CS5330 Randomized Algorithms: RP1 - Randomized Decision Trees

Main Content: Pages 6 - 12

Kiran Gopinathan, Jishnu Mohan

1 Introduction

A pertinent issue when dealing with randomised algorithms is the question of whether the use of randomisation improves the runtime of an algorithm (if at all) and by how much? and what if bounded errors are allowed? Unfortunately, it turns out that answering such questions in a general model of computation is typically difficult as it is challenging to establish strong lower bounds on runtimes - finding non-trivial lower bounds for several standard algorithmic problems still remain as open questions. Spurred by this fact, some researchers have instead considered assessing randomisation within the context of other weaker computational models, such as query complexity, where lower bounds are easier to find. Query complexity is a form of complexity analysis that considers a simplified model of computation with algorithms represented entirely by binary decision trees, and asks about the number of bits of an input that an algorithm will have to read to compute a function. In 1986, Saks et al.[1] proved a tight lower bound of $O(D(f)^{0.753\ldots})$ on the query complexity of a randomised algorithm for computing NAND formulae (where $D(f)$ is the complexity of an optimal deterministic algorithm computing the same value), and conjectured that this separation is the best possible query complexity improvement over all boolean functions that one can obtain by introducing randomisation. Recent breakthroughs[2][3] in this area have disproven this bound by demonstrating a specific family of functions for which there exist randomized algorithms that produce even greater improvements over optimal deterministic strategies, up to the theoretical maximum of $O(\sqrt{D(f)})$.

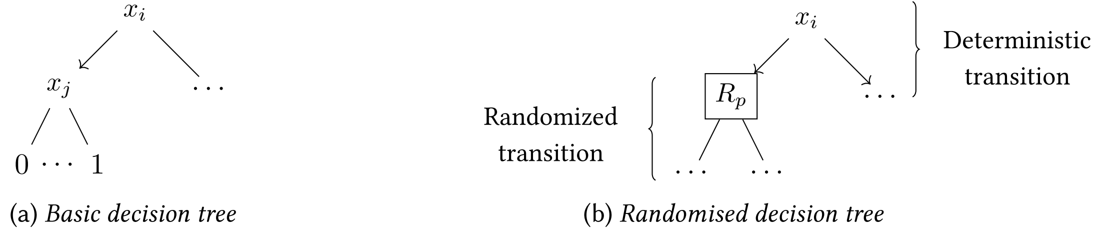
In this report, we provide a survey of the main developments into this question, providing a carefully curated walk through each of the key proofs and algorithms. Finally, we conclude by investigating how these techniques can be applied to one of the remaining open problems in this area - that of whether there exists a function exhibiting the optimal cubic separation between bounded-error query complexity $R(f)$ and deterministic query complexity $D(f)$. The remainder of this report will adopt the following structure: Section 2 provides an overview of query complexity research, presenting the key definitions and covering a few standard properties, as well as discussing Saks *et al.*'s original hypothesis[1]. We end this section with an overview of the best known bounds between deterministic and randomised query complexity. In Section 3, we introduce the specific family of functions that will be used in the subsequent analysis - the Göös-Pitassi-Watson (GPW) function [4] - proving a few of its basic properties and providing intuition for why it is a particularly suitable candidate for randomised solutions. Following this, Section 4 presents an algorithm devised by Mukhopadhyay *et al.* [2] and proves that it calculates the GPW function with bounded error (Monte Carlo) in $O(\sqrt{D(f)})$ queries. Then, Section 5 discusses how GPW can be modified to increase the opportunities for randomisation, presenting algorithms by Ambainis *et al.* [3] which use modifications to the GPW function to obtain even greater optimal separations. Finally, Section 6 concludes the report, providing a summary of the various bounds, discussing how these techniques can be extended to attack the open problem of whether the lower bound of a cubic separation between bounded-error query complexity $R(f)$ and deterministic query complexity $D(f)$ is tight.

2 Randomized Query Complexity

In this section we introduce the reader to the field of query complexity, beginning with a review of the basic definitions and notations of query complexity, and also presenting a few of the important standard properties. The section then moves to describe the context and motivation behind Saks *et al.*'s conjecture, and discusses its importance and implications. Finally, the section ends with an overview of the current state of the field, discussing the best known separations between deterministic and randomised query complexity.

2.1 Definitions and Standard Properties

Query complexity proposes a measure of computational complexity wherein the cost of an algorithm is not measured by the number of steps it takes, but rather by the number of bits of the input it reads (queries) in order to



(a) *Basic decision tree*

(b) *Randomised decision tree*

Fig. 1: *Query complexity computational model*

carry out its computation. More formally, problems in query complexity involve computing some boolean function $f : \{0,1\}^n \to \{0,1\}$, and the complexity of algorithms calculating this function are measured by the number of bits of the input that they have to read in the worst case.

Computational model It turns out then, that when analysing algorithms under this particular restrictive cost model, all algorithms can be entirely encoded as some variant of a binary decision tree, where each node of the tree branches based on some bit of the input. More formally, the exact formulation of the decision tree slightly varies depending on the class of algorithm - deterministic or randomised:

- **Deterministic Trees** - In the case of a deterministic algorithm, the algorithm can be completely encoded by a decision tree like the one presented in Figure 1a. Each interior node of the tree is labelled with an index i into the input, and the leaf nodes are annotated with either 1 or 0. Then, to execute this tree on some given input, we start from the root, and then while we are on an interior node, we read the i-th bit of input as specified by the node and recursively transition either left or right depending on its value. Once a leaf is reached, we terminate and output the value attached to the leaf.
- **Randomised Trees** - In the case of randomised algorithms, there are two equivalent encodings in terms of decision trees. The first approach is presented in Figure 1b, wherein a randomised algorithm is encoded as decision tree with an additional randomised type of node (represented in the diagram with R_p) which is evaluated by sampling from a Bernoulli random variable of some parameter p, and then taking the branch corresponding to the outcome. The other approach of encoding randomised algorithms is to consider them as distributions over the space of all possible deterministic decision trees. It is fairly easy to show that these two encodings are equivalent as we can always convert one to the other. We will use this fact as a given and freely transition between the two formulations as required in our proofs.

Tree complexity Due to the relatively simple structure of these computational models, it becomes fairly easy to measure the query complexity, as this simply resolves to some expression of the maximum depth of the corresponding binary tree. Once again, there are some slight differences depending on whether the algorithm is deterministic or randomised:

- **Deterministic Complexity** - The deterministic complexity of a boolean function f, denoted by $D(f)$, is defined as the maximum number of queries $\mathbb{Q}$ made by an optimal algorithm A_D in the worst case.

  Definition 2.1. The **deterministic complexity** $D(f)$ of a boolean function f is given by:

  $$D(f) = \min_{A_D} \max_x \mathbb{Q}[A_D(x)]$$

  where A_D is drawn from the set of deterministic algorithms computing f.

  By our earlier reasoning, if we represent the optimal algorithm as a decision tree, then this simply becomes the maximum depth of the corresponding tree.
  One other measure of query complexity that also shows up in the literature is the non-deterministic complexity $N(f)$:

  Definition 2.2. The **non-deterministic complexity** $N(f)$ of a boolean function f is given by:

  $$N(f) = \min_{A_{NF}} \max_x \mathbb{Q}[A_{NF}(x)]$$

  where A_{NF} is over the set of non-deterministic algorithms computing f.

  It is also easy to show that this measure of query complexity corresponds to the maximum degree of any clause in the disjunctive normal form of the logical representation of the function f - for this reason, it is also sometimes referred to as $\deg(f)$.

- **Randomised/Quantum Complexity** - In the case of randomised algorithms, there is a wider variety of complexity measures, depending on whether we wish to allow for errors or not.
  At it's most simplest, the zero-error randomised complexity of a function f, denoted by $R_0(f)$, is the maximum number of queries $\mathbb{Q}$ on any input for any choice of randomness Q for the optimal randomised algorithm A that computes f with no error.

  Definition 2.3. The **zero-error randomised complexity** $R_0(f)$ of a boolean function f is defined as:

  $$R_0(f) = \min_{A_{RF}} \max_{x} E[\mathbb{Q}[A_{RF}(x)]]$$

  This zero error formulation is also referred to as a Las-Vegas style randomised algorithm, as it always computes the correct answer, but may have variable run time based on the random choices it makes.
  Alternatively, we may instead take a Monte-Carlo style approach wherein we compute a function f with a randomised algorithm A_{RF}, while allowing a small probability of error c - i.e
  $\forall x, P[A_{RF}(x) \neq f(x)] \leq c$.
  In this case, we characterise the complexity of the function f in terms of the bounded-error randomised complexity $R(f)$, which measures the maximum number of queries $\mathbb{Q}$ taken by a bounded-error optimal algorithm A_{RF} in the worst case:

  Definition 2.4. The **bounded-error randomised complexity** $R(f)$ of a boolean function f is defined as:

  $$R(f) = \min_{A_{RF}} \max_{x,Q} \mathbb{Q}[A^Q_{RF}(x)]$$

  where A^Q_{RF} denotes the execution of a randomised algorithm A_{RF}, drawn from the set of all randomised algorithms which compute f with some small probability of error, where the randomness has been fixed according to a specification Q.

  Finally, in certain cases the function f that we're computing may have asymmetric behaviours on 1 and 0 inputs - i.e it may be easier to determine 0-outputs than 1-outputs. To capture these situations, we can also consider the 1-bounded-error randomised complexity $R_1(f)$, which measures the maximum number of queries taken by a 1-bounded-error algorithm (i.e only makes errors for x such that $x \in f^{-1}(1)$) in the worst case:

  Definition 2.5. The **1-sided bounded-error randomised complexity** $R_1(f)$ of a boolean function f is defined as:

  $$R_1(f) = \min_{A_{RF}} \max_{x,Q} \mathbb{Q}[A^Q_{RF}(x)]$$

  where A_{RF} is drawn from the set of all randomised algorithms which compute f with some small probability of error for $x \in f^{-1}(1)$, and no error otherwise.

