Illustrated Introduction to the Truncated Fourier Transform

Paul Vrbik*
Department of Computer Science
The University of Western Ontario
London, Ontario, N6A 5B7
pvrbik@csd.uwo.ca

November 7, 2011

Abstract

I summarize the univariate parts of two papers by Joris van der Hoeven's. These papers introduce the Truncated Fourier Transform (TFT) which is a variation of the Discrete Fourier Transform (FFT) that allows one to work with input vectors that are *not* a power of two. I offer many illustrations to accompany the mathematics.

1 Introduction

Fundamentally, the Discrete Fourier Transform (DFT) is a process that allows for the rapid evaluation of polynomials at points. The efficient implementation of the DFT as an algorithm is given by the Fast Fourier Transform (FFT). These FFTs are so commonly used that the term "FFT" is often mistakingly used for "DFT". These FFTs have major applications in at least two (fairly disjoint) research disciplines: Signal Processing and Computer Algebra.

For instance, let $\mathcal{R}$ be a ring of constants with $2 \in \mathcal{R}$ a unit. If $\mathcal{R}$ has a primitive nth root of unity ω with $n = 2^p$ (i.e. $\omega^{n/2} = -1$) then the Fast Fourier Transform can be used to compute the product of two polynomials $P, Q \in \mathcal{R}[x]$ with $\deg(PQ) < n$ in $O(n \log n)$ operations in $\mathcal{R}$. However, when $\deg(PQ)$ is sufficiently far from a power of two *many* computations are done to establish evaluation points that are ultimately not needed.

This problem was solved by the signal processing community by using a method called FFT-pruning [3]. However, the difficult inversion of this process, is due to van der Hoeven [4][5].

I outline the DFT, including a method for its non-recursive implementation, and then develop the "pruned" variant — which we call the Truncated Fourier Transform (TFT). Finally I show how the TFT can be inverted.

*This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

2 The Discrete Fourier Transform

Let $\mathcal{R}$, n, and ω be given as in the introduction. The Discrete Fourier Transform*, with respect to ω, of an n-tuple $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \mathcal{R}^n$ is the n-tuple $\widehat{\mathbf{a}} = (\widehat{a}_0, \ldots, \widehat{a}_{n-1}) \in \mathcal{R}^n$ with

$$\widehat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

Alternatively we can see these n-tuples as containing the coefficients of polynomials from $\mathcal{R}[x]$ and define the DFT wrt ω as the mapping which takes $A = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ to the n-tuple $(A(\omega^0), \ldots, A(\omega^{n-1}))$. We denote this by:

$$\mathrm{DFT}_\omega\,(a_0, \ldots, a_n) = (A(\omega^0), \ldots, A(\omega^{n-1})).$$

The DFT can be computed efficiently using binary splitting. This method requires that we evaluate only at ω^{2^i} for $i \in \{0, \ldots, p-1\}$, rather than at all $\omega^0, \ldots, \omega^{n-1}$. To compute the DFT of $\mathbf{a}$ with respect to ω we write

$$(a_0, \ldots, a_{n-1}) = (b_0, c_0, \ldots, b_{n/2-1}, c_{n/2-1})$$

and recursively compute the DFT of $(b_0, \ldots, b_{n/2-1})$ and $(c_0, \ldots, c_{n/2-1})$ wrt ω^2:

$$\begin{aligned}\mathrm{DFT}_{\omega^2}(b_0, \ldots, b_{n/2-1}) &= (\widehat{b}_0, \ldots, \widehat{b}_{n/2-1});\\ \mathrm{DFT}_{\omega^2}(c_0, \ldots, c_{n/2-1}) &= (\widehat{c}_0, \ldots, \widehat{c}_{n/2-1}).\end{aligned}$$

Finally we construct $\widehat{\mathbf{a}}$ according to

$$\begin{aligned}\mathrm{DFT}_\omega(a_0, \ldots, a_{n-1}) = (&\widehat{b}_0 + \widehat{c}_0, \ldots, \widehat{b}_{n/2-1} + \widehat{c}_{n/2-1}\omega^{n/2-1}\\ &\widehat{b}_0 - \widehat{c}_0, \ldots, \widehat{b}_{n/2-1} - \widehat{c}_{n/2-1}\omega^{n/2-1}).\end{aligned}$$

Equivalently, using the polynomial interpretation, split A into its even and odd parts, evaluate each part at ω^2 and then reconstruct to retrieve $\widehat{A}$.

Clearly this description has a natural implementation as a recursive algorithm; but, in practice it is sometimes more efficient to implement an in-place algorithm that eliminates the overhead of creating recursive stacks:

Definition 1. We denote by $[i]_p$ the bitwise reverse† of i at length p. Suppose $i = i_0 2^0 + \cdots + i_p 2^p$ and $j = j_0 2^0 + \cdots + j_p 2^p$ with $i_\ell, j_\ell \in \{0, 1\}$ then

$$[i]_p = j \iff i_k = j_{p-k} \text{ for } k \in \{0, \ldots, p\}.$$

Example 1. Denote by x_2 a number x written in binary.

$[3]_5 = 24$ because $3 = 00011_2$ whose reverse is $11000_2 = 24$.

$[11]_5 = 26$ because $11 = 01011_2$ whose reverse is $11010_2 = 26$.

*In signal processing community this is called the "decimation-in-time" variant of the FFT.

†In [4] the word "mirror" instead of reverse is used.

For the non-recursive (in-place) DFT algorithm we require *one* vector of length n. Initially, at step zero, this vector is

$$\mathbf{x_0} = (x_{0,0}, \ldots, x_{0,n-1}) = (a_0, \ldots, a_{n-1})$$

and is updated (incrementally) at steps $s \in \{1, \ldots, p\}$ by the rule

$$\begin{bmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{bmatrix} = \begin{bmatrix} 1 & \omega^{[i]_s m_s} \\ 1 & -\omega^{[i]_s m_s} \end{bmatrix} \begin{bmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{bmatrix} \tag{1}$$

for all $i \in \{0, \mathbf{2}, \ldots, n/m_s - 2\}$ and $j \in \{0, \ldots, m_s - 1\}$, where $m_s = 2^{p-s}$.

