Top-*k* String Auto-Completion with Synonyms

Pengfei Xu and Jiaheng Lu

Department of Computer Science, University of Helsinki
`first.last@helsinki.fi`

Abstract. Auto-completion is one of the most prominent features of modern information systems. The existing solutions of auto-completion provide the suggestions based on the beginning of the currently input character sequence (i.e. prefix). However, in many real applications, one entity often has synonyms or abbreviations. For example, "DBMS" is an abbreviation of "Database Management Systems". In this paper, we study a novel type of auto-completion by using synonyms and abbreviations. We propose three trie-based algorithms to solve the top-*k* auto-completion with synonyms; each one with different space and time complexity trade-offs. Experiments on large-scale datasets show that it is possible to support effective and efficient synonym-based retrieval of completions of a million strings with thousands of synonyms rules at about a microsecond per-completion, while taking small space overhead (i.e. 160-200 bytes per string). The implementation of algorithms is publicly available at http://udbms.cs.helsinki.fi/?projects/autocompletion/download.

1 Introduction

Keyword searching is a ubiquitous activity performed by millions of users daily. However, cognitively formulating and physically typing search queries is a time-consuming and error-prone process [3,6] . In response, keyword search engines have widely adopted auto-completion as a means of reducing the efforts required to submit a query. As users enter their query into the search box, auto-completion suggests possible queries the user may have in mind.

The existing solutions of auto-completion provide the suggestions based on the beginning of the currently input character sequence (i.e. prefix). Although this approach provides satisfactory auto-completion in many cases, it is far from optimal since it fails to take into account the semantic of users' input characters. There are many practical applications where syntactically different strings can represent the same real-world object [10]. For example, "Bill" is a short form of "William" and "Database Management Systems" can be abbreviated as "DBMS". These equivalence information suggests semantically similar strings that may have been missed by simple prefix based approaches. For instance, based on the DBLP dataset, when a user enters "Andy Pa" in the search box (see Fig. 1), the system should suggest "Andrew Palvo", because there is no record with the prefix "Andy Pa" and "Andy" is a nickname of "Andrew". Similarly, on an E-commerce site, a user may type part of an abbreviation of a product name because she does not know the full name stored in a database. In a gene/protein database, one of the major obstacles that hinder the effective use is term variation [13], including acronym variation (e.g. "IL-2" and "interleukin-2"), and term abbreviation (e.g. "Ah

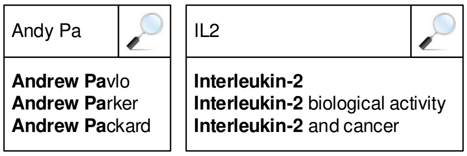


Fig. 1: Auto-completion with synonyms.

receptor" and "Ah dioxin receptor"). *Therefore, this calls for auto-completion with synonyms to improve its usability and effectiveness.* For brevity we use "*synonym*" to describe any kind of equivalent pairs which may include synonym, acronym, nickname, abbreviation, variation and other equivalent expressions.

Often, when only a few characters of the lookup string have been entered, there are too many completions for auto completion to be useful. We thus consider a top-k synonym-based auto-completion strategy that provides the suggestion for the only top-k results according to predefined ranking scores and synonym sets. Given a large set of strings, an auto-completion system needs to be speedy enough to keep up with the user's key strokes. Meanwhile, we would like to fit all strings in the limited main memory. Hence, we need a both time-efficient and space-efficient data structure that enables us to return top-k completions without checking all the data in the synonyms set and the string collection.

In this paper, we propose three data structures to support efficient top-k completion queries with synonyms for different space and time complexity trade-offs:
(i) *Twin tries (TT)*: Two tries are constructed to present strings and synonym rules respectively in order to minimize the space occupancy. Each trie is a compact data structure, where the children of each node are ordered by the highest score among their respective descendants. Applicable synonym rules are indicated by pointers between two tries. An efficient top-k algorithm is developed to search both tries to find the synonym rules.
(ii) *Expansion trie (ET)*: A fast lookup-optimized solution by integrating synonym rules with the corresponding strings. Unlike TT, ET uses a single expended trie to represent both synonym and string rules. Therefore, by efficiently traversing this trie, ET is faster than TT to provide top-k completions. Meanwhile ET often takes larger space overhead than TT, because ET needs to expand the strings with their applicable rules.
(iii) *Hybrid tries (HT)*: An optimized structure to strike a good balance between space and time cost for TT and ET. We find a balance between lookup speed and space cost by judiciously selecting part of synonym rules to expand the strings. We show that given a predefined space constraint, the optimal selection of synonym rules is NP-hard, which can be reduced to a 0/1 knapsack problem with item interactions. We provide an empirically efficient heuristic algorithm by extending the branch and bound algorithm.

Large scale evaluation of search queries on three real datasets demonstrate the effectiveness of the proposed approaches. For example, on the US postal address dataset with 1M strings, the twin tries achieve a size of 160 bytes per string, which requires an average of only 5ms to compute the top-10 completion on a simulated workload. In comparison, the expansion trie reduces the completion time to 0.1ms, but increases the size to 200 bytes per string. The hybrid tries have a balanced performance, by achieving 1-2ms per query, with the space overhead of 172 bytes per string. The implementation is available at http://udbms.cs.helsinki.fi/?projects/autocompletion/download.

2 Related work

There is a plethora of literature on query auto-completion, especially in the field of information retrieval. We report here the results closest to our work. Readers may refer to a recent survey [2] for more comprehensive review.

