University of Milan
Department of Mathematics
"Federigo Enriques"

Project Report on Algorithms and Data Structures

Vittorio Mocchi

June 2023

Contents

Chapter 1

Introduction

1.1 Introduction to the Problem

The project revolves around the management of a pizzeria and the definition of its menu based on customer preferences. Through market research, the preferences of a group of potential customers regarding ingredients to add, prefer, or avoid in pizzas are collected. The objective is to extract useful information from these preferences, such as the number of customers requesting, liking, or excluding certain ingredients, in order to order ingredients by popularity. Customer urgency levels based on imposed constraints are also considered. Additionally, an attempt is made to determine a menu that satisfies all customers with the fewest possible pizzas, using lower and upper estimates obtained through heuristic algorithms. The distribution of ingredient popularity and the management of customer incompatibilities are important elements to consider in solving the problem.

The problem can be divided into five sections, which will be analyzed in detail in the following chapters:

- **Data Reading**: The input data is stored within data structures `IngredientOpinions` and `Customer`.
- **Popularity and Preferences Calculation, Exclusion Frequencies**: Ingredient popularity, preferences-based demands, and ingredient exclusion frequencies based on customer preferences are calculated.
- **Incompatible Pairs**: The number and pairs of incompatibilities are calculated and printed.
- **First Greedy Algorithm: Incompatible Groups**: The number of pizzas required to satisfy all customers is computed, estimating using a greedy algorithm with an incompatible customer group.

- **Second Greedy Algorithm: Menu Creation**: A menu satisfying all customers with the minimum number of pizzas is created using a greedy algorithm.

1.2 Notes

1.2.1 Code Organization

The code is organized into multiple files: In the file `Main.c`, various functions defined in files `Letturafile_Stampa.c`, `Compare_Sorting.c`, `Calcola_Stampa_CoppieIncompatibili.c`, `Calcola_Stampa_GruppiIncompatibili.c`, `Menu.c`, and `LiberaMemoria.c` are called. All functions in these files are called in the header. The code is compiled with gcc using the command:

```
gcc Main.c Menu.c LiberaMemoria.c Letturafile\_Stampa.c Compare\_Sorting.c

Calcola\_Stampa\_GruppiIncompatibili.c Calcola\_Stampa\_CoppieIncompatibili.c

-o out
```

It is then executed using `./out ingredienti00.txt esempio00.txt`.

1.2.2 Terminology

- In the analysis of the time complexity of our algorithms, we will refer to NC and NI to indicate the number of customers and number of ingredients, respectively.
- When using logarithms in the study of time complexity, it refers to the base 2 logarithm.

1.2.3 Sorting Algorithm

In the code, it's necessary to sort both ingredients and customers multiple times based on various criteria. For this reason, a generic implementation of `mergesort` has been implemented. It takes an array and a sorting criterion as input and returns the sorted array. I chose `mergesort` as the sorting algorithm because, while it requires more memory than heapsort, it is stable and thus more reliable for larger data quantities. Regarding the comparison with quicksort, the latter has a time complexity of $O(n^2)$ and is less efficient in certain situations.

1.2.4 Definition of Structures

- **Structure:** `struct Cliente`

  This contains: the unique identifier of the client; an array of characters representing the client's name, with a maximum size of N; the number of requests made by the client; a pointer to a pointer to a character representing a list of requests made by the client. Each request is a string of characters; the number of ingredients excluded by the client; a pointer to a pointer to a character representing a list of the excluded ingredients by the client. Each excluded ingredient is a string of characters; the number of ingredients liked by the client; a pointer to a pointer to a character representing a list of the ingredients liked by the client. Each liked ingredient is a string of characters; a pointer to the next Cliente structure, allowing the creation of a linked list of clients.

- **Structure:** `struct Ingredients_opinions`

  This contains: an array of characters representing the name of the ingredient, with a maximum size of N; the number of times the ingredient was requested; the number of times the ingredient was excluded; the number of times the ingredient was liked; a pointer to the next Ingredients_opinions structure, allowing the creation of a linked list of opinions on the ingredients.

