A Study of Variants of PPA-complete Problems

Charalampos Kokkalis
Jesus College
University of Oxford

A thesis submitted for the degree of
MCompSci Computer Science

Trinity Term 2022

Abstract

In contrast to the well-known concept of decision problems in computational complexity theory, search problems require algorithms to return a certificate of a solution instead of simply verifying its existence. In 1991, Megiddo and Papadimitriou introduced a subclass of NP Search Problems, that of Total NP Search Problems (TFNP), a solution to which is guaranteed to exist due to some non-constructive, graph-theoretic argument. In this project, we focus on a well-studied subclass of TFNP, the class PPA (which stands for Polynomial Parity Argument), whose problems are guaranteed to have a solution due to the handshaking lemma. We consider extensions of two of the most important problems in PPA, the problems LEAF and LONELY. Using a variety of reduction ideas from previous work in the field, we establish new membership and hardness results for these extensions in the subclasses of TFNP. Additionally, we provide a list of open problems, which could potentially be solved by extending the work of this project.

Acknowledgments

I would like to deeply thank my supervisor, Professor Paul Goldberg, for introducing me to the problems of this project, providing me with valuable ideas and sources to study, and meticulously examining my proofs and early drafts of the report.

I would also like to express my gratitude towards all of my tutors and professors who inspired me throughout my years at the university, motivating me to produce my best work, and making these four years a very enjoyable learning experience.

Last but not least, I want to thank my family and friends for their unfaltering support and encouragement, always being there for me in difficult moments.

Contents

Chapter 1

Introduction

The first problems studied in computational complexity were decision problems, where given an instance of the problem, the algorithm aims to return either “yes” or “no”. The first identified NP-complete problem was that of deciding whether a propositional formula has a satisfying assignment, making it evaluate to “true”. This is the well-known SAT problem. A natural extension to this problem would be to investigate the complexity of actually finding this assignment (if existent), rather than just verifying that there is one. This gives rise to the class of search problems. The equivalent class of search problems whose solutions are verifiable in polynomial-time is FNP.

In many cases, search problems can be efficiently converted into decision problems. For example, the search equivalent of the SAT problem, namely FSAT, which given a propositional formula looks to either return a satisfying assignment or the string “no” if the formula is unsatisfiable, is in a sense not harder than SAT, since we could use a polynomial number of calls to a SAT oracle to solve the FSAT instance. However, there are some search problems where the corresponding decision version is not computationally equivalent to the original one. A particular such case, which we investigate in this project, is when a solution is guaranteed to exist due to some mathematical argument, however it remains computationally challenging to compute it. Such problems are called *total search problems*, and the corresponding class of total search problems whose solutions are verifiable in polynomial-time is TFNP (standing for Total Function Non-deterministic Polynomial-time), introduced by Megiddo and Papadimitriou in [MP91].

As with any interesting complexity class, one of the first emerging questions is the existence of complete problems for that class. For reasons we will discuss in Chapter 2, it seems unlikely that there are TFNP-complete problems. Nonetheless, there has been a lot of research in identifying subclasses of TFNP and looking for complete problems in those. These subclasses are generally defined by the nature of the argument that guarantees the existence of a solution to the search problem. In this paper we will focus on results for one of these subclasses, the PPA class (standing for Polynomial Parity Argument). We will study extensions of two PPA-complete problems, the LEAF problem (which defines the class) and the LONELY problem. We aim to classify the complexity of these problems, and achieve completeness results where possible.

1.1 Our Contributions

Besides discussion on the theory behind total search problems, our work provides the following original contributions to the field:

- We show membership of the one-dimensional MAX-DISTANCE-d LONELY problem in the class FP.
- We show existence of an adversary oracle for the NO-OVERLAPPING-EDGES LONELY problem.
- We prove that, first 2D LEAF, but also k-D LEAF for any positive integer k are PPA-complete problems.
- We describe a relaxation of the 2D LONELY problem and prove it is PPA-complete.

Furthermore, we extend this relaxation to k dimensions, establishing another PPA-completeness result.

1.2 Report Outline

- Chapter 2: Definition of TFNP, investigation of its division in subclasses, the notion of non-constructive proofs.
- Chapter 3: Definition of PPA, key problems LEAF and LONELY.
- Chapter 4: Explore 1-dimensional variants of LONELY problem, namely MAX-DISTANCE-d LONELY and NO-OVERLAPPING-EDGES LONELY.
- Chapter 5: Define higher-dimensional variants of LEAF, prove PPA-completeness.
- Chapter 6: Define higher-dimensional variants of LONELY (including a relaxed version), provide proof for PPA-completeness of RELAXED k-D LONELY.
- Chapter 7: Summary and exploration of open questions.

Chapter 2

Search Problems and the Class TFNP

Computational problems in which, rather than a simple "yes" or "no" answer, we are interested in returning a solution (when there is one) are called search problems. In the remaining of this chapter, we will formalize some specific classes of search problems and their properties.

2.1 Definitions

In the case where the solutions can be checked in polynomial-time, we end up with what are called *NP Search Problems* and the class FNP.

Definition 1 (NP Search Problems). *An NP search problem is specified as a relation $R(\cdot, \cdot)$ such that:*

- *Given an input x, we are looking for a y (called the* certificate*) such that $R(x, y)$.*
- *$R(x, y)$ is checkable in time polynomial in $|x|$ and $|y|$.*

Similarly, the FP class consists by problems for which there exist polynomial-time algorithms that on input x return a y such that $R(x, y)$.

An example of a problem in FNP is the search equivalent of the well-known Boolean satisfiability problem (SAT), called FSAT (for Function Satisfiability), where, given a Boolean formula, we ask for either a satisfying assignment to its variables or the string "no" in case it is unsatisfiable. FSAT is FNP-complete. In the case of SAT, we get a polynomial-time equivalent search problem, which we can easily see by describing an algorithm that, given an oracle for SAT, solves FSAT in polynomial-time. This algorithm begins by assigning a random truth value to the first literal, calling the SAT oracle, and proceeding with that value if the oracle returns "yes" to the new instance or flipping it if it returns no. The algorithm proceeds to take a number of steps linear on the number of literals in the formula before solving FSAT.

However, there are cases where the decision problem could be trivial compared to the search problem. This often happens in problems where existence of a solution is guaranteed due to some mathematical argument. Search problems with guaranteed existence of a solution are called total search problems.

Definition 2 (NP Total Search Problems)**.** *An NP search problem is total if for all inputs x there is a y of length $|y| = poly(|x|)$ such that $R(x, y)$.*

The class of total search (or function computation) problems in NP is called TFNP, and was first introduced by Megiddo and Papadimitriou in [MP91]. Finally, it is useful to introduce the notion of a polynomial-time reduction for search problems, similarly to that for decision problems, which we will use throughout the paper:

Definition 3 (Polynomial-time many-one reductions for search problems)**.** *Let $R, S \in FNP$. Then, R many-one reduces to S if there exist polynomial-time computable functions f, g such that*

$$(f(x), y) \in S \Rightarrow (x, g(x, y)) \in R.$$

We denote this as $R \leq_m S$. To keep notation simple, we abbreviate this to $R \leq S$ in this project.

