Syntactic Methods (Strings and Grammars)

Jason Corso

SUNY at Buffalo

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.
- Examples
  - String of letters in English
  - DNA bases in a gene sequence (AGCTTC...)

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.
- Examples
  - String of letters in English
  - DNA bases in a gene sequence (AGCTTC...)
- There are a number of differences in the way we need to approach the pattern recognition in this case.

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.
- Examples
  - String of letters in English
  - DNA bases in a gene sequence (AGCTTC...)
- There are a number of differences in the way we need to approach the pattern recognition in this case.
  1. The characters in the string are nominal and have no obvious notion of distance.

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.
- Examples
  - String of letters in English
  - DNA bases in a gene sequence (AGCTTC...)
- There are a number of differences in the way we need to approach the pattern recognition in this case.
  1. The characters in the string are nominal and have no obvious notion of distance.
  2. Strings need not be of the same length.

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.
- Examples
  - String of letters in English
  - DNA bases in a gene sequence (AGCTTC...)
- There are a number of differences in the way we need to approach the pattern recognition in this case.
  1. The characters in the string are nominal and have no obvious notion of distance.
  2. Strings need not be of the same length.
  3. Long-range interdepencies often exist in strings.

Recognition with Strings

- Take a different view and consider the situation where the patterns are represented as sequences of nominal discrete items.
- Examples
  - String of letters in English
  - DNA bases in a gene sequence (AGCTTC...)
- There are a number of differences in the way we need to approach the pattern recognition in this case.
  1. The characters in the string are nominal and have no obvious notion of distance.
  2. Strings need not be of the same length.
  3. Long-range interdepencies often exist in strings.
- Notation
  - Assume each discrete character is taken from an alphabet $\mathcal{A}$.
  - Use the same vector notation for a string: $\mathbf{x} =$"AGCTTC".
  - Call a particularly long string *text*.
  - Call a contiguous substring of $\mathbf{x}$ a *factor*.

Key String Problems

- **String Matching**: Given $\mathbf{x}$ and *text*, determine whether $\mathbf{x}$ is a factor of *text*, and, if so, where it appears.

Key String Problems

- **String Matching**: Given $\mathbf{x}$ and *text*, determine whether $\mathbf{x}$ is a factor of *text*, and, if so, where it appears.
- **Edit Distance**: Given two strings $\mathbf{x}$ and $\mathbf{y}$, compute the minimum number of basic operations—character insertions, delections, and exchanges—needed to transform $\mathbf{x}$ into $\mathbf{y}$.

Key String Problems

- **String Matching**: Given $\mathbf{x}$ and *text*, determine whether $\mathbf{x}$ is a factor of *text*, and, if so, where it appears.
- **Edit Distance**: Given two strings $\mathbf{x}$ and $\mathbf{y}$, compute the minimum number of basic operations—character insertions, delections, and exchanges—needed to transform $\mathbf{x}$ into $\mathbf{y}$.
- **String Matching with Errors**: Given $\mathbf{x}$ and *text*, find the locations in *text* where the "cost" or "distance" of $\mathbf{x}$ to any factor of *text* is minimal.

Key String Problems

- **String Matching**: Given $\mathbf{x}$ and *text*, determine whether $\mathbf{x}$ is a factor of *text*, and, if so, where it appears.
- **Edit Distance**: Given two strings $\mathbf{x}$ and $\mathbf{y}$, compute the minimum number of basic operations—character insertions, delections, and exchanges—needed to transform $\mathbf{x}$ into $\mathbf{y}$.
- **String Matching with Errors**: Given $\mathbf{x}$ and *text*, find the locations in *text* where the "cost" or "distance" of $\mathbf{x}$ to any factor of *text* is minimal.
- **String Matching with the "Don't–Care" Symbol**: This is the same as basic string matching, but with the special symbol–$\emptyset$, the *don't care* symbol–which can match any other symbol.

String Matching

- The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of *text*.

String Matching

- The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of *text*.
- Assume the number of characters in *text* is greater than that in $\mathbf{x}$: $|text| > |\mathbf{x}|$ or $|text| \gg |\mathbf{x}|$.

String Matching

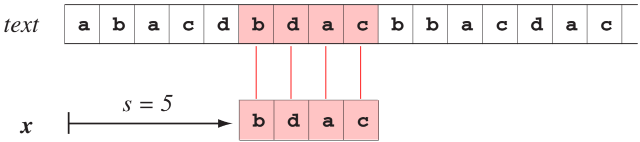
- The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of *text*.
- Assume the number of characters in *text* is greater than that in $\mathbf{x}$: $|text| > |\mathbf{x}|$ or $|text| \gg |\mathbf{x}|$.
- Define a **shift** s as an offset needed to align the first character of $\mathbf{x}$ with the character number $s+1$ in *text*.

String Matching

- The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of *text*.
- Assume the number of characters in *text* is greater than that in $\mathbf{x}$: $|text| > |\mathbf{x}|$ or $|text| \gg |\mathbf{x}|$.
- Define a **shift** s as an offset needed to align the first character of $\mathbf{x}$ with the character number $s+1$ in *text*.
- The basic problem of string matching is to find whether or not there is a **valid shift**, one where there is a perfect match between each character in $\mathbf{x}$ and the corresponding one in *text*.

String Matching

- The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of *text*.
- Assume the number of characters in *text* is greater than that in $\mathbf{x}$: $|text| > |\mathbf{x}|$ or $|text| \gg |\mathbf{x}|$.
- Define a **shift** s as an offset needed to align the first character of $\mathbf{x}$ with the character number $s + 1$ in *text*.
- The basic problem of string matching is to find whether or not there is a **valid shift**, one where there is a perfect match between each character in $\mathbf{x}$ and the corresponding one in *text*.



