Artificial Intelligence Report

Optimal Sample Selection System

Group 7

Luo Yemao 2009853G-I011-0041

Luo Wei 2009853G-I011-0070

Lin Longjun 2009853F-I011-0014

Contents

1 Overview of Problem

Figure 1 is an overview of the problem. We have m samples and extract n samples randomly from these samples. We group these samples, and each group contains j samples. We are looking for a sampling method that can take several **Combination_K**s of samples of length k, and the sample set covers all sample groups. The requirement of the question for covering sample grouping is that, randomly select several **Combination_S**s of length s in the sample grouping, and **Combination_K** only needs to completely include all the samples in one **Combination_S** in the several **Combination_S**s. In this case, we can say that The **Combination_K** covered this sample group. The key point of the question is how to obtain a **Combination_K** that can completely cover all sample groups in a short period of time, and the number of **Combination_K**s in the combination is as small as possible.

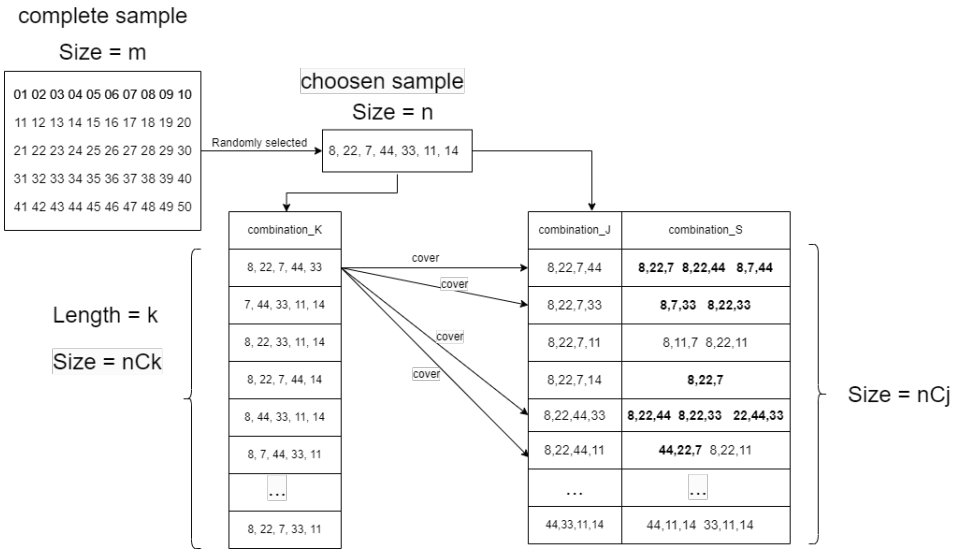

Figure 1: Overview of the Problem

2 Solution to the Problem

Figure 2 is a overview of our idea.

Our algorithm proceeds as follows:

1. Generate Chosen Samples. Randomly take n samples from the full set of samples.

2. Generate Possible Results. We use backtracking to pick out all combinations of length j from n. Each **Combination_J** is stored in the list, and all **Combination_J** are placed in a list of length nCj.

3. Generate Cover List. We use backtracking to pick out all combinations of length k from n. Each **Combination_K** is stored in the list, and all **Combination_K** are placed in a list of length nCk.

4. Get Results. We use greedy search to search Possible Result List and calculate the number of covering **Combination_J** in the cover list for each **Combination_K**. We select the **Combination_K** that can cover the most **Combination_J** and remove this **Combination_K** to the Result List from Possible Result List, then delete the covered **Combination_J** in the Cover List, and repeat the above steps until the Cover List is empty.

