Chapter 1

Introduction

1.1 Background

The amount of email sent daily is 31 billion and the original size is about 440,606 terabytes in 2002. Most of the Internet user has at least one mail box and email become the second most popular way of communication after telephone. As the amount of information in emails grows, the amount of redundant emails also relatively increases fast. According to the test data for this system, the average rate of redundant messages in a normal mail box is 36%. This problem leads to the need of redundancy detecting methods for email systems.

There are some existing approaches for message redundancy problem. However, most these approaches do not fully use the contents of the email messages to detect redundancies. The system described in this report implements and tests some string matching approaches for this problem.

1.2 Definitions

1.2.1 Email message

Email messages experimented in this project are plain-text email messages written in English. An email can also contain attachments. According to this definition, the system cannot process correctly email messages that have other format than plain-text or not written in English. This exception also includes messages that contain special characters or symbols.

1.2.2 Redundant email message

Redundant messages could be simply defined as messages whose contents are repeated in other messages. However, a more detailed definition will be helpful in describing and finding solutions to the redundancy problem.

In this project, we assume that a message in a mail folder can only be repeated in later messages of that mail folder. This assumption considers messages arrive to the mail folder in time-order. Therefore, a message cannot be quoted in previous messages. If the messages in the mail folder are copied from other folder, we can sort them base on sending time before processing to keep the assumption true.

A message in a mail box is considered as redundant if the main content of the message is repeated in later message in that mail box. The main content of a message is the text body of the message excludes the salutation, the conclusion and the signature.

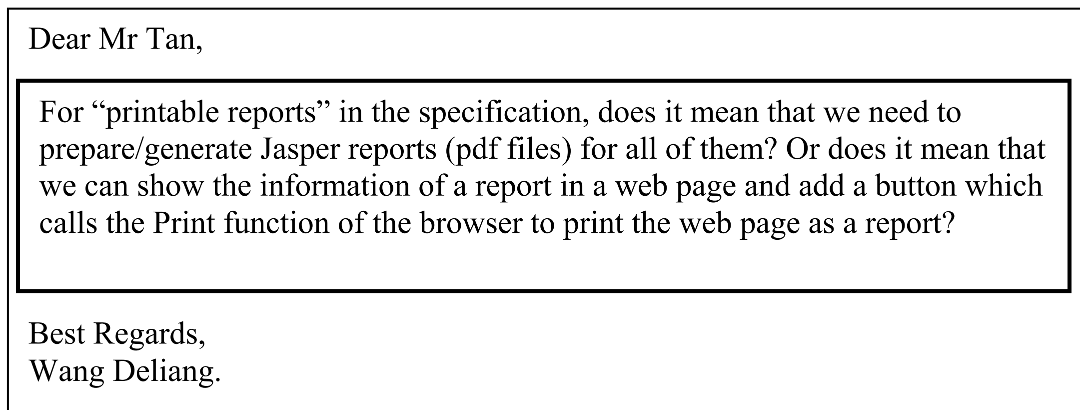
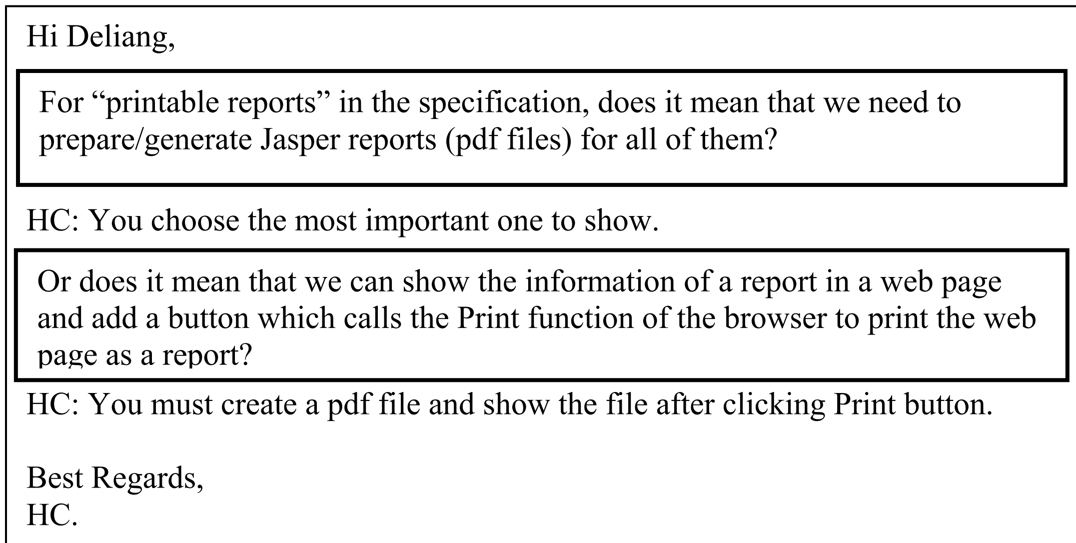


Figure 1.1: Illustration of an email message

Figure 1.1 represents the structure of a typical email message. The main content of the message is enclosed in the bold border box. Main content can be further divided into sub parts and quoted in later messages.


Hi Deliang,

For “printable reports” in the specification, does it mean that we need to prepare/generate Jasper reports (pdf files) for all of them?

HC: You choose the most important one to show.

Or does it mean that we can show the information of a report in a web page and add a button which calls the Print function of the browser to print the web page as a report?

HC: You must create a pdf file and show the file after clicking Print button.

Best Regards,
HC.

Figure 1.2: An email message quotes another message

Figure 1.2 represents another message that quotes the main content of the message in Figure 1.1. This main content is divided into two smaller blocks and quoted separately in the second message. In this scenario, the first message is considered as repeated in the second message. We will call the first message “redundant message” and second message “container message”.

The definition can be further extended so that the main content can be divided and quoted in many messages. We can also allow some small unimportant portions of the main content to be deleted in the later messages. The amount of acceptable deletions and the constraints will be discussed in Chapter 3.

1.3 Motivation

According to the definitions stated in previous part, there are some challenges in finding a solution for the problem.

The first challenge is identifying the main content of an email message. According to the test data of this project, the rate of messages quoted without salutations, conclusions and signatures is 30% of the redundant messages. Therefore, it is necessary to identify the main content of a message for redundancy checking.

Main content of a message can be divided into smaller portions and quoted in other messages poses the next challenge. We should define what should be an acceptable method for dividing a message. For example, a message cannot be divided into words since a single word tends to appear in many non-relevant messages. We also want to have some constraints to ensure that redundant message is not scattered in too many other container messages. In such cases, recovering the initial redundant messages from those portions is troublesome.

The last challenges is detecting if a small unimportant portion of the main content is deleted in container messages. For example, in the message of Figure 1.1, we can delete some words of the last sentence but still keep the meaning of the sentence.

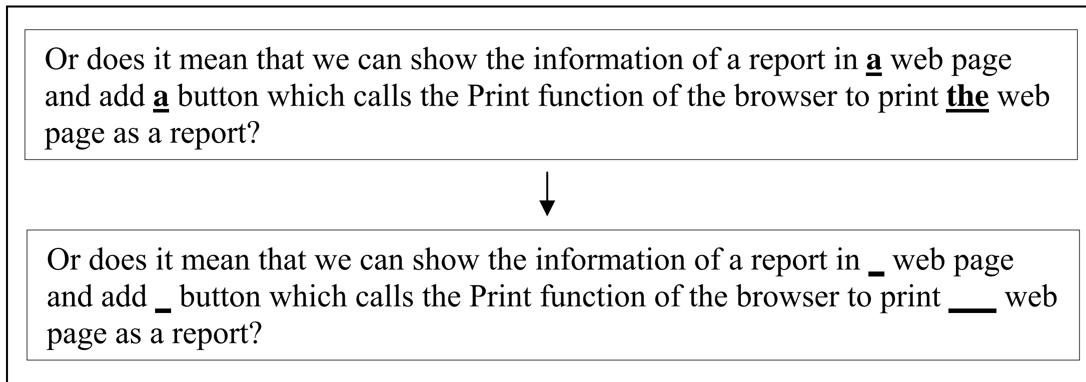


Figure 1.3: Acceptable deletions in message

Figure 1.3 is an example of possible deletion of unimportant portions. If the modified message is quoted in another message, we consider it as redundant.

1.4 Organization and Structure

This report is organized into 5 Chapters:

Chapter 1: presents the background, problem statement and motivation of the project.

Chapter 2: examines some prior work.

Chapter 3: explains benchmarking, testing and final methods for redundancy problem.

Chapter 4: presents the experiments and result of all methods in Chapter 3.

Chapter 5: summarize the previous chapters, presents the limitation of final method and suggestions for future work.

Chapter 2

Prior art

There are some existing solutions to message redundancy problem. In this chapter, we discuss main points of these solutions before representing the main approach in the next chapter.

2.1 Real-time data feeds system

One solution to the redundancy in email message is stated in US patent 5404488 (Michael Kerrigan and Michael A. Dempsey, 1995). The system presented in this patent classifies messages into known groups. The system caches the most recent message of each group. These cached messages can be used to verify redundancy.

This solution may help in detecting redundant messages but also eliminate information in previous messages since only the most recent one is cached for each group. For example, the system can be used for financial data feeds. The groups can be the stock symbols and the most recent message of each symbol is kept. Using this approach, the information about previous messages like history of share price will be lost.