Naive String Matching

```
begin initialize A, x, n ← |text|,  m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s + 1 ... s + m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

Naive String Matching

```
begin initialize 𝒜, x, n ← |text|,  m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s+1 ... s+m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

Naive String Matching

```
begin initialize 𝒜, x, n ← |text|, m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s + 1 ... s + m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

Naive String Matching

```
begin initialize 𝒜, x, n ← |text|,  m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s + 1 ... s + m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

Naive String Matching

```
begin initialize A, x, n ← |text|, m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s + 1 ... s + m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

Naive String Matching

```
begin initialize A, x, n ← |text|, m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s + 1 ... s + m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

Naive String Matching

```
begin initialize A, x, n ← |text|, m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1 ... m] = text[s+1 ... s+m]
      then print "pattern occurs at shift" s
    s ← s+1
  return
end
```

- Although this algorithm will compute the string match, it does so quite inefficiently. Worst case complexity is $\Theta((n-m+1)m)$.

Naive String Matching

```
begin initialize A, x, n ← |text|,  m ← |x|
  s ← 0
  while s ≤ n − m
    if x[1... m] = text[s + 1 ... s + m]
      then print "pattern occurs at shift" s
    s ← s + 1
  return
end
```

- Although this algorithm will compute the string match, it does so quite inefficiently. Worst case complexity is $\mathbf{\Theta}((n-m+1)m)$.
- The weakness comes from the fact that it does not use any information about a potential shift s to compute the next possible one s.

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|, m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|,  m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|, m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|,  m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|, m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize A, x, n ← |text|, m ← |x|
  s ← 0
  F(x) ← last-occurrence function
  G(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + G(0)
        else s ← max[G(j), j − F(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|,  m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
              s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize 𝒜, x, n ← |text|, m ← |x|
  s ← 0
  ℱ(x) ← last-occurrence function
  𝒢(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
              s ← s + 𝒢(0)
        else s ← max[𝒢(j), j − ℱ(text[s + j])]
  return
end
```

Boyer-Moore String Matching

