Lecture 1 — Overview and Sequencing the Genome

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — Sunday, January 12, 2014

1 Administrivia

Welcome to 15-210 Parallel and Sequential Data Structures and Algorithms. This course will teach you methods for designing, analyzing, and programming sequential and parallel algorithms and data structures. The emphasis will be on fundamental concepts that will be applicable across a wide variety of problem domains, and transferable across a broad set of programming languages and computer architectures. *There is no textbook for the class.* We will (electronically) distribute lecture notes and supplemental reading materials as we go along. We will try to generate a draft of lecture notes and post them before classes.

The syllabus for the course is available on the Blackboard page for the course. Please read it very carefully, especially the policy on collaboration.

You will likely find this course to be difficult. There are several reasons for why. First, the material covered in the course, with emphasis on parallelism, will be new to many of you. Second, the way we design our algorithms and implement them, with emphasis on higher-order programming (where functions are first-class values), can be difficult to grasp quickly, though over time you will likely not be able to imagine thinking without them. Third, our assignments will involve programming in Standard ML (SML).

It is thus important for you to mentally prepare yourself for a difficult course. If you do your work, we are confident that you will finish this class with a satisfactory grade. Whenever needed, the TAs and the instructor will be happy to help you with any questions you may have. However we would like to reiterate that there is no substitute for doing your own work.

How to learn well. As we said in the class, we will be making available lecture notes and the slides we use in the classes. Some of you would be very comfortable with just reading these carefully. Some of you would feel comfortable by taking additional notes during the lectures. That is perfectly fine and if you do, we suggest you take it alongside or on the lecture notes so that you have the context. We also suggest that as you read the course notes, write down a list of questions and try to solve them and develop a habit of formulating variants of exercises and homeworks that you are given and solving them and coming up with new questions. Finally, if you are not used to doing these, we would recommend that you start working on them and being patient.

Since this is still an early incarnation of 15-210, we would appreciate feedback any time. Please come talk to us if you have suggestions and/or concerns. If you find any errors or confusing explanation please let us know so we can improve these notes.

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

2 Course Overview

This course aims to cover the algorithms and data structures, with an emphasis on parallelism.

Question 2.1. *Why should we care about parallelism?*

There are many reason for why parallelism is important. Fundamentally, parallelism is simply more powerful than sequential or serial computation where there is only one line of computation. In parallel computation, we can perform multiple tasks at the same time. Another reason is efficiency in terms of energy usage. As it turns out performing a computation twice as fast sequentially requires eight times as much energy. Precisely speaking, energy consumption is a cubic function of frequency (speed). With parallelism, we don't need more energy to speed up a computation, at least in principle. For example, to perform a computation in half the time, we need two lines of computation (instead of one) that ran half the time of the sequential, thus consuming the same amount of energy. In reality, there are some overheads and we will need more energy, usually only a constant fraction more. These two factors—power and energy—has gained importance in the last decade catapulting parallelism to the forefront of computing.

Parallel hardware. Today, it is nearly impossible to avoid parallelism. For example, when you do a simple web search, you are engaging a data center in some part of the world (likely near your geographic location) that houses thousands of computers. Many of these computers (perhaps as many as hundreds, if not thousands) take up your query and sift through data to give you an accurate response as quickly as possible. This form of parallelism may be viewed as large-scale parallelism, as it involves a large number of computers.[1]

Another form of parallelism involves much smaller numbers of processors. For example, portable computers today have chips that can have as many as 10 processors. Such processors, sometimes called *multicore chips*, are predicted to spread and provide increasing amount of parallelism over the years. For example, using current chip technology, it is not difficult to put together several multicore chips in a desktop machine to include 60 cores. While multicore chips were initially used only in laptops and desktops, they are also becoming used common in smaller mobile devices such as phones due to their low-energy consumption (many mobile phones today have 4- or 8- core chips.)

In addition to the aforementioned parallel systems, there has been much interest in developing hardware for specific tasks. For example, Graphics Processing Units (GPUs) can fit as much as 1000 chips onto a single unit. Intel's Mic (or Xeon Phi) architecture can host several hundred processors on a single chip.

