Approximately Repetitive Structure Detection for Wrapper Induction

Xiaoying Gao, Peter Andreae and Richard Collins

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
Xiaoying.Gao, Peter.Andreae, Richard.Collins@mcs.vuw.ac.nz

Abstract. In recent years, much work has been invested into automatically learning wrappers for information extraction from HTML tables and lists. Our research has focused on a system that can learn a wrapper from a single unlabelled page. An essential step is to locate the tabular data within the page. This is not trivial when the structures of data tuples are similar but not identical. In this paper we describe an algorithm that can automatically detect approximate repetitive structures within one sequence. The algorithm does not rely on any domain knowledge or HTML heuristics and it can be used in detecting repetitive patterns and hence to learn wrappers from a single unlabeled tabular page.

1 Introduction

The amount of information on the Web is continuing to grow rapidly and there is an urgent need to create information extraction systems that can turn some of the online information from "human-readable only" to "machine readable". Information extraction systems that extract data tuples from particular information sources are often called wrappers. Building wrappers by hand is problematic because the number of wrappers needed is huge and the format of many sources is frequently updated. One solution is to be found in wrapper induction systems that learn wrappers from example Web pages.

A lot of wrapper induction systems have been constructed [1–4], especially for information extraction from HTML tables and lists; this research differs from most other systems in that our system aims to learn from a single unlabeled tabular page, and the page does not have to contain HTML tables or lists nor be generated by a program using predefined templates.

Some researchers have developed systems that learn wrappers from unlabelled Web pages [5–7], but these systems all require at least two training pages. Also, these systems are based on the assumption that the pages are generated by programs using predefined templates. There are many cases in which there is only one page available and the page is manually crafted so that the data formats are not regular and are often updated. Our research focuses on learning a wrapper from one single page, where the page contains a set of tuples of data items presented in a format that a human reader would perceive as a regular structure, even if the underlying HTML is not completely regular.

This paper focuses on the task of automatically detecting a region of approximately repetitive tabular data on a single page and identifying the sequence of "approximate repeat units" (ARUs) in that region that contain the data tuples. Extracting the data tuples from the ARUs is a separate process that is described in another paper [8].

The algorithm we describe is domain independent and does not rely on any HTML heuristics. So it does not require the tabular data to be presented in HTML tables or lists. Even in a page with tables or lists, the ARUs do not necessarily correspond to table rows or list items. For example, a page containing data tuples with five fields might present each tuple as two adjacent table rows with two fields in the first row and three fields in the second row. In this case, each ARU consists of two rows of the HTML table.

In this paper, we first formalise our problem, and then introduce our algorithm that automatically detects approximate repetition within one sequence. We then present our experimental results on using this algorithm in wrapper induction, and finally conclude and discuss future work.

2 Formalisation of the Problem

A web page can be represented by a string of tokens $t_1, t_2, \ldots t_n$, where each token represents a sequence of characters on the page. The tokens must be abstractions of the character strings so that tokens can be matched to determine their similarity. We currently use a different token type for each type of HTML tag, and two tokens types for text — one for numbers and the other for all other text. For a more sophisticated tokeniser, the tokens would not be atomic, and there would be a multivalued scale of similarity of tokens. However, for the sake of clarity of exposition of the algorithm, we will use single letters to represent tokens for most of the paper, and will ignore partial similarities of tokens. We assume that the page contains at least one region of semi-structured data so that a set of data tuples are presented on the page, and that the tokens in the region contain either values of the data tuples or formatting/presentation information.

The problem addressed in this paper is to find a region of the page $(t_k, \ldots t_m)$ that can be partitioned into two or more subsequences of tokens $ARU_1, ARU_2, \ldots, ARU_r$ where the ARU_i are all similar to each other. The measure of similarity should be such that ARUs corresponding to the data tuples on a page will be considered similar. Part of the problem is to define a syntactic definition of similarity that accommodates the kinds of errors and irregularities on real pages, but still captures the repetitive structure. For example, if a page is represented in a token string "XYABCDEABDEAPCDEXYET", the system should identify the sequence of three ARUs: "ABCDE", "AB-DE", "A**P**CDE". Note that the second ARU has a missing token ("C" is missing) and the third ARU has a mismatching token ("P" instead of "B"). Alternatively, we could describe the first and third ARU as having an additional token "C".

If a page has nested approximately repetitive structures, our task is to detect the exterior repetitive structure. The interior repetitive structure could then be found by reusing this algorithm on one of the ARUs. If a page has multiple approximately repetitive structures, our algorithm should be able to find all of them.