Equation (1), being a relation among four values at the two steps s and $s-1$, can be illustrated as in Figure 1. We call this relation a "butterfly" after the shape it forms. We may say that m_s controls the width of this butterfly — the value of which decreases as s increases. Note that two

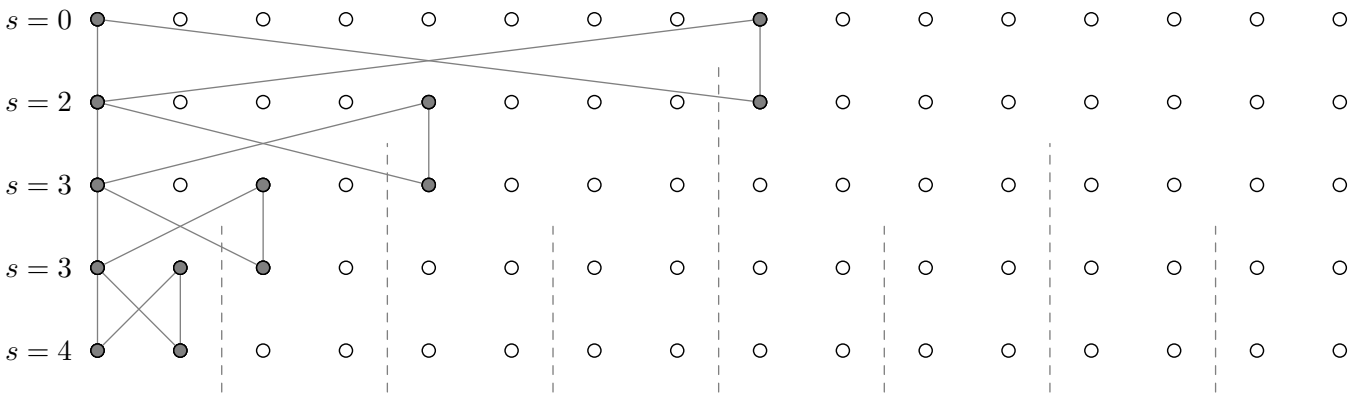


Figure 1: Butterflies. Schematic representation of Equation (1). Solid dots represent the $x_{s,i}$. The top row corresponds to $s = 0$. In this case $n = 16 = 2^4$.

additions and *one* multiplication are done in Equation (1) as one product is merely the negation of the other.

Using induction over s, it can be shown that

$$x_{s,im_s+j} = \left(\mathrm{DFT}_{\omega^{m_s}}(a_j, a_{m_s+j}, \ldots, a_{n-m_s+j})\right)_{[i]_s},$$

for all $i \in \{0, \ldots, n/m_s - 1\}$ and $j \in \{0, \ldots, m_s - 1\}$ [4] . In particular, when $s = p$ and $j = 0$ we have

$$x_{p,i} = \widehat{a}_{[i]_p}$$
$$\widehat{a}_i = x_{p,[i]_p}$$

for all $i \in \{0, \ldots, n-1\}$. That is, $\widehat{\mathbf{a}}$ is a (specific) permutation of $\mathbf{x_p}$ as illustrated in Figure 2.

The key property of the DFT is that it is straightforward to invert (i.e. to recover $\mathbf{a}$ from $\widehat{\mathbf{a}}$).

$$\mathrm{DFT}_{\omega^{-1}}(\widehat{\mathbf{a}})_i = \mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_{\omega}(\mathbf{a}))_i = \sum_{k=0}^{n-1}\sum_{j=0}^{n-1} a_i \omega^{(i-k)j} = n a_i \tag{2}$$

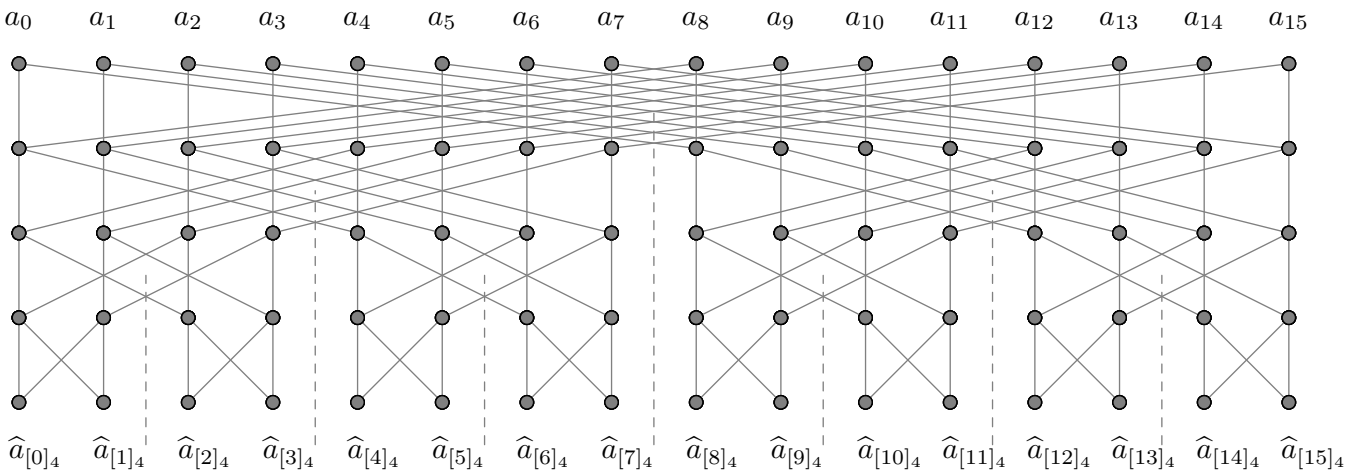


Figure 2: The Discrete Fourier Transform for $n = 16$. The top row, corresponding to $s = 0$, represents the values of $\mathbf{x_0}$. The bottom row, corresponding to $s = 4$, is a permutation of $\widehat{\mathbf{a}}$ (the result of the DFT on $\mathbf{a}$).

since

$$\sum_{j=0}^{n-1} \omega^{(i-k)j} = 0$$

whenever $i \neq k$. This yields a polynomial multiplication algorithm that does $O(n \log n)$ operations in $\mathcal{R}$. For the sake of brevity I will refer the reader to [1, §4.7] for the outline of this algorithm.

3 The Truncated Fourier Transform

The motivation behind the Truncated Fourier Transform[‡] is the observation that many computations are wasted when the length of $\mathbf{a}$ (the input) is not a power of two. This is entirely the fault of the strategy where one "completes" the ℓ-tuple $\mathbf{a} = (a_0, \ldots, a_{\ell-1})$ by setting $a_i = 0$ when $i \geq \ell$ to artificially extend the length of $\mathbf{a}$ to the nearest power of two (so the DFT can be executed as usual).