- **Structure:** `struct Pizza`

  This contains: a pointer to a pointer to a character representing a list of the pizza's ingredients. Each ingredient is a string of characters; the number of ingredients present in the pizza; the number of satisfied customers with this pizza; a pointer to a pointer to a character representing a list of the names of satisfied customers with this pizza. Each name is a string of characters.

Chapter 2

Lettura dei dati

2.1 Sub-problem introduction

To complete the first part of the project, it's necessary to develop a program capable of reading information from two text files. The first file contains a list of available ingredients, sorted in alphabetical order. Each ingredient is represented by a single word with a maximum length of 50 characters. The file begins with the total number of ingredients and continues with a line for each ingredient.

The second file provides a list of interviewed clients, also sorted in alphabetical order. Each block of information for a client consists of four lines, containing the client's name, the requested ingredients, the liked ingredients, and the excluded ingredients.

2.2 Functions

In the main, I set the files to read and call the InterpretaLineaComando function, which reads from the terminal the files to open.

- **Function:** `initialize_opinions`

  The function performs a for loop that iterates NC times. Inside the loop, the following operations are performed:

  Read a string from the fscanf file. Set three integer variables to 0 nRichieste, nEsclusioni, and nGradimenti. Since the loop is run NI times and the operations within the loop have a constant cost, the time complexity of the function is O(NI). Since the function does not allocate dynamic memory or use other data structures, the space complexity of the function is O(1), meaning it does not use additional space relative to the input provided.

- **Function** `initialize_clients`
  The initialize_clients function has the job of initializing an array of Cliente structures, reading data from a file pointed by fp2, and updating an array of Ingredients_opinions structures. The function reads client information from the file, such as the name, the requested, liked, and excluded ingredients. In addition, the function updates the number of requests, likes, and exclusions for each ingredient in the opinions_array.

  Regarding the time cost, the function has three nested loops: the outer loop for clients, the middle loop for ingredients, and the inner loop for requests, likes, and exclusions. The outer loop is run NC times, the middle loop is run NI times, and the inner loop is run at most NI times (in the worst-case scenario where a client has requests, likes, or exclusions for all ingredients). Therefore, the overall time cost is O(NC * NI$\hat{2}$).

  The space cost of the function is mainly determined by the dynamic allocation of memory for the requested, liked, and excluded arrays. For each client, NI pointers are allocated for each array and subsequently, memory is allocated for N characters for each element of such arrays. Therefore, the overall space cost is O(NC * NI).

Chapter 3

Calculation of Popularity and Need

3.1 Introduction to the Subproblem

The problem is about creating a program that sorts and prints ingredients based on their popularity, following the criteria described in the project. Subsequently, we calculate the customers' "requirements", meaning the total number of ingredients requested or excluded by each customer, and the list of customers is printed in the specified order. Finally, it is required to print the sample frequency of ingredient exclusions, ignoring null values and maintaining a defined order.

3.2 Scheme

For the first part of the problem, we call the function `mergesort_opinions`.

- **Function:** `print_opinions`
  This function, firstly, sorts the ingredients by popularity using a mergesort algorithm. Then, it prints the name of the ingredient, the number of people who have excluded it, who request it, and who like it.

  The time cost of this function is the sum of the calculation time of the mergesort and the for loop to print. Given that the for loop is executed for each ingredient, thus $O(NI)$ and the calculation time of mergesort_opinions is $O(NI * log(O(NI))$; Therefore, the time cost is $O(NI * log(NI))$. The function, except for the mergesort, does not allocate dynamic memory and only uses local variables as counters. Therefore, the space cost is that of the mergesort, $O(NI)$.

- **Function:**`print_clients`

This function, firstly, sorts the customers by their requirements using a `mergesort` algorithm. Then, it prints the number of customers, the customer's name, the number of strong constraints (exclusions and requests) they have expressed, and the number of ingredients they like.