From the definitions of these classes, it is obvious that FP $\subseteq$ TFNP $\subseteq$ FNP. However, there is still no proof of either a proper inclusion or an equality in either side. Any solution to these problems would have tremendous effects in complexity theory. If TFNP = FP, that would imply that, among other things, P = NP $\cap$ coNP. On the other hand, an existence of an FNP-complete problem in TFNP would imply that NP = coNP [MP91]. Therefore, there has been a lot of interest in the study of this intermediate class, TFNP. A natural first step would be to look for TFNP-complete problems. However, this seems unlikely to happen; TFNP is a *semantic class*, as it is defined in terms of a property over all inputs (the property is that the Turing machine has at least one conclusive computation on all inputs), and semantic classes seem to have no complete problems.

2.2 TFNP subclasses

Consequently, the immediate next step would be to split TFNP further into *syntactically definable* subclasses which we can study and find complete problems for. Before introducing these classes, it is important to define the notion of a non-constructive proof:

Definition 4 (Non-constructive proof)**.** *A proof is non-constructive if it uses some mathematical argument to show the existence of a solution, without providing an efficient way to compute it.*

The subclasses of TFNP, first introduced in [Pap94], classify some of the problems according to the non-constructive argument that guarantees the existence of a solution to them. In [GP17], the class PTFNP (for "provable TFNP") is defined, which contains the union of these well studied subclasses of TFNP. We proceed by briefly mentioning the different subclasses of PTFNP together with the corresponding non-constructive argument defining each one of them.

- PLS: Problems where existence of a solution is guaranteed by the fact that "every directed acyclic graph has a sink".
- PPP: Problems where existence of a solution is guaranteed by the fact that "given

a function mapping from a finite set to a smaller one, there must exist a collision" (Pigeonhole Principle).

- PPA: Problems where existence of a solution is guaranteed by the fact that "every graph with an odd-degree node must have another odd-degree node".
- PPAD: Problems where existence of a solution is guaranteed by the fact that "every directed graph with an unbalanced node must have another unbalanced node".
- PPADS: Same as PPAD but looking for oppositely unbalanced nodes.
- CLS (first defined in [DP11]): Optimization problems over a continuous domain in which a continuous potential function is being minimized and the optimization algorithm has access to a continuous improvement function (guaranteed existence of a solution to the optimization problem is due to the continuity of the aforementioned functions). Recent advances in [FGHS21] showed that actually CLS = PPAD $\cap$ PLS.

It remains unknown whether PTFNP=TFNP, since there are TFNP problems that have not been classified as members of any of the subclasses in PTFNP. A well-known example of this is the FACTORING problem (the problem of factoring integers). [1]

Following multiple research papers establishing the relationships and inclusions between these complexity classes, the current state can be summarized in Figure 2.1, where a directed edge from class A to class B indicates that $A \subseteq B$.

In this project, we will focus on the class PPA, investigating problems that arise from extending certain known PPA-complete problems. Before we proceed to exploring the PPA class, it is important to define the two different models that we will use throughout this report to describe TFNP problems.

[1]Recent progress on this has showed that FACTORING can be reduced to both PPP-complete and PPA-complete problems using randomized reductions, which could be derandomized assuming the Generalized Riemann hypothesis.[BO06] [Jer12] Therefore, any breakthrough in proving the GRH would have significant impact on the characterization of the relationships between the subclasses of TFNP.

$$\begin{tikzcd}
 & \text{TFNP} & \\
 & \text{PTFNP} \arrow[u] & \\
\text{PLS} \arrow[ur] & \text{PPA} \arrow[u] & \text{PPP} \arrow[ul] \\
 & & \text{PPADS} \arrow[u] \\
 & & \text{PPAD} \arrow[u] \arrow[uul] \\
 & \text{CLS} \arrow[ur] \arrow[uuul] & \\
 & \text{FP} \arrow[u] &
\end{tikzcd}$$

Figure 2.1: The inclusion relations among classes in PTFNP and TFNP

2.3 Circuit Model vs Oracle Model

As with many computational problems, it is common to also define TFNP problems in terms of circuits. By convention, binary strings are considered to be *type-0* objects. Then, simple functions, which take binary strings as inputs are naturally called *type-1* or *"unrelativized"* objects. We can view the circuits defining the TFNP problems as such type-1 objects. In [BCE$^+$98], the authors reformulated the TFNP class and its subclasses in terms of *type-2* (also called *"relativized"*) problems, where the input consists of not only strings, but also functions and relations (type-1 objects), which are presented as oracles. This is what we call the *oracle model* (or also known as the *black-box model*), as it uses oracle access to the function/circuit that specifies the naturally induced graph of each problem.

This reformulation of the problems allows for further separation of the subclasses of PTFNP in the oracle model. Additionally, we can now resort to adversarial arguments to prove the hardness of problems. This is because in the black-box model the adversary could be arbitrarily sophisticated, causing any algorithm to take exponentially many steps to find a solution. This was not the case for the circuit model, since it would not be possible to encode such a complex strategy for an adversary using a circuit of bounded size. We will establish a result using this technique of designing an adversary in the oracle model in Section 4.2.

Chapter 3

The Class PPA

In this project, we will focus on specific problems in the class PPA and their extensions. First introduced by Papadimitriou [Pap94], the PPA class (standing for Polynomial Parity Argument) includes TFNP problems for which the non-constructive argument that guarantees the existence of a solution is the *handshaking lemma*:

Lemma 1 (The Handshaking Lemma)**.** *Any undirected graph that has an odd-degree vertex must have at least one other vertex whose degree is also odd.*

Before defining the class PPA and giving the natural complete problem for it, we will show how to formalize the connection between the more abstract idea of a graph and the underlying Turing machine specifying the structure of the problem. Let M a polynomial-time deterministic Turing machine with alphabet $\Sigma = \{0, 1\}$ for a problem A, and x an input of the problem.

Definition 5 (Configuration Space)**.** *The configuration space* $C(x)$ *is* $\Sigma^{p(|x|)}$*, the set of all strings of length at most* $p(|x|)$ *for some polynomial* p*.*

Given a configuration $c \in C(x)$, M outputs a set $M(x, c)$ of configurations in time $O(p(n))$. We can now get the corresponding graph as follows:

Definition 6 (Configuration Graph)**.** *The configuration graph $G(x)$ has all configurations in the configuration space $c \in C(x)$ as vertices. Additionally, it has an edge from c to c' if $c \in M(x, c')$ and $c' \in M(x, c)$.*

By the definition of the graph $G(x)$, we can see that it is symmetric. We are now ready to define the natural PPA-complete problem, which defines the class.

