

# Chapter 2

## Algorithms and Data Structures in Next-Generation Sequencing

**Abstract** This chapter provides an overview of prevalent data structures and algorithms that are commonly utilized in bioinformatics. In particular, we place emphasis on data structures and algorithms that are employed in bioinformatic techniques during next-generation sequence assembly.

### 2.1 Data Structures

From a computational point of view, DNA, RNA, and proteins can be regarded as merely strings that consist of a finite set of letters (comprising of individual four letter alphabets for DNA and RNA, and a 20-letter alphabet for proteins). The efficient processing of these strings is necessary for almost all assembly and error correction techniques that are applied to bioinformatics sequences. Therefore, such techniques make extensive use of several data structures such as suffix/prefix trees, suffix arrays, graphs, hash tables, and Bloom filters. In this section, we will briefly describe these data structures and explain their applications in bioinformatics algorithms.

#### 2.1.1 Strings

A string is a finite sequence of elements, typically characters, which are chosen from a [set](#) called an [alphabet](#). For example, if  $\Sigma$  is a nonempty finite set that denotes an alphabet, then elements of  $\Sigma$  are called symbols or characters and any finite sequence of characters from  $\Sigma$  is called a string over  $\Sigma$  [1]. Thus, DNA sequences can be considered as strings over  $\Sigma = \{A, G, C, T\}$  while RNA sequences can be regarded as strings over  $\Sigma = \{A, G, C, U\}$ .

The string length is a nonnegative number that indicates the number of characters in the string. A string that does not contain any characters is called an empty string (denoted by  $\epsilon$ ). The length of an empty string is 0. Certain sets of strings are of

special interest. For example, the set of all strings over  $\Sigma$  (denoted as  $\Sigma^*$ ) is the [Kleene closure](#) of  $\Sigma$  whereas the set of all strings over  $\Sigma$  that have a specific length  $n$  is denoted as  $\Sigma^n$  ( $\Sigma^0 = \{\epsilon\}$  for any alphabet  $\Sigma$ ). In the context of bioinformatics, subsequences of a fixed length  $n$  are usually called  $k$ -mers. These  $k$ -mers can be used to identify regions of interest within bioinformatics sequences. They are also very useful in the determination of sequence alignment and assembly algorithms. Consider the following case for DNA where  $\Sigma = \{A, G, C, T\}$ , then  $\Sigma^2 = \{AA, AG, AC, AT, GA, GG, GC, GT, CA, CG, CC, CT, TA, TG, TC, TT\}$ , and  $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$

Defining an order in a set of strings is vital for some applications. For example, strings in suffix arrays (see below) are ordered lexicographically. Under this ordering scheme, strings are arranged based on the alphabetical order of their component characters. For example, if  $\Sigma = \{A, G, C, T\}$ , strings over  $\Sigma$  are ordered lexicographically based on the relationship  $\epsilon < A < AA < AAA < \dots < AAAC < AAAG < A AAT < AAC \dots$

A string of length  $n$  has  $n$  suffixes/prefixes varying in length from 1 to  $n$ . For example, the string ACGT has four prefixes (A, AC, ACG, and ACGT) and four suffixes (T, GT, CGT, ACGT).