However, despite the fact that we may only want ℓ components of $\widehat{\mathbf{a}}$, the DFT will calculate *all* of them. Thus computation is wasted. I illustrate this in Figures 3 and 4. This type of wasted calculation is relevant when using the DFT to multiply polynomials — their products are rarely of degree one less some power of two.

The definition of the TFT is similar to that of the DFT with the exception that the input and output vector ($\mathbf{a}$ resp. $\widehat{\mathbf{a}}$) are not necessarily of length some power of two. More precisely the TFT of an ℓ-tuple $(a_0, \ldots, a_{\ell-1}) \in \mathcal{R}^\ell$ is the ℓ-tuple

$$\left(A(\omega^{[0]_p}), \ldots, A(\omega^{[\ell-1]_p})\right) \in \mathcal{R}^\ell.$$

where $n = 2^p$, $\ell < n$ (usually $\ell \geq n/2$) and ω a n-th root of unity.

[‡]The TFT is exactly equivalent to a technique called "FFT pruning" in the signal processing literature [3].

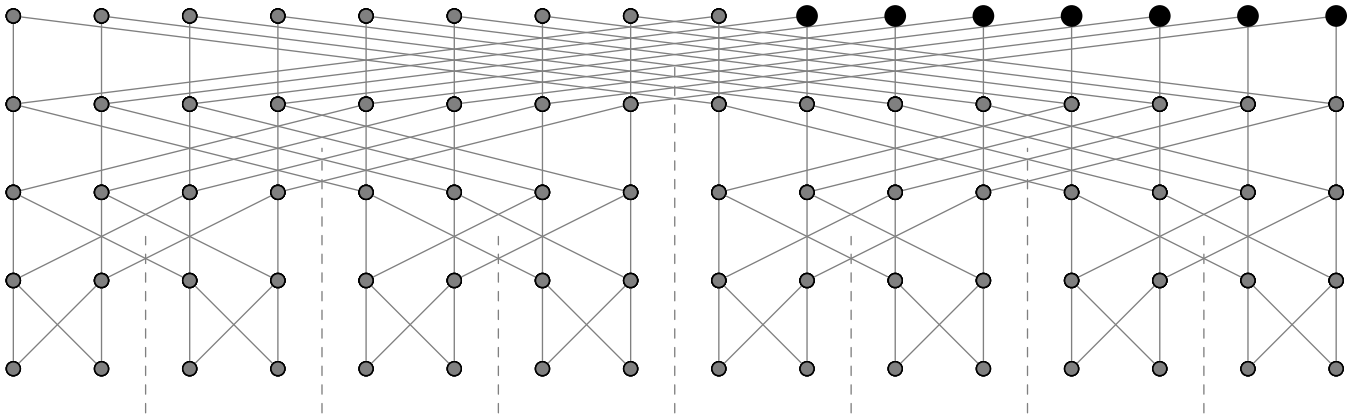


Figure 3: The DFT with "artificial" zero points (large black dots).

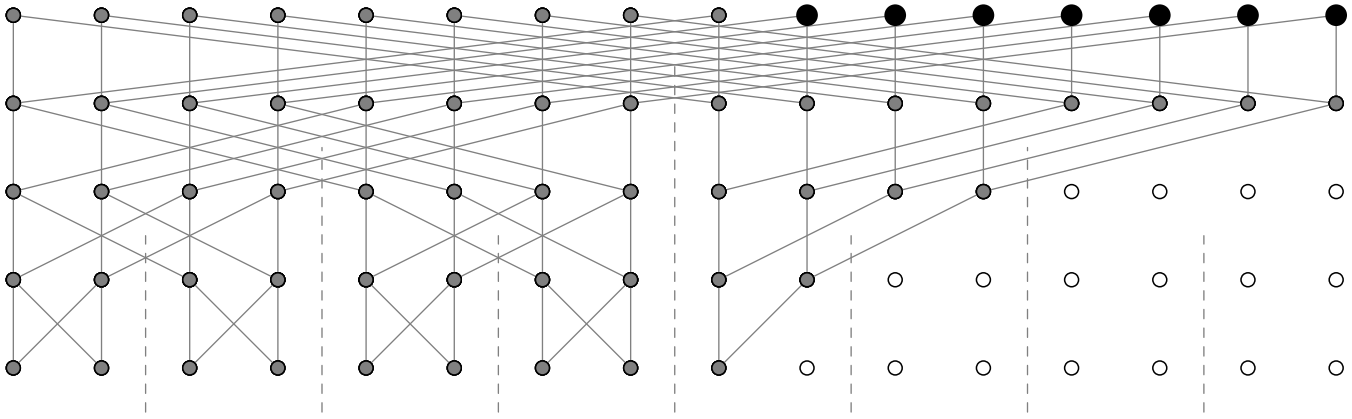


Figure 4: Removing all unnecessary computations from Figure 3 gives the schematic representation of the TFT.

Remark 1. van der Hoeven gives a more general description of the TFT where one can chose an initial vector $(x_{0,i_0}, \ldots, x_{0,i_n})$ and target vector $(x_{p,j_0}, \ldots, x_{p,j_n})$. Provided that the i_k's are distinct one can carry out the TFT by considering the full DFT and removing all computations not required for the desired output. For this paper I restrict my discussion to that of the scenario in Figure 4 (where the input and output are the same initial segments) because it is can be used for polynomial multiplication, and because it yields the most improvement.

If we allow ourselves to operate in a size n array it is actually straightforward to modify the in-place DFT algorithm from the previous section to instead execute the TFT. (It should be emphasized that this only saves computation; not space. For a "true" in-place TFT algorithm that operates in an array of size ℓ, see Harvey and Roche's work in [2].) At stage s it suffices to compute

$$(x_{s,0}, \ldots, x_{s,j}) \text{ with } j = \lceil \ell/m_s \rceil m_s - 1$$

where $m_s = 2^{p-s}$.[§]

Theorem 1. *Let $n = 2^p$, $1 \leq \ell < n$ and $\omega \in \mathcal{R}$ be a primitive n-th root of unity in $\mathcal{R}$. Then the TFT of an ℓ-tuple $(a_0, \ldots, a_{\ell-1})$ wrt ω can be computed using at most $\ell p + n$ additions and $\lfloor (\ell p + n)/2 \rfloor$ multiplications of powers of ω.*

Proof. Let $j = (\lceil \ell_s/m_s \rceil)m_s - 1$; at stage s we compute $(x_{s,0}, \ldots, x_{s,j})$. So, in addition to $x_{s,0}, \ldots, x_{s,\ell-1}$ we compute

$$(\lceil \ell/m_s \rceil)m_s - 1 - \ell \leq m_s$$

more values. Therefore, in total we compute at most

$$\begin{aligned} p\ell + \sum_{s=1}^{p} m_s &= p\ell + 2^{p-1} + 2^{p-2} + \cdots + 1 \\ &= p\ell + 2^p - 1 < p\ell + n \end{aligned}$$

values $x_{s,i}$. The result follows from this. □

4 Inverting The Truncated Fourier Transform

Unfortunately, the inverse TFT cannot be inverted by merely doing another TFT with $1/\omega$ and adjusting the output by some constant factor (like the DFT). Simply put: we are missing information and have to account for this.

