Counting complexity: compressed Hamming distance, vertex covers, and recent highlights.

Ioannis Nemparis

May 3, 2015

When I started counting my blessings, my whole life turned around.
– Willie Nelson, The Tao of Willie: A Guide to Happiness

Contents

List of Figures

Abstract

This thesis consists of three parts. The first is concerned about the Compressed Hamming distance. The second answers some natural questions about counting Vertex Covers in lines, cycles and trees. The third one presents techniques for counting algorithms design and milestone results.

Part I
Compressed Hamming Distance

1 Introduction

We start by defining the problem. The Hamming Distance is probably the first problem tackled, when being introduced in coding theory. Given two alphabets and two strings S,T of same size, one of each alphabet, the goal is to return an integer indicating the number of symbols with the same index number inside the strings being different. For example notice that for $S = 000$ and $T = 000$ the Hamming Distance (HD from now on) is zero as the strings are identical. Contrary for S=000 and T=001 the HD is 1 as the strings differ in only one symbol (the

symbol having index number 3 in S is 0 but in T it is 1). Notice that this problem has a linear time algorithm. Read the two tapes and check if $S_i = T_i$ if yes then add +1 to a counter. When moving to the compressed version the texts are not given explicitely. Instead the input consists of two Straight Line Programs. Straight Line Programs (SLPs from now on) are a method of compressing big strings. A SLP is a set of rules implicating how the text can be reconstructed. For example: $\begin{array}{rcl} X_5 & \leftarrow & X_4X_4 \\ X_4 & \leftarrow & X_3X_2 \\ X_3 & \leftarrow & X_2X_1 \\ X_2 & \leftarrow & \text{b} \\ X_1 & \leftarrow & \text{a} \end{array}$ The X_is are called rules. This SLP recontructs the string: $babbab$ (we will use "text" and "string" and "eval(X)" to refer to the uncompressed string, hopefully without confusion for the reader). The smaller subtexts corresponding to the rules (X_i s) with smaller i than 5 ($X_1, .., X_4$) are a,b,ba,bab respectively. The uncompressed string corresponding to the SLP is the one corresponding to the rule with the maximum i (i_{max} from now on). Notice that each SLP decompresses to only one string, but the converse does not hold and many different SLPs may correspond to the same string. The SLPs have the following two properties. They must have some rule $X_i \leftarrow c$ where c is a terminal symbol. Secondly, every other rule must be $X_m \leftarrow X_kX_l$ with $m > k, l$. So, computing the compressed HD (CHD from now on) is: Given two SLPs, compute the HD of the uncompressed texts they correspond to. Or more formally:

Definition 1. *Given two SLPs X,Y, compute the HD of eval(X) and eval(Y).*

2 Previous work

It is shown in [10] that computing the exact CHD is a $\#P - complete$ problem. Also to the best of our knowledge there is no algorithm for the problem, other than the naive approach of decompressing the text and finding the HD with the naive linear algorithm. This is out of question in the general case, as the text is generally too big to uncompress. In this article, we will propose 2 algorithms and some heuristics for special cases, as well as set the basis for an impossibility result.

3 Algorithm 1

SLPs is a compression method that has many structural properties that help greatly in creating algorithms.

First of all notice that we can easily compute the size of the intermediate subtexts. This can be easily done by the fundamental observation:

Remark 1. *Decompressing a text is not needed to find its size.*

The size of every intermediate subtext can be computed using the following recursive function:

$$size(X_i) = \begin{cases} 1 & \text{if } X_i \leftarrow c \\ size(X_k) + size(X_l) & \text{if } X_i \leftarrow X_k X_l \end{cases}$$

The algorithm is based on dynamic programming, filling a matrix of size $m \times n$ (m,n are the number of rules of the two SLPs respectively, given as input). It computes the Hamming distances of all the previous subtexts of any size between them. The entry $M[i][j]$ in this matrix is the H.D between the subtexts X_i, Y_j. It is highly probable, that the subtexts of the two SLPs will not have the same sizes. In this case, that specific entry in the matrix will contain a value (i.e $5 + X_3 + X_5$) meaning that the Hamming distance is 5 up to the point, where there exist symbols to compare in both texts, plus a X_3 concatenated by a X_5, cause the X_i rule happened to be longer than the Y_j. We begin by computing the first subtexts of very small size exhaustively and proceed by using them to compute the bigger ones. Let there be two 2-symbol alphabets with rules X_i and Y_j. After computing the first 2×2 upper left square, we continue by computing the i', j' such that $i' = \min i : \forall j (i', j) not - computed$ and $j' = \max j : length(Y_j) \leq length(X_i)$ and complete the rest of the column with smaller i's. Finally we compute the rows from right to left starting from the row j' and moving to smaller j's. Eventually,

we will reach the $M_{i_{max},j_{max}}$ which holds the answer.

Input: Two SLP's X,Y
Output: The Hamming distance of eval(X),eval(y)
res=0;
Compute upper 2*2 square ;
`/* easy comparison between 1 symbol long texts. */` i_{prev} = 3;
repeat
  Find i',j' ;
  for $k = i'$; $k \geq 1$; $k--$ **do**
    `CompDist(`*k,j',M,res*`)` ;
    $M[k][j'] = res$;
  end
  for $k = i'$; $k >= i_{prev}$; $i--$ **do**
    for $j = j' - 1$;$j >= 1$;$j--$ **do**
      `CompDist(`*i,j,M,res*`)` ;
      $M[i][j] = res$;
    end
  end
  $i_{prev} = i'$;
until $i_{prev} = i_{max}$;

Algorithm 1: CHD(X,Y)

Input: A value of the Matrix M
Output: The "arithmetic" part of the H.D
return arithmetic part of M[i][j];
`/* i.e for 5+`X_2 `this will return 5 */`

Algorithm 2: get_val(M[i][j])

Input: A value of the Matrix M
Output: The "strings remaining" part of the H.D
return string part of M[i][j];
`/* i.e for 5+`X_2 `this will return` X_2 `*/`

Algorithm 3: get_subtexts(M[i][j])

Input: 2 arrays of SLPs X,Y
Output: CHD of X,Y
Remark: X, Y are arrays of concatenated SLP's
res=0 ;
while $(X[] \wedge Y[]) \neq$ *empty* **do** `/* both nonempty */`
  if *M[a][b] not empty* **then** ; `/* a,b are the indeces of the subtexts in the leftmost SLPs in X[] Y[] respectively */`
    if *length*(X_a) >*length*(Y_b) **then**
      `CompDist(`$get\_subtexts(M[a][b]) \oplus X \setminus Xhead$, $Y \setminus Yhead, M, res + get\_val(M[a][b])$`)`
    else
      `CompDist(`$X \setminus Xhead$, $get\_subtexts(M[a][b]) \oplus Y \setminus Yhead, M, res + get\_val(M[a][b])$`)`
    end
  else
    find i,y,i',y' such that $X \leftarrow X_i X_y, Y \leftarrow Y_{i'} Y_{y'}$;
    if *length*(X_i)≥*length*($Y_{i'}$) **then**
      `CompDist(`$get\_subtexts(M[i][i']) \oplus X_y, Y_{y'}, M, res + get\_val(M[i][i'])$`)`
    else
      `CompDist(`$X_y, get\_subtexts(M[i][i']) \oplus Y_{y'}, M, res + get\_val(M[i][i'])$`)`
    end
  end
end
if *(X[]* $\wedge$ *Y[])* = *empty* **then** `/* both empty */`
  break;
`/* the result is in res. */`
end
else if *(X[] empty)* **then**
  $res = res \oplus Y[]$;
  break;
end
else
  $res = res \oplus X[]$;
  break;
end

Algorithm 4: CompDist(X[],Y[],M,result)

$$
\begin{array}{lcl}
X_5 & \leftarrow & X_4X_3 \\
X_4 & \leftarrow & X_3X_1 \\
X_3 & \leftarrow & X_2X_1 \\
X_2 & \leftarrow & \mathsf{b} \\
X_1 & \leftarrow & \mathsf{a}
\end{array}
\qquad
\begin{array}{lcl}
Y_6 & \leftarrow & Y_5Y_2 \\
Y_5 & \leftarrow & Y_4Y_3 \\
Y_4 & \leftarrow & Y_2Y_1 \\
Y_3 & \leftarrow & Y_2Y_2 \\
Y_2 & \leftarrow & \mathsf{b} \\
Y_1 & \leftarrow & \mathsf{a}
\end{array}
$$

corner	X_1	X_2	$\ldots$	X_5
Y_1	0	1		
Y_2	1	0		
$\vdots$				
Y_6				

corner	X_1	X_2	X_3	$\ldots$	X_5
Y_1	0	1			
Y_2	1	0	$\Uparrow$		
Y_3			0		
$\vdots$					
Y_6					

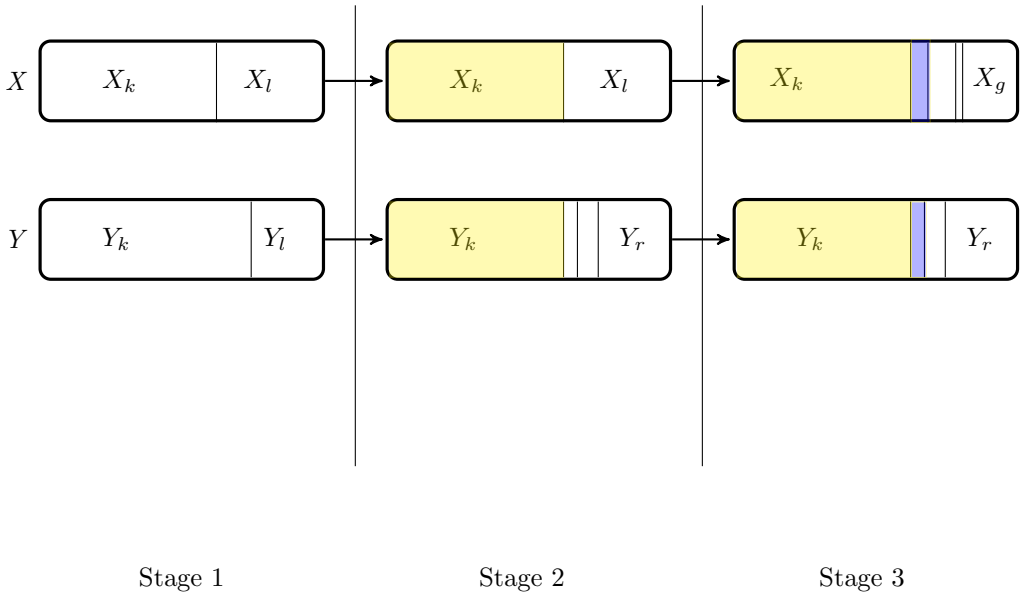
corner	X_1	X_2	X_3	$\ldots$	X_5
Y_1	0	1	$1+x_1$		
Y_2	1	0	x_1		
Y_3		$\Leftarrow$	0		
$\vdots$					
Y_6					

corner	X_1	X_2	X_3	X_4	X_5
Y_1	0	1	$1+x_1$	$1+X_1X_2$	$1+X_1X_2X_3$
Y_2	1	0	X_1	X_1X_2	$X_1X_2X_3$
Y_3	$1+Y_2$	Y_2	1	X_2	X_2X_3
Y_4	$1+Y_1$	Y_1	0	$1+X_2$	$1+X_2X_3$
Y_5	$1+Y_2Y_3$	Y_2Y_3	$1+Y_3$	$1+Y_1$	$2+X_1$
Y_6					2

3.1 Correctness of Algorithm 1

The first step is to compare the X_k and $Y_{k'}$ of the respective $X_i \leftarrow X_kX_l$ and $Y_i \leftarrow Y_{k'}Y_{l'}$. Here we need to note the following. First that as $k < i$ and $k' < i$

Figure 1: Execution of algorithm 1.



the $M[k][k']$ value is computed in the upper left area of the matrix, so we can use it . For the moment, the result is the CHD between $X_k, Y_{k'}$. As expected, $X_k, Y_{k'}$ won't have the same size and there will be some subtexts "left" from the longer between $X_k, Y_{k'}$ (suppose X w.l.o.g). Thus, the new subproblem is the CHD between the first of the subtexts concatenetated in front of X_l and Y_l. Following the same idea and concatenating always the remaining subtexts appropriately in front, leads to a concatenation of strings (the matrices X[] and Y[] above). These matrices may contain more and more strings at each iteration, but in the same time, the sum of their lengths strictly decreases with each iteration. This holds because at each step, we trace back to the matrix. As the subtext sizes are never zero, the distance is computed and thus progress in the computation is made. Consequently the computation will eventually finish.