2.2 Email message thread

Redundant messages can be detected through grouping messages under threads. This is the approach that has been used in bulletin board systems and USENET newsgroups. US patent 5905863 (Kimberly A. Knowles and David Dolan Lewis, 1999) proposed the similar approach for email system. This approach focuses on constructing message threads using “reply-to” field of an email message header.

According to this method, if message A is a reply-message to message B then we conclude that message B is repeated in A.

However, using this approach to identify redundancies in email messages has some limitations. A reply-message may be sent without using the “reply” command of the emails system. In this case, user can simply copy the content of the first message in the second one and send it as a new email message. Using “reply-to” field in the header cannot detect such redundancy cases. A user can also use the “reply” command to send a non-relevant message to another user. This case happens usually as users may want to avoid re-typing the receiver’s email address. The system will mark the first message as repeated in the non-relevant one which is not true.

These limitations occur since the system does not consider the content of the messages. Using the “reply-to” header field only may yield a better performance in term of time. However, the limitations of this approach lower the accuracy in detecting redundant messages.

2.3 Content comparison approach

The final approach, which is also the main approach of the methods discuss in Chapter 3, is string matching algorithms. By comparing main contents of messages, we found that the system could detect up to 85% of the redundancies. The overall methods have been presented in European patent No. 1327192 (Chong-See Kwok and Limsoon Wong, 2005).

According to this patent, the process of detecting redundant messages can be divided into two steps. The first step is cleansing the messages in which the following information will be removed:

- Header information (such as “to:”, “from:”, “subject:” , “date:”)
- White spaces (such as tab, carriage return, new line, space, etc.)
- Punctuation symbols (such as comma, semi-colon, colon and period)
- Email forwarding and quoting symbols (such as “>” and “----“)

For the cleansing step, the system can take further action such as convert all the character to lower case (or upper case) and remove signature information and other formatting layout of the messages.

After this step, the system takes the cleansed message list (sorted in time-order) and does the redundancy checking. Given a cleansed message M, these methods check whether it is repeated in a list of other cleansed message $M_{1},\ldots, M_n$ (the later messages in the global message list).

The first checking method mentioned in this patent is simply check if the M is a substring of any M_i message. However, this method can only detect redundancies if M is quoted completely in others. The second method concatenates $M_1,\ldots, M_n$ into a single string. The system checks whether this string can be divided into segments (can be empty) $T_1S_1\ldots T_mS_mT_{m+1}$ where $S_1\ldots S_m$ is equal to M.

Moreover, a small percentage of $S_1,\ldots,S_m$ are allowed to remain unmatched in $M_1,\ldots M_n$. The third method detects those cases by treating M and concatenation of $M_1, \ldots, M_n$ as two string to be aligned and use dynamic string algorithms.

These above methods have different degrees of efficiency. The first method is the most efficient and the last is the least. However, the sensitivity is in inverse relation to their efficiency.

The last method describe in this patent is the combination of these three methods in order to utilize the efficiency and sensitivity. First, this method use the fast substring test to determine if most of the lines in M are repeated in other email

messages. If this condition is true, the alignment algorithm will be applied for the remaining lines. The steps of the final method are as follows:

- Divide M into non-overlapping segments $S_1, \ldots, S_k$, each of about 50-70 character long.
- Concatenate $M_1, \ldots M_n$ into a single string N.
- If most (for example, 60%) of $S_1, \ldots, S_k$ do not appear as a substring in N, then report that M is not repeated in $M_1, \ldots, M_n$.
- Otherwise, use the string alignment method to decide whether M is repeated in $M_1, \ldots, M_n$.

With the combination of fast substring test and dynamic alignment algorithm, the fourth method has high potential to solve the redundancy problem stated in Chapter 1. In the remaining chapters, we will discuss in detail the implementation and experiment result of these methods.

Chapter 3

Algorithms and Methods

3.1 General redundant message detecting process

For all the redundancy detecting methods presented in this chapter, the process is included in five steps:

1. Connect to the email server and access the examined email folder.
2. Download all the messages in the email folder.
3. Process the contents of messages.
4. Do the redundancy checking on the message list.
5. Compare the output against real redundancy result.

Step 1, 2, 5 are identical for all the methods. The differences between redundancy checking methods are in step 3 and 4. Each method has a corresponding way to process messages' contents. Therefore, the time-performance for each method is the sum of processing messages time and checking redundancy time.

3.2 Benchmarking method

The benchmarking exercise in this project using a method which is quite similar to the method stated in US patent 5905863 (Kimberly A. Knowles and David Dolan Lewis, 1999). The method in this patent uses the "reply-to" field in the header to indicate if a message is repeated in another message. Instead of using the "reply-to" field, benchmarking method uses the subject of the message as the key to detect redundancy.

3.2.1 Messages processing

For each message in the email folder, we extract the subject and do the cleansing process described in section 2.3. The cleansing process also includes changing all the characters to lower case. After this step, the message subject only contains digits and English lower case characters. From this point to the end of this report, the cleansing process on a string is the same for all other methods.

3.2.2 Redundancy checking

To indicate that a message is repeated in another message, this method checks if the subject of the first message is a substring of subject of a second message. For each redundant message, there is a list that holds all the container messages. If a message is found as repeated in another message, we add the second one to the container message list of the redundant message.

```
for each message i in message list {
   if subject of i is not empty {
      for each later message j in the message list {
         if subject of i is a proper substring of subject of j {\
            add j to container message list of i
         }
      }
   }
}
```

Figure 3.1: Pseudo code for benchmarking method

This method base on the common observation that subject of reply email often contains the subject of the quoted email. In most of the cases, the subject of reply emails is auto-generated by the email system if user uses the reply command. The auto-generated subject often has the form “RE: <subject of quoted email>”. Therefore, the quoted email subject is included in side the subject of reply email. We do not consider empty subject since empty string is a substring of every string.

When the user does not use the reply command, most of the time he or she will also include the quoted email subject inside the reply email subject. This task is needed to indicate that this email is the reply to the first email. Comparing to the "reply-to" header field method, this is an advantage.

Subject of a message often a short sentence or phrase with length seldom exceeds 60 characters. This property reduces the precision of this method since short messages tend to be a substring of another message than longer one.

For example, a message has the subject "Meeting" with the content discuss about the meeting of this week. The week after, the user send another email with the same subject for the next meeting. Other users start to reply to the second email with subject "RE: Meeting". This method won't mark first "Meeting" email as repeated in second "Meeting" email since we put the proper substring constraint. However, the system will incorrectly indicate that the first "Meeting" message is repeated in reply messages of second "Meeting" message.

For the bulletin board and USENET newsgroup systems, the rate of incorrectly detecting may be even higher since messages in these systems often belong to some particular subjects. This is a disadvantage compare to the "reply-to" field method.

Other than the differences discuss above, the benchmarking method is quite similar to the method mentioned in Kimberly and David patent. The system cannot detect redundancy if the reply message using non-related subject. In the case that users using reply command to avoid re-type the address and do not change the auto-generated subject, benchmarking method will incorrectly mark this case as redundant.

3.2.3 Benchmarking result

This method has low performance in term of accuracy but it has the best time performance compare with other methods experimented in this project. The

benchmarking method process has the overall precision 54% and recall 53% when testing on the data collection. The detailed experiment result of all the methods discussed in this chapter will be presented in Chapter 4.

3.3 Message-base substring test method

The first examined method in this project is using normal substring test algorithm to detect whether a message is totally repeated in another message.

3.3.1 Messages processing

For this method, we only focus on the body of the message. Therefore, only this part is extracted and cleansed. As mentioned in previous section and Chapter 2, the cleansing process eliminate as much irrelevant as possible. Only digits and English characters are kept, the remaining character is removed. The process also includes converting all the characters to lower case. The next step cannot process without this lower case converting.

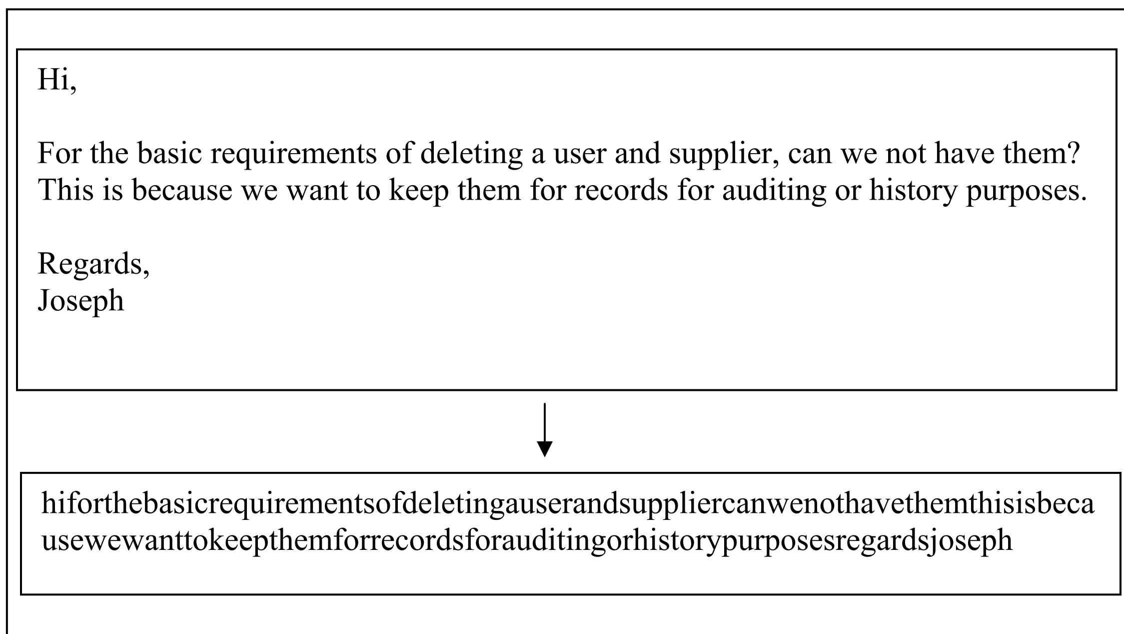


Figure 3.2: Illustration of cleansing message

Figure 3.2 presents the body of an email message before and after cleansing. Note that the cleansed message is broken into two lines to make it printable. The actual cleansed one is a single long line of characters.

