

Connections: Computer Science and Molecular Biology

Anne Condon
condoncs.ubc.ca

May 29, 2006

1 Caveats

This is a first draft of notes to accompany the CS & Molecular Biology module of CPSC101/WMST201: Connecting with Computer Science. Feedback is very welcome! Send comments to Anne Condon (condoncs.ubc.ca).

Ultimately, the goals of this module are to introduce ways in which computer science influences and advances biology, and vice versa. We want to look more deeply than just the use of computer tools by biologists. The bias is towards examples that serve to introduce interesting ideas from computer science - the emphasis would be very different if these notes were written by a biologist! The challenge is to provide real insight in just a few pages, without overly compromising discussion of the chosen example. Our approach is to “cut to the chase” by providing a minimum of biological background and motivation, and then to focus in on the computational core of a compelling biological problem. We try to keep biological jargon to a minimum.

If you are knowledgeable in biology, you will probably think about many interesting details ignored in our discussion. I encourage you to think how the solution to the problem discussed here might be adapted to better model biological aspects of the problem. If you are not knowledgeable in biology, rest assured that you can follow, and hopefully enjoy the module anyway. And, the computer science concepts that are covered are sufficiently rich that they apply in other, non-biological contexts, too.

In the near future, this module will be expanded to include a “hands-on” component, providing exposure to genomic databases and perl scripting, as well as some profiles of scientists who bring an in-depth knowledge of both computer science and biology to bear on their work.

2 Introduction

The publication of the first draft of the human genome in 2001 was a landmark event for biology. For well over a century, we have known that many of our physical and behavioural traits are inherited from our parents. That inheritance is scripted in our genes and stored in each and every one of our cells. Knowledge of our genetic makeup not only tells us more about ourselves and our evolutionary heritage, but also leads to better diagnosis and management of disease. As we discuss below, the great accomplishment of the human genome project was to bring our genetic script to light.

The human genome project also illustrates the central role of computer science in enabling biological advances. Sequencing the genome would not have been possible without sophisticated database systems, computer algorithms, visualization tools, high-performance clusters of computing machines, and support for international collaboration. Because of its enormous impact on biology, and because it provides a rich context in which to illustrate computer science concepts, we use the human genome project as our focus in these notes.

Following a little background on DNA and proteins in Section 3, we focus on a key step of the overall process of gene sequencing in Section 4. In this step, called *fragment assembly*, the sequences of short gene fragments are pieced, or assembled, back together, by computer algorithms, to reveal whole genes.

The connections between computer science and biology extend well beyond the application of computer methods in understanding our genome and the workings of our cells. Other parts of this course touch on different connections, relating to the complexity of patterns in nature, biologically inspired methods for solving computational problems, and computational processes in the brain.

3 DNA, proteins, genes

An adult human body has trillions of cells. Each cell is a hive of activity, in which proteins are the actors. Not only do proteins keep the inner workings of the cell in good order, but they digest food, coordinate the transport of nutrients around the body, signal thoughts in our brain, and much more.

A *protein molecule* is a chain of basic molecular units, called amino acids, strung together like tiny beads on a long necklace. Twenty types of amino acids occur commonly in living organisms. Many proteins quickly wear protein

themselves out, and so the basic process of life requires cells to constantly make new proteins. Thus, nature needs a medium in which to record the composition of the needed proteins, and also to pass along this information to our offspring. This medium is DNA.

A *DNA molecule*, often called *DNA strand*, is also a sequence of molecular units. DNA's units are of four types, called A, C, G, and T (which denote Adenine, Cytosine, Guanine, and Thymine). Just as a word has a left and right end, or a worm has a head and tail, so too does a DNA strand have two chemically distinct ends, called the 5' and 3' (pronounced 5 prime and 3 prime) ends. When we describe a molecule using its letters, for example, AGGTC, the convention is that the left end, as written is the 5' end. So, the molecule AGGTC is different from the molecule CTGGA. Chemically, DNA is a very stable molecule compared, say, with proteins, which makes it well suited for information storage in the cell. However, with our computer science hats on, we will appeal to the advantages of abstraction and not concern ourselves further with the chemical details of DNA's units. Instead, we will view DNA simply as a digital information storage medium. In each and every one of our cells are about three billion units of DNA. To store the raw information in the DNA of all of the cells of an adult human would require trillions of CD's!