Example 2. Let $\mathcal{R} = \mathbb{Z}/13\mathbb{Z}$, $n = 2^2 = 4$, with $\omega = 5$ a n-th primitive root of unity. Setting $A = a_0 + a_1 x + a_2 x^2$, the TFT of $\mathbf{a} = (a_0, a_1, a_2)$ is

$$\begin{bmatrix} A(\omega^0) \\ A(\omega^2) \\ A(\omega^1) \end{bmatrix} = \begin{bmatrix} A(1) \\ A(-1) \\ A(5) \end{bmatrix} = \begin{bmatrix} a_0 + a_1 + a_2 \\ a_0 - a_1 + a_2 \\ a_0 + 5a_1 - a_2 \end{bmatrix}$$

[§]This is a correction to the bound given in [5] as pointed out in [4].

Now to show that the TFT of this w.r.t. $1/\omega$ is *not* $\mathbf{a}$ define

$$\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_0 + a_1 + a_2 \\ a_0 - a_1 + a_2 \\ a_0 + 5a_1 - a_2 \end{bmatrix}$$

The TFT of $\mathbf{b}$ w.r.t $1/\omega = -5$ is

$$\begin{bmatrix} B(\omega^0) \\ B(\omega^{-2}) \\ B(\omega^{-1}) \end{bmatrix} = \begin{bmatrix} B(1) \\ B(-1) \\ B(5) \end{bmatrix} = \begin{bmatrix} b_0 + b_1 + b_2 \\ b_0 - b_1 + b_2 \\ b_0 - 5b_1 - b_2 \end{bmatrix} = \begin{bmatrix} 3a_0 + 5a_1 + a_2 \\ a_0 - 6a_1 - a_2 \\ -5a_0 + a_1 - 3a_2 \end{bmatrix}$$

which is not some constant multiple of $\mathrm{TFT}_\omega(\mathbf{a})$.

This discrepancy is caused by the completion of $\mathbf{b}$ to $(b_0, b_1, b_2, 0)$. Namely we should instead complete b to $(b_0, b_1, b_2, A(-5))$ to correspond to the DFT of $\mathbf{a}$ w.r.t ω.

Essentially to invert the TFT we follow the paths from $\mathbf{x_p}$ back to $\mathbf{x_0}$. We will use the fact that whenever two values among

$$x_{s,im_s+j}, x_{s-1,im_s+j}$$

and

$$x_{s,(i+1)m_s+j}, x_{s-1,(i+1)m_s+j}$$

are known then we can deduce the other values. That is, if two values of some butterfly are known then the other two values can be calculated using Equation (1) as the relevant matrix is invertible. Moreover, these relations only involve shifting (multiplication and division by two), additions, subtractions and multiplications by roots of unity — an ideal scenario for implementation.

Again observe (as with DFT) that $x_{p-k,0}, \ldots, x_{p-k,2^k-1}$ can be calculated from $x_{p,0}, \ldots, x_{p,2^k-1}$. This is because all the butterfly relations necessary to move up like this never require $x_{s,2^k+j}$ for any $s \in \{p-k, \ldots, p\}$, $j > 0$. This is illustrated in Figure 5. More generally we have that

$$x_{p,2^j+2^k}, \ldots, x_{p,2^j+2^k-1}$$

is sufficient information to compute

$$x_{p-k,2^j}, \ldots, x_{p-k,2^j+2^k-1}$$

provided that $0 < k \leq j < p$. (In Algorithm 1, that follows, we call this a "self-contained push up").

5 Inverse TFT Algorithm

We present a simple recursive description of the inverse TFT algorithm for the case we have restricted ourselves to (all zeroes packed at the end). The algorithm operates in a length n array $\mathbf{x} = (\mathbf{x}_0, \ldots, \mathbf{x}_{n-1})$ for which we assume access to ($n = 2^p$ corresponds to ω, a n-th primitive root of unity). Initially, the contents of the array is

$$\mathbf{x} = (x_{p,0}, \ldots, x_{p,\ell-1}, 0, \ldots, 0)$$

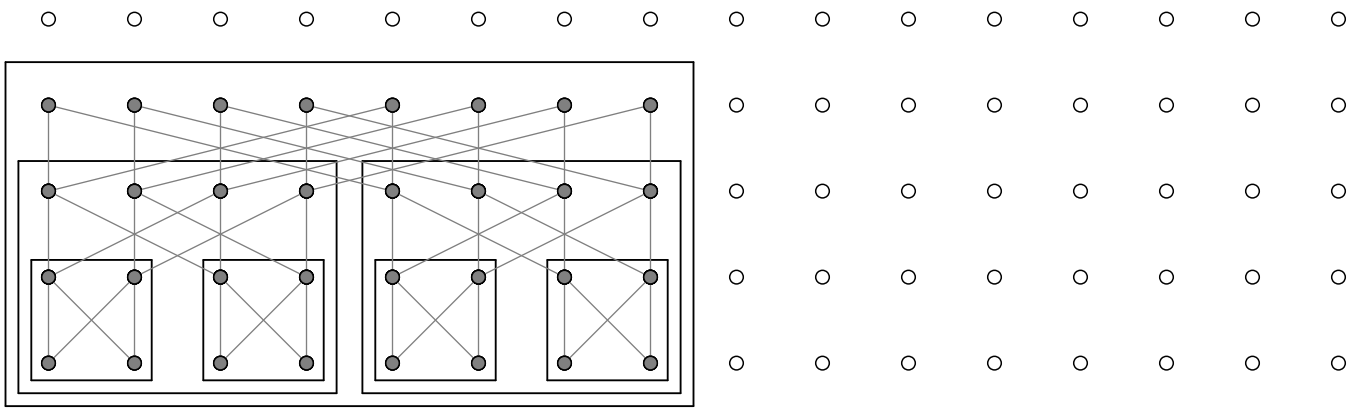


Figure 5: The computations in the boxes are self contained.

where $(x_{p,0}, \ldots, x_{p,\ell-1})$ is the result of the TFT on $(x_{0,0}, \ldots, x_{0,\ell-1}, 0, \ldots, 0)$ — ultimately, the output of the computation.