3.1 The LEAF Problem

The arguments to the LEAF problem (C, x) describe a graph G on $2^{|x|}$ nodes, where each node has maximum degree 2 and the circuit C succinctly encodes the edges of the graph, such that for some vertex u in the graph, $C(u)$ is a set of 0, 1, or 2 nodes adjacent to u. A *leaf* is a node of degree one. We are given that the node $0...0 = 0^n$, where $n = |x|$, is a leaf, called the *standard leaf* in G. Using the definition of M above, and restricting the size of the set of configurations returned by M to be at most 2, this means that given an input x to the problem A, we have $M(x, 0...0) = \{1...1\}$ and $0...0 \in M(x, 1...1)$. We now have the tools to define the search problem LEAF as follows:

Definition 7 (The LEAF problem)**.** *Given the input x, find a leaf of the graph $G(x)$ other than the standard one* (0...0)*.*

Then, we can define the class PPA as:

Definition 8 (The class PPA)**.** *PPA is the class that contains the LEAF problem and all other problems reducible to it.*

Finally, another useful notion for the remaining of this paper is PPA-completeness. Similarly to how completeness is defined for other classes (like NP), we have:

Definition 9 (PPA-completeness)**.** *A problem A is PPA-complete if it belongs in PPA and any other problem $A' \in PPA$ reduces to it.*

3.2 The LONELY problem

Another computational problem of interest is the *LONELY* problem. Similarly to LEAF, the input to the LONELY problem (C, x) describes a graph G on $2^{|x|}$ nodes, where each node has maximum degree 1 and the circuit C succinctly encodes the edges of the graph, such that for some vertex u in the graph, $C(u)$ is a set of 0 or 1 nodes adjacent to u. Therefore, the Turing machine M, given a configuration $c \in C(x)$ will output $M(x, c)$ which will be either an empty set or a single configuration. A *lonely* node is a node of zero degree. We are given that the node $0...0 = 0^n$ is a *standard lonely* node in G, meaning that $M(x, 0...0) = \{\}$. Two nodes u, v (represented as bitstrings of length n, where $n = |x|$) are considered to be neighbors of each other if and only if:

- $u \neq v$
- $C(u) = v$
- $C(v) = u$

Then, the LONELY problem is defined as:

Definition 10 (The LONELY problem). *Given the input x, find a lonely node of the graph $G(x)$ other than the standard one* (0...0).

The problem is known to be *PPA-complete*, as it is many-one equivalent to LEAF (we can reduce it to LEAF, but also we can reduce from LEAF to it). The details of these reductions are given in [BCE+98].

Before proceeding to investigate related problems, it is practical to introduce some simplifications in the language we use for defining these problems and reducing from one to another, following the discussion of [GKSZ19].

Remark 1 (Simplifications in description of problems and reductions). *We will use the following conventions (interchangeably with the default ones) when describing our search problems and the various reductions:*

1. *We will be describing algorithms instead of circuits in many cases to encode the graphs. It is known that it is feasible to simulate a polynomial-sized circuit by a polynomial-time algorithm.*

2. *In contrast to how we defined LEAF and LONELY, we will allow our graphs to have any number of nodes (rather than just powers of* 2 *encoded as bitstrings* $\{0,1\}^n$*). This can be achieved in all cases where the nodes we use can be efficiently indexed. It is important to ensure that we are not introducing any new solutions to the problem in this process; this is also not an issue, as for example in the LEAF problem we can add nodes with no neighbors, and in the LONELY problem we can keep adding the nodes in pairs, connected to each other.*

3. *We will also often define the edges of the graph directly, instead of specifying how to compute the neighbors of a vertex. It will be easy to see how a small number of queries to a corresponding circuit used as an oracle could compute these locally.*

Finally, it is useful to mention another well-known type of reductions, the Turing reductions:

Definition 11 (Turing reductions for search problems)**.** *Let* $R, S \in FNP$*. Then, a Turing reduction from* R *to* S *is an oracle machine that solves problem* R *given an oracle for problem* S*. We denote this as* $R \leq_T S$*.*

An *oracle machine* can be thought of as a Turing machine that has access to a black-box (the *oracle*) which is able to solve the corresponding problem in a single operation. For example, in the definition of Turing reductions above, the oracle used by the oracle machine can answer queries about instances of the S problem efficiently, using a single operation. In [BJ11], the authors proved that the TFNP subclasses PLS, PPAD, PPADS, and PPA are all closed under Turing reductions, which allows us to use both types of reductions (Turing and many-one) in our proofs as equivalent.

In the next chapter we will investigate the first kind of extensions to the PPA problems we mentioned, namely the one-dimensional extensions to the LONELY problem.

Chapter 4

Variants of the LONELY Problem in One Dimension

Although this is not explicitly part of the definition, we could consider the graphs corresponding to the problems in the previous section to be aligning nodes in a straight line (the line of positive integers). To achieve this, we can consider the bitstring that corresponds to the description of each node and convert this to an integer (as we would normally convert a binary number to a decimal one). Therefore, our nodes can be imagined as placed on a straight line from 0 to $2^n - 1$. This way, we can naturally define the length of an edge between two nodes, by calculating their distance on the line. The original version of the LONELY problem allows for edges of arbitrary length. Knowing that LONELY is PPA-complete, it is natural to consider other extensions to it, investigating whether allowing for more solutions (or in a sense adding more "structure" to the problem) could make it easier. In this chapter, we will discuss two variants of the LONELY problem, where we introduce some restriction on the properties of these edges.

4.1 The MAX-DISTANCE-d LONELY Problem

The first example of such an extension would be to restrict the allowed length of the edges. As the instance contains 2^n nodes, these are succinctly encoded using bitstrings of length n. Therefore, we could define the distance between two nodes to be the absolute value of the difference of their corresponding binary numbers. We can restrict the problem to allow edges of length at most d, for some constant d.

Definition 12 (The MAX-DISTANCE-d LONELY problem)**.** *Given an input instance x (just like in the LONELY problem) and an integer d, find either a lonely node of the graph $G(x)$, or return two nodes u, v such that they are neighbors and their distance (as defined above) is greater than d.*

In the remaining of this section we prove the following result about this problem:

Theorem 1. *The MAX-DISTANCE-d LONELY problem is in FP.*

To prove membership in FP, we will demonstrate an algorithm that solves the search problem in polynomial time. Our algorithm is similar to a binary search, making a number of d queries to figure out which side of the nodes (if we imagine positioning them in a straight line lexicographically ordered according to the bitstring representing them) has a solution. We provide pseudocode for this in Algorithm 1.

The reason this “shortcut” to a solution works is that by querying d nodes in a row, we are guaranteed that nodes in each side could only be neighbors with nodes on the same side, because of the constraint on the distance d. The algorithm is guaranteed to terminate since at each step the interval we are considering is half the size of the previous one. As far as the running time is concerned, we require a total of $\log 2^n$ steps (for the binary search), where at each step we make at most d queries to the oracle, which results to an algorithm of time complexity $O(d \log 2^n) = O(dn)$, which is polynomial in n if d is also polynomial in n. Thus, the search problem is solvable in a polynomial number of calls to the oracle and consequently belongs to the FP complexity class.

