Outline

1. Automatic Wrapper Generation
2. Pattern Matching
3. RoadRunner
4. Conclusion
Outline

1. Automatic Wrapper Generation
2. Pattern Matching
3. RoadRunner
4. Conclusion
Problems of wrapper induction

Wrapper induction (supervised) has two main shortcomings:
Problems of wrapper induction

Wrapper induction (supervised) has two main shortcomings:

- It is unsuitable for a large number of sites due to the manual labeling effort
Problems of wrapper induction

Wrapper induction (supervised) has two main shortcomings:

- It is unsuitable for a large number of sites due to the manual labeling effort.
- Wrapper maintenance is very costly.
  - The Web is a dynamic environment.
  - Sites change constantly.
  - Since rules learnt by wrapper induction systems mainly use formatting tags, if a site changes its formatting templates, existing extraction rules for the site become invalid.
Unsupervised learning

- Due to these problems, automatic (or unsupervised) extraction has been studied
- Why is fully automatic extraction possible?
Due to these problems, automatic (or unsupervised) extraction has been studied.

Automatic extraction is possible because data records (tuple instances) in a Web site are usually encoded using a very small number of fixed templates.

It is possible to find these templates by mining repeated patterns.
The data extraction problem

- We have an abstract model of structured data on the Web (i.e., nested relations), and a HTML mark-up encoding of the data model.
- The general problem of data extraction is to recover the hidden schema from the HTML mark-up encoded data.
An Example

\[ \text{schema} = AB+C?D \]

- Note that, since we do not know the original schema, we do not know that \( A = \text{title}, B = \text{author}, \) etc.
Outline

1 Automatic Wrapper Generation
2 Pattern Matching
3 RoadRunner
4 Conclusion
The key is to find the encoding template from a collection of encoded instances of the same type

A natural way to do this is to detect repeated patterns from HTML encoding strings

String edit distance and tree edit distance are obvious techniques for the task
String edit distance

- String edit distance: the most widely used string comparison technique.
- The edit distance of two strings, \( s_1 \) and \( s_2 \), is defined as the minimum number of point mutations required to change \( s_1 \) into \( s_2 \), where a point mutation is one of:
  1. change a letter
  2. insert a letter
  3. delete a letter
For two strings \( s_1 \) and \( s_2 \), the edit distance \( d(s_1, s_2) \) is defined as:

\[
d(\epsilon, \epsilon) = 0
\]

\[
d(s, \epsilon) = d(\epsilon, s) = |s|
\]

\[
d(s_1 + c_1, s_2 + c_2) = \min(d(s_1, s_2) + r(c_1, c_2),
\]

\[
d(s_1 + c_1, s_2) + 1,
\]

\[
d(s_1, s_2 + c_2) + 1
\]

where \( r(c_1, c_2) = 0 \) if \( c_1 = c_2 \) and \( r(c_1, c_2) = 1 \) otherwise.
Dynamic programming

- We can use a matrix $m[0..|s_1|, 0..|s_2|]$ to hold the edit distances.
- The value in each cell is computed iteratively, from $m[0, 0]$
- The last cell, $m[|s_1|, |s_2|]$ will hold the required value of edit distance.

The matrix is defined as:

- $m[0, 0] = 0$
- $m[i, 0] = i$
- $m[0, j] = j$
- $m[i, j] = \min(m[i - 1, j - 1] + r(s_1[i], s_2[j]), m[i - 1, j] + 1, m[i, j - 1] + 1)$
An example

- To compute the edit distance between “XGYXYXYX” and “XYXYXYTX”

The string alignment is

```
XGYXYXYX
X-YXYXYTX
```

- The string alignment is

```
XGYXYXYX
X-YXYXYTX
```
An example

To compute the edit distance between “XGYXYXYX” and “XYXYXYTX”

The string alignment is

```
XGYXYXY-X
X-YXYXYTX
```

Question: How can this help in finding Web page templates?
Tree Edit Distance

- Tree edit distance between two trees $A$ and $B$ (labeled ordered rooted trees) is the cost associated with the minimum set of operations needed to transform $A$ into $B$
- The set of operations used to define tree edit distance includes three operations:
  - node removal
  - node insertion
  - node replacement
- A cost is assigned to each of the operations
Definition

Let $X$ be a tree and $X[i]$ be the $i$-th node of $X$.