3.2 Complexity of Algorithm 1

The time complexity of the algorithm can be computed by the following function.

$$T(C.H.D(X_i, Y_i)) = T(M[k][k'])+T(C.H.D(get\_subtetexts(M[k][k'])\oplus X_l, Y_l))$$

We may assume that the first factor of the sum on the right part of the equation takes time $O(1)$ time as it is precomputed (just look the table at the appropriate place). The time to compute each value of $M[i][j]$ is mainly consumed in the process of "comparing" the subtexts left as $M[k][k']$ dictates (the second factor of the sum).

Lemma 3.1. *When comparing 2 texts of any size (not necessary the same between them), the resulting compressed text left, are a concetenation of at most $O(n)$ subtexts.*

Proof. First some terminology. We will address to the two texts to be compared as the first and the second texts. At the first decomposition, they will be addressed as subtexts and after > 1 decompositions as subsubtexts. Recall also, that n is the cardinality of rules of the SLP with the most rules. The idea is to recursively decompose the biggest text, starting from its left subtext. If the left subtext's length is smaller than that of the second text we ignore it, rewind that decomposition and decompose the right subtext. As long as the right subsubtexts begin after the place where the second text ends, they are added in the list of subtexts resulting in the remaining compressed text. Generally, either the left or the right subsubtext, at each stage, will span both on the left and the right of the place, where the second text

ends (else the algorithm finishes). In that occasion, the decomposition continues taking that subsubtext (the crossing one). The time needed for this procedure is at most $O(n)$. We will give a proof for the case, where the left subsubtext is always longer that the right. The case in a general SLP, when the inverse can happen, is totally symmetric.

Decomposing the left subsubtext can be done to a maximun of n times (n rules in total). Note that, at the worst case, when the index of the leftmost subsubtext in the (n-1)-th decomposition is that of the second subtext +1, the n-th decomposition will cover at least half of the second subtext (because the left subsubtext is always bigger that the right one). Consquently, half of the second subtext is covered with at most n decompositions. At the same time, on the right of the index of the second text, the solution has already some parts "created" due to the process. When the second subtext will be totally covered, on the right side of the index there will be a concatenation of rules (subsubtexts forming the remaining compressed text). Those rules can not be more than $O(n)$. They equal the number of decompositions in each "half-covering" of the second subtext and they are at most

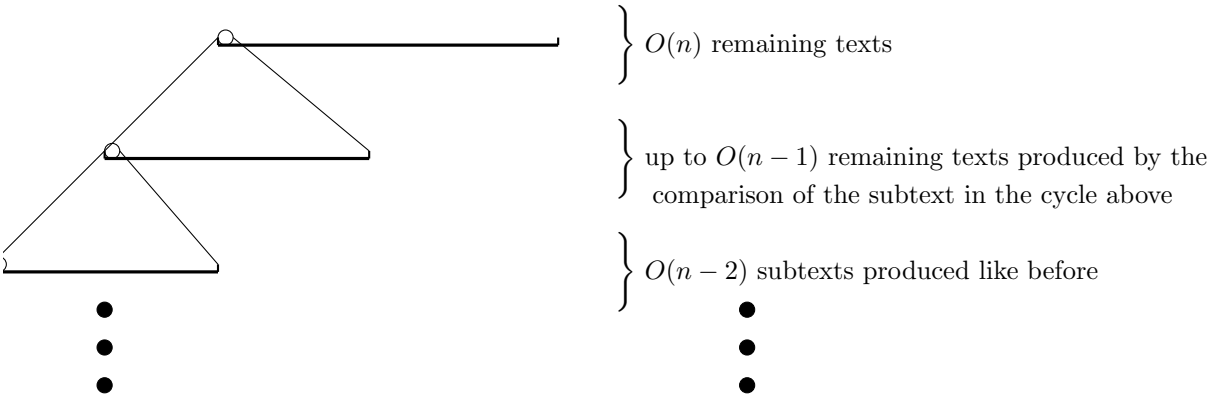
$$F(n) = F(\frac{n}{2}) + n$$

in total, meaning they are at most $O(n)$. □

Returning now to something mentioned above, let $X_i \leftarrow X_k X_l$ and $Y_i \leftarrow Y_{k'} Y_{l'}$. The time complexity of the algorithm is $T(C.H.D(X_i, Y_i)) = T(M[k][k'])+ T(C.H.D(get\_subtetexts(M[k][k'])\oplus X_l, Y_l))$ supposedly $length(X_l) < length(Y_{l'})$. What follows is the attempt to quantify how much of the matrix M can be efficiently computed. Let the worst case, where $T(compare(X_k, Y_{k'}))$ returns $O(n)$ remaining subtexts, with indeces $O(n)$. So, at worst, they will need $O(n)$ decompositions and they might produce at each of the $O(n)$ steps up to $index$ of the subtext many remaining subtexts. An example is shown below. So at worse, we have a tree structure with a root of n vertices (n possible remaining subtexts) and each having degree n-1 (number of maximum possible remaining subtexts produced when compared to another subtext belonging to the second SLP), for every n. So we have a tree, with leaves equal to the biggest number of comparisons needed. This number of leaves is $O(\frac{n}{2}!)$.

For the matrix to be filled up to a $log^c n$ index needs at most $O(log^c n)!$ time. When dealing with SLP of around 10^9 construction rules (meaning some gigabytes of compressed text) and a $c = 2$, the running time is approximated by the polynomial $T(n) = n^{105}$, which is intractable. However, we can compare subtexts

Figure 2: Worst case of Algorithm 1. Factorial time



derived from the first $logn$ rules of both SLPs efficiently. If we decompose the left subtexts of the two max index rules of both SLPs $n - logn$ times, we can compare the two leftmost subtexts and return an exact answer based on the M matrix. We can continue in the sense described above, comparing subtexts going from left to right between them(always appending in front the probable remaining subtexts). Practically, this means we can cut the depth of the tree at any given level getting only the last $depth$ factors of the factorial (i.e $n * (n-1) * (n-2)$ when cut in the third level). This approach, as the naive approach, doesn't give any information about what happens in the rest of the text. In the worst case, this can be arbitrarily bad. For example, this algorithm may show that the first n symbols of the uncompressed texts of the two SLPs are identical and thus have $H.D = 0$. The remaining symbols can be pairwise different and thus the algorithm approximates the H.D as 0, while it actually can be at most $2^n - n$.

3.3 Case the heuristic performs better

However there are some instances when the heuristic proposed works better. We consider the parameter $compressability = \frac{Usize}{Csize}$, as the fraction of the size of the

uncompressed text over the number of rules. The algorithm performs well, when Usize gets close to Csize. Unfortunately, this means that the rules are not vastly outnumbered by the size of the uncompressed text and thus the compression somehow failed. The really interesting cases are those, where the compression was very good, as in the contrary situation, probably the exhaustive algorithm can perform well too. Although comparing the rules $X_{log^{c'}n}, Y_{log^{c}n}$ demands filling $log^{c'*c}n$ cages in the matrix M, in practice $O(logn)$ are many times enough, because as the compressibility of the text is large, the patterns repeat themselves. In this case, the complexity is close to $O(log(n)!)$ which can be approximated by $O(n^3)$ even for data of size $O(n^{21})$ meaning some zettabytes (The internet in 2013 was supposed to be 4 zettabytes).

4 Algorithm 2

In this section we will discuss a second algorithm, of $O(n^{c+3})$ compelxity that can compute exactly the C.H.D, if it happens to be up to $O(n^c)$ in polynomial time. c is a constant that can be set at any value. For this, we will use the algorithm described by Yury Lifshits [10]. His algorithm gets 2 compressed texts and in time $O(n^3)$ returns if they are identical or not. n is the size of the SLPs. Our algorithm decomposes the SLPs and at each iteration uses the Lifshits algorith. (denoted as Lif. from now on) as a blackbox. If the algorithm returns yes, the H.D. is 0 and thus the computation does not continue in this branch. On the other hand, if it returns no, there is at least a different symbol, so the H.D. is at least 1. Starting from the i_{max} rule of the SLP, we can decompose at least to a $c * logn$ depth as there will be at most n^c oracle calls at that level. There is only one technicality before writing the algorithm. For eval(X),eval(Y) to be compared, they must be of same size. Thus, there are new rules created, so that the resulting subtexts are of same size. In $O(n)$ time, we can find the subtext of max size of the first SLP that has smaller length than that of the text produced by the second SLP (let's call that index h and suppose the first SLP has a longer left subtext than the second one). With an analysis similar to that of the lemma proven above, we can show that in $O(n)$ time we can find up to $O(n)$ subtexts such that $length(X_h \oplus subtexts) = length(Y_k)$ where Y_k is the left subtext of the first decomposition of the i_{max} rule of Y. In the same time, for each subtext appended, we had another one that started after $length(Y_k)$ place.

Transform Part 1. *Let* $length(X_h \oplus X_{10} \oplus X_{20}) = length(Y_k)$. *Let the remaining*

texts be a matrix remain[]. In this case we create a new SLP with:

$$X'_k = \begin{cases} X_k & \textit{if } k \leq h \\ X'_k \leftarrow X'_{k-1}(remain[i-h]) & \textit{if } k > h \end{cases}$$

Now for the subtexts starting after $length(Y_k)$ place we have another matrix remain2[]. Find the max index rule in remain2 (assume f) and assume m the size of the Y SLP. The new SLP is:

Transform Part 2.