Question 2.2. *Can you think of consequences of these developments in hardware?*

Parallel thinking. These developments in hardware make the specification, the design, and the implementation of parallel algorithms a very important topic. Parallel computing requires a somewhat

[1] Of course, terms such as "large" are relative by definition. What we call large-scale today may be consider small scale in the future.

different way of thinking than sequential computing. Developing the intellectual skills for *parallel thinking* is an important goal of this class.

Question 2.3. *What is the advantage of using a parallel algorithm instead of a sequential one?*

Parallel software. The most important advantage of using a parallel instead of a sequential algorithm is the ability to perform sophisticated computations quickly enough to make them practical. For example without parallelism computations such as Internet searches, realistic graphics, climate simulations would be prohibitively slow. One way to quantify such an advantage is to measure the performance gains that we get from parallelism. Here are some example timings to give you a sense of what can be gained. These are on a 32 core commodity server machine (you can order one on the Dell web site).

	Serial	**Parallel**	
		1-core	32-core
Sorting 10 million strings	2.9	2.9	.095
Remove duplicates 10M strings	.66	1.0	.038
Min spanning tree 10M edges	1.6	2.5	.14
Breadth first search 10M edges	.82	1.2	.046

In the table, the serial timings use sequential algorithms while the parallel timings use parallel algorithms. Notice that the speedup for the parallel 32 core version relative to the sequential algorithm ranges from approximately 12 (min spanning tree) to approximately 32 (sorting). Currently, obtaining such performance requires developing efficient and performant parallel algorithms and highly tuned implementations. In this course, we will focus on the first challenge.

Challenges of parallel software. The many forms of parallelism, ranging from small to large scale, and from general to special purpose, currently requires many different languages, libraries, and implementation techniques. For example, it is unlikely to obtain the kinds of speedups that we discussed above from unoptimized software. This diversity of hardware and software makes it a challenge 1) to develop parallel software and 2) to learn and to teach parallelism. For example, we can easily spend weeks talking about how we might optimize a parallel sorting algorithm for a specific hardware.

This course: parallelism. Maximizing speedup by highly tuning an implementation is not the goal of this course. In this course, we aim to cover the general design principles for parallel algorithms that can be applied in essentially all parallel systems, from the data center to the multicore chips on mobile phones. We will learn to think about parallelism at a high-level, learning general techniques for designing parallel algorithms and data structures, and learning how to approximately analyze their costs. As in 15-150, in this class we will be using *work* and *span* to analyze costs.

The algorithms that we design will mostly be purely functional.

Question 2.4. *Can you think of reason for why purely functional programming can help in designing and implementing parallel algorithms?*

One reason for why purely functional programming can help in the design and implementation of parallel algorithms is that purely functional programs are safe for parallelism: they can be executed in parallel without any modifications. In an imperative setting, we need to worry about race conditions since parallel threads of execution might modify shared states in different orders from one run of the code to the next. Race conditions makes it much harder to think in parallel and reason about the correctness and the efficiency of parallel algorithms. Another reason is that functional languages (even when imperative) enable expressing computations succinctly and effectively by using higher-order features. Thinking in higher order functions encourages working at a higher level of abstraction, moving us away from the one-at-a-time (loop) way of thinking that is detrimental to parallelism.

Even though the algorithms that we design are purely functional, this does not mean that they cannot be implemented in imperative languages—one just needs to be much more careful when coding imperatively. Some imperative parallel languages, in fact, encourage programming purely functional algorithms. The techniques that we describe thus applicable in the imperative setting as well.

This all being said, most of what is covered in a traditional algorithms course will be covered in this course, but perhaps in a somewhat different way.

This course: specification and implementation. In this course, we will carefully distinguish between interfaces/specifications and design/implementation.

An *interface* or *specification* defines precisely what we want of a function or a data structure. A *design* or an *implementation* describes how to meet the specification. In other words specifications and designs refer to the *what* and the *how*. What we want a function or data structure to achieve and how to do that.