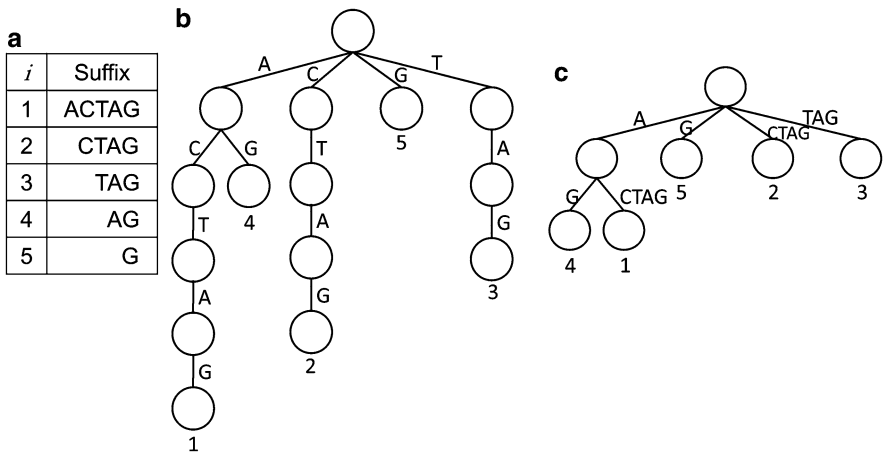
### 2.1.2 Suffix/Prefix Trees

Suffix/prefix trees for a string, or a set of strings, are created by entering all the suffixes/prefixes of this string(s) into a tree structure [2]. Suffix trees in particular are very useful in solving complex string problems. A suffix trie, also called a keyword tree [3], is a special kind of suffix tree. In a suffix trie, every edge is labeled by a single character. A suffix tree is formed by concatenating all the internal nodes in the suffix trie. Figure 2.1 shows an example that illustrates the difference between the suffix trie and suffix tree for the string ACTAG. In this figure, each leaf is labeled by a number that indicates the starting position  $i$  of the corresponding suffix in the string.

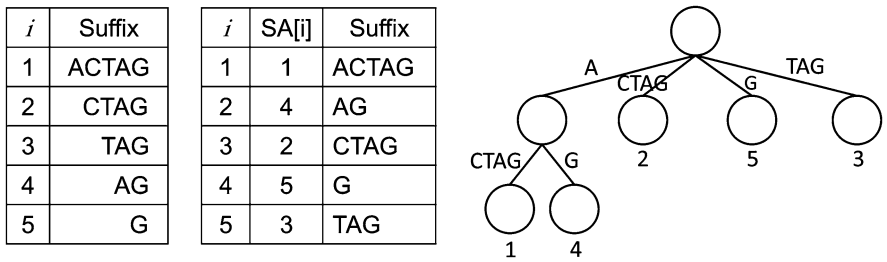
In one of many beneficial applications, suffix trees can be used efficiently to search for a specific string pattern in a large collection of strings [3]. In fact, using only  $O(n)$  time, a string of length  $n$  can be searched for in any collection of strings regardless of the size of the collection. For this reason, several assembly and error correction techniques for NGS data make use of suffix trees. The usage of suffix trees is limited, however, due to its large space requirement. As such, a suffix tree requires  $O(n|\Sigma|\log n)$  bits to represent a string of length  $n$  over an alphabet  $\Sigma$  [2].

### 2.1.3 Suffix Arrays

Suffix arrays have similar functionality as suffix trees but with reduced space requirements. A suffix array requires  $O(n \log n)$  bits only to store a string of length  $n$  over an alphabet  $\Sigma$ , regardless of the size of  $\Sigma$  [2].



**Fig. 2.1** Suffix trie versus suffix tree. (a) The suffixes, (b) suffix trie, (c) suffix tree for the string ACTAG



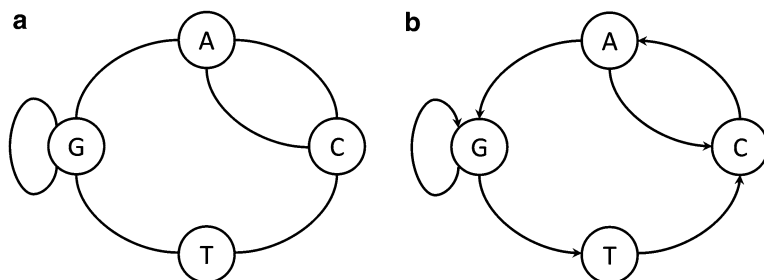
**Fig. 2.2** Suffix array versus suffix tree. Suffix array for the string in Fig. 2.1

In a suffix array, suffixes are sorted lexicographically in increasing order. Figure 2.2 shows the suffix array (SA) of the same string in Fig. 2.1. Note the correspondence between the order of the suffixes in the suffix array and the leaves in the suffix tree.

Since the suffix array stores only the start positions of the ordered suffixes of a string of length  $n$ , each entry in the array needs  $\log n$  bit of space, and since there are  $n$  entries (number of suffixes) in a suffix tree, the space requirement of a suffix array is  $O(n \log n)$ .

