Algorithms and Data Structures for Computational Biology

Math, Theory, and Practice for Large-Scale Biological Data Analysis

K. Jun Gao

May 4, 2023

© Copyright 2023 by K. Jun Gao

Contents

I Basic Mathematics **1**

1 Graphs and Combinatorics **3**

1.1 Graph Theory 3
1.2 Eulerian and Hamiltonian Circuits 7
1.3 Hall's Theorem 13
1.4 Partially Ordered Sets 15
1.5 Counting 16

2 Probability **23**

2.1 Review of Basic Probability Theory 23
2.2 Concentration Inequalities 25
2.3 Moment Generating Functions 28

II Information and Compression **31**

3 Measure of Information and Complexity **33**

3.1 Entropy 33
3.2 Entropy of Biological Sequences 34
3.3 Kolmogorov Complexity 36
3.4 Lempel-Ziv Complexity 38

4 Entropy Coding **43**

4.1 Worst-Case Entropy 43
4.2 Zero-Order Empirical Entropy 44
4.3 Symbol Codes 44
4.4 Lower Bounds 48

III Index Data Structures **51**

5 Suffix Tree **53**

5.1 Suffix Tries . . . 53

5.2 Ukkonen's Linear-Time Construction . . . 55

6 Suffix Array **59**

6.1 String Sorting . . . 59

6.2 Naive Construction From Suffix Tree . . . 64

6.3 A Divide-and-Conquer Approach . . . 65

6.4 Kärkkäinen-Sanders Algorithm . . . 65

7 Burrows-Wheeler Transform and FM Index **71**

7.1 Burrows-Wheeler Transform . . . 71

7.2 FM Index . . . 74

7.3 Bidirectional BWT Index . . . 80

IV Data in High-Dimensional Space **83**

8 Geometry of High-Dimensional Objects **85**

8.1 Most Volume of High-dimensional Objects is Near the Surface . . . 85

8.2 Most Points in a Unit Ball Are Nearly Orthogonal . . . 86

9 Comparing Data in High Dimension **91**

9.1 Johnson-Lindenstrauss Lemma and Random Projection . . . 91

9.2 Alignment-Free Sequence Comparison . . . 93

V Randomness and Randomization **99**

10 Markov Chain and Random Process **101**

10.1 Definitions . . . 101

10.2 Fundamental Theorem of Markov Chain . . . 103

10.3 The Metropolis-Hastings Algorithm . . . 107

10.4 Gibbs Sampling . . . 108

10.5 Hidden Markov Model . . . 110
10.6 The Viterbi Algorithm . . . 111

11 Random Graph Theory 113
11.1 Bulk Properties of Random Graphs . . . 113
11.2 Structures in Random Graphs . . . 114
11.3 Phase Transitions in Random Graphs . . . 115
11.4 Modeling Protein-Protein Interaction Network . . . 117

12 Hashing 119
12.1 Hash Functions . . . 119
12.2 Strong Universality (2-Independence) . . . 119
12.3 Finite Fields . . . 121
12.4 Universal Hashing of Variable-Length Strings . . . 122
12.5 Applications of Hashing . . . 122
12.6 Locality Sensitive Hashing . . . 124

13 Probabilistic Sampling and Sketching 131
13.1 Frequency Moments . . . 131
13.2 Distinct Elements . . . 131
13.3 Second Frequency Moment Sketch . . . 136
13.4 Majority Element and Misra-Gries . . . 136
13.5 CountMin Sketch . . . 138

Bibliography 142

Index 142

Part I

Basic Mathematics

Chapter 1

Graphs and Combinatorics

Graphs are ubiquitous in computational biology. Its application ranges from assembly graph in genome assembly to interaction networks in systems biology. Many algorithms in computational biology involves the use of graph and concepts from graph theory. In this chapter, will introduce some basic definitions and some important results from graph theory.

1.1 Graph Theory

We begin by providing a formal definition of graphs.

Definition 1.1 (Undirected Graph)**.** *An* ***undirected graph*** *is a pair (V, E) where V is a finite set, and E is a collection of subsets of V of size 2. The set V is called the vertex set, and members of V are called* ***vertices****. The set E is called the edge set and members of E are called* ***edges****.*

Similarly, we can define a directed graph as graphs where the edges are directed. Formally,

Definition 1.2 (Directed Graph)**.** *A* ***directed graph*** *is a pair (V, E) where V is a finite set, and E is a collection of ordered pairs of V of size 2 ($E \subseteq V \times V$).*

We generally use G to denote a graph. For $x, y \in V$, the edge between x and y is denoted $\{x, y\}$, or sometimes xy or yx. The set notation $\{\cdot\}$ is to highlight the unordered nature of an undirected edge. If $x, y \in V$ and $\{x, y\} \in E$, we say that x and y are **adjacent**.

1.1.1 Classic Graphs

Definition 1.3 (Complete Graph)**.** *For $n \geq 1$, we let K_n denote the complete graph $([n], E)$ where $E = \{e \subseteq [n] \mid |e| = 2\}$.*

Conversely, if for all distinct $x, y \in V$, $\{x, y\} \notin E$, the graph is called an independent graph, denoted I_n.

Definition 1.4 (Path Graph)**.** *For $n \geq 1$, we let P_n denote the path graph $([n], E)$ where $E = \{\{i, i+1\} \subseteq [n] \mid i \in [n]\}$.*

Definition 1.5 (Cycle Graph)**.** *For $n \geq 3$, we let C_n denote the cycle graph $([n], E)$ where $E = \{\{0, 1\}, \{1, 2\}, \ldots, \{n-2, n-1\}, \{n-1, 0\}\}$.*

1.1.2 Subgraphs

Definition 1.6 (Subgraph)**.** *Given a graph $G = (V, E)$, a* ***subgraph*** *H of G is a pair $H = (W, E')$ such that $W \subseteq V$ and $E' \subseteq \{s \in E \mid s \subseteq W\}$.*

Sometimes, we abuse notation and write $H \subseteq G$.

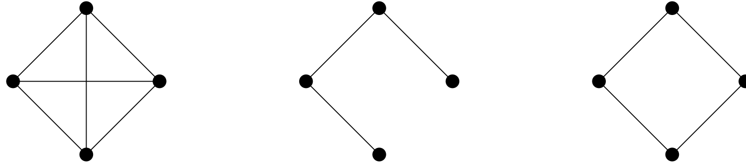


Figure 1.1: From left to right: complete graph, path graph, cycle graph with 4 vertices.

Definition 1.7 (Induced Subgraph). *Given a graph $G = (V, E)$, an **induced subgraph** $H \subseteq G$ is one of the form (W, E') where*

$$E' = \{\{x, y\} \in E \mid \{x, y\} \in W\}$$

for $W \subseteq V$.

Note that induced subgraphs are uniquely defined by the vertex set whereas a subgraph is not. A subgraph need not contain all edges between the vertices in the subgraph.

1.1.3 Isomorphism

Definition 1.8 (Isomorphic Graphs). *Let $G = (V_1, E_1)$ and $H = (V_2, E_2)$ be graphs. We say that G and H are isomorphic if there is a bijection $f : V_1 \to V_2$ such that $\{x, y\} \in E_1 \iff \{f(x), f(y)\} \in E_2$.*

Theorem 1.9. *Let G be a graph with n vertices. G is isomorphic to a subgraph of K_n.*

Proof. Consider $G = (V, E)$. Label the vertices such that $V = \{v_0, \ldots, v_{n-1}\}$. Let $f : V \to [n]$ be a function such that $f(v_i) = i$. Consider the set $E' = \{\{i, j\} \subseteq [n] \mid \{v_i, v_j\} \in E\}$. Clearly, $H = ([n], E')$ is a subgraph of K_n by definition. Further, we claim that f is an isomorphism between G and H.

To show that f is an isomorphism, it suffices to show that $\{v_i, v_j\} \in E \iff \{f(i), f(j)\} \in E'$. For the forward direction, suppose $\{v_i, v_j\} \in E$, then $\{i, j\} \in E'$ by construction. This immediately implies that $\{f(v_i), f(v_j)\} \in E'$ by construction of f. For the reverse direction, suppose $\{i, j\} \in E'$, which is equivalent to $\{f(v_i), f(v_j)\} \in E'$. By definition of f, this is only true when $\{v_i, v_j\} \in E$. Therefore, f is an isomorphism between G and H.

It follows that G is isomorphic to H, which is a subgraph of K_n. □

1.1.4 Degree

Definition 1.10 (Degree). *Given a graph $G = (V, E)$ and a vertex $v \in V$, we let $\deg_G(v) = |\{u \in V \mid \{u, v\} \in E\}|$, which is the number of vertices ahring an edge with v.*

Lemma 1.11 (The Handshaking Lemma). *Let $G = (V, E)$ be a graph, and let $|E| = e$. Then,*

$$\sum_{v \in V} \deg_G(v) = 2e$$

Proof. For each $v \in V$, let $E_V = \{\{v, u\} \mid \{v, u\} \in E\}$. Notice that for distinct u, v, $|E_u \cap E_v| \leq 1$ and is equal to one exactly when u and v form an edge. No three distinct sets overlap. It follows that

$$e = |E| = \left| \bigcup_{v \in V} E_v \right| = \sum_{v \in V} |E_v| - \sum_{u,v \in V} |E_v \cap E_u| = \sum_{v \in V} \deg_G(v) - e$$

This implies $\sum_{v\in V} \deg_G(v) = 2e$. □

For directed graphs, we define ***in-degree*** and ***out-degree*** separately. We use $\deg_+$ to denote the in-degree (i.e. number incoming edges) and $\deg_-$ to denote the out-degree (i.e. number of outcoming edges).

1.1.5 Connectivity

Definition 1.12 (Path). *A **path** in a graph $G = (V, E)$ is a sequence of **distinct** vertices $x_1, \ldots, x_n$ that are each successively connected with an edge. We call the number of edges in the sequence the length of the path.*

Definition 1.13 (Walk). *A **walk** in a graph $G = (V, E)$ is a sequence of vertices that are each successively connected with an edge.*

Note that a walk does not have the distinctness requirement, meaning repeating vertices in a walk is allowed.

Definition 1.14 (Connectivity). *We call a graph G connected if every pair of distinct vertices is connected by a path.*

Definition 1.15 (Connected Component). *Given a graph G, we call a maximally connected vertex set a **connected component**. More formally, if we denote the equivalence relation $\sim$ on V given by*

$$v \sim w \iff v = w$$

*(i.e. v is connected to w), the **connected components** are the equivalence classes given by this relation.*

Now that the edges are directed, the original definition of connectedness is no longer symmetric and thus is no longer an equivalence relation. For directed graphs, there are two different notion of connectivity.

Definition 1.16 (Strongly and Weakly Connected). *A directed graph D is **weakly connected** if $\forall x, y \in V$, there exists either an x, y-walk or a y, x-walk. D is **strongly connected** if $\forall x, y \in V$, $\exists$ both a walk from x to y and a walk from y to x.*

1.1.6 Trees and Spanning Trees

Definition 1.17 (Tree). *A **tree** is a graph $G = (V, E)$ with the property that every two vertices are connected by a **unique** path.*

Lemma 1.18. *Every acyclic and connected graph G with $n \geq 2$ vertices has a leaf $v \in V$ such that* $\deg(v) = 1$.

Proof. Let G be an acyclic and connected graph. Note that since G is connected, $\deg_G(v) \geq 1$ for all $v \in V$. We prove the contrapositive. Suppose for all $v \in V$, $\deg_G(v) > 1$. Fix a vertex v. Starting from v, follow a sequence of distinct edges until a vertex repeats. It is possible to visit all vertices without repeating edges because each vertex has two incident edges. Further, because every vertex has degree of at least 2, once we have visited every vertex, there should still be at least one edge unvisited. When we follow that edge, we will arrive at a vertex that has previously been visited. This creates a cycle. So, G is not acyclic. □

Theorem 1.19. *Let G be a **connected graph** with n edges. The following are equivalent:*

1. *G is a tree*
2. *G does not contain a cycle*

3. G has $n-1$ edges

Proof. Let G be a connected graph.

($1 \implies 2$): Assume that G is a tree. We show that G does not contain a cycle by contradiction, so suppose not and G contains a cycle. Let the cycle be $c = x_1x_2\dots x_n$. Notice that $x_1x_2\dots x_n$ is a path from x_1 to x_n. Since c is a cycle, $\{x_1, x_n\} \in E$. So, x_1x_n is also a path from x_1 to x_n. However, this is a contradiction to the path uniqueness requirement in the definition of a tree. So G does not contain a cycle.

($2 \implies 3$): We prove this implication by induction. Assume G is acyclic and connected.

Base Case: $n = 1$. There is $0 = n-1$ edge. The implication holds.

Inductive Step: Let $n \in \mathbb{N}$ and $n \geq 2$. As induction hypothesis, suppose all acyclic and connected graphs with $m \leq n$ vertices have $m-1$ edges. Let G be a graph with $n+1$ vertices. Suppose that G is also connected and acyclic. By Lemma 1.18, G has at least one vertex v such that $\deg_G(v) = 1$. Construct G' by removing v and the one edge incident to v. G' has n vertices, so by induction hypothesis, has $n-1$ edges. If we add back the removed edge, we can see that G has n edges.

By induction, this implication holds.

($3 \implies 1$): Let G be a connected graph with $n-1$ edges. To show that G is a tree, we need to show that every two vertices are connected by a unique path. It suffices to show that deleting any edge from G leaves the graph disconnected. Suppose for contradiction that deleting an edge $\{u, v\}$ does not disconnect the graph. This implies that there is another path from u to v containing at least 2 edges. This also implies that G is connected with $n-2$ edges. But this is impossible because the minimum number of edges in a connected graph with n vertices is $n-1$. Therefore, every pair of vertices is connected via a unique path since deleting any edge breaks this path and thus disconnects the graph. By definition, this means G is a tree. □

Definition 1.20. *Let $T = (V, E)$ be a tree. We call a vertex v a **leaf** if $\deg_T(v) = 1$.*

Lemma 1.21. *If $G = (V, E)$ is a tree with $n \geq 2$ vertices, then G has at least two leaves.*

Proof. We prove this lemma by strong induction on n for $n \geq 2$.

Base Case: $n = 2$. Since T is a tree, T is connected and has exactly one edge. Clearly, both vertices in T have degree 1.

Inductive Step: Suppose, as inductive hypothesis, that for some $n \geq 2$, every tree with $2 \leq k \leq n$ vertices has at least two leaves. Let T be a tree with $n+1$ vertices. Pick an edge $\{u, v\} \in E$ and form a new graph T' by deleting $\{u, v\}$. More formally, $T' = (V, E - \{\{u, v\}\})$. Since T is a tree, there is no other path between u and v (otherwise we would have a cycle). It follows that in T', u and v are disconnected. Deleting an edge will not create a cycle either, so T' is a forest with two connected components T_u and T_v. Consider the following cases:

Case 1: Both T_u and T_v contain at least 2 vertices. Then, we can apply our induction hypothesis. Take one leaf from each component. If both are endpoints of the deleted edge $\{u, v\}$, they are no longer leaves in T, but we still have at least two leaves, one from T_u and one from T_v. In general, T_u must have a leaf x that is not u and T_v must have a leaf y that is not v. Adding $\{u, v\}$ back does not affect x and y so they will remain leaves, which implies that there exist at least two leaves.

Case 2: If each component of T' has only one vertex, then T is isomorphic to K_2, which has two leaves.

Case 3: If exactly one of the components has only one vertex, then it must become a leaf when we reconnect $\{u, v\}$ to form T.

In all cases, we have that T has at least two leaves. By induction, every tree with at least 2 vertices has at least two leaves. ☐

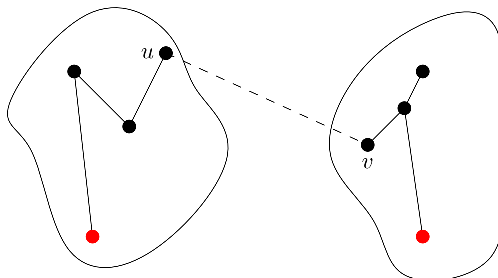


Figure 1.2: In each of the two components, there exists at least one leaf that is not u or v (colored red). When we reconnect $\{u, v\}$, those vertices will remain leaves.

1.1.7 Spanning Trees

Definition 1.22 (Spanning Subgraph)**.** *Given a graph $G = (V, E)$, we call a subgraph $H \subseteq G$ **spanning** if $H = (V, E')$.*

Definition 1.23 (Spanning Tree)**.** *Given a graph G, a **spanning tree** is a spanning subgraph $T \subseteq G$ that is a tree.*

Theorem 1.24. *Every connected graph has a spanning tree.*

Proof. By induction on the number of edges.

Base Case: If G is connected and has no edges, G contains one single vertex. G is trivially a tree and a spanning tree.

Inductive Step: Suppose G has $m \geq 1$ edges. If G is a tree, then we are done because it is trivially a spanning tree. Otherwise, G has cycles. For each cycle, remove an edge from the cycle to disconnect the cycle. By definition of a cycle, the graph is still connected after the removal of edges from the cycles. The resulting graph is still connected but has no cycle, which by Theorem 1.19, means the resulting graph is a tree. By definition of a spanning tree, this means the resulting graph is a spanning tree.

By induction, a connected graph with n edges has a spanning tree for all $n \in \mathbb{N}$. This implies that all connected graphs have a spanning tree. ☐

1.2 Eulerian and Hamiltonian Circuits

1.2.1 Path, Walk, Trail, and Circuits in a Graph

Recall the following definition of path and walks.

Definition 1.25 (Path)**.** *A **path** in a graph $G = (V, E)$ is a sequence of **distinct** vertices $x_1, \ldots, x_n$ that are each successively connected with an edge. We call the number of edges in the sequence the length of the path.*

Definition 1.26 (Walk)**.** *A **walk** in a graph $G = (V, E)$ is a sequence of vertices that are each successively connected with an edge.*

In addition, we define a trail and a circuit in a graph as follows.

Definition 1.27 (Trails)**.** *A **trail** is a walk with no repeating edges.*

Definition 1.28 (Circuits)**.** *A **circuit** in a graph $G = (V, E)$ is a sequence of distinct vertices $v_1, \ldots, v_k$ such that for all $i \in \{1, \ldots, k-1\}$, $\{v_i, v_{i+1}\} \in E$ and $\{v_k, v_1\} \in E$. Note, a circuit induces a copy of a subgraph isomorphic to C_k for $k \geq 3$. Specially, we define a **single vertex** a circuit as well.*

In other words, circuit is a closed (starts and ends with the same vertex) trail.