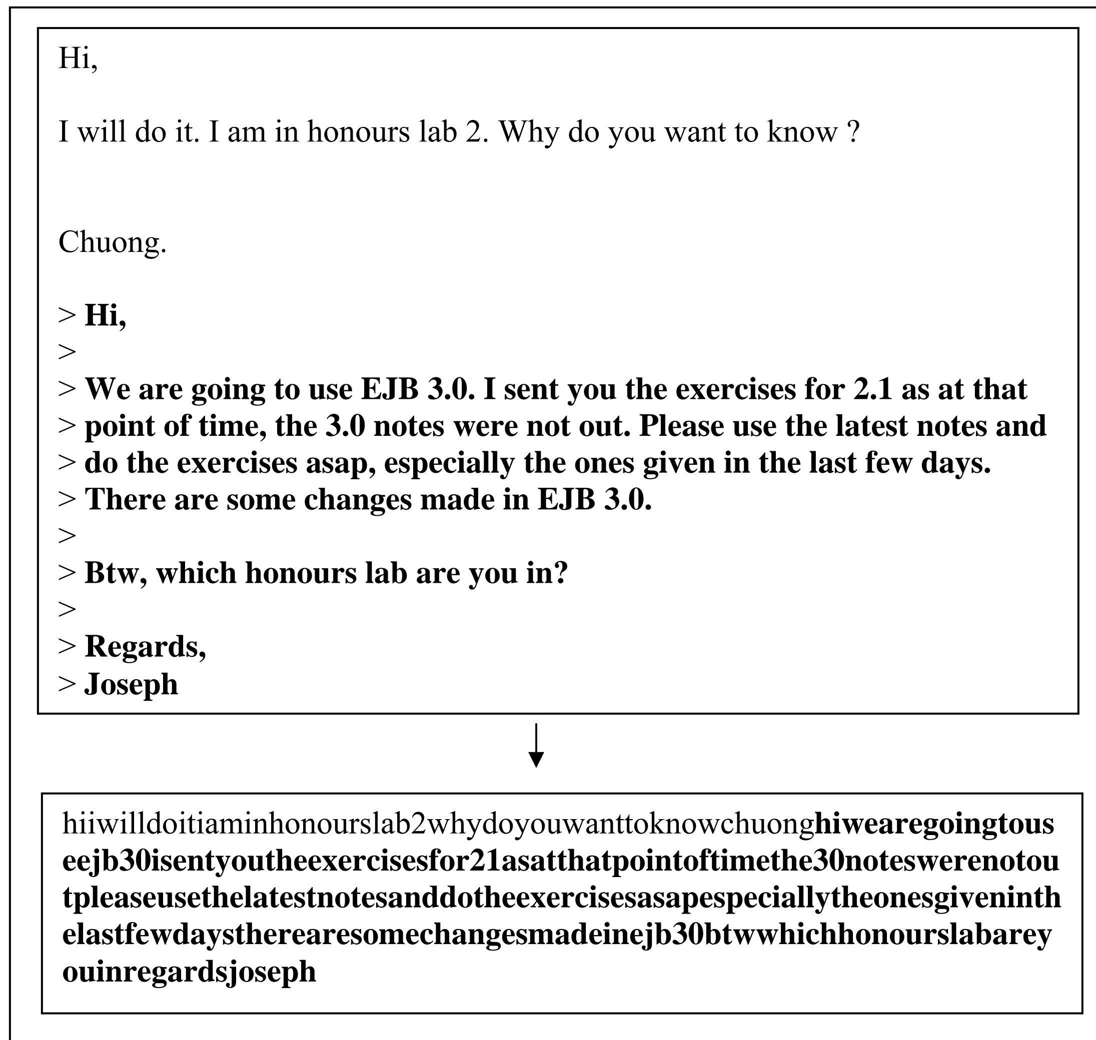
3.3.2 Redundancy checking

After the first step, the message list now contains only cleansed messages. The system go through the messages list, does the substring test of a message and later messages in the list. For any redundancy detected, the system creates a record in the result which simply says which message is repeated in which message.

```
for each message i in message list {

   if body of i is not empty {

      for each later message j in the message list {

         if body of i is a substring of body of j {
            add j to container message list of i
          }
      }
   }
}
```

Figure 3.2: Pseudo code for message-base substring test method

This method can detect all the cases that the message is totally quoted as a continuous block in another message. Figure 3.3 illustrate a typical redundancy case that the method can detect. In this example, we can realize that email system generated characters in container message will be removed in the cleansing process. This will ensure that the first message is a substring of the second one if it is repeated.



Hi,

I will do it. I am in honours lab 2. Why do you want to know ?

Chuong.

> **Hi,**
>
> **We are going to use EJB 3.0. I sent you the exercises for 2.1 as at that**
> **point of time, the 3.0 notes were not out. Please use the latest notes and**
> **do the exercises asap, especially the ones given in the last few days.**
> **There are some changes made in EJB 3.0.**
>
> **Btw, which honours lab are you in?**
>
> **Regards,**
> **Joseph**

hiiwilldoitiaminhonourslab2whydoyouwanttoknowchuong**hiwearegoingtous eejb30isentyoutheexercisesfor21asatthatpointoftimethe30noteswerenotou tpleaseusethelatestnotesanddotheexercisesasapespeciallytheonesgiveninth elastfewdaysthereareso mechangesmadeinejb30btwwhichhonourslabarey ouinregardsjoseph**

Figure 3.3: A container message before and after cleansing

From the data collection in this project, the rate of totally redundancy as this case is high (70% of the redundant message). This result can be explained as users often use reply command to quickly reply to received-emails. However, this rate may change according to different users or different kind of message systems (bulletin board and USENET newsgroup).

3.3.3 Boyer-Moore algorithm

As mentioned in the Kwok-Wong patent, we can use fast substring algorithms instead of normal substring algorithm to speed up the string comparison process. Applying Boyer-Moore algorithm (R. S. Boyer and J. S. Moore, 1977) in this method, the overall redundancy detecting time is improved considerable. The detail of results will be discussed in Chapter 4. For the rest methods mentioned in this chapter, we use Boyer-Moore algorithm for substring tests.

3.4 Phrase-base substring test method

Although message-base substring method can detect up to 70% of redundant email, it cannot detect cases that repeated messages are divided in to smaller portions and quoted in many other messages. The method presented in this section divides the examined message into atomic unit called phrases and searches for these phrases in later messages.

3.4.1 Phrases

To detect the case that message is divided into smaller portions, we should define what should be an acceptable portion as mentioned in section 1.3. This portion should be the atomic part of a message. We do not allow a portion to change seriously in the container messages; otherwise, we consider it as not repeated.

For this project, such a portion is called a phrase. A phrase is defined as a part of the message body that line between two of the following characters: period, question mark, exclamation mark. A phrase must have length greater than zero after being cleansed.

It is reasonable to choose period, question mark and exclamation mark as delimiters of a phrase. In writing, the portion that is in between these characters is a sentence and it represents some particular idea. Therefore, we do not allow serious changes in a phrase to ensure that the meaning of the quoted message is preserved.