In keeping with our "Illustrative" description we use pictures, like Figure 6, to indicate what values are known (solid dots) and what value to calculate (empty dot). For instance, "push down $\boldsymbol{x}_k$ with Figure 6", is shorthand for: use $\mathbf{x}_k = x_{s-1,\, im_s+j}$ and $\mathbf{x}_{k+m_s+j} = x_{s-1,\,(i+1)m_s+j}$ to determine $x_{s,\, im_s+j}$. (We emphasize with an arrow that this new value should also overwrite the one at $\mathbf{x}_k$). This calculation is easily accomplished using (1) with this caveat: the values i and j are not explicitly known. What *is* known is s (and therefore m_s) and some array position k. Observe that i is recovered by $i = k$ quo m_s (the quotient of k/m_s).

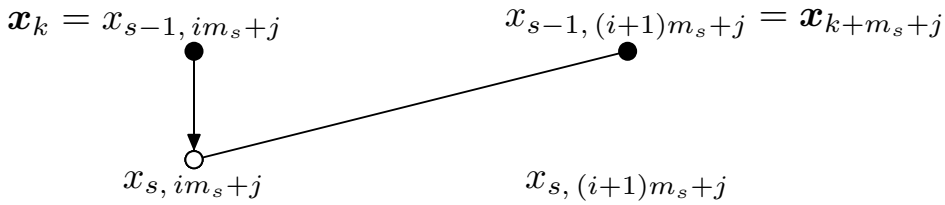


Figure 6: The relation given in (1).

The full description of the inverse TFT follows in Algorithm 1 (note that the initial call is **InvTFT**$(0,\, \ell-1,\, n-1,\, 1)$). A visual depiction of this Algorithm is given in Figure 9. A proof of its correctness follows.

Theorem 2. *Algorithm 1, initially called with* **InvTFT** $(0,\, \ell-1,\, n-1,\, 1)$ *and given access to the 0-indexed, length n array*

$$\mathbf{x} = (x_{p,0}, \ldots, x_{p,\ell-1}, 0, \ldots, 0) \tag{3}$$

will terminate with

$$\mathbf{x} = (x_{0,0}, \ldots, x_{0,\ell-1}, 0, \ldots, 0) \tag{4}$$

where (3) is the result of the TFT on (4).

Algorithm 1: InvTFT(head, tail, last, s)

Initial call: InvTFT(0, $\ell - 1$, $n - 1$, 1);

middle $\leftarrow \dfrac{\text{last} - \text{head}}{2} + \text{head}$;

LeftMiddle $\leftarrow \lfloor \text{middle} \rfloor$;

RightMiddle $\leftarrow$ LeftMiddle $+ 1$;

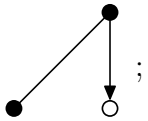
if head $>$ tail **then**

  Base case—do nothing;

  return null;

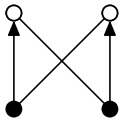
else if tail $\geq$ LeftMiddle **then**

  Push up the self-contained region $\mathbf{x}_{\text{head}}$ to $\mathbf{x}_{\text{LeftMiddle}}$;

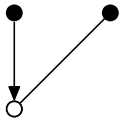
  Push down $\mathbf{x}_{\text{tail}+1}$ to $\mathbf{x}_{\text{last}}$ with  ;

  InvTFT(RightMiddle, tail, last, $s + 1$);

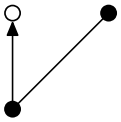
  $s \leftarrow p - \log_2(\text{LeftMiddle} - \text{head} + 1)$;

  Push up (in pairs) $(\mathbf{x}_{\text{head}}, \mathbf{x}_{\text{head}+m_s})$ to $(\mathbf{x}_{\text{LeftMiddle}}, \mathbf{x}_{\text{LeftMiddle}+m_s})$ with  ;

else if tail $<$ LeftMiddle **then**

  Push down $\mathbf{x}_{\text{tail}+1}$ to $\mathbf{x}_{\text{LeftMiddle}}$ with  ;

  InvTFT(head, tail, LeftMiddle, $s + 1$);

  Push up $\mathbf{x}_{\text{head}}$ to $\mathbf{x}_{\text{LeftMiddle}}$ with  ;

Termination. Consider the integer sequences given by

$$\alpha_i = \text{tail}_i - \text{head}_i \in \mathbb{Z} \qquad \text{and} \qquad \beta_i = \text{tail}_i - \left\lfloor \frac{\text{last}_i - \text{head}_i}{2} + \text{head} \right\rfloor \in \mathbb{Z}$$

for head_i/tail_i the values of head/tail at the ith recursive call (give last_i the same definition).

If $\text{head}_i > \text{tail}_i$ then we have termination. Otherwise, either branch (7) executes giving

$$\begin{aligned} \text{head}_{i+1} &= \left\lfloor \frac{\text{last}_i - \text{head}_i}{2} \right\rfloor + \text{head}_i > \text{head}_i \\ \text{tail}_{i+1} &= \text{tail}_i \\ \text{last}_{i+1} &= \text{last}_i \end{aligned}$$

and thus $\alpha_{i+1} < \alpha_i$; or branch (13) executes giving

$$\begin{aligned} \text{head}_{i+1} &= \text{head}_i \\ \text{tail}_{i+1} &= \text{tail}_i \\ \text{last}_{i+1} &= \left\lfloor \frac{\text{last}_i - \text{head}_i}{2} \right\rfloor + \text{head}_i > \text{head}_i. \end{aligned}$$

and thus $\beta_{i+1} < \beta_i$.

