Chapter 3

Example: Genome Sequencing

Sequencing of a complete human Genome (genetic material encoded in the DNA) represents one of the greatest scientific achievements of the century. The efforts started a few decades ago and includes the following major landmarks:

1996	sequencing of first living species
2001	draft sequence of the human Genome
2007	full human Genome diploid sequence

Efficient parallel algorithms played a crucial role in all these achievements. In this lecture, we will take a look at some algorithms behind the results—and the power of problem abstraction which will reveal surprising connections between seemingly unrelated problems.

As with many "real world" applications just defining precisely the problem that models our application is interesting in itself. We therefore start with the vague and not well-defined problem of "sequencing the human genome" and convert it into a precise algorithmic problem. To come up with this problem we will assume that the sequencing is done using a particular method, called the shotgun method.

3.1 The Shotgun Method

Question 3.1. *Do you have any ideas about what makes genome sequencing hard?*

What makes sequencing the genome hard is that there is currently no way to read long strands with accuracy. Current DNA sequencing machines are only capable of efficiently reading relatively short strands, e.g., 1000 base pairs, compared to the over three billion contained in the whole human genome. Scientists therefore cut strands into shorter fragments and then reassemble the pieces.

Figure 3.1: A jigsaw puzzle.

Figure 3.2: Water lilies by Claude Monet. We think of the patches of color as water lilies even though they are just that, patches of color.

Primer walking. A technique called "primer walking" can be used to cut the DNA strands into consecutive fragments and sequence each one. Each step of the process is slow because one needs the result of one fragment to "build" in the wet lab the molecule needed to find the following fragment. Note that primer walking is an inherently sequential technique as a step depends on the previous, making it difficult to parallelize and thus speed up.

Question 3.2. *Can you think of a way to parallelize primer walking?*

One possible way to parallelize primer walking is to divide the genome into a many fragments and sequence them all in parallel. The shortcoming of this approach is that we don't know how to put them together, because we have mixed up the fragments and lost their order.

Example 3.3. *When cut, the strand* cattaggagtat *might turn into,* ag, gag, catt, tat, *destroying the original ordering.*

Question 3.4. *The problem of putting together the pieces is a bit like solving a jigsaw puzzle. But it is harder. Can you see why? Can you think of a way of turning this into a jigsaw puzzle that we can solve?*

The shotgun method. When we cut a genome into fragments we lose all the information about how the fragments are connected. If we had some additional information about how to connect them, we can imagine solving this problem just as we solve a jigsaw puzzle.

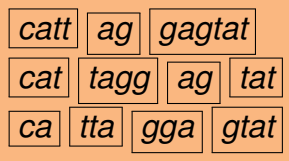
Question 3.5. *Can you think of a way to relate different pieces?*

One way to get additional information about how to join the fragments is to make multiple copies of the original sequence and generate many fragments that overlap. When a fragment overlaps with two others, it can tell us how to connect those two. This is the idea behind the shotgun (sequencing) method, which today seems to be the standard technique for genome sequencing.

Example 3.6. *For example, for the sequence* cattaggagtat*, we produce three copies:*

cattaggagtat
cattaggagtat
cattaggagtat

We then divide each into fragments



Note how each cut is "covered" by an overlapping fragment telling us how to reverse the cut.

Based on this idea, the shotgun method works as follows.

1. Take a DNA sequence and make multiple copies.
2. Randomly cut the sequences using a "shotgun" (actually using radiation or chemicals).
3. Sequence each of the short fragments, which can be performed in parallel with multiple sequencing machines.
4. Reconstruct the original genome from the fragments.

Steps 1–3 are done in a wet lab, while step 4 is the interesting algorithmic component.

Question 3.7. *In step 4, is it always possible to reconstruct the sequence?*

Unfortunately it is not always possible to reconstruct the exact original genome in step 4. For example, we might get unlucky and cut all sequences in the same location. Even if we do cut them in different locations there are many DNA strings that lead to the same collection of fragments. A particularly challenging problem is repetition. For example, just repeating the original string twice can lead to the same set of fragments if the two sequences are always cut at their seam.