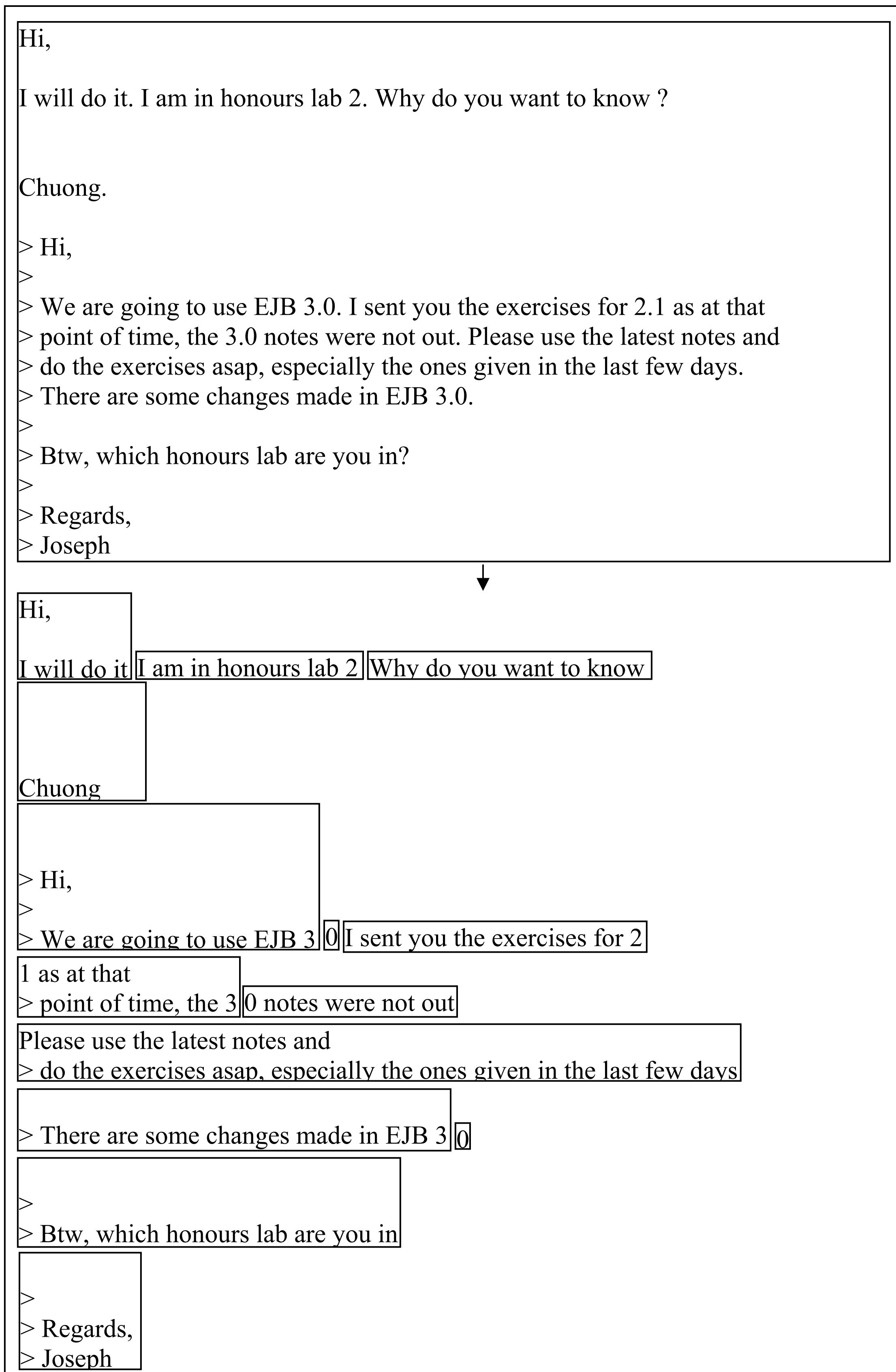


Figure 3.4: Illustration of dividing message into phrases

3.4.2 Messages processing

In the messages processing step, beside cleansing the message body, the system also divide the message into phrases using period, question mark and exclamation mark as delimiters. The system also cleanses phrases before continuing with redundancy detecting step.

Figure 3.4 presents an example of dividing message into phrases. Note that we only remove the delimiters and separate phrases, other characters are still kept.

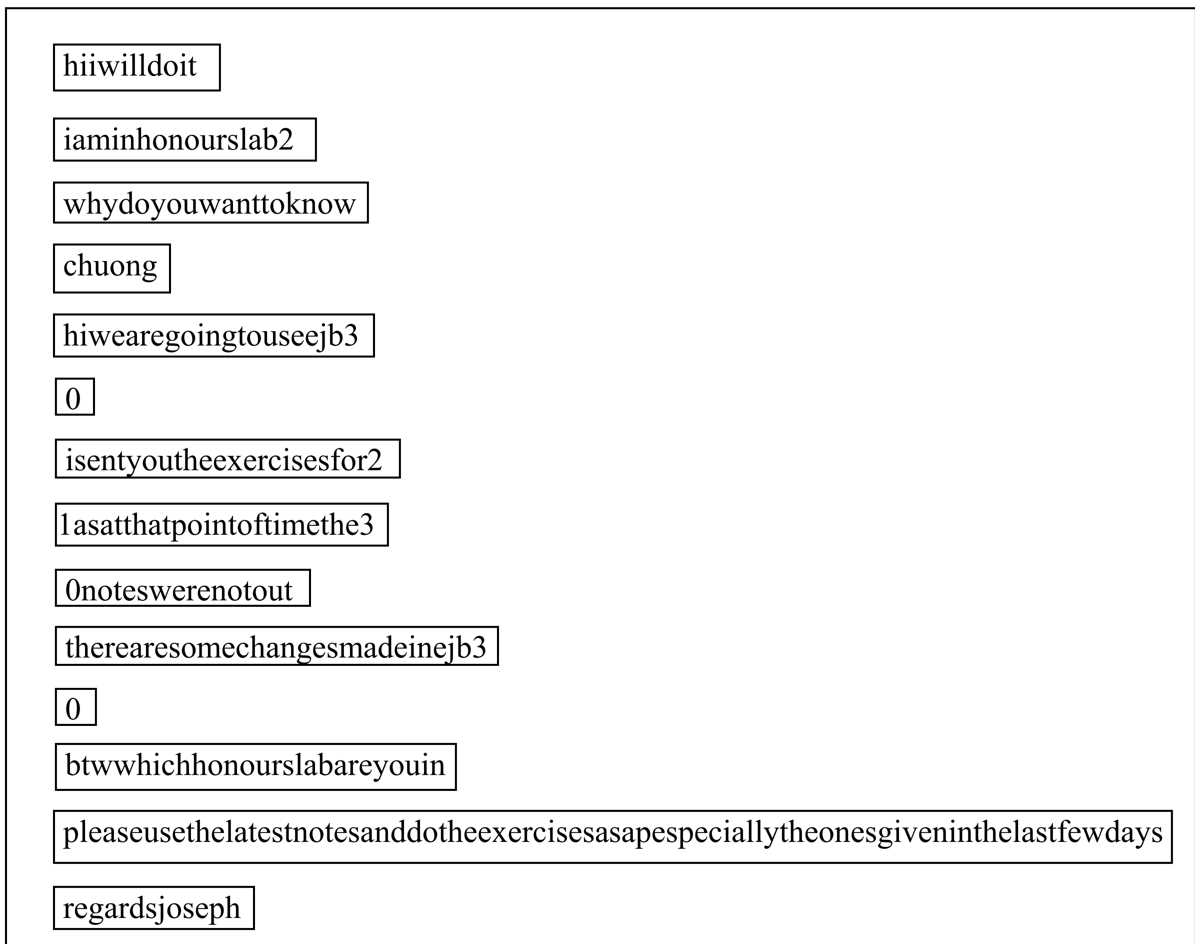


Figure 3.5: List of cleansed phrases of a message

Figure 3.5 presents the phrase list of the message in Figure 3.4. We can see that sometimes a phrase may not be a literal sentence. For example, the sentence “There are some changes made in EJB 3.0.” is separate into 2 phrases: “There are some changes made in EJB 3” and “0” which is not a literal sentence. We may want to further extend the definition phrase to limit those cases. However, in this project, we use the current definition.

3.4.3 Redundancy checking

After processing step, each message has a list of phrases. The checking method is the same as message-base method. The only modification is instead of searching for the whole message, the system check phrases one by one to indicate which later messages contain current phrase. Note that we still keep the whole cleansed body of the message so that we could do a substring test on the message body.

```
for each message i in message list {
   for each phrase k in phrase list of i {
      for each later message j in the message list {
         if k is a substring of body of j {
            add j to redundancy list of k
         }
      }
   }
}
```

Figure 3.6: Pseudo code for message-base substring test method

Each phrase of a message now holds a list of messages that contain the phrase as a substring. From this list, we can indicate whether a message is redundant or not. If all the lists of phrases are not empty (each phrase is at least quoted in another message), then we can conclude that the message is redundant. These lists can further indicate a message is quoted totally or partially in another message.

However, this method is highly sensitive compare to the message-base method since short phrases tend to repeated in many non-relevant messages. For example, phrase “0” in message of Figure 3.5 will be marked as repeated in all messages that have “0” as substring. Figure 3.7 presents such a message.

> Hi all,
>
> I will come late tomorrow since I have another meeting at 4:3**0**. You can do other parts first.
>
> See you tomorrow.
>
> Chuong.

Figure 3.7: A non-relevant message that has “0” as substring

3.5 Edit distance method

Fast substring methods allow us to find redundancies if the phrases are quoted without modification. As mentioned in section 1.3, we still allow small number of phrases to be modified in the container message. To handle such cases, we use edit distance matrix (V. I. Levenshtein, 1966).

3.5.1 Edit distance matrix calculation

The edit distance between two strings is the number of edit operations such as insertion, deletion and substitution needed to convert one string to another string. We can use dynamic programming to calculate edit distance.

Edit distance matrix of two strings S1, S2 is constructed using the following rule:

$$D(i, j) = \min[D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + t(i, j)]$$

In this formula, D(i, j) is the edit distance of string S1[1..i] and string S2[1..j]. The term t(i, j) has value 0 if S(i) = S(j) and value 1 if S(i). Using the above formula together with base conditions D(i, 0) = i and D(0, j) = j, we can calculate all the values in edit distance matrix. The final value D(n, m) where n is the length of S1 and m is the length of S2 is the edit distance of two strings.

		s	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
s	1	***0***	***1***	***2***	3	4	5	6	7
u	2	1	1	2	***2***	3	4	5	6
n	3	2	2	2	3	***3***	4	5	6
d	4	3	3	3	3	4	***3***	4	5
a	5	4	3	4	4	4	4	***3***	4
y	6	5	4	4	5	5	5	4	***3***

Figure 3.8: An example of edit distance matrix

Figure 3.8 presents the edit distance of two strings S1 equals to “sunday” and S2 equals to “saturday”. The optimal alignments are highlighted. We may have more than one optimal sequence of operation for any two strings.