2.1.4 Graphs

A graph is a set of nodes, also called vertices, that are connected by edges. Graphs can be categorized as directed or undirected. Edges in directed graphs have a



**Fig. 2.3** Graph types: (a) Undirected graph. (b) Directed graph

specific direction whereas edges in undirected graphs have no direction [4]. Figure 2.3 shows examples of both types of graphs.

A path from one vertex to another in a graph is called a *walk*. A closed walk is a walk that starts and ends at the same vertex. A closed walk that has no repeated edges or vertices (other than the starting and ending vertex) is called a *cycle*, e.g., the walk A–G–G–T–C–A in Fig. 2.3b represents a cycle.

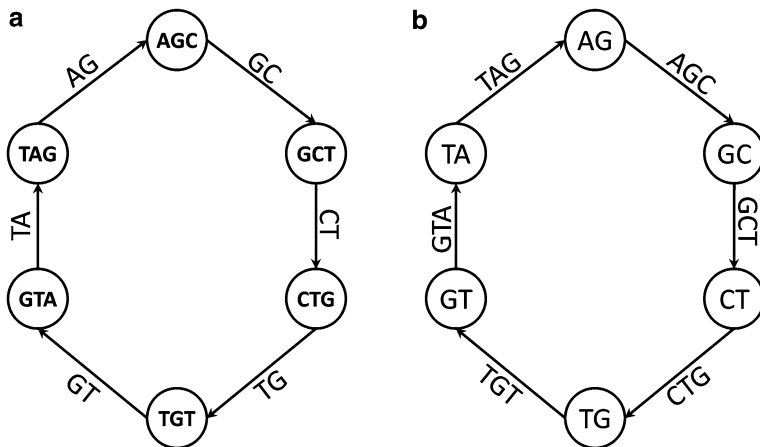
There are two types of cycles that are very useful for genome sequence assembly, namely, *Hamiltonian* cycles and *Eulerian* cycles. In a Hamiltonian cycle, every node of the graph is visited exactly once, whereas in an Eulerian cycle, every edge of the graph is visited exactly once.

Genome sequence assembly techniques which are based on Sanger sequencing make use of graphs to reconstruct long contiguous sequences from shorter reads by representing each read by a vertex and representing overlap between reads by edges that connect each pair of reads as illustrated in Fig. 2.4a. Therefore, the problem of assembling reads represented by graph vertices is reduced to finding a Hamiltonian cycle in the graph [5].

In the next-generation sequencing environment, the instruments utilized produce billions of short sequencing reads. As such, finding a Hamiltonian cycle in a graph that contains a very large number of nodes is a challenging computational problem. Therefore, the method described above for representing reads in a graph is not suitable for next-generation sequencing data.

Fortunately, discovering the location of an Eulerian cycle is much more efficient. However, this requires representing next-generation sequence reads as edges rather than vertices. To realize this, modern next-generation sequence assemblers utilize De Bruijn graphs. De Bruijn graphs were originally designed to find the shortest circular superstring that contains all possible substrings of a specific length  $k$  over a given alphabet [5]. This problem is referred to as the superstring problem. The analogy between the superstring problem and the problem of assembling billions of short sequencing reads into a single genome has attracted several researchers to De Bruijn graphs.

An important factor in the assembly of next-generation sequence short reads using De Bruijn graphs is that every distinct  $k-1$  prefix or suffix of each  $k$ -mer is represented by a node. Accordingly, each pair of nodes is then connected with a

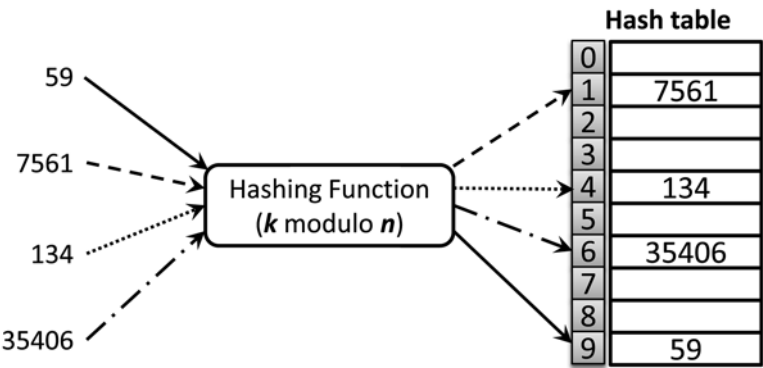


**Fig. 2.4** Genome assembly using graphs. (a) Nodes and edges represent reads and overlap between reads, respectively. (b) An example of a De Bruijn graph where suffixes and prefixes of reads are represented as nodes and the corresponding  $k$ -mers as edges