```
begin initialize A, x, n ← |text|,  m ← |x|
  s ← 0
  F(x) ← last-occurrence function
  G(x) ← good-suffix function
  while s ≤ n − m
    j ← m
    while j > 0 and x[j] = text[s + j]
      j ← j − 1
      if j = 0
        then print "pattern occurs at shift" s
             s ← s + G(0)
        else s ← max[G(j), j − F(text[s + j])]
  return
end
```

Boyer-Moore String Matching

begin initialize $\mathcal{A}$, $\mathbf{x}$, $n \leftarrow |\,\textit{text}\,|$, $m \leftarrow |\mathbf{x}|$
  $s \leftarrow 0$
  $\mathcal{F}(\mathbf{x}) \leftarrow$ last-occurrence function
  $\mathcal{G}(\mathbf{x}) \leftarrow$ good-suffix function
  while $s \leq n - m$
    $j \leftarrow m$
    while $j > 0$ and $\mathbf{x}[j] = \textit{text}[s+j]$
      $j \leftarrow j - 1$
      if $j = 0$
        then print "pattern occurs at shift" s
          $s \leftarrow s + \mathcal{G}(0)$
        else $s \leftarrow \max[\mathcal{G}(j), j - \mathcal{F}(\textit{text}[s+j])]$
  return
end

- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are

- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are
  1. At each candidate shift s, the character comparisons are done in reverse order.

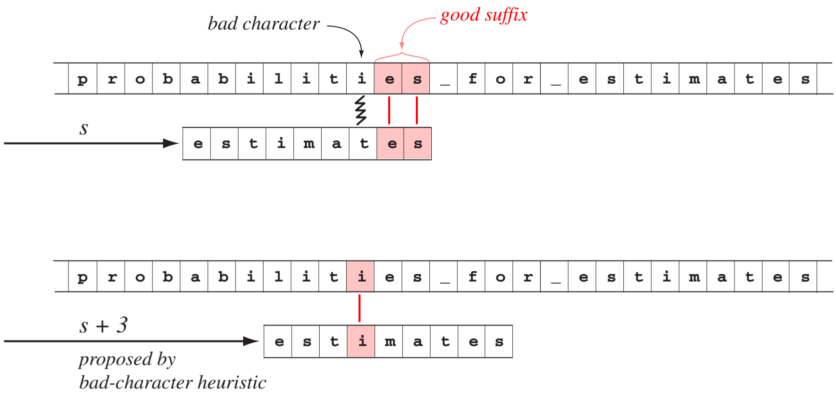
- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are
  1. At each candidate shift s, the character comparisons are done in reverse order.
  2. The increment of a new shift need not be 1.

- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are
  1. At each candidate shift s, the character comparisons are done in reverse order.
  2. The increment of a new shift need not be 1.
- The real power in Boyer-Moore comes from two heuristics that govern how much the shift can be safely incremented by without missing a valid shift.

- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are
  1. At each candidate shift s, the character comparisons are done in reverse order.
  2. The increment of a new shift need not be 1.
- The real power in Boyer-Moore comes from two heuristics that govern how much the shift can be safely incremented by without missing a valid shift.
- **The bad-character heuristic** utilizes the rightmost character in *text* that does not match the aligned character in $\mathbf{x}$.

- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are
  1. At each candidate shift s, the character comparisons are done in reverse order.
  2. The increment of a new shift need not be 1.
- The real power in Boyer-Moore comes from two heuristics that govern how much the shift can be safely incremented by without missing a valid shift.
- **The bad-character heuristic** utilizes the rightmost character in *text* that does not match the aligned character in $\mathbf{x}$.
  - The "bad-character" can be found as efficiently as possible because evaluation occurs from right-to-left.

- Two key differences in the Boyer-Moore algorithm over the Naive algorithm are
  1. At each candidate shift s, the character comparisons are done in reverse order.
  2. The increment of a new shift need not be 1.
- The real power in Boyer-Moore comes from two heuristics that govern how much the shift can be safely incremented by without missing a valid shift.
- **The bad-character heuristic** utilizes the rightmost character in *text* that does not match the aligned character in $\mathbf{x}$.
  - The "bad-character" can be found as efficiently as possible because evaluation occurs from right-to-left.
  - It will then propose to increment the shift by an amount to align the rightmost occurrence of the bad character in $\mathbf{x}$ with the bad character identified in *text*. Hence, we are guaranteed that no valid shifts have been skipped.


bad character
good suffix
p r o b a b i l i t i e s _ f o r _ e s t i m a t e s
s
e s t i m a t e s
p r o b a b i l i t i e s _ f o r _ e s t i m a t e s
s + 3
e s t i m a t e s
proposed by
bad-character heuristic

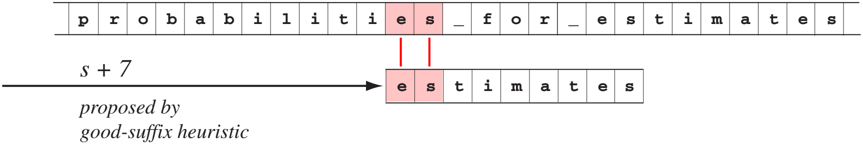
- **The good-suffix heuristic** also proposes a safe shift and works in parallel with with the bad-character heuristic.

- **The good-suffix heuristic** also proposes a safe shift and works in parallel with with the bad-character heuristic.
- A **suffix** of $\mathbf{x}$ is a factor of $\mathbf{x}$ that contains the final character in $\mathbf{x}$.

- **The good-suffix heuristic** also proposes a safe shift and works in parallel with with the bad-character heuristic.
- A **suffix** of $\mathbf{x}$ is a factor of $\mathbf{x}$ that contains the final character in $\mathbf{x}$.
- A **good suffix**, or matching suffix, is a set of rightmost characters in *text*, at shift s that match those in $\mathbf{x}$.
  - The good suffix is likewise found efficiently due to the right-to-left search.

- **The good-suffix heuristic** also proposes a safe shift and works in parallel with with the bad-character heuristic.
- A **suffix** of $\mathbf{x}$ is a factor of $\mathbf{x}$ that contains the final character in $\mathbf{x}$.
- A **good suffix**, or matching suffix, is a set of rightmost characters in *text*, at shift s that match those in $\mathbf{x}$.
  - The good suffix is likewise found efficiently due to the right-to-left search.
- It will propose to increment the shift so as to align the next occurrence of the good suffix in $\mathbf{x}$ with that identified in *text*.

