The Power of Free Branching in a General Model of Backtracking and Dynamic Programming Algorithms

SASHKA DAVIS
IDA/Center for Computing Sciences
Bowie, MD
sashka.davis@gmail.com

JEFF EDMONDS
Dept. of Computer Science
York University
jeff@cse.yorku.ca

RUSSELL IMPAGLIAZZO
Dept. of Computer Science
University of California, San Diego
russell@cs.ucsd.edu

ALLAN BORODIN
Dept. of Computer Science
University of Toronto
bor@cs.toronto.edu

June 27, 2020

Abstract

The *Priority Branching Tree* (pBT) model was defined in [ABBO$^+$11] to model recursive backtracking and a large sub-class of dynamic programming algorithms. We extend this model to the *Prioritized Free Branching Tree* (pFBT) model by allowing the algorithm to also branch in order to try different priority orderings without needing to commit to part of the solution. In addition to capturing a much larger class of dynamic programming algorithms, another motivation was that this class is closed under a wider variety of reductions than the original model, and so could be used to translate lower bounds from one problem to several related problems. First, we show that this free branching adds a great deal of power by giving a surprising moderately exponential (2^{n^c} for $c < 1$) upper bound in this model. This upper bound holds whenever the number of possible data items is moderately exponential, which includes cases like k-SAT with a restricted number of clauses. We also give a $2^{\Omega(\sqrt{n})}$ lower bound for 7-SAT and a strongly exponential lower bound for Subset Sum over vectors mod 2 both of which match our upper bounds for very similar problems.

1 Introduction

Algorithms are often grouped pedagogically in terms of basic paradigms such as greedy algorithms, divide-and-conquer algorithms, local search, dynamic programming, etc. There have been many moderately successful efforts over the years to formalize these paradigms ([BNR03, DI09, ABBO$^+$11, BODI11, **?**, **?**]). The models defined capture many algorithms known (intuitively) to be in the DP paradigm. These works establish lower bounds for the given models; that is, either that certain approximation ratios cannot be improved, or that exponential size algorithms are required within the model. These models also provide a formal distinction between the power of different paradigms.

The general framework for many models is that such algorithms view the input one data item at a time, and after each data item, the algorithm is forced to make irrevocable decision(s) about the solution as related to the received data item. (For example, should the data item be included or not in the solution, should the item be scheduled on machine 1 or 2, etc.). The technique for lower bounds is information-theoretic in that the algorithm does not know the rest of the input before needing to commit to part of the solution. These techniques borrowed from those dealing with online algorithms [BEY98, FW98], where an adversary

chooses the order in which the items arrive. However, the difference here is that, while inputs arrive in a certain order, the algorithm determines that order, not an adversary. A very successful formalization of greedy-like algorithms was developed in [BNR03] with the *priority model*. In this model, the algorithm is able to choose a *priority order* (greedy criteria) specifying the order in which the data items are processed. [ABBO$^+$11] extends these ideas to the *priority branching tree (pBT)* model which allows the algorithm when reading a data item to branch so that it can make, not just one, but many different irrevocable decisions about the data item. In an adaptive pBT algorithm, each such branch of the tree is allowed to specify a different order in which to read the remaining data items. The solution of the algorithm is is obtained by taking the best branch. The width of the algorithm is the maximum number of branches that are alive at any given level (i.e. time) of the tree. This relates to the space (and total time) of the algorithm. In order to keep this width down the algorithm is allowed to later terminate a branch (and back track) when this computation path does not seem to be working well. Surprisingly, the resulting model can also simulate a large fragment of dynamic programming algorithms. This model was extended further in [BODI11] to the *Prioritized Branching Program model (pBP)* as a way of capturing more of the power of dynamic programming.

One weakness of both the pBT and pBP models is that they are not closed under reductions that introduce *dummy data items*. In a typical reduction, we might translate each data item for one problem into a gadget consisting of a small set of corresponding data items for another. In addition, many reductions also have gadgets that are always present, independent of the actual input. For example, when we reduce 3-SAT to exactly-3-out-of-five SAT, we might introduce two new variables per clause, and if the clause is satisfied by $t \geq 1$ literals, we can set $3 - t$ of these new variables to true. The set of new variables and which clauses they are in depends only on the number of clauses in the original formula, not the structure of these clauses. Setting the value of one of these "dummy" variables does constrain the original satisfying assignment, but not any particular original variable. Such a data item in the extreme is what we call a *free data item*. These are data items such that the irrevocable decision that must be made about it does not actually affect the solution in any way. The reason that adding these free data items to the pBT model increases its power is because when reading them the computation is allowed to branch without needing to read a data item that matters.

Note: Which do you like priority free branching tree or prioritized free branching tree. I don't have an opinion. We extend the pBT model to the *prioritized Free Branching Tree (pFBT)* model by allowing the algorithm to also branch in order to try different priority orderings without reading a data item and hence needing to fix the part of the solution relating to this data item. In addition to closing the model under reductions, it also allows for a much wider range of natural algorithms. For example, one such branch of an algorithm for the Knapsack Problem might sort the data items according to price, another according to weight, and another according to price per weight. The algorithm will branch initially and return the best of these three greedy (i.e. priority algorithm) solutions. Obviously, free branching is simply a non-deterministic branching but we will use the word "free" to contrast it with branching only on the decisions for data items.

The contributions of this paper are both to develop a new general proof technique for proving lower bounds in this seemingly not much more powerful model and (in contrast) for finding new surprising algorithms showing that this change in fact gives the model a great deal of additional (albeit unfair) power. While the known lower bounds for 3-SAT in pBT are the tight $2^{\Omega(n)}$, for pFBT, we are only able to prove a $2^{\Omega(n^{1/2})}$ lower bound for 7-SAT. The upper bounds show that perhaps this is due more to the model's surprising strength, rather than to our inability to prove tight lower bounds. More precisely, any problem whose set $|\mathcal{D}|$ of possible data items is relatively small has an upper bound in the model of roughly exponential in $\sqrt{n}$. This covers the case of any sparse SAT formula where the number of occurrences of each variable is bounded. There is some formal evidence that this is actually the most difficult case of SAT ([IPZ01]).

The idea behind the "unreasonable algorithm" is as follows. When the algorithm reads data item D,

it learns all the data items appearing before D in the priority ordering that are *not* in the instance. Being allowed to try many orderings and abandoning those that are not useful, the algorithm is able to learn all of the data items from the domain $\mathcal{D}$ that are not in the instance and in so doing effectively learns what the input is. The algorithm is computationally unbounded (with respect to making decisions) and hence can then instantly know a valid solution. Having $m = \mathcal{O}(n \log |\mathcal{D}|)$ free data items, i.e. those whose decision does not affect the solution, brings the tree width down to $\mathcal{O}(n \log |\mathcal{D}|)^3$ even without free branching, i.e. in pBT. **NOTE: I took out the point at which m starts to help. I could also break it into multiple sentences if you like.**

Figure 1 gives a summary of those results.

Computational Problem	$\|\mathcal{D}\|$	W in pBT	W in pFBT	W with m f.d.i.
General	d	$2^{\mathcal{O}(n)}$	$2^{\mathcal{O}(\sqrt{n \log d})}$	$2^{\mathcal{O}\left(\frac{n \log d}{m+\sqrt{n \log d}}\right)}$
k-SAT $\mathcal{O}(1)$ clauses per var	$n^{\mathcal{O}(1)}$		$2^{\mathcal{O}(\sqrt{n \log n})}$	
k-SAT m' clauses	$2^{n^{k-1}}$		$2^{\mathcal{O}(k^{1/2} n^{1/3} m'^{1/3} \log n)}$	
Matrix Inversion	$n^{\mathcal{O}(1)}$	$2^{\Omega(n)}$	$2^{\Omega(\sqrt{n})}$	$2^{\Omega\left(\frac{n}{m+\sqrt{n}}\right)}$
7-SAT $\mathcal{O}(1)$ clauses per var				
Matrix Parity	$n2^n$		$2^{\Omega(n)}$	$2^{\Omega\left(\frac{n^2}{m+n}\right)}$
Subset Sum			$2^{\Omega(n/\log n)}$	
SS no carries or in pFBT$^-$			$2^{\Omega(n)}$	

Figure 1: This table summarizes the results in the paper. Here pBT and pFBT stand for the two models of computation *priority Branching Trees* and *prioritized Free Branching Tree*. The prioritizing allows the model to specify an order in which to read the data items defining the input instance. Branching allow the algorithm to make different irrevocable decisions about the data items read and to try different such orderings. The free branches allow the tree to branch without reading and hence needing to fix the part of the solution relating to a data item. Free data items, f.d.i., are data items whose irrevocable decision does not effect the solution. Here $\mathcal{D}$ is the full domain of all possible data items that potentially could be used to define the problem. The algorithm that uses free branching and/or free data items in a surprising and unfair way depends on this domain being small. 'W' is short for the width of the branching tree. This is a measure of how much space (or total time over all executed branches) the algorithm uses. The problem "SS no carries" is the Subset Sum problem with the non-standard change that the sums are done over GF-2. A slightly better lower bound can be obtained for this because the reduction to it does not need dummy/free data items. Similarly the lower bound is obtained in another the model pFBT$^-$ which restrict the use of such dummy data items.

NOTE: Yes. This is what Russel did not like too. Do you think it worth including or would you prefer it to be deleted entirely? I just don't care.

The second way of getting around the challenges of doing reductions is to restrict the model pFBT to pFBT$^-$. The input instance is allowed to specify some of its data items to be *known*. The pFBT$^-$ algorithm knows that such a data item is in the input instance even though it has not yet received it or made an irrevocable decision about it. If, despite this knowledge, an pFBT$^-$ algorithm wants to read such data items, it must do so separately from reading the unknown ones. Similarly, the input instance is allowed to pair some of the data items to be *equivalent*. The pFBT$^-$ algorithm does not know which pairs of data items will be in the instance, but does know that if one is then the other will be as well. An input reading state of a pFBT$^-$ computation tree can either read such a pair together or when one of the pair is read, the other automatically becomes a known data item as previously described. These restrictions prevent the algorithm

from inadvertently learning what unknown data items are not in the instance (because they would appear before the known data items in the priority order). Having removed their power, the $2^{\Omega(n)}$ width lower bound for the matrix parity problem goes through in the pFBT$^-$ model no matter how many known/free data items there are. The reduction then gives the $2^{\Omega(n)}$ width lower bound for Subset Sum in pFBT$^-$ model.

The paper is organized as follows. Section 2 formally defines the computational problems and the pFBT model. It also motivates the model in terms of its ability to express online, greedy, recursive backtracking, and dynamic programming algorithms. Section 3 provides the upper bounds arising from having free branching and/or free data items and small domain size. We also extend this algorithm to obtain subexponential algorithms when there are few (i.e. $m - o(n^2)$) clauses. In Section 4 we present the reductions from the hard problems 7-SAT and Subset Sum to the easy problems Matrix Inversion and Matrix Parity. In Section 5 we provide a general technique for proving lower bounds within pFBT and then use the technique to obtain the results for the matrix problems. The sections after Section 2, can be read in any order.

2 Definitions

This section defines the computational problems in Section 2.1, intuition about the model in Section 2.2, the formal definition of the pFBT model in Section 2.3, and pFBT$^-$ in Section 2.4.

2.1 The Computational Problems

Here we give a formal definition of a general computational problem and also state the k-SAT, the Matrix Inversion, the Matrix Parity Problem, and the Subset Sum problem in the priority algorithm framework:

A General Computational Problem: A *computational problem* with priority model $(\mathcal{D}, \Sigma)$ and a family of objective functions $f^n : \mathcal{D}^n \times \Sigma^n \mapsto \mathbb{R}$ is defined as follows. The input $I = \langle D_1, D_2, \ldots, D_n \rangle \in \mathcal{D}^n$ is specified by a set of n *data items* D_i chosen from the given domain $\mathcal{D}$. A solution $S = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \in \Sigma^n$ consists of a *decision* $\sigma_i \in \Sigma$ made about each data item D_i in the instance. For an optimization problem, the desired solution S maximizes $f(I, S)$. For a search problem, $f(I, S) \in \{0, 1\}$.

k-SAT: The input consists of the AND of a set of clauses, each containing at most k literals and the output is a satisfying assignment, assuming there is one. A data item contains the name x_j of the variable and the description of all clauses in which it participates.

- **Unrestricted:** The number of potential data items in the domain is $|\mathcal{D}| = n \cdot 2^{2(2n)^{k-1}}$, because there are $2(2n)^{k-1}$ clauses that the given variable x_i might be in.
- m **clauses:** Restricting the input instance to contain only m clauses, does not restrict the domain of data items $\mathcal{D}$.
- $\mathcal{O}(1)$ **clause per variable:** If k-SAT is restricted so that each variable is in at most $\mathcal{O}(1)$ clauses, then $|\mathcal{D}| = n \cdot [2(2n)^{k-1}]^{\mathcal{O}(1)} = n^{\mathcal{O}(1)}$.

The Matrix Inversion Problem (MI): This problem was introduced in [ABBO$^+$11] and slightly modified by us. It is defined by a matrix $M \in \{0, 1\}^{n \times n}$ that is fixed and known to the algorithm. It is non-singular, has exactly seven ones in each row, at most K ones in each column, and is a $(r, 7, c)$-*boundary expander* (defined in Section 5.2), with $K \in \Theta(1)$, $c = 3$ and $r \in \Theta(n)$.

The input to this problem is a vector $b \in \{0, 1\}^n$. The output is an $x \in \{0, 1\}^n$ such that $Mx = b$ over GF-2. Each data item contains the name x_j of a variable and the indices of the at most K bits b_i from b involving this variable, i.e. those i for which there is a j such that $M_{\langle i,j \rangle} = 1$. Note $|\mathcal{D}| = n2^K$.

The Matrix Parity Problem (MP): The input consists of a matrix $M \in \{0,1\}^{n \times n}$. The output x_i, for each $i \in [n]$, is the parity of the i^{th} row, namely $x_i = \bigoplus_{j \in [n]} M_{\langle i,j \rangle}$. This is easy enough, but the problem is that this data is distributed in an awkward way. Each data item contains the name x_j of a variable and the j^{th} column of the matrix. Note $|\mathcal{D}| = n2^n$.