directed edge if a  $k$ -mer whose prefix is one of the two nodes and whose suffix is the other node exists. For example, AGCTGTAG is a small genome sequence from which the following three short reads can be sequenced: AGCT, CTGT, and GTAG. In splitting these reads into all possible  $k$ -mers of length  $k=3$ , 6 different 3-mers are obtained: AGC, GCT, CTG, TGT, GTA, and TAG. For these 3-mers, we are able to list all possible  $k-1$  prefixes and suffixes: AG, GC, CT, TG, GT, TA, and AG. As illustrated in Fig. 2.4b, a De Bruijn graph is constructed by representing these suffixes and prefixes as nodes and representing the corresponding  $k$ -mers (those having prefixes and suffixes as nodes in the graph) as edges. Therefore, rather than perform the computationally expensive process of finding a Hamiltonian cycle in a graph, modern assemblers prefer to identify an Eulerian cycle in De Bruijn graphs during genome reconstruction.

### 2.1.5 Hash Tables

The identification of repeats in a DNA sequence is critical for many applications, including the study of genome evolution and divulging the characteristics of different types of tumors. A hash table is a data structure that is commonly utilized to efficiently locate repeats within a sequence. Moreover, certain error correction techniques for next-generation sequencing, such as RACER [6], employ the use of hash tables to achieve efficient storage of  $k$ -mers. The basic idea behind hashing is simple. In a given collection of data, each data entry  $x$  is stored as a record in an array. The location of this record is computed using a hashing function  $h(x)$  that assigns each data entry to a unique integer that stands for a key to that particular location in the



**Fig. 2.5** Example hash tables

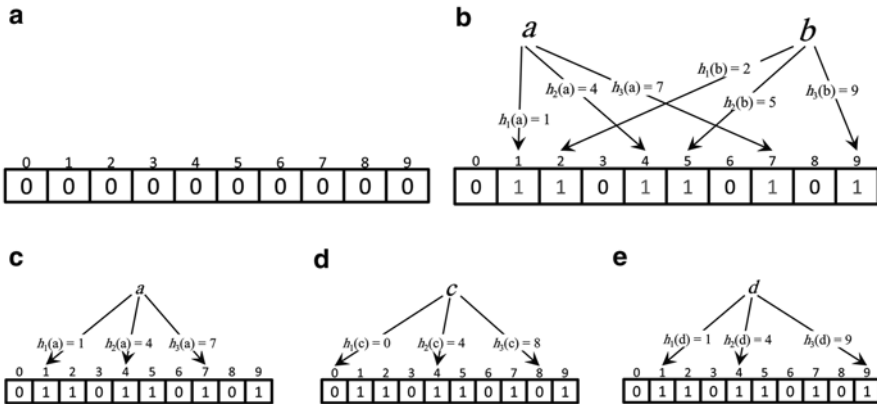
array. Accordingly, a hash table is used to preserve the set of keys that index different records in the array. Understandably, a hashing function would map similar data entries (or  $k$ -mers in the case of genome sequences) to the same index in the hash table. In other words, repeated  $k$ -mers are collected under a single slot (chained records) that is indexed using a unique key stored in a hash table [3].

Figure 2.5 shows a simple example of hashing functions. In this case, each entry ( $k$ ) is stored in the hash table at position  $(k \bmod n)$ , where  $n$  ( $=10$ ) represents the size of the hash table. A similar hashing function was used for storing  $k$ -mers in RACER [6].

