15-210 Assignment 09

Karan Sikka
ksikka@cmu.edu
Section C
November 30, 2012

1: Task 2.2

Table values

```
5  3  2  4
9  3  9  10
5  10 4  11
9  11 10 12
```

2: Task 2.3

The lowest cost seam consists of the points (3,0),(2,0),(1,1),(0,2)

3: Task 2.4

$$m(i,j) = g(i,j) + min(m(i-1,j-1), m(i-1,j), m(i-1,j+1))$$

Algorithm:
Generate the matrix of gradients in $\Theta(mn)$ work and $\Theta(1)$ span using `generateGradients` since you must at the least and at most do a constant amount of work at each matrix entry, and there are no dependencies. Create a mxn 2d sequence filled with 0's. This will contain the matrix shown in part 2.2. We will populate this table sequentially by rows.

Set the top row equal to the top row of the gradient matrix, since the seam costs start out as 0. Then, for each cell in the next row, set its value to $g(i,j) + min(m(i-1,j-1), m(i-1,j), m(i-1,j+1))$. Iterate sequentially. Populating this table takes $O(mn)$ work and $O(n)$ span.

To find the minimum-cost seam, first find the index of the cell with the lowest value in the bottom row of the seam-cost matrix. Add that index to the head of a linked-list which will store the seam. Finding this cell takes $O(m)$ work and $O(\log(m))$ span, since it can be implemented as a reduce.

Consider the row above the bottom row. Find the cell with the minimum value either directly above the first cell added to the seam, above and to the right, or above and to the left. Add the index of that cell to the seam list and iterate until all rows of the seam-cost matrix have been explored. This step takes $O(n)$ since it does constant work at each of n rows, and it has $O(n)$ span since it iterates accross the n rows sequentially.

Finally, convert the list of indices comprising the seam into a sequence. This has work and span of $O(n)$.

The work of the algorithm is $\Theta(mn)$, and the span is $\Theta(m+n)$. The step with the most work is the generation of the table of costs. The step with the largest span is the either the table generation (span of m) or the backtracking to find the seam (span of n).

4: Task 3.1

7

5: Task 3.2

6

6: Task 3.3

$C(\emptyset, s_j) = 1$
$C(,s_0) = 0$

$C(S_i, s_j) =$
If the last character in S and s are equal, then $C(S_i, s_{j-1}) + C(S_{i-1}, s_{j-1})$ else $C(S_i, s_{j-1})$

What this conceptually does is compare the last characters in S and s. If they are not equal, shorten the string and try again. If they are equal, shorten the string AND shorten the string and subsequence. In the base case, 1 is added for successfully seeing the entire subsequence in the string.

This can be a top-down dynamic programming solution if you memoize the function.

7: Task 3.4

The number of subproblems in the DAG will be $O(nm)$. This is clear from the subproblem definition, which is a tuple of i and j, which are in ranges of size n and m respectively.

The memoization overhead can be constant, since you could store and lookup the pre-computed values in a 2d array which would function as a table keyed on (i, j).

The span of the DAG is $O(m)$, since the length of the string decreases by one every recursive call, so there are at most m recursive calls.

8: Task 3.5

"dogged" would greedily select "dog" to be a word, but "ged" is not a word, so the naive greedy algorithm would fail here.

9: Task 3.6

Let $S_{i,j}$ be the substring of S from index i to index j, including the character at i and excluding the character at j. Returns an empty string if indices are out of range or invalid.

Given a string S, let $B(S_{i,j})$ be true iff $S_{i,j}$ can be parsed as a sequence of valid words. Note that B is true for an empty string.

10: Task 3.7

```
B(s) =
  if length s = 0 then return true
  for i from 1 to k:
    if D[s_{0,i}] and B(S_{i,length(s)}) then
      return true
  return false
```

11: Task 3.8

There are at most $O(nk)$ vertices in the DAG, because that's roughly how many substrings of upto length k there are (if you draw out the memoization table). The longest path is the same since there is no parallelization.

The work would be $O(nk^2)$ due to the dictionary lookups. The span would be the same since this algorithm has no parallelization.

12: Task 3.9

Redefine B to return a list of indices option, indicating the indices of the last letters of the valid words, or NONE if the string cannot be parsed into valid words. If the string cannot be broken down into valid words, the returned list will be empty. Spaces should be inserted in all locations except for the last index in the returned list. To handle this, you could define a wrapper B', which returns all but the last element of the list returned by B.

The code could simply be modified by changing the return type of B and making small adjustments as follows:

```
B(s) =
  if length s = 0 then return SOME []
  for i from 1 to k:
    if D[s_(0,i)] and Option.isSome B(S_{i,length(s)}) then
      return SOME (i :: Option.valOf B(S_{i,length(s)}))
  return NONE

B'(s) = case B(s) of NONE => NONE |
        SOME [] => SOME [] |
        SOME l => SOME (dropLastElem l)
```