Billions of years before we could contemplate the beauty of it all, nature somehow invented a simple and elegant system that uses DNA to specify which proteins should be manufactured in the cell. This system is called the *genetic code*. The same system is used in every known living organism (with minor variations in a small number of microscopic organisms). In the genetic code, three-unit segments of DNA, called codons, are used to specify one protein. Note that this encoding principle is essentially the same as the ASCII code system used to encode characters of text using eight "0" and "1" bits (see Chapter 8 of the text). The usage of the word "code" here is *not* intended to suggest computer code, or lists of instructions, but rather a system for information representation. Table 1 gives the details of the genetic code. Notice the redundancy in the code - as many as six codons can specify the same amino acid.

Exercises:

- 3.1 Why do you think the genetic code uses three DNA units per codon? Why not one or two units per codon? (How many codons would there be if there were two units per codon?) Why not four or more units per codon?

TTT phenylalanine	TCT serine	TAT tyrosine	TGT cysteine
TTC phenylalanine	TCC serine	TAC tyrosine	TGC cysteine
TTA leucine	TCA serine	TAA stop	TGA stop
TTG leucine	TCA serine	TAG stop	TGG tryptophan
CTT leucine	CCT proline	CAT histidine	CGT arginine
CTC leucine	CCC proline	CAC histidine	CGC arginine
CTA leucine	CCA proline	CAA glutamine	CGA arginine
CTG leucine	CCG proline	CAG glutamine	CGG arginine
ATT isoleucine	ACT threonine	AAT asparagine	AGT serine
ATC isoleucine	ACC threonine	AAC asparagine	AGC serine
ATA isoleucine	ACA threonine	AAA lysine	AGA arginine
ATG methionine	ACG threonine	AAG lysine	AGG arginine
GTT valine	GCT alanine	GAT aspartic acid	GGT glycine
GTC valine	GCC alanine	GAC aspartic acid	GGC glycine
GTA valine	GCA alanine	GAA glutamic acid	GGA glycine
GTG valine	GCG alanine	GAG glutamic acid	GGG glycine

Table 1: The genetic code. Each entry in the table lists a codon - three capital letters representing DNA units - and either an amino acid, or the word “stop”. For example, TCT specifies the amino acid serine. The stop codons indicate to the cell’s translational machinery that the end of the gene has been reached. The code is *redundant* meaning that more than one codon can specify the same amino acid. Thus, the protein fragment Methionine-Isoleucine-Phenelalanine-Aspartic Acid-Glycine is coded by ATGATCTTTGACGGG and also by ATGATTTTTGATGGT.

A *gene* contains a sequence of codons, and thus specifies a protein. To manufacture a protein, molecular machinery in the cell must somehow translate the code of a DNA gene into the corresponding sequence of amino acids. genes

The above description ignores many details. For example, you have likely seen the elegant double helix structure of DNA, which is formed from two entwined strands. DNA in this form is called double-stranded DNA. Abstractly, in terms of information content, it turns out that double stranded DNA is no different from the single-stranded DNA described above, so in the interests of clarity we will stick with single-stranded DNA. The important points of our discussion will be just as relevant to double-stranded DNA. As another example, interspersed both within and between some genes, particularly human genes, is a lot of so-called “junk” DNA. No-one is sure what purpose much of this junk DNA serves, but its existence makes it hard to identify the actual genes. The word *genome* is typically used to refer both to the genes and to the junk DNA. In fact, the genome looks somewhat like a badly-edited document that evolved over time, with quick fixes and patches introduced as needed to adapt to changing needs, and whole sections awkwardly crossed out. If your English instructor were grading for well-organized, clear, concise writing, probably the human genome would get a “D”. Nevertheless, the fact that you can read and understand these notes shows that the code works well enough most of the time. genome

With current technologies, it is now possible to make significant progress in finding answers to the following questions: What are our genes? What are our proteins? What do those proteins do? The human genome project was a major investment, a first step towards answering these questions. Much of the research focus to date has been on genes known to be implicated in diseases such as cancer, since knowledge of such genes results in more accurate diagnosis, and in some cases, more effective treatment. In addition, genes of other organisms have also been sequenced. For example, genes of bacteria and viruses provide valuable information that can help in development of a vaccine. Genes of important crops provide insight on how the crops are naturally resistant to disease. Also, by comparing DNA across many organisms, it becomes easier to distinguish genes from “junk” - since the DNA sequences of genes tend to be more highly conserved across organisms - and to understand evolutionary relationships among organisms.

In the next section, we’ll look at how long strands of DNA are actually sequenced.