1.2.2 Eulerian Circuit

There are two special types of circuits that we would like to characterize. They are the Eulerian circuits and Hamiltonian circuit.

Definition 1.29 (Eulerian Circuit)**.** *Given a graph $G = (V, E)$, an **Eulerian circuit** is a sequence of vertices $x_0, \ldots, x_t$ such that*

- $x_0 = x_t$
- $\forall i \in [t].\, \{x_i, x_{i+1}\} \in E$
- $\forall e \in E.\, \exists$ *unique* $i \in [t].\, e = \{x_i, x_{i+1}\}$ *(i.e. every edge appears exactly once in an Eulerian circuit)*

We say a graph is ***Eulerian*** if and only if it has an **Eulerian circuit**. An Eulerian circuit is also referred to as an **Euler tour** in some texts. The notion of an Eulerian circuit and graph appears in the famous problem of the **bridges of Königsberg**.

In 1736, Euler gave his famous characterization of an Eulerian graph, stated as follows

Theorem 1.30 (Euler, 1736)**.** *A connected graph $G = (V, E)$ is Eulerian if and only if all its vertices have even degree.*

Proof. The forward direction of the proof is quite straightforward whereas the reverse direction requires a slightly more involved proof by induction.

($\implies$): Let G be a connected graph. Assume that G is Eulerian so it must have an Eulerian circuit. Note that in an Eulerian circuit, every time we enter a vertex, we must also leave the vertex. This is the case for all vertices because otherwise we would have an infinite graph. Hence, all vertices in G must have even degree.

($\impliedby$): Let G be a graph. We proceed by strong induction on the number of edges.

Base Case: G is a graph with $m = 0$ edge. The result trivially holds.

Inductive Step: Let $m \in \mathbb{N}$ be arbitrary. Assume that for all $k \in \mathbb{N}$ such that $0 \leq k < m$, the implication holds. Let $G = (V, E)$ be a connected graph with m edges. Further, assume that $\deg_G(v)$ is even for all $v \in V$. Since the graph is connected and every vertex has even degree, it follows immediately that $\deg_G(v) \geq 2$ for all $v \in V$. This also implies that G contains a cycle. Let $c = v_1 \ldots v_k$ be such cycle of maximal length and E' be the edges contained in this cycle.

If c contains all edges exactly once, we are done. Hence, suppose $E' \neq E$ and consider the graph $G' = (V, E \setminus E')$. It has connected components $S_1, \ldots, S_l$. Since $E' \neq E$, each of the connected components S_i contains strictly fewer edges than $|E| = m$. For every $v \in G$, an even number of edges of G at v are in the cycle c, so we we remove these edges, each vertex in the remaining graph should still have even degree. Apply the induction hypothesis to the components, which asserts that each of the components $S_1, \ldots, S_l$ possess an Eulerian circuit. Further, since c is a cycle, $C = (\{v_1, \ldots, v_k\}, E')$ itself is also Eulerian. Now, we recursively construct an Eulerian circuit, say x, in the original graph G. Start from v_1, find the component S_i containing v_1, and concatenate the Eulerian tour in S_i to x. Next, move from v_1 to v_2 along $\{v_1, v_2\} \in E'$. If $\{v_1, v_2\} \notin E$, then they must have been in the same connected component, in which case we skip v_2 and move to v_3. Repeat this until we have walked through every edge in each one of the l connected components and the edges in E' connecting each component. It is clear that x is Eulerian since it visits every edge exactly once.

By induction, the implication holds. □

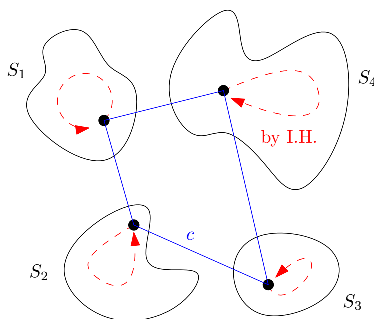


Figure 1.3: Construct an Eulerian circuit in the original graph G by first finding a cycle c, remove the cycle, find an Eulerian circuit within each component $S_1, \ldots, S_l$, and concatenate these circuits via the cycle c.

1.2.3 Hamiltonian Circuit

Definition 1.31 (Hamiltonian Graph)**.** *We call a graph with n vertices **Hamiltonian** if it admits a circuit of length n, which is to say that the graph has a spanning subgraph that is isomorphic to C_n.*

A sufficient condition for Hamiltonian graphs.

Theorem 1.32 (Dirac, 1952)**.** *Let $G = (V, E)$ be a graph with n vertices where $n \geq 3$. Suppose that for every $v \in V$, $\deg_G(v) \geq \lceil \frac{n}{2} \rceil$. Then, G is Hamiltonian.*

Proof. Let $G = (V, E)$ be a graph with $n \geq 3$ vertices. Assume that $\deg_G(v) \geq \lceil \frac{n}{2} \rceil$ for all $v \in V$. Let $\delta(G)$ denote the minimum degree. That is, $\delta(G) = \min\{\deg_G(v) \mid v \in V\}$. Then, the assumption is equivalent to that $\delta(G) \geq \lceil \frac{n}{2} \rceil$.

We claim that G is connected. We prove the claim by contradiction. So suppose not, consider the component $G' = (V_{G'}, E_{G'})$ of G with the fewest number of vertices. Since $\delta(G) \geq \lceil \frac{n}{2} \rceil$, each vertex is connected to at least $\lceil \frac{n}{2} \rceil$ other vertices. Since C is a component that is not connected to vertices in other components, $|V_{G'}| \leq \lceil \frac{n}{2} \rceil$. But then, $\delta(G') < |V_C| \leq \lceil \frac{n}{2} \rceil$, which contradicts the assumption that $\deg_G(v) \geq \lceil \frac{n}{2} \rceil$ for all $v \in V$, including those in G'.

Since G is connected, we can find the longest path in G. Let $P = v_0 \dots v_k$ be a longest path in G of length k (with k edges and $k+1$ vertices). We claim that there exists some $0 \le i \le k-1$ such that $\{v_0, v_{i+1}\} \in E$, $\{v_i, v_k\} \in E$, and $\{v_i, v_{i+1}\} \in E$ as shown in Figure 1.4.

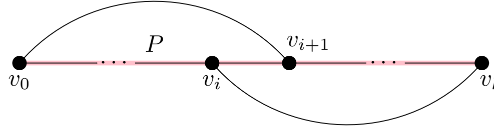


Figure 1.4: The longest path $P = v_0 \dots v_k$ with $k+1$ vertices is colored in red. There exists some $0 \le i \le k$ such that $\{v_0, v_{i+1}\} \in E$ and $\{v_i, v_k\} \in E$.

Such adjacent vertices v_i and v_{i+1} such that v_i is adjacent to v_k and v_{i+1} is adjacent to v_0 must exists. By way of contradiction, suppose v_i and v_{i+1} do not exist. Then, for every vertex adjacent to v_0, there must exists some vertex adjacent to it that is NOT adjacent to v_k. Similarly, for every vertex adjacent to v_k, there must exists some adjacent vertex that is NOT adjacent to v_0. Note that these two sets of vertices are disjoint and do not include v_k. This implies that

$$\deg_G(v_0) + \deg_G(v_k) + 1 \le k + 1$$

since we are not overcounting and the number of vertices being counted is at most the length of path P. The additional 1 on the LHS of the inequality came from couting v_k as it is not included in either $\deg_G(v_0)$ or $\deg_G(v_k)$. Now, since $\deg_G(v) \le \lceil \frac{n}{2} \rceil$ for all $v \in V$,

$$n + 1 \le \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + 1 \le \deg_G(v_0) + \deg_G(v_k) + 1 \le k + 1$$

so $n+1 \le k+1$. This implies that $n < k+1$ since both n and k are integers. But this leads to a contradiction because the number of vertices on the path P cannot be more than the total number of vertices in the entire graph.

The existence of such v_i and v_{i+1} allows us to construct a cycle $C = v_0 \to v_{i+1} \rightsquigarrow_P v_k \to v_i \rightsquigarrow_P v_0$. We claim this cycle is Hamiltonian.

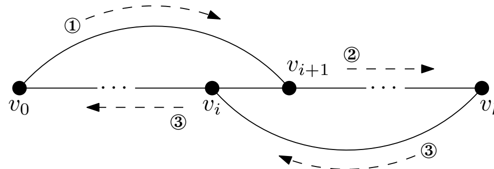


Figure 1.5: The Hamiltonian path is $C = v_0 \to v_{i+1} \rightsquigarrow_P v_k \to v_i \rightsquigarrow_P v_0$.

To see why C is Hamiltonian, again we use a contradiction proof. Suppose C is not Hamiltonian. Then, by definition, there must be some vertex $w \in V$ such that w is not on C. But since G is connected, w must be adjacent to some vertices, say $v_w \in V$. Without loss of generality, suppose that this v_w is on the cycle C. There must also be a v_{w+1} immediately adjacent to v_w. By construction, the cycle C contains $k+1$ edges (that's all edges on P along with $\{v_0, v_{i+1}\}, \{v_i, v_k\}$ and without $\{v_i, v_{i+1}\}$). We then consider the path from v_w to v_{w+1} by following the edges on the cycle. This leads to a path of length k, namely $p = v_w \rightsquigarrow v_0 \to v_{i+1} \rightsquigarrow v_k \to v_i \rightsquigarrow v_{w+1}$. Now, we extend the left end of this path to w since w is adjacent to v_w and still get back a valid path. The new path $P' = w \to v_w \rightsquigarrow v_0 \to v_{i+1} \rightsquigarrow v_k \to v_i \rightsquigarrow v_{w+1}$ is one edge longer than p. However, this contradicts the maximality assumption for P since now we would have a path, P', that contains more vertices than P. Therefore, C is indeed a Hamiltonian path.

It follows immediately that G is Hamiltonian by definition. □

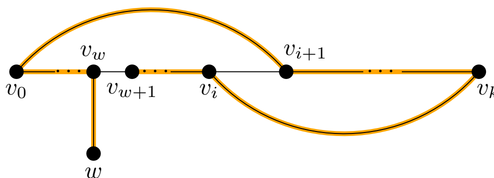


Figure 1.6: With the existence of a vertex w outside of the previously constructed cycle, we would have a longer path (colored in orange), contradicting the maximality of P.

1.2.4 Graph Coloring

Definition 1.33 (Independent Set)**.** *Given a graph $G = (V, E)$, we say $A \subseteq V$ is **independent** if and only if no no vertices in A are adjacent.*

The only independent sets in K_n are singletons. We can prove this using a contradiction and the definition of a complete graph and independent set.

Definition 1.34 (Bipartite Graph)**.** *We say a graph $G = (V, E)$ is **bipartite** if and only if we can partition V into two **disjoint independent** sets.*

A bipartite graph $(V_1 \cup V_2, E)$ is a complete bipartite graph iff every $v_1 \in V_1$ is connected to every vertex in V_2 and vice versa. We denote a complete bipartite graph by $K_{|V_1|,|V_2|}$ (K with a subscript denoting the size of the left and right partition, respectively).

Theorem 1.35. *A graph is bipartite if and only it does not contain a circuit of odd length.*

Proof.

($\implies$): Let $G = (V, E)$ be a bipartite graph. In particular, $V = A \cup B$ for some $A, B \subseteq V$ such that $A \cap B = \emptyset$ and for all $\{a, b\} \in E$, $a \in A$ and $b \in B$. Suppose, for contradiction, that G contains an odd-length cycle $C = v_1 v_2 \dots v_n v_1$ of length n. Without loss of generality, suppose that v_i and v_{i+1} alternates between A and B. So, $v_1 \in A$, $v_2 \in B$, $v_3 \in A$, and so on. If the cycle is not in that particular order, we can reindex the vertices and still have the same cycle.

Then, for $k \in \{1, 2, 3, \dots, n\}$,

$$v_k \in \begin{cases} A & k \text{ is odd} \\ B & k \text{ is even} \end{cases}$$

Since C is a cycle of odd length, n is odd. It follows that $v_n \in A$. But then, since $v_1 \in A$ and $\{v_n, v_1\} \in E$, this is a contradiction to the assumption that G is bipartite.

($\impliedby$): Let $G = (V, E)$ be a graph. Without loss of generality, assume that G is connected. Otherwise, we can consider the connected components individually. Assume that G contains no odd cycle. Let $w \in V$ be a vertex in G.

Let A be the set of vertices whose shortest distance from w is even, and let B be the set of vertices whose shortest distance from w is odd. That is,

$$\begin{aligned} A &= \{v \in V \mid d(v, w) \equiv 0 \mod 2\} \\ B &= \{v \in V \mid d(v, w) \equiv 1 \mod 2\} \end{aligned}$$

Since G is connected, every vertex is either at an even distance or odd distance from $w \in V$. A vertex cannot be both at an even distance and an odd distance from w at the same time. Hence, $A \cup B = V$ and $A \cap B = \emptyset$. This implies that A and B are a valid partition of V.

Now, we would like to show that G is bipartite. It suffices to show for all vertices $a_1, a_2 \in V$ and $b_1, b_2 \in B$, $\{a_1, a_2\} \notin E$ and $\{b_1, b_2\} \notin E$. To prove this fact, we suppose the contrary and derive a contradiction. So, suppose that there does exist such $x, y \in A$ or $x, y \in B$ such that $\{x, y\} \in E$. Fix such x, y. We can assume that $x \neq y \neq w$. Otherwise, we have $w = x$ and $d(x, w) = 0$. Since x and y are in the same partition, $d(y, w)$ is even and $d(y, x) = 0$. However, this is not possible since $d(y, x) = 1$, which is odd. By a similar argument, we can show that $w \neq y$ either.

To obtain a contradiction, we consider the shortest path from x to w and the shortest path from y to w. Let p be the shortest path from x to w, and let q be the shortest path from y to w. Let z be the last common vertex of p and q. Note that z may be w. We also note that $|p|$ and $|q|$ have the same parity since we assumed that y, x are in the same partition.

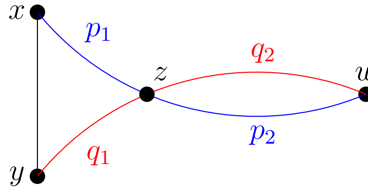


Figure 1.7: p is the shortest path from x to w. q is the shortest path from y to w. p_1 is the part of p from x to the last common vertex of p and q. Similarly, q_1 is the part of q from y to the last common vertex of p and q.

Let $p_1 = x \rightsquigarrow_p z$ be the part of the path p from x to z. Similarly, let $p_2 = z \rightsquigarrow_p w$, $q_1 = y \rightsquigarrow_q z$, and $q_2 = z \rightsquigarrow_q w$. We claim that $|p_2| = |q_2|$ since otherwise we can obtain a shorter path from x to w or from y to w. Further, we claim that $|p_1|$ and $|q_1|$ have the same parity becuase $|p|$ and $|q|$ have the same parity and the second part of both paths, p_2 and q_2, are of the same length. Recall that $\{x, y\} \in E$. Then, $C = x \rightsquigarrow_{p_1} z \rightsquigarrow_{q_1} y \rightarrow x$. Since $|p_1|$ and $|q_1|$ have the parity, $|C| = |p_1| + |q_1| + 1$ is odd. This is becuase $|p_1| + |q_1|$ can be expressed as $2k$ for some $k \in \mathbb{Z}$. This is an odd-length cycle, which is a contradiction to our initial assumption that G has no odd cycle. The only additional assumption leading to this contradiction is that G is not bipartite. Hence, G must be bipartite. □

1.2.5 Coloring

Definition 1.36 (Proper Coloring)**.** *Let $G = (V, E)$ be a graph. A (proper)* ***coloring*** *of G is a function $\phi : V \rightarrow [k]$ such that for all $i \in [k]$, $\phi^{-1}(i)$ is independent. Equivalently, ϕ is a (proper) coloring of G iff $\forall \{a, b\} \in E. \phi(a) \neq \phi(b)$. We call ϕ a k-coloring.*

Definition 1.37 (Chromatic Number)**.** *The* ***chromatic number*** *of a graph G, is the smallest k such that there is a proper k coloring of G. The chromatic number of G is denoted by $\chi(G)$.*

Theorem 1.38. *A graph is 2-colorable if and only if it does not contain an odd-length cycle.*

Proof. Theorem 1.35 states that a graph is bipartite iff there is no odd-length cycle. To prove this theorem, it suffices to prove that a graph is 2-colorable if and only if the graph is bipartite.

($\Longrightarrow$): Let $G = (V, E)$ be a 2-colorable graph. Take the coloring. Assign vertices with one color to V_1 and vertices with another color to V_2. V_1 and V_2 is a partition of V. By definition of a 2-color, for all $x, y \in V_1$, $\{x, y\} \notin E$ and for all $x, y \in V_2$, $\{x, y\} \notin E$.

($\Longleftarrow$): Let $G = (V, E)$ be a bipartite graph where $V = V_1 \cup V_2$ is a partition. Assign one color to all vertices in V_1 and assign another color to all vertices in V_2. It is easy to prove that this is a valid 2-coloring directly from the definition of a bipartite graph. □

1.2.6 Matching

Definition 1.39 (Matching)**.** *A matching M of G is a subgraph that is a disjoint union of edges.*

We say M is ***maximal*** if for all matchings M' such that $M' \subseteq M$, $M = M'$ (in other words, we cannot expand the matching). We say M is ***maximum*** if for all matchings M', $|E(M)| \geq |E(M')|$.

Definition 1.40 (Perfect Matching)**.** *A **perfect matching** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, the neighborhood of X, denoted $N(X)$ is*

$$N(X) = \{y \in V \mid (x, y) \in E \text{ for some } x \in X\}$$

For a vertex $v \in V(G)$, we say that v is ***saturated*** by M if $v \in V(M)$. Otherwise, if $v \in V(G) \setminus V(M)$, we say that v is ***unsaturated***. If every vertex is saturated by M, then M is a perfect matching.