Here we make the same calculation as we did for the function `print_opinions`. Since the for loop is executed for each customer, thus $O(NC)$ and the calculation time of `mergesort_clients` is $O(NC * log(O(NC))$; Therefore, the time cost is $O(NC * log(O(NC))$. Also in this case, the function does not allocate dynamic memory except for the `mergesort` as it only uses local variables as counters. Therefore, the space cost is that of the `mergesort`, $O(NC)$.

- **Function:** `exclusions_frequencies`
  This function calculates the frequency of exclusions for each ingredient in the `opinions_array` and stores them in the 'frequenze' array. The function first initializes the 'frequenze' array to zero and then scans the `opinions_array` to update the frequency of exclusions for each ingredient. The time cost of this function is $O(NC + NI)$, as I first initialize the 'frequenze' array to 0 for NC times, then go through the `opinions_array` NI times. The space cost is $O(1)$ as no additional memory is allocated, and the 'frequenze' array is modified in-place.

- **Function:** `print_exclusions_frequencies`
  This function prints the frequencies of exclusions for each ingredient, contained in the `exclusions_frequencies` array. The function scans the `exclusions_frequencies` array and prints the number of exclusions for each customer, only if the frequency value is greater than zero. The time cost of this function is $O(NC)$, as the `exclusions_frequencies` array is scanned once. The space cost is $O(1)$ as no additional memory is allocated, and the operations are performed in-place.

Chapter 4

Incompatible Pairs

4.1 Introduction to the sub-problem

In this section of the project, we will focus on finding incompatible pairs, meaning pairs of customers such that one customer requests an ingredient that the other excludes. To do this, we will create a graph described by its respective adjacency matrix.

4.2 Scheme

To solve the request, we will use two functions that are found within the file `Calculate_Print_IncompatiblePairs`.

- **Function:**`calculate_Adjacency_Matrix` This function calculates the adjacency matrix that represents the compatibility between customers. An element in the matrix is set to 1 if the two corresponding customers are incompatible, otherwise it is set to 0. The function performs the following operations:

  1. Sorts the customers array using `mergesort`.
  2. Allocates the adjacency matrix `adj` and initializes all elements to 0.
  3. Finds the pairs of incompatible customers and sets the corresponding elements of the `adj` matrix to 1.

  To clarify, we have two nested loops to go through the pairs of customers $O(NC^2)$ and another two nested loops to compare the requests and exclusions of each pair $O(n_r \times n_e)$ in the worst case, where n_r is the maximum number of requests per customer and n_e is the maximum number of exclusions per customer. Multiplying these two terms, we get a time cost of $O(NC^2 \times NI^2)$. The space cost is $O(NC^2)$, as a matrix of dimension $NC \times NC$ is allocated.

- **Function:** `print_Adjacency_Matrix`
  This function prints the incompatible pairs of customers and the total number of incompatible pairs, using the adjacency matrix calculated by the `calculateAdjacencyMatrix` function. The function traverses the adjacency matrix and, if it finds an element set to 1, increments the count of incompatible pairs and adds the names of the incompatible customers to the auxiliary string.

  The time cost of this function is $O(NC^2)$, as the adjacency matrix is traversed once with nested loops.

  The space cost is $O(NC^2)$. This is due to the allocation of the auxiliary string used to store the names of the incompatible customers.

Chapter 5

First Greedy Algorithm: Incompatible Group

5.1 Introduction to the Sub-problem

This part of the project deals with obtaining an approximate but valid estimate of the number of pizzas needed to satisfy a group of incompatible customers. It starts by selecting the incompatible customer with the maximum number of other incompatible customers. Then, a greedy algorithm is used to gradually add the customer who is incompatible with all those already in the group and involves the maximum total number of customers. The objective of the criterion is to maintain the maximum number of potential incompatible pairs within the group.

5.2 Outline

- **Function:** `findIncompatibleGroup`

  The function `findIncompatibleGroup` finds a group of incompatible customers based on the information on the adjacency matrix indicating the incompatibilities between customers. The resulting group represents the minimum number of pizzas that must be ordered.

