Lecture 17:
Edit Distance

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Topic: Problem of the Day

Problem of the Day

Suppose you are given three strings of characters: X, Y, and Z, where $|X| = n$, $|Y| = m$, and $|Z| = n + m$. Z is said to be a *shuffle* of X and Y iff Z can be formed by interleaving the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string.

1. Show that *cchocohilaptes* is a shuffle of *chocolate* and *chips*, but *chocochilatspe* is not.

2. Give an efficient dynamic-programming algorithm that determines whether Z is a shuffle of X and Y. Hint: The values of the dynamic programming matrix you construct should be Boolean, not numeric.

Questions?

Topic: The Gas Station Problem

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different instances of your recurrence is bounded by a polynomial.
3. Specify an order of evaluation for the recurrence so you always have what you need.

The Gas Station Problem

Suppose we are driving from NY to Florida, and we know the positions of all gas stations g_1 to g_n we will pass on route. What is the minimum number of gas stations we will have to fill up at to make it down there?

The m_i be the mile marker where station g_i is located, and R be the driving range of the car on a full tank in miles.

Recursive Idea

Let $G[i]$ be the minimum number of fillups needed to get to gas station g_i.
If we know the best cost to get to all gas stations before i that are in driving range, we can compute $G[i]$:

$$G[i] = \min_{\substack{j < i, \text{ where} \\ ((m_j - m_i) < R)}} G[j] + 1$$

The boundary case is $G[1] = 0$.

Observations

- This gives an $O(n^2)$ algorithm to minimize the number of stations.
- This problem *could* have been solved as BFS/shortest path on an unweighted directed graph.
- Many dynamic programming algorithms are in fact shortest path problems on the right DAG, in disguise.
- The dynamic programming formulation can be extended (with additional state) to finding the cheapest trip if gas stations charge different prices.

Minimum Average Station Price

Suppose we know the price of gas $p(s)$ at every station s we will pass.

What is the set of stations we should stop at to minimize average price?

Here we are not worrying about how much gas we buy at each stop, just the per-gallon price.

Recursive Idea

Let $G[i,k]$ be the minimum total per gallon price needed to get to gas station g_i in exactly k stops.

$$G[i,k] = \min_{\substack{j < i, \text{ where} \\ ((m_j - m_i) < R)}} G[j,k-1] + p(j)$$

The boundary case is $G[1,0] = 0$.
Ultimately we want the k that minimize $G[n,k]/k$.
This doesn't quite capture the desire to put more gas in the car at cheaper stations.

Cheapest Filling Schedule

Let's say that we must buy gas in integer numbers of gallons at a time.

What is the filling schedule which gets us to the destination at lowest total cost?

Recursive Idea

Let $G[i,k]$ be the minimum cost possible to get to gas station g_i arriving with k gallons of gas left in the tank.

$$G[i,k] = \min_{c} \min_{\substack{j < i \\ ((m_j - m_i) < R_c)}} G[j,c] + p(j) \cdot (k - c')$$

$$G[i,k] = \min_{c=0}^{k-1} G[i,c] + p(i) \cdot (k - c)$$

Here c' is the amount of gas remaining after starting with c gallons and driving from j to i.
The boundary case is $G[1,0] = 0$.

Questions?

Topic: Problem of the Day

Problem of the Day

A certain string processing language allows the programmer to break a string into two pieces. Since this involves copying the old string, it costs n units of time to break a string of n characters into two pieces.

Suppose a programmer wants to break a string into many pieces. The order in which the breaks are made can affect the total amount of time used.

For example suppose we wish to break a 20 character string after characters 3,8, and 10:

- If the breaks are made in left-right order, then the first break costs 20 units of time, the second break costs 17 units of time and the third break costs 12 units of time, a

total of 49 steps.

- If the breaks are made in right-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, a total of only 38 steps.

Give a dynamic programming algorithm that, given the list of character positions after which to break, determines the cheapest break cost in $O(n^3)$ time.

Questions?

Topic: Edit Distance

Edit Distance

Mispellings 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.

A reasonable distance measure minimizes the cost of the *changes* which have to be made to convert one string to another.