4 Genome sequencing

Imagine you are in a lab and have managed to extract a piece of DNA from your favourite organism - or perhaps from one of your own cells. It is a long, gelatinous strand - let's say about a million units in length. We'll call it our *target* strand. Let's suppose that you have several copies of this strand at your disposal. How are you going to figure out the sequence of one million A's, C's, G's, and T's in this mess?

It's not at all clear why a computer will be of any help. Perhaps it's better to first find out what technologies biologists can bring to the table. Indeed, biologists have developed very clever laboratory methods for sequencing DNA. Unfortunately, these methods only work on short DNA fragments, on the order of hundreds of units. But, biologists also know how to chop DNA strands up into short fragments, of the size that their sequencing machines can handle. Great, you say - let's just chop our strand up, feed the pieces into the sequencing machine, and we're done.

There are some problems with this. One is that the sequencing methods are somewhat error-prone. A bigger problem, however, is that biologists don't have a lot of control over their chopping mechanism. To keep this discussion simple, let's just say that the points at which the target DNA gets chopped are quite random, but in such a way that the fragments between chopping points are usually short enough to be handled by a sequencing machine. Moreover, there is no way to keep track of how the resulting fragments are ordered in the target strand. So, you'll end up with the sequences of hundreds of thousands of DNA fragments, but no way to piece them together. It's as if someone chopped up the great works of Confucius into tiny snippets, and asked you to recreate the original works. And you don't even know how to read ancient Chinese script. Sounds hopeless.

But you are not the type to give up easily. When faced with a challenging problem, it sometimes helps to think concretely about a simple example, and explore solutions for this example. We'll imagine that the target strand is now much, much shorter, so we can easily write it down - say between 20 and 30 units long, and your sequencing machine can sequence strands between 6 and 10 units long. (Strands which are too short or too long are not sequenced at all.) We'll also assume that the sequencing machine never makes errors. You have at hand two copies of the target strand. You chop them into fragments, and sequence the fragments that are of suitable length. You get back three fragments, say CAAGACCAA, TTACCGGCC, and CAACAAATTA. Call these fragments X , Y , and Z in that order.

You notice that the ends of the fragment Z *match* ends of X and Y . For

example, CAA at the left end of Z matches CAA at the right end of X . We call CAA the $X \rightarrow Z$ -overlap (the “ $S \rightarrow Y$ ” notation is intended to convey that the overlap is at the right end of X and the left end of Y). Similarly, TTA is the $Z \rightarrow Y$ -overlap. Figure 1 makes precise some useful concepts, such as overlaps, which are used in the following discussion.

We could align the fragments so that these matching parts line up:

```

CAAGACCAA
  CAACAAATTA
    TTACCGGGCC

```

If we read off the units left to right, we get CAAGACCAACAAATTACCGGGCC. Call this strand A . A is an *assembly* of the three fragments X, Y, Z , obtained by aligning the fragments by starting with X , then aligning Z so that the $X \rightarrow Z$ -overlap at the right end of X and the left end of Z line up, and finally aligning Y so that the $Z \rightarrow Y$ -overlap at the right end of Z and the left end of Y line up. Note that A contains X, Y , and Z as substrands (why?). A is one possibility for the target, since the three fragments X, Y , and Z could plausibly have been obtained from two copies of S .

The above example illustrates why it is useful to obtain fragments from several copies of your target gene. The randomness inherent in the chopping mechanism is actually helpful, since fragments are likely to overlap, and these overlaps may provide valuable clues as to how the fragments can be assembled to reveal the target.

Of course, there are other plausible ways to piece together the fragments – see Figure 1. We need to make an intelligent guess as to which one is most likely to be the true target. One criterion to use in selecting the “winner” is the length of the resulting strand: the shorter, the better. Roughly, the rationale for using this criterion is that, the more overlap there is between two strands, the more evidence we have that they really are from the same part of the original DNA strand. The shorter the assembly, the more overlap there must be between fragments in the assembly.

Exercises:

- 4.1 Suppose that one strand W in a collection \mathcal{C} of fragments is a substrand of another. (For example, suppose the list is $X = \text{CAAGACCAA}$, $Y = \text{TTACCGGGCC}$, $Z = \text{CAACAAATTA}$, and $W = \text{CAAAT}$; here, W is a substrand of Z .) A shortest assembly of the overall collection \mathcal{C} is the same as

Strand: a sequence of A's, C's, G's, and T's.

Substrand of a strand: a contiguous subsequence (possibly all) of the strand.

Substrand-free collection of strands: A collection of strands, no strand of which is a substrand of another strand (except for itself)

Prefix: a substrand at the *left end* of a strand.