### 2.1.6 Bloom Filters

Counting  $k$ -mers is a crucial preprocessing step for several bioinformatics applications such as genome/transcriptome sequence assembly, error correction techniques for next-generation sequence reads, and metagenomic sequencing. Several  $k$ -mer counting and abundance analysis software packages have been presented recently [7, 8]. In order to ensure rapid and memory-efficient counting of  $k$ -mers, these packages depend on memory-efficient data structures such as Bloom filters. A Bloom filter is an efficient probabilistic data structure that tells us whether an element is present or not in a data set. Bloom filters do not return false negatives. However, the efficiency of Bloom filters comes at the expense of a controlled amount of false positives. In other words, if a Bloom filter tells us that an element does not exist in a data set, we can be assured that the element is definitely not present. However, a Bloom filter may inaccurately indicate that an element is present in a particular data set when it does not.

Figure 2.6 illustrates how Bloom filters work. A Bloom filter is simply an array of bits that are initially set to zeroes as shown in Fig. 2.6a. To store an element in a Bloom filter, the element is hashed several times using multiple hashing functions



**Fig. 2.6** Example of a Bloom filter of three hashing function. (a) A Bloom filter initialized to zeros, (b) two elements are inserted into the Bloom filter, (c) an example of true positive, (d) an example of true negative, and (e) an example of false positive

to obtain different hash values. These hash values should lie in a range between 0 and the size of the Bloom filter. Bits located at positions indicated by the resulting hash values are set to 1. Figure 2.6b–e illustrates how elements are inserted in a Bloom filter of three hash functions, and shows examples of a true positive, true negative, and false positive, respectively.

## 2.2 Algorithms

Bioinformatics is a field of science that studies how to relate the principles of computer science to tackle important biological questions. This can be accomplished by transforming biological queries into computational models and searching for (or developing) efficient algorithms in terms of accuracy and complexity to broach these subjects. An algorithm is a step-by-step description of a procedure of calculation, data processing, or automated reasoning [9]. In this section, we briefly discuss how a popular biological problem such as biological sequence alignment can be formulated as a computer science problem. Additionally, we will introduce algorithms and give a brief overview of a commonly utilized computer science algorithm that has been applied to a variety of biological problems, i.e., the greedy algorithm.

### 2.2.1 Alignment Algorithms

A typical approach to understand the functionality of a newly discovered gene is to search for close matches in a previously stored database of known genes. In order to

**Fig. 2.7** Example of a scoring matrix

	-	A	C	G	T
-		-3	-4	-2	-1
A	-3	5	-1	-2	-1
C	-4	-1	5	-3	-2
G	-2	-2	-2	-3	-2
T	-1	-1	-2	-2	5

measure how close two genes are, they should be aligned with respect to each other so that the number of matches between corresponding characters in the aligned sequences is optimized. This biological problem is analogous to the well-known computer science problem of string edit distance, which aims to measure the distance between two strings through aligning or matching them up [9]. For instance, the following alignment of the two DNA strings  $x = \text{CTGCG}$  and  $y = \text{ACCGCT}$  show that the number of matches between them is 3.

- CT - GCG  
 AC-CGCT

The alignment score is calculated based on a scoring matrix that specifies the scores of matches, mismatches, insertions, and deletions. Figure 2.7 shows an example of a scoring matrix. According to this matrix, the alignment score of the above alignment is

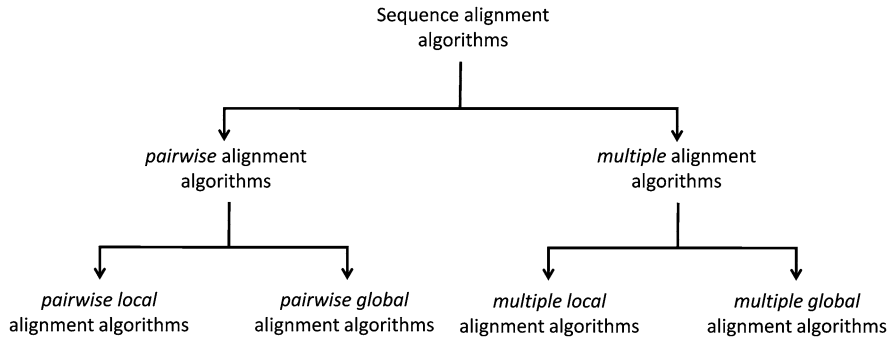
$$\delta(-, T) + \delta(C, C) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(G, T) = 7$$

There are two main classes of biological sequence alignments: *global* alignments and *local* alignments. In contrast to local alignments where only portions of sequences are aligned, the entire sequences are aligned in global alignments. Therefore, global alignments are useful for aligning closely related sequences whereas local alignments are more suitable when comparing distantly related sequences [4].

