Chapter 5

Example: Genome Sequencing

The human genome is the full nucleic acid sequence for humans; it consists of nucleotide ***bases*** of A (Adenine), Cytosine (C), Guanine (G), or Thymine (T). It contains over 3 billion ***base pairs***, each of which consist of two nucleotide bases bound by hydrogen bonds. It can be written as a sequence or a string of bases consisting of the letters, "A", "C", "G", and "T". The sequence, if printed as a book, would be about as tall as the Washington Monument. The human genome is present in each cell of the human body. It appears to have all the information needed by the cells in our bodies; it is expected that its deeper understanding will lead to insights into the nature of life.

Sequencing of a complete human genome represents one of the greatest scientific achievements of the century. When the human genome project was first proposed in mid 1980's, the technology available could only sequence couple of hundred bases at a time. After a few decades, the efforts led to the several major landmark accomplishments. In 1996, the DNA of the first living species (fruit fly) was sequenced. This was followed, in 2001, by the draft sequence of the human genome. Finally in 2007, the full human genome diploid was sequenced.

Efficient parallel algorithms played a crucial role in all these achievements. In this chapter, we review some of the 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, 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. Our starting point is the "shotgun method" for genome sequencing, which was the method used for sequencing the human genome.

5.1 The Shotgun Method

What makes sequencing the genome hard is that there is currently no way to read long strands with accuracy. Current DNA "reading" techniques are only capable of efficiently reading relatively short strands, e.g., 10-1000 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.

Primer walking. A technique called "primer walking" can be used to cut the DNA strands into consecutive fragments and sequence each one. The process is slow and sequential because the result of one fragment is used to construct (in a lab) the molecule needed to find the following fragment. One possible way to parallelize primer walking is to divide the genome into 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. This approach can be viewed as making a jigsaw puzzle out a picture that we have not seen before; it can be difficult and even impossible to solve such a jigsaw puzzle.

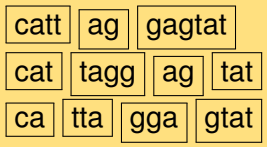
Example 5.1. When cut, the strand cattaggagtat might turn into, ag, gag, catt, tat, destroying the original ordering.

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, however, we could imagine solving this problem just as we solve a jigsaw puzzle. One way to get additional information on joining the fragments is to make multiple copies of the original sequence and generate many fragments that overlap. Overlaps between fragments then be used to connect them. This is the idea behind the shotgun (sequencing) method, which was the primary method used to first sequence the human genome.

Example 5.2. 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.

The shotgun method works as follows.

1. Take a DNA sequence and make multiple copies.
2. Randomly cut the sequences using a "shotgun" (in reality, using radiation or chemicals).

3. Sequence each of the short fragments, which can be performed in parallel.

4. Reconstruct the original genome from the fragments.

Steps 1–3 are done in a wet lab, while step 4 is the algorithmically interesting component. 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 cut them in different locations, there can be many DNA sequences that lead to the same collection of fragments. A particularly challenging problem is repetition: there is no easy way to know if repeated fragments are actual repetitions in the sequence or if they are a product of the method itself.

5.2 Defining the Problem

Given that there might be a many 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 can find.

It is not easy to define the solution precisely. 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.

To start with, 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. There can, however, be multiple superstrings for any given set of fragments. Which one 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***.

Problem 5.3. [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 $x \in A$ as a substring of r.

In the definition the notation Σ^*, the "Kleene star", 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".

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.

Figure 5.1: 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.

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. In reality, we design algorithms by starting with simple ideas based on several well-known techniques and refine them until the desired result is reached. We now consider three algorithmic techniques and apply them to this problem, deriving an algorithm from each.

5.3 Brute-Force Algorithm 1

As applied to the genome-sequencing problem, the brute-force technique involves trying all candidate superstrings of the input fragments and selecting the shortest one.

One idea is to consider all strings $x \in \Sigma^*$, and for each x to check if every fragment is a substring. Although we won't describe how here, such a check can be performed efficiently. We then pick the shortest x that is indeed a superstring of all fragments. The problem, however, is that there are an infinite number of strings in Σ^* so we cannot check them all. Fortunately, we don't need to: we only need to consider strings up to the total length of the fragments m, since we can easily construct a superstring by concatenating all fragments. Since the length of such a string is m, the shortest superstring has length at most m. 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.