Example: CAAG is a prefix of strand X . G is another prefix of X .

Suffix: a substrand at the *right end* of a strand.

Example: CCAA is a suffix of strand X . CAA is both a prefix and suffix of X . Also, a strand is always a prefix and suffix of itself.

$R \rightarrow S$ -overlap of strands R, S : if S is a substrand of R then the $R \rightarrow S$ -overlap is S ; otherwise, it is the maximum-length suffix of R which is also a prefix of S .

Example: C is the $Y \rightarrow Z$ -overlap. The $Z \rightarrow X$ -overlap and $X \rightarrow Y$ -overlap are *empty*, that is, contain no unit at all.

$R \rightarrow S$ -overhang of strands R, S : what remains of S , when the $R \rightarrow S$ -overlap is removed from the left end of S .

Example: CAAATTA is the $X \rightarrow Z$ -overhang. Since the $X \rightarrow Y$ -overlap is empty, the $X \rightarrow Y$ -overhang equals Y .

Assembly of a collection \mathcal{C} of strands: a strand A obtained from \mathcal{C} by the following algorithm:

First, pick any strand and remove it from \mathcal{C} . Add this strand to the assembly, by initializing A to be this strand. Then, repeatedly remove a strand from \mathcal{C} and add to the (partial) assembly, until no strands are left in \mathcal{C} . To add a strand S to A , simply append the $A \rightarrow S$ -overhang to the right end of A .

Example: If the collection \mathcal{C} consists of X, Y , and Z , then one assembly (discussed in the main text) is the strand $A = \text{CAAGACCAACAAATTACCGGGCC}$, created by adding X, Z , and Y in that order. Another assembly is $\text{CAAGACCAATTACCGGGCCAACAAATTA}$: to create this assembly, strands are added in the order X, Y, Z . Yet another assembly is $\text{CAACAAATTACCGGGCCAAGACCAA}$ (why?).

Shortest assembly of a collection \mathcal{C} of strands: an assembly of \mathcal{C} whose length is less than or equal to the length of any other assembly of \mathcal{C} .

Figure 1: Fragment assembly notation. Examples refer to three DNA strands: $X = \text{CAAGACCAA}$, $Y = \text{TTACCGGGCC}$, and $Z = \text{CAACAAATTA}$.

the shortest assembly of the collection obtained by removing W from \mathcal{C} . Can you explain why?

- 4.2 Give two distinct assemblies for the following collection of fragments: AATCTG, CCGCAA, TGTAATCCG, and CTGTAAAT.
- 4.3 Let \mathcal{C} be a collection of strands. Define a *superstrand for \mathcal{C}* to be any strand that contains all strands of \mathcal{C} as substrands. Explain why the following facts are true: (i) an assembly of collection \mathcal{C} is a superstrand for \mathcal{C} ; (ii) a superstrand for \mathcal{C} may not be an assembly of \mathcal{C} , and (iii) any shortest superstrand for \mathcal{C} is always a shortest assembly of \mathcal{C} .
- 4.4 Suppose that \mathcal{C} is a *substrand-free* collection of strands. Suppose that R is the last strand added to the assembly A of \mathcal{C} before S is added. Show that the $A \rightarrow S$ -overhang equals the $R \rightarrow S$ -overhang. Give an example to show that this may not be true if \mathcal{C} is not substrand-free.

While biologists might use additional criteria in judging the quality of an assembly of fragments, it's reasonable to focus on one criterion first, namely finding the shortest assembly. A solution for this problem can later be refined to account for other criteria. We can precisely define the Fragment Assembly Problem, or FAP, for short, as follows (see Figure 1 for notation used here).

Fragment Assembly Problem (FAP): An instance of FAP is a *substrand-free* collection of DNA strands. The problem is to find the *shortest assembly* of the strands. The “substrand-free” restriction will simplify things for us later on. Note that if our instance is not substrand-free, we can always just discard any substrands in our instance, and solve the reduced instance to get the correct solution to our original instance. In Exercise 1 you are asked to justify why this is so.

Fragment
Assembly
Problem
(FAP)

Now, we have a cleanly-stated computational problem in hand. Maybe a computer will be useful after all.

5 A digression: the traveling salesman

Let's take a break from DNA sequencing for a moment. Here's a fun problem to take your mind off things. A salesman, starting from his home, needs to fly to several cities and then return home. He has a list of one-way flights between each pair of cities, and the cost of each flight. What is the cheapest

route that the salesman can follow, in order to visit each city exactly once and end up back in his home city?