3 The Approximate Repeat Units Detection Algorithm

The ARUs detection algorithm is based on the observation that if a sequence of tokens contains a region consisting of m repeat units and the sequence is matched against itself, offset by the length of k repeat units,then the first $m-k$ repeat units will match against the last $m-k$ repeat units. Therefore, if we match a token sequence against itself at all possible offsets to find matching subsequences, we should be able to identify any repetitive regions and also the length of the repeat units. If the repeat units are identical, such matching is easy. If the repeat units are only approximately the same, with some mismatched, missing or additional tokens, then a more complicated matching, such as the Smith-Waterman algorithm [9], is required to find a good approximate match.

The limitation of the Smith-Waterman algorithm is that it will only find the best matching subsequence of two sequences, whereas we need to find *all* good matching subsequences within a single sequence. Our algorithm uses a dynamic programming algorithm, similar to Smith-Waterman, to construct a matrix representing all the matching subsequences, and then analyses the matrix to identify the approximately repetitive regions and the approximate repeat units.

3.1 Step1: Building the matrix

Given a sequence of n tokens, $t_1, t_2, \dots t_n$, the first step of the algorithm constructs an $n \times n$ matrix H of scores, where H_{ij} is the similarity score of the highest scoring pair of subsequences ending at the tokens t_i and t_j respectively. The score $H_{ij} = 0$ if there are no approximately similar subsequences ending at t_i and t_j. The similarity score of two subsequences is the sum of the token similarities of paired tokens minus any penalties for additional tokens in one subsequence that are not paired with tokens in the other subsequence.

Since the sequence is being matched against itself, $H_{ij} = H_{ji}$, and therefore the algorithm only needs to compute the upper right triangle of the matrix. Also, we are not interested in the trivial match of the whole sequence against itself exactly, so we set $H_{ii} = 0$. The algorithm builds the matrix from the top-left corner: for each cell it finds the best way to extend a pair of subsequences to include the new tokens. We can extend pairs of subsequences to include the tokens t_i and t_j in three different ways:

- We can extend a pair of subsequences ending at t_{i-1} and t_{j-1} by one token each, to include t_i paired with t_j.

- We can extend a pair of subsequences ending at t_{i-k} and t_j by including $t_{i-k+1}, t_{i-k+2}, \ldots, t_i$ as additional tokens in the first subsequence, not paired with any tokens in the second subsequence.
- We can extend a pair of subsequences ending at t_i and t_{j-k} by including $t_{j-k+1}, t_{j-k+2}, \ldots, t_j$ as additional tokens in the second subsequence, not paired with any tokens in the first subsequence.

The algorithm chooses whichever extension results in the best score for the new pair of subsequences.

The algorithm records which extension had the best score by storing a backpointer from (i, j) to the cell that it was extended from. The backpointers are stored in a matrix BP. Figure 1 shows the matrix H constructed for a short sequence of 11 tokens: "XABCDEABDEY". Note that the subsequence $t_2 \ldots t_6$ ("ABCDE") matches the subsequence $t_7 \ldots t_{10}$ ("ABDE"), with just one additional token. Each cell contains a score, which is greater than 0 if the two subsequences have an adequate similarity. The matrix BP is overlaid so that cells calculated by extending the subsequences of a previous cell have backpointers to previous cell. The path of backpointers from a cell shows the matching subsequences ending at that cell. The cells with bold scores represent the locally best matching subsequence. Note that the additional token (t_4) in the first subsequence appears in the path as a vertical backpointer.

	0	1	2	3	4	5	6	7	8	9	10	11
0		X	A	B	C	D	E	A	B	D	E	Y
1	X	0	0	0	0	0	0	0	0	0	0	0
2	A		0	0	0	0	0	**1**	0	0	0	0
3	B			0	0	0	0	0	**2**	0.7	0.3	0
4	C				0	0	0	0	**0.7**	1.7	0.3	0
5	D					0	0	0	0.3	**1.7**	1.3	0
6	E						0	0	0	0.3	**2.7**	1.3
7	A							0	0	0	1.3	2.3
8	B								0	0	1.0	1.0
9	D									0	0	0
10	E										0	0
11	Y											0

Fig. 1. Similarity Score Matrix H

The similarity score of tokens t_i and t_j is given by $S_{i,j} = 1$ if t_i and t_j match, and $S_{i,j} = -\frac{1}{3}$ if they do not match. The penalty for a string of k additional tokens is $1 + \frac{k}{3}$. We place a limit, r, on the number of consecutive mismatching and additional tokens, so that $H_{ij} = 0$ if $S_{i-k,j-l} < 0$ for all $1 \le k, l \le r$.

