Dynamic Programming
Edit distance and its variants

Tyler Moore
CSE 3353, SMU, Dallas, TX
Lecture 17

Some slides created by or adapted from Dr. Kevin Wayne. For more information see http://www.cs.princeton.edu/~wayne/kleinberg-tardos. Some code reused from Python Algorithms by Magnus Lie Hetland.

Edit distance

- Misspellings make approximate pattern matching an important problem
- If we are to deal with inexact string matching, we must first define a cost function telling us how far apart two strings are, i.e., a distance measure between pairs of strings.
- The edit distance is the minimum number of changes required to convert one string into another

String edit operations

- We consider three types of changes to compute edit distance:
  - Substitution: Change a single character from pattern \( s \) to a different character in text \( t \), such as changing “shot” to “spot”
  - Insertion: Insert a single character into pattern \( s \) to help it match text \( t \), such as changing “ago” to “agog”.
  - Deletion: Delete a single character from pattern \( s \) to help it match text \( t \), such as changing “hour” to “our”
- This definition of edit distance is also called Levenshtein distance
- Can you think of any other natural changes that might capture a single misspelling?

Edit distance application #1

- Spell checkers identify words in a dictionary with close edit distance to the misspelled word
- But how do they order the list of suggestions?
Edit distance: recursive algorithm design

- **Match:** no substitutions
  \[
  d(s_{i-1}, t_{j-1}) = \begin{cases} 
  0 & \text{if } i = 0 \\
  1 & \text{otherwise}
  \end{cases}
  \]

- **Insertion**
  \[
  d(s_{i-1}, t_{j}) = d(s_{i-1}, t_{j-1}) + 1
  \]

- **Deletion**
  \[
  d(s_{i}, t_{j-1}) = d(s_{i-1}, t_{j-1}) + 1
  \]

Recursive edit distance code

```python
def string_compare(s, t):
    # start by prepending empty character to check 1st char
    s = "" + s
    t = "" + t
    P = {}
    @memo
    def edit_dist(i, j):
        if i == 0: return j
        if j == 0: return i
        # case 1: check for match at i and j
        if s[i] == t[j]:
            c_match = edit_dist(i-1, j-1)
        else:
            c_match = edit_dist(i-1, j-1) + 1
        # case 2: there is an extra character to insert
        c_ins = edit_dist(i, j-1) + 1
        # case 3: there is an extra character to remove
        c_del = edit_dist(i-1, j) + 1
        return min(c_match, c_ins, c_del)
    return edit_dist(len(s)-1, len(t)-1)
```

```
Towards a dynamic programming alternative

- We note that there are only $|s|$ possible values for $i$ and $|t|$ possible values for $j$ when invoking $\text{edit_dist}(i,j)$ recursively.
- This means there are at most $|s| \cdot |t|$ recursive function calls to cache in an iterative version.
- The table is a two-dimensional matrix $C$ where each of the $|s| \cdot |t|$ cells contains the cost of the optimal solution of this subproblem.
- We just need a clever way to calculate the cost for each entry based on only a small subset of already-computed values.

Evaluation order

- To determine the value of cell $(i,j)$ we need three values to already be computed: the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$.
- Any evaluation order with this property will do, including the row-major order used in the upcoming code.
- But there are plenty of other valid orderings.
- Think of the cells as vertices, where there is an edge $(i,j)$ if cell $i$'s value is needed to compute cell $j$. Any topological sort of this DAG provides a proper evaluation order.

Edit distance: dynamic programming code

```python
def iter_string_compare_lists(s, t):
    C, s, t = [], "\"" + s, "\"" + t # prepend empty character for edge case
    C.append(range(len(t)+1)) # initialize cost data structure
    for i in range(len(s)):
        C.append([i+1])
    for i in range(1, len(s)):
        for j in range(1, len(t)):
            # case 1: check for match at i and j
            if s[i] == t[j]:
                c_match = C[i-1][j-1]
            else:
                c_match = C[i-1][j-1]+1
            # case 2: there is an extra character to insert
            c_ins = C[i][j-1]+1
            # case 3: there is an extra character to remove
            c_del = C[i-1][j]+1
            c_min = min(c_match, c_ins, c_del)
            C[i].append(c_min)
    return C[i][j]
```

Edit distance: DP with cost table as dictionary