3.2 Defining the Problem

Given that there might be an infinite number of solutions and that we may not always hope to find the actual genome, we wish to define a problem that makes precise the "best solution" that we wish to find.

Question 3.8. *How can we make this intuitive notion of a "best solution" precise?*

It is not easy to make this notion of best solution precise. This is why, it can be as difficult and important to formulate a problem as it is to solve it. But as we will see, we can come pretty close to a realistic solution.

Question 3.9. *Can you think of a property that the result needs to have in relation to the fragments?*

Note that since the fragments all come from the original genome, the result should at least contain all of them. In other words, it is a *superstring* of the fragments. As mentioned earlier, however, there will be multiple superstrings for any given set of fragments.

We can take one more step in making the problem more precise by constructing the "best" superstring.

Question 3.10. *Which of the many superstrings should we pick?*

How about the shortest superstring? This would give us the simplest explanation, which is often desirable. The principle of selecting the simplest explanation is often referred to as Occam's razor (see Figure **??**) .

Problem 3.11 (The Shortest Superstring (SS) Problem)**.** *Given an alphabet set Σ and a set of finite-length strings $S \subseteq \Sigma^+$, return a shortest string r that contains every $s \in S$ as a substring of r.*

Figure 3.3: William of Occam (or Ockham, 1287-1347) posited that among competing hypotheses that explain some data, the one with the fewest assumptions or shortest description is the best one. The term "razor" apparently refers to shaving away unnecessary assumptions, although here is a more modern take on it.

In the definition the notation Σ^+, the "Kleene plus", means the set of all possible non-empty strings consisting of characters Σ. Note that, for a string s to be a *substring* of another string r, s must as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg".

We will define the genome-sequencing problem as the problem of finding the shortest superstring that contains all the fragments, i.e., given a set of fragments, construct the shortest string that contains all of them. For genome sequencing, we have $\Sigma = \{a, c, g, t\}$. We have thus converted a vague problem, sequencing the genome, into a concrete problem, the SS problem. As suggested by the discussion thus far and further discussed at the end of this chapter, the SS problem might not be exactly the right abstraction for the application of sequencing the genome, but it is a good start.

Having specified the problem, we are ready to design an algorithm, in fact a few algorithms, for solving it. Let's start with a brute-force algorithm.

3.3 Brute-Force Algorithm 1

As discussed in Section 1.3 the brute-force technique consist of trying all candidate solutions and selecting the best (or any) valid solution. For the SS problem a solution is valid if it is a superstring of all the input strings, and we want to pick the shortest of these strings.

We consider two brute force solutions. The first is simply to consider all strings $r \in \Sigma^+$,

and for each r to check if every string $s \in S$ is a substring. It turns out that such a check can be done efficiently, although we won't describe how here. We then pick the shortest r that is indeed a superstring of all $s \in S$. The problem, however, is that there are an infinite number of strings in Σ^+ so we cannot check them all, but perhaps we can be a little smart about picking the candidate solutions r.

Question 3.12. *Do we really have to consider all strings?*

We only need to consider strings up to length $m = \sum_{s \in S} |s|$ since we can easily construct a superstring by concatenating all strings. Since the length of such a string is m, the shortest superstring has length at most m.

Question 3.13. *How many string of length m are there?*

Unfortunately, there are still $|\Sigma|^m$ strings of length m; this number is not infinite but still very large. For the sequencing the genome $\Sigma = 4$ and m is in the order of billions, giving something like $4^{1{,}000{,}000{,}000}$. There are only about $10^{80} \approx 4^{130}$ atoms in the universe so there is no feasible way we could apply the brute force method directly. In fact we can't even apply it to two strings each of length 100.

3.4 Understanding the Structure of the Problem

This brute force-algorithm is rather disappointing. We can improve it quite a bit it turns out but we have make some observations.