Figure 2 (a) illustrates a specific instance of this problem and introduces needed notation. Formally, we can define the Traveling Salesman Problem (TSP) as follows:

Traveling Salesman Problem (TSP): An instance of TSP is a *directed network*, in which *costs* (nonnegative numbers) label the links. The problem is to find a *cheapest tour* of the network.

Traveling
Salesman
Problem
(TSP)

Exercises:

- 5.1 For the TSP instance given in Figure 2 (a), find a tour of cost 14 that is different from the tour depicted in Figure 2 (b).
- 5.2 Figure 3 depicts another instance of TSP. Can you find the cheapest tour for this instance?

If an instance of TSP has many cities, it becomes surprisingly tricky to reliably find the cheapest tour in an efficient manner. This is in part because there are so many possible tours. There can be up to $n!$ (that is, n factorial, which is $n(n-1)(n-2)\dots 1$) possible tours for an instance with n cities. Even if there are only 1,000 cities, and you could calculate the cost of each possible tour in a microsecond (that is, 1 millionth of a second, or 10^{-6} seconds), it would take more time than since the creation of the universe to check them all! In contrast, many other natural problems defined on networks can be solved efficiently. This seems like a rather vague thing to say - for example, what exactly do we mean by an efficient solution here? - but this notion can be made completely rigorous. Examples of problems with efficient solutions include: figuring out whether two nodes in a network are connected, or what is the cheapest way to get from one node to another.

In fact, TSP has become very famous - the “grand challenge of computing”, in a sense. Researchers have worked hard on developing the best methods they can for the TSP. Some known methods are guaranteed to find the best solution, being very efficient on many instances of the problem, and only rarely are inefficient. Other methods are guaranteed to run efficiently, but may not always find the best solution. There is even a \$1,000,000 prize for the first person who either finds an efficient algorithm that is guaranteed to find the cheapest tour in any instance of the problem, or can prove that no such algorithm exists. You might well wonder why someone would award such a large prize for what seems like a rather contrived problem. We’ll return to this later, but for now, let’s get back to genome sequencing.

