Approximate Matching

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Search when exact equality is too strict

Approximate matching finds text regions close to a pattern under a stated error model. It supports spelling correction, biological sequence search, record linkage, and noisy measurements. "Similar" is not an algorithmic specification: the distance, threshold, occurrence boundaries, and output policy must all be defined.

Dynamic programming gives a general baseline. Faster methods exploit a small threshold, reject impossible candidates cheaply, pack states into words, or preprocess the text.

2 Learning goals

After studying this note, you should be able to:

- distinguish Hamming and edit-distance occurrences;
- adapt pairwise edit-distance DP to substring search;
- derive generalised block-edit transitions and precompute their applicable endpoints;
- recover the start and edit script of a match;
- explain when a diagonal band is safe and when it is not;
- prove no-false-negative partition and q-gram filters;
- trace a bit-parallel matcher for mismatches; and
- select a method using pattern length, threshold, word size, and candidate frequency.

3 Prerequisites

You should know the edit-distance recurrence from the dynamic-programming note and exact search from the exact-matching note. The recurrence and required exact-search facts are restated here so this chapter can be read independently.

4 Problem models and output semantics

Let pattern $P[1..m]$ and text $T[1..n]$ be strings over alphabet Σ.

- **At most k mismatches:** report length-m text windows whose Hamming distance from P is at most k. Only substitutions are allowed.
- **At most k edits:** report text substrings whose insertion/deletion/substitution distance from P is at most k. Under unit costs, their lengths may range from $\max\{0, m-k\}$ to $m+k$.
- **Best match:** find a substring minimising a chosen distance, possibly subject to a length or location restriction.

One can report every qualifying pair `(start,end)`, one best start for each end, nonoverlapping matches, or maximal regions. These are different output problems. The basic DP below gives the best score for each endpoint and can recover one tied start by traceback. Enumerating all tied starts can require additional output-sensitive work.

5 Dynamic programming for substring search

Let $D[i, j]$ be the minimum edit cost of matching pattern prefix $P[1..i]$ to a suffix of text prefix $T[1..j]$. Thus column j is anchored at text boundary j, although final deletions need not consume a text character. For unit costs use

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 & \text{delete } P_i, \\ D[i,j-1]+1 & \text{insert } T_j, \\ D[i-1,j-1]+[P_i \neq T_j] & \text{match or substitute.} \end{cases}$$

The crucial boundaries are

$$D[i, 0] = i, \qquad D[0, j] = 0.$$

The free top row discards any text prefix before a candidate start. In pairwise global distance it would instead be $D[0, j] = j$.

```
approximate_endpoints(P, T, k):
    for j = 0 .. n: D[0,j] = 0
    for i = 1 .. m: D[i,0] = i
    for j = 1 .. n:
        for i = 1 .. m:
            D[i,j] = min(D[i-1,j]+1,
                         D[i,j-1]+1,
                         D[i-1,j-1] + (P[i] != T[j]))
        if D[m,j] <= k: report endpoint j
```

The invariant is exactly the state definition. Consider the final operation of an optimum alignment: deletion, insertion, or diagonal match/substitution gives the three candidates. Conversely, appending the stated operation to an optimal predecessor gives a feasible alignment. Induction over the grid proves the recurrence.

The full method takes $\Theta(mn)$ time and space. Two columns suffice for scores in $O(m)$ space. To recover a start and edit script, retain predecessors or recompute a region when an endpoint is found.

6 Worked endpoint and traceback example

For pattern `abc` and text `zabxc`, the substring-search table is:

D	ε	z	a	b	x	c
ε	0	0	0	0	0	0
a	1	1	0	1	1	1
b	2	2	1	0	1	2
c	3	3	2	1	1	1

With $k = 1$, endpoint 4 represents `abx` with one substitution; endpoint 5 represents `abxc` with insertion `x`. Tracing the second path from $D[3, 5]$ reaches row zero at column 1, so the occurrence starts at text position 2. The leading `z` costs nothing because the top row is zero.

7 Generalised edit operations

An edit rule may replace a whole string α by another string β, rather than consuming at most one character from each side. Store every finite-cost rule as

$$r = (\alpha_r, \beta_r, w_r), \qquad \alpha_r, \beta_r \in \Sigma^*, \qquad w_r \geq 0,$$

where the two strings are not both empty. Equivalently, every pair $\alpha \to \beta$ has a cost, with $+\infty$ meaning that the operation is disallowed. An implementation stores only the finite catalogue.

