More fuzzy string matching!
Plan for today:

Tries
Simple uses of tries
Fuzzy search with tries
Levenshtein automata
A trie is essentially a **prefix tree**:

A: 15
i: 11
in: 5
inn: 9
to: 7
tea: 3
ted: 4
ten: 12
Simple uses of tries:

Key lookup in $O(m)$ time, predictably.

(Compare to hash table: best-case $O(1)$, worst-case $O(n)$, depending on key)

Fast longest-prefix matching

IP routing table lookup

For an incoming packet, find the closest next hop in a routing table.
Simple uses of tries:

Fast longest-prefix matching

Useful for autocompletion:

“All words/names/whatevers that start with XYZ...”
The problem with tries:
When the space of keys is sparse, the trie is not very compact:

(One) Solution: PATRICIA Tries
(One) Solution: PATRICIA Tries

Key ideas: edges represent more than a single symbol; nodes with only one child get collapsed.
One could explicitly represent edges with multiple symbols...

... but that would complicate matching.
Instead, each internal node stores the *offset* for the next difference to look for:
Instead, each internal node stores the offset for the next difference to look for:
Instead, each internal node stores the offset for the next difference to look for:
Instead, each internal node stores the *offset* for the next difference to look for:
Instead, each internal node stores the offset for the next difference to look for:
Instead, each internal node stores the *offset* for the next difference to look for:
Instead, each internal node stores the *offset* for the next difference to look for:

![Diagram showing the process of finding differences in a sequence of words.](image-url)
Instead, each internal node stores the offset for the next difference to look for:
Instead, each internal node stores the *offset* for the next difference to look for:
Plan for today:

Tries
Simple uses of tries
Fuzzy search with tries
Levenshtein automata
Fuzzy search with tries

Problem: we want to search a dictionary for words similar to a query.

Example: “Smyth” and “Zmith” should retrieve “Smith”,

“Levenstien” should retrieve “Levenshtein”, etc.

By “similar,” we mean “edit distance less than some threshold $\delta$. ”
One solution:
Compute pairwise edit distance between our query \( q \) and every word \( w_i \) in our dictionary;

Match if \( \text{sim}(q, w_i) \leq \delta \)

A (slightly) better solution:
Speed up pairwise edit distance computation using \textit{prefix pruning}. 
Prefix pruning’s key idea:

If we only care whether strings $r$ and $s$ have an edit distance less than some threshold...

...we can do early termination of our computation as soon we exceed that threshold.

One solution:
Compute pairwise edit distance between our query $q$ and every word $w_i$ in our dictionary;

Match if $\text{sim}(q, w_i) \leq \delta$

A (slightly) better solution:
Speed up pairwise edit distance computation using \textit{prefix pruning}. 
Neither are very good solutions for any kind of “on-line” use case:

Query autocompletion, fuzzy searching, spellchecking, etc.

(our dictionary is large, number of searches is high, etc. etc.)
A better solution: use a trie!

1. Build a trie out of our dictionary;

2. Iterate through $q$; at each point, identify a set of active nodes of the trie.

A node $n$ is “active” with respect to a prefix $q_i$ if the edit distance between $q_i$ and the prefix represented by $n$ is $\leq \delta$. 
A better solution: use a trie!

1. Build a trie out of our dictionary;

2. Iterate through $q$; at each point, identify a set of active nodes of the trie.

3. Stop when we reach the end of $q$ or no longer have active nodes.

Active nodes that happen to be leaves represent matches.
A better solution: use a trie!

1. Build a trie out of our dictionary;

2. Iterate through $q$; at each point, identify a set of active nodes of the trie.

3. Stop when we reach the end of $q$ or no longer have active nodes.

Active nodes that happen to be leaves represent matches.
Intuition: at each symbol $i$ in $q$, the set of active nodes will be related to the set from $q_{i-1}$.

So, we don’t need to visit every node in the trie!
Intuition: at each symbol $i$ in $q$, the set of active nodes will be related to the set from $q_{i-1}$.

So, we don’t need to visit every node in the trie!

(a) Initialize  
(b) query “n”
Intuition: at each symbol \( i \) in \( q \), the set of active nodes will be related to the set from \( q_{i-1} \).

So, we don’t need to visit every node in the trie!

(a) Initialize  
(b) query “n”  
(c) query “nl”

\( O \) ED = 0  \( \bigcirc \) ED = 1  \( \bigcirc \) ED = 2

Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW ’09
Intuition: at each symbol $i$ in $q$, the set of active nodes will be related to the set from $q_{i-1}$.

So, we don’t need to visit every node in the trie!
Intuition: at each symbol $i$ in $q$, the set of active nodes will be related to the set from $q_{i-1}$.

So, we don’t need to visit every node in the trie!

(a) Initialize  (b) query “n”  (c) query “nl”  (d) query “nli”  (e) query “nlis”
Related problem: the “similarity join”

We have two bags of words, $R$ and $S$.

Goal: identify pairs of similar words.

