Extending Dictionary-based Entity Extraction to Tolerate Errors

Guoliang Li† Dong Deng‡ Jianhua Feng†
†Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China.
‡School of Software, Beihang University, Beijing 100191, China.
liguoliang@tsinghua.edu.cn, buaasoftdavid@gmail.com, fengjh@tsinghua.edu.cn

ABSTRACT

Entity extraction (also known as entity recognition) extracts entities (e.g., person names, locations, companies) from text. Approximate (dictionary-based) entity extraction is a recent trend to improve extraction quality, which extracts substrings in text that approximately match predefined entities in a given dictionary. In this paper, we study the problem of approximate entity extraction with edit-distance constraints. A straightforward method first extracts all substrings from the text and then for each substring identifies its similar entities from the dictionary using existing methods for approximate string search. However many substrings of the text have overlaps, and we have an opportunity to utilize the shared computation across the overlaps to avoid unnecessary duplicate computations. To this end, we propose a heap-based framework to efficiently extract entities. We have implemented our techniques, and the experimental results show that our method achieves high performance and outperforms existing studies significantly.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation, Performance

Keywords

Approximate Entity Extraction, Edit Distance, Heap

1. INTRODUCTION

Dictionary-based entity extraction extracts all the substrings from given text that match predefined entities in a dictionary. For example, consider text "*Maros Hadjileftheriou, Nick Koudas, Divesh Srivastava: Incremental maintenance of length normalized indexes for approximate string matching*", and a dictionary with two entities "`Marios Hadjieleftheriou`" and "`Nick Koudas`". Dictionary-based entity extraction extracts the predefined entity "`Nick Koudas`" from the text. It has many real applications in the fields of information retrieval, molecular biology, bioinformatics, and natural language processing. However, usually text may contain typographical or orthographical errors [5], especially the text crawled from the Web. For example, the substring "`Maros Hadjileftheriou`" in the above text has typographical errors. However the traditional (exact) entity extraction cannot extract this substring from the text, since the substring does not exactly match the predefined entity "`Marios Hadjieleftheriou`". Approximate (dictionary-based) entity extraction is a recent trend to address this problem, which can extract substrings from the text that *approximately* match the predefined entities.

Many similarity functions have been proposed to quantify the similarity between two strings, such as jaccard similarity, cosine similarity, and edit distance. In this paper, we study the problem of approximate entity extraction with edit-distance constraints, which, given a dictionary of entities, text, and an edit-distance threshold, finds all the substrings from the text that have edit distances to an entity in the dictionary no larger than the given threshold. For instance, in the above example, approximate entity extraction can extract the substring "`Maros Hadjileftheriou`" which is similar to a predefined entity "`Marios Hadjieleftheriou`".

A straightforward method to this problem first extracts all the substrings from the text, and then for each substring identifies its similar entities in the dictionary using existing methods for approximate string search [4]. As many such substrings have overlaps, this method involves unnecessary duplicate computations across the overlaps. For example, consider the above text, we first generate its substrings, such as "*Maros Hadjileftheriou*", "*aros Hadjileftheriou nick*", "*ros Hadjileftheriou nick koudas*", then we find their similar entities from the predefined dictionary. We observe that these substrings have many overlaps, and we have an opportunity to utilize the shared computation across overlaps to eliminate the unnecessary duplicate computations. Although there have been recent studies on approximate entity extraction [5, 2, 1], they do not focus on using the shared computation across overlaps to improve performance. To address this problem, in this paper we propose heap-based methods which can fully utilize the shared computation.

2. PRELIMINARIES

In this section, we first formalize the problem of approximate entity extraction with edit-distance constraints. Then we introduce two existing methods to address this problem.

2.1 Problem Formulation

Edit Distance: In this paper, we use edit distance to quantify the similarity between two strings. Formally, the edit distance between two strings r and s, denoted as $\textsc{ed}(r, s)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform r to s. For example, $\textsc{ed}(\texttt{marios}, \texttt{maras}) = 2$. In this paper two strings are *similar* if their edit distance is no larger than a given edit-distance threshold τ. Based on this notation, we formalize the problem of approximate entity extraction with edit-distance constraints as follows.

