A List-Decoding Algorithm for Polar Codes

Connections between Viterbi’s decoding algorithm and the Successive Cancellation List decoding algorithm

<table>
<thead>
<tr>
<th></th>
<th>Viterbi</th>
<th>SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoding algorithm?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Code family</td>
<td>convolutional</td>
<td>polar</td>
</tr>
<tr>
<td>Optimal?</td>
<td>✓</td>
<td>almost, sometimes</td>
</tr>
<tr>
<td>Recursion?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data structures?</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Successive Cancellation

<table>
<thead>
<tr>
<th>decide on</th>
<th>based on</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{u}_0$</td>
<td>$W_0(y_0^{n-1}</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\hat{u}_1$</td>
<td>$W_1(y_0^{n-1}, \hat{u}_0</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\hat{u}_2$</td>
<td>$W_2(y_0^{n-1}, \hat{u}_0</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\hat{u}_i$</td>
<td>$W_i(y_0^{n-1}, \hat{u}_0^{i-1}</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\hat{u}_{n-1}$</td>
<td>$W_{n-1}(y_0^{n-1}, \hat{u}_0^{n-2}</td>
</tr>
</tbody>
</table>
SC not good enough

Legend:
- • successive cancellation, $n = 2048$, $k = 1024$
- ▲ LDPC (Wimax standard, $n = 2304$)

- Is the SC decoder under-performing?
- Are the polar codes themselves weak at this length?
A critical look at SC

Successive Cancellation Decoding

for $i = 0, 1, \ldots, n - 1$ do

if $\hat{u}_i$ is frozen then set $\hat{u}_i$ accordingly;
else

if $W_i(y_0^{n-1}, \hat{u}_0^{i-1}|0) > W_i(y_0^{n-1}, \hat{u}_0^{i-1}|1)$ then

set $\hat{u}_i \leftarrow 0$;
else

set $\hat{u}_i \leftarrow 1$;

Potential weaknesses (interplay):

- Once an unfrozen bit is set, there is “no going back”. A bit that was set at step $i$ can not be changed at step $j > i$.
- Knowledge of the value of future frozen bits is not taken into account.
List decoding of polar codes

**Key idea:** Each time a decision on $\hat{u}_i$ is needed, split the current decoding path into two paths: try both $\hat{u}_i = 0$ and $\hat{u}_i = 1$. 

![Diagram of decoding paths](attachment:diagram.png)
List decoding of polar codes

**Key idea:** Each time a decision on $\hat{u}_i$ is needed, split the current decoding path into two paths: try both $\hat{u}_i = 0$ and $\hat{u}_i = 1$. 
List decoding of polar codes

**Key idea:** Each time a decision on $\hat{u}_i$ is needed, split the current decoding path into two paths: try both $\hat{u}_i = 0$ and $\hat{u}_i = 1$. 

![Diagram of list decoding of polar codes]
List decoding of polar codes

**Key idea:** Each time a decision on $\hat{u}_i$ is needed, split the current decoding path into two paths: **try both $\hat{u}_i = 0$ and $\hat{u}_i = 1$.**

When the number of paths grows beyond a prescribed threshold $L$, discard the worst (least probable) paths, and keep only the $L$ best paths.
**List decoding of polar codes**

**Key idea:** Each time a decision on $\hat{u}_i$ is needed, split the current decoding path into two paths: **try both $\hat{u}_i = 0$ and $\hat{u}_i = 1$.**

When the number of paths grows beyond a prescribed threshold $L$, discard the worst (least probable) paths, and keep only the $L$ best paths.
List decoding of polar codes

**Key idea:** Each time a decision on $\hat{u}_i$ is needed, split the current decoding path into two paths: **try both** $\hat{u}_i = 0$ and $\hat{u}_i = 1$.

When the number of paths grows beyond a prescribed threshold $L$, discard the worst (least probable) paths, and keep only the $L$ best paths.

