Motif finding algorithms

Introduction

The central dogma of molecular biology outlines the basic idea that instructions encoded in DNA are acted upon by converting them into proteins through transcription from DNA to RNA followed by translation of the RNA into protein. This basic explanation ignores the fact that whatever actions are taken by proteins need to be coordinated in some fashion. For example, a set of proteins need to be created in a well-defined order in order to replicate a cell, another set of proteins are necessary to respond to changes in nutrients, or to resist an attack from viruses or toxins. This chapter describes some computational approaches used to understand how cells coordinate the activity of genes—a process called **gene regulation**.

Learning outcomes

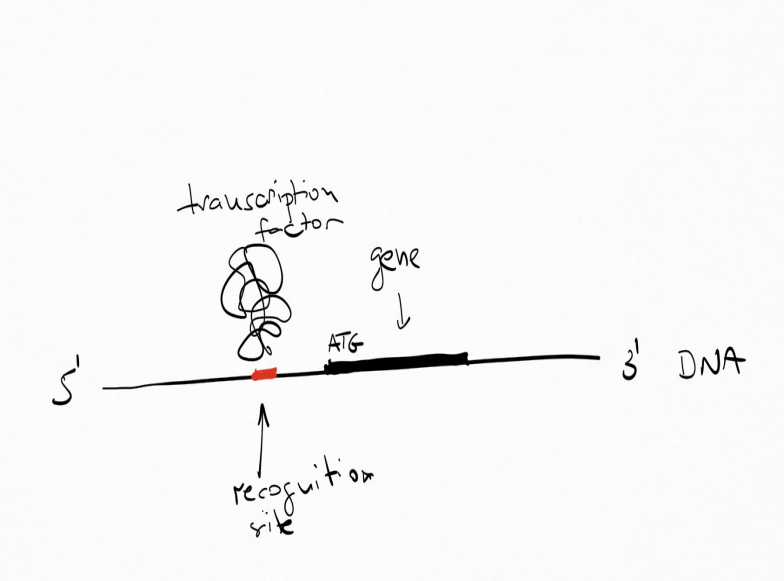
After completing this chapter, students should be able to:

- Describe the concept of motifs and why they are important for understanding biology
- Describe the difference between motifs and other signals found in DNA
- Describe the concept of a consensus string/motif and correctly construct the consensus of a given motif.
- Implement a brute force algorithm to find motifs
- Describe the benefit of modeling motif profiles as probability distributions
- Score a sequence against a profile using probabilities
- Compute the entropy of a motif matrix
- Articulate the need for pseudo-counts (Laplace's rule)
- Describe and/or implement the Gibbs sampling algorithm
- Describe why the Gibbs sampler allows sub-optimal solutions

Genes and motifs

At a very high level, the transcription process that converts DNA into RNA, and thus "turns on" a gene, is triggered when a special protein—called a **transcription factor**—binds to the DNA just "upstream" of the gene, i.e., just before the gene in the 5'-3' orientation. In other words, the gene gets activated if

the transcription factor binds to the DNA right before the gene.



To enact the complex "programs" that determine how cells react to stimuli and replicate, organisms rely on **regulatory networks**—complex networks that link together a large connection of transcription factors to the genes that they control, some of which themselves encode other transcription factors. To elucidate the structure of these networks, it is important to decipher the connections between transcription factors and the genes that they can regulate.

One useful bit of information can be derived from experiments that measure the **expression** of genes—that is, whether the cell produces the RNA corresponding to the gene, indicating the gene is "turned on". There are a number of techniques for figuring this out, which can also give some information about the level of expression (i.e., determining how active is a gene), however these technologies are beyond the scope of this chapter. Here we simply assume that an experiment can indicate which set of genes is activated when the cell receives a particular stimulus (e.g., when it's fed glucose). For different stimuli we expect that different sets of genes will be active. How can we go from this information to figuring out what "signals" are hidden in the DNA to tell the organism exactly which genes to turn on or off? This is the primary focus of this chapter.