The algorithm is given in Figure 2.

Initialise H:
  $H_{i0} = H_{0i} = 0$ for $0 \le i \le n$
  $H_{ij} = 0$ for $1 \le j \le i \le n$
For $1 \le i \le j \le n$
  If $S_{i-k,j-l} < 0$ for all $1 \le k, l \le r$
    $H_{i,j} = 0, BP_{ij} = \text{null}$
  Else
    Compute max of
      $S_0 = 0$
      $S_1 = H_{i-1,j-1} + S_{i,j}$
      $S_2 = \max_{1 \le k < i} \left(H_{i-k,j} - (1 + \frac{k}{3})\right)$
      $S_3 = \max_{1 \le k < j} \left(H_{i,j-k} - (1 + \frac{k}{3})\right)$
    Set H_{ij} = maximum score.
    If maximum score was S_0: $BP_{ij} = \text{null}$
    Else set BP_{ij} to the cell from which the best score was extended.

Fig. 2. Algorithm to calculate the matrix H

3.2 Step 2: Finding best paths

Our algorithm finds the approximate repeat units by analysing the paths defined by the backpointer matrix, BP, and the corresponding scores in the matrix H. The paths in BP represent pairs of matched subsequences. The set of paths always forms a forest of trees, since nearby paths merge and share a common root. For example, figure 1 contains just a single tree of paths, with all the paths ending at the cell $(2, 7)$. matched subsequences.

We are only interested in the "best path" in each tree, representing the best matching pair of subsequences beginning at the root. The best path in a tree is the path from the root to the cell in the tree with the highest score in the matrix H. The best path collection is built by tracing paths back from every non-null cell in BP. The best path for each root is defined to start from the closest cell to the root with a score greater than or equal to 1, and to end at the cell with the best score.

For the token sequence in Figure 1, there is just one root, and the best path starts at $(2, 7)$ and ends at $(6, 10)$ corresponding to the pair of subsequences "ABCDE" and "ABDE". There is only one root because there is only one repetitive region, and the region contains only two approximate repeat units, so that there is only one offset of the sequence (an offset of 5 tokens) that generates a good match. Figure 3 shows a bigger example using the token string "XYABCED-ABDEAPCDETXYET", which contains a region with three ARU's, and some small regions with minor repetition. All the best paths are shown in Table 3.2. The region with three ARUs generates two best paths, one with an offset of 5 (matching the first two ARUs against the last two ARUs) and one with an offset of 9, (matching the first ARU against the last ARU).

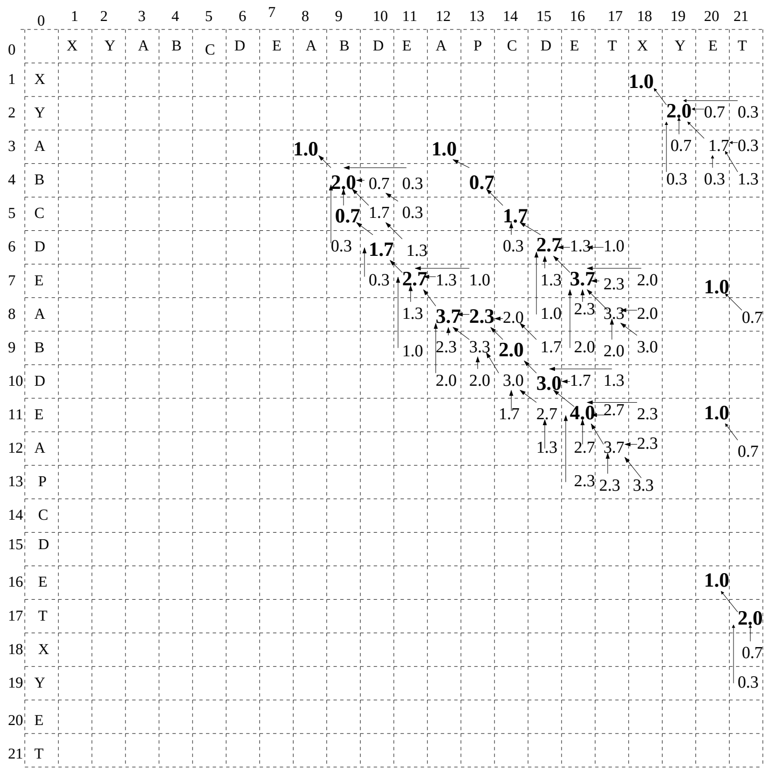

Fig. 3. Example with larger repetitive region

3.3 Step 3: Identifying repetitive regions and ARUs

A repetitive region with k ARUs will typically give rise to $k-1$ best paths, each one representing a match of the first m ARUs with the last m ARUs. The offsets of the paths correspond approximately to multiples of the length of the ARUs. The paths will also typically begin at the beginning of the first ARU and end at the end of the last ARU. Our algorithm exploits this structure to identify repetitive regions and the ARUs.

