An Illustrated Introduction to the Truncated Fourier Transform

Paul Vrbik.
School of Mathematical and Physical Sciences
The University of Newcastle
Callaghan, Australia
paulvrbik@gmail.com

February 18, 2016

Abstract

The Truncated Fourier Transform (TFT) is a variation of the Discrete Fourier Transform (DFT/FFT) that allows for input vectors that do *not* have length 2^n for n a positive integer. We present the univariate version of the TFT, originally due to Joris van der Hoeven, heavily illustrating the presentation in order to make these methods accessible to a broader audience.

1 Introduction

In 1965 Cooley and Tukey developed the *Discrete Fourier Transform* (DFT) to recover continuous functions from discrete samples. This was a landmark discovery because it allowed for the digital manipulation of analogue signals (like sound) by computers. Soon after a variant called the *Fast Fourier Transform* (FFT) eclipsed the DFT to the extent that FFT is often mistakenly substituted for DFT. As its name implies, the FFT is a method for computing the DFT faster.

FFTs have an interesting application in Computer Algebra. Let $\mathcal{R}$ be a ring 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 FFT computes the product of two polynomials $P, Q \in \mathcal{R}[x]$ with $\deg(PQ) < n$ in $O(n \log n)$ operations in $\mathcal{R}$. Unfortunately, when $\deg(PQ)$ is sufficiently far from a power of two *many* computations wasted. This deficiency was addressed by the signal processing community using a method called FFT-pruning [3]. However, the difficult inversion of this method is due to van der Hoeven [4][5].

In Section 2 we outline the DFT, including a method for its non-recursive implementation. In Section 3 we develop the "pruned" variant, called Truncated Fourier Transform (TFT).

Finally, in Section 4, we show how the TFT can be inverted and outline the algorithm for doing so in Section 5.

2 The Discrete Fourier Transform

For this paper let $\mathcal{R}$ be a ring with $2 \in \mathcal{R}$ a unit and $\omega \in \mathcal{R}$ an nth root of unity. The Discrete Fourier Transform[*], with respect to ω, of vector $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \mathcal{R}^n$ is the vector $\hat{\mathbf{a}} = (\hat{a}_0, \ldots, \hat{a}_{n-1}) \in \mathcal{R}^n$ with

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

Alternatively we can view these n-tuples as encoding the coefficients of polynomials from $\mathcal{R}[x]$ and define the DFT with respect to ω as the mapping

$$\begin{gathered} \mathrm{DFT}_\omega : \mathcal{R}[x] \to \mathcal{R}^n \\ A(x) = a_0 + \cdots + a_{n-1}x^{n-1} \mapsto (A(\omega^0), \ldots, A(\omega^{n-1})). \end{gathered}$$

We let the relationship between A and its coefficients be implicit and write

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

when $A = a_0 + \cdots + a_{n-1}x^{n-1}$.

The DFT can be computed efficiently using binary splitting. This method requires evaluation 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 A with respect to ω we write

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

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

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

Finally, we construct $\hat{\mathbf{a}}$ according to

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

This description has a natural implementation as a recursive algorithm, but in practice it is often more efficient to implement an in-place algorithm that eliminates the overhead of creating recursive stacks.

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

Definition. Let i and p be a positive integers and let $i = i_0 2^0 + \cdots + i_p 2^p$ for $i_0, \ldots, i_p \in \{0, 1\}$. The length-$p$ bitwise reverse of i is given by

$$[i]_p := i_p 2^0 + \cdots + i_0 2^p.$$

Example. $[3]_5 = 24$ and $[11]_5 = 26$ because

$$\begin{aligned}[3]_5 &= [1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4]_5 \\ &= 0 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 \\ &= 24\end{aligned}$$

and

$$\begin{aligned}[11]_5 &= [1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4]_5 \\ &= 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 \\ &= 26.\end{aligned}$$

Notice if we were to write 3, 24, 11, and 26 as a binary numbers to five digits we have 00011 reverses to 11000 and 01011 reverses to 11010 — in fact this is the inspiration for the name "bitwise reverse."

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

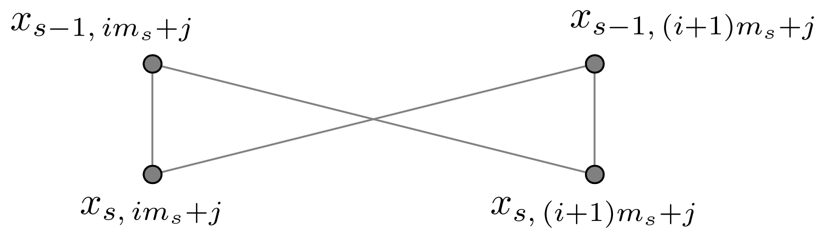
$$\mathbf{x_0} = (x_{0,0}, \ldots, x_{0,n-1}) := (a_0, \ldots, a_{n-1})$$

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

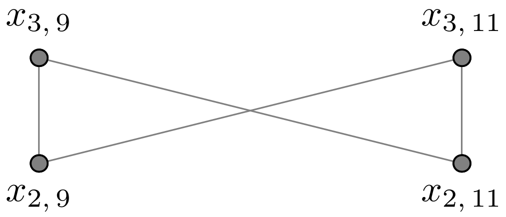
$$\begin{bmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{bmatrix} := \begin{bmatrix} 1 & \omega^{[i]_s m_s} \\ 1 & -\omega^{[i]_s m_s} \end{bmatrix} \begin{bmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{bmatrix} \tag{1}$$

where $m_s = 2^{p-s}$ and for all $i \in \{0, 2, \ldots, n/m_s - 2\}$, $j \in \{0, \ldots, m_s - 1\}$. Note that two additions and *one* multiplication are done in (1) as one product is merely the negation of the other.