- **The good-suffix heuristic** also proposes a safe shift and works in parallel with with the bad-character heuristic.
- A **suffix** of $\mathbf{x}$ is a factor of $\mathbf{x}$ that contains the final character in $\mathbf{x}$.
- A **good suffix**, or matching suffix, is a set of rightmost characters in *text*, at shift s that match those in $\mathbf{x}$.
  - The good suffix is likewise found efficiently due to the right-to-left search.
- It will propose to increment the shift so as to align the next occurrence of the good suffix in $\mathbf{x}$ with that identified in *text*.



- The **last-occurrence function**, $\mathcal{F}(\mathbf{x})$ is simply a table containing every letter in the alphabet and the position of its rightmost occurrence in $\mathbf{x}$.

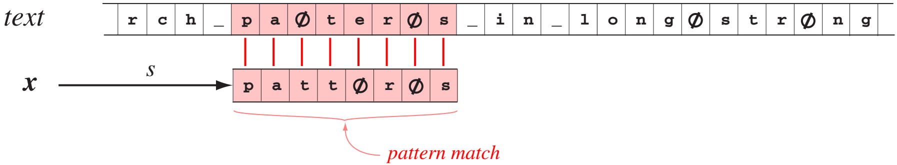
- The **last-occurrence function**, $\mathcal{F}(\mathbf{x})$ is simply a table containing every letter in the alphabet and the position of its rightmost occurrence in $\mathbf{x}$.
- The **good-suffix function**, $\mathcal{G}(\mathbf{x})$ creates a table that for each suffix gives the location of its second right-most occurrence in $\mathbf{x}$.

- The **last-occurrence function**, $\mathcal{F}(\mathbf{x})$ is simply a table containing every letter in the alphabet and the position of its rightmost occurrence in $\mathbf{x}$.
- The **good-suffix function**, $\mathcal{G}(\mathbf{x})$ creates a table that for each suffix gives the location of its second right-most occurrence in $\mathbf{x}$.
- These tables can be computed only once and can be stored offline. They hence do not significantly affect the computational complexity of the method.

- The **last-occurrence function**, $\mathcal{F}(\mathbf{x})$ is simply a table containing every letter in the alphabet and the position of its rightmost occurrence in $\mathbf{x}$.
- The **good-suffix function**, $\mathcal{G}(\mathbf{x})$ creates a table that for each suffix gives the location of its second right-most occurrence in $\mathbf{x}$.
- These tables can be computed only once and can be stored offline. They hence do not significantly affect the computational complexity of the method.
- These heuristics make the Boyer-Moore string searching algorithm one of the most attactive string-matching algorithms on serial computers.

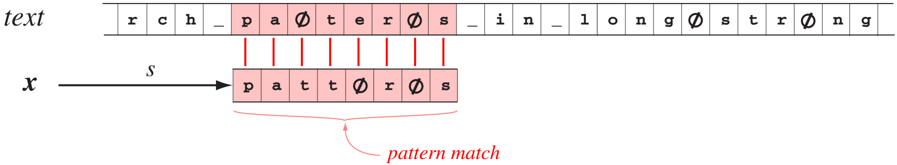
String Matching with Wildcards

- Formally, this is the same as string-matching, with the addition that the symbol $\varnothing$ can match anything in either $\mathbf{x}$ or *text*.



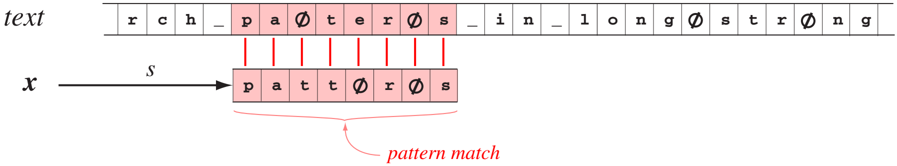
String Matching with Wildcards

- Formally, this is the same as string-matching, with the addition that the symbol $\emptyset$ can match anything in either $\mathbf{x}$ or *text*.
- An obvious thing to do is modify the Naive algorithm and include a special condition, but this would maintain the computational inefficiencies of the original method.



String Matching with Wildcards

- Formally, this is the same as string-matching, with the addition that the symbol $\varnothing$ can match anything in either $\mathbf{x}$ or *text*.
- An obvious thing to do is modify the Naive algorithm and include a special condition, but this would maintain the computational inefficiencies of the original method.
- Extending Boyer-Moore is quite a challenge...



Edit Distance

- The fundamental idea behind edit distance is based on the **nearest-neighbor** algorithm.

Edit Distance

- The fundamental idea behind edit distance is based on the **nearest-neighbor** algorithm.
- We store a full set of strings and their associated category labels. During classification, a test string is compared to each stored string and a "distance" is computed. Then, we assign the category of the string with the shortest distance.

Edit Distance

- The fundamental idea behind edit distance is based on the **nearest-neighbor** algorithm.
- We store a full set of strings and their associated category labels. During classification, a test string is compared to each stored string and a "distance" is computed. Then, we assign the category of the string with the shortest distance.
- But, how do we compute the distance between two strings?

Edit Distance

- The fundamental idea behind edit distance is based on the **nearest-neighbor** algorithm.
- We store a full set of strings and their associated category labels. During classification, a test string is compared to each stored string and a “distance” is computed. Then, we assign the category of the string with the shortest distance.
- But, how do we compute the distance between two strings?
- **Edit distance** is a possibility, which describes how many fundamental operations are required to transform $\mathbf{x}$ into $\mathbf{y}$, another string.

The fundamental operations are as follows.

1. **Substitions**: a character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.

The fundamental operations are as follows.

1. **Substitions**: a character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.
2. **Insertions**: a chracter in $\mathbf{y}$ is inserted into $\mathbf{x}$, thereby increasing the length of $\mathbf{x}$ by one character.

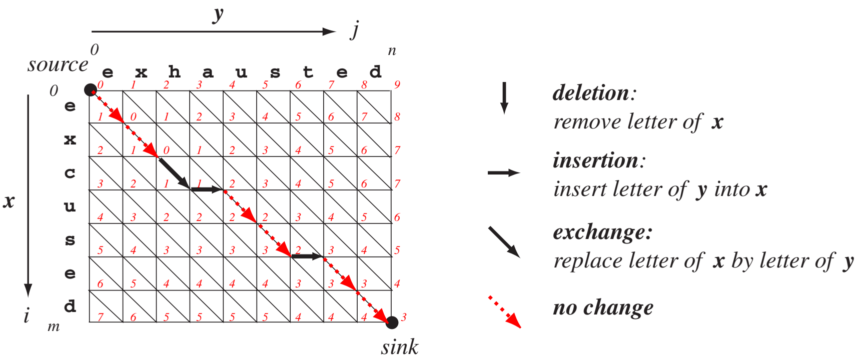