Neither branch can run forever since $\beta < 0$ means condition (13) fails forcing termination or α to decrease; and $\alpha < 0$ causes termination. □

Correctness (sketch). We argue inductively (through illustration) that, for $n \in \{0, \ldots, p-1\}$

$$\mathbf{x} = (x_{p-n-1,\,0}, \ldots, x_{p-n-1,\,2^{n+1}-1})$$

can always be calculated from

$$\mathbf{x} = (x_{p,\,0}, \ldots, x_{p,\,\ell-1},\ x_{p-n-1,\,\text{tail}+1}, \ldots,\ x_{p-n-1,\,2^{n+1}-1})\,.$$

Taking this as the inductive hypothesis refer to Figure 7 and Figure 8. □

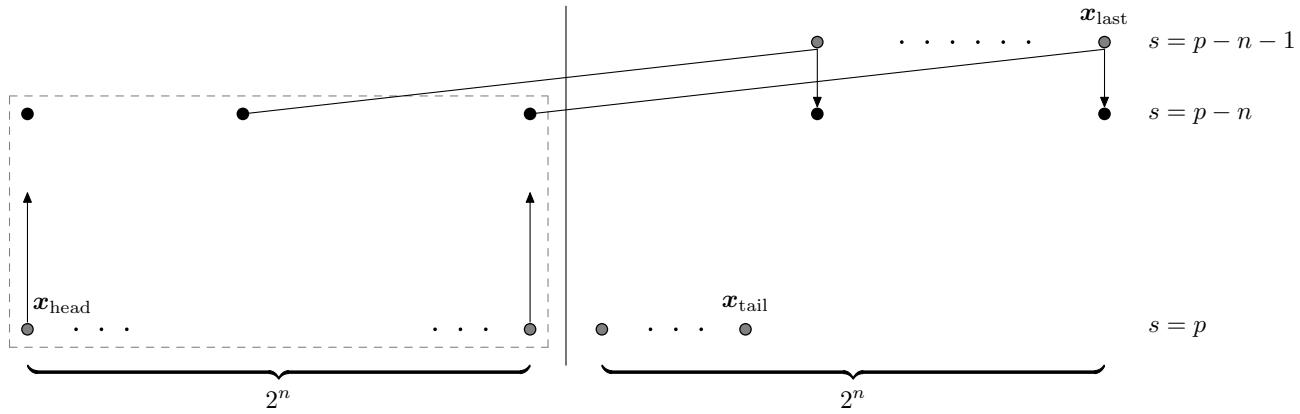
6 Conclusions

The Truncated Fourier Transform is a novel and elegant way to reduce the number of computations of a DFT-based computation by a possible factor of two (which may be significant). Additionally, with the advent of Harvey and Roche's paper [2], it is possible to save as much space as computation. The hidden "cost" of working with the TFT is the increased difficulty of determining the inverse TFT. Although in most cases this is still less costly than the inverse DFT, the algorithm is no doubt more difficult to implement.

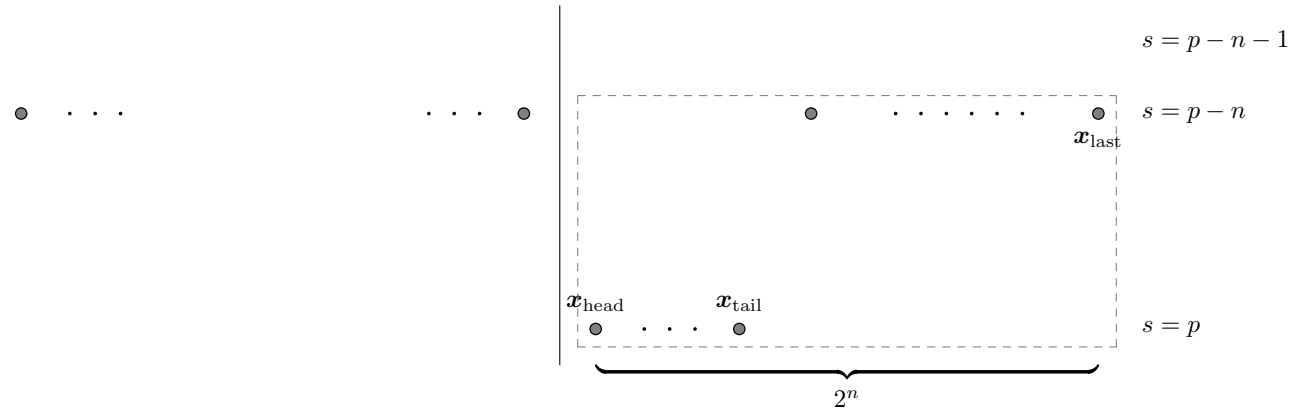
Acknowledgements

The author wishes to thank Dr. Dan Roche and Dr. Éric Schost for reading the draft of this paper and offering suggestions.

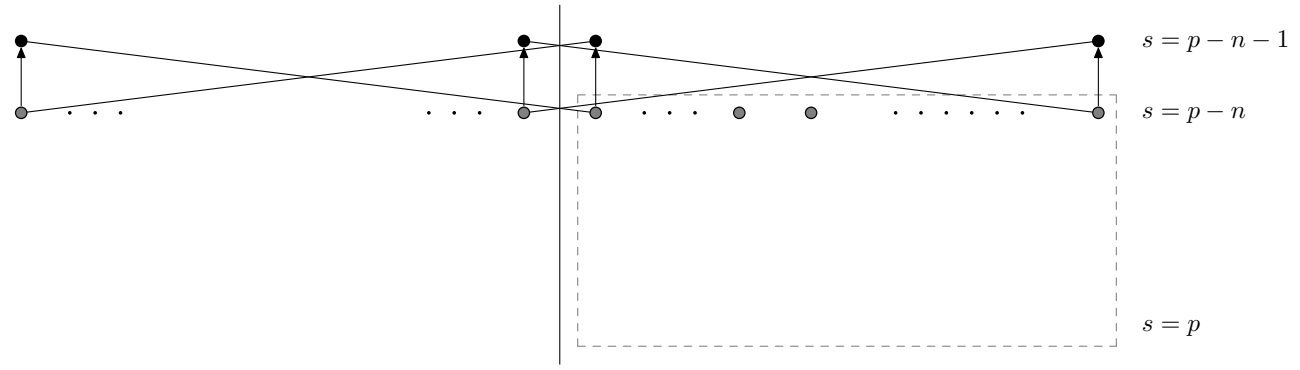
Figure 7: tail $\geq$ LeftMiddle (i.e. at least half the values are at $x = p$).



(a) Line (8): push up the self contained (dashed) region. This yields values sufficient to push down at line (9).