Auto-completion with prefix matching can be easily implemented with a trie. Therefore, it is straightforward to extend trie to support top-k prefix matching. Li et al. [9] precompute and materialize the top-k completion of each possible word prefix and store them with each internal node of a trie. This requires a predetrmined k. Surajit et al. [3] provided solutions for error-tolerating auto-completion based on edit distance constraints, which during the lookup, maintains an *error* variable while searching for all possible candidate nodes. The collection of candidate strings are fetched by gathering strings stored under all leaf nodes under all candidates nodes. Xiao et al. [14] further extended the approach, by proposing a novel neighborhood generation-based algorithm, which can achieve up to two orders of magnitude speedup over existing methods for the error-tolerant query auto-completion problem. These solutions, however, are based on string similarity, which does not expose the semantic relations between words. For example, "iPod" and "iPad" have an edit distance only 1, but they should not be considered as the same word. In contrast, the edit distance between "DASFAA" and "International Conference on Database Systems for Advanced Applications" is big, but they refer to the same conference.

In [5], Hyvonen et al. proposed *semantic-based* auto-completion, which can include synonym-based, context-based and multilingual auto-completion. Unfortunately, this paper only mentions the concept of *semantic-based* auto-completion, but no algorithms are proposed. In this paper, we make the technical contribution by proposing space and time efficient algorithms to explore the synonym relations for top-k auto-completion.

Finally, synonym pairs can be obtained in many ways, such as existing dictionaries and synonyms mining algorithms [12]. Recently, Lu et al. [10] studied how to use the synonyms to improve the effectiveness of table joins. In this paper, with different research theme, we strike to use the synonyms to provide meaningful top-k auto-completion.

3 Preliminaries and problem description

In this section, we describe some of the data structures and primitives used in this paper and define our research problem.

Dictionary and synonym rule sets. Dictionary is a scored string set $\mathbb{D}$ in forms of pairs (s, r) where $s \in \Sigma^*$ is a string drawn from an alphabet Σ and r is an integer score. A synonym rule set $\mathbb{R}$ is a set of synonym pair. Let $\mathbb{R}$ denote a collection of synonym rules, i.e., $\mathbb{R} = \{r : lhs \to rhs\}$. A rule can be applied to s if lhs is a substring of s; the result of applying the rule is the string s' obtained by replacing the substring matching lhs with rhs. Given a string p, we say a replaced string p' from p, which is obtained from some non-overlapping substrings of p by applying the rules to generate new string p'. We can apply any number of synonym rules one after another. However, a token that is generated as a result of production application cannot participate in a subsequent production.

Problem Description. Given a dictionary of strings and a collection of synonym rules, the goal is to suggest the top k strings with the highest scores with considerations of synonym pairs. Formally, we define the problem of top-*k* completion with synonyms as follows.

Problem 1 (Top-k completion with synonyms). Given a dictionary string $p \in \Sigma^*$*, an integer* k*, and a synonym set* $\mathbb{R}$*, a top-k completion query in the scored dictionary string set* $\mathbb{D}$ *returns the* k *highest-scored pairs in* $\mathbb{D}_p = \{s \in \mathbb{D} \mid p$ *is a prefix of s, or there exists a replaced string p' of p using* $\mathbb{R}$*, such that* p' *is a prefix of s.*$\}$ ☐

Example 1. See Figure 1. Given three dictionary strings $\mathbb{D}$ *including "Andrew Pavlo", "Andrew Parker" and "Andrew Packard" and one synonym rule* $\mathbb{R}$ *= {"Andy"* $\rightarrow$ *"Andrew"}. If a user enters "Andy Pa". Then all three strings are returned as top-3 completions. Note that none of results can be returned based on the traditional prefix-based auto-completion.* ☐

4 Twin tries (TT)

A trie, or radix tree or prefix tree, is a tree data structure that encodes a set of strings, represented by concatenating the characters of the edges along the path from the root node to each corresponding leaf. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. To encode the score and support top-k completions, we assign to each leaf node the score of the string it represents, while each intermediate node holds the maximum score among its descendants. We employ two tries, named *dictionary trie* ($\mathcal{T}_D$) and *rule trie* ($\mathcal{T}_R$), which hold all dictionary strings and the synonym rules, respectively. Moreover, we introduce *synonym links*, which are edges from $\mathcal{T}_R$ and pointing to corresponding nodes in $\mathcal{T}_D$. To support top-k lookups, each synonym link is assigned with an integer offset, denoted by $link.delta$, which equals to the length of $rule.lhs$ minus length of $rule.rhs$. An example of the mentioned structure can be found in Fig. 2. Algorithm 1 gives the method for building TT.

We introduce a heuristic algorithm (see Alg. 2) to find the best completion results that extends the search string. Specifically, starting from the root node, we iteratively search in the dictionary trie for any matching prefixes of search string. For the unmatched part, we look up in the rule trie for any possible synonym rules. If there are multiple link targets in this rule, we select the appropriate one by comparing the deepest locus node and the node prior to $rule.lhs$ (Line 18 in Alg. 2). To support an efficient top-k lookup, we also introduce $node.depth$, which is the number of edges from $node$ to the trie's root node.

Example 2. Consider twin tries in Fig. 2. Assume that the search string is "abmp" and $k = 1$*. The process for getting completion result with Alg. 2 can be found in the following table*[1]*. "✓" and "×" represents a string is found or not found in corresponding trie:* p_{r_a} *in dictionary trie,* p_{r_r} *in rule trie.* ☐

---

[1] In this table, we use the denotation `ab`$\underline{\texttt{c}}$ to represent a node with label "c" with parent node labeled "b", in path *root – a – b – c*.