String Edit Operations

There are three natural types of changes:

- *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".

Recursive Algorithm

We can compute the edit distance with recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.

If we knew the cost of editing the three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly.

We *can* learn this cost, through the magic of recursion:

Recurrence Relation

Let $D[i, j]$ be the minimum number of changes to convert the first i characters of string S into the first j characters of string T.

Then $D[i, j]$ is the minimum of:

- $D[i-1, j-1]$ if $S[i] = T[j]$
- $D[i-1, j-1] + 1$ if $S[i] \neq T[j]$
- $D[i, j-1] + 1$ for an insertion into S
- $D[i-1, j] + 1$ for a deletion from S

Recursive Edit Distance Code

```
#define MATCH      0       /* enumerated type symbol for match */
#define INSERT     1       /* enumerated type symbol for insert */
#define DELETE     2       /* enumerated type symbol for delete */


int string_compare_r(char *s, char *t, int i, int j) {
    int k;              /* counter */
    int opt[3];         /* cost of the three options */
    int lowest_cost;    /* lowest cost */

    if (i == 0) {       /* indel is the cost of an insertion or deletion */
        return(j * indel(' '));
    }

    if (j == 0) {
        return(i * indel(' '));
    }
                        /* match is the cost of a match/substitution */

    opt[MATCH]  = string_compare_r(s,t,i-1,j-1) + match(s[i],t[j]);
    opt[INSERT] = string_compare_r(s,t,i,j-1) + indel(t[j]);
    opt[DELETE] = string_compare_r(s,t,i-1,j) + indel(s[i]);

    lowest_cost = opt[MATCH];
```

```
    for (k = INSERT; k <= DELETE; k++) {
        if (opt[k] < lowest_cost) {
            lowest_cost = opt[k];
        }
    }

    return(lowest_cost);
}
```

Speeding it Up

This program is absolutely correct but takes exponential time because it recomputes values again and again and again!

But there can only be $|s| \cdot |t|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the parameters of recursive calls.

By storing the values for each of these (i, j) pairs in a table, we can avoid recomputing them and just look them up as needed.

The Dynamic Programming Table

The table is a two-dimensional matrix m where each of the $|s| \cdot |t|$ cells contains the cost of the optimal solution of this subproblem, as well as a parent pointer explaining how we got to this location:

```
typedef struct {
    int cost;               /* cost of reaching this cell */
    int parent;             /* parent cell */
} cell;


cell m[MAXLEN+1][MAXLEN+1];     /* dynamic programming table */
```

Differences with Dynamic Programming

The dynamic programming version has three differences from the recursive version:

- First, it gets its intermediate values using table lookup instead of recursive calls.
- Second, it updates the `parent` field of each cell, which will enable us to reconstruct the edit-sequence later.
- Third, it is instrumented using a more general `goal_cell()` function instead of just returning `m[|s|][|t|].cost`. This will enable us to apply this routine to a wider class of problems.

We assume that each string has been padded with an initial blank character, so the first real character of string `s` sits in `s[1]`.

Evaluation Order

To determine the value of cell (i, j) we need three values sitting and waiting for us, namely, 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 this program.

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.

Dynamic Programming Edit Distance

```
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;    /* counters */
    int opt[3];             /* cost of the three options */

    for (i = 0; i <= MAXLEN; i++) {
        row_init(i, m);
        column_init(i, m);
    }

    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i-1][j-1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j-1].cost + indel(t[j]);
            opt[DELETE] = m[i-1][j].cost + indel(s[i]);

            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++) {
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
            }
        }
```

```
    }

    goal_cell(s, t, &i, &j);
    return(m[i][j].cost);
}
```

Example

Below is an example run, showing the cost and parent values turning “thou shalt” to “you should” in five moves:

