Advanced Algorithms

Group Project Spring 2018

Alice Karnsund (MMA)
Therese Stålhandske (MMA)

May 18, 2018

1 Introduction

In this project we are considering a simplified version of the problem of inferring phylogenies. We are given a set S, consisting of $n > 0$ distinct binary strings, which are the so called single-nucleotide polymorphism (SNP) profiles. The object is to find the optimal phylogeny. This corresponds to finding the spanning forest which includes all members of S, that has the minimum total weight connecting the strings. In the case of a spanning forest all nodes in S do not have to be connected, thus a spanning forest consists of a set of spanning trees.

The whole problem can be viewed as a non-directed graph $G = (V, E)$, such as the one shown in Figure 1. In our case the set of nodes, V, corresponds to the strings in S, i.e $V = S$, and the edges connecting them are the set E. Where an edge between two nodes $u, v \in S$ exists only if they fulfill the criteria $d(u, v) \leq k$, where k is an integer fulfilling $0 < k \leq m$ and $m > 0$ is the length of a string, same for all in S. Thus the aim is to find the shortest distance (i.e minimum weight) connecting the nodes without the path being cyclic, and not necessarily connecting all nodes in just one tree. If two branches (u_1, v_1) and (u_2, v_2) has the same weight, the path (u_1, v_1) will be chosen instead of (u_2, v_2) if it fulfills the following criteria,

$$d(u_1, v_1) = d(u_2, v_2) \wedge (min\{i_{u_1}, i_{v_1}\} < min\{i_{u_2}, i_{v_2}\} \vee \\ (min\{i_{u_1}, i_{v_1}\} = min\{i_{u_2}, i_{v_2}\} \wedge max\{i_{u_1}, i_{v_1}\} \leq max\{i_{u_2}, i_{v_2}\})$$

Where i_u specifies the position of string u in the input.

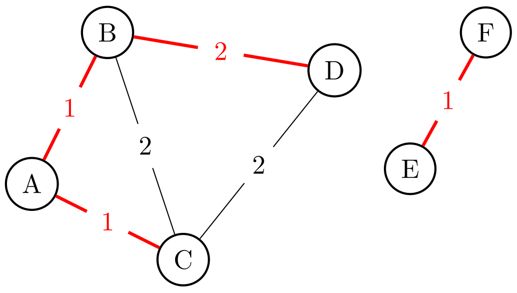

Figure 1: Spanning forest

Figure 1 shows an example of a minimum spanning forest, shown by the red edges.

2 Implementation Decisions

As mentioned in the introduction the the aim for this project is to find the spanning forest in the non-directed graph G. There exist a couple of useful algorithms to find minimum spanning trees and minimum spanning forests. For example Prim's algorithm and Kruskal's algorithm. Prim's algorithm is designed to find a minimum spanning tree and thus has to be applied to all partial graphs. Kruskal's algorithm on the other hand can be applied directly even though the graph might be divided into unconnected parts. Both these algorithms are greedy algorithms. A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.[1] This type of strategy is not generally guaranteed to find a globally optimal solution. But in the case of minimum spanning trees and minimum spanning forests, it can be shown that certain greedy strategies do yield a spanning tree or forest with minimum weight.

2.1 The Kruskal Algorithm

For this project we decided to use the Kruskal Algorithm since no modification of the algorithm is needed for it to find the minimum spanning forest instead of just a single tree.

The algorithm approaches the problem seen from an edge perspective rather than that of the nodes. Between two different trees, it finds the safe edge of least weight to add to the growing forest. If the two trees T_1 and T_2 are connected by the light edge (u, v), Corollary 23.2 in [2] assures that (u, v) is a safe edge for T_1.

In words, the algorithm may be written,

- Create a graph $G = (V, E)$, where V are all the nodes, each representing a separate tree. And E is the set containing all edges in the graph.
- While E is nonempty and G is not yet spanning
  - remove an edge with the minimum weight from E
  - if this edge connects two different trees T_1 and T_2 then add it to the spanning forest, which then combines the two trees to one
- The algorithm terminates when a minimum spanning forest (or tree) of the graph G has been found.

[1]Thomas H. Cormen et al. Introduction to Algortithms, Second Edition. 2nd edition, 2001. Page 317

[2]Thomas H. Cormen et al. Introduction to Algortithms, Second Edition. 2nd edition, 2001. Page 480

2.2 Implementation

The project is implemented in Python version 3.4. The implementation is divided into two parts:

1. Calculating distances between strings and constructing a graph
2. Find the MSF using Kruskal Algorithm

The data was given in a number of files, where each file had the following layout. The first line holds a single integer n, specifying the number of binary strings that will follow. Line 2 to line $n + 1$ contains the set of binary strings, i.e. all are combination of the values $\{0, 1\}$. All strings have the same length, m. Finally the last line in the input file holds a single integer, k, which indicates the distance threshold.