Subset Sum (SS): The input consists of a set of n n-bit integers, each stored in a data item. The goal is to accept a subset of the data items that adds up to a fixed known target T. Again $|\mathcal{D}| = n2^n$.

Subset Sum No Carries: Another version of SS does the GF-2 sum bit-wise so that there are no carries to the next bits. **NOTE: See the last line in Table figure 1. The bound for this problem is slightly better. I am happy to delete it or not.**

2.2 Definitions of Online, Greedy, Recursive Back-Tracking, and Dynamic Programming Algorithms

Ideally a formal model of an algorithmic paradigm should be able to capture the complete power and weaknesses of the paradigm. In this section, we focus on the defining characteristics of online, greedy, recursive back-tracking, and dynamic programming. The formal definition of the pFBT model appears in Section 2.3.

To make the discussion concrete, consider the Knapsack Problem specified as follows. A data item $D_i = \langle p_i, w_i \rangle$ specifies the price and the weight of the i^{th} object. The decision option is $\sigma_i \in \Sigma = \{0,1\}$, where 1 and 0 are the two options, meaning to accept and not to accept the object, respectively. The goal is to maximize $f(I,S) = \sum_i p_i \sigma_i$ as long as $\sum_i w_i \sigma_i \leq W$.

An deterministic *online* algorithm for such a problem would receive the data items D_i one at a time and must make an irrevocable decision σ_i about each as it arrives. For example, an algorithm for the Knapsack Problem may choose to accept the incoming data item as long as it still fits in the knapsack. We will assume that the algorithm has unbounded computational power based on what it knows from the data items it has seen already. The algorithm's limitation of power arises because it does not know the data items that it has not yet seen. After it commits to a *partial solution* $PS^{in} = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ by making decisions about the *partial instance* $PI^{in} = \langle D_1, D_2, \ldots, D_k \rangle$ seen so far, an adversary can make the future items $PI^{future} = \langle D_{k+1}, D_{k+2}, \ldots, D_n \rangle$ be such that there is no possible way to extend the solution with decisions $PS^{future} = \langle \sigma_{k+1}, \sigma_{k+2}, \ldots, \sigma_n \rangle$ so that the final solution $S = \langle PS^{in}, PS^{future} \rangle$ is an optimal solution for the actual instance $I = \langle PI^{in}, PI^{future} \rangle$. This lower bound strategy is at the core of all the lower bounds in this paper.

A *greedy* algorithm is given the additional power to specify a priority ordering function (greedy criteria) and is ensured that it will see the data items of the instance in this order. For example, an algorithm for the Knapsack Problem may choose to sort the data items by the ratio $\frac{p_i}{w_i}$. In order to prove lower bounds as done with the online algorithm, [BNR03] defined the *priority* model. It allows the algorithm to specify the priority order without allowing it to see the future data items by having it specify an ordering $\pi \in \mathcal{O}(\mathcal{D})$ of all possible data items. The algorithm then receives the next data item D_i in the actual input according this order. An *adaptive* algorithm is allowed to reorder the data items every time it sees an new data item (also known as fully adaptive priority algorithm in [BNR03, DI09]). **NOTE: Sounds good. Did you correct it?** An added benefit of receiving data item D is that the algorithm learns that every data item D' before D in the priority ordering is not one of the remaining data items in the input instance. The set of data items learned to *not* belong to the instance is denoted as PI^{out}.

It is a restriction on the power of the greedy algorithm to require it to make a single irrevocable decision σ_i about each data item before knowing the future data items. In contrast, *backtracking* algorithms search for a solution by first exploring one decision option about a data item and then if the search fails, then the algorithm backs up and searches for a solution with a different decision option. The computation forms a

tree. To model this [ABBO$^+$11] define [1] *prioritized Branching Trees (pBT)*. Each path of the computation tree is determined by priority ordering; each node reveals one data item D_i (according to the priority ordering for the path leading to this node, and makes zero, one, or more decisions σ_i about it. If more than one decision is made, the tree branches to accommodate these. If zero decisions are made then the computation path terminates. The computation returns the solution S that is best from all the nonterminating paths.

A computation of the algorithm on an instance I dynamically builds a tree of *states* as follows. Along the path from the root state to a given state u in the tree, the algorithm has seen some partial instance $PI_u = \langle D_{\langle u,1\rangle}, D_{\langle u,2\rangle}, \ldots, D_{\langle u,k\rangle}\rangle$ and committed to some partial solution $PS_u = \langle \sigma_{\langle u,1\rangle}, \sigma_{\langle u,2\rangle}, \ldots, \sigma_{\langle u,k\rangle}\rangle$ about it. At this state (knowing only this information) the algorithm specifies an ordering $\pi_u \in \mathcal{O}(\mathcal{D})$ of all possible data items. The algorithm then receives the next unseen data item $D_{\langle u,k+1\rangle}$ in the actual input according to this order. The algorithm then can make one decision $\sigma_{\langle u,k+1\rangle}$ that is irrevocable for the duration of this path, to fork making a number of such decisions, or to terminate this path of the search all together. The computation returns the solution S that is best from all the nonterminating paths.

A curious restriction of this model is that it does not allow the following completely reasonable knapsack algorithm. Return the best solution after running three parallel greedy algorithms, one with the priority function being largest p_i, another smallest w_i, and another largest $\frac{p_i}{w_i}$. Being fully adaptive, each state along each branch is allowed to choose a different ordering $\pi \in \mathcal{O}(\mathcal{D})$ for its priority function. However, the computation is only allowed to fork when it is making different decisions about a newly seen data item and this only occurs after seeing a data item. We generalize their model by allowing the algorithm to fork without seeing a data item. This is facilitated by, in addition to the *input reading* states described above, having *free branching* states which do nothing except branch to some number of children. This ability introduces a surprising additional power to the algorithms. We call this new model *prioritized Free Branching Tree* (pFBT) algorithms. See Section 2.3 for the formal definition.

Though it is not initially obvious how, pFBT (pBT) are also quite good at expressing some *dynamic programming* algorithms. **NOTE: Sounds good.** Certain dynamic programming algorithms derive a collection of sub-instances to the computational problem from the original instance such that the solution to each sub-instance is either trivially obtained or can be easily computed from the solutions to the subsubinstances. Generally, one thinks of these sub-instances being solved from the *smallest* to *largest*, but one can equivalently continue in the recursive backtracking model where branches are pruned when the same *subinstance* has previously been solved. Each state u in the pFBT computation tree can be thought of the root of the computation on the subinstance $PI^{future} = \langle D_{k+1}, D_{k+2}, \ldots, D_n\rangle$ consisting of the yet unseen data items, whose goal is to find a solution $PS^{future} = \langle \sigma_{k+1}, \sigma_{k+2}, \ldots, \sigma_n\rangle$ so that $S = \langle PS_u, PS^{future}\rangle$ is an optimal solution for the actual instance $I = \langle PI_u^{in}, PI^{future}\rangle$. Here, $PI_u^{in} = \langle D_{\langle u,1\rangle}, \ldots, D_{\langle u,k\rangle}\rangle$ is the partial instance seen along the path to this state and $PS_u = \langle \sigma_{\langle u,1\rangle}, \ldots, \sigma_{\langle u,k\rangle}\rangle$ is the partial solution committed to. It is up to the algorithm to decide which pairs of states it considers to be representing the "same" computation. For example, in the standard dynamic programming algorithm for the Knapsack Problem, the data items are always viewed in the same order. Hence, all states at level k have the same "subinstance" $PI^{future} = \langle D_{k+1}, D_{k+2}, \ldots, D_n\rangle$ yet to be seen. However, because different partial solutions $PS_u = \langle \sigma_{\langle u,1\rangle}, \ldots, \sigma_{\langle u,k\rangle}\rangle$ have been committed to along the path to different states u, their computation tasks are different. As far as the future computation is concerned, the only difference between the different subproblems is the amount $W' = W - \sum_{i\le k} w_i\sigma_{\langle u,i\rangle}$ of the knapsack remaining. Hence, for each such W', the algorithm should kill off all but one of these computation paths. This reduces the width from 2^k to being the range of W'. For a given value W', which of the paths should be kept is clearly the one that made the best initial partial solution $PS_u = \langle \sigma_{\langle u,1\rangle}, \ldots, \sigma_{\langle u,k\rangle}\rangle$ according to maximizing $\sum_{i\le k} p_i\sigma_{\langle u,i\rangle}$.

[1] What follows is a description of a strongly adaptive pBT algorithm pBT algorithm as defined in [ABBO$^+$11]. The weaker fixed order pBT (respectively, adaptive order pBT) models initially fix an ordering (respectively, create the same adaptive ordering for each path) rather than allow different priority orderings on each path.

An additional challenge in implementing the standard dynamic programming algorithm for the Knapsack Problem in the pBT model is the following. In the model, the algorithm does not know which path should live until it knows partial instance PI_u^{in}, however, after this partial instance is received, the pBT model does not allow cross talk between these different paths. This is where the unbounded computational power of the pBT model comes in handy. Independently, each path, knowing PI_u^{in}, knows what every other paths in the computation is doing. Hence, it knows whether or not it should be terminated. If required, it quietly terminates itself. In this way, pBT is able to model many dynamic programming algorithms.

Note: What you say about Bellman-Ford sounds right. I am not completely up on it. However, Bellman-Ford's dynamic programming algorithm for the shortest paths problem cannot be expressed in the model. Each state in the computation tree at depth k will have traversed a sub-path of length k in the input graph from the source node s, and the future goal is to find to a shortest path from the current node to the destination node t. States in the computation tree whose sub-path end in the same input graph node v are considered to be computing the same subproblem. However, because the different states u have read different paths $PI_u^{in} = \langle D_{\langle u,1\rangle}, \ldots, D_{\langle u,k\rangle}\rangle$ of the input graph, they cannot know what the other computation states are doing. Without cross talk, they cannot know whether they are to terminate or not. [BODI11] formally proves that pBT requires exponential width to solve shortest paths and defines another model called *prioritized Branching Programs* (pBP) which captures this notion of memoization by merging computation states that are computing the "same" subproblem. The computation then forms a DAG instead of a tree.

2.3 The Formal Definition of pFBT

We proceed with a formal definition of a pFBT algorithm $\mathcal{A}$. On an instance I, a pFBT algorithm builds a computation tree $\mathcal{T}_{\mathcal{A}}(I)$ as follows. Each node in the tree represents a *state*, which is either a *input reading* or *free branching* state. Consider a state u_p of the tree along a path p. It is labeled with $\langle PI_p, PS_p, f_p\rangle$, constituting the *partial instance*, the *partial solution*, and the *free branching* seen along this path. The partial information about the instance is split into two types $PI_p = \langle PI_p^{in}, PI_p^{out}\rangle$. Here $PI_p^{in} = \langle D_{\langle p,1\rangle}, \ldots, D_{\langle p,k\rangle}\rangle$ and $PS_p = \langle \sigma_{\langle p,1\rangle}, \ldots, \sigma_{\langle p,k\rangle}\rangle$, where $D_{\langle p,i\rangle} \in \mathcal{D}$ is the data item seen and $\sigma_{\langle p,i\rangle} \in \Sigma$ is the decision made about it in the i^{th} input reading state along path to p. $PI_p^{out} \subseteq \mathcal{D}$ denotes the set of data items inadvertently learned by the algorithm to be *not* in the input instance. Finally, $f_p = \langle f_{\langle p,1\rangle}, \ldots, f_{\langle p,k'\rangle}\rangle$, where $f_{\langle p,i\rangle} \in \mathbb{N}$ indicates which branch is followed in the i^{th} free branching state along path p. **NOTE: No. That was a mistake. Thanks** An input reading state u_p first specifies the priority function with the ordering $\pi_{\mathcal{A}}(u_p) \in 2^{\mathcal{D}}$ of the possible data items. Suppose $D_{\langle p,k+1\rangle}$ is the first data item from $I \setminus PI_p^{in}$ according to this total order. The input reading state u_p must then specify the decisions $c_{\mathcal{A}}(u_p, D_{\langle p,k+1\rangle}) \subseteq \Sigma$ to be made about $D_{\langle p,k+1\rangle}$.[2] For each $\sigma_{\langle p,k+1\rangle} \in c_{\mathcal{A}}(u_p, D_{\langle p,k+1\rangle})$, the state u_p has a child u'_p with $D_{\langle p,k+1\rangle}$ added to PI_p^{in}, each data item D' appearing before $D_{\langle p,k+1\rangle}$ in the ordering $\pi_{\mathcal{A}}(u_p)$ added to PI_p^{out}, and $\sigma_{\langle p,k+1\rangle}$ added to PS_p. A free branching state u_p must specify only the number $F_{\mathcal{A}}(u_p)$ of branches it will have. For each $f_{\langle p,k'+1\rangle} \leq F_{\mathcal{A}}(u_p)$, the state u_p has a child u'_p with $f_{\langle p,k'+1\rangle}$ added to f_p. The width $W_{\mathcal{A}}(n)$ of the algorithm $\mathcal{A}$ is the maximum over instances I with n data items, of the maximum over levels k, of the number of states u_p in the computation tree at level k.

Though this completes the formal definition of the model pFBT, in order to give a better understanding of it, we will now prove what such an algorithm knows about the input instance I when in a given state u_p. This requires understanding how the computation tree $\mathcal{T}_{\mathcal{A}}(I)$ changes for different instances I. If these $\mathcal{T}_{\mathcal{A}}(I)$ were allowed to be completely different for different instances I, then for each I, $\mathcal{T}_{\mathcal{A}}(I)$ could simply know the answer for I. Lemma 1 proves that the partial instance $PI_p = \langle PI_p^{in}, PI_p^{out}\rangle$ constitutes the sum

[2] If one wanted to measure the time until a solution is found, then one would want to specify the order in which these decisions $\sigma_{\langle p,k+1\rangle} \in c_{\mathcal{A}}(u_p, D_{\langle p,k+1\rangle})$ were tried.

knowledge that algorithm $\mathcal{A}$ knows about the instance. Namely, if we switched algorithm $\mathcal{A}$'s input instance from being I to being another instance I' consistent with this information, then $\mathcal{A}$ would remain in the same state u_p. Define $I \vdash PI_p$ to mean that instance I is consistent with p, i.e. contains the data items in PI_p^{in} and not those in PI_p^{out}.