Algorithm 1 Algorithm for MAX-DISTANCE-d LONELY

```
1: interval $\leftarrow [0...0, 1...1]$
2: repeat
3:     $l, r \leftarrow 0$
4:     $m \leftarrow$ middle point of current interval
5:     checking_interval $\leftarrow [m, m + d - 1]$
6:     for $i$ from 0 to $d$ do
7:         Query $n = C(m + i)$
8:         if $n = \{\}$ then
9:             return "$m + i$ is a lonely node"
10:        else if dist$(m + i, m) > d$ then
11:            return "$m + i$ and $n$ are neighbors of distance more than $d$"
12:        else if $n$ is to the left of the checking_interval then
13:            $l \leftarrow l + 1$
14:        else if $n$ is to the right of the checking_interval then
15:            $r \leftarrow r + 1$
16:    if the number of unmatched nodes to the left of the checking_interval is odd then
17:        interval $\leftarrow$ [previous interval's left end,$m$]
18:    else
19:        interval $\leftarrow [m + d$, previous interval's right end]
20: until one of the if statements returns                ▷ Guaranteed to be the case
```

4.2 The NO-OVERLAPPING-EDGES LONELY Problem

Another extension to the LONELY problem could be forcing the edges connecting neighbors to be non-overlapping. We can easily see that this belongs to the PPA class (by simply reducing to the standard LONELY problem, mapping each instance of NO-OVERLAPPING-LONELY to itself), so it remains to check the hardness of the problem. We will now demonstrate an adversary that would force any algorithm to resort to an exhaustive search, which indicates that the problem is unlikely to be in FP.

Theorem 2. *There exists an adversary oracle causing any algorithm to exhaustively query all nodes to find a solution to the NO-OVERLAPPING-EDGES LONELY problem.*

For the rest of this section, we describe this adversary oracle. The goal of the adversary is to respond to all of the algorithm's queries to the circuit, creating an instance of the problem that is hard to solve. A simple adversary to the NO-OVERLAPPING-EDGES LONELY problem could begin by keeping track of intervals in $[0, 2^n - 1]$ that contain only unmatched nodes. Initially, all nodes are unmatched, and the adversary will update this accordingly in each step. Then, when the algorithm queries for the neighbor of node u, the adversary will return v, which is the first unmatched node to the right of u (if we think of nodes as being aligned in a straight line depending on their index). If there are no unmatched nodes to the right of u, then we start considering the nodes from the beginning of the line. After finding v, we update the intervals of unmatched nodes accordingly, removing u and v. This adversary is guaranteed to keep giving new neighbors until there is only one node (the solution) available, so it remains to show that the edges (connection between neighbors) it gives are not overlapping, meaning that there is no x that has already been assigned such that:

- if $u < v$ then $u < x < v < C[x]$
- if $u > v$ (in the case where there were no unmatched nodes left to the right of u) then $v < x < u < C[x]$ or $x < v < C[x] < u$

To see why this is not the case, notice that in all bad scenarios we have that for some $z \in \{u, v\}$, $x < z < C[x]$ (also using the fact that $C[C[x]] = [x]$). This however would be impossible to be chosen by our adversary, as at each step the neighbor returned is the closest node to the right of the queried node, meaning that $C[x]$ would have been z, resulting in a contradiction. Therefore, the edges returned by the adversary are not overlapping, and thus the adversary achieves forcing the algorithm to exhaustively check all (exponentially many) nodes to find the LONELY node. Thus, working in the oracle model, the corresponding type-2 problem to NO-OVERLAPPING-EDGES LONELY is not in FP. This indicates that the original, type-1 problem is indeed unlikely to be in FP.

Chapter 5

Extension of the LEAF Problem to Higher Dimensions

5.1 The 2D LEAF Problem

We can think of the 2D LEAF problem as an extension of the LEAF problem in two dimensions, with the restriction of allowing nodes to only connect with their immediate neighbors. Instead of 2^n points, this time we have a $2^n \times 2^n$ grid of nodes, where the only possible edges are those of length one (connecting nodes up, down, left, or right). Therefore, the corresponding graph $G(x)$ is now two-dimensional. The nodes of the graph are given by the set $\{(i,j)|i,j \in [0, 2^n - 1]\}$ and the edges are a subset of $\{((i,j),(i+1,j))|i \in [0, 2^n - 2], j \in [0, 2^n - 1]\} \cup \{((i,j),(i,j+1))|i \in [0, 2^n - 1], j \in [0, 2^n - 2]\}$. Given that the standard leaf is at $(0^n, 0^n)$, the search problem 2D LEAF looks for another leaf node in $G(x)$:

Definition 13 (The 2D LEAF problem). *Given the input x, find a leaf of the corresponding graph $G(x)$ other than the standard one* $(0^n, 0^n)$.

To begin with, we can place this new problem in the hierarchy of TFNP subclasses by showing its membership in the class PPA:

Theorem 3. *The 2D LEAF Problem is in PPA.*

Proof. To see why this is the case, we will reduce 2D LEAF to the LEAF problem. The reduction is trivial, mapping all the points from the 2D grid to a straight line and maintaining the same connectivity between vertices. It is easy to see that the given solution (at $(0^n, 0^n)$ in the 2D LEAF instance) corresponds to the given solution 0^n in the LEAF instance, while leaf node in the 2D instance is also mapped to a leaf node in the 1D instance. This completes our polynomial-time reduction 2D LEAF≤LEAF, and therefore, since LEAF∈PPA, it is also the case that 2D LEAF∈PPA. □

5.2 Reducing LEAF to 2D LEAF

We will now establish the first hardness result on this project. In the rest of this section, we present the proof for the following:

Theorem 4. *The 2D LEAF Problem is PPA-complete.*

We will map an instance $(C_1, |x|)$ of the LEAF problem to an instance $(C_2, |x'|)$ of the 2D LEAF problem. Here, $|x'| = 4x^4$, as we are working with a $2|x|^2 \times 2|x|^2$ grid. For each node i (where i is the bitstring corresponding to its position in the LEAF instance), we introduce an edge from $(0, i(2|x|))$ to $(0, i(2|x|) + |x| - 1)$ representing it in the 2D LEAF instance. By an edge between these points, we imply that we connect every node in the straight line between the two endpoints to both of its horizontal neighbors. By definition of the problem, each node has $0, 1$, or 2 neighbors. We will now describe the remaining edges of the 2D LEAF instance.