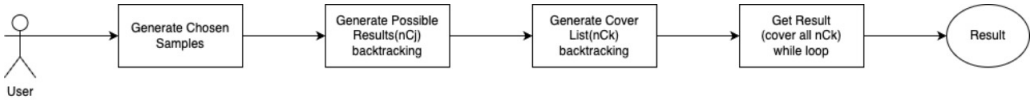

Figure 2: Overview of the Process

3 The inadequacy of general solution

1. **Getting Trapped in Local Optima:** During the process of obtaining the result, we use a greedy strategy to choose the combination that covers the most sets, which may lead to getting trapped in local optima. The greedy strategy chooses the combination that can cover the most sets in each iteration, rather than considering the global optimal solution. In some cases, the local optimal solution may not be the global optimal solution. For example, when there are multiple combinations covering the most sets, the final result obtained from the subsequent series of operations based on the combinations we prioritize in the search will be different, possibly leading to a non-optimal solution.

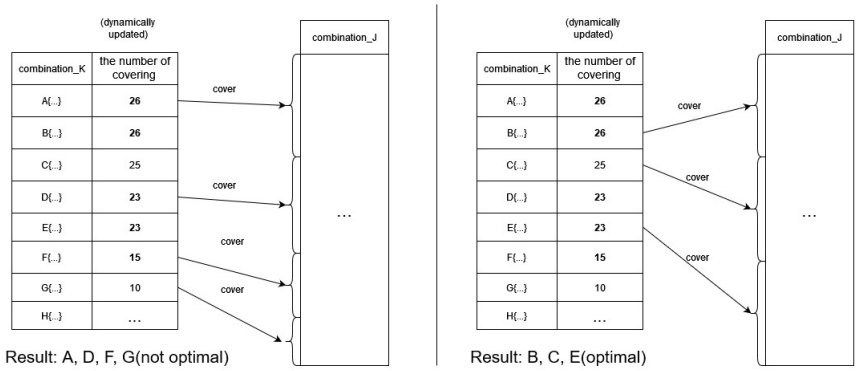

Figure 3: Example of Local Optima

2. **Huge Time and space:** This approach has a high time and space complexity. When getting temp optimal in each iteration, it is necessary

to calculate all k-element and j-element subsets in the n elements, i.e., O(nCk * nCj). As n, k, and j increase, the time and space requirements will grow rapidly. In addition, during the process of obtaining the result, if an exhaustive approach is used to traverse all possible result combinations, it will further increase the computational requirements.

4 Contribution in Optimizing Algorithms

1. **Save Time Using Filters**

   We use the filter method to skip elements that absolutely do not meet specific conditions. In our code, filters are applied to Possible Results List and Cover List, to reduce the number of elements traversed.

   - Possible Results List. In the process of selecting the **Combination_K** that can cover the most **Combination_J**, we apply the filter to skip over **Combination_K** that are definitely not the answer in this round of traversing. For example, the **Combination_K** selected in this round is [1, 2, 3, 4, 5, 6, 7], and in the next round of searching Possible Result List, we will directly skip the **Combination_K** which has the same element as the last result. In this example, it means that we would skip the **Combination_K** which contains 1.
   - Cover List. When traversing Cover List, we also use the filter to skip over **Combination_J** that are definitely not the answer. For example, when we calculate the number of **Combination_J** that can be covered by [1, 2, 3, 4, 5] now, we will ignore **Combination_J** whose first element is not in the [1, 2, 3, 4, 5] combination, such as [6, 7, 8].

   This helps to reduce unnecessary traversals and improve code efficiency.To ensure accuracy, we conducted accuracy testing and run-time testing on programs with and without filters. The test cases are in Table 1, and the test results of time cost are showed in Figure 4 and the differences compared with given answer showed in Figure 5. We can clearly see that Filter can greatly reduce program run-time without reducing the accuracy of the results.