Standard properties As is typical with complexity analysis, past research has established several relations between these complexity measures. Below we present a few of the relevant relations to provide context for the separations covered in this report, however as the focus of this report is on more recent developments, we will not discuss their proofs in detail, and recommend the reader consult the referenced papers for a more complete coverage.

- **Query Complexity Hierarchy** - The first obvious relation between these complexity measures arises from the fact that each subsequent measure represents a more powerful model of computation - i.e a bounded error randomised algorithm can easily execute a 0-error randomised algorithm, and similarly a randomised algorithm can easily simulate a deterministic one. From this we conclude:

  $$R(f) \leq R_0(f) \leq D(f)$$

- **Maximum Randomisation Separation** - One interesting question that has already been partially answered is a bound on the maximum possible improvement in query complexity when introducing randomisation.
  This result for the 0-error case was first derived by demonstrating a bound on the separation between the deterministic $D(f)$ and non-deterministic complexity $N(f)$ for any function f by Blum *et al.* [5]:

  $$D(f) \leq N(f)^2$$

This bound is demonstrated by devising an algorithm that computes the function f by repeatedly trying each one of the possible non-deterministic queries (i.e clauses of the DNF of f), and using the results of f to minimise the number of iterations - in the interests of time, we won't cover this proof here.
From this relation, we can then simply use the fact that $N(f) \leq R_0(f)$ to conclude that:

$$R_0(f) = \Omega(D(f)^{\frac{1}{2}})$$

As such any randomisation can at best have a quadratic improvement in the query complexity of a deterministic algorithm, however, until recently, it was not known whether this is tight, as most known randomised algorithms have had a weaker speedup than this. In fact, others have even conjectured the existence of a tighter lower bound.
As in general $N(f) \not\leq R(f)$, this result does place any bounds on the maximum separation when allowing bounded errors. In fact, a separate lower bound for this case was proven separately by Nissan *et al.* [6], who showed that the maximum separation for bounded error is at most cubic[1]:

$$R(f) = \Omega(D(f)^{\frac{1}{3}})$$

Unlike the previous bound, this relation is not known to be tight, and no prior research has found any functions exhibiting such a cubic separation.

2.2 Maximum separations between complexity measures

A long-standing open problem in the field of query complexity has been, until recently, the question of the maximum possible separation between randomised and deterministic algorithms. In 1986, Saks and Wigderson[1] were investigating the use of randomisation in efficiently computing AND-OR trees for use in artificially intelligence. As part of this research, they developed a randomised variant of alpha-beta pruning and proved that the optimal complexity of randomised 0-error algorithm for this function $R_0(f)$ was exactly $\Theta(D(f)^{0.753...})$. As, at the time, this was the greatest separation between randomised and deterministic algorithms that had been found, Saks and Wigderson then conjectured that the known quadratic lower bound on deterministic versus random separation was not tight, and that $O(D(f)^{0.753})$ was in fact the optimal separation. This conjecture has remained standing for over 20 years since it was proposed, and has only recently been disproven.

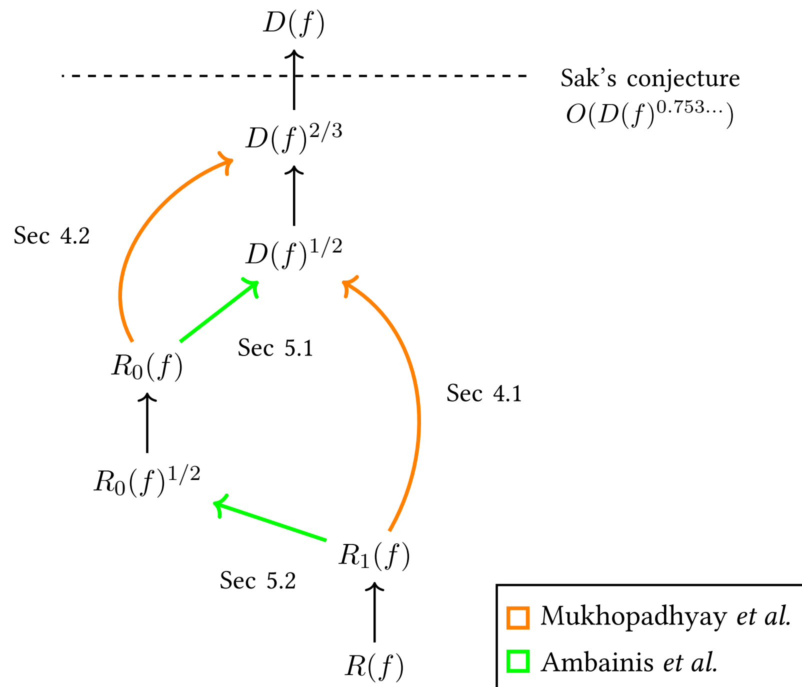


Fig. 2: *Separations between Deterministic and Randomised Query Complexity*

This breakthrough came about as a result of a publication by Göös *et al.* [4] in the orthogonal field of communication complexity. Specifically, Göös *et al.*. devised a novel strategy of using pointers within the definition of a boolean function to reduce the cost of calculating a 1-instance. Subsequently, other researchers in query complexity realised that this type of function had certain properties that made it particularly efficient to verify using randomisation.

[1]This paper also proved $R_1(f) = \Omega(D(f)^{\frac{1}{2}})$, although this is known to be tight.

Following this observation, a number of later papers built on GPW's strategy to develop new results in query complexity, and in particular, disprove Saks and Wigderson's conjecture, as well as produce several improvements in the best known separations between various randomised and deterministic complexity classes. Figure 2 presents an overview of the main separations that were developed during this phase - note that the functions f for each comparison are not necessarily the same.

In the remainder of this report, we will present the core concepts behind these developments, starting with a discussion of Göös *et al.*'s function in the next section.

3 Göös-Pitassi-Watson Function

In this section we gradually build up the intuition and context behind the Göös-Pitassi-Watson function, henceforth referred to as GPW, providing its definition and proving some of its basic properties.

3.1 Motivation for Göös-Pitassi-Watson Function

When Göös *et al.* initially developed the GPW function they were actually working on problems in communication complexity, an orthogonal field of analysis considering games in which two parties must work together to calculate a joint boolean function while minimising the number of messages they exchange. Over the course of their research, Göös *et al.* had managed to reduce their analysis to a smaller problem in query complexity, and, owing to this difference in perspective, were interested in finding boolean functions with particular properties not typically considered by standard analysis in this area. More specifically, given the overlying context of communicating between parties, Göös *et al.* were interested in finding a boolean function that had **unambiguous 1-certificates** - *i.e.* given any input for which the function outputs 1, there should be exactly 1 unique collection of bits of the input that are necessary and sufficient to verify the output of the function as 1. As it turns out, enforcing unambiguity on certain boolean functions can also make them particularly amenable to admitting randomised solutions.

To build up our intuition for this, we will begin our analysis on a less complex boolean function. Consider a function $f : \{0,1\}^{r \times s} \rightarrow \{0,1\}$, defined as treating its input as a matrix, and then outputting 1 if and only if it contains a unique column (the principal column) of all 1s (see Figure 3). For simplicity, we'll assume that $r = s = n$. Also, note that this function is not unambiguous, as there may be multiple "proofs" of that an input is a valid 1-instance - specifically, to prove a 1-instance, we just need to select the principal column, and point out at least a single 0 in every other column, of which there may be multiple entries to choose from.

1	1	0	1	0	0
1	1	0	1	0	0
1	1	0	1	1	1
1	1	0	1	1	0
1	0	1	1	1	0
0	1	0	1	0	0

Fig. 3: *Boolean matrix function input*

At first glance, this function seems to be an ideal win for randomisation, as a deterministic algorithm must always take $r \times s = O(n^2)$ queries in the worst case (owing to the possibility of unlucky runs as presented in the diagram), whereas randomised algorithms always have the chance of solving the problem in $r = O(n)$ (in the case that the algorithm reads exactly the bits from the column and a 0 from one cell in every other column). However, a closer analysis reveals that this function has certain inherent complexity that forces poor performance even for randomised solutions. More specifically, the possibility of unlucky runs (see Figure 3) means that in the worst case the algorithm may take considerable time to find both the 1-column and witness 0-cells for the other columns - *i.e.* consider the case where each non-principal column has exactly 1 0-cell, then if each column is sampled randomly in turn, the expected time till the 0-cell is found is $O(n)$ (as the sampling is essentially a geometric random variable with probability $\frac{1}{n}$, hence as there are n columns in the input, the overall search takes $O(n^2)$).

In particular, Göös *et al.* noticed that if a randomised algorithm commits to a particular column as being the principal column and then it later turns out to contain a 0, then the queries spent reading the values of this column are entirely wasted. It turns out that the primary reason for this cost is the ambiguity in the function, which limits the information that can be inferred by reading values from one column, thereby requiring the algorithm to waste its queries in the presence of an unlucky run.

In attempting to correct for this issue by enforcing unambiguity, the GPW function was conceived.