Definition 1.41 (Alternating Path)**.** *Let M be a matching in G. An **alternating path** $P \subseteq G$ is a path where edges alternate between M and $G \setminus M$. An alternating path is an M-**augmenting path** if the end points of the graph are unsaturated by M.*

Lemma 1.42. *If $M \subseteq G$ has an augmenting path, then M is not maximum.*

Theorem 1.43 (Berge's Lemma)**.** *M is a maximum matching if and only if it has no augmenting paths.*

Proof.
(maximum $\implies$ no augmenting path): Consider the contrapositive. The claim holds by Lemma 1.42.

(no augmenting path $\implies$ maximum): Consider the contrapositive. Then, suppose M is not maximum. By definition, this means we have some M' such that $|M'| > |M|$. Let $H = M \triangle M' = (M \cup M') \setminus (M \cap M') = (M \setminus M') \cup (M' \setminus M)$. We claim that $\deg_H(v) \leq 2$ for all $v \in H$. This is because M and M' are matchings and every vertex is incident to at most one edge from each of M and M'. Then, H is a union of paths and cycles with edges alternating between $M \setminus M'$ and $M' \setminus M$. Every cycle in H has the same number of edges from M and M'. Since $|M'| > |M|$, H must contain a component with more edges of M' than of M. This component is a path that starts and ends with an edge in M'. Hence, H is an M-augmenting path. □

1.3 Hall's Theorem

The question is: when does a bipartite graph have a perfect matching. This is exactly what Hall's theorem answers. Hall's theorem establishes the necessary and sufficient condition for a perfect matching in bipartite graphs.

Theorem 1.44 (Hall's Theorem)**.** *Let G be a bipartite graph where $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. G contains a perfect matching if and only if $|N(S)| \geq |S|$ for all $S \subseteq V_1$.*

Proof. By induction on the size of V_1.

Base case: $|V_1| = 1$. The theorem trivially holds.

Inductive step: Let V_1 be a set of vertices such that $|V_1| = k$ for some $k \geq 2$. Assume that for all vertex sets of size smaller than k, the theorem holds. Suppose bipartite graph $G = (V_1 \cup V_2, E)$ satisfies Hall's condition.

Case 1: For all $S \subsetneq V_1$, $|N(S)| \geq |S| + 1$. Let (a, b) be an edge where $a \in V_1$ and $b \in V_2$. Let G' be the subgraph induced by $V - \{a, b\}$. Clearly, $|V_1 - \{a\}| \leq |N(V_1 - \{a\})|$. Here's a more careful argument of why G' satisfies Hall's condition. Let $S' \subseteq V_1 - \{a\}$, and let $N'(S')$ denote the neighborhood of S' in the graph G' induced by $V - \{a, b\}$. Further, $S' \subseteq V_1 - \{a\} \subset V_1$, so by assumption that G satisfies Hall's condition,

$$|N(S')| - 1 \geq |S'|$$

Since only b has been removed from the induced subgraph G', we also have $|N'(S')| \geq |N(S')| - 1$. It follows that $|N'(S')| \geq |S'|$ and Hall's condition holds for G'.

By inductive hypothesis, G' contains a perfect matching M'. Since (a, b) connects $a \in V_1$ with $b \in V_2$, $\{(a, b)\}$ is a perfect matching. Hence, $M = M' \cup \{(a, b)\}$ is a perfect matching in G.

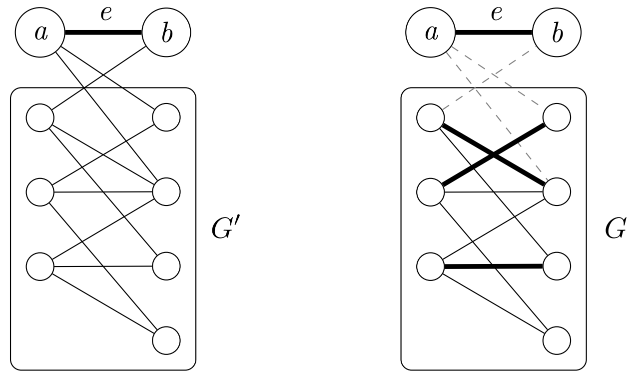


Figure 1.8: Case 1. Note that $N(S')$ can have one fewer vertex than $N'(S')$ (this happens when there is an edge from some vertex in S' to b, shown as dashed lines). $N(S')$ can also have the same number of vertices as $N'(S')$ if there is no edge going from vertices in S' to b. Hence the inequality $|N'(S')| \geq |N(S')| - 1$ holds.

Case 2: There exists some $S \subsetneq V_1$ such that $|S| = |N(S)|$. Since S is a proper subset of V_1, $|S| < |V_1|$. Let G_1 be the subgraph induced by $S \cup N(S)$. S is a proper subset of V_1, and V_1 satisfies Hall's condition. It follows that any subset of S must also be a subset of V_1 and hence satisfies Hall's hypothesis. By induction hypothesis, G_1 has a perfect matching M_1.

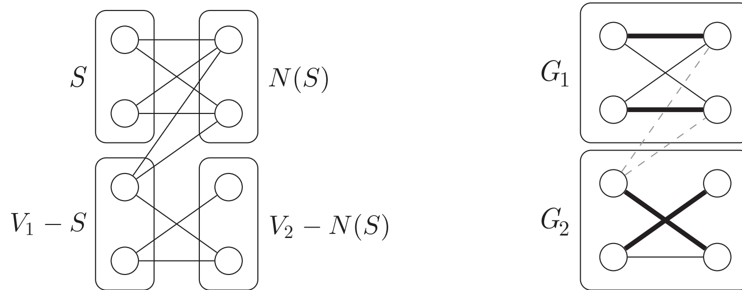


Figure 1.9: Case 2. We partition V_1 into S and $V_1 - S$. The reasoning behind the construction $(S \cup S') \cup (N(S) \cup N(S'))$ when showing that G_2 satisfies Hall's condition is that there any neighboring vertices of S' in G not included in $N_{G_2}(S')$ should be included in $N(S)$, which allows us to derive a contradiction if G_2 does not satisfy Hall's condition.

Let G_2 be the subgraph induced by $(V_1 - S) \cup (V_2 - N(S))$. We claim that G_2 also has a perfect matching. It suffices to show that G_2 satisfies Hall's condition. Suppose not, then there exists some $S' \subseteq (V_1 - S)$ such that $|S'| > |N_{G_2}(S')|$ where $N_{G_2}(S')$ denotes the neighborhood of S' in the subgraph G_2. More precisely,

$N_{G_2}(S') = N(S) \cap (V_2 - N(S))$. Consider the subgraph induced by $(S \cup S') \cup (N(S) \cup N(S'))$. Since $S \cup S'$ is a subset of V_1 and G satisfies Hall's condition

$$|N(S \cup S')| \geq |S \cup S'|$$

Since $N(S)$ and $N_{G_2}(S')$ are disjoint,

$$\begin{aligned}
|N(S \cup S')| &= |N(S) \cup N_{G_2}(S')| \\
&= |N(S)| + |N_{G_2}(S')| \\
&= |S| + |N_{G_2}(S')| \\
&< |S| + |S'| \\
&= |S \cup S'|
\end{aligned}$$

which contradicts the assumption that G satisfies Hall's condition. So, G_2 must also satisfy Hall's condition and by induction hypothesis, have a perfect matching M_2.

M_1 and M_2 are perfect matchings within their individual subgraphs that are disjoint. Taking the union of M_1 and M_2 yields a perfect matching M in G.

By induction, Hall's theorem holds for bipartite graphs of all sizes. □

1.4 Partially Ordered Sets

Graph is an important object in combinatorics and discrete mathematics. It is also closely related to another topic that we will discuss in this chapter, partially ordered sets. As it turns out, we can model a set equipped a certain type of ordering using graphs.

Definition 1.45 (Poset (partially ordered set))**.** *A **poset** $\mathbb{P}$ is a pair $\mathbb{P} = (X, P)$ where X is a set and $P \subseteq X \times X$ is a relation that is*

- *reflexive:* $\forall a \in X. (a, a) \in P$
- *anti-symmetric:* $a \neq b \wedge (a, b) \in P \implies (b, a) \notin P$
- *transitive:* $(a, b) \in P \wedge (b, c) \in P \implies (a, c) \in P$

Instead of writing $(a, b) \in P$, we use the notation $a \leq_{\mathbb{P}} b$.

Definition 1.46 (Embedding, Isomorphism, Automorphism)**.** *Given a poset $\mathbb{P} = (X, P)$ and $\mathbb{Q} = (Y, Q)$, an **embedding** from $\mathbb{P}$ into $\mathbb{Q}$ is an injective map $f : X \to Y$ with the property that $a \leq_{\mathbb{P}} b$ if and only if $f(a) \leq_{\mathbb{Q}} f(b)$. If the embedding is surjective, we call it an **isomorphism**. If $\mathbb{Q} = \mathbb{P}$, we call it an **automorphism**.*

For example, consider $X = \{\star, \circ, \diamond\}$ with $P = \{(\star, \star), (\circ, \circ), (\diamond, \diamond), (\diamond, \star)\}$, and $Y = \{\star, \circ, \diamond, \Box\}$ with $Q = \{(\star, \star), (\circ, \circ), (\diamond, \diamond), (\Box, \Box), (\Box, \star)\}$. Let $\mathbb{P} = (X, P)$ and $\mathbb{Q} = (Y, Q)$. Consider the function $f : X \to Y$ such that $f(\star) = \star$, $f(\circ) = \circ$, and $f(\diamond) = \Box$. f is an embedding because $\diamond \leq_{\mathbb{P}} \star$ if and only if $f(\diamond) = \Box \leq_{\mathbb{Q}} \star = f(\star)$. However, f is not an isomorphism because $\diamond$ is not in the range of f so f is not surjective.

Definition 1.47 (Dual)**.** *Given a poset $\mathbb{P} = (X, P)$, we call the poset $\mathbb{P}^d = (X, P^d)$ where $a \leq_{\mathbb{P}^d} b \iff b \leq_{\mathbb{P}_d} a$ the **dual** of $\mathbb{P}$. We say a poset is self-dual if it is isomorphic to its dual.*

Definition 1.48 (Cover)**.** *Given a poset $\mathbb{P} = (X, P)$ and a point $a \in X$, we say a is **covered by** a point $b \in X$ if $a <_{\mathbb{P}} b$ and there is no c such that $a <_{\mathbb{P}} c <_{\mathbb{P}} b$.*

Definition 1.49 (Cover Graph)**.** *Given a poset $\mathbb{P} = (X, P)$, we call the graph $G = (X, E)$ given by $\{x, y\} \in E$ if and only if x covers y or y covers x, the* ***cover graph*** *associated to $\mathbb{P}$.*

If we draw the cover graph in an oriented fashion where lower vertices correspond to the $\leq_{\mathbb{P}}$-smaller elements, we have a special kind of cover graph known as **Hasse diagram**.

Let $\mathbb{P} = (X, P)$ be a poset where $X = \{a, b, c, d, e, f\}$ and $P = \{(a, c), (b, c), (b, d), (d, e), (a, e), (e, f)\}$. One possible cover graph and the Hasse diagram is shown below.

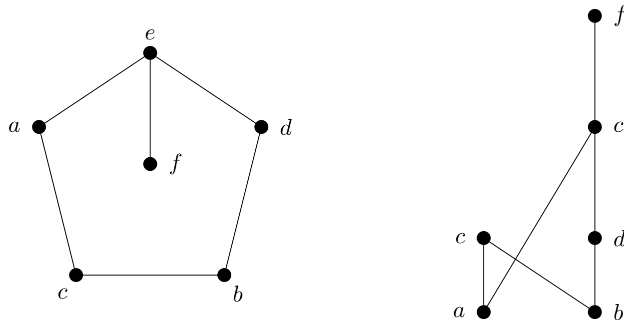


Figure 1.10: Cover graph and Hasse diagram for the poset described above.

1.4.1 Linear (Total) Order

Definition 1.50 (Comparability)**.** *Given a poset $\mathbb{P} = (X, P)$, we say two points $a, b \in X$ are* ***comparable*** *if either $a <_{\mathbb{P}} b$ or $b <_{\mathbb{P}} b$ If two points are not comparable, we call them* ***incomparable****.*

Definition 1.51 (Total/Linear Order)**.** *Given a poset $\mathbb{P} = (X, P)$, we say $\mathbb{P}$ is* ***linearly ordered*** *or* ***totally ordered*** *if no two distinct points are incomparable.*

1.5 Counting

1.5.1 The Pigeonhole Principle

Theorem 1.52 (The Pigeonhole Principle)**.** *Let X be a set of n objects. Suppose $\{X_1, \ldots, X_k\}$ from a partition of X (i.e. a family of disjoint sets whose union is X). If $k < n$, then $\exists i \in \{1, \ldots, k\}$ such that $|X_i| \geq 2$.*

Proof. Suppose for contradiction that for all $i \in \{1, \ldots, k\}$, $|X_i| = 1$. Since $\{X_1, \ldots, X_k\}$ is a partition,

$$n = |X| = \sum_{i=1}^{k} |X_i| = k.$$

But this is contradiction since $k < n$. □

The Pigeonhole Principle is often used to show that there are more than one element in a set with the same property or to bound the size of sets. We can extend the Pigeonhole Principle to a more general case where one set is of size m and the other is of $km + 1$.

Theorem 1.53 (Generalized Pigeonhole Principle)**.** *Fix $k, m \in \mathbb{N}$, sets X and Y with $|X| = km + 1$ and $|Y| = m$, and a function $f : X \to Y$. Show that there exists an element $y \in Y$ with at least $k+1$ preimages.*

Proof. Let X be a set with $|X| = km + 1$ for some $k, m \in \mathbb{N}$ and Y be a set such that $|Y| = m$. Let f be a function mapping from X to Y.

For $y \in Y$, let $P_y = \{x \in X \mid f(x) = y\}$ be the set of preimages for $y \in Y$. We would like to show that $|P_y| \geq k + 1$. Suppose for contradiction that $|P_y| < k + 1$ for all $y \in Y$. Since f is a well-defined function, f does not map any $x \in X$ to more than one element in Y. This implies that for every distinct $y_1, y_2 \in Y$, $P_{y_1} \cap P_{y_2} = \emptyset$. Further, since f is a well-defined function, every element in X is mapped to an element in Y. So, $\bigcup_{y \in Y} P_y = X$. By definition, $\{P_y\}_{y \in Y}$ is a partition of X. Then, it follows that

$$\begin{aligned} |X| &= \Big| \bigcup_{y \in Y} P_y \Big| \\ &= \sum_{y \in Y} |P_y| && P_y\text{'s are disjoint} \\ &\leq \sum_{y \in Y} k && |P_y| < k+1 \text{ for all } y \in Y \\ &= km && |Y| = m \end{aligned}$$

which immediately implies that $|X| \leq km$. However, this is a contradiction because by assumption, $|X| = km + 1$.

Hence, our assumption that $|P_y| < k + 1$ for all $y \in Y$ is false. Therefore, there exists some $y \in Y$ such that $|P_y| \geq k + 1$, which is equivalent to saying that there exists an element $y \in Y$ with at least $k + 1$ preimages by definition. □

1.5.2 Permutation and Combination

Let $X = \{a_1, \dots, a_n\}$ be a set of distinct object.

Definition 1.54 (Permutation)**.** *A* ***permutation*** *of length k is an X-string of length k such that there is no repetition.*

Given integers n and k with $n \geq k$, we let $P(n, k)$ denote the number of permuations of $[n]$ of length k.

Example 1.55. *Suppose $X = \{a, b, c, d\}$. abc is a permutation of length 3. $\emptyset$ is a permutation of length 0. However, bab is not a permutation because of the repeating character b.*

Remark 1.56. *Because a permutation requires there be no repetition, there is no permutation of length k for X of size n if $k > n$.*

Definition 1.57 (Combination)**.** *Given a set of objects X, a* ***combination*** *of size k is a subset $A \subseteq X$ of size k.*

Example 1.58. *What are the subsets of $\{1, 2, 3\}$?*

$$\emptyset = \{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$$

There are 3 combinations of $\{1, 2, 3\}$ of size 2.

We can use the following formulas to compute permutation and combination.

$$P(n, k) = \frac{n!}{(n-k)!}$$

$$C(n, k) = \binom{n}{k} = \frac{n!}{(n-1)!k!}$$

1.5.3 Binomial Theorem

Theorem 1.59 (Binomial Theorem)**.** *For any $x \in \mathbb{R}$, for any $n \geq 0$ natural number*

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

Proof.

$$\text{LHS} = (1+x)^n = \underbrace{(1+x)(1+x)\cdots(1+x)}_{n \text{ times}}$$

When we expand and collect the like terms, we get a polynomial of the form

$$\sum_{k=0}^{n} c_k x^k = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

For k, the only way to get x^k in the expansion of the LHS is if for k of the n terms in the product to contribute to to x, and the rest $n-k$ of the terms to contribute to 1.

In total, we have $\binom{n}{k}$ ways to get x^k in the expansions. Hence, $c_k = \binom{n}{k}$. □

The bionmial theorem holds for more than one variable.

Theorem 1.60 (Binomial Theorem (two variables))**.** *For any $x, y \in \mathbb{R}$, $n \in \mathbb{N}$ such that $n \geq 0$,*

$$(x+y)^n = \sum_{k=0}^{k} \binom{n}{k} x^k y^{n-k}$$

1.5.4 Bijection

Suppose we have sets A and B. $f : A \to B$ is a function.

Definition 1.61 (Injective)**.** *We say f is* ***injective*** *or one-to-one if $\forall a_1 \neq a_2 \in A.\, f(a_1) \neq f(a_2)$.*

Definition 1.62 (Surjective)**.** *We say f is* ***surjective*** *or onto if $\forall b \in B.\, \exists a \in A.\, f(a) = b$.*

Definition 1.63 (Bijection)**.** *We say that f is* ***bijective*** *if f is both injective and surjective.*

Remark 1.64. *If there is a bijective mapping $f : A \to B$, then A and B have the same size (cardinality).*

This is useful because it allows us to count things using things that we already know how to count if we can find a bijection between the two sets.

1.5.5 Induction

Theorem 1.65 (Well-Ordering Principle)**.** *Every non-empty set of positive integers (or natural numbers) contains a least element (an element that is less than or equal to every other elements).*

From the well-ordering principle, one can derive the principle of weak induction.