Lemma 1. *Suppose that for input I there exists a path $p \in \mathcal{T}_{\mathcal{A}}(I)$ identified with $\langle PI_p, PS_p, f_p \rangle$. Suppose that a second input I' is consistent with this same partial instance PI_p, i.e. $I' \vdash PI_p$. It follows that there exists a path $p' \in \mathcal{T}_{\mathcal{A}}(I')$ identified with the same $\langle PI_{p'}, PS_{p'}, f_{p'} \rangle = \langle PI_p, PS_p, f_p \rangle$.*

Note: Allan, you are so right. This proof was bad. Russell added the idea of a super tree. Like you, I was not convinced it removed the need for induction. When I started writing up the proof it was much more subtle than I had previously thought. I removed the idea of a super tree because I don't feel that it added much. I also shortened the statement of the lemma itself. How do you feel about them now?

Proof. The proof uses the fact that pFBT considers deterministic algorithms that explore an unknown instance and given the same knowledge they always make identical decisions. The proof will be by induction on the length of the path. Suppose by way of induction that it is true up to some depth. Now consider a path $p \in \mathcal{T}_{\mathcal{A}}(I)$ followed by some input I whose length is one longer. Let $u_{(p-1)}$ and u_p denote the second last and last states in this path. If the second input I' is consistent with the last partial instance PI_p, i.e. $I' \vdash PI_p$, then it is consistent with the previous partial instance $PI_{(p-1)}$, i.e. $I' \vdash PI_{(p-1)}$, because it says less. And hence by the induction hypothesis there exists a path $(p'-1) \in \mathcal{T}_{\mathcal{A}}(I')$ identified with the same $\langle PI_{(p'-1)}, PS_{(p'-1)}, f_{(p'-1)} \rangle = \langle PI_{(p-1)}, PS_{(p-1)}, f_{(p-1)} \rangle$.

If this second last state is an input reading state, then its next task is to specify a priority function $\pi_{\mathcal{A}}(u_{(p-1)})$. The implication in this notation is that which priority function the pFBT algorithm $\mathcal{A}$ selects depends only on the label $\langle PI_{(p-1)}, PS_{(p-1)}, f_{(p-1)} \rangle$ of the state $u_{(p-1)}$. And hence, both inputs I and I' will be subjected to the same priority function.

A more subtle proof is that both inputs I and I' in this state will receive the same data item. Let $D = D_{\langle p,k+1 \rangle}$ denote the one received on input I. It follows that the partial instance PI_p labeled in the tree $\mathcal{T}_{\mathcal{A}}(I)$ specifies that this data item is in the input I. Because I' agrees with this partial instance, this data item must also be in I'. By way of contradiction, lets assume that on input I', despite I' containing D, the state receives D'. This must be because the priority function for I' puts a higher priority on D' than on D. But then we showed that the same is true on I. It follows that the partial instance PI_p labeled in the tree $\mathcal{T}_{\mathcal{A}}(I)$ specifies that this data item D' is not in the input I. Because I' agrees with this partial instance, this data item must also not be in I'. This contradicts the fact that D' was received under I'. Hence, we know that on both inputs the state received the same data item $D_{\langle p,k+1 \rangle}$.

Because on the two inputs I and I', the state has the same priority function and receives the same data item D, it follows that in both it learns that this data item D is in the input and that all the data items D' appearing before D in the priority order is not in the instance. Hence, their partial instance $PI_{p'} = PI_p$ remains the same. From this, we know that on both inputs the state will make the same decision $c_{\mathcal{A}}(u_p, D_{\langle p,k+1 \rangle})$, making their partial solutions $PS_{p'} = PS_p$ the same as well.

If the second last state in this path is a free branching state, then both inputs will have the same set of free branches $f_{\langle p,k'+1 \rangle}$. We will dictate that on I' the path p' follows the same free branch that p follows on I. This completes the inductive step that there exists a path $p' \in \mathcal{T}_{\mathcal{A}}(I')$ identified with the same $\langle PI_{p'}, PS_{p'}, f_{p'} \rangle = \langle PI_p, PS_p, f_p \rangle$. □

2.4 The Formal Definition of pFBT$^-$

NOTE: Yes. This is what Russel did not like too. Do you think it worth including or would you prefer it to be deleted entirely? I just don't care.

We now formally define the restricted model pFBT$^-$. Recall that *free data items* are defined to be additional data items that have no bearing on the solution of the problem. The reason adding these free data items to the pBT model increases its power is that when the algorithm reads data item D, it inadvertently learns that all data items appearing before D in the priority ordering are *not* in the instance. This allows a pBT algorithm to solve any computational problem with width $W = (n \log |\mathcal{D}|)^3$. Though this makes the proving of lower bounds in the model even more impressive, this aspect of the model is clearly not practical. The pFBT$^-$ model restricts the algorithm in a way that excludes the possibility of it implementing this impractical algorithm while still allowing any practical algorithm that is in pFBT to still be in pFBT$^-$.

The input instance in the model pFBT$^-$ is allowed to specify some of its data items to be *known*. The pFBT$^-$ algorithm knows that such a data item is in the input instance even though it has not yet received it. Each input reading state u_p of a pFBT$^-$ computation tree either specifies an order in which all of the possible unknown data items occur before the known ones in its priority order or visa versa. It is not allowed to mix the two types of data items together in the ordering. Similarly, the input instance is allowed to pair some of the data items to be *equivalent*. The pFBT$^-$ algorithm does not know which pairs of data items will be in the instance, but does know that if one is then the other will be as well. An input reading state of a pFBT$^-$ computation tree can either read such a pair together or when one of the pair is read, the other automatically becomes a known data item as previously described.

We argue that any practical algorithm that is in pFBT is still in pFBT$^-$ by arguing that there is no (direct) reason that an algorithm should want to receive a *known* data item. Already knowing that it is in the instance, it gains no new information, but it is still required to make an irrevocable decision about it, which it may or may not be prepared to do. The algorithm should either order the known data items first in order to get these data items out of the way, or after in order to avoid receiving them until the end when knowing the entire input instance is easy to make an irrevocable decision about them.

The flaw in this argument is that mixing the known and unknown data items is useful. The surprising algorithm from Theorem 2 does this to learn every data item that is not in the instance without receiving a single real unknown data item. No plausible poly-time algorithm, however, is able to do this. First, it requires the algorithm to keep track of an exponential amount of information. Second, it requires the algorithm to have unbounded computational power with which to instantly compute the solution for the now known instance.

3 Upper Bounds

In this section, we provide a surprising algorithm for any computational problem where the algorithm has access either to free branching and/or free data items, and the computational problem has a small set of possible data items $\mathcal{D}$. Next we will extend the generic algorithm to an algorithm for the k-SAT problem with only m clauses.

Theorem 2 (Algorithm with Free Branching and/or Free Data Items)**.** *Consider any computational problem with known input size n and $|\mathcal{D}| = d$ potential data items. There is a pFBT algorithm for this problem with width $W_{\mathcal{A}}(n) = 2^{\mathcal{O}(\sqrt{n \log d})}$. When m free data items are added to the problem, the required width decreases to $2^{\mathcal{O}(\frac{n \log d}{m + \sqrt{n \log d}})}$. When $m = \mathcal{O}(n \log d)$, it decreases to $\mathcal{O}(n \log d)^3$ even without free branching, i.e. in pBT.*

Proof. We first consider the case without free data items. We "construct" the deterministic algorithm by proving that for every fixed input instance, the probability of our randomized algorithm failing to find a correct solution is less than one over the number $\binom{d}{n} \leq d^n$ of instances. Hence, by the union bound we know that the probability that a randomly chosen setting of coin flips fails to work for all input instances is less than one. Hence, there must exist an algorithm that solves every input instance correctly.

In its first stage, the algorithm learns what the input instance is, not by learning each of the data items in the instance but by learning the complete set of $|\mathcal{D}|-n$ data items that are not in the input instance. It does this without having to commit to values for more than $\ell = \mathcal{O}(\sqrt{n \log d})$ variables. The algorithm branches when making a decision about each of these variables. Hence, the width is $\approx |\Sigma|^{\ell} = 2^{\mathcal{O}(\sqrt{n \log d})}$. In its second stage, the algorithm uses its unbounded computational power to instantly compute the solution for the now known instance. This solution will be consistent with one of its $|\Sigma|^{\ell}$ branches.

We now focus on the how the algorithm can use free branches to learn lots of data items that are not in the instance. The algorithm randomly orders the potential data items and learns the first one that is in the input instance. The items earlier in the ordering are learned to be not in the instance and are added to PI^{out}. This is expected to consist of a $\frac{1}{n}$ fraction of the remaining potential items. However, if instead, the algorithm free branches to repeat this experiment $F = 2^{\ell} = 2^{\mathcal{O}(\sqrt{n \log d})}$ times, then we will see below that with high probability at least one of these branches will result a in $\Theta(\frac{\ln(F)}{n}) = \Theta(\frac{\sqrt{\log d}}{\sqrt{n}})$ fraction of the remaining potential data items being added to PI^{out}. The algorithm might want to keep the one branch that performs the best, but this would require "cross talk" between the branches. Instead, it keeps any branch that performs sufficiently well. The threshold q_i is carefully set so that the expected number of branches alive stays about the constant r from one iteration to the next.

algorithm $pFBT(n, d)$

$\langle \boldsymbol{pre-cond} \rangle$**:** n is the number of data items in the input instance and d is the number of potential data items

$\langle \boldsymbol{post-cond} \rangle$**:** Forms a pFBT tree such that whp one of the leaves knows the solution

begin
- $\ell = \mathcal{O}(\sqrt{n \log d})$
- $F = 2^{\ell}$
- $r_0 = r = \mathcal{O}(\ell^2 n \log d) = \mathcal{O}(n \log d)^2$
- Width $\leq (2r) \times F \times |\Sigma|^{\ell} = 2^{\mathcal{O}(\sqrt{n \log d})}$
- Free branch to form r_0 branches
- Loop: $i = 1, 2, \ldots, \ell$
  - $\langle \boldsymbol{loop-invariant} \rangle$**:** The current states in the pFBT tree form a $r_{i-1} \times |\Sigma|^{i-1}$ rectangle. The r_{i-1} rows were formed using free branches. For each such row, the states know a set PI^{in}_{i-1} of $i-1$ of the data items known to be in the input instance. The column of $|\Sigma|^{i-1}$ states within this row were formed from decision branches in order to make all $|\Sigma|^{i-1}$ possible decisions on these $i-1$ variables. Except for making different decision, these states are identical. Each of the r_{i-1} rows of states also knows a set PI^{out}_{i-1} of the data items known to be not in the instance. PI^{out}_{i-1} has grown large enough so that there are only $d_{i-1} = d - |PI^{in}_{i-1}| - |PI^{out}_{i-1}|$ remaining potential data items.
  - Each state free branches with degree F forming a total of $r_{i-1}F$ rows
  - For each of these $r_{i-1}F$ rows of states
    - Randomly order the d_{i-1} remaining potential data items
    - D = the first data item that is in the input
    - q = # of data items before D in the order
    - $q_i = \Theta(\frac{d_i \ln(F)}{n})$, set so that $\Pr(q \geq q_i) = \frac{1}{F}$
    - if($q \geq q_i$) then

add D to PI_{i-1}^{in} and these q earlier data items to PI_{i-1}^{out}
$d_i = d_{i-1} - q_i - 1$ remaining potential data items
Make a decision branch to decide each possibility in Σ about D
else
Abandon all the branches in this row
end if
end for
r_i = the total # of these $r_{i-1}F$ rows that survive
if($r_i \notin [1, 2r]$) abort
end loop
We know what the input instance is by knowing all the data items that are not in it.
We use our unbounded power to compute the solution.
This solution will be consistent with one of our $|\Sigma|^\ell$ states.
end algorithm

The first thing to ensure is that $d_\ell = d - |PI_\ell^{in}| - |PI_\ell^{out}|$ becomes $n - \ell$ so the remaining possible data items must all be in the instance. At the beginning of the i^{th} iteration, there are d_{i-1} remaining potential data items. The algorithm randomly chooses one of these to be first in the ordering. It is one of those in our fixed input instance with probability at most $\frac{n}{d_{i-1}}$. Conditional on this one not being in the instance, the next has probability at most $\frac{n}{d_{i-1}-1}$ of being in the instance, while the last of concern has probability at most $\frac{n}{d_{i-1}-q-1} = \frac{n}{d_i}$. If all of these events occur, then at least q_i data items appear before the first that is in the instance, i.e. $\Pr[q \geq q_i] \geq (1-\frac{n}{d_i})^{q_i} \approx e^{-nq_i/d_i}$. (This approximation is good until the end game. See below for how the time for that is bounded.) Because we have F chances and we want the expected number of these to survive to be one, we set $q_i = \frac{d_i \ln(F)}{n}$ so that this probability is $\frac{1}{F}$. For the states that achieve $q \geq q_i$, the number of remaining potential data items is $d_i = d_{i-1} - q_i - 1 \leq d_{i-1} - \frac{d_i \ln(F)}{n} = d_{i-1} - \frac{d_i \ell}{n} = \frac{d_{i-1}}{1+\frac{\ell}{n}}$. Hence, in the end $d_\ell = \frac{d}{(1+\frac{\ell}{n})^\ell} \approx \frac{d}{e^{\frac{\ell^2}{n}}} = \frac{d}{e^{\frac{\sqrt{n \ln d^2}}{n}}} = 1$. Before this occurs, the input instance is known.

