7(2):1–21, 2015.

[9] Avrim Blum and John Dunagan. Smoothed analysis of the perceptron algorithm for linear programming. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 905–914, USA, 2002. Society for Industrial and Applied Mathematics.

[10] John Dunagan, Daniel A. Spielman, and Shang-Hua Teng. Smoothed analysis of condition numbers and complexity implications for linear programming. *Mathematical Programming*, 126(2):315–350, 2011.

[11] David Arthur and Sergei Vassilvitskii. Worst-case and smoothed analysis of the ICP algorithm, with an application to the k-means method. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 153–164, USA, 2006. IEEE.

[12] David Arthur, Bodo Manthey, and Heiko Röglin. Smoothed analysis of the k-means method. *Journal of the ACM*, 58(5):1–31, 2011.

[13] Adam Tauman Kalai, Alex Samorodnitsky, and Shang-Hua Teng. Learning and smoothed analysis. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 395–404, USA, 2009. IEEE.

[14] Heiko Röglin and Shang-Hua Teng. Smoothed analysis of multiobjective optimization. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 681–690, USA, 2009. IEEE.

[15] Jonathan A. Kelner and Evdokia Nikolova. On the hardness and smoothed complexity of quasi-concave minimization. In *48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 472–482, USA, 2007. IEEE.

[16] Juraj Hromkovič. *Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms*. Springer, Berlin, Heidelberg, first edition, 2005.

[17] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, 2009.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Company, USA, 1979.

[19] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity*. Wiley series in discrete mathematics and optimization. John Wiley & Sons, Hoboken, New Jersey, second edition, 2014.

[20] Geoffrey R. Grimmett and David R. Stirzaker. *Probability and random processes*. Oxford University Press, New York, NY, USA, third edition, 2001.

[21] Gerald B. Folland. *Real Analysis: Modern Techniques and Their Applications*. John Wiley & Sons, USA, 1999.

[22] William Beckner. Inequalities in Fourier analysis. *Annals of Mathematics*, 102(1):159–182, 1975.

[23] Giorgio Gallo, Peter L Hammer, and Bruno Simeone. Quadratic knapsack problems. In *Combinatorial Optimization*, pages 132–149. Springer, Berlin, Heidelberg, 1980.

[24] Bahram Alidaee, Gary A. Kochenberger, and Ahmad Ahmadian. 0-1 quadratic programming approach for optimum solutions of two scheduling problems. *International Journal of Systems Science*, 25(2):401–408, 1994.

[25] Carlos E. Ferreira, Alexander Martin, C. Carvalho de Souza, Robert Weismantel, and Laurence A. Wolsey. Formulations and valid inequalities for the node capacitated graph partitioning problem. *Mathematical Programming*, 74(3):247–266, 1996.

[26] Alberto Caprara, David Pisinger, and Paolo Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11(2):125–137, 1999.

[27] Franklin Djeumou Fomeni and Adam N. Letchford. A dynamic programming heuristic for the quadratic knapsack problem. *INFORMS Journal on Computing*, 26(1):173–182, 2013.

[28] Endre Boros and Peter L. Hammer. The max-cut problem and quadratic 0-1 optimization; polyhedral aspects, relaxations and bounds. *Annals of Operations Research*, 33(3):151–180, 1991.

[29] Giorgio Ausiello, Alberto Marchetti-Spaccamela, Pierluigi Crescenzi, Giorgio Gambosi, Marco Protasi, and Viggo Kann. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, Berlin, Heidelberg, 1999.

[30] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, Boston, MA, 1972.

[31] Simon Foucart and Holger Rauhut. *A Mathematical Introduction to Compressive Sensing*. Birkhäuser, New York, NY, USA, 2013.

[32] Dongdong Ge, Xiaoye Jiang, and Yinyu Ye. A note on the complexity of L_p minimization. *Mathematical Programming*, 129(2):285–299, 2011.

[33] Arthur E. Hoerl and Robert W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 42(1):80–86, 2000.

[34] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

Appendix A Python Code for Example 2

A.1 Bubble Sort Algorithm

```
from itertools import permutations
import time
import matplotlib.pyplot as plt

perm = permutations(range(0,6))
runTimeList = []

# Bubble sort
def bubbleSort(arr):
  length = len(arr)
  swapped = False
  for i in range(length-1):
    for j in range(length-1):
      if arr[j] > arr[j+1]:
        # swap
        arr[j], arr[j+1] = arr[j+1], arr[j]
        swapped = True
    if not swapped:
      return arr

# Record the running time of bubbleSort
for arr in perm:
  start = time.process_time()
  arr = list(arr)
  arrSorted = bubbleSort(arr)
  end = time.process_time()
  duration = end - start
  runTimeList.append(duration)
```

A.2 Drawing Figure 1

```
caseNum = len(runTimeList)
caseList = range(caseNum)
worstCaseTime = max(runTimeList)
averageCaseTime = sum(runTimeList) / len(runTimeList)

plt.figure(figsize=(8,5), dpi=300)
plt.plot(runTimeList, color='tab:blue', label='running time')
plt.axhline(y=worstCaseTime, color='tab:red', label='worst-case running
    time')
plt.axhline(y=averageCaseTime, color='tab:pink', label='average-case
    running time')
plt.xlabel("instances")
plt.ylabel("running time")
plt.legend(loc='best')

plt.show()
```

A.3 Drawing Figure 2

```
smoothedTimeList = []

for i in range(caseNum):
  smoothedTime = (runTimeList[i]
  + runTimeList[(i-1) % caseNum]
  + runTimeList[(i-2) % caseNum]
  + runTimeList[(i+1) % caseNum]
  + runTimeList[(i+2) % caseNum]) / 5
  smoothedTimeList.append(smoothedTime)

smoothedComplexity = max(smoothedTimeList)

plt.figure(figsize=(8,5), dpi=300)
plt.plot(runTimeList, color='lightblue', label='running time')
plt.plot(smoothedTimeList, color='tab:orange', label='smoothed running time
    ')
plt.axhline(y=smoothedComplexity, color='tab:cyan', label='smoothed
    complexity measure')
plt.xlabel("instances")
plt.ylabel("running time")
plt.legend(loc='best')

plt.show()
```