The fundamental operations are as follows.

1. **Substitions**: a character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.
2. **Insertions**: a chracter in $\mathbf{y}$ is inserted into $\mathbf{x}$, thereby increasing the length of $\mathbf{x}$ by one character.
3. **Deletions**: a character in $\mathbf{x}$ is deleted, thereby decreasing the length of $\mathbf{x}$ by one character.

The fundamental operations are as follows.

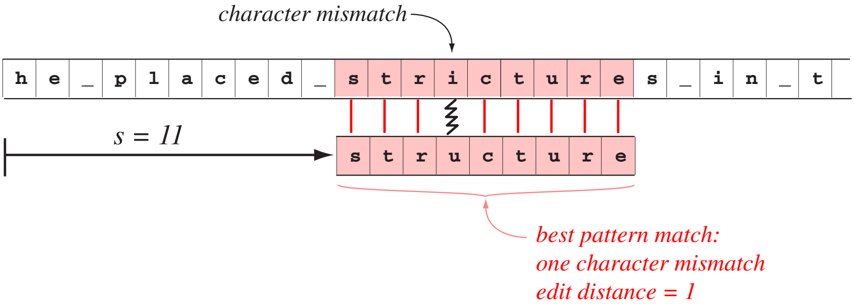
1. **Substitions**: a character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.
2. **Insertions**: a chracter in $\mathbf{y}$ is inserted into $\mathbf{x}$, thereby increasing the length of $\mathbf{x}$ by one character.
3. **Deletions**: a character in $\mathbf{x}$ is deleted, thereby decreasing the length of $\mathbf{x}$ by one character.
4. **Transpositions**: two neighboring characters in $\mathbf{x}$ change positions. But, this is not really a fundamental operation because we can always encode it by two substitutions.

The basic Edit Distance algorithm builds an $m \times n$ matrix of costs and uses it to compute the distance. Below is a graphic describing the basic idea. For more details read section 8.5.2 on your own.



String Matching with Errors

- Problem: Given a pattern $\mathbf{x}$ and *text*, find the shift for which the edit distance between $\mathbf{x}$ and a factor of *text* is minimum.
- Proceed in a similar manner to the Edit Distance algorithm, but need to compute a second matrix of minimum edit values across the rows and columns.



String Matching Round-Up

- We've covered the basics of string matching.
- How does these methods relate to the temporal ones we saw last week?
- While learning has found general use in pattern recognition, its application in basic string matching has been quite limited.

Grammatical Methods

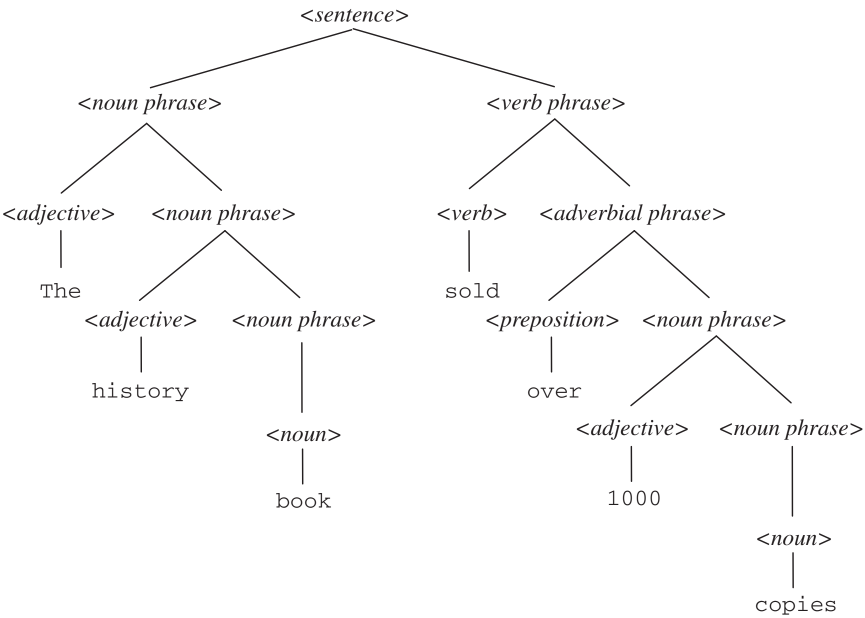
- The earlier discussion on string matching paid no attention to any models that might have underlied the creation of the sequence of characters in the string.

Grammatical Methods

- The earlier discussion on string matching paid no attention to any models that might have underlied the creation of the sequence of characters in the string.
- In the case of grammatical methods, we are concerned with the set of rules that were used to generate the strings.

Grammatical Methods

- The earlier discussion on string matching paid no attention to any models that might have underlied the creation of the sequence of characters in the string.
- In the case of grammatical methods, we are concerned with the set of rules that were used to generate the strings.
- In this case, the structure of the strings is fundamental. And, the structure is often **hierarchical**.



<sentence>
<noun phrase>
<verb phrase>
<adjective>
<noun phrase>
<verb>
<adverbial phrase>
The
sold
<adjective>
<noun phrase>
<preposition>
<noun phrase>
history
over
<adjective>
<noun phrase>
<noun>
1000
book
<noun>
copies

Grammars

- The structure can easily be specified in a **grammar**.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

1 **Symbols**: These are the characters taken from an alphabet $\mathcal{A}$, as before. They are often called **primitive or terminal symbols**. The null or empty string ϵ of length 0 is also included.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

1 **Symbols**: These are the characters taken from an alphabet $\mathcal{A}$, as before. They are often called **primitive or terminal symbols**. The null or empty string ϵ of length 0 is also included.

2 **Variables**: These are also called nonterminal or intermediate symbols and are taken from a set $\mathcal{I}$.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

1. **Symbols**: These are the characters taken from an alphabet $\mathcal{A}$, as before. They are often called **primitive or terminal symbols**. The null or empty string ϵ of length 0 is also included.
2. **Variables**: These are also called nonterminal or intermediate symbols and are taken from a set $\mathcal{I}$.
3. **Root Symbol**: This is a special variable from which all sequences of symbols are derived. The root symbol is taken from a set $\mathcal{S}$.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

1. **Symbols**: These are the characters taken from an alphabet $\mathcal{A}$, as before. They are often called **primitive or terminal symbols**. The null or empty string ϵ of length 0 is also included.
2. **Variables**: These are also called nonterminal or intermediate symbols and are taken from a set $\mathcal{I}$.
3. **Root Symbol**: This is a special variable from which all sequences of symbols are derived. The root symbol is taken from a set $\mathcal{S}$.
4. **Production Rules**: The set of operations, $\mathcal{P}$ that specify how to transofrm a set of variables and symbols into othe variables and symbols. These rules determine the core structures that can be produced by the grammar.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

1. **Symbols**: These are the characters taken from an alphabet $\mathcal{A}$, as before. They are often called **primitive or terminal symbols**. The null or empty string ϵ of length 0 is also included.
2. **Variables**: These are also called nonterminal or intermediate symbols and are taken from a set $\mathcal{I}$.