The algorithm attempts to group the best paths according to the following criteria:

- Paths starting at cells in approximately the same row, and

Table 1. Best paths in the example shown in Figure 3

No	Best paths	offset	Score	Subsequences
1	(3, 8) to (11, 16)	5	4	ABCDEAB-DE : AB-DEAPCDE
2	(3, 12) to (7, 16)	9	3.7	ABCDE : APCDE
3	(1, 18) to (2, 19)	17	2	XY : XY
4	(7, 20) to (7, 20)	13	1	E : E
5	(11, 20) to (11, 20)	9	1	E : E
6	(16, 20) to (17, 21)	4	2	ET : ET

- Paths ending at cells in approximately the same column, and
- Paths with offsets (the difference between the row and column of the starting cell) that are approximately multiples of the smallest offset in the group.

We consider a number to be approximately equal to another if their difference is at most r - the maximum number of consecutive mismatches or additional tokens. Paths that are not grouped with any other paths are discarded. In the example in Figure 3, the first two paths are grouped together and the other four are discarded.

From each group of paths, the algorithm constructs the description of a repetitive region and its ARUs:

- If the lowest row of the starting cells of the paths in the group is min, then the repetitive region starts at token t_{min}.
- If the highest column of the ending cells of the paths in the group is max, then the repetitive region ends at token t_{max}.
- Each column of a starting cell of a path in the group is the beginning of an ARU.

In the example in Figure 3, the repetitive region consists of tokens $t_3 \dots t_{16}$ and the ARUs consist of tokens $t_3 \dots t_7$ (“ABCDE”), $t_8 \dots t_{11}$ (“ABDE”), and $t_1 2 \dots t_{16}$ (“APCDE”).

4 AutoWrapper and Experimental Results

AutoWrapper[8] is a system we built that learns a wrapper from a single, unlabeled, tabular Web page. The earlier version[8] used handcrafted heuristics to identify candidate rows of the table. We have now extended AutoWrapper to use the ARU detection algorithm described above to find the tabular region and the approximate repeat units; Autowrapper then uses these as the candidate rows to build a wrapper.

We tested AutoWrapper on two sets of data and the experimental results are shown in Tables 2 and 4. The first set of data shown in Table 2 was downloaded from RISE (Repository of Online Information Sources Used in Information Extraction Tasks) at http://www.isi.edu/info-agents/RISE. Experimental results for the five Web sites are known in the literature [10, 3, 5] and

Table 2 shows a comparison of AutoWrapper with the three other systems — Wien [10], Stalker [3] and RoadRunner [5]. The second set of data used in Table 4 were downloaded from the VUW (Victoria University of Wellington) web site at http://www.mcs.vuw.ac.nz. The pages are semi-structured, but there is only one page of each kind, so that automatic extraction from one single page is essential.

Table 2. Comparative results

No.	Site	Source and URL	Wien	Stalker	RoadRunner	AutoWrapper
1	OKRA	RISE	√	√	√	√
2	BigBook	RISE, bigbook.com	√	√	√	√
3	La Weekly	RISE, laweekly.com	×	√	√	√not perfect
4	Address Finder	RISE, iaf.net	×	√	×	×
5	Quote Server	RISE	×	√	–no info	√

Table 3. Results on pages from http://www.mcs.vuw.ac.nz

No.	Site	Source and URL	Results
6	Graduate course	VUW, /courses/grad-courses.shtml	√
7	People, Staff page	VUW, /people/	×
8	Publications	VUW, /comp/Publications/index-byname.html	√ not perfect
9	Seminars	VUW, /events/seminars/upcoming-seminars.html	√
10	SE research group	VUW, /research/se-vuw/about.shtml	√

It can be seen from Table 2 that AutoWrapper gets better results than Wien since AutoWrapper works well for data with optional or missing items while Wien cannot; Stalker achieves better results than AutoWrapper, but Stalker requires labeled examples (as does Wien); AutoWrapper achieves results comparable to RoadRunner, but RoadRunner requires at least two pages, whereas AutoWrapper only needs one page.