Starting from any position (i, j) of the matrix, we can always trace back the optimal sequences of operations taken to convert string S1[1..i] to S2[1..j]. We can simply use a recursive function to find these sequences. The recursive function applied on (i, j) checks which of the three terms D(i – 1, j) + 1, D(i, j – 1) + 1 and D(i – 1, j – 1) + t(i, j) is equal to D(i, j) and apply itself on this term. In the case that we have more than one term equal to D(i, j), recursive function is applied on all of these terms and we will have many optimal sequences. The base case of the recursive is the origin (0, 0).

In term of converting S1 to S2, if we move from (i – 1, j) to (i, j) then we take a deletion operation on string S1 at position i. If we move from (i, j – 1) to (i, j) then we take an insertion operation on string S1 after position i. If we move from (i –

1, j – 1) to (i, j), it will be an substitution if S1(i) ≠ S2(j), otherwise, it is a match and no operation is taken.

In term of converting S2 to S2, it is the same as above except that (i – 1, j) to (i, j) is an insertion and (i, j – 1) to (i, j) is a deletion.

For this project, we make a small modification to the calculation of edit distance matrix. We use the matrix to check if a string is an approximate substring of another string. The operations that we allow be applied on the first string are deletion and insertion. As a phrase is quoted in another message, we only allow deletion of some unimportant tokens or insertion of other tokens to the original phrase in the container message. Therefore, any substitution operation is considered as a deletion followed by an insertion. To achieve this, we modify the cost of substitution to 2. t(i, j) equals to 0 if S1(i) = S2(j) and 2 if S1(i) ≠ S2(j).

3.5.2 Redundancy checking

The messages processing step for this method is identical to the process described in section 3.4.2. We divide messages into phrases and do the cleansing process. To check if a phrase is repeated in a message, we calculate the edit distance matrix of the phrase and the message body. If the number of operations taken to convert the phrase to the message body equals to the difference in length of these two strings, we conclude that the phrase is repeated in the message body.

With the condition edit distance equals to difference in length, we only allow insertions into the phrase to convert it to the examined message body. This means that the message body must contain all the characters in the phrase and the internal orders of these characters are kept. However, this method also allows insertions of other characters in between phrase’s characters. This limitation makes edit distance method even more sensitive than then phrase-base substring method.

```
for each message i in message list {

   for each phrase k in phrase list of i {

      for each later message j in the message list {
         if (length of j >= length of k) {

            if ed(k, j) == (length of j – length of k) {

               add j to container message list of k

             }
         }
      }
   }
}
```

Figure 3.9: Pseudo code for edit distance method

Figure 3.10 presents an example of incorrectly redundancy detecting using the edit distance method. The number of insertion needed to convert the phrase to the message exactly equals to the difference in length of two strings. The shorter the length of the phrase (i.e. less complicated), the higher chance it will appear in a non-relevant message. The longer messages also tend to be marked as containing non-relevant phrases.

> I will come late tomorrow.

> **I will** do the implementation part first. Please remember to **come** for the meeting. Coming **late** or absence is not acceptable. See you **tomorrow**.

Figure 3.10: An example of incorrectly redundancy detecting using edit distance only

3.6 Constraint edit distance method

Using the edit distance only cannot constraint the number of insertion into the phrase. Moreover, this method does not allow deletion a small portion of the phrase. In this section, we discuss a method that use edit distance matrix with constraints to solve those problems.

3.6.1 Trace back algorithm

As mentioned in section 3.5.1, we can find all the optimal alignments using edit distance matrix. An optimal alignment can be presented by a path starting from position (n, m) to the origin (0, 0). Any move in the path corresponding to an operation. In term of converting S1 (left column of the matrix) to S2 (top row of the matrix), left move is an insertion, up move is a deletion. Diagonal move is a match or a substitution depend on the values of S1(i) and S2(j).

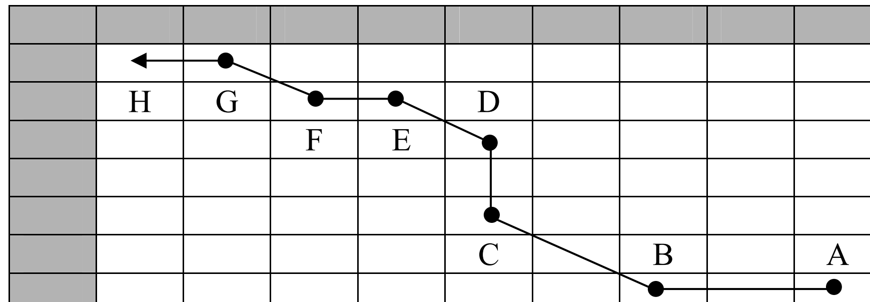


Figure 3.11: Path of an optimal alignment

In Figure 3.11, AB, EF and GH segments are sequences of insertions while CD is a sequence of deletion. BC and FG are sequences of matches and substitution.

We want to control the number of extra insertion into the phrase. With the initial condition that the length of S1 is less than or equal to S2, there will be a number of insertions at both end of S1 as S1 is converted to S2. These insertions are represented by AB and GH segments and they are allowed. We want to constraint insertions that are in between matches such as the EF segments. The number of

deletions and substitution is simply to control since we only need to consider the up moves and diagonal moves with S1(i) ≠ S2(j).

Therefore, we use a recursive function that take in edit distance matrix, a position (i, j), number of allowed insertions, number of allowed deletions, maximum insertions, maximum deletions and a boolean to indicate if sequence of matches has started or not.

```
function boolean traceBack (matrix, (i,j), ins, del, maxIns, maxDel, st) {
   if (i == 0) return (ins <= maxIns) & (del <= maxDel);
   else if (j == 0) return (ins <= maxIns) & (del + i <= maxDel);
   else {
      boolean result;
      if (matrix[i-1][j] + 1 == matrix[i][j] & del < maxDel)  {
         result = traceBack (matrix, (i-1, j), ins, del + 1, maxIns, maxDel, st);
      }
      if (result) return true;
      else if (matrix[i][j-1] + 1 == matrix[i][j]){
         int newIns = ins;
         if (st) newIns = newIns + 1;
         if (newIns <= maxIns) {
          result = traceBack (matrix, (i-1, j), newIns, del, maxIns, maxDel, st);
         }
      }
      if (result) return true;
      else if (matrix[i-1][j-1] == matrix[i][j]) {
         result = traceBack (matrix, (i-1, j-1), ins, del, maxIns, maxDel, true);
         return result;
      }
   }
}
```

Figure 3.12: Recursive trace back algorithm with constraints

The algorithm in Figure 3.12 look for next optimal move by comparing matrix[i][j] with matrix[i –1][j] + 1, matrix[i][j – 1] + 1 and matrix[i – 1][j – 1]. If a move is an optimal move, the recursive call will be applied on the corresponding position to check for maximum allowed insertions and deletions. Note that in this algorithm, we do not allow substitutions (only take matches with t(i, j) = 0). For the next move, we adjust the current number of insertions and deletions and pass in as parameters for the next recursive call.

The base cases are i or j equals to 0 which means we reach the top or left side of the matrix. At this point, only insertions or deletions are available. The insertions (reaching the top side) are legal since we cannot have up or diagonal moves from this point. Therefore, we simply check the final insertions and deletions taken and return the result. For the deletions (reaching the left side), we need to add to the current number of deletions the remaining deletions left and return the final result.

Combining with the edit distance checking, the final method allows us to constraint the number of max operations, max insertions in between and max deletions.

```
function checkWithConstraints (matrix, maxOps, maxIns, maxDel) {
   if (matrix[n][m] <= maxOps) {
      return traceBack (matrix, (n, m), 0, 0, maxIns, maxDel, false);
   } else return false;
}
```

Figure 3.12: Check substring with constraints function

3.6.2 Redundancy checking

Although the checkWithConstraints allow both insertions and deletions applied on the examined string, we may want to further constraint the max operations, max insertions and max deletions input to ensure that the meaning of the phrase will not change seriously in the container message.

Therefore, we only allow either insertions or deletions but not both. For the insertion case, we simply call the checkWithConstraints function with max operation is the difference in length of two strings. The max deletion is set to 0 and the max insertion.

For the deletion case, we need to increase the max operations corresponding to number of allowed deletions. For example, S1 has length n and S2 has length m and S2 contains S1 with some deletions. Then m = (n – i) + k where k is the length of remaining part of S2. To convert S1 to S2 we must at most insert k characters to S1 and delete i characters from S1. Therefore, the max operations are k + i which equals to m – n + 2i.

To check whether phrase S1 is contains in S2 with i allowed deletions, we use the following function call:

```
checkWithConstraints (matrix, m – n + 2i, 0, i);
```

However, in some special cases, the edit distance between S1 and S2 is less than m – n + 2i operations and these optimal alignments needed other operations to convert S1 to S2. In this case the algorithm return false and cannot detect string S1 is quoted in S2.

The example in Figure 3.13 presents such a case. The word “with” is deleted when the sentence is quoted. However, there is a phrase “with other teams” appears later in the container message. Therefore, to convert S1 to S2, we only need m – n insertions to S1 and this is the optimal alignment. The number of allowed insertions is zero so checkWithConstraints will return false for this case.

