# Web Intelligence

**Course Code: DJ19CEC801 | Sem VIII | A.Y. 2024-2025**

**By Mr. Vivian Lobo**

**Assistant Professor**

**Department of Computer Engineering**

**SVKM's Dwarkadas J. Sanghvi College of Engineering**

| Program: Final Year B.Tech. in Computer Engineering | Semester: VIII |
|---|---|
| Course: Web Intelligence | Course Code: DJ19CEC801 |
| Course: Web Intelligence Laboratory | Course Code: DJ19CEL801 |

| Teaching Scheme (Hours / week) | | | | Evaluation Scheme | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Semester End Examination Marks (A) | | | Continuous Assessment Marks (B) | | | Total marks = (A+ B) | | |
| Lectures | Practical | Tutorial | Total Credits | Theory | | | Term Test 1 | Term Test 2 | Avg. | | | |
| | | | | 75 | | | 25 | 25 | 25 | 100 | | |
| | | | | Laboratory Examination | | | Term work | | Total Term work | 50 | | |
| 3 | 2 | - | 4 | Oral | Practical | Oral &Practical | Laboratory Work | Tutorial / Mini project / presentation/ Journal | | | | |
| | | | | 25 | - | - | 15 | 10 | 25 | | | |

**Pre-requisite: Statistics, Machine Learning, Data Mining**

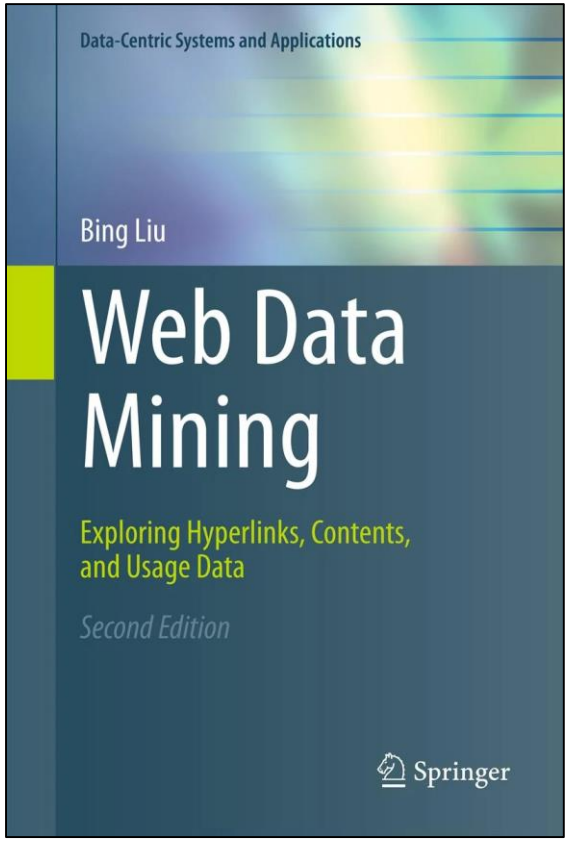
**Course Objectives:**

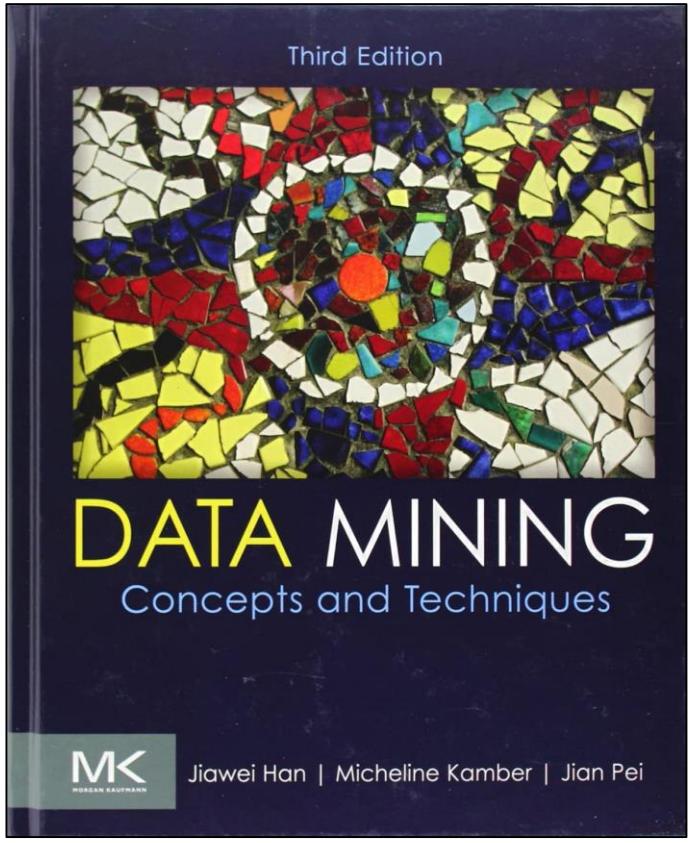
1. To gain a background in Web mining techniques
2. To extract knowledge from the social web for web analytics
3. To enable students to solve complex real-world problems for sentiment analysis and Recommendation systems.

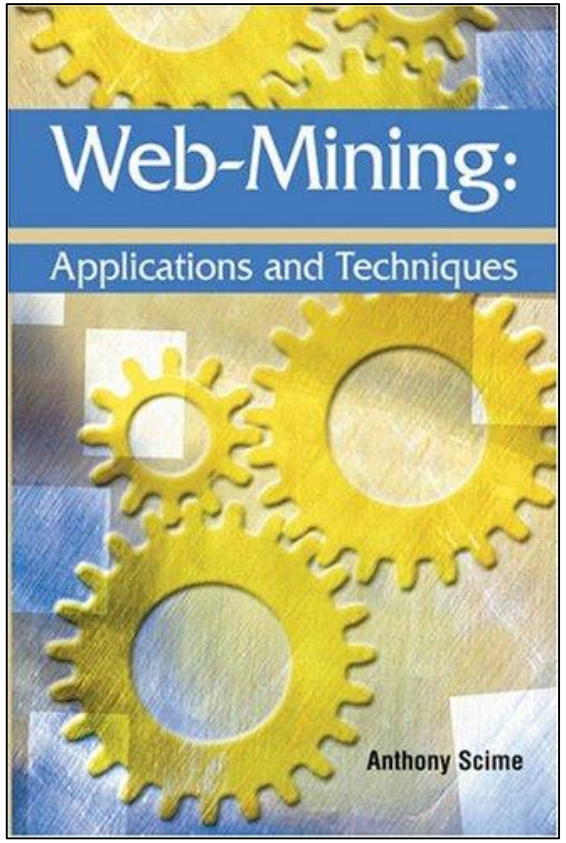
**Outcomes:** On successful completion of course learner will be able to:

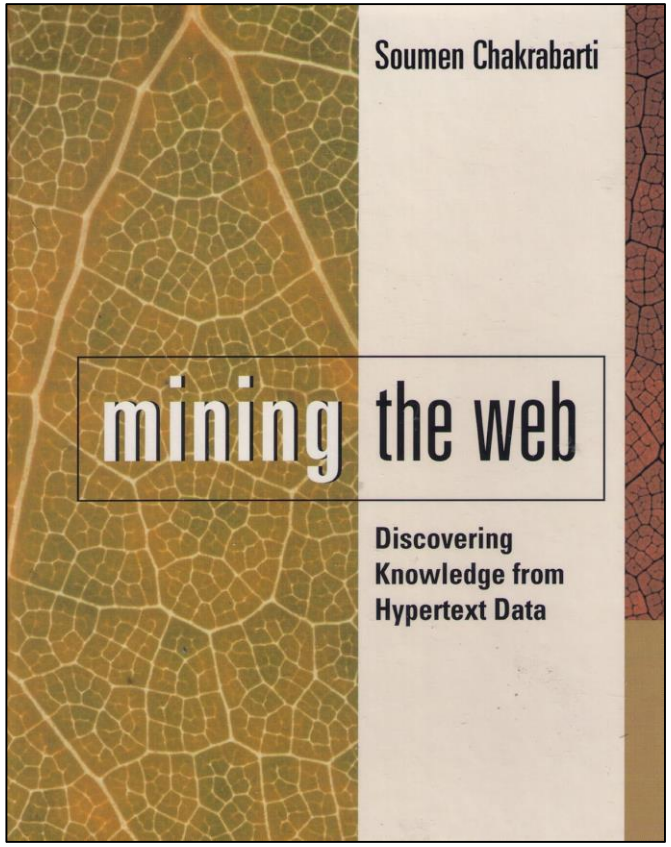
1. Interpret the terminologies and perspectives of Web Mining.
2. Perform social network analysis to identify communities and network properties in social media sites.
3. Extract and Integrate information from the web for real-world scenarios.
4. Design new solutions to opinion extraction and sentiment classification problems
5. Provide solutions to the emerging problems with social media using Recommendation systems

# Books Recommended




**Books Recommended:**

1. Web Data Exploring Hyperlinks, Contents, and Usage , Bing Liu , Springer, Second Edition

**Reference books:**

1. Data Mining: Concepts and Techniques, Second Edition Jiawei Han, Micheline Kamber (Elsevier Publications)
2. Web Mining: Applications and Techniques by Anthony Scime
3. Mining the Web: Discovering Knowledge from Hypertext Data by Soumen Chakrabarti

# Coursera Courses Recommended


https://www.coursera.org/learn/social-media-analytics-introduction#modules

# Evaluation Scheme (DJ19)

**Evaluation Scheme:**

***Semester End Examination (A):***

*Theory:*

1. Question paper based on the entire syllabus, summing up to 75 marks.
2. Total duration allotted for writing the paper is 3 hrs.

# Evaluation Scheme (DJ19)

*Laboratory:*

1. Oral Examination will be based on the entire syllabus including, the practical's performed during laboratory sessions.

***Continuous Assessment (B):***

*Theory:*

1. Two term tests of 25 marks each will be conducted during the semester out of which; one will be a compulsory term test (on minimum 02 Modules) and the other can either be a term test or an assignment on live problems or a course project.
2. Total duration allotted for writing each of the paper is 1 hr.
3. Average of the marks scored in both the two tests will be considered for final grading.

*Laboratory: (Term work)*

The distribution of marks for term work shall be as follows:

i. Laboratory work (Performance of Experiments): 15 Marks
ii. Journal Documentation (Write-up and Assignments: 10 marks

The final certification and acceptance of term work will be subject to satisfactory performance of laboratory work and upon fulfilling minimum passing criteria in the term work.

# List of Experiments

**Subject:** Web Intelligence Laboratory **Laboratory Course Code:** DJ19CEL801

**Class:** B.Tech. **SEM:** VIII

*S/W/Tools to be used: Python, Jupyter Notebook, Google Colab*

| Sr. No. | Title of Experiments | COs |
|---|---|---|
| 1. | Implementation of Page rank estimation | CO2 |
| 2. | Design a crawler to gather web information | CO2 |
| 3. | Implement a wrapper induction technique to gather data from the web | CO3 |
| 4. | Latent Semantic Indexing | CO1 |
| 5. | Use linguistic techniques for schema matching | CO3 |
| 6. | Perform Opinion spam detection | CO4 |
| 7. | Using Google Analytics, perform Audience Analysis, Acquisition Analysis, Behavior Analysis, Conversion Analysis | CO5 |
| 8. | Apply analytics to social media activity (Using FB, Twitter, Instagram or any social media dataset) | CO5 |

## Module 3:

# Structured Data Extraction

| | | |
|---|---|---|
| 3 | **Structured Data Extraction**<br><br>Structured Data Extraction: Wrapper Generation, Preliminaries, Wrapper Induction, Instance-Based Wrapper Learning, Automatic Wrapper Generation: Problems, String Matching and Tree Matching, Building DOM Trees, Extraction Based on a Single List Page, Extraction Based on Multiple Pages. | 8 |

# Road map

2

- **Introduction**
- Data Model and HTML encoding
- Wrapper induction
- Automatic Wrapper Generation: Two Problems
- String Matching and Tree Matching
- Multiple Alignments
- Building DOM Trees
- Extraction Given a List Page: Flat Data Records
- Extraction Given a List Page: Nested Data Records
- Extraction Given Multiple Pages
- Summary

# Introduction

3

- A large amount of information on the Web is contained in regularly structured data objects.
  - often data records retrieved from databases.
- Such Web data records are important: lists of products and services.
- Applications: e.g.,
  - Comparative shopping, meta-search, meta-query, etc.
- **We introduce:**
  - Wrapper induction (supervised learning) A program for extracting structured data is usually called a **wrapper**.
  - automatic extraction (unsupervised learning)

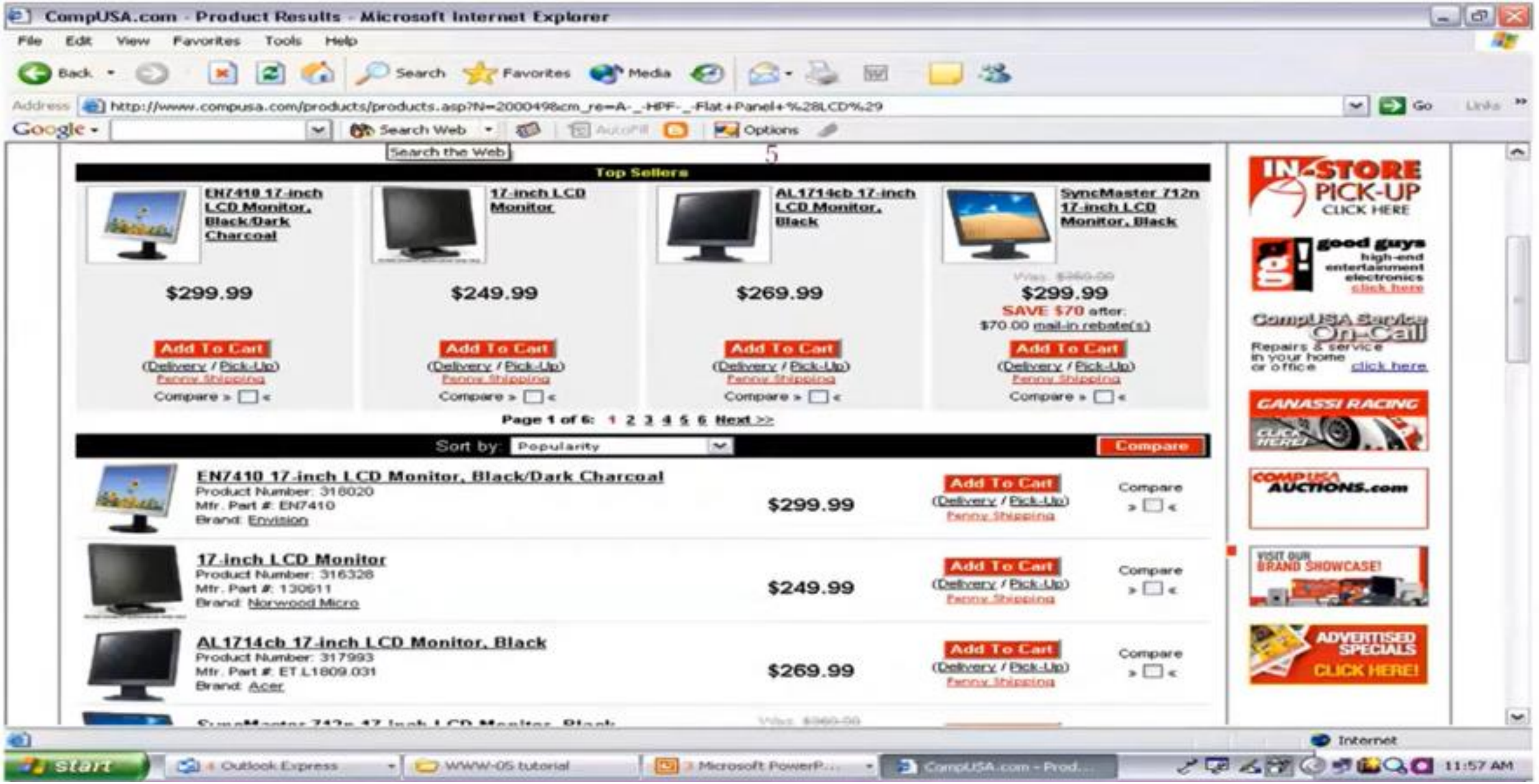
**Wrapper induction:** This is the supervised learning approach, and is semi-automatic. The work started around 1995-1996. In this approach, a set of extraction rules is learned from a collection of manually labeled pages or data records. The rules are then employed to extract target data items from other similarly formatted pages.