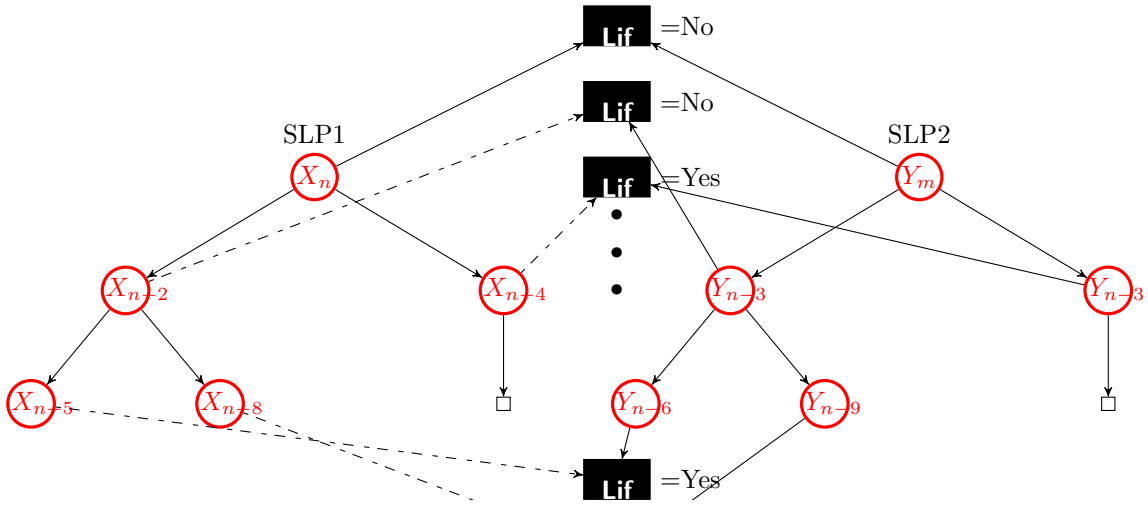
$$Y'_l = \begin{cases} X_l & \textit{if } l \geq 1, l \leq f \\ Y_{l-f} & \textit{if } l \geq f+1, l \leq f+m \\ remain2[1]remain2[2] & \textit{if } l = f+m+1 \\ Y'_{l-1}(remain2[l+1-m-f]) & \textit{if } l > f+m+1, i \leq f+m+|remain2|-1 \\ Y'_{l-1}Y'_{f+m} & \textit{if } l = f+m+|remain2| \end{cases}$$

Input: Two SLPs X,Y
Output: The exact C.H.D if it is $O(n)$, or $> O(n)$ if it is bigger

```
Right[] ← Y ;
Left[] ← X ;
for i=1 to O(logn) do
    Dist =0 ;
    Decompose(Right[]) ;
    Decompose(Left[]) ;
        /* In later iterations we should take care,
    that the matrices have alternately 1 subtext
    from the Left matrix and one from Right then
    again from Left etc */ place = 1 ;
    place2= 1;
    put left subtexts of X,Y in Left and right subtexts in Right ;
    for j=place to sizeofLeft j+=2 do
        SameSizeTransform(Left[j],Left[j+1]) ;
        if Lifshits(Left[j],Left[j+1])= true then
            erase both from the Left matrix     /* No distance here
            */ ;
            place -=2 ;
        else
                /* At least 1 different symbol */;
            Distance ++ ;
        end
    end
    for j=place2 to sizeofRight j+=2 do
        SameSizeTransform(Right[j],Right[j+1]) ;
        if Lifshits(Right[j],Right[j+1])= true then
            erase both from the Right matrix    /* No distance here
            */ ;
            place2 -=2 ;
        else
                /* At least 1 different symbol */;
            Dist ++ ;
        end
    end
    Print(Distance at least Dist) ;
end
```

Algorithm 5: CHD(X,Y)

Figure 3: Execution of Algorithm 2



The algorithm goes as follows: Get X_n and Y_m. Decompose them. If the right subtexts between them and left subtexts between them do not produce uncompressed texts of equal size, use the transform above. Run $Lif(X_k, Y_k)$ and $Lif(X_l, Y_l)$. If yes is returned do nothing. There is no distance to be found in this part of the texts. If no is returned, then at least a symbol is different and thus the H.D is at least 1. We can repeat that process at most $O(n^c)$ times to be able to finish the computation in polymial time. The algorithm can compute exactly C.H.D up to $O(n^c)$ for SLPs with size n.

4.1 Complexity of Algorithm 2

Depending on the $c*logn$ depth we choose, we have: $T(n) = O(n^c)*T(transform)+ O(n^c) * T(Lifshits\_call) = O(n^{c+1}) + O(n^{c+3}) = O(n^{c+3})$.

Remark 2. *There is a slight inaccuracy in the algorithm. The first for ranges till $O(logn)$. That is not exactly correct (it is the worst case where there are different symbols all along and the decompositions create a complete tree). It may happen that the only H.D. is concentrated in some area and thus decomposition*

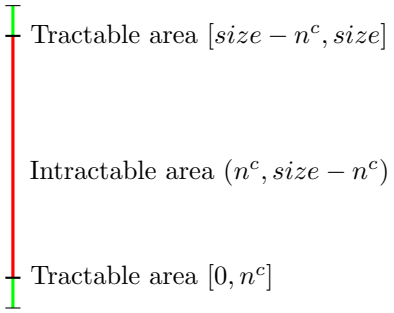
can go in greater depth. The importance lays in the Lifshits blackbox calls being polynomial. So it would be more accurate to have the counter range inside $O(n^c)$ calls and not $O(logn)$ tree depth in the worst case. The algorithm was written this way for greater simplicity.

4.1.1 Modifying Algorithm 2

As we cited before, algorithm 2 can compute C.H.D up to $O(n^c)$. With more closer look however, it is a bit stronger.

Remark 3. *Assume a 2 symbol alphabet. Modifying the terminal symbols and changing them mutually (i.e from $X_1 \leftarrow a, X_2 \leftarrow b$ to $X_1 \leftarrow b, X_2 \leftarrow a$, algorithm 2 can compute C.H.D in the interval $[2^n - n, 2^n]$.*

Figure 4: Execution of Algorithm 1.



Remark 4. *When the alphabet becomes a > 2-symbol one, then the 'symmetric' argument used above does not hold. Thus, algorithm 2 can only answer for C.H.D $\leq n^c$ and only.*

5 Formal Proofs

In this section we will show 2 lemmata that through the paper have been repeated without proof. First, that given a SLP we can create a new one, with polynomially more rules, where the left subtexts are always bigger than the right subtexts. Secondly, we will show a simple algorithm, that creates the two matrices above, remain and remain2, used directly and implicitely many times.

Lemma 5.1. *Given a polynomial-size SLP P1. We can create a new one P2 in polynomial time ($O(n^2)$) such that P2 is has the same size and P2's rules consist of two subtexts, where the left has always bigger size than the right.*

Proof.

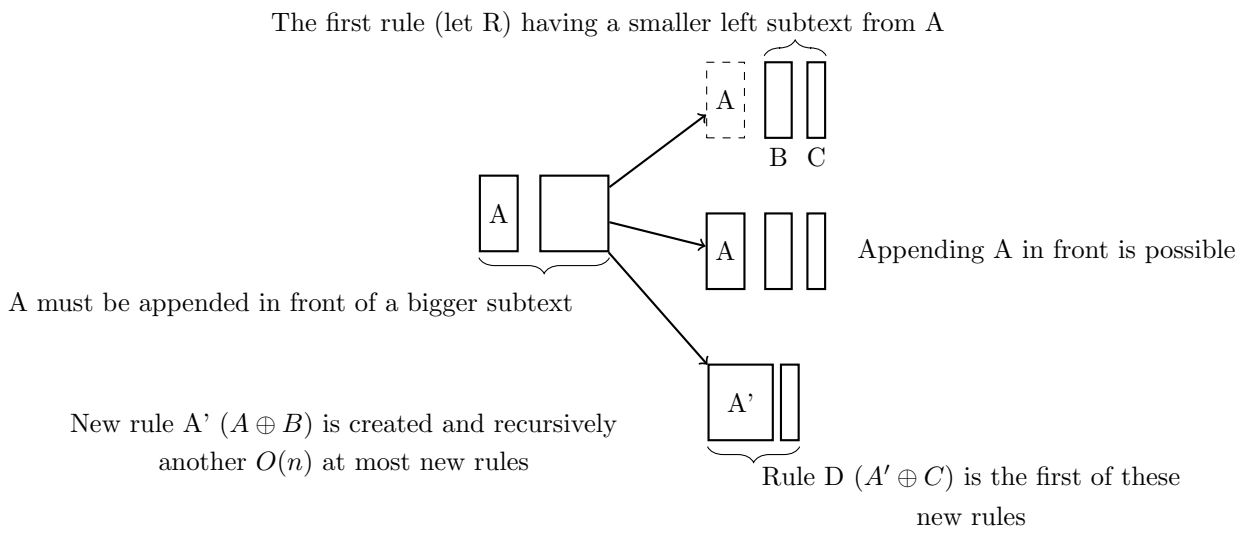
Input: A SLP P1
Output: A SLP P2 with the above properties
for *every rule starting from those with smaller lengths* **do**
  if $length(left\_subtext) \geq length(right\_subtext)$ **then**
    Copy the rule to P2 ;
  else
    Recursively decompose the left subsubtext of the right subtext ;
    if $length(left\_subtext) \geq length(some\_left\_subsubtext)$ **then**
      create new rule appending in front of that subsubtext the left subtext;
      add it just before the rule of the if condition;
    end
    Do not copy the "problematic" rule of P1 to P2;
  end
end

Algorithm 6: P2-Transfrom

The idea of the algorithm is simple. As we can not add smaller subtexts in front of a bigger one, we simply decompose the bigger one's big (left) subtext, until it will be small enough for the right subtex to be appended to it (at most n decompositions). After placing that rule, it is easy to see that the final texts produced is exactly the same. Note that we are able to do this recursive process because the base case (terminal symbols) have the same length. As replacing one "problematic rule" might take up to $O(n)$ time, the total execution time of the P2-Transform algorithm is at most $O(n^2)$ if all rules except the base cases happen to be problematic.

□

Figure 5: P2 transform



Lemma 5.2. *The algorithm R computes the matrices remains and remains2, needed for the transform shown in the examples of the above section in polynomial time.*

Proof. Let two sets of rules X and Y, producing final texts of different size.Our goal is to return 2 matrices of subtexts. The first matrix contains the biggest subtext of X with length smaller than Y and some more subtexts. The sum of lengths of those plus the first one equals that of Y, and they produce the uncompressed text of X up to the $|Y|$-th symbol. The second matrix contains subtexts, the concatenation of which produces the uncompressed text from $|Y|$-th symbol till $|X|$-th symbol of X. W.l.o.g we assume that X,Y have the property of the above lemma

and $length(X) > length(Y)$.

Input: X,Y,length of Y
Output: The two strings of subtexts described above.
remain=[] ;
remain2=[];
length = $length\_of\_Y$;
while *length(Left subtext of X) ¿ length(Y)* **do**
  Decompose(X);
  if *left subtext of X's size is bigger than length* **then**
    Shift to the right the contents of remain2 ;
    Put right subtext of X in remain2[1] ;
    $R(left\_subtext\_of\_X, length)$;
  end
  else if *left subtext of X's size is smaller than length* **then**
    Shift to the right the contents of remain ;
    Put left subtext of X in remain2[1] ;
    $length = length - size\_of\_left\_subtext\_of\_X$;
    $R(right\_subtext\_of\_X, length)$;
  end
  else
    return remain[],remain2[] ;
  end
end

Algorithm 7: The R algorithm

The algorithm tries to find a subtext of X of closest possible size to Y (ideally the same and return). If it fails, it will take the closest subtext found and try to add 1 subtext to get same size. After it tries to add 2 subtexts, 3 etc. Because we hypothesized that the left subsubtext is always longer than the right, each of these stages covers at least half of the size of Y. Consequently after $O(n)$ decompositions, each stage covers half of Y. So $O(n)$ stages are enough ($T(n) = T(\frac{n}{2}) + n$). This algorithm has running time of $O(n)$.

□

6 More questions

Question 1: Will using, instead of the Lifshits blackbox, the Largest Common Subsequence of 2 SLPs (shown to be $P^{NP} - hard$) derive any interesting theo-

retical result? Question 2: If there exists some preacquired knowledge over the SLPs, does that help towards computing the C.H.D? I.e SLPs that compress the DNA of humans are expected to be 99 % equal. Can that information lead to an algorithm determining exactly or even approximating the value between 99 and 100?