Iter.	$\mathbf{p_{r_a}}$	$\mathbf{p_{r_r}}$	Note
1	Pop first element from queue: m = ε (root of $\mathcal{T}_D$), p_r = *abmp*		
1.1	ε ✓	*abmp* ×	ε is found in $\mathcal{T}_D$, but *abmp* is not found in $\mathcal{T}_R$.
1.2	*a* ✓	*bmp* ×	*a* is found in $\mathcal{T}_D$, but *bmp* is not found in $\mathcal{T}_R$.
1.3	*ab* ✓	*mp* ✓	*mp* is found in $\mathcal{T}_R$. The target of its links are $\underline{\texttt{c}}$ and $\texttt{ab}\underline{\texttt{c}}$. $\texttt{ab}\underline{\texttt{c}}$ is the correct link target. Push it to queue.
1.4	*abm* ×		Break loop.
2	Pop first element from queue: $m = abc$, $p_r = \emptyset$		
2.1	*abc* ✓	$\emptyset$	Node $\texttt{ab}\underline{\texttt{c}}$ is a leaf, so add it to result set. p_{r_r} is empty, so push all children of $\texttt{ab}\underline{\texttt{c}}$ to queue (but it has no child).
3	The queue is empty. Therefore the final result is "*abc*".		

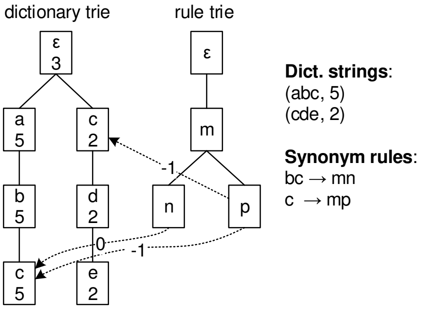


Fig. 2: TT example

Complexity analyses. The worst-case time complexity of top-k auto-completion (Alg. 2) is O($pm + p^2n^2 + klm\log(klm) + st$), where p is the length of search string, l is the maximum length of dictionary string, m and n is the maximum number of children per node in $\mathcal{T}_D$ and $\mathcal{T}_R$ respectively, s is the maximum number of links per rule and t is the maximum number of applicable rules per dictionary string. Specifically, we need to examine up to pm nodes in the dictionary trie to check whether a substring is from dictionary. We also need to lookup $(pn + \cdots + 2p + p)$ nodes in the rule trie in order to find possible synonym links. After we find one result, we need to scan upward l nodes on the way to root, which is totally O(kl) time corresponding to k completions. As the algorithm inserts all nodes in corresponding path to the priority queue, we may add up to klm nodes, contributing an additional O($klm\log(klm)$) term. Finally, O(st) time is required to determining the correct synonym link.

Algorithm 1: Generation of TT

```
Input: Set of dictionary strings ($\mathbb{D}$) and set of synonym rules ($\mathbb{R}$)
Output: Twin tries $\langle \mathcal{T}_D, \mathcal{T}_R \rangle$

for each of rules in $\mathbb{R}$, add its $rhs$ to $\mathcal{T}_R$
foreach $s \in \mathbb{D}$ do
    add $s$ to $\mathcal{T}_D$
    foreach $r \in \mathbb{R}$ do
        if $r$ can be applied onto $s$ then
            $f \leftarrow$ deepest locus node of $r$ in $\mathcal{T}_R$
            foreach $lo \in$ all locus points of $r$ on $s$ do
                $l \leftarrow$ node from $\mathcal{T}_D$, which represents $r$.lhs in decendents of $lo$
                $f$.links.add($l$, $r$.lhs.length - $r$.rhs.length)      // (target, delta)
recursively set every score of every node in $\mathcal{T}_D$ to the maximum among its descendants
return $\langle \mathcal{T}_D, \mathcal{T}_R \rangle$
```

Algorithm 2: Top-*k* completions with TT

```
Input: Dictionary trie $\mathcal{T}_D$, rule trie $\mathcal{T}_R$, search string $p$ and $k > 0$
Output: List of top-$k$ completions $C$
$Q \leftarrow$ empty priority queue; $C \leftarrow$ empty priority list; $Q$.push($\langle$root node of $\mathcal{T}_D$, 0$\rangle$)
while $Q \neq \emptyset$ do
    $\langle m, i_{p_r} \rangle \leftarrow Q$.pop()      // (current node, index of remaining $p$)
    $p_r \leftarrow p$.substring(0, $i_{p_r}$)
    for $i$ from 0 to $p_r$.length do
        ($p_{r_a}$, $p_{r_r}$) $\leftarrow p_r$.split($i$)
        $l \leftarrow$ deepest locus node of $p_{r_a}$ in descendants of node $m$
        if $l$ is not found then break the for loop
        else if $l$ is a leaf node then
            $C$.add(full string of $l$)
            if $|C| = k$ then return $C$
        if $p_{r_r}$ is empty string then
            foreach $c \in l$.children do $Q$.push($\langle c, i_{p_r} + i \rangle$)
        else
            $ns \leftarrow$ locus points of $p_{r_r}$ in $\mathcal{T}_R$
            foreach $n \in ns$ do
                foreach $lk \in n$.links do
                    $dest \leftarrow$ from $lk.target$, go up ($lk.depth + lk.delta$) levels
                    if $l$ and $dest$ is the same node then
                        $Q$.push($\langle lk$.target, $i_{p_r}$ + $i$ + $lk$.target.depth$\rangle$)

return $C$
```

5 Expansion trie (ET)

In thi section, we describe a compressed trie data structure to combine both dictionary and synonym strings into one trie, called Expansion Trie (ET).

The baseline algorithm is to generate a set of new strings by applying permutations of rules onto the dictionary strings, then add them to trie. The baseline algorithm has two problems: (i) Dictionary and synonym nodes are mixed together in the final trie, and thus it is hard to tell whether a string is from dictionary; (ii) the algorithm is extremely slow because the generation of permutations for all applicable rules in strings.