**Automatic extraction:** This is the unsupervised approach started around 1998. Given a single or multiple pages, it automatically finds patterns or grammars from them for data extraction. Since this approach eliminates the manual labeling effort, it can scale up data extraction to a huge number of sites and pages.

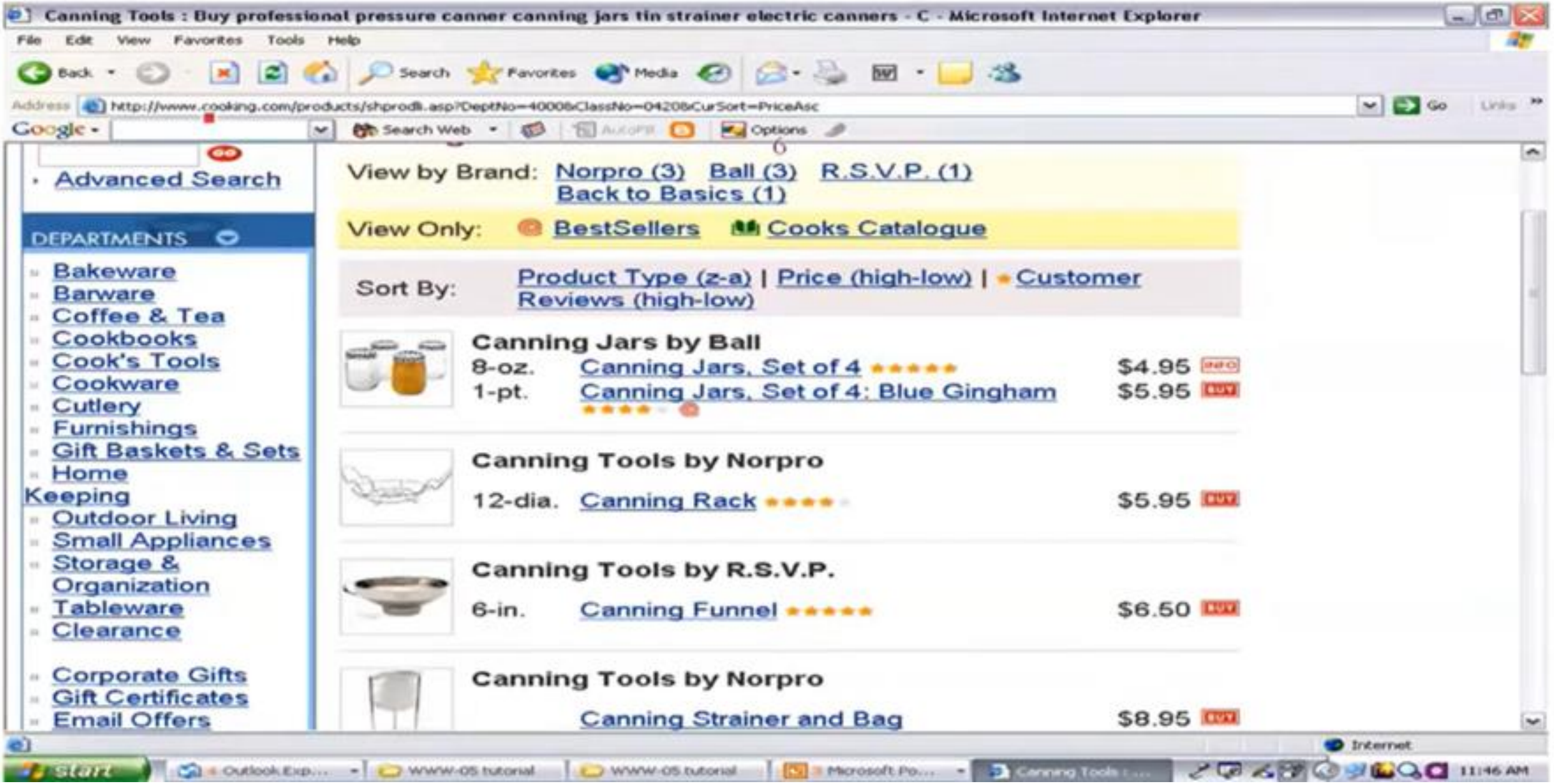
# Two types of data rich pages

4

- List pages
  - Each such page contains one or more lists of data records.
  - Each list in a specific region in the page
  - Two types of data records: flat and nested
- Detail pages
  - Each such page focuses on a single object.
  - But can have a lot of related and unrelated information




**Flat List Page**




**Nested List Page**


Ball Canning Jars, Set of 4 - Cooking.com - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://www.cooking.com/products/shprodde.asp?SKU=142008
Go
Google Search Web AutoFill Options
SEARCH
Keyword or Item#
GO
Advanced Search
DEPARTMENTS
Bakeware
Barware
Coffee & Tea
Cookbooks
Cook's Tools
Cookware
Cutlery
Furnishings
Gift Baskets & Sets
Home
Keeping
Outdoor Living
Small Appliances
Storage & Organization
Tableware
Clearance
Corporate Gifts
Canning Tools > Canning Jars
Canning Jars, Set of 4
(8-oz.)
by Ball
Click image for larger view
Also available in:
1-pt. - Blue Gingham Red Gingham
Ball has been synonymous with canning and preserving for decades, so it's no wonder they've added these decorative jars to their collection. These 8-oz. Canning Jars with Gingham Lids (colors may vary) add some fun to the process and color to
Our Price: $4.95
CHECKOUT
OUT OF STOCK
Email me when in stock
Sku# 142008
Rated by 2 Reviewers
See Reviews
RELATED ITEMS
Clearly Delicious
Elisabeth Lambert Ortiz's treasury of preserved, pickled &...
$19.96 BUY
Blue Gingham Canning Jars, Set of 4
"Ball" has been synonymous with canning and preserving for decades...
$5.95 BUY
Internet
start
Outlook Express
WWW-05 tutorial
Microsoft PowerP...
Ball Canning Jars, Set...
11:53 AM

**Detail Page**


(A) An example of a nested data record

| | | | | | |
|---|---|---|---|---|---|
| image 1 | Cabinet Organizers by Copco | 9-in. | Round Turntable: White | ***** | $4.95 |
| image 1 | Cabinet Organizers by Copco | 12-in. | Round Turntable: White | ***** | $7.95 |
| image 2 | Cabinet Organizers | 14.75x9 | Cabinet Organizer (Non-skid): White | ***** | $7.95 |
| image 3 | Cabinet Organizers | 22x6 | Cookware Lid Rack | **** | $19.95 |

(B) Extraction results

Fig. 9.3(A) contains some nested data records, which makes the problem more interesting and also harder. The first product, "Cabinet Organizers by Copco," has two sizes (9-in. and 12-in.) with different prices. These two organizers are not at the same level as "Cabinet Organizers by Copco".

**Our objective:** We want to extract the data and produce the data table given in Fig. 9.3(B). "image 1" and "Cabinet Organizers by Copco" are repeated for the first two rows due to the nesting.

## 1st Approach to Data Extraction:

# Wrapper Induction (Supervised Learning)

# Wrapper Induction

- A wrapper induction system learns data extraction rules from a set of labeled training examples.
- Labeling is usually done manually, which simply involves marking the data items in the training pages/examples that the user wants to extract.
- The learned rules are then applied to extract target data from other pages with the same mark-up encoding or the same template.

# Rules use landmarks

24

- The extraction rules are based on the idea of **landmarks**.
  - Each landmark is a sequence of *consecutive* tokens.
- Landmarks are used to locate the beginning and the end of a target item.
- Rules use landmarks

# Extraction using two rules

23

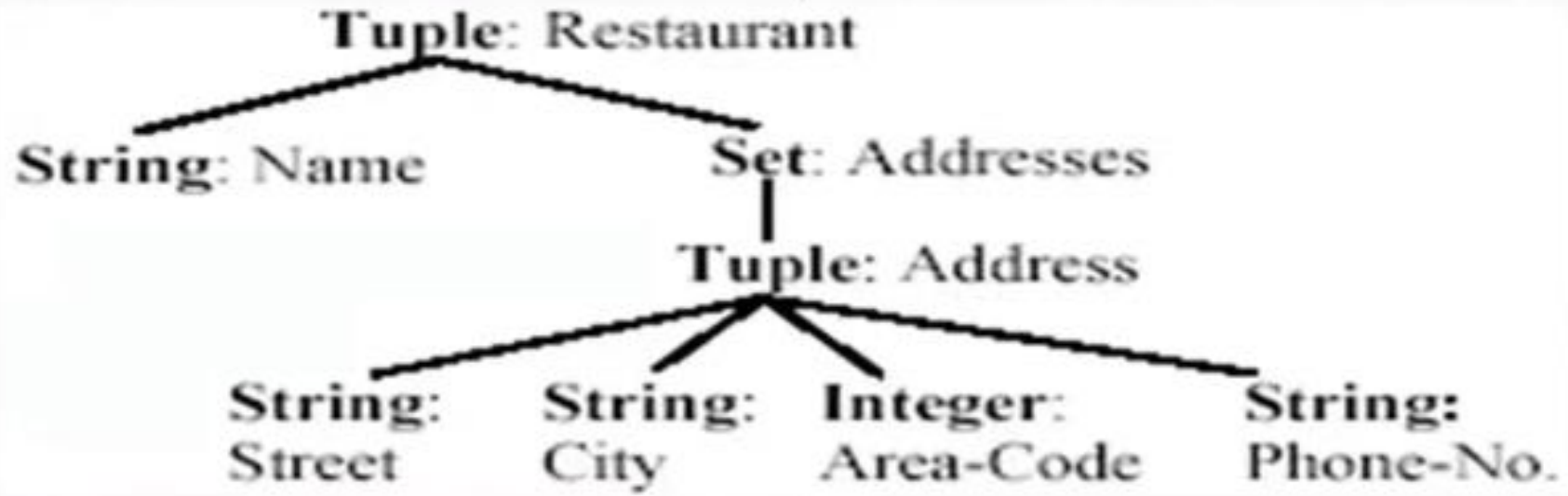
- Each extraction is done using two rules,
  - **a start rule** and **a end rule**.
- The start rule identifies the beginning of the node and the end rule identifies the end of the node.
  - This strategy is applicable to both leaf nodes (which represent data items) and list nodes.
- For a list node, **list iteration rules** are needed to break the list into individual data records (tuple instances).

# Hierarchical representation: type tree



**Restaurant Name: Good Noodles**

- 205 Willow, *Glen*, Phone 1-*773*-366-1987
- 25 Oak, *Forest*, Phone (800) 234-7903
- 324 Halsted St., *Chicago*, Phone 1-*800*-996-5023
- 700 Lake St., *Oak Park*, Phone: (708) 798-0008



# Data extraction based on EC tree



The extraction is done using a tree structure called the *EC* tree (**embedded catalog tree**).

The *EC* tree is based on the type tree above.


To extract each target item (a node), the wrapper needs a rule that extracts the item from its parent.

# Rules are not unique



- Note that a rule may not be unique. For example, we can also use the following rules to identify the beginning of the name:

  **R3**: *SkiptTo*(Name *_Punctuation_* *_HtmlTag_*)

or **R4**: *SkiptTo*(Name) *SkipTo*(<b>)

- **R3** means that we skip everything till the word "Name" followed by a punctuation symbol and then a HTML tag. In this case, "Name *_Punctuation_* *_HtmlTag_*" together is a landmark.
  - *_Punctuation_* and *_HtmlTag_* are **wildcards**.

# Extract area codes

27

1. Identify the entire list of addresses. We can use the start rule *SkipTo*(<br><br>), and the end rule *SkipTo*(</p>).
2. Iterate through the list (lines 2-5) to break it into 4 individual records (lines 2 - 5). To identify the beginning of each address, the wrapper can start from the first token of the parent and repeatedly apply the start rule *SkipTo*(<li>) to the content of the list. Each successive identification of the beginning of an address starts from where the previous one ends. Similarly, to identify the end of each address, it starts from the last token of its parent and repeatedly apply the end rule *SkipTo*(</li>).

Once each address record is identified or extracted, we can extract the area code in it. Due to variations in the format of area codes (some are in italic and some are not), we need to use disjunctions. In this case, the disjunctive start and the end rules are respectively **R5** and **R6**:

**R5:** **either** SkipTo( ( )
**or** SkipTo(-)

**R6:** **either** SkipTo( ) )
**or** SkipTo()

In a disjunctive rule, the disjuncts are applied sequentially until a disjunct can identify the target node.

# Learning extraction rules

28

- Stalker uses sequential covering to learn extraction rules for each target item.
  - In each iteration, it learns a perfect rule that covers as many positive examples as possible without covering any negative example.
  - Once a positive example is covered by a rule, it is removed.
  - The algorithm ends when all the positive examples are covered. The result is an ordered list of all learned rules.

# Refinement

34

- To specialize a disjunct by adding more **terminals** to it.
- A **terminal** means a token or one of its matching wildcards.
- We hope the refined version will be able to uniquely identify the positive items in some examples without matching any negative item in any example in *E*.
- Two types of refinement
  - Landmark refinement
  - Topology refinement

- The next iteration of LearnRule() is left with E1 and E3.
- LearnDisjunct() will select E1 as the Seed Two candidates are then generated:

  D3: SkipTo( **** )

  D4: SkipTo( *_HtmlTag_* )
- Both these two candidates match early in the uncovered examples, E1 and E3. Thus, they cannot uniquely locate the positive items.
- Refinement is needed.

