Truncated Fourier Transform

Mike Ghesquiere

December 14, 2012

Introduction

$\mathcal{R}$ is a ring for which:

- $2, 1/2 \in \mathcal{R}$
- $\mathcal{R}$ is *effective* (there are effective algorithms to compute $+$, $-$ and $\times$).

If $\mathcal{R}$ has a primitive n-th root of unity with $n = 2^p$ then the product of $P, Q \in \mathcal{R}[X]$ with $\deg(PQ) < n$ can be computed in $\mathcal{O}(n \log n)$ using the (Discrete) *Fast Fourier Transform FFT*.

If $\mathcal{R}$ does not contain a primitive n-th root of unity then we require an additional $\mathcal{O}(\log\log n)$ to construct a ring extension to carry out the multiplication.

Fast Fourier Transform (FFT)

Let ω be a primitive n-th root of unity with $n = 2^p$. Define:

$$\begin{aligned} FFT_\omega : \quad & \mathcal{R}^n \to \mathcal{R}^n \\ & (a_0, ..., a_{n-1}) \mapsto (\hat{a}_0, ..., \hat{a}_{n-1}) \end{aligned} \tag{1}$$

where

$$\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \tag{2}$$

Fast Fourier Transform (FFT)

Consider the binary splitting:

$$(a_0, a_1, ..., a_{n-1}) = (b_0, c_0, ..., b_{n/2-1}, c_{n/2-1}).$$

$$FFT_{\omega^2}(b_0, b_1, ..., b_{n/2-1}) = (\hat{b}_0, ..., \hat{b}_{n/2-1})$$
$$FFT_{\omega^2}(c_0, c_1, ..., c_{n/2-1}) = (\hat{c}_0, ..., \hat{c}_{n/2-1})$$

Then we have:

$$FFT_\omega(a_0, ..., a_{n-1}) = \begin{pmatrix}(\hat{b}_0 + \hat{c}_0), ..., (\hat{b}_{n/2-1} + \hat{c}_{n/2-1})\omega^{n/2-1} \\ (\hat{b}_0 - \hat{c}_0), ..., (\hat{b}_{n/2-1} - \hat{c}_{n/2-1})\omega^{n/2-1}\end{pmatrix}$$

This gives a natural recursive implementation.

In-place Fast Fourier Transform

In practice, it is more efficient to use an in-place variant.

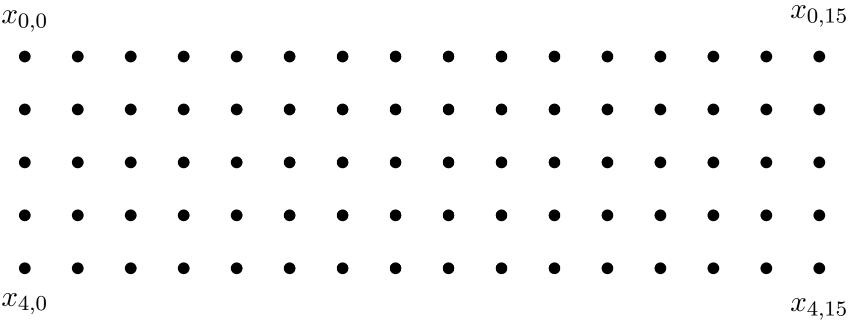
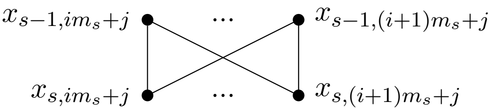


Figure: The case for $n = 16 = 2^4$

In-place Fast Fourier Transform



$$\begin{pmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{pmatrix} = \begin{pmatrix} 1 & \omega^{rev_s(i)m_s} \\ 1 & -\omega^{rev_s(i)m_s} \end{pmatrix} \begin{pmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{pmatrix} \tag{3}$$

$m_s = 2^{p-s}$.
$rev_s(i)$ is the bitwise reverse of i at length s. ($rev_5(11) = 26$)
$i \in \{0, 2, ..., n/m_s - 2\}$
$j \in \{0, ..., m_s - 1\}$

In-place Fast Fourier Transform

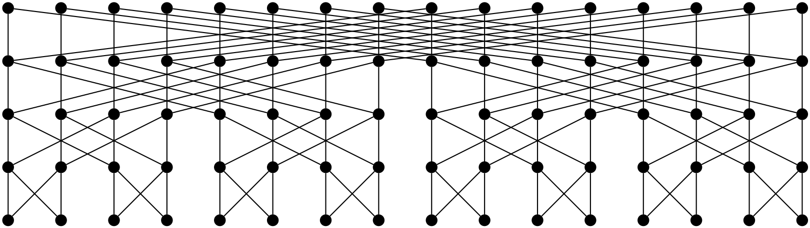


Figure: Schematic representation of a TFT for $n = 16$.