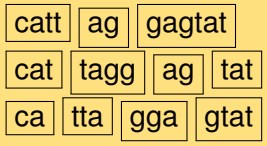
5.4 Understanding the Structure of the Problem

To improve over the brute force-algorithm described above, let's take a closer look at the problem to make some observations.

Observation 1: Snippets. First, observe that we can ignore strings that are contained in other strings, because they don't contribute to the solution. 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***.

Observation 2: Ordering of the snippets. Since no snippet can be contained in another, in the result superstring, snippets cannot start at the same position. Furthermore, if one starts after another, it must finish after the other. This leads to our second observation: in any superstring, the start positions of the snippets is a strict (total) order, which is the same order as their finish positions. In other words, we only need to consider permutations of the snippets.

Example 5.4. In our example, we had the following fragments.



Our snippets are now:

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

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

Consider a superstring such as cattaggagtat. The starting points for the snippets are: 0 for catt, 2 for tta, 3 for tagg, 5 for gga.

In the following discussion we will use $n = |S|$, i.e. the number of snippets, and $m = \sum_{i=1}^{n} |S_i|$, i.e. the total size of the snippets.

5.5 Brute Force Algorithm 2

Based on our new understanding of the problem, let's consider a second brute force solution. In addition to requiring less computational work, this solution will also help us understand other possible solutions. In the last section, we concluded that we only have to consider permutations of the snippets. We can thus consider all permutations, and find the permutation that gives us the shortest superstring. The question remains of how, once we pick a permutation, we then

find the shortest superstring for that permutation. For this purpose we will use the following theorem.

Theorem 5.5 (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 permutation.*

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 snippets yields 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 the theorem it is true for i then it is true for $i+1$. Consider adding the snippet s_{i+1} after r_i, we know that s_{i+1} does not fully overlap with the previous snippet (s_i) by the definition of snippets. Therefore when we add it on using the maximum overlap, the resulting string r_{i+1} will be r_i with some new characters added to the end. The string r_{i+1} is a superstring of $s_0, \ldots, s_{i+1}$ because it includes r_i, which by induction is a superstring of $s_0, \ldots, s_i$, and because it includes s_{i+1}. It is also be the shortest since r_i is the shortest for $s_1, \ldots s_i$ and a shorter string would not include s_{i+1}, because we have already eliminated the maximum overlap between s_{i+1} and r_i. □

Example 5.6. For our running example, consider the following permutation

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 permutation. In this case, it is also the overall shortest superstring.

Since the shortest superstring naturally defines a total order on the snippets, we can compute the shortest superstring by trying all permutations of the snippets, finding the shortest superstring for that permutation by removing the overlaps, and pick the overall shortest. Let's make our algorithm more precise and calculate the work and span of this algorithm. First we need an algorithm for finding the maximum overlap between two strings s and t. To this end, we can again use the brute force-technique: consider each suffix of s and check if it is a prefix of t and select the longest such suffix that is also a prefix. The work of this algorithm is $|s| \cdot |t|$, i.e., the product of the lengths of the strings. The span for checking a particular suffix is also a prefix is $O(\lg t)$ by using a divide-and-conquer algorithm. Since we can try all positions in parallel, the span is thus $O(\lg |t|)$. The span for selecting the maximum over all matches is $O(\lg s)$. The total span is thus $O(\lg s + \lg t) = O(\lg s + t)$.

Now that we know how to find the overlap reasonably efficiently, we can try each permutation, and eliminate overlaps. This would, however, require repeatedly computing the overlap between many of the same snippets. Since we only remove the overlap between successive snippets, there are only $O(n^2)$ pairs that we have to consider. We can thus ***stage*** the algorithm into two stages.

- First, compute the overlaps between each pair of snippets and store them in a table for quick lookup.
- Second, try all permutations and compute the shortest superstring by removing overlaps as defined by the table.