7 Heuristics for specific instances

7.1 Using LCS as blackbox

It was shown in [10] that LCS (largest common subsequence) is $P^{NP} - hard$ (Given 2 SLPs. Does a common subsequence of size $\geq n$ exists? n denotes some random size). On first glance, this does not help towards an algorithmic result, but the subsequence as defined by Lifshits is a more general notion. We would prefer a 'sequential' subsequence. This problem is shown by Matsubara et al to be in P [16]. Probably, we could modify the algorithm 2 and change the Lifshits blackbox for a LCS blackbox. In the next section, we will show something simpler.

7.2 Parametrizing approximation based on the length of the LCS

If it happens, that the SLPs have common subsequences of at most constant size (compared to the size of the SLP) then a very simple algorithm can return a constant approximation to the H.D.

Input: SLP1,SLP2
Output: Approximation of H.D.
$t = |eval(SLP1)|$;
m = Matsubara(SLP1,SLP2) ;
`/* Consider m constant to the size of the SLP */`
return $\lfloor \frac{t}{m} \rfloor$;

Algorithm 8: The Blackbox-Algorithm 1

7.3 Complexity of Blackbox-algorith 1

Finding m will take $O(n^4 logn)$. The algorithm assumes naively that the whole text consist of LCSs of constant size. Consequently there is at least a different symbol between them. So the worst case will be if the total of the distance is

equal to the total length subtracted the LCS size. In that case the approximation ratio is given by $r = \frac{2^{|SLP|}/m}{2^{|SLP|}-m} = \frac{1}{m}$. Computing this approximation in this specific input is as difficult as finding the LCS.

7.4 Using dynamic programming again

Another specific instance, where we can probably approximate the C.H.D is when there is a big difference between the cardinality of the same symbol in the two texts. First, we compute in linear time the number of times the two symbols appear in both compressed texts. This can be done with the following algorithm, that fills an array with n rows and 2 columns. Each value in the first column equals the $\#symbol1s$ in the i-th rule and each value in the second column equals the $\#symbol2s$ in the i-th rule. Consequently the following algorithm can fill this array in linear time.

Input: An SLP
Output: The frequency of each symbol in all subtexts F[i][j]
fill $F[0][0], F[0][1], F[1][0], F[1][1]$;
for $i = 2$*;* $i < n$*;* $i++$ **do**
  LeftX,RightX = Decompose(X_i) ;
  $F[i][0] = F[LeftX][0] + F[RightX][0]$;
  $F[i][1] = F[LeftX][1] + F[RightX][1]$;
end
/* frequencies equal $\frac{F[n-1][0]}{max\_length}, \frac{F[n-1][1]}{max\_length}$ */ ;

Algorithm 9: Frequencies of symbols

We can easily create an approximation algorithm by the following simple remark:

Remark 5. *If there are* d *more* $symbol1s$ *in the first SLP. Then the distance is at least d.*

Proof. The proof is straightforward from the fact that those d symbols will surely be compared with $symbol2s$, as the final sizes of the evaluations are equal. □

Thus we can derive a parametrized lower and the upper bound from this remark.

Corollary 7.1. $d \leq C.H.D \leq |size| - d$. *Where size is the total size of the uncompressed text.*

If d happens to be close to $|size| - d$, both bounds are close to each other, and picking one value at random between the interval will be a good approximation.

Figure 6: Dynamic algorith: $Min \quad CHD \leq CHD \leq Max \quad CHD$

The length of the green line corresponds to the number of $\#symbol1$
The length of the red line corresponds to the number of $\#symbol2$
The H.D equals the orange surface

8 Classification and Inapproximability

Definition 2. *$C.H.D \pm n^c$ is the function problem that given 2 SLPs will return a value at most n^c far from the C.H.D.*

Lemma 8.1. *$C.H.D \pm n^c$ belongs in $\#P$.*

Proof. The proof resembles the one in [10] which shown that $C.H.D \in \#P$. We can take the one position comparison function $G(T, S, y) = yes$ if $T_y = S_y$ and modify it so as $G(T, S, y) = yes$ if $T_y = S_y$ and $y \leq |size| - 2n^c$. For $y > |size - 2n^c|$, $G(T, S, y) = yes$ if $G(T, S, y - 1) = no$ and vice versa. This function codes exactly the problem as we defined it above and is polynomially computable as we can 'navigate' through the decomposition tree, knowing the lengths and end up in T_y in polynomial time. □

Lemma 8.2. *$C.H.D \pm n^c \geq C.H.D$ under randomised Cook reductions, under the assumption of an oracle with uniform behavior (We explain this notion below).*

Proof. The problem of $C.H.D \pm n^c$ returns a number 'close' to the value of C.H.D. The problem is that there is no other information derived from the call of the $C.H.D \pm n^c$ oracle. The oracle is deterministic and thus will return the same value once queried with the same 2 SLPs. We will describe a process that will lead to the solution of C.H.D, if the oracle shows a property we name uniform behavior.

Definition 3. *An oracle M of the $C.H.D \pm n^c$ is said to have uniform behavior, if the distribution of the value returned by it, is uniform in $[C.H.D - n^c, C.H.D + n^c]$ on the set of inputs with same C.H.D.*

We will now describe a procedure that in randomised polynomial time would solve the C.H.D problem given the oracle of $C.H.D \pm n^c$ with uniform behavior. M(SLP1,SLP2) will return a value at most n^c far from the real answer. The second step of this procedure, creates 2 new SLPs. SLP1' and SLP2'. SLP1' is equal to SLP1 with the addition of 1 symbol at the end. This can be easily done in constant time. SLP2' is exactly the same with SLP2 with the addition of the same symbol that was appended at the end of SLP1 to make SLP1'. Consequently the distance stays the same, but the deterministic oracle, most likely will return a different value than the one in the previous step. At each step we will keep record of the min and max values returned till now. This procedure (method 1) will continue iteratively until $max - min = 2n^c$. If this condition is met, we know for sure that they correspond to $C.H.D - n^c$ and $C.H.D + n^c$ respectively. Consequently, $C.H.D = max - n^c$. □

8.0.1 Time complexity of method 1

The uniform behavior assumption is necessary to guarantee the reasonable time complexity of our method. It is easy to see, that the problem tackled here is the coupon collector problem with $2n^c$ coupons.

Definition 4. *The coupon collector problem: We want to fill an album with n coupons. The coupons are bought from a merchant, and their packing does not allow the coupon collector to see the coupon before buying it. How many coupons should the coupon collector buy on expectation to fill the album, given that the coupons have equal probabilities of appearence?*

So if we get every coupon (every value in the $[C.H.D - n^c, C.H.D + n^c]$), we will find the desired values and answer the C.H.D problem. It is known that, for n coupons, the coupon collector should buy, on expectation, $n \cdot H_n$ with H_n being the n-th harmonic number. $H(n) = O(logn)$. It is obvious to see that after $n^c \cdot H_{n^c} = O(n^c logn)$. Thus our procedure finishes in reasonable time. In addition, the variance is shown to be $< \frac{\pi^2}{6} n^{2c}$ which is also acceptable.

8.0.2 Implications of method 1

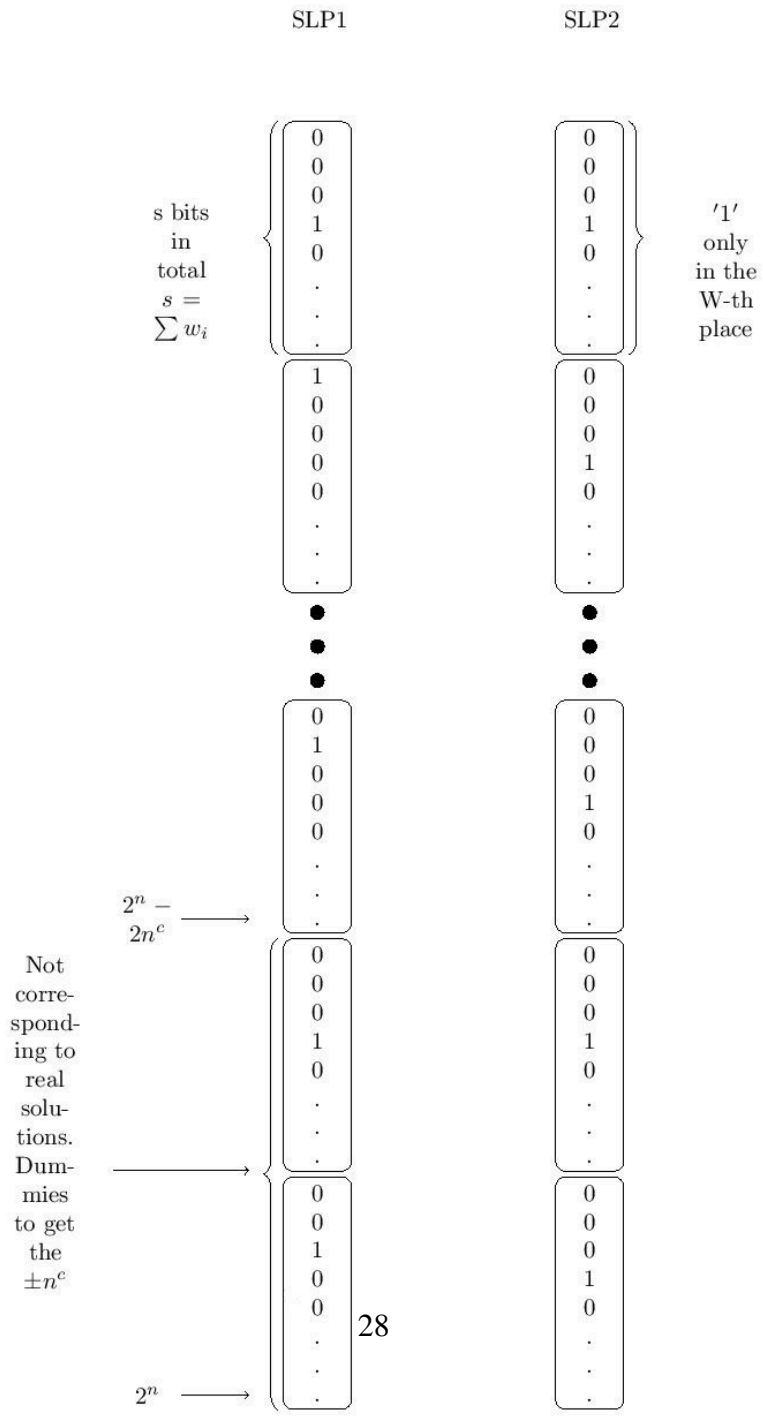
Our method shows that under our assuptions $C.H.D. \in BPP^{C.H.D \pm n^c}$. It would be interesting to answer the question, whether or not the randomisation can be omitted. If the answer is positive, the problem would be shown to be $\#P - complete$. However, even that result would not directly imply anything about the existence of an algorithm approximating well the C.H.D.

Part II
About #Vertex Cover in path and cycle graphs

9 Counting in polynomial time

Sometimes, the counting problem can be solved in polynomial time. Such examples are computing the number of spanning trees, or computing the number

Figure 7: Correctness of the reduction



of perfect matchings in a planar graph. We will show two polynomial time algorithms sloving such problems.

10 Counting Vetrex Covers in a line graph

Although the solution of the problem is already known in the literature [19], our solution has slight ramifications not previously known.