3.2 Definition

The GPW function is defined as a property $f : \Sigma^{r \times s} \to \{0, 1\}$ on a $n \times n$ matrix as in our prior example - again, for simplicity we'll assume that $r = s = n$. However, unlike in that previous instance, the elements of the matrix, are not just values in $\{0, 1\}$, but are drawn from a set Σ, defined as:

$$\Sigma = \{0, 1\} \times ([r] \times [s] \cup \{\bot\})$$

More specifically, each element of Σ is a product of a value $v \in \{0, 1\}$ and a nullable pointer $p \in [r] \times [s] \cup \{\bot\}$ to another position in the matrix. We briefly also note that as the pointers themselves can be encoded in $O(\log(rs))$ space, this modification only changes the size of an input by a poly-logarithmic factor $O(\log(n))$, and thus does not significantly affect the complexity.

With this definition, it becomes possible to extend the previous task of determining the existence of a unique principal column of all 1s to have unambiguous 1-certificates. In particular, Göös-Pitassi-Watson define their function to return 1 on a matrix M if and only if the following holds:

1. There is a unique principal column C of the matrix with entries which have their values all equal to 1.
2. The principal column C of M has exactly 1 entry with a non-null pointer $p^* \neq \bot$.
3. The pointer p^* starts a pointer chain that only visits 0 valued entries, and visits all other columns than C in $n - 1$ steps

Figure 4 illustrates an example 1-instance of the function.

The core modification that was made to the prior function is that the "proofs" of validity are now encoded in the function itself - rather than validating a unique principal column by requiring at least one 0-entry in every other column (of which there may be multiple entries that can be chosen as a witness), the pointer chain from the principal column now directly points out the specific 0-entry cells. As such, this new function has unambiguous 1-certificates, as the principal column and the pointer chain uniquely specify the bits required to verify the output of the function in the case of a 1-instance.

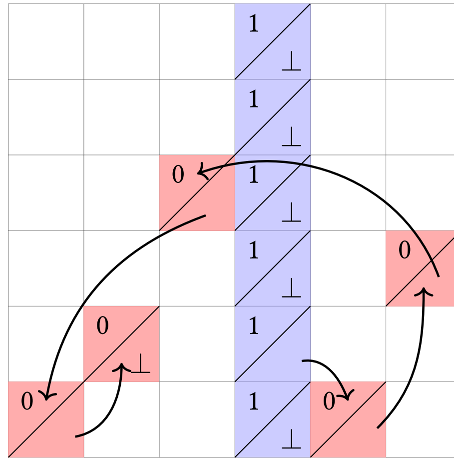


Fig. 4: *Göös-Pitassi-Watson function input*

As explored in more depth in Section 4, this additional structure in the input format also provides opportunities for efficient randomised solutions. Intuitively, this is because the presence of a pointer chain that passes through every column, means that even if a randomised algorithm commits to a single column and it turns out to be an unlucky run as before, the queries are not wasted, as it will find an element on the pointer chain and can use this to eliminate other columns.

As we will see in the next subsection, this additional structure can not always be effectively exploited by deterministic algorithms, and there still exists worst-case inputs for which the deterministic query complexity is large.

3.3 Deterministic Query Complexity

We will now show that despite this additional structure, the worst case deterministic query complexity for this function is still large. In order to do this, we will construct an adversarial input strategy that requires that any deterministic algorithm makes $\tilde{\Omega}(n^2)$ queries.

The strategy is as follows:

1. Respond to each initial query for an entry with a value of $1, \perp$ and track which entries of the matrix have been seen so far.
2. Whenever the algorithm queries a cell which is the last unseen cell in its respective column, respond with either a $0, \perp$, if this is the first unqueried cell, or $0, p$, where p is a pointer to the last 0 cell that was returned.
3. When the algorithm finally queries the last unseen cell, return either $1, p$ or $0, \perp$ to force the input instance to be either a 1-instance or a 0-instance.

The intuition behind this strategy is that each response to a query is crafted to ensure that the search continues for as long as possible. For example, step 1 ensures that a deterministic algorithm will never find the principal column first, as whenever it tries to query all the elements of any single column, the very last element will always turn out to be 0. Additionally, the choice to ensure that the pointer for each column always points to a previously seen cell means that the deterministic algorithm can never learn any additional information by following the pointer chain, as it only visits previously seen entries.

Following this reasoning, the value of the GPW function on this adversarial input will not be known until every cell has been queried[2], which means that any correct deterministic algorithm will have to query all $rs = n^2$ entries of the matrix in the worst case before it can calculate the value of the function. As each entry is $\log(rs) = O(\log(n))$ size, the overall deterministic query complexity of this function is $O(n^2 \log(n)) = \tilde{O}(n^2)$ (where the $\tilde{O}$ hides poly-logarithmic factors).

We will now explore how introducing randomisation can circumvent this strategy and obtain even smaller query complexity bounds.

4 Deterministic vs Monte Carlo separation and Corollaries

In this section, we demonstrate how the GPW function can solved efficiently using randomised algorithms, basing our analysis on the original proof by Mukhopadyay *et al.* [2]. We begin by presenting an algorithm for the 1-bounded error case and proving that it can be executed in $\tilde{O}(n)$ time, thereby establishing a quadratic separation from the deterministic complexity. Then, in order to extend this to the 0-error case, we also consider 0-bounded error, and we present an algorithm that we prove can be executed in $\tilde{O}(n^{2/3})$ time. We use this algorithm to obtain a $\tilde{O}(n^{2/3})$ solution for the 0-error case, and thereby refute Saks *et al.*'s original conjecture. Finally, we discuss the various issues with the GPW that prevent randomised solutions obtaining the optimal quadratic separations, proving some lower bounds on the maximum separation that is possible using this function.

4.1 Upper bound on 1-sided error Randomised Complexity

We will now consider how to design an randomised algorithm that efficiently computes the GPW function with 1-bounded error. When dealing with 1-sided error, as no errors are permitted for $x \in f^{-1}(0)$, the algorithm must only return an 1 if it finds a valid 1-certificate, but can conservatively return a 0 at any point. In other words, a 1-sided bounded error randomised algorithm can be trivially obtained if we can devise a search procedure that finds 1-certificates with sufficient probability given an $x \in f^{-1}(1)$ - i.e we run the search algorithm, if it returns a valid 1-certificate then we return 1, else return 0.

To simplify the reasoning we'll perform the analysis in terms of "cells" of the input matrix that have been read (i.e a cell is read if any/all of it's bits are read). As each cell constitutes $\log(rs) = O(\log(n))$ bits, this will add at most a factor of $\log(n)$ to our results, which will not affect the subsequent complexity analysis.

4.1.1 Intuition

The logic underpinning Mukhopadyay *et al.*'s algorithm quite closely follows the random sampling strategy we briefly mentioned in Section 3.1, wherein we eliminate columns by randomly sampling for 0-cells in them. As before, in the worst case, if each non-principal column contains exactly one 0-cell, then the sampling of each column can be treated as a geometric random variable with parameter $\frac{1}{r}$, and thus the expected number of steps to eliminate a

[2]Functions with this property are referred to as *evasive* in the literature.

single column is $O(r)$. If we were to simply repeat this process for each of the s columns, then the overall complexity would be $O(rs) = O(n^2)$ which wouldn't improve on the deterministic case.

The key observation found by Mukhopadyay *et al.* is that the pointers from cells on the principal chain must also link to cells in other columns that may have not yet been removed. As such, we can modify the algorithm to follow the pointer chains of each cell they find, and use it to eliminate other columns. Additionally, using the fact that each column must contain exactly one 0-cell on the principal chain, we can conclude that, in expectation, one such cell will be found once every r samples. As can be found formally proven in Appendix A, it also turns out that, with high probability, each cell we find on the principal chain will contain a constant fraction of the unseen columns ahead of it. Combining these two facts, we can then conclude that with high probability the search will take $O(r \log(s))$ rather than the $O(rs)$ result from the previous naive strategy.

Finally, once the principal column has been found, the verification can be done in $O(r + s)$ time by simply verifying the principal column contains only 1s and then traversing the principal chain.

4.1.2 Challenges

While this strategy seems to be suitable, there is a particular challenge that we have failed to account so far - if we simply follow all the pointer chains of each 0-cell we find without validation, then then this allows an adversary to force the algorithm to waste queries by creating fake pointer chains (Figure 5) that only visit columns we have already seen. In the worst case, each pointer chain could be of length $\Omega(s)$, and thereby increase the time complexity of the search to $O(rs) = O(n^2)$, once again failing to exhibit any separation from the deterministic solutions.

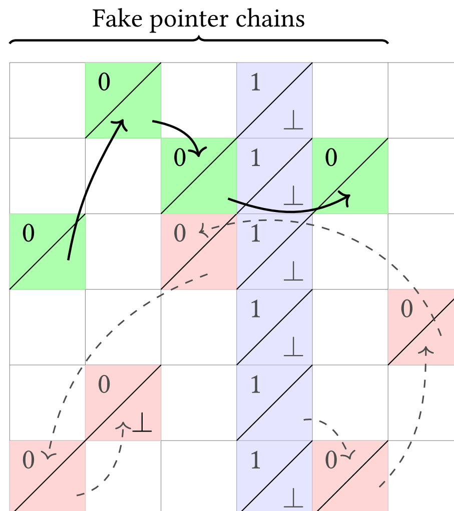


Fig. 5: *Göös-Pitassi-Watson Long pointer chains*

The key observation to solving this problem is to realise that fake pointer chains of themselves are not an issue - as long as the chain visits sufficiently many unseen columns, it is not important whether it actually a principal chain or not, as the analysis will hold regardless. Additionally, as we are sampling randomly from the columns, our position on the pointer chain is effectively chosen uniformly at random, and thus we can expect that when following the principal chain, we will see undiscovered columns at a consistent rate.