This chapter uses a left-to-right **non-overlapping alignment model**: one rule consumes a block of the original pattern and a block of the chosen text substring, and its output is not edited again. Unrestricted repeated string rewriting is a different problem and is not computed by this acyclic DP.

Ordinary weighted edit distance is the special case containing identity rules $a \to a$, deletions $a \to \varepsilon$, insertions $\varepsilon \to b$, and substitutions $a \to b$. Multi-character rules can represent historical spelling, pronunciation, dialect, or transliteration, such as `hw` $\to$ `f` and `a` $\to$ `aa`. The direction matters: here rules transform the pattern into a text substring, so the cost need not be symmetric.

7.1 Recurrence

Let $D[i, j]$ be the minimum cost of aligning pattern prefix $P[1..i]$ with text substring material ending at boundary j. A rule r is applicable at (i, j) when α_r is a suffix of $P[1..i]$ and β_r is a suffix of $T[1..j]$. Write $a_r = |\alpha_r|$ and $b_r = |\beta_r|$. Then

$$D[i, j] = \min_{\substack{r \text{ applicable at } (i,j) \\ a_r + b_r > 0}} \{D[i - a_r, j - b_r] + w_r\}.$$

For global distance initialise $D[0, 0] = 0$ and every other entry to $+\infty$. Insertions and deletions, if permitted, generate the remaining boundary values through their own rules. For approximate substring search instead initialise

$$D[0, j] = 0 \qquad \text{for every } j.$$

Then $D[m, j] \leq k$ reports an occurrence ending at boundary j, and traceback to row zero recovers a start. Every transition strictly increases $i + j$, even if one rule side is empty, so row-major evaluation is a valid topological order. Correctness follows by induction: every nonempty alignment has one final rule represented by the recurrence, and appending any applicable rule to a predecessor constructs a valid alignment.

7.2 Precompute both rule sides

Rescanning every α_r and β_r at every matrix cell is unnecessarily expensive. For each distinct nonempty α, run KMP once over P and record the boundaries where it ends. Do the same for each distinct nonempty β over T. An empty side matches every boundary. When the catalogue contains many distinct strings, one Aho–Corasick automaton per side replaces the repeated KMP scans with one multi-pattern scan.

It helps to separate two layers. Endpoint matching answers only where each rule side occurs; the DP then joins compatible pattern and text endpoints into a minimum-cost path. KMP or Aho–Corasick accelerates the first layer but does not choose edits or remove the need to process genuinely applicable rules.

```
rule_endpoints(S, side, rules):
    occurrence_ends[epsilon] = [0, 1, ..., length(S)]
    for each distinct nonempty string q among the rule sides:
        occurrence_ends[q] = empty list
        for start in kmp_all_occurrences(S, q):
            occurrence_ends[q].append(start + length(q))

    for each rule id r:
        ends[r] = occurrence_ends[side(r)]  // share equal-side lists
    return ends
```

The lists for both sides determine exactly which cells admit each rule. Materialising these candidates makes the later DP bound explicit:

```
rule_candidates(P, T, rules):
    Pend = rule_endpoints(P, alpha, rules)
    Tend = rule_endpoints(T, beta, rules)
    applicable[0 .. m, 0 .. n] = empty lists

    for each rule id r:
        for i in Pend[r]:
            for j in Tend[r]:
                applicable[i,j].append(r)
    return applicable
```

Now evaluate the acyclic grid in row-major order:

```
generalised_edit(P, T, rules, substring_search, threshold):
    applicable = rule_candidates(P, T, rules)
    fill D[0 .. m, 0 .. n] with infinity
    fill predecessor with NONE

    if substring_search:
        for j = 0 .. n: D[0,j] = 0
    else:
        D[0,0] = 0

    for i = 0 .. m:
        for j = 0 .. n:
            for r in applicable[i,j]:
                a = length(alpha[r]); b = length(beta[r])
                if a + b = 0: continue
                candidate = D[i-a,j-b] + weight[r]
                if candidate < D[i,j]:
                    D[i,j] = candidate
                    predecessor[i,j] = r

    if substring_search:
        return all j with D[m,j] <= threshold
    return D[m,n]
```

Suppose there are g rules, u distinct nonempty α strings, v distinct nonempty β strings, and their total stored length is L. Let E be the number of occurrence endpoints reported for these distinct strings; rules with an identical side share one endpoint list. Repeated KMP preprocessing and assigning the shared lists to rules takes $O(g + um + vn + L + E)$ time. Aho–Corasick reduces the scanning part, giving $O(g + m + n + L + E)$ under its alphabet-transition model.