$$x_{s,im_s+j} = (FFT_{\omega^{m_s}}(a_j, a_{m_s+j}, ...a_{n-m_s+j}))_{rev_s(i)}$$

In particular:

$$x_{p,i} = \hat{a}_{rev_p(i)}$$

Truncated Fourier Transform

However, FFT is only defined for $n = 2^p$. If $a \in \mathcal{R}^l$ for $n/2 < l < n$ then we must carry out FFT at precision n.

We now present the *Truncated Fourier Transform* (TFT) for vectors in $\mathcal{R}^l$ which reduces to FFT in the case where l is a power of two. However it also behaves more smoothly for intermediate degrees.

Truncated Fourier Transform (TFT)

At stage s it is sufficient to compute $(x_{s,0}, ..., x_{s,(\lceil l/m_s \rceil m_s)-1})$.

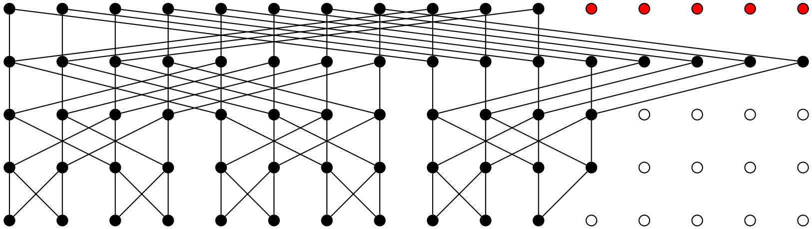


Figure: Schematic representation of a TFT for $n = 16$ and $l = 11$.

Inverting TFT

Our algorithm for inverting the TFT will rely on reconstructing the original graph.

$$\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 1 & \alpha \\ 1 & -\alpha \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

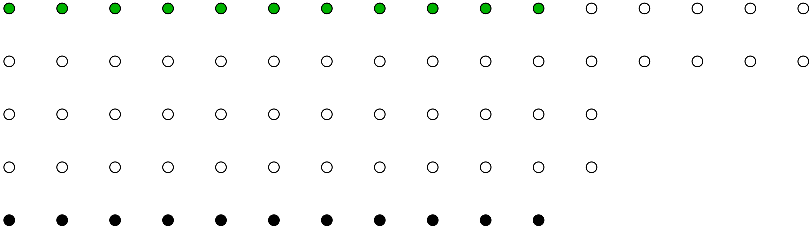
$$\begin{pmatrix} a \\ b \end{pmatrix} = 2^{-1} \begin{pmatrix} 1 & 1 \\ \alpha^{-1} & -\alpha^{-1} \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} \tag{4}$$

$$\begin{pmatrix} d \\ a \end{pmatrix} = \begin{pmatrix} 1 & -2\alpha \\ 1 & -\alpha \end{pmatrix} \begin{pmatrix} c \\ b \end{pmatrix} \tag{5}$$

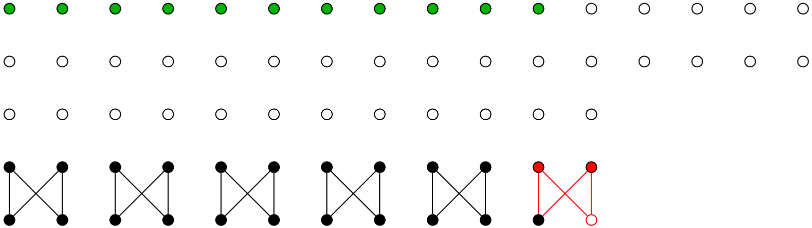
$$\begin{pmatrix} c \\ b \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -\alpha^{-1} & \alpha^{-1} \end{pmatrix} \begin{pmatrix} d \\ a \end{pmatrix} \tag{6}$$

Equations for (a, c) in terms of (b, d) and vice versa also exist.