As is often the case, the last 20% of the work takes 80% the time. The approximation $(1 - \frac{n_i}{d_i})^{q_i} \approx e^{-n_i q_i/d_i}$ fails in the end game when the number of remaining possible data items d_i gets close to the number of unseen input items n_i, say $d_i \leq 2n_i$. This means the number of data items that must still be eliminated is at most n_i. Not being done means that $d_i \geq n_i + 1$ giving that $\Pr[q \geq q_i] \geq (1 - \frac{n_i}{d_i})^{q_i} \geq \frac{1}{d_i}^{q_i}$. Setting this to F^{-1} and solving gives that $q_i \geq \frac{\log_2(F)}{\log_2(d_i)} = \frac{\mathcal{O}(\sqrt{n \log d})}{\log_2(d_i)}$. If this many data items are eliminated each round then the number of required rounds to eliminate the remaining n_i data elements is at most $\frac{n_i}{q_i} \leq \mathcal{O}(\sqrt{n \log d})$. This at most doubles the number of rounds already being considered.

We must also bound the probability that the algorithm aborts to be less than one over the number d^n of instances. It does so if $r_\ell \notin [1, 2r]$ because it needs at least one branch to survive and we don't want the computation tree to grow too wide. Lemma 3 proves that this failure probability is at most $2^{-\Theta(r/\ell^2)}$, which is d^{-n} as long as $r = \Theta(\ell^2 n \log d) = \Theta(n \log d)^2$. This gives a total width of $W_{\mathcal{A}}(n) \leq (2r) \times F \times |\Sigma|^\ell = 2^{\mathcal{O}(\sqrt{n \log d})}$ as required. This completes the proof when there are no free data items.

Free data items provide even more power than free branching does. Just as done above the set PI_i^{out} of data items known not to be in the instance grows every time one of these data items is received. However, because they have no bearing on the solution of the problem, the algorithm need not branch making all possible decisions Σ^m about them. The algorithm is identical to the one above, except that $m' = \min(m, n)$ of the free data items are randomly ordered into the remaining real data items. The next data item received will either be one of the n real data items or one of these m' free data items. As before, let q denote the number of data items before this received data item in the priority ordering, i.e. the number added to PI_{i-1}^{out}. $\Pr[q \geq q_i] \geq (1 - \frac{n+m'}{d_i})^{q_i} \geq e^{-2nq_i/d_i}$. During this first stage, the algorithm wants to receive all m

of the free data items and only receive ℓ of the real data items. If a real data item is read after already having received ℓ of them, then the branch dies. $\Pr[q \geq q_i$ and the data item received is a free one$] \geq e^{-2nq_i/d_i} \times \frac{m'}{n+m'}$. Still having $F = 2^\ell$ chances and wanting the expected number of these to survive to be one, we set $q_i = \frac{d_i(\ln(F)-\ln(\frac{m'}{n+m'}))}{2n} \geq \frac{d_i\ell}{3n}$ so that this probability is $\frac{1}{F}$. ($\ln(F) = \ell$ will be set to $\mathcal{O}(\frac{n\log d}{m}) \gg \ln(\frac{m'}{n+m'})$.) Hence, $d_i = d_{i-1} - q_i - 1 \leq \frac{d_{i-1}}{1+\frac{\ell}{3n}}$, giving in the end $d_{m+\ell} = \frac{d}{(1+\frac{\ell}{3n})^{m+\ell}} \approx \frac{d}{e^{\frac{\ell(m+\ell)}{3n}}} < 1$, when $\ell = \Omega(\frac{n\log d}{m+\sqrt{n\log d}})$. The probability that the algorithm aborts because $r_\ell \notin [1, 2r]$, by Lemma 3 is at most $2^{-\Theta(r/(m+\ell)^2)}$, which is d^{-n} as long as $r = \Theta((m+\ell)^2 n\log d)$. This is $r = \Theta(n\log d)^2$ when $m = 0$ and grows to $\Theta(n\log d)^3$ when $m = \mathcal{O}(n\log d)$. This gives a total width of $W_{\mathcal{A}}(n) \leq (2r) \times F \times |\Sigma|^\ell = 2^{\mathcal{O}(\frac{n\log d}{m+\sqrt{n\log d}})}$ as required.

The remaining point is that this can be achieved without free branching, i.e. in pBT, when $m = \mathcal{O}(n\log d)$ and $W_{\mathcal{A}}(n) \leq 2r = \mathcal{O}(n\log d)^3$. In this case, a branch terminates every time a real data item is received, i.e. $\ell = 0$. Only $F = 2$ branches are needed each time a free data item is received. Though we do not have free branches, this can be achieved by having the algorithm branch on the Σ different possible decisions for this received free data item. □

We now bound the probability that the number of rows starting at r goes to zero or expands past $2r$.

Lemma 3. *Start with r rabbits. Each iteration $i \in [\ell]$, all the rabbits currently alive have F babies (and dies). Each baby lives independently with probability $\frac{1}{F}$. Hence, the expected number of rabbits remaining is again r. The game succeeds if the population does not die out and there are never more than $2r$ rabbits. The probability of failure is at most $2^{-\Theta(r/\ell^2)}$.*

Proof. We are given $\mathrm{Exp}[r_i] = r_{i-1}$. Set $h = \frac{\sqrt{r/2}}{\ell}$. Using Chernoff bounds, the probability that r deviates by more than $h\sqrt{r_{i-1}}$ is at most $2^{-\Theta(h^2)}$. The probability that this occurs in any at least one of the ℓ iterations is at most $\ell 2^{-\Theta(h^2)} = 2^{-\Theta(r/\ell^2)}$. Otherwise, r_i is always below $2r$ and each of the ℓ iterations changes r_i by at most $h\sqrt{2r} = \frac{r}{\ell}$. □

In reality, these changes perform a random walk, because each of these changes is in a random direction. But even if they are all in the same direction, the total change is at most r, which neither zeros, nor doubles r_i.

The width $2^{\mathcal{O}(\sqrt{n\log|\mathcal{D}|})}$ of the algorithm described in the last section is $2^{\mathcal{O}(\sqrt{n\log n})}$ for k-SAT when restricted to having only $\mathcal{O}(1)$ clauses per variable, because then the domain $\mathcal{D}$ of data items is no bigger than $n^{\mathcal{O}(1)}$. However, for general k-SAT problem, this result does not directly improve the width because there are $|\mathcal{D}| = n \cdot 2^{2(2n)^{k-1}}$ potential data items. Despite this, this algorithm can surprisingly be extended to $2^{\mathcal{O}(n^{1/3}m^{1/3}\log n)}$ when the instance is restricted to having only m clauses.

Theorem 4 (k-SAT m-clauses)**.** *When there are only m clauses, then k-SAT can be done with $2^{\mathcal{O}(k^{1/2}n^{1/3}m^{1/3}\log n)}$ width.*

For $k = 3$, this is $2^{\mathcal{O}(n^{2/3}\log n)}$ when there are only $m = \Theta(n)$ clauses and $2^{o(n)}$ when there are only $m = o(n^2)$ clauses. The problem is that there may be $m = \Theta(n^3)$ clauses.

Proof. The first step is to ask for all variables that appear in at least $\ell = k^{1/2}n^{-1/3}m^{2/3}$ clauses. There are at most $\frac{km}{\ell} = k^{1/2}n^{1/3}m^{1/3}$ of these. Branching on each of these requires width $2^{\mathcal{O}(k^{1/2}n^{1/3}m^{1/3})}$ width. For each of the remaining variables, only $k\ell\log(2n)$ bits are needed to specify the k variables in each of the at most ℓ clauses that they are in. Hence, the number of possible data items for each of these is at most $d = (2n)^{k\ell}$. Theorem 2 then gives a pFBT algorithm for this problem with width $w = 2^{\mathcal{O}(\sqrt{n\log d})} = 2^{\mathcal{O}(\sqrt{nk\ell\log n})} = 2^{\mathcal{O}(k^{1/2}n^{1/3}m^{1/3}\log n)}$. □

4 Reductions from Hard to Easy Problems

This section presents the reductions to show that if pBT or pFBT cannot do well on the simple matrix problems then they cannot do well on the hard problems either.

Theorem 5 (Lower Bounds for Hard Problems)**.** *7-SAT requires width $2^{\Omega(n)}$ in pBT and $2^{\Omega(\sqrt{n})}$ in pFBT. Subset Sum requires width $2^{\Omega(n/\log n)}$ in pFBT and $2^{\Omega(n)}$ in pFBT$^-$. Finally, Subset Sum without carries requires width $2^{\Omega(n)}$ in pFBT.*

Proof. Lemma 6 gives a reduction to 7-SAT from the Matrix Inversion Problem in both the pBT and the pFBT models. Lemma 7 gives a reduction in pFBT to Subset Sum (with and without carries) from the Matrix Parity Problem with free data items. Lemma 8 gives a reduction from Subset Sum in the pFBT$^-$ model to the Matrix Parity Problem with no free data items in the pFBT model. The results then follow from the lower bounds for the matrix problems given in Theorem 9. □

Lemma 6. *[Reduction to 7-SAT] If 7-SAT can be solved with width W in the pBT (pFBT) model, then the Matrix Inversion Problem can be solved with the same width.*

Recall that the Matrix Inversion Problem is defined by a fixed and known matrix $M \in \{0,1\}^{n\times n}$ with at most seven ones in each row and $K \in \mathcal{O}(1)$ in each column. The input is a vector $b \in \{0,1\}^n$. The output is $x \in \{0,1\}^n$ such that $Mx = b$. Each data item contains the name x_j of a variable and the value of the at most K bits b_i from b involving this variable, i.e. those for which $M_{\langle i,j\rangle} = 1$.

Proof. Given a pBT (pFBT) algorithm for 7-SAT, we design a pBT (pFBT) algorithm for the Matrix Inversion Problem as follows. Consider a matrix inversion data item. Let x_j be the variable specified and b_i be one of the at most K bits from b involving this variable, i.e. $M_{\langle i,j\rangle} = 1$. Because the i^{th} row of M has at most seven ones, its contribution to $Mx = b$ is that the parity of these corresponding seven x variables must be equal to b_i. This parity can be expressed as the AND of $\frac{1}{2}2^7 = 64$ clauses involving these seven variables. These $64K$ clauses involving variable x_j for each of the K bits b_i are included in the one 7-SAT data item corresponding to this matrix inversion data item. Note that as required for the 7-SAT problem, this constructed data item contains a variable x_j and the list of clauses containing this variable. Each clause contains at most seven variables. As with a matrix inversion data item, the only new information that an algorithm who is aware of the reduction learns from a new 7-SAT data item is the value of the at most K bits b_i from b involving this variable. Finally, note that the output of both the matrix inversion problem and of 7-SAT is $x \in \{0,1\}^n$ such that $Mx = b$. □

Lemma 7. *[Reduction to Subset Sum with free data items] If Subset Sum can be solved with width W in pFBT, then the Matrix Parity Problem, with the introduction of $m = \mathcal{O}(n \log n)$ free data items, can be solved with the same width. If the version of SS without carries can be solved then only $m = \mathcal{O}(n)$ free data items need to be added to the Matrix Parity Problem.*

Recall that the Matrix Parity Problem is given as input a matrix $M \in \{0,1\}^{n\times n}$. For each $j \in [n]$, there is a data item containing the name x_j of a variable and the j^{th} column of the matrix. The output x_i, for each $i \in [n]$, is the parity of the i^{th} row, namely $x_i = \bigoplus_{j\in[n]} M_{\langle i,j\rangle}$.

Proof. Given a pFBT algorithm for SS (Subset Sum), we design a pFBT algorithm for the MP (Matrix Parity Problem) as follows. We map the matrix M instance for MP to a $2n + \mathcal{O}(n \log n)$ integer instance for SS. These integers will be such that if you write them in binary in separate rows j, lining up the $2^{i'}$ bit places into columns, then at least part of this will be the transpose of M. Let D_j denote the MP data item containing the variable name x_j and the j^{th} column of the matrix M. This data item will get mapped to two SS data items $D_{\langle j,-\rangle}$ and $D_{\langle j,+\rangle}$. Each such integer is $4n \log_2 n$ bits long. Of these $2n$ are special.

More specifically, for $i, j \in [n]$, we make the $2^{c_{j'}}$ and the 2^{r_i} bit places special, where $c_{j'} = j' \cdot 2\log n$ and $r_i = 2n\log n + i \cdot 2\log n$. Consecutive special bits are separated by at least $2\log_2 n$ zeros so that the sum over all the numbers of one of the special bits does not carry into the next special bit. To identify the column-integer relation, the ${c_{j'}}^{th}$ bit of $D_{\langle j,-\rangle}$ will be one iff $j' = j$. To store the j^{th} column of the matrix, for $i \in [n]$, the ${r_i}^{th}$ bit is $M_{\langle i,j\rangle}$. The second integer $D_{\langle j,+\rangle}$ associated with data item D_j is exactly the same as $D_{\langle j,-\rangle}$ except that the ${r_j}^{th}$ bit of $D_{\langle j,-\rangle}$ storing the diagonal entry $M_{\langle j,j\rangle}$ is complemented. The SS instance will have another $\mathcal{O}(n\log n)$ integers in dummy data items, in order to deal with carries from one bit to the next. For each bit index $k \in [1, \lceil \log n \rceil]$ and each row $i \in [n]$, the integer $E_{\langle i,k\rangle}$ will be one only in the $r_i + k^{th}$ bit. The target T will be one in the ${c_j}^{th}$ bit and the $r_i + \lceil \log n \rceil^{th}$ bit for each $i, j \in [n]$.