We illustrate the dependencies of the $x_{s,i}$ values in (1) with



We call this a "butterfly" after the shape it forms and may say m_s controls the width — the value of which decreases as s increases. By placing these butterflies on a $s \times n$ grid (Figure 1) we can see which values of $\boldsymbol{x}_s$ are required to compute particular entires of $\boldsymbol{x}_{s+1}$ (and vice-versa). For example



denotes that $x_{3,9}$ and $x_{3,11}$ are required to determine $x_{2,9}$ and $x_{2,11}$ (and vice-versa).

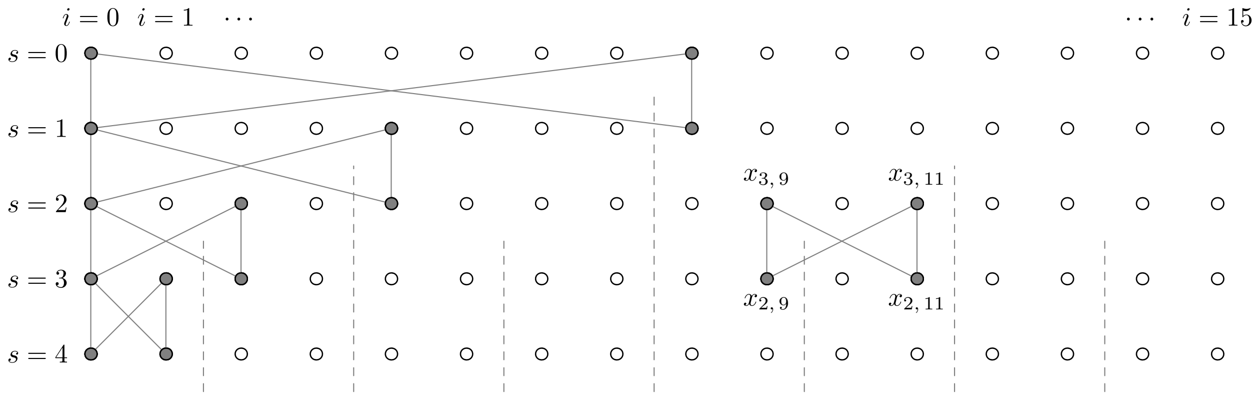


Figure 1: Schematic representation of Equation (1) using butterflies to illustrate the value dependencies at various steps s. The grid has rows $s = 0, \ldots, 4$ and columns $n = 0, \ldots, 15$ for illustrating the $x_{s,i}$ values.

Using induction over s,

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

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

$$x_{p,i} = \hat{a}_{[i]_p} \quad \text{and} \quad \hat{a}_i = x_{p,[i]_p}$$

for all $i \in \{0, \ldots, n-1\}$. That is, $\hat{\mathbf{a}}$ 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, that is to recover $\mathbf{a}$ from $\hat{\mathbf{a}}$:

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

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 $\mathcal{R}$ (see [1, §4.7] for the outline of this algorithm).

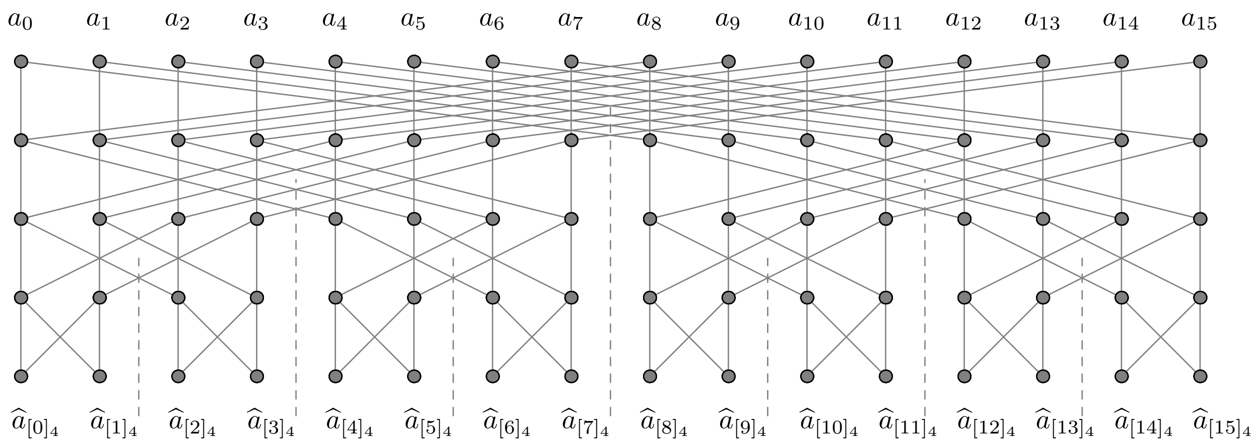


Figure 2: The Discrete Fourier Transform for $n = 16$. The top row, corresponding to $s = 0$, are the initial values $\mathbf{a}$. The bottom row, corresponding to $s = 4$, is a permutation of $\hat{\mathbf{a}}$ (the result of the DFT on $\mathbf{a}$).