Theorem 10.1.

$$\#VC(n) = \begin{cases} 1 & \textit{if } n = 1\ (only\ 1\ vertex) \\ fibonacci(n+2) & \textit{if } n > 1\ (vertices) \end{cases}$$

We will prove this by induction. Before starting the proof, we note that, by size of the cover we will mean how many vertices are picked. Also, when refering to a vertex as for example the last one, we can imagine that every vertex has an increasing integer as its label.
For the line graph consisting of only 1 vertex, obviously there is only 1 vertex cover.
For the line graph of size 2, again it is easy to notice that there are 3 vertex covers. One of size 2 selecting both vertices of the graph and another 2 symmetric ones of size 1 picking alternately one out of the two vertices.
For every other n sized graph we make the following remark.

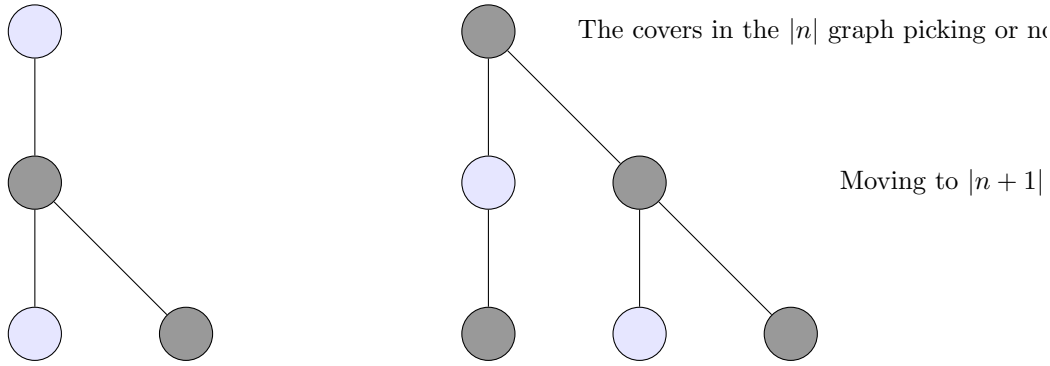
Remark 6. *Given the vertex covers of the $|n-1|$ line graph, we can partition them in two groups. Those that pick the last vertex and those that do not do so.*

Lemma 10.2. *Those covers of the $|n-1|$ line graph, that do not pick the $n-1$th vertex can become covers for the $|n|$ line graph only if the vertex n is added. Otherwise the edge between $n-1$ and n will not be covered.*

Remark 7. *Picking any cover for a $|n|$ graph and dropping the last k vertices will result in a cover for a $|n-k|$ line graph.*

Lemma 10.3. *There is no other cover for the $|n|$ line graph, than those created by the procedure of adding to the covers of the $n-1$ graph the last vertex (or not) according to the above 2 lemmata.*

Figure 8: Creating covers for longer path graphs



Proof. Suppose there was some cover C for the graph $|n|$. This C does not get produced from a $|n-1|$ cover. Let's suppose it gets produced from a $|n-2|$ cover. In this case we take two cases. If vertex $n-2$ is picked then the $|n-2|$ cover is also a $|n-1|$ which is contradictory. If it is not picked then C does not cover the edge $n-2->n-1$ and thus it is not a cover. □

Proof. Let's take the base case of size 2 with 1 cover that does not pick the last vertex and 2 that pick it. Moving to a graph of size 3: The 2 covers picking the last vertex will lead to twice as much covers (the new edge is already covered no matter what) half of them containing the $3rd$ 'new' vertex and half not. As for those (1 here) that did not contain the $n-1$th (2nd) vertex, they will produce a set of vertices with exactly the same cardinality and always containing the last vertex. So for a size 3 graph we have $4+1=5$ covers created. 2 not containing the last vertex and 3 containing it. 5 is the 5th fibonacci number. Following always this process we will produce $2Fib(n)+Fib(n-1)=Fib(n+2)$ covers. □

10.1 Probability of picking a vertex in a cover

Now that the total number of covers is known, we will specify the fraction of covers a specific node takes part. We get the following theorem:

Theorem 10.4. *Let a $|n|$ line graph and a node with index i, $i < n$. Then from the total Fib(n+2) covers and $2 < i < n-1$, node i is not picked in $Fib(index) \cdot Fib(n - index + 1)$ covers.*

Proof. Suppose node i is not picked. For a set of nodes to be a cover, every edge must be covered. Consequently nodes $i-1, i+1$ are surely picked to cover the edges $i-1, i$. Now that edges $i-1, i$ are covered, to guarantee the property a cover, the two subgraphs (node1 to node $i-2$ and node $i+2$ to n) must be covered. The first cover has $Fib(i)$ covers and the second one $Fib(n-i+1)$ covers. Thus, the probability of a node i is not getting picked in a cover is $\frac{Fib(i) \cdot Fib(n-i+1)}{Fib(n+2)}$. For i=1 or i= n and for i=2 or i=n-1 the number of covers is $Fib(n), Fib(n-1)$ respectively. □

10.2 Counting covers of specific sizes

We showed and proved, that the number of vertex covers in a $|n|$ vertex line graph equals the $n+2$ Fibonacci number. This was already known in the litterature. However the way we construct the solution gives us something more.

Theorem 10.5. *The sum of the values in the diagonals of the Pascal triangle equals the Fibonacci numbers.*

Theorem 10.6. *From left to right, the elements of the diagonal correspond to the the number of vertex covers of decreasing sizes for the corresponding line graph.*

Example 1. *The eighth diagonal is $[1, 6, 10, 4]$. Its elements sum to 21 (eighth Fibonacci number, which means it corresponds to a line graph with 6 vertices) and furthermore it states that there is 1 cover of size 6, 6 covers of size 5, 10 covers of size 4 and 4 covers of size 3.*

This is our contribution to the problem and the proof follows.

Lemma 10.7. *There can not be a cover of size less than $\lfloor \frac{n}{2} \rfloor$.*

Proof. Suppose there is a cover with strictly less than the size above. This means there are $\lfloor \frac{n}{2} \rfloor - 1$ chosen vertices and $\lfloor \frac{n}{2} \rfloor + 1$ (+2 for odd n) not chosen vertices. Consequently as there are 2 (3) not chosen vertices more, in the $|n|$ line graph at least 1 (2) edge is not covered. Thus, this can not be a cover. □

Proof. We will now prove the theorem stated above. For n=1, there is only 1 cover is size 1, which is trivial to see (equal to the sum of elements of the second pascal triangle diagonal). Now for $n > 1$ we can prove that inductively. The basis is n=2. For n=2, there is 1 cover of size 2 and 2 covers of size 1. Indeed those covers' numbers can be found on the fourth diagonal of the pascal triangle. Now we will describe the step of the induction. Every value in the pascal triangle is equal to the sum of values in the previous raw on its 'left' and 'right'. In our setting, this will mean that the number of the $|m|$ covers of the $|n|$ graph is equal to the number of the $|m-1|$ covers of the $|n-1|$ graph plus the number of the $|m-1|$ covers of the $|n-2|$ graph, if we add 1 chosen vertex (the nth) to the first and 2 vertices the n-1th chosen and the nth not chosen to the two cases respectively. Notice that by applying those local moves,we just described how the smaller sized covers in a smaller graph become larger in a larger graph. One last thing to prove here is that there are no more vertex covers of size m for the size n graph. This follows from the fact that our way of constructing covers of all sizes guarantess that there are at least x covers of size a and x' of size b and x" of size c ... If somehow there was a cover of some size not constructed by this procedure, this would mean that some other cover of some other size should not exist as the sum of x+x'+x"+... equals the total number of covers. But this is impossible as our method always constructs valid covers from smaller valid ones. Thus we have totally characterized the structure of the problem and the meaning of the 'above' line's left and right values that produced a new value of the Pascal triangle.

□

11 Counting Vertex Covers in a cycle graph

The problem is already tackled in the literature. The following theorem gives the solution to the problem.

Theorem 11.1. *Let a $|n|$ (vertex) cycle. The number of vertex covers is equal to $Fib(n+1) + Fib(n-1)$ which is known as the n-th Lucas number L_n.*

Proof. For n=1 we have only 1 cover. For $n \geq 2$ we will prove it by induction. Induction basis: For n=2, there are 3 covers (2 symmetric ones of size 1 and 1 of size 2). Notice that for n=2, $Fib(3) + Fib(1) = 2 + 1 = 3$. Induction step: Suppose that for size n there are $Fib(n+1) + Fib(n-1)$ covers. We will show that, for size n+1 there are $Fib(n+2) + Fib(n)$ covers.