A mapping $M$ between a tree $A$ and a tree $B$ is a set of ordered pairs $(i, j)$, where $i$ is a node in tree $A$ and $j$ is a node in tree $B$, such that, for every $(i_1, j_1), (i_2, j_2) \in M$:

1. $i_1 = i_2$ iff $j_1 = j_2$
2. $A[i_1]$ is on the left of $A[i_2]$ iff $B[j_1]$ is on the left of $B[j_2]$
An Example Mapping
The tree matching problem

- In the general setting:
  - mapping can cross levels, e.g., node $d$ in tree A and node $d$ in tree B.
  - replacements are also allowed, e.g., node $f$ in A and node $a$ in B
- General matching/mapping algorithms are computationally complex
  - In general $O(|A|^2|B|^2)$, although different solutions can have different complexities
- In practice we can use more restrictive algorithms
- An example is Simple Tree Matching (STM)
  - Liu & Zhai 2005
Simple Tree Matching

- STM is a top-down algorithm
- Instead of computing the edit distance of two trees, it evaluates their similarity by producing the maximum matching through dynamic programming
- Restrictions:
  - Node labels must match
  - Matching must occur at the same level
- Complexity: $O(|A||B|)$
Simple Tree Matching algorithm

**Simple_Tree_Matching**(A, B)
1. if the roots of the two trees A and B contain distinct symbols or there is a visual conflict between A and B
2. then return (0);
3. else \( m := \) the number of first-level sub-trees of A;
4. \( n := \) the number of first-level sub-trees of B;
5. Initialization: \( M[i, 0] := 0 \) for \( i = 0, \ldots, m; \)
   \( M[0, j] := 0 \) for \( j = 0, \ldots, n; \)
6. for \( i = 1 \) to \( m \) do
7. \hspace{1em} for \( j = 1 \) to \( n \) do
8. \hspace{2em} \( M[i, j] := \max(M[i, j-1], M[i-1, j], M[i-1, j-1]+W[i, j]) \);
   \hspace{2em} where \( W[i, j] = \) Simple_Tree_Matching\((A_i, B_j)\)
9. return \( (M[m, n]+1) \)
An Example

Matching = 5
An Example

Matching = 5

Question: How can this help in finding Web page templates?
Outline

1. Automatic Wrapper Generation
2. Pattern Matching
3. RoadRunner
4. Conclusion
The RoadRunner Algorithm

- RoadRunner is an algorithm for automatic wrapper generation
- Uses a nested-tuple representation of the page schema
  - Similar to what we have seen before
- Uses a RE-based model of the extraction program
- It infers the schema and program from a given set of Web pages
Union-Free Regular Expressions

Definition

Given an alphabet of symbols $\Sigma$ and a special token $\#pcdata \notin \Sigma$,

A UFRE over $\Sigma$ is a string over $\Sigma \cup \{\#pcdata, +, ?, (, )\}$ defined as follows:

- The empty string $\epsilon$ and all elements of $\Sigma \cup \{\#pcdata\}$ are UFRE
- If $A$ and $B$ are UFRE, then $AB$, $(A)^+$ and $(A)^?$ are regular expressions
- Note: $(A)^*$ stands for $((A)^+)^*$

Question: what is the difference from full regular expressions?
UFRE and nested tuples

- RoadRunner explores the close correspondence between nested tuples and union-free regular expressions
RoadRunner explores the close correspondence between nested tuples and union-free regular expressions

\#pcdata \Rightarrow \text{value fields}

+ \Rightarrow \text{(nested) lists}

? \Rightarrow \text{optional fields}
RoadRunner explores the close correspondence between nested tuples and union-free regular expressions.

- `#pcdata` ⇒ value fields
- `+` ⇒ (nested) lists
- `?` ⇒ optional fields

UFRE have limitations.
RoadRunner explores the close correspondence between nested tuples and union-free regular expressions

\[ \#pcdata \Rightarrow \text{value fields} \]
\[ + \Rightarrow \text{(nested) lists} \]
\[ ? \Rightarrow \text{optional fields} \]

UFRE have limitations

- No disjunctions: it is not possible to code “either A or B”
RoadRunner explores the close correspondence between nested tuples and union-free regular expressions

\[ \#pcdata \Rightarrow \text{value fields} \]
\[ + \Rightarrow (\text{nested}) \text{ lists} \]
\[ ? \Rightarrow \text{optional fields} \]

UFRE have limitations
- No disjunctions: it is not possible to code "either A or B"

However, they are still a good approximation
- Capture most cases
- Simpler to process than full regular expressions
Given a set of HTML strings $s_1, s_2, \ldots, s_k$ that encode a set of items $i_1, i_2, \ldots, i_k$ of type $\tau$, it is possible to find the minimal UFRE $\sigma$ whose language $L(\sigma)$ contains $s_1, s_2, \ldots, s_k$.

The UFRE $\sigma$ can also be used as a wrapper to extract $i_1, i_2, \ldots, i_k$.