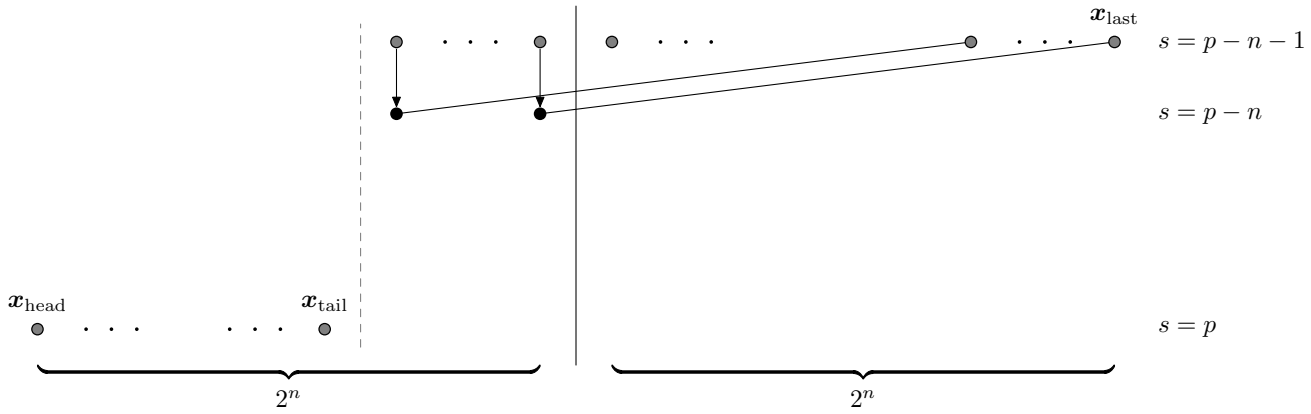


(b) This enables us to make a recursive call on the dashed region (line (12)). By our induction hypothesis this brings all points at $s = p$ to $s = p - n$.

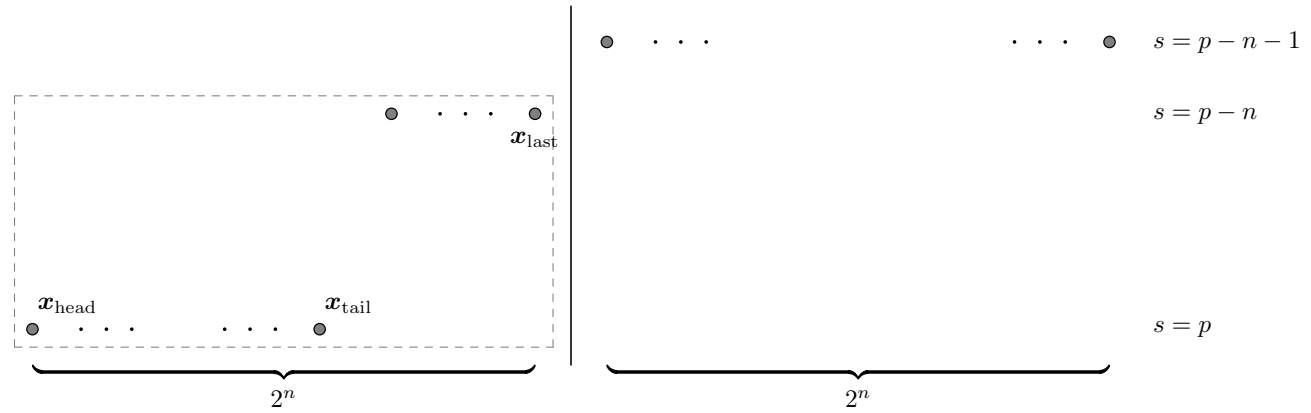


(c) Sufficient points at $s = p - n$ are known to move to $s = p - n - 1$ at line (13).

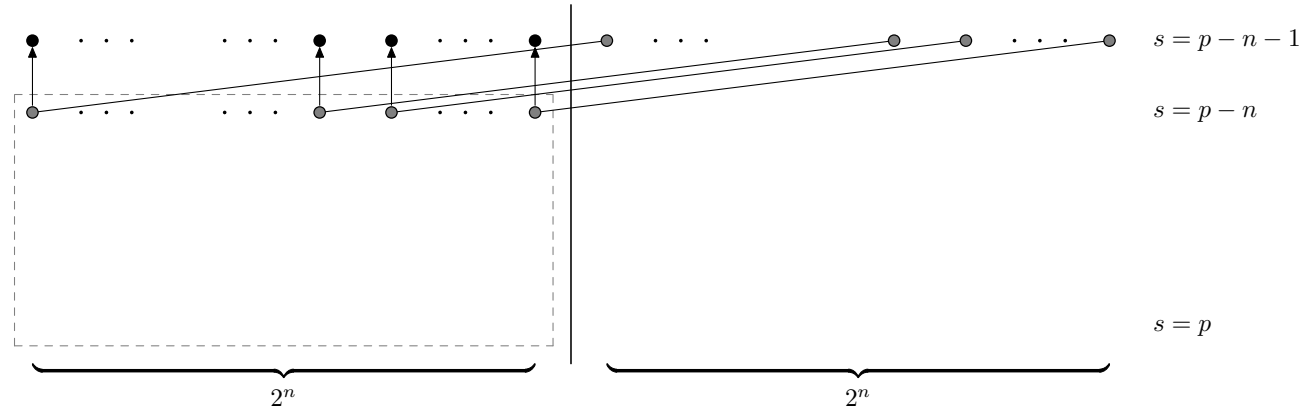
Figure 8: tail < LeftMiddle (i.e. less than half the values are at $x = p$).



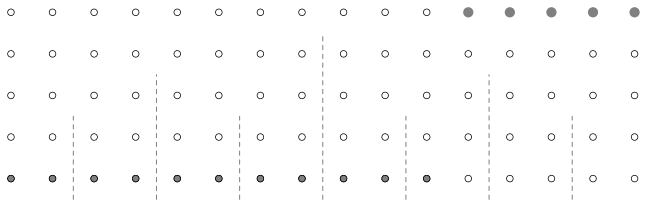
(a) Initially there is sufficient information to push down at line (14).



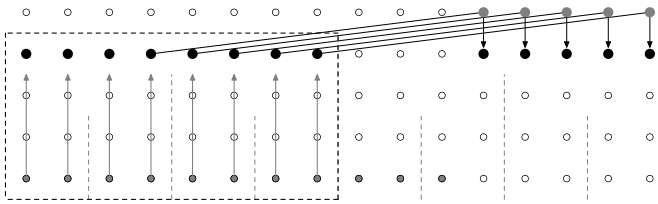
(b) This enables us to make the prescribed recursive call at line (15).