	Interface (specification)	**Implementation (design)**
Functions	Problem	Algorithm
Data	Abstract Data Type	Data Structure

Functions and data. In computing, is is broadly possible to distinguish between *functions* that perform actual computation and *data* which serve as the subject of computation. We can thus distinguish between specifying and implementing the behavior of functions and data (Table 2).

A *problem* specifies precisely the intended input/output behavior—a function—in an abstract form. It is an abstract (precise) definition of the problem but does not describe how it is solved.

An *algorithm* enables us to solve a problem; it is an implementation that meets the specification. Typically, a problem will have many algorithmic solutions.

Example 2.5. *For example, the sorting problem specifies what the input is (e.g., a sequence of numbers) and the intended output (e.g., an ordered sequence of numbers); the* `quicksort` *and insertion sort are algorithms for solving the sorting problem.*

Similarly an *abstract data type* (ADT) specifies precisely an interface for operating on data in an abstract form. It does not specify how the data is structured.

A *data structure* implements the interface by organizing the data in a particular. For an ADT, the interface is specified in terms of a set of operations on the type.

Example 2.6. *For example, a priority queue is an ADT with operations that might include* `insert`*,* `findMin`*, and* `isEmpty?`*. Various data structures can be used to implement a priority queue, including binary heaps, arrays, and balanced binary trees.*

The terminology ADTs vs. data structures is not as widely used as problems vs. algorithms. In particular, sometimes the term data structure is used to refer to both the interface and the implementation. We will try to avoid such usage in this class.

Question 2.7. *Why do we need to distinguish between interfaces and implementations.*

There are several critical reasons for keeping a clean distinction between interface and implementation. One reason is to enable proofs of correctness, e.g. to show that an algorithm properly implements an interface. Many software disasters have been caused by badly defined interfaces. Another reason is to enable reuse and layering of components. One of the most common techniques to solve a problem is to reduce it to another problem for which you already know algorithms and perhaps already have code. We will look at such an example today. A third reason is that when we compare the performance of different algorithms or data structures it is important that we are not comparing apples with oranges. We have to make sure the algorithms we compare are solving the same problem, because subtle differences in the problem specification can make a significant difference in how efficiently that problem can be solved.

For these reasons, in this course we will put a strong emphasis on defining precise and concise interfaces and then implementing those abstractions using algorithms and data structures. When discussing solutions to problems we will emphasize general techniques that can be used to design them, such as divide-and-conquer, the greedy method, dynamic programming, and balance trees. It is important that in this course you learn how to design your own algorithms/data structures given an interface, and even how to specify your own problems/ADTs given a task at hand.

3 An Example: Sequencing the Genome

As an example of how to define a problem and develop parallel algorithms that solve it we consider the task of sequencing the Genome. Sequencing of a complete human genome 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

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

Question 3.1. *Why do you think sequencing the genome is a difficult problem?*

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 approximately few billion in the whole genome. We therefore resort to cutting strands into shorter fragments and then reassembling the pieces.

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 about a way to parallelize primer walking?*

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

Exercise 1. *When cut, the strand* cattaggagtat *might turn into,* ag, gag, catt, tat, *destroying the original ordering.*

Question 3.3. *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 a fragments, we lose all the information that we have about how to connect them back. If we had some information about how to relate different pieces, we can imagine solving this problem just as we solve a jigsaw puzzle (by relying on our "meta" knowledge about what the whole picture that we are trying to reconstruct is).

Figure 1: A jigsaw puzzle.

Question 3.4. *Can you think of a way to relate different pieces?*

We can relate different pieces, if we could make copies of the original sequence and generate many fragments. When a fragment overlaps with two other, it can tell us how to relate them. This is the idea behind the shotgun (sequencing) method, which today seems to be the standard technique for genome sequencing today. Shotgun method works as follows:

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

Example 3.5. *For example, for the sequence cattaggagtat, we produce three copies:*

cattaggagtat
cattaggagtat
cattaggagtat

We then divide each into fragments

catt	*ag*	*gagtat*	
cat	*tagg*	*ag*	*tat*
ca	*tta*	*gga*	*gtat*

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

Steps 1–3 are done in a wet lab. Algorithms come into play in Step 4.