3 The Truncated Fourier Transform

The motivation behind the Truncated Fourier Transform (TFT) 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 $\hat{\mathbf{a}}$, the DFT will calculate *all* of them. Thus computation is wasted. We illustrate this in Figures 3 and 4. This type of wasted computation 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. $\hat{\mathbf{a}}$) 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

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

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

Remark. van der Hoeven [4] gives a more general description of the TFT where one can choose 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 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. In this paper, we restrict our discussion to that of the scenario in Figure 4 (where the input and output are the same initial segments) because it can be used for polynomial multiplication, and because it yields the most improvement. $\diamond$

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

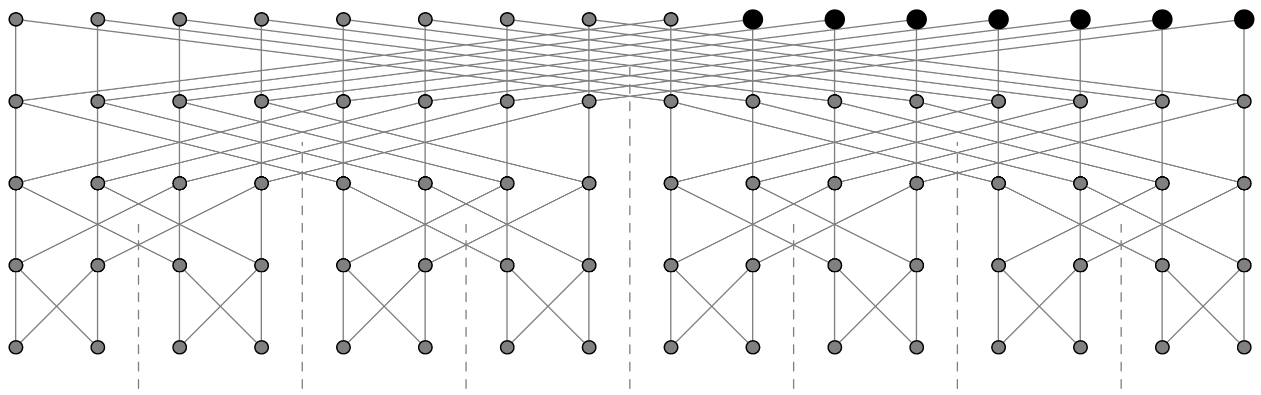


Figure 3: The DFT with “artificial” zero points (large black dots).

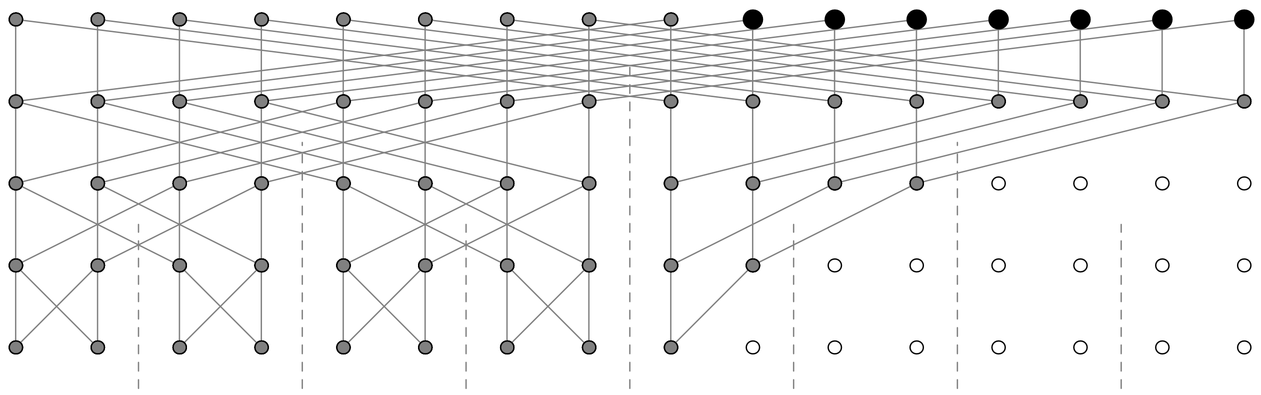


Figure 4: Removing all unnecessary computations from Figure 3 gives the schematic representation of the TFT.