DEFINITION 1 (APPROXIMATE ENTITY EXTRACTION). *Given a dictionary of entities $E = \{e_1, e_2, \ldots, e_n\}$, text S, and an edit-distance threshold τ, approximate entity extraction finds all pairs $\langle s, e_i\rangle$ such that $\textsc{ed}(s, e_i) \le \tau$, where s is a substring of S and $e_i \in E$.*

For instance, consider text S and dictionary E in Table 1. Suppose the edit-distance threshold $\tau = 2$. ⟨"`maros hadji`", "`marios hadji`"⟩, and ⟨"`nic kouds`", "`nick koudas`"⟩, ⟨"`dievesh criva`", "`divesh sriva`"⟩ are three example results. Especially, although the substring of the text "*sigmodmaros hadji*" misses a space between "`sigmod`" and "`maros`" (a typographical error), our method still can extract "`maros hadji`".

Table 1: A dictionary of entities and text.

(a) Dictionary E

ID	Entities	Length
1	nick koudas	11
2	marios hadji	12
3	divesh sriva	12
4	dievemaros	10
5	rivakoudieva	12

(b) Text S

Text
sigmodmaros hadjileftheriou, nic kouds, dievesh crivastava: incremental maintenance of length normalized indexes for approximate string matching

2.2 Two Existing Methods

***q*-gram:** A q-gram of a string s is a substring of s with length q. The q-gram set of s, denoted as $G(s)$, is the set of all of s's q-grams. For example, the 3-gram set of "`nick_koudas`" is {`nic`, `ick`, `ck_`, `k_k`, `_ko`, `kou`, `oud`, `uda`, `das`}. Two strings r and s are similar only if they share enough common q-grams [3]. Formally if $\textsc{ed}(r, s) \le \tau$, then

$$|G(r) \cap G(s)| \ge \max(|r|, |s|) - q + 1 - \tau * q$$

must hold, where $|G(r) \cap G(s)|$ is the size of $G(r) \cap G(s)$ and $|r|$ is the length of string r. This is called count filtering [3]. Based on this concept, we discuss two existing methods.

Approximate String Search based Method: Based on count filtering, q-gram-based methods [4] are proposed to address the approximate string search problem, which, given a set of strings, a query string, and an edit-distance threshold, finds all similar strings of the query string from the set. Existing methods usually employ a filter-and-refine framework and we also use this framework to address our problem. Firstly we construct q-gram-based index structures for all entities. For each q-gram, we use an inverted list to maintain those entities that contain the q-gram. For example, consider the entities in Table 1(a), we can build its q-gram index structures as shown in Table 2. For instance, the inverted list of gram "`kou`" is {1,5}. This denotes that entity 1 and entity 5 contain the gram. Secondly, given text S, we extract all of its substrings from S. For each substring s, we

Table 2: q-gram indexes of entities in Table 1.

q-grams	Inverted Lists	q-grams	Inverted Lists
nic	1	ive	3
ick	1	ves	3
ck_	1	esh	3
k_k	1	sh_	3
_ko	1	h_s	3
kou	1,5	_sr	3
oud	1,5	sri	3
uda	1	riv	3,5
das	1	iva	3,5
mar	2,4	die	4,5
ari	2	iev	4,5
rio	2	eve	4
ios	2	vem	4
os_	2	ema	4
s_h	2	aro	4
_ha	2	ros	4
had	2	vak	5
adj	2	ako	5
dji	2	udi	5
div	3	eva	5

generate s's q-grams, and then find the entities that contain at least δ q-grams in $G(s)$ using the q-gram indexes, where $\delta = |s| - q + 1 - \tau * q$. That is we find the entities that have at least δ occurrences in the inverted lists of q-grams of s, and all such entities are taken as candidates. Finally we verify the candidates and get the final results. For instance, consider a substring s = "`dievesh`", the inverted lists of its q-grams is {4, 5}, {4, 5}, {4}, {3}, {3}. If $\tau = 1$ and $q = 3$, $\delta = |s| - 3 + 1 - 1 * 3 = 2$. As entities 3, 4, 5 have at leat two occurrences, they are candidates.