Question 3.6. *In step 4, is it always possible to reconstruct the sequence?*

It is not always possible to reconstruct the exact original genome in step 4. For example, we might get unlucky and not be able to reverse a cut. But also there are many DNA strings that lead to the same collection of fragments. 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.

Defining the problem. We will therefore do our best is constructing the original sequence.

Question 3.7. *How can we make this intuitive notion of "doing our best precise"?*

It is not easy to make this notion of "doing our best" precise. This is why, it can be as difficult and important to formulate a problem as it is to solve it.

Question 3.8. *Can you think of a property that the result needs to have in relation to the snippets?*

Note that since the fragments all come from the original genome, the result should contain all snippets. In other words, it is a *superstring* of the fragments.

Now of all the superstrings, which one should we pick? We can take one more step in making the problem more precise by constructing the "best" superstring. How about the shortest superstring? This would give us the simplest solution, which is often desirable. This is how we will define the problem.

Definition 3.9 (The Shortest Superstring (SS) **Problem**)**.** Given an alphabet set Σ and a set of finite strings $S \subseteq \Sigma^+$, return a shortest string r that contains every $s \in S$ as a substring of r.

In this definition the notation Σ^+, the "Kleene plus", means the set of all possible non-empty strings consisting of symbols from Σ. Note that in this definition, we require each $s \in S$ to appear as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg". That is, given sequence fragments, construct the shortest string that contains all the fragments. The idea is that the simplest string is the best.

Having specified the problem, we are ready to design an algorithm for solving it. Let's start with a few observations.

Observation 1: Snippets. Note that we can ignore strings that are contained in other strings. That is, for example, if we have gagtat, ag, and gt, we can throw out ag and gt. We will refer to the fragments that are not contained in others as *snippets*.

Example 3.10. *In our example, we had the following fragments.*

Our snippets are now:

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

Observation 2: defining a solution. When designing an algorithm, an important questions is how to define the result.

Question 3.11. *Consider the result sequence, can two snippets start at the same position in the result sequence?*

Note that two snippets cannot start at the same position, because otherwise there would be a string that is a substring of another. Since they start at distinct positions, we can totally order the strings in S. This ordering thus defines a solution.

We are now ready to solve the 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. 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.

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.

Question 3.12. *Before we start looking at algorithms let's think about why this problem may be hard. Consider jigsaw puzzles. What makes them "hard"?*

4 Algorithm Design Technique 1: Brute Force

Brute-force technique simply consist of trying all candidate solutions and selecting the best. It is often not efficient, because there can be many candidates. It can, however, be the only known way to solve a problem.

Definition 4.1. Enumerate all possible candidate solutions for a problem, score each candidate, and return a best solution.

Question 4.2. *What are the candidate solutions for the genome sequencing problem?*

When sequencing the genome, recall that we are searching for the shortest superstring. Since we know that each result consist of snippets that start at a unique position and since we don't need to duplicate snippets, the result is simple a permutation of all the snippets with overlaps removed. Thes when sequencing the genome with the brute-force technique, candidate solutions consist of such permutations.

Question 4.3. *How can we score each permutation?*

To score each permutation, we can use the length of the permutation after removing the overlaps.

Example 4.4. *For our running example, the permutation*

catt tta tagg gga gagtat

gives us cattaggagtat *after removing the overlaps (the excised parts are underlined). The score for this permutation is 12. Note that this result happens to be the original string and is also the shortest superstring.*

Exercise 2. *Try a couple other permutations and determine the length after removing overlaps.*

Question 4.5. *Does trying all permutations always give us the shortest superstring?*

As our intuition might suggest, by trying all permutations, we can indeed find the shortest superstring. The proof of this intuiton hints at an algorithm that we will look at in a moment.