Principle of Weak Induction: Suppose we have a family of statements $P(n)$ for all $n \in \mathbb{N}$, and suppose the following holds:

1. $P(1)$ holds, and
2. if $P(k)$ holds, then $P(k+1)$ holds.

Then, we can conclude $P(n)$ is true for all $n \in \mathbb{N}$.

Remark 1.66. *The base case does not have to be for $n = 1$ if we are not proving the statement for all $n \in \mathbb{N}$. Say, for example, that we want to show $P(n)$ holds for all $n \in \mathbb{N}$ such that $n \geq 5$. Then, we can use $P(5)$ as the base case.*

Further, from the principle of simple induction, one can derive the seemingly stronger but actually equivalent principle of strong induction.

Principle of Strong Induction: Suppose we have a family of statements $P(n)$ for all $n \in \mathbb{N}$. Suppose the following are true:

1. $P(1)$ is true, and
2. if $P(m)$ is true for all $1 \leq m \leq k$, then $P(k+1)$ is true.

Then, we can conclude $P(n)$ is true for all $n \in \mathbb{N}$.

1.5.6 Principle of Inclusion-Exclusion

Definition 1.67. *Let S be a set. Fix a collection $A_i \subseteq S$ indexed by $i \in \{1, \ldots, n\}$. Given $I \subseteq \{1, \ldots, n\}$, we let $A_I = \bigcap_{i \in I} A_i$. In particular, we set $A_\emptyset = S$.*

We are all familiar with the addition principle: given two disjoint sets A, B such that $A \cap B = \emptyset$,

$$|A \cup B| = |A| + |B|$$

In the case where A and B are not disjoint, we can simply remove the count for the common elements among A and B that were double-counted.

$$|A \cup B| = |A| + |B| - |A \cap B|$$

We further observe that four three sets A, B, C, we have

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| - |A \cap B \cap C|$$

The generalization of this is called the Inclusion-Exclusion Principle. There are multiple equivalent formulations of the Inclusion-Exclusion Principle.

Theorem 1.68 (Inclusion-Exclusion Principle). *Let S be a set. Let $A_1, \ldots, A_n \subseteq S$ be subsets of S indexed by $i \in \{1, \ldots, n\}$. Then, the number of elements in S that is in none of A_i is*

$$\left|\bigcap_{i=1}^{n} \overline{A_i}\right| = \left|S \setminus \bigcup_{i=1}^{n} A_i\right| = \sum_{I \subseteq [n]} (-1)^{|I|} |A_I|$$

and equivalently,

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\substack{I \subseteq [n] \\ I \neq \emptyset}} (-1)^{|I|+1} |A_I|$$

If stated without using the notation introduced in Definition 1.67, the Inclusion-Exclusion Principle can be stated as follows.

Theorem 1.69. *If $A_i \subseteq X$ for $1 \leq i \leq n$, then*

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n} \left((-1)^{k+1} \sum_{\{i_1, \ldots, i_k\} \subseteq [n]} \left|\bigcap_{j=1}^{k} A_{i_j}\right| \right)$$

and equivalently

$$\left|\bigcap_{i=1}^{n} \overline{A_i}\right| = |X| + \sum_{k=1}^{n} \left((-1)^{k} \sum_{\{i_1, \ldots, i_k\} \subseteq [n]} \left|\bigcap_{j=1}^{k} A_{i_j}\right| \right)$$

We give a proof of the Inclusion-Exclusion Principle using induction.

Proof.

Base Case: $n = 1$. A_1 is the only subset of S in the collection. $|\bigcup_{i=1}^{n} \overline{A_i}| = |S| - |A_i| = \sum_{I \in [1]} (-1)^{|I|} |A_I|$.

Inductive Step: Let $n \in \mathbb{N}$ be arbitrary. Assume that the principle holds for all $k \leq n$. Consider a collection of subsets $A_1, \ldots, A_{n+1} \subseteq S$. Let $B_i = A_i \cap A_{n+1}$ for $i \in \{1, \ldots, n\}$. Let $B_I = A_{n+1} \cap \bigcap_{i \in I} A_i$. Note that $B_\emptyset = A_{n+1}$.

We know by induction hypothesis that

$$\left|S \setminus \bigcup_{i=1}^{n} A_i\right| = \sum_{I \subseteq [n]} (-1)^{|I|} |A_I| \qquad \left|A_{n+1} \setminus \bigcup_{i=1}^{n} A_i\right| = \sum_{I \subseteq [n]} (-1)^{|I|} |B_I|$$

In other words, $\sum_{I \subseteq [n]} (-1)^{|I|} |A_I|$ counts the number of elements in S that is not in $A_1, \ldots, A_n$. Similarly,

$\sum_{I\subseteq[n]}(-1)^{|I|}|B_I|$ counts the number of elements in S that satisfied only A_{n+1}. Now, consider

$$\begin{aligned}
\sum_{I\subseteq[n+1]}(-1)^{|I|}|A_I| &= \sum_{I\subseteq[n]}(-1)^{|I|}|A_I| + \sum_{I\subseteq[n]}(-1)^{|I|+1}|A_{I\cup\{n+1\}}| \\
&= \sum_{I\subseteq[n]}(-1)^{|I|}|A_I| - \sum_{I\subseteq[n]}(-1)^{|I|}|A_{I\cup\{n+1\}}| && \text{take out } -1 \\
&= \left|S\setminus\bigcup_{i=1}^{n}A_i\right| - \sum_{I\subseteq[n]}(-1)^{|I|}|A_{I\cup\{n+1\}}| && \text{by I.H.} \\
&= \left|S\setminus\bigcup_{i=1}^{n}A_i\right| - \sum_{I\subseteq[n]}(-1)^{|I|}|B_I| && \text{by definition of } B_I \\
&= \left|S\setminus\bigcup_{i=1}^{n}A_i\right| - \left|A_{n+1}\setminus\bigcup_{i=1}^{n}A_i\right| && \text{by I.H.} \\
&= \left|S\setminus\bigcup_{i=1}^{n+1}A_i\right|
\end{aligned}$$

□

Let's consider the following example.

Example 1.70. *Count the number of integer solutions to the equation $x_1+x_2+x_3+x_4 = 100$ with $x_1, x_2 \leq 10$ and $x_i \geq 0$ for all $i \in \{1,2,3,4\}$.*

Let S be the set of solutions with $x_i \geq 0$ for all $i \in \{1,2,3,4\}$. Let A_1 be the set of solutions with $x_1 \geq 10$ and $x_i \geq 0$ for all $i \in \{1,2,3,4\}$. Let A_2 be the set of solutions with $x_2 \geq 10$ and $x_i \geq 0$ for all $i \in \{1,2,3,4\}$.

Notice that the quantity that we want to count is equal to $|S \setminus (A_1 \cup A_2)|$. If a solution in S fails any of our conditions, it must be in either A_1 or A_2. We can count $|S|$ using the stars-and-bars technique (since we are essentially trying to partition 100 elements into 4 groups by placing 3 bars). So it follows that we have $|S| = \binom{100+3}{3}$. We can do the same to count $|A_1|$ and $|A_2|$. It follows from the Principle of Inclusion-Exclusion that

$$\begin{aligned}
|S\setminus(A_1\cup A_2)| &= |S| - (|A_1|+|A_2|) + |A_1\cap A_2| \\
&= \binom{100+3}{3} - \left(\binom{90+3}{3}+\binom{90+3}{3}\right) - \binom{80+3}{3}
\end{aligned}$$

Chapter 2
Probability

2.1 Review of Basic Probability Theory

2.1.1 Probability Space

A ***probability space*** $(\Omega, \Pr)$ consists of a finite or countable set Ω called the ***sample space***, and the ***probability function*** $\Pr : \Omega \to \mathbb{R}$ such that for all $\omega \in \Omega$, $\Pr(\omega) \geq 0$ and $\sum_{\omega \in \Omega} \Pr(\omega) = 1$.

We call an element $\omega \in \Omega$ a sample point, or ***outcome***, or simple event. A sample space models some random "experiment" where Ω contains all possible outcomes of the experiment, and $\Pr(\omega)$ gives the probability that we are going to get outcome ω. We always discuss probability in relation to a sample space.

If $\Pr(\omega) = \Pr(\omega')$ for all $\omega, \omega' \in \Omega$, we say that the probability is ***uniform*** over Ω.

For a set of events $A \subseteq \Omega$, we define the probability of A to be the sum of the probability of each event in A:

$$\Pr(A) = \sum_{\omega \in A} \Pr(\omega).$$

It is easy to see that for any two events A and B,

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B).$$

From this, we can derive our first probability bound.

Theorem 2.1 (Union Bound)**.** *Let A and B be two events.*

$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$$

2.1.2 Conditional Probability

Definition 2.2 (Conditional Probability)**.** *The probability of A conditional on B is defined as*

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

*The **conditional probability** $\Pr(A \mid B)$ is only defined when $\Pr(B) > 0$.*

Theorem 2.3 (Bayes' rule)**.**

$$\Pr(A \mid B) = \frac{\Pr(B \mid A) \cdot \Pr(A)}{\Pr(B)}$$

Theorem 2.4 (Law of Total Probability)**.**

$$\Pr(A) = \sum_{n} \Pr(A \mid B_n) \cdot \Pr(B_n)$$

Definition 2.5 (Independence)**.** *Two events A and B are **independent** if*

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B)$$

*If $\Pr(B) > 0$, this implies that $\Pr(A \mid B) = \Pr(A)$. More generally, events $A_1, \ldots, A_k$ are **mutually independent** if*

$$\Pr\left(\bigcap_{i=1}^{k} A_k\right) = \prod_{i=1}^{k} \Pr(A_i)$$

2.1.3 Random Variable and Their Expectations

Definition 2.6 (Random Variable)**.** *Given a probability space $(\Omega, \Pr)$, a **random variable** X is a function $X : \Omega \to \mathbb{R}$. Suppose that the range of X is $\mathcal{X}$. The probability distribution of X is the function $p : \mathcal{X} \to [0, 1]$ such that*

$$p(x) = \Pr(X = x).$$

Definition 2.7 (Expectation)**.** *The **expected value** or **expectation** of a random variable X is defined as*

$$\mathbb{E}[X] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr(\omega) = \sum_{x \in \mathcal{X}} xp(x).$$

The expectation is **linear**. That is, for r.vs $X_1, \ldots, X_k$ defined on the same probability space,

$$\mathbb{E}[X_1 + \cdots + X_k] = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_k].$$

Definition 2.8 (Conditional Expectation)**.** *The **conditional expectation** of X with respect to event A is defined as*

$$\mathbb{E}[X \mid A] = \sum_{x \in \mathcal{X}} x \cdot \Pr(X = x \mid A)$$

where $\mathcal{X}$ is the set of values taken by X.

Similarly, we have the following result

Theorem 2.9 (Law of Total Expectation)**.**

$$\mathbb{E}[\mathbb{E}[X \mid Y]] = \mathbb{E}[X].$$

Definition 2.10 (Variance)**.** *Given a r.v. X, the **variance** of X is defined as*

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$$

Alternatively, from linearity of expectation,

$$\begin{aligned}
\mathrm{Var}(X) &= \mathbb{E}[(X - \mathbb{E}[X])^2] \\
&= \mathbb{E}[X^2 - 2X\,\mathbb{E}[X] + \mathbb{E}[X]^2] \\
&= \mathbb{E}[X^2] - 2\,\mathbb{E}[X]\,\mathbb{E}[X] + \mathbb{E}[X]^2 \\
&= \mathbb{E}[X^2] - \mathbb{E}[X]^2
\end{aligned}$$

Theorem 2.11 (Law of Total Variance)**.** *Let X and Y be two r.vs and assume that the variance of X exists, then,*

$$\mathrm{Var}(X) = \mathbb{E}[\mathrm{Var}(X \mid Y)] + \mathrm{Var}(\mathbb{E}[X \mid Y])$$

Definition 2.12 (Covariance). *The covariance of two random variables X and Y is*

$$\mathrm{Cov}(X,Y) = \mathbb{E}[(X - \mathbb{E}[X]) \cdot (Y - \mathbb{E}[Y])].$$

If the covariance of two r.vs is 0, we say they are ***uncorrelated****.*

Again, using linearity of expectation, we can derive the alternative expression:

$$\mathrm{Cov}(X,Y) = \mathbb{E}[XY] - \mathbb{E}[X]\,\mathbb{E}[Y]$$

The **variance is not linear**, but it does have the following properties:

$$\mathrm{Var}(X + a) = \mathrm{Var}(X)$$

and

$$\mathrm{Var}(aX) = a^2\mathrm{Var}(X).$$

We can compute the variance of the sum of two r.vs as follows

$$\begin{aligned}
\mathrm{Var}(X+Y) &= \mathrm{Cov}(X+Y,\ X+Y) \\
&= \mathrm{Cov}(X,X) + \mathrm{Cov}(X,Y) + \mathrm{Cov}(Y,X) + \mathrm{Cov}(Y,Y) \\
&= \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X,Y)
\end{aligned}$$

This derivation uses the following properties of covariance:

$$\mathrm{Cov}(X,X) = \mathrm{Var}(X) \qquad \mathrm{Cov}(X,Y) = \mathrm{Cov}(Y,X) \qquad \mathrm{Cov}(aX,bY) = ab\mathrm{Cov}(X,Y)$$

and

$$\mathrm{Cov}(aX+bY, cW+dV) = ac\mathrm{Cov}(X,W) + ad\mathrm{Cov}(X,V) + bc\mathrm{Cov}(Y,W) + bd\mathrm{Cov}(Y,V).$$

2.2 Concentration Inequalities

The intuition behind all concentration inequalities is similar: we expect the behavior of a r.v. often to be not too far off from its expected behavior.

Theorem 2.13 (Markov's Inequality). *Let X be a non-negative random variable. Then, for $a > 0$,*

$$\Pr(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

Proof. Let X be continuous with probability density p. The discrete case is analogous. Then,

$$\begin{aligned}
\mathbb{E}[X] &= \int_0^\infty xp(x)dx = \int_0^a xp(x)dx + \int_a^\infty xp(x)dx \\
&\geq \int_a^\infty xp(x)dx \\
&\geq a\int_a^\infty p(x)dx \\
&= a\,\Pr(x \geq a)
\end{aligned}$$

□

Corollary 2.14. $\Pr(X \geq b\,\mathbb{E}[X]) \leq \frac{1}{b}$

Theorem 2.15 (Chebyshev's Inequality)**.** *Let X be a random variable with bounded variance. Then for $c > 0$,*

$$\Pr(|X - \mathbb{E}[X]| \geq c) \leq \frac{\mathrm{Var}(X)}{c^2}.$$

Proof. We note that

$$\Pr(|X - \mathbb{E}[X]| \geq c) = \Pr(|X - \mathbb{E}[X]|^2 \geq c^2).$$

Define r.v. $Y = |X - \mathbb{E}[X]|^2$. Then, Y is clearly non-negative and $\mathbb{E}[Y] = \mathrm{Var}(X)$. By Markov's inequality,

$$\Pr(|X - \mathbb{E}[X]| \geq c) = \Pr(Y \geq c^2) \leq \frac{\mathbb{E}[Y]}{c^2} = \frac{\mathrm{Var}(X)}{c^2}.$$

□

Theorem 2.16 (Law of Large Numbers)**.** *Let $X_1, \ldots, X_n$ be independent samples of a random variable X. Then,*

$$\Pr\left(\left|\frac{X_1 + \ldots + X_n}{n} - \mathbb{E}[X]\right| \geq \epsilon\right) \leq \frac{\mathrm{Var}(X)}{n\epsilon^2}.$$

Proof. By Chebyshev,

$$\begin{aligned}\Pr\left(\left|\frac{X_1 + \ldots + X_n}{n} - \mathbb{E}[X]\right| \geq \epsilon\right) &\leq \frac{\mathrm{Var}\left(\frac{X_1+\ldots+X_n}{n}\right)}{\epsilon} \\ &= \frac{1}{n^2\epsilon^2}\mathrm{Var}(X_1 + \ldots + X_n)\end{aligned}$$

Because $X_1, \ldots, X_n$ are independent,

$$\begin{aligned}\Pr\left(\left|\frac{X_1 + \ldots + X_n}{n} - \mathbb{E}[X]\right| \geq \epsilon\right) &\geq \frac{1}{n^2\epsilon^2}\mathrm{Var}(X_1 + \ldots + X_n) \\ &= \frac{1}{n^2\epsilon^2}\left(n\mathrm{Var}(X)\right) \\ &= \frac{\mathrm{Var}(X)}{n\epsilon^2}.\end{aligned}$$

□

This is sometimes called the weak law of large numbers (WLLN). The strong law of large number states that

$$\Pr\left(\lim_{n\to\infty} \frac{X_1 + \ldots + X_n}{n} = \mathbb{E}[X]\right) = 1.$$

However, we will focus on the weak law since it leads to other results related to tail bounds. We can use tail bounds to analyze typical behaviors of random variables. There are some very counterintuitive properties of high-dimensional space. We will prove some results about high-dimensional space by treating points in the space as random variables and apply the concentration inequalities.

Theorem 2.17. *Let r be a positive, even integer, then*

$$\Pr(|X| \geq a) \leq \mathbb{E}[X^r]/a^r$$

Proof. direct application of Markov's inequality. ☐

Let $x_i = (y_i - z_i)^2$ be a random variable with bounded variance. By LLN,

$$\Pr\left(\left|\frac{x_1 + \ldots + x_d}{d} - \mathbb{E}[X]\right| \geq \epsilon\right) \leq \frac{\mathrm{Var}(X)}{d\epsilon^2}.$$

We can prove stronger bounds using stronger results. To this end, we consider the master tail bounds theorem. It is called so because it allows us to prove other tail bounds are related results more easily such as **Chernoff bounds**.

Theorem 2.18 (Master Tail Bounds Theorem)**.** *Let $X = X_1 + \ldots + X_n$ where $X_1, \ldots, X_n$ are mutually independent r.vs with 0 mean and variance at most σ^2. Let $0 \leq a \leq \sqrt{2}n\sigma^2$. Assume that $|\mathbb{E}[X_i^s]| \leq \sigma^2 s!$ for $s = 3, 4, \ldots, \left\lfloor \frac{a^2}{4n\sigma^2} \right\rfloor$. Then,*

$$\Pr\left(|X| \geq a\right) \leq 3e^{\frac{-a^2}{12n\sigma^2}}.$$

Proof. We will only sketch a proof here.