Observation 1: Snippets. Note that in solving the superstring problem we can ignore strings that are contained in other strings. For example, if we have gagtat, ag, and gt, we can throw out ag and gt. In the context of the genome sequencing problem, we will refer to the fragments that are not contained in others as *snippets.*

Example 3.14. *In our example, we had the following fragments.*

catt ag gagtat
cat tagg ag tat
ca tta gga gtat

Our snippets are now:

$$S = \{\mathsf{catt}, \mathsf{gagtat}, \mathsf{tagg}, \mathsf{tta}, \mathsf{gga}\}.$$

The other fragments $\{\mathsf{cat}, \mathsf{ag}, \mathsf{tat}, \mathsf{ca}, \mathsf{gtat}\}$ *are all contained within the snippets.*

Since no snippet can be contained in another, in the result superstring, snippets cannot start at the same position, and if one starts after another, it must finish after the other. This leads to our second observation.

Observation 2: Ordering of the snippets. In any superstring, the start positions of the snippets is a strict (total) ordering, which is the same ordering as their finish positions.

We are now ready to solve the SS problem by designing algorithms for it. Designing algorithms may appear to be an intimidating task, because it may seem as though we would need brilliant ideas that come out of nowhere. Like the water lilies of Monet, this is just an appearance. In reality, we design algorithms by starting with simple ideas based on several well-known techniques and refine them until we reach the desired result, much like a painter constructing a painting with simple but deliberate brush strokes.

In the rest of this section, we will consider three algorithmic techniques that can be applied to this problem and derive an algorithm from each.

3.5 Brute Force Algorithm 2

We now consider a second brute force solution. In addition to requiring less computational work, the approach will also help us understand other possible solutions, which we look at next. In the last section we mentioned that snippets have to start in some ordering in any superstring. If this is the case, we can try all possible orderings of the snippets. One of the orderings has to be the right ordering for the shortest superstring. The question remains of how once we pick an ordering, we then find the shortest superstring for that ordering. For this purpose we will use the following theorem.

Theorem 3.15 (Removing Overlap)**.** *Given any start ordering of the snippets $s_1, s_2, \ldots, s_n$, removing the maximum overlap between each adjacent pair of snippets (s_i, s_{i+1}) gives the shortest superstring of the snippets for that start ordering.*

Example 3.16. *For our running example, consider the following ordering*

catt tta tagg gga gagtat

When the maximum overlaps are removed (the excised parts are underlined) we get cattaggagtat*, which is indeed the shortest superstring for the given start ordering (indeed it is also the overall shortest).*

Proof. The theorem can be proven by induction. The base case is true since it is clearly true for a single snippet. Inductively, we assume it is true for the first i snippets, i.e. that removing the maximum overlap between adjacent snippets among these i is the shortest superstring of

$s_1, \ldots, s_i$ starting in that order. We refer to this superstring as r_i. We now prove that if it is true for r_i then it is true for r_{i+1}. Note that when we add the snippet s_{i+1} after r_i it cannot fully overlap with the previous snippet (s_i) by the definition of snippets. Therefore when we add it on using the maximum overlap, the string r_{i+1} will be r_i with some new characters added to the end. This will still be a superstring of all the first i snippets since we did not modify r_i, plus it will also be a superstring of s_{i+1}, since we are including it at the end. It will also be the shortest since r_i is the shortest for $s_1, \ldots s_i$ (starting in that order) and by picking the maximum overlap we added the least number of additional characters to get a superstring for $s_1, \ldots s_{i+1}$ □

This theorem tells us that we can try all permutations of the snippets, calculate the length of each one by removing the overlaps and pick the best.

Exercise 3.17. *Try a couple other permutations and determine the length after removing overlaps.*

We now look at the work required for this second brute-force algorithm. There are $n!$ permutations on a collection of n elements each of which has to be tried. For $n = 10$ strings this is probably feasible, which is better than our previous technique that did not even work for $n = 2$. However for $n = 100$, we'll need to consider $100! \approx 10^{158}$ combinations, which is still more than the number of atoms in the universe. As such, the algorithm is still not going to be feasible for large n.

Question 3.18. *Can we come up with a smarter algorithm that solves the problem faster?*

Unfortunately the SS problem turns out to be NP-hard, although we will not show this.