Let

$$C = \sum_r e_P(r) e_T(r),$$

where $e_P(r)$ and $e_T(r)$ count endpoints of the two sides. The outer loop reads all g rules and the nested endpoint loops construct exactly C candidates. Initialising and filling the dense grid therefore takes $O(g + mn + C)$ time after matching.

The value, predecessor, and dense candidate-bucket grids use $O(mn + C)$ space. The candidate payload itself is $O(C)$, and a sparse map can avoid allocating empty buckets. If storing all candidates is undesirable, per-boundary rule bit sets can instead be intersected on demand in $O(mn\lceil g/w \rceil + C)$ word operations.

In repetitive strings, many rules may really apply at the same cell. KMP and Aho–Corasick eliminate repeated character comparison, not that output-sensitive work. A two-row optimisation is not automatic because a rule may jump back several pattern positions.

7.3 Worked rule trace

Transform `ahwrika` into `aafrika`. In addition to zero-cost identity rules and unit-cost single-character edits, suppose the catalogue contains

$$a \to aa \text{ at cost } 0.2, \qquad hw \to f \text{ at cost } 0.3.$$

The first rule gives $D[1, 2] = D[0, 0] + 0.2$. KMP marks `hw` as ending at pattern boundary 3 and `f` as ending at target boundary 3, so the second rule gives

$$D[3, 3] = D[1, 2] + 0.3 = 0.5.$$

The remaining `r`, `i`, `k`, and `a` use identity rules, giving $D[7,7] = 0.5$. Ordinary unit-cost Levenshtein distance can instead substitute `h` by `a` and `w` by `f`, with total cost 2; the block rules encode that these particular changes should be considerably cheaper.

A unit-edit diagonal band is not automatically safe here. One cheap rule may consume strings of very different lengths and jump across many diagonals. Any safe band must be derived from the permitted rule lengths and a lower bound connecting their costs to net length change.

8 Thresholds, cutoffs, and safe bands

While verifying two strings from a **fixed common start**, any alignment using at most k insertions and deletions satisfies $|i - j| \leq k$: the difference in consumed prefix lengths changes by at most one per insertion or deletion. It is therefore safe to compute only that diagonal band. For strings of comparable length this takes $O((k+1)m)$ time and $O(k+1)$ or $O(m)$ working space, depending on layout.

Substring search is different because the start offset is unknown. One fixed band around the table's main diagonal is not safe: a valid occurrence far into the text lies around another diagonal. Safe choices include:

- use an exact/filtering step to propose a start and band only its verifier;
- run a separate band relative to each candidate start; or
- use a specifically derived Ukkonen/Landau–Vishkin-style algorithm that tracks the furthest reachable position on each error diagonal.

Capping values above k at $k+1$ is useful but does not alone prove $O(kn)$ time. In repetitive input, many states may remain active. State the particular threshold algorithm and its assumptions rather than attributing every narrow-looking computation to "banding."

9 Filter and verify by partitioning

A **filter** returns candidates cheaply. It may return false positives, but a correct filter must not discard a true match. A verifier then computes the real distance.

Assume $0 \leq k < m$, so P can be partitioned into $k+1$ nonempty, nonoverlapping pieces. Under at most k mismatches, at most k pieces contain a changed position, so at least one piece occurs exactly at its expected offset. Search all pieces with KMP or Aho–Corasick, convert their hits into candidate starts, deduplicate, and verify the full window. When $k \geq m$, this particular nonempty-piece filter provides no useful partition guarantee.

For edit distance, each edit can disrupt at most one disjoint piece, so an unchanged piece still occurs exactly, but preceding insertions and deletions shift its text position. If a piece begins at pattern offset a and occurs at text position h, a candidate start can lie within k positions of $h - a$. The verifier must also allow candidate lengths $m - k$ through $m + k$. This positional slack is essential for a no-false-negative guarantee.

