Project 2 Report

1. Optimization Problem and Description

The optimization problem in this project is the "Shortest Superstring" problem. Given a finite set of strings, the target of this function is to find the shortest common superstring according to this set of strings. It is the shortest possible string such that every element in the string set appears as the substring (a consecutive block) of the superstring. The shortest superstring can be used in data compression and DNA sequencing.

(1) The input of this problem is an array of strings $S = \{s_1, s_2, \dots, s_n\}$. Without loss of generality, we assume that no string s_i inside array is a substring of another s_j, $where\ i \neq j$.

(2) The target of this problem is to find the shortest string (measured in string length) that contains each element in the array as its substring. Given the string array input $S = \{s_1, s_2, \dots, s_n\}$, the objective function of this problem is:

$$\min|S^*|$$
$$\text{subject to } \forall s_i \ \in S^*$$

(3) The expected output of this problem is a single string which satisfy the definition of shortest superstring. One example of input and output of this problem is:
Input: ["TACGA","ACCC","CTAAG","GACA"]
Output: "TACGACACCCTAAAG"

(4) This problem is an NP hard problem. In order to find the global optimal, we need to check every possible order of combination of input array of strings. Given sufficiently large problem size, the global optimal solution may be almost impossible to calculate due to the factorial time complexity nature of this problem.

2. Algorithm Implementation and Description

(1) Description of the first greedy algorithm
In the first algorithm, we start with the longest string inside the input array as the final superstring S^*. During each iteration step, we find the string s_i from the remaining input array that has the max overlap with the final superstring.

The local optimal is based on always adding string with largest overlap with current superstring. Two strings are overlapping if prefix of one string is same as suffix of other string or vice versa. There are two possible ways to define the largest overlap:

a. The length of the matching prefix and suffix is maximum.

$$overlap(S^*, a) = overlap\_length(S^*, a)(tail\ of\ a\ and\ head\ of\ b)$$

b. The Increment length of combining the two strings, the formula is represented as follow:

$$overlap(a, b) = len(b) - overlap\_length(a, b)(tail\ of\ a\ and\ head\ of\ b)$$

In this project, we choose the second definition of largest overlap for the two greedy algorithms because it is more closing to the global optimal for large-size input array.

After we find the string of max overlap with the final superstring, we merge this pair of string (i.e. s_i and S^*) into a new string and replace the final superstring S^* with the new one. Then, we remove the s_i from the string array. When we merge all string inside the string array with superstring S^*, we find the result.

(2). Pseudo-code of the first greedy algorithm

```
====================================================================
Greedy Shortest Superstring I (S)
==============================================================
1.  input: An array of strings S.
2.  output: A short superstring $S^*$ of input array S
3.  let T ← S
4.  let n be the length of T
5.  Initialize $S^*$ be the longest string inside T
6.  while n > 1 do
7.              If overlap (a, $S^*$) > 0
8.                  a ← where overlap (T[i], $S^*$) is the maximum
9.              else
10.                 a ← T[0]
11.         replace $S^*$ with a new string obtained by overlapping a and $S^*$
12.         update n
13. end while
14. return $S^*$
====================================================================
```

(3). Description of the second greedy algorithm

In the second greedy algorithm, the local optimal is achieved by always pick the pair of strings with the largest overlap. Given the input array of string S[], the steps are shown as follow:

a. Create an auxiliary array of strings T[] and copy all elements of S[] to T[].

b. While T[] contains more than one strings, find the most overlapping string pair (a, b) in T[].Then replace the string pair (a, b) with the string obtained after combining them.

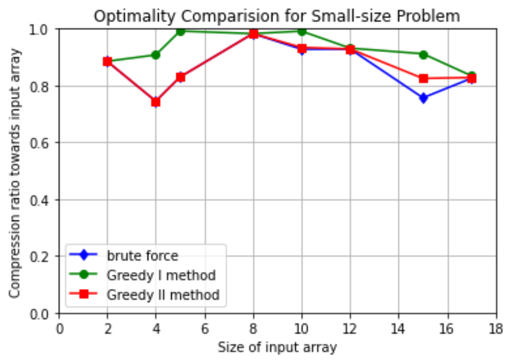
c. When there is only one string left in T[], this T[0] is the shortest superstring result.