If we only allow ourselves to operate in a size n vector it is straightforward to modify the in-place DFT algorithm from the previous section to execute the TFT. (It should be emphasized that this only saves computation and not space. For a "true" in-place TFT algorithm that operates in an array of size ℓ, see Harvey and Roche's [2].) At stage s it suffices to compute $(x_{s,0}, \ldots, x_{s,j})$ 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 nth root of unity in $\mathcal{R}$. The* TFT *of an ℓ-tuple $(a_0, \ldots, a_{\ell-1})$ with respect to 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

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

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

4 Inverting The Truncated Fourier Transform

Unfortunately, the TFT cannot be inverted by merely doing another TFT with ω^{-1} and adjusting the output by some constant factor like inverse of the DFT. Simply put: we are missing information and must account for this.

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

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

Now, to show the TFT of this with respect to ω^{-1} is *not* $\mathbf{a}$, define

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

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

The TFT of $\mathbf{b}$ with respect to $\omega^{-1} = -5$ 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* a constant multiple of $\mathrm{TFT}_\omega(\mathbf{a})$.

This discrepancy is caused by the completion of $\mathbf{b}$ to $(b_0, b_1, b_2, 0)$ — we should have instead completed b to $(b_0, b_1, b_2, A(-5))$.

To invert the TFT we use the fact that whenever two values among

$$x_{s,im_s+j}, x_{s-1,im_s+j} \text{ and } x_{s,(i+1)m_s+j}, x_{s-1,(i+1)m_s+j}$$

are known, that the other values can be deduced. That is, if two values of some butterfly are known then the other two values can be calculated using (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.

As with DFT, observe 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 $s \in \{p-k, \ldots, p\}$ and $j > 0$. 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

Finally, in this section, 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 $\boldsymbol{x} = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$ for which we assume access; here $n = 2^p$ corresponds to ω, a nth primitive root of unity. Initially, the content of the array is

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

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

In keeping with our "Illustrated" description we use pictures, like Figure 6, to indicate what values are known (solid dots •) and what value to calculate (empty dot ∘). For

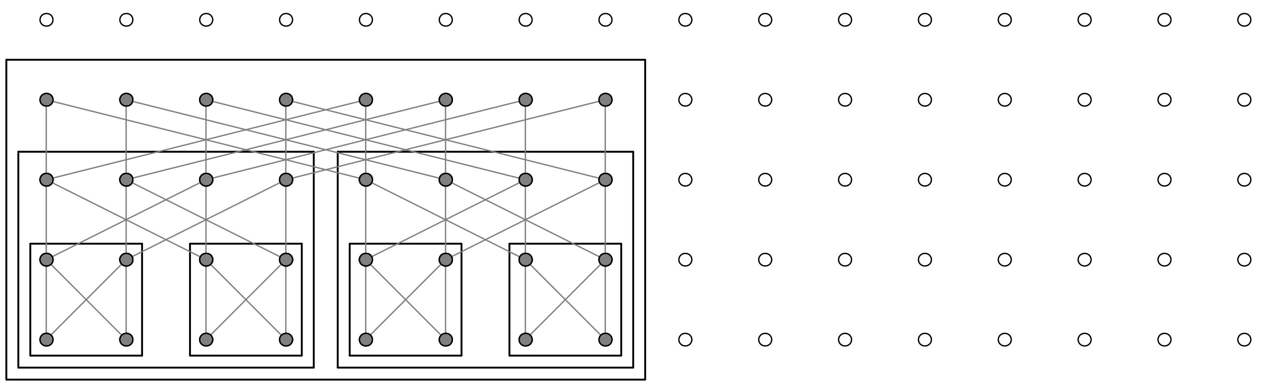


Figure 5: The computations in the boxes are self contained.

instance, "push down $\boldsymbol{x}_k$ with Figure 6", is shorthand for: use $\boldsymbol{x}_k = x_{s-1,\, im_s+j}$ and $\boldsymbol{x}_{k+m_s+j} = x_{s-1,\,(i+1)m_s+j}$ to determine $x_{s,\,im_s+j}$. We emphasize with an arrow that this new value should also overwrite the one at $\boldsymbol{x}_k$. This calculation is easily accomplished using (1) with a 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).

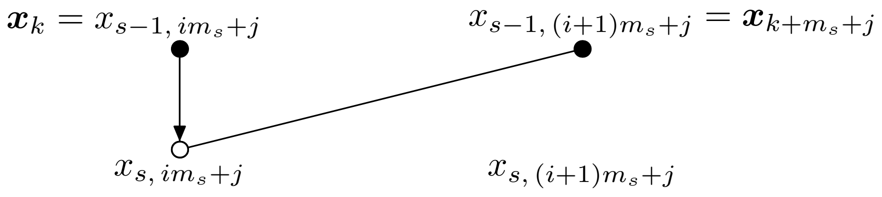

$\boldsymbol{x}_k = x_{s-1,\, im_s+j}$
$x_{s-1,\,(i+1)m_s+j} = \boldsymbol{x}_{k+m_s+j}$
$x_{s,\, im_s+j}$
$x_{s,\,(i+1)m_s+j}$

Figure 6: Overwrite $\boldsymbol{x}[im_s + j]$ with $\boldsymbol{x}[im_s + j] + \omega^{[i]_s m_s}\boldsymbol{x}[(i+1)m_s + j]$.