For each edge from node u to node v in the LEAF instance, we will create a path in the 2D LEAF instance, which will move vertically some distance on the grid starting from an endpoint of the corresponding edge of each node, and then horizontally connecting the two vertical lines (we informally call that last part a "bridge"). The height of the vertical lines used to connect u and v will be $y(u, v) = 2(u + |x|v)$. This guarantees that no two bridges can overlap horizontally. For the case where a horizontal section may need to cross a vertical section of a different bridge, we will construct a crossover gadget, which will make the reduction valid. Since each node can have two neighbors, we need to specify which of the two endpoints of the edge corresponding to the node in the 2D version will be connected to which neighbor. We choose to connect the leftmost node with the first neighbor in lexicographic order. We need at most 2 extra calls to C_1 to further decide which one of the neighbor's nodes will be the other endpoint of this path. In case a node has only one neighbor, we choose to always connect the second endpoint of the edge corresponding to it in the 2D version for consistency. An example demonstrating this construction can be seen in Figure 5.2, where the 2D instance is shown right below the 1D instance, with all the edges in both cases drawn in blue. We have drawn a circle with a dotted line around each edge corresponding to a node in the 1D instance, and have placed each node of the 1D instance right above the corresponding edge (and thus dotted circle) in the 2D instance. As we can see, the solutions to the 2D case (located at the red points, the squared ones being the standard ones) directly correspond to the solutions of the 1D case.

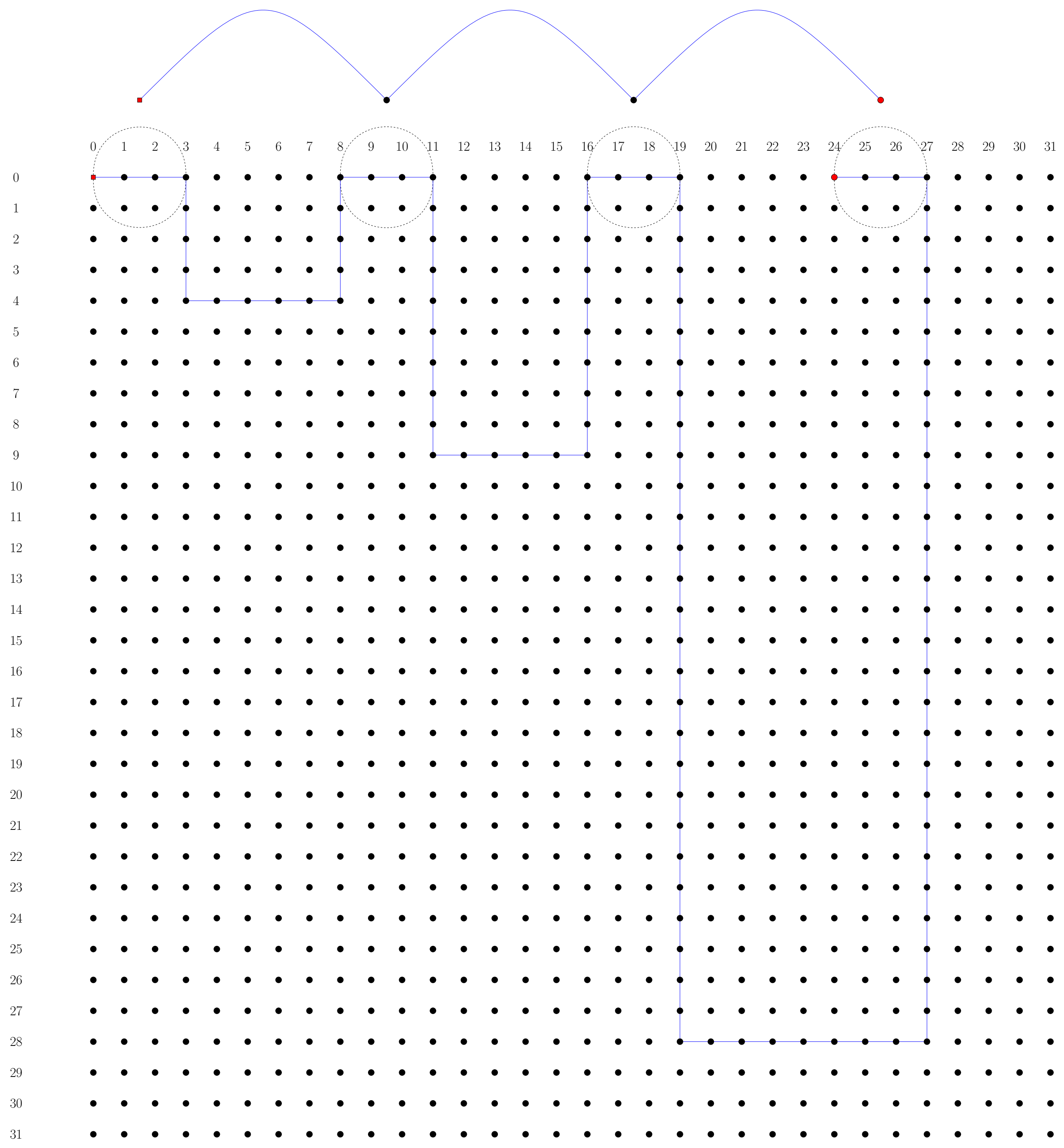


Figure 5.1: Demonstrating the bridges in a simple case. 1D instance above, corresponding 2D instance below.

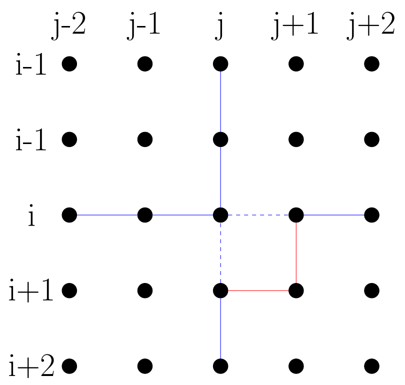


Figure 5.2: The Crossover Gadget

We will now examine what happens in the more difficult case of our reduction, which occurs when a vertical line of one bridge might cross the horizontal line of another bridge (we can identify these cases in polynomial time, since the horizontal segments efficiently encode the values of the corresponding vertices that they connect in the 1D version). This must be fixed, since we require all nodes in the 2D LEAF problem to have at most 2 neighbors. We achieve this by introducing a crossover gadget. Similar gadgets have been introduced in the past for other 2D problems, for example in proving that the 2D Brouwer fixed-point problem is PPAD-complete [CD09].

The reason we multiplied the function returning the heights of the bridges by two was to avoid having the aforementioned points of degree 4 in consecutive lines. We can now describe the crossover gadget as shown in Figure 5.2; the blue dotted lines correspond to edges that are removed, whereas the red lines are the new edges introduced. The red lines are introduced to the side "away from the vertex", meaning that in the example of the crossover gadget we assumed that $(0, j)$ is connected to $(0, j - 1)$ (if it was connected to $(0, j + 1)$ instead, the red lines would have been symmetrical to the current ones but to the left of the vertical line). To illustrate the functionality of the crossover gadget, consider the example in Figure 5.2, containing three different crossover gadgets.