Using these two observations, an obvious strategy presents itself - specifically, when following pointer chains, we keep track of the rate at which unseen columns are found - if the rate is too low, then we are probably not on the principal chain, and thus can terminate the search early and return to sampling. As proven in Appendix A, adopting this strategy then allows the algorithm to achieve speed-up using pointer chains without also introducing the potential for adversarial attacks.

We direct the reader to Appendix A for a more formal proof.

4.1.3 Bounding the Separation between $R_1(f)$ and $D(f)$

Putting these results together, we find that we can compute GPW in $\tilde{O}(r + s)$ queries - this is quite impressive, as simply verifying the principal column alone takes $r + s$ queries, so when using randomisation, the 1-certificate can be found with at most a poly-logarithmic factor of extra queries. If we set $r = s = n$, then the deterministic query complexity as before is $\tilde{O}(rs) = \tilde{O}(n^2)$, whereas the 1-bounded error randomised query complexity is $\tilde{O}(r + s) = \tilde{O}(n)$. Hence, $R_1(f) = O(D(f)^{\frac{1}{2}})$, thereby exhibiting an optimal quadratic separation.

4.2 Extending to 0-error Randomised Complexity

In this section, we will consider how to bound the query complexity of $R_0(f)$ following the strategy by Mukhopadyay *et al.*, and discuss how to use this result to refute Saks *et al.*'s conjecture.

4.2.1 Intuition

While the previous section demonstrated a quadratic separation between the randomised bounded error $R_1(f)$ and deterministic query complexity $D(f)$, Saks *et al.*'s conjecture is specifically about 0-error randomised algorithms, not covered by this analysis. Fortunately, there is a simple way to build upon the prior results to provide a bound for the 0-error case. In particular, if we can devise an algorithm to compute the negation of GPW ($\overline{\text{GPW}}$) with 1-bounded randomised error in a number of queries that is bounded by some function $g(r, s)$, then we can obtain a simple algorithm to calculate GPW with 0 error with $O(r + s + g(r, s))$ queries in expectation. This is by simply running both algorithms, and returning any result with 0-error guarantees (i.e returning 1 if the algorithm for computing GPW returns 1 and 0 if the algorithm for computing $\overline{\text{GPW}}$ returns 1). While there is a possibility that both may return results without 0-error guarantees, by the fact that these events have a constant probability bound, we can simply repeat the process until we obtain a favourable outcome, and compute a 0-error result in an constant factor of extra expected time.

Following the same reasoning as in the previous section, we will be able to implement a bounded error randomised algorithm for the $\overline{\text{GPW}}$ if we can devise a search strategy for the 1-certificates with sufficient success probability.

As this time we're dealing with $\overline{\text{GPW}}$, the 1-certificates are not all the same structure, but rather can be split into one of the two forms:

1. A single 0-cell witness in each of the s columns.
2. A principal column with no valid pointer chain.

Both of these two requirements seem to be plausible to find using the earlier strategy of random sampling, but the lack of a guaranteed existence of an unambiguous principal pointer chain suggests that the complexity may be too large.

The key observation made by Mukhopadyay *et al.* for tackling this problem is to notice that while these two constraints do in fact capture all 1-certificates of $\overline{\text{GPW}}$, there is a slightly weaker property on pointer chains that can be used to eliminate inputs and is also easier to compute.

In particular, if we use span(i) to denote the set of columns accessible via a non-repeating 0-cell-visiting pointer chain from a entry in column i, then if we can find two columns i, j such that $i \notin$ span(j) and $j \notin$ span(i), then this is sufficient to conclude that the output of $\overline{\text{GPW}}$ is 1 (as there is no possible pointer chain that could cover all columns).

This weaker condition happens to be easier to compute, as it allows the use of a pointer-based elimination strategy as with the previous analysis. More specifically, provided the number of 0s in each column are small ($\leq \sqrt{r}$), we can again consistently eliminate a constant proportion of the remaining columns by simply sampling from the columns and discarding any columns that are in the span (taking $\tilde{O}(r+\sqrt{r}s)$ to perform). Using this strategy, the elimination process will take at most $\tilde{O}((r + \sqrt{r}s)\log(s))$ to complete with sufficient probability.

Finally, in order to satisfy the constraint that the number of 0s in each column are sufficiently small, we can preprocess the input by eliminating columns with more than $\sqrt{r}$ 0-cells by random sampling as in the previous analysis. In particular, for these violating columns, as each draw has a probability of at least $\frac{\sqrt{r}}{r} = \frac{1}{\sqrt{r}}$ of obtaining a 0-cell, we will be able to eliminate all the columns with more than $\sqrt{r}$ 0-cells after sampling $\tilde{O}(\sqrt{r})$ times from each column with sufficient probability. As such, the overall complexity is $\tilde{O}(r + \sqrt{r}s)$.

We direct the reader to Appendix B for a more formal proof.

4.2.2 Refuting Saks *et al.*'s Conjecture

Following the earlier reasoning, the overall 0-error complexity of GPW $R_0(f)$ can thus be shown to be $\tilde{O}(r + \sqrt{r}s)$. If we set $r = s^2$, then this becomes $\tilde{O}(s^2 + s^2) = \tilde{O}(s^2)$, whereas the deterministic complexity is $\tilde{O}(rs) = \tilde{O}(s^3)$. Hence, there is a separation of $\frac{2}{3}$ between the 0-error randomised query complexity and the deterministic query complexity $R_0(f) = \tilde{O}(D(f)^{\frac{2}{3}})$, thereby refuting Saks's et al's conjecture.

While we will not cover it here, Mukhopadyay *et al.* also separately prove that this bound is optimal for the GPW function. This means that while the GPW function can disprove Saks *et al.*'s conjecture, it can not answer the

question of whether a quadratic lower bound on the separation between $R_0(f)$ and $D(f)$ is actually tight. Instead, in order to tackle this remaining open problem, other researchers have considered modifications of the GPW function. We will investigate these modifications in more detail in the next section.

5 Achieving quadratic separation and Further relations

At this point, we have seen that the GPW function is sufficient to both disprove Saks' conjecture and show a quadratic separation between $R_1(f)$ and $D(f)$. However, the separation of $\frac{2}{3}$ shown between $R_0(f)$ and $D(f)$ is still different from the lower bound of $1/2$. In the next section, we will discuss the work by Ambainis *et al.* [3] that improve these results by introducing some extensions to the GPW function. With this, we can see that the quadratic separation between $R_0(f)$ and $D(f)$ is a tight bound, and that there is also a function that provides a quadratic separation between $R_0(f)$ and $R_1(f)$

5.1 Quadratic Separation between Las Vegas and Deterministic

A quadratic separation between $R_0(f)$ and $D(f)$ can be shown by extending the GPW function with **back pointers**. A second bit of information is added to the *witness* zeroes in each column. Each *witness* must have a pointer back to the principal column. Each input is therefore defined by a triple - a boolean value, a forward pointer to another input or null, and a back pointer to another input or null.

The function is defined such that it is 1 only if the following conditions are all satisfied for a $n \times m$ grid of boolean values:

- There is exactly 1 column with all values as 1. (The *principal column*).
- For each other column j, there is at least one 0 which has a backpointer back to the principal column. This is a *witness* zero. It has a forward pointer to another witness zero or null.
- All of the principal column cells have both pointers set to null, except one *special element* which has a forward pointer to a witness. Following the forward pointers until a null is seen from this element traverses one witness in each column.

The deterministic query complexity for this function is $O(m^2)$ for the case where $n = 2m$. We can see this by constructing an adversarial strategy based on Ambainis *et al.*'s work. Let k be the number of queries made for a column. The adversary returns 1 (and null pointers) for $k \leq m$ queries for the column, and returns 0s with back pointers to the $k - m$ column. The adversary also maintains a counter for the last response with back pointer set to $k - m$ initialised to null, which is used to set the forward pointer for the next response with the same backpointer value (which must be in a different column by construction) So after all cells in a column are queried, there are m 1s and m 0s where each 0 points back to a different column and may point forward to a 0 in a different column that shares a backpointer destination.

This construction leads to an answer of 0 for the function. However, to set the answer to 1, we just need some column which only has responded to queries with 1s (to pick as the principal column) and ability to create a path of witnesses in the other columns. Picking the principal column is sufficient since based on the adversary response, there are already possible paths that traverses zeroes in each column and point back to the principal column. Therefore, as long as there at least m unanswered queries in a column the function is undetermined. At least m^2 queries to make sure that the number of unanswered queries is $< m$ so m^2 is also a lower bound.

A randomised algorithm can solve this in $\approx O(n + m)$ queries. Notice that once a column with all 1s is found, the remaining of the verification can be done in $n + m$ by finding the special element and traversing the path of witnesses. The algorithm is as follows:

- First, pick any column and query all its elements.
- If they are all 1, move to the verification step.
- Otherwise, sample the set of possible principal columns by looking at the destinations of the back pointers on the zeros.
- If the set is empty, reject the function as there is no possible witness.
- If the sampling surfaces any zeroes, remove the column from possibly being the principal column. Determine the sampling threshold to ensure that the probability of seeing zeroes is some small constant.
- Repeat by picking a new column to query all elements from the set of possible principal columns.
- By picking an appropriate threshold, halve the probability of seeing zeroes is halved in each iteration such that in logarithmic number of iterations the all-1s column is found if it exists.

This gives us the optimal separation between R_0 and D.

5.2 Quadratic Separation between Las Vegas and Monte Carlo

A quadratic separation between $R_0(f)$ and $R_1(f)$ can be shown by extending the GPW function with **back pointers** and changing the path of witnesses to a **a balanced binary tree**. Additionally, only half of the 'leaves' of the tree (equivalent to the witness in the original GPW) have backpointers to the principal column. The intuition here is that by making only half the witnesses useful, we make the function hard for 0-error but easy for one-sided bounded error.