Thus, solving the extraction problem consists in finding $\sigma$. 
The matching technique

ACME: Align, Collapse under Mismatch, and Extract

- Two main items are processed:
  1. The **sample**: a list of tokens
  2. The **wrapper**: a UFRE

- Given two Web pages $p_1$ and $p_2$:
  - Use $p_1$ as the initial wrapper
  - Use $p_2$ as the sample
  - Parse the sample using the wrapper
  - Generalize the wrapper by solving *mismatches*
Mismatches

Two types of mismatches:

- **String mismatches**: different strings occur at the same position of wrapper and sample
Mismatches

Two types of mismatches:

- **String mismatches:** different strings occur at the same position of wrapper and sample
  - Indicate the occurrence of field values
Mismatches

Two types of mismatches:

1. **String mismatches**: different strings occur at the same position of wrapper and sample
   - Indicate the occurrence of field values

2. **Tag mismatches**: different tags occur at the same position of wrapper and sample, or a string occurs where a tag should occur
Mismatches

Two types of mismatches:

1. **String mismatches**: different strings occur at the same position of wrapper and sample
   - Indicate the occurrence of field values

2. **Tag mismatches**: different tags occur at the same position of wrapper and sample, or a string occurs where a tag should occur
   - Indicate the occurrence of optional fields
   - Or the occurrence of lists
Discovering fields

Wrapper generalization:
Discovering fields

Wrapper generalization:

- Replaced the mismatch with \#pcdata
Discovering optional fields

Pattern location:

Wrapper generalization:
Discovering optional fields

Pattern location:

- Pattern is in the wrapper: locate `<IMG src=.../>` in the wrapper
- Pattern is in the sample: locate `<UL>` in the sample

Wrapper generalization:
Discovering optional fields

Pattern location:

- Pattern is in the wrapper: locate `<IMG src=.../>` in the wrapper
- Pattern is in the sample: locate `<UL>` in the sample

Wrapper generalization:

- Insert optional pattern in wrapper
  - e.g. (`<IMG src=.../>`)?
Discovering lists

1. Find the terminal tag

2. Find the initial tag
Discovering lists

Discover the **square location**:

1. **Find the terminal tag**
   - We know that some squares have been matched

2. **Find the initial tag**
Discovering lists

Discover the square location:

1. **Find the terminal tag**
   - We know that some squares have been matched
   - Thus, if we go one step backwards, we find the finishing tag

2. **Find the initial tag**
Discovering lists

Discover the **square** location:

1. Find the **terminal tag**
   - We know that some squares have been matched
   - Thus, if we go one step backwards, we find the finishing tag

2. Find the **initial tag**
   - The mismatch indicates the end of the list on one side and the beginning of the square on the other
     - We don’t know which side is which (is it </UL> or </LI>?)
Discovering lists

Discover the **square** location:

1. **Find the terminal tag**
   - We know that some squares have been matched
   - Thus, if we go one step backwards, we find the finishing tag

2. **Find the initial tag**
   - The mismatch indicates the end of the list on one side and the beginning of the square on the other
     - We don’t know which side is which (is it </UL> or </LI>?)
   - Search the wrapper for </UL> . . . </LI>
Discovering lists

Discover the **square** location:

1. **Find the terminal tag**
   - We know that some squares have been matched
   - Thus, if we go one step backwards, we find the finishing tag

2. **Find the initial tag**
   - The mismatch indicates the end of the list on one side and the beginning of the square on the other
     - We don’t know which side is which (is it \texttt{</UL>} or \texttt{</LI>}?)
   - Search the wrapper for \texttt{</UL>}\ldots\texttt{</LI>}
   - Search the sample for \texttt{</LI>}\ldots\texttt{</LI>}

---

12: \texttt{</LI>}
13: \texttt{</LI>}
14-16: \texttt{</I>Title:\texttt{/I>}
17: \texttt{Comp. Sys.}
18: \texttt{</LI>}
19: \texttt{</UL>}
20: \texttt{</HTML>}
13: \texttt{</LI>}
14: \texttt{</LI>}
15-17: \texttt{</I>Title:\texttt{/I>}
18: \texttt{HTML Scripts}
19: \texttt{</LI>}
20: \texttt{</LI>}
21-23: \texttt{</I>Title:\texttt{/I>}
24: \texttt{Javascript}
25: \texttt{</LI>}

- `string mismatch (#PCDATA)`
- `tag mismatch (+)`
- `terminal tag search and square matching`
Square matching (to check if it is a true square)
Square matching (to check if it is a true square)

- Search for matches backwards
  - Match 25 and 19, match 24 and 18, ...
Discovering lists (cont.)

- **Square matching** (to check if it is a true square)
  - Search for matches backwards
    - Match 25 and 19, match 24 and 18, ...
  - Stop when full square is matched
    - I.e., when 20 and 14 match
Discovering lists (cont.)