E1: 205 Willow, Glen, Phone 1-773-366-1987
E2: 25 Oak, Forest, Phone (800) 234-7903
E3: 324 Halsted St., Chicago, Phone 1-800-996-5023
E4: 700 Lake St., Oak Park, Phone: (708) 798-0008

**Fig. 9.9.** Training examples: four addresses with labeled area codes

# Landmark refinement

35

- Landmark refinement: Increase the size of a landmark by concatenating a terminal.
  - E.g.,

**D5**: SkipTo( - <i>)

**D6**: SkipTo( _*Punctuation*_ <i>)

# Topology refinement

- Topology refinement: Increase the number of landmarks by adding 1-terminal landmarks, i.e., *t* and its matching wildcards

D7: SkipTo(205) SkipTo()
D8: SkipTo(Willow) SkipTo()
D9: SkipTo(,) SkipTo()
D10: SkipTo() SkipTo()
D11: SkipTo(Glen) SkipTo()
D12: SkipTo(1) SkipTo()
D13: SkipTo(-) SkipTo()
D14: SkipTo(Phone) SkipTo()

D15: SkipTo(_*Numeric*_) SkipTo()
D16: SkipTo(_*Alphabetic*_) SkipTo()
D17: SkipTo(_*Punctuation*_) SkipTo()
D18: SkipTo(_*HtmlTag*_) SkipTo()
D19: SkipTo(_*Capitalized*_) SkipTo()
D20: SkipTo(_*AlphaNum*_) SkipTo()
D21: SkipTo() SkipTo()

# The final solution

38

- We can see that **D5**, **D10**, **D12**, **D13**, **D14**, **D15**, **D18** and **D21** match correctly with E1 and E3 and fail to match on E2 and E4.
- Using BestDisjunct in Fig. 13, **D5** is selected as the final solution as it has longest last landmark (- <i>).
- **D5** is then returned by **LearnDisjunct()**.
- Since all the examples are covered, LearnRule() returns the disjunctive (start) rule either **D1** or **D5**

**R7:** **either** *SkipTo*( ( )
**or** *SkipTo*(- <i>)

# Example: Extract area codes

30

E1: 205 Willow, Glen, Phone 1-773-366-1987
E2: 25 Oak, Forest, Phone (800) 234-7903
E3: 324 Halsted St., Chicago, Phone 1-800-996-5023
E4: 700 Lake St., Oak Park, Phone: (708) 798-0008

**Fig. 9.** Training examples: four addresses with labeled area codes

# Summary

39

- The algorithm learns by sequential covering
- It is based on landmarks.
- The algorithm is by no mean the only possible algorithm.
- Many variations are possible. There are entirely different algorithms.
- In our discussion, we used only the *SkipTo*() function in extraction rules.
  - *SkipUntil*() is useful too.

# Identifying informative examples

40

- Wrapper learning needs manual labeling of training examples.
- To ensure accurate learning, a large number of training examples are needed.
- Manual labeling labor intensive and time consuming.
- Is it possible to automatically select (unlabelled) examples that are informative for the user to label.
  - Clearly, examples of the same formatting are of limited use.
  - Examples that represent exceptions are informative as they are different from already labeled examples.

## 2nd Approach to Data Extraction:

# Automatic Extraction (Unsupervised Learning)

# Automatic wrapper generation

47

- **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 is possible

48

- 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.

# Two data extraction problems

49

- In Sections 8.1.2 and 8.2.3, we described an abstract model of structured data on the Web (i.e., nested relations), and a HTML mark-up encoding of the data model respectively.
- The general problem of data extraction is to recover the hidden schema from the HTML mark-up encoded data.
- We study two extraction problems, which are really quite similar.

# Problem 1: Extraction given a single list page

50

- **Input**: A single HTML string $S$, which contain $k$ non-overlapping substrings $s_1, s_2, \ldots, s_k$ with each $s_i$ encoding an instance of a set type. That is, each $s_i$ contains a collection $W_i$ of $m_i$ ($\geq 2$) non-overlapping sub-substrings encoding $mi$ instances of a tuple type.
- **Output**: $k$ tuple types $\sigma_1, \sigma_2, \ldots, \sigma_k$, and $k$ collections $C_1, C_2, \ldots, C_k$, of instances of the tuple types such that for each collection $C_i$ there is a HTML encoding function $enc_i$ such that $enc_i$: $C_i \rightarrow W_i$ is a bijection.

*A bijection is **a relation between two sets** such that each element of either set is paired with exactly one element of the other set.*

# Example:

Imagine the HTML string looks like this:

```html
<div class="collection1">
  <p>John</p><p>25</p>
  <p>Alice</p><p>30</p>
</div>

<div class="collection2">
  <p>Product1</p><p>100</p>
  <p>Product2</p><p>200</p>
</div>
```

- The first section ("collection1") contains tuples of people's names and ages (e.g., ("John", 25), ("Alice", 30)).
- The second section ("collection2") contains tuples of product names and prices (e.g., ("Product1", 100), ("Product2", 200)).

For each section:

1. You define a tuple type for each collection. For collection1, the tuple type might look like `("Name", "Age")`, and for collection2, it could be `("ProductName", "Price")`.
2. You create collections ($C_1$, $C_2$) for each section, where $C_1$ would be [("John", 25), ("Alice", 30)] and $C_2$ would be [("Product1", 100), ("Product2", 200)].
3. You then create an encoding function that maps each collection back to its original HTML structure. For example, the encoding function for $C_1$ would map each tuple in $C_1$ to the corresponding HTML elements like `<p>John</p><p>25</p>`.

In essence, the task is to extract data from an HTML string, organize it into structured collections, and then ensure you can map each collection back to its original HTML encoding in a clear, one-to-one way.

# Problem 2: Data extraction given multiple pages

51

- **Input**: A collection $W$ of $k$ HTML strings, which encode $k$ instances of the same type.
- **Output**: A type $\sigma$, and a collection $C$ of instances of type $\sigma$, such that there is a HTML encoding *enc* such that $enc: C \rightarrow W$ is a bijection.

*A bijection is **a relation between two sets** such that each element of either set is paired with exactly one element of the other set.*

## Simplified Example:

Imagine you have 3 HTML pages (W = {page1, page2, page3}):

- **page1**: Contains a person's name and age (e.g., `<p>John</p><p>25</p>`)
- **page2**: Contains a person's name and age (e.g., `<p>Alice</p><p>30</p>`)
- **page3**: Contains a person's name and age (e.g., `<p>Bob</p><p>22</p>`)

From these pages, you want to:

1. **Define a type**: In this case, the type would be something like `("Name", "Age")` because each HTML page represents a person with a name and an age.
2. **Create a collection C**: You would extract the data from the pages and organize it into instances of this type. So, C could look like this:
    - `C = [("John", 25), ("Alice", 30), ("Bob", 22)]`
3. **Create an encoding function**: This function would map each item in C back to its original HTML string. For example, the instance `("John", 25)` would map back to the HTML string `<p>John</p><p>25</p>`.

In essence, the task is to extract the data from several HTML pages, organize it into a structured collection, and then create a way to map the collection back to the original pages, ensuring that each item in the collection corresponds to exactly one HTML string.

**Wrapper maintenance** is the process of updating and adapting **wrappers** used in web data extraction. A **wrapper** is a program or script designed to extract structured information from **semi-structured or unstructured** web pages, such as **news articles, product listings, or reviews.**

Since websites **change frequently** (e.g., layout, structure, or HTML elements), the wrappers must be **maintained and updated** to keep extracting data correctly.

## Why is Wrapper Maintenance Important?

- **Websites change their structure** (HTML tags, layout, class names).
- **Old wrappers may stop working** and extract incorrect or missing data.
- **Maintaining accuracy** in web scraping is essential for **business intelligence, search engines, and data analysis.**

## Challenges in Wrapper Maintenance

1. **Frequent Website Updates** – Sites often change their HTML, breaking existing scrapers.
2. **Anti-Scraping Measures** – Some sites use **CAPTCHAs, IP blocking, or JavaScript obfuscation** to prevent scraping.
3. **Scalability Issues** – Maintaining wrappers for **many websites** can be costly and time-consuming.
4. **Diverse Web Page Structures** – Different websites follow different formats, making standardization hard.

## Types of Wrapper Maintenance Approaches

1. **Manual Maintenance**
   - Developers update the wrapper code **whenever a site changes.**
   - Effective for small-scale projects but **not scalable** for many websites.
2. **Automated Wrapper Adaptation**
   - Uses **machine learning** to detect changes in web pages and adjust the wrapper **automatically.**
   - More efficient but **complex to implement.**
3. **Robust Wrappers**
   - Instead of extracting data from **fixed HTML tags**, they use **patterns, heuristics, or XPath queries** to be **more flexible.**
   - Example: Instead of looking for a product price in `<span class="price">`, the wrapper **identifies patterns** like "$XX.XX".

## Wrapper Maintenance

Once a wrapper is generated, it is applied to other Web pages that contain similar data and are formatted in the same ways as the training examples. This introduces new problems.

1. If the site changes, does the wrapper know the change? This is called the **wrapper verification problem**.
2. If the change is correctly detected, how to automatically repair the wrapper? This is called the **wrapper repair problem**.

One way to deal with both problems is to learn the characteristic patterns of the target items, which are then used to monitor the extraction to check whether the extracted items are correct. If they are incorrect, the same patterns can be used to locate the correct items assuming that the page changes are minor formatting changes. This is called **re-labeling**. After re-labeling, **re-learning** is performed to produce a new wrapper. These two tasks are very difficult because contextual and/or semantic information is often needed to detect changes and to find the new locations of the target items. Wrapper maintenance is still an active research area.

# Instance-Based Wrapper Learning

**Instance-Based Wrapper Learning (IBWL)** is a technique used in **web data extraction** where the wrapper learns by observing **specific examples (instances)** of structured data on a web page. Instead of manually coding rules, IBWL **automatically generalizes patterns** from labeled examples and applies them to extract similar data from new pages.

## How Instance-Based Wrapper Learning Works?

1. **Provide Labeled Examples** – The system is given **sample data** from a webpage, such as **product names, prices, or descriptions.**
2. **Identify Patterns** – The wrapper learns from the **HTML structure, XPath, CSS classes**, or **positioning** of the given examples.
3. **Generalize Extraction Rules** – It creates **generalized patterns** that can extract similar data from other pages of the same website.
4. **Apply to New Pages** – The wrapper uses the learned patterns to extract structured data from **unseen pages.**

## Example: Extracting Product Data from an E-Commerce Website

Imagine a web page listing products with the following **HTML structure:**

```html
<div class="product">
    <h2 class="title">Smartphone X</h2>
    <span class="price">$699</span>
</div>

<div class="product">
    <h2 class="title">Laptop Y</h2>
    <span class="price">$1199</span>
</div>
```

### Step 1: Provide Labeled Examples

We manually label:

- **Product Name:** `"Smartphone X"`, `"Laptop Y"`
- **Price:** `"$699"`, `"$1199"`

### Step 2: Identify Patterns

The wrapper detects that:

- **Product names** appear inside `<h2 class="title">`
- **Prices** appear inside `<span class="price">`

### Step 3: Generalize Extraction Rules

The wrapper learns:

- Extract text inside `h2.title` for product names.
- Extract text inside `span.price` for prices.

### Step 4: Apply to New Pages

For a new product listing:

```html
<div class="product">
    <h2 class="title">Tablet Z</h2>
    <span class="price">$499</span>
</div>
```

The wrapper **automatically extracts:**

- **Product Name:** `"Tablet Z"`
- **Price:** `"$499"`

## Advantages of Instance-Based Wrapper Learning

✅ **No Need for Manual Rule Writing** – Learns from examples instead of predefined rules.

✅ **Handles Structural Variations** – Adapts to similar pages with minor changes.

✅ **More Scalable** – Can generalize to multiple pages of the same website.

## Limitations

⚠️ **Fails on Drastic Page Structure Changes** – If the website layout changes significantly, the wrapper might need retraining.

⚠️ **Needs Enough Examples** – The wrapper requires a few labeled instances to learn patterns accurately.

# String Matching and Tree Matching

# Some useful algorithms



- The key is to finding 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. We describe these techniques.

- In the given text, the focus is on finding an encoding template from a set of encoded instances, which are of the same type, such as HTML encoding strings.

- The key idea is to detect repeated patterns in these strings. Since HTML encoding strings contain nested structures (due to HTML tags), these can be modeled as trees, specifically DOM (Document Object Model) trees.

- Tree matching and string matching are identified as useful techniques to identify patterns in these encoded instances.

- Tree matching is particularly useful because it can handle the nested nature of HTML elements, and both tree and string matching algorithms are employed to find such patterns effectively.

# String edit distance

57

- String edit distance: the most widely used string comparison technique.
- The **edit distance** of two strings, *s1* and *s2*, is defined as the minimum number of *point mutations* required to change *s1* into *s2*, where a point mutation is one of:
  - (1) change a letter,
  - (2) insert a letter, and
  - (3) delete a letter.

# String Edit Distance

String **edit distance** (also known as **Levenshtein distance**) is perhaps the most widely used string matching/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 character, (2) insert a character, and (3) delete a character.

The **Levenshtein distance** (also known as edit distance) is a metric for measuring the difference between two strings. It represents the minimum number of operations required to transform one string into another, where the allowable operations are:

1. **Insertion**: Adding a character to the string.
2. **Deletion**: Removing a character from the string.
3. **Substitution**: Replacing one character in the string with another.

For example, the Levenshtein distance between "kitten" and "sitting" is 3:

- **kitten → sitten** (substitute 'k' with 's')
- **sitten → sittin** (substitute 'e' with 'i')
- **sittin → sitting** (insert 'g' at the end)

The distance is 3 because it takes 3 operations (substitution and insertion).

The Levenshtein distance is often used in applications such as spell checking, DNA sequence analysis, and natural language processing for determining the similarity or difference between strings.

# String edit distance (definition)

58

Assume we are given two strings $s_1$ and $s_2$. The following recurrence relations define the edit distance, $d(s_1, s_2)$, of two strings $s_1$ and $s_2$:

$$d(\varepsilon, \varepsilon) = 0 \qquad // \ \varepsilon \text{ represents an empty string}$$

$$d(s, \varepsilon) = d(\varepsilon, s) = |s| \qquad // \ |s| \text{ is the length of string } s$$

$$d(s_1+ch_1, s_2+ch_2) = \min(d(s_1, s_2) + r(ch_1, ch_2), d(s_1+ch_1, s_2) + 1, d(s_1, s_2+ch_2) + 1)$$

where $ch_1$ and $ch_2$ are the last characters of $s_1$ and $s_2$ respectively, and $r(ch_1, ch_2) = 0$ if $ch_1 = ch_2$; $r(ch_1, ch_2) = 1$, otherwise.

# Dynamic programming

59

We can use a two-dimensional matrix, $m[0..|s_1|,0..|s_2|]$ to hold the edit distances. The low right corner cell $m(|s_1|+1, |s_2|+1)$ will furnish the required value of the edit distance $d(s_1, s_2)$.

$m[0, 0] = 0$

$m[i, 0] = i,\ \ i = 1, 2..., |s_1|$

$m[0, j] = j,\ \ j = 1, 2 ..., |s_2|$

$m[i, j] = \min(m[i\text{-}1, j\text{-}1] + r(s_1[i], s_2[j]), m[i\text{-}1, j] + 1, m[i, j\text{-}1] + 1\ ),$

where $i = 1, 2..., |s_1|, j = 1, 2..., |s_2|$, and $r(s_1[i], s_2[j]) = 0$ if $s_1[i] = s_2[j]$; $r(s_1[i], s_2[j]) = 1$, otherwise.

**Example 1:** We want to compute the edit distance and find the alignment of the following two strings:

$s_1$: X G Y X Y X Y X
$s_2$: X Y X Y X Y T X

The edit distance matrix is given in Fig. 9.17. The final edit distance value is 2, which is the value in the bottom right corner cell. Fig. 9.17 also shows the trace back path. Notice that a diagonal line means match or change, a vertical line means insertion, and a horizontal line means deletion. Thus, the final alignment of our two strings is:

$s_1$: X G Y X Y X Y – X
$s_2$: X – Y X Y X Y T X

| | $s_1$ | X | G | Y | X | Y | X | Y | X |
|---|---|---|---|---|---|---|---|---|---|
| $s_2$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| X | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Y | 2 | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| X | 3 | 2 | 2 | 2 | 1 | 2 | 3 | 4 | 5 |
| Y | 4 | 3 | 3 | 2 | 2 | 1 | 2 | 3 | 4 |
| X | 5 | 4 | 4 | 3 | 2 | 2 | 1 | 2 | 3 |
| Y | 6 | 5 | 5 | 4 | 3 | 2 | 2 | 1 | 2 |
| T | 7 | 6 | 6 | 5 | 4 | 3 | 3 | 2 | 2 |
| X | 8 | 7 | 7 | 6 | 5 | 4 | 3 | 3 | 2 |

**Steps to Fill the Matrix**

1. **Initialize the first row and first column:**
   - The first row represents converting an empty string to $s_1$ by inserting characters.
   - The first column represents converting an empty string to $s_2$ by inserting characters.
2. **Fill each cell (i, j) using the recurrence relation:**
   - If **characters match** ($s_1[j] = s_2[i]$), no edit is needed:

     $$\text{cost} = \text{value from the diagonal (i-1, j-1)}$$

   - If **characters differ**, consider:
     - **Substitution** (diagonal cell + 1)
     - **Insertion** (left cell + 1)
     - **Deletion** (above cell + 1)
     - Take the **minimum of these three values.**

**Fig. 9.17.** The edit distance matrix and back trace path

# Tree Edit Distance

61

- 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, and
  - node replacement.