Each input is defined by a quadruple - a boolean value, a left pointer to another input or null, a right pointer to another input or null, and a back pointer to another input or null. Additionally, a balanced binary tree T is defined which has 1 node per column in the input grid, and each node is uniquely labeled with a column. For some column $j \in m$, $T(j)$ is defined as the path from the root to the node with its label (as a sequence of lefts and rights).

The function is defined such that it is 1 only if the following conditions are all satisfied:

- There is exactly 1 column with all values as 1. (The principal column).
- In the principal column there is exactly 1 cell with all pointers as null. (The special element).
- For each non principal column j, we define $l(j)$ as the result of following the path specified by $T(j)$ starting from the special element. The condition is that $l(j)$ exists, is in column j and has value equals 0. (These are called the leaves of the tree, similar to the witnesses seen earlier).
- Exactly $m/2$ leaves have backpointers to the special element. The remaining $m/2-1$ leaves have backpointers somewhere else or null.

A Las Vegas algorithm takes $\Omega(nm)$ queries. This can be shown via Yao's principle, by constructing a probability distribution on the input that will take at least mn queries for a deterministic algorithm.

The input is constructed such that all pointers are null, and for each column 1 cell is set to 0 if $l_x(i) = j$, sampled over a uniform distribution over all functions $l_x : [m] \to n$ while the others are 1. An algorithm should return negative for this input. To prove that at least mn queries are required, it must be shown that with fewer queries, an adversary can still construct a positive input (and so the algorithm cannot be correct).

This is shown in 2 parts - first, if the algorithm hasn't seen $O(m)$ zeroes, a positive input can still be constructed. By this assumption, there are at least $m/2+1$ columns without zeroes. To construct the positive input, divide these columns into three exclusive sets - 1) a single column b that will be the principal column, 2) a set G of size $m/2$ to contain the leaves of the tree which have backpointers and 3) any remaining columns. For the principal column, set $a = (l_b, b)$ to be the special element. For the columns $c \in G$ set the values of (l_c, c) to be 0 and have backpointers to a. Finally for any remaining columns, set the values of (l_x, x) to be 0 and have null backpointers. There are still enough unqueried cells to add the remaining cells needed for the tree with the pointers to preserve the structure, to create a positive input. Second is to prove that it takes $O(mn)$ queries to get to a state with $O(m)$ zeroes with the input distribution. Formally, this is shown by Ambainis *et al.* by an application of Markov's inequality. However, this is intuitive since in the chosen input distribution, each column has exactly 1, randomly selected zero, and so there is no query process that takes fewer than $O(mn)$ queries to observe $O(n)$ zeroes.

The intuition behind the R_1 algorithm for this function which takes $O(n+m)$ time is to focus on the fact that exactly $m/2$ of the non-principal columns have leaves with backpointers to the column. Let such a column be *good*, so that a positive input has exactly $n/2$ good columns while a negative input has none. A building block for the approach is that there exists an algorithm with $O(1)$ queries on a vector to differentiate between all zeroes and exactly $m/2$ ones. Then, if there is an algorithm to determine the goodness of a column in $O(n+m)$ queries, that can be directly used to build an R_1 algorithm to evaluate the function.

The procedure to identify whether a given column j is good or not is as follows:

- Start with a set I of potential leaves in j, $|I| < n$ and a set B of potential principal columns, $|B| < m-1$. Building this can be done in $O(n)$ time.
- For each element in I, follow the tree path outward, if any. If the path ends in a column k with value 0, than it can be removed from B.
- After iterating over all elements, if B has only one element, then verify whether it is a principal column and whether the constructing conditions are satisfied. If yes, accept, else reject.

In each iteration, either 1 leaf or 1 potential principal column is ruled out, so this runs in $O(n+m)$.

Considering the case where $n = m$, then Las Vegas algorithms have $O(n^2)$ query complexity while Monte Carlo algorithms have $O(n)$ query complexity, which is a quadratic separation.

6 Open problems and Concluding Remarks

In the previous sections of this report, we have presented a tour of the recent breakthroughs in query complexity research that have arisen due to the discovery of the GPW function. While many of the long-standing open problems in this area have been solved following this flurry of developments, there has been one question in particular - *whether a cubic lower bound on the maximum separation between $R(f)$ and $D(f)$ is tight?* (see Section 2.1) - that has not had any similar success. In this final section of the report, we investigate this problem, discussing whether any of the techniques and insights used in the prior analysis can be extended to this problem, and providing some preliminary directions for future work.

6.1 Cubic Separation between $D(f)$ and $R(f)$

We will decompose investigating this open problem through the following 2 steps: (i) first, we will try to summarise the key insights from the previous sections in terms of a sequence of requirements that any proposed function should probably satisfy, following this, (ii) we will propose a few functions that meet these requirements and consider their potential as directions for future work.

6.1.1 Key Insights

From analysing the components of the GPW function that made the prior developments possible, we identify the following three requirements that seem to be useful for exhibiting large separations between deterministic and randomised algorithms:

1. **Evasive function** - in the worst case, a deterministic algorithm should need to read all bits of its input to compute the value of the function. The intuition for this property is fairly simply - as we are trying to exhibit as large a separation between deterministic and randomised algorithms as possible, it would be helpful if the deterministic complexity is as large as possible.
2. **Low (approximate) certificate complexity** - the number of queries to verify an output as either 1 or 0 should be small relative to the input size. Additionally, as we are now dealing with bounded-error $R(f)$, it is sufficient for the certificate complexity to be small only when verifying with probability of error - i.e allowing the use of functions such as the majority (see Section 5.2). The intuition for this requirement is that the verification complexity is a rough lower bound on the complexity of the overall function, as algorithms will likely need to verify solutions to ensure error guarantees.
3. **Pointer-aided sampling** - the function should make use of pointers in such a way that it is still possible to learn information from bad samples by following pointer chains. This property draws from one of the key steps in the development of the GPW function, wherein the use of pointers allowed converting an inefficient $\tilde{O}(rs)$ sampling strategy to an efficient $\tilde{O}(r + s)$ with at most a poly-logarithmic factor increase in the complexity.

In the next section, we will consider using these requirements to evaluate possible functions for this open problem.

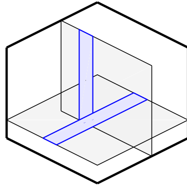
6.1.2 Potential Directions - Extending GPW to 3D

Following on from insights in the previous section, we now move to investigate a selection of functions that we feel show promise for tackling this problem.

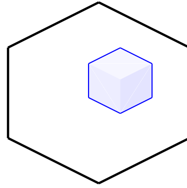
Before we begin, we must first reevaluate whether the GPW function is a suitable function for this analysis. Note that, as the certificate complexity provides a rough lower bound[3] on the query complexity of a randomised algorithm for computing a function f, GPW seems to unsuitable for this task. This is because its 1-certificate complexity alone is $\Omega(n)$, whereas the deterministic complexity is $\Omega(n^2)$. As such, it seems likely that the greatest separation we might hope to be able to establish between $R(f)$ and $D(f)$ will be around quadratic, failing to meet our cubic requirements. However, while this reasoning does invalidate the use of the GPW function, it also suggests a natural extension to circumvent this problem. More specifically, as one of the main limiting factors is the $\Omega(n^2)$ bound on the deterministic implementation, one way to avoid this problem is to simply extend the GPW function from acting on an 2D $n \times n$ matrix to a 3D $n \times n \times n$ one.

While this seems to be a straightforward transformation, a closer inspection reveals that there are in fact multiple

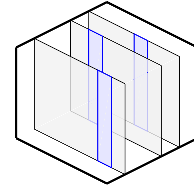
[3]It is not a exact bound as we allow bounded errors *i.e.*- it may be possible to approximately verify an input in fewer queries.



(a) *Orthogonal multi-dimensional GPW*



(b) *Generalised multi-dimensional GPW*



(c) *Majority multi-dimensional GPW*

plausible ways in which the GPW function could be extended to 3D - we will now investigate and evaluate a few of these approaches:

a) **Orthogonal multi-dimensional GPW** - A natural way of extending GPW to 3D is to simply run GPW simultaneously on multiple dimensions - i.e we split the input into n matrices of size $n \times n$ along each dimension, and require that at least one instance of GPW along each dimension is a 1-instance. Using this strategy, the verification time (for 1-instances) is still linear in terms of the input $O(2 \times (r + s)) = O(r + s)$, while making full use use of the n^3 space.
Analysis: The main flaw with this strategy is that, as the instances of GPW along each dimension overlap, it is possible for an algorithm to infer information about the GPW instances on a given dimension by querying along another dimension. As a result, the function ceases to be evasive, and thus we conclude that this is not a suitable direction.

b) **Generalised multi-dimensional GPW** - An alternative approach is to take a more holistic view of the GPW function and attempt to generalise its logic to higher dimensions. More specifically, rather than running multiple instances of GPW in the additional space as before, we will now devise a variant GPW to act over the entire matrix, instead subdividing the input matrix into smaller cubes of size k^3 where k is some arbitrary size. These cubes will act as substitutes for columns in 2D GPW, and thus a 1-instance corresponds to a single cube with all 1s and a pointer chain traversing through all other cubes and demonstrating at least one 0-cell in the cube.
Analysis: While this seems like a natural extension of GPW to multiple dimensions, by sticking too closely to the original structure, its certificate complexity ends up being too large. In particular, if we set k to be $\sqrt{n}$, then the input will consist of $(\sqrt{n})^3 = n^{3/2}$ large cubes each of size $n^{3/2}$ cells - as such, in order to verify a 1-instance, we must at least read $n^{3/2}$ bits (all $n^{3/2}$ cells in the principal "block" and then at least one 0-cell in the remaining $n^{3/2} - 1$ blocks), producing a rough lower bound on the optimal separation as quadratic rather than cubic.

c) **Majority multi-dimensional GPW** - Having seen that running GPW in multiple dimensions fails to produce an evasive function due to the interactions between the orthogonal planes, a natural alternative direction is to consider running multiple instances of GPW on *parallel planes*. The problem with this strategy alone is that, as verifying a single instance of GPW takes linear time, verifying all the instances along any single dimension will take quadratic time, thereby making the function unlikely to admit a cubic separation. To circumvent this issue we draw from the analysis by Ambainis *et al.* and rather than requiring all the parallel instances to be valid, simply a *majority of the instances*.
Analysis: As noted by Ambainis *et al.*, when using a majority vote in a boolean function, it is possible for a randomised algorithm to compute the answer with bounded error using just $O(1)$ samples. As such, this strategy allows us to extend the GPW to multiple-dimensions without affecting the certificate complexity too much and retaining evasiveness. We can also extend the pointer structures to connect between the principal chains of parallel instances, and thereby also optimise for sampling strategies.