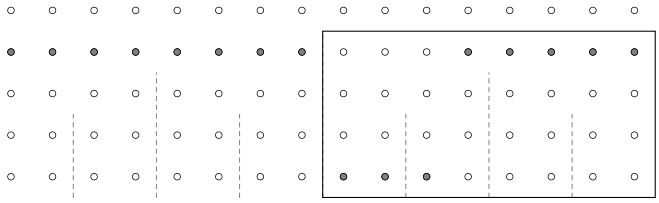
(c) By the induction hypothesis this brings the values in the dashed region to $s = p-n$, leaving enough information to move up at line (16).



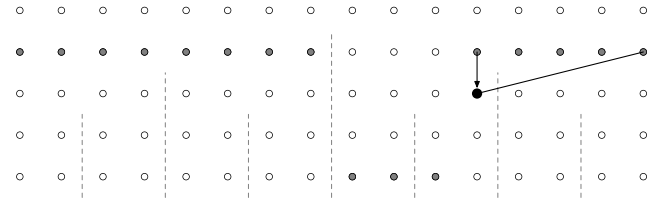
(d) Initial state of the algorithm. Grey dots are the result of the forward TFT; larger grey dots are zeros.



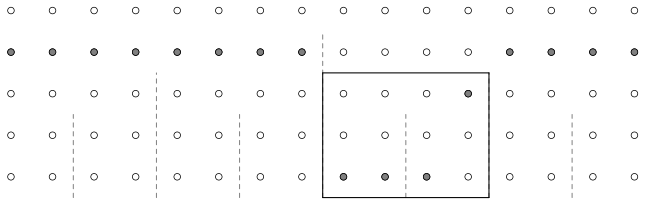
(e) tail≥LeftMiddle. Push up; calculate $x_{1,0}, \ldots, x_{1,7}$ from $x_{4,0}, \ldots, x_{4,7}$ (contained region). **Then** push down.



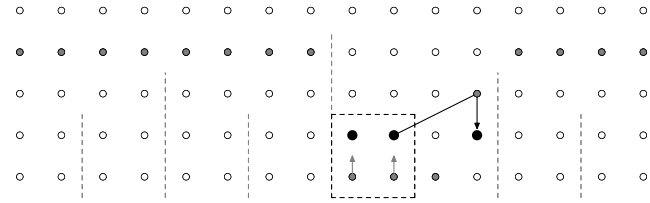
(f) Recursive call on right half.



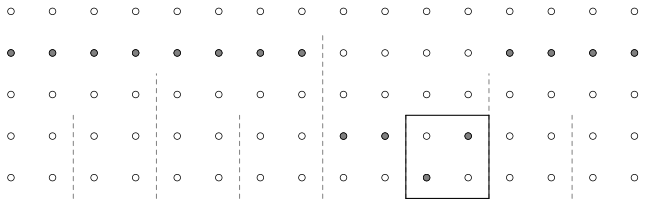
(g) tail<LeftMiddle. Push down with.



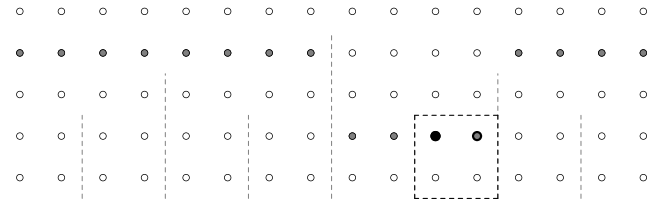
(h) Recursive call on left half.



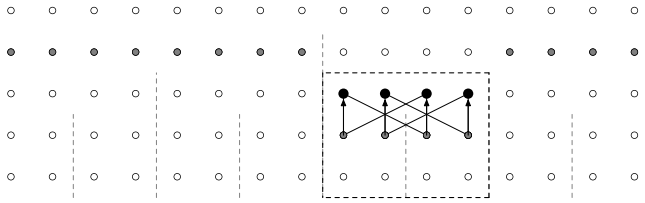
(i) tail≥LeftMiddle. Push up the contained (dashed) region then push down.



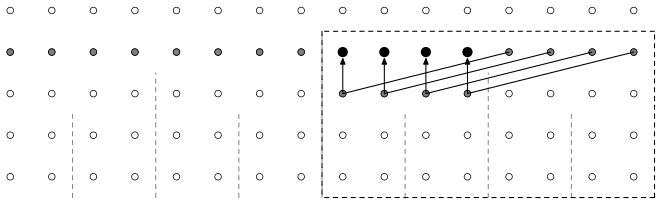
(j) Recursive call on right half.



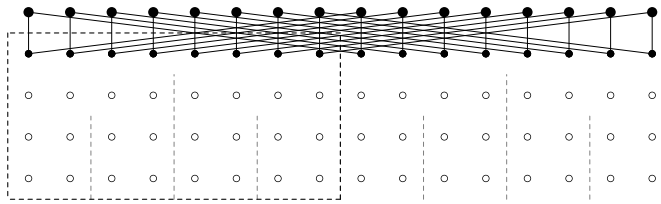
(k) Hiding details. The result of (g).



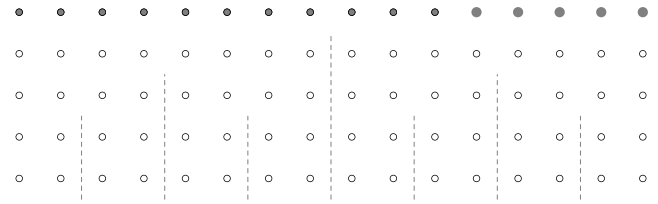
(l) Finish step (e) by pushing up.



(m) Finish step (c) by pushing up.



(n) Resolve the original call by pushing up.



(o) Done.

Figure 9: Schematic representation of the recursive computation of the inverse TFT for $n = 16$ and $\ell = 11$.

References

[1] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[2] David Harvey and Daniel S. Roche. An in-place truncated fourier transform and applications to polynomial multiplication. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 325–329, New York, NY, USA, 2010. ACM.

[3] H.V. Sorensen and C.S. Burrus. Efficient computation of the dft with only a subset of input or output points. *Signal Processing, IEEE Transactions on*, 41(3):1184 –1200, mar 1993.

[4] J. van der Hoeven. Notes on the Truncated Fourier Transform. Technical report, Université Paris-Sud, Orsay, France, 2008.

[5] Joris van der Hoeven. The truncated fourier transform and applications. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 290–296, New York, NY, USA, 2004. ACM.