Algorithmic Bioinformatics DD2450, spring 2010, Lecture 1

Lecturer Jens Lagergren
Several current and previous students
will be acknowledged in a separate document.

March 30, 2010

1 Dynamic programming

Dynamic programming is a technique for solving optimization problems that have a structure containing overlapping subproblems. The idea in dynamic programming is to only compute solutions to subproblems once and store them so they can be used again if required. Solving a problem using dynamic programming typically follows these steps.

1. Given a problem instance, identify the subproblem instances.
2. Construct a recursion expressing the optimal value of the problem instance in terms of the optimal values of the subproblem instances.
3. Identify base cases for which the optimal value is known.
4. Compute and store the values of the optimal solutions of subproblem instances. Iterate over the subproblem instances in such an order that the optimal values required by the recursion have already been computed when a subproblem instance is reached.
5. Locate where the optimal value will end up and optionally also construct the optimal solution using the optimal values of the subproblems.

1.1 Longest common subsequence

A nice example of dynamic programming is finding the longest common subsequence (LCS) of two strings.

Definition 1. *If $X = x_1, \ldots, x_m$ is a string and $i_1, \ldots, i_k$ are integers such that $1 \leq i_1 < i_2 < \cdots < i_k \leq m$, then $x_{i_1}, \ldots, x_{i_k}$ is a subsequence of X.*

Example 1. *acba is a subsequence of the string $X = abcbba$.*

Definition 2. *A longest common subsequence (LCS) of strings X and Y is a string Z such that:*

1. *Z is a subsequence of X.*
2. *Z is a subsequence of Y.*
3. *Z has the maximal length among all strings satisfying conditions 1 and 2.*

1.2 Dynamic programming solution to the LCS problem

LCS can be formulated as a computational problem in the following manner

Input: Strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$.

Output: An LCS of X and Y.

A naive solution would be to perform an exhaustive search, which would have exponential time complexity. Instead this can be done in polynomial time with a dynamic programming algorithm.

Definition 3. *Let $X^i = x_1, \ldots, x_i$ be the prefix of X with length i.*

Observations

1. If $Z = z_1, \ldots, z_k$ is an LCS of X^i and Y^j ($i \geq 1, j \geq 1$), the following holds
   (a) $(x_i = y_j = z_k) \vee (x_i \neq y_j \wedge z_k \neq x_i) \vee (x_i \neq y_j \wedge z_k \neq x_i)$
   (b) if $x_i = y_j = z_k$, then Z^{k-1} is an LCS of X^{i-1} and Y^{j-1}.
   (c) if $x_i \neq y_j \wedge z_k \neq x_i$, then Z is an LCS of X^{i-1} and Y^j.
   (d) if $x_i \neq y_j \wedge z_k \neq y_j w$, then Z is an LCS of X^i and Y^{j-1}.
2. Let the matrix $c(i,j)$ represent the length of an LCS of X^i and Y^j then $c(i,j)$ can be computed recursively as

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & \text{if } x_i = y_j \\ \max\{c(i-1,j), c(i,j-1)\} & \text{otherwise} \end{cases}$$

3. The LCS of $\emptyset$ (the empty string) and any string is $\emptyset$.
4. Computing $c(i,j)$ requires the values $c(i-1,j-1)$, $c(i-1,j)$ and $c(i,j-1)$. They will always be available if each row is computed from left to right starting at $c(1,1)$.
5. The length of the LCS is the value at $c(m,n)$. By using back-pointers describing which of the three cases that led to the value in each element $c(i,j)$, it is also possible to extract the corresponding LCS.

Algorithm 1 LCS of two strings using dynamic programming

```
Input: Strings $X$ and $Y$ of length $m$ and $n$ respectively.
Output: LCS of $X$ and $Y$.
  // Initialize base cases
  for $i = 0$ to $m$ do
    $C[i, 0] = 0$
  end for
  for $j = 0$ to $n$ do
    $C[0, j] = 0$
  end for
  // Compute optimal values for other cases, save backtrack information
  for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
      if $X[i] == Y[j]$ then
        $C[i, j] = C[i-1, j-1] + 1$
        $B[i, j] = \nwarrow$
      else if $C[i, j-1] \geq C[i-1, j]$ then
        $C[i, j] = C[i, j-1]$
        $B[i, j] = \leftarrow$
      else
        $C[i, j] = C[i-1, j]$
        $B[i, j] = \uparrow$
      end if
    end for
  end for
  // Construct optimal solution
  $LCS = \emptyset$
  $i = m$
  $j = n$
  while $i \neq 0$ and $j \neq 0$ do
    if $B[i, j] = \nwarrow$ then
      prepend $X[i]$ to $LCS$
      $(i, j) = (i-1, j-1)$
    else if $B[i, j] = \leftarrow$ then
      $(i, j) = (i, j-1)$
    else
      $(i, j) = (i-1, j)$
    end if
  end while
  return $LCS$
```

Example 2. *The following is the optimal value matrix when computing the LCS of $X = caabb$ and $Y = acbca$. A traceback is marked with back-pointers, showing that one (of several possible) LCS is ab. The characters chosen to form the subsequence are underlined.*

$$
\begin{array}{c|ccccccccccc}
c(i,j) & & & \underline{a} & & c & & \underline{b} & & c & & a \\
\hline
 & 0 & & 0 & & 0 & & 0 & & 0 & & 0 \\
c & 0 & & 0 & & 1 & & 1 & & 1 & & 1 \\
a & 0 & & 1 & & 1 & & 1 & & 1 & & 2 \\
 & & \nwarrow & & & & & & & & & \\
\underline{a} & 0 & & 1 & & 1 & & 1 & & 1 & & 2 \\
 & & & \uparrow & & & & & & & & \\
b & 0 & & 1 & \leftarrow & 1 & & 2 & & 2 & & 2 \\
 & & & & & & \nwarrow & & & & & \\
\underline{b} & 0 & & 1 & & 1 & & 2 & \leftarrow & 2 & \leftarrow & 2
\end{array}
$$

Complexity of the algorithm

Time: $O(mn)$, since there are $(m + 1)(n + 1)$ cells and each is computed in constant time.

Memory: $O(mn)$, since each cell uses constant memory. To get only the length of the LCS the memory complexity can be reduced to $O(\min(m, n))$ by saving only the previous line of the matrix $c(i, j)$.