As such, from our analysis, we find that the Majority-GPW extension seems to be the most promising direction forward. In the next section, we will try to explore the behaviours of these functions empirically to see if this can lead to any insights.

6.1.3 Experiments

Having found a suitable function - *Majority multi-dimensional GPW* (MGPW) - from the analysis in the previous section, we now try to characterise the specific behaviours of this function in more detail through a few empirical experiments. These experiments and their associated code have been compiled together into a small framework[7] for performing empirical experiments on randomised query complexity functions. The framework is open source and can be publicly accessed at: `https://github.com/Gopiandcode/query-complexity-framework`.

Given the time constraints, we did not devise a novel algorithm for solving MGPW, but rather investigated how

Mukhopadyay *et al.*'s pointer-following sampling strategy (Section 4.1) performs when used to solve majority GPW.

Our algorithm to solve MGPW was to simply randomly draw l (a constant) planes (each containing a distinct GPW instance) from the input, and then use Mukhopadyay *et al.*'s strategy to evaluate whether they are true (a *valid plane*) or false (an *invalid plane*). The algorithm would then accept or reject if the number of valid planes exceeds a specific bound[4].

Having implemented this function, we performed a series of experiments to investigate the behaviours of this strategy:

- **False-positives rate** - Our first experiment was to investigate the specific error-rate of this strategy. In particular, as this strategy of using sampling to evaluate a majority only has bounds on correctness when dealing with a 1-instance input[5], our algorithm may still error with high probability when dealing with a 0-instance[6]. As such, a important question for the viability of this strategy is how severely the error rate changes as the number of valid planes in the input increases.
  Our methodology to investigate this question was as follows: we first fixed an input size of $n = 20$, and then, varying the number of valid planes from 0 to 9, generated a single random 0-instance input for each option. We then ran the algorithm 100 times on each generated input and evaluated how the false positive rate of the algorithm would change as the number of valid planes were increased. The results of this experiment are presented in Figure 7.

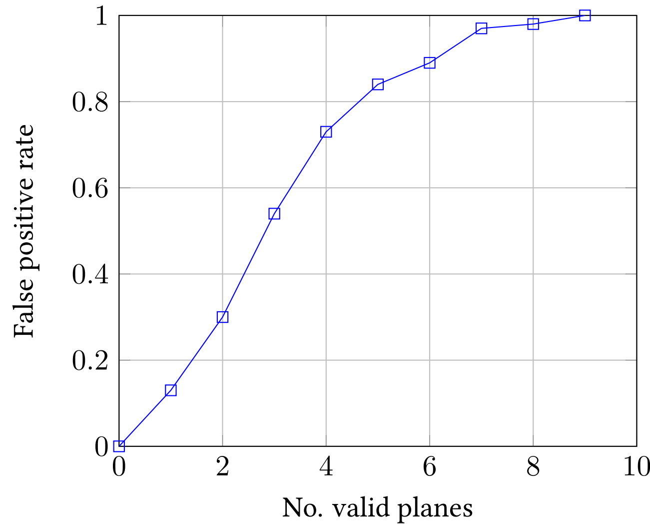


Fig. 7: *MGPW False-positive rate with limited valid planes*

  As we predicted, the false positive rate increases quite severely as the number of valid planes becomes closer to the number of invalid planes. While this is a significant issue for achieving bounded error guarantees, it is important to note that the correctness of the function may be improved if we also ran Mukhopadyay *et al.*'s $\overline{GPW}$ 1-bounded error algorithm (Section 4.2) and repeating the sampling algorithm but testing for invalid-plane majority.
- **Empirical query complexity** - Our second experiment investigates how the query complexity of this algorithm varies with the input. Our main aim with this experiment was to evaluate whether our analysis of achieving a linear verification time was reasonable.
  Our methodology was as follows: first, varying the input size from 4 to 20, we generated a single true and false instance for each input size, and then repeatedly ran the algorithm 100 times on these inputs and took the average number of queries for each input. The results of this analysis are presented in Figure 8.
  As we would expect, as the number of samples increases, the query complexity does correctly become linear, verifying our earlier analysis. The slight abnormality for input sizes $n \leq 10$ is due the fact that the constant number of samples l used in the majority estimation algorithm is set to 10, so for input sizes less than that, we just use half the input size, thereby leading to the odd non-linear behaviour.
- **Impact of invalid pointer chains** - Our final experiments sought to investigate whether the structure of the GPW function could be further optimised for bounded error separation. In particular, given the special

[4]The specific bound was $n/2 - \sqrt{n}$, which has a probability of $\geq \frac{2}{3}$ of occurring when dealing with a 1-instance.
[5]Achieved using a Chernoff bound, as each sample has a probability $\geq \frac{1}{2}$ of being correct.
[6]This is most likely to occur when the number of 1-instances in the input is very close to the number of 0-instances.

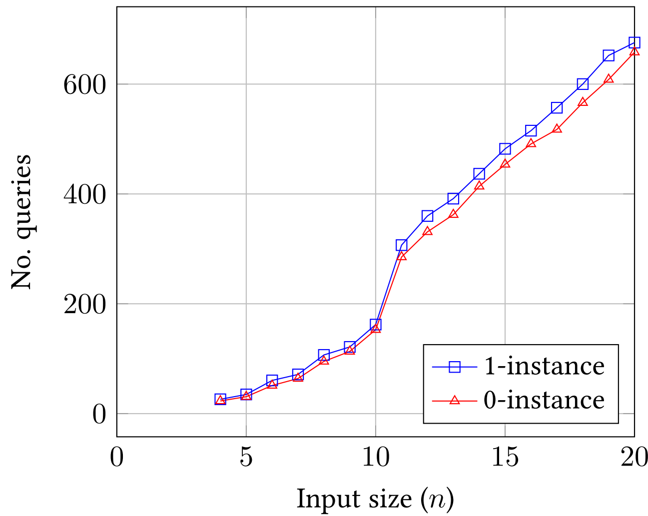


Fig. 8: *MGPW Empirical query complexity*

case reasoning for fake pointer chains that the prior research had to include, we were interested in working out whether fake pointer chains have a significant influence on the query complexity of this algorithm. Our methodology was as follows: first, varying the input size from 4 to 20, we generated 100 random inputs that had random pointers for non principal chain columns. For each input size, we ran the algorithm 10 times for each input and took the average over all times and inputs. The results of this experiment are presented in Figure 9.

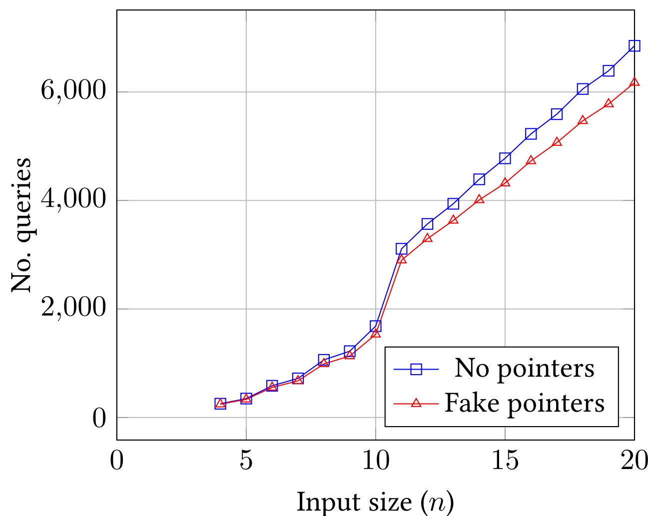


Fig. 9: *MGPW impact of fake pointers (1-instance)*

Surprisingly, contrary to our expectations, there is a slight *decrease* rather than increase in the number of queries when we introduce fake pointers. This is surprising as we were expecting the fake pointers to increase the query complexity by forcing the algorithm to follow fake pointer chains. One possible explanation for this phenomena could be that as we are randomly generating pointers, the likelihood of an actual pointer chain forming by chance is fairly small, and so most pointers have a high probability of helping the algorithm to find the remaining columns.

6.2 Concluding Remarks

This report has aimed to cover the main developments in query complexity research over the past years, providing an account of the insights and observations that have finally lead to the refutation of Saks *et al.*'s longstanding conjecture. We began with the Göös-Pitassi-Watson function, discussing its definitions and the intuition behind it's suitability for randomisation. Following this, we presented an algorithm by Mukhopadyay *et al.* that was able to solve GPW with bounded error, and described how this result could be used to obtain an algorithm computing GPW with 0-error, refuting Saks's conjecture, but also failing to exhibit the optimal quadratic separation. We then considered Ambainis *et al.*'s extensions to GPW and investigated how they were not only able to achieve this optimal separation, but also able to exhibit the first known quadratic separation between 0-error and bounded error randomised algorithms. Finally, in the last sections of this report, we considered how the methodology of

introducing pointer-based optimisations as done with the GPW function, could be used to attack another open problem in this space - that of whether it is possible to exhibit a cubic separation between deterministic and 2-sided error randomised algorithms.