Similarity Join based Method: We can also model the substrings of the text as a set, and then we can join the set of entities and the set of substrings to generate the similar pairs using existing similarity-join based methods. Traditional methods [6] usually employ a prefix-filter-based framework and we can also use this framework to address our problem. We first sort the q-grams of each entity and each substring, for example based on IDF or in dictionary order. Then we keep $\tau * q + 1$ q-grams for each string and each entity (Xiao et al. proposed to reduce the prefix length in [6]). Consider a string s, let $G^p(s)$ denote the set of s's first $\tau * q + 1$ q-grams. Given an entity e and a substring s, they are similar only if $G^p(e) \cap G^p(s) \neq \phi$. Based on this feature, we can use the prefix set to find similar pairs.

Obviously many substrings share common q-grams, and the two methods cannot utilize the shared computation across the common grams. To address this problem, we propose efficient algorithms to improve performance.

3. A HEAP-BASED FRAMEWORK

In this section, we propose a heap-based framework to address the problem of approximate entity extraction.

3.1 A Filter-and-Refine Framework

We have an observation that some substrings of S will not produce any results, and we define *valid substrings* that are potentially similar to some entities.

Valid Substring: Let L_{min} and L_{max} respectively denote the minimal entity length and the maximal entity length in the dictionary. Obviously all the substrings of text S with length smaller than $L_{min} - \tau$ or larger than $L_{max} + \tau$ can be pruned based on length filtering. We call the substrings of text S with length between $L_{min} - \tau$ and $L_{min} + \tau$ *valid substrings*, which may have similar entities in the dictionary. For instance, consider the dictionary and text in Table 2. $L_{min} = 10$ and $L_{max} = 12$. Suppose $\tau = 1$. The substrings of the text with length between $L_{min} = 10 - 1$ and $L_{max} = 12 + 1$ could have similar entities in the dictionary, and all other substrings will not produce any results. For instance, the substring "`maros hadjileftheriou`" cannot be similar to any entity, and we can prune it.

A Filter-and-Refine Framework: We employ a filter-and-refine framework to check whether each valid substring has similar entities in the dictionary. In the filter step, we generate the candidate pairs; and in the refine step, we verify the candidates to get the final results, by computing their real edit distance using dynamic programming. In this paper, we focus on the filter step. We employ the gram-based method to address this problem. Given a valid substring s and an entity e_i, if e_i's occurrence number in the inverted lists of q-grams in $G(s)$ is no smaller than $\delta = |s| - q + 1 - \tau * q$, $\langle s, e_i \rangle$ is a candidate as discussed in Section 2.

3.2 Multi-Heap based Method

In this section, we propose a multi-heap based method. Firstly, we construct q-gram-based index structures for all entities. Secondly for each substring of the text, we generate its q-grams, construct a heap on top of the inverted lists of its q-grams, and adjust the heap to find similar entities for the substring. Formally, let $S[i, l]$ denote a substring of S with length l, starting with the i-th character (including the i-th character). Obviously $S[i, l]$ is a valid substring if $L_{min} - \tau \le l \le L_{max} + \tau$. For each valid substring $S[i, l]$, we first generate its q-gram set $G(S[i, l])$ and get inverted lists of every gram in $G(S[i, l])$. Then we construct a min-heap $H(S[i, l])$ using the first element of each inverted list. Obviously the top element on the heap is the minimal entity among all entities in the inverted lists. Next, we adjust the heap, get the next top element, and count the occurrence numbers of each entity. Finally, we output the entities that have at least $\delta = \max(L_{min} - \tau, l) - q + 1 - \tau * q$ occurrences.