Let W_1 and S_1 be the work and span for the first phase of the algorithm, i.e., for calculating all pairs of overlaps in our set of input snippets $s = \{s_1, \ldots, s_n\}$. Let $m = \sum_{x \in S} |x|$. Using our algorithm, $\texttt{overlap}(x, y)$ for finding the maximum overlap between two strings x and y, we have

$$\begin{aligned}
W_1 &\leq \sum_{i=1}^{n}\sum_{j=1}^{n} W(\texttt{overlap}(s_i, s_j))) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} O(|s_i||s_j|) \\
&\leq \sum_{i=1}^{n}\sum_{j=1}^{n} (c_1 + c_2|s_i||s_j|) \\
&= c_1 n^2 + c_2 \sum_{i=1}^{n}\sum_{j=1}^{n} (|s_i||s_j|) \\
&= c_1 n^2 + c_2 \sum_{i=1}^{n} \left(|s_i| \sum_{j=1}^{n} |s_j| \right) \\
&= c_1 n^2 + c_2 \sum_{i=1}^{n} (|s_i| m) \\
&= c_1 n^2 + c_2 m \sum_{i=1}^{n} |s_i| \\
&= c_1 n^2 + c_2 m^2 \\
&\in O(m^2) \qquad \text{since } m \geq n.
\end{aligned}$$

and since all pairs can be considered in parallel,

$$\begin{aligned}
S_1 &\leq \max_{i=1}^{n} \max_{j=1}^{n} S(\texttt{overlap}(s_i, s_j))) \\
S_1 &\leq \max_{i=1}^{n} \max_{j=1}^{n} O(\lg |s_i| + |s_j|) \\
&\in O(\lg m)
\end{aligned}$$

We therefore conclude that the first stage of the algorithm requires $O(m^2)$ work and $O(\lg m)$ span.

Moving onto the second stage, we want to compute the shortest superstring for each permutation. Given a permutation, we know that all we have to do is remove the overlaps. Since there are n overlaps to be considered and summed, this requires $O(n)$ work, assuming that we can lookup the overlaps in constant time. Since we can lookup the overlaps in parallel,

the span is constant for finding the overlaps and $O(\lg n)$ for summing them. Therefore the cost for handling each permutation is $O(n)$ work and $O(\lg n)$ span. Unfortunately, there are $n!$ permutations to consider. Even though we have designed reasonably efficient algorithms for computing the overlaps and the shortest superstring for each permutation, there are too many permutations for this algorithm to be efficient. For $n = 10$ strings the algorithm is probably feasible, which is better than our previous brute-force algorithm, which 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. Thus, the algorithm is still not feasible if the number of snippets is more than a couple dozen.

Unfortunately the SS problem turns out to be NP-hard, although we will not show this. 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 strongly typed languages (e.g., the ML family of languages) is NP-hard but we use them all the time, even on large programs.

For this particular problem, we know efficient approximation algorithms that are theoretically good: they guarantee that the length of the superstring is within a constant factor of the optimal answer. Furthermore, these algorithms tend to perform even better in practice than the theoretical bounds suggest. We will discuss such an algorithm in Section 5.7.

Remark 5.7. The technique of staging used above is a key technique in algorithm design as well as implementation. The basic idea is to identify a computation that is repeated many times in the future and pre-compute it, storing it in some fast lookup data structure. Later instances of that computation can then be recalled via lookup instead of re-computing every time it is needed.

5.6 Reduction to the Traveling-Salesperson Problem

Another technique in algorithm design is to reduce an algorithmic problem to another problem that we know how to solve. It is sometimes quite surprising that problems that seem very different can be reduced to each other. Here, we shall reduce the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson (TSP) problem. The TSP problem is a canonical NP-hard problem dating back to the 1930s and has been extensively studied. 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 asymmetric TSP problem requires finding a Hamiltonian cycle of the graph such that the sum of the arc (directed edge) weights along the cycle is the minimum of all such cycles. Recall that a cycle is a path in a graph that starts and ends at the same vertex and that a ***Hamiltonian cycle*** is a cycle that visits every vertex exactly once. The symmetric version of the problem can be viewed as considering only graphs where for each arc (u, v), there is also a reverse arc (v, u) with the same weight.