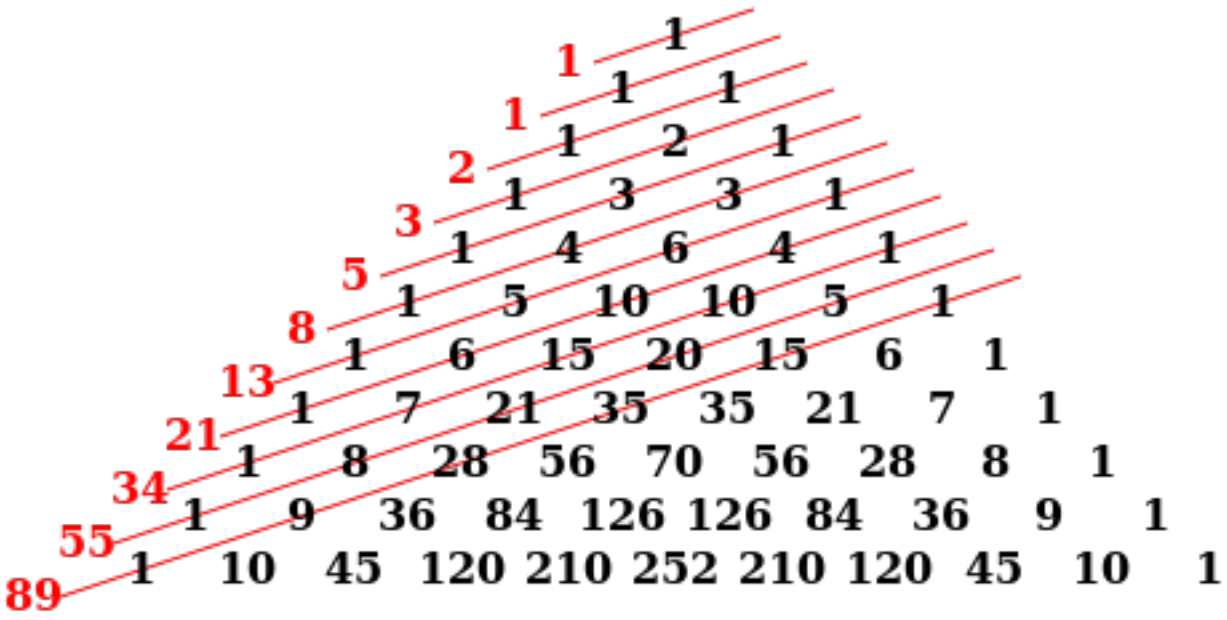


Figure 9: The diagonals of the Pascal triangle

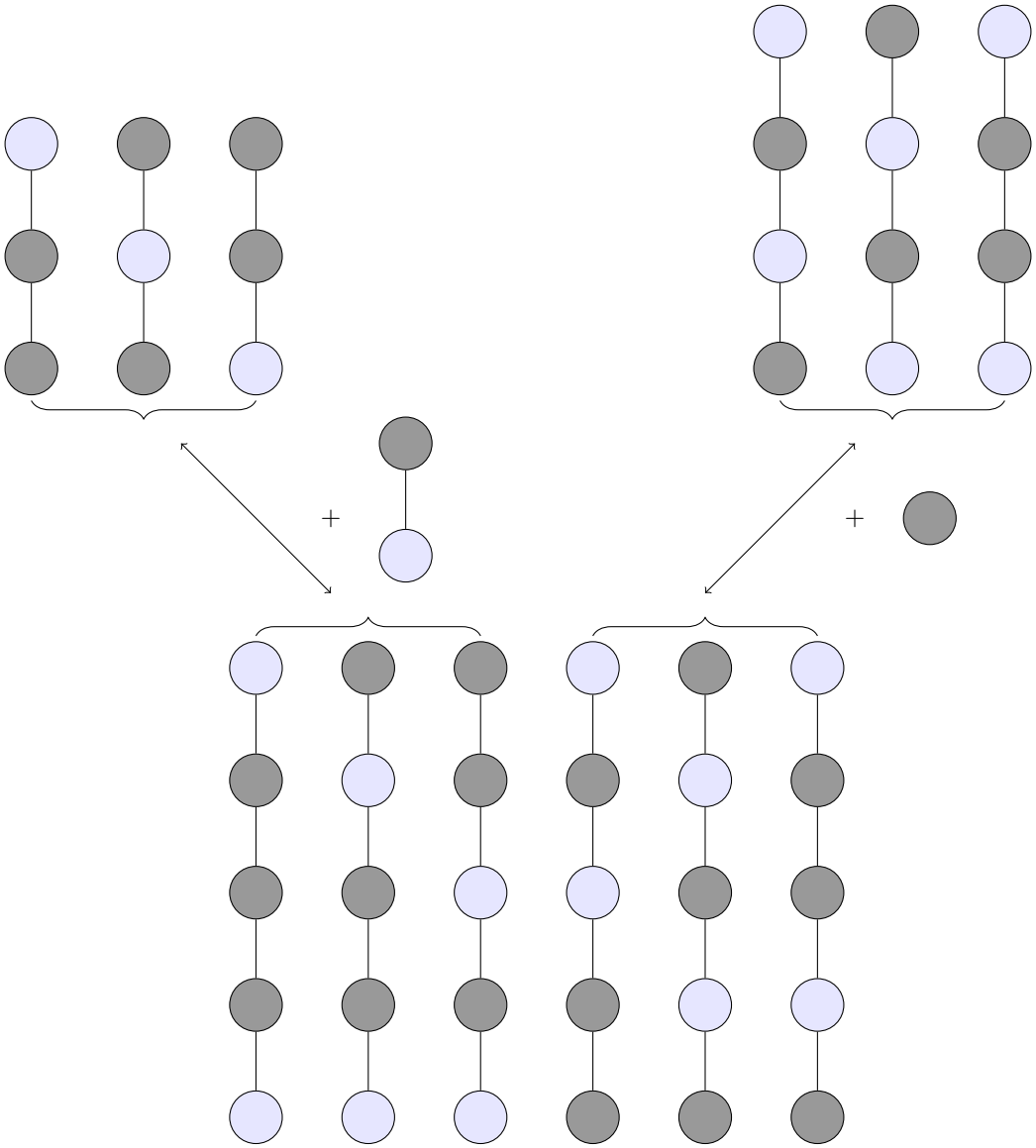


Figure 10: $|2|$ cover for $|3|$ and $|4|$ line graph making $|3|$ covers for $|5|$ line graph

Lemma 11.2. *There are at least $Fib(n+1) + Fib(n-1)$ covers in the $|n+1|$ cycle.*

Proof. Let the last vertex n, that is connected to the first vertex 1. Suppose, w.l.o.g., that the new vertex n+1 is placed between n and 1. If vertex n+1 is picked every cover for the $|n|$ cycle becomes a cover for the $|n+1|$ cycle. □

Lemma 11.3. *There are $Fib(n)$ covers where n+1 is not picked.*

Proof. Let all those that have both $n, 1$ picked. Consequently, the $|n+1|$ vertex may also not be picked. Notice that, those covers differ from every cover mentioned above. Subtracting now vertices $n, n+1, 1$, there is a $|n-2|$ line graph left. Consequently there are $Fib(n)$ covers for it, which produce the covers we described by adding the picked $n, 1$ and not picking $n+1$. □

Lemma 11.4. *There are $Fib(n-2)$ covers produced by non cover structures.*

Proof. If vertex $n+1$ is picked, edges $n-1, n$ are covered ($n-1$ was not covered before). Edges $n-2, 2$ must get somehow covered, and as vertices 1,n are not picked, then vertices $2, n-2$ should absolutely be picked. Subtracting now these 5 vertices $(1, 2, n-1, n, n+1)$ the remaining graph is now a $|n-4|$ line graph with $Fib(n-2)$ covers. Notice again that these covers differ from any cover described above. □

By the above 3 lemmata and the remark that that every possible cover belongs to these 3 categories, we get the proof of the theorem ($Fib(n+1) + Fib(n-1) + Fib(n) + Fib(n-2) = Fib(n+2) + Fib(n)$). See figure below. □

11.1 Probability of picking a vertex in a cover

Here the probability is not parametrised according to the index. It is straight forward from the third lemma of the above section, that between the covers of a $|n|$ cycle, there are $Fib(n-1)$ that do not contain the vertex i.

Theorem 11.5. *The probability of a node i not being picked in a cover of a $|n|$ cycle is equal to $\frac{Fib(n-1)}{Fib(n-1)+Fib(n+1)}$.*

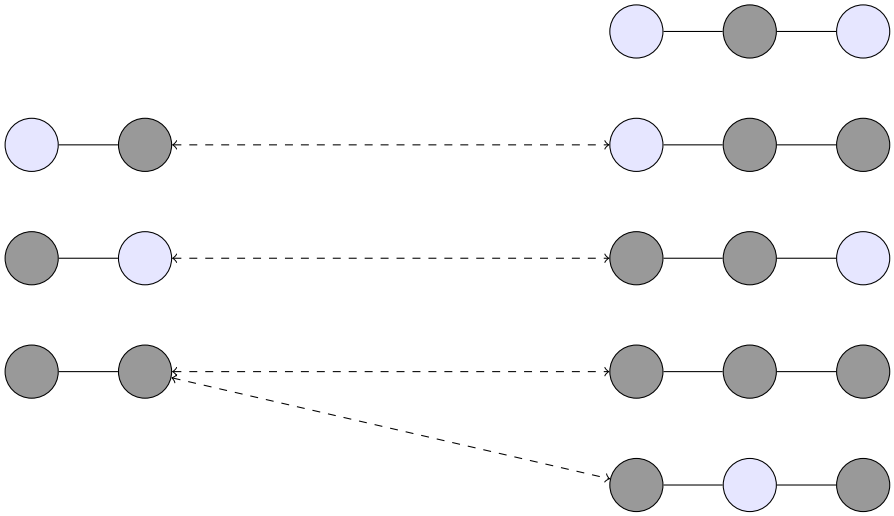


Figure 11: The cases of putting the n+1 vertex between n and 1, that construct covers

11.2 Counting covers of specific sizes

It comes naturally, as in the previous section, to try and find the number of all covers of same size. This subsection tackles this problem.

Lemma 11.6. *The smallest cover is of size* $\lceil \frac{n}{2} \rceil$.

Proof. Anything smaller, by the pigeonhole principle, will produce 2 consecutive nodes not picked and thus it can not be a cover. □

The proof will somehow follow the logic of the proof described above. We will describe 3 ways of creating covers of different sizes, all mutually different. Because there are at least those many covers and it happens that their sum is exactly the one expected, the number of covers attributed of each size is the appropriate one.

Lemma 11.7. *Let a* $|n-3|$ *line graph and a* $|m|$ *cover C. Adding 3 nodes* $n-2, n-1, n$ *leads to a* $|n|$ *cycle. If* $n-2, n$ *are picked then* $C' = C \cup \{n-2, n\}$ *is a cover of size* $|m+2|$.

Proof. Vertices $n-2, n$ cover the edges 'getting out' of the $|n-3|$ cover. They also cover the edges linking them with vertex $n-1$. Consequently C' is a cover. □

Not only C' are covers, but adding +2 we can determine the number of covers of each size, because we can reduce the problem to its counterpart in the line graph case.

Lemma 11.8. *Let a* $|n-2|$ *line graph and a* $|m|$ *cover C. Adding 2 nodes* $n-1, n$ *leads to a* $|n|$ *cycle. If both nodes are picked then* $C'' = C \cup \{n-1, n\}$ *is a cover of size* $|m+2|$.

The proof greatly resembles the previous one. Notice that its cover C' is always different from C" as $n-1$ is never picked in C' covers, but always picked in C" ones. We have omitted the case where vertex n is not picked. We will take care of this case here.

Lemma 11.9. *Let a* $|n-3|$ *line graph and a* $|m|$ *cover C. The line graph contains the vertices* $2, ..., n-2$. *Adding 3 nodes* $n-1, n, 1$ *leads to a* $|n|$ *cycle. If only nodes* $n-1, 1$ *are picked then* $C''' = C \cup \{n-1, 1\}$ *is a cover of size* $|m+2|$.

Proof. C"' is obviously a cover. Also it is easy to see, that C"' is always different from C" and C as they differ in the vertex n.$\#C' + \#C'' + \#C''' = Fib(n-1) + Fib(n) + Fib(n-1) = Fib(n+1) + Fib(n-1)$ □

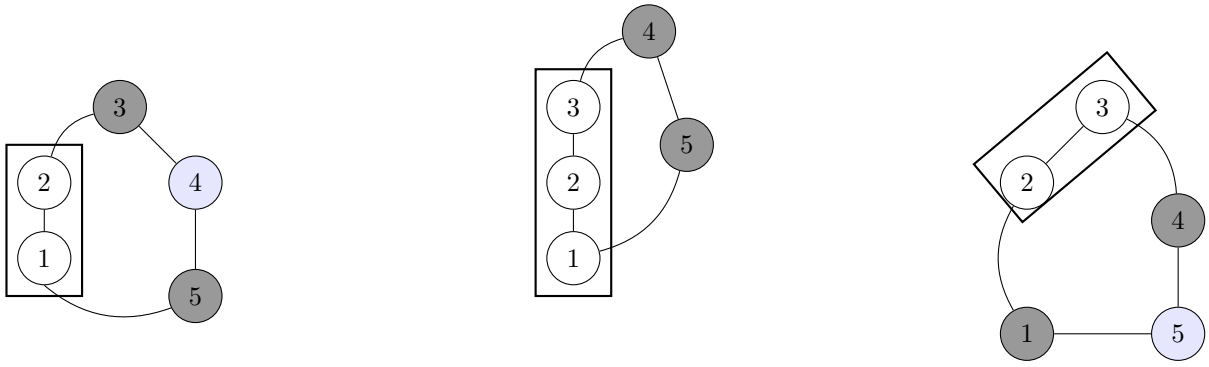


Figure 12: Line graph covers leading to cycle covers of +2 size

Thus, we can compute the number of covers of any specific size by performing the 3 above computations and adding the numbers corresponding to the same sizes. Intuitively, if the diagonals corresponding to the $|n-3|$ and $|n-2|$ graphs have the same number of elements, we can double the elements of the corresponding to the $|n-3|$ graph diagonal and add the corresponding element of the $|n-2|$ graph diagonal, after each element is shifted to the right by 1. If they differ by 1, then we double and add the ones in the same places without shifting. For example, for a 5 vertices cycle we have: $(1,3,1)+(2\cdot 2,1\cdot 2)=(5,5,1)$. 5 $|3|$ covers, 5 $|4|$ covers, 1 $|5|$ cover. On the other hand for a 6 vertices cycle: $(0,3,4,1)+(1\cdot 2,3\cdot 2,1\cdot 2)=(2,9,6,1)$.