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 Algorithm 1 is given in Figure 9. A sketch of 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 zero-indexed length n array*

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

will terminate with

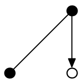
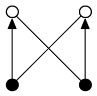
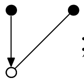
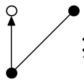
$$\boldsymbol{x} = (x_{0,0}, \ldots, x_{0,\ell-1},\, 0, \ldots, 0) \tag{4}$$

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

Termination. Let head$_i$, tail$_i$, and last$_i$ be the values of head, tail, and last at the ith

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

```
Initial call: InvTFT(0, $\ell - 1$, $n - 1$, 1);
middle $\leftarrow \frac{\text{last} - \text{head}}{2} + \text{head}$;
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 $\boldsymbol{x}_{\text{head}}$ to $\boldsymbol{x}_{\text{LeftMiddle}}$;
    Push down $\boldsymbol{x}_{\text{tail}+1}$ to $\boldsymbol{x}_{\text{last}}$ with  ;
    InvTFT(RightMiddle, tail, last, $s + 1$);
    $s \leftarrow p - \log_2(\text{LeftMiddle} - \text{head} + 1)$;
    Push up (in pairs) ($\boldsymbol{x}_{\text{head}}$, $\boldsymbol{x}_{\text{head}+m_s}$) to ($\boldsymbol{x}_{\text{LeftMiddle}}$, $\boldsymbol{x}_{\text{LeftMiddle}+m_s}$) with  ;
else if tail < LeftMiddle then
    Push down $\boldsymbol{x}_{\text{tail}+1}$ to $\boldsymbol{x}_{\text{LeftMiddle}}$ with  ;
    InvTFT(head, tail, LeftMiddle, $s + 1$);
    Push up $\boldsymbol{x}_{\text{head}}$ to $\boldsymbol{x}_{\text{LeftMiddle}}$ with  ;
```

recursive call. Consider the integer sequences given by

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

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 $\alpha < 0$ causes termination and $\beta < 0$ means either α strictly decreases or condition (13) fails, forcing termination. □

Sketch of correctness. Figure 7 and Figure 16 demonstrate that self contained regions can be exploited to obtain the initial values required to complete the inversion. That is to say, for $n \in \{0, \ldots, p-1\}$, that

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

can always be calculated from

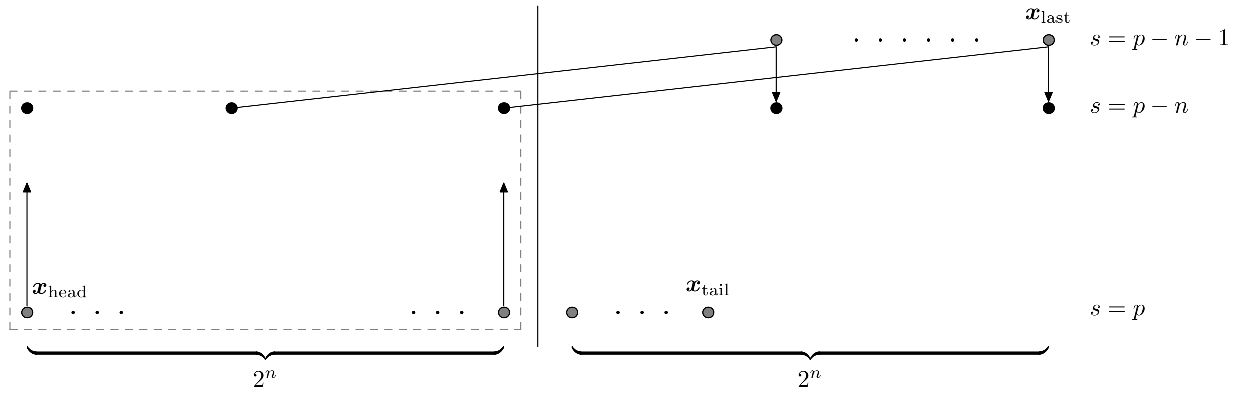
$$\boldsymbol{x} = \left(x_{p,0}, \ldots, x_{p,\ell-1},\ x_{p-n-1,\text{tail}+1}, \ldots,\ x_{p-n-1,2^{n+1}-1}\right).$$

