CS 141 Final Report

The Team

Stephen Bolaris -- sbola004 | Michael Romero -- mrome007 | Jon Deans -- jdean007
Aaron Gonzales -- agonz049 | Nathanial Hapeman -- nhape001

Name

dnasearch

Syntax

```
dnasearch <database.fa> <query.fa> [number of results] [alignment flag]
dnasearch [d = <database.fa>] [q = <query.fa>] [n = # results] [a = alignment flag]
```

Description

The program "dnasearch" can read in a database file of one or more Fasta sequences and perform a global alignment of an input Fasta sequence. The user can specify the number of sequences to output, and whether or not to print the alignments or just the scores.

Options

d	The name of a Fasta formatted file containing one or more sequences align against.
q	The name of a Fasta formatted file containing the input sequence to align with.
n	The number of results to show.
a	Binary flag. 1 if alignments should be printed, 0 if alignments should be omitted.

Example dnasearch d = bact.txt q = mydna n = 27 a = 0

Documentation

Sorting

For sorting we implemented a doubly linked list to hold the top K scores. Additionally, the list included a tail pointer. As we added nodes to the list we kept track of the minimum and assigned it to the tail pointer. This allowed us to only add nodes that were above the minimum cutting down on the time it took to run the algorithm. The purpose of having a doubly linked list was to delete the tail node when a node was added to the list. If it was implemented as a singly linked list, we would have to traverse to the end of the list in order to remove and properly delete the tail node. This also allowed us to keep the list to a minimum of 100 nodes. We had other ways of implementing this but this way seemed efficient since it only went through the main list once. This also could have been implemented with an array and it would have been fairly effective since you can perform binary search but the drawback is that an array would have to shift all the nodes to the right if a higher value was added to the list. This coupled with the log(n) search makes it a poor candidate.

Similarity
For computing similarity we implemented a function based on the recurrence relation:

```
maxAlignment(i,j) = max(maxAlignment(i-1,j)+(-0.2or-0.05),
                        maxAlignment(i,j-1)+(-0.2or-0.05),
                        maxAlignment(i-1,j-1)+match/mismatch score).
```

and used iterative dynamic programming in order to optimize the function. The algorithm constructs a table that contains Nodes that are assigned scores based on the recurrence relation above and edge pointers that point to Nodes that resulted in the current score.

Traceback
To traceback the sequence of the alignment we merely follow the traceback pointers from the last node in the sequence to the nodes which created the alignment score we are reporting. If we traceback diagonally we print a match / mis-match, if we travel horizontally we insert a gap into the query sequence, otherwise if travelling vertically we insert a gap in the reference.

function: void thePaths(.....)
input: Node * start, char * ref, char * que
output: void

The traceback function thePaths(...) is a simple tree traversal using Depth First Search as a basis of traversing the Nodes. The function takes in a Node pointer, which is used as the starting point for the traversal, and two char arrays representing the reference and query strings. Then the function goes through the Depth First Search Process of traversing a graph by first pushing a node into the stack then looping until the Stack is empty and checking the current Node's neighbors and pushing them onto the Stack. For this implementation there is no need to mark or check nodes as visited since the table is basically a tree. In the loop the function first creates two integers 'rx' and 'qx that represent positions in the reference and query string respectively, then a temporary pointer to the Node object in the traceBackNode object 'curr', and then two string objects, 'currR' and 'currQ', representing the current alignment between the reference and query so far. Before all the neighbors of the current node are pushed onto the Stack the function first checks whether the neighboring node was a result of a deletion, insertion, or a match by accessing information through its edges. If the neighbor was a result of a match two string objects are created, the first holds a character from the reference string concatenated by the current reference alignment string so far represented by 'currR', the seconds holds a character from the query string concatenated by the current query alignment string so far represented by 'currQ'. If the neighbor was a result of a deletion two string objects are created, the first holds a space concatenated by the current reference alignment string so far represented by 'currR', the seconds holds a character from the query string concatenated by the current query alignment string so far represented by 'currQ'. If the neighbor was a result of an insertion two string objects are created, the first holds a character from the reference string concatenated by the current reference alignment string so far represented by 'currR', the seconds holds a space concatenated by the current query alignment string so far represented by 'currQ'. A traceBackNode object, containing the neighboring node and the two string objects representing the current reference and query alignments, is then pushed onto the stack. Then the process repeats until the stack is empty. In order to print out the reference and query alighments, within the loop the function checks whether the current node 'curr' have no neighbors, this signifies that it is the end of a path, if 'curr' has no neighbors then the function prints out the two string objects 'currR' and 'currQ' which hold the current alignment between the reference and query strings.

function: void maxAlign(.....)
input: Graph &g, vector<Node*> &lastRow, vector<Node*> topRow, vector<Node*> fcol, char * align
output: void

The function maxAlign constructs the table for dynamic programming based on the recurrence relation:

```
        maxAlignment(i,j) = max(maxAlignment(i-1,j)+(-0.2or-0.05),
                                maxAlignment(i,j-1)+(-0.2or-0.05),
                                maxAlignment(i-1,j-1)+match/mismatch score).
```

Each cell of the table contained a Node object that held an integer that is assigned the score, another integer that represented a position on the reference string (used for the traceback function), and a vector of Edge pointers used to indicate which Nodes lead to the current score. The first row and column of the table were pre-calculated and passed in the function as vectors in order to access the first solutions for the recurrence relation above. The function then iterates from i = 1 through the length of the reference string, then iterates from j = 1 through the length of the query string. At each position (i,j) a score is calculated based on the recurrence relation above, the penalty -0.2 or -0.05 is determined by checking whether a previous deletion or insertion happened at position (i-1,j) and (i,j-1) respectively, if there was then the penalty score is -0.05, else -0.2. The match/mismatch score is determined through a separate table of scores. After determining the score the function then checks where it got that score. New edges are created and pointed to the nodes that lead to the current score, so there is a maximum of three edges that'll possibly point to position (i-1,j), (i,j-1), and (i-1,j-1) respectively. The current position i is also stored in the node, this is used for the traceback function.

function: int scores(...)
input: char ref, char qry
output: integer

The function scores(...) returns an integer based on the costs of matching char ref, and char qry. The integer values returned are based off of the match/mismatch score table provided in the project 3 manual.