A Unified Algorithm for Accelerating Edit-Distance Computation via Text-Compression

Danny Hermelin, Gad M. Landau, Shir Landau and Oren Weimann
Edit Distance: Quick Review

- The min cost of transforming one string into another via insertion/deletion/replacement.
- One of the fundamental problems in computer science.
- Standard solution: dynamic programming (DP). Time complexity on strings of length $N$: $O(N^2)$.
- Recent approximation algorithms: Rabani et al.
Edit Distance: Quick Review

$$T[i,j] = \min \begin{cases} 
T[i-1,j] + \text{cost of deleting } a_i \\
T[i,j-1] + \text{cost of inserting } b_j \\
T[i-1,j-1] + \text{cost of replacing } a_i \text{ with } b_j 
\end{cases}$$
Acceleration via Compression

- Use compression to accelerate the above DP solution

Basic idea:

1. Compress the strings
2. Compute edit-distance of compressed strings
Acceleration via Compression

- Run-Length encoding
  - Bunke and Csirik ’95
  - Series of results: Apostolico et al. $O(n^2 \log n)$ for LCS. Arbel et al. $O(nN)$ for edit-distance.

- LZW-LZ78
  - Crochemore et al ‘03
    - $O(nN)$
    - Constant size alphabets: $O(N^2/\log N)$

- Masek, Paterson ’80
  - Exploit repetitions + “Four-Russians technique” $O(N^2/\log^2 N)$ for any strings, rational scoring function
  - Bille, Farach-Colton ‘05 extend to general alphabets

\[ N = \text{total length of strings} \]
\[ n = \text{length of compression} \]
A Unified Acceleration

- Find a general compression-based edit distance acceleration for any compression scheme...
- Can handle two strings that compress well on different schemes
- Towards breaking the quadratic barrier of edit-distance computation
Basic idea of the Crochemore *et al.* algorithm

1. Divide DP-grid into blocks
2. Build a repository of DIST tables for all blocks
3. Compute edit distance by computing boundaries of each block
   - propagate DP-values using SMAWK
Definition: *xy-partition of* $G$:
- Partitioning of $G$ into blocks:
  - Boundary size of blocks: $O(x)$
  - $O(y)$ blocks in each row and each column
A Unified Acceleration

Running-time:

- Constructing the repository:
  - \( \#\text{DIST} \times O(x^2 \log x) \) time (Apostolico et al. ‘90)

- Propagating the DP-values:
  - \( O(Ny) \) time (SMAWK).

\[ N = \text{total length of strings} \]
\[ n = \text{length of compression} \]
A Unified Accelerator

- Find a good xy-partition for any pair of compressible strings.
- How can we achieve this?

Using Straight-Line Programs
Straight-line Programs (SLP)

- Context-free grammar
- Every grammar generates exactly one string
- Allow 2 types of productions:
  - $X_i \rightarrow a$ (a is a unique terminal)
  - $X_i \rightarrow X_pX_q$ ($i > p,q$)
Example: S = abaababaabaab

Use Fibonacci SLP:

\[ \begin{align*}
X_1 & \rightarrow a \\
X_2 & \rightarrow b \\
X_3 & \rightarrow X_2X_1 \\
X_4 & \rightarrow X_3X_2 \\
X_5 & \rightarrow X_4X_3 \\
X_6 & \rightarrow X_5X_4 \\
X_7 & \rightarrow X_6X_5
\end{align*} \]
Straight-line Programs (SLP)

- Why SLP?
  - Result of most compression schemes can be transformed into SLP (Rytter ‘03)
    - LZ, RLE, Byte-Pair, Dictionary methods…
    - Compressed approximation:
      - String length: $N$
      - Encoding produces $n$ blocks
      - Get SLP of size $m=O(n\log N)$ in $O(m)$ time
      - $m$ within $\log N$ factor from minimal SLP
Straight-line Programs (SLP)

- Rytter, Lifshits - used SLP for accelerating pattern matching via compression
- Lifshits –
  - various hardness results for SLP e.g.: edit-distance, Hamming distance
  - $O(n^3)$ for determining equality of SLPs
- Tiskin –
  - $O(nN^{1.5})$ algorithm for computing longest common subsequence between two SLPs
  - Can be extended at constant factor to compute edit distance between SLPs
Constructing the \textit{xy}-partition

- Use SLP to create a \textit{xy}-partition of $G$
  - At most $O(n^2)$ DIST tables.
Constructing the xy-partition

- For any $x$, we can construct an xy-partition with $y = O(nN/x)$ in $O(N)$ time.
  - We will choose $x$ later.
  - Use SLP parse tree.
Choose $nN/x$ “key vertices” in tree s.t. each vertex is variable generating substring of length $O(x)$

1. Find $O(N/x)$ variables in $A'$ generating disjoint substrings of length between $x$ and $2x$
2. Substrings in $A$ not yet covered can be generated using $O(n)$ additional variables for each $2$ found in step 1
3. Total – $O(nN/x)$ vertices ($A$ is the concatenation of all generated substrings of key vertices)
Putting it all together

- Using SLP to compute edit-distance
  1. Create xy-partition of G according to SLP
  2. Build a repository of DIST tables of blocks in xy-partition
  3. Compute edit distance by computing boundaries of each block (propagate DP values using SMAWK)

- Total running time $O(n^2x^2\lg x + Ny)$

- constructing repository of all DIST tables
- propagating DP values
Putting it all together

- Total running time $O(n^2x^2 \log x + Ny)$

- For all $x$ we can build $xy$-partition with $y = O(nN/x)$.

- Choose $x$ so as to balance both terms above.

- Total: $O(n^{1.34}N^{1.34})$ time.
Extensions

1. $O(n^{1.4}N^{1.2})$ time for rational Scoring:
   - use recursive construction of DIST tables, compute repository in $O(n^2x^{1.5})$
   - Based on:
     - $x^x$ DIST table stored succinctly in $O(x)$ space (Schmidt)
     - This allows to merge 2 DIST tables in $O(x^{1.5})$ time (Tiskin)

2. Arbitrary scoring and “Four Russians”:
   - $\Omega(lg N)$ speedup for any string (not necessarily compressible)
   - Short enough substrings must appear many times (Masek and Paterson)
   - With SLP we expand this idea to arbitrary scoring functions
Thank You!!!