□

6 Conclusions

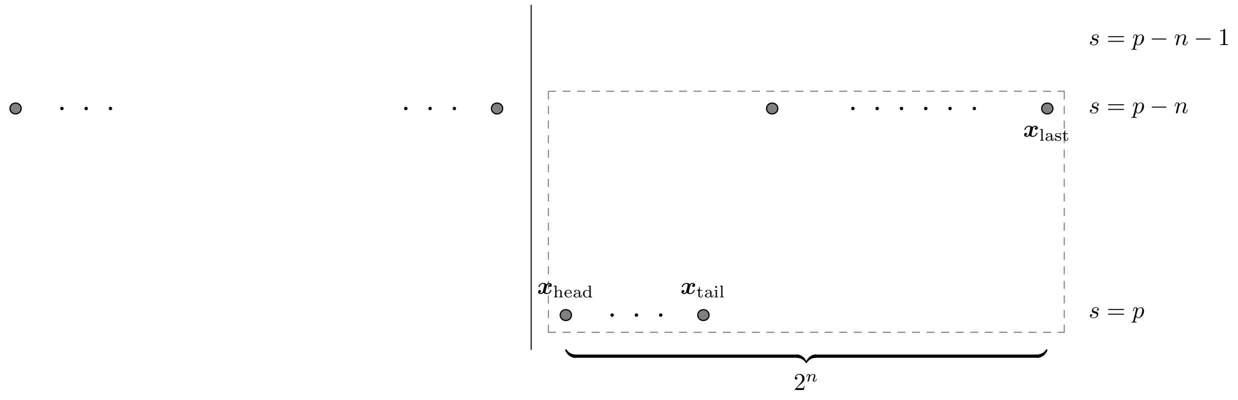
The Truncated Fourier Transform is a novel and elegant way to reduce the number of computations of a DFT-based computation by a possible factor of two (which may be significant). Additionally, with the advent of Harvey and Roche's paper [2], it is possible to save as much space as computation. The hidden "cost" of working with the TFT algorithm 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.

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



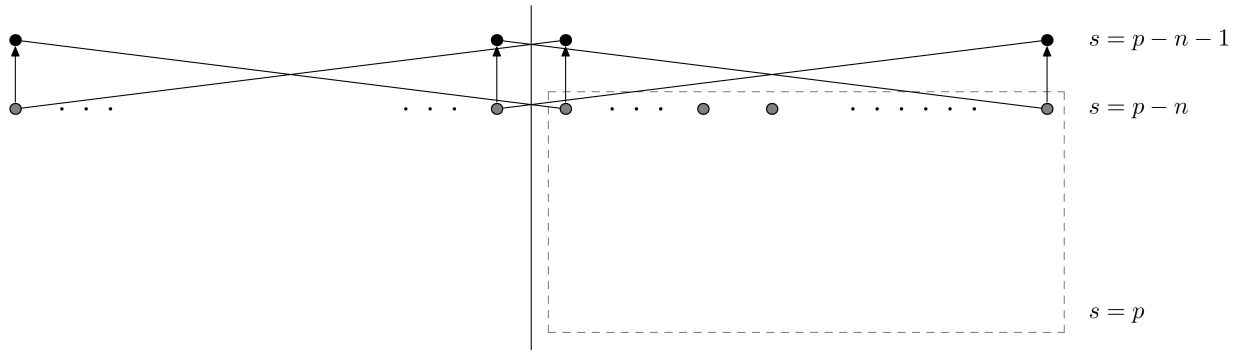
$\boldsymbol{x}_{\text{last}}$
$s = p - n - 1$
$s = p - n$
$\boldsymbol{x}_{\text{head}}$
$\boldsymbol{x}_{\text{tail}}$
$s = p$
2^n
2^n

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

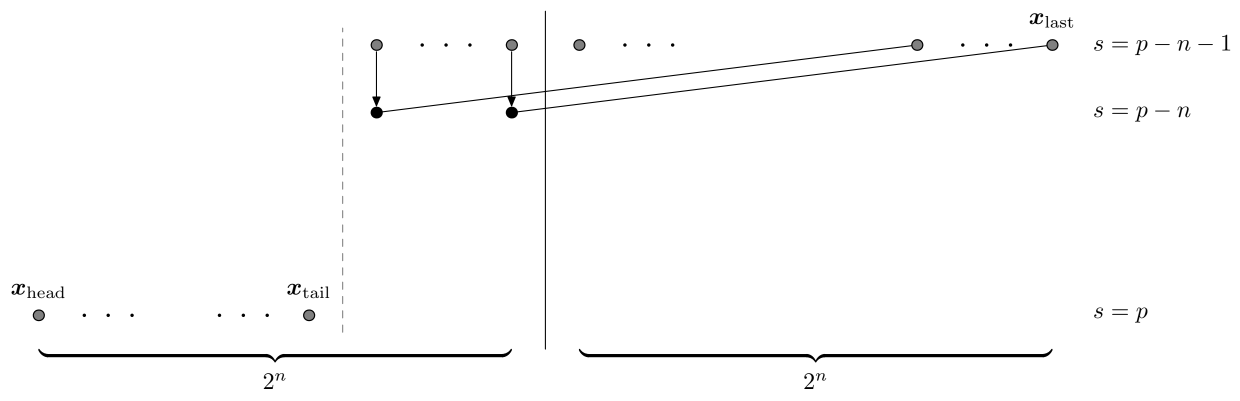


$s = p - n - 1$
$s = p - n$
$\boldsymbol{x}_{\text{last}}$
$\boldsymbol{x}_{\text{head}}$
$\boldsymbol{x}_{\text{tail}}$
$s = p$
2^n

(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 $s = p - n$.