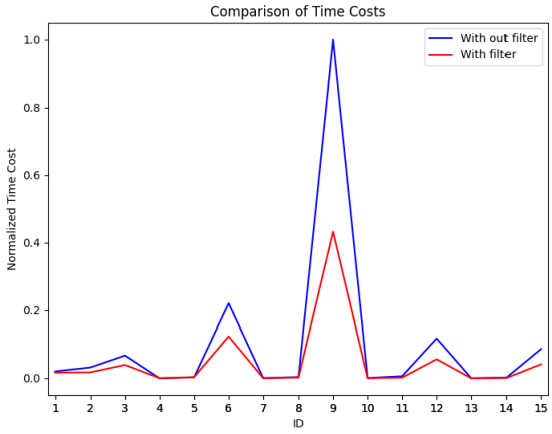

Figure 4: Time Cost in Test cases

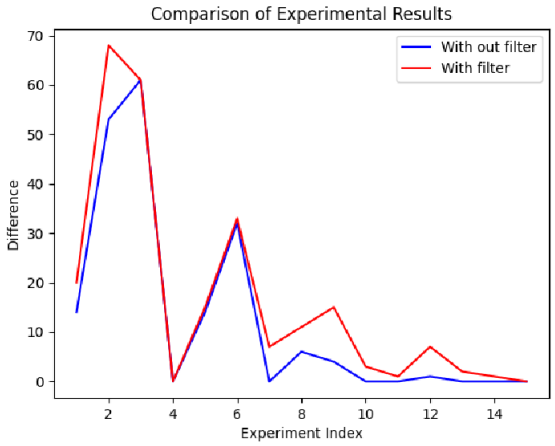

Figure 5: Differences Compared with Given Answer

2. **Reducing Time and Memory Usage with Fork and Join**

   We use the Fork/Join framework to decompose tasks into smaller tasks for parallel execution. In our code, we utilize parallelStream to achieve parallel processing. parallelStreamis a feature introduced in Java 8, which takes advantage of multi-core CPUs by automatically decomposing tasks into smaller tasks for parallel execution. Although the use of multi-threading does not directly reduce the time complexity of the problem, it significantly reduces the actual execution time by parallelizing the computation. Figure 6 is an overview of our idea. We partition the Possible Result List and calculate the number of coverage **Combination_J** for each **Combination_K**

in each part of the Possible Result List, selecting the **Combination_K** that can cover the most **Combination_J** as the sub_result for each part. And then in these sub_results, we select **Combination_K** from these sub_results that can cover the maximum number of **Combination_J** as the result for this round of traversal. Similarly, when traversing Cover List, we partition the Cover List and traverse each part in parallel. This can significantly reduce program execution time in multi-core CPU environments. To ensure accuracy, we conducted accuracy testing and performance testing on programs using Fork and Join. Figure 7 shows the improvement in performance after using Join and Fork. Figure 8 shows the accuracy after using Join and Fork. The results show that after using Fork and Join, the running time is greatly reduced, and only about 10% of the error is brought .