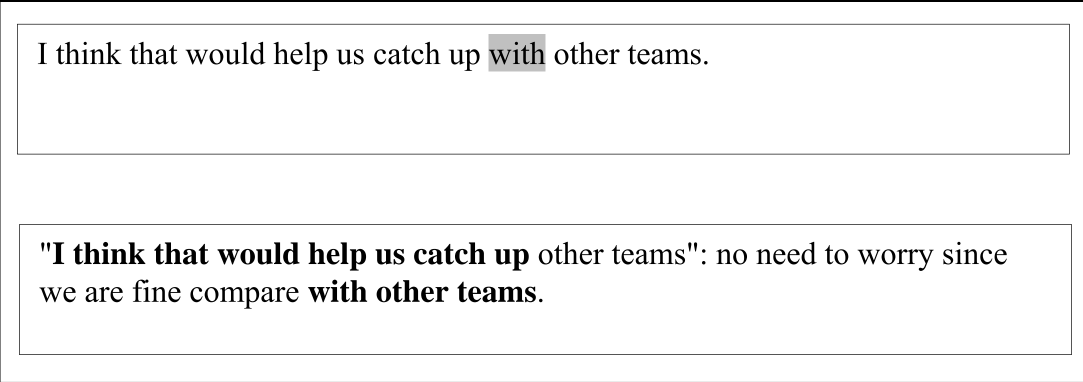


Figure 3.13: An example of undetected repeated phrase with deletions

Another limitation of this method is the time-performance. Since we use recursive method to calculate the path from the position (n, m) to the origin (0, 0), the number of recursive call is n + m. This is in case we have only one optimal alignment. The length of a phrase is often around 50 to 70 characters long. However, the length of a message can reach thousands. By experiments, we realize that for such long string, the system will throw stack over flow memory exceptions. Therefore we put in one assumption to avoid such cases. We assume that for any phrase quoted in another message, it cannot be separated in many phrases in that message. This means that we only allow the phrase to be quoted in another phrase (may equal to itself with some modifications). This assumption is reasonable since the phrase meaning may change if we divide it into smaller portions.

According to the above assumption, we check the current phrase against other messages' phrase instead the messages themselves. The pseudo code for this redundancy checking is presented in Figure 3.14

```
for each message i in message list {

   for each phrase k in phrase list of i {

      for each later message j in the message list {

         for each phrase p in phrase list of j {

            if compare(k, p) {

               add j to container message list of k
            }
         }
      }
   }
}
```

Figure 3.14: Pseudo code for redundancy check process of constraint edit distance method

As mentioned before, we only allow only either insertions or deletions but not both. The amount of insertions and deletions in this method is 10% of the length of examined phrase. The detail of the phrase to phrase comparison is described in figure 3.15.

```
function Boolean compare (k, p) {

   n = length of k;

   m = length of p;

   maxDel = (n / 10);

   maxIns = (n/ 10);

   matrix = calculateEditDistance(k, p);

   if (length of k > length of p) {

      result = checkWithConstraints (matrix, m – n + maxDel * 2, 0, maxDel);

   } else {

      result = checkWithConstraints (matrix, m – n, maxIns, 0);

      if (!result) checkWithConstraints (matrix, m – n + maxDel * 2, 0, maxDel);

   }

   return result;
}
```

Figure 3.15: Pseudo code for compare function

3.7 Combine method

In the last method of this project, we combine the phrase-base substring test method with constraint edit distance method to yield the best performance for redundancy checking.

According to this compare function, if the length of phrase k is greater than then length of phrase p then we check if p is a result of a deletion sequence on k. The number of insertions allowed is zero for this check. This check can detect the case that phrase p is phrase k with some deletions.

If the length of k is less than or equal to p, we check if k is repeated in p. Two checking steps are performed for this case. The first test covers the case in which p quotes k and also inserts some words in between. The second test detects if p quotes k with some words is deleted. If one of these test return true, we add message j to container message list of k.

3.7.1 Messages processing

For this method, we still use the phrase-base approach as described in previous sections. We only add in the salutation, conclusion and signature detecting process for this step.

As mention in section 1.3, the unimportant information like salutation, conclusion and signature are often deleted in the container message. Therefore, in the message processing step, we also check if those parts are existed. Later if some phrases cannot be found as repeated in other messages and they are those unimportant parts, we still consider the message as redundant.

In this step, we also classified phrases into two types: normal phrase and short phrase. The only difference is the length of two types. We use a constant to

indicate the minimum length for a normal phrase. If the length of the phrase is less than this constant, we classify it as short phrase. In the experiments described in Chapter 4, the value of the constant PHR_LEN_THESHOLD is 15.

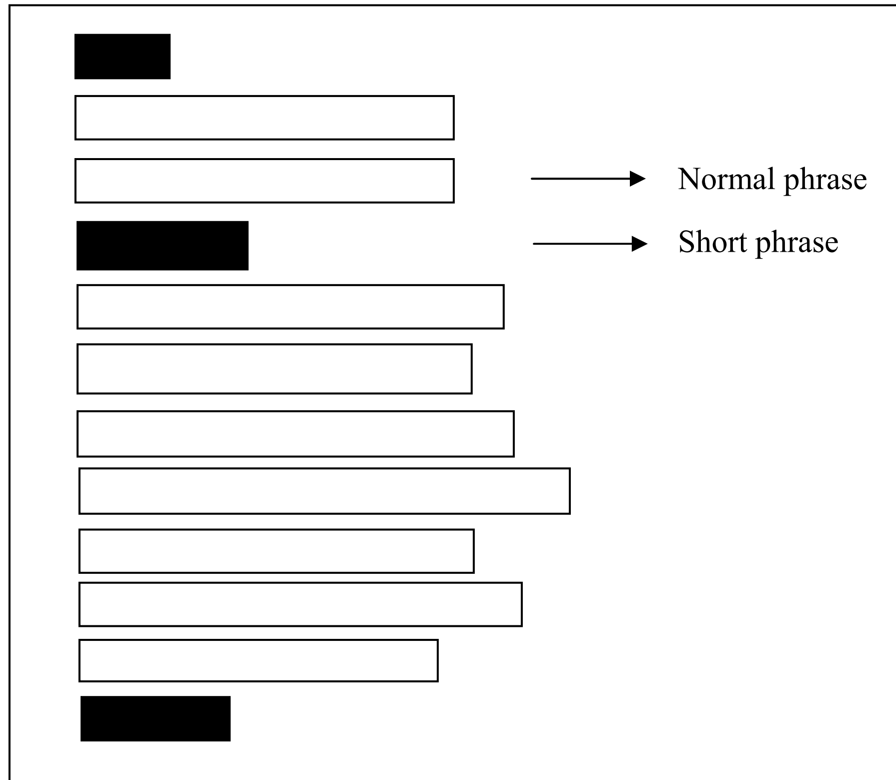


Figure 3.15: An example of phrases classification

3.7.2 Redundancy checking

For each message in the message list, we only do redundancy checking if the length of the message is not less than the constant MSG_LEN_THRESHOLD. Since email users sometimes use email to transfer files as attachments, the text content of the message is often very short in this case. For example, a user may want to transfer a picture to a friend. He or she sends an email with the content “Picture” and the attachment is the picture itself. We do not want to mark the message as redundant if there is another unrelated message that contains the word “picture”. To avoid such cases, we perform redundancy checking if the message is not too short. For this method, we set the MSG_LEN_THRESHOLD to 15.

The redundancy checking process for each message in this method includes 2 rounds. In the first round, we search for the normal phrase in the later message list using the fast substring test. After this round, we check if most of the normal phrases are found as redundant, we continue to the second round. If all the phrase in the message are short phrase, we simple search for the content of the message in later messages as in message-base substring test method.

The condition to check the second round changes base on the number of normal phrases in the message. If number of normal phrase is less than 5 then only 50% of the phrases must be found in first round as redundant. If the number of normal phrases is from 5 to 10 then 80% must be found. Otherwise, 90% of the normal phrases must be found. This policy based on the observation of current data test collection and can be configured for different system or users.

In this round, all the short phrases and remaining normal phrases are examined. Checking redundancy for these phrases is highly sensitive. In the case of short messages, they tend to be a substring of many non-relevant messages. For not found normal phrases, we must use the constraint edit distance method to search.

Therefore, we must narrow the search range of these phrases. For each of them, the method finds the immediately previous and next normal phrases that are marked as redundant in first round. The method retrieves redundancy message list of these two phrases and search for the second round phrase in these lists. Figure 3.16 presents this process. Note that we also search for unfound normal phrases (the grey bar in the figure) in the nearest redundant normal phrases.