12 Counting Vertex covers in a complete binary tree

In this section we will give a function tha counts the number of vertex covers in a full complete binary tree. We regard a tree with a root and 2 children as the

complete tree of size 1. Let the following sequence be T.

$$T(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } i = 2 \\ T^2(i-1) + T^4(i-2) & \text{else} \end{cases}$$

The covers correspond to the $n+3$-th number of this sequence. We will prove this by induction:

Induction Basis: For $i = 1$ note that the $|1|$ full complete binary tree, is exactly the $|3|$ path graph. As shown above this graph has 5 vertex covers.

Example 2. *For $i = 2$: Recall the $i = 1$ case. There are 5 covers. 4 where the root is picked in the cover and 1 where it is not picked. Now, suppose the tree of size 2 (with 7 nodes). It consists of 2 trees of size 1 and a root. If its root is picked then any combination of covers in the left and right subtrees is a valid cover for the new tree. Thus there are 5^2 covers with a picked vertex. Additionaly, only the 4 out of 5 covers can combine between them to create covers for the larger tree. Hence, there are another 4^2 covers. In total there are 25+16=41 covers.*

Induction step: We will generalise the above example. Suppose there are T(n) covers in the tree of height n. Then, by considering 2 n-height full complete binary trees and picking the root, every possible combination of covers leads to a cover. Hence there are at least $T^2(n)$ covers. Additionaly, if the root is not picked, then the 2 roots of the left and right subtrees have to be picked. Then, there are at $T^2(n-2)$ combinations from the left subtree and another $T^2(n-2)$ combinations from the right subtree. All the combinations of these are $T^4(n-2)$. Hence $T(n) = T^2(i-1) + T^4(i-2)$.

13 Impossibility result of counting 'suboptimal' structures

As shown thus far, counting problems are difficult. Hence, we aim for approximate counting. However, another relaxation might be to approximate the quality of the structures counted. For example, counting minimum vertex covers can be difficult, but what about vertex covers, with 3 times the cardinality of the minimum vertex cover? In this paragraph we will show that this problem is difficult.

13.1 The reduction

Suppose the problem gap vertex cover. Let (G, k) be an instance of vertex cover, where G is a graph on n vertices, and k is the threshold. Suppose furthermore that the minimum vertex cover of G has size either at most k or at least ck for some $c > 1$. This version is already NP-hard. Our construction is based on this NP-hard problem.

Consider now n copies of G, with the addition of a clique of size n^D. If G has a minimum vertex cover of size $\ell \leq k$, then the new graph has a vertex cover of size $n\ell + 1$. (ℓ nodes in each of the n copies and 1 node to cover the clique). How many covers can there be at most of size $3(n\ell + 1)$? We need at least nl vertices to cover the n copies. Hence, there are at most all the combinations of picking $n\ell$ nodes of the n copies times picking $2n\ell + 3$ from the clique plus $n\ell + 1$ nodes of the n copies times $2n\ell + 2$ from the clique until only 1 is picked from the clique and the rest $3n\ell + 2$ from the n copies.

If the minimum vertex cover of G is $\ell \geq ck$, then the new graph has a minimum vertex cover of $n\ell + 1$, and at least $\binom{n^D}{2n(ck)+3} \approx n^{2Dcnk+3D}$ vertex covers of size $3(n\ell + 1)$*(1)*. (Having the nodes covering the n copies fixed).

For $D > 3$, the covers in the case where G had a cover of at most k the sum we described is bounded by $(2nk + 3)\binom{n^2}{nk}\binom{n^D}{2nk+3} \leq n^3 n^{2nk} n^{2Dnk+3D} = n^{(2D+2)nk+3D+3}$*(2)*. For D=3, $\frac{(2)}{(1)} = n^{3.7nk-3} = e^{\Omega(n \log n)}$, and thus the gap between the two cases is at least $e^{\Omega(n \log n)}$.

This reduction shows that it is NP-hard to approximate the number of vertex covers of thrice the minimum size within $e^{O(n \log n)}$. If an algorithm returns a value below the 'at most' bound shown above or above the 'at least' bound, then we could deduce an answer for the gap vertex cover problem known to be NP-hard. Otherwise, it is a $e^{\Omega(n \log n)}$ approximation. Even if the value returned is inside the exponential gap between the two cases, the ratio can not be less than $e^{\Omega(n \log n)}/2$.

Part III
Counting algorithms design and Important Results

In the this section we will refer to some important results, mainly concerning the class $\#P$ and then we will roughly present some methods used to derive counting algorithms.

14 Approximate counting

We will begin with a very classic result by Larry Stockmeyer from 1983 [23]. Suppose a circuit $C(X)$ with polynomial sized inputs $x = x_1, x_2, ..., x_n$. How many values satisfy $C(X) = c$ (where c is a certain constant)? This problem is $\#P - complete$. Let $S = \{x|C(x) = c\}$. We want to approximate the $|S|$.

Remark 8. *Approximating $|S|$ by a constant factor say 2, can lead to $1 + \frac{1}{n^c}$ approximation.*

Proof. Suppose there is a 2 approximation. We now contruct a new circuit $C'(x, y) = C(x) \wedge C(y)$. This C' has $|S^2|$ solutions. This means that having a 2 approximation for D, implies that we can have an $\sqrt{2}$ approximation for $|S|$. This can be generalised in a C" with n conjunctions this leads to a $2^{\frac{1}{m}} = 1 + O(\frac{1}{m})$ approximation. □

Notice that a constant approximation is not necessary. In fact, any polynomial approximation will lead to an approximation arbitrary close to the result of a counting function [8]. Imagine a $p(n)$ (polynomial) approximation of a counting function with value a. Creating polynomially many copies of the instance, we have a $p(n)a^{p(n)}$ approximation which can yield a better approximation for a. In fact $(p(n)a^{p(n)})^{\frac{1}{p(n)}} = p(n)^{\frac{1}{p(n)}} a^{p(n)\frac{1}{p(n)}} = (1 + O(\sqrt{\frac{2}{p(n)-1}}))a = (1 + \epsilon)a$. However, a 2 approximation is enough for our result here.

Theorem 14.1. *There is a randomised algorithm that determines $|S|$ to within a factor of 2 in polynomial time, provided it has access to an NP-oracle [23].*

Notice that removing somehow the oracle would imply that $P = NP$, as we would know if the problem has 0 solutions or not.

Proofsketch 1. *We will show something far easier. Suppose we want to know if $|S|$ is really large, or really small. We take a hash function $h : \{0,1\}^n \to \{0,1\}^m$ where $m << n$. Then, notice that, if there are two distinct x,y and both x,y are in $\{S\}$ then there is a collision $h(x) = h(y)$(1) . If $|S|$ is small then, it is likely that (1) will have no solutions, but if $|S|$ is large, then (1) is likely to have solutions. The theorem can be shown after careful analysis of the probabilities of (1) having solutions, for given hash functions. Note that the detection of a collision requires exactly one oracle call: Are there $x, y | x \neq y \wedge h(x) = h(y) = c \wedge x, y \in \{S\}$?*

Another interesting result is the following:

Theorem 14.2. *Suppose that M is a m-state deterministic finite state automaton. Then, the number of length n inputs that are accepted by M can be computed exactly in polynomial time in m, n.*

Proof. We take each step of the FSA (at most n steps). At each step, we count (via dynamic programming) the number of inputs able to reach in (2,3,4..,n steps) each of the m states. After we reach the n-th step, we can add the numbers corresponding to the accepting states [13]. □

14.1 Open problems in this specific area

- Can Stockmeyer result be improved?
- Let a complexity class probably weaker than polynomial time. What is the cost of approximate counting? In the above class the cost of exact counting is polynomial. (Reachability however is known to be logspace complete, but counting s-t paths is $\#P - complete$).

15 Finding specific bits of a counting function

Lemma 15.1. *Given a PP oracle we can determine the most significant bit of a $\#P$ function.*

Lemma 15.2. *Given a $\oplus P$ we can determine the least significant bit of a $\#P$ function.*

Definition 5. *Let function BB(n) be the maximum number of 1s that a n-state Turing machine can leave in a halting computation that starts with a blank tape and uses only the characters 1 and blank.*

Theorem 15.3. *The above problem is easily shown to be undecidable.*

The result is obvious as according to the value of BB(n), answering the halting problem is possible.

Definition 6. *Let function $Bf(n,k)$ be the number of n-state Turing machines, that when run on blank tape halt with at least k-many 1s on the tape.*

Theorem 15.4. *The $Bf(n,k)$ function is uncomputable. However computing $Bf(n,k) \mod m$ for some m is computable (in polynomial time) [13].*

Proofsketch 2. *Let our computation take place in infinite Turing machines, both on the left and on the right. Consecuently every turing machine with computation evolving leftwise has a sibling machine performing the same computation rightwise. There are other symmetries too. The permutation of the halting states for example. Consequently, $Bf(n,k)$ is always a multiple of $m = 2(n-1)!$.*

16 Efficient counting. Fptas and Fpras

According to the writer, one of the most interesting $NP - complete$ problems is the knapsack problem. Recall that it can be solved in time nW. More precisely, we can make a $|weight, items|$ two dimensional array. Each entry of this array is filled according to the following criterion:

$$max(m[i-1,j], m[i-1, j-w[i]] + v[i])$$

Notice that, if we swap the max with a $+$ (an addition of both factors) we get the solution to the counting problem. Both criteria take constant time to be computed and thus the problem of counting solution $\leq W$ can be solved in nW too.

Theorem 16.1. *There is an FPTAS for the 0-1 knapsack problem.*

The above algorithm is known to be pseudopolynomial. That is explained due to the W being part of the time complexity. If W is relatively small, (polynomial to the size of n), then the above algorithm runs in polynomial time. In general, this condition can not be guaranteed. However, if we perform a special scaledown on the weights of the items and then perform the same dynamic programming algorithm considering the modified weights, the quality of the solution and the time complexity are adjusted according to the scaledown performed, yielding a FPTAS [30].

A question that rises naturally is if our previous remark that enabled to count solutions, continues to be valid.

16.1 An FPRAS for $\#knapsack$

Dyer answered the above question [4]. Once again the capacity of the knapsack together with the weights are scaled down by the same value. The new weights are rounded if necessary. Then the dynamic programming algorithm counts the number of solutions. Due to the rounding, the new problem might have more solutions. In the paper, it is shown that for a suitable scaledown, the solutions of the initial problem are modified by a $O(n)$ factor. Now given the exact solution for this new instance, an efficient sampler is constructed and then rejection sampling is used to only sample solutions of the initial problem.

However a problem remains:

Lemma 16.2. *Even an exponentially small variance to the capacity can change the number of solutions up to a constant factor.*

Due to the above, randomisation is needed.

16.2 Another FPRAS for $\#knapsack$

Before presenting another FPRAS about the problem, we will present briefly a very important result, useful for the next algorithms.

Lemma 16.3. *The problems of almost uniform sampling and approximate counting are polynomial time reducible.*

The above has important algorithmic implications [8]. For example, if a random walk on the vertices-solutions converges fast to the uniform distribution and as a result we can uniformly sample efficiently, then this implies the existence of a polynomial time approximation algorithm for the corresponding counting problem.