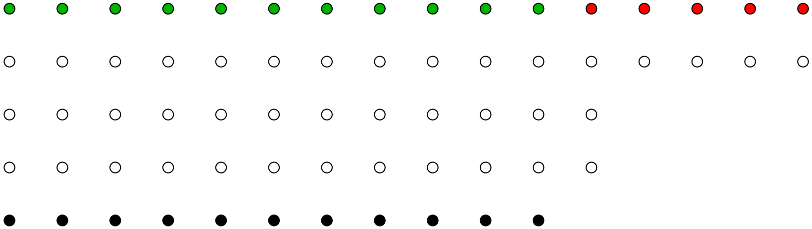
Inverting TFT



Inverting TFT



Inverting TFT



Inverting TFT

More prceisely, our algorithm takes two input:

- $\vec{a} = (x_{p,k}, ..., x_{p,l-1})$
- $\vec{b} = (x_{s,l}, ..., x_{s,(\lceil l/m_s \rceil m_s)-1})$

and will return:

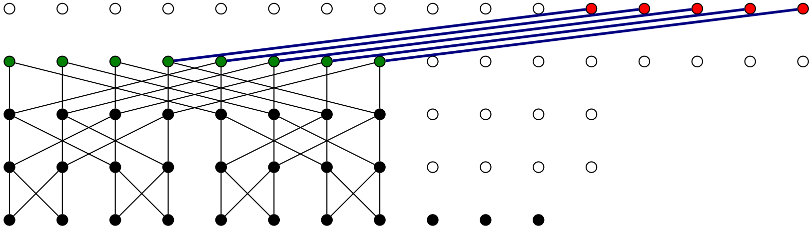
- $(x_{s,k}, ..., x_{s,l-1})$

Inverting TFT - Alternative algorithm

Case 1: length($\vec{a}$) $\geq$ length($\vec{b}$).

1. Using Equation (4) repeatedly, we "push up" the vector $(x_{p,k}, ..., x_{p,k+2^i-1})$ where i is maximal such that $2^i \leq l - k$.
2. Using the newly computed $(x_{p(s+1),k}, ..., x_{(s+1),k+2^i-1})$ and $\vec{b}$ with Equation (5) we can find
   - $(x_{(s+1),l}, ..., x_{(s+1),(\lceil l/m_s \rceil m_s)-1})$ and
   - $(x_{s,l-k}, ..., x_{s,k+2^i-1})$
3. We compute $(x_{(s+1),k+2^i}, ..., x_{(s+1),l})$ by making a recursive call on $(x_{p,k+2^i}, ..., x_{p,l-1})$ and $(x_{(s+1),l}, ..., x_{(s+1),(\lceil l/m_{(s+1)} \rceil m_{(s+1)})-1})$

Inverting TFT - Alternative algorithm



Inverting TFT - Alternative algorithm

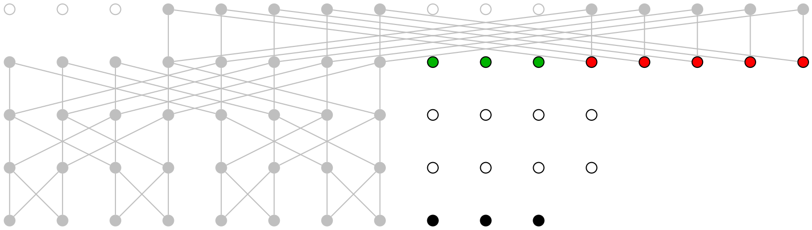


Figure: First recursive call.

Inverting TFT - Alternative algorithm

Case 2: $\text{length}(\vec{a}) < \text{length}(\vec{b})$.

1. Using Equation (3), push down $\vec{b}$ to $(x_{(s+1),l}, \ldots, x_{(s+1),(\lceil l/m_{(s+1)} \rceil m_{(s+1)})-1})$.
2. Recurse to calculate $\vec{c} = (x_{(s+1),k}, \ldots, x_{(s+1),l-1})$
3. Using Equation (5), compute $(x_{s,k}, \ldots, x_{s,l-1})$ using $\vec{c}$ and $\vec{b}$.

Inverting TFT - Alternative algorithm

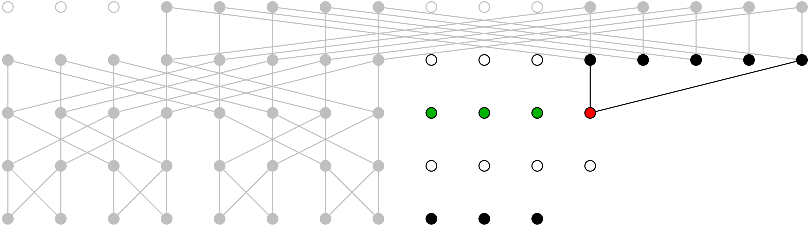


Figure: After pushing down, about to make second recursive call.