At the end, select the single **most likely** path.
List-decoding: complexity issues

The idea of branching while decoding is not new:


Our contribution

- In a naive implementation, the time would be $O(L \cdot n^2)$.
- We show that this can be done in $O(L \cdot n \log n)$ time and $O(L \cdot n)$ space.

We will return to the complexity issue later. For now, let’s see how decoding performance is affected.
Approaching ML performance

List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size. Good: our decoder is essentially optimal. Bad: Still not competitive with LDPC. . .

Conclusions: Must somehow “fix” the polar code.
Approaching ML performance

List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.

Legend:

- ○ $n = 2048$, $L = 1$
- △ $n = 2048$, $L = 2$

Word error rate

Signal-to-noise ratio ($E_b/N_0$) [dB]
Approaching ML performance

List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.

Legend:
- $n = 2048$, $L = 1$
- $n = 2048$, $L = 2$
- $n = 2048$, $L = 4$
Approaching ML performance

List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size. Good: our decoder is essentially optimal. Bad: Still not competitive with LDPC. . .

Conclusions: Must somehow "fix" the polar code.
Approaching ML performance

List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.

Good: our decoder is essentially optimal.

Bad: Still not competitive with LDPC.

Conclusions: Must somehow "fix" the polar code.
Approaching ML performance

Legend:
- $n = 2048, L = 1$
- $n = 2048, L = 2$
- $n = 2048, L = 4$
- $n = 2048, L = 8$
- $n = 2048, L = 16$
- $n = 2048, L = 32$

Signal-to-noise ratio $\left( E_b/N_0 \right)$ [dB]

Word error rate
List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.
List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.

- Good: our decoder is essentially optimal.
- Bad: Still not competitive with LDPC...
Approaching ML performance

List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.

- Good: our decoder is essentially optimal.
- Bad: Still not competitive with LDPC...
- Conclusions: Must somehow “fix” the polar code.
A simple concatenation scheme

- Recall that the last step of decoding was “pick the most likely codeword from the list”.
- An error: the transmitted codeword is not the most likely codeword in the list.
- However, very often, the transmitted codeword is still a member of the list.
- We need a “genie” to single-out the transmitted codeword.
- Idea: Let there be $k + r$ unfrozen bits. Of these,
  - Use the first $k$ bits to encode information.
  - Use the last $r$ unfrozen bits to encode the CRC value of the first $k$ bits.
  - Pick the most probable codeword on the list with correct CRC.
Approaching LDPC performance

Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0$ dB.

![Graph showing Bit error rate vs. Signal-to-noise ratio for Successive cancellation and List-decoding (L = 32)]

- **Successive cancellation**
- **List-decoding (L = 32)**
Approaching LDPC performance

Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0$ dB.

---

**Graph:**
- **Red line and circles:** Successive cancellation
- **Purple line and triangles:** List-decoding ($L = 32$)
- **Magenta dashed line and downward triangles:** WiMax turbo ($n = 960$)
- **Green dashed line and downward triangles:** WiMax LDPC ($n = 2304$)
Approaching LDPC performance

Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0$ dB.
Approaching LDPC performance

Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0$ dB.
Quadratic complexity of list decoding

Naive implementation recap

- In a naive implementation, the decoding paths are independent. They don’t share information.
- Each decoding path has a set of variables associated with it. For example, at stage $i$, each decoding path must remember the values of the bits $\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{i-1}$.
- It turns out (as we shall see) that each decoding path has $\Theta(n)$ memory associated with it.
- When a path is split in two, one decoding path is left with the original variables while the other must be handed a copy of them.
- Each copy operation takes $O(n)$ time.
- Thus, the overall time complexity is $O(L \cdot n^2)$. 
A very short introduction to polar codes

\[ U_0 \rightarrow X_0 \rightarrow Y_0 \]
\[ U_1 \rightarrow X_1 \rightarrow Y_1 \]

polar coding
A very short introduction to polar codes

\[ W^- : \mathcal{X} \to \mathcal{Y}^2 \]

\[
\begin{align*}
U_0 & \quad \xrightarrow{} \quad X_0 \quad \xrightarrow{} \quad Y_0 \\
U_1 & \quad \xrightarrow{} \quad X_1 \quad \xrightarrow{} \quad Y_1 \\
\text{Ber}(\frac{1}{2}) & \quad \xrightarrow{} \quad W
\end{align*}
\]

\[ W^-(y_0, y_1 | u_0) = \sum_{u_1 \in \mathcal{X}} \frac{1}{2} W(y_0 | u_0 \oplus u_1) W(y_1 | u_1). \]
A very short introduction to polar codes

