Approximation Algorithms

In chapter 2 we mentioned that, for many problems, efficient polynomial algorithms are still unknown and unlikely ever to be found. For such problems, computer scientists often find a compromise in *approximation algorithms* that produce an approximate solution rather than an optimal one.[6] The *approximation ratio of algorithm $\mathcal{A}$ on input π* is defined as $\frac{\mathcal{A}(\pi)}{OPT(\pi)}$, where $\mathcal{A}(\pi)$ is the solution produced by the algorithm $\mathcal{A}$ and $OPT(\pi)$ is the correct (optimal) solution of the problem.[7] The *approximation ratio, or performance guarantee of algorithm $\mathcal{A}$* is defined as its maximum approximation ratio over *all* inputs of size n, that is, as

$$\max_{|\pi|=n} \frac{\mathcal{A}(\pi)}{OPT(\pi)}.$$

We assume that $\mathcal{A}$ is a *minimization* algorithm, i.e., an algorithm that attempts to minimize its objective function. For maximization algorithms, the approximation ratio is

$$\min_{|\pi|=n} \frac{\mathcal{A}(\pi)}{OPT(\pi)}.$$

In essence, an approximation algorithm gives a worst-case scenario of just how far off an algorithm's output can be from some hypothetical perfect algorithm. The approximation ratio of SIMPLEREVERSALSORT is at least $\frac{n-1}{2}$, so a biologist has no guarantee that this algorithm comes anywhere close to the correct solution. For example, if n is 1001, this algorithm could return a series of reversals that is as large as 500 times the optimal. Our goal is

6. Approximation algorithms are only relevant to problems that have a numerical objective function like minimizing the number of coins returned to the customer. A problem that does not have such an objective function (like the Partial Digest problem) does not lend itself to approximation algorithms.
7. Technically, an approximation algorithm is not correct, in the sense of chapter 2, since there exists some input that returns a suboptimal (incorrect) output. The approximation ratio gives one an idea of just how incorrect the algorithm can be.

$$0\ \underleftarrow{2\ 1}\ \underrightarrow{3\ 4\ 5}\ \underleftarrow{8\ 7\ 6}\ 9$$

Figure 5.2 Breakpoints, adjacencies, and strips for permutation 2 1 3 4 5 8 7 6 (extended by 0 and 9 on the ends). Strips with more than one element are divided into decreasing strips (←) and increasing strips (→). The boundary between two nonconsecutive elements (in this case, 02, 13, 58, and 69) is a breakpoint; breakpoints demarcate the boundaries of strips.

to design approximation algorithms with better performance guarantees, for example, an algorithm with an approximation ratio of 2, or even better, 1.01. Of course, an algorithm with an approximation ratio of 1 (by definition, a correct and optimal algorithm) would be the acme of perfection, but such algorithms can be hard to find. As of the writing of this book, the best known algorithm for sorting by reversals has a performance guarantee of 1.375.

Breakpoints: A Different Face of Greed

We have described a greedy algorithm that attempts to maximize $prefix(\pi)$ in every step, but any chess player knows that greed often leads to wrong decisions. For example, the ability to take a queen in a single step is usually a good sign of a trap. Good chess players use a more sophisticated notion of greed that evaluates a position based on many subtle factors rather than simply on the face value of a piece they can take.

The problem with SIMPLEREVERSALSORT is that $prefix(\pi)$ is a naive measure of our progress toward the identity permutation, and does not accurately reflect how difficult it is to sort a permutation. Below we define *breakpoints* that can be viewed as "bottlenecks" for sorting by reversals. Using the number of breakpoints, rather than $prefix(\pi)$, as the basis of greed leads to a better algorithm for sorting by reversals, in the sense that it produces a solution that is closer to the optimal one.