References

[1] M. Saks and A. Wigderson, "Probabilistic boolean decision trees and the complexity of evaluating game trees," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, IEEE, 1986, pp. 29–38.

[2] S. Mukhopadhyay and S. Sanyal, "Towards better separation between deterministic and randomized query complexity," *arXiv preprint arXiv:1506.06399*, 2015.

[3] A. Ambainis, K. Balodis, A. Belovs, T. Lee, M. Santha, and J. Smotrovs, "Separations in query complexity based on pointer functions," *J. ACM*, vol. 64, no. 5, Sep. 2017, ISSN: 0004-5411. DOI: 10.1145/3106234. [Online]. Available: https://doi.org/10.1145/3106234.

[4] M. Goos, T. Pitassi, and T. Watson, "Deterministic communication vs. partition number," *SIAM Journal on Computing*, vol. 47, no. 6, pp. 2435–2450, 2018.

[5] M. Blum and R. Impagliazzo, "Generic oracles and oracle classes," in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, IEEE, 1987, pp. 118–126.

[6] N. Nisan, "Crew prams and decision trees," *SIAM Journal on Computing*, vol. 20, no. 6, pp. 999–1007, 1991.

[7] K. Gopinathan and J. Mohan, *QUERY COMPLEXITY FRAMEWORK: FRAMEWORK FOR EMPIRICALLY EXPLORING QUERY COMPLEXITY QUESTIONS*, https://github.com/Gopiandcode/query-complexity-framework, 2020.

A Formal Analysis of $R_1(\textbf{GPW})$

We will now provide a more formal analysis of the strategy presented in the previous subsection and demonstrate that it does in fact produce a valid 1-bounded error randomized algorithm that executes with $\tilde{O}(r+s)$ queries. The first part of this section will describe the exact algorithm in a formal notation, and the subsequent parts will tackle proving the query complexity and correctness of the algorithm separately.

Listing 1 and 2 present a formal description of the general algorithm. As we can see, the code mostly follows the strategy outlined earlier - we repeatedly sample from the matrix and use the pointer chains to eliminate columns. Once the elimination is complete, we return 1 if we have actually found a valid 1-certificate, and otherwise return 0.

```
1:  Initialize S to contain all columns {1, ..., s}
2:  for t ← 1 to Ar log(s) do
3:      Sample a random column j from S
4:      Sample a random row i from [r]
5:      Update S using FollowPointer(S, i, j)
6:  end for
7:  if S contains exactly one remaining column that is a 1-certificate then
8:      return 1
9:  else
10:     return 0
11: end if
```

Listing 1: *High level structure of 1-bounded error algorithm*

We implement the rate-aware pointer following algorithm by means of a dynamic loop bound step $\leq 100s\frac{|\mathcal{D}|}{|\mathcal{S}|}$. In particular, each colum we discard in a single call increases the maximum bound for that iteration by $100\frac{s}{|\mathcal{S}|}$ steps, where $\mathcal{S}$ is the set of remaining columns. As a simple sanity-check, we can quickly verify that as $100\frac{s}{|\mathcal{S}|}$ is always greater than 1, we will never incorrectly reject a pointer chain of just consecutive unseen columns (i.e in the case that our initial sample is lucky and our first sample is the first element of the pointer chain). Additionally, as we will see in the subsequent analysis, it also means that as the number of remaining columns decreases, the reward for finding an undiscarded column increases, which accounts for the reduced rate of column-finding as the search progresses.

```
1:  function FollowPointer(S, i, j)
2:      step ← 0
3:      D ← ∅
4:      while step ≤ 100s|D|/|S| do
5:          step ← step + 1
6:          Read the value x_{i,j} and pointer p_{i,j} of cell i, j
7:          if x_{i,j} ≠ 0 then break
8:          if j ∈ S/D then Update D with D ∪ {j}
9:          if p_{i,j} = ⊥ then break
10:         Update (i, j) with p_{i,j}
11:     end while
12:     return S/D
13: end function
```

Listing 2: *Rate tracking follow pointer subroutine*

Query Complexity We will now prove that the query complexity of this function is $\tilde{O}(r+s)$.

We note that the final verification steps at line 7 of Listing 1 of ensuring that a column is a valid 1-certificate takes at most $\tilde{O}(r+s)$ queries as we simply need to read one column of length r and the principal pointer chain of length $s-1$. As such, provided we can place an $\tilde{O}(r+s)$ bound on the $Ar\log(s)$ sampling steps, then the overall bound will hold.

As such, let q_i denote the number of queries made in the i-th iteration of the sampling step, let s_i be the size of the remaining columns at that iteration, and let d_i denote the number of columns that are discarded in that iteration. Note that $d_i = s_i - s_{i+1}$.

As queries in the sampling step are only made inside the FollowPointer subroutine, we will first place a simpler bound on the number of queries made in a single call to this routine. In particular, as each iteration of the while loop (line 4, Listing 2) makes at most 1 query, we can bound the number of queries by the maximum bound of the loop $q_i \leq$ `step`, hence:

$$q_i \leq 100s\frac{d_i}{s_i} + 1$$

With the $+1$ as in the final iteration of the loop,

Now to bound the total number of queries, we consider the summation over all t steps:

$$\sum_{i=1}^{t} q_i \leq 100s \sum_{i=1}^{t} \frac{d_i}{s_i} + t$$

Expanding $d_i = \sum_1^{d_i} 1$:

$$= 100s \sum_{i=1}^{t} \sum_{j}^{d_i} \frac{1}{s_i} + t$$

As $d_i < s_i + 1$:

$$\leq 100s \sum_{i=1}^{t} \sum_{j}^{d_i} \frac{1}{s_i - j + 1} + t$$

Expanding the summation:

$$= 100s \sum_{i=1}^{t} \left(\frac{1}{s_i} + \cdots + \frac{1}{s_i - d_i + 1} \right) + t$$

Using the fact that $d_i = s_i - s_{i+1}$:

$$= 100s \sum_{i=1}^{t} \left(\frac{1}{s_i} + \cdots + \frac{1}{s_{i+1} + 1} \right) + t$$

Expanding the summation:

$$= 100s \left\{ \left(\frac{1}{s_1} + \cdots + \frac{1}{s_2 + 1} \right) + \cdots + \left(\frac{1}{s_t} + \cdots + \frac{1}{s_{t+1} + 1} \right) \right\} + t$$

So overall, the sum iterates over $\frac{1}{s_i}$ for i from 1 to t. From here, using the fact that s_{t+1} represents the size of the set at step t and that $s_1 = s$:

$$\leq 100s \left\{ \sum_{i=s_t+1}^{s} \frac{1}{i} \right\} + t$$

Expanding the definition of t and using the standard logarithmic bound for an inverse sum:

$$\begin{aligned} &\leq 100s \log(s) + Ar \log(s) \\ &\leq \tilde{O}(r + s) \end{aligned}$$

As such, the sampling has the complexity bound as the verification of $\tilde{O}(r + s)$, giving the overall algorithm a complexity bound of $\tilde{O}(r + s)$.

Bounded Error We will now prove that this function will find a 1-certificate with probability at least $\frac{2}{3}$ when given an input $x \in f^{-1}(x)$.

Let $|S|_t$ denote the number of remaining columns at time t. Observe that for a 1-input, the search succeeds if $\mathcal{S}$ at the end has exactly 1 element.

If we can show that with sufficient probability ($\geq \frac{1}{8r}$), each call to FollowPointer reduces the number of remaining columns by a constant faction ($\frac{1}{2}$), then the overall statement can be easily proven by considering conditional expectations.

$$
\begin{aligned}
E[|S|_t||S|_{t-1}] &\leq \left(1 - \frac{1}{8r}\right)|S|_{t-1} + \frac{1}{8r}\frac{|S|_{t-1}}{2} \\
&\leq \left(1 - \frac{1}{16r}\right)|S|_{t-1}
\end{aligned}
$$

Hence, by induction:

$$
\begin{aligned}
E[|S|_t] &\leq \left(1 - \frac{1}{16r}\right)^t s \\
&\leq \exp\left\{-\frac{1}{16r}t\right\} s \\
&\leq \exp\left\{-\frac{1}{16r}Ar\log(s)\right\} s \\
&\leq s^{1-\frac{A}{16}}
\end{aligned}
$$

Thus for a sufficiently large constant A, we can apply Markov's inequality to conclude that the probability of the search failing is less than $\frac{1}{3}$.

Now, all that remains is to prove that each call to FollowPointer reduces the number of columns by a constant fraction with sufficient probability.

In order to prove this, we can first prove a slightly more general property that places bounds on the deviation of the value of various subsequences of a summation from the total value.

Theorem A.1 (Subsequence Summation Bound). *Suppose we have a summation of l non-negative values $x_1, \ldots, x_l$ such that $\sum_{i=1}^{l} x_i = N$ for some value N.*

We define a index j as bad (and otherwise good) if it denotes the start of a sub-sequence of some length $D \in [1, l]$ such that the value of the summation is significantly greater than than the average value:

$$
\sum_{i=j}^{j+D-1} x_i \geq 100D\frac{N}{l}
$$

Then, the number of bad indices in the first half of the sum is bounded:

$$
|\{j \leq l/2 \wedge \ j \text{ is good}\}| \geq \frac{l}{4}
$$

Proof. To prove this, we first divide the array into disjoint subsets $K_1, \ldots K_m$, by repeatedly drawing out the associated ranges for a bad index starting from the smallest remaining bad index (see Figure 10).