Apply Markov to x^r for large and even r (we require r to be even so that $x^r \geq 0$). Then, by Markov

$$\Pr(|X| \geq a) = \Pr(x^r \geq a^r) \leq \frac{\mathbb{E}[X^r]}{a^r}$$

We can then compute $\mathbb{E}[X^r]$ by expanding out $X = X_1 + \ldots + X_n$ and using properties of expectation and the technical assumptions introduced in the theorem. In the end, we will find that $\mathbb{E}[X^r]$ is bounded by $3e^{\frac{-a^2}{12n\sigma^2}}$. ☐

An even more useful tail bound inequality is the Chernoff bound.

Theorem 2.19 (Chernoff Bounds)**.** *Let $X = X_1 + \ldots + X_n$ where $X_i \sim$ Bernoulli and is i.i.d. Then, for any $\epsilon \in [0, 1]$*

$$\Pr(|X - \mathbb{E}[X]| \geq \epsilon\,\mathbb{E}[X]) \leq 3e^{-\epsilon^2\,\mathbb{E}[X]/12}.$$

There are other variants of the Chernoff bounds. The bound as stated above can be proven directly from the Master Tail Bounds Theorem, whereas the more standard proof using moment-generating functions gives a better constant in the exponent.

Proof. Note that $p = \mathbb{E}[X_i]$ for all i. This is the same parameter p in the Bernoulli distribution from which X_i is sampled from. Let $Y_i = X_i - \mathbb{E}[X_i]$. Then, $\mathbb{E}[Y_i] = 0$ and $\mathbb{E}[Y_i^2] = \mathrm{Var}(X_i) = p(1-p)$.

When $s \geq 3$,

$$|\,\mathbb{E}[Y_i^s]| = |\,\mathbb{E}[(X_i - \mathbb{E}[X_i])^s]| = |p(1-p)^s + (1-p)(-p)^s|$$

Hence, it follows that $|\,\mathbb{E}[Y_i^s]| \leq p$ for all $s \geq 2$. We verify the assumption for the Master Tail Bounds Theorem

- the variance $\mathrm{Var}(Y_i)$ is bounded by $\sigma^2 = p$;
- setting $a = \epsilon np$, we have $a < \sqrt{2}n\sigma^2$;
- setting $s = \epsilon^2 np/4$, we have $s \geq a^2/(4n\sigma^2)$.

Then, by the Master Tail Bounds Theorem,

$$\begin{aligned}\Pr(|X - \mathbb{E}[X]| \geq \epsilon\,\mathbb{E}[X]) &= \Pr(|Y_1 + \ldots + Y_n| \geq a) \\ &\leq 3e^{-a^2/(12n\sigma^2)} \\ &= 3e^{-\frac{c^2n^2p^2}{12np}} \\ &= 3e^{-\mathbb{E}[X]\epsilon^2/12}.\end{aligned}$$

□

2.3 Moment Generating Functions

Definition 2.20 (Moment Generating Function)**.** *For a random variable X, the* ***moment-generating function*** *is*

$$M_X(\lambda) = \mathbb{E}[e^{\lambda X}].$$

The rth derivative of $M_X(\lambda)$ is called the rth ***moment*** *of X.*

$$\frac{d^rM}{d\lambda^r}(0) = \mathbb{E}[X^r].$$

The first moment is the mean; the second central moment is the variance; the third normalized central moment is the skewness.

Having introduced the necessary definition, we are ready to state the Chernoff bound and sketch a proof.

Theorem 2.21. *Let $X_1, \ldots, X_n$ be n independent Bernoulli random variables. Let $S = X_1 + \ldots + X_n$, and let $\mu = \mathbb{E}[S] = np$. Then, for any $\delta > 0$,*

$$\Pr(s > (1+\delta)\mu) \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu.$$

Proof. For $\lambda > 0$, $e^{\lambda x}$ is monotone, so

$$\Pr(S > (1+\delta)\mu) = \Pr(e^{\lambda S} > e^{\lambda(1+\delta)\mu}).$$

By Markov's inequality,

$$\Pr(e^{\lambda S} > e^{\lambda(1+\delta)\mu}) \leq \frac{\mathbb{E}[e^{\lambda S}]}{e^{\lambda(1+\delta)\mu}}.$$

We consider the expansion of $\mathbb{E}[e^{\lambda S}]$. Because the X_i's are independent,

$$\mathbb{E}[e^{\lambda S}] = \mathbb{E}\left[e^{\lambda \sum_{i=1}^n X_i}\right] = \mathbb{E}\left[\prod_{i=1}^n e^{\lambda X_i}\right] = \prod_{i=1}^n \mathbb{E}[e^{\lambda X_i}].$$

Since X_i is Bernoulli, the r.v. $e^{\lambda X_i}$ is equal to e^{λ} with probability p and 1 with probability $1-p$. Hence,

$$\begin{aligned}\mathbb{E}[e^{\lambda S}] &= \prod_{i=1}^{n} \mathbb{E}[e^{\lambda X_i}] \\ &= \prod_{i=1}^{n} (e^{\lambda} p + (1-p)) \\ &= \prod_{i=1}^{n} (p(e^{\lambda}-1)+1) \\ &< \prod_{i=1}^{n} e^{p(e^{\lambda}-1)}.\end{aligned}$$

The last inequality holds because $1+x < e^x$ for all $x > 0$. So far, we have shown that

$$\Pr(S > (1+\delta)\mu) \leq \frac{\mathbb{E}[e^{\lambda S}]}{e^{\lambda(1+\delta)\mu}} \qquad \text{and} \qquad \mathbb{E}[e^{\lambda S}] < \prod_{i=1}^{n} e^{p(e^{\lambda}-1)}.$$

It follows that

$$\Pr(S > (1+\delta)\mu) \leq \frac{\mathbb{E}[e^{\lambda S}]}{e^{\lambda(1+\delta)\mu}} < e^{-\lambda(1+\delta)\mu} \cdot \prod_{i=1}^{n} e^{p(e^{\lambda}-1)}$$

and the theorem follows by setting $\lambda = \ln(\delta+1) > 0$. □

Intuitively, the Chernoff bound tells us that if X is the sum of many i.i.d. Bernoulli trials (coin flips), it is extremely unlikely that X will deviate from even a little bit from its expected value.

Bibliography

Master Tail Bounds theorem and the proof of Chernoff bound using it are results from the book *Foundations of Data Science*, [2].

Part II

Information and Compression

Chapter 3
Measure of Information and Complexity

The field of information theory was first formulated by Claude Shannon in 1948. It aims to quantify how different an observed distribution of states is from an expected distribution. For example, a relevant case where information theory could be helpful in bioinformatics is when we want to quantify how different the amino acids are distributed in a biological system compared to a stochastic process.

3.1 Entropy

Suppose we have a set of possible events whose probabilities of occurrence are $p_1, p_2, \cdots, p_n$. These probabilities are known but that is all we know concerning which event will occur. Can we find a measure of how much "choice" is involved in the selection of the event or of how uncertain we are of the outcome? If there is such a measure, say $H(p_1, \cdots, p_n)$, it is reasonable to require of it the following properties:

1. H should be continuous in the p_i
2. If all the p_i are equal, $p_i = 1/n$, then H should be a monotonic increasing function of n. With equally likely events there is more choice, or uncertainty, when there are more possible events.
3. If a choice be broken down into two successive choices, the original H should be the weighted sum of the individual values of H.

Entropy should be a measure of randomness, uncertainty, or complexity of a random source. It is usually considered in relation to some random variable that models a data source such as letters in a DNA string.

Definition 3.1 (Self-Information)**.** *Given an event ω with probability p, the information content or self-information is*

$$I(\omega) = -\log \Pr(\omega) = -\log p.$$

For a random variable X with probability mass function p_X, the self-information of the event $X = x$ is

$$I_X(x) = -\log p_X(x) = \log \left(\frac{1}{p_X(x)} \right).$$

Theorem 3.2 (Shannon, 1948)**.** *The only H satisfying the three assumptions above is one of the form*

$$H = -K \sum_{i=1}^{n} p_i \log p_i$$

where K is some positive constant.

Alternatively, it can be written as an expectation.

Definition 3.3 (Entropy)**.** *Let X be a random variable with range $\mathcal{X}$ and distribution $p : \mathcal{X} \to [0,1]$ such that $p(x) = \Pr(X = x)$. Then,*

$$H(X) = -\sum_{x_i \in \mathcal{X}} \Pr(x_i) \log(\Pr(x_i)) = \mathbb{E}[I_X(X)] = \mathop{\mathbb{E}}_{x_i \in \mathcal{X}} (-\log \Pr(x_i)).$$

It follows from the definition of entropy H that,

- $H = 0$ if and only if all p_i, except one are zero, and the one remaining has $p = 1$. This is because if there is no uncertainty, the entropy is zero.
- For a given n, H is maximal if all p_i are equal. This means that the outcomes are uniformly distributed. In this case, $H = \log n$.

As an example, consider the entropy of equiprobable nucleotides where $p_A = p_C = p_G = p_T = 0.25$.

$$H_{\max}^{nuc} = -\sum_{i \in \{A,C,G,T\}} p_i \log_2 p_i = -4 \times \frac{1}{4} \log_2 \frac{1}{4} = 2$$

We can define entropy of multiple random variables similarly.

Definition 3.4 (Entropy (multiple r.v.))**.** *Let $X_1, \ldots, X_m$ be m random variables. Entropy is defined by their joint probability distribution $p_{X_1,\ldots,X_m}$ as*

$$H(X_1, \ldots, X_m) = -\sum_{x_1 \in \mathcal{X}_1} \cdots \sum_{x_m \in \mathcal{X}_m} p_{X_1,\ldots,X_m}(x_1, \ldots, x_m) \cdot \log(p_{X_1,\ldots,X_m}(x_1, \ldots, x_m)).$$

We define conditional entropy based on the conditional probability of the random variables.

$$H(X \mid Y) = -\sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p_{X,Y}(x,y) \log\left(\frac{p_{X,Y}(x,y)}{p_Y(y)}\right)$$

where $p_{X,Y}(x,y) = \Pr[X = x, Y = y]$ and $p_Y(y) = \Pr[Y = y]$.

3.2 Entropy of Biological Sequences

Let Σ be a finite alphabet. In the context of biological sequence, $|\Sigma| = 4$ for DNA and RNA sequences and 20 for amino acid sequences. Then, from the previous section, we know that the entropy (uncertainty) is

$$H = -\sum_{i=1}^{|\Sigma|} p_i \log(p_i)$$

bits per symbol where p_i is the probability that the symbol $\Sigma[i]$ occurs.

Now, if we fix our attention to a specific position j in the sequence, we can calculate the entropy at the given position as

$$H(j) = -\sum_{i \in \Sigma} f(i,j) \log(f(i,j))$$

where $f(i,j)$ is the frequency that the symbol $\Sigma[i]$ appears at the jth position in aligned sequences.

We define the information at position j of an alignment to be the decrease in uncertainty when the site is aligned.

$$R_{seq}(j) = H - H(j).$$

Then, the information of the entire sequence can be calculated by summing over the site-specific information over all sites of a sequence.

$$R_{seq} = \sum_j R_{seq}(j).$$

3.2.1 Sequence Logo

The site-specific information content can be used to create a plot known as the ***sequence logo***. Sequence logos plot features of aligned sets of each sequence. The horizontal column represents the positions in an alignment, and vertical corresponds to the information for each residue observed at that position, represented using letter corresponding to the amino acid and height proportional to the observed frequency in that position. If a residue is conserved, we should see one or few specific residues dominate that position.

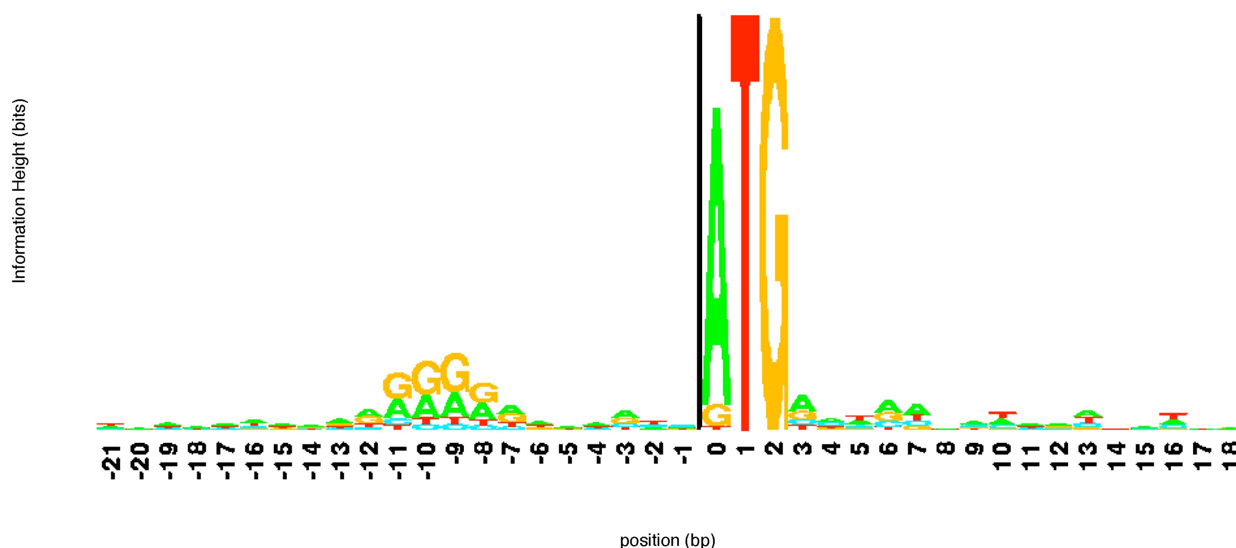


Figure 3.1: Sequence logo showing that the start codon ATG is highly conserved.

3.2.2 Kullback-Leibler Divergence

Kullback and Leibler used Shannon's concept of entropy to compute a measure of relative entropy, thus allowing the comparison of the complexity between two sequences. KL divergence uses the frequencies of symbols or words (i.e. k-mers) and takes the sum of their entropies.

Definition 3.5 (Kullback-Leibler Divergence)**.** *Let P and Q be two discrete distributions defined on the same sample space, say $\mathcal{X}$. The KL divergence or relative entropy from Q to P is defined as*

$$D_{KL}(P\|Q) = \sum_{x\in\mathcal{X}} P(x)\log\frac{P(x)}{Q(x)}.$$

We often model the distribution using the observed frequency of each letter or word. As an example, let's compute the empirical KL divergence of between sequence $S =$ `ATGTGTG` and $T =$ `CATGTG`. First, we compute the frequency of each base in each sequence.

Base / Sequence	A	C	G	T
S	1	0	3	3
T	1	1	2	2

Base / Sequence	A	C	G	T
S	0.14	0	0.43	0.43
T	0.17	0.17	0.33	0.33

Then, we can calculate the KL divergence as follows.

$$\begin{aligned}
D_{KL}(S\|T) &= \sum_{b\in\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}} p_x(b)\log\left(\frac{p_x(b)}{p_y(b)}\right) \\
&= 0.14\cdot\log\left(\frac{0.14}{0.17}\right) + 0 + 0.43\cdot\log\left(\frac{0.43}{0.33}\right) + 0.43\cdot\log\left(\frac{0.43}{0.33}\right) \\
&= 0.24.
\end{aligned}$$

3.3 Kolmogorov Complexity

We now shift our focus to other measures of complexity and will come back to entropy and information theory later when we discuss encoding methods for compression. ***Kolmogorov complexity*** is an abstract measure of incompressibility. In layman's term, Kolmogorov complexity of a string is the length of a program that generates this string. It uses complexity and computability as a proxy to measuring complexity of strings.

Consider

$$x = 0101010101010101$$

and

$$y = 110001010110010$$

Intuitively, y appears to have more information than x, which is simply repeated 01. The idea here is that the more we can compress a string, the less information it contains.

More formally, we define the Kolmogorov complexity of a string as follows.

Definition 3.6 (Shortest Description)**.** *If $x \in \{0,1\}^*$, then the* ***shortest description*** *x, denoted $d(x)$ is the lexicographically minimal string $\langle M, w\rangle$ such that $M(w)$ is an algorithm that halts with only x as output.*

Definition 3.7 (Kolmogorov Complexity)**.** *The* ***Kolmogorov complexity*** *of x, denoted $K(x)$ is $|d(x)|$.*

Immediately from the definition, we have the following results.

Theorem 3.8. *There is a constant c such that for all $x \in \{0,1\}^*$,*

$$K(x) \le |x| + c$$

The amount of information in x is not much more than $|x|$. The Kolmogorov complexity of a string is at most a fixed constant more than its length.

Proof. Define M

M on input w, halts. On any string x, $M(x)$ halts with x on its tape.

$$K(x) \le |\langle M, x\rangle| \le 2|\langle M\rangle| + |x| + 1 \le |x| + c$$

So we can let c be the length of the algorithm that computes the identity function. ☐

Theorem 3.9. *There exists a constant c such that for all $x \in \{0,1\}^*$,*

$$K(xx) \le K(x) + c$$

This says if a string is repetitive, such string has no more information than x.

Proof. Consider the algorithm M defined as follows

$M =$ "on input $\langle N, w\rangle$, where N is an algorithm and w is a string
1. Run N on w until it halts and produces an output string s
2. Output the string ss "

Let $\langle M, w\rangle$ be the shortest description of x, then $\langle N, \langle M, w\rangle\rangle$ is a description of xx.

$$K(xx) \leq |\langle N, \langle M, w\rangle\rangle| \leq 2|\langle N\rangle| + K(x) + 1 \leq K(x) + c$$

So letting $c = 2|\langle N\rangle| + 1$, the theorem holds. □

Corollary 3.10. *There is a constant s such that for all $n \geq 2$ and $x \in \{0,1\}^*$,*

$$K(x^n) \leq K(x) + c\log n$$

In particular, $K((01)^n) \in O(\log n)$.