For `ALGORITHM` and $k = 2$ mismatches, use pieces `ALG`, `ORI`, and `THM`. Every qualifying length-nine window preserves at least one complete piece. An exact hit is only a candidate; Hamming verification still decides whether the other positions contain at most two errors.

Filtering helps when few candidates survive. On repetitive text or very short pieces, verification can dominate and the worst case can return to quadratic behaviour.

10 q-gram counting filters

A q-gram is a length-q substring, with $1 \leq q \leq m$. Pattern P has $N = m - q + 1$ overlapping q-grams counted by position. One substitution lies in at most q of them. Therefore a window with at most k mismatches has at least

$$\max(0, N - kq)$$

position-aligned q-grams equal to those of P. Rejecting a window below this threshold has no false negatives for the stated mismatch model.

An inverted q-gram index maps each gram to text or document positions. Candidate generation can count hits before verification. Multiset counting, position-aware counting, and document-level counting are not interchangeable. With insertions and deletions, positions shift and q-grams are created as well as destroyed; a safe edit-distance filter must use a theorem derived for its exact counting convention and must allow positional slack. Overlap is why one cannot count every disagreeing q-gram as an independent edit.

11 Bit-parallel mismatch search

For substitutions only, extend Shift-And with one bit vector R_e for each allowed error count e. Bit i says that prefix $P[1..i+1]$ matches a suffix of the processed text using at most e mismatches. Let $M[c]$ mark positions of character c, and initialise every R_e to zero. Using old vectors from before the next text character c:

$$R'_0 = ((R_0 \ll 1) \mid 1) \,\&\, M[c],$$
$$R'_e = (((R_e \ll 1) \mid 1) \,\&\, M[c]) \mid ((R_{e-1} \ll 1) \mid 1), \quad e \geq 1.$$

The first term extends with a matching character; the second spends one mismatch. A match with at most k errors ends when bit $m-1$ of R_k is set. Update error layers from old copies or from high e downward so a text character is not consumed twice.

If $m \leq w$, this costs $O((k+1)n)$ word operations and $O(k+|\Sigma|)$ words. Multiple words extend longer patterns. Insertions and deletions require additional terms because one side may advance alone; Myers' bit-vector algorithm encodes full edit-distance columns more compactly. Its speed is still a word-RAM result, not constant time independent of m/w.

12 Character classes, wildcards, and practical tools

A pattern position may denote a character class, such as `[A-G]`, or a wildcard. Bit masks handle single-position classes naturally by setting the corresponding bit in every permitted character mask. An unbounded wildcard such as `.*` changes the automaton structure and cannot be treated as one ordinary position.

Nonuniform edit costs require the weighted DP model from the previous note. Command-line fuzzy-search tools such as `agrep` may expose mismatches, insertions, deletions, classes, or separate operation costs. Before comparing speed, verify their distance semantics, Unicode/byte model, line-boundary policy, and whether reported results are exact under that policy.

13 Method trade-offs

Method	Main time bound	Best use
Full substring DP	$O(mn)$	general baseline and traceback
Generalised-rule DP	$O(g+mn+C)$ after endpoint matching	transliteration and spelling variants
Banded fixed-start verifier	$O((k+1)m)$	a known candidate start
$k+1$ piece filter	exact-search cost plus verification	small k, selective pieces
q-gram index	candidate dependent	repeated queries or documents
Bit-parallel mismatches	$O((k+1)n\lceil m/w \rceil)$ word operations	short patterns, small k

14 Connections

- This note changes the boundary conditions of the pairwise DP from the dynamic-programming note.
- Generalised block rules extend weighted edit distance with an explicit operation catalogue.
- Its exact-piece filters reuse KMP or Aho–Corasick from exact matching.
- Error-layer bit vectors are compact NFA simulations, connecting to the automata note.
- q-gram inverted indexes lead toward the preprocessing perspective of full-text indexing.