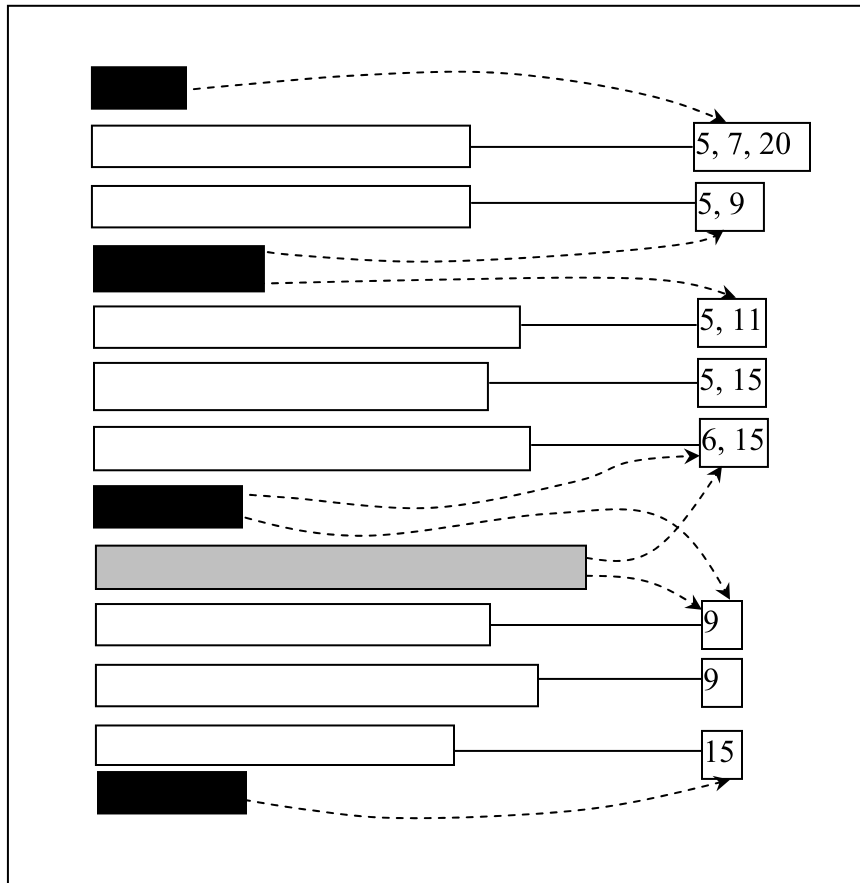


Figure 3.16: Searching for second round phrases in the container message list of nearest redundant phrases

Minimizing the search range to redundancy lists of nearest messages is acceptable since we want phrases of a message are not quoted separately in too many later messages. In the case of short phrases and not found normal phrases, it is even more reasonable to search for them in messages that are identified as containers of other phrases in the same message. The constraint allows second round phrases to be quoted only in messages that contain previous or next found normal phrase. This improves the precision rate as most of the incorrectly detections due to sensitivity are eliminated.

For short phrase, we use the substring test to check if they are redundant. We do not allow these phrases to change in the container message since. We use the same method described in previous method for the normal phrases that are not found as

redundant in first round. Only one small modification in this method: only compare not found normal phrase with normal phrases of the examined message.

After the second round, we continue to check those phrases are still not marked as redundant. If they are salutation, conclusion or signature, of the message, we still conclude that the message is redundant base on remaining phrases result list. Otherwise we consider this message is not redundant.

The current methods for checking unimportant parts are quite simple. In the message processing state, we treat the first comma of the first sentence as a phrase delimiter. If there is the salutation in the email, the first phrase (after cleansing) will start with one of the following term “hi”, “hello”, “hey”, “dear” or “mydear”. If the first phrase is not found in any message and it start with one of the above string, we can omit this phrase.

For the conclusion, the acceptable phrases are “bestwishes”, “regard”, “bestregard”, “goodwishes”, “yourssincerely”, “sincerelyyours”. The signature can only be the last phrase. It must has length less than 9 characters or a phrase with number of words is less than 7 while each word must start with a upper case character.

This method combines all the techniques that used in previous sections. It can detect most of the redundant messages in the test data collection. However, in some cases, deleting some words from a phrase cannot be detected as described in section 3.6.2. The current methods used to detect salutations, conclusions and signatures also have limitations. They cannot detect all these portions for some complicated cases. Moreover, some other terms can be incorrectly marked as unimportant portions and removed in the container message while this method still allows such cases.

For example, this method cannot long signatures which includes address and other information of sender:

ZHENG Jun, Raymond
Teaching Assistant
Department of Information Systems
National University of Singapore
S15 #04-28, 3 Science Drive 2, Singapore 117543
e-mail: diszj@nus.edu.sg
Tel: (65) 6516-4246

Other terms whose first character is upper case will be considered as signature if they are in the last phrase of the message. Some examples are “Happy New Year” and “Merry Christmas”. If these terms are the last phrase, the system will allow them to be removed in the container message.

Chapter 4

Experiments and Results

4.1 System specification

The system implements methods presented in Chapter 3 is written in Java 1.6.0. The system runs on a 1.73 GHz Pentium (R) M machine with 1GB memory running Windows XP.

We use the JavaMail 1.4 package to establish connection to the Unix mail server. The IMAP and SSL protocol are used. Testing email collection is stored in an email folder and the system downloads all these emails before processing. These are first two steps of the redundancy checking process mentioned in section 3.1.

4.2 Test data specification

Test data used in this project is a collection of 286 email messages. These emails are chosen from different mail boxes. Since this is testing data for redundancy checking, emails are chosen so that the rate of redundancy is higher compare to the source mail boxes. Note that the redundancy rate is various based on different type of message systems and users.

The detail information of the data collection are described in table 4.1

No. email messages	286
No. redundant messages	105
No. single redundancies	218
No. combo redundancies	9
No. total redundancies	227

Table 4-1: Detail information of data collection

The single redundancies are cases in which a message is totally quoted in another message. These cases also include removing unimportant portions like salutation, conclusion and signature in container messages. Combo redundancies indicate cases in which part of a message is quoted in another message and all the part of the redundant message (except unimportant portions) must be found in at least one. We will discuss some examples of redundancy result in next section.

4.3 Redundancy result

In order to compare the output results of different methods with the real result, we define a redundancy result structure to represent such output. Associating with each redundant message, there is a redundancy result which has the structure described below.

```
RedundancyResult ::= MsgID NUMBER SingleContainer* ComboContainer?
SingleContainer  ::= MsgID
ComboContainer   ::= COMBO NUMBER MsgID*
```

MsgID is the Message-ID field in the header of an email message. Note that for each message in a mail server, Message-ID is unique. Therefore, we can use Message-ID to differentiate between messages in a mail box.

The MsgID in the definition of RedundancyResult is the ID of the redundant message. The next term, NUMBER, is the number of SingleContainer and ComboContainer in the RedundancyResult. A SingleContainer is simply a Message-ID of a container message that quotes whole redundant message.

The ComboContainer is a list of messages that quote part of the redundant message. The union of these parts must be the original message. The NUMBER in the

definition indicates the size of the list. For each RedundancyResult, there is at most one ComboContainer.

Figure 4.1 presents an example of RedundancyResult without ComboContainer. The first line is the ID of the redundant message. The second line is the number indicates the number of SingleContainer (there is no ComboContainer in this example). The next two lines is the message ID of container message that quote the redundant message.

```
<200402082305.HAA01789@sf3.comp.nus.edu.sg>
2
<200402112305.HAA17952@sf3.comp.nus.edu.sg>
<Pine.GSO.4.58.0407281147400.5063@sf3.comp.nus.edu.sg>
```

Figure 4.1: An example of RedundancyResult without ComboContainer

Figure 4.2 presents another example of RedundancyResult with ComboContainer. In this example, we have 3 SingleContainer and the ComboContainer so the number is 4. The ComboContainer has 3 messages with IDs indicate by the last 3 lines.

```
1310.172.18.197.226.1098293138.squirrel@172.18.197.226>
4
<001001c4b742$167ad610$ce134adb@winxp>
<73F60BB0A14F2D4B91B99A94DE6565A25ED38F@MBOX22.stu.nus.edu.sg>
<2431.172.18.136.136.1098363189.squirrel@172.18.136.136>
COMBO
3
<3291.172.18.193.160.1100537758.squirrel@172.18.193.160>
<001601c4cbd9$34b9cc70$561b4adb@winxp>
<2331.172.18.181.170.1100620974.squirrel@172.18.181.170>
```

Figure 4.2: An example of RedundancyResult with ComboContainer.

We define the total redundancies of a RedundancyResult is the total number of container messages that quote part or whole redundant message. For example, the total redundancies of RedundancyResult in Figure 4.1 are 2 and Figure 4.2 is 6. This definition will be used in section 4.5 to calculate the accuracy of output result.

With this definition structure, we can easily indicate which message is quoted in which messages and whole or part of the message is quoted. The redundancy result of an email folder is a list of RedundancyResult. Each redundant message in the folder associates with a RedundancyResult.

4.4 Redundancy result construction

For each method mentioned in Chapter 3, every redundant message (benchmarking method and message-base substring test method) or redundant phrase (phrase base substring test method, edit distance methods and combine method) has a list of container messages. The construction of RedundancyResult discussed in section 4.3 is different for message-base and phrase-base methods.

For message-base methods, we can only detect which message is repeated in which message. Therefore the RedundancyResult only contains SingleContainer. To construct the RedundancyResult, we simply go through the container message list in each redundant message and create corresponding list of SingleContainer.

For phrase-base method, each redundant message has a list of phrases. Each phrase also has a list of container messages. To construct the RedundancyResult for the redundant message, we first find the intersection I and union U of all phrases' container message list. The intersection I is the list of SingleContainer. We create S which is a list of messages such that S = U – I. For each phrase p, we check whether the intersection of container list of p and S is empty or not. If all of these intersections are not empty then S is the COMBO list. Otherwise, there is no COMBO list. Figure 4.3 presents an example of constructing SingleContainer and ComboContainer lists. The numbers in this figure represent the message IDs.