Figure 5.2: 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.

Problem 5.8. [The Asymmetric Traveling Salesperson (aTSP) Problem] Given a weighted directed graph, find the shortest cycle that starts at some vertex and visits all vertices exactly once before returning to the starting vertex.

The idea behind the reduction of the SS problem to the TSP problem comes from something that we have learned from our brute-force algorithm: the shortest superstring problem can be solved by trying all permutations. In particular we shall make the TSP problem try all the permutations for us. For the reduction, we set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. The graph is ***complete***: it contains an arc between any two vertices, guaranteeing the existence of a Hamiltonian cycle.

To specify the reduction, let $\texttt{overlap}(s_i, s_j)$ denote the maximum overlap for s_i followed by s_j. For example, for "tagg" and "gga", we have $\texttt{overlap}$ ("tagg","gga") $= 2$. Now we build a graph $D = (V, A)$.

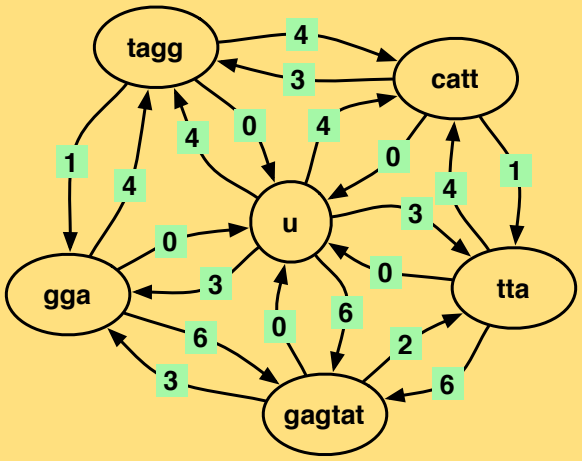
- The vertex set V has one vertex per snippet and a special "source" vertex u where the

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, |"gga"| − overlap("tagg", "gga") = 3 − 2 = 1.

- The weights for arcs incident to source u are set as follows: $(u, s_i) = |s_i|$ and $(s_i, u) = 0$. That is, if s_i is the first string in the permutation, then the arc (u, 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, u) is free.

Example 5.9. To see this reduction in action, the snippets in our running example, {catt, gagtat, tagg, tta, gga} results in the graph, a subgraph of which is shown below (not all arcs are shown).



As intended, in this graph, a Hamiltonian cycle corresponds to a permutation in the brute force method: we start from the source and follow the cycle to produce the permutation. Furthermore, the sum of the arc weights in that cycle is equal to the length of the superstring produced by the permutation. Since the TSP finds the minimum weight cycle, it also finds the permutation that leads to the shortest superstring. Therefore, if we could solve the TSP problem, we can solve the shortest superstring problem.

We have thus reduced the shortest-superstring problem, which is NP-hard, to another NP-hard problem: TSP. We constructed the reduction by using an insight from a brute-force algorithm: that we can solve the problem by trying out all permutations. The advantage of this reduction is that we know a lot about TSP, which can help, because for any algorithm that solves or approximates TSP, we obtain an algorithm for the shortest-superstring problem, and thus for sequencing the genome.

Example 5.10. [Hardness of shortest superstring] We can also establish that the shortest superstring problem is NP-hard by using the same idea as in the reduction described above but by reversing the direction of the reduction and reducing TSP to the shortest superstring problem. To see this, note that for any cycle that starts at the source vertex, we have a corresponding permutation. The total weight of the arcs on such a cycle is exactly the length of the shortest superstring with overlaps removed. Thus, if we have an algorithm for finding the shortest superstring, then we can solve the TSP problem. Since constructing the graph only requires polynomial work this reduction is efficient. But we know that TSP problem is NP hard. We thus conclude that the shortest superstring problem is also NP hard.

Remark 5.11. In addition to designing algorithms, reductions can be used to prove that a problem is NP-hard or NP-complete. For example, if we reduce an NP-hard (NP-complete) problem A to another problem B by performing polynomial work, then B must be NP-hard (NP-complete).