Example:

$R = \{ \text{kobe, ebay, ...} \}$

$S = \{ \text{bag, koby, ...} \}$

We would want to identify pairs such as $<\text{kobe, koby}>$
Again, one solution is pairwise edit distance calculation...

... but if $R$ and $S$ are very large, that will be incredibly time consuming, even with prefix pruning!

One solution: use the trie search method!

Build a trie representing $R$;

For every string $s$ in $S$, identify the active nodes $A_s$ of $R$’s trie; for each leaf node $r$ in $A_s$, produce $<s,r>$. 

Another solution: use sub-trie pruning

Intuition: given the set of active nodes $A_n$ for a particular trie node $n$...

... we can say that only children of nodes in $A_n$ could possibly be similar to children of node $n$.

We can use this fact to speed up extraction of similar pairs.

Let us consider the case where our two sets are actually one set ($R = S$), and we simply want to identify similar pairs.
Algorithm:

1. Build a trie for our set of words;

2. Traverse the trie in preorder. At each node, compute its set of active nodes $A$.

3. At each leaf node $n$, identify any leaf nodes in $A_n$; these are similar pairs.

As we traverse, we must keep the current node’s ancestor’s set of active nodes in memory; total time complexity is $O(\delta |A_T|)$.
In this section, we propose a trie-traversal-based method.

### 3.1 Trie-traverse algorithm

Fig. 4

Trie-join: a trie-based method for efficient string similarity joins

441

To self-join, that is, find all similar string pairs recursively (Lines 5–6).

Example 1

Consider the string set and the corresponding trie structure in Fig. 5.

SimilarPair

calls itself to compute the similar string pairs of its descendants (Lines 6–7).

The pseudo-code of the above-mentioned incremental algorithm. If

Time complexity of computing the similar pair $\tau$'s descendants (Lines 6–7).

is $O(\max(|C|, |\tau|)|A_\tau|)$, where $C$ is the set of the active nodes of ancestors of the current node.

The space complexity is $O(|R|)$, since each node has at most $|R|$ children.

The time complexity of traversing the trie nodes, we need to maintain the trie and the active node only can be computed from its ancestors within $O(|R|)$ steps. Therefore, the time complexity of the algorithm to find all similar string pairs is $O(|R|^2 |A_\tau|)$.

The maximal value of $|A_\tau|$ is $|R|$ for the root node.

For ease of presentation, in the following, we focus on

For each trie node, $S$-\(\sim\)\(\neq\)\(\sim\)\(\neq\) and computes the active-node set for a node in the trie $T_{\tau}$. Intuitively, $T_{\tau}$ utilizes dual subtrie pruning to avoid duplicated computation in

When reaching a leaf node, the algorithm outputs similar string pair.

For each trie node, $S$-$\sim$\(\neq\)\(\sim\)\(\neq\)\(\sim\)\(\neq\) and computes the active-node set for a node in the trie $T_{\tau}$. Intuitively, $T_{\tau}$ utilizes dual subtrie pruning to avoid duplicated computation in

When reaching a leaf node, the algorithm outputs similar string pair.

We can do better!

Intuition: given the set of active nodes $A_n$ for a particular trie node $n$...

... we can say that only children of nodes in $A_n$ could possibly be similar to children of node $n$.

Also: if node $u$ has node $v$ in its active set, $v$ must also have $u$ in its set!
Traverse Trie called 3.1 Trie-traverse algorithm

Fig. 4 Trie-join: a trie-based method for efficient string similarity joins

For ease of presentation, in the following, we focus on $\mathcal{R} = \{S \in \mathcal{A} : \text{Sim}(S, \tau) \geq \epsilon\}$ for string set $\mathcal{A}$ and a threshold $\epsilon$. To find all similar string pairs recursively (Lines 5–6).

In the worst case, the time complexity of computing $|\mathcal{T}|$ is $\mathcal{O}(\max_{\tau \in \mathcal{R}} |\tau|)$, since each edit distance to the root node of $\mathcal{T}$ is at most $\max_{\tau \in \mathcal{R}} |\tau|$. For a string $s \in \mathcal{A}$, its active node set $\mathcal{A}(s)$ is the set of active nodes that are required to compute the edit distance of $s$ to some string in $\mathcal{R}$.

Consider the string set and the corresponding trie structure in Fig. 5. An example to use $\text{findSimilarPair}(\mathcal{R}, \tau)$ as a similarity join algorithm.

Similarly, we compute active node sets of all the trie nodes in the trie, i.e., we compute $\mathcal{A}(s)$ for every leaf node $s$. The space complexity is $\mathcal{O}(\max_{\tau \in \mathcal{R}} |\tau| \cdot |\mathcal{T}|)$, where $|\mathcal{T}|$ is the number of trie nodes.

We always compute its parent's active-node set before its own active-node set. Consider node 2, we use its parent's active note set to find all similar string pairs of node 2's descendants (Lines 6–7). If node 2 is a leaf node, it calls a subroutine $\text{findSimilarPair}\left(\mathcal{R}, \mathcal{A}(s)\right)$ to compute the similar string pairs of node 2.