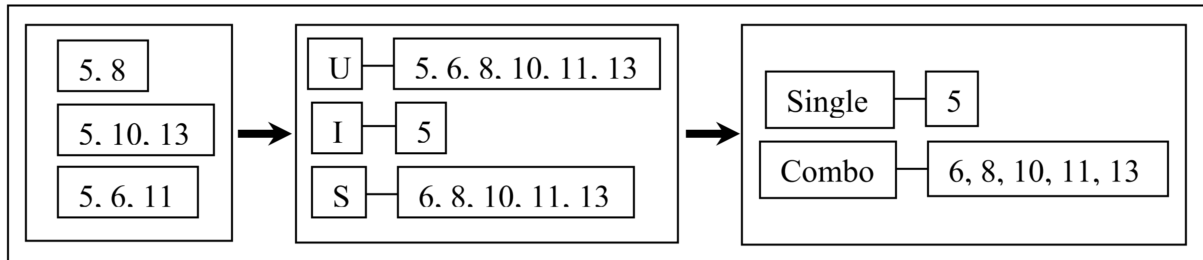


Figure 4.3: Constructing SingleContainer and ComboContainer lists from redundant message phrases' lists

4.5 Redundancy result comparison

4.5.1 General precision and recall

Precision and recall are two measures that are widely used in information retrieval (IR). In this project, we also use these measures to evaluate the output results of redundancy checking methods.

	Real redundant messages	Real non-redundant messages
System indicates as redundant	a	b
System indicates as non-redundant	c	d

Table 4-2: Parameters for calculating general precision and recall

According to Table 4-2, the message-base precision and recall measures of a method are calculated by the following formulae:

$$\text{General precision} = \frac{a}{a + b}$$

$$\text{General recall} = \frac{a}{a + c}$$

From the list of real RedundancyResult and system output RedundancyResult, we simply find redundant messages that are in the output list but not in the real list. The number of these messages is parameter b. Note that the size of output list is (a + b) and size of real result list is (a + c). a can be found by using the simple formula: a = (a + b) – b. With all the given parameters, we could calculate the precision and recall.

4.5.2 Detailed precision and recall

Using the general precision and recall can only give the detailed accuracy at the message level. Two different list of RedundancyResult may give the same result since the general precision and recall can only tell which message is redundant but not quoted in which message.

Using the RedundancyResult structure, we can use the number of SingleContainer and ComboContainer instead of the message only. The parameters for new formulae are described in Table 4.3

	Total redundancies in real result	Total redundancies not in real result
Total redundancies in system output	a’	b’
Total redundancies not in system output	c’	d’

Table 4-3: Parameters for calculating detailed precision and recall

The term “Total redundancies” indicates the sum of RedundancyResults’ total redundancies in a list of RedundancyResult. For example, “Total redundancies

in real result" is the sum of RedundancyResults' total redundancies in the real RedundancyResult list.

Term (a' + c') is the total redundancies in real result list and (a' + b') is the total redundancies in system output list. We apply the same precision and recall formulae to find the detailed measures.

$$\text{Detailed precision} = \frac{a'}{a' + b'}$$

$$\text{Detailed recall} = \frac{a'}{a' + c'}$$

We find b' by comparing two RedundancyResult lists. For each RedundancyResult r_1 in the output result list, we find if there is a RedundancyResult r_2 with the same MsgID in the real list. If this condition is true, we count all container message IDs that is in r_1 but not in r_2 (including container message ID in the ComboContainer). If there is no such r_2, the number counted will be the total redundancies of r_1. b' is the sum of all number of unfound redundancies.

4.6 Experiments result

4.6.1 Benchmarking method

The redundancy result output for benchmarking method contains only SingleContainer since this is a message-base method. We can only indicate which message is repeated in which message. The detailed result is in Table 4-4 and Table 4-5.

Measure	Value
General precision	0.544
General recall	0.533

Table 4-4: Precisions and recalls of benchmarking method

Process	Time (ms)
Messages processing	203
Redundancy checking	32
Total	235

Table 4-5: Time performance of benchmarking method

The RedundancyResult of this method only contains SingleContainer since it is a message-base method. Therefore, we only calculate the general measures. The precision and recall measures for this method are low because we only use the subject to check for redundancies. Especially, the precision rate is very low. This means that the rate of incorrectly redundancy detecting is high.

The measures in this method are unpredictable since the redundancies in contents of messages are not totally related to the subjects of the messages. The result also depends on the types of message system and users.

However, the time performance is the best comparing other methods. The benchmarking method needs only 32 ms to do the redundancy detecting. Message processing step takes most of the total time since it includes the cleansing process.

4.6.2 Message-base substring test method

This method is a message-base method like the benchmarking method so we do not calculate the detailed precision and recall. The experiment results are presented in Table 4-6 and Table 4-7.

Measure	Value
General precision	0.986
General recall	0.676

Table 4-6: Precisions and recalls of message-base substring test method

Process	Time (ms)
Messages processing	484
Redundancy checking	204
Total	688

Table 4-7: Time performance of message-base normal substring test method

The expected precision of this method is 1.0. This means that the method will never mark non-redundant messages as redundant. However, for short email messages with attachments mentioned in section 3.7.2, this method will incorrectly marked the message as repeated in non-relevant messages. The recall measure is low since this method cannot detect the cases in which message are divided into smaller portions.

Table 4-7 presents the time performance with normal substring test. The implementation with Boyer-Moore algorithm in Table 4-8 shows a better result. Time to perform redundancy checking has been reduced from 204 ms to 141 ms.

Process	Time (ms)
Messages processing	478
Redundancy checking	141
Total	619

Table 4-8: Time performance of message-base fast substring test method

Although this method is 3 time slower than the benchmarking method, the precision and recall measures are better and more stable.

4.6.3 Phrase-base substring test method

The experiment results of this method is showed in Table 4-9 and Table 4-10

Measure	Value
General precision	0.986
General recall	0.733
Detailed precision	0.743
Detailed recall	0.801

Table 4-9: Precisions and recalls of phrase-base substring test method

Process	Time (ms)
Messages processing	447
Redundancy checking	3422
Total	3869

Table 4-10: Time performance of message-base substring test method

This method also has the same problem with short message as the previous one. Therefore the general precision is not 1.0. The general recall is improved since more redundant messages are detected (parts are quoted separately). However, the detailed precision is quite low. This is because short phrases (usually not literal sentences) tend to appear in many non-relevant messages.

Time for processing messages is almost the same with previous method but the redundancy checking time is much greater (24 times). However, 4 second to process 286 messages is acceptable.

4.6.4 Constraint edit distance method

Since the edit distance method is highly sensitive and impractical, we only do experiment on edit distance method with constraint. Experiment results for constraint edit distance method is showed in Table 4-11 and Table 4-12.

Measure	Value
General precision	0.966
General recall	0.787
Detailed precision	0.374
Detailed recall	0.830

Table 4-11: Precisions and recalls of constraint edit distance method

Process	Time (ms)
Messages processing	375
Redundancy checking	529547
Total	529922

Table 4-12: Time performance of constraint edit distance method

For general measures, this method has a high performance with precision is 0.966 and recall is 0.787. This means that most of the redundant messages is detected with small incorrectly detecting rate. However, the detailed precision is quite low. This means that there are too many phrases (most are short phrases) are marked as repeated in non-relevant messages.

The time needed to perform redundancy checking is about 8 minutes. This means that the calculation for edit distance matrix and trace back algorithm is ineffective in time performance.

4.6.5 Combine method

Experiment results for constraint edit distance method is showed in Table 4-13 and Table 4-14.

Measure	Value
General precision	0.989
General recall	0.843
Detailed precision	0.980
Detailed recall	0.878

Table 4-13: Precisions and recalls of combine method

Process	Time (ms)
Messages processing	422
Redundancy checking	6813
Total	7235

Table 4-14: Time performance of combine

The combine method combines all the previous method and further put in some constraints to improve the precision. There is only one incorrectly detecting case (indicating that the last phrase is signature). Most of the redundancies that this method cannot detect are those emails with complicated signatures.

Since most of the phrase redundancy checking is substring test (most of the normal phrases and short phrases if the method continue with the second round), the processing time is much better than using the edit distance method only. The system needs 7 second to process 286 messages which is acceptable.

Chapter 5 Conclusion

5.1 Summary

In this project, we discuss, implement and experiment various methods to solve the redundant messaging problem. The final method combines all the previous methods with constraints in order to yield the best result in both time and accuracy. This method is practical since the accuracy is high and the processing time is acceptable. This method also partial solves all the challenges stated in section 1.3. It allows the redundant messages to be divided and quoted in other messages. Moreover, unimportant portion of the redundant message can be removed and some of the remaining parts can be quoted with modifications.

5.2 Limitations

Although all the challenges in section 1.3 are solved with the combine method, there are still some limitations. For some of the deletion modification on the phrases, the constraint edit distance method cannot recognize. The process of identifying salutation, conclusion and signature of the redundant message cannot apply for many complicated cases. In some special cases, this process even incorrectly indicates normal phrases as these portions. These cases deduce the precision of the system.

5.3 Future work

As mentioned in previous section, we need to further improve accuracy of unimportant portions detecting process. A better process will improve the precision and recall measure of the combine method. The unimportant portions can also include quoted message header. This part is often auto-generated in the reply message by the

email system. If we can eliminate such parts, it will reduce the time cost to detect redundancy and improve the overall accuracy.

Although the redundant messages can only correctly indicate while we examined the body content of email messages, other parts of the email may also provide useful information that can help in redundancy detecting. These parts include message header fields (subject, sending time, sender address, reply-to field) and attachments. We can also further improve the system so that it could handle other types of content rather than plain-text (html, for example) and in different languages.