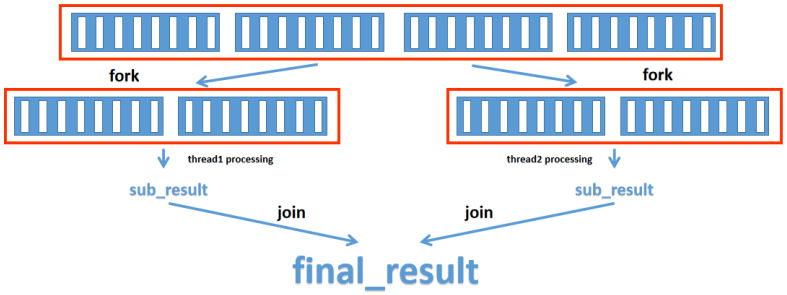

Figure 6: Overview of Using Join and Fork

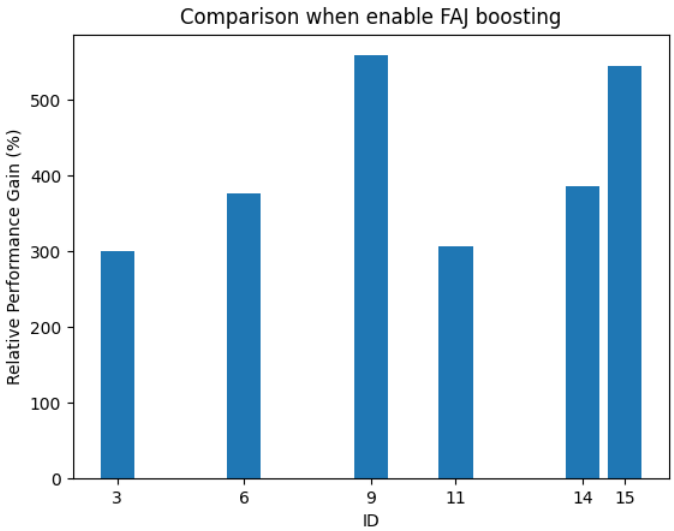

Figure 7: Relative Performance after Using Join and Fork

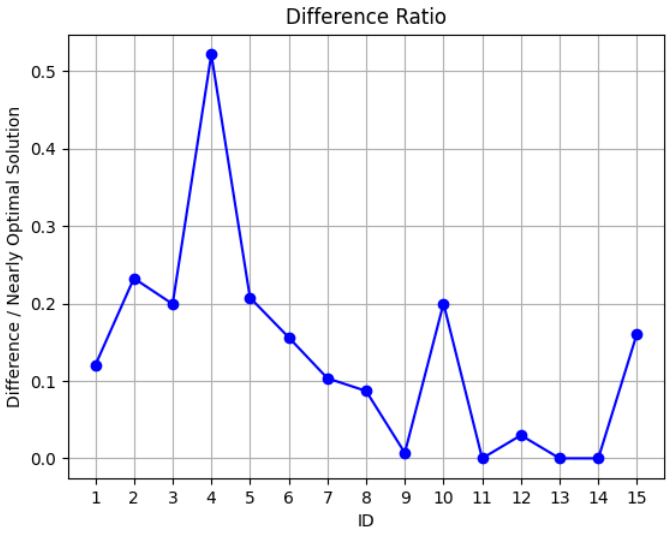

Figure 8: Differences Ratio after Using Join and Fork

3. **Reduce Space Usage**

   According to the problem constraints, our goal is to find at least s samples from the j-element subsets within a group of k samples. Therefore, when determining if a group is covered, we do not need to exhaustively enumerate all possible values of s. As long as we find a group that contains at least s samples from the **Combination_J**, we can consider the group to be covered without further consideration of other values of s.

   This optimization reduces the amount of data we need to store and process. By avoiding the enumeration of all possible values of s, we save computational resources and memory. This makes our solution more efficient, especially when dealing with large amounts of data. For instance, in the getCandidateResult and removeCoveredResults methods, we only check if the condition of at least s samples is met, rather than checking every possible value of s. In this way, we can focus on achieving the core goal of the problem while reducing computational and storage overhead.

5 Experiment Results

We run our program using the given test cases in moodle. The test cases are listed below.

ID	m	n	k	j	s	Nearly optimal solution
1	45	10	6	5	5	50
2	45	12	6	5	5	132
3	45	13	6	5	5	245
4	45	10	6	4	4	37
5	45	12	6	4	4	42
6	45	16	6	4	4	162
7	45	11	6	6	5	26
8	45	12	6	6	5	42
9	45	16	6	6	5	280
10	45	12	6	5	4	18
11	45	13	6	5	4	28
12	45	16	6	5	4	65
13	45	12	6	6	4	6
14	45	13	6	6	4	12
15	45	16	6	6	4	38

Table 1: Given Test Cases in Moodle

We will test our OSSS program in two aspects, the first one is the run time which is the performance, and the second one is the run result which is the accuracy. Also, all the tests will be run both on computers and mobiles, using three different CPUs, which are Apple M1 Pro, Intel i7-12700H, and Snapdragon 8 Gen2.