Proofsketch 3. *Let two sets A, B with $A \subseteq B$. If uniform sampling is possible for B, by repeated sampling we can estimate the probability of the event that the sample is also in A. Hence, the probability $\frac{|A|}{|B|}$ is approximated, which is close to enumerating the elements of A, mainly when counting B is easy, and thus computing the size $|A|$.*

Intuitively, the method is very easy to capture. Imagine a set of people and we want to count the number of those with blond hair. Instead of counting each one, we can pick some uniformly at random from the whole set and check if they have

blond hair or not. If the sample is big enough, our picture about the percentage of blond people is good and thus we can give an expectation of their total number by multiplying it with the whole.

There is another FPRAS, due to Morris and Sinclair [18] about $\#knapsack$ dating from 1999 following a different approach. Let a n-dimentional hypercube. This hypercube has 2^n vertices. Hence, there is a 1-1 correspondance between the powerset of items and the vertices of the cube. Let now a hyperplane trancate the cube. This hyperplane corresponds to the capacity of the knapsack. Thus, all the feasible solutions lie under this hyperplane. Can solutions to the knapsack problem be sampled efficiently? The answer is yes and it is shown by a rapidly mixing markov chain. Morris and Sinclair proposed a random walk starting from a solution. The random walk can be modeled and analysed as a Markov Chain. The last state of the chain is shown to be a solution with a probability asymptotically close to the respective one of the uniform distribution. By the above lemma an FPRAS immediately follows.

16.3 An FPTAS for $\#knapsack$

The matrix that will be filled [22] differ from the folklore one.The one dimension is the subset of weights of size i like the original, but the rows instead of all possible capacities are swapped with the number of solutions a. Finally, each entry consists of an integer which corresponds to the minimum capacity such that there are a solutions with i items. Unfortunately, this function $\tau(i, a)$ can not be computed due to a ranging possibly up to 2^n. However notice that the following function would solve the problem:

$$\tau(i, a) = \min_{\alpha \in [0,1]} max(\tau(i-1, \alpha a), \tau(i-1, (1-\alpha)a) + w_i)$$

Notice the similarity with the dynamic programming algorithm (ignore the min for the moment). α shows that an α fraction does not contains item i and an $1-\alpha$ percentage does not. A second problem appearing here is that α ranges in the continum, and thus computing the min is untractable.

For this reason a T function is defined as follows:

$$T(i, a) = \min_{\alpha \in [0,1]} max(T(i-1, \lceil j + ln_Q \alpha \rceil), T(i-1, \lceil j + ln_Q(1-\alpha) \rceil) + w_i)$$

The number of solutions now corresponds to j. Notice that the second parameter of the T function. $Q = 1 + \frac{\epsilon}{n+1}$ and $s = nlog_Q 2$ which is $O(\frac{n^2}{\epsilon})$. This s value

is where j will range. The second parameter of T is roughly the Q logarithm of the second factor of τ ($ln_Q\alpha, ln_Q(1-\alpha) < 0$).

This T solves both problems. First, j can not reach exponentially large values. And secondly, both $\lceil j+ln_Q\alpha \rceil, \lceil j+ln_Q(1-\alpha) \rceil$ are step functions of α. For every $j \in \{0, 1, ..s\}$ the values in $\{Q^{-j}, ..., Q^0\} \cup \{1-Q^0, .., 1-Q^{-j}\}$ are enough to find the previous intractable min. This solution to the problem of the minimisation over the continum was the only really not trivial part. The answer Z' returned by the algorithm is $Q^{j'+1}$ where $j' = max\{j : T[n, j] \leq C\}$. Finally, it is shown that, $\tau(i, Q^{j-i}) \leq T[i, j] \leq \tau(i, Q^j)$. This lasts inequality shows that

$$\frac{Z'}{OPT} \leq \frac{Q^{j'+1}}{Q^{j'-n}} = Q^{n+1} \leq e^{\epsilon}$$

.

17 Counting and Quantum Computation

We will very briefly mention some results showing theoretic interest:

Theorem 17.1. *The 'quantum' part of every quantum computation can be replaced by the approximate evaluation of the Jones polynomial over a braid [17].*

The Jones polynomial is a polynomial used in knot theory. Computing exactly this polynomial is a $\#P - complete$ problem. However even a good approximation of this polynomial would be enough and it would imply that $BQP \subseteq P^A$ where A is an oracle that returns this approximation.

Theorem 17.2. *For any quantum turing machine M running in time t(n) [6], there is a function $f \in GapP$ such that*

$$Pr(M(x)accepts) = \frac{f(x)}{5^{2t(|x|)}}$$

However, evaluating a GapP function is computationally intractable in general. Would an approximation suffice?

Theorem 17.3. *In both cases a RAA would suffice [1].*

A RAA is a randomised Additive Approximation algorithm. It is like an FPRAS but it is weaker in the approximation ratio ($\epsilon u|I|$ against $\epsilon f(|I|)$ respectively. u is some upper bound of the function.

18 Counting and Parametrized complexity

A famous result by Valiant was the $\#P-completeness$ of $\#perfect-matchings$ in a bipartite graph. This result was astonishing because the decision version of the problem belonged in P. Naturally, the following question is yield: Is there a problem, that exhibits the same behavior in parametrized complexity? (Decision in FPT and counting in $\#W[1]$ or some other counting class?)

The answer is yes. Finding a k-path (simple path of length k) in a graph is in FPT [21]. However, counting k-paths is $\#W[1] - hard$ [5]. What's more interesting is that there are problems in P with counting versions not only $\#P - hard$ but also $\#W[1] - hard$. Such a problem is the well known counting k-matchings in a bipartite graph.

After intense search it turned out that the most appropriate notion of approximation algorithm for counting problems is an FPRAS. Naturally, the notion was generalized to an FPTRAS when there is a need to approximate count the parametrized versions of problems. There are some FPTRASes in the literature like $p - \#k - paths$ and $p - \#k - cycles$ [5] and $p - \#k - matchings$.

19 Holographic Algorithms and matchgate computations

This is an algorithmic design technique, which was introduced by Leslie Valiant [27] and produced some exotic P algorithms for counting problems.

Roughly speaking, the technique has 3 steps. First of all, a holographic reduction must be constructed. This specific type of reduction is a many to many type of reduction that has one interesting property. It preserves the number of solutions. What is more interesting is that it is not a simple parsimonious reduction as it may not always map solutions to solutions and guarantess only to preserve the number of solutions between the two problems. In the Valiant framework, the problem reduced to is a specific constraint problem where every constraint is realizable by matchgates.

Definition 7. *Matchgates are a specific quantum circuit. One interesting property of matchgates is that, when computing functions using only neighbour qubits, they can be efficiently simulated by classical circuits [26, 9].*

Finally, there is a second reduction from this matchgate problem to the problem of computing the perfect matchings of a planar graph.This problem belongs

in P. The FKT (Fisher Kasteleyn Temperley) algorithm [24], a rather exotic algorithm, computes in polynomial time the solution to the $\#perfect - matchings$ in planar graphs problem.

The first reduction uses Pfaffians to encode and process the information needed.

Definition 8. *The determinant of a skew-symmetric matrix ($-A = A^T$) can always be written as the square of a polynomial in the matrix entries. This polynomial is called the Pfaffian of the matrix.*

The second reductions creates an exponential sum that achieves exponential cancellations and thus is computed efficiently.

What is more interesting is that Valiant found actual problems where this framework would yield a polynomial algorithm, after the creation of the framework [28]. More precisely, for the following two problems, an appropriate holographic reduction is proved to exist: $\#_7Pl, Rtw, Mon, 3CNF, \#_7Pl, 3/2Bip, VC$. Surprisingly, the same problems $mod2$ were shown to be $\#P - complete$. Recently Cai [2] proved a more general result showing that the above problems modulo any Mersenne prime belong in P. He also generalised the framework, reducing the problem through the holographic reduction to a more general problem efficiently computed by Fibonacci gates [3]. The second reduction also changes, reducing the "Fibonacci gates" problem to an artificial edge coloring problem having a counting version which once again belongs in P. This improvement makes it possible to count structures not necessarily related to planar graphs (see above).

References

[1] Magnus Bordewich, Michael H. Freedman, László Lovász, and D. Welsh. Approximate counting and quantum computation. *CoRR*, abs/0908.2122, 2009.

[2] Jin-yi Cai and Pinyan Lu. Holographic algorithms: From art to science. *J. Comput. Syst. Sci.*, 77(1):41–61, 2011.

[3] Jin-yi Cai, Pinyan Lu, and Mingji Xia. Holographic algorithms by fibonacci gates and holographic reductions for hardness. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 644–653, 2008.

[4] Martin E. Dyer. Approximate counting by dynamic programming. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 693–699, 2003.

[5] Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004.

[6] Lance Fortnow and John D. Rogers. Complexity limitations on quantum computation. *J. Comput. Syst. Sci.*, 59(2):240–252, 1999.

[7] M. Jerrum. *Counting, Sampling and Integrating: Algorithms and Complexity*. Lectures in Mathematics. ETH Zürich. SPRINGER VERLAG NY, 2003.

[8] Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.

[9] Richard Jozsa and Akimasa Miyake. Matchgates and classical simulation of quantum circuits. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 464(2100):3089–3106, 2008.

[10] Yury Lifshits. Processing compressed texts: A tractability border. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings*, pages 228–240, 2007.

[11] Yury Lifshits and Markus Lohrey. Querying and embedding compressed texts. In *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, pages 681–692, 2006.

[12] Chengyu Lin, Jingcheng Liu, and Pinyan Lu. A simple FPTAS for counting edge covers. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 341–348, 2014.

[13] Richard J. Lipton and Kenneth W. Regan. *People, Problems, and Proofs - Essays from Gödel's Lost Letter: 2010*. Springer, 2013.

[14] Markus Lohrey. Word problems and membership problems on compressed words. *SIAM J. Comput.*, 35(5):1210–1240, 2006.

[15] Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[16] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009.

[17] Michael J. Larsen Michael H. Freedman, Alexei Kitaev and Zhenghan Wang. Topological quantum computation. *Bull. Amer. Math. Soc. 40 (2003), 31-38*, 40:31–38, 2003.

[18] Ben Morris and Alistair Sinclair. Random walks on truncated cubes and sampling 0-1 knapsack solutions. *SIAM J. Comput.*, 34(1):195–226, 2004.

[19] S. D. Noble. Evaluating a weighted graph polynomial for graphs of bounded tree-width. *Electr. J. Comb.*, 16(1), 2009.

[20] Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[21] Hadas Shachnai and Meirav Zehavi. Representative families: A unified tradeoff-based approach. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 786–797, 2014.

[22] Daniel Stefankovic, Santosh Vempala, and Eric Vigoda. A deterministic polynomial-time approximation scheme for counting knapsack solutions. *SIAM J. Comput.*, 41(2):356–366, 2012.

[23] Larry J. Stockmeyer. On approximation algorithms for #p. *SIAM J. Comput.*, 14(4):849–861, 1985.

[24] H. N. V. Temperley and Michael E. Fisher. Dimer problem in statistical mechanics-an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.

[25] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[26] Leslie G. Valiant. Quantum circuits that can be simulated classically in polynomial time. *SIAM J. Comput.*, 31(4):1229–1254, 2002.

[27] Leslie G. Valiant. Holographic algorithms (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 306–315, 2004.

[28] Leslie G. Valiant. Accidental algorithms. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 509–517, 2006.

[29] Dror Weitz. Counting independent sets up to the tree threshold. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 140–149, 2006.

[30] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.