Question 3.19. *Is there no way to efficiently solve an instance of an NP-hard problem?*

When a problem is NP hard, it means that there are *instances* of the problem that are difficult to solve. NP-hardness doesn't rule out the possibility of algorithms that quickly compute near optimal answers or algorithms that perform well on real world instances. For example the type-checking problem for the ML language is NP-hard but we use ML type-checking all the time without problems, even on programs with thousands of variables.

For this particular problem, we know efficient approximation algorithms that (1) give theoretical bounds that guarantee that the answer (i.e., the length) is within a constant factor of the optimal answer, and (2) that in practice perform even better than the bounds suggest.

Figure 3.4: A poster from a contest run by Proctor and Gamble in 1962. The goal was to solve a 33 city instance of the TSP. Gerald Thompson, a Carnegie Mellon professor, was one of the winners.

3.6 Reduction to the Traveling-Salesperson Problem

Another approach to solving a problem is to reduce it to another problem which we understand better and for which we know algorithms, or possibly even have existing code. It is sometimes quite surprising that problems that seem very different can be reduced to each other. Note that reductions are sometimes used to prove that a problem is NP-hard (i.e. if you prove that using polynomial work you can reduce an NP-complete problem A to problem B, then B must also be NP-complete). That is *not* the purpose here. Instead we want the reduction to help us solve our problem.

In particular we consider reducing the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson (TSP) problem.

Question 3.20. *Are you all familiar with the TSP problem?*

The TSP problem is a canonical NP-hard problem dating back to the 1930s and has been extensively studied, e.g. see Figure **??**. The two major variants of the problem are *symmetric* TSP and *asymmetric* TSP, depending on whether the graph has undirected or directed edges, respectively. The particular variant we're reducing to is the asymmetric version, which can be described as follows.

Problem 3.21 (The Asymmetric Traveling Salesperson (aTSP) Problem)**.** *Given a weighted directed graph, find the shortest path that starts at a vertex s and visits all vertices exactly once before returning to s.*

That is, find a Hamiltonian cycle of the graph such that the sum of the edge weights along the cycle is the minimum of all such cycles (a cycle is a path in a graph that starts and ends at the same vertex, and a Hamiltonian cycle is a cycle that visits every vertex exactly once). You can think of the TSP problem as the problem of coming up with best possible plan for your annual road trip.

Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the permutations for us.

Question 3.22. *Can we set up the TSP problem so that it tries all permutations for us?*

For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. The graph will be complete, containing an edge between any two vertices, and thus guaranteeing the existence of a Hamiltonian cycle.

Let $\texttt{overlap}(s_i, s_j)$ denote the maximum overlap for s_i followed by s_j.

Example 3.23. *For "tagg" and "gga", we have* $\texttt{overlap}(\text{“tagg”}, \text{“gga”}) = 2$.

The Reduction. Now we build a graph $D = (V, A)$.

- The vertex set V has one vertex per snippet and a special "source" vertex Z where the cycle starts and ends.
- The arc (directed edge) from s_i to s_j has weight $w_{i,j} = |s_j| - \texttt{overlap}(s_i, s_j)$. This quantity represents the increase in the string's length if s_i is followed by s_j.

  For example, if we have "tagg" followed by "gga", then we can generate "tagga" which only adds 1 character giving a weight of 1—indeed, $|\text{“gga”}| - \texttt{overlap}(\text{“tagg”}, \text{“gga”}) = 3 - 2 = 1$.

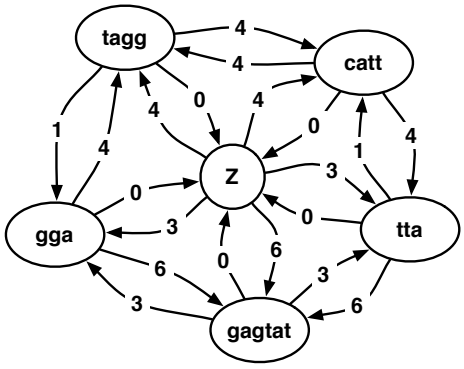


Figure 3.5: Reduction to TSP example (not all edges are shown).

