Description

The code written for this task is quite flexible and we can parametrize it for execution time or for result quality. The overall algorithm (pseudo code) is given below.

```
Function: getMarkedTable( [in] rows ) returns html table (of match rows)
  for each row r
    company_name = r[“Company”];
    for each keyword k in company_name with length > alpha
      if(k exists in hash)
        increment hash[k].count
        insert (hash[k].company_indices_of_rows, r.index)
        optimized_hash[k] = hash[k];
      else
        hash[k] = new keyword( count = 1, company_index = r.index );
      end if
  end for
  marked_rows = new hash();
  for each keyword k in optimized_hash
    if(optimized_hash[k].count < beta)
      for any pair (a, b) in optimized_hash[k].company_indices_of_rows
        if(similarity(rows[a][“Company”], rows[b][“Company”]) > 80%)
          if(match_addresses(rows[a], rows[b]) == true)
            marked_rows[a] = b;
            marked_rows[b] = a;
          end if
        else if (similarity(rows[a][“Company”], rows[b][“Company”]) > 50%)
          if(user prompt is enable)
            Ask user if rows[a][“Company”] and rows[b][“Company”] match?
            If given yes
              if(match_addresses(rows[a], rows[b]) == true)
```

```
                        marked_rows[a] = b;
                        marked_rows[b] = a;
                    end if
                end if
            end if
        end if
    end for
  end if
 end for
 html_table = new table;
 for each pair (current_row_index => match_row_index) in marked_rows
    if( strtolower(rows[current_row_index][“CustomerNo”]) starts with “abo” and
        strtolower(rows[matched_row_index][“CustomerNo”]) starts with “ge”)
            color = GREY
    else if ( strtolower(rows[current_row_index][“CustomerNo”]) starts with “ge”)
            color = YELLOW
    end if
    html_table.insert(rows[current_row_index] in proper formatting and color)
 end for
 return html_table;
End Function
```

The Algorithm

In simple words the algorithm actually creates a hash table of keywords that exist in some company names. At each entry in the hash (where key is actually the keyword for O(1) access), we maintain the count of companies where the given keyword exists and the hash table of indices of those company rows in the original rows (that are provided in argument). More than 99% of keywords existed in very few company names. We ignored very few keywords (that exists in N number of company names where N is in thousands) with the assumption that if there is possible 50+ % or 80%

match then it will be tackled by other keywords in those names (that exist in them). This optimization actually results in comparison of very very few company names that may be potential matches. We save the indices of those rows that became actual matches and using those indices (that are match 80% or 50% with users intervention) we get the final table in html format that actually comprises the matched rows. We tackle the Customer type etc as required.

Important Points:

The important things to note are

- The alpha, beta and the option to prompt the user for comparison can be given to the program
- The prompt is made optional because sometimes we may want to see the result for 80%+ matches
- The alpha is actually the minimum length of the word that keyword that should take part in the comparison, we can make it 1 if we are interested to see more accuracy (program parameter).
- The beta is the maximum number of company names in which a given keyword is existing. I observed that in three cases the number of companies where a keyword existed were 500, 900 and 4500. I kept the maximum count to 500 (in the program, but can be given as parameter for more accuracy). There is another observation in this point that, if a keyword, that exists in 4500 company names, is ignored for which few of companies (where this keyword existed) were actually matching more than 80% then other keywords (whose count might be very less, like, 2, 3, 10 etc) will make it sure that these companies (rows) are marked. So ignoring a keyword that exists in 4500 company names didn't effect the quality but reduced the execution time considerably.
- I have shown those rows in the final html that actually have matches in the actual cv file. I have added two columns (first two) that actually show the row number and matching row number in the actual cvs file.

In the first try, I tried to write the algorithm in a brute force way but that was so much time consuming and was not flexible. This program improves the performance with reasonable quality of results. Although we can increase the execution time and accuracy by setting the parameters.