15 Common mistakes

- Saying "similar" without defining distance, threshold, and reported region.
- Using Hamming distance when insertions or deletions shift alignment.
- Reporting only endpoints when starts are required.
- Applying a main-diagonal band to substring search with unknown start.
- Allowing a generalised rule's output to be edited again despite using the non-overlapping DP model.
- Assuming that $\alpha \to \beta$ and $\beta \to \alpha$ have the same cost.
- Assuming KMP removes the cost of processing every rule occurrence pair.
- Designing a fast filter without proving it has no false negatives.
- Ignoring overlapping q-grams when deriving a threshold.
- Updating bit-vector error layers in an order that reuses the current character.
- Comparing fuzzy-search tools without checking their semantics.

16 Self-check

1. Why is the top row zero while the left column still increases?
2. Trace one optimal path for `abc` against the occurrence `abxc`.
3. Why does rule `a` $\to$ `aa` lead from $(i-1, j-2)$ to (i, j)?
4. What work does KMP remove from the generalised-rule DP, and what candidate work remains?
5. Prove the fixed-start band condition $|i - j| \le k$.
6. Why is the same band not globally safe in a long text?
7. Derive the exact-hit position range for a partition piece under k edits.
8. For $m = 20$, $q = 4$, and $k = 2$ mismatches, what q-gram threshold follows?
9. Which term in the bit recurrence spends a mismatch?

17 Revision summary

- Approximate matching begins with an explicit error model and output convention.
- A zero top row adapts edit-distance DP to substring endpoints.
- Generalised rules replace non-overlapping blocks; KMP or Aho–Corasick precomputes where their two sides end.
- Diagonal bands are safe relative to a known alignment origin, not an unknown global start.
- Filters require a proof that every true match produces at least one candidate.
- q-gram overlap determines the correct counting threshold.
- Bit parallelism accelerates many states at once but retains word-size assumptions.

18 Implementations and hands-on exploration

- **Python pairwise distance:** RapidFuzz Levenshtein supports integer insertion, deletion, and substitution weights plus score cutoffs. Each weight is shared by an operation type; it does not implement symbol-dependent or multi-character $\alpha \to \beta$ rules.
- **C/C++ and Python alignment:** Edlib implements unit-cost Levenshtein distance and global, prefix, and infix modes using Myers bit vectors, with optional locations and paths. The C API treats one symbol as one byte and reports zero-based, inclusive end positions; convert explicitly to this note's one-past-the-end boundaries. Edlib does not expose weighted, affine-gap, or generalised block costs.

- **Rust bit-parallel search:** `bio::pattern_matching::myers` reports substring matches within unit edit distance and can recover starts and edit paths over byte slices. `find_all_end` returns an inclusive end index, while `find_all` returns a half-open `(start,end)` range. The simple form is limited by its 64- or 128-bit vector; the block form supports longer patterns with threshold-dependent work.
- **C++ indexed search:** SeqAn3 index search configures total, substitution, insertion, and deletion limits over an FM or bidirectional FM index, with explicit all, single-best, all-best, and strata hit policies. These are error counts, not arbitrary edit-operation costs.
- **Python fuzzy regex:** `regex` allows bounded or weighted single-character insertions, deletions, and substitutions inside a regular expression. Default, `ENHANCEMATCH`, and `BESTMATCH` choose different search policies, and none supplies a catalogue of multi-character block rules. Unicode-string spans use Python string indices; bytes-pattern spans use byte indices.

18.1 Small experiments

1. Enumerate every pair of strings over `{a,b}` of length at most 4. Assert that your unit-cost DP, RapidFuzz, and Edlib global mode return the same distance, then test how each API signals that a cutoff was exceeded.
2. Run Edlib in infix mode for pattern `abc` and text `zabxc`, request locations and a path, and compare its chosen best span with the qualifying endpoints in the worked table. Explain any tie. Separately log the applicable cells of the `a` → `aa` and `hw` → `f` rules to see what endpoint preprocessing contributes.

19 Sources and further study

- Approximate matching lecture deck, Jaak Vilo.
- Dynamic Programming I and II, Jaak Vilo, for the generalised edit model.
- Information Retrieval of Word Form Variants in Spoken Language Corpora Using Generalized Edit Distance, Orasmaa, Käärik, Vilo, and Hennoste, LREC 2010.
- Esko Ukkonen, "Finding approximate patterns in strings," *Journal of Algorithms* 6 (1985).
- Gene Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *JACM* 46(3), 1999.