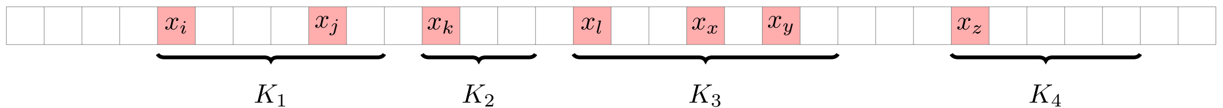


Fig. 10: *Subdividing array by bad indices*

Clearly, by construction, every bad index is now contained in one of these subsets, and as each subset contains exactly the D elements associated with its smallest bad index, we can bound the size as follows:

$$
100|K_i|\frac{N}{l} < \sum_{i \in K_i} x_i
$$

Hence, by rearranging:

$$|K_i| < \frac{1}{100} \times \frac{l}{N} \times \sum_{i \in K_i} x_i$$

From this, we note that the number of bad indices b is now trivially bounded by the size of the union of each of these disjoint sets, hence:

$$\begin{aligned} b &\leq \sum_{i=1}^{m} |K_i| \\ &\leq \frac{1}{100} \times \frac{l}{N} \times \sum_{i=1}^{m} \sum_{j \in K_i} x_j \end{aligned}$$

As the total number of elements in the subsets is at most all the elements in the array:

$$\leq \frac{1}{100} \times \frac{l}{N} \times \sum_{i=1}^{l} x_i$$

By the assumption that the total sum is N:

$$\begin{aligned} &\leq \frac{1}{100} \times \frac{l}{N} \times N \\ &\leq \frac{l}{100} \end{aligned}$$

As such, the number of good indices in the range 1 to $l/2$, is at least $l/2 - l/100$, which is greater than $l/4$ □

Finally to apply this to our situation, suppose we are at some iteration t of the algorithm, and the remaining columns are defined by the set $\mathcal{S}$, define $m_1, \ldots m_{|\mathcal{S}|-1}$ to be the order in which the principal chain visits the elements of $\mathcal{S}$. Additionally, let x_i denote the number of seen columns (i.e columns in $[s]/\mathcal{S}$) that the principal chain passes when moving from m_i to m_{i+1}.

We will now consider the summation of these x_is, which represents the total number of previously seen columns that are encountered during the pointer following search (including the column we start on). Clearly, the following trivial bounds apply to the summation:

$$|\mathcal{S}| - 1 \leq \sum_{i=1}^{|S|-1} x_i \leq s - 1$$

Notice, that our pointer following algorithm will fail if and only if the index it chooses is "bad" (using the definition from the previous theorem), as the rate at which unseen columns are found will be too low.

Using this observation and the previous theorem, the lemma trivially follows. When we sample an element, with probability $\frac{1}{r}$ it will be on the pointer chain, with probability $\frac{1}{2}$ it will be in the first half, and with probability $\frac{1}{4}$ it will be one of the "good" indices.

Hence, with probability $\frac{1}{8r}$, the pointer following algorithm will not terminate and will consume at least half of the remaining columns.

B Formal Analysis of $R_1(\overline{\textbf{GPW}})$

In this section we will provide a formal proof of the 1-bounded randomized query complexity of $\overline{\text{GPW}}$. This section begins with a formal description of the exact algorithm and then proves the query complexity and probability of correctness separately.

Listing 3 presents the formal description of the algorithm. As described earlier, the algorithm first preprocesses the input by eliminating all columns with greater than $\sqrt{r}$ 0-cells by random sampling. After this is done, the algorithm simply uses the pointer-following technique to either find a 1-certificate or to eliminate columns. At the end, if there are more than 1 remaining columns, the algorithm conservatively returns 1 and otherwise deterministically calculates the exact answer.

Query Complexity We will now show that this function takes $\tilde{O}(r + \sqrt{r}s)$ queries to compute.

```
1: Initialize S to all columns in {1, ..., s}
2: for c in S do
3:     for t = 1 to 10√r log(s) do
4:         Sample a cell v randomly from c
5:         if v is a 0-cell then update S with S/{c}
6:     end for
7: end for
8: for t = 1 to B log(s) do
9:     if |S| ≤ 1 then break
10:    Sample two columns i, j from S
11:    if more than √r cells with 0 value found then return 1
12:    if i ∉ span(j) ∧ j ∉ span(i) then return 0
13:    else
14:        Update S with S/(span(i) ∪ span(j))
15:    end if
16: end for
17: if |S| > 1 then return 0
18: else
19:    Deterministically check if 0 or 1 instance.
20: end if
```

Listing 3: *High level structure of 1-bounded error $\overline{GPW}$ algorithm*

Proof. This proof is fairly trivial, and follows from the bounds of each of the loops of the algorithm. The first loop iterates $10\sqrt{r}\log(s)$ times for each of the s columns and makes a single query each iteration, hence contributes to the complexity by $\tilde{O}(\sqrt{r}\log(s) \times s) = \tilde{O}(\sqrt{r}s)$. The second loop iterates $B\log(s)$ times, and in each iteration calculates the span of 2 random columns, taking at most $\tilde{O}(r + \sqrt{r}s)$ queries for each column (r queries to read the column, and $\sqrt{r}s$ queries to follow the pointer chains from each 0-cell), hence contributes to the overall complexity by $\tilde{O}(B\log(s) \times (r + \sqrt{r}s)) = \tilde{O}(r + \sqrt{r}s)$. The final verification step takes at most $\tilde{O}(r + s)$ queries.

As such, by summing all these bounds, the overall complexity is $\tilde{O}(r + \sqrt{r}s)$ as claimed. □

Bounded Error We will also prove that this function will return 1 when fed an $x \in f^{-1}(1)$ with probability at least $\frac{2}{3}$.

This proof follows by three main steps: (i) first, we prove that the first loop has a high probability of eliminating any columns with more than $\sqrt{r}$ 0-cells, then conditioning on this, (ii) the second step involves proving that when sampling columns we have a high probability of either finding a 1-certificate or eliminating a large number of remaining columns, finally, (iii) we combine these facts to demonstrate that with high probability the remaining columns after both loops will less than or equal to 1.

(i) **Removing 0-cell dense columns** - Suppose a column c contains more than $\sqrt{r}$ 0-cells. As such, each time we sample from c, there is at least a probability of $\frac{\sqrt{r}}{r} = \frac{1}{\sqrt{r}}$ that we obtain a 0-cell and thereby eliminate c. From this, the probability that c is not eliminated after all $10\sqrt{r}\log(s)$ is at most $(1 - \frac{1}{\sqrt{r}})^{10\sqrt{r}\log(s)} \leq \exp\{-10\log(s)\} = s^{-10}$. By the union bound, the probability that all such column are eliminated is at least $1 - s^{-9}$.

(ii) **Span-based column elimination** - We will now show that each time we sample columns i, j randomly from the set of remaining columns $\mathcal{S}$ in the second loop, then one of the following must happen:
   (a) $P[|\text{span}(i) \cap \mathcal{S}| \geq \frac{|\mathcal{S}|}{10}] \geq \frac{1}{10}$
   (b) $P[i \notin \text{span}(j) \wedge j \notin \text{span}(i)] \geq \frac{1}{2}$

   As such, each time we sample, we either find a 1-certificate with high probability or are able to eliminate a constant fraction of the remaining columns.

   To prove this, suppose that that the first statement is false, then the with probability at least $\frac{9}{10}$, $|\text{span}(i) \cap \mathcal{S}| < \frac{|\mathcal{S}|}{10}$. As such, when we randomly sample another column j, with probability at most $\frac{9}{10}$, $j \notin \text{span}(i)$. Combining these two events, the probability that $j \in \text{span} i$ is at most 0.2. The same analysis holds for the converse case $i \in \text{span} j$, and so by the union bound, with probability at least $\frac{1}{2}$, $i \notin \text{span}(j) \wedge j \notin \text{span}(i)$.

(iii) **Remaining Columns** - Let $|S|_t$ denote the number of remaining columns at the tth iteration of the second

loop (i.e $|S|_0$ is the number of columns after completing the first loop).
We will now show that $E[|S|_t||S|_{t-1}] \leq \left(\frac{99}{100}\right)|S|_{t-1}$.
Each time we sample a column, using the result of the previous step, we know that we either a) find a 1-certificate with at least $\frac{1}{2}$ probability or b) we eliminate at least $\frac{1}{10}$th of the remaining columns with a probability of at least $\frac{1}{10}$. We'll consider these two cases seperately:

a) If we find a 1-certificate, then we terminate the loop, hence $|S|_t = 0$:

$$E[|S|_t||S|_{t-1}] \leq \frac{1}{2}0 + \frac{1}{2}\mathcal{S}_t t - 1 \leq \left(\frac{99}{100}\right)|S|_{t-1}$$

b) Assuming we eliminate the columns with probability at least $\frac{1}{10}$:

$$E[|S|_t||S|_{t-1}] \leq \left(1 - \frac{1}{10}\right)|S|_{t-1} + \frac{1}{10}\frac{1}{10}|S|_{t-1} \leq \left(\frac{99}{100}\right)|S|_{t-1}$$

Hence, in either case, this claim holds.
Applying induction, we can conclude that $E[|S|_t] \leq \left(\frac{99}{100}\right)^{B\log(s)} s$.
We conclude by applying Markov to note that for a large enough constant B, the probability of the event that $|S|_t > 1$ (and thereby there is a possibility that the algorithm could be wrong) will be at most $\frac{1}{3}$.