To address the above problems, we propose a new algorithm for generating the ET, which minimizes running time and memory cost by removing duplicate nodes. We treat every dictionary string as a unique path from root to leaf, while all its available synonym rules as "branches" that attached to it. At the end of each branch, a synonym link points it back to the dictionary string. Additionally, we set the score of every *synonym node* (new nodes introduced by expanding syn-

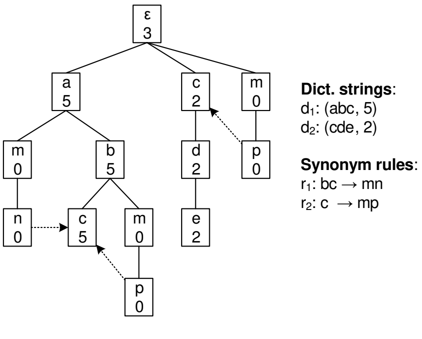


Fig. 3: ET example

Algorithm 3: Generation of ET

```
   Input: Set of dictionary strings ($\mathbb{D}$), set of synonym rules ($\mathbb{R}$)
   Output: Expansion trie $\mathcal{T}$
1  add all strings in $\mathbb{D}$ to $\mathcal{T}$
2  foreach $s \in \mathbb{D}$ do
3      foreach $r \in \mathbb{R}$ do
4          if $r$ can be applied onto $s$ then
5              foreach $f \in$ all locus points of $r$ on $s$ do
6                  $l \leftarrow$ deepest node of $r$.lhs in decendents of $f$
7                  $p \leftarrow f$.parent              // synonym nodes attach here
8                  add each char of $r$.rhs with $score = 0$ as descendants of $p$
9                  $e \leftarrow$ select the last node just added (i.e. deepest node in synonym nodes)
10                 $e$.links.add($l$)
11 recursively set every score of every node in $\mathcal{T}$ to the maximum among its descendants
12 return $\mathcal{T}$
```

Algorithm 4: Top-*k* completions with ET

```
   Input: Expansion trie $\mathcal{T}$, search string $p$ and $k > 0$
   Output: List of top-k completions $C$
1  $Q \leftarrow$ empty priority queue; $C \leftarrow$ empty priority list; $H \leftarrow$ empty hash table
2  $locus \leftarrow$ deepest locus point of $p$ in $\mathcal{T}$
3  if $locus$ is a dictionary character then $Q$.push($locus$)
4  foreach $l \leftarrow$ target node of $locus$.links do $Q$.push($l$)
5  while $Q \neq \emptyset$ do
6      $m \leftarrow Q$.pop()
7      if $m$ is a leaf node then
8          add full string of $m$ to $C$ with $m$.score
9          if $|C| = k$ then return $C$
10     if $m$ is the last node representing p, or there is more chars from p after $m$ then
11         Q.push($m$.links.target)
12     if there is more chars from p after $m$ then
13         $Q$.push(node holds next character of $p$ after $m$)
14     else push all non-synonym nodes of $m$.children to $Q$
15 return $C$
```

onym rules on dictionary trie) to 0, because we do not give such suggestion. The pseudocode of the proposed algorithm can be found in Alg. 3.

Example 3. Given dictionary strings $d_1 : (abc, 5)$ *and* $d_2 : (cde, 2)$ *and synonym rules* $r_1 : bc \rightarrow mn$ *and* $r_2 : c \rightarrow mp$*, the ET generated by Alg. 3 can be seen in Fig. 3.* □

We introduce a similar solution to perform top-k suggestions on ET as in Alg. 4. Specifically, we find the deepest node in the trie that matches the search string as much as possible (Line 2 in Alg. 4) and insert it into a priority queue. We also insert the target node of its synonym links (if any) to the queue. Then we iteratively pop the node with highest score as the current node. We add it to the result if it is a leaf node. Otherwise, if there is still any other remaining character, we find the corresponding child node and add it to the queue. When all characters are processed, we add all the children of the current node to the queue (Lines 12 to 14 in Alg. 4). This procedure loops until k completions have been found or the priority queue becomes empty.

Complexity analyses. The worst-case time complexity of top-k on ET is O($pm + klm\log(klm)$). According to the proposed algorithm, we need to examine up to pm nodes in the trie to find the locus node. After reaching one leaf node, we need to scan upward l nodes on the way to root, which is totally O(kl) time corresponding to k completions. Add up to klm nodes to the binary heap contributes an additional O($klm\log(klm)$) term. Although we use the same notation "m" here, one should notice that its value is larger compared to TT because the expansion of rules introduced more nodes, thus the maximum number of children is increased.

6 Hybrid tries (HT)

By comparing the top-k time complexity of TT and ET, it can be seen that the latter will need more time as it needs to (i) look up the rule trie iteratively for every substring of p, (ii) check all synonym links in order to find the correct one. Therefore, we propose a solution that selects some synonym rules and expands them while leaving the remaining ones in rule trie, so that fewer synonym links need to be checked, which leads to a smaller running time. Note that *the more rules we expand, the more space it takes*. Therefore the problem can be defined as follows:

Problem 2 (Optimal construction of HT). Let S_{TT} and S_{ET} be the space cost of TT and ET, given a space threshold $S \in [S_{TT}, S_{ET}]$, $\mathbb{D}$ and $\mathbb{R}$, our task is to build two tries $\langle \mathcal{T}_D, \mathcal{T}_R \rangle$ to minimize the top-k lookup time while satisfying the space constraint S.

With endeavors to make the lookup search based on HT more efficient, our approach is to solve Problem 2 with a combinatorial optimization algorithm based on the frequency of rules in applicable strings. Therefore, the policy of selecting rules for expansion turns into *maximizing the total number of applicable rules on dictionary strings within space constraints*.