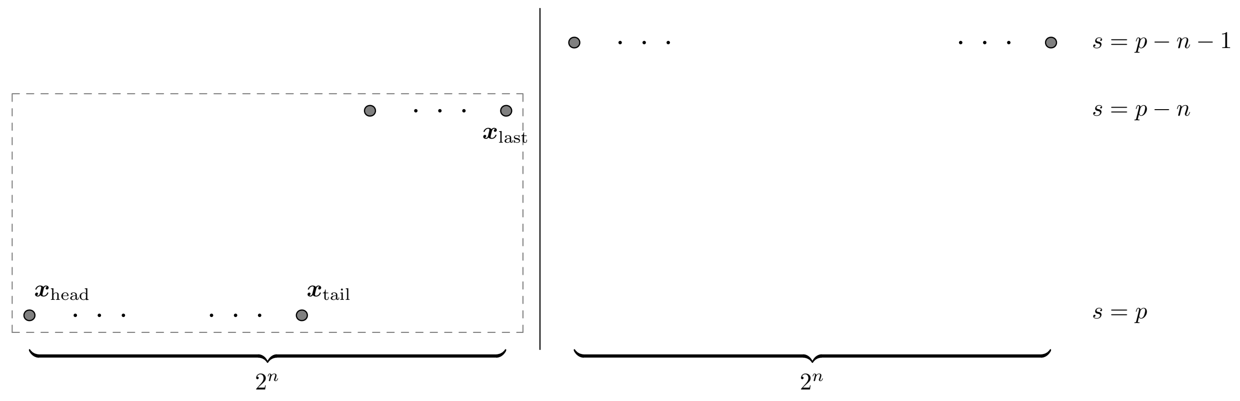


(c) Sufficient points at $s = p - n$ are known to move to $s = p - n - 1$ at line (13).



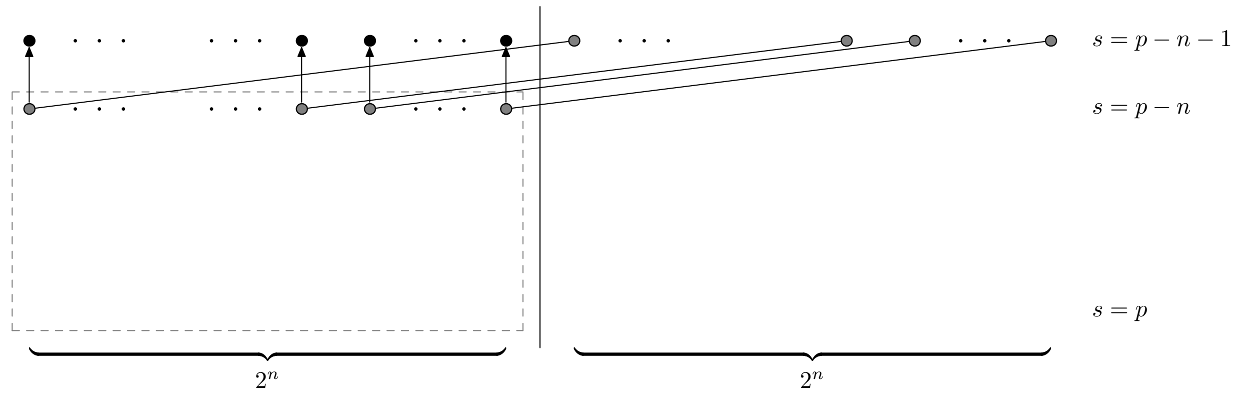
$\boldsymbol{x}_{\text{last}}$
$s = p - n - 1$
$s = p - n$
$\boldsymbol{x}_{\text{head}}$
$\boldsymbol{x}_{\text{tail}}$
$s = p$
2^n
2^n

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



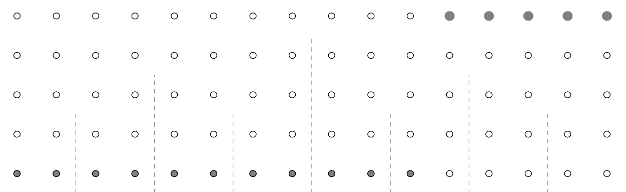
$s = p - n - 1$
$s = p - n$
$\boldsymbol{x}_{\text{last}}$
$\boldsymbol{x}_{\text{head}}$
$\boldsymbol{x}_{\text{tail}}$
$s = p$
2^n
2^n

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

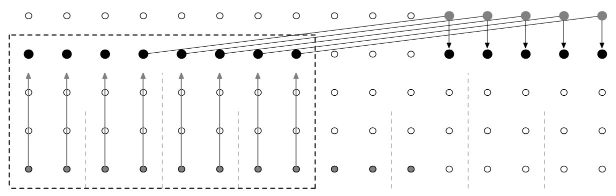


(f) By the induction hypothesis this brings the values in the dashed region to $s = p - n$, leaving enough information to move up at line (16).