According to the number of sequences to be aligned, sequence alignment algorithms can be categorized into two categories; namely, *pairwise* alignment algorithms and *multiple* alignment algorithms. Pairwise alignment algorithms aim at finding the optimal alignment of only two sequences. On the other hand, the goal of multiple sequence alignment algorithms is to find the best alignment of three or more sequences. Figure 2.8 shows a general classification of sequence alignment algorithms.

For next-generation sequence reads, aligners take into account the application areas of next-generation sequencing technologies (e.g., metagenomics [10], cancer genomics [11], or analysis of mRNA expression [12]) as well as their unique characteristics (such as short read lengths, the large number of short reads to be mapped, and platform-dependent sequencing error rates) [13]. Therefore, such aligners have



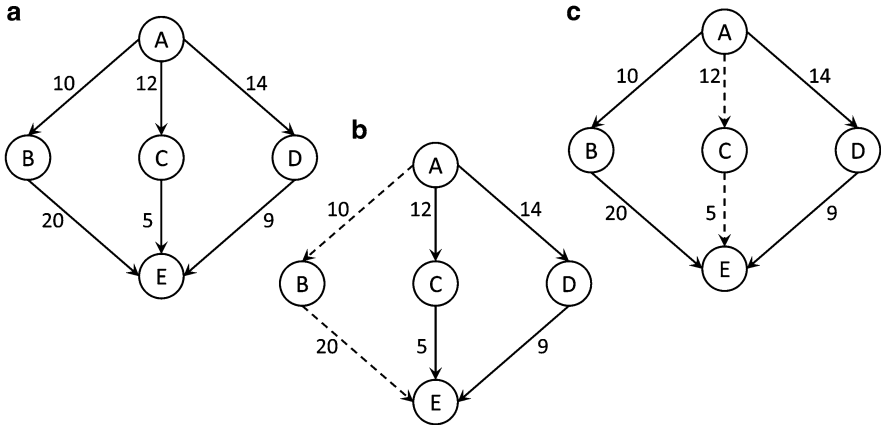


**Fig. 2.8** Classification of sequence alignment algorithms

extra features compared to general alignment techniques. For example, performing alignment for short reads generated from certain next-generation sequencing technologies is computationally more intensive than the alignment of longer reads. Several short read alignment algorithms and software packages have been proposed in the last few years. Out of these aligners, Novoalign [14], SHRiMP [15], Bowtie [16], SOAP [17], Burrows-Wheeler Alignment (BWA) [18], mrFAST [19], mrs-FAST [20] are among the most popular.

### 2.2.2 Greedy Algorithm

Although dynamic programming can find optimal solutions for many optimization problems such as the pairwise sequence alignment issue discussed in the previous subsection, it is not always the strategy of choice for a wide range of optimization problems, especially large-scale ones, due to its high computational cost. Fortunately, the greedy algorithm provides a viable alternative strategy with reduced computational requirements. The basic idea behind this strategy is to adopt the best (optimal) choice at each possible (local) stage in the hopes that a global optimum is reached at the final stage [1]. Due to the fact that it chooses the best local solution, the greedy algorithm is also called the best first search algorithm [21]. This can be explained using the example shown in Fig. 2.9. As may be inferred from the figure, the problem here is to find the shortest path from node A to node E. Since there is no direct path from A to E, an initial decision should be made regarding the move from A to one of the three nodes B, C, or D. As shown in Fig. 2.9b, the greedy algorithm choice (dashed line) for this local step is to move to B since the distance from A to B is the shortest (best) among the three choices. It should be noted that obtaining the optimal solution using the greedy strategy is not guaranteed. This is because the greedy algorithm considers only the stage at hand and does not look ahead to the following stages. It is clear from this example that the first choice made by the algorithm does not lead to the optimal solution shown in Fig. 2.9c. Taking the next



**Fig. 2.9** Problem of finding the shortest path from A to E (a) solution using the greedy algorithm (b) optimal solution (c)

stages into consideration, one should choose node C instead of B since the distance from A to E through C is 17 ( $12 + 5$ ) while the distance from A to E thorough B is 30 ( $10 + 20$ ). However, the greedy algorithm often finds near-to-optimal solutions in relation to several types of optimization problems [9].