It will be convenient for us to extend the permutation $\pi_1 \cdots \pi_n$ by $\pi_0 = 0$ and $\pi_{n+1} = n+1$ on the ends. To be clear, we do not move π_0 or π_{n+1} during the process of sorting. We call a pair of neighboring elements π_i and π_{i+1}, for $0 \leq i \leq n$, an *adjacency* if π_i and π_{i+1} are consecutive numbers; we call the pair a *breakpoint* if not. The permutation in figure 5.2 has five adjacencies (2 1, 3 4, 4 5, 8 7, and 7 6) and four breakpoints (0 2, 1 3, 5 8, and 6 9). A permutation on n elements may have as many as $n+1$ breakpoints (e.g., the permutation 0 6 1 3 5 7 2 4 8 on seven elements has eight breakpoints) and as few as 0 (the identity permutation 0 1 2 3 4 5 6 7 8).[8] Every breakpoint corresponds to a pair of elements π_i and π_{i+1} that are neighbors in π but not in the identity permutation. In fact, the identity permutation is the only permutation with no breakpoints at all. Therefore, the nonconsecutive elements π_i and π_{i+1} forming a breakpoint must be separated in the process of trans-

forming π to the identity, and we can view sorting by reversals as the process of eliminating breakpoints. The observation that every reversal can eliminate *at most* two breakpoints (one on the left end and another on the right end of the reversal) immediately implies that $d(\pi) \geq \frac{b(\pi)}{2}$, where $b(\pi)$ is the number of breakpoints in π. The algorithm BREAKPOINTREVERSALSORT eliminates as many breakpoints as possible in every step in order to reach the identity permutation.

```
BREAKPOINTREVERSALSORT(π)
1  while b(π) > 0
2      Among all reversals, choose reversal ρ minimizing b(π · ρ)
3      π ← π · ρ
4      output π
5  return
```

One problem with this algorithm is that it is not clear why BREAKPOINTREVERSALSORT is a better approximation algorithm than SIMPLEREVERSALSORT. Moreover, it is not even obvious yet that BREAKPOINTREVERSALSORT terminates! How can we be sure that removing some breakpoints does not introduce others, leading to an endless cycle?

We define a *strip* in a permutation π as an interval between two consecutive breakpoints, that is, as any maximal segment without breakpoints (see figure 5.2). For example, the permutation 0 2 1 3 4 5 8 7 6 9 consists of five strips: 0, 2 1, 3 4 5, 8 7 6, and 9. Strips can be further divided into *increasing* strips (3 4 5) and *decreasing* strips (2 1) and (8 7 6). Single-element strips can be considered to be either increasing or decreasing, but it will be convenient to

8. We remind the reader that we extend permutations by 0 and $n + 1$ on their ends, thus introducing potential breakpoints in the beginning and in the end.

define them as decreasing (except for elements 0 and $n+1$ which will always be classified as increasing strips).

We present the following theorems, first to show that endless cycles of breakpoint removal cannot happen, and then to show that the approximation ratio of the algorithm is 4. While the notion of "theorem" and "proof" might seem overly formal for what is, at heart, a biological problem, it is important to consider that we have modeled the biological process in mathematical terms. We are proving analytically that the algorithm meets certain expectations. This notion of proof without experimentation is very different from what a biologist would view as proof, but it is just as important when working in bioinformatics.

Theorem 5.1 *If a permutation π contains a decreasing strip, then there is a reversal ρ that decreases the number of breakpoints in π, that is, $b(\pi \cdot \rho) < b(\pi)$.*

Proof: Among all decreasing strips in π, choose the strip containing the smallest element k ($k = 3$ for permutation $0\,\overleftarrow{1\,2}\,\overleftarrow{7\,6\,5}\,\overleftarrow{8}\,\overleftarrow{4\,3}\,9$). Element $k-1$ in π cannot belong to a decreasing strip, since otherwise we would choose a strip ending at $k-1$ rather than a strip ending at k. Therefore, $k-1$ belongs to an increasing strip; moreover, it is easy to see that $k-1$ terminates this strip (for permutation $0\,\overleftarrow{1\,2}\,\overleftarrow{7\,6\,5}\,\overleftarrow{8}\,\overrightarrow{4\,3}\,9$, $k-1=2$ and 2 is at the right end of the increasing strip $0\,1\,2$). Therefore elements k and $k-1$ correspond to two breakpoints, one at the end of the decreasing strip ending with k and the other at the end of the increasing strip ending in $k-1$. Reversing the segment between k and $k-1$ brings them together, as in $0\,1\,2\,\underline{7\,6\,5\,8\,4\,3}\,9 \rightarrow 0\,\overrightarrow{1\,2\,3\,4}\,\overleftarrow{8}\,\overrightarrow{5\,6\,7}\,9$, thus reducing the number of breakpoints in π. □

