Dynamic Programming: The Edit Distance Problem

http://books.google.com/books?id=IcnSCDcDocMC

Chapter 11

Dynamic Programming

- What do you remember about Dynamic Programming?

Greedy vs. Dynamic

- *Greedy* algorithms focus on making the best local choice at each decision point
- Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency)

Dynamic Programming

- Dynamic programming algorithms are defined by recursive algorithms/functions that describe the solution to the entire problem in terms of solutions to smaller problems
- Efficiency in any such recursive algorithm requires storing enough information to avoid repeating computations we have done before

Dynamic Programming

- Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results
- The trick is to see that the naive recursive algorithm repeatedly computes the same subproblems over and over and over again. If so, storing the answers to them in a table instead of recomputing can lead to an efficient algorithm
- Thus we must first hunt for a correct recursive algorithm - later we can worry about speeding it up by using a results matrix

Edit Distance

- Levenshtein (1966) introduced edit distance between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other

Edit Distance

- Misspellings and changes in word usage (``Thou shalt not kill'' morphs into ``You should not murder.'') 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

Edit Distance

- There are three natural types of changes:
  - *Substitution* - Change a single character from pattern to a different character in text , such as changing ``shot'' to ``spot''.
  - *Insertion* - Insert a single character into pattern to help it match text , such as changing ``ago'' to ``agog''.
  - *Deletion* - Delete a single character from pattern to help it match text , such as changing ``hour'' to ``our''.

Examples

- What is the minimum distance between:
  - cat and cast
  - brain and barn

Applications

- File Revision: The Unix command diff
- Spelling Correction
- Plagiarism Detection
- Molecular Biology: distance between two DNA sequences (alphabet is A, C, G, T)

Output

- 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 Algorithm

```
#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 );
}
```

Helper Functions

```
int match(char c, char d)
{
  if (c == d) return 0;
  else return 1;
}
int indel(char c)
{
  return 1;
}
```


Verifying string_compare

- s = cast
- t = cat
- i = 4
- j = 3

What is the Problem then?

Speeding it up

- The important observation is that 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.

Dynamic Programming Table

```
typedef struct
{
  int cost;
  int parent;
} cell;
cell m[MAXLEN+1][MAXLEN+1];
```


DP Edit Distance

```
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;
                                }
                        }
```

Helper Function

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


Helper Function

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

Helper Function

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


Example

	-	C	A	T
-				
C				
A				
S				
T				

Example

- Where is the shortest distance?
- How can we construct the path?

Reconstructing the Path

```
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;
        }
}
```

Your Turn

- What is the edit distance between the following two DNA sequences:
  - CTACCG
  - TACATG
- How can one be converted to the other?