Greedy algorithms are utilized in several applications in different areas of bioinformatics. These applications include genome rearrangement and locating regulatory motifs in DNA sequences [3]. Furthermore, several recently proposed alignment and assembly algorithms for next-generation sequence data greatly benefit from the greedy algorithm. In fact, the first short read genome assemblers are based on the greedy algorithm [22–28]. The notion of applying the greedy algorithm to the next-generation sequence assembly problem is straightforward. To decide which read or contig should be added to the current one, the read (or contig) are sorted according to their overlap scores and the one with highest score is chosen [29]. However, it should be noted that next-generation sequence assemblers based on the greedy algorithms can easily get stuck at local maxima, and therefore some researchers recommend employing greedy algorithms to simple genomes only when the available computer processing power is limited.

## References

1. Cormen TH, Leiserson EL, Rivest RL, Stein C (2009) Introduction to Algorithms (3rd. Ed.). MIT Press, Cambridge, MA
2. Sung WK (2010) Algorithms in Bioinformatics: A Practical Introduction. Chapman & Hall/CRC, Boca Raton, FL, USA
3. Jones N, Pevzner P (2004) An introduction to Bioinformatics Algorithms (Computational Molecular Biology). MIT Press, Cambridge, MA, USA

4. Chacko E, Ranganathan S (eds) (2011) Chapter10: Graphs in Bioinformatics. in Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications. John Wiley & Sons, Inc., Hoboken, NJ. doi:[10.1002/9780470892107.ch10](https://doi.org/10.1002/9780470892107.ch10)
5. Compeau PEC, Pevzner PA, Tesler G (2011) How to apply de Bruijn graphs to genome assembly. *Nat Biotech* 29 (11):987-991. doi:[10.1038/nbt.2023](https://doi.org/10.1038/nbt.2023)
6. Ilie L, Molnar M (2013) RACER: Rapid and accurate correction of errors in reads. *Bioinformatics*. doi:[btt407](https://doi.org/10.1093/bioinformatics/btt407)
7. Melsted P, Pritchard J (2011) Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC bioinformatics* 12 (1):1-7. doi:[10.1186/1471-2105-12-333](https://doi.org/10.1186/1471-2105-12-333)
8. Zhang Q, Pell J, Canino-Koning R, Chuang Howe CA, Brown T (under review) These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. Preprint arXiv:1309:2975. In review, PLoS One
9. Dasgupta S, Papadimitriou C, Vazirani U (2006) Algorithms. McGraw-Hill Science/Engineering/Math, Berkshire, UK, USA
10. Qin J, Li R, Raes J, Arumugam M, Burgdorf KS et al. (2010) A human gut microbial gene catalogue established by metagenomic sequencing. *Nature* 464 (7285):59-65. doi:[10.1038/nature08821](https://doi.org/10.1038/nature08821)
11. Guffanti A, Iacono M, Pelucchi P, Kim N, Solda G et al. (2009) A transcriptional sketch of a primary human breast cancer by 454 deep sequencing. *BMC Genomics* 10:163. doi:[10.1186/1471-2164-10-163](https://doi.org/10.1186/1471-2164-10-163)
12. Sultan M, Schulz MH, Richard H, Magen A, Klingenhoff A et al. (2008) A global view of gene activity and alternative splicing by deep sequencing of the human transcriptome. *Science* 321 (5891):956-960. doi:[10.1126/science.1160342](https://doi.org/10.1126/science.1160342)
13. Li H, Homer N (2010) A survey of sequence alignment algorithms for next-generation sequencing. *Brief Bioinform* 11 (5):473-483. doi:[10.1093/bib/bbq015](https://doi.org/10.1093/bib/bbq015)
14. Novocraft Technologies (2012) Novoalign. <http://novocraft.wordpress.com/2012/07/02/novoalign-v2-08-02-novoaligncs-v1-02-02-and-novosort-v1-0-released/>
15. Rumble SM, Lacroute P, Dalca AV, Fiume M, Sidow A et al. (2009) SHRiMP: accurate mapping of short color-space reads. *PLoS Comput Biol* 5 (5):e1000386. doi:[10.1371/journal.pcbi.1000386](https://doi.org/10.1371/journal.pcbi.1000386)
16. Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol* 10 (3):R25. doi:[10.1186/gb-2009-10-3-r25](https://doi.org/10.1186/gb-2009-10-3-r25)
17. Li R, Li Y, Kristiansen K, Wang J (2008) SOAP: short oligonucleotide alignment program. *Bioinformatics* 24 (5):713-714. doi:[10.1093/bioinformatics/btn025](https://doi.org/10.1093/bioinformatics/btn025)
18. Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25 (14):1754-1760. doi:[10.1093/bioinformatics/btp324](https://doi.org/10.1093/bioinformatics/btp324)
19. Alkan C, Kidd JM, Marques-Bonet T, Aksay G, Antonacci F et al. (2009) Personalized copy number and segmental duplication maps using next-generation sequencing. *Nat Genet* 41 (10):1061-1067. doi:[10.1038/ng.437](https://doi.org/10.1038/ng.437)
20. Hach F, Hormozdiari F, Alkan C, Birol I, Eichler EE et al. (2010) mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nat Methods* 7 (8):576-577. doi:[10.1038/nmeth0810-576](https://doi.org/10.1038/nmeth0810-576)
21. Russell S, Norvig P (2009) Artificial Intelligence: A Modern Approach (3rd Ed.). Prentice Hall, New Jersey, USA
22. Warren RL, Sutton GG, Jones SJ, Holt RA (2007) Assembling millions of short DNA sequences using SSAKE. *Bioinformatics* 23 (4):500-501. doi:[10.1093/bioinformatics/btl629](https://doi.org/10.1093/bioinformatics/btl629)
23. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ et al. (2009) ABySS: a parallel assembler for short read sequence data. *Genome research* 19 (6):1117-1123. doi:[10.1101/gr.089532.108](https://doi.org/10.1101/gr.089532.108)
24. Dohm JC, Lottaz C, Borodina T, Himmelbauer H (2007) SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res* 17 (11):1697-1706. doi:[gr.6435207](https://doi.org/10.1101/gr.6435207)