For example, BREAKPOINTREVERSALSORT may perform the following four steps when run on the input $(0\,8\,2\,7\,6\,5\,1\,4\,3\,9)$ in order to reduce the number of breakpoints:

$$
\begin{array}{ll}
(0\ \overleftarrow{8}\ \overleftarrow{2}\ \overleftarrow{7\ 6\ 5}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{3}\ 9) & b(\pi) = 6 \\
(0\ \overleftarrow{2\ 8\ 7\ 6\ 5}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{3}\ 9) & b(\pi) = 5 \\
(0\ \overrightarrow{2\ 3\ 4}\ \overleftarrow{1}\ \overrightarrow{5\ 6\ 7\ 8}\ 9) & b(\pi) = 3 \\
(0\ \overleftarrow{4\ 3\ 2\ 1}\ \overrightarrow{5\ 6\ 7\ 8}\ 9) & b(\pi) = 2 \\
(\overrightarrow{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}\ 9) & b(\pi) = 0
\end{array}
$$

In this case, BREAKPOINTREVERSALSORT steadily reduces the number of breakpoints in every step of the algorithm. In other cases, (e.g., the permutation $(0\,\overrightarrow{1\,5\,6\,7}\,\overrightarrow{2\,3\,4\,8}\,\overrightarrow{9})$ without decreasing strips), no reversal reduces the number of breakpoints. In order to overcome this, we can simply find any

increasing strip (excluding π_0 and π_{n+1}, of course) and flip it. This creates a decreasing strip and we can proceed.

```
IMPROVEDBREAKPOINTREVERSALSORT(π)
1  while b(π) > 0
2      if π has a decreasing strip
3          Among all reversals, choose reversal ρ minimizing b(π · ρ)
4      else
5          Choose a reversal ρ that flips an increasing strip in π
6      π ← π · ρ
7      output π
8  return
```

The theorem below demonstrates that such "no progress" situations do not happen too often in the course of IMPROVEDBREAKPOINTREVERSALSORT. In fact, the theorem quanitifes exactly how often those situations could possibly occur and provides an approximation ratio guarantee.

Theorem 5.2 IMPROVEDBREAKPOINTREVERSALSORT *is an approximation algorithm with a performance guarantee of at most 4.*

Proof: Theorem 5.1 implies that as long as π has a decreasing strip, IMPROVEDBREAKPOINTREVERSALSORT reduces the number of breakpoints in π. On the other hand, it is easy to see that if all strips in π are increasing, then there might not be a reversal that reduces the number of breakpoints. In this case IMPROVEDBREAKPOINTREVERSALSORT finds a reversal ρ that reverses an increasing strip(s) in π. By reversing an increasing strip, ρ creates a decreasing strip in π implying that IMPROVEDBREAKPOINTREVERSALSORT will be able to reduce the number of strips at the next step. Therefore, for every "no progress" step, IMPROVEDBREAKPOINTREVERSALSORT will make progress at the next step which means that IMPROVEDBREAKPOINTREVERSALSORT eliminates at least one breakpoint in every two steps. In the worst-case scenario, the number of steps in IMPROVEDBREAKPOINTREVERSALSORT is at most $2b(\pi)$ and its approximation ratio is at most $\frac{2b(\pi)}{d(\pi)}$. Since $d(\pi) \geq \frac{b(\pi)}{2}$, IMPROVEDBREAKPOINTREVERSALSORT has a performance guarantee[9] bounded above by $\frac{2b(\pi)}{d(\pi)} \leq \frac{2b(\pi)}{\frac{b(\pi)}{2}} = 4$. □