Having mapped an instance of the MP problem to an instance of the SS problem, we must now prove that the solutions to these two instances correspond to each other. Let S_{sum} be a solution to the SS instance described above. We first note that for each j, S_{SS} must accept exactly one of $D_{\langle j,+\rangle}$ and $D_{\langle j,-\rangle}$ because these are the only integers with a one in the ${c_j}^{th}$ bit and the target T requires a one in this bit. Let S_{MP} be the solution to the MP instance in which $x_j = 1$ if and only if $D_{\langle j,+\rangle}$ is accepted by S_{SS}. We now prove that this is a valid solution, i.e. $\forall i \in [n],\ x_i = \bigoplus_{j\in[n]} M_{\langle i,j\rangle}$. Consider some index i. Because S_{SS} is a valid solution, we know that its integers sum up to T and hence the ${r_i}^{th}$ bit of the sum is zero. Because there is no carry to the ${r_i}^{th}$ bit, we know that the parity of the ${r_i}^{th}$ bit of the integers in S_{SS} is zero. The dummy integers $E_{\langle i',k\rangle}$ are zero in this bit. For $j \neq i$, both $D_{\langle j,+\rangle}$ and $D_{\langle j,-\rangle}$ have $M_{\langle i,j\rangle}$ in this bit and as seen exactly one of them is in S_{SS} and hence these integers contribute $M_{\langle i,j\rangle}$ to the parity. For $j = i$, it is slightly trickier. $D_{\langle i,-\rangle}$, if it is in S_{SS}, also contributes $M_{\langle i,i\rangle}$ to the parity, while $D_{\langle i,+\rangle}$ contributes the compliment $M_{\langle i,i\rangle} \oplus 1$. In this first case, $x_i = 0$ and in the second $x_i = 1$. Hence, we can simplify this by saying that together $D_{\langle i,-\rangle}$ and $D_{\langle i,+\rangle}$ contribute $M_{\langle i,i\rangle} \oplus x_i$. Combining these gives that the parity of the ${r_i}^{th}$ bit of the integers in S_{SS} is $0 = \left[\bigoplus_{j\neq i} M_{\langle i,j\rangle}\right] \oplus \left[M_{\langle i,i\rangle} \oplus x_i\right]$. This gives as required that $x_i = \bigoplus_{j\in[n]} M_{\langle i,j\rangle}$. And hence, S_{MP} is a valid solution.

Now we must go in the opposite direction. Let S_{MP} be a solution for our MP problem. Put $D_{\langle j,+\rangle}$ in S_{SS} if $x_j = 1$, otherwise put $D_{\langle j,-\rangle}$ in it. For each row i, let $N = 2^{\lceil \log n \rceil}$ and $N_i = N - (x_i + \sum_{j\in[n]} M_{\langle i,j\rangle})$ and let the binary expansion of N_i be $[N_i]_2 = \left\langle N_{\langle i,\lceil \log n\rceil\rangle}, \ldots, N_{\langle i,0\rangle}\right\rangle$, so that $N_i = \sum_{k\in[0,\lceil \log n\rceil]} N_{\langle i,k\rangle} 2^k$. For $k \neq 0$, include the integer $E_{\langle i,k\rangle}$ in S_{SS} if and only if $N_{\langle i,k\rangle} = 1$. We must now prove that the integers in S_{SS} add up to T. The same argument as above proves that the ${c_j}^{th}$ and the ${r_i}^{th}$ bits of the sum are as needed for T. What remains is to deal with the carries. The ${c_j}^{th}$ bits will not carry. The ${r_i}^{th}$ bits in the $D_{\langle j,+\rangle}$ or $D_{\langle j,-\rangle}$ integers of S_{SS} add up to $x_i + \sum_{j\in[n]} M_{\langle i,j\rangle}$. For $k \in [1, \lceil \log n \rceil]$, the integer in $E_{\langle i,k\rangle}$ is one only in the $r_i + k^{th}$ bit and hence can be thought of contributing 2^k to the ${r_i}^{th}$ bit. Together, they contribute $\sum_{k\in[1,\lceil \log n\rceil]} N_{\langle i,k\rangle} 2^k$, which by construction is equal to $N_i - N_{\langle i,0\rangle}$. Because S_{MP} is a solution, $N_i = N - (x_i + \sum_{j\in[n]} M_{\langle i,j\rangle})$ is even, making $N_{\langle i,0\rangle} = 0$. The total contribution to the ${r_i}^{th}$ bit is then $(x_i + \sum_{j\in[n]} M_{\langle i,j\rangle}) + N_i = N = 2^{\lceil \log n\rceil}$, which carries to be zeros everywhere except in the $r_i + \lceil \log n \rceil^{th}$ bit. This agrees with the target T.

What remains to prove is that the MP computation tree can state-by-state mirror the SS computation tree. In addition to the MP problem having n data items D_i, it will have $m = \mathcal{O}(n\log n)$ free data items. For each $i \in [n]$, the MP problem will have a free data item labeled $D_{\langle i,2^{nd}\rangle}$ and for each $i \in [n]$ and $k \in [1, \lceil \log n \rceil]$, it will have one labeled $E_{\langle i,k\rangle}$. In each state of the computation tree, the SS algorithm specifies an priority ordering of its data items $D_{\langle i,+\rangle}$, $D_{\langle i,-\rangle}$, and $E_{\langle i,k\rangle}$. The simulating MP algorithm constructs as follows its ordering of its data items D_i, $D_{\langle i,2^{nd}\rangle}$, and $E_{\langle i,k\rangle}$. If neither $D_{\langle i,+\rangle}$ nor $D_{\langle i,-\rangle}$ have been seen yet, then the first occurrence of them in this ordering is replaced by D_i. The second occurrence of them can be ignored because it will never come into play. If at least one of $D_{\langle i,+\rangle}$ or $D_{\langle i,-\rangle}$ has been seen already, then the occurrence of the other one in this SS ordering is replaced by the free data item $D_{\langle i,2^{nd}\rangle}$.

Any dummy data item $E_{\langle i,k\rangle}$ in the SS ordering are replaced by the corresponding free data item $E_{\langle i,k\rangle}$ in the MP ordering. If according to this SS priority ordering, SS receives the first of $D_{\langle i,\pm\rangle}$, then MP according to its mirrored ordering will receive D_i. Note that MP learns the same information (or more[3]) about its instance that SS does. If on the other hand, SS receives the second of $D_{\langle i,\pm\rangle}$ or receives a dummy data item $E_{\langle i,k\rangle}$, then MP will receive the corresponding free data item. The MP algorithm (and we can assume the SS algorithm) knows that its instance is coming from this reduction and hence they knew before receiving it that this dummy/free data item is in the instance and hence neither gains any new information from this fact when it is received. The information of interest that they both inadvertently learn is that all data items D' in the ordering before this received data item are learned to not be in the instance and as such are added to PI^{out}. Because the MP algorithm always learns the same information that SS does, MP is able to continue simulating SS's algorithm.

If we are reducing from the carry free version of Subset Sum then the dummy data items $E_{\langle i,k\rangle}$ are not needed. □

Lemma 8. *[Reduction to Subset Sum in pFBT$^-$] If Subset Sum can be solved with width W in the pFBT$^-$ model, then the Matrix Parity Problem without free data items can be solved in pFBT with the same width.*

Proof. Given a pFBT$^-$ algorithm for SS, our goal is to construct a pFBT algorithm for MP. We map the matrix M instance for MP to an instance for SS as before with integers $D_{\langle j,-\rangle}$, $D_{\langle j,+\rangle}$, and $E_{\langle i,k\rangle}$. We must assume that the Subset Sum algorithm, SSA, knows that it is receiving an input from the reduction. Hence, the data items $E_{\langle i,k\rangle}$ are *known* in that the algorithm knows that they are in the actual input instance even though it has not yet received them. The data items $D_{\langle j,-\rangle}$ and $D_{\langle j,+\rangle}$ are paired to be *equivalent* in that the algorithm knows that if one is the input then the other will be as well. The pFBT$^-$ model then does not allow SSA to have *mixed* reading states that intertwine the unknown and known data item together in its priority orderings. Consider an *unknown* reading state of SSA, i.e. one that puts all of the possible unknown data items before the known ones. The simulating MP algorithm constructs its ordering of its data items D_i by replacing the first occurrence of $D_{\langle i,\pm\rangle}$ with D_i. If SSA receives one of $D_{\langle i,\pm\rangle}$, then MP receives D_i. If SSA receives a known data item, then this means that the current computation path is done because the input instance is completely known. Now consider a *known* reading state of SSA, i.e. one that puts all of the known ones before the possible unknown ones. Both SSA and MP know that these known data items are in SSA's instance and hence know that SSA will receive the first one in this order. In fact, there was no point in this read state at all. If SSA forks in order to make more than one irrevocable decision about this known data item then MP makes this same branches using a free branch. This free branch was one of the initial motivators of having free branches. □

5 Lower Bounds for the Matrix Problems

This section will prove the following lower bounds for the matrix problems.

Theorem 9 (Lower Bounds for the Matrix Problems)**.** *The Matrix Inversion Problem requires width $2^{\Omega(n)}$ in the pBT model and width $2^{\Omega(\sqrt{n})}$ in the pFBT model. The Matrix Parity Problem requires width $2^{\Omega(n)}$ in the pFBT model. If included in these problems are m* free data items*, then widths $2^{\Omega\left(\frac{n}{m+\sqrt{n}}\right)}$ and $2^{\Omega\left(\frac{n^2}{m+n}\right)}$ are still needed.*

This Section will be organized as follow. We will begin by developing in Section 5.1 a general technique for proving lower bounds on the width of the computation tree of any algorithm solving a given problem

[3]SS may not be able differentiate between $D_{\langle i,+\rangle}$ and $D_{\langle i,-\rangle}$, while MP can.

in either the pBT or the pFBT models. Theorem 10 formally states and proves that this technique works. We will then show how it is applied to get all of the results in Theorem 9. This proof still depends on four lemmas specific to the problems at hand. Section 5.2 proves Lemmas 12 and 13 for the Matrix Parity Problem and Section 5.3 proves Lemmas 16 and 17 for the Matrix Inversion Problem. Finally, Section 5.4 proves that the required boundary expander matrices exist.

5.1 The Lower Bound Technique

As said, we begin developing a general technique for proving lower bounds for pBT and pFBT models which we will use to prove Theorem 9 subject to four lemmas proved later.

For each input instance I, the computation tree $\mathcal{T}_{\mathcal{A}}(I)$ has height $n = |I|$ measured in terms of the number of data items received. Let l be a parameter, $\Theta(n)$ or $\Theta(\sqrt{n})$ depending on the result. We will focus our attention on *partial paths* p from the root to a state at level l in $\mathcal{T}_{\mathcal{A}}(I)$. Recall that such states are uniquely identified with $\langle PI_p^{in}, PI_p^{out}, PS_p, f_p \rangle$. The *partial instance* $PI_p = \langle PI_p^{in}, PI_p^{out} \rangle$ consists of the l data items known to be in the instance and those inadvertently learned to be not in the instance. Lemma 1 proves that PI_p constitutes the sum knowledge that algorithm $\mathcal{A}$ knows about the instance I in this state. With only this limited knowledge, it is unlikely that the *partial solution* PS_p that the algorithm has irrevocably decided about the data items in PI_p^{in} are correct. Formally, we prove that $\Pr_I[S(I) \vdash PS_p \mid I \vdash PI_p]$ is small, where $I \vdash PI_p$ is defined to mean that instance I is consistent with p, i.e. contains the data items in PI_p^{in} and not those in PI_p^{out} and $S(I) \vdash PS_p$ is defined to mean that the decisions PS_p are consistent with the/a solution for I.

With free branching, in depth only $l_{upper} = \mathcal{O}(\sqrt{n \log d})$, the unreasonable upper bound in Theorem 2 manages to learn the entire input I not by having PI_p^{in} contain all of its data items but by having PI_p^{out} contain all of the data items not in it. This demonstrates how having PI_p^{out} extra ordinarily large, can cause $I \vdash PI_p$ to give the algorithm sufficient information about the instance I that it can deduce a correct partial solution PS_p causing $\Pr_I[S(I) \vdash PS_p \mid I \vdash PI_p]$ to be far too big. Hence, we define a path p to be *bad* if this is the case.

A pFBT algorithm $\mathcal{A}$ is allowed to fork both in a *free branching state* in order to try different priority orderings and in an *input reading* state in order to try different irrevocable decisions. Each of the computation paths p for a given tree $\mathcal{T}_{\mathcal{A}}(I)$ can be uniquely identified by a *tuple of forking indexes* $f = \langle \sigma_1, \ldots, \sigma_l,\ f_1, \ldots, f_l \rangle$, where $\sigma_i \in \Sigma$ is indicates the decision made in the i^{th} input reading state and $f_i \in \mathbb{N}$ indicates which branch is followed in the i^{th} free branching state. Note that if the width of the computation tree $\mathcal{T}_{\mathcal{A}}(I)$ is bounded by $W_{\mathcal{A}}(n)$, then each f_i is at most $W_{\mathcal{A}}(n)$. This allows us to define $Forks_{pFBT} = [\Sigma \cup [W_{\mathcal{A}}(n)]]^l$ be the set of possible tuples of forking indexes. In contrast, a pBT algorithm is not allowed free branching, and hence $Forks_{pBT} = \Sigma^l$. The only difference between these two models that we will use is the sizes of these sets. Once we fix such a tuple f (independent of an input I), we can then define the *algorithmic strategy* $\mathcal{A}_f$ used by the branching program $\mathcal{A}$ to be the mapping between the input I and the $\langle PI_p^{in}, PI_p^{out}, PS_p, f_p \rangle$ identifying the state at the end of the path identified by tuple f.

This next theorem outlines the steps sufficient to prove that the width of the algorithm must be high.

Theorem 10 (The Lower Bound Technique)**.**

1. *Define a probability distribution $\mathcal{P}$ on a finite family of hard instances.*
   - *For both the Matrix Parity and Matrix Inversion Problems the distribution is uniform on the legal instances with n data items.*
2. *Only consider partial paths p from the root to a state at level l in the tree $\mathcal{T}_{\mathcal{A}}(I)$.*
   - *$l = \Theta(n)$ or $\Theta(\sqrt{n})$ depending on the result.*

3. *A path p is defined to be* good *if its set PI_p^{out} is good.*

   - *For the Matrix Inversion Problems, PI_p^{out} is defined to be* good *if it is sufficiently small. For the Matrix Parity Problems, it only must be sufficiently small with respect to at least one variable x_j.*

4. *Prove that if p is a good path at level l, then $\Pr_I[S(I) \vdash PS_p \mid I \vdash PI_p] \leq pr_{good}$, namely that the information about the instance I gained from $PI_p = \langle PI_p^{in}, PI_p^{out} \rangle$ is not sufficient to effectively make the irrevocable decisions PS_p about the data items in PI_p^{in}.*

   - *If the algorithm simply guesses the decision PS_p for each of the l data items in PI_p^{in}, then it is correct with probability $|\Sigma|^{-l} = 2^{-l}$. For both the Matrix Parity and the Matrix Inversion problems, we are able to bound $pr_{good} \leq 2^{-\Omega(l)}$. See Lemmas 13 and 17.*

5. *Prove that with probability at least $\frac{1}{2}$, every set PI_p^{out} in $\mathcal{T}_{\mathcal{A}}(I)$ is sufficiently small so that it is considered* good*. Consider a tuple of forking indexes $f \in Forks$. Consider some algorithmic strategy $\mathcal{A}_f$ along these forking indexes. Prove that $\Pr_I\left[\mathcal{A}_f \text{ produces a } PI^{out} \text{ that is bad}\right] \leq pr_{bad} \leq \frac{1}{2|Forks|}$. Note that $\frac{1}{2|Forks|}$ gives us the probability $\frac{1}{2}$ after doing the union bound.*

   - *For the Matrix Inversion Problem, we are able to bound $pr_{bad} \leq 2^{-\Omega(n)}$. For the Matrix Parity Problem, we are able to do much better bounding it by $2^{-\Omega(n^2)}$. See Lemmas 16 and 12.*

These three step gives that any pFBT (pBT) algorithm $\mathcal{A}$ requires width $W_{\mathcal{A}}(n) \geq \frac{1}{2pr_{good}}$.