Lemma 4.6. *Given a finite set of finite strings $S \subseteq \Sigma^+$, the brute force method finds the shortest superstring.*

Proof. Let r^* be any shortest superstring of S. We know that each string $s \in S$ appears in r^*. Let i_s denote the beginning position in r^* where s appears. Since we have eliminated duplicates, it must be the case that all i_s's are distinct numbers. Now let's look at all the strings in S, $s_1, s_2, \ldots, s_{|S|}$, where we number them such that $i_{s_1} < i_{s_2} < \cdots < i_{s_{|S|}}$. It is not hard to see that the ordering $s_1, s_2, \ldots, s_{|S|}$ gives us r^* after removing the overlaps. □

Question 4.7. *Is the brute-force approach a good parallel algorithm?*

The approach of trying all permutations is easy to parallelize. Each permutation can be tested in parallel and it is also easy to generate all permutations in parallel. To understand whether this is a

good parallel algorithm or not, we have to consider the work and the span.

Although highly parallel, the brute-force algorithm has to examine a very large number of combinations, resulting in too much computational work. In particular, there are $n!$ permutations on a collection of n elements. This means that if the input consists of $n = 100$ strings, we'll need to consider $100! \approx 10^{158}$ combinations, which for a sense of scale, is more than the number of atoms in the universe. As such, the algorithm is not going to be feasible for large n.

Question 4.8. *Can we come up with a smarter algorithm that solves the problem faster?*

Unfortunately, we don't know if we can give an asymptotically faster algorithm for this problem, because the problem is NP-hard.

Question 4.9. *Is there no way to 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 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.

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) in practice do even better than the bounds suggest.

5 Algorithm Design Technique 2: Reduction

Another approach to solving a problem is to reduce it to another problem which we understand better and for which we know algorithms. 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 an NP-complete problem A can be reduced to problem B with polynomial work, 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 5.1. *Are you all familiar with the TSP problem?*

This is a canonical NP-hard problem dating back to the 1930s and has been extensively studied, e.g. see Figure 2. 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.

Figure 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.

Definition 5.2 (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 5.3. *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 5.4. *For "tagg" and "gga", we have* $\texttt{overlap}$ *("tagg","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 Λ 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, $|\text{"gga"}| - \texttt{overlap}(\text{"tagg"}, \text{"gga"}) = 3 - 2 = 1$.
- The weights for arcs incident to Λ are set as follows: $(\Lambda, s_i) = |s_i|$ and $(s_i, \Lambda) = 0$. That is, if s_i is the first string in the permutation, then the arc (Λ, s_i) pays for the whole length s_i.

To see this reduction in action, the input $\{\text{catt}, \text{acat}, \text{tta}\}$ results in the graph in Figure 3.

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. In figure 4, we have two Hamiltonian cycles in this graph, the left with length 10 corresponding to `cattacatta` and the right with length 8, corresponding to `cattacat`:

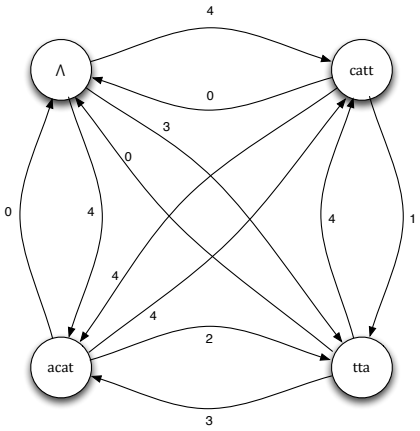

Figure 3: Asymmetric instance reduced from the SSSP instance.