```python
def iter_string_compare(s, t):
    C, s, t = {}, "\"" + s, "\"" + t # prepend empty character for edge case
    C[0, j] = j
    for i in range(1, len(s)):
        C[i, 0] = i
    for i in range(1, len(s)):
        for j in range(1, len(t)):
            # case 1: check for match at i and j
            if s[i] == t[j]:
                c_match = C[i-1,j-1]
            else:
                c_match = C[i-1,j-1]+1
            # case 2: there is an extra character to insert
            c_ins = C[i,j-1]+1
            # case 3: there is an extra character to remove
            c_del = C[i-1,j]+1
            c_min = min(c_match, c_ins, c_del)
            C[i,j] = c_min
    return C[i,j]
```
### Building edit distance cache

**s:** run  
**t:** drain

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>r</th>
<th>a</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>←1</td>
<td>←2</td>
<td>←3</td>
<td>←4</td>
</tr>
<tr>
<td>r</td>
<td>↑1</td>
<td>↕1</td>
<td>↕1</td>
<td>←2</td>
<td>←3</td>
</tr>
<tr>
<td>u</td>
<td>↑2</td>
<td>↕2</td>
<td>↕2</td>
<td>←3</td>
<td>←3</td>
</tr>
<tr>
<td>n</td>
<td>↑3</td>
<td>↕3</td>
<td>↕3</td>
<td>↕3</td>
<td>↕3</td>
</tr>
</tbody>
</table>

Steps to turn “run” into “drain”
- Insert d
- Keep r
- Substitute a for u
- Insert i
- Keep n

### Edit distance exercises

- **Build cost table by hand following DP algorithm**
  - **s:** bear, **t:** pea
  - **s:** farm, **t:** for
- **Performance cost of DP edit distance**
  - Operations: \( \Theta(|s| \cdot |t|) \)
  - Storage: \( \Theta(|s| \cdot |t|) \)

### Variation of edit distance: approximate substring matching

Suppose we want to find the best close match to a smaller word in a larger string (e.g., find the closest match to “Tulsa” in “SMU Tulda Rice”)

We need to modify our existing code in two ways
- Cost table initialization: all starting costs \( C[0,j] \) should be set to 0
- Return the finishing cell \( C[i,k] \) that minimizes the overall cost

### Substring matching code

```python
def iter_substring_match(s, t):
    C, s, t = {}, "", "", t  # prepend empty character for edge case
    for j in range(len(t)):  # initialize cost data structure
        C[0, j] = 0  # changed: ignore cost of preceding unmatched text
    for i in range(1, len(s)):
        C[i, 0] = i
    for i in range(1, len(s)):  # go through all chars of s
        for j in range(1, len(t)):
            # case 1: check for match at i and j
            if s[i] == t[j]:
                c_match = C[i-1, j-1]
            else:
                c_match = C[i-1, j-1] + 1
            # case 2: there is an extra character to insert
            c_ins = C[i, j-1] + 1
            # case 3: there is an extra character to remove
            c_del = C[i-1, j] + 1
            c_min = min(c_match, c_ins, c_del)
            C[i, j] = c_min

    finj = min([(C[i, k], k) for k in range(1, len(t)-1)])
    return "with edit dist %i, %s morphs into %s finishing at position %i" % finj
```

```bash
13 / 18
```

```bash
14 / 18
```
Excercise: substring matching cache

s: Tulsa

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>U</th>
<th>T</th>
<th>u</th>
<th>l</th>
<th>d</th>
<th>a</th>
<th>-</th>
<th>R</th>
<th>i</th>
<th>c</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
</tr>
<tr>
<td>u</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
<td>↑1</td>
<td>↓0</td>
<td>←1</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
</tr>
<tr>
<td>l</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑2</td>
<td>↑1</td>
<td>↑0</td>
<td>←1</td>
<td>←2</td>
<td>←3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
</tr>
<tr>
<td>s</td>
<td>↑4</td>
<td>↑4</td>
<td>↑4</td>
<td>↑4</td>
<td>↑3</td>
<td>↑2</td>
<td>↑1</td>
<td>↑1</td>
<td>←2</td>
<td>←3</td>
<td>↑4</td>
<td>↑4</td>
<td>↑4</td>
</tr>
<tr>
<td>a</td>
<td>↑5</td>
<td>↑5</td>
<td>↑5</td>
<td>↑5</td>
<td>↑4</td>
<td>↑3</td>
<td>↑2</td>
<td>↑2</td>
<td>↑2</td>
<td>←3</td>
<td>←4</td>
<td>↑5</td>
<td>↑5</td>
</tr>
</tbody>
</table>

Substring ending at position 9 ("Tulda") is the closest substring to "Tulsa"

Variation of edit distance: longest common subsequence

- We might want to find the longest scattered sequence of characters within both strings
- For example, the longest common subsequence of "republican" and "democrat" is "eca"
- To get the longest subsequence, we can still allow insertions and deletions, but substitutions are forbidden
- We can change the edit distance code to behave as before on matches where the last characters are the same, but never select a substitution