5.7 Greedy Algorithm

We now consider our final algorithm-design technique, the "greedy" technique for solving the SS problem. When applying the greedy design technique, we consider the current solution at hand and make a greedy ***locally optimal*** decision to reduce the size of the problem. We then repeat the same process of making a locally optimal decision, hoping that eventually these decisions lead us to a global optimum. For example, a greedy algorithm for the TSP can visit the closest unvisited city (the locally optimal decision), removing thus one city from the problem.

The greedy technique is a heuristic that in some cases returns an optimal solution, but in many cases it does not. Nevertheless, greedy algorithms are popular because they tend to be simple. We note that, 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 key step in designing a greedy algorithm is to decide the locally optimal decision. In the case of the SS problem, observe that we can minimize the length of the superstring by maximizing the overlap among the snippets. Thus, at each step of the algorithm, 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.

The pseudocode for our algorithm is given in Algorithm 5.12. The algorithm relies on a function `overlap` (x, y) to compute the overlap of the strings x and y, and the function `join` (x, y), which places x after y and removes the maximum overlap. For example, `join` ("tagg", "gga") = "tagga". 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 x and y 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 T

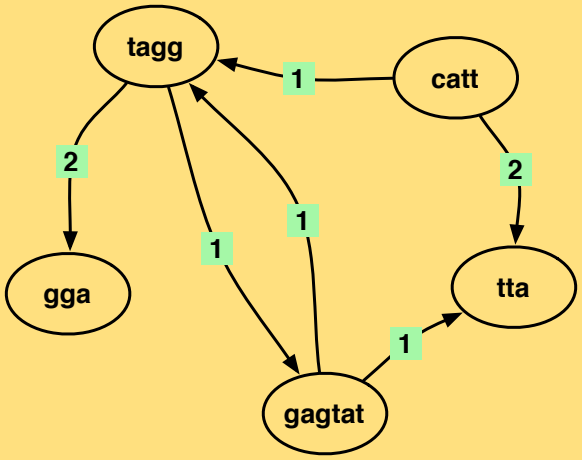
Algorithm 5.12. [Greedy Approximate SS]

```
greedyApproxSS S =
  if |S| = 1 then
    return x ∈ S
  else
    let
      T = { (overlap(x, y), x, y) : x ∈ S, y ∈ S, x ≠ y }
      (_, x, y) = arg max_{(o,_,_)∈T} o
      z = join(x, y)
      S' = S ∪ {z} \ {x, y}
    in
      greedyApproxSS S'
    end
```

is a set of triples each corresponding to an overlap and the two strings that overlap. The notation $\arg\max_{(o,\_,\_)\in T} o$ is mathematical notation for selecting the element of T that maximizes the first element of the triple, which is the overlap. After finding the pair (x, y) with the maximum overlap, the algorithm then replaces x and y with $z = \texttt{join}(x, y)$ in S to obtain the new set of snippets S'. The new set S' contains one element less than S. 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 set S. However, the superstring returned is not necessarily the shortest superstring.

Example 5.13. Consider the snippets in our running example,
$\{$ catt, gagtat, tagg, tta, gga $\}$.
The graph below illustrates the overlaps between different snippets. An arc from vertex u to v is labeled with the size of the overlap when u is followed by v. All arcs with weight 0 are omitted for simplicity.



Given these overlaps, the greedy algorithm could proceed as follows:

- join tagg and gga to obtain tagga,
- join catt and tta to obtain catta,
- join gagtat and tagga to obtain gagtatagga, and
- join gagtatagga and catta to obtain gagtataggacatta.

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. Let's analyze the work and the span of the greedy algorithm.

From the analysis of our brute-force algorithm, we know that we can find the overlaps between the strings in $O(m^2)$ work and $O(\lg m)$ span. Thus the $\arg\max$ for finding the maximum overlap can be computed in $O(m^2)$ work and $O(\lg m)$ span using a simple reduce. The other steps have less work and span. Therefore, not including the recursive call each call to `greedyApproxSS` costs is $O(m^2)$ work and $O(\lg m)$ span. Observe now that each call to `greedyApproxSS` reduces the number of snippets: S' contains one fewer element than S, so there are at most n calls to `greedyApproxSS`. These calls are sequential because one call must complete before the next call can take place. Hence, the total cost for the algorithm is $O(nm^2)$ work and $O(n \lg m)$ span. The algorithm is therefore highly parallel. There are ways to make the algorithm more efficient, but leave that as an exercise to the reader.