  1. The function starts by initializing some variables and allocating memory for the incompatible and group arrays.
  2. Calculates the number of incompatibilities for each customer using a double for loop that traverses the adjacency matrix and increments the counter `incompatible[i]` if the corresponding value in the matrix is 1.
  3. Finds the customer with the most incompatibilities and adds them to the group.

4. Enters an infinite while loop where it continues to search for and add incompatible customers to the group. The loop breaks when no customer incompatible with all group members can be found.
5. In the end, the size of the group is assigned to the pointer size, the incompatible array is freed, and the group array is returned.

The time cost of this function is $O(NC^3)$, as a double for loop iteration is used to calculate the number of incompatibilities for each customer within a while loop that runs a maximum of NC times. The space cost is $O(NC)$, due to the allocation of the incompatible and group arrays.

- **Function:** `print_incompatible_names`
The function `print_incompatible_names` prints the minimum number of pizzas required and the names of the incompatible customers in the group found by the function `findIncompatibleGroup`. Here's how it works:

  1. Calls the function `findIncompatibleGroup` to get the group of incompatible customers and the size of the group.
  2. Allocates memory for the array incompatible_names and assigns the names of the incompatible customers using the obtained group.
  3. Sorts the array incompatible_names using `mergesort` and a comparison function `compareCustomerNames`.
  4. Prints the minimum number of pizzas required and the names of the incompatible customers in sorted order.
  5. Frees the memory allocated for the group and incompatible_names arrays.

  The time cost of this function is directly related to the function `findIncompatibleGroup` as it is the relevant factor, and thus $O(NC^3)$. The same applies to the space cost, which is therefore $O(NC)$.

  What we are doing with this algorithm is similar to the maximal clique problem in a graph. It refers to any of the problems related to finding particular complete subgraphs ("cliques") in a graph, i.e., sets of elements where each pair of elements is connected. The maximal clique problem has a non-polynomial cost; obviously, we obtain a polynomial cost as what we are looking for is a local maximum of which we are not sure is the absolute maximum.

Chapter 6

Second Greedy Algorithm: Menu Creation

6.1 Introduction to the Sub-problem

In this problem, we are given a list of customers and their preferences regarding pizza ingredients. The goal is to create a pizza menu that satisfies the preferences of as many customers as possible, using a greedy algorithm. The function `buildMenu` has been implemented to solve this problem, taking as input the list of customers, the number of customers, the number of ingredients, and the customers' opinions on the ingredients.

The function `buildMenu` follows an iterative approach to create the menu. In each iteration, it creates an empty pizza and tries to add ingredients that satisfy the most number of unsatisfied customers. Once the current pizza can no longer be expanded, it is added to the menu, and the satisfied customers are removed from the list of unsatisfied customers. This process is repeated until there are unsatisfied customers or until a preset iteration limit is reached.

During the process, the function `printPizza` is used to print the ingredients and satisfied customers for each pizza on the menu. The ingredients and customers are sorted alphabetically before being printed.

6.2 Outline

- **Function:** `build_Menu`
  The function `buildMenu` takes as input a pointer to the list of customers (customer), the number of customers (NC), the number of ingredients (NI), and an array of opinions on the ingredients `opinion_array`. The goal is to create a pizza menu that satisfies as many customers as possible.

  To do this, the function performs the following steps:

1. Sorts the array of opinions on ingredients alphabetically using the function `mergesort`.
2. Creates a list of unsatisfied customers from the input array of customers and initializes the counter of unsatisfied customers.
3. Enters a while loop that continues until there are unsatisfied customers or a limit of 200 iterations is reached. In each iteration of the loop:
   (a) Creates a copy of the list of unsatisfied customers.
   (b) Initializes another while loop that continues until a limit of 200 iterations is reached or the current pizza is "finished" (can no longer be expanded). In each iteration of the second loop:
      i. Finds the ingredient with the fewest exclusions among the temporary customers.
      ii. If the found ingredient is excluded by all temporary customers, the pizza is considered "finished".
      iii. Otherwise, removes customers who exclude the found ingredient, adds the ingredient to the current pizza, and removes the ingredient from the list of ingredients.
   (c) Removes satisfied customers from the list of unsatisfied customers.
   (d) Adds the current pizza to the menu and increments the counter of pizzas on the menu.
   (e) Frees the memory occupied by the list of ingredients and the list of temporary customers.
4. Prints the number of pizzas on the menu and the information of each pizza using the function `printPizza`.
5. Destroys the menu and the list of unsatisfied customers, freeing the allocated memory.