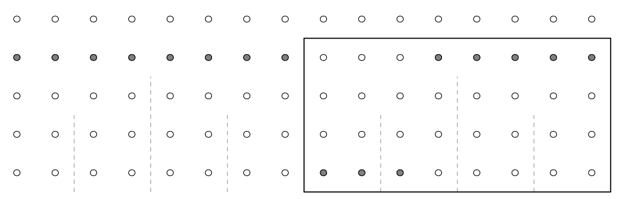
Figure 8: tail < LeftMiddle (i.e. less than half the values are at $x = p$).



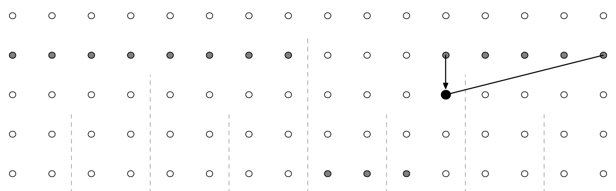
(a) Initial state of the algorithm. Grey dots are the result of the forward TFT; larger grey dots are zeros.



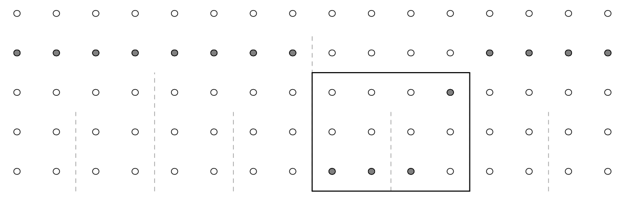
(b) tail≥LeftMiddle. Push up; calculate $x_{1,0},\ldots,x_{1,7}$ from $x_{4,0},\ldots,x_{4,7}$ (contained region). **Then** push down.



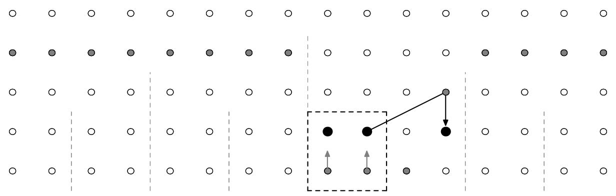
(c) Recursive call on right half.



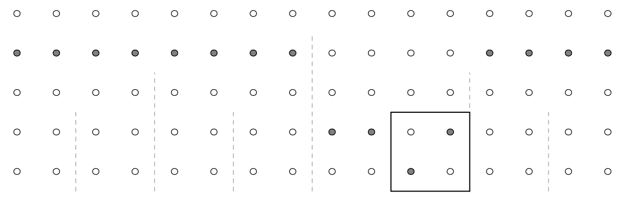
(d) tail<LeftMiddle. Push down with.



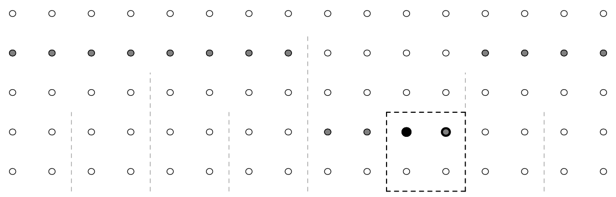
(e) Recursive call on left half.



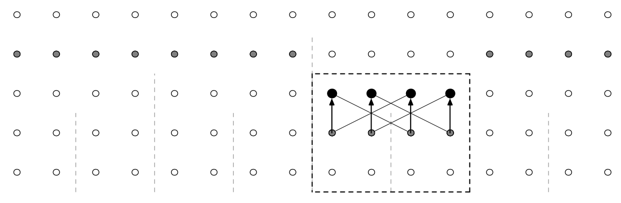
(f) tail≥LeftMiddle. Push up the contained (dashed) region then push down.



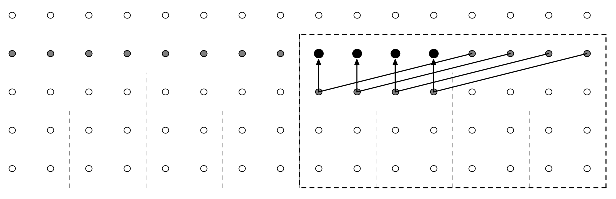
(g) Recursive call on right half.



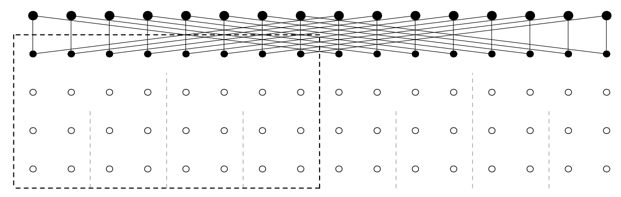
(h) Hiding details. The result of (g).



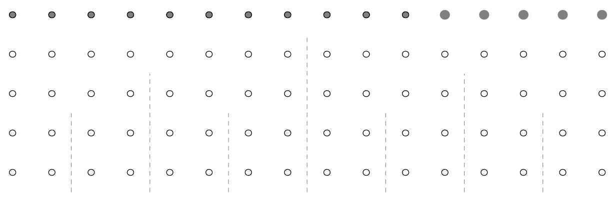
(i) Finish step (e) by pushing up.



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



(k) Resolve the original call by pushing up.



(l) Done.

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

Acknowledgements

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

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.