In the graph $G = (V, E)$, E is the set of all hamming distances between the set of nodes V, i.e. all binary strings, that has a length less than or equal to k. To find these we made use of a naive approach, simply calculating the hamming distance between all pairs of the binary strings and then omitting the edges with an hamming distance above k. This approach has a time complexity of at most $\mathcal{O}(n^2)$ and thus is very unattractive for big data sets. This was the most time consuming algorithm used in this implementation. But it do exists other approaches that do not have to take into account all pairs of strings to find E. This is brought up further in the "Discussion and Conclusion" section.

The hamming distances is a quantity in the information theory that specifies the difference between two strings of equal length. For example, consider the two binary strings below. They have equal elements at the same positions apart from at 5 places, thus the hamming distance between these two strings is 5.

$$1001010000101000$$
$$1010000000111001$$

In our specific implementation we let E be a list consisting of elements of the form (h, u, v), where h is the hamming distance, less than or equal to k, and (u, v) are the two corresponding nodes. Finally, for simplicity, we let the set V of all nodes, be a list of elements ranging from 1 to $n + 1$. That is, we named the nodes after what position the stings had in the input.

Further, in the Kruskal Algorithm we implement an union-find data structure with a union-by-rank heuristic, that supports the following operations:

- FIND(x)
- UNION(x,y)

UNION(x,y) creates the union of set x and y, using rank, to replace the single sets of x and y. FIND(x) is implemented as a recursively function that returns the parent of the set.

The implementation time of the Kruskal's algorithm, depends strongly on the implementation of the disjoint-set data structure. The implementation described above have the implementation time of $\mathcal{O}(m \log n)$. Because of the size of the given problem, where the number of nodes are restricted, we chose not to apply additional heuristics to improve the running time. The implemented psuedocode for the Kurskal algorithm is shown below.

```
KURSKAL(V,E)

    1. MST = set()
    2. for each node in V:
    3.     parent[node] = node
    4.     weight[node] = 0
    5. sort edges by weight
    6. for each edge (u,v) in E:
    7.     if FIND(u) != FIND(v):
    8.         UNION(u,v)
    9.         add edge to MST
    10. return MST
```

The time complexity for line 1 is constant, $\mathcal{O}(1)$. For line 2-4, time complexity depends on the number of nodes, $\mathcal{O}(n)$. Sorting the edges are done with comparison sort with the time complexity $\mathcal{O}(llogl)$, where l is the number of edges in E. The disjoint-set operations are optimized to $\mathcal{O}(\log n)$ in worst case. As we do n disjoint-set operations, the lines 6-9 have a time complexity of $\mathcal{O}(l \log n)$. In summary, the total time complexity of the algorithm is $\mathcal{O}(l \log l)$, showing the dependence of the on the number of edges in the graph.

The space complexity is linear dependent on the number of edges, $\mathcal{O}(l)$.

3 Results

Each part of the implementation of the algorithm described in section 2 is evaluated separately, using several families of inputs measuring both time and space.

3.1 Evaluation of string matching

The evaluation of the string matching algorithm is done by generating n randomly binary strings of a given size m with maximum distance of k. For each choice of n, m, k the space and time complexity is calculated as the average over 10 or 100 iterations.

3.1.1 Space Analysis

For evaluation of the space complexity, the values of n are varied and then in effect also the number of edges, which is the interesting factor for the space complexity.

Value of n	Number of edges with weight $\leq k$	Space (kB)
10	18	2
100	2166	181
1000	225 926	27 570

Table 1: Space analysis when varying n, measured in kilobytes (kB). $k = 2$ and $m = 7$ is kept constant.

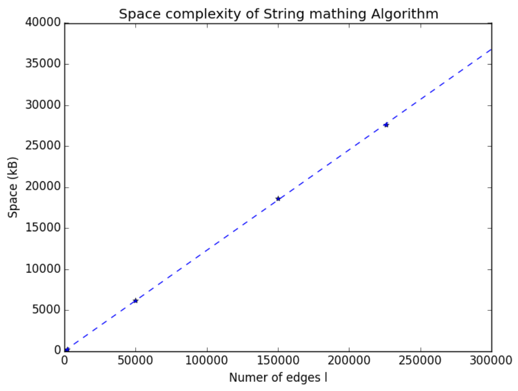

Figure 2: Space as a function of the number of edges. The dashed line is a 1-degree polynomial fit to the points.