For example, consider a valid substring "`dievesh criva`". Suppose $\tau = 2$. We have $\delta = \max(10-2, 13) - 3 + 1 - 2*3 = 5$. We first generate its gram set {`die`, `iev`, `eve`, `ves`, `esh`, `sh_`, `h_c`, `_cr`, `cri`, `riv`, `iva`} and get the inverted lists of the q-grams as shown in Figure 1. Then, we construct a heap on top of the first elements of each inverted list. Next, we adjust the heap and get the entities {3, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5}, in ascending order. We count the occurrence numbers of each entry and report the entities that have at least $\delta = 5$ occurrences. Here we get a candidate (entity 3). Finally, we verify the candidates and get the final result.

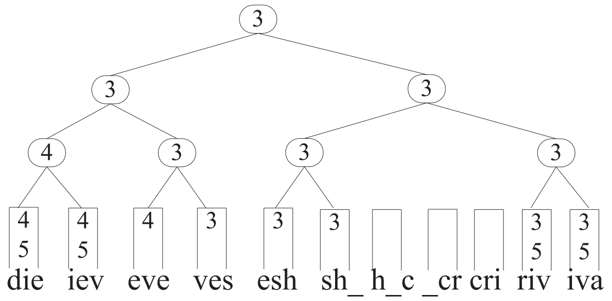


Figure 1: A heap structure for the valid substring "`dievesh criva`".

3.3 Single-Heap based Method

In this section, we propose a new heap based method, which only needs to construct a single heap on top of inverted lists of q-grams in $G(S)$, denoted as $H(S)$. For each valid substring, we use a occurrence pair $\langle e_v, o_v \rangle$ to maintain the entity (e_v) and its occurrence numbers in the substring (o_v, initialized as 0). We use the heap $H(S)$ to count the occurrence numbers of each entity in each valid substring. Firstly consider the top element $\langle e_h, g_i \rangle$ on heap $H(S)$, where e_h is an entity from the inverted list of gram g_i. Next we discuss how to find valid substrings that contain the gram and then increase the occurrence number of e_h in these valid substrings. Without loss of generality, we first consider the valid substrings with length l. Obviously the valid substring $S[i, l]$ contains the gram, and all the valid substrings $S[j > i, l]$ cannot contain gram g_i. In addition, note that the number of q-grams of a valid substring with length l is $l - q + 1$, thus the valid substring taking g_i as the last gram is $S[i - (l - q + 1) + 1, l]$ as shown in Figure 2. Accordingly, the valid substrings with length l that contain gram g_i are $S[i - (l - q + 1) + 1, l], S[i - (l - q + 1) + 2, l], \ldots, S[i, l]$.

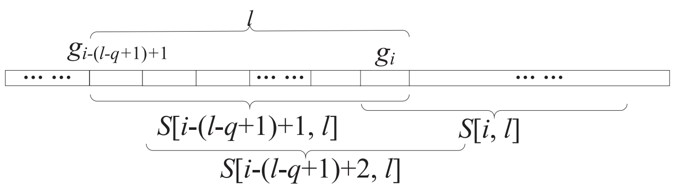


Figure 2: Valid substrings with length l that contain gram g_i.

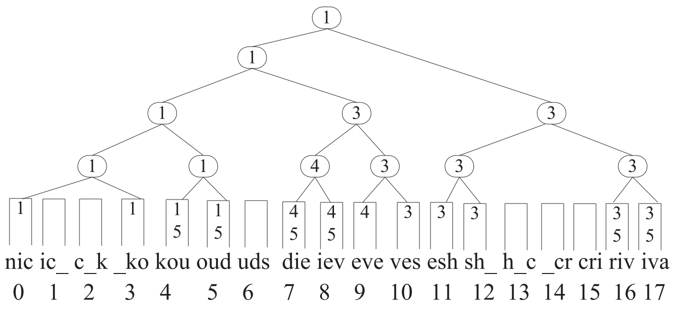
For any length $l (L_{min} - \tau \le l \le L_{max} + \tau)$, we update the occurrence numbers of e_h in the corresponding valid substrings as follows. For each valid substring s that contains gram g_i, if its kept entity e_v is the same as e_h, we increase its occurrence number o_v by one, and if the occurrence number (o_v) is larger than or equal to the threshold $\max(|s|, L_{min} - \tau) - q + 1 - \tau * q$, we return it as a candidate; otherwise, we set the current entity $e_v = e_h$ and set its occurrence number $o_v = 1$. Note that we only need to use a tuple for each valid substring to maintain the information.