A cost is assigned to each of the operations.

# Tree Edit Distance/Tree Matching

## What is Tree Edit Distance?

Tree edit distance measures the minimum number of operations (insertions, deletions, or substitutions) required to transform one tree into another. Since trees are hierarchical structures (unlike strings which are linear), the operations involved are slightly different:

1. **Insertion**: Adding a new subtree into the tree.
2. **Deletion**: Removing a subtree from the tree.
3. **Substitution**: Replacing one subtree with another.

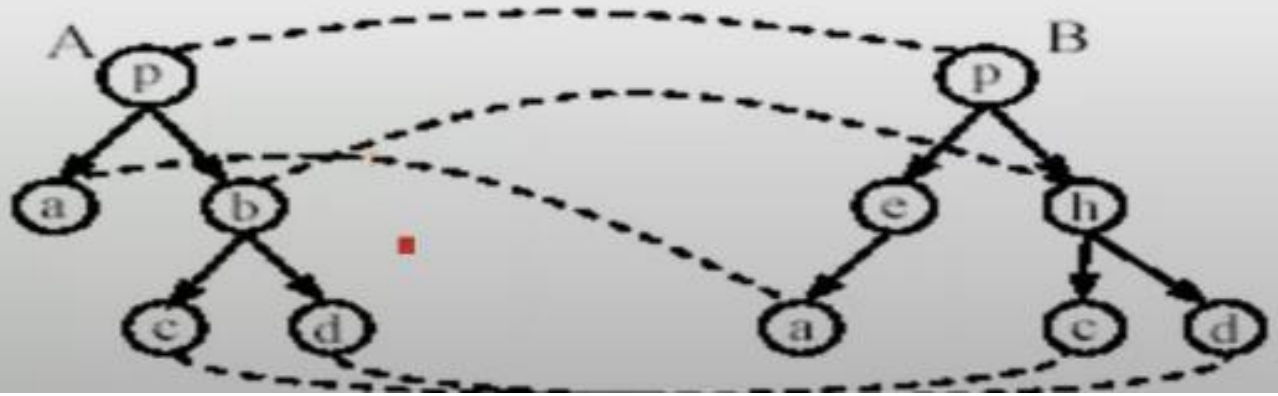
# Definition

62

Let $X$ be a tree and let $X[i]$ be the $i$th node of tree $X$ in a preorder walk of the tree. A *mapping* $M$ between a tree $A$ of size $n_1$ and a tree $B$ of size $n_2$ is a set of ordered pairs $(i, j)$, one from each tree, satisfying the following conditions for all $(i_1, j_1), (i_2, j_2) \in M$:

(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 $B[j_2]$;
(3) $A[i_1]$ is an ancestor of $A[i_2]$ *iff* $B[j_1]$ is an ancestor of $B[j_2]$.

Intuitively, the definition requires that each node appears no more than once in a mapping and the order among siblings and the hierarchical relation among nodes are both preserved. Fig. 16 shows a mapping example.



# Simple tree matching

63

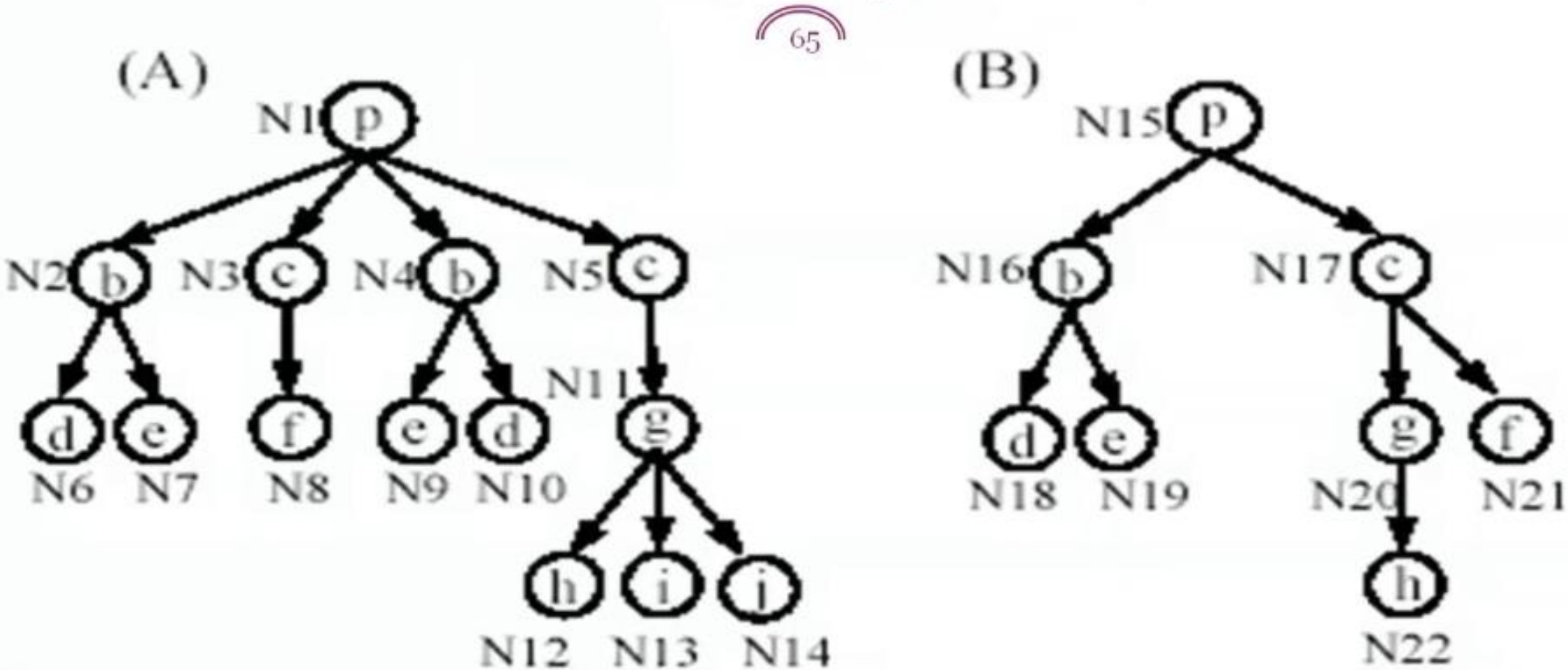
- In the general setting,
  - mapping can cross levels, e.g., node $a$ in tree $A$ and node $a$ in tree $B$.
  - Replacements are also allowed, e.g., node $b$ in $A$ and node $h$ in $B$.
- We describe a restricted matching algorithm, called **simple tree matching** (STM), which has been shown quite effective for Web data extraction.
  - 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.

# Simple Tree Matching algo



```
Algorithm: STM(A, B)
1.   if the roots of the two trees A and B contain distinct symbols then
2.       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, ..., m;
                              M[0, j] := 0 for j = 0, ..., n;
6.         for i = 1 to m do
7.             for j = 1 to n do
8.                 M[i,j]:=max(M[i,j-1], M[i-1, j], M[i-1, j-1]+W[i, j]);
                       where W[i,j] = STM(A_i, B_j)
9.             end-for
10.        end-for
11.        return (M[m, n]+1)
12.  end-if
```

# An example


65
(A)
N1 p
N2 b
N3 c
N4 b
N5 c
N11
d
e
f
e
d
g
N6 N7 N8 N9 N10
h
i
j
N12 N13 N14
(B)
N15 p
N16 b
N17 c
d
e
g
f
N18 N19
N20 N21
h
N22

# Tree Edit Distance/Tree Matching

## Example of Tree Edit Distance

Let's use the following two **rooted trees** as an example.

**Tree 1:**

mathematica


**Tree 2:**

mathematica


We want to calculate the **tree edit distance** between these two trees.

### Step 1: Identify Differences

- **Root node**: Both trees have the root node `A`, so no operation is needed here.
- **Left child of A**:
  - Tree 1 has `B`, while Tree 2 has `C`. This is a **substitution**.
  - Additionally, Tree 1's `B` has children `D` and `E`, but Tree 2's `C` has children `E` and `F`, so further changes are needed.

# Tree Edit Distance/Tree Matching

## Example of Tree Edit Distance

Let's use the following two **rooted trees** as an example.

**Tree 1:**

mathematica


**Tree 2:**

mathematica


We want to calculate the **tree edit distance** between these two trees.

### Step 2: Apply Edit Operations

Now, let's break down the operations:

- The **left child of A** in Tree 1 (node `B`) is replaced with node `C` in Tree 2 (substitution).
- Subtree under `B` in Tree 1 (`D` and `E`) needs to be replaced with subtree under `C` in Tree 2 (`E` and `F`).
  - The subtree `D` is deleted.
  - The subtree `F` is inserted.
- The **right child of A** is the same (`C` in Tree 1 vs `D` in Tree 2), so no changes needed there.

# Tree Edit Distance/Tree Matching

## Example of Tree Edit Distance

Let's use the following two **rooted trees** as an example.

**Tree 1:**

mathematica


**Tree 2:**

mathematica


We want to calculate the **tree edit distance** between these two trees.

### Step 3: Resulting Operations

- **Substitution**: Replace node `B` with node `C`.
- **Deletion**: Remove node `D` under `B`.
- **Insertion**: Add node `F` under `C`.

So, the **tree edit distance** between these two trees is **3 operations** (1 substitution, 1 deletion, and 1 insertion).

Just like **string edit distance** measures how different two strings are based on character operations, **tree edit distance** measures how different two trees are based on tree operations (insertions, deletions, and substitutions). The concept can be applied in various domains, such as comparing hierarchical structures like XML documents or syntactic trees in natural language processing.

# Multiple alignment

67

- Pairwise alignment is not sufficient because a web page usually contain more than one data records.
- We need multiple alignment.
- We discuss two techniques
  - Center Star method
  - Partial tree alignment.

# Center star method

68

- This is a classic technique, and quite simple. It commonly used for multiple string alignments, but can be adopted for trees.
- Let the set of strings to be aligned be $S$. In the method, a string $s_c$ that minimizes,

$$\sum_{s_i \in S} d(s_c, s_i) \qquad (3)$$

- is first selected as the **center string**. $d(sc, si)$ is the distance of two strings.
- The algorithm then iteratively computes the alignment of rest of the strings with $sc$.

# The algorithm


**CenterStar**(S)

1. choose the center star $s_c$ using Equation (3);
2. initialize the multiple sequence alignment $M$ that contains only $s_c$;
4. **for** each $s$ in $S$-$\{s_c\}$ **do**
5. let $c^*$ be the aligned version of $s_c$ in $M$;
6. let $s'$ and $c^{*\prime}$ be the optimally aligned strings of $s$ and $c^*$;
7. add aligned strings $s'$ and $c^{*\prime}$ into the multiple alignment $M$;
8. add spaces to each string in $M$, except, $s'$ and $c^{*\prime}$, at locations where new spaces are added to $c^*$
9. **endfor**
10. return multiple string alignment $M$

# An example



**Example 2**: We have three strings, i.e., $S$ = {ABC, XBC, XAB}. ABC is selected as the center string $s_c$. Let us align the other strings with ABC.

Iteration 1: Align $c^*$ ($= s_c$) with $s$ =XBC:

```
c*':        A  B  C
            |  |
s':         X  B  C
```

Update $M$: A B C → A B C
X B C

Iteration 2: Align $c^*$ with $s$ = XAB:

```
c*':        -  A  B  C
               |  |
s':         X  A  B  -
```

Update $M$: A B C → − A B C
X B C − X B C
X A B −

# The shortcomings

- Assume there are $k$ strings in $S$ and all strings have length $n$, finding the center takes $O(k^2n^2)$ time and the iterative pair-wise alignment takes $O(kn^2)$ time. Thus, the overall time complexity is $O(k^2n^2)$.

For our data extraction task, this method has two shortcomings:

1. the algorithm runs slowly for pages containing many data records and/or data records containing many tags (i.e., long strings) because finding the center string needs $O(k^2n^2)$ time.
2. if the center string (or tree) does not have a particular data item, other data records that contain the same data item may not be aligned properly. For example, the letter X's in the last two strings (in bold) are not aligned in the final result, but they should.

# Shortcomings (cont ...)

- Giving the cost of 1 for "changing a letter" in edit distance is problematic (e.g., A and X in the first and second strings in the final result) because of optional data items in data records.
- The problem can be partially dealt with by disallowing "changing a letter" (e.g., giving it a larger cost). However, this introduces another problem.
- For example, if we align only ABC and XBC, it is not clear which of the following alignment is better.

(1)
A – B C
– X B C

(2)
– A B C
X – B C

# The partial tree alignment method

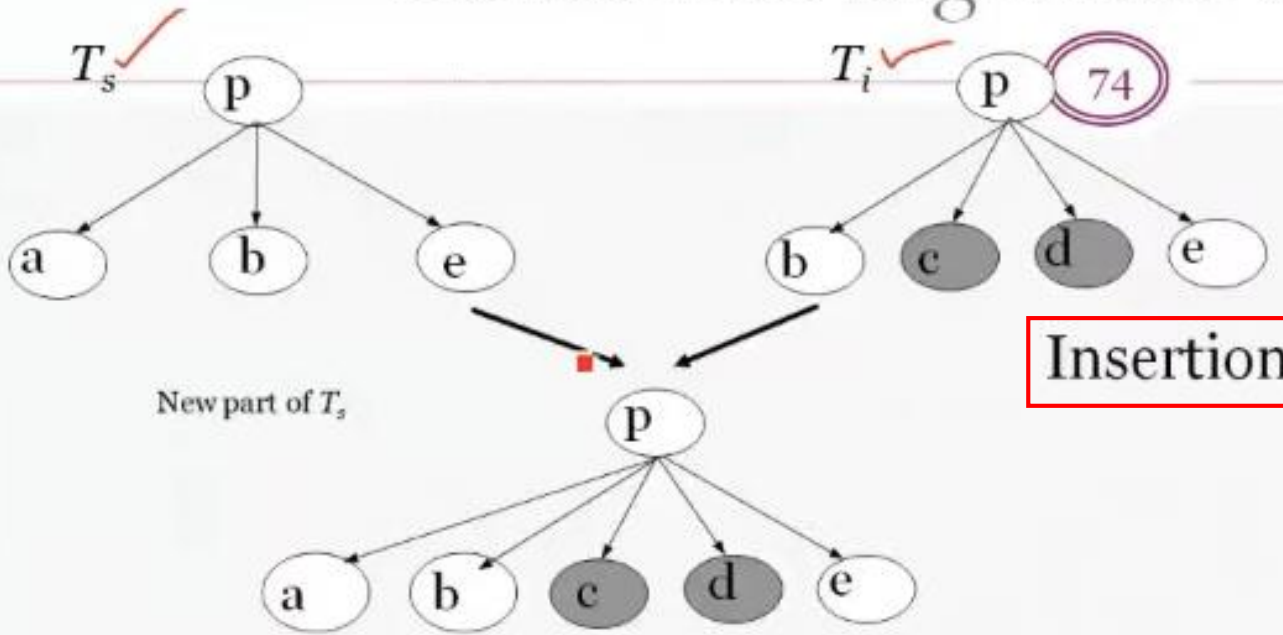


- Choose a seed tree: A seed tree, denoted by $T_s$, is picked with the maximum number of data items.
- The seed tree is similar to center string, but without the $O(k^2n^2)$ pair-wise tree matching to choose it.
- Tree matching:

  For each unmatched tree $T_i$ ($i \neq s$),
  - match $T_s$ and $T_i$.
  - Each pair of matched nodes are linked (aligned).
  - For each unmatched node $n_j$ in $T_i$ do
    - expand $T_s$ by inserting $n_j$ into $T_s$ if a position for insertion can be uniquely determined in $T_s$.