3.1.2 Time Analysis

Our implementation of string matching depends on the number of strings, n, and the length of the strings m. For time analysis, n is varied in the range of 10 to 10000 and m is varied in the range of 10 to 2000, while the k is kept constant. Table 2 shows the results of the implementation time when varying the number of strings and table 3 shows the results when varying the length of the strings.

Number of strings n	Average time (s)
10	0.00013
50	0.00281
100	0.01125
500	0.28242
1000	1.13800
5000	28.86115
10000	122.76675

Table 2: Implementation time of string matching when varying n, $n \in (10, 50, 100, 500, 1000, 5000, 10000)$. $k = 2$ and $m = 7$. Averaged over 10 iterations.

As was mentioned earlier this is a naive algorithm and has a time complexity of $\mathcal{O}(n^2)$. This relationship can easily be seen from Figure 3 below, and also from Table 2 by looking at the time values for $n = 10, 100, 1000, 10000$.

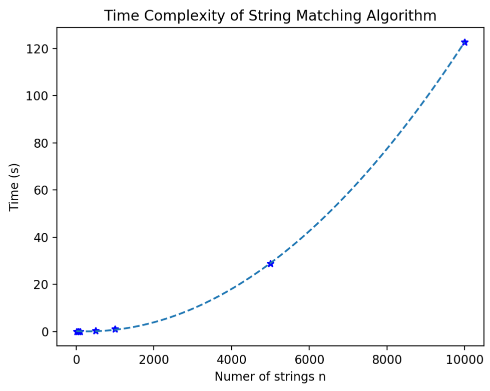

Figure 3: Time complexity of the string matching algorithm. The blue stars represents the values in Table 2. The dashed line is a 2-degrees polynomial fit to the points.

We further investigate the impact of m when keeping n and k constant. Table 3 together with Figure 4 shows a linear result, and thus the time complexity is $\mathcal{O}(m)$.

String length	Average time (s)
10	0.01288
50	0.03964
100	0.07089
500	0.30519
1000	0.63306
1500	0.93039
2000	1.24663

Table 3: Implementation time of string matching when varying m, $m \in (10, 50, 100, 500, 1000, 1500, 2000)$. $n = 100$ and $k = 2$. Averaged over 10 iterations.

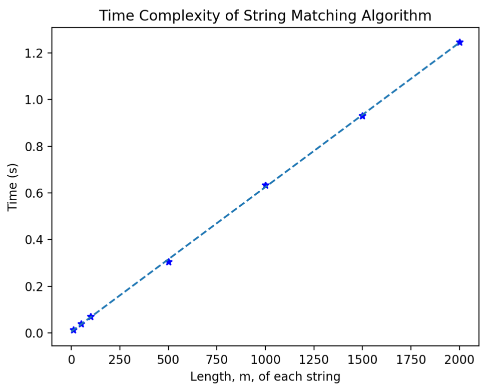

Figure 4: Time complexity of the string matching algorithm. The blue stars represents the values in Table 3. The dashed line is a 1-degree polynomial fit to the points.

We can from this draw the conclusion that if n is much greater than m the time complexity for the string matching algorithm will be $\mathcal{O}(n^2)$, but if m is much greater than n the time complexity will be less and approach a complexity of $\mathcal{O}(m)$. Thus this implementation is more feasible for smaller data sets, i.e. small n values.

3.2 Evaluation of Kruskal algorithm

Evaluation of the Kruskal algorithm is done in a similar manner as described above in section 3.1. The results of the space and time analysis are shown in sections 3.2.1 - 3.2.2.

3.2.1 Space Analysis

For the space analysis, n, k are varied and results are shown in tables 4 - 5.

Value of n	Average space (bytes)
10	589
100	6109
1000	57839

Table 4: Average space analysis when varying n, measured in bytes over 100 iterations. $k = 2$ and $m = 7$ is kept constant.

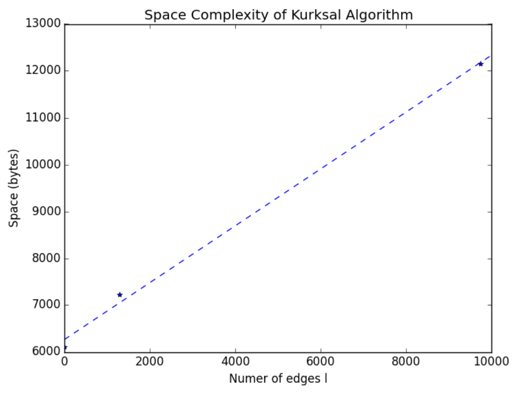

Figure 5: Space as a function of the number of edges, points taken from Table 4. The dashed line is a 1-degree polynomial fit to the points.

Value of k	Average number of edges $\leq k$	Average space (bytes)
8	0	6109
16	1293	7231
32	9736	12149

Table 5: Average space analysis when varying k, measured in bytes over 100 iterations. $n = 100$ and $m = 50$ are kept constant.