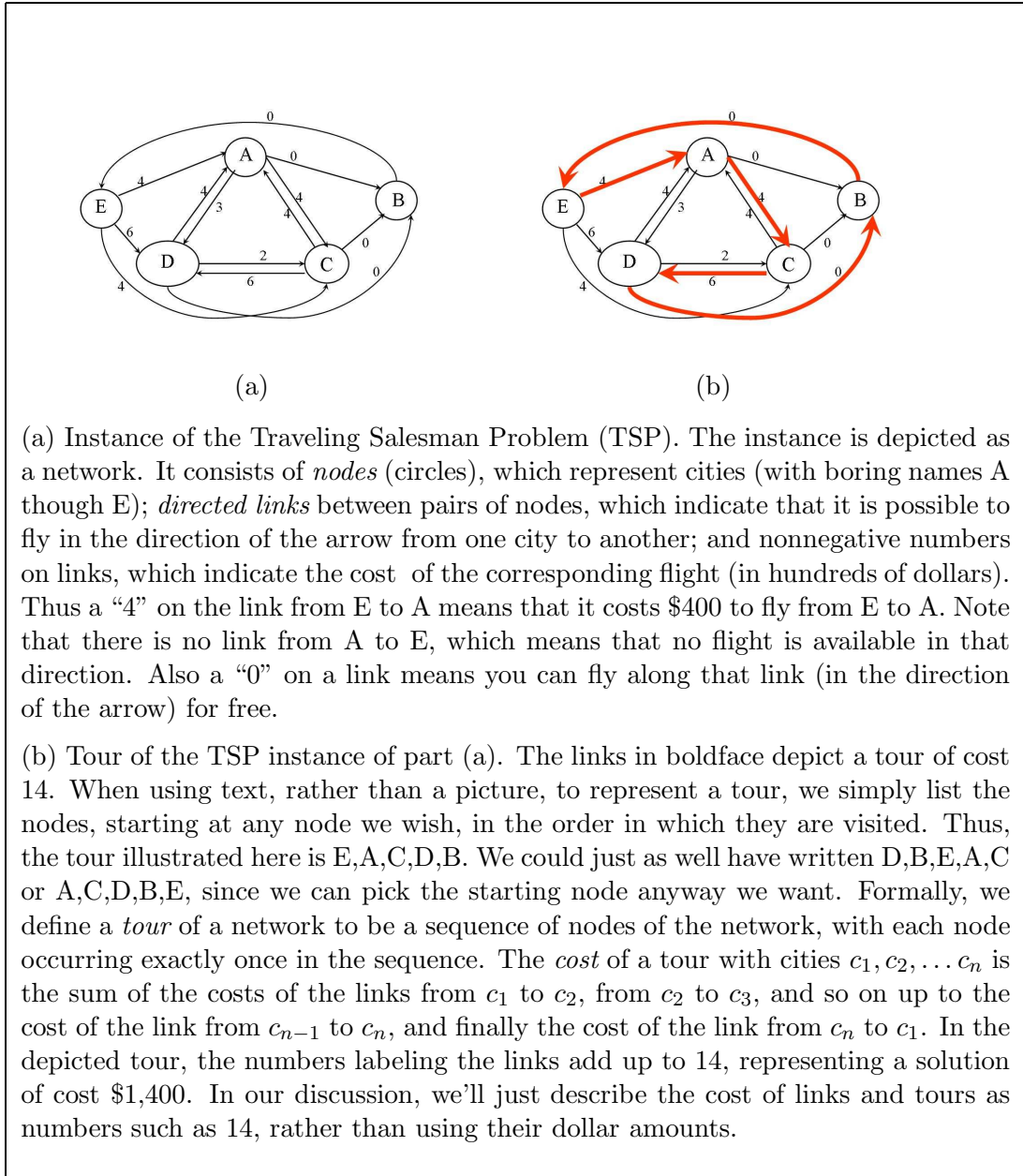


Figure 2: Traveling salesman notation.

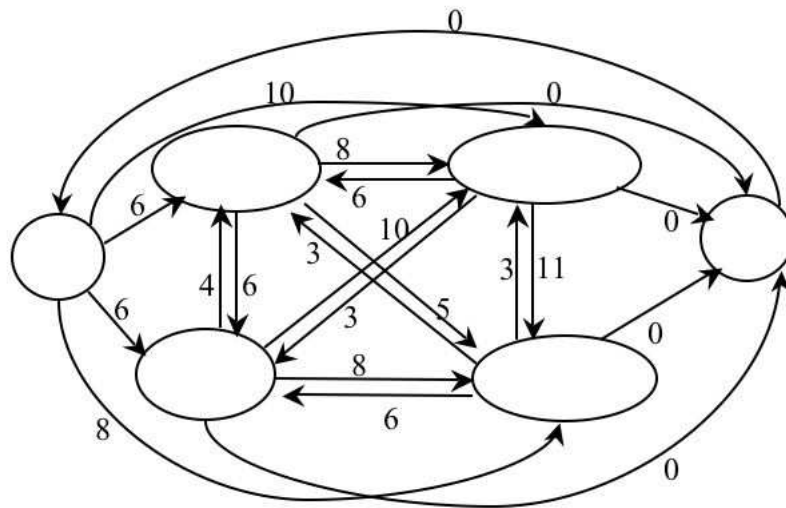


Figure 3: Another instance of the Traveling Salesman Problem. Finding the cheapest tour of this instance is no fun... isn't this what computers are supposed to be good for?

6 Back to genome sequencing

Remember that, at the end of Section 4, our goal was to solve the FAP. Based on our digression in Section 5, imagine you have at your disposal some computer software that is considered a state-of-the-art TSP solver. Given a description of a network, this software can spit out a low-cost tour of the network in very little time. If only you could somehow use the TSP solver to assemble your fragments.

It turns out that you can. In some sense, TSP is really FAP in disguise! We will show how you can easily convert any instance of FAP into an instance of TSP. This is a very useful insight, because it tells us that we can reap the benefits of decades of research on good methods for solving the TSP, and use them for solving FAP.

Can you think of ways to convert a FAP instance into a TSP instance? FAP seeks shortest assemblies, while TSP seeks cheapest tours. So, perhaps in our conversion, assemblies should somehow correspond to tours. Also, assemblies correspond to sequences of fragments (indicating the order in which the fragments are added to the assembly), and tours are sequences of nodes, so perhaps fragments should convert to cities. For an example, see Figure 4 (a). In what follows, we refer to nodes and fragments which label nodes interchangeably.

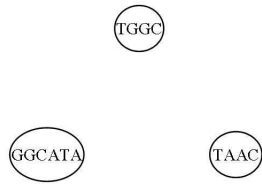
Links between nodes in the network indicate the order in which cities can be visited in a tour. Since fragments can be assembled in any order, it makes sense to add links, in both directions, between each pair of “nodes” (fragments). See Figure 4 (b).