\( W^+ : \mathcal{X} \rightarrow \mathcal{X} \times \mathcal{Y}^2 \)

\[ W^+(u_0, y_0, y_1 | u_1) = \frac{1}{2} W(y_0 | u_0 \oplus u_1) W(y_1 | u_1). \]
A closer look at successive cancellation

\[ u_0 \rightarrow x_0 \rightarrow y_0 \]

\[ u_1 \rightarrow x_1 \rightarrow y_1 \]
A closer look at successive cancellation

\[ u_0 \rightarrow x_0 \rightarrow y_0 \]
\[ u_1 \rightarrow x_1 \rightarrow y_1 \]
A closer look at successive cancellation

\[ (P(y_0 y_1 | u_0 = 0), P(y_0 y_1 | u_0 = 1)) \]

\[ (P(y_0 | x_0 = 0), P(y_0 | x_0 = 1)) \]

\[ (P(y_0 y_1 \hat{u}_0 | u_1 = 0), P(y_0 y_1 \hat{u}_0 | u_1 = 1)) \]

\[ (P(y_0 | x_1 = 0), P(y_0 | x_1 = 1)) \]

\[ (P(y_1 y_1 \hat{u}_0 | u_1 = 0), P(y_1 y_1 \hat{u}_0 | u_1 = 1)) \]

\[ (P(y_1 | x_1 = 0), P(y_1 | x_1 = 1)) \]

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[
(P(y_0 y_1 | u_0 = 0), P(y_0 y_1 | u_0 = 1))
\]

\[
(P(y_0 | x_0 = 0), P(y_0 | x_0 = 1))
\]

\[
(P(y_0 y_1 \hat{u}_0 | u_1 = 0), P(y_0 y_1 \hat{u}_0 | u_1 = 1))
\]

\[
(P(y_1 | x_1 = 0), P(y_1 | x_1 = 1))
\]

\[
\hat{u}_0
\]

\[
\hat{u}_1
\]

\[
\hat{x}_0
\]

\[
\hat{x}_1
\]

\[
\text{probability pair variable}
\]

\[
\text{boolean variable (bit)}
\]
A closer look at successive cancellation

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1))\]
\[(P(y_0|u_0 = 0), P(y_0|u_0 = 1))\]
\[(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1))\]
\[(P(y_1|u_0 = 0), P(y_1|u_0 = 1))\]

Probability pair variable

Boole variable (bit)
A closer look at successive cancellation

\[
\begin{align*}
(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) & \triangleright \hat{u}_0 \\
(P(y_0|u_0 = 0), P(y_0|u_0 = 1)) & \triangleright \hat{x}_0 \\
(P(y_0y_1\hat{u}_0|u_1 = 0), P(y_0y_1\hat{u}_0|u_1 = 1)) & \triangleright \hat{u}_1 \\
(P(y_1|u_1 = 0), P(y_1|u_1 = 1)) & \triangleright \hat{x}_1
\end{align*}
\]

\[\begin{align*}
\triangle & \text{probability pair variable} \\
\square & \text{boolean variable (bit)}
\end{align*}\]
A closer look at successive cancellation

\[ (P(y_0 y_1 | u_0 = 0), P(y_0 y_1 | u_0 = 1)) \]

\[ (P(y_0 | x_0 = 0), P(y_0 | x_0 = 1)) \]

\[ (P(y_0 y_1 \hat{u}_0 | u_1 = 0), P(y_0 y_1 \hat{u}_0 | u_1 = 1)) \]

\[ (P(y_0 | x_1 = 0), P(y_0 | x_1 = 1)) \]

\[ (P(y_1 y_1 \hat{u}_0 | u_1 = 0), P(y_1 y_1 \hat{u}_0 | u_1 = 1)) \]

\[ (P(y_1 | x_1 = 0), P(y_1 | x_1 = 1)) \]