Inverting TFT - Alternative algorithm

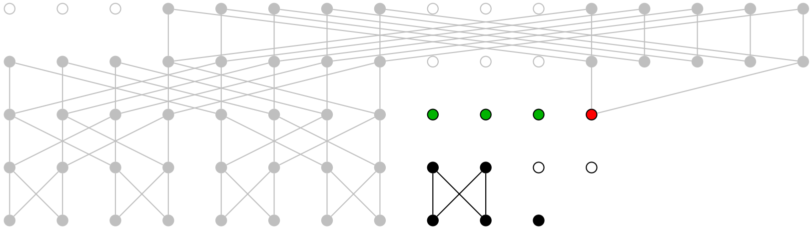


Figure: length($\vec{a}$) $\geq$ length($\vec{b}$). We carry out case 1 again.

Inverting TFT - Alternative algorithm

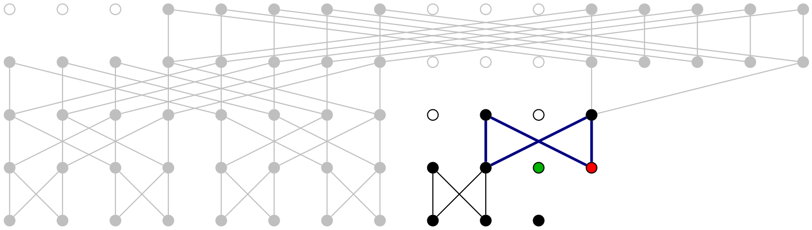


Figure: Recurse again.

Inverting TFT - Alternative algorithm

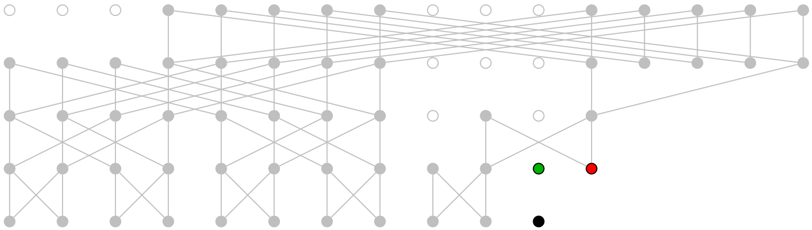


Figure: $p = (s + 1)$ so the push up is trivial.

Inverting TFT - Alternative algorithm

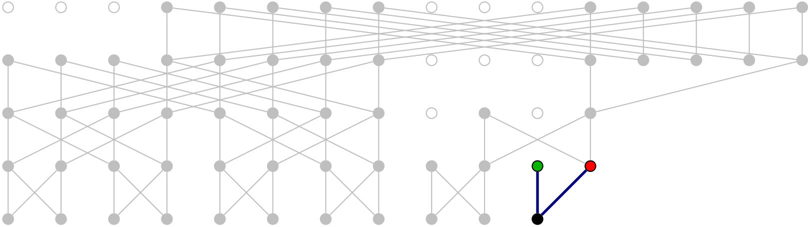
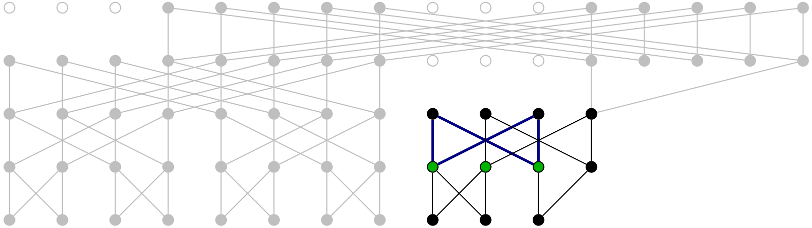
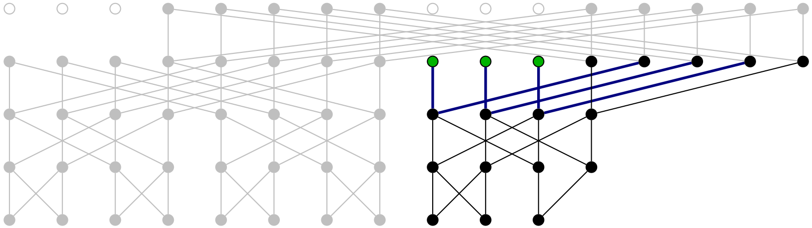


Figure: $\vec{a}$ and $\vec{b}$ have equal lengths so we return.

Inverting TFT - Alternative algorithm



Inverting TFT - Alternative algorithm



Truncated Fourier Transform (TFT)