Proof. Let

$N =$ "on input $\langle n, \langle M, w\rangle\rangle$
run $M(w)$ and print x for n times"

If $\langle M, w\rangle$ is a shortest description of x, then,

$$\begin{aligned} K(x^n) &\leq K(\langle N, \langle n, \langle M, w\rangle\rangle\rangle) \\ &\leq 2|\langle N\rangle| + 2\lceil \log n\rceil + |\langle M, w\rangle| + 2 \\ &= K(x) + O(\log n) \end{aligned}$$

□

3.3.1 Invariance Theorem

The Kolmogorov complexity of a string is independent of the model (as long as we restrict ourselves to classical computational models). The model does not matter. It does not matter what language we use to implement the algorithm considered in the definition of Kolmogorov complexity. If we use another programming lanuage, we will not get significantly shorter description. Intuitively, we can always write a interpreter to translate from one language to another, and the size of the compiler is constant.

Theorem 3.11 (Invariance Theorem)**.** *For every interpreter p, there is a constant c such that for all $x \in \{0,1\}^*$,*

$$K(x) \leq K_p(x) + c$$

This theorem implies that we only change the Kolmogorov complexity of x by a constant c by using a different programming language.

We omit the proof of the invariance theorem, but for those who are interested, the proof can usually be found in any computation complexity text.

However, Kolmogorov complexity is less useful in practice due to its incomputability. That is, the Kolmogorov complexity of a given string cannot be computed under a reasonable model of computation (e.g. Turing machine). This is why we will consider another more practical and efficiently computable measure of complexity.

3.4 Lempel-Ziv Complexity

Definition 3.12 (Reproducibility). *Let S, Q, R be sequences defined over an alphabet Σ. An extension $R = SQ$ of S is **reproducible** from S, denoted $S \to R$, if there exists an integer $p < |S|$ such that $Q[k] = R[p + k - 1]$ for $k = 1, \dots, |Q|$.*

Consider the sequences `AACGT` and `AACGTCGTCG`. The second sequence is reproducible from the first one with $p = 3$. In other words, R is reproducible from S if R can be obtained by copying elements from the pth position in S to the end of S. As each copy extends the length of the new sequence beyond $|S|$, the number of elements copied can be greater than $|S| - p + 1$. Thus, this is a simple copying procedure of S starting from position p, which can carry over to the added part, Q.

Definition 3.13 (Producibility). *A sequence S is **producible** from its prefix $S[1 \dots j]$, denoted $S[1, j] \Rightarrow S$, if $S[1 \dots j] \to S[1 \dots |S| - 1]$ ($S[1 \dots |S| - 1]$ is reproducible from $S[1 \dots j]$).*

For example, `AACGT A C` can be produced from its own prefix with $p = 2$. Note the $|S| - 1$ in the definition of production. Production allows for an extra symbol (that doesn't have to be part of the prefix) at the end of the copying process which is not permitted in reproduction. We call this a ***novel letter production***. Therefore, an extension which is reproducible is always producible but the reverse may not always be true. As an example, note that `AACGT AC C` is producible from its own prefix `AACGT` even though the last `C` is not obtainable from the prefix `AACGT` starting at $p = 2$.

Any sequence S can be constructed using a production process where at the ith step, $S[1 \dots h_{i-1}] \Rightarrow S[1 \dots h_i]$; for the base case, $\epsilon = S[1 \dots 0] \Rightarrow S[1 \dots 1]$. Consider an m-step production of the string S. We define the ***history*** of S, $H(S)$ as follows

$$H(S) = S[1 \dots h_1] \cdot S[h_1 + 1 \dots h_2] \cdots S[h_{m-1} + 1 \dots h_m].$$

The ith group of the history $H_i(S) = S[h_{i-1} + 1, h_i]$ is called the ith ***component*** of $H(S)$.

For example, let $S =$ `AACGTACC`. Below are all valid production histories of S.

`A · A · C · G · T · A · C · C;`   `A · AC · G · T · A · C;`   `A · AC · G · T · ACC.`

In the first production history, every letter is newly generated at each step. In the second production history, the `A` in the second component (`AC`) is copied from the first component. In the last production history, both the second and the last component contains letters copied from a prefix (`A` for the second component and `AC` for the last component).

As we mentioned earlier, every reproducible string is producible but not vice versa.

Definition 3.14. *Let S be a string and $H(S)$ be its history. If $S[1 \dots h_i]$ is not reproducible from $S[1 \dots h_{i-1}]$, we call $H_i(S)$ **exhaustive**. A history is called exhaustive if each of its components, except maybe the last one, is exhaustive.*

In other words, for $H_i(S)$ to be exhaustive, the ith step of the production must be a production only and not a reproduction. That is, $S[1 \dots h_i]$ cannot be constructed using the only copying process and the last letter is introduced using the novel letter production rule. Again, consider $S =$ `AACGTACC` and its three production histories.

An important result from Lempel and Ziv's 1976 paper is that every string has a unique exhaustive history.

Lemma 3.15 (Lempel and Ziv, 1976). *Every string S has a unique exhaustive history $E(S)$.*

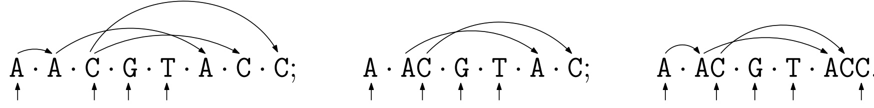


Figure 3.2: The last history is exhaustive because each component has a letter introduced using the novel letter production. The curves on the top of each history represents letters introduced using the copying process as well the source letter. The arrow at the bottom indicates letters introduced using novel letter production.

3.4.1 Complexity of Production history

Definition 3.16 (Production History Complexity)**.** *Let S be a string with some production history $H(S)$. The **production history complexity** of S with respect to $H(S)$ is the number of components in the history $H(S)$ of S, denoted $c_H(S)$.*

Definition 3.17 (LZ Complexity)**.** *The **Lempel-Ziv (LZ) complexity** of S is defined to be*

$$c(S) = \min_{H \in \mathcal{H}_S} \{c_H(S)\}$$

where $\mathcal{H}_S$ is the set of all valid production histories of S.

Having defined the relevant terminology, we now present perhaps the most celebrated results in the paper by Lempel and Ziv.

Theorem 3.18 (Lempel and Ziv, 1976)**.** *$c(S) = c_E(S)$ where $c_E(S)$ is the number of components in the exhaustive history $E(S)$ of S.*

This result is intuitive in view of the innovative nature of an exhaustive component. For, in any given state of a production process, the next production step is longest when the resulting component is exhaustive.

Proof. Let $H_S = \{h_1, h_2, \ldots, h_m\}$ be the set of right indices of the components in some production history $H(S)$ where $m = c_H(S)$ and $1 = h_1 < h_2 < \cdots < h_m = |S|$. H_S defines a partition of S. Similar, let $E_S = \{e_1, e_2, \ldots, e_k\}$ be defined similarly for the exhaustive history $E(S)$. Similarly, $k = c_E(S)$ and $1 = e_1 < e_2 < \cdots < e_k = |S|$.

Let η be a mapping from E_S to H_S, defined by

$$\eta(e_i) = \max\{h \in H_S \mid h \leq e_i\}, \qquad i = 1, 2, \ldots, k.$$

Clearly, $\eta(e_1) = h_1 = 1$ and $\eta(e_k) = h_m = |S|$. If $k > 2$, consider any i such that $2 \leq i \leq k-1$ and let $\eta(e_i) = h_j$ for some j. By definition, $j < m$ and $e_i < h_{j+1}$. Since h_{j+1} is reached in a single production step from h_j and e_i is the furthest position reachable in one step from e_{i-1}, it follows that $e_{i-1} < h_j$.

Hence, for each i such that $2 \leq i \leq k-1$, we have $e_{i-1} < \eta(e_i) \leq e_i$ so $\eta(e_i)$ is indeed a one-to-one mapping from E_S onto some subset of H_S. Since $E(S)$ is unique and $H(S)$ is chosen to be an arbitrary production history, $|E(S)| \leq |H(S)|$ for all production history $H(S)$. □

We also present an upper bound on the LZ-complexity of any sequence with an alphabet size $|\Sigma| = \sigma$.

Theorem 3.19. *For every $S \in \Sigma^n$ where $|\Sigma| = \sigma$,*

$$c(S) < \frac{n}{(1-\epsilon_n)\log_\sigma(n)}$$

where

$$\epsilon_n = 2 \cdot \frac{1 + \log_\sigma \log_\sigma(\sigma n)}{\log_\sigma n}.$$

We omit the proof of this theorem since it is slightly too technical, but the idea behind the proof is as follows.

Proof sketch. The idea is to first bound the maximum possible number of distinct words N that a sequence of length n over the alphabet Σ can be parsed (partitioned) into. Consider the sequence formed by all distinct words of length one, followed by all distinct words of length two, up to all distinct words of length k. The length of such sequence is $n_k = \sum_{i=0}^{k} i\sigma^i$, and the number of distinct words N_k is equal to $\sum_{i=0}^{k} a^i$. We can simplify these two summations using geometric series. This gives us

$$c(S) \leq N_k + 1 < \frac{n_k}{k-1}.$$

Now consider strings of arbitrary length n. Any positive integer n can be expressed as $n = n_k + \Delta_k$ for some k and $0 \leq \Delta_k < (k+1)\sigma^{k+1}$. The increase in the number of distinct words due to the increase in length by Δ_k is at most $\Delta_k/(k+1)$, giving us for any string of arbitrary length n,

$$c(S) < \frac{n_k}{k-1} + \frac{\Delta_k}{k+1} < \frac{n_k + \Delta_k}{k-1} = \frac{n}{k-1}.$$

Further, for each n,

$$k < \log_\sigma(n) < k + 1 + 2\log_\sigma(k+1).$$

And we get $k - 1 > \log_\sigma(n) - \epsilon_n \log_\sigma(n)$ from subtracting $2 + 2\log_\sigma(k+1)$ from both sides of the above inequality $\log_\sigma(n) < k + 1 + 2\log_\sigma(k+1)$. □

3.4.2 Additional Properties of LZ Complexity

In this subsection, we will discuss some additional properties of LZ complexity. We will present results without proof, but we encourage interested readers to read Lempel and Ziv's original paper for a more rigorous presentation of the results with proofs.

Theorem 3.20 (Almost all sequences of sufficiently large length are complex)**.** *For every positive ϵ, and alphabet size $|\Sigma| = \sigma$,*

$$\lim_{n\to\infty} \Pr\left(c(S) < \frac{n(1-\epsilon)}{\log_\sigma n} \mid |S| = n\right) = 0.$$

Theorem 3.21 (Subadditivity)**.** *$c(QS) \leq c(Q) + c(S)$.*

Proof. Let $E(Q)$ and $E(S)$ be exhaustive histories of Q and E, respectively. Then, $H(QS) = E(Q) \cdot E(S)$ is indeed a history of QS. Then, by Theorem 3.18,

$$c(QS) \leq c_H(QS) = c_E(Q) + c_E(S) = c(Q) + c(S).$$

□

This property is very useful because it allows us to easily define a distance function based on the difference in LZ complexity of two given sequences. We will come back to this idea in the Part IV when we discuss alignment-free sequence comparisons. For a sneak peek, consider the following function.

$$d(S, Q) = \frac{c(SQ) - \min\{c(S), c(Q)\}}{\max\{c(S), c(Q)\}}.$$

It can be shown that d is a distance function. That is, it satisfies (1) identity; (2) symmetry; and (3) triangle inequality. It also turns out to be a good measure of evolutionary distance for genomic sequences and can be used to construct phylogenetic trees.

In a sense, LZ complexity is a more concrete notion of complexity compared to say Kolmogorov complexity. Kolmogorov complexity does not provide an explicit construction of the minimum-length description and the Kolmogorov complexity itself may not even be computable, whereas LZ complexity is not noly easily computable, but also has a more constructive definition. The Lempel-Ziv complexity of a string can be computed in $O(n)$ time using the following algorithm. Without loss of generality, suppose that the input to the following procedure is a binary string.

```
LZ-Complexity(S)
 1  p, u, v = 0, 1, 1
 2  v-max = v
 3  c_S = 1
 4  while u + v ≤ |S|
 5      if S[p + w] == S[u + v]
 6          v = v + 1
 7      else
 8          v-max = max{v, v-max}
 9          p = p + 1
10          if i == u
11              c_S = c_S + 1
12              p, u, v = 0, u + v-max, 1
13              v-max = v
14          else v = 1
15  if v ≠ 1
16      c_S = c_S + 1
17  return c_S
```

The algorithm keeps track of three pointers, p, u, and v. p is the p-pointer in the definition of reproducibility – the starting position of a reproduction. u is the length of the current prefix that is used to reproduce new letters in the current component. v is the length of the current component and v-max is the final length of the current component. We iteratively check each possible positions for p to find the max length of each component. After the max component is found, we reset p and check the next component. Whenever we finish a new component, we update c_S.

This greedy algorithm correctly computes the LZ complexity in linear time because $u + v$ is monotonically increasing by at least one each iteration so the loop runs exactly $|S|$ iterations.

Bibliography

The desired property of an entropy measure is taken directly from Shannon's 1948 paper [23]. Kullback-Leibler divergence is introduced in the paper [16]. Kolmogorov complexity is often introduced in an introductory course on computation complexity theory. A good reference is Introduction to the Theory of Computation by Sipser [24]. Lempel-Ziv complexity was introduced in the 1967 paper by Lempel and Ziv [17]. It is widely used in many compression algorithms.

Chapter 4

Entropy Coding

In this chapter, we will use the concepts discussed in the previous chapter to come up with concise encoding of data. This is the foundation of many compression algorithms and will also help us with designing and analyzing succinct data structures, which we will discuss in more details in the next part.

We will focus on the problem of finding the shortest code for elements from a (potentially large) universe.

Problem 4.1 (Shortest Code Word)**.** Let U be a universe. Find an encoding f that outputs the shortest code that uniquely identifies every element in U. In other words, f is injective.

4.1 Worst-Case Entropy

Definition 4.2 (Worst-Case Entropy)**.** *Let U be our universe. We define the **worst-case entropy** of U as*

$$H_{wc}(U) = \log_2 |U|. \tag{4.1}$$

Now suppose we were to come up with a coding scheme that uniquely identify each element in the universe U. If we require the codes to all have the same length, thhen the length of the code is at least $\log_2 |U|$ and the theoretical optimum is $\lceil \log_2 |U| \rceil$. If codes are allowed to have various lengths, the length of the longest code must be at least $\log_2 |U|$. For variable-length encoding, we can modify Problem 4.1 so that it seeks to minimize the expected code length. The expected code length is defined as follows.

$$\bar{\ell} = \sum_{u \in U} \Pr(u) \cdot \ell(u).$$

Let's look at some examples of worst-case entropy.

Example 4.3. *A coin has worst-case entropy of 1 bit because the only two possible states are head and tail. Meanwhile, a die has worst-case entropy of $\log_2 6 \leq 3$ bits because there are 6 possible states.*

If our universe U contains all length-n strings over the alphabet Σ where $|\Sigma| = \sigma$, then

$$H_{wc}(U) = \log_2 \sigma^n = n \log_2 \sigma.$$

Equation 4.1 is called the worst-case entropy because it is equal to the Shannon entropy when all outcomes in the sample space are equally probable. Recall that the Shannon entropy of a random variable X over the sample space Ω is $-\sum_{\omega \in \Omega} \Pr(\omega) \log(\Pr(\omega))$. When every outcome is equally probable,

$$H(X) = \sum_{\omega \in \Omega} \frac{1}{|\Omega|} \log_2 |\Omega| = \log_2 |\Omega| = H_{wc}(\Omega).$$

4.2 Zero-Order Empirical Entropy

In practice, we often do not have knowledge of the random variable that emits the data of which we want to measure the entropy. In this case, we define a random variable that models a *memoryless* source, "trained" by the frequency of occurrence of the observed (empirical) data.

Definition 4.4 (Zero-Order Empirical Entropy)**.** *Suppose we have a memoryless binary source that emits a binary string B. We define the zero-order empirical entropy as*

$$H_0(B) = H(X) \tag{4.2}$$

where $X \sim \mathrm{Bern}(m/n)$. By definition of the Bernoulli distribution, if we let m be the number of 1's and n to be the length of B,

$$H_0(B) = H(X) = \frac{m}{n} \log_2 \frac{n}{m} + \frac{n-m}{n} \log_2 \frac{n}{n-m}.$$

4.3 Symbol Codes

Having introduced the necessary definitions, we will know look at how to utilize the notion of entropy and empirical entropy in designing and analyzing coding schemes.

Definition 4.5 (Symbol Code)**.** *A* ***symbol code*** *(variable-length code) is a function $C : \Sigma \to A^*$ where Σ is the source alphabet and A is the code alphabet. The output of C is called a* ***codeword****. The extension of C is the function $C^* : \Sigma^* \to A^*$ such that*

$$\forall n \geq 0,\, \forall x_1, \ldots, x_n \in \Sigma,\, C^*(x_1 \cdots x_n) = C(x_1) \cdots C(x_n).$$

Morse code is an example of a symbol code.

Example 4.6 (Morse Code)**.** *Morse code is a symbol code defined for the source alphabet $\{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}\} \cup \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}$ and code alphabet $\{\bullet, -, \sqcup\}$ where $\sqcup$ represents a pause or empty space.*

Let's consider some other example.

Example 4.7. *Let $\Sigma = \{\mathtt{a}, \mathtt{g}, \mathtt{c}, \mathtt{t}\}$ and $A = \{\mathtt{0}, \mathtt{10}, \mathtt{110}, \mathtt{111}\}$. Let C be defined as follows.*

$$C(\mathtt{a}) = \mathtt{0} \qquad C(\mathtt{g}) = \mathtt{10} \qquad C(\mathtt{c}) = \mathtt{110} \qquad C(\mathtt{t}) = \mathtt{111}.$$

So $C^*(\mathtt{aacg}) = C(\mathtt{a}) \cdot C(\mathtt{a}) \cdot C(\mathtt{c}) \cdot C(\mathtt{g}) = \mathtt{0011010}$.

Example 4.8 (An ambiguous code)**.** *Let C be defined as follows.*

$$C(\mathtt{a}) = \mathtt{0} \qquad C(\mathtt{g}) = \mathtt{1} \qquad C(\mathtt{c}) = \mathtt{01} \qquad C(\mathtt{t}) = \mathtt{10}.$$

However, notice that $C^*(\mathtt{ag}) = \mathtt{01}$ but $C^*(\mathtt{c}) = \mathtt{01}$ as well. Therefore, this code is ambiguous because $\mathtt{01}$ can be interpreted in two different ways.