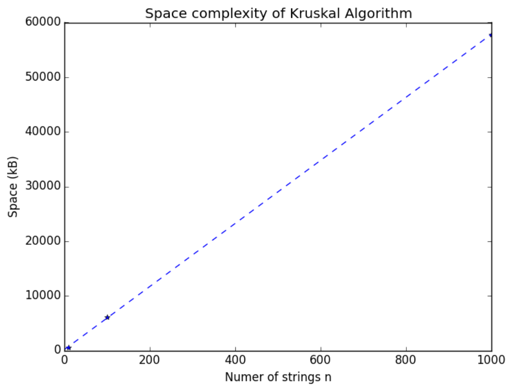

Figure 6: Space as a function of the number of strings, points taken from Table 5. The dashed line is a 1-degree polynomial fit to the points.

3.2.2 Time analysis

The number of edges are controlled by the variable k and the number of nodes. The average results, when varying n and k are shown in tables 6 and 7 respectively.

Number of strings	Average time (s)
10	0.00015
100	0.01412
1000	1.6688
10000	57.897

Table 6: Running time of Kruskal algorithm with varying n. $k = 2$ and $m = 7$ are kept constant.

Value of k	Average number of edges $\leq k$	Average time (s)
8	0	0.000172
16	48	0.00478
32	5440	0.42861
50	9900	0.99098

Table 7: Running time of Kruskal algorithm with varying k. $n = 100$ and $m = 50$ are kept constant.

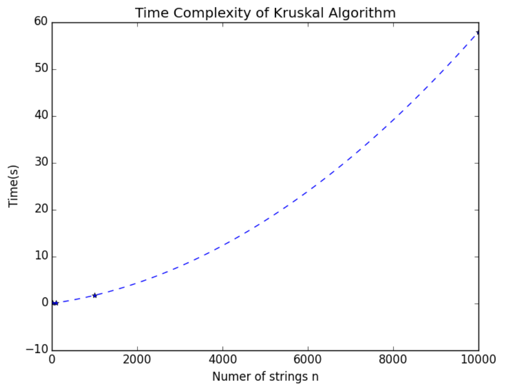

Figure 7: Time complexity as a function of the number of strings, points taken from Table 6. The dashed line is a 2 polynomial fit to the points.

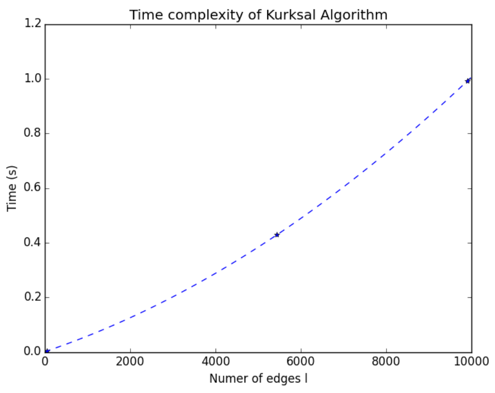

Figure 8: Time complexity as a function of the number of edges, points taken from Table 7. The dashed line is a n log n polynomial fit to the points.

4 Discussion and Conclusion

The space complexity for the string matching and Kruskal algorithm, as shown in section 3.1.1 and section 3.2.1, is linear dependent on the number of edges in the graph. This coincide with the theoretical calculations done in section 2.2. Experimental time complexity for the Kruskal algorithm, shown in section 3.2.2, is comparable to theoretical upper bound. As the number of edges increases, the average time increases with $\mathcal{O}(l \log l)$. The time complexity for the string matching algorithm also agrees with the theory, that the algorithm has an upper bound time complexity of $\mathcal{O}(n^2)$, which can be seen from Figure 3.

This implementation supports the fact that the Kruskal's algorithm manage to find a global minimum using a greedy approach. Anyhow, the main focus in this paper has been the consideration of the time and space complexity. And as mentioned, there are ways to speed up the algorithms and thus making it more feasible to handle even greater data sets.

The most time consuming algorithm is the string matching part where we calculate the hamming distance between all pairs of strings. To speed up this part, one may use approximating string matching techniques, where computations between all pairs of strings are not necessary for small values of k. Example of such techniques are for instance LCP, Longest Common Prefix, which has a time complexity of $\mathcal{O}(m^2)$ which compared to $\mathcal{O}(n^2)$ is a significant improvement. Another technique would be to divide all strings into $(n+1)$ parts and then use a hash-table, $\mathcal{O}(n)$ where n is the number of entries occupied in the hash table, or a bloom filter, which has a time complexity of $\mathcal{O}(k)$, where k is the number of hash functions.