One subtle difference between FAP and TSP is that an assembly has a distinct start and end, whereas a tour returns to its starting node. To reflect this difference, it is useful to add two extra nodes to our TSP instance, which we label *start* and *end*. We add links from *start* to each node labeled by a fragment, since we may start the assembly with any fragment. Also, we add links from each fragment node to *end*, since we may end the assembly with any fragment. And finally, we add a link from *end* back to *start*, so that a tour can get from *end* back to *start*. See Figure 4 (c).

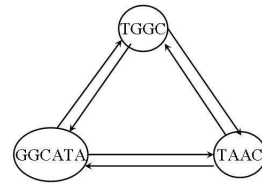
Our conversion so far ensures that the following property holds.

Property 1 *A tour of our network corresponds to an assembly of the fragments, and vice versa.*

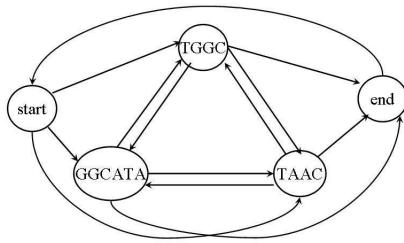
An example, pick a tour, any tour - let’s say the tour indicated by boldface arrows in Figure 4 (d). If we begin at the *start* node, fragment nodes are visited in the order: TGGC, GGCATA, TAAC (followed by a visit to *end* and



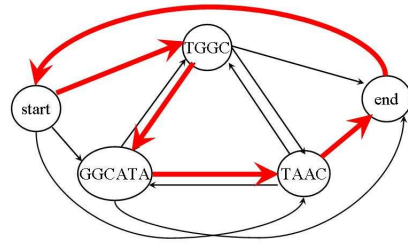
(a) Create one node per fragment.



(b) Add links in both directions between nodes.



(c) Add *start* and *end* nodes, and additional links.



(d) A tour of the network corresponds to an assembly of the fragments. The tour depicted here corresponds to the assembly TGGCATAAC.

Figure 4: Converting a FAP instance into a TSP instance. The FAP instance has three fragments, namely GGCATA, TGGC, and TAAC.

finally back to *start*). This corresponds to the assembly in which fragments are added in the same order:

```
TGGC
  GGCATA
    TAAC
```

To complete our conversion of a FAP instance to a TSP instance, we still need to add costs to the links between nodes of the TSP instance. How should these costs be chosen? A reasonable strategy is to choose costs so that the following property holds:

Property 2 *The cost of a tour equals the length of the corresponding assembly.*

If we can achieve this, then the cheapest tour of the network would indeed correspond to the shortest assembly of the fragments. Before reading the next paragraph, you may want to think yourself about a rule for assigning costs to links, referring to the example of Figure 4.

As each fragment S is added to the assembly A , the length of the assembly increases by an amount equal to the length of the $A \rightarrow S$ -overhang. From Exercise 4, we know that if R is the last strand added to the assembly A before S is added, then the $A \rightarrow S$ -overhang is the same as the $R \rightarrow S$ -overhang. This suggests that the cost of a link from a fragment R to a fragment S should be the length of the $R \rightarrow S$ -overhang. And, the cost of the link from *start* to a fragment S should be the length of S , since if the assembly starts with S , then in the initial assembly step the length of the partial assembly is exactly the length of S . Finally, links directed into or out of the *end* node have cost “0”, since these links do not increase the length of the corresponding assembly. Figure 5 shows the costs of several links for the network obtained in the example of Figure 4. The overall conversion method is summarized in Algorithm 1.

We’re done! We have described a general and efficient method for converting an instance of FAP to an instance of TSP. As a second example, the network of Figure 3 is the result of converting the instance given in Exercise 2. Properties 1 and 2 ensure that the cheapest tour of the TSP instance is indeed a shortest assembly of the FAP instance from which it is constructed.