  The expanded seed tree $T_s$ is then used in subsequent matching.

# Partial tree alignment of two trees

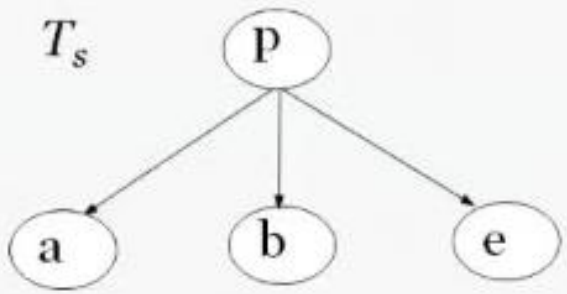


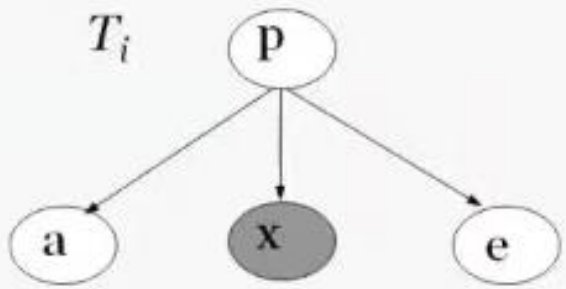




Insertion is possible

Insertion is not possible



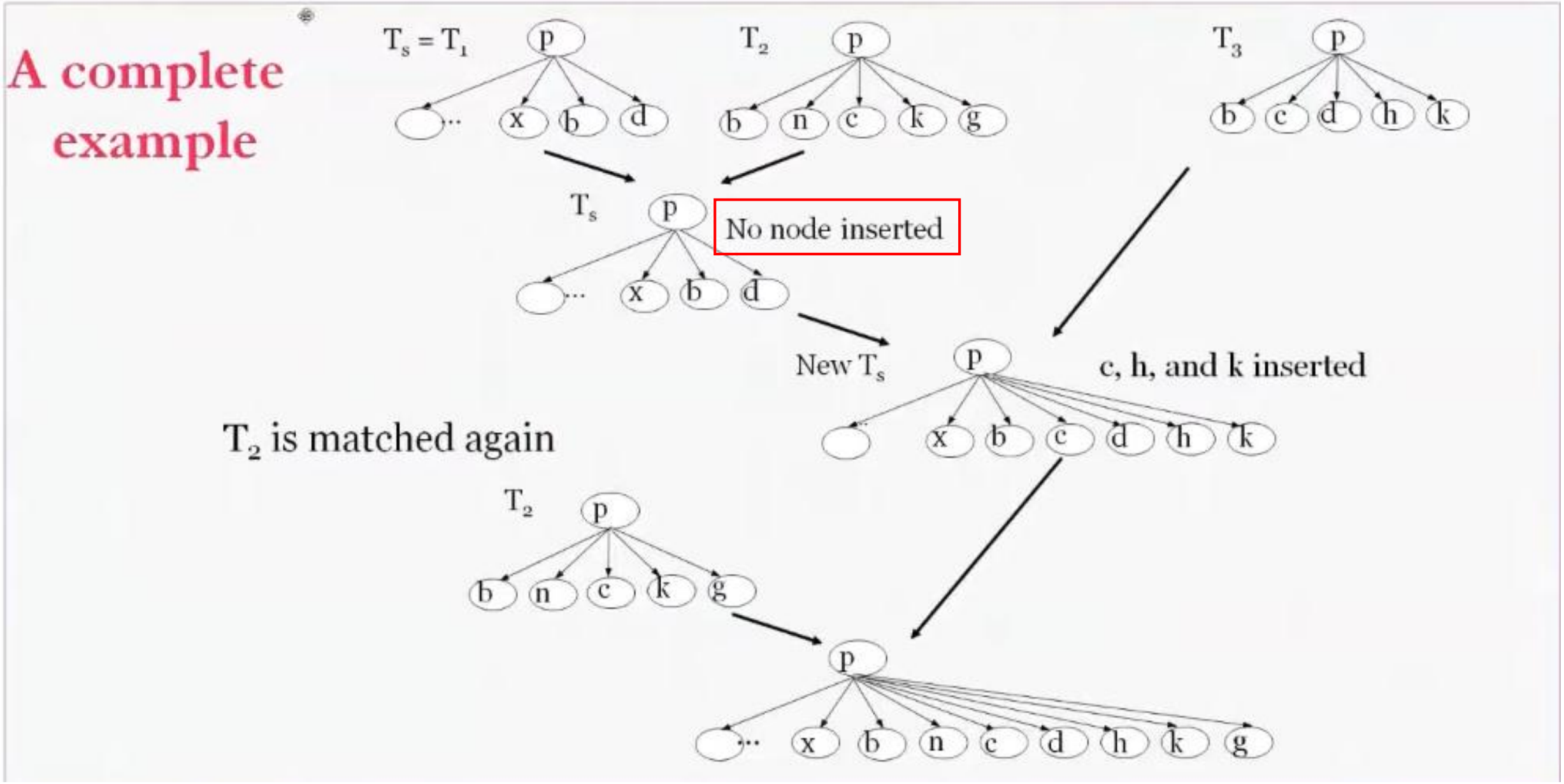
# Partial alignment of two trees

75

```
Algorithm PartialTreeAlignment(S)
1.  Sort trees in S in descending order of the number of unaligned data items;
2.  T_s ← the first tree (which is the largest) and delete it from S;
3.  R ← ∅;
4.  while (S ≠ ∅) do
5.     T_i ← select and delete next tree from S;     // follow the sorted order
6.     STM(T_s, T_i);                                // tree matching
7.     AlignTrees(T_s, T_i);                         // based on the result from line 6
8.     if T_i is not completely aligned with T_s then
9.        if InsertIntoSeed(T_s, T_i) then           // True: some insertions are done
10.          S = S ∪ R;
11.          R ← ∅
12.       endif;
13.       if there are still unaligned items in T_i that are not inserted into T_s then
14.          R ← R ∪ {T_i}
15.       endif;
16.    endif;
17. endwhile;
18. Output data fields from each T_i to a data table based on the alignment results.
```

**Fig. 21.** The partial tree alignment algorithm

# A complete example



# Output Data Table

| | … | x | b | n | c | d | h | k | g |
|---|---|---|---|---|---|---|---|---|---|
| $T_1$ | … | 1 | 1 | | | 1 | | | |
| $T_2$ | | | 1 | 1 | 1 | | | 1 | 1 |
| $T_3$ | | | 1 | | 1 | 1 | 1 | 1 | |

# Building DOM Trees

# Building DOM trees

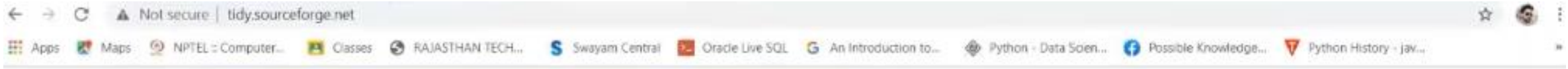
79

- We now start to talk about actual data extraction.
- The usual first step is to build a DOM tree (tag tree) of a HTML page.
  - Most HTML tags work in pairs. Within each corresponding tag-pair, there can be other pairs of tags, resulting in a nested structure.
  - Building a DOM tree from a page using its HTML code is thus natural.
- In the tree, each pair of tags is a **node**, and the nested tags within it are the **children** of the node.

# Two steps to build a tree



- **HTML code cleaning:**
  - Some tags do not require closing tags (e.g., <li>, <hr> and <p>) although they have closing tags.
  - Additional closing tags need to be inserted to ensure all tags are balanced.
  - Ill-formatted tags need to be fixed. One popular program is called **Tidy**, which can be downloaded from http://tidy.sourceforge.net/.
- **Tree building:** simply follow the nested blocks of the HTML tags in the page to build the DOM tree. It is straightforward.




## HTML Tidy Legacy Website

Welcome to the HTML Tidy Legacy Website! We at HTACG are trying hard to keep this site up to date, but you will certainly find newer information about HTML Tidy at http://www.html-tidy.org and on our newer Github repository.

## HTML Tidy Project

At the current stage of HTML Tidy's long history, current maintenance and development is provided by HTACG, which is fortunate to count among its members some of the very earliest contributors to HTML Tidy.

We continue to have two primary goals. First, to provide a home where all the patches and fixes that folks contribute can be collected and incorporated into the program. Second, a library form of Tidy has been created to make it easier to incorporate Tidy into other software.

### Table of Contents



### Get Tidy

The latest version of Tidy is always available as source code in our repository, and we try to make as many binaries available as possible on our binaries website.

Also consider checking your package manager for a recent version of Tidy!

### News

**4 September 2015**

HTACG are proud to announce the first major release of HTML Tidy in years. Tidy version 5.0.0 is stable, ready for mass adoption, and finally *officially* supports modern HTML5.

Version 5.0.0 source, or check to see if there's a binary for your OS.

SOURCEFORGE

**Tidy User Links**

These links are available to anyone to submit a bug report or to view the Tidy mailing list archives.

Bug Reports / Feature Requests
User List Archives
Developer List Archives

**Source Forge Projects**

HTML Tidy
mod_tidy for Apache2

**Accessibility**

Tools and Resources:
A-Prompt
Specifications:
Section 508
W3C WAI

**LibTidy Applications**

Active:
Balthisar Tidy - for Mac OS X
Out of Date:
Tidy UI - basic GUI for Win32
wxTidy - editor based on wxWindows
validator using PHP - by Nuno Lopes

**LibTidy Bindings**

By Charles Reitzel
C++
Perl
COM

# Building tree using tags & visual cues



- Correcting errors in HTML can be hard.
- There are also dynamically generated pages with scripts.
- Visual information comes to the rescue.
- As long as a browser can render a page correct, a tree can be built correctly.
  - Each HTML element is rendered as a rectangle.
  - Containments of rectangles representing nesting.

# An example



```
1  <table>
2      <tr>
3          <td>…</td>
4          <td>…</td>
5      </tr>
6      <tr>
7          <td>…</td>
8          <td>…</td>
9      </tr>
10 </table>
```

| left | right | top | bottom |
|---|---|---|---|
| 100 | 300 | 200 | 400 |
| 100 | 300 | 200 | 300 |
| 100 | 200 | 200 | 300 |
| 200 | 300 | 200 | 300 |
| 100 | 300 | 300 | 400 |
| 100 | 200 | 300 | 400 |
| 200 | 300 | 300 | 400 |

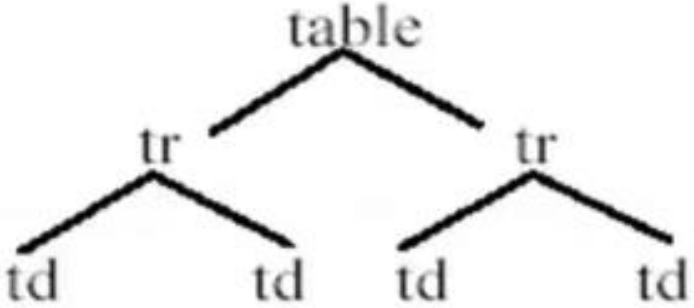



**Fig. 23.** A HTML code segment, boundary coordinates and the resulting tree

A **DOM (Document Object Model)** tree represents the structure of an HTML or XML document as a hierarchical tree of nodes. Each node in the tree corresponds to a part of the document, such as an element, attribute, or text content.

When we **build a DOM tree**, we essentially parse the document (HTML or XML) and create a tree-like structure that can be easily accessed and manipulated programmatically.

**Key Points About the DOM Tree:**

1. **Node Types**: The DOM tree consists of different types of nodes:
   - **Element nodes**: These represent HTML or XML elements (e.g., `<div>`, `<p>`).
   - **Text nodes**: These represent the text inside elements.
   - **Attribute nodes**: These represent attributes of elements (e.g., `id`, `class`).
   - **Document nodes**: The root of the DOM tree represents the entire document.
2. **Hierarchy**: The DOM tree has a parent-child relationship. For example, a `<div>` element can have multiple child elements like `<p>` or `<a>`.
3. **Access**: You can traverse and manipulate the DOM tree using various programming languages like JavaScript, Python, etc. Most commonly, DOM trees are built and accessed using JavaScript in web browsers.

# Building a DOM Tree (HTML Example)

Let's take a simple **HTML document** as an example:

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <div id="main">
      <h1>Welcome to My Page</h1>
      <p>This is an example paragraph.</p>
    </div>
  </body>
</html>
```

**Step-by-Step Breakdown:**

1. **Start with the document**: The root of the DOM tree is the entire document ( `<!DOCTYPE html>` is ignored, but it's part of the document).
2. **Parse the `<html>` tag**: The `<html>` element is the root element in the DOM tree, containing the `<head>` and `<body>` as child nodes.
3. **Parse the `<head>` tag**: Inside the `<html>` element, the `<head>` contains metadata like the `<title>` tag.
4. **Parse the `<body>` tag**: The `<body>` contains the visible content, like the `<div>` tag.
5. **Parse the `<div>` tag**: The `<div>` has an `id` attribute ("main") and contains two child elements, `<h1>` and `<p>`.
6. **Parse the `<h1>` tag**: The `<h1>` tag contains the text "Welcome to My Page."
7. **Parse the `<p>` tag**: The `<p>` tag contains the text "This is an example paragraph."

## DOM Tree Structure for the Example:

bash

Copy Edit

```
Document
 └── html
      ├── head
      │    └── title
      │         └── "Example"
      └── body
           └── div (id="main")
                ├── h1
                │    └── "Welcome to My Page"
                └── p
                     └── "This is an example paragraph."
```

### Explanation of the DOM Tree Structure:

1. **Root Node**: The root node is the entire document (`Document`), which contains the `<html>` tag.
2. **Element Nodes**: Each HTML tag (like `<html>`, `<head>`, `<body>`, `<div>`, `<h1>`, and `<p>`) corresponds to an element node.
3. **Text Nodes**: The text inside the `<title>`, `<h1>`, and `<p>` tags are text nodes.
4. **Attributes**: The `id="main"` on the `<div>` element is an attribute node.

html

Copy Edit

```html
<!DOCTYPE html>
<html>
<head>
    <title>Sample Page</title>
</head>
<body>
    <h1>Welcome</h1>
    <div>
        <p>This is a paragraph.</p>
        <ul>
            <li>Item 1</li>
            <li>Item 2</li>
        </ul>
    </div>
</body>
</html>
```

**Breakdown of the DOM Tree:**

1. The **root node** is `[Document]`, which represents the entire HTML document.
2. `<html>` is the **first child** of `[Document]` and contains:
   - `<head>` (with a `<title>` element inside)
   - `<body>` (containing `<h1>`, `<div>`, and `<ul>`)
3. `<div>` has:
   - A `<p>` element inside it.
   - A `<ul>` (unordered list) with two `<li>` items inside.

# Step 2: Corresponding DOM Tree Representation

Below is the **hierarchical structure** of the HTML document:

php-template

Copy Edit