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[ (P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) \]

\[ \hat{u}_0 \]

\[ (P(y_0|u_0 = 0), P(y_0|u_0 = 1)) \]

\[ x_0 \rightarrow y_0 \]

\[ \hat{x}_0 \]

\[ \hat{u}_0 \]

\[ (P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) \]

\[ u_1 \]

\[ (P(y_1|u_1 = 0), P(y_1|u_1 = 1)) \]

\[ x_1 \rightarrow y_1 \]

\[ \hat{x}_1 \]

\[ \hat{u}_1 \]

\[ (P(y_0y_1\hat{u}_0|u_1 = 0), P(y_0y_1\hat{u}_0|u_1 = 1)) \]

\[ \hat{u}_1 \]

\[ (P(y_0y_1\hat{u}_0|u_1 = 0), P(y_0y_1\hat{u}_0|u_1 = 1)) \]

\[ \hat{u}_1 \]

\[ \hat{u}_0 \]

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[(P(y_0 y_1 | u_0 = 0), P(y_0 y_1 | u_0 = 1))\]

\[\hat{u}_0 \quad √\]

\[u_0 \]

\[(P(y_0 y_1 | u_0 = 0), P(y_0 y_1 | u_0 = 1))\]

\[\hat{u}_0 \quad ✓\]

\[u_1 \]

\[(P(y_0 y_1 | u_0 = 0), P(y_0 y_1 | u_0 = 1))\]

\[\hat{u}_1 \quad ?\]

\[\hat{u}_1 \quad ?\]

\[(P(y_0 | x_0 = 0), P(y_0 | x_0 = 1))\]

\[x_0 \rightarrow y_0 \]

\[\hat{x}_0 \]

\[(P(y_1 | x_1 = 0), P(y_1 | x_1 = 1))\]

\[x_1 \rightarrow y_1 \]

\[\hat{x}_1 \]

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[
(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1))
\]

\[u_0\]

\[\hat{u}_0\]

\[u_1\]

\[\hat{u}_1\]

\[x_0 \rightarrow y_0\]

\[x_0 \rightarrow \hat{x}_0\]

\[x_1 \rightarrow y_1\]

\[x_1 \rightarrow \hat{x}_1\]

\[\hat{u}_0\]

\[\hat{u}_1\]

\[\triangle\] probability pair variable

\[\square\] boolean variable (bit)
A closer look at successive cancellation

\[ (P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) \]
\[ \hat{u}_0 \]
\[ (P(y_0y_1|\hat{u}_0 = 0), P(y_0y_1|\hat{u}_0 = 1)) \]
\[ \hat{u}_0 \]
\[ (P(y_0y_1\hat{u}_0|u_1 = 0), P(y_0y_1\hat{u}_0|u_1 = 1)) \]
\[ \hat{u}_1 \]
\[ (P(y_0|u_0 = 0), P(y_0|u_0 = 1)) \]
\[ x_0 \rightarrow y_0 \]
\[ \hat{x}_0 \]
\[ (P(y_0|u_0 = 0), P(y_0|u_0 = 1)) \]
\[ \hat{u}_1 \]
\[ (P(y_0|u_1 = 0), P(y_0|u_1 = 1)) \]
\[ x_1 \rightarrow y_1 \]
\[ \hat{x}_1 \]