5.1 Computers

ID	Time cost(s)	Result size	Nearly optimal solution	Difference
1	0.061	55	50	5
2	2.075	171	132	39
3	1.690	305	245	60
4	0.018	23	37	12
5	0.354	55	42	13
6	3.794	192	162	30
7	0.062	28	26	2
8	0.365	46	42	4
9	14.345	282	280	2
10	0.118	15	18	3
11	0.104	28	28	**0**
12	2.107	68	65	3
13	0.073	6	6	**0**
14	0.062	12	12	**0**
15	1.723	32	36	6

Table 2: Test Results Using OSSS for Computers With **Apple M1 Pro**

ID	Time cost(s)	Result size	Nearly optimal solution	Difference
1	0.045	55	50	5
2	1.749	171	132	39
3	1.141	305	245	60
4	0.012	23	37	12
5	0.278	55	42	13
6	2.7	192	162	30
7	0.076	28	26	2
8	0.404	46	42	4
9	11.8	282	280	2
10	0.160	15	18	3
11	0.061	28	28	**0**
12	1.852	68	65	3
13	0.061	6	6	**0**
14	0.41	12	12	**0**
15	1.24	32	36	6

Table 3: Test Results Using OSSS for Computers With **Intel i7-12700H**

5.2 Mobiles

ID	Time cost(s)	Result size	Nearly optimal solution	Difference
1	0.319	56	50	6
2	13.41	171	132	39
3	13.021	305	245	60
4	0.124	23	37	12
5	2.377	55	42	13
6	57.9	192	162	30
7	0.533	28	26	2
8	3.492	46	42	4
9	245.73	282	280	2
10	1.324	15	18	3
11	1.710	28	28	**0**
12	29.34	68	65	3
13	0.704	6	6	**0**
14	1.049	12	12	**0**
15	24.3	32	36	6

Table 4: Test Results Using OSSS for Mobiles With **Snapdragon 8 Gen2**

With the experimental results, we believe that our implementation of OSSS is a good approach with outstanding performance in both efficiency and accuracy.

6 Software `OSSS`

6.1 User interface

Our UI file is `Page.java`. We use Java swing to complete the layout and design of the page. The user interface has two windows, divided into a home page in Figure 9 and a history page in Figure 10. The home page has 6 input boxes for inputting m, n, k, j, s and chosen samples, an output box for outputting results and the overall time consumption, and a progress bar to monitor the percentage of program completion. The history interface shows all the running records in the database. Users can see the previous records on this page.