We now prove that the technique works.

Proof. Step 5 in the technique proves that for a random instance I, the probability that $\mathcal{T}_{\mathcal{A}}(I)$ contains a bad partial path is at most $\frac{1}{2}$. Hence, any working algorithm $\mathcal{A}$ must solve the problem with probability at least $\frac{1}{2}$ using a full path whose first l levels is a good partial path. This requires that there exists a partial path $p \in \mathcal{T}_{\mathcal{A}}(I)$ such that the irrevocable decisions PS_p made along it are consistent with the/a solution $S(I)$ of the instance, namely

$$\tfrac{1}{2} \leq \Pr_I[\exists p \in \mathcal{T}_{\mathcal{A}}(I)_l, \text{ such that } p \text{ is good and } S(I) \vdash PS_p]$$

This probability deals only with paths in $\mathcal{T}_{\mathcal{A}}(I)$, which in turn depends on the randomly selected instance I. In contrast, the Step 4 probability, $\Pr_I[S(I) \vdash PS_p \mid I \vdash PI_p] \leq pr_{good}$, talks about fixed sets PI_p and PS_p that can depend on the algorithm $\mathcal{A}$ as a whole but not on our current choice of I. To understand the algorithm $\mathcal{A}$ independent of a particular instance I, define $Paths = \{p = \langle PI_p, PS_p, f_p \rangle \mid \exists I' \text{ such that } p \text{ is a partial path of length } l \text{ in } \mathcal{T}_{\mathcal{A}}(I') \}$ and $Paths_g$ those that are good. Note that $p \in \mathcal{T}_{\mathcal{A}}(I) \Rightarrow [p \in Paths \text{ and } I \vdash PI_p]$. Hence,

$$\Pr_I[\exists p \in \mathcal{T}_{\mathcal{A}}(I)_l, \text{ such that } p \text{ is good and } S(I) \vdash PS_p]$$

$$\leq \Pr_I[\exists p \in Paths_g, \text{ such that } I \vdash PI_p \text{ and } S(I) \vdash PS_p]$$

The union bound, conditional probabilities, and plugging in the probability from Step 4 translates this probability into the following.

$$\leq \sum_{p \in Paths_g} \Pr_I[I \vdash PI_p \text{ and } S(I) \vdash PS_p]$$

$$= \sum_{p \in Paths_g} \Pr_I [S(I) \vdash PS_p \mid I \vdash PI_p] \cdot \Pr_I [I \vdash PI_p]$$

$$\leq pr_{good} \cdot \sum_{p \in Paths_g} \Pr_I [I \vdash PI_p]$$

Translating from the algorithm $\mathcal{A}$ as a whole back to the just those paths in $\mathcal{T}_{\mathcal{A}}(I)$ requires understanding how the computation tree $\mathcal{T}_{\mathcal{A}}(I)$ changes for different instances I. Lemma 11 proves that if $p = \langle PI_p, PS_p, f_p \rangle$ is a path that algorithm $\mathcal{A}$ follows for some instance I' and I is consistent with what is learned in this path, then this same p will be followed when $\mathcal{A}$ is given instance I, namely if $p \in Paths$ and $I \vdash PI_p$, then $p \in \mathcal{T}_{\mathcal{A}}(I)$. This bounds the previous probability with the following.

$$\leq pr_{good} \cdot \sum_{p \in Paths} \Pr_I [p \in \mathcal{T}_{\mathcal{A}}(I)]$$

The width of $\mathcal{T}_{\mathcal{A}}(I)$ at level l is equal to the number of paths of length l in $\mathcal{T}_{\mathcal{A}}(I)$. This width is bounded by $W_{\mathcal{A}}(n)$. Hence, the above sum of probabilities is can be interpreted as follows.

$$= pr_{good} \cdot \mathrm{Exp}_I[\text{width of } \mathcal{T}_{\mathcal{A}}(I) \text{ at level } l] \ \leq \ pr_{good} \cdot W_{\mathcal{A}}(n)$$

Together this gives us $W_{\mathcal{A}}(n) \geq \frac{1}{2pr_{good}}$ as required. □

We now state and prove the lemma needed in the above proof that considers how the computation tree $\mathcal{T}_{\mathcal{A}}(I)$ changes for different instances I.

Lemma 11. *If $p = \langle PI_p, PS_p, f_p \rangle$ is a path that algorithm $\mathcal{A}$ follows for some instance I' and I is consistent with what is learned in this path, then this same p will be followed when $\mathcal{A}$ is given instance I, namely if $p \in Paths$ and $I \vdash PI_p$, then $p \in \mathcal{T}_{\mathcal{A}}(I)$.*

Proof. Suppose $p \in Paths$ and $I \vdash PI_p$. Recall $p \in Paths$ means that there exists an instance I' such that $p = \langle PI_p, PS_p, f_p \rangle$ is the label of a partial path of length l in $\mathcal{T}_{\mathcal{A}}(I')$. It follows that $I' \vdash PI_p$. This gives us everything needed for Lemma 1, namely we have two inputs I and I' that are both consistent with the partial instance PI_p, i.e. $I \vdash PI_p$ and $I' \vdash PI_p$ and there exists a path $p \in \mathcal{T}_{\mathcal{A}}(I')$ identified with $\langle PI_p, PS_p, f_p \rangle$. This lemma then gives that $p \in \mathcal{T}_{\mathcal{A}}(I)$. □

Our lower bound results follow from this technique and the four Lemmas specific to the problems at hand.

Proof. (Theorem 9) In all cases, the technique gives that $W_{\mathcal{A}}(n) \geq \frac{1}{2pr_{good}} \geq 2^{\Omega(l)}$. See Lemmas 13 and 17. What remains is to ensure that the probability pr_{bad} of a bad path is at most $\frac{1}{2|Forks|}$. Lemmas 16 states that for the Matrix Inversion Problem, $pr_{bad} \leq 2^{-\Omega(n)}$. In the model pBT, $|Forks_{pBT}| = 2^l$, giving $|Forks_{pBT}| \cdot pr_{bad} \leq \frac{1}{2}$ even when $l \in \Theta(n)$. In the pFBT model, however, $|Forks_{pFBT}| = [2W_{\mathcal{A}}(n)]^l = [2^{\Theta(l)}]^l = 2^{\Theta(l^2)}$, giving $|Forks_{pFBT}| \cdot pr_{bad} \leq \frac{1}{2}$ only when $l \in \Theta(\sqrt{n})$. Lemmas 12 states that for the Matrix Parity Problem $pr_{bad} \leq 2^{-\Omega(n^2)}$. Hence we have no problem setting $l \in \Theta(n)$ either way. If included in this problem are m free data items then what changes is that instead of just considering l levels of the computation, we consider the computation up to the point at which l real data items have been received and any number of free data items have been received. This involves up to $l + m$ levels in total. This increases $|Forks_{pFBT}|$ from $[2W_{\mathcal{A}}(n)]^l = 2^{\Theta(l^2)}$ to $[2W_{\mathcal{A}}(n)]^{l+m} = 2^{\Theta(l(l+m))}$. Hence, when $pr_{bad} \leq 2^{-\Omega(n)}$, $l \in \Theta\left(\frac{n}{m+\sqrt{n}}\right)$ is needed and when $pr_{bad} \leq 2^{-\Omega(n^2)}$, $l \in \Theta\left(\frac{n^2}{m+n}\right)$ is needed. □

5.2 The Matrix Parity Problem

This section proves the required good and bad path lemmas needed for the Matrix Parity Problem. Here is a reminder the relevant definitions.

- MP with distribution $\mathcal{P}$: The distribution $\mathcal{P}$ uniformly chooses a matrix $M \in \{0,1\}^{n \times n}$ for the input instance. The j^{th} data item contains the name x_j of a variable and the j^{th} column of the matrix. The output x_i, for each $i \in [n]$, is the parity of the i^{th} row, namely $x_i = \bigoplus_{j \in [n]} M_{\langle i,j \rangle}$.
- Height l: Define $l = \epsilon n$ to be the level to which the partial paths p are considered.
- Partitioning $\mathcal{D}$: For each $j \in [n]$, let $\mathcal{D}_j \subset \mathcal{D}$ denote the set of possible data items labeled by x_j and hence containing the j^{th} column of the matrix. Similarly, for a given path p, PI^{out} can be partitioned into $PI^{out}_j \subseteq \mathcal{D}_j$. Define $PI^{?}_j \subseteq \mathcal{D}_j$ to be the set of data items for which it remains unknown whether or not it is in the instance I. If $j \notin PI^{in}$ (i.e. PI^{in} does not contain a data item from $\mathcal{D}_j$), then $PI^{?}_j = \mathcal{D}_j - PI^{out}_j$. If $j \in PI^{in}$, then $PI^{?}_j = \emptyset$, because having received one data item from $\mathcal{D}_j$, the algorithm knows that no more from this domain are possible.
- Good Path: A computation path p and the set PI^{out} arising from it are considered to be *q-good* if $\exists j \notin PI^{in}$, $|PI^{?}_j| \geq q|\mathcal{D}_j|$, where q is set to be $2^{-(1-\varepsilon)l} = 2^{-(1-\varepsilon)\epsilon n}$. Note that this effectively means that at most $(1-\varepsilon)\epsilon n$ bits about the j^{th} column have been revealed.

The next lemma then bounds the probability that a computation path is *bad*. Consider a tuple of forking indices $f \in Forks$. Consider some algorithmic strategy $\mathcal{A}_f$ along these forking indexes halting after l levels.

Lemma 12. $pr_{bad} = \Pr_I [\mathcal{A}_f \textit{ is a } q\textit{-bad path}] \leq (3q)^{n-l}$.

This gives $pr_{bad} \leq 2^{-\frac{1}{2}\epsilon n^2}$ for the matrix parity problem as claimed.

Proof. The algorithmic strategy $\mathcal{A}_f$ follows one path p until l data items have been read. During each of these l steps, it chooses a permutation of the remaining data items and then learns which of the items in the randomly chosen input appears first in this ordering. The item revealed to be in the input is put into PI^{in} and all those appearing before it in the specified ordering are put into PI^{out}. We will, however, break this process into sub-steps, both in terms of defining the orders and in terms of delaying fully choosing the input until the information is needed. In each sub-step, the computation $\mathcal{A}_f$ specifies one data item D to be next in the current ordering. Then the conditioned distribution $\mathcal{P}$ randomly decides whether to put D into the input. Above we partitioned the data items into $\mathcal{D}_j \subset \mathcal{D}$ based on which variable x_j it is about. Though the algorithm deals with all such $\mathcal{D}_j$ in parallel, they really are independent. When taking a step in the j^{th} game, $\mathcal{A}_f$ selects one data item D from $PI^{?}_j = \mathcal{D}_j - PI^{out}_j$ to be next in the current ordering. Given $\mathcal{A}_f$ knows that exactly one of the data items in this set are in the input and they are equally likely to be, $\mathcal{P}$ decides that D is in the input with probability $1/|PI^{?}_j|$. If it is, then this j^{th} game is done. Otherwise, D is removed from $PI^{?}_j$. Given the statement of the lemma being proved bounds the probability that this path p followed is *bad*, we need to assume that the algorithm $\mathcal{A}_f$ is doing everything in its power to make this the case. Hence, we say that $\mathcal{A}_f$ lose this *game* if this path is good, i.e. if after PI^{in} contains l data items it is the case that $\exists j \notin PI^{in}$, $|PI^{?}_j| \geq q|\mathcal{D}_j|$. In order to make the game less confusing and to give $\mathcal{A}_f$ more chance of winning, we will not stop the game when PI^{in} contains l data items, but instead we will let him play all n games to completion. He wins the j^{th} game if he can manage to shrink $PI^{?}_j$ to be smaller than $q|\mathcal{D}_j|$ without adding j to PI^{in}. If he accidentally add j to PI^{in}, then he loses this j^{th} game. It is ok for him to lose l of these n games, because, during the part of his computation we consider, we allow PI^{in} to grow to be of size

l. However, if he loses $l+1$ of the games, then we claim that this computation path is *good*. At least one of these lost games had its j added to PI^{in} after we stopped considering his computation and at the point in time $j \notin PI^{in}$ and $|PI^?_j| \geq q|\mathcal{D}_j|$. Now let us bound the probability that he is able to win the j^{th} game. Effectively what he is able to do in this game is to specify the one order in which it will ask about the data items in $\mathcal{D}_j$. The algorithm wins this game if and only if the data item D_j from $\mathcal{D}_j$ that is in the input lays in the last q fraction of this ordering. Because D_j is selected uniformly at random, the probability of the algorithm winning this game is q. For $j \in [n]$, let X_j be the random variable indicating whether $\mathcal{A}_f$ wins the j^{th} game and let $X = \sum_{j\in[n]} X_j$ denote the number of games won. We proved that $\Pr[X_j = 1] = q$. We also proved that $\Pr[\text{path is bad}] \leq \Pr[X \geq n - l]$. Define $\mu = \text{Exp}[X] = qn$ and $1 + \delta = \frac{1}{q}(1 - \frac{l}{n})$ so that $(1+\delta)\mu = n - l$. Chernoff bounds gives that $\Pr[X \geq (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq (3q)^{n-l}$. □

We now prove that the partial information about the input I gained from a good $PI = \langle PI^{in}, PI^{out}\rangle$ is not sufficient to effectively make irrevocable decisions PS about the data items in PI^{in}. This is the only result in the lower bounds section that depends on anything specific about the computation problem and it does not depend on the specifics of the model of computation.