```
[Document]
 ├── <html>
 │    ├── <head>
 │    │    ├── <title> Sample Page </title>
 │    ├── <body>
 │         ├── <h1> Welcome </h1>
 │         ├── <div>
 │         │    ├── <p> This is a paragraph. </p>
 │         │    ├── <ul>
 │         │         ├── <li> Item 1 </li>
 │         │         ├── <li> Item 2 </li>
```

## Use Cases for DOM Trees:

1. **Web Scraping**: When scraping HTML pages, you often need to build a DOM tree to parse and extract information.
2. **Dynamic Web Pages**: In JavaScript, the DOM tree is manipulated to update content dynamically on a webpage (e.g., adding/removing elements, modifying styles).
3. **Document Manipulation**: In applications like word processors or code editors, the DOM tree is used to structure and manipulate documents.
4. **Event Handling**: DOM trees are also crucial for handling events (like clicks, form submissions, etc.), as each element can have event listeners attached to it.

# Extraction Given a List Page: Flat Data Records

84

- Given a single list page with multiple data records,
  - Automatically segment data records
  - Extract data from data records.
- Since the data records are flat (no nested lists), string similarity or tree matching can be used to find similar structures.
  - Computation is a problem
  - A data record can start anywhere and end anywhere

**Example:**

Imagine you visit an e-commerce website like Amazon. You see a list of products, each with an image, title, price, and rating. These product listings appear in a **grid or list format**—this section is a **data region** because all items follow the same pattern.

# Two important observations

85

- **Observation 1**: A group of data records that contains descriptions of a set of similar objects are typically presented in a contiguous region of a page and are formatted using similar HTML tags. Such a region is called a **data region**.
- **Observation 2**: A set of data records are formed by some child sub-trees of the same parent node.

**Example:**

Each product listing on Amazon is structured as an **individual block** inside a larger container (like a `<div>` in HTML). All product listings belong to the same **parent div**, and each product's information (title, price, image) is a **child element** of that parent.

# An example

86

1. 

Customer Rating:

**Apple iBook Notebook M8600LL/A (600-MHz PowerPC G3, 128 MB RAM, 20 GB hard drive)**

Buy new: **$1,194.00**

Usually ships in 1 to 2 days

| Best use: (what's this?) | Business: | Portability: | Desktop Replacement: | Entertainment: |
|---|---|---|---|---|
| | ●●●○○ | ●●●●● | ●●●○○ | ●●●○○ |

600 MHz PowerPC G3, 128 MB SRAM, 20 GB Hard Disk, 24x CD-ROM, AirPort ready, and Mac OS X, Mac OS X,Mac OS 9.2,Quick Time,iPhoto,iTunes 2,iMovie 2,AppleWorks,Microsoft IE

2. 

Customer Rating:

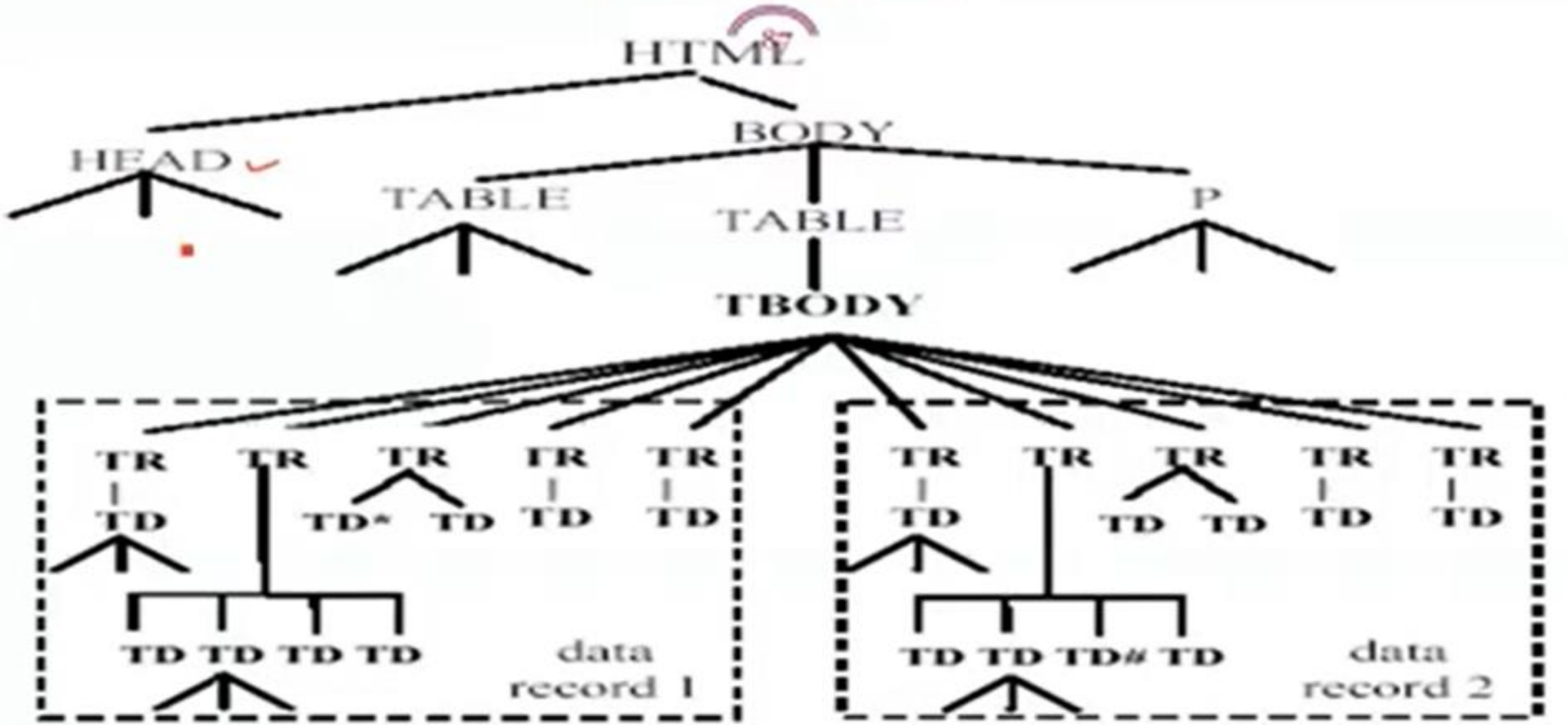
**Apple Powerbook Notebook M8591LL/A (667-MHz PowerPC G4, 256 MB RAM, 30 GB hard drive)**

Buy new: **$2,399.99**

| Best use: (what's this?) | Portability: | Desktop Replacement: | Entertainment: |
|---|---|---|---|
| | ●●●●○ | ●●●●○ | ●●●●○ |

667 MHz PowerPC G4, 256 MB SDRAM, 30 GB Ultra ATA Hard Disk, 24x (read), 8x (write) CD-RW, 8x; included via combo drive DVD-ROM, and Mac OS X, QuickTime, iMovie 2, iTunes(6), Microsoft Internet Explorer, Microsoft Outlook Express, ...

# The DOM tree


HTML
HEAD
BODY
TABLE
TABLE
P
TBODY
TR
TD
data
record 1
data
record 2

# The Approach

88

Given a page, three steps:

- **Building the HTML Tag Tree**
  - Erroneous tags, unbalanced tags, etc
- **Mining Data Regions**
  - Spring matching or tree matching
- **Identifying Data Records**

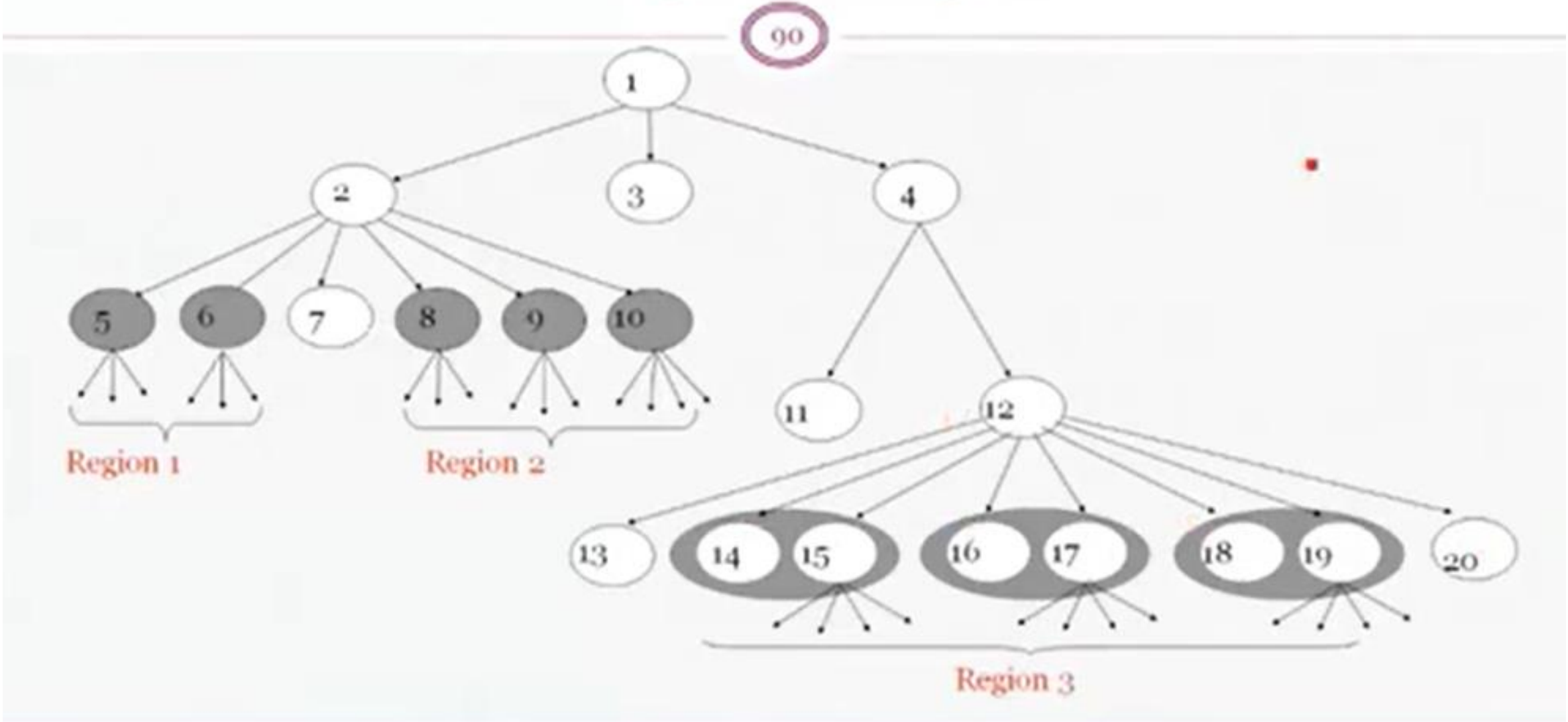
Rendering (or visual) information is very useful in the whole process

# Mining a set of similar structures

89

- **Definition:** A *generalized node* (a *node combination*) of length $r$ consists of $r$ ($r \geq 1$) nodes in the tag tree with the following two properties:
  - the nodes all have the same parent.
  - the nodes are adjacent.
- **Definition:** A *data region* is a collection of two or more generalized nodes with the following properties:
  - the generalized nodes all have the same parent.
  - the generalized nodes all have the same length.
  - the generalized nodes are all adjacent.
  - the similarity between adjacent generalized nodes is greater than a fixed threshold.

# Mining Data Regions



# Mining data regions

91

- We need to find where each generalized node starts and where it ends.
  - perform string or tree matching
- Computation is not a problem anymore
  - Due to the two observations, we only need to perform comparisons among the children nodes of a parent node.
  - Some comparisons done for earlier nodes are the same as for later nodes (see the example below).

# Comparison

92

We use Fig. 27 to illustrate the comparison process. Fig. 27 has 10 nodes below a parent node *p*. We start from each node and perform string (or tree) comparison of all possible combinations of component nodes. Let the maximum number of components that a generalized node can have be 3 in this example.

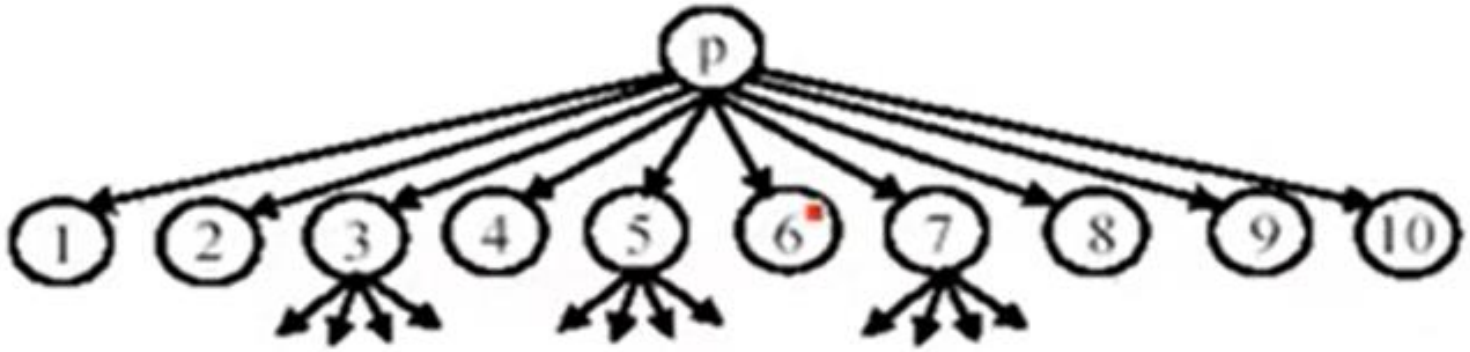



**Fig. 27.** Combination and comparison

# Comparison (cont ...)

93

Start from node 1: We compute the following string comparisons.

- (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10)
- (1-2, 3-4), (3-4, 5-6), (5-6, 7-8), (7-8, 9-10)
- (1-2-3, 4-5-6), (4-5-6, 7-8-9)

(1, 2) means that the tag string of node 1 is compared with the tag string of node 2. The tag string of a node includes all the tags of the sub-tree of the node. (1-2, 3-4) means that the combined tag string of nodes 1 and 2 is compared with the combined tag string of nodes 3 and 4.

Start from node 2: We only compute:

- (2-3, 4-5), (4-5, 6-7), (6-7, 8-9)
- (2-3-4, 5-6-7), (5-6-7, 8-9-10)

# The MDR algorithm

94