- The weights for arcs incident to Z are set as follows: $(Z, s_i) = |s_i|$ and $(s_i, Z) = 0$. That is, if s_i is the first string in the permutation, then the arc (Z, s_i) pays for the whole length s_i. If s_i is the last string we have already paid for it, so the arc (s_i, Z) is free.

To see this reduction in action, the snippets in our running example, $\{$catt, gagtat, tagg, tta, gga$\}$ results in the graph shown in Figure 3.5 (not all edges are shown).

Question 3.24. *What does a Hamiltonian cycle in the graph starting at the source correspond to? What about the total weight of the edges on a cycle?*

As intended, in this graph, a cycle through the graph that visits each vertex once corresponds to a permutation in the brute force method. Furthermore, the sum of the edge weights in that cycle is equal to the length of the superstring produced by the permutation.

Question 3.25. *Is there a cycle in the graph for each permutation?*

Note that since the graph is complete, we can construct a cycle for each permutation by visiting the corresponding vertices in the graph in the specified order. Conversely, we each cycle corresponds to a permutation. We have thus established an equivalence between permutations and the Hamiltonian cycles in the graph.

Since TSP considers all Hamiltonian cycles, it considers all orderings in the brute force method. Since the TSP finds the min-cost cycle, and assuming the brute force method is correct, then TSP finds the shortest superstring. Therefore, if we could solve the TSP problem, we would be able to solve the shortest superstring problem.

TSP is also NP-hard. What we have accomplished so far is that we have reduced one NP hard problem to another, but the advantage is that there is a lot known about TSP, so perhaps this helps.

3.7 Greedy Algorithm

We now consider a third technique, the "greedy" technique, and a corresponding algorithm.

Definition 3.26 (The Greedy **Technique**). *Give a set of elements, on each step remove at least one element by making a locally optimal decision based on some criteria.*

For example, a greedy technique for the TSP could be to always visit the closest unvisited city. Each step makes a locally optimal decision, and each step removes one element, the next city visited.

Question 3.27. *Does the greedy technique always return the optimal solution?*

The greedy technique (or approach) is a heuristic that in some cases returns an optimal solution, but in many cases it does not. For a given problem there might be several greedy approaches that depend on the types of steps and what is considered to be locally optimal. The greedy approach for the SS problem we now consider does not guarantee that we will find the optimal solution, but it can guarantee to give a good approximation. Furthermore it works very well in practice. Greedy algorithms are popular because of their simplicity.

Question 3.28. *Considering that we want to minimize the length of the result, what should our "greedy choice" be?*

To choose an appropriate greedy approach for the SS problem, note that to minimize the length of the superstring we would need to maximimize the overlap among the snippets. Thus we can greedily pick a pair of snippets with the largest overlap and join them by placing one immediately after the other and removing the overlap. This can then be repeated until there is only one string left.

To describe the algorithm more precisely, we define a function $\texttt{join}(s_i, s_j)$ that places s_j after s_i and removes the maximum overlap. For example, $\texttt{join}(\text{“tagg”}, \text{“gga”}) = \text{“tagga”}$. The pseudocode for our algorithm is given in Algorithm 3.29.

Given a set of strings S, the *greedyApproxSS* algorithm checks if the set has only 1 element, and if so returns that element. Otherwise it finds the pair of distinct strings s_i and s_j

Algorithm 3.29 (Greedy Approximate SS)**.**

```
1  function greedyApproxSS (S) =
2    if |S| = 1 then
3      S_0
4    else
5      let
6        val O = {(overlap(s_i, s_j), s_i, s_j) : s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
7        val (o, s_i, s_j) = arg max_{(x,_,_)∈O} x
8        val s_k = join(s_i, s_j)
9        val S' = ({s_k} ∪ S)\{s_i, s_j}
10     in
11       greedyApproxSS (S')
12     end
```

in S that have the maximum overlap. It does this by first calculating the overlap for all pairs (Line 6) and then picking the one of these that has the maximum overlap (Line 7). Note that O is a set of triples each corresponding to an overlap and the two strings that overlap. The notation $\arg\max_{(x,\_,\_)\in O} x$ is mathematical notation for selecting the element of O that maximizes the first element of the triple, which is the overlap. After finding the pair (s_i, s_j) with the maximum overlap, the algorithm then replaces s_i and s_j with $s_k = \texttt{join}(s_i, s_j)$ in S.