(4). Pseudo-code of the second greedy algorithm

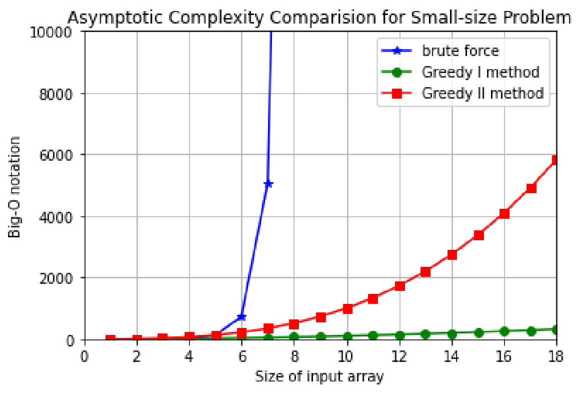
```
==============================================================
Greedy Shortest Superstring II (S)
==============================================================
1.  input: An array of strings S.
2.  output: A short superstring $S^*$ of input array S
3.  let T ← S
4.  let n be the length of T
5.  while n > 1 do
6.          for i = 1 to n
7.              for j = i+1 to n
8.                  if overlap (a, $S^*$) > 0
9.                      if overlap (T[i], T[j]) > overlap (a, b)
10.                         a ← T[i],   b← T[j]
11.                 else
12.                     a ← T[0],   b← T[m]
13.         replace a and b with a new string obtained by overlapping a and b in T
14.         update n
15.  end while
16.  return T[0]
====================================================================
```

3. Comparative Performance Analysis

(1). Comparative performance analysis result for small-size problem



(2) Asymptotic complexity analysis result for small-size problem

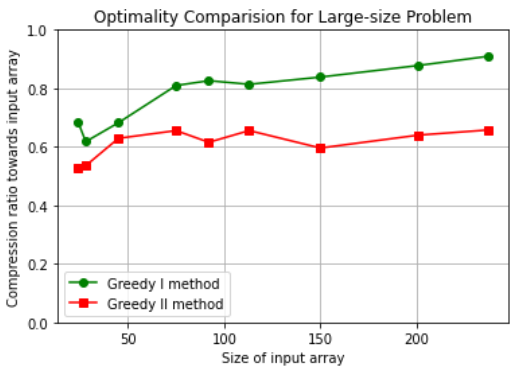


(3) Explanation for the observed result in small-size problem

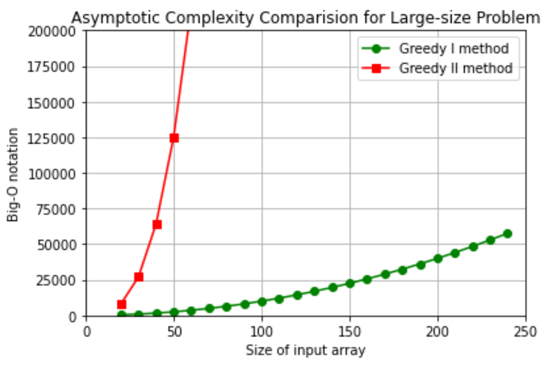
In the small-size problem performance analysis, the compression ratio (i.e. the measure of optimality) is the length of shortest superstring compare to the length of all string concatenation without overlap. From the result graph above, we can see that the second greedy algorithm achieve almost the same performance with brute-force method. As for the first greedy algorithm, the compression ratio is little higher than the rest two methods which means that the string concatenation in Greedy I is not always global optimal.

As for the time asymptotic complexity result, the brute-force is O(n!) which is significantly larger than the Greedy I algorithm (i.e. $O(n^2)$) and Greedy II algorithm ((i.e. $O(n^3)$)). Greedy II algorithm achieve better optimality in the sacrifice of asymptotic complexity.

(4) Comparative performance analysis result for large-size problem



(5) Asymptotic complexity analysis result for large-size problem



(6) Explanation for the observed result in large-size problem

Due to the high complexity of brute force algorithm, the memory required to compute the result given input array size of 20 or more beyond the capacity of my laptop. However, for the two Greedy methods, they can solve relatively large-size problem (i.e. greater than 20 and less than 500) without causing out of memory exception.

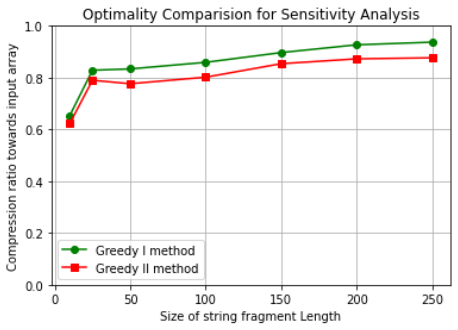
For optimality, the performance of the second greedy algorithm is better than the first greedy algorithm. This is because the first greedy method fixed one string when calculating the pair-wise overlap value. This is more likely to result in local optimal in the early steps.