3. **Root Symbol**: This is a special variable from which all sequences of symbols are derived. The root symbol is taken from a set $\mathcal{S}$.
4. **Production Rules**: The set of operations, $\mathcal{P}$ that specify how to transofrm a set of variables and symbols into othe variables and symbols. These rules determine the core structures that can be produced by the grammar.

- Thus, we denote a grammar by $G = (\mathcal{A}, \mathcal{I}, \mathcal{S}, \mathcal{P})$.

Grammars

- The structure can easily be specified in a **grammar**.
- Formally, a **grammar** consists of four components.

1. **Symbols**: These are the characters taken from an alphabet $\mathcal{A}$, as before. They are often called **primitive or terminal symbols**. The null or empty string ϵ of length 0 is also included.
2. **Variables**: These are also called nonterminal or intermediate symbols and are taken from a set $\mathcal{I}$.
3. **Root Symbol**: This is a special variable from which all sequences of symbols are derived. The root symbol is taken from a set $\mathcal{S}$.
4. **Production Rules**: The set of operations, $\mathcal{P}$ that specify how to transofrm a set of variables and symbols into othe variables and symbols. These rules determine the core structures that can be produced by the grammar.

- Thus, we denote a grammar by $G = (\mathcal{A}, \mathcal{I}, \mathcal{S}, \mathcal{P})$.
- The **language** generated by a grammar, $\mathcal{L}(G)$, is the set of all strings (possibly infinite) that can be generated by G.

Consider an abstract example:

- Let $\mathcal{A} = \{$a,b,c$\}$.
- Let $\mathcal{S} = \{$S$\}$.
- Let $\mathcal{I} = \{$A,B,C$\}$.

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

Consider an abstract example:

- Let $\mathcal{A} = \{a,b,c\}$.
- Let $\mathcal{S} = \{S\}$.
- Let $\mathcal{I} = \{A,B,C\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & S & \rightarrow aSBA \text{ OR } aBA \\ \mathbf{p}_2: & bB & \rightarrow bb \\ \mathbf{p}_3: & cA & \rightarrow cc \\ \mathbf{p}_4: & AB & \rightarrow BA \\ \mathbf{p}_5: & bA & \rightarrow bc \\ \mathbf{p}_6: & aB & \rightarrow ab \end{array} \right\}$$

---

root	S

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA

Consider an abstract example:

- Let $\mathcal{A} = \{$a,b,c$\}$.
- Let $\mathcal{S} = \{$S$\}$.
- Let $\mathcal{I} = \{$A,B,C$\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA
$\mathbf{p}_2$	aabBAA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA
$\mathbf{p}_2$	aabBAA
$\mathbf{p}_3$	aabbAA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA
$\mathbf{p}_2$	aabBAA
$\mathbf{p}_3$	aabbAA
$\mathbf{p}_4$	aabbcA

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA
$\mathbf{p}_2$	aabBAA
$\mathbf{p}_3$	aabbAA
$\mathbf{p}_4$	aabbcA
$\mathbf{p}_5$	aabbcc

Consider an abstract example:

- Let $\mathcal{A} = \{\text{a,b,c}\}$.
- Let $\mathcal{S} = \{\text{S}\}$.
- Let $\mathcal{I} = \{\text{A,B,C}\}$.

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & \text{S} & \rightarrow \text{aSBA OR aBA} \\ \mathbf{p}_2: & \text{bB} & \rightarrow \text{bb} \\ \mathbf{p}_3: & \text{cA} & \rightarrow \text{cc} \\ \mathbf{p}_4: & \text{AB} & \rightarrow \text{BA} \\ \mathbf{p}_5: & \text{bA} & \rightarrow \text{bc} \\ \mathbf{p}_6: & \text{aB} & \rightarrow \text{ab} \end{array} \right\}$$

---

root	S
$\mathbf{p}_1$	aBA
$\mathbf{p}_6$	abA
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA
$\mathbf{p}_2$	aabBAA
$\mathbf{p}_3$	aabbAA
$\mathbf{p}_4$	aabbcA
$\mathbf{p}_5$	aabbcc

These are two examples of **productions**.

Another example...English.

- The alphabet is all English words:
  $\mathcal{A} = \{$the, history, book, sold, over, ...$\}$.

Another example...English.

- The alphabet is all English words:
  $\mathcal{A} = \{$the, history, book, sold, over, ...$\}$.
- The variables are the parts of speech:
  $\mathcal{I} = \{\langle\text{noun}\rangle, \langle\text{verb}\rangle, \langle\text{noun phrase}\rangle, \langle\text{adjective}\rangle, \dots\}$.

Another example...English.

- The alphabet is all English words:
  $\mathcal{A} = \{$the, history, book, sold, over, ...$\}$.
- The variables are the parts of speech:
  $\mathcal{I} = \{\langle\text{noun}\rangle, \langle\text{verb}\rangle, \langle\text{noun phrase}\rangle, \langle\text{adjective}\rangle, \dots\}$.
- The root symbol is $\mathcal{S} = \{\langle\text{sentence}\rangle\}$.

Another example...English.

- The alphabet is all English words:
  $\mathcal{A} = \{$the, history, book, sold, over, ...$\}$.
- The variables are the parts of speech:
  $\mathcal{I} = \{\langle\text{noun}\rangle, \langle\text{verb}\rangle, \langle\text{noun phrase}\rangle, \langle\text{adjective}\rangle, \dots\}$.
- The root symbol is $\mathcal{S} = \{\langle\text{sentence}\rangle\}$.
- A restricted set of production rules is

$$\mathcal{P} = \left\{ \begin{array}{rcl} \langle\text{sentence}\rangle & \rightarrow & \langle\text{noun phrase}\rangle\langle\text{verb phrase}\rangle \\ \langle\text{noun phrase}\rangle & \rightarrow & \langle\text{adjective}\rangle\langle\text{noun phrase}\rangle \\ \langle\text{verb phrase}\rangle & \rightarrow & \langle\text{verb phrase}\rangle\langle\text{adverb phrase}\rangle \\ \langle\text{noun}\rangle & \rightarrow & \text{book OR theorem OR}\dots \\ \langle\text{verb}\rangle & \rightarrow & \text{describes OR buys OR}\dots \\ \langle\text{adverb}\rangle & \rightarrow & \text{over OR frankly OR}\dots \end{array} \right\}$$