Question 3.30. *Is the algorithm guaranteed to terminate?*

The new set S' is one smaller than S and that the algorithm recursively repeats this process on this new set of strings until there is only a single string left. It thus terminates after $|S|$ recursive calls.

The algorithm is greedy because at every step it takes the pair of strings that when joined will remove the greatest overlap, a locally optimal decision. Upon termination, the algorithm returns a single string that contains all strings in the original S. However, the superstring returned is not necessarily the shortest superstring.

Exercise 3.31. *In the code we remove s_i, s_j from the set of strings but do not remove any strings from S that are contained within s_k = $\texttt{join}(s_i, s_j)$. Argue why there cannot be any such strings.*

Exercise 3.32. *Prove that algorithm* `greedyApproxSS` *indeed returns a string that is a superstring of all original strings.*

Exercise 3.33. *Give an example input S for which* `greedyApproxSS` *does not return the shortest superstring.*

Exercise 3.34. *Consider the following greedy algorithm for TSP. Start at the source and always go to the nearest unvisited neighbor. When applied to the graph described above, is this the same as the algorithm above? If not what would be the corresponding algorithm for solving the TSP?*

Parallelizing the greedy algorithm. Although the greedy algorithm merges pairs of strings one by one, we note there is still significant parallelism in the algorithm, at least as described. In particular we can calculate all the overlaps in parallel, and the largest overlap in parallel using a reduction. We will look at the cost analysis in more detail in the next lecture.

Approximation quality. Although `greedyApproxSS` does not return the shortest superstring, it returns an "approximation" of the shortest superstring. In particular, it is known that it returns a string that is within a factor of 3.5 of the shortest and conjectured that it returns a string that is within a factor of 2. In practice, it typically performs much better than the bounds suggest. The algorithm also generalizes to other similar problems.

Of course, given that the SS problem is NP-hard, and `greedyApproxSS` does only polynomial work (see below), we cannot expect it to give an exact answer on all inputs—that would imply **P** = **NP**, which is unlikely. In literature, algorithms such as `greedyApproxSS` that solve an NP-hard problem to within a constant factor of optimal, are called *constant-factor approximation algorithms*.

Remark 3.35. *Often when abstracting a problem we can abstract away some key aspects of the underlying application that we want to solve. Indeed this is the case when using the Shortest Superstring problem for sequencing genomes. In actual genome sequencing there are two shortcomings with using the SS problem. The first is that when reading the base pairs using a DNA sequencer there can be errors. This means the overlaps on the strings that are supposed to overlap perfectly might not. Don't fret: this can be dealt with by generalizing the Shortest Superstring problem to deal with approximate matching. Describing such a generalization is beyond the scope of this course, but basically one can give a score to every overlap and then pick the best one for each pair of fragments. The nice thing is that the same algorithmic techniques we discussed for the SS problem still work for this generalization, only the "overlap" scores will be different.*

The second shortcoming of using the SS problem by itself is that real genomes have long repeated sections, possibly much longer than the length of the fragments that are sequenced. The SS problem does not deal well with such repeats. In fact when the SS problem is applied to the fragments of an initial string with longer repeats than the fragment sizes, the repeats or parts of them are removed. One method that researchers have used to deal with this problem is the so-called double-barrel shotgun method. *In this method strands of DNA are cut randomly into lengths that are long enough to span the repeated sections. After cutting it up one can read just the two ends of such a strand and also determine its length (approximately). By using the two ends and knowing how far apart they are it is possible to build a "scaffolding" and recognize repeats. This method can be used in conjunction with the generalization of the SS discussed in the previous paragraph. In particular the SS method allowing for errors can be used to generate strings up to the length of the repeats, and the double barreled method can put them together.*

.