Next, we delete the top element $\langle e_h, g_i \rangle$ from heap $H(S)$, adjust the heap to get the next top element $\langle e'_h, g'_i \rangle$, and update the occurrence numbers of e'_h in the corresponding valid substrings as discussed above. Interactively, we can get all the similar substrings. We give a running example to walk through the single-heap-based method.

For example, in our running example, consider text "`nic koouds dievesh criva`", we construct a single heap on top of the text as shown in Figure 3. We have $L_{min} = 10$ and $L_{max} = 12$. Suppose $\tau = 2$. For the first entity 1 selected from g_0, we only need to increase its occurrence number in valid substrings $S[0, l]$ for $L_{min} - \tau \le l \le L_{max} + \tau$, i.e., $S[0, 8], S[0, 9], \ldots, S[0, 14]$. For the next entity 1 selected from g_3, we need to increase its occurrence number in valid substrings $S[0, l], S[1, l], S[2, l], S[3, l]$ for $L_{min} - \tau \le l \le L_{max} + \tau$. Similarly, we can count the occurrence numbers of each entity in every valid substring. For instance, the occurrence number of entity 1 ("`nick koudas`") in $S[0, 8]$ is 4. As the occurrence number of entity 1 is larger than $\delta = 11 + 1 - q - \tau * q = 3$, the valid substring $S[0, 8]$ ("`nic kouds`") is a candidate for entity 1 ("`nick koudas`").

4. EXPERIMENTS

We compared our algorithms with state-of-the-art methods, approximate-string-based methods (AppSearch) [4], similarity-join-based methods (ED-Join) [6], and NGPP [5]. For AppSearch,

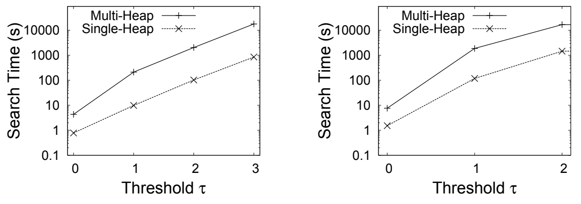

nic ic_ c_k _ko kou oud uds die iev eve ves esh sh_ h_c _cr cri riv iva
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

(a) A heap structure

⟨ entity, g_i ⟩	⟨ valid substrings, occurrence number ⟩
$\langle 1,0\rangle$	$\langle S[0,8],1\rangle$; $\langle S[0,9],1\rangle$; $\langle S[0,10],1\rangle$; $\langle S[0,11],1\rangle$; $\langle S[0,12],1\rangle$; $\langle S[0,13],1\rangle$; $\langle S[0,14],1\rangle$
$\langle 1,3\rangle$	$\langle S[0,8],2\rangle$; $\langle S[0,9],2\rangle$; $\langle S[0,10],2\rangle$; $\langle S[0,11],2\rangle$; $\langle S[0,12],2\rangle$; $\langle S[0,13],2\rangle$; $\langle S[0,14],2\rangle$; $\langle S[1,8],1\rangle$; $\langle S[1,9],1\rangle$; $\langle S[1,10],1\rangle$; $\langle S[1,11],1\rangle$; $\langle S[1,12],1\rangle$; $\langle S[1,13],1\rangle$; $\langle S[1,14],1\rangle$; $\langle S[2,8],1\rangle$; $\langle S[2,9],1\rangle$; $\langle S[2,10],1\rangle$; $\langle S[2,11],1\rangle$; $\langle S[2,12],1\rangle$; $\langle S[2,13],1\rangle$; $\langle S[2,14],1\rangle$; $\langle S[3,8],1\rangle$; $\langle S[3,9],1\rangle$; $\langle S[3,10],1\rangle$; $\langle S[3,11],1\rangle$; $\langle S[3,12],1\rangle$; $\langle S[3,13],1\rangle$; $\langle S[3,14],1\rangle$
...	...