Let r_i be the *item* (synonym rule) at index i in $\mathbb{R}$, $\{v_1, v_2, ..., v_{|\mathbb{R}|}\}$ be the frequency (time-of-use) of items, $\{w_1, w_2, ..., v_{|\mathbb{R}|}\}$ be its weight, i.e. space cost when expanding it to the dictionary trie, $\{x_1, x_2, ..., x_{|\mathbb{R}|}\}$ be a set of integers either 0 or 1, it can be seen that the problem is similar with a 0/1 knapsack problem, which is known NP-hard:

$$\text{maximize} \quad \sum_{i=1}^{|\mathbb{R}|} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{|\mathbb{R}|} w_i x_i \leq S$$

However, our problem is not such straightforward because the space cost of a synonym rule may be smaller depends on the presence of other rules in the trie. Consider dictionary string $abcde$ and two rules $r_1 : abc \rightarrow mn$ and $r_2 : abc \rightarrow mnp$. Expanding r_1 adds two synonym nodes $\mathtt{\underline{m}}$ and $\mathtt{m\underline{n}}$. Then when expanding r_2, it uses existing nodes $\mathtt{\underline{m}}$ and $\mathtt{m\underline{n}}$ which are generated by r_1 before. Thus only one new node $\mathtt{mn\underline{p}}$ is created. By considering such interactions, we are able to formalize Problem 2 more precisely as follows:

Problem 3 (0/1 Knapsack Problem with Item Interactions).

$$\text{maximize} \sum_{i=1}^{|\mathbb{R}|} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{|\mathbb{R}|} f_i(x_i, x_j | j \in P_i) \leq S$$

$f_i(\cdot)$ is the weight function that returns the weight of item r_i with knowledges of x_i, current items in knapsack (their indexes are stored in C), and P_i as indexes of all items which have interactions with r_i. Specifically, the weight function can have three types of return values: (i) $f_i(\cdot) = 0$ when $x_i = 0$, i.e. item r_i is not selected. (ii) $f_i(\cdot) = w_i$ when $x_i = 1$ and $\nexists x_j = 1 | j \in (P_i \cap C)$. (iii) $f_i(\cdot) \in (0, w_i)$, otherwise.

It is possible to adapt the dynamic programming (DP) method to Problem 3, by sorting the items so that all items which r_i depends on are located before r_i. This ensures all interacted items are processed before r_i itself. However, in our problem the cyclical cannot be ignored [1]: we can say that the weight of r_1 depends on the presence or absence of r_2, but it is also true to say r_2 depends on r_1, since r_2 can also provide the two synonym nodes which reused by r_1. Due to the hardness of the problem, some approximate methods are proposed, by grouping interacted items as a single knapsack item [8] or cutting weak interactions [11]. However, all such solutions are not able to give a bound of the estimation error. In this paper, we present a new solution following a branch and bound (B&B) fashion by tightening the upper- and lower-bound with considerations of item interactions, which gives an exact solution subject to total value.

We now introduce three terms used in our algorithm. All items can be classified into one of three categories at any specific stage of B&B algorithm [7]: (i) *Included*: the item is explicitly included in the solution. According to our definition, item r_i is an included item when $x_i = 1$. (ii) *Excluded*: the item is explicitly excluded in the solution, i.e. $x_i = 0$. (iii) *Unassigned*: the item is not processed yet. At any given stage, this type of items should only contains further items that has not been tested in any branch.

Tight upper-bounds For knapsack with independent items, the method for obtaining an upper-bound is based on the solution of *fractional knapsack problem*, where a part of an item can be take into the knapsack when the space does not fit it as a whole. A greedy algorithm by Dantzig et al. [4] can be employed to obtain an optimal result. In our case, we sort items by assuming all interactions already existed. That is, for item r_i, we assume every item $\forall j \in P_i, x_j = 1$. We use $w_{min,i}$ to indicate this minimum weight. This can guarantee that the greedy algorithm returns a solution which is the largest one among all feasible upper-bounds.

Tight lower-bounds A classic method to obtain the lower-bound is to look forward down the unassigned items in current solution, and greedy take (in order) items into knapsack until the weight budget left cannot fit the next item. We extend this method by assuming every interacted item r_j is either excluded or unassigned, i.e. $\forall j \in P_i, x_j = 0$.

Measuring exact weight We add the exact space consumption for expanding r_i to knapsack in each branch operation. One straightforward solution can be "scan all items to accumulate any possible savings". Unfortunately, this method ignores that fact that most items are not interacted and will be very slow when $|\mathbb{R}|$ is large because each scan requires O($|\mathbb{R}|$) time. As in our solution, we perform a *partition* prior to B&B by grouping all items to several parts: r_i has interactions with all other items in the same part, but not with items in other parts. As the result, saving can only be obtained when r_i is included together with items from the same part, otherwise it has its original weight.