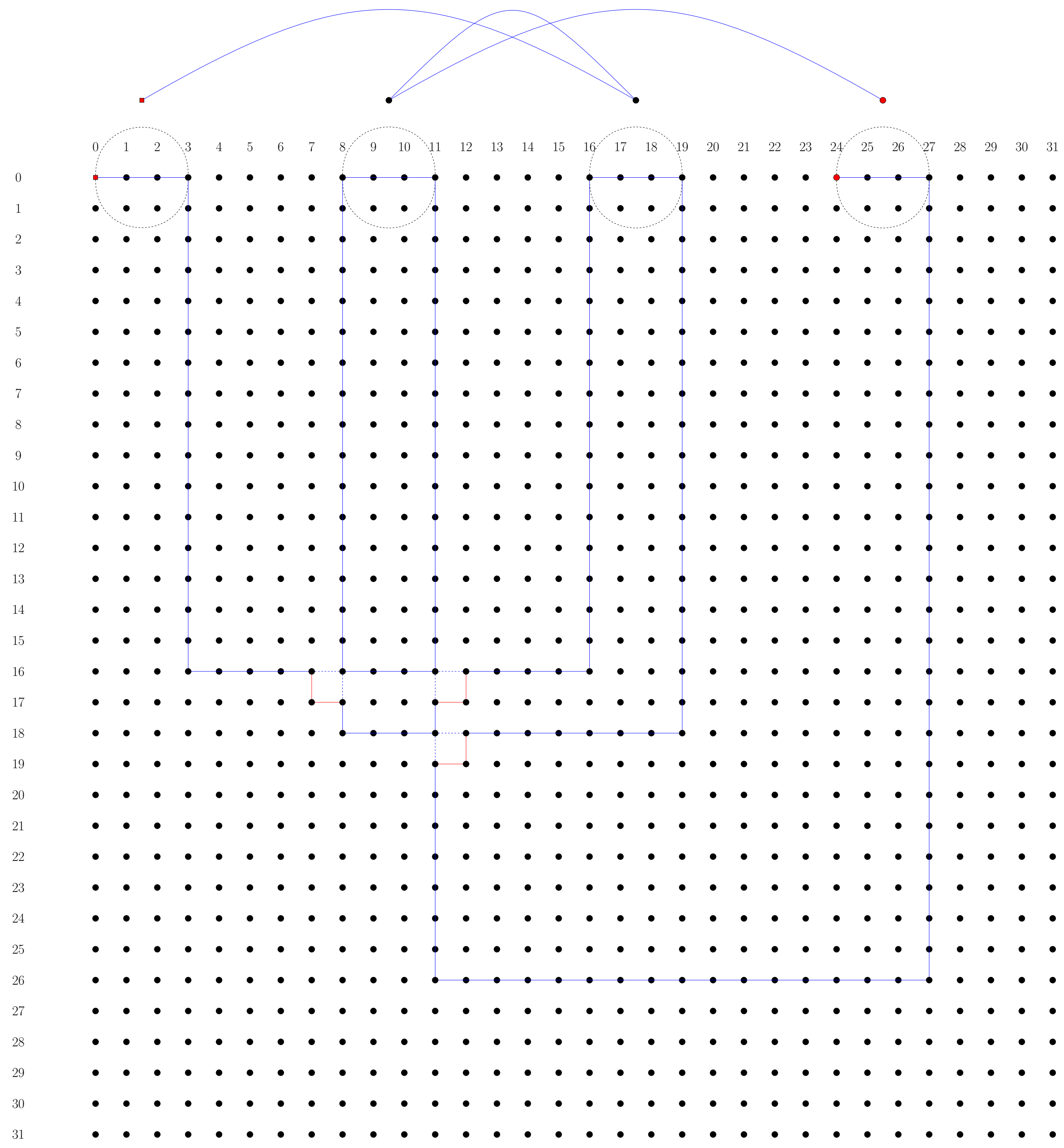


Figure 5.3: Demonstrating the bridges in the case where crossover gadgets are required. 1D instance on top, corresponding 2D instance below

This concludes our polynomial-time reduction LEAF$\leq$2D LEAF. Since LEAF is PPA-complete, this implies that 2D LEAF is also PPA-hard (we can reduce any PPA problem to LEAF and from there to 2D LEAF). This, together with membership in PPA (which we proved in Theorem 3), implies that 2D LEAF is PPA-complete.

5.3 The k-D LEAF Problem

We can now also examine a further extension of the LEAF problem, namely considering the k-dimensional version of it. Essentially this remains a case of the LEAF problem, where each node has degree either 0, 1, or 2 and given a node of degree 1 (the given leaf) we are looking for another leaf node. The only restriction is that nodes can only be connected along one dimension, meaning that all edges will have length exactly 1. Therefore, the definition of vertices and the set of possible edges of the graph is just the obvious extension to that of the 2D case.

Definition 14 (The k-D LEAF Problem)**.** *Given the input x, find a leaf of the corresponding graph $G(x)$ other than the standard one (represented by the k-tuple $(0^n, ..., 0^n)$).*

As before, we can immediately classify this problem in PPA:

Theorem 5. *The k-D LEAF Problem is in PPA.*

Proof. To show this, we will once again reduce to the LEAF problem. The reduction is essentially identical to Theorem 3, mapping all nodes to a single line and keeping the same edges. Reducing k-D LEAF to LEAF suffices to prove that k-D LEAF$\in$PPA. □

5.4 Reducing 2D LEAF to k-D LEAF

Aiming for another PPA-completeness result, we use the remaining of this section to prove the following theorem:

Theorem 6. *The k-D LEAF Problem is PPA-complete.*

We will demonstrate a reduction from the 2-dimensional version of LEAF to the k-dimensional version, as that would imply PPA-hardness of k-D LEAF (since we already showed that 2D LEAF is PPA-hard).

Given a 2-dimensional instance L of the LEAF problem, we will construct a corresponding k-dimensional one L' by simply mapping all the points of L to the 2D plane defined by the first two axis of the k-dimensional space, matching nodes identically to how they were matched in L, and then leaving all nodes in the remaining of the k-D space unmatched. This way, the given leaf in L is mapped to the given leaf in L', and the leaves of L' are in 1-1 correspondence with the leaves in L. Therefore, we conclude that 2D LEAF $\leq$ k-D LEAF. Since we have the chain of reductions LEAF$\leq$2D LEAF$\leq$ k-D LEAF$\leq$LEAF, and we know that LEAF is PPA-complete, we deduce that all these problems are PPA-complete.