12:   </LI>
13:    <LI>
14-16:  <I>Title:</I>
17:    Comp. Sys.
18:    </LI>
19:    </UL>
20:   </HTML>

13:    </LI>
14:    <LI>
15-17:  <I>Title:</I>
18:    HTML Scripts
19:    </LI>
20:    <LI>
21-23:  <I>Title:</I>
24:    Javascript
25:    </LI>

string mismatch (#PCDATA)
tag mismatch (+)
terminal tag search and
square matching

Wrapper generalization
Wrapper generalization

- Replace sequences of squares in the wrapper with the generalized expression

\[
(<\text{LI}><\text{I>Title:</I>}\#\text{pcdata}</LI>)^+ 
\]
- Wrapper after solving mismatches:

```html
<HTML>
  Books of:<B>#PCDATA</B>
  (<IMG src=.../> )?
  <UL>
    ( <LI><I>Title:</I>#PCDATA</LI> )+
  </UL></HTML>
```

Note: we must look for lists before optionals. Why?
The need for recursion

- Solving one mismatch may cause other mismatches to be found
  - E.g., when matching squares on a list
- These are called internal mismatches
- They must be solved recursively
The need for recursion (example)
Backtracking

- When matching, not all alternatives lead to the correct solution
- Example:
  - When looking for candidate squares, we may find many incorrect initial and terminal tags
- If a solution is not found, backtracking must be performed
Matching can be seen as solving an AND-OR tree
Matching is an exponential problem
- Worst case: exploring the full AND-OR tree
- Can be optimized through pruning

Pruning strategies:
Matching is an exponential problem
- Worst case: exploring the full AND-OR tree
- Can be optimized through **pruning**

**Pruning strategies:**
1. Limit the number of candidate optionals/squares
   - I.e. prune the OR nodes
Lowering the complexity

- Matching is an exponential problem
  - Worst case: exploring the full AND-OR tree
- Can be optimized through pruning

Pruning strategies:

1. Limit the number of candidate optionals/squares
   - I.e. prune the OR nodes
2. Do not backtrack if the chance of being wrong is very low
   - E.g., if there was a match for a list (externally), do not check if it was an optional
Lowering the complexity

- Matching is an exponential problem
  - Worst case: exploring the full AND-OR tree
- Can be optimized through pruning

Pruning strategies:

1. Limit the number of candidate optionals/squares
   - I.e. prune the OR nodes
2. Do not backtrack if the chance of being wrong is very low
   - E.g., if there was a match for a list (externally), do not check if it was an optional
3. Ignore unlikely patterns
   - E.g., optionals/iterators delimited by optional patterns, such as ((<HR>)?<LI>#{pcdata}</LI>)+
Outline

1 Automatic Wrapper Generation
2 Pattern Matching
3 RoadRunner
4 Conclusion
Summary

Wrapper induction

Advantages:

- Only the target data are extracted as the user can label only data items that he/she is interested in.
- Due to manual labeling, there is no integration issue for data extracted from multiple sites as the problem is solved by the user.

Disadvantages:

- It is not scalable to a large number of sites due to significant manual efforts. Even finding the pages to label is non-trivial.
- Wrapper maintenance (verification and repair) is very costly if the sites change frequently.
Automatic extraction

- Advantages:
Automatic extraction

- Advantages:
  - It is scalable to a huge number of sites due to the automatic process.
Automatic extraction

- Advantages:
  - It is scalable to a huge number of sites due to the automatic process.
  - There is little maintenance cost.
Automatic extraction

- Advantages:
  - It is scalable to a huge number of sites due to the automatic process.
  - There is little maintenance cost.

- Disadvantages:
Automatic extraction

- **Advantages:**
  - It is scalable to a huge number of sites due to the automatic process.
  - There is little maintenance cost.

- **Disadvantages:**
  - It may extract a large amount of unwanted data because the system does not know what is interesting to the user. Domain heuristics or manual filtering may be needed to remove unwanted data.
Automatic extraction

- **Advantages:**
  - It is scalable to a huge number of sites due to the automatic process.
  - There is little maintenance cost.

- **Disadvantages:**
  - It may extract a large amount of unwanted data because the system does not know what is interesting to the user. Domain heuristics or manual filtering may be needed to remove unwanted data.
  - Extracted data from multiple sites need integration, i.e., their schemas need to be matched.
Conclusions

- In terms of extraction accuracy, it is reasonable to assume that wrapper induction is more accurate than automatic extraction. However, there is no reported comparison.

Applications

- Wrapper induction should be used in applications in which the number of sites to be extracted and the number of templates in these sites are not large.
- Automatic extraction is more suitable for large scale extraction tasks which do not require accurate labeling or integration.

Still an active research area!
Questions?
References
