CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC17: EDIT DISTANCE

Lecture slide courtesy of Prof. Steven Skiena

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.
- 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:

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(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) return(j * indel(' '));
    if (j == 0) return(i * indel(' '));

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

Correct but very slow

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 $|s|*|t|$ 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 $|s| * |t|$ 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;

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

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

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

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

```
int string_compare(char *s, char *t)
{
        int i,j,k; /* counters */
        int opt[3]; /* cost of the three options */
        for (i=0; i<MAXLEN; i++) {
                row_init(i);
                column_init(i);
        }
        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 not” to “you should not” in five moves:

P	T pos	0	y 1	o 2	u 3	- 4	s 5	h 6	o 7	u 8	l 9	d 10	- 11	n 12	o 13	t 14
:		$\boxed{0}$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	$\boxed{\mathbf{1}}$	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	$\boxed{\mathbf{2}}$	$\boxed{\mathbf{3}}$	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

- The edit sequence from “thou-shalt-not” to “you-should-not”
- is DSMMMMMISMSMMMM

RECONSTRUCTING THE PATH

- Solutions to a given dynamic programming problem are described by paths through the dynamic programming matrix, starting from the initial configuration (the pair of empty strings (0, 0)) down to the final goal state (the pair of 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.

RECONSTRUCT PATH CODE

- However, clever use of recursion can do the reversing for us:

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

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

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

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

CUSTOMIZING EDIT DISTANCE

- *Table Initialization*
  - The functions *row_init()* and *column_init()* initialize the zeroth row and column of the dynamic programming table, respectively.

```
row_init(int i)
{
        m[0][i].cost = i;
        if (i>0)
                m[0][i].parent = INSERT;
        else
                m[0][i].parent = -1;
}
```

```
column_init(int i)
{
        m[i][0].cost = i;
        if (i>0)
                m[i][0].parent = DELETE;
        else
                m[i][0].parent = -1;
}
```

- *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.

```
goal_cell(char *s, char *t, int *i, int *j)
{
        *i = strlen(s) - 1;
        *j = strlen(t) - 1;
}
```

- *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,
  - so that a match in the middle is not prejudiced against.
- 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.

CUSTOMIZATIONS FOR SUBSTRING MATCHING

```
row_init(int i)
{
        m[0][i].cost = 0; /* note change */
        m[0][i].parent = -1; /* note change */
}

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.
- We get the alignment we want by changing the match-cost function to make substitutions expensive:

```
int match(char c, char d)
{
	if (c == d) return(0);
	else 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.