Algorithm 5: Construction of HT

```
Input: Set of dictionary strings ($\mathbb{D}$), set of synonym rules ($\mathbb{R}$) and space threshold ($S$)
Output: Hybrid tries $\langle \mathcal{T}_D, \mathcal{T}_R \rangle$
$P \leftarrow$ partition rules in $\mathbb{R}$
sort $\mathbb{R}$ by items' minimum weight (i.e. assume all interactions exist)
$\langle \mathbb{R}_{in}, \mathbb{R}_{ex} \rangle \leftarrow$ solve knapsack problem with branch and bound, with bound functions
  $upper\_bound(r_i)$ and $lower\_bound(r_i)$, $r_i$ is the current item in the decision tree. in
  each branch, the exact weight of $r_i$ is obtained by exact_weight ($r_i$, $P_{r_i}$, $X_{inc}$), where
  $P_{r_i}$ is the part $r_i$ belongs to, $X_{inc}$ is the set of included items at current step
$\mathcal{T}_D \leftarrow$ build dictionary trie with $\mathbb{D}$ and expand rules in $\mathbb{R}_{in}$ following Alg. 3
$\langle \mathcal{T}_D, \mathcal{T}_R \rangle \leftarrow$ build rules trie with $\mathbb{D}$ and $\mathbb{R}_{ex}$ following Alg. 1, while let $\mathcal{T}_D$ be the
  ready-to-use dictionary trie
return $\langle \mathcal{T}_D, \mathcal{T}_R \rangle$
Function upper_bound ($r_i$)
    $ub_i \leftarrow r_i$.weight                              // take current weight
    while $ub_i < S$ do
        take $r_i$, add its minimum weight to $ub_i$; $i \leftarrow i + 1$
    $ub_i \leftarrow ub_i + \frac{v_i}{w_{min,i}} \times (S - ub_i)$ // take a fraction of next item
        using its minimum weight
    return $ub_i$
Function lower_bound ($r_i$)
    $lb_i \leftarrow r_i$.weight
    while $lb_i < S$ do
        take $r_i$, add its original weight to $lb_i$; $i \leftarrow i + 1$
    return $lb_i$
Function exact_weight ($r_i$, $P_{r_i}$, $X_{inc}$)
    $w_{real} \leftarrow w_i$
    foreach $r | r \neq r_i, r \in P_{r_i}$ do
        if $\exists r \in X_{inc}$ then $w_{real} \leftarrow \min(w_{real}, f_i(x_i, r))$
    return $w_{real}$
```

The partition allows us to use a heuristic in each branch operation by scanning though only items in the same part with r_i, instead of all items.

Construction of HT and top-*k* completions Our algorithm for solving Problem 2 is given in Alg. 5. Specifically, we highlight our three extensions to B&B algorithm as separate functions.

Example 4. Given dictionary strings $d_1 : (abc, 5)$ and $d_2 : (cde, 2)$ and synonym rules $r_1 : bc \rightarrow mn$ and $r_2 : c \rightarrow mp$, the HT generated by expanding r_1 according to Alg. 5 is illustrated in Fig. 4.

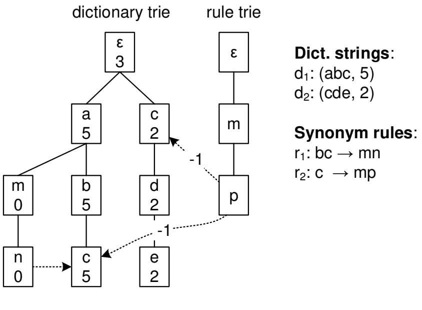


Fig. 4: HT example

We can preform top-*k* completions queries on HT by extending Alg. 2: every time when checking possible synonym rules in p_r (before Line 12), we push the target of *l.links* with `foreach t ← target of l.links do Q.push(t).`

Because the top-*k* completions algorithm on HT is similar with TT, their worst-time complexity is also the same. However, the value of s (i.e. maximum number of links per synonym rule) is smaller since we reduced the number of synonym links per rule by moving some rules to the dictionary tire.

7 Experimental analysis

To evaluate the effectiveness of the proposed top-*k* completion techniques, Twin Tries (**TT**), Expansion Trie (**ET**) and Hybrid Tries (**HT**), we compare their effectiveness on the following datasets from different application scenarios on a Intel i7-4770 3.4GHz processor with 8GB of RAM, complied with OpenJDK 1.8.0 on Ubuntu 14.04 LTS.

7.1 Datasets

We use three datasets: conference publications and book titles (**DBLP**), US addresses (**USPS**), and gene/protein data (**SPROT**). These datasets differ from each other in terms of rule-number, rule-complexity, data-size and string-length. Our goal in choosing these diverse sources is to understand the usefulness of algorithms in different real world environments.

DBLP: We collected 24,810 conference publications and book titles from DBLP website (http://dblp.uni-trier.de/). We obtained 214 synonym pairs between the common words and their abbreviations used in computer science field listed on IEEE website.

USPS: We downloaded common person names, street names, city names and states from the United States Postal Service website (http://www.usps.com). We then generated 1,000,000 records as dictionary strings, each of which contains a person name, a street name, a city name and a state. We also gathered extensive information about the common nicknames and format of addresses, from which we obtained 341 synonym pairs. The synonym pairs covers a wide range of alternate representations of common strings, e.g. `Texas` → `TX`.

SPROT: We obtained 1,000,000 gene/protein records from the UniProt website (http://www.uniprot.org/). Each record contains an entry name, a protein name, a gene name and its organism. In this dataset, each protein name has $5 \sim 22$ synonyms. We generated 1,000 synonym rules describing relations between different names.

Table 1 gives the characteristics of the three datasets. The scores of each string are randomly generated in this experiment.

7.2 Data structure construction

Space We evaluate the compactness of the generated data structures by reporting in Tab. 2 the average number of bytes per string (including score and relations e.g. *node.parent*). For comparison, we also report the size of the data structure generated by the baseline method (**BL**) described in the expansion trie (see Sec. 5). Across the

Table 1: Characteristics of datasets.