Our system achieves about 70% success rate in both sets of data, considering Nos.3 and 8 as partial success. The experiments demonstrate that our system works on tabular pages with mismatch and additional tokens (Nos. 3, 5, 6, 10), data tuples presented in multiple rows (No. 1), non-table-list sites (Nos. 3, 10), and manually crafted sites (No. 10). For all web sites, AutoWrapper only requires a single unlabeled Web page. The algorithm is domain independent and does not rely on any HTML heuristics, and it works on semi-structured pages in different domains.

AutoWrapper failed on No. 4 and No. 7 due to the diversity of their data formats. The data tuples in No. 4 can have three to seven fields and the fields can come in a different order. No. 7 has a special format with six tables (as group names) and six lists (staff in each group), where each table is followed by a list and the length of the list varies from 1 to 20. AutoWrapper can detect the repetitive structure in Nos.3 and 8, but can not extract all the information. No. 3 has a list of credit cards presented as plain text at the end of a text paragraph and our tokeniser can not distinguish the list from the rest of the text. No. 8 presents the author as the title and publications as a list, where our system can extract the publications but the author is ignored as optional tokens.

5 Related Work

Most wrapper induction systems require labeled training examples [1, 3, 11]. There are three systems [5–7] that learn wrappers from unlabeled pages. RoadRunner [5] automatically generates wrappers by comparing HTML pages and analysing their similarities and differences, and it works well on tables and lists generated based on predefined templates. It outperforms other wrapper induction systems in that it can handle nested lists. Another system [6] uses AutoClass for automatic classification of data and uses grammar induction of regular languages for wrapper induction. It induces the structure of lists by exploiting the regularities both in the format of the pages and the data contained in them, while most other systems only consider formats and treat one text string as one single field. This system can therefore extract multiple data fields from one text string and it works on text-rich tables or lists. Another approach that also use grammatical inference for wrapper learning is [7]. All three systems require at least two pages for training the system, while our system requires a single page.

The problem of finding patterns in strings is also studied in the field of molecular biology. A lot of approaches are devoted to the discovery of patterns in biosequences. Our ways of calculating $H_{i,j}$ are similar to the Smith-Waterman algorithm [9], first published in a biology journal in 1970s. It can find the two subsequences of characters with the maximum similarity from two string sequences. Our algorithm finds patterns within one sequence and it needs one input instead of two. We also introduce a limit on the number of consecutive mismatching and additional tokens.

6 Conclusion and Future Work

We sucessfully built a wrapper induction system that can learn wrappers from one single unlabeled tabular training page. Our system is based on an algorithm that automatically detects approximate repeats within one sequence. The algorithm is domain independent and does not rely on any HTML heuristics. Our system AutoWrapper was tested on two data sets and achived about 70% success rate. The experiments show that our system works well on data with mismatching and additional tokens, on manaully crafted pages that are not program generated as well as on tabular pages that are not HTML tables or lists.

One limition we have noticed is that the tokens in a page are not equally important. We are exploring ways to improve the algorithm by introducing weights to tokens and introducing partial similarity for token comparison. In the future, we will also explore other application areas of this algorithm, particularly how it can improve the performance of our information extraction agents [12].

References

1. N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *IJCAI-97*, pages 729–735, Nagoya, Japan, 1997.
2. D. Freitag. Information extration from html: Application of a general machine learning approach. In *AAAI-98*, 1998.
3. I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *The 3rd conference on Autonomous Agents(Agent'99)*, 1999.
4. S. Soderland. Learning to extract text-based information from the world wide web. In *Proceedings of Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, 1997.
5. Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.
6. Kristina Lerman, Craig Knoblock, and Steven Minton. Automatic data extraction from lists and tables in web sources. In *Automatic Text Extraction and Mining workshop (ATEM-01), IJCAI-01, Seattle, WA*, 2001.
7. Theodore W. Hong and Keith L. Clark. Using grammatical inference to automate information extraction from the Web. *Lecture Notes in Computer Science*, 2168:216–223, 2001.
8. Xiaoying Gao, Mengjie Zhang, and Peter Andreae. Learning information extraction patterns from tabular web pages without manual labelling. In *IEEE/WIC International Conference on Web Intelligence (WI'03) October 13 - 17, 2003 Halifax, Canada*, pages 495–498, 2003.
9. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. of Mol. Biol.*, 147:195–197, 1981.
10. N. Kushmerick. *Wrapper Induction for Information Extraction*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1997.
11. William Cohen, Matthew Hurst, and Lee S. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *In The Eleventh International World Wide Web Conference WWW-2002*, 2002.
12. X. Gao and L. Sterling. Knowledge-based information agents. In J. G. Carbonell and J. Siekmann, editors, *Lecture Notes in Artificial Intelligence*, pages 229–238. Springer, 2001.