2 File Challenge

Hugh O'Reilly

March 12, 2018

a) First, we will want to iterate through each of the files and create two mappings such that one of the mappings maps the unique identifiers in one file to their space-delimited fields and the other mapping does the same thing for the other file. Then, we will iterate through the keys of one of the mappings, and if any of the keys match a key in the other mapping, we will create a new mapping where the common unique identifier is the key and the value is a string that consists of the combined space-delimited fields. Finally, we will output a file formatted with this third mapping.

b)

Algorithm 1: TwoFileChallenge.

Input: Two files $file1$ and $file2$ where each line of each file consists of a unique identifier $unique\_identifier$ followed by space-delimited fields. For each file, a unique identifier is present only once. The list of unique identifiers in the first file may not be the same as in the second file.

Output: One file that contains, on each line, a merged line from the input where the unique identifiers match.

```
1  Initialize map1 to be an empty mapping;
2  Initialize map2 to be an empty mapping;
3  foreach line in file1 do
4      map1_unique_identifier ← following space-delimited fields;
5  foreach line in file2 do
6      map2_unique_identifier ← following space-delimited fields;
7  Initialize map3 to be an empty mapping;
8  foreach u ∈ map1 do
9      // note: the set of all u ∈ map1 represents the set of all keys in map1
10     foreach v ∈ map2 do
11         // note: the set of all v ∈ map2 represents the set of all keys in map2
12         if u = v then
13             map3_u ← map1_u concatenated with map2_v;
14 Initialize file3 to be an empty file;
15 foreach w ∈ map3 do
16     // note: the set of all w ∈ map3 represents the set of all keys in map3
17     Concatenate w with map3_w, separated by a space, and place it on its own line in
        file3;
18 return file3;
```

c) The main pro of this approach is that it's thorough, and it will find and match all of the proper unique identifiers. However, the main con is that it has computational limitations proportional to its input size. For example, suppose m is the number of lines in $file1$ and n is the numbers of lines in $file2$. Then, in the worst case, this algorithm runs in $O(mn)$. (Note: the worst case occurs when none of the unique identifiers in the first file match the unique identifiers in the second.) As the sizes of the two files grow, this algorithm could become quite inefficient. Should m and n be very large, I would sort the files alphabetically according the first letters of their unique identifiers. This way, I could use binary search instead of linear search, making my algorithm more efficient.