Regarding the time complexity of the function, we observe that the major contribution comes from the function `removeSatisfiedCustomers` which is inside two while loops that run for a maximum of NC and NI times, thus $O(NC^2NI^2(NC+NI))$. For space complexity, we find the function `copyCustomerList` inside the while loop, thus obtaining $O(NI * NC^2)$.

The sub-functions used in the function `buildMenu` are analyzed below.

- **Function:** `countExclusions`
  This function counts the number of customers who exclude a particular ingredient. The time complexity of this function is $O(NC * NI)$ as two concatenated for loops are traversed with a maximum number of iterations of NC and NI. The space complexity is constant, $O(1)$.

- **Function:** `copyCustomerList`
  This function creates a copy of a list of customers. The time complexity is $O(NI * NC)$ as I scroll through the function `copyCustomer` a maximum of NC times, which has a cost of $O(NI)$ as it only contains the function `copyStringArray`. For the space complexity, for the same reason, I will have $O(NI * NC)$.

- **Function:** `findLeastExcludedIngredient`
  This function finds the ingredient with the fewest exclusions among the customers in the list. The time complexity of this function is $O(NI^2 * NC)$ as there is the function `countExclusions` inside a for loop. It has a time complexity of $O(NC * NI)$ as two concatenated for loops are traversed with a maximum number of iterations of NC and NI. The space complexity is constant.

- **Function:** `removeCustomersExcludingIngredient`
  This function removes customers who exclude a specified ingredient from the list of customers. The time complexity of this function is $O(NI * NC)$ as it is necessary to traverse the entire list of customers and check if each customer excludes the specified ingredient. The space complexity is constant $O(1)$.

- **Function:** `ingredientExcludedByAll`
  This function checks if an ingredient is excluded by all customers in the list. The time complexity is $O(NI * NC)$, as it is necessary to traverse the entire list of customers and check if each customer excludes the specified ingredient. The space complexity is constant, $O(1)$.

- **Function:** `removeSatisfiedCustomers`
  The function `removeSatisfiedCustomers` removes satisfied customers from the list, checking that they have all the requests met and do not have exclusions present in the pizza. The time complexity is $O(NC * NI^2)$, as we have inside the while loop that runs for NC times in the worst case two for loops that run a maximum of NI times.

- **Function:** `removeIngredientFromIngredientList`
  The function `removeIngredientFromIngredientList` removes ingredients from the list of ingredients. The time complexity of this function is $O(NI)$. The space complexity is $O(1)$, as only a few variables like prev and current are used.

- **Function:** `createIngredientListFromArray`
  The function `createIngredientListFromArray` creates a dynamic list of ingredients from an array. The time complexity of this function is $O(NI)$. The space complexity is $O(NI)$, as n nodes are created for the dynamic list.

- **Function:** `createCustomerListFromArray`
  The function `createCustomerListFromArray` creates a dynamic list of

customers from an array. The time complexity of this function is $O(NC)$. The space complexity is $O(NC)$, as NC nodes are created for the dynamic list.

- **Function:** `printPizza`
  The function `printPizza` takes a pointer to a Pizza structure and prints the ingredients and satisfied customers of the pizza. The time complexity of this function is $O(NI * log(NI) + NC * log(NC))$, as it prints the ingredients and customers once each. The space complexity is $O(NI)$, as it creates a copy of the ingredients and satisfied customers before printing them.

Chapter 7

Conclusion

To conclude, the time and space complexity of the code will be given by the function `buildMenu`, as it is the most expensive from both points of view. Obviously, all the smaller complexities will be irrelevant, and thus we obtain a total time complexity of $O(NC^2NI^2(NI + NC))$. As for the total space complexity, it is $O(NI * NC^2)$.