9. To be clear, we are not claiming that IMPROVEDBREAKPOINTREVERSALSORT will take four times as long, or use four times as much memory as an (unknown) optimal algorithm. We

A Greedy Approach to Motif Finding

In chapter 4 we saw a brute force algorithm to solve the Motif Finding problem. With a disappointing running time of $O(l \cdot n^t)$, the practical limitation of that algorithm is that we simply cannot run it on biological samples. We choose instead to rely on a faster greedy technique, even though it is not correct (in the sense of chapter 2) and does not result in an algorithm with a good performance guarantee. Despite the fact that this algorithm is an approximation algorithm with an unknown approximation ratio, a popular

tool based on this approach developed by Gary Stormo and Gerald Hertz in 1989, CONSENSUS, often produces results that are as good as or better than more complicated algorithms.

GREEDYMOTIFSEARCH scans each DNA sequence only once. Once we have scanned a particular sequence i, we decide which of its l-mer has the best contribution to the partial alignment score $Score(\mathbf{s}, i, DNA)$ for the first i sequences and immediately claim that this l-mer is part of the alignment. The pseudocode is shown below.

```
GREEDYMOTIFSEARCH(DNA, t, n, l)
 1  bestMotif ← (1, 1, ..., 1)
 2  s ← (1, 1, ..., 1)
 3  for s_1 ← 1 to n − l + 1
 4      for s_2 ← 1 to n − l + 1
 5          if Score(s, 2, DNA) > Score(bestMotif, 2, DNA)
 6              BestMotif_1 ← s_1
 7              BestMotif_2 ← s_2
 8  s_1 ← BestMotif_1
 9  s_2 ← BestMotif_2
10  for i ← 3 to t
11      for s_i ← 1 to n − l + 1
12          if Score(s, i, DNA) > Score(bestMotif, i, DNA)
13              bestMotif_i ← s_i
14      s_i ← bestMotif_i
15  return bestMotif
```

---

are saying that IMPROVEDBREAKPOINTREVERSALSORT will return an answer that contains no more than four times as many steps as an optimal answer. Unfortunately, we cannot determine exactly how far from optimal we are for each particular input, so we have to rely on this upper bound for the approximation ratio.

GREEDYMOTIFSEARCH first finds the two closest l-mers—in the sense of Hamming distance—in sequences 1 and 2 and forms a $2 \times l$ *seed matrix*. This stage requires $l(n - l + 1)^2$ operations. At each of the remaining $t - 2$ iterations GREEDYMOTIFSEARCH extends the seed matrix into a matrix with one more row by scanning the ith sequence (for $3 \leq i \leq t$) each of the remaining $t-2$ sequences and selecting the one l-mer that has the maximum $Score(\mathbf{s}, i)$.

This amounts to roughly $l \cdot (n - l + 1)$ operations in each iteration. Thus, the running time of this algorithm is $O(ln^2 + lnt)$, which is vastly better than the $O(ln^t)$ of SIMPLEMOTIFSEARCH or even the $O(4^l nt)$ of BRUTEFORCEMEDIANSTRING. When t is small compared to n, GREEDYMOTIFSEARCH really behaves as $O(ln^2)$, and the bulk of the time is actually spent locating the l-mers from the first two sequences that are the most similar.

As you can imagine, because the sequences are scanned sequentially, it is possible to construct input instances where GREEDYMOTIFSEARCH will miss the optimal motif. One important difference between the popular CONSENSUS motif finding software tool and the algorithm presented here is that CONSENSUS can scan the sequences in a random order, thereby making it more difficult to construct inputs that elicit worst-case behavior. Another important difference is that CONSENSUS saves a large number (usually at least 1000) of seed matrices at each iteration rather than only the one that GREEDYMOTIFSEARCH saves, making CONSENSUS less likely to miss the optimal solution. However, no embellishment of this greedy approach will be guaranteed to find an optimal motif.