Lemma 13. *If $\langle PI, PS, f\rangle$ is the label of a q-good path p of length l, then $pr_{good} = \Pr_I[S(I) \vdash PS \mid I \vdash PI] \leq \frac{1}{q}2^{-l}$.*

Here $q = 2^{-(1-\varepsilon)l}$, giving $pr_{good} \leq 2^{-\varepsilon l}$ as needed.

Proof. Assume the data items in PI^{in} have told the algorithm the first l columns of M, but required him to make irrevocable guesses PS of the parities $x_1, \ldots, x_l$ of the first l rows of M. PI^{out} being q-good means that $\exists j \notin PI^{in}$, $|PI^?_j| \geq q|\mathcal{D}_j|$, where $D_j \in PI^?_j \subseteq \mathcal{D}_j = \{0,1\}^n$ is a vector that remains possible for the j^{th} column of M. In order to be nice to the algorithm, we will give the algorithm all of M except for this column j. Then for each row $i \in [1,l]$, we now know the parity $\bigoplus_{j'\neq j} M_{\langle i,j\rangle}$ of these columns and know the parity $x_i = \bigoplus_{j\in[n]} M_{\langle i,j\rangle}$ that the algorithm has promised for the entire row. Hence, we can compute the required value for $M_{\langle i,j\rangle}$ in order for the algorithm to be correct. Let A be the set of vectors α for the j^{th} column such that the algorithm is correct, namely $A = \{\alpha \in \{0,1\}^n \mid \forall i \in [1,l], \alpha_i = \text{ the required } M_{\langle i,j\rangle}\}$. Note that $|A| = 2^{n-l}$ because there is such an α for each way of setting the remaining $n - l$ bits of this column. The adversary now randomly selects $\alpha \in PI^?_j$ that will be the j^{th} column in the input M. If the selected α is in A, then the algorithm wins. The probability of this, however, is $|A|/|PI^?_j| \leq \left[2^{n-l}\right] / \left[q2^n\right] = \frac{1}{q}2^{-l}$. □

This completes the lower bound for the Matrix Parity Problem.

5.3 The Matrix Inversion Problem

This section proves the required good and bad path lemmas needed for the Matrix Inversion Problem. Here is a reminder the relevant definitions.

- Fixed Matrix M: The problem MI is defined by a fixed matrix $M \in \{0,1\}^{n\times n}$ that is non-singular, has exactly seven ones in each row, at most K ones in each column, and is a $(r,7,3)$-*boundary expander*, with $K \in \Theta(1)$, and $r \in \Theta(n)$.
- Boundary Expander: M is a $(r,7,c)$-*boundary expander* if it has exactly seven ones in each row and if for every set $R \subseteq [n]$ of at most r rows, the *boundary* $\partial_M(R)$ has size at least $c|R|$. The boundary is defined to contain the column j if there is a single one in the rows R of this column. Section 5.4 sets $K = \Theta(1)$, $c = 3$, and $r = \Theta(n)$ and proves that such a matrix exits.

- MI with distribution $\mathcal{P}$: The distribution $\mathcal{P}$ uniformly chooses a vector $b \in \{0,1\}^n$. The j^{th} data item contains the name x_j of a variable and the value of the at most K bits b_i from b involving this variable, i.e. those for which $M_{\langle i,j \rangle} = 1$. The output is $x \in \{0,1\}^n$ such that $Mx = b$ over GF-2.
- Height l: Define l to be the level to which the partial paths p are considered. If we are proving the $2^{\Omega(n)}$ width lower bound for the pBT model, then set $l = \frac{n}{2^{K+3}} = \Theta(n)$. If we are proving $2^{\Omega(\sqrt{n})}$ for the pFBT, then set $l = \Theta(\sqrt{n})$.
- Good Path: A computation path p and the set PI^{out} arising from it are considered to be Q-*good* if $|PI^{out}| \leq Q$. Here $Q = r - Kl = \Theta(n)$, which is reasonable given that the number of possible data items is $|\mathcal{D}| = 2^K n = \Theta(n)$.

At each point along the computational path p, the algorithmic strategy $\mathcal{A}_f$ knows that the input is consistent with the current partial instance PI. Knowing what the algorithm learns from this is complex. Hence, we will change the game by having the adversary reveal extra information to the algorithms so that at each point in his computation, what he knows is only some r' bits of the input b. If the path is *good*, then this number does not exceed r, i.e. the first parameter of expansion.

Lemma 14. *Let $PI = \langle PI^{in}, PI^{out} \rangle$ be arbitrary sets of data items. $I \vdash PI$ reveals at most $r' = K \cdot |PI^{in}| + |PI^{out}|$ bits of b. More formally, let $B = \{b \in \{0,1\}^n \mid I \vdash PI\}$ be the set of instances $b \equiv I$ consistent with the partial instance PI. The adversary can reveal more information to the algorithm, restricting this set to $B' \subseteq B$, so that the vectors in B' have fixed values in r' of its bits and all $2^{n-r'}$ possibilities in the remaining $n - r'$ bits. Note that when PI^{out} is good, the algorithm learns at most $K \cdot |PI^{in}| + |PI^{out}| \leq K \cdot l + (r - Kl) = r$ bits of b.*

Proof. When the algorithm learns that data item D is in PI^{in}, he learns learns the value of at the most K bits of $b \in \{0,1\}^n$ corresponding to the j^{th} column of M. When he learns that D is in PI^{out}, he learns a setting in $\{0,1\}^k$ that $k \leq K$ of bits in b do not have. The adversary chooses one of the k bits of b that the data item is about and that the algorithm does not yet know the value of and reveals that this bit of b has the opposite value from that given in the data item. The algorithm, seeing this as the reason the data item is not in the input, learns nothing more about the input. In the end, what the algorithm learns from $I \vdash PI$ is $K \cdot |PI^{in}| + |PI^{out}|$ bits of b. □

Lemma 15. *Let $PI = \langle PI^{in}, PI^{out} \rangle$ denotes the current state computation. Let D be a data item that is still possibly in the input I. It follows that $\Pr_I [D \in I \mid I \vdash PI] \geq q = \frac{1}{2^K}$.*

Proof. The proof of Lemma 14 changes the game so that what the algorithm knows at each step of the computation is some r' of bits of the vector $b \in \{0,1\}^n$. The conditional distribution $\mathcal{P}$ uniformly at random chooses the remaining $n - r'$ bits. Each data item D specifies a variable x_j and up to K bits of the vector b. If this D disagrees with some of the r' revealed bits of b, then D in not still possible. Otherwise, the probability that D is in the input is $\frac{1}{2^{K'}} \geq \frac{1}{2^K}$, where $K' \leq K$ is the number of the specified bits that have not been fixed. □

The next lemma then bounds the probability that the computation path given by $\mathcal{A}_f$ is *bad*.

Lemma 16. $pr_{bad} = \Pr_I [\mathcal{A}_f \text{ is a } Q\text{-bad path}] \leq 2^{-\Omega(n)}$.

Proof. Generally, we consider $\mathcal{A}_f$'s computation until it receives l data items. However, instead, let us consider it for Q sub-steps as done in Lemma 12. In each of these sub-steps, the algorithm specifies one data item D from those that are still possible and is told whether or not D is in the input. If it is then D is added to PI^{in} and if not it is added to PI^{out}. For each $t \in [Q]$, let X_t be the random variable indicating

whether the t^{th} such data item is added to PI^{in}. By Lemma 15, $\Pr[X_t = 1] \geq q$. Note that $X = \sum_{t \in [Q]} X_t$ denotes the resulting size of PI^{in}. If $X \geq l$, then by this point in the computation $|PI^{in}| \geq l$, so the part of the computation we normally consider has already stopped, but $|PI^{out}| = Q - |PI^{in}| \leq Q$ and hence this computation is considered to be good. It follows that $\Pr[\text{path is bad}] \leq \Pr[X < l]$. Define $\mu = \text{Exp}[X] = qQ$ and $\delta = 1 - \frac{l}{qQ}$ so that $(1-\delta)\mu = l$. Chernoff bounds gives that $\Pr[X < (1-\delta)\mu] < e^{-\mu\delta^2/2} = e^{-qQ(1-\frac{l}{qQ})^2/2}$. We defined $l \leq \frac{r}{2^{K+3}}$, $Q = r - Kl$, $q = \frac{1}{2^K}$, giving that $\frac{qQ}{l} = \frac{qr}{l} - qK \geq 8 - 1$ and that $\frac{l}{qQ} \leq \frac{1}{2}$. This gives $pr_{bad} \leq e^{-\frac{Q}{2^{K+3}}} \leq 2^{-\Omega(n)}$. □

Lemma 14 proves that for good paths, $I \vdash PI$ reveals at most r bits of b. We now prove that this is not sufficient to effectively make irrevocable decisions PS about the data items in PI^{in}.

Lemma 17. *If p is a good path, then* $\Pr_I[S(I) \vdash PS \mid I \vdash PI] = pr_{good} \leq 2^{-\Omega(l)}$.

Proof. Our probability distribution $\mathcal{P}$ on inputs I is uniform on the settings of b. Because there is a bijection mapping $x = M^{-1}b$ between the settings of b and the settings of x, it is equivalent to instead consider the uniform distribution on setting of x. Let $R \subseteq [n]$ denote the indexes i of the at most r bits of b that have been fixed by Lemma 14 from $I \vdash PI$ and let $\bar{b}_R \in \{0,1\}^{|R|}$ be the values that these $|R|$ bits have been fixed to. Let M_R denote the corresponding rows of the matrix M. These rows impose the requirement $M_R \cdot x = \bar{b}_R$. An interesting effect of translating from a distribution on b to one on x is that the equations $M_{([n]-R)} \cdot x = b_{([n]-R)}$ no longer matter. In addition, PS commits the algorithm to the values of exactly l variables. Let $Fixed \subseteq [n]$ denote the indexes j of these variables x_j and let $\bar{x}_{Fixed}$ denote the values committed to them. To conclude,

$$\begin{aligned} pr_{good} &= \Pr_I[S(I) \vdash PS \mid I \vdash PI] = \Pr_{x \in_u \{0,1\}^n}\left[x_{fixed} = \bar{x}_{fixed} \mid M_R \cdot x = \bar{b}_R\right] \\ &= \frac{|\{x \mid x_{fixed} = \bar{x}_{fixed} \text{ and } M_R \cdot x = \bar{b}_R\}|}{|\{x \mid M_R \cdot x = \bar{b}_R\}|} \end{aligned}$$

What we now need to compute is both the number of solutions x to the subsystem of equations $M_R \cdot x = \bar{b}_R$ and the number of these consistent with $x_{fixed} = \bar{x}_{fixed}$. Towards this goal, we will run the following Gaussian-like elimination algorithm. It selects, satisfies, and eliminates equations $i \in R$ one at a time by setting and eliminating a variable x_j. We easily establish the loop invariant by setting $R_0 = R$ and $C_0 = [n]$.

GAUSSIAN-LIKE ELIMINATION
$R_0 \leftarrow R$ and $C_0 \leftarrow [n]$
while $(c|R_t| > l)$
  % Loop Invariant:
    LI1: We have a set $R_t \subseteq R$ of rows and a set $C_t \subseteq [n]$ of columns with $Fixed \subseteq C_t$.
      Let $M_{\langle R_t, C_t\rangle}$ denote the submatrix with the selected rows and columns.
      Let x_{C_t} denote the vector of selected variables.
    LI2: $pr_{good} = \frac{|\{x_{C_t} \mid x_{fixed}=\bar{x}_{fixed} \text{ and } M_{\langle R_t,C_t\rangle} \cdot x_{C_t} = \bar{b}_{R_t}\}|}{|\{x_{C_t} \mid M_{\langle R_t,C_t\rangle} \cdot x_{C_t} = \bar{b}_{R_t}\}|}$.
    LI3: $M_{\langle R_t,C_t\rangle}$ is a $(r, 7, 3)$-*boundary expander*.
  1. Pick $x_j \in \partial(R_t) - Fixed$
  2. Let $i \in R_t$ be the row in which x_j appears.
  3. $x_j \leftarrow$ value that satisfies $M_i \cdot x = \bar{b}_i$
  4. $R_{t+1} \leftarrow R_t - i$ and $C_{t+1} \leftarrow C_t - j$
  5. $t = t + 1$

We prove as follows that this code maintains the loop invariant. Assume by way of induction that it is true for t. We now prove that it is true for $t{+}1$. By the definition of $M_{\langle R_t, C_t\rangle}$ being a $(r, 7, 3)$-*boundary expander* and from the fact that $|R_t| \leq |R| \leq r$, we know that the *boundary* $\partial_M(R_t) \subseteq C_t$ has size at least $c|R_t|$, which by not having exited more is than l. Hence, line 1 is able to pick a variable $x_j \in \partial(R_t) - Fixed$. The boundary $\partial_M(R_t)$ is defined to contain the column j if there is a single one in the rows R_t of this column. Let $i \in R_t$ be the row in which x_j appears. The equations $M_{\langle R_t - i, C_t\rangle} \cdot x_{C_t} = \bar{b}_{(R_t - i)}$ do not use this variable x_j and hence are not effected by how it is set. The equation $M_{\langle i, C_t\rangle} \cdot x_{C_t} = \bar{b}_i$ can be solved and satisfied by setting the value of x_j to be this linear combination of the variables in $R_t - i$. Being satisfied, the equation i is removed from $R_{t+1} = R_t - i$. Not effecting the remaining equations, the variable x_j can be removed from them, $C_{t+1} = C_t - j$. Being set, the variable x_j adds no degrees of freedom and hence does not effect the size of either $\{x_{C_t} \mid x_{fixed} = \bar{x}_{fixed}$ and $M_{\langle R_t, C_t\rangle} \cdot x_{C_t} = \bar{b}_{R_t}\}$ or of $\{x_{C_t} \mid M_{\langle R_t, C_t\rangle} \cdot x_{C_t} = \bar{b}_{R_t}\}$. Hence, the ratio of their sizes remains being equal to pr_{good}, maintaining LI2. Deleting row i from $M_{\langle R_t, C_t\rangle}$ does not effect it being a $(r, 7, 3)$-*boundary expander* because the definition considers every subset of rows. Neither does deleting column j because once row i has been deleted, this j^{th} column is all zeros. This proves that all the loop invariants have been satisfied. The Gaussian-like elimination algorithm terminates when $c|R_t| \leq l$. The size of $\{x_{C_t} \mid x_{fixed} = \bar{x}_{fixed}$ and $M_{\langle R_t, C_t\rangle} \cdot x_{C_t} = \bar{b}_{R_t}\}$ is at most $2^{|C_t| - l}$ because C_t is the set of remaining variables and $x_{fixed} = \bar{x}_{fixed}$ fixes the value of l of them. The size of $\{x_{C_t} \mid M_{\langle R_t, C_t\rangle} \cdot x_{C_t} = \bar{b}_{R_t}\}$ is at least $2^{|C_t| - l/3}$ because only $|R_t| \leq \frac{l}{3}$ equations restrict the values of the $|C_t|$ variables. LI2 then gives that $pr_{good} \leq \frac{2^{|C_t| - l}}{2^{|C_t| - l/3}} = 2^{2/3\, l}$, completing the proof. □

5.4 Boundary Expander Matrix

Here we need to prove that a square matrix $A \in \{0,1\}^{n \times n}$ exists such that

1. Each row has exactly 7 ones.
2. Each column has at most $K = 7\Delta$ ones, for some constant $\Delta \geq 1$.
3. A is of full rank and hence is non-singular.
4. A is a $(r, 7, 3)$ boundary expander with $r \in \theta(n)$. This means that for any set I of at most r rows, the boundary $|\partial_A(I)|$ has size is at least $3|I|$. The boundary is defined to contain column j if there is a single one in the rows I of this column.