Another example...English.

- The alphabet is all English words: $\mathcal{A} = \{$the, history, book, sold, over, ...$\}$.
- The variables are the parts of speech: $\mathcal{I} = \{\langle\text{noun}\rangle, \langle\text{verb}\rangle, \langle\text{noun phrase}\rangle, \langle\text{adjective}\rangle, \dots\}$.
- The root symbol is $\mathcal{S} = \{\langle\text{sentence}\rangle\}$.
- A restricted set of production rules is

$$\mathcal{P} = \left\{ \begin{array}{rcl} \langle\text{sentence}\rangle & \rightarrow & \langle\text{noun phrase}\rangle\langle\text{verb phrase}\rangle \\ \langle\text{noun phrase}\rangle & \rightarrow & \langle\text{adjective}\rangle\langle\text{noun phrase}\rangle \\ \langle\text{verb phrase}\rangle & \rightarrow & \langle\text{verb phrase}\rangle\langle\text{adverb phrase}\rangle \\ \langle\text{noun}\rangle & \rightarrow & \text{book OR theorem OR} \dots \\ \langle\text{verb}\rangle & \rightarrow & \text{describes OR buys OR} \dots \\ \langle\text{adverb}\rangle & \rightarrow & \text{over OR frankly OR} \dots \end{array} \right\}$$

- Of course, this subset of the rules for English grammar does not prevent the generation of meaningless sentences like *Squishy green dreams hop heuristically*.

Types of String Grammars

- **Type 0: Unrestricted or Free**. There are no restrictions on the production rules and thus there will be no constraints on the strings they can produce.
  - These have found little use in pattern recognition because so little information is provided when one knows a particular string has come from a Type 0 grammar, and learning can be expensive.

Types of String Grammars

- **Type 0: Unrestricted or Free**. There are no restrictions on the production rules and thus there will be no constraints on the strings they can produce.
  - These have found little use in pattern recognition because so little information is provided when one knows a particular string has come from a Type 0 grammar, and learning can be expensive.
- **Type 1: Context-Sensitive**. A grammar is called context-sensitive if every rewrite rule is of the form

$$\alpha I \beta \rightarrow \alpha x \beta \tag{1}$$

where both α and β are any strings of intermediate or terminal symbols, I is an intermediate symbol, and x is an intermediate or terminal symbol.

- **Type 2: Context-Free**. A grammar is called context-free if every production rule is of the form

$$I \to x \tag{2}$$

where I is an intermediate symbol and x is an intermediate or terminal symbol.

- **Type 2: Context-Free.** A grammar is called context-free if every production rule is of the form

$$I \rightarrow x \tag{2}$$

where I is an intermediate symbol and x is an intermediate or terminal symbol.

- Any context free grammar can be converted into one in **Chomsky normal form** (CNF), which has rules of the form:

$$A \rightarrow BC \qquad \text{and} \qquad A \rightarrow z \tag{3}$$

where A, B, C are intermediate symbols and z is a terminal symbol.

- **Type 3: Finite State of Regular**. A grammar is called regular if every production rule is of the form

$$\alpha \rightarrow z\beta \qquad \text{OR} \qquad \alpha \rightarrow z \tag{4}$$

where α and β are made up of intermediate symbols and z is a terminal symbol.

- **Type 3: Finite State of Regular**. A grammar is called regular if every production rule is of the form

$$\alpha \to z\beta \qquad \text{OR} \qquad \alpha \to z \tag{4}$$

where α and β are made up of intermediate symbols and z is a terminal symbol.

- These grammars can be generated by a finite state machine.

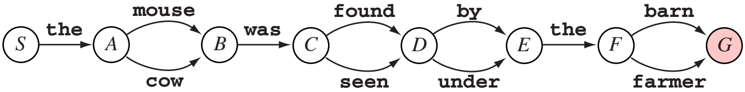


FIGURE 8.16. One type of finite-state machine consists of nodes that can emit terminal symbols ("`the`," "`mouse`," etc.) and transition to another node. Such operation can be described by a grammar. For instance, the rewrite rules for this finite-state machine include $S \to$ `the`A, $A \to$ `mouse`B *OR* `cow`B, and so on. Clearly these rules imply this finite-state machine implements a type 3 grammar. The final internal node (shaded) would lead to the null symbol ϵ. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Parsing – Recognition with Grammars

- Given a test sentence, $\mathbf{x}$, and c grammars, $G_1, G_2, \ldots, G_c$, we want to classify the test sentence according to which grammar could have produced it.

Parsing – Recognition with Grammars

- Given a test sentence, $\mathbf{x}$, and c grammars, $G_1, G_2, \ldots, G_c$, we want to classify the test sentence according to which grammar could have produced it.
- **Parsing** is the process of finding a derivation in a grammar G that leads to $\mathbf{x}$, which is quite more difficult than directly forming a derivation.

Parsing – Recognition with Grammars

- Given a test sentence, $\mathbf{x}$, and c grammars, $G_1, G_2, \ldots, G_c$, we want to classify the test sentence according to which grammar could have produced it.
- **Parsing** is the process of finding a derivation in a grammar G that leads to $\mathbf{x}$, which is quite more difficult than directly forming a derivation.