25. Jeck WR, Reinhardt JA, Baltrus DA, Hickenbotham MT, Magrini V et al. (2007) Extending assembly of short DNA sequences to handle error. *Bioinformatics* 23 (21):2942-2944. doi:[10.1093/bioinformatics/btm451](https://doi.org/10.1093/bioinformatics/btm451)
26. Reinhardt JA, Baltrus DA, Nishimura MT, Jeck WR, Jones CD et al. (2009) De novo assembly using low-coverage short read sequence data from the rice pathogen *Pseudomonas syringae* pv. *oryzae*. *Genome research* 19 (2):294-305. doi:[10.1101/gr.083311.108](https://doi.org/10.1101/gr.083311.108)
27. Kao WC, Chan AH, Song YS (2011) ECHO: a reference-free short-read error correction algorithm. *Genome research* 21 (7):1181-1192. doi:[10.1101/gr.111351.110](https://doi.org/10.1101/gr.111351.110)
28. Goldberg SM, Johnson J, Busam D, Feldblyum T, Ferriera S et al. (2006) A Sanger/pyrosequencing hybrid approach for the generation of high-quality draft assemblies of marine microbial genomes. *Proc Natl Acad Sci U S A* 103 (30):11240-11245. doi:0604351103
29. Miller JR, Koren S, Sutton G (2010) Assembly algorithms for next-generation sequencing data. *Genomics* 95 (6):315-327. doi:[10.1016/j.ygeno.2010.03.001](https://doi.org/10.1016/j.ygeno.2010.03.001)

Next Generation Sequencing Technologies and  
Challenges in Sequence Assembly

El-Metwally, M.Sc, S.; Ouda, O.M.; Helmy, M.

2014, XII, 118 p. 12 illus., 1 illus. in color., Softcover

ISBN: 978-1-4939-0714-4