First we define a distribution on $\Delta n \times n$ binary matrices A, with $\Delta n \geq n$ rows, such that each row has exactly 7 ones and each column exactly 7Δ ones. Then we show that the probability that A is not a good boundary expander is less than $\frac{1}{2}$. Next we argue that the probability that A is not full rank is also less than $\frac{1}{2}$. Hence, there must exist such an A that is both a good boundary expander and of full rank. Then we are done because any n linearly independent rows of that matrix will inherit the expansion of the big matrix, is non-singular, and the number of ones in each column will be at most 7Δ.

The $\Delta n \times n$ binary matrix is randomly chosen by randomly choosing a perfect matching in the following bipartite graph. On the left side we have Δn equations, each with 7 slots (for *distinct* variables). On the right side we have n variables, each with 7Δ slots. An edge from a slot of equation i to a slot of variable j puts a one at matrix entry $A[i, j]$. The graph being a perfect matching ensures that there are exactly 7 ones in each row and exactly 7Δ ones in each column. This perfect matching is chosen randomly by filling one at a time each slot on the left by randomly picking an unmatched slot on the right and match them.

Lemma 18. *For any sufficiently large n and constant Δ, the probability that matrix $A \in \{0,1\}^{\Delta n \times n}$ is not an $(r, 7, 3)$ boundary expander is less than $\frac{1}{2}$.*

Proof. Let I denote an arbitrary set of rows I, $|I| = t \leq r$, defining t equations. Each such equation randomly chooses 7 of the variable slots for a total of $7t$. Suppose that $0 \leq i \leq 7t$ of these variables slots have been chosen already. Let B denote the current number of boundary variables, i.e. the number for which exactly one of its slots have been selected. Similarly, let D denote the number that have duplicate slots filled. This leaves $n - B - D$ variables with none of its slots filled. We will watch these values dynamically as more slots are chosen. We need to bound the probability of the bad event that after $i = 7t$ such steps $B < 3t$.

We bound the probability that the next step will increase B by one to be

$$p_+ = \frac{7\Delta(n - B - D)}{7\Delta n - i} \geq 1 - \frac{B + D}{n}$$

This is because there are $n - B - D$ variables with all 7Δ of there slots empty and filling one of these will make such a variable a boundary variable. Here $7\Delta n - i$ is the total number of slots currently unfilled.

We will now prove that we may assume that at all times $B + D \leq 5t$. Other than B increasing, the other two options are as follows. If the next slot chosen is that of a boundary variable, B decreases and D increases, keeping $B + D$ constant. If it is of a duplicate variable neither changes. On the other hand, if B ever increases during more than $5t$ of the steps, then in the remaining $7t - 5t = 2t$ steps, B be unable to decrease below $3t$ as required for A to fail.

We simplify this process by assuming that each step B either increases or decreases by one each step with the probability of decrease being $p_- = 1 - p_+ \leq \frac{B+D}{n} \leq \frac{5t}{n}$. The probability that this random walk of length $7t$ ends with $B < 3t$ then is bounded by the probability that this decrease occurs at least $2t$ times. We will set $t \leq r \leq \frac{1}{35}n$, so that the expected number of decreases is at most t, giving that it is unlikely to have $2t$ and exponentially less likely to have more. This gives

$$p_{tbad} \leq \sum_{k \geq 2t}^{7t} \binom{7t}{k} {p_-}^k (1 - p_-)^{7t-k} \leq \sum_{i \geq 0} 2^{-i} \times \text{max term} \leq 2\binom{7t}{2t} {p_-}^{2t}.$$

We then do the union bound to bound the probability that there exists a set of size $t \leq r$ whose boundary is at most $3t$.

$$p = \sum_{t=1}^{r} \binom{\Delta n}{t} p_{tbad} \leq \sum_{t=1}^{r} \left(\frac{e\Delta n}{t}\right)^t (2) \left(\frac{7e}{2}\right)^{2t} \left(\frac{5t}{n}\right)^{2t} \leq \sum_{t=1}^{r} 2 \left(\frac{6152\Delta t}{n}\right)^t \leq \sum_{t=1}^{r} 2 \left(\frac{1}{5}\right)^t < \frac{1}{2}$$

Here we ensure $\frac{6152\Delta t}{n} \leq \frac{1}{5}$ by setting $r = \frac{n}{31{,}000\Delta} \in \Theta(n)$. In this case, such an expander exists. □

Lemma 19. *For any sufficiently large n and constant Δ, the probability that matrix $A \in \{0,1\}^{\Delta n \times n}$ does not have full rank is less than $\frac{1}{2}$.*

Proof. Suppose A is not full rank. Then there exits a new equation that is not in the span of the Δn equations given by the rows of A. Let $F \subseteq [n]$ be the subset of variables such that this new equation is $\sum_{i \in F} x_i = 0$. This F does in fact witness the fact that A is not full rank iff each row of A, viewed as the values of the n variables, satisfies this equation. When $|F| \leq r' \in \Theta(n)$, we will show that F likely fails to be such a witness because there is a row of A that contains exactly one variable from F and hence $\sum_{i \in F} x_i = 1$ for this row. In contrast, when $|F| > r'$, we will show that F likely fails because there is a row that contains exactly seven variables from F and hence when doing Boolean algebra we also get that $\sum_{i \in F} x_i = 1$. If all such F fail to be a witness then A is full rank.

The amusing thing is that the case with small F is exactly the same boundary expansion property as proved in Lemma 18 and hence the same proof with slightly different numbers works. The only difference is that we switch the roles of rows and the columns and the boundary does not need to be of size $3t$ but only of size one. Repeating the statement is that for any set F of at most r' columns, the boundary $|\partial_A(F)|$ has

size is at least 1. The boundary is defined to contain row j if there is exactly one one in the columns F of this row. The required changes in the numbers are as follows.

$$p_+ = \frac{7(\Delta n - B - D)}{7\Delta n - i} \geq 1 - \frac{B+D}{\Delta n} \geq 1 - \frac{3.5\Delta t}{\Delta n} = 1 - \frac{3.5t}{n} = 1 - p_-$$

$$p_{tbad} \leq \sum_{k \geq 3.5\Delta t} \binom{7\Delta t}{k} {p_-}^k (1-p_-)^{7\Delta t - k} \leq 2\binom{7\Delta t}{3.5\Delta t} {p_-}^{3.5\Delta t}.$$

$$p = \sum_{t=1}^{r'} \binom{n}{t} p_{tbad} \leq \sum_{t=1}^{r'} \left(\frac{en}{t}\right)^t 2^{7\Delta t} \left(\frac{3.5t}{n}\right)^{3.5\Delta t} \leq \sum_{t=1}^{r'} \left(\frac{20t}{n}\right)^{3\Delta t}$$

This is strictly less than $\frac{1}{4}$ as long as $\frac{20t}{n} \leq \frac{1}{8}$, which we can assure by setting $r' = \frac{n}{160} \in \Theta(n)$. In this case, all F smaller than r' fail to be witnesses that A is not full rank.

We now switch to proving that for $|F| > r' = \frac{n}{c}$, F likely fails to be a witness because there is a row (equation) such that all seven of its variables are from F. Consider one such row. When randomly selecting a variable for this row, the probability that it is in F is $\frac{1}{c}$. The probability that all seven variables fall in F is $\frac{1}{c^7}$. The probability that this fails to happen for all Δn rows is $\left(1 - \frac{1}{c^7}\right)^{\Delta n}$. The number of different sets $F \subseteq [n]$ is at most 2^n. Hence, probability that there exists a large witness F is at most $2^n \cdot \left(1 - \frac{1}{c^7}\right)^{\Delta n} < \frac{1}{4}$ for any $\Delta \geq c^7$. □

By Lemmas 18 and 19 for reasonably large n there exists a non-singular binary matrix which is $(r, 7, 3)$ boundary expander in which each column sums up to at most a constant 7Δ.

6 Conclusions

We define the *prioritized Free Branching Tree* (pFBT) model extending the *Prioritized Branching Trees* (pBT) model use to model recursive back-tracking and a large sub-class of dynamic programming algorithms. We give some surprising upper bounds and matching lower bounds.

7 Bug

The definition (5.4) of the Boundary Expander graphs (and the resulting problems) clearly say "Each row has exactly 7 ones." But in the constuction it may have fewer.

Lets bound prob that EVERY row has EXACTLY 7 ones. The bug occurs when one of the Δn equations has two of its 7 slots be matched to two of the 7Δ slots for the SAME one of the n variables,

There are $\binom{7}{2}\Delta n$ such pairs of equation slots. The probability that this slot does not hit the same variable is $(1 - \frac{1}{n})$. The probability that things are good because none of these pairs collide is $(1 - \frac{1}{n})^{35\Delta n} = e^{-35\Delta}$. This is awfully small.

The probability diestribution is defined as "This perfect matching is chosen randomly by filling one at a time each slot on the left by randomly picking an unmatched slot on the right and match them."

If we condition this probability on not having the same variable appear twice in the same equation:

1. There there are more ones in the matrix.

2. I claim the prob that it A is an expander could go up. Fix some column j and some small set of rows. "The boundary is defined to contain column j if there is a single one in the rows I of this column." This column j could fail this condition because it has no ones in I or because it has more than one.

The first reason is much more likely. Our new conditioning of having ones in the matrix improves this probability.

3. I assume that the probability that A is full rank also goes up. ??

References

[AB04] Spyros Angelopoulos and Allan Borodin. On the power of priority algorithms for facility location and set cover. *Algorithmica*, 40(4):pp.271–291, 2004.

[ABBO+11] Michael Alekhnovich, Allan Borodin, Joshua Buresh-Oppenheim, Russell Impagliazzo, Avner Magen, and Toniann Pitassi. Toward a model for backtracking and dynamic programming. *Computational Complexity*, 20(4):pp. 679–740, 2011.

[AHI05] Michael Alekhnovich, Edward A. Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *J. Autom. Reasoning*, 35(1-3):51–72, 2005.

[AJPU07] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory of Computing*, 3(1):81–102, 2007.

[AS92] Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 1992.

[BBLM10] Allan Borodin, Joan Boyar, Kim S. Larsen, and Nazanin Mirmohammadi. Priority algorithms for graph optimization problems. *Theor. Comput. Sci.*, 411(1):239–258, 2010.

[BCM11] Allan Borodin, David Cashman, and Avner Magen. How well can primal-dual and local-ratio algorithms perform? *ACM Trans. Algorithms*, 7(3):29, 2011.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[BNR03] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (Incremental) Priority Algorithms. *Algorithmica*, 37(4):pp.295–326, 2003.

[BODI11] Joshua Buresh-Oppenheim, Sashka Davis, and Russell Impagliazzo. A stronger model of dynamic programming algorithms. *Algorithmica*, 60(4):938–968, 2011.

[Bol01] Bela Bollobas. *Random Graphs.* Cambridge University Press, 2001.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[DI09] Sashka Davis and Russell Impagliazzo. Models of greedy algorithms for graph problems. *Algorithmica*, 54(3):269–317, 2009.

[FW98] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of boundded random variables. *J. of Amer. Statist. Assoc.*, 58(301):13–30, 1963.

[IKR85] Oscar H. Ibarra, Sam M. Kim, and Louis E. Rosier. Some characterizations of multihead finite automata. *Information and Control*, 67(1-3):114–125, 1985.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addisn Wesley, 2005.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, 2005.

[MV04] Jiri Matousek and Jan Vondrak. *The Probabilistic Method, Lecture Notes*. Manuscript, 2004.

[PPZ99] Ramamohan Paturi, Pavel Pudlák, and Francis Zane. Satisfiability coding lemma. *Chicago J. Theor. Comput. Sci.*, 1999, 1999.

[RSZ02] Alexander Russell, Michael E. Saks, and David Zuckerman. Lower bounds for leader election and collective coin-flipping in the perfect information model. *SIAM J. Comput.*, 31(6):1645–1662, 2002.

[Woe01a] Gerhard J. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization*, pages 185–208, 2001.

[Woe01b] Gerhard J. Woeginger. When does a dynamic programming formulation guarantee the existence of an fptas? *Electronic Colloquium on Computational Complexity (ECCC)*, (084), 2001.