	T		y	o	u	-	s	h	o	u	l	d
P	pos	0	1	2	3	4	5	6	7	8	9	10
:		**0**	1	2	3	4	5	6	7	8	9	10
t:	1	1	1	2	3	4	5	6	7	8	9	10
h:	2	2	**2**	2	3	4	5	5	6	7	8	9
o:	3	3	3	**2**	3	4	5	6	5	6	7	8
u:	4	4	4	3	**2**	3	4	5	6	5	6	7
-:	5	5	5	4	3	**2**	3	4	5	6	6	7
s:	6	6	6	5	4	3	**2**	3	4	5	6	7
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6
a:	8	8	8	7	6	5	4	3	3	**4**	5	6
l:	9	9	9	8	7	6	5	4	4	4	**4**	5
t:	10	10	10	9	8	7	6	5	5	5	5	**5**

The edit sequence from “thou-shalt” to “you-should” is `DSMMMMMISMS`

Questions?

Topic: Variant of Edit Distance

Reconstructing the Path

Dynamic programming solutions are described by paths through the dynamic programming matrix, starting from the initial configuration (the empty strings $(0, 0)$) down to the final goal state (the full strings $(|s|, |t|)$).

Reconstructing these decisions is done by walking backward from the goal state, following the `parent` pointer to an earlier cell. The `parent` field for `m[i,j]` tells us whether the transform at (i, j) was `MATCH`, `INSERT`, or `DELETE`.

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

Reconstruct Path Code

```
void reconstruct_path(char *s, char *t, int i, int j,
                      cell m[MAXLEN+1][MAXLEN+1]) {
    if (m[i][j].parent == -1) {
        return;
    }

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1, m);
        match_out(s, t, i, j);
        return;
    }

    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j-1, m);
        insert_out(t, j);
        return;
    }

    if (m[i][j].parent == DELETE) {
        reconstruct_path(s, t, i-1, j, m);
        delete_out(s, i);
        return;
    }
}
```

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

Customizing Edit Distance

- *Table Initialization* – The functions `row_init()` and `column_init()` initialize the zeroth row and column of the dynamic programming table, respectively.
- *Penalty Costs* – The functions `match(c,d)` and `indel(c)` present the costs for transforming character c to d and inserting/deleting character c. For edit distance, `match` costs nothing if the characters are identical, and 1 otherwise, while `indel` always returns 1.

- *Goal Cell Identification* – The function `goal_cell` returns the indices of the cell marking the endpoint of the solution. For edit distance, this is defined by the length of the two input strings.
- *Traceback Actions* – The functions `match_out`, `insert_out`, and `delete_out` perform the appropriate actions for each edit-operation during traceback. For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application.

Substring Matching

Suppose that we want to find where a short pattern s best occurs within a long text t, say, searching for "Skiena" in all its misspellings (Skienna, Skena, Skina, ...).

Plugging this search into our original edit distance function will achieve little sensitivity, since the vast majority of any edit cost will be that of deleting the body of the text.

We want an edit distance search where the cost of starting the match is independent of the position in the text.

Likewise, the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text.

Substring Matching Customizations

```
void row_init(int i, cell m[MAXLEN+1][MAXLEN+1]) {
    m[0][i].cost = 0;        /* NOTE CHANGE */
    m[0][i].parent = -1;     /* NOTE CHANGE */
}


void goal_cell(char *s, char *t, int *i, int *j) {
    int k;    /* counter */

    *i = strlen(s) - 1;
    *j = 0;

    for (k = 1; k < strlen(t); k++) {
        if (m[*i][k].cost < m[*i][*j].cost) {
            *j = k;
        }
    }
}
```

Longest Common Subsequence

The *longest common subsequence* (not substring) between "democrat" and "republican" is *eca*.

A common subsequence is defined by all the identical-character matches in an edit trace. To maximize the number of such traces, we must prevent substitution of non-identical characters.

```
int match(char c, char d) {
    if (c == d) {
        return(0);
    }
    return(MAXLEN);
}
```

Maximum Monotone Subsequence

A numerical sequence is *monotonically increasing* if the ith element is at least as big as the $(i-1)$st element.

The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence.

Thus a longest increasing subsequence of "243517698" is "23568."

Reduction to LCS

In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order.

Any common sequence of these two must (a) represent characters in proper order in S, and (b) use only characters with increasing position in the collating sequence, so the longest one does the job.

Questions?