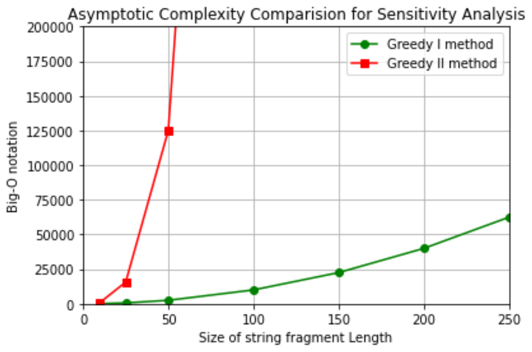
As for the time asymptotic complexity result, the Greedy II algorithm is almost 100 thousand times more than Greedy I method when the input size is 50. For larger problem size, the complexity is far more than Greedy II method. Greedy II method is almost unable to solve with problem size over 500 under the constrain of my laptop (4GB memory).

5. Sensitivity Analysis

In this project, we focus on two sensitive case study. The length of string fragment and the type of possible character set for each string fragment.

(1). Sensitivity Case 1: The length of input string fragment



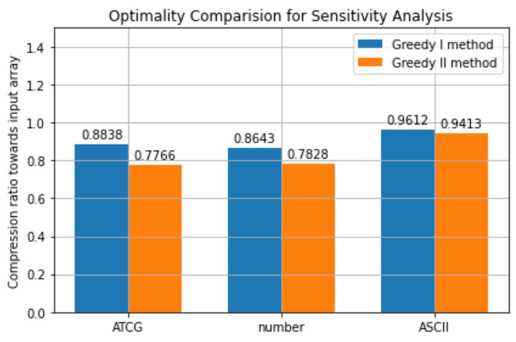


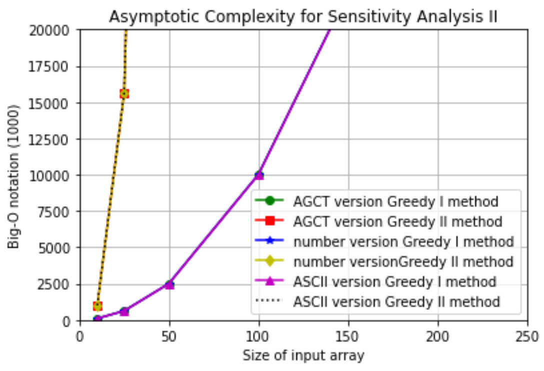
In this simulation, the problem size is 200 with variation on the length of each fragment string inside the input array. From the above two result, we can see that the compression ratio keeps increasing as the increase of string fragment size. This is because the overlap occurs only on the head and tail of the strings. Large fragment size will reduce the overall string compression efficiency. The relationship between two greedy algorithms does not change. Greedy II perform better than Greedy I. Thus, the length of string fragment does not have significantly influence on the optimality. This is because we are focus on the prefix and suffix of fragments and the possibility of prefix of string A match the suffix string B. The increment on fragment length will increase the compression ratio of both Greedy algorithms.

As for the asymptotic complexity, the relationship between two Greedy algorithms does not change. Greedy II is more complex than Greedy I. This may attribute to the fact that the fragment length only has almost no influence on the overlap computing part. Most of string pairs only match for several characters. The increment in the middle part of the fragment contribute almost nothing to the asymptotic complexity.

(2). Sensitivity Case 2: The size of possible character set for each string

In this case, we test on three different possible character set of the input string fragments. The length of input array is 50 with variation on character set type. {A, T, C, G} only, numbers, and ASCII characters.





According to the simulation result, the optimality for the Greedy I and Greedy II method reduce slightly as the increase of the size of possible character set. The ASCII character set and number character set will reduce the compression ability of the algorithm because it lower down the probability of matching between prefix and suffix. Thus, both two Greedy approach is kind of sensitive to the type of character set for the string fragments. As for asymptotic, the complexity for the three character type is the same.

This is because the matching operation only relate to whether the prefix and suffix of two string is same or not regardless of what exactly the two characters are. Thus, the asymptotic complexity does not change while there do exist some change on optimality.