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[
\begin{align*}
(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) & \\
(P(y_0y_1\hat{u}_0|u_1 = 0), P(y_0y_1\hat{u}_0|u_1 = 1)) & \\
(P(y_0y_1\hat{u}_1|u_0 = 0), P(y_0y_1\hat{u}_1|u_0 = 1)) & \\
(P(y_0y_1\hat{u}_1|u_0 = 0), P(y_0y_1\hat{u}_1|u_0 = 1)) & \\
(P(y_0|0 = 0), P(y_0|0 = 1)) & \\
(P(y_0|0 = 0), P(y_0|0 = 1))
\end{align*}
\]

probability pair variable

boolean variable (bit)
A closer look at successive cancellation

\[
\begin{align*}
(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) & \quad \triangledown \quad \checkmark \quad u_0 \quad \triangleleft \\
(P(y_0y_1|u_0 = 0), P(y_0y_1|u_0 = 1)) & \quad \triangledown \quad \checkmark \quad \hat{u}_0 \\
(P(y_0y_1|u_1 = 0), P(y_0y_1|u_1 = 1)) & \quad \triangledown \quad \checkmark \quad u_1 \quad \triangleleft \\
(P(y_0y_1|u_1 = 0), P(y_0y_1|u_1 = 1)) & \quad \triangledown \quad \checkmark \quad \hat{u}_1 \\
(x_0 \to y_0 & ) \quad \square \quad \checkmark \\
(x_1 \to y_1 & ) \quad \square \quad \checkmark \quad \hat{x}_0 \quad \hat{x}_1 \\
(P(y_0|x_0 = 0), P(y_0|x_0 = 1)) & \quad \triangledown \quad \checkmark \quad x_0 \\
(P(y_1|x_1 = 0), P(y_1|x_1 = 1)) & \quad \triangledown \quad \checkmark \quad x_1
\end{align*}
\]
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level \( t \) is \( O(n/2^t) \).
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 

$$4 3 2 1 0$$

$u_0$ $y_0$
$u_1$ $y_1$
$u_2$ $y_2$
$u_3$ $y_3$
$u_4$ $y_4$
$u_5$ $y_5$
$u_6$ $y_6$
$u_7$ $y_7$
$u_8$ $y_8$
$u_9$ $y_9$
$u_{10}$ $y_{10}$
$u_{11}$ $y_{11}$
$u_{12}$ $y_{12}$
$u_{13}$ $y_{13}$
$u_{14}$ $y_{14}$
$u_{15}$ $y_{15}$
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point

The memory needed to hold the variables at level $t$ is $O(n/2^t)$.
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point

The memory needed to hold the variables at level \( t \) is \( O(n/2^t) \).
A larger example

Key point

The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 

---

**Diagram Description:**

The diagram above illustrates a circuit or a graph structure with nodes labeled as $u_0, u_1, u_2, \ldots, u_{15}$. Each node is connected to its corresponding output node labeled as $y_0, y_1, y_2, \ldots, y_{15}$. The connections are directed, indicating the flow of data or signals from input to output. The structure is typical of binary tree or recursive circuit designs, where the output of each node is fed into the next level, and the process continues recursively.
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point

The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point

The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$.
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$.
A larger example

Key point
The memory needed to hold the variables at level \( t \) is \( O(n/2^t) \).
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$.
A larger example

Key point

The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point

The memory needed to hold the variables at level \( t \) is \( O(n/2^t) \).
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point

The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$. 
A larger example

Key point
The memory needed to hold the variables at level $t$ is $O(n/2^t)$.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

**Key point**

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

**Key point**

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point

Level $t$ is written to once every $O(2^{m-t})$ stages.
A larger example

Key point
Level $t$ is written to once every $O(2^{m-t})$ stages.
Application to list decoding

- In a naive implementation, at each split we make a copy of the variables.

- We can do better:
  - At each split, flag the corresponding variables as belonging to both paths.
  - Give each path a unique variable (make a copy) only before that variable will be written to.
  - If a path is killed, deflag its corresponding variables.

- Thus, instead of wasting a lot of time on copy operations at each stage, we typically perform only a small number of copy operations.

This was a mile high view, there are many details to be filled (book-keeping, data structures), but the end result is a running time of $O(L \cdot n \log n)$ with $O(L \cdot n)$ memory requirements.