- **Bottom-Up Parsing** starts with the test sentence $\mathbf{x}$ and seeks to simplify it so as to represent it as the root symbol.

Parsing – Recognition with Grammars

- Given a test sentence, $\mathbf{x}$, and c grammars, $G_1, G_2, \ldots, G_c$, we want to classify the test sentence according to which grammar could have produced it.
- **Parsing** is the process of finding a derivation in a grammar G that leads to $\mathbf{x}$, which is quite more difficult than directly forming a derivation.
- **Bottom-Up Parsing** starts with the test sentence $\mathbf{x}$ and seeks to simplify it so as to represent it as the root symbol.
- **Top-Down Parsing** starts with the root node and successively applies productions from $\mathcal{P}$ with the goal of finding a derivation of the test sentence $\mathbf{x}$.

Bottom-Up Parsing

- The basic approach is to use candidate productions from $\mathcal{P}$ "backwards", which means we want to find the rules whose right hand side matches part of the current string. Then, we replace that part with a segment that could have produced it.
  - This is the general method of the Cocke-Younger-Kasami algorithm.

Bottom-Up Parsing

- The basic approach is to use candidate productions from $\mathcal{P}$ "backwards", which means we want to find the rules whose right hand side matches part of the current string. Then, we replace that part with a segment that could have produced it.
  - This is the general method of the Cocke-Younger-Kasami algorithm.
- We need the grammar to be expressed in Chomsky normal form.
  - Recall, this means that all productions must be of the form $A \rightarrow BC$ or $A \rightarrow z$.

Bottom-Up Parsing

- The basic approach is to use candidate productions from $\mathcal{P}$ "backwards", which means we want to find the rules whose right hand side matches part of the current string. Then, we replace that part with a segment that could have produced it.
    - This is the general method of the Cocke-Younger-Kasami algorithm.
- We need the grammar to be expressed in Chomsky normal form.
    - Recall, this means that all productions must be of the form $A \rightarrow BC$ or $A \rightarrow z$.
- The method will build a **parse table** from the "bottom up."

Bottom-Up Parsing

- The basic approach is to use candidate productions from $\mathcal{P}$ "backwards", which means we want to find the rules whose right hand side matches part of the current string. Then, we replace that part with a segment that could have produced it.
  - This is the general method of the Cocke-Younger-Kasami algorithm.
- We need the grammar to be expressed in Chomsky normal form.
  - Recall, this means that all productions must be of the form $A \rightarrow BC$ or $A \rightarrow z$.
- The method will build a **parse table** from the "bottom up."
- Entries in the table are candidate strings in a portion of a valid derivation. If the table contains the source symbol S, then indeed we can work forward from S to derive the test sentence $\mathbf{x}$.

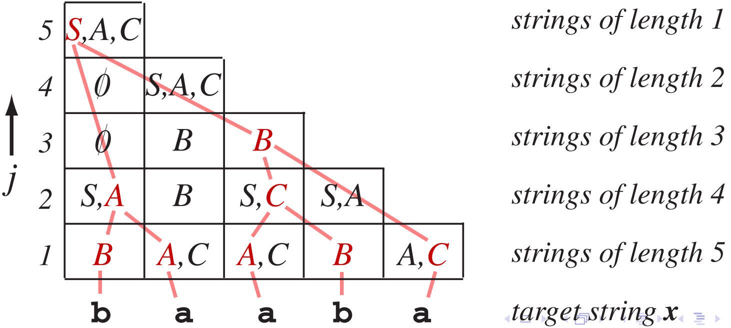
Bottom-Up Parsing

- The basic approach is to use candidate productions from $\mathcal{P}$ "backwards", which means we want to find the rules whose right hand side matches part of the current string. Then, we replace that part with a segment that could have produced it.
  - This is the general method of the Cocke-Younger-Kasami algorithm.
- We need the grammar to be expressed in Chomsky normal form.
  - Recall, this means that all productions must be of the form $A \rightarrow BC$ or $A \rightarrow z$.
- The method will build a **parse table** from the "bottom up."
- Entries in the table are candidate strings in a portion of a valid derivation. If the table contains the source symbol S, then indeed we can work forward from S to derive the test sentence $\mathbf{x}$.
- Denote the individual terminal characters in the string to be parsed as x_i for $i = 1, \ldots, n$.

- Consider an example grammar G with two terminal symbols, $\mathcal{A} = \{a, b\}$, three intermediate symbols, $\mathcal{I} = \{A, B, C\}$, the root symbol S, and four production rules,

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1 : & S \rightarrow AB \text{ OR } BC \\ \mathbf{p}_2 : & A \rightarrow BA \text{ OR } a \\ \mathbf{p}_3 : & B \rightarrow CC \text{ OR } b \\ \mathbf{p}_4 : & C \rightarrow AB \text{ OR } a \end{array} \right\} .$$

- The following is the parse table for the string $\mathbf{x} =$ "baaba".



- If the top cell contains the root symbol S then the string is parsed.

- If the top cell contains the root symbol S then the string is parsed.
- See Algorithm 4 on Pg. 427 DHS for the full algorithm.

- If the top cell contains the root symbol S then the string is parsed.
- See Algorithm 4 on Pg. 427 DHS for the full algorithm.
- The time complexity of the algorithm is $O(n^3)$ and the space complexity is $O(n^2)$ for a string of length n.

- If the top cell contains the root symbol S then the string is parsed.
- See Algorithm 4 on Pg. 427 DHS for the full algorithm.
- The time complexity of the algorithm is $O(n^3)$ and the space complexity is $O(n^2)$ for a string of length n.
- We will not cover grammar inference, learning the grammar.