(b) Occurrence tuples of valid substrings

Figure 3: A single-heap-based method for text "`nic kouds dievesh criva`".



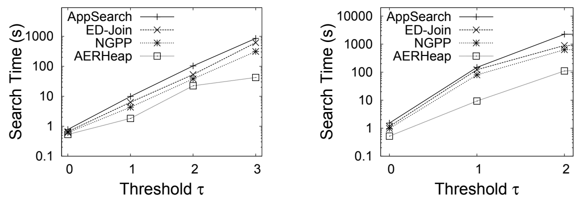
(a) DBLP ($q = 5$) (b) PubMed ($q = 3$)

Figure 4: Performance comparison of multi-heap-based methods and single-heap-based methods.

we modified the implementation of q-gram-based listmerger algorithms in Flamingo Project[1] to support approximate entity extraction. We also extended state-of-the-art similarity-join algorithm ED-Join to support approximate entity extraction. We downloaded the binary codes of ED-Join [6] and NGPP [5] from "Similarity Joins" project website[2]. We varied the gram length q and reported the best performance of each method. All the algorithms were implemented in C++ and compiled using GCC 4.2.3 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz processor and 4 GB memory. We used two real datasets, DBLP [3] and PubMed [4]. DBLP is a computer-science publication dataset. We selected 95,293 author names as entities and 10,000 papers with author and title as text. PubMed is a medical publication dataset. We selected 152,096 commonly used medical terms as entities and 10,000 publications with title and author names as text.

We first compared the multi-heap-based method and the single-heap-based method by varying edit-distance thresholds. Figure 4 shows the experimental results. We see that the single-heap-based method outperforms the multi-heap-based method by an order of magnitude, and even two orders of magnitude in some cases. There are two reasons that the single-heap-based method is better than the multi-heap-based method. Firstly, the multi-heap-based method needs to scan each gram inverted list of the text many times and the single-heap-based method only needs to scan them once. Secondly the multi-heap-based method needs to construct and adjust larger numbers of heaps, and the single-heap-based method only adjusts a single heap.

Then we compared our best method AERHeap (using length pruning techniques) with existing methods, AppSearch, ED-Join, and NGPP. Figure 5 shows the results. We see that our method AERHeap achieved the highest performance. This is because our method can use the shared computation across overlapped grams, but the existing methods cannot.



(a) DBLP ($q = 5$) (b) PubMed ($q = 3$)

Figure 5: Comparison with existing methods.

5. CONCLUSION

In this paper, we have studied the problem of approximate entity extraction with edit-distance thresholds. We proposed a heap-based framework to address this problem. The single-heap-based method only needs to construct a single heap and can extract all entities by scanning each gram inverted list of text only once. We have implemented our algorithms, and tested our method on three real datasets. The experimental results show that our method achieves high performance and outperforms state-of-the-art studies.

6. ACKNOWLEDGEMENT

This work is partly supported by the National Natural Science Foundation of China under Grant No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2009AA011906, and the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303103.

7. REFERENCES

[1] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, pages 805–818, 2008.
[2] S. Chaudhuri, V. Ganti, and D. Xin. Mining document collections to facilitate accurate approximate entity matching. *PVLDB*, 2(1):395–406, 2009.
[3] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
[4] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[5] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, pages 759–770, 2009.
[6] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[1]http://flamingo.ics.uci.edu/
[2]http://www.cse.unsw.edu.au/∼weiw/project/simjoin.html
[3]http://www.informatik.uni-trier.de/∼ley/db
[4]http://www.ncbi.nlm.nih.gov/pubmed/