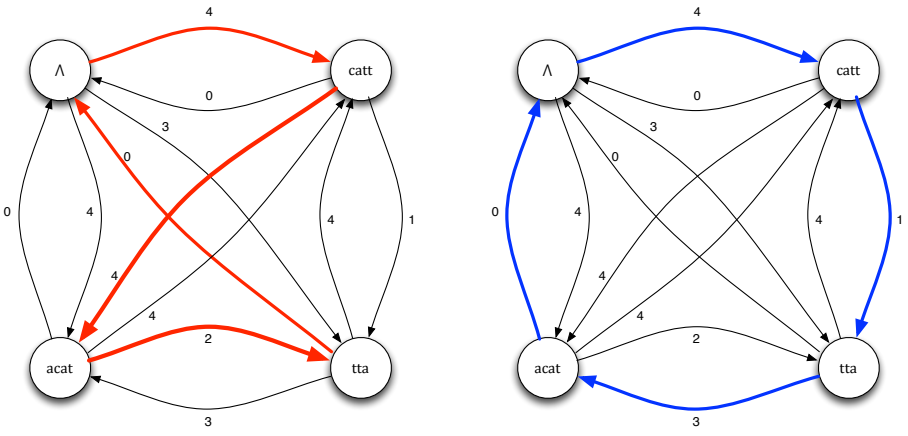

Figure 4: Two Hamiltonian paths in the graph.

Question 5.5. *What does a Hamiltonian cycle in the graph starting at the root corresponds 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 5.6. *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.

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 TSP, 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 this helps.

6 Algorithm-Design Technique 3: Greedy

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

Definition 6.1 (The Greedy **Technique**)**.** Given a sequence of steps, on each step make a locally optimal decision based on some criteria without ever backtracking on previous decisions.

Question 6.2. *Does the greedy technique always return the optimal solution?*

The greedy technique (or approach) is a heuristic that when applied to many problems does not necessarily return an optimal solution. In our case the greedy algorithm indeed is not guaranteed to find the shortest superstring but we can guarantee that it gives a good approximation, and furthermore it works very well in bounds on how close it is to the optimal, and it works very well in practice. Greedy algorithms are popular because of their simplicity.

Pseudo Code 6.5 (Greedy Approximate SS Algorithm).

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

Figure 5: A Greedy Algorithm for the Shortest Superstring (SS) Problem.

Remark 6.3. You can think of a greedy algorithm as taking a walk in a hilly terrain where each step in taken in the direction of the steepest slope in the hope that it will lead to the highest peak. Thus, if you start at a place close to the summit and all peaks are close to each other in terms of elevation, then you will find peak that is guaranteed to have an altitude that is close to the summit.

Question 6.4. *Considering that we want to minimize the length of the result, what should our "greedy choice" be?*

To minimize the length of the string, we should maximize overlap. Thus our algorithm will simple pick a pair of snippets with the largest overlap and join them by appending them and removing the overlap.

To describe the greedy algorithm, we'll define a function $\texttt{join}(s_i, s_j)$ that appends s_j to s_i and removes the maximum overlap. For example, $\texttt{join}(\text{“tagg”}, \text{“gga”}) = \text{“tagga”}$.

Figure 5 shows pseudocode for our greedy algorithm. In this course the pseudocode we use will be purely functional and easy to translate into ML code or code for just about any functional language. In fact it should not be hard to translate it to imperative languages, especially if they support higher-order functions. Primarily, the difference from ML is that we will use standard mathematical notation, such as subscripts, and set notation (e.g. $\{f(x) : x \in S\}$, $\cup$, $|S|$).

Given a set of strings S, the `greedyApproxSS` algorithm finds the pair of strings s_i and s_j in S that are distinct and have the maximum overlap—the `maxval` function takes a comparison operator (in this case comparing the first element of the triple) and returns a maximum element of a set (or sequence) based on that comparison. The algorithm then replaces s_i and s_j with $s_k = \texttt{join}(s_i, s_j)$ in S.

Question 6.6. *Is the algorithm guaranteed to terminate?*

Note that 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.

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

Exercise 5. *Give an example input S for which* `greedyApproxSS` *does not return the shortest superstring.*

Exercise 6. *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*. It also seems that the relationship between the asymmetric TSP and SSSP is still an active research area: a recent result [2] presents a 2.37–approximation algorithm for SSSP based on approximate ATSP.

[2] Katarzya Paluch, "Better Approximation Algorithms for Maximum Asymmetric Traveling Salesman and Shortest Superstring", `http://arxiv.org/abs/1401.3670`, Jan 2014

Remark 6.7. 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.