```
Algorithm MDR(Node, K, T)
1  if TreeDepth(Node) >= 3 then
2      CombComp(Node.Children, K);
3      DataRegions ← IdenDRs(Node, K, T);
4      if (UncoveredNodes ← Node.Children − ⋃_{DR∈DataRegions} DR ) ≠ ∅ then
5          for each ChildNode ∈ UncoveredNodes do
6              DataRegions ← DataRegions ∪ MDR(ChildNode, K, T);
7      return DataRegions
8  else return ∅
```

**Fig. 28.** The overall algorithm

## Understanding the MDR Algorithm from the Given Pseudocode

The given MDR algorithm works recursively to **identify and extract data regions** from a **tree-like structure** (e.g., a DOM tree in web pages). Here's a step-by-step breakdown:

1. **Check Depth of Node** ( `TreeDepth(Node) >= 3` )
   - The algorithm processes only nodes that are sufficiently deep in the tree structure (depth ≥ 3).
   - This avoids processing nodes that are too high in the hierarchy (like `<html>`, `<body>`).
2. **Combine Similar Components** ( `CombComp(Node.Children, K)` )
   - It groups or combines similar child nodes of the current node using a similarity measure (parameter `K`).
   - This step helps in identifying repetitive structures, such as product listings on an e-commerce page.
3. **Identify Data Regions** ( `IdenDRs(Node, K, T)` )
   - Extracts potential data regions based on structure and similarity.
   - This function finds meaningful data groups within the node.

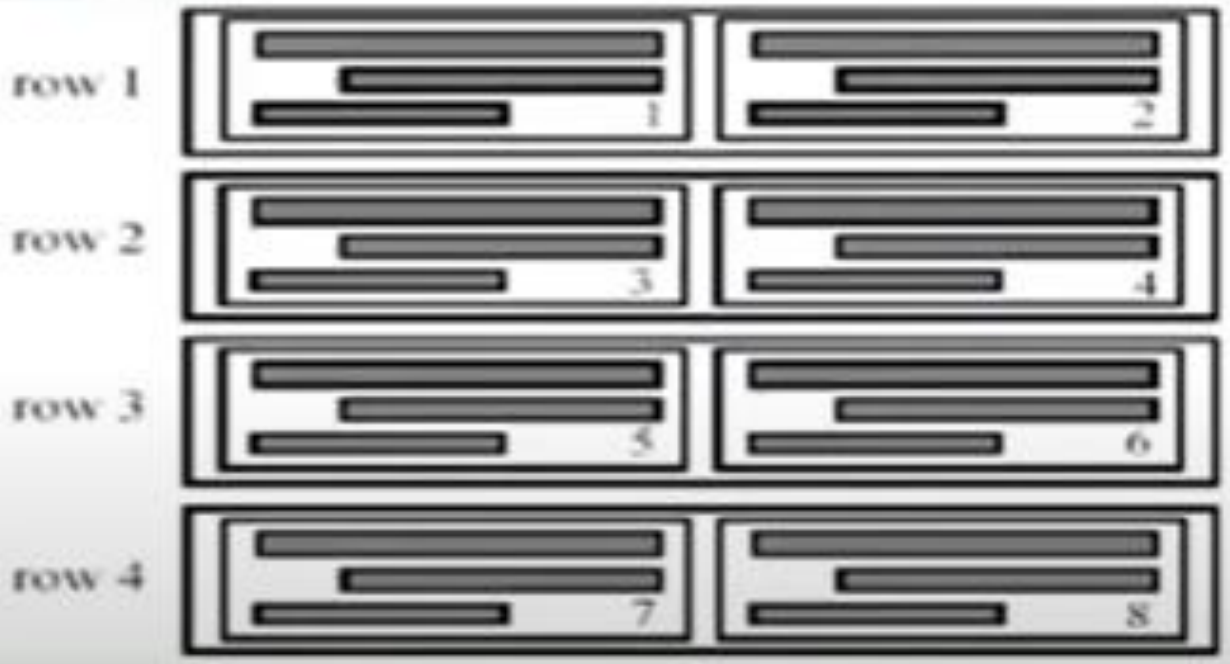
4. **Find Uncovered Nodes** ( `UncoveredNodes` )
    - After extracting identified data regions, some child nodes may still remain uncovered.
    - These are nodes that are not part of any identified data region.
5. **Recursive Processing of Uncovered Nodes** ( `for each ChildNode ∈ UncoveredNodes do` )
    - The algorithm continues recursively on uncovered nodes to find more data regions.
    - This ensures deeper hierarchical structures are explored.
6. **Return Data Regions** ( `return DataRegions` )
    - The extracted structured regions are returned as the final output.
7. **Base Case** ( `else return ∅` )
    - If the conditions are not met, the function returns an empty result.

## Applications of MDR Algorithm

- **Web Data Extraction** – Extracts structured information (e.g., product listings, news articles) from web pages.
- **Web Scraping** – Helps identify meaningful sections in an HTML document for automated data collection.
- **Search Engine Optimization (SEO)** – Identifies key content areas to improve search ranking.
- **Information Retrieval** – Helps in organizing and structuring web content for efficient querying.

# Find data records from generalized nodes

- A generalized node may not represent a data record.
- In the example on the right, each row is found as a generalized node.
- This step needs to identify each of the 8 data record.
  - Not hard
  - We simply run the MDR algorithm given each generalized node as input
- There are some complications (read the notes)



## Example Scenario:

Imagine a webpage displaying a list of products. Each **row (generalized node)** contains **two product records** inside it.

## HTML Code Example:

html Copy Edit

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Generalized Nodes and Data Records</title>
    <style>
        .row {
            display: flex;
            border: 2px solid black;
            padding: 10px;
            margin: 10px;
        }
        .record {
            flex: 1;
            border: 1px solid gray;
            padding: 10px;
            margin: 5px;
        }
    </style>
</head>
<body>
    <h2>Product List</h2>

    <!-- Row 1: Generalized Node -->
    <div class="row">
        <div class="record">Product 1: Laptop - $1000</div>
        <div class="record">Product 2: Smartphone - $800</div>
    </div>

    <!-- Row 2: Generalized Node -->
    <div class="row">
        <div class="record">Product 3: Tablet - $500</div>
        <div class="record">Product 4: Smartwatch - $200</div>
    </div>

    <!-- Row 3: Generalized Node -->
    <div class="row">
        <div class="record">Product 5: Headphones - $100</div>
        <div class="record">Product 6: Keyboard - $50</div>
    </div>

    <!-- Row 4: Generalized Node -->
    <div class="row">
        <div class="record">Product 7: Mouse - $30</div>
        <div class="record">Product 8: USB Drive - $20</div>
    </div>

</body>
</html>
```

## Explanation of the Code:

1. Each `.row` represents a generalized node (like a container that holds multiple items).
2. Each `.record` inside a row is an individual data record (product information).
3. The webpage shows 4 rows, each containing 2 product records (just like the diagram in your image).

## How does this relate to the image?

- The rows in the image are similar to the `<div class="row">` containers.
- The individual numbered records in the image (1-8) match the `<div class="record">` elements inside each row.
- The goal is to extract each product separately instead of treating the entire row as one item.

# Extraction Given a List Page: Nested Data Records

99

- We now deal with the most general case
  - Nested data records
- Problem with the previous method
  - not suitable for nested data records, i.e., data records containing nested lists.
  - Since the number of elements in the list of each data record can be different, using a fixed threshold to determine the similarity of data records will not work.

## 1. Understanding Nested Data Records

- A nested data record is a structured piece of data within a web page that contains sub-records or nested lists.
- For example, on an e-commerce website, a product listing may contain multiple attributes, including a list of reviews, specifications, or subcategories.
- These nested structures complicate data extraction because traditional methods assume a flat structure where each data record has a fixed format.

## 2. Challenges with Previous Methods

- Not Suitable for Nested Lists

- Traditional web data extraction techniques **struggle with nested data records** because they are designed for **simple, repetitive structures.**
- If a web page contains **nested lists within records**, these methods may fail to correctly segment and extract the data.

- Variation in the Number of Elements

- Each data record may have a **different number of elements** (e.g., one product may have 3 reviews, another may have 5).
- **Fixed threshold-based similarity detection** does not work well because the structure varies across records.

# Extraction Given a List Page: Nested Data Records

- Once a list of data records is identified, we can align and extract data items from them.
- Approaches (align multiple data records):
  - Multiple string alignment
    - Many ambiguities due to pervasive use of table related tags.
  - Multiple tree alignment (partial tree alignment)
    - Together with visual information is effective

# Solution idea

100

- The problem, however, can be dealt with as follows.
  - Instead of traversing the DOM tree top down, we can traverse it post-order.
  - This ensures that nested lists at lower levels are found first based on repeated patterns before going to higher levels.
  - When a nested list is found, its records are **collapsed** to produce a single template.
  - This template replaces the list of nested data records.
- When comparisons are made at a higher level, the algorithm only sees the template. Thus it is treated as a flat data record.

Let's say we are extracting product information from an e-commerce webpage. The page contains a list of products, and each product has a list of customer reviews (nested data records).

**Before Applying the Solution (Normal Approach - Top-Down Traversal)**

- The algorithm starts from the top of the webpage and goes downward.
- When it reaches a product, it sees a list of reviews inside it.
- The reviews have different numbers of comments (some products have 2 reviews, others have 5).
- This makes it hard to compare products because they have different structures.

**After Applying the Solution (New Approach - Post-Order Traversal & Collapsing Nested Lists)**

1. The algorithm **starts from the bottom** of the webpage (reviews first).
2. It **groups all reviews** for each product into **one template** (instead of separate review records).
    - Example:
        - Before:
            - **Product 1:** Review A, Review B, Review C
            - **Product 2:** Review X, Review Y
        - After collapsing:
            - **Product 1:** Reviews = "Summary of Reviews (A, B, C)"
            - **Product 2:** Reviews = "Summary of Reviews (X, Y)"
3. Now, at the **higher level**, when the algorithm processes the product list, it only sees:
    - **Product Name, Price, and a Single Reviews Field (Template)**
4. This makes all products **look similar in structure**, making it easier to extract and compare them.

## Final Output After Applying the Solution

| Product Name | Price | Reviews (Collapsed) |
|---|---|---|
| Phone A | $500 | Summary of Reviews (A, B, C) |
| Phone B | $300 | Summary of Reviews (X, Y) |

## Key Benefit:

✅ Instead of dealing with **complex nested data**, we **collapse** it into a single **template**, making the entire process **faster and more efficient**.

# The NET algorithm

101

```
Algorithm NET(Root, τ)
1  TraverseAndMatch(Root, τ);
2  for each top level node Node whose children have aligned data records do
3     PutDataInTables(Node);
4  endfor

TraverseAndMatch (Node, τ)
1  if Depth(Node) ≥ 3 then
2     for each Child ∈ Node.Children do
3        TraverseAndMatch(Child, τ);
4     endfor
5     Match(Node, τ);
6  endif
```

**Fig. 31.** The overall NET algorithm

## Breakdown of the NET Algorithm from the Given Pseudocode

The algorithm consists of two main parts:

1. **Main NET Algorithm (** `NET(Root, τ)` **)** – Starts the process and organizes the extracted data.
2. **Recursive Traversal (** `TraverseAndMatch(Node, τ)` **)** – Recursively searches and matches structured data.

### 1. Main NET Algorithm ( `NET(Root, τ)` )

- **Step 1:** Calls the `TraverseAndMatch` function on the **root node** to start analyzing the webpage's structure.
- **Step 2-4:** Once aligned data records (structured patterns) are found, they are **stored in tables** using `PutDataInTables(Node)` .
    - This ensures that extracted data is organized in a useful format, like a structured dataset or table.

### 2. Recursive Traversal ( `TraverseAndMatch(Node, τ)` )

- **Step 1:** Checks if the node is deep enough ( `Depth(Node) ≥ 3` ).
    - This avoids unnecessary processing of **high-level** nodes (like `<html>` or `<body>` ).
- **Step 2-4:** Recursively visits each child node using `TraverseAndMatch(Child, τ)` .
    - This ensures the algorithm **processes all nested structures** in a **bottom-up manner.**
- **Step 5:** Calls `Match(Node, τ)` , which attempts to **detect patterns** and **align similar data records** for extraction.

## Key Idea of the NET Algorithm

✅ It traverses the DOM tree in a structured way to find patterns in aligned data records (e.g., product lists, news articles).

✅ It processes deeper nodes first to ensure small patterns are grouped before extracting larger data structures.

✅ It organizes extracted data into tables, making it easier to use for analysis, machine learning, or further processing.

## Example: Extracting Data from an E-commerce Website

Imagine a webpage listing **laptops** with details like **name, price, and ratings**. The NET algorithm would:

1. Start at the **root node** and traverse the webpage structure.
2. Identify repeated patterns (e.g., multiple `<div>` sections containing product details).
3. Align and match these records into a structured format.
4. Store the extracted data in a table like this:

| Laptop Name | Price | Rating |
|---|---|---|
| Laptop A | $500 | 4.5 ⭐ |
| Laptop B | $700 | 4.7 ⭐ |

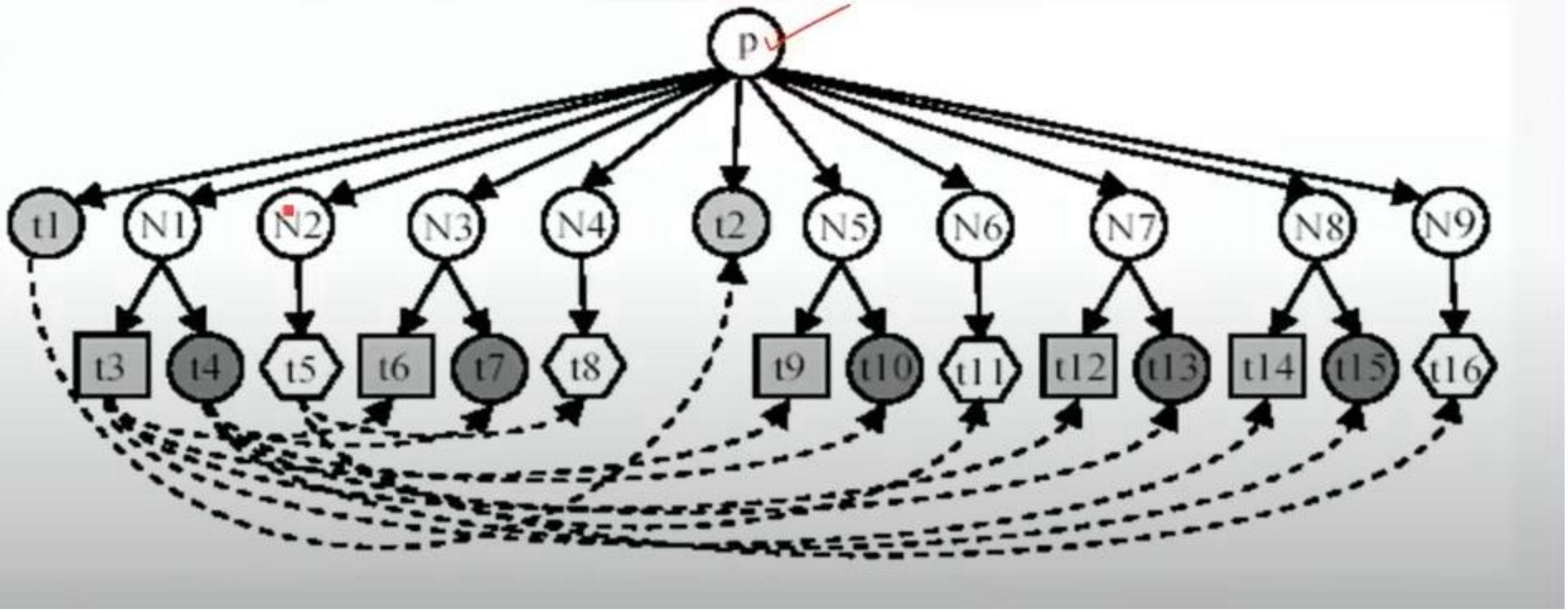
# The MATCH algorithm

It performs tree matching on child sub-trees of *Node* and template generation. $\tau$ is the threshold for a match of two trees to be considered sufficiently similar.

```
Match(Node, τ)
1   Children ← Node.Children;
2   while Children ≠ ∅ do
3       ChildFirst ← select and remove the first child from Children;
4       for each ChildR in Children ← Children − {ChildFirst} do
5           if TreeMatch(ChildFirst, ChildR) > τ then
6               AlignAndLink();
7               Children ← Children − {ChildR}
8       endfor
9       if some alignments (or links) have been made with ChildFirst then
10          GenNodeTemplate(ChildFirst)
11  endwhile
12  If consecutive child nodes in Children are aligned then
13      GenRecordTemplate(Node)
```

# An example

103



# GenNodeTemplate

104

- It generates a **node template** for all the nodes (including their sub-trees) that match *ChildFirst*.
  - It first gets the set of matched nodes *ChildRs*
  - then calls PartialTreeAlignment to produce a template which is the final seed tree.
- Note: AlignAndLink aligns and links all matched data items in *ChildFirst* and *ChildR*.