Name of Dataset	# of Strings	String Len (avg/max)	# of Synonym Rules	Rules Per String (avg/max)
DBLP	24,810	60 / 295	368	2.51 / 11
USPS	1,000,000	25 / 43	341	2.15 / 12
SPROT	1,000,000	20 / 28	1,000	2.11 / 12

datasets, the baseline method produce the largest trie structure, about 4KB per dictionary string for DBLP dataset. For larger dataset like USPS and SPROT, it crashes because of exponentially increasing number of new strings. The ET structure takes the second largest space consumption, while TT consistently takes the smallest size, about 58% smaller than ET on SPROT dataset. Finally, the HT structure (we set the space threshold to $0.5 \times (S_{ET} - S_{TT})$) takes a larger size than TT but smaller than ET.

Table 2: Data structure sizes in bytes per string.

Name of Dataset	BL	TT	ET	HT
DBLP	4,250.98	528.71	638.76	578.12
USPS	Failed	160.49	200.03	172.64
SPROT	Failed	128.82	217.55	161.25

To better understand how the space is used, we present in Fig. 5 the storage breakdown of each of the techniques on SPROT dataset. We break the total space down to (i) Space taken by dictionary nodes, including labels, scores and relations like $node.parent$ and $node.children$, (ii) Expanded synonym nodes: size of synonym nodes in the dictionary trie and (iii) Unexpanded synonym nodes: size of synonym nodes in rule trie $\mathcal{T}_R$. For ET, the number of synonym nodes in the trie is about 15 times more than in rule trie (TT) due to the numerous different locus points. The latter eliminates multiple copies of nodes, but will incur some sacrifice in top-k speed. For HT, the most frequent rules are expanded like ET, while half size of less-frequent rules are left in the rule trie. This results a moderate space consumption between TT and ET.

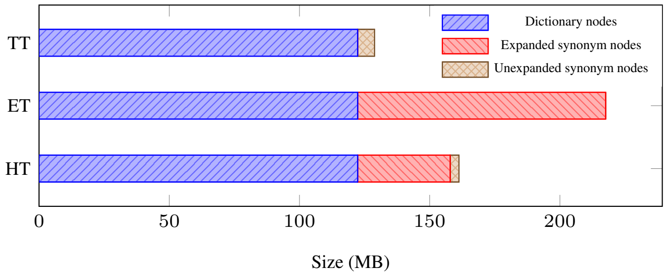
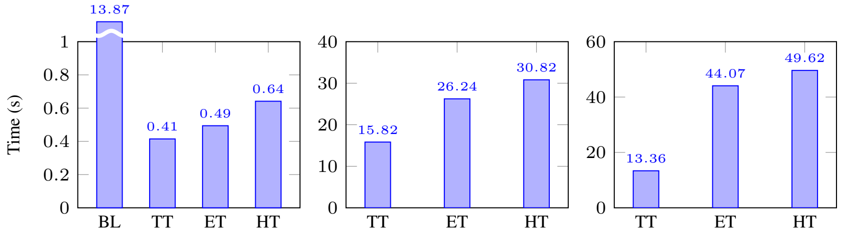


Fig. 5: Data structure size breakdown on SPROT dataset

Time In addition to the space cost, we also measure their running time on three dataset and report them in Fig. 6. For small dataset like DBLP, all four methods finish within a reasonable time, however, the baseline method is nearly 30 times slower than the other three. It also failed to finish within 300 seconds on large dataset USPS and SPROT. For the other three methods, TT is always the fastest on all datasets, because it does not need to create synonym nodes for each application, but use the existing ones and add a new synonym link. The HT runs the slowest due to the additional computation in the B&B method.

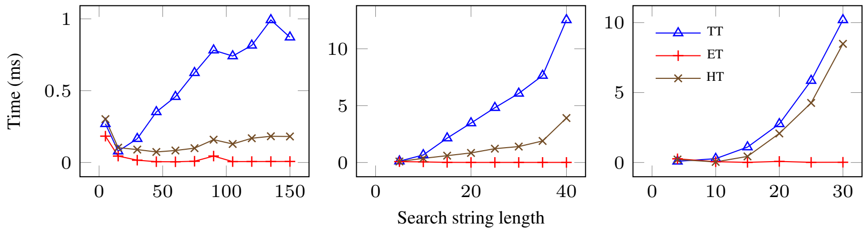


(a) DBLP (b) USPS (c) SPROT

Fig. 6: Construction time

7.3 Top-k efficiency

This set of experiments studies the overall efficiency of our auto-completion. We generate test queries by randomly applying synonym rules onto the dictionary strings, then we randomly pick a substring of each new string, formed 50,000 query strings for each dataset. We ran every query string based on the TT, ET and HT structures and plotted the running time in Fig. 7. We observed that for shorter queries (length 2 to 10), all three algorithms runs very fast, less than 0.5ms for small dataset and 1ms for large ones. However, the running time of TT and HT grows as the length of query becomes longer. The primary reason for this is that they need to lookup every substring of query in the rule trie, which consumes more time (Lines 4 to 7 in Alg. 2). Determining the correct link further slows down the speed. Besides, as HT expanded some synonym rules, its speed is faster for the reason that less synonym links being checked. In contrast, ET runs the fastest in all experiments, whose running time is not affected by the length of search strings.



(a) DBLP (b) USPS (c) SPROT

Fig. 7: Top-10 auto-completion lookup time

We observe that running time of HT is more like TT especially on SPROT dataset. As the space threshold is the key parameter to control the construction of HT, we preform one more set of experiments to deeply study the effect of this parameter on the lookup running time. We define a ratio $\alpha \in [0, 1]$ where $\alpha = \frac{S}{S_{ET} - S_{TT}}$. We select several values for α, build HT and then perform top-10 lookup. The speed of top-10 operations corresponding to different αs is illustrated in Fig. 8. The result shows that when α becomes larger, i.e. larger space threshold for HT, the top-k lookup becomes faster. When $\alpha = 0$ and $\alpha = 1$, the time is exactly the same with TT and ET, respectively. This experiment shows that if we select a space threshold between 75% and 100% of $S_{ET} - S_{TT}$, we can expect to have more than 50% performance boost compared with TT while performing lookup.