Chapter 6

Extension of the LONELY Problem to Higher Dimensions

6.1 The 2D LONELY Problem

We can think of the 2D LONELY problem as an extension of the LONELY problem in two dimensions, with the restriction of allowing nodes to only connect with their immediate neighbors. Instead of 2^n points, this time we have a $2^n \times 2^n$ grid of nodes, where the only possible edges are those of length one (connecting nodes up, down, left, or right). The set of vertices as well as possible edges is therefore the same as in the 2D LEAF problem. Given the standard lonely node is at $(0^n, 0^n)$, the search problem 2D LONELY looks for another lonely node in the corresponding graph $G(x)$:

Definition 15 (The 2D LONELY Problem)**.** *Given the input x, find a lonely node of the corresponding graph $G(x)$ other than the standard one $(0^n, 0^n)$.*

6.2 A Relaxation of the 2D LONELY Problem

Additionally, as an intermediate step, we introduce another problem, a relaxation of the 2D LONELY problem, where we allow for existence of diagonal edges (still of length one). This results in the same definition of the search problem, with the only difference being in the properties of the graph $G(x)$:

Definition 16 (The RELAXED 2D LONELY Problem). *Given the input x, find a lonely node of the corresponding graph $G(x)$ other than the standard one $(0^n, 0^n)$.*

Similarly to what we did in the previous chapter, we begin by showing membership of these problems in PPA:

Theorem 7. *Both the 2D LONELY problem and its relaxation (as defined above) belong in the PPA class.*

Proof. Once again, the proof here is a simple reduction to the original LONELY problem. We proceed by mapping all nodes to a straight line and maintaining the same connections. This guarantees that the standard lonely node in the 2D instance is mapped to the standard lonely node in the 1D instance, and any other lonely nodes (solutions) of the 2D instance also correspond to lonely nodes in the described 1D instance. Therefore, we have reduced both problems to the original LONELY problem, which is in PPA, and so its 2-dimensional extensions are also in PPA. □

6.3 Reducing 2D LEAF to RELAXED 2D LONELY

We now present the second hardness result of this project, which we will also generalize later:

Theorem 8. *The RELAXED 2D LONELY problem is PPA-complete.*

The rest of this section presents the details of a reduction from 2D LEAF (which we proved to be PPA-complete in the previous section) to 2D LONELY. We will map an instance $(C_1, |x|^2)$ of the 2D LEAF problem to a $(C_2, 4|x|^2)$ instance of the RELAXED 2D

LONELY problem. For every node u in the 2D LEAF instance, we will have nodes u,u' in the corresponding instance of 2D LONELY, and if there is an edge from u to v in the 2D LEAF instance, there will be an edge from either u or u' to either v or v' in the 2D LONELY instance. The relationship between u and u' in the new graph is that if the coordinates of u are (i, j), then u' is at $i + 1, j$ (we obviously translate i and j to binary when needed to be passed as inputs to the circuit in order to succinctly encode exponentially many nodes). To describe the reduction, it suffices to demonstrate the functionality of the circuit C_2, using a polynomially-bounded number of calls to the circuit C_1.

Before we proceed, it is useful to present the definition of lexicographic ordering to 2 dimensions as follows:

$$(x_1, y_1) \preceq (x_2, y_2) \Leftrightarrow x_1 < x_2 \text{ or } (x_1 = x_2 \text{ and } y_1 \leq y_2)$$

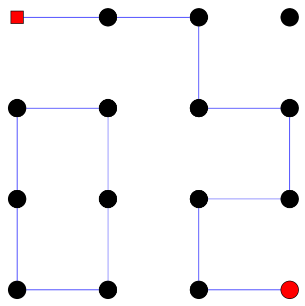
Now, for a call $C_2(v)$ or $C_2(v')$, the procedure to return the neighbors is the following:

- if the y coordinate of v is greater than $|x|$, then $C_2(v) = \{v'\}$ and $C_2(v') = \{v\}$.
- if $C_1(v) = \{\}$, then $C_2(v) = \{v'\}$ and $C_2(v') = \{v\}$.
- if $C_1(v) = \{u\}$, then:
  - if $v \prec u$: $C_2(v) = \{\}$ and $C_2(v') = \{$**either** u or $u'\}$
  - else $(v \succ u)$: $C_2(v') = \{\}$ and $C_2(v) = \{$**either** u or $u'\}$
- if $C_1(v) = \{u, w\}$: Assume WLOG that $u \prec w$. Then, v is connected to **either** u or u' and v' is connected to **either** w or w'.

In the above description, we see that it remains to show how to pick one of the two cases wherever we mentioned **either**. This requires another call to the C_1 circuit. Say we want to connect v to **either** u or u'. Then:

- if $C_1(u) = \{v\}$:
    - if $v \prec u$ then $C_2(v) = \{u\}$
    - else $(v \succ u)$: $C_2(v) = \{u'\}$
- else $(C_1(u) = \{v, w\})$:
    - if $v \prec w$ then $C_2(v) = \{u\}$
    - else $(v \succ w)$: $C_2(v) = \{u'\}$

As we see, with only a constant number of calls to the C_1 circuit, we have completely defined C_2. Additionally, a node has exactly one neighbor in 2D LEAF if and only if one of the two corresponding nodes in the relaxed 2D LONELY instance is left unmatched. Therefore, we have shown that 2D LEAF $\preceq$ relaxed-2D LONELY. In Figure 6.1, we see an example demonstrating the reduction from a 4×4 instance of 2D LEAF to an 8×8 instance of the relaxation of the 2D LONELY problem. The 4×4 grid of dotted circles in the lower graph corresponds to the nodes of the first one, where as the remaining nodes are connected in pairs to avoid additional solutions.



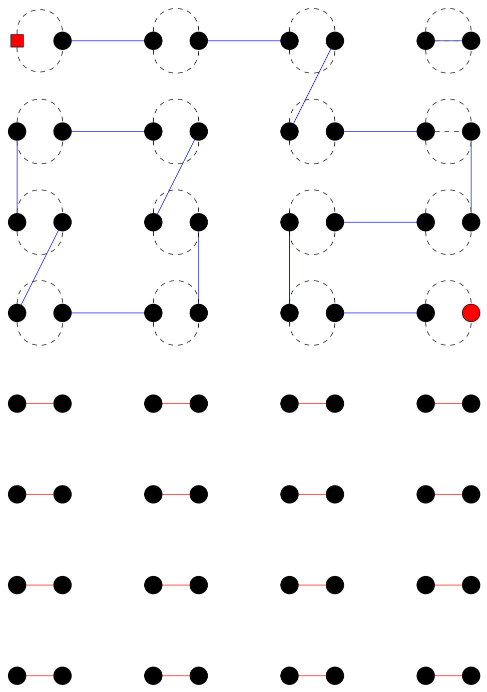


Figure 6.1: Demonstrating the reduction from 2D LEAF (top instance) to the relaxation of 2D LONELY (bottom instance).

6.4 Reducing k-D LEAF to the Relaxation of k-D LONELY

Having already showed that LEAF reduces to k-D LEAF for any k, it is worth investigating what happens as we also extend the relaxation of the LONELY problem to k dimensions. Once again, the set of vertices and possible edges is the natural extension to that of the 2D case, still allowing for edges of length at most 1:

Definition 17 (The RELAXED k-D LONELY Problem). *Given the input x, find a lonely node of the corresponding graph $G(x)$ other than the standard one (represented by the k-tuple $(0^n, ..., 0^n)$).*

Again, we can see the membership in PPA as in the 2D case:

Theorem 9. *The RELAXED k-D LONELY problem is in PPA.*

Proof. It suffices to reduce the problem to the original LONELY problem. The reduction is mapping all nodes to a single line and keeping the same edges, in a similar manner to the previous ones. Since this trivial construction maps standard lonely nodes in k-dimensions to standard ones in the one-dimensional case, and does the same to all other lonely nodes, we get that RELAXED k-D LONELY$\leq$LONELY and thus RELAXED k-D LONELY$\in$ PPA. $\square$

We now proceed to generalize the result from Theorem 8 to k dimensions:

Theorem 10. *The RELAXED k-D LONELY problem is PPA-complete.*

To show this, we provide a reduction from k-D LEAF to RELAXED k-D LONELY. The reduction follows the idea of the aforementioned 2D case, where we map one node of the LEAF instance to two nodes of the relaxed LONELY instance.