Since the `greedyApproxSS` algorithm does only polynomial work, and since the TSP prob-

lem is NP hard, we cannot expect it to give an exact answer on all inputs—that would imply **P** = **NP**, which is unlikely. Although `greedyApproxSS` does not return the shortest superstring, it returns a good 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; it is conjectured that the algorithm returns a string that is within a factor of 2. In practice, the greedy algorithm typically performs better than the bounds suggest. The algorithm also generalizes to other similar problems. Algorithms such as `greedyApproxSS` that solve an NP-hard problem to within a constant factor of optimal, are called ***constant-factor approximation algorithms***.

Remark 5.14. 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 feature of this change 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.

5.8 Problems

5-1 String concatenation
You are given a set of strings $s_1, \ldots, s_n$ and a target string t.

- Use the brute-force technique to come up with an algorithm that gives you a subset of the strings and the specific order in which they can be concatenated to obtain the target string.
- Analyze the work of your algorithm.
- Analyze the span of your algorithm.

5-2 Compact string concatenation
You are given a set of strings $s_1, \ldots, s_n$ and another target string t. Use the brute-force technique to come up with an algorithm that gives you a subset of the strings and the specific order in which they can be concatenated while removing overlaps to obtain the target string. For example if you concatenate ”parallel” and ”elision”, you will get ”parallelision”. Analyze the work and the span of your algorithm.

5-3 Start positions of snippets
Consider the set of snippets for a genome sequence, and let s be a superstring of the snippets, such that each snippet is contained in the superstring as substring. Prove than no two snippets can start at the same position in s.

5-4 End positions of snippets
Consider the set of snippets for a genome sequence, and let s be a superstring of the snippets, such that each snippet is contained in the superstring as substring. Prove than no two snippets can end at the same position in s.

5-5 Substring check
Given a string s of length n, and a string t of length m: design an algorithm that checks whether s is a substring of t, and analyze the work and span of your algorithms.

5-6 Hamiltonian paths on complete graphs
Prove or disprove: a Hamiltonian path on a complete directed graph corresponds to a permutation of vertices. A permutation of vertices corresponds to a Hamiltonian path.

5-7 Hamiltonian cycle with start
Prove using the reduction technique that if the TSP problem in NP hard, then so is the problem of finding the shortest Hamiltonian cycle that starts and ends at a specified vertex.

5-8 Contained Strings
In the greedy algorithm `greedyApproxSS` (Algorithm 5.12), we remove x, y from the set of strings but do not remove any strings from s that are contained within xy = `join` (x, y). Argue why there cannot be any such strings.

5-9 Correctness of greedy approximation
Prove that algorithm `greedyApproxSS` (Algorithm 5.12) returns a string that is a superstring of all original strings.

5-10 Exactness of greedy approximation
Give an example input for which `greedyApproxSS` (Algorithm 5.12) does not return the shortest superstring.

5-11 Improved greedy
Improve the greedy algorithm's cost bounds by presenting a more efficient implementation.

5-12 Generous grandmother
Your rich grandmother enjoys collecting precious items, such as jewelry and gold-plated souvenirs. When you visit her for Spring break (instead of going to Cancun), she is very pleased and decides to reward you. She gives you a bag and a scale and instructs you to take anything you want as long as the bag does not hold any more than 10 pounds. Delighted by the surprising (based on your parents' stories of their childhood) generosity of your grandmother, you also realize that your grandmother forgot to take the price tags off the items, which gives you an idea about the value of these items.

- Design a brute force algorithm for selecting the items to take away with you. Is your algorithm optimal?
- What is the work and span of your brute-force algorithm?
- Design a greedy algorithm for selecting the items to take away with you. Why is your algorithm greedy?
- Is your greedy algorithm optimal?
- What is the work and span of your greedy algorithm?