What exactly is it about our last example that made it ambiguous? The coding function itself is still injective, but the sequence of codewords generated is not uniquely decodable.

Definition 4.9 (Unique Decodability)**.** *C is* ***uniquely decodable*** *if C^* is surjective.*

If we compare the last example with previous examples of uniquely decodable codes, we can notice that the only property about the code appears to be that in the last example, some codeword is a prefix of other codewords. For example, the codeword for `a` is a prefix of the codeword for `c`; and the codeword for `b` is a prefix of the codeword for `d`. This makes it impossible to differentiate between `ab` and `c` or `ba` and `d`. On the other hand, all the uniquely decodable codes that we have looked at so far share the same property that no codeword is a prefix of another codeword. We call this type of codes ***prefix-free*** codes.

4.3.1 Prefix-Free Code and Huffman Coding

Definition 4.10 (Prefix-Free Code)**.** *C is a **prefix-free code** if no codeword is a prefix of another codeword.*

Prefix-free codes are sometimes also referred to as prefix codes or instantaneous codes.

Proposition 4.11. *If C is prefix-free, then C is uniquely decodable.*

A simple way to construct a prefix code is to use Huffman's greedy algorithm. Huffman coding is a prefix code where more frequent source code has shorter codeword. Huffman's algorithm utilizes a labeled binary tree where the leaves are characters in the source alphabet. All left edges (edge between a parent and its left child) are labeled `0` while all right edges are labeled `1`. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree.

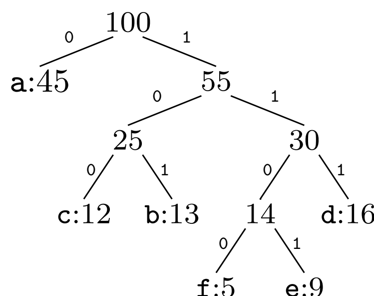


Figure 4.1: A Huffman tree for the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$ with frequency $45, 13, 12, 16, 9, 5$, respectively.

The codeword for a given character in the source alphabet is the concatenation of labels on the root to leaf path. For example, the codeword for `f` in the above example is `1100`.

Decoding is also easy with a Huffman tree. We parse a code by starting from the root and reading code characters while following the edge labels until we reach a leave, after which we start over and start processing the next codeword. For example, given a code `001011101`, we start processing by reading the first character in the code, `0`. We immediately encounters a leaf, giving us `a 01011101`. We repeat this process for the next code character `0`, resulting in `a a 1011101`. Next, start from `1` and continue to process code characters until we reaches a leaf. In this case, we read three characters `101` and gets `a a b 1101`. Finally, start from `1` and repeat the same process, which gives us `a a b e` as our final result.

The construction of a Huffman tree uses a greedy strategy. We start from a set of disjoint nodes and join root nodes (nodes with no parents) iteratively by their frequencies, starting from the nodes with the lowest frequency, until we get a full binary tree.

```
HUFFMAN-ENCODE(S)
1  Q = PRIORITY-QUEUE(S)
2  for i = 1 to |S| - 1
3        z = CREATE-NODE()
4        z.left = EXTRACT-MIN(Q)
5        z.right = EXTRACT-MIN(Q)
6        z.freq = z.left.freq + z.right.freq
7        INSERT(Q, z)
8  return EXTRACT-MIN(Q)
```

4.3.2 Kraft-McMillan Inequality

We have now seen a way to construct a prefix-free code, but what is the minimum possible length of the codewords output by a prefix-free coding scheme? Kraft-McMillan inequality gives a necessary and sufficient condition for the existence of a prefix-free code for a given set of codeword length.

Definition 4.12. *A B-ary code is a code $C : \Sigma \to A^*$ such that $|A| = B \geq 1$.*

We now state the theorem.

Theorem 4.13 (McMillan's theorem)**.** *For any uniquely decodable B-ary code C,*

$$\sum_{x \in \Sigma} \frac{1}{B^{\ell(x)}} \leq 1$$

where $\ell(x) = |C(x)|$.

Theorem 4.14 (Kraft's theorem)**.** *If $\ell : \Sigma \to \{0, 1, \ldots, \}$ satisfies*

$$\sum_{x \in \Sigma} \frac{1}{B^{\ell(x)}} \leq 1$$

then there exists a B-ary uniquely decodable prefix-free code C such that $|C(x)| = \ell(x)$ for all $x \in \Sigma$.

These two theorems combined tell us that there is an instantaneous binary code with lengths $\ell_1, \ldots, \ell_n$ such that $\sum_{i=1}^{n} 1/B_{\ell_i} \leq 1$ if and only if there is a uniquely decodable code with these lengths.

Proof of McMillan's Inequality. In the following proofs, we use n to denote the size of the source alphabet $|\Sigma|$. Let C be a uniquely decodable B-ary code. Without loss of generality, let $\ell_1 \leq \cdots \leq \ell_n$ be the lengths of the codewords, sorted in non-decreasing order of their lengths. Note $\ell_n = \max_i\{\ell_i\}$. Let

$$r = \sum_{i=1}^{n} \frac{1}{B^{\ell_i}}. \tag{4.3}$$

For any positive integer n,

$$r^k = \left(\sum_{i=1}^{n} \frac{1}{B^{\ell_i}}\right)^k = \sum_{x_{i_1}, \ldots, x_{i_k} \in \Sigma} \frac{1}{B^{\ell_{i_1}}} \cdots \frac{1}{B^{\ell_{i_k}}}. \tag{4.4}$$

The summation on the RHS is over all combinations of source alphabet symbols $x_{i_1} \cdots x_{i_k} \in \Sigma^k$. Define $j = \ell_{i_1} + \cdots + \ell_{i_k}$. Let $N_{j,k}$ be the number of sequences of k codewords with total length j. Then, clearly, since C is uniquely decodable (and thus is injective),

$$N_{j,k} = \left| \left\{ x_{i_1} \cdots x_{i_k} \in \Sigma^k : \sum_{t=1}^{k} \ell_{i_t} = j \right\} \right| \leq B^j. \tag{4.5}$$

We can then rewrite Equation 4.4 as

$$r^k = \sum_{x_{i_1},\ldots,x_{i_k} \in \Sigma} \frac{1}{B^{\sum_{t=1}^{k} \ell_{i_t}}} = \sum_{j=1}^{n\ell_n} \frac{N_{j,k}}{2^j} \tag{4.6}$$

by splitting the summation based on all possible $j = \sum_{t=1}^{k} \ell_{i_t}$. By Inequality 4.5,

$$r^k = \sum_{j=1}^{k\ell_n} \frac{N_{j,k}}{2^j} \leq \sum_{j=1}^{k\ell_n} \frac{2^j}{2^j} = k\ell_n. \tag{4.7}$$

This implies that $r \leq 1$ because otherwise if $r > 1$, r^k will dominate over $k\ell_n$, creating a contradiction to the inequality $r^k \leq k\ell_n$. Therefore, $r = \sum_{i=1}^{n} \frac{1}{B^{\ell_i}} \leq 1$ so the McMillan inequality holds. □

We also sketch an algorithmic proof of Kraft's theorem. We will show how to construct the prefix-free code satisfying the properties of the consequence of Kraft's theorem and prove that the algorithm is correct.

Proof of Kraft's Theorem. Assume that Kraft's inequality holds:

$$\sum_{i=1}^{n} \frac{1}{B^{\ell_i}} \leq 1. \tag{4.8}$$

We would like to construct a prefix-free code C with codewords of lengths exactly $\ell_1, \ldots, \ell_n$. Without loss of generality, order the lengths so that $\ell_1 \leq \cdots \leq \ell_n$. Like we did when constructing Huffman code, we consider a binary tree that represents the code. The codewords correspond to leaves of the tree and each branch corresponds to adding another code symbol. See Figure 4.1 for an example. The tree is full, meaning that each node has either 0 or 2 children. We can extend the tree into a perfect binary tree (every internal node has exactly two children) by extending the tree to the depth of the longest codeword. After the extension, each codeword that was previously a leaf either remains a leaf or becomes an internal node. If a codeword becomes an internal node, let the subtree rooted at this internal node represent the codeword. See Figure 4.2 to see an example coding tree and its extension. More generally, for B-ary code, the tree is B-ary, but for the sake of our argument, we assume without loss generality that $B = 2$.

The shorter the codeword, the larger the subtree in the extended tree. For a binary code, the fraction of the leaves belonging to a codeword of length ℓ in the extended coding tree is $1/2^\ell$. For example, in the extended tree shown in Figure 4.2, the each codeword of length 3 takes up 1 out of the 8 leaves, whereas each codeword of length 2 takes up 2 out of 8 leaves because the subtree for codewords of length 2 is rooted at the second to last level.

The idea is to create a one-to-one mapping between the codewords in the extended tree and the corresponding codeword lengths. To achieve this, we repeat the following steps for $i = 1, \ldots, n$, from shortest to longest:

1. Pick a node at depth ℓ_i that is not a subtree previously used, and let the code for codeword i be the one at that node.

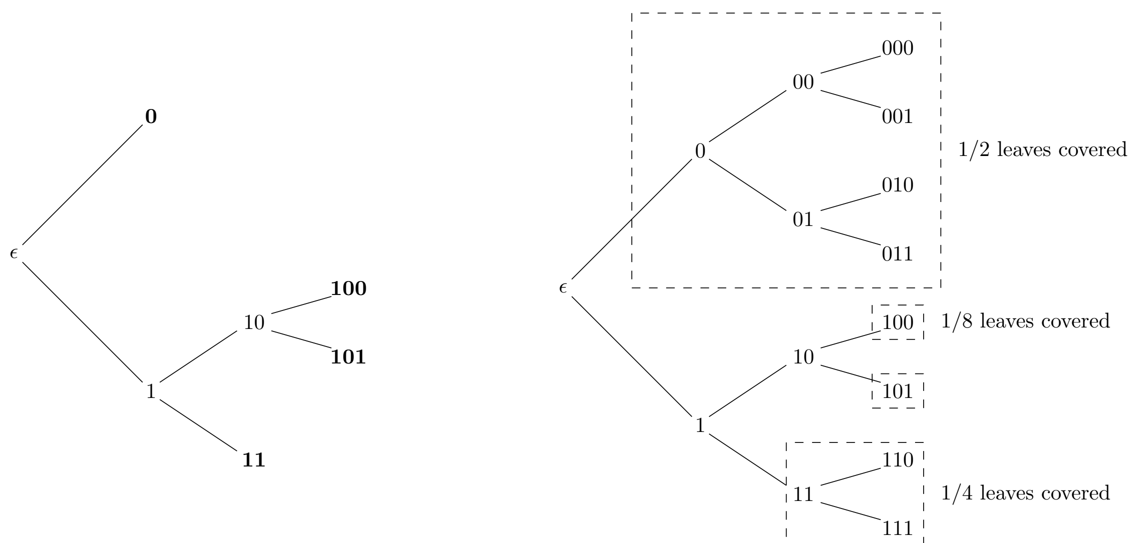


Figure 4.2: A coding tree and its extension. The codewords in the extended tree is indicated by boxes.

2. Mark all nodes in the subtree rooted at this node as being used so that they are not considered in the subsequent iterations.

We claim that at the beginning of each iteration, there will always be an unused node. There are 2^{ℓ_b} nodes at depth ℓ_b because the tree is perfect. When we pick a node at depth ℓ_a, the number of nodes that become unavailable at depth ℓ_b is $2^{\ell_b - \ell_a}$. When we pick a node at some depth ℓ_j, after having picked earlier nodes at depth ℓ_i where $i < j$ and $\ell_i \leq \ell_j$, the number of nodes left to choose from at depth ℓ_j is

$$2^{\ell_j} - \sum_{i=1}^{j-1} 2^{\ell_j - \ell_i} = 2^{\ell_j} \left(1 - \sum_{i=1}^{j-1} \frac{1}{2^{\ell_i}}\right).$$

By assumption, $\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1$ so $\sum_{i=1}^{j-1} \frac{1}{2^{\ell_i}} < \sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1$, and it follows that

$$2^{\ell_j} - \sum_{i=1}^{j-1} 2^{\ell_j - \ell_i} > 0.$$

Therefore, there will always be a node available to be chosen as the codeword for each length ℓ_i. By induction on the number of iterations, there is a codeword for ℓ_i for all $i = 1, \ldots, n$. By excluding all nodes in a subtree when the root of the subtree is selected, we ensure that our code is prefix-free. The code constructed as such is indeed prefix-free and has codewords of every lengths in $\ell_1, \ldots, \ell_n$. □

4.4 Lower Bounds

In this last section of the chapter, we present some lower bound results regarding symbol codes, linking data compression to entropy and complexity. The idea behind the lower bound results is that each symbol $x_i \in \Sigma$ of the source alphabet contains $-\log_2 p_i$ bits of information, so the encoding of this symbol should have at least that many bits or otherwise we will “lose” information during compression. We will now formalize this intuition and prove a theorem regarding the lower bound on the expected codeword length.

Lemma 4.15. *For any two discrete probability distributions p and q over the source alphabet $|\Sigma|$,*

$$-\sum_{i=1}^{|\Sigma|} p_i \log(p_i) \leq -\sum_{i=1}^{|\Sigma|} p_i \log(q_i).$$

Proof. For all $x > 0$, $\ln x \leq x - 1$ so $\log_2 x \leq (x-1)/\ln 2$. Subtract the RHS from LHS and consider

$$\sum_{i=1}^{|\Sigma|} p_i \left[\log_2\left(\frac{1}{p_i}\right) - \log_2\left(\frac{1}{q_i}\right)\right] = \sum_{i=1}^{|\Sigma|} p_i \log_2\left(\frac{p_i}{q_i}\right).$$

By our observation that $\log_2 x \leq (x-1)/\ln 2$,

$$\sum_{i=1}^{|\Sigma|} p_i \log_2\left(\frac{p_i}{q_i}\right) \leq \frac{1}{\ln 2}\sum_{i=1}^{|\Sigma|} p_i\left(\frac{q_i}{p_i} - 1\right) = \frac{1}{\ln 2}\left(\sum_{i=1}^{|\Sigma|} p_i - \sum_{i=1}^{|\Sigma|} q_i\right) = 0.$$

The last equality follows because p and q are probability distributions. So it follows that RHS − LHS ≥ 0 so LHS $\leq$ RHS. □

Using this lemma, we can prove the main theorem of this section.

Theorem 4.16. *Let X be a random source with a finite source alphabet. Any uniquely decodable binary code for X must have expected length of at least $H(X)$.*

Proof. Let n be the source alphabet size so the codeword lengths are $\ell_1, \ldots, \ell_n$. Let $r = \sum_{i=1}^{n} 2^{-\ell_i}$ and $q_i = 2^{-\ell_i}/r$. $q_1, \ldots, q_n$ forms a probability distribution. This is the distribution of the codeword lengths. Then,

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \leq -\sum_{i=1}^{n} p_i \log_2 q_i = \sum_{i=1}^{n} p_i \log_2(2^{\ell_i} \cdot r) = \sum_{i=1}^{n} p_i(\ell_i + \log_2 r).$$

Since the code is uniquely decodable, $r \leq 1$ by Kraft-McMillan's inequality. Thus, $\log_2 r \leq 0$. Hence,

$$\bar{\ell} = \sum_{i=1}^{n} p_i \ell_i \geq \sum_{i=1}^{n} p_i(\ell_i + \log_2 r) \geq H(X).$$

□

4.4.1 Shannon-Fano Codes

Shannon-Fano codes are constructed so that the codeword for symbol i with emission probability p_i has length

$$\ell_i = \lceil \log_2 1/p_i \rceil.$$

Kraft's inequality tells us such code exists because

$$\sum_{i=1}^{|\Sigma|} \frac{1}{2^{\ell_i}} \leq \sum_{i=1}^{|\Sigma|} \frac{1}{2^{\log_2(1/p_i)}} = \sum_{i=1}^{|\Sigma|} p_i = 1.$$

Theorem 4.17. *The expected length of a Shannon-Fano code for a source X with symbol distribution p is $1 + H(X)$.*

Proof.

$$\begin{aligned}\sum_{i=1}^{n} p_i \ell_i &= \sum_{i=1}^{n} p_i \lceil \log_2(1/p_i) \rceil \\ &< \sum_{i=1}^{n} p_1 (1 + \log_2(1/p_i)) \\ &= \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} p_i \log_2(1/p_i) \\ &= 1 + H(X).\end{aligned}$$

□

This corollary immediately follows from the theorem.

Corollary 4.18. *For any source X, any uniquely decodable code C has expected length $\bar{\ell}$ such that*

$$H(X) \leq \bar{\ell} \leq H(X) + 1.$$

Bibliography

For a survery on coding algorithms including Shannon-Fano code, see [13]. Kraft-McMillan's inequality are described separately in Kraft's PhD thesis and a published paper by McMillan [15] [20].

Part III

Index Data Structures

Chapter 5

Suffix Tree

5.1 Suffix Tries

Let us recall the definition of a suffix.

Definition 5.1 (Suffix)**.** *For any string S, $S[i \dots j]$ is the* ***substring*** *starting at position i and ending at position j; $S[1 \dots i]$ is the prefix of S ending at i; and $S[j \dots |S|]$ is the* ***suffix*** *of S starting at position j. A proper substring, prefix, or suffix is a substring, prefix, or suffix that is neither the entire string S nor the empty string.*

Then, we define a trie and a suffix trie as follows.