# GenRecordPattern

105

- This function produces a regular expression pattern for each data record.
- This is a grammar induction problem.
- Grammar induction in our context is to infer a regular expression given a finite set of positive and negative example strings.
  - However, we only have a single positive example. Fortunately, structured data in Web pages are usually highly regular which enables heuristic methods to generate "simple" regular expressions.
  - We need to make some assumptions

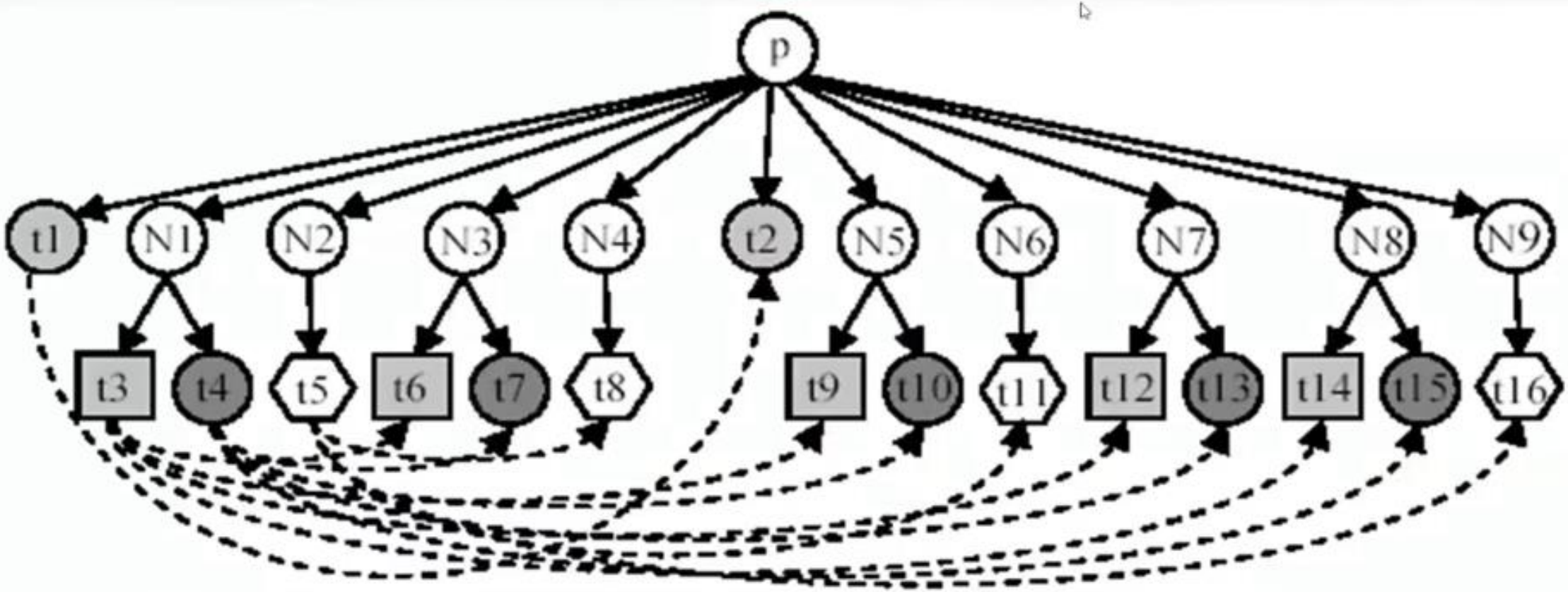
# Assumptions

106

- Three assumptions
  - The nodes in the first data record at each level must be complete.
  - The first node of every data record at each level must be present.
  - Nodes within a flat data record (no nesting) do not match one another.
- On the Web, these are not strong assumptions. In fact, they work well in practice.

# An example

108



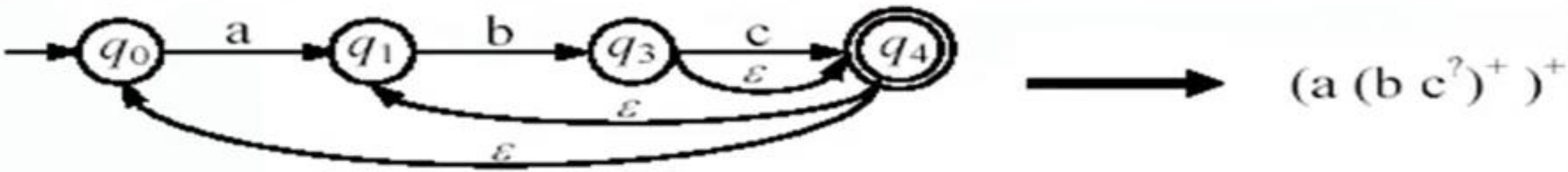
# An example

- Line 1 simply produces a string for generating a regular expression.

For our example, we obtain the following:

| | t1 | N1 | N2 | N3 | N4 | t2 | N5 | N6 | N7 | N8 | N9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *String*: | a | b | c | b | c | a | b | c | b | b | c |

The final NFA and the regular expression




$(a\ (b\ c^{?})^{+}\ )^{+}$

# Example (cont ...)

- We finally obtain the following

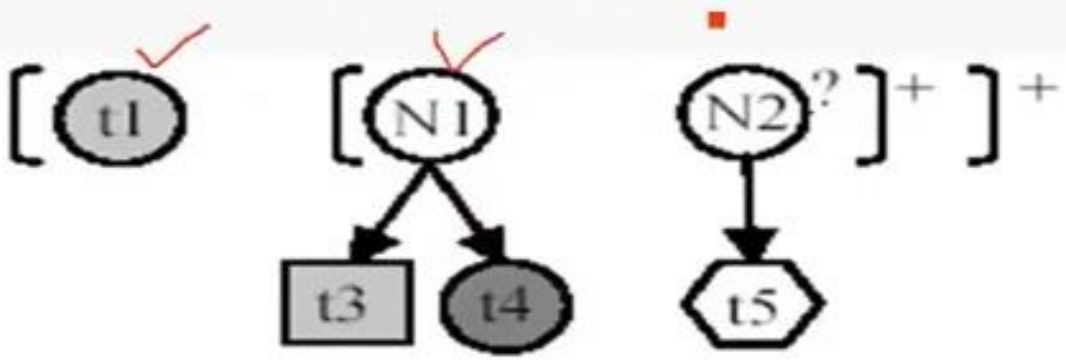



**Fig. 36.** The regular expression pattern produced from Fig. 33.

# Data extraction

- The function PutDataInTables (line 3 of NET) outputs data items in a table, which is simple after the data record templates are found.
- An example

| | | | |
|---|---|---|---|
| t1 | t3 | t4 | t5 |
| t1 | t6 | t7 | t8 |
| t2 | t9 | t10 | t11 |
| t2 | t12 | t13 | |
| t2 | t14 | t15 | t16 |

**Fig. 37.** The output data table for the example in Fig. 33.

# Generating NFA

107

**GenRecordPattern**(*Node*)

1 *String* ← Assign a distinctive symbol to each set of matched children of *Node*;
2 create an NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0\}$, $q_0$ is the start state, $\Sigma$ is the symbol set containing all symbols appeared in *String*, $\delta$ is the transition function, $F = \varnothing$ is the set of accept states;
3 $q_c \leftarrow q_0$; // $q_c$ is the current state
4 **for** each symbol $s$ in *String* in sequence **do**
5 **if** $\exists$ a transition $\delta(q_c, s) = q_n$ **then**
6 $q_c \leftarrow q_n$ // transit to the next state;
7 **else if** $\exists\ \delta(q_i, s) = q_j$, where $q_i, q_j \in Q$ **then** // $s$ appeared before
8 **if** $\exists\ \delta(q_f, \varepsilon) = q_i$, where $\delta(q_i, s) = q_j$ and $f \geq c$ **then**
9 TransitTo($q_c$, $q_f$);
10 **else** create a transition $\delta(q_c, \varepsilon) = q_i$, where $\delta(q_i, s) = q_j$
11 $q_c \leftarrow q_j$
12 **else** create a new state $q_{c+1}$ and a transition $\delta(q_c, s) = q_{c+1}$;
13 $Q = Q \cup \{q_{c+1}\}$;
14 $q_c \leftarrow q_{c+1}$
15 **if** $s$ is the last symbol in *String* **then**
16 Assign the state with the largest subscript the accept state $q_r$, $F = \{q_r\}$;
17 TransitTo($q_c$, $q_r$);
18 **endfor**
19 generate a regular expression pattern based on the NFA $N$;
20 Substitute all the node templates into the regular expression pattern.

# The NET algorithm

101

```
Algorithm NET(Root, τ)
1  TraverseAndMatch(Root, τ);
2  for each top level node Node whose children have aligned data records do
3     PutDataInTables(Node);
4  endfor

TraverseAndMatch (Node, τ)
1  if Depth(Node) ≥ 3 then
2        for each Child ∈ Node.Children do
3              TraverseAndMatch(Child, τ);
4        endfor
5        Match(Node, τ);
6  endif
```

**Fig. 31.** The overall NET algorithm

# Data extraction

- The function PutDataInTables (line 3 of NET) outputs data items in a table, which is simple after the data record templates are found.
- An example

| | | | |
|---|---|---|---|
| t1 | t3 | t4 | t5 |
| t1 | t6 | t7 | t8 |
| t2 | t9 | t10 | t11 |
| t2 | t12 | t13 | |
| t2 | t14 | t15 | t16 |

**Fig. 37.** The output data table for the example in Fig. 33.

# An more complete example



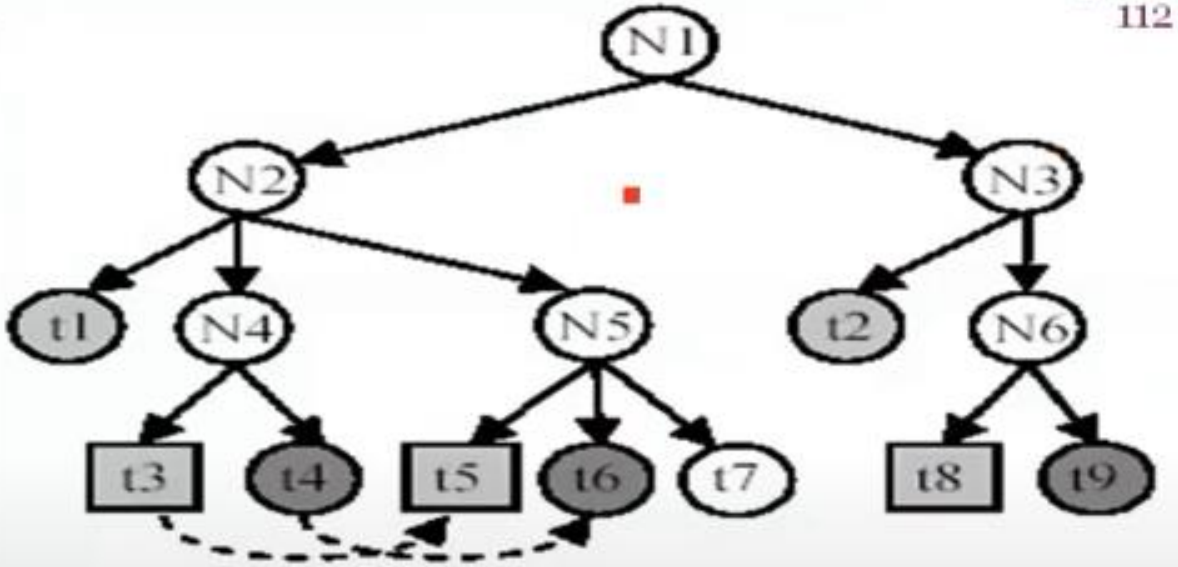



**Fig. 38.** Aligned data nodes are linked

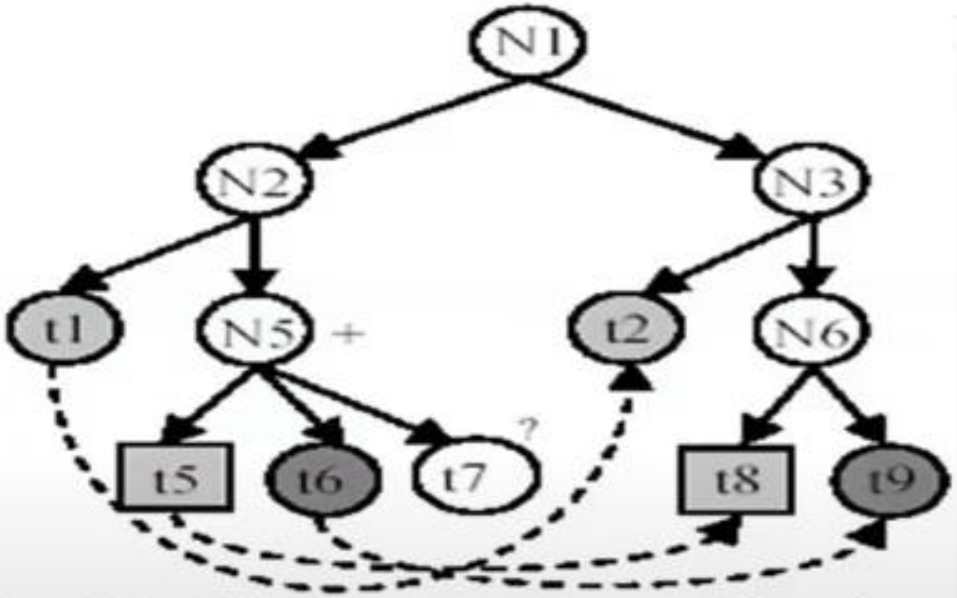



**Fig. 39.** Alignment after collapsing

| | | | |
|---|---|---|---|
| t1 | t3 | t4 | |
| t1 | t5 | t6 | t7 |
| t2 | t8 | t9 | |

**Fig. 40.** The output data table for the example in Fig. 38

# Extraction Given Multiple Pages



- We now discuss the second extraction problem described in Section 8.3.1.
  - Given multiple pages with the same encoding template, the system finds patterns from them to extract data from other similar pages.
  - The collection of input pages can be a set of list pages or detail pages.
- Below, we first see how the techniques described so far can be applied in this setting, and then
  - describe a technique specifically designed for this setting.

# Using previous techniques



- **Given a set of list pages**
  - The techniques described in previous sections are for a single list page.
  - They can clearly be used for multiple list pages.
- If multiple list pages are available, they may help improve the extraction.
  - For example, templates from all input pages may be found separately and merged to produce a single refined pattern.
  - This can deal with the situation where a single page may not contain the complete information.

# Given a set of detail pages

115

- In some applications, one needs to extract data from detail pages as they contain more information on the object. Information in list pages are quite brief.
- For extraction, we can treat each detail page as a data record, and extract using the algorithm described in Section 8.7 and/or Section 8.8.
  - For instance, to apply the NET algorithm, we simply create a rooted tree as the input to NET as follows:
    - create an artificial root node, and
    - make the DOM tree of each page as a child sub-tree of the artificial root node.