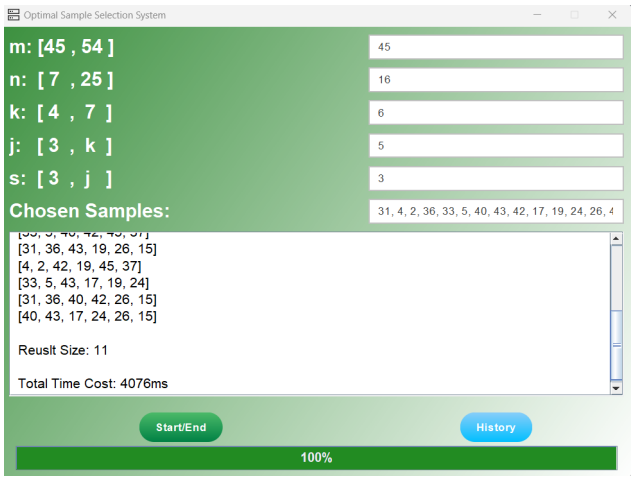

Figure 9: Home Page

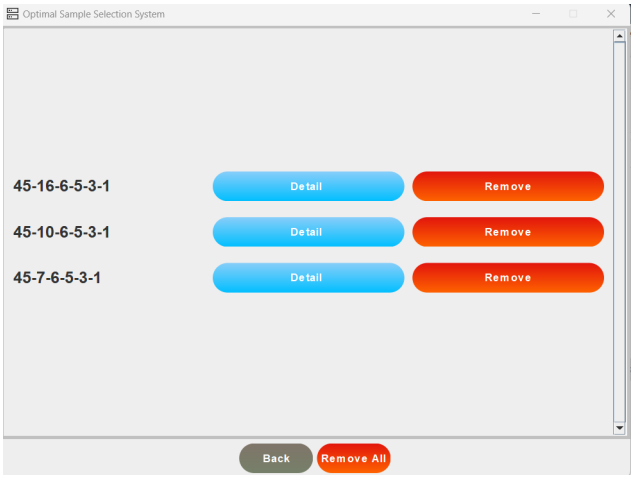

Figure 10: History Page

6.2 Database

6.2.1 Computers

For computers, we wrote a Java file that contains a class named `DBHelperUI`, which allows reading and writing files to a directory named `./db/`. Our implementation method fully combines object-oriented thinking, and DBHelperUI can be called at any time in the UI to operate the database, and finally, we achieve a function that can save the results, delete the results, and view the results. The implementation of our db is based on the thinking of serialize. The result for each run will be stored and serialized before being saved to disk. Also, a db cache file will be automatically refreshed and used to help to accelerate fetching records from disk. The following are the explanations, input, and output of these methods.

- `save()`: Used to write data into a file. Returns `true` on successful write, `false` otherwise.
- `load()`: Used to read data from a specified file and return the data.
- `remove()`: Used to delete a specified file. Returns `true` on successful deletion, `false` otherwise.
- `loadDB()`: Used to load the database and cache. Returns `true` on successful load, `false` otherwise.
- `readAllDBFiles()`: Used to read all files in a specified directory and return an unordered list of file names.
- `removeAll()`: Used to delete all files and cache files in a specified directory.

Method Name	Input Parameters	Return Value Type
`save()`	File name (`String`) Data to write (`String`)	Success status (`boolean`)
`load()`	File name (`String`)	Data read (`String`)
`remove()`	File name (`String`)	Success status (`boolean`)
`loadDB()`	None	Success status (`boolean`)
`readAllDBFiles()`	None	List of file names (`List<String>`)
`removeAll()`	None	Success status (`boolean`)

Table 5: Input Parameters and Return Value Types for Each Method

6.2.2 Mobiles

For mobiles, we used the embedded database SQLite in the Android platform. To make our usage of the database easier, we also wrote a Java file to provide some functions for database manipulation. Also, we would sort all the results by the auto-increasing id given automatically. This means the user can easily find the latest run result. The following are the explanations, input, and output of these methods.

- `save()`: Used to write data into a file. Returns `true` on successful write, `false` otherwise.
- `load()`: Used to read data from a specified file and return the data.
- `remove()`: Used to delete a specified file. Returns `true` on successful deletion, `false` otherwise.
- `loadAll()`: Used to read all files in a specified directory and return an unordered list of file names.
- `removeAll()`: Used to delete all files and cache files in a specified directory.
- `nameBuilder()`: Used to generate file name.

Method Name	Input Parameters	Return Value Type
`save()`	File name (`String`) Data to write (`String`)	Success status (`boolean`)
`load()`	File name (`String`)	Data read (`String`)
`remove()`	File name (`String`)	Success status (`boolean`)
`loadDB()`	None	Success status (`boolean`)
`loadAll()`	None	List of file names (`List<String>`)
`removeAll()`	None	Success status (`boolean`)
`nameBuilder()`	Parameters m, n, k, j, s (`int`)	File name (`String`)

Table 6: Input Parameters and Return Value Types For Each Method

It is clear that the `DBHelper` class methods on our different platforms are basically the same, and the input and output parameters are almost identical. This is to enhance the compatibility of the code, and we no longer need to modify other classes to use the DBHelper class. However, the implementation methods of the database on mobile devices and computers are completely different. On mobile devices, we have adapted to SQLite, which means that our data will be more secure under the protection of the DBMS, and our interaction with the data is no longer simply reading from the disk, but obtaining data from the DBMS through SQL.

7 Code Repository

The code will be open source on these GitHub links:

- Software on PC. https://github.com/YemaoLuo/AIAlgorithm
- Application on Mobile Phone. https://github.com/YemaoLuo/AIApplication

8 Acknowledgments

We would like to express our heartfelt gratitude to our esteemed teacher,**Zeng Xiangcai**, for his invaluable guidance and unwavering support throughout the course of our experiments and report writing.