Definition 5.2 (Suffix Trie)**.** *A* ***trie*** *is the smallest tree such that each edge is labeled with a character from the alphabet Σ, each node has at most one outgoing edge labeled with c for each $c \in \Sigma$, and each node has a key that is the concatenation of the edge labels along the path from the root to that node. A* ***suffix trie*** *is a trie where each root-to-leaf path represents a suffix.*

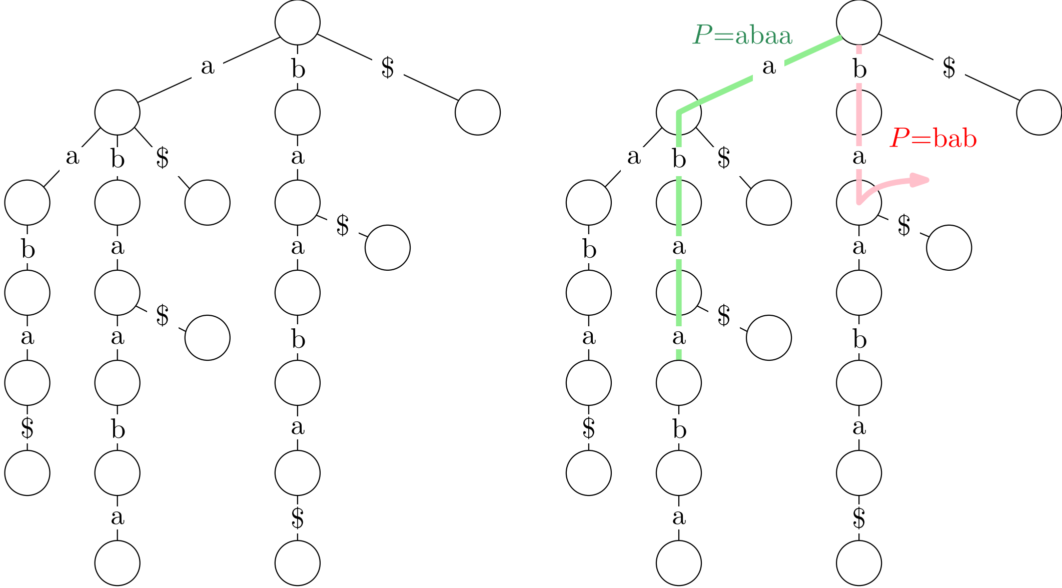


Figure 5.1: Suffix trie for $T = abaaba$. On the right: the search path for $P = abaa$ and $P = bab$. When searching for a pattern that is not in T, we "fall off" the trie.

In a regular tree (e.g. binary search tree), the key is stored at each node. In a trie, the keys are *implicitly* represented by the edge labels along the path. Figure 5.1 shows a suffix trie constructed for $T = abaaba$.

It is important to add the terminator character `$` at the end of the string. If we remove the terminator `$`, it is not hard to see the result trie may no longer be a valid suffix trie. We assume that `$` is *lexicographically smaller than all characters* in Σ.

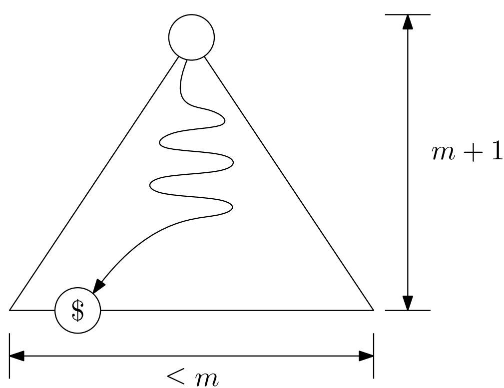


Figure 5.2: Max width and height of a suffix trie. The path from the root to the deepest leaf represents the longest suffix (the whole string plus the terminator).

5.1.1 Search in Suffix Trie

Search for Pattern: It is easy to search for a pattern P given a suffix trie. We can **start from the root and follow the edges labeled with the characters** in P until we either finish reading the pattern and find a match, or "fall off" the trie, in which case we can return that a match is not found.

```
SEARCH-TRIE(P, T)
1  cur = T.root
2  for c in P
3      if c ∉ cur.edges
4          return FALSE
5      else cur = cur.edges[c]
6  return cur ≠ NULL
```

Assume that at each node, we maintain a hash table for each outgoing edges. Then, the algorithms runs in expected time $\Theta(|P|)$.

Search for Suffix: Similarly, if we want to see if a given pattern P is a suffix of T, we can run the same algorithm and checks if the node at the end of the path has an outgoing edge labeled `$`.

Search for Number of Occurrences: If we are interested in the number a pattern P occurs as a substring in T, we can run SEARCH-TRIE. Once we arrive at the end of our search path, we run a **depth-first search** from the node at the end of the search path and count the number of leaf nodes reachable from that node. Since a trie is a tree, DFS runs in $O(|V|)$ time. In this case, it takes $O(|P| + |T|)$ time to find the number of occurrences of a given pattern.

SEARCH FOR LONGEST REPEATED SUBSTRING: Find the deepest (internal) node with more than one children.

5.1.2 Space Complexity of Suffix Trie

The simplest way to construct a suffix tree is to first construct a suffix trie and convert it to a suffix tree by repeatedly coalescing the paths. This takes $O(n^2)$ time and space. It takes $O(n^2)$ space because we need to store the intermediate suffix trie.

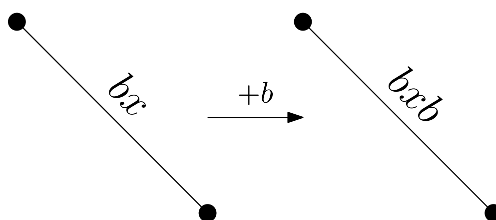


Figure 5.3: Type 1 insertion for suffix bx of $S = axabx$.

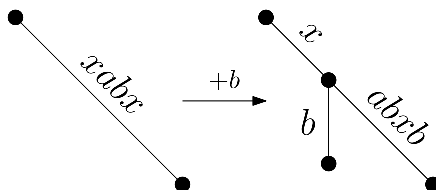


Figure 5.4: Type 2 insertion for suffix x of $S = axabx$.

5.2 Ukkonen's Linear-Time Construction

5.2.1 Types of Extensions

Suppose that we have $S[j \dots i] = \beta$ be a suffix of $S[1 \dots i]$. In some iteration j, the algorithm finds the end of β and extends the path by adding $S[i+1]$ to the path. This ensures that the suffix $S[j \dots i+1]$ is included in the new tree. Observe that there are three types of insertions:

Type 1. In the current tree, path β ***leads to a leaf***. In this case, $S[i+1]$ is ***added to the end of the label*** on that edge.

Type 2. There is ***no path*** from the end of β that starts with character $S[i+1]$, but at least one labeled path continues from the end of β. In this case, a ***new leaf edge*** starting from the end of β is created and labeled $S[i+1]$, which leads to a ***new leaf node*** with number j.

Type 3. Some ***path*** from the end of β starts with character $S[i+1]$. In this case, $\beta \cdot S[i+1]$ is ***already in the implicit suffix tree***. So we ***do nothing***.

5.2.2 Suffix Links

Definition 5.3 (Suffix Link)**.** Let $x\alpha$ be an arbitrary string, where $x \in \Sigma$ is a *single character* and $\alpha \in \Sigma^*$ is a (possibly empty) *substring*. For an internal node v with root-to-node path label $x\alpha$, if there is another node $s(v)$ with root-to-node path label α, then we create a pointer from v to $s(v)$, called a ***suffix link***.

The reason we have suffix link is because we want to *access the insertion point (end of a suffix) efficiently.* Suppose we insert a new character to the sequence $x\alpha$. Once we inserted the new character to the suffix $x\alpha$, we also need to insert the character to the end of α. Without suffix link, we would have to traverse back to the root and search for the insertion point all over again. The use of suffix link is especially useful for jumping from one suffix to the next during extension.

Moreover, the suffix links induce a subtree called the ***suffix link tree***. More formally, given a text T

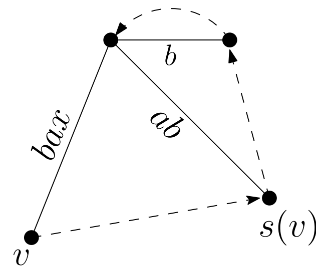


Figure 5.5: A suffix link from v (representing xab) to $s(v)$ (representing ab). The other two suffix links are also shown. Note that if a node's path label has no proper suffix, we create a link to the root.

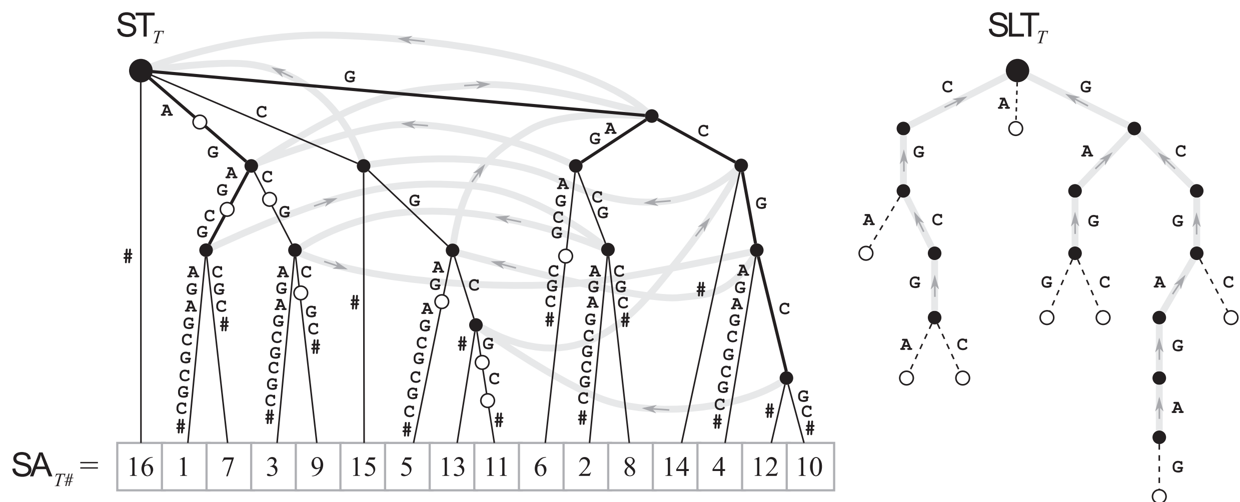


Figure 5.6: A suffix tree and its corresponding suffix link tree for $T =$ `AGAGCGAGAGCGCGC`.

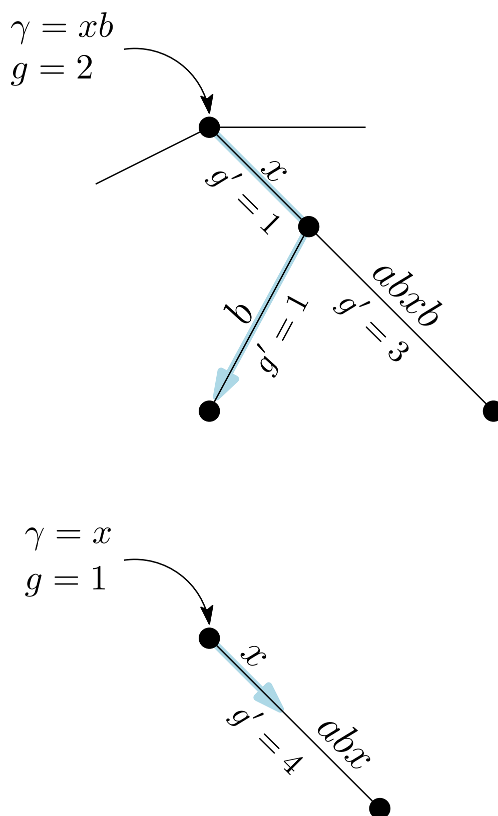


Figure 5.7: Top: Case 1 where $g \geq g'$, skip to the next node; Bottom: Case 2 where $g < g'$, go the the g-th character on the current edge.

represented by a suffix tree (V, E) with suffix links, let $\ell(v)$ denote the label on the path from the root to v. Then, $L = \{(v, a) \mid v \in V, s(v) \in V, \ell(v) = x\ell(s(v)), x \in \Sigma\}$ be the set of all suffix links, and we define the tree $\mathrm{SLT}(T) = (V, L)$ labeled by ℓ as the ***suffix link tree***.

5.2.3 Count and Skip

After reaching $s(v)$ via a suffix link, we still need to travel down the path labeled γ in order to add a new character at the end of γ, either by branching off or creating a new leaf node. However, this walk along the γ path takes $O(|\gamma|)$ if implemented directly. An alternative to this would be to ***store the number of characters of on each edge*** and skip nodes wheneve we can.

Let g denote the length of γ, the next suffix to which we want to append the new character. We start the search for the insertion point starting from $s(v)$.

Recall that no two edges coming out of $s(v)$ can have labels starting with the same character. We can then use one comparison to determine the edge that we need to follow. In particular, let $h = 1$ initially and before each iteration, compare the hth character of γ with the first character of every edge coming out of the current node. Let g' be the number of characters on the edge that we have just identified. Then, consider the following two cases:

1. $g \geq g'$: Skip to the node at the end of the edge. Set $g = g - g'$, $h = h + g'$, and repeat.

2. $g < g'$: Skip to the gth character on the edge and stop.

5.2.4 Edge Label Compression

Another issue with our high-level "algorithm" for Ukkonen's algorithm is that every edge is explicitly labeled with the suffix they represent. A label of an edge can be as large as $\Theta(m)$, and there can be at most $\Theta(m)$ edges, making the total space required for a suffix tree $\Theta(m^2)$ in this case. This makes it impossible to build such a tree in $O(m)$ time. Fortunately, this issue can be solved using a simple trick: instead of storing the strings explicitly, we can store a pair of indicies representing the ***starting and ending position of the substring*** represented by each edge. That way, each edge can be maintained using only $\Theta(\log m)$ space.

In the ***word RAM model***[1], we assume that every word of size $\log m$ bits can be read and written efficiently in constant time. Hence, it is more plausible to build a suffix tree with compressed edge labels in linear time.

5.2.5 Key Observations

No More Insertions After Type 3

In any given phase $i+1$, if there is a Type 3 extension j (the new suffix $S[j \dots i+1]$ is already in the tree), then any further extensions in the current phase will also be of Type 3. When there is a Type 3 extension, the path labeled $S[j \dots i]$ in the current tree must have already contained $S[i+1]$. Then clearly, so does $S[j' \dots i]$ for all $j < j' \leq i+1$ because they are all suffixes of $S[j \dots i]$.

Once a Leaf, Always a Leaf

At some point in the algorithm, if a leaf j is created for the suffix starting at position j, then the leaf will remain a leaf throughout the algorithm. This is because the algorithm never extends a leaf. If a path leads to a leaf, it will be a Type 1 insertion, in which case we extend the edge label without explicitly adding new nodes.

Bibliography

Ukkonen's linear time construction algorithm was presented in [25]. The presentation of this chapter is largely based on the book *Algorithms on Strings, Trees, and Sequences* by Gusfield [9].

[1] In complexity theory, we have focused on Turing machine as our preferred model of computation. However, in analysis of algorithms, we usually use the word RAM model (often without explicitly stating it) as it is a more realistic model of how modern computers work.

Chapter 6
Suffix Array

Suffix array is another important data structure for string matching. It stores the positions of all suffixes of a string T sorted in lexicographic order. More formally,

Definition 6.1 (Suffix Array)**.** *The **suffix array*** SA_T *of a text* $T = t_1 t_2 \ldots t_n$ *is a permutation of* $[1, \ldots, n]$ *such that* $SA_T[i] = j$ *if and only if* $T[j \ldots n]$ *has position* i *in the list of all suffixes of* T *taken in lexicographic order.*

Before we dive into the construction and applications of suffix arrays, we will first look at some techniques for sorting strings that will be useful for coming up with suffix array construction algorithms.

6.1 String Sorting

6.1.1 Sorting Strings of Fixed Length

Recall the counting sort algorithm. Let A be a sequence of integers ranging from 1 to k. Counting sort sorts the list by first building a vector *count* of size n where $count[i]$ stores the number of entries of A that is i, and then printing the sorted list by printing $count[i]$ copies of i for $i = 1, \ldots, k$.

```
Counting-Sort(A, B, k)
 1  count = empty array of length k
 2  offset = empty array of length k
 3  offset[1] = 1
 4  for i = 1 to k
 5      count[i] = 0
 6  for j = 1 to |A|
 7      count[A[j]] = count[A[j]] + 1
 8  for i = 1 to k
 9      offset[i] = offset[i − 1] + count[i − 1]
10  for j = 1 to |A|
11      B[offset[A[j]]] = A[j]
12      offset[A[j]] = offset[A[j]] − 1
```

In this pseudocode, at the end of the loop on Line 6-7, $offset[i]$ stores the position of the next i in the sorted order when we move from the *front to the end of the array*. *offset* is sometimes referred to as a *block pointer array*. Alternatively, we can implement without the block pointer array by modifying C so that $C[i]$ stores the number of elements less than or equal to i, which tells us the position of the next i in sorted order as we move from the *end to the front of the array*. Notice that counting sort beats the well-known $\Omega(n \log n)$ lower bound for sorting because it is not a comparison-based sorting algorithm.

Assume now that A is an array of key-value pairs ($A[i].p$, $A[i].v$). Further, assume that A is already sorted by the value v of each entry. We run counting sort for another round on the primary key p. Repeating this

inductively for every digit of a number from the least significant bit to the most significant bit (or in our case, every letter of a string), we will have ourselves ***radix sort*** – an algorithm for sorting list of arbitrary numbers or strings of fixed length.

Sort ↓	Sort ↓	Sort ↓	
326	690	704	326
453	751	608	435
608	453	326	453
835	704	835	608
751	835	435	690
435	435	751	704
704	326	453	751
690	608	690	835

Sort ↓	Sort ↓	Sort ↓	
cat	him	ham	bat
him	ham	cat	cat
ham	cat	bat	ham
bat	bat	him	him

Figure 6.1: Example of radix sort. We repeat counting sort using every digit as sorting key, from least to most significant bit.

```
Radix-Sort(A, B, k, d)
1  for i = 1 to d
2      sort A using Counting-Sort(A, B, k) with ith bit as primary key
```

Clearly, we can sort an array of n strings of fixed length d and alphabet size σ in $O(d(\sigma + n))$ time. The correctness is also obvious: if two strings differ on the first character, counting sort using a primary key will put them in the correct relative order; on the other hand, if two strings agree on the first character, they stay together in proper relative order due to stability of counting sort.

6.1.2 Sorting Strings of Variable Length

We can easily modify radix sort to work with strings with variable-length strings by adding left paddings to the string. However, this can be inefficient because it uses $\Omega(nm)$ time where $m = \max\{|A[i]| \mid i \in \{1, \dots, n\}\}$ is the length of the longest string even if the length of distinct prefix is much shorter. Instