Solving the Shortest Common Superstring Problem

<u>CS674, Spring 1999</u>

Sarita Challagulla

Geoff Rothman

Shelia Sittinger

Chris Wells

Introduction

Given a list of strings, the *shortest common superstring* problem is to find the shortest possible string in which all of the given strings can be found. Stated more formally, Given a set of strings $\boldsymbol{S} = \{s_1, \ldots s_k\}$, and an integer d, find a string L s.t. s_i is in L, 1<=i<=k, and |L| <=d. The matching optimization problem is to find a string L s.t. |L| is as small as possible. For example, given the set

green, red, enumerated, rat, bothered, there, both

What is the shortest string which contains all of these strings?
(greenumeratedbothered) <-- Our Shortest Common Superstring
(green rat both red)
(enumerated there)
(bothered)

This problem is a formal statement of two problems in computer science today – file compression and DNA sequencing[Fri 96] [Arm95].

The problem was shown to be NP-complete by Maier and Storer in 1977, by reduction from the vertex cover problem[Mai77].

Integer Programming

Integer Programming is a powerful tool in the mathematical modeling and solution of various optimization and decision problems. The Shortest Common Superstring problem can be transformed into an integer programming problem with the help of the following steps.

Steps For Transformation

1. Eliminate proper substrings from the input.
2. Build the follow table.
3. Transform the follow table into an optimization equation.
4. Build the simplex tableau.

The follow table gives information about whether a particular string can be followed by another string or not and also gives the number of characters which are saved when string s_i is followed by string s_j. The entry t_{ij}, in the ith row and jth column of the follow table is the number of characters saved when string s_i is followed by string s_j, plus 1. The steps to build a follow table and to transform the follow table into the optimization equation are discussed with the help of the following example.

Consider the with the input :
a = “ringer”
b = “string”
c = “ernest”
d = “ing”

The first step is to eliminate any substrings. So string d is eliminated and the remaining strings a, b, c are considered in building the follow table. Two other strings are also used for the transformation, the start string which is represented by “%” and the termination string which is represented by “$”. The string which follows the start string is the first string in the sequence of strings which comprise the superstring and the string which is followed by the termination string is the last string. We build the follow table as given below.

	%	a	b	c	$
%	0	1	1	1	0
a	0	0	1	3	1
b	0	5	0	1	1
c	0	1	3	0	1
$	0	0	0	0	0

A string s_i cannot follow itself as that would be a repetition and the entry t_{ii} in the follow table is a 0 denoting that such an operation is not allowed. If $t_{ij} > 0$, it means that string s_i can be followed by string s_j. Consider the entry t_{ac}, if a is followed by c i.e. “ringer” by “ernest”, the overlapping characters between the two strings are ‘e’ and ‘r’ . The number of overlapping characters is 2 and it is incremented by 1 to obtain the entry t_{ac}. This is done in order to represent that if there are no overlapping characters between two strings, the entry would be 1 denoting that one of the strings can follow the other but the number of characters saved by concatenating the two strings is 0. Consider the entry t_{bc}, if b is followed by c i.e. “string” by “ernest”, no characters will be saved but such a concatenation is allowed and the entry, t_{bc} is equal to 1. The start string cannot follow any other string. Hence, the column corresponding to the start string is filled with zeroes. Similarly, the termination string cannot be followed by any other string and the row corresponding to it is filled with zeroes. The other entries in the table can be filled by checking for the number of overlapping characters and incrementing it by 1.

Variables:

For each pair of strings $< s_i, s_j>$, we introduce the variable x_{ij}. If the variable is set to 1 then the string s_i is followed by the string s_j in the superstring and if it is set to 0, there is no such sequence in the superstring. For example, if $x_{ac} =1$ then a is followed by c in the resulting superstring.

Constraints and Optimization Equation:

The constraints are that each string in the superstring can be followed by a particular string only once and each string can follow another string only once. This can be expressed as

$\forall i,\ \sum x_{ij} = 1$ [Each string can be followed by only one string]
$i<>j,\ 1<=i,j<=n$

$\forall j,\ \sum x_{ij} = 1$ [Each string can follow only one string]
$i<>j,\ 1<=i,j<=n$

The optimization problem is to minimize the length of the superstring which can be done by maximizing the total number of overlapping characters. Each variable x_{ij} takes a value of either 0 or 1 to ensure that each string is followed by and follows another string exactly once. In linear integer programming form this becomes:

Maximize $\sum t_{ij} * x_{ij}$ subject to the constraints

$\forall i,\ \sum x_{ij} = 1$ [each string can be followed by only one string]

$\forall j,\ \sum x_{ij} =1$ [each string follows only one string]

Hence, our optimization equation would be

Maximize $x_{\%a} + x_{\%ab} + x_{\%c} + x_{ab} + 3x_{ac} + x_{a\$} + 5x_{ba} + x_{bc} + x_{b\$} + x_{ca} + 3\,x_{cb} + x_{c\$}$

Subject to the following constraints.

$x_{\%a} + x_{\%b} + x_{\%c} = 1$ [% can be followed by only a, b, or c]

$x_{ab} + x_{ab} + x_{a\$} = 1$ [a can be followed by only b, c or $]

$x_{ba} + x_{bc} + x_{b\$} = 1$ [b can be followed by only a, c or $]

$x_{ca} + x_{cb} + x_{c\$} = 1$ [c can be followed by only a, b or $]

$x_{\%a} + x_{ba} + x_{ca} = 1$ [a can follow only %, b or c]

$x_{\%b} + x_{ab} + x_{cb} = 1$ [b can follow only %, a or c]

$x_{\%c} + x_{ac} + x_{bc} = 1$ [c can follow only %, a or b]

$x_{a\$} + x_{b\$} + x_{c\$} = 1$ [$ can follow only a , b or c]

To solve the above equations, we introduce artificial variables y1, y2 , …, y8 and the new equations are

$x_{\%a} + x_{\%b} + x_{\%c} + y_1 = 1$ [% can be followed by only a, b, or c]

$x_{ab} + x_{ab} + x_{a\$} + y_2 = 1$ [a can be followed by only b, c or $]

$x_{ba} + x_{bc} + x_{b\$} + y_3 = 1$ [b can be followed by only a, c or $]

$x_{ca} + x_{cb} + x_{c\$} + y_4 = 1$ [c can be followed by only a, b or $]

$x_{\%a} + x_{ba} + x_{ca} + y_5 = 1$ [a can follow only %, b or c]

$x_{\%b} + x_{ab} + x_{cb} + y_6 = 1$ [b can follow only %, a or c]

$x_{\%c} + x_{ac} + x_{bc} + y_7 = 1$ [c can follow only %, a or b]

$x_{a\$} + x_{b\$} + x_{c\$} + y_8 = 1$ [$ can follow only a , b or c]

The completed Simplex table follows on the next page.

The Simplex table is

$X_{\% a}$	$X_{\% b}$	$X_{\% c}$	$X_{\% \$}$	X_{aa}	X_{bb}	X_{ac}	$X_{a\$}$	X_{ba}	X_{bb}	X_{bc}	$X_{b\$}$	X_{ca}	X_{cb}	X_{cc}	$X_{c\$}$	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	b_j
1	1	1	0	0	1	3	1	5	0	1	1	1	3	0	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1
0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	1

The size of the Simplex table grows exponentially with the size of the input in integer programming.

Enumeration Methods

Branch and Bound: Checking Strings

Enumerating Possible Solutions

We present an algorithm that will enumerate all the possible Shortest Common Superstring solutions for the feasibility checker to analyze. We build a tree that contains all the possible permutations of the strings in the input. To improve efficiency our goal is to bound as many branches of the tree as possible. The following EasyBound algorithm is passed these variables:

neweststring - the string to be added to the Partiallist
last- the string before the newest in the superstring
Partiallist- an array of indices of the strings in the present superstring
best- the maximum savings seen so far
Completelist- the best permutation found so far
anticipated- the sum of maximum savings for each string not in the partial string
currentsavings- the actual savings from the strings in the partial string
Follow[i][j]- a 2D array with savings for the i'th string that is followed by the j'th string.

```
EasyBound( neweststring, laststring, Partiallist, Completelist, best,
currentsavings, anticipatedsavings)
{
  Add neweststring to end of Partiallist
  laststring := neweststring

  if (Partiallist is full
      if (currentsavings is better than previous best)
          copy Partiallist into Completelist
          best = currentsavings;

       return best, Completelist;

   else
   for Every string
       stringid = stringnumber
     if (this string is not in Partiallist)
         if ( best <= the best we can save with this string following
                         the last string)

         EasyBound(stringid, last, Partial, best,
                 Beststring, anticipate-Follow[stringid][stringid],
                 currentsavings + Follow[last][stringid], Follow);
         Partial[stringid] = -1;  // reset this string
```

Problem Instance

For our introductory problem instance, we will use an array "Array [1..k]" of strings where k is the number of strings in the user input. We will also need to generate a follow table (as in the integer programming solution) to determine savings from the strings.

Solution Candidates

Our solution candidate is a list of strings in the order in which they appear in the superstring. In my introductory example, I will use these three strings: the, theatre & red. The best solutions turn out to be [theatred] or [redtheatre].

Objective Function

Our objective function is the sum of the savings from the strings in the list and the maximum savings possible from the other strings. This problem can be solved in constant time, with the follow table. We begin the tree with a maximum of the letters saved for each string (except the first). Each time we add a string, we recalculate the currentsavings and the anticipated savings.

```
Newsavings(currentsavings, anticipated, newstring, last, Follow) {
   currentsavings := currentsavings +Follow[last][newstring]
   anticipated := anticipated - maxsavings[newstring]
}
```

The complexity of this function is constant. The correctness follows from the fact that when we start, anticipated contains maxsavings from all strings except the first. At the end (when a full string is found) maxsavings is zero, since the each strings maxsavings has been subtracted. At the beginning, currentsavings is zero. At the end, the correct amount has been added for each new string.

Feasibility Solution Checker

We have developed the following algorithm to check the feasibility of a solution passed in to us. It was derived from a solution by Knuth, Morris, Pratt [Cor97].

```
CheckSolution(T,L,k)
// preconditions: T is the (presumed) superstring, L is the list of
strings to be incorporated, k the number of strings
// postcondition: returns true if all strings in the list are in the
superstring, false otherwise.
for i:=1 to k
  okay=KMP-Matcher(T, L[i])
  if(okay = false) return false
end for
return true
```

KMP-Matcher

```
KMP-Matcher(T, P)
// preconditions: T is the superstring, P is the substring to find in T
// post: returns true if P is a substring of T, false otherwise

n:= length[T]
m=length[P]
ComputePrefixFunction(P, prefix) [O(m) complexity]
q:=0
for i=1 to n  [O(n) complexity]
    while q>0 and P[q+1] <>T[i]
        q:=prefix[q]  // q is index of substring and will be <=
superstring
    if P[q+1] = T[i]
        q:=q+1
    if q=m
        return true
return false
```

ComputePrefixFunction

```
ComputePrefixFunction(P, prefix)
//preconditions: P contains a string
//post: prefix[q] is the length of the longest prefix of P that is a
proper suffix of P[1..q]
m:=length[P]
prefix[1]:=0
k:=0
for q:=2 to m
        while k>0 and P[k+1] <> P[q]
                k:=prefix[k]
        if P[k+1] = P[q]
                k:=k+1
        prefix[q] = k
end for
return
```

ComputePrefixFunction saves time by finding out how far you need to shift the index in the superstring in order to get a proper match w/ the substring. It eliminates the need to backtrack when traversing the superstring.
Example:

```
Superstring: stringeringing
Substring:     ringing (1st Try)
    (e vs r)       ringing (2nd Try)
    (match found)   ringing (3rd Try)
```

The complexity of this function is O(m). The for loop executes m-1 times, k is always less than q, and since prefix[k] is always less than k, the while loop decreases k (but only to 0). So the inner loop amortized cost is O(1), and the outer is O(m). We could compute the prefix function once for each string we wanted to match, thus making this part of the initial overhead rather than the feasible solution finder.

Complexity of Feasibility Checker

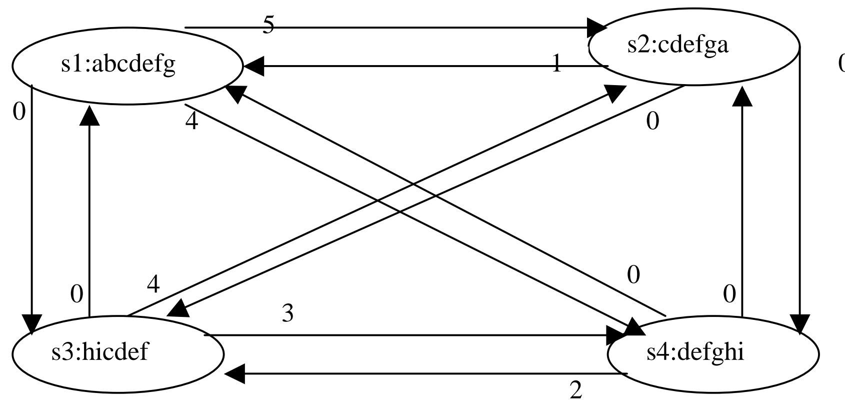
KMP-Matcher is O(m+n). The while loop executes in constant time (overall it can't run any more than "n" times), and the outer loop executes n times; m is the length of the substring.

CheckSolution is O((m+n)*k). The loop calls KMP-Matcher k times where k is the number of substrings. This is blatantly obvious that we don't want to call the Feasibility Checker very often!

Branch and Bound: Greedy Method

Using the previous branch and bound method, no attempt was made to find better superstrings first. In many cases, this kind of optimization speeds the solution of the problem. We have therefore implemented another branch and bound algorithm, which uses a greedy method to decide which string to add next.

In order to search the set of possible solutions, we must decide on the format of a solution. Consider each string as a node in a fully connected, directed, edge-weighted graph. The weight of each edge is the number of letters we save when appending the source string to the sink string. For example:



To find a minimum superstring, we must find a maximum spanning tree on these edges, with the constraint that each vertex has at most one incoming edge and one outgoing edge. Therefore, a solution to the problem is a list of k-1 of these edges, such that two strings are included once each (for the beginning and the end) and all other strings are included twice.

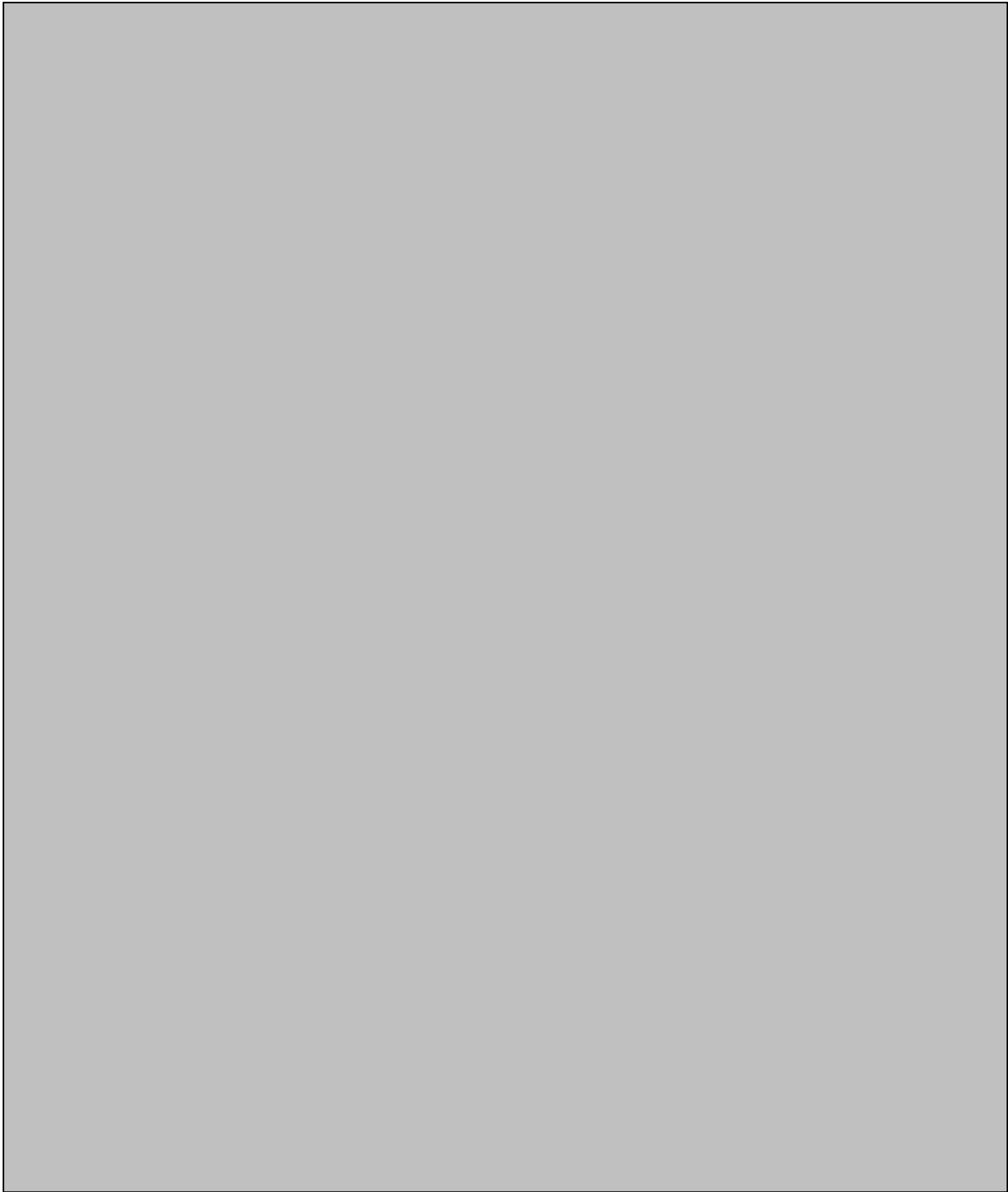
Objective Function
To evaluate a solution of this form, we add the letters saved on each edge together, subtracting the total from the number of letters in the initial strings. In the example above, for instance, the solution [s1,s2], [s2,s3], [s3,s4] would be evaluated as:
25 - 5-0-3, or 17. The actual string would be abcdefgahicdefghi, which has 17 letters.

```
Size(Edge, list)
    for each Edge in list
        subtract savings from total
return total
```

The complexity of the algorithm is linear. The correctness follows from the simplicity.

Feasibility
The feasibility of a solution can be checked by determining a) whether any string is represented as following or preceding more than one other string, and b)whether there is a cycle in the solution string. To determine this, we will keep a record of the position of each string in the selected set, and a record of which substring of the current superstring each string is in. For example, if we have added the edge from s1 to s4, and the edge from s3 to s2 to our set, s1 and s3 are preceding strings, s2 and s4 are following strings, and s1 and s4 are in the substring of s1, while s2 and s3 are in the substring of s2. Thus we know that edges starting with either s1 or s3 are infeasible, as are edges ending with s2 and s4, or edges containing both s1 and s3 or edges containing s2 and s4.

Correctness
Correctness of the feasibility algorithm: There are 4 possible values for InOut for each of the two strings of an edge, therefore 16 cases. Lines 1 through 3 correctly label infeasible 12 of these cases - those in which one or both of the strings are in an incorrect position. Lines 4 through 8 deal with the case of adding a new edge to the end of a substring. Lines 9 through 16 deal with the case of concatenating two substrings together with a new edge, including the update of the Substr table. Lines 17 through 22 deal with adding an entirely new edge to the set, and lines 22 through 28 deal with adding a new edge to the beginning of a substring, including updating the Substr table to reflect the new head of the substring.
Complexity of the feasibility algorithm: The algorithm is O(k). The worst case would be the case that each edge is added to the "front" of the preceding substring.



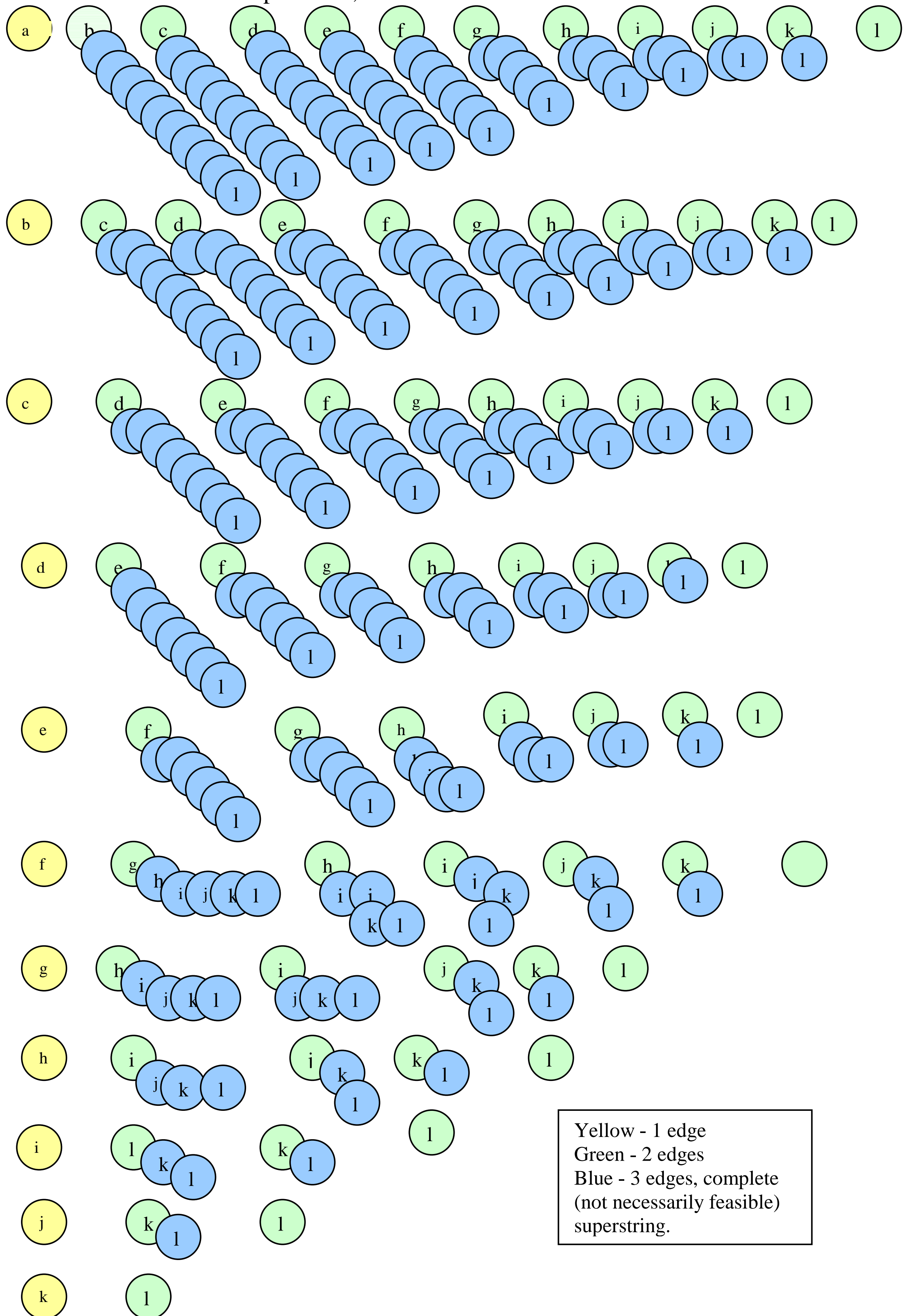
Enumeration of solutions

With k strings given, there are k^2 edges. We will sort the edges by weight, which, since the weights have a range [0, ..., k-1] can be done in $O(k^2)$ time. In the given example, we will have the following table:

EDGE	1ST STRING	2ND STRING	SAVINGS
a	s1: abcdefg	s2: cdefga	5
b	s1: abcdefg	s4: defghi	4
c	s3: hicdef	s2: cdefga	4
d	s3: hicdef	s4: defghi	3
e	s4: defghi	s3: hicdef	2
f	s2: cdefga	s1: abcdefg	1
g	s1: abcdefg	s3: hicdef	0
h	s2: cdefga	s3: hicdef	0
i	s2: cdefga	s4: defghi	0
j	s3: hicdef	s1: abcdefg	0
k	s4: defghi	s1: abcdefg	0
l	s4: defghi	s2: cdefga	0

With this method of representing the problem, the search space for a solution is the possible combinations of k-1 edges. There are k(k-1) edges, so the number involved is $\binom{k*(k-1)}{k-1}$ in this case. The complete search space is represented on the next page.

Search space tree, unbounded



The Branch and Bound Algorithm

To minimize the search space, we use a lower bounding algorithm, calculating the best superstring we can find on each branch of the search tree. We discover the best savings for each string (as the second string of an edge), while we are building the edge table. We will drop the minimum best savings, because only k-1 strings can be added second. Adding the k-1 best savings together gives us the initial possible savings (PS). Now the lower bound on a branch is equal to the maximum string length (max) minus the current savings(CS) minus the minimum of either the best savings on the remaining strings (PS), or the number of edges to be added times the weight of the current node. (Note that, if the value of the current edge is 1, there are no edges remaining which will save more than 1 letter each.) Upon reaching a node, we will add back the best savings for the second string of that edge to PS and subtract the actual savings of the edge from CS. This is the algorithm:

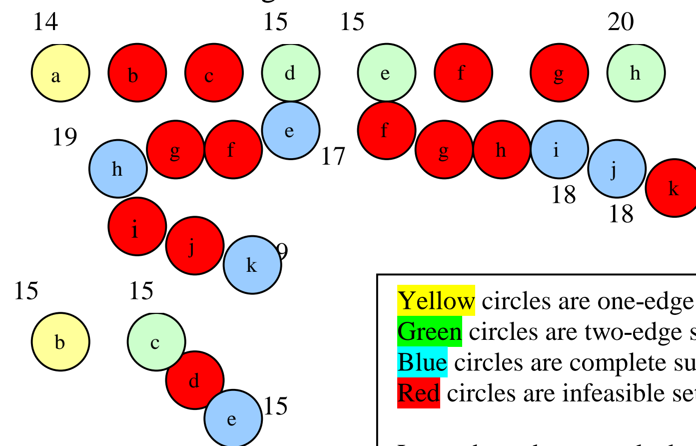
```
EvaluateNode( weight, BS, CS, PS, count)
pre: weight is the weight of the edge to be added.  BS is the best savings from the second
string in the edge. CS is the number of letters we have currently saved, PS is the possible
savings on future edges and count is the number of edges to be added to the set of
substrings.
post: PS is the minimum of PS-BS and more*weight, weight has been added to CS.
Therefore, total possible savings is PS + CS.

PS = PS-BS
if (more*weight) < PS
        then PS = more*weight
CS = CS + weight
```

Correctness of the bounding algorithm: We assume we can save the maximum from every string but one. When we actually add a string, we change the current savings by the amount actually saved, and the possible savings by the amount we hoped to save. These will be equal but opposite changes when the maximum is saved, or less if the maximum of that string is not saved. Checking the more*weight allows us to bound off the tree at the point at which all savings have been checked.

Complexity of the bounding algorithm: the algorithm is O(1).

A Bounded Tree: Infeasible edges are red.

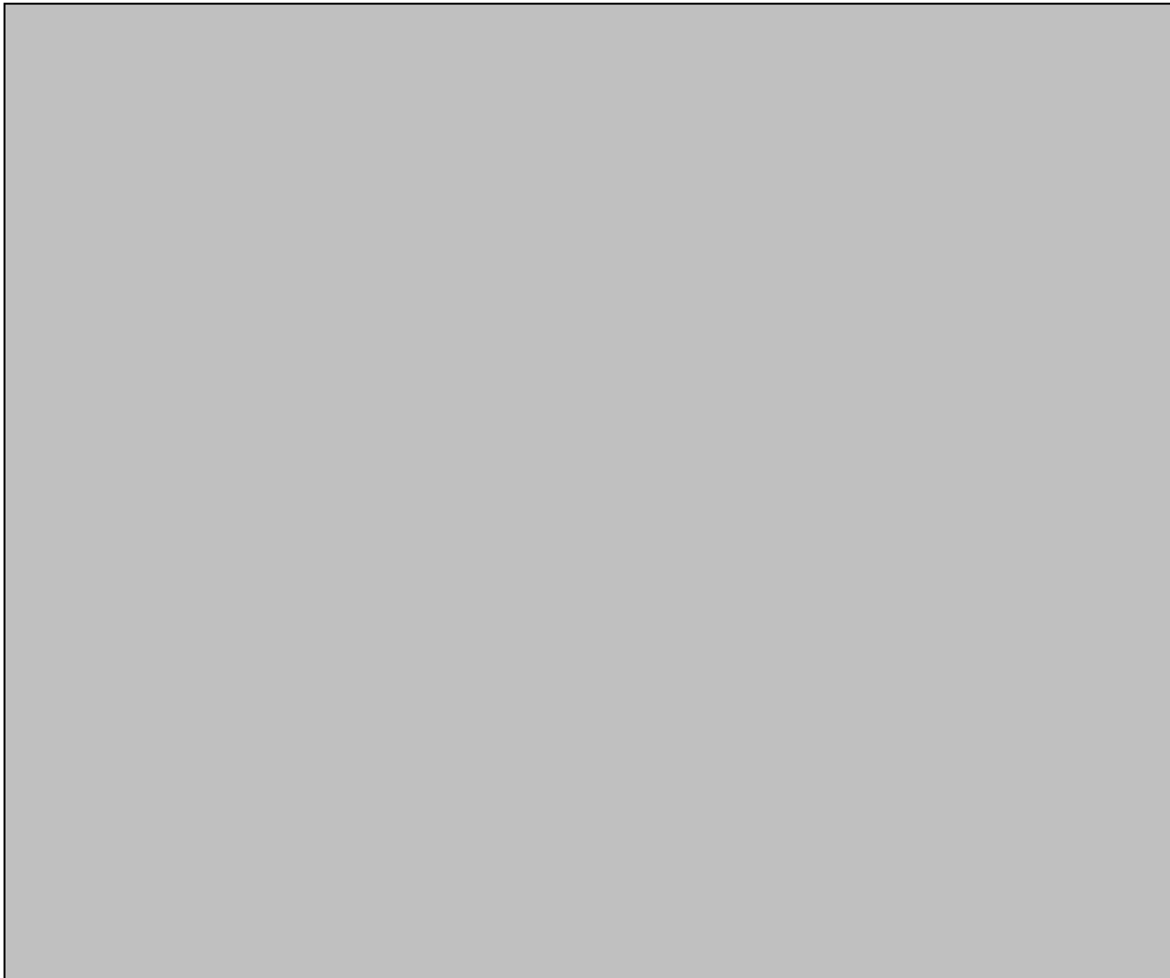


Yellow circles are one-edge sets.
Green circles are two-edge sets.
Blue circles are complete superstrings
Red circles are infeasible sets.

Lower bounds are marked outside of yellow, and green circles. Superstring lengths are marked outside blue circles.

This tree is much smaller than the last, since once we have found a superstring of the same size as the lower bound of its branch, we need not look at any more edges.

The Branching Algorithm



Correctness of Branching Algorithm:
At each edge, we evaluate the lower bound of that branch. If the bound is lower than our current best, we check for feasibility. If an edge is feasible, and more edges are needed for a complete set, we call the algorithm again to find another edge to add to the set. All branches with lower bounds than the current best are explored.

Complexity of the Branching Algorithm:
There are up to $\binom{k*(k-1}{k-1}$ nodes on the tree, so the algorithm is $O(k^2!)$.

The Greedy Method:

In many NP-complete problems, sometimes we are interested in a quickly obtainable *good* solution, rather than a slow-to-compute *optimal* solution. In these instances, we use approximation heuristics. One such method is the Greedy Method.

We apply the Greedy Method in this section to our shortest common superstring problem. First, using Knuth-Morris-Pratt ($O(k^2)$), we remove all strings in the input that are substrings of other strings in the input. Next, we construct the Edge Table. An edge is an indicator that one string is following another in the superstring. For instance, suppose, in the input, that string 1 was string, and that string 2 was ringer. Edge s1-s2 would mean that stringer appeared somewhere in the superstring. Similarly, edge s2-s1 would mean that ringerstring appeared somewhere in the superstring. Note that an edge must also store the number of characters saved: in the case of stringer, the **weight** of the edge would be 4. The Edge Table is a 3-tuple table with k^2 entries (one for each pair of strings). The first part of the tuple is the weight of the edge. The second and third parts of the tuple are, respectively, the string that precedes in the edge, and the string that follows in the edge. As an example, for the input

honest
nester
estnes
ring

the Edge Table would be

Wt	4	3	0	0	0	1	0	3	0	0	0	0
1st	1	1	1	2	2	2	3	3	3	4	4	4
2nd	2	3	4	1	3	4	1	2	4	1	2	3

Next, we sort the Edge Table by weight in decreasing order:

Wt	4	3	3	1	0	0	0	0	0	0	0	0
1st	1	1	3	2	1	2	2	3	3	4	4	4
2nd	2	3	2	4	4	1	3	1	4	1	2	3

Before we begin to apply the Greedy Method, we need to store certain additional information. Specifically, we need to be able to determine whether a given string can be followed or preceded (since a string, once preceded by another string, cannot be preceded again...likewise when a string is followed). To this end we have the Current Savings Table, which has one entry per string. Each entry contains the following information: In/Out, which determines whether the string can be preceded, followed, both, or neither; First String, the first string of the input in the current string's portion of the final superstring; and Next String, which points to the next string in the current solution.

In/Out
0 = Not in   1 = First in   2 = Last in   3 = Surrounded

In/Out	String	First String	Next String
0	honest	1	0
0	nester	2	0
0	estnes	3	0
0	ring	4	0

The Greedy Method, then, traverses the Sorted Edge Table. For each edge, the Greedy Method determines if the edge is feasible, and if it is, adds it to the set of accepted edges. It then updates the Current Savings Table, and if the number of current edges is equal to k-1, terminates.

Feasibility
Feasibility for an edge depends on the two strings in the edge. If the Current Savings Table indicates that the first string cannot be followed, or that the second string cannot be preceded, then the edge is infeasible. Additionally, if the two strings in the edge have the same first string, then they are already connected in an indirect manner. Adding the edge in this case would produce a cycle in the superstring, so this case is also infeasible.

When an edge is found to be feasible, the Greedy Method's current state must be updated. First, the second string in the edge—and every string currently following it, directly or indirectly—has its First String field in the Current Savings Table updated to point to the string pointed to by the First String field of the first string of the edge. Then the Next String field of the first string of the edge is pointed to the second string of the edge. Finally, the In/Out field of each string is updated. If the first string of the edge is not following another string, its In/Out field is changed from 0 to 1, indicating it can still be the second string in an edge. Otherwise, its In/Out field is updated to 3, meaning that it can no longer be in any feasible edge. If the second string of the edge does not precede another string, it is marked as being able to be followed in the future. Otherwise, its In/Out field is changed to 3, indicating that it is surrounded, and can no longer be a part of any feasible edge.

So, in our example, the edge s1-s2 is added first, because it saves us 4 characters. Our superstring at this point is "honester". Next, the edge s1-s3 is checked for feasibility. Because string 1 is already followed by string 2, this edge is infeasible. The next feasible edge is s2-s4, with a weight of 1. Our superstring is now "honestering". The next—and final—feasible edge is s3-s1, with a weight of 0, giving us our approximate solution: "estneshonestering". Note that "honestnestering" is also a superstring, and is shorter than our Greedy Method solution.

The time complexity of this Greedy Method requires careful analysis. Checking each possible edge requires $O(k^2)$ iterations of a loop. Each iteration of this loop performs a constant number of steps, unless the edge is feasible. But only O(k) edges can ever be

feasible, so only O(k) iterations of the loop need be considered for this case. When an edge is found to be feasible, an update must be performed. In the worst case, every string's entry in the Current Savings Table must be updated, which is O(k). Therefore, for feasible edges, the loop is $O(k^2)$. The Sorted Edge Table must also be constructed and sorted. Sorting the table takes $O(k^2 \log k)$, and this is the longest running part of the algorithm. Therefore, the Greedy Method presented here for Shortest Common Superstring is $O(k^2 \log k)$.

Results

<u>First set of Data</u>

```
string 0:  honest
string 1:  nester
string 2:  estnes
string 3:  ring
```

Greedy Method results:

```
Save:  0, 0, 4, 1,
   Length:       17   Superstring:  estneshonestering
Order of strings:  2, 0, 1, 3,
Branch and Bound w/ Greedy results:
Save:  0, 0, 4, 1,
Save:  0, 3, 3, 1,
   Length:       15   Superstring:  honestnestering
Order of strings:  0, 2, 1, 3,
```

Standard Branch and Bound

```
the number of letters saved was: 7
The number of letters in the superstring is 15
The best string is:
honestnestering
```

<u>Second Set of Data</u>

```
string 0:  babbaba
string 1:  baaaa
string 2:  cabbaca
string 3:  cabc
string 4:  cbaac
string 5:  cccbcb
string 6:  acabbb
string 7:  accaacc
string 8:  baaab
string 9:  bcabb
```

Greedy Method results:

```
Save:  0, 0, 2, 4, 3, 1, 2, 1, 2, 2,
   Length:       40   Superstring:
baaabcabcabbacabbbabbabaaaaccaacccbcbaac
```

Branch and Bound w/ Greedy results:

```
Save:  0, 0, 2, 4, 3, 1, 2, 1, 2, 2,
Save:  0, 1, 2, 1, 2, 2, 1, 2, 4, 3,
   Length:       39   Superstring:
baaababbabaaaaccaacccbcbaacabcabbacabbb
```

Standard Branch and Bound

```
the number of letters saved was: 18
The number of letters in the superstring is 39
The best string is:
babbabaaaaccaacccbcbaacabcabbacabbbaaab
```

Third Set of Data

string 0: 1010110
string 1: 1011001
string 2: 1001101
string 3: 001010
string 4: 111101

Greedy Method results:

Save: 0, 4, 0, 1, 4,
   Length: 24   Superstring:
001010110111101011001101
Order of strings: 3, 0, 4, 1, 2,

Branch and Bound w/ Greedy results:

Save: 0, 4, 0, 1, 4,
Save: 0, 3, 1, 2, 4,
   Length: 23   Superstring: 11110101100101011001101
Order of strings: 4, 0, 3, 1, 2,

Standard Branch and Bound

the number of letters saved was: 11
The number of letters in the superstring is 22
The best string is:
0010101101100110111101

Fourth Set of Data

string 0: 111001011
string 1: 101000011
string 2: 0111000
string 3: 001111
string 4: 010000101
string 5: 010001
string 6: 100100001

Greedy Method results:

Save: 0, 3, 3, 1, 2, 3, 0,
   Length: 43   Superstring:
0011110010111000100010000101000011100100001
Order of strings: 3, 0, 2, 5, 4, 1, 6,

Branch and Bound w/ Greedy results:

Save: 0, 3, 3, 1, 2, 3, 0,
   Length: 43   Superstring:
0011110010111000100010000101000011100100001
Order of strings: 3, 0, 2, 5, 4, 1, 6,

Standard Branch and Bound

the number of letters saved was: 13
The number of letters in the superstring is 42
The best string is:
001111001011100010001000010100001100100001

Fifth Set of Data
string 0: feast
string 1: spoof
string 2: lisp
string 3: eastern
string 4: ernest
string 5: stop
string 6: topic
string 7: picture
string 8: regard
string 9: ardent
string 10: dentist
string 11: clasp
Greedy Method results:
Save: 0, 0, 2, 1, 4, 3, 2, 3, 3, 2, 3, 4,
   Length: 40   Superstring:
clasplispoofeasternestopicturegardentist
Order of strings: 11, 2, 1, 0, 3, 4, 5, 6, 7, 8, 9, 10,
Branch and Bound w/ Greedy results:
Save: 0, 0, 2, 1, 4, 3, 2, 3, 3, 2, 3, 4,
   Length: 40   Superstring:
clasplispoofeasternestopicturegardentist
Order of strings: 11, 2, 1, 0, 3, 4, 5, 6, 7, 8, 9, 10,
Standard Branch and Bound:
the number of letters saved was: 27
The number of letters in the superstring is 40
The best string is:
lispoofeasternestopicturegardentistclasp

In most cases, then, the Standard Branch and bound found the best string. Occasionally the Greedy algorithm found an equally good solution, but even when the solution was not exact, it was still within only a few letters of the optimal string.

Bibliography

[Arm95] Chris Armen and Clifford Stein, **A 2-2/3 Approximation for the Shortest Superstring Problem**, Technical report, Dartmouth College, Computer Science, Number PCS-TR95-262, June 1995.

[Cor89] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms,* The MIT Press, 1989.

[Fri 96] A. Frieze and W. Szpankowski, Greedy Algorithms for the Shortest Common Superstring that Are Asymptotically Optimal, *Lecture Notes in Computer Science*, Vol. 1136, 1996.

[Gar70] M. Garey and D. Johnson, *Computers and Intractability*, W.H. Freeman, San Francisco, 1970.

[Mai77] D. Maier and J. A. Storer, A note on the complexity of superstring problem, Technical Report, Computer Science Laboratory, Princeton University, Number 233, 1977.

Appendix
C++ implementations

Easy Branch and Bound

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#define MAXSTRLEN 50
#define MAXSTRINGS 25

void checkstrings(const char *string1, const char *string2,
            short int& save12,short int& save21);
// pre: second string is not longer than first string
//post:  save12 is overlap when second string follows first
//      save21 is overlap when first string follows second
void ComputePrefix(const char *string1,short int prefix[]);
//pre: string1 contains a string
//post: prefix[i] is the longest prefix of string1 that is a suffix of
string1[i]
short KMPMatcher(const char *string1, const char *string2);
//pre: second string is not longer than first string
//post: returns 1 if second string is a substring, 0 otherwise.

void EasyBound(short newest, short current, short last, short
Partial[], short Number,
            short &best, short Beststring[],
             short hopesave, short currsave, short
Follow[][MAXSTRINGS]);

//pre: newest is the string to be tried next in the superstring,
// current is the number of strings in the current partial string,
// last is the string before the newest in the superstring,
// Partial is indices of the strings in the present superstring,
// Number is the number of strings in a full superstring,
// best is the maximum savings seen so far,
// Beststring is the best permutation found so far,
// hopesave is the sum of the maximum savings for each string not in
the partial
// currsave is the actual savings from the strings in the partial
string
// Follow[i][j] is a 2-dimensional array with the savings
// for the ith string being followed by the jth string.
// post:

int main(){
   short best, maxsave;
   short Partial[MAXSTRINGS];
   short Beststring[MAXSTRINGS];
   short Follow[MAXSTRINGS][MAXSTRINGS];
   short i,j, large, small;
   char list[MAXSTRINGS][MAXSTRLEN];
   char filename[80];
   short Number = 0;
   short total;         // the total number of letters in all strings
   short int savefirst, savesecond;
```

```
   ifstream datafile;

   cout<<"What file has the list of strings?";
   cin>>filename;
   datafile.open(filename);
   if (datafile.fail()) exit(1);

   datafile>>Number;
   for (i=0; i< Number; i++)
      datafile>>list[i];

   // check each string as a substring
   i=0;
   while( i< Number - 1 ){
      j=i+1;
      while (j < Number){
         if (strlen(list[i]) <= strlen(list[j])){

            if( KMPMatcher(list[j], list[i])){
              // list[i] is a substring of list[j]
              //copy last string into ith position
              strncpy(list[i],list[Number-1], MAXSTRLEN);
              Number--; // we have one string fewer
              j=i; // check new ith element
            }  // end if substring found
         }else if (KMPMatcher(list[i], list[j]) ){
               // list[j] is a substring of list[i]
               // copy last string into jth position
               if (j<Number-1) {
                 strncpy(list[j], list[Number-1],MAXSTRLEN);
                 j--;     // recheck the jth string, it's new
               } // end if not the last string
               Number--;
         }  //end if substring found
         j++;
      }//end while j < Number
      i++;
   }//end while i< Number-1

// Now that substrings are eliminated, find maximum savings
// and maximum length
   total = strlen(list[Number-1]);
   for (i=0; i<Number-1; i++) {
        total += strlen(list[i]);
        for (j=i+1; j<Number; j++){
         if (strlen(list[i]) > strlen(list[j]) ){
             large = i; small = j;
         }else {
             large = j; small = i;
         }
         checkstrings(list[large],list[small],savefirst,savesecond);
         Follow[large][small] = savefirst;
         Follow[small][large] = savesecond;
        }  // end for j
   }// end for i

// Now put maximum savings for the string into Follow[i][i]
```

```
   for (i=0; i<Number; i++) {
       Follow[i][i] = 0;
       for (j=0; j<Number; j++)
        if ((j !=  i) && (Follow[j][i] > Follow[i][i]))
            Follow[i][i] = Follow[j][i];
   }
//print the Follow table
   cout<<"Follow Table:"<<endl;
   for (i = 0; i<Number; i++) {
      cout<< list[i]<<"\t";
      for (j=0; j<Number; j++) {
       cout <<Follow[i][j]<<" ";
      }
      cout <<endl;
   }

//Now do the branch and bound search
   best = -1;
  for (i=0; i< Number; i++) {
      maxsave = 0;

      for(j=0; j<Number; j++){
       Partial[j] = -1;
       if (i != j)
         maxsave += Follow[j][j];
       }

     EasyBound(i, 1, i, Partial, Number, best, Beststring,
             maxsave, 0, Follow);
   }

   cout <<"The total number of letters in the string was:
"<<total<<endl;
   cout << "the number of letters saved was: "<<best<<endl;
   cout <<"The number of letters in the superstring is "<<total-
best<<endl;
   cout << "The best string is: \n" ;
   cout <<list[Beststring[0]];
   for (i=1; i<Number; i++)  {
     for (j = Follow[Beststring[i-1]][Beststring[i]];
j<strlen(list[Beststring[i]]); j++){
       cout << list[Beststring[i]][j];
     }
   }
   cout << endl;

  return 0;
}

void checkstrings(const char *string1,const char *string2,
            short int& save12, short int& save21){

   short int first = strlen(string1);
   short int second = strlen(string2);
   short int i,j, count;
   save12 =0;
   save21=0;
```

```
   // find the overlap from first string into second string
   i=first - second;   //index into longer (first) string

   do {
       j=0;             //index into shorter (second) string
       while (( i<first)&&(string2[j] != string1[i])) {
       i++;
       }
       if (i<first){
        for (count = 0; (i<first)&&
string2[j]==string1[i];i++,j++,count++);

        if ( i == first ){
           save12 = count;
        }
        i++;
       }
    }while (i < first);  // until last letter in first string is
checked.

   // Find the overlap from second string into first string
   i=second-1;                //index to string1
   do {
      j=second-1;            //index to string2
      while ((i>=0)&&( string2[j] != string1[i])) // find last letter
of string2
       i--;                        // inside string1 (longest overlap)


      if (i>=0){    //check for match back to zero
       for (count = 0; i>=0 && string2[j]==string1[i];i--,j--,count++);

       if (i < 0){
          save21 = count; // zero if no match
       }
       i--; //check next letter
      }
   }while (i>=0);  // until first letter of first string is checked

    return;
}


void ComputePrefix(const char *string1,short int prefix[]) {

   short int position =-1;
   short int length = strlen(string1);

   prefix[0] = -1;
   for (short int index = 1; index <length; index++) {
      while ( (position >=0) && (string1[position+1] !=
string1[index])) {
       position = prefix[position];
      }

      if(string1[position+1] == string1[index])
```

```
          position++;

      prefix[index] = position;
   }
   return;
}

short KMPMatcher(const char *string1, const char *string2){

   short int prefix[MAXSTRLEN];
   short int first, second, position;

   first = strlen(string1);
   second = strlen(string2);
   ComputePrefix(string2, prefix);

   position = -1;
   for(int index=0; index< first; index++){

      while(position >=0 && (string2[position+1] != string1[index]))
        position = prefix[position];

      if(string2[position+1] == string1[index])
        position++;

      if (position == second -1)
        return 1;     // second string is a substring of first
   }
   return 0;
}

void EasyBound(short newest, short current, short last, short
Partial[],
             short Number, short &best, short Beststring[],
             short hopesave, short currsave, short Follow[][MAXSTRINGS]
)
{

  Partial[newest] = current;
  last = newest;               // this string is now last

  if (current == Number) {
     if (best < currsave) {
       for (int i = 0; i<Number; i++)
       {
            Beststring[Partial[i]-1] = i;
       } // end for i
       best = currsave;
    } // end if better string found
       return;
   } // end if partial string is full
   else{
   for (int stringid= 0; stringid< Number; stringid++){

     if (Partial[stringid]<0) { // this element is not in the current
superstring
      if ( best <= (currsave + hopesave -
```

```
            Follow[stringid][stringid] + Follow[last][stringid]) ){
        // the best we can save with string[i] following the last string

          EasyBound(stringid, (current+1), last, Partial, Number, best,
                  Beststring, hopesave-Follow[stringid][stringid],
                  currsave + Follow[last][stringid], Follow);
          Partial[stringid] = -1;  // reset this string
        } //end branching
      } // end if string not in list

   } // try another missing string
}} // end of EasyBound
```

The Greedy and Greedy Branching algorithms

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "strings.cc"

//True if x is a substring of y
#define IS_SUB_STRING(x, y) isSubString(x, y)

//Returns the number of characters saved if y follows x
#define SUFFIX(x, y, z, w) suffix(x, y, z, w)

bool isSubString(char *x, char *y)
{
   return (strlen(x) <= strlen(y) && KMPMatcher(x, y));
}

void suffix(char *x, char *y, short &save1, short &save2)
{
   checkstrings(x, y, save1, save2);
}

typedef struct _Edge
{
   size_t save;
   size_t s1;
   size_t s2;
} Edge;

enum Placement { NOT, FIRST, LAST, SURROUNDED };

typedef struct _Save
{
   size_t save;     //Number of characters truncated from this string's
prefix
   Placement place; //Whether this string is first, last, surrounded,
or not in
   size_t first;    //First string in this string's portion
   size_t next;     //Next string in this string's portion
} Save;
```

```
//Declare the edge table used in Greedy and GBB
Edge *edgeTable;

//This function reads the input strings in from stdin, fills in the
input
//table, and returns the size of the input
size_t getInput(char ** &table, size_t &max_length)
{
   size_t i, j;
   size_t n, N = 0;
   char buffer[1024];
   cin>>n;      //Get the size of the input
   table = new char *[n];
   //Read in each string, and copy it into the table
   for (i = 0; i < n; i++)
   {
      cin>>buffer;
      //This loop determines whether of not the new string
      //is a substring of a previously input string, or vice versa
      for (j = 0; j < N; j++)
      {
         if (IS_SUB_STRING(buffer, table[j]))
            j = N+1;
         else if (IS_SUB_STRING(table[j], buffer))
         {
            max_length -= strlen(table[j]);
            delete [] table[j];
            break;
         }
      }
      //The new string is added to the list of strings, or replaces
      //a string that it is a superset of.
      if (j != N+1)
      {
         table[j] = new char[strlen(buffer) + 1];
         strcpy(table[j], buffer);
         N = (j == N) ? N+1 : N;
         max_length += strlen(table[j]);
      }
   }
   return N;
}

//This function compares two edges and returns
//negative:  e1 > e2
//0:  e1 == e2
//positive:  e1 < e2
int edgeCompare(const void *e1, const void *e2)
{
   return ((Edge *)e2)->save - ((Edge *)e1)->save;
}

//This function builds the edge table and sorts it for the given input
void fillEdge(Edge *&edgeTable, char **input, size_t n)
{
   edgeTable = new Edge[n*n - n];
```

```
   size_t index = 0;
   short save1, save2;
   for (size_t i = 0; i < n; i++)
      for (size_t j = i+1; j < n; j++)
      {
         SUFFIX(input[i], input[j], save1, save2);
         //Fill in edge s1-s2
         edgeTable[index].save = save1;
         edgeTable[index].s1 = i;
         edgeTable[index].s2 = j;
         index++;
         //Fill in edge s2-s1
         edgeTable[index].save = save2;
         edgeTable[index].s1 = j;
         edgeTable[index].s2 = i;
         index++;
      }
   qsort(edgeTable, index, sizeof(Edge), edgeCompare);
}

//This function determines whether or not the input edge is feasible,
//based on the current state of the Current Savings Table
bool feasible(Edge E, Save *save)
{
   //The input edge is infeasible if any of the following conditions
are true:
   if (save[E.s1].first == save[E.s2].first ||
                           //If the two strings are in the same portion
       (save[E.s1].place != NOT && save[E.s1].place != LAST) ||
                           //If the first string cannot be followed
       (save[E.s2].place != NOT && save[E.s2].place != FIRST))
                           //If the second string cannot be preceeded
      return false;
   else
      return true;
}

//This function updates the Current Savings Table based on the current
Edge
void update(Edge e, Save *save, size_t n)
{
   Save *s1 = &save[e.s1], *s2 = &save[e.s2];
   s1->next = e.s2;
   //Update the placement information of s1 and s2
   s1->place = (s1->place == NOT) ? FIRST : SURROUNDED;
   s2->place = (s2->place == NOT) ? LAST : SURROUNDED;
   s2->save = e.save;
   //Update the first indices of all strings following s2 (including
s2)
   while (s2 != NULL)
   {
      s2->first = s1->first;
      s2 = (s2->next < n+1) ? &save[s2->next] : (Save *)NULL;
   }
}

//This function restores the Current Savings Table assuming
```

```
//the indicated edge was the last edge added
void unupdate(Edge e, Save *save, size_t n)
{
   Save *s1 = &save[e.s1], *s2 = &save[e.s2];
   s1->next = n+1;
   //Update the placement information of s1 and s2
   s1->place = (s1->place == SURROUNDED) ? LAST : NOT;
   s2->place = (s2->place == SURROUNDED) ? FIRST : NOT;
   s2->save = 0;
   //Update the first indices of all strings following s2 (including
s2)
   while (s2 != NULL)
   {
      s2->first = e.s2;
      s2 = (s2->next < n+1) ? &save[s2->next] : (Save *)NULL;
   }
}

//This method builds a superstring based on a finished CS Table
void buildSuperString(Save *save, char **input, size_t n, char
*superstring)
{
   size_t s = save[0].first;
   superstring[0] = '\0';
   while (s != n+1)
   {
      strcat(superstring, input[s] + save[s].save);
      s = save[s].next;
   }
}

//This routine initializes a Current Savings Table
void initSave(Save *save, size_t n)
{
   for (size_t i = 0; i < n; i++)
   {
      save[i].save = 0;
      save[i].place = NOT;
      save[i].first = i;
      save[i].next = n+1;
   }
}

//The Greedy Method approximates the optimal superstring, stores
//it in superstring, and returns its length
size_t Greedy(char **input, Edge *edgeTable, size_t n, char
*superstring)
{
   Save save[n];
   size_t savings = 0;
   size_t i;
   size_t count = 0;
   //Initialize the current savings table
   initSave(save, n);
   //Go through each edge.  If the current edge is feasible, add it.
   for (i = 0; i < n*n-n && count < n-1; i++)
      if (feasible(edgeTable[i], save))
```

```
      {
         update(edgeTable[i], save, n);
         count++;
         savings += edgeTable[i].save;
      }
   buildSuperString(save, input, n, superstring);
   return strlen(superstring);
}


//The Greedy Method approximates the optimal superstring, stores
//it in superstring, and returns its length
void BBR(char **input, Edge *edgeTable, Save *save, size_t n,
                        size_t edge, size_t max, size_t &best, size_t
saved,
                        size_t cansave, size_t count, char
*superstring)
{
   size_t i;
   if (count == n - 1) //Terminate the recursion when n-1 edges have
been added
   {
      if (max - saved < best) //We've found a new best!
      {
         buildSuperString(save, input, n, superstring);
         best = max - saved;
      }
      return;
   }
   //Go through each remaining edge.  If the current edge is feasible,
add it.
   for (i = edge; i < n*n-n; i++)
      if (feasible(edgeTable[i], save))
      {
         if (max - saved - cansave < best)
         {
            //Update the Current Savings Table
            update(edgeTable[i], save, n);
            BBR(input, edgeTable, save, n, i+1, max, best,
                      saved + edgeTable[i].save, cansave -
edgeTable[i].save,
                      count + 1, superstring);
            //Restore the Current Savings Table
            unupdate(edgeTable[i], save, n);
         }
      }
}


size_t BB(char **input, Edge *edgeTable, size_t n, char *superstring)
{
   Save save[n];
   size_t maxSave[n];
   size_t i, cansave = 0, count = 0, max = 0;
   for (i = 0; i < n; i++)
      maxSave[i] = 0;
   //Determine the most characters that can be saved for each string
```

```
   for (i = 0; i < n*n-n; i++)
      if (maxSave[edgeTable[i].s2] < edgeTable[i].save)
         maxSave[edgeTable[i].s2] = edgeTable[i].save;
   //Sum the max saved characters for each string
   //Sum the maximum number of characters that can appear in a
superstring
   for (i = 0; i < n; i++)
   {
      max += strlen(input[i]);
      cansave += maxSave[i];
   }
   //Initialize the current savings table
   initSave(save, n);
   BBR(input, edgeTable, save, n, 0, max, max, 0, cansave, count,
superstring);
   return max;  //At this point, max is best
}

#define FILL(i, x, y, z) e[i].save = x; e[i].s1 = y-1; e[i].s2 = z-1

void testFill(Edge *e)
{
   FILL(0, 4, 1, 2);
   FILL(1, 3, 1, 3);
   FILL(2, 3, 3, 2);
   FILL(3, 1, 2, 4);
   FILL(4, 0, 1, 4);
   FILL(5, 0, 2, 1);
   FILL(6, 0, 2, 3);
   FILL(7, 0, 3, 1);
   FILL(8, 0, 3, 4);
   FILL(9, 0, 4, 1);
   FILL(10, 0, 4, 2);
   FILL(11, 0, 4, 3);
}


//This routine prints out the Edge Table
void printEdgeTable(Edge *edgeTable, size_t n)
{
   for (size_t i = 0; i < n*n-n; i++)
   {
      cout<<"Edge:  "<<edgeTable[i].save;
      cout<<",  "<<edgeTable[i].s1;
      cout<<",  "<<edgeTable[i].s2<<endl;
   }
}

int main(int argv, char **argc)
{
   Edge *edgeTable;
   size_t n, i, max_length = 0;
   char **input, *superstring;
   //Read in the strings from the input
   n = getInput(input, max_length);
   superstring = new char[max_length + 1];
   for (i = 0; i < max_length; i++)
```

```
      superstring[i] = '\0';
   fillEdge(edgeTable, input, n);

   if (argv > 1 && !strcmp(argc[1], "-e"))
      printEdgeTable(edgeTable, n);
cout<<"Greedy Method results:  "<<endl;
cout<<"   Length:       "<<Greedy(input, edgeTable, n, superstring);
cout<<"   Superstring:  "<<superstring<<endl;

cout<<"Brand and Bound w/ Greedy results:   "<<endl;
cout<<"   Length:       "<<BB(input, edgeTable, n, superstring);
cout<<"   Superstring:  "<<superstring<<endl;

   return 1;
}
```