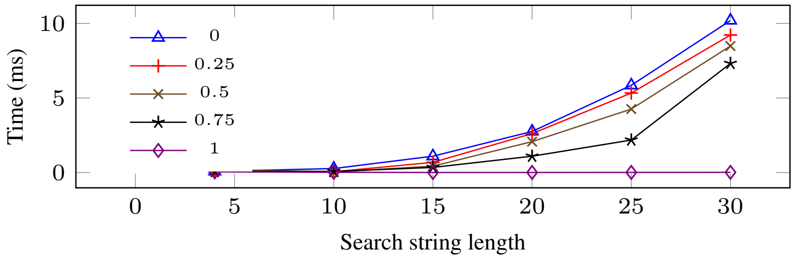
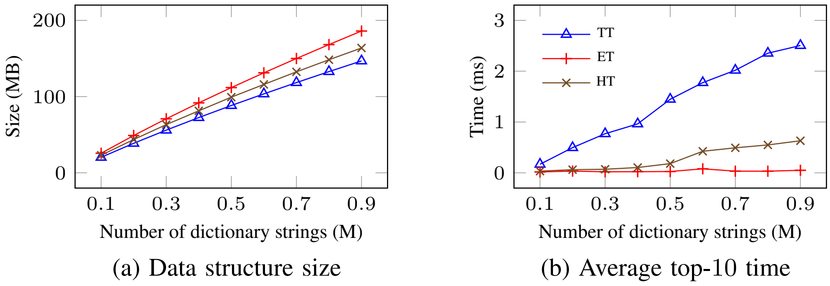


Fig. 8: Top-10 auto-completion lookup time of HT on SPROT dataset, in respect of different space ratios α



(a) Data structure size

(b) Average top-10 time

Fig. 9: Data structure size and average top-10 time related to number of dictionary strings on USPS dataset

7.4 Scalability

To assess the scalability of the data structures, we compare the structure size and top-10 speed on different subsets of the USPS dataset. We generate these subsets by taking the top-N items in decreasing score order. Figure 9a shows that the sizes of all three structures increase linearly, where TT and TT are the smallest and largest, respectively. In Fig. 9b, the average time per completion for ET does not increase as the dataset grows, while TT and HT become slower as number of dictionary string becomes larger. This is because the increasing number of strings brings more synonym links need to be checked. However, compared with TT, who has a sharp increasing trend (about 3ms per million strings), the time of HT grows slowly, only from 0.18 to 0.6ms while data grows from 0.5M to 0.9M.

8 Conclusion and future work

In this paper, we have presented three data structures, i.e. TT, ET and HT, to address the problem of top-*k* completion with synonyms, each with different space and time complexity trade-offs. Experiments on large-scale datasets show that our algorithms can support synonym-based retrieval of completions of strings at about a microsecond per-completion for more than 1 million dictionary strings and thousands of synonym rule while taking small memory space. As our future work, it would be interesting to work on the problem called "*synonym ambiguity*". For instance, "DB" can be either "Database" or "Development Bank" depending on different contexts. We will explore the context of words to select appropriate synonym rules for auto-completion.

References

1. Burg, J.J., Ainsworth, J.D., Casto, B., Lang, S.: Experiments with the "oregon trail knapsack problem". Electronic Notes in Discrete Mathematics 1, 26–35 (1999)
2. Cai, F., de Rijke, M.: A survey of query auto completion in information retrieval. Foundations and Trends in Information Retrieval 10(4), 273–363 (2016)
3. Chaudhuri, S., Kaushik, R.: Extending autocompletion to tolerate errors. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. pp. 707–718. SIGMOD '09, ACM, New York, NY, USA (2009)
4. Dantzig, G.B.: Discrete-variable extremum problems. Operations research 5(2), 266–277 (1957)
5. Hyvönen, E., Mäkelä, E.: Semantic autocompletion. In: The Semantic Web–ASWC 2006, pp. 739–751. Springer (2006)
6. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: Proceedings of the 18th International Conference on World Wide Web. pp. 371–380. WWW '09, ACM, New York, NY, USA (2009)
7. Kolesar, P.J.: A branch and bound algorithm for the knapsack problem. Management science 13(9), 723–735 (1967)
8. LeFevre, J., Sankaranarayanan, J., Hacigümüs, H., Tatemura, J., Polyzotis, N., Carey, M.J.: MISO: souping up big data query processing with a multistore system. In: SIGMOD Conference. pp. 1591–1602. ACM (2014)
9. Li, G., Ji, S., Li, C., Feng, J.: Efficient type-ahead search on relational data: a TASTIER approach. In: ACM SIGMOD. pp. 695–706 (2009)
10. Lu, J., Lin, C., Wang, W., Li, C., Xiao, X.: Boosting the quality of approximate string matching by synonyms. ACM Trans. Database Syst. 40(3), 15 (2015)
11. Schnaitter, K., Polyzotis, N., Getoor, L.: Index interactions in physical design tuning: Modeling, analysis, and applications. PVLDB 2(1), 1234–1245 (2009)
12. Singh, R., Gulwani, S.: Learning semantic string transformations from examples. PVLDB 5(8), 740–751 (2012)
13. Tsuruoka, Y., McNaught, J., Tsujii, J., Ananiadou, S.: Learning string similarity measures for gene/protein name dictionary look-up using logistic regression. Bioinformatics 23(20), 2768–2774 (2007)
14. Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., Sadakane, K.: Efficient error-tolerant query autocompletion. Proc. VLDB Endow. 6(6), 373–384 (Apr 2013)