So, let's try to formalize what we are trying to figure out. We'll assume that we know where all the genes are in the genome, and that the signal we are looking for occurs right before the start of the genes within some small range (e.g., 300-500 letters before the ATG). If some subset of the genes get expressed at the same time, then we can infer that the same signal is hidden within the DNA that precedes these genes. In other words, we are looking for some similarity between the upstream regions of the set of genes that are co-expressed.

At a first glance, knowing what we know from trying to figure out where the origin of replication is located, we could try to look for k-mers that are shared by all the upstream regions of the genes of interest. We can formalize this task as:

Given n strings of length L Find all k-mers that occur in all of the strings

If you think a little bit about this problem, you'll realize you can solve it by modifying your code for finding the most frequent k-mer. It just takes a bit more bookkeeping.

It turns out, however, that the answer is not that simple. Transcription factors can bind to imperfect patterns, i.e., each upstream region may have a slightly different version of signal detected by the transcription factor. Simply counting k-mers won't be useful as we're no longer looking for exact matches but need to tolerate some differences. To figure out how to solve this problem, we need to think a bit more carefully about how to define the problem. The picture below may help clarify what we are looking for.

CACCT**AGCT**AGCT
GTGGCTACG**ATCT**
CGGC**ATAT**AATGA
GCCAGA**GGCT**ACA

In this figure, you see four DNA strings, corresponding to the upstream regions of four genes, and inside each of them I have highlighted in underlined bold letters the "signal" to which a transcription factor binds. Note that they are all different but somewhat similar, as you can see better if you align them all on top of each other:

AGCT
ATCT
ATAT
GGCT

Because these strings are not identical, but similar, they are usually described as a **motif**, and we are trying to figure out the motif recognized by a particular transcription factors by comparing to each other the upstream regions of the genes turned on by the transcription factor. But saying that the four strings in the motif "look" similar is not sufficient. We need to define this similarity more formally in order for this definition to be understood by a computer.

Scoring motifs

A first definition of a "motif score" involves computing how far we are from having a perfect motif, i.e., when all the four strings are identical. To do so, we simply count the number of letters in the motif matrix that differ from the majority letter in the corresponding column. In the example above, we highlight the divergent letters as lower case letters:

```
AGCT
AtCT
AtaT
gGCT

score(Motif) = 4
```

Note that the second column has two Gs and two Ts, and I have chosen the Ts to be the non-majority letters, but the score would be the same if I had selected the Gs instead.

Another way to think about this score is to try to infer what k-mer best encapsulates the information in the matrix. We can define the **consensus sequence** of the motif matrix as the k-mer (in our case 4-mer) that contains the majority letter in each column, i.e., AGCT in our case. Thinking briefly about this, you should be able to convince yourself that:

$$score(Motif) = \sum_i d(consensus(Motif), string_i)$$

where $d(consensus(Motif), string_i)$ is the Hamming distance between the consensus sequence and the i^{th} string (row) in the motif.

Note, however, that the score function we are using here is not very sensitive. Both of the motifs shown below have the same score, yet the first one seems more "motif-y".

```
ACA ACA
ATA ATA
ATA AGA
ACA ACA
```

Both motifs have a score of 2 because there are two letters in the second column that differ from the majority/consensus letter, yet in the first motif the two divergent letters are the same whereas in the second one they are different. We need to think of some other way to capture the difference between these motifs. We can find a solution by computing the motif's **profile**—a matrix that records the frequency of each of the letters in each column.

Specifically, we construct a matrix that has 4 rows, each for each DNA letter, and the number of columns equal to the "width" of the motif (the number of columns in the motif). Each cell in the profile represents the fraction of the letters in the corresponding column that is comprised by the letter represented by the corresponding row. This makes more sense in an example:

```
Motif:
  A    C    A
  A    T    A
  A    G    A
  A    C    A

Profile:
A 1.00 0.00 1.00
C 0.00 0.50 0.00
G 0.00 0.25 0.00
T 0.00 0.25 0.00
```

In the first column all letters are A, thus the value for A is 1, while the values for C, G, and T are 0. In the second column, there are no As, thus the value in the corresponding row is 0, but we have 50% C (hence the 0.5 in row 2 column 2), and 25% Gs and Ts (in the third and fourth rows).

Hopefully this makes sense, but how does it help us figure out a better score. What we've effectively done is transform the motif into a set of probabilities. The profile contains for each column the probability that we observe a particular letter in that column. Information theory allows us to compute the information content of a set of probabilities, more precisely their **entropy**:

$$H(p_1, p_2, \ldots, p_n) = -\sum_i p_i \log_2 p_i$$

You can now see that there's a difference between a column that contains just two letters (C, C, T, T) and one that has three (C, C, T, G) since the corresponding columns in the profile matrix and the corresponding entropies are:

H(0, 0.5, 0, 0.5) = 1
H(0, 0.5, 0.25, 0.25) = 1.5

(if you are curious how I came up with these numbers, it helps to remember that log base 2 of a power of 2 is easy to figure out as $\log_2 2^k = k$. Thus, $\log_2(0.5) = -1$, $\log_2(0.25) = -2$, and so on.)

Now you can see that the first column has a lower entropy, thus it's more "conserved".

To compute the entropy of the whole motif you simply add up the entropy values for each column.

Here are some questions for you:

- What is the entropy of a column that only contains one letter?
- What is the entropy of a column that contains equal counts of each letter (i.e., the profile is [0.25, 0.25, 0.25, 0.25])?
- What is the highest possible entropy? Can you prove it?

Finding "hidden" motifs

Now that we have some ways of computing the "score" of a motif, we can come back to the problem that we started with—figuring out a way to find the motif that determines that a set of genes are expressed at the same time. More specifically, given *t* strings of length *L* and a value of *k* (the k-mer "width" of the motif we are looking for), find a string of length *k* within each of the *t* strings such that the motif created by these *t* k-mers has a relatively small score.

Read the underlined text a few times and think if it's making you as uncomfortable as it is making me. The problem it defines is vague. What does "relatively small score" mean? How would one even pick a cut-off value?

When computer scientists are faced with such a situation, we tend to make things simpler and more concrete by looking for the motif with the lowest score. The problem definition, thus, becomes:

Given *t* strings of length *L* and a value of *k* (the k-mer "width" of the motif we are looking for), find a string of length *k* within each of the *t* strings such that the motif created by these *t* k-mers has the minimum score value over all possible choices of motifs.

This definition eliminates ambiguity, and also makes it easier to prove things about the algorithms we may come up with (though we won't do that here).

Let's think a little bit about this problem definition and how we may solve it. The simplest way is an exhaustive search:

```
            for each k-mer k1 in string 1 :
              for each k-mer k2 in string 2 :
               for each k-mer k3 in string 3 :
                 ...
(1)              current_score = score(Motif(k1, k2, k3, ...))
                 if current_score < best_score:
                  best_score = current_score
                   best_motif = Motif(k1, k2, k3, ...)
```

This algorithm is guaranteed to find the lowest scoring motif. However, what is it's runtime?

If you haven't figured out the answer yet, we have *t* nested loops (one for each string in the input), and each loop iterates over the *O(L)* k-mers in each string, so the inner code segment gets executed $O(L^t)$ times. The inside of the loop itself has to compute the score of a motif with *k* columns and *t* rows, for an additional multiplicative factor of *kt*, leading to a total runtime of $O(ktL^t)$. Let's plug in some numbers to see what this means in practice. As we mentioned a reasonable value for L is somewhere between 100 and 500, so let's pick the lower side L=100. Let's assume a fairly small motif of k=8 and a set of maybe 20 genes (t=20). The runtime becomes $O(8 * 20 * 100^{20})$ or $1.6 * 10^{42}$ which happens to be a pretty large number. You can easily code this algorithm but may have to wait for years to get an answer.

Incidentally, the calculation that we just made also gives us a way to define the size of the **search space** within which we are looking for the motif. The size of the search space is related to the computational complexity of the problem we are trying to solve—knowing this size is exponential gives us a hint that perhaps the motif finding problem is difficult, perhaps even NP-hard (though while an exponential search space is necessary for a problem to be NP-hard, there are problems that have an exponential search space that can be solved in polynomial time).

Is there any way we can solve this problem faster? Remember that we earlier defined the score in two different ways, either as the score of the full motif matrix, or as a sum of the Hamming distances between the consensus string and each of the strings selected in the motif. Both scores give us exactly the same number, so we could turn the problem around and suggest a different algorithm:

```
            for each string of length k called consensus:
              for each string si in the input: # each of the t strings
                                               # within which the motif
(2)                                            #"hides"
                find k-mer ki such that dist(consensus, ki)is minimized
                                                 over all k-mers
                add ki to the motif
              current_score = score(Motif(k1, k2, k3, ...))
              if current_score < best_score:
                best_score = current_score
                best_motif = Motif(k1, k2, k3, ...)
```

Is this algorithm any faster than the one before? Let's unpack it a bit. The outer loop iterates over all DNA strings of length k, i.e. , 4^k times. The inner loop gets executed t times, and within each of the loops we are comparing the selected consensus k-mer to all possible k-mers in the current string, for a total run time of O(kL). Thus, the total runtime is O(tkL 4^k). Plugging in the same numbers as before we get O(20 * 8 * 100 * 4^8) = 1,048,576,000 = O(10^9)—a large number but substantially smaller than the 10^{42} we computed earlier.

Just by turning the problem around we managed to significantly speed up the algorithm.

Probabilistic search

The last algorithm described in the previous section relied on guessing a possible consensus sequence for the motif, then finding k-mers within each string that were similar to the consensus, in terms of the Hamming distance. Could we do a bit better if we knew more about the potential motif than just its consensus?

As discussed when we defined scores for motifs, converting the motif into a profile matrix allows us more nuance than we could get from simply counting disagreements between the k-mers in the motif. Perhaps we can use the same idea in searching. One strategy, that's quite common in a number of computer science applications, is to think of the profile not as a representation of the actual letters in the motif, but as a "recipe" for creating k-mers that could belong to the motif. Viewing the profile through this **generative** point of view, you can create k-mers by flipping a biased 4-sided coin for each column which yields an A, C, G, or T according to the probabilities encoded in the profile. Of course we don't want to generate random k-mers, but this point of view also allows us to calculate how likely a particular k-mer is to have been generated by the profile, as follows.

Let's take the profile

```
A  0.5  0.4  0.0  0.1
C  0.0  0.0  0.8  0.3
G  0.1  0.3  0.1  0.4
T  0.4  0.3  0.1  0.2
```

and the k-mer: `AGTT`

The "fit" between the k-mer and the profile can be defined by the conditional probability `p(AGTT|profile)` (probability of AGTT given the profile), which is simply the product of the probabilities of the letters in the corresponding columns:

```
A  0.5  0.4  0.0  0.1
C  0.0  0.0  0.8  0.3
G  0.1  0.3  0.1  0.4
T  0.4  0.3  0.1  0.2
```

```
p(AGTT|profile) = 0.5 * 0.3 * 0.1 * 0.2 = 0.003
```

Note that we have to look in the right column to find the probability for each letter.

Given this new tool, we can revisit the algorithm described above that searched for a consensus sequence within each of the strings. Assuming you have a guess at what the profile of the motif is, you can easily write code that finds the k-mer in a string that best matches the profile—**the profile most probable k-mer**.

Specifically, here's some pseudocode to perform this search:

```
            for each k-mer k in string:
              if p(k, profile) > maxP:
(3)            maxP = p(k, profile)
               maxK = k
            return maxK
```

You can easily work out the code to compute `p(k, profile)`.

By applying this piece of code to each of the strings representing the upstream regions of the genes of interest, we can build a motif that is most similar to the chosen profile.

But, wait a second. Our goal was to find a motif hidden in the strings without actually knowing the profile. If we don't know the motif how can we compute the profile? And if we know the motif, and can compute the profile, then why do we need to find the motif again? While what we did seems somewhat silly, it opens up the door for a new way of searching the large space of possible motifs to find the correct one.

First, note that both of the previous motif finding algorithms (1 and 2) are performing an exhaustive search of a large space of possible solutions. Algorithm 1 searches through all possible motifs that can be created from k-mers in the input strings. Algorithm 2 searches through all possible consensus sequences, then finds an appropriate motif for each of these. Perhaps we can do better.

The key idea behind this "smarter" search is to start with a guess about what the motif looks like, then keep improving this guess by selecting k-mers that better match the corresponding profile than the k-mers in the current motif. There are many different ways in which we can do this and here we focus on one such approach called **Gibbs sampling**.

The Gibbs sampler

The Gibbs sampler iterates between two stages—one that estimates the motif, and a second one that improves the motif, i.e., changes the k-mers in the motif in such a way that the score is decreased (defined as either an entropy or the divergence from the consensus sequence).

To start the algorithm, the Gibbs sampler builds an arbitrary motif by selecting either the first k-mer or a random k-mer from each input string. For simplicity, let us assume that we select the first k-mer from each string to create this motif, as shown below:

CACCTAGCTAGCT
GTGGCTACGATCT
CGGCATATAATGA
GCCAGAGGCTACA

yielding the motif matrix

```
CACC
GTGG
CGGC
GCCA
score = 9
```

As a second step in the algorithm, the Gibbs sampler selects one of the strings, at random, from which it will select a new k-mer. Let us assume we are selecting the third string. The corresponding k-mer (CGGC) is removed from the motif and a profile matrix is constructed from the remaining motif:

```
CACC
GTGG

GCCA

A 0.00 0.33 0.00 0.33
C 0.33 0.33 0.66 0.33
G 0.66 0.00 0.33 0.33
T 0.00 0.33 0.00 0.00
```

Now, the Gibbs sampler uses the profile P to find the profile most probable k-mer from the 3rd string, as follows:

```
CGGCATATAATGA p(CGGC|P)=0
CGGCATATAATGA p(GGCA|P)=0
CGGCATATAATGA p(GCAT|P)=0
CGGCATATAATGA p(CATA|P)=0
CGGCATATAATGA p(ATAT|P)=0
CGGCATATAATGA p(TATA|P)=0
CGGCATATAATGA p(ATAA|P)=0
CGGCATATAATGA p(TAAT|P)=0
CGGCATATAATGA p(AATG|P)=0
CGGCATATAATGA p(ATGA|P)=0
```

Ok, this is pretty strange—none of the k-mers matches the profile well, actually all of them have probability 0. What is happening here? The problem comes from the fact that some letters are missing from each column, leading to 0s in the corresponding column of the profile. Since we multiply the probabilities, any time the k-mer contains such a letter in the right position, the score will be 0.

To solve this problem, we can assign a non-zero probability to each cell in the profile matrix, so that each letter has a chance of being considered even if we haven't observed it in the motif created so far. Some people call this idea **Laplace's rule** in honor of the French mathematician Laplace who mused about the non-zero probability that the sun may not rise one day even though it has always risen previously. In practical terms,we use a concept called a **pseudo-count**. Rather than starting with 0s when we count the numbers of As, Cs, Gs, and Ts in each column, we start with 1. For the motif above, the count matrix and profile matrix, respectively are:

```
          A  1  2  1  2
          C  2  2  3  2
          G  3  1  2  2
          T  1  2  1  1
     Totals  7  7  7  7


A 0.14 0.29 0.14 0.29
C 0.29 0.29 0.43 0.29
G 0.43 0.14 0.29 0.29
T 0.14 0.29 0.14 0.14
```

Note that, at this point, the profile matrix does not contain any 0s, rather now we allow a small probability (14%) that the unseen letters could be observed.

Using this new profile matrix, we can recompute the probabilities for the k-mers in the third sequence:

```
CGGCATATAATGA p(CGGC|P)=0.003
CGGCATATAATGA p(GGCA|P)=0.007
CGGCATATAATGA p(GCAT|P)=0.002
CGGCATATAATGA p(CATA|P)=0.003
CGGCATATAATGA p(ATAT|P)=0.0007
CGGCATATAATGA p(TATA|P)=0.001
CGGCATATAATGA p(ATAA|P)=0.001
CGGCATATAATGA p(TAAT|P)=0.0007
CGGCATATAATGA p(AATG|P)=0.001
CGGCATATAATGA p(ATGA|P)=0.003
```

Thus, we can see that the second k-mer (GGCA) has the highest probability (0.007) and it will be selected to complete the motif:

```
CACC
GTGG
GGCA
GCCA
score = 7
```

Also note that the new motif has a lower score than the one we started with, so this search process is getting us closer to the motif we are looking for.

This concludes the second stage of the Gibbs sampler.

We, then, repeat the process:

```
        1: pick a random string S
           remove the corresponding k-mer from the motif
(4)        construct the profile matrix (with Laplace's rule)
           find the profile most probable k-mer K in S
           insert K into the motif
           repeat from 1
```

How many times do we have to repeat this algorithm? Ideally, we'd do so until the score does not improve anymore, indicating that we found a minimal motif. Note that we are not guaranteed to find the best motif (the one with the lowest score over all possible motifs) because the algorithm can get stuck in local minima. By starting with a new set of initial k-mers we may be able to obtain a better motif. That is one argument in favor of picking a random set of k-mers instead of just selecting the first k-mer in each string.

In addition to checking whether the algorithm **converges** (the score doesn't change from iteration to iteration), we can also put a limit on the number of iterations in case we want to ensure we will get some answer within a reasonable amount of time.

While the Gibbs sampler is not guaranteed to find the best motif (particularly if it is stopped before converging), it turns out it is very effective in practice. Typically it finds "pretty good" motifs (with scores that are quite close to the best possible score) in a fraction of the runtime of exhaustive search algorithms such as those we have discussed at the beginning of this chapter.

A common problem encountered by search algorithms such as the Gibbs sampler is that they can get stuck in local minima. To reduce the chances this happens, we want to allow the algorithm to make sub-optimal choices. Specifically, we want to allow the algorithm to select any k-mer in the string being processed even if it is not the "profile most probable" k-mer. This is similar in spirit to the Laplace rule modification that allows for a small probability that every letter could be present at every column of the motif. To achieve this functionality in the Gibbs sampler, we will replace the selection of the k-mer from:

```
            find the profile most probable k-mer K in S
```

to:

```
            select a k-mer K from S with probability of its fit to motif
```

We are effectively flipping a biased coin (or more precisely a multi-faced die) that returns a k-mer with the probability defined by its "fit" with motif. The profile most probable k-mer will be the most likely to be selected, but the Gibbs sampler will occasionally select a less good k-mer, allowing it to explore a broader search space and avoid some local minima. The full algorithm is:

```
        1: pick a random string S
           remove the corresponding k-mer from the motif
(5)        construct the profile matrix (with Laplace's rule)
           select a k-mer K from S with probability of its fit to motif
           insert K into the motif
           repeat from 1
```

Final notes

What we described here is the basic Gibbs sampling algorithm. This algorithm has many applications outside of bioinformatics as well, for example when estimating probability distributions in Bayesian analysis and other machine-learning applications.

The Gibbs sampler is one of a class of random search algorithms that are used to find the "needle in a haystack"—a solution within a very large search space. Designing and using such algorithms can be quite an art as there are many tricks that can be employed to avoid finding local minima.

Exercises