To specify the reduction, we first need to extend our definition of lexicographic ordering to k-tuples. We define this in a recursive way, the base case being the 2 dimensional one:

$$(x_1, x_2, ..., x_k) \preceq (y_1, y_2, ..., y_k) \Leftrightarrow (x_1 < y_1) \vee (x_1 = y_1 \wedge ((x_2, x_3, ..., x_k) \preceq (y_2, y_3, ..., y_k)))$$

Given two nodes u, v connected in the k-D LEAF instance G (succinctly encoded by the concatenation of k bitstrings of length n each), we will have four nodes u, u', v, v' in the RELAXED k-D LONELY instance G', and the rule to connect them will be the same as in the 2D instance, using the extension of lexicographic ordering that we just defined. Given this construction, we see that the leaf nodes of G correspond to the lonely nodes of G'. Consequently, we get that k-D LEAF$\leq$RELAXED k-D LONELY, so RELAXED k-D LONELY is also PPA-complete.

Chapter 7

Conclusion

7.1 Summary

Overall, besides stating some of the fundamental theorems and questions in the field of total search problems, this project aimed to shed light into the class PPA, by considering extensions to some of the most popular problems in the class. In some cases, such as the k-D LEAF problem and the RELAXED k-D LONELY problem (where k could be any positive integer in both cases), we discovered new PPA-complete problems. Additionally, we showed that another extension, the MAX-DISTANCE-d LONELY problem is in FP. Finally, we provided a detailed discussion on various other extensions of the problems. Our analysis to these, although inconclusive, could still be a first step towards precisely classifying them.

7.2 Open Problems/Future Work

During the exploration of the topics of these project, we came up with multiple interesting problems that would be worth researching in the future, some of which could have valuable impact in the classification of TFNP problems. In this section, we will list these open questions and provide a brief discussion on each one of them.

1. Could we establish some hardness result for the version of the LONELY problem with no overlapping edges?

   In Section 4.2 we described the functionality of an adversary that would give the oracle's responses to each query in a way that makes the search for the solution take exponentially long. This is an indicator that the problem is not likely to be in FP. However, it remains to classify it more precisely in a subclass of TFNP, which would be achieved by designing a reduction from a complete problem of a subclass to it.

2. Is the 2D LONELY problem PPA-hard (or hard for some other subclass of PPA)?

   Although we proved that a relaxed version of the 2D LONELY problem is PPA-complete, it still remains to decide what the complexity of the original version of the problem is. As an intermediate step to that, we could also look for a description of an adversary (similarly to what we did for the NO-OVERLAPPING-EDGES LONELY problem), which would suggest that the problem is not in FP.

3. More generally, does existence of an adversary oracle which causes exhaustive search guarantee some hardness result for a problem in TFNP?

   This is a more general question, a positive answer to which would allow for a new approach to classifying the complexity of total search problems, while a negative one would mean that we need more than just the description of an adversary to show that a problem is not in FP under normal assumptions.

4. Are there any other subclasses of TFNP of theoretical interest that could be related to the problems we investigated or some simple extensions of these problems?

   In [GKSZ19], the class PPA$_q$ is defined as a modulo-q analog of the PPA class that we focused on in this project. Essentially, this generalizes the non-constructive argument to state that "if a bipartite graph has a node of degree not a multiple of q, then there is another such node.". The equivalent LEAF$_q$ and LONELY$_q$ problems are defined and proved to be PPA_q-complete in the paper, therefore it would make sense to consider their extensions to higher dimensions, taking a similar approach to that of our project.

Bibliography

[BCE$^+$98] Paul Beame, Stephen Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. *Journal of Computer and System Sciences*, 57(1):3–19, 1998.

[BGIP01] Sam Buss, Dima Grigoriev, Russell Impagliazzo, and Toniann Pitassi. Linear gaps between degrees for the polynomial calculus modulo distinct primes. *Journal of Computer and System Sciences*, 62(2):267–289, 2001.

[BJ11] Samuel Buss and Alan Johnson. Propositional proofs and reductions between NP search problems. *Annals of Pure and Applied Logic*, 163, 08 2011.

[BO06] Joshua Buresh-Oppenheim. On the TFNP complexity of factoring. 12 2006.

[BOM04] J. Buresh-Oppenheim and T. Morioka. Relativized NP search problems and propositional proof systems. In *Proceedings. 19th IEEE Annual Conference on Computational Complexity, 2004.*, pages 54–67, 2004.

[BR97] Paul Beame and Søren Riis. More on the relative strength of counting principles. In *In: Proceedings of the DIMACS workshop on Feasible Arithmetic and Complexity of Proofs*, pages 13–35. American Mathematical Society, 1997.

[CD09] Xi Chen and Xiaotie Deng. On the complexity of 2D Discrete Fixed Point problem. *Theoretical Computer Science*, 410(44):4448–4456, 2009.

[DP11] Constantinos Daskalakis and Christos Papadimitriou. Continuous Local Search. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 790–804, 05 2011.

[FGHS21] John Fearnley, Paul W. Goldberg, Alexandros Hollender, and Rahul Savani. The complexity of gradient descent: CLS = PPAD $\cap$ PLS. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 46–59, New York, NY, USA, 2021. Association for Computing Machinery.

[GKRS18] Mika Göös, Pritish Kamath, Robert Robere, and Dmitry Sokolov. Adventures in Monotone Complexity and TFNP. In Avrim Blum, editor, *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*, volume 124 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[GKSZ19] Mika Göös, Pritish Kamath, Katerina Sotiraki, and Manolis Zampetakis. On the complexity of modulo-q arguments and the Chevalley-Warning theorem. 2019.

[Gol11] Paul W. Goldberg. A survey of PPAD-completeness for computing Nash Equilibria. *Surveys in Combinatorics 2011*, page 51–82, 2011.

[GP17] Paul Goldberg and Christos Papadimitriou. Towards a unified complexity theory of total functions. *Journal of Computer and System Sciences*, 94, 12 2017.

[Jer12] Emil Jerábek. Integer factoring and modular square roots. *CoRR*, abs/1207.5220, 2012.

[MP91] Nimrod Megiddo and Christos H. Papadimitriou. On total functions, existence theorems and computational complexity. *Theor. Comput. Sci.*, 81(2):317–324, apr 1991.

[Pap90] Christos H. Papadimitriou. On graph-theoretic lemmata and complexity classes. *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, 1990.

[Pap94] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994.