Exercises:

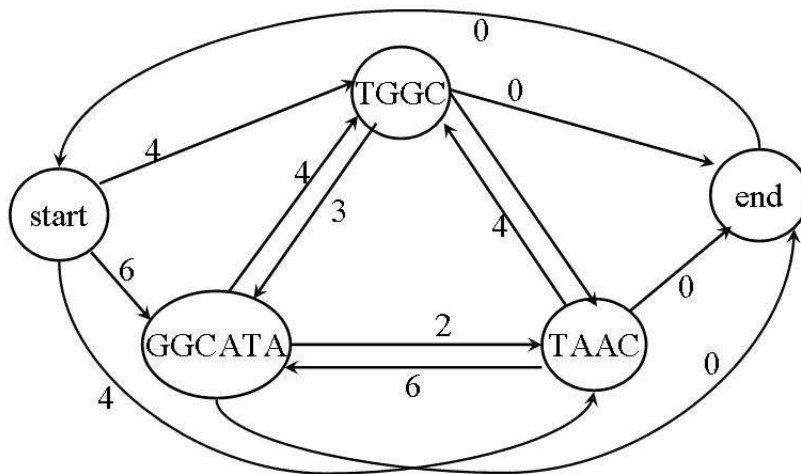


Figure 5: Converting a FAP instance into a TSP instance, continued from Figure 4. The cost on link from *start* to TGGC is 4, since if TGGC is the first fragment added to the assembly, it contributes a length of 4. The cost of the link from TGGC to GGCATA is 3, since the TGGC→GGCATA-overhang is ATA, which has length 3.

Algorithm 1 Algorithm to convert an instance of the Fragment Assembly Problem to the Traveling Salesman Problem

algorithm FAP-to-TSP

input: an instance of the Fragment Assembly Problem (FAP),
that is, a substrand-free collection of fragments.

output: an instance of the Traveling Salesman Problem (TSP),
whose shortest tour has cost equal to the length of shortest
assembly of the input FAP instance.

First, create the nodes and links of the TSP instance (see Figure 4):

- (a) Add one node per fragment. Label each node with a distinct fragment.
- (b) Add two directed links, one in each direction, between each pair of nodes.
- (c) Add two extra nodes, labeled *start* and *end*.
Add a link from *start* to each fragment node.
Add a link from each fragment node to *end*.
Add a link from *end* node to *start*.

Next, label the links with costs (see Figure 5):

- For each pair of fragments R and S , label the link from R to S
with the length of the $R \rightarrow S$ -overhang.
 - Label the link from *start* to fragment S with the length of S .
 - Label the link from fragment S to *end* with “0”.
 - Label the link from *end* to *start* with “0”.
-

6.1 Convince yourself that Properties 1 and 2 hold for the conversion from FAP to TSP, as given in Algorithm 1.

7 Why a \$1,000,000 prize?

The amazing thing is: there are thousands of interesting and important problems, ALL of which are the TSP in disguise! That is, given an instance of any of these problems, it is possible to efficiently convert it into an instance of TSP, in such a way that if we knew the cheapest tour of the TSP instance, we could easily infer the best solution to the original problem instance. If we had a fast algorithm to figure out the optimal solution to TSP, we would have a fast algorithm for finding an optimal solution to all of these thousands of problems. This is why there is a \$1,000,000 reward for determining whether there is a solution to the TSP problem.

It is also true that, given an instance of TSP, it is possible to efficiently convert it into an instance of FAP. We will not discuss how this can be done. However, this conversion tells us that we are not likely to find an efficient method that always finds the shortest assembly for any FAP instance. If we could, we could efficiently solve an instance of TSP by first converting it into an instance of FAP and then using our efficient algorithm for FAP. We would then be able to claim the \$1,000,000 prize!

We've done all of this work to develop a method for converting FAP instances into TSP instances, but there isn't an efficient method that is guaranteed to solve TSP in the first place. But, I hope that these notes have been instructive in conveying the following points.

First, some problems just don't seem to have efficient algorithms. Such problems are often called *intractable* problems. If you continue to work on computational methods for solving problems in the future, it's good to keep this in mind. You may be better off considering a heuristic technique that works well on practical instances of the problem, than trying for a solution that is guaranteed to work efficiently on all instances. At least we know that more information about techniques for solving TSP or related problems might help us.

Second, the result that TSP is a disguise for thousands of natural and important computational problems is a famous insight of computer science. Hopefully you too find it fascinating. This insight will be passed on to students like you, long after html and javascript and many other things you are learning about are obsolete.

8 Concluding thoughts on our genomes

While we now know our genomic script, we understand little of its meaning. In just a few years, the estimate of the number of our genes has been dramatically revised, from over 100,000 down to 25,000 or less. We still don't know exactly how many genes we have. We know very little about what most of those genes do, or how the marvelous complexity of a human can be coded in so few genes. We can expect a lot of progress in understanding our genome over the next several decades. Along with the methods of the wet lab, methods of computer science will continue to help unlock many secrets of the functioning of our bodies.