For each trie node, we first calculates the active-node set of the root node, then traverses the trie in preorder. When reaching a leaf node, we output SimilarPair for all strings in $\mathcal{R}$.

The space complexity is $\mathcal{O}(\max_{\tau \in \mathcal{R}} |\tau| \cdot |\mathcal{T}|)$, where $|\mathcal{T}|$ is the number of trie nodes.
We can compute active nodes as we build the trie, and eliminate duplicate calculations.

By increasing our space complexity, we can reduce the time complexity to \( O(\frac{\delta}{2}|A_T|) \).

There are extensions to the idea that allow for different sets of strings, more space-efficient construction, etc.

See the Feng, et al. article (cited below) for more details!
In this paper we study research challenges that arise naturally in this computing paradigm. The main challenge is efficiency on a large amount of data is especially challenging because of two reasons. First, we allow the query keywords to appear in different attributes with an arbitrary order, and present an incremental algorithm for computing the intersection of the inverted lists efficiently (Section 4.1). Its main idea is to use forward lists of keyword IDs for checking whether a record matches query conditions even a prefix.

We develop novel solutions to these problems. We present several incremental-search algorithms for answering a query approximately. Even though experimental evaluation of the developed techniques supports search on 3.95 million MEDLINE records (http://www.ncbi.nlm.nih.gov/pubmed), this approach is greatly limited by the query speed. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Another prototype, available at http://psearch.ics.uci.edu, supports search on 3.95 million public records. At third prototype, available at http://psearch.ics.uci.edu, supports search on the UC Irvine people directory (http://psearch.ics.uci.edu).

For queries with multiple keywords, we study various techniques for computing the intersection of the inverted lists computationally expensive. Second, we want to support fuzzy search by finding records with keywords that match query approximately. Because of two reasons. First, we allow the query keywords to appear in different attributes with an arbitrary order, and develop a novel algorithm for computing key-tails (Section 4.2). (4) the system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.

1. We first study the case of queries with a single keyword, and present an incremental algorithm for computing key-tails (Section 4.2). (4) The system can already find person records that might be similar to the query keywords, such as a person name "William Kropp" and present an incremental algorithm for computing key-tails. (5) In addition to the "name" attribute. The matched prefixes are highlighted. The word "autocomplete" could have different attributes of the records. In particular, (2) the system searches for the best answers "on the fly" as the user types in a keyword query; (2) interactive, for each keystroke on the client browser, the requirement of a high efficiency. Specifically, we make the following contributions.
Plan for today:

Tries
Simple uses of tries
Fuzzy search with tries
Levenshtein automata
Levenshtein automata

A different approach to solving the fuzzy matching problem uses finite-state automata.

The basic idea: construct an acceptor that will recognize an input string with up to $\delta$ edits.

Then, walk through the acceptor and our dictionary, emitting any final states we visit.
Levenshtein automata

The basic idea: construct an acceptor that will recognize an input string with up to $\delta$ edits.

We have seen something not entirely dissimilar:
Levenshtein automata

In a sense, our old friend the edit-distance transducer is a step along the path towards a Levenshtein transducer.

The difference: the edit-distance transducer will allow infinite insertions or deletions...

... and we need to limit the total number of such events.
Levenshtein automata

Character position (mantissa)
Levenshtein automata

![Levenshtein automata diagram]

**Number of edits thus far (exponent)**

Levenshtein automata
Levenshtein automata

Substitution
Levenshtein automata

![Levenshtein automata diagram]

Deletion

Levenshtein automata

Fig. 4

For this reason, a uniform notion of subsumption.

Subsumption triangles (cf. Example 4)

\( \pi \):=

\[ \begin{align*}
\Phi &= 8. \\
\Phi &= 5
\end{align*} \]

Fig. 5

Definitions

Base positions can be considered as "error-free" positions.

Accepting positions can be considered as final states of the non-deterministic Levenshtein automata described above.

The following lemma indicates the background for the notion of subsumption.

\[ \text{Example 4} \]

Let \( e < f \) and \( i \) are subsumed by \( j \).

\[ \text{Lemma 2} \]

For \( j \) and \( i \) are subsumed by \( n \).

Position

\[ \text{Insertion} \]
Levenshtein automata


foof    doof
Levenshtein automata

foof    doof

Levenshtein automata


foof    doof    dora
Levenshtein automata

![Levenshtein automata diagram](http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata)

foof  doof  døra
Levenshtein automata

Levenshtein automata


foof  doof  dora  foods  feed
Levenshtein automata

foof     doof     dora     foods     feed

This is a non-deterministic representation; actually using NFAs in practice is often tricky.

Luckily, NFAs can be determinized, which is generally how Levenshtein automata are actually used.
On its own, having a Levenshtein automaton of a query word improves even the naïve approach (pairwise comparison):

Instead of a large set of $O(nm)$ computations, we have a large set of $O(n)$ computations!
Levenshtein automata

We can do better, however.

Represent our dictionary as a trie, DAWG, etc....

... and walk through it and our determinized automaton together in tandem.

At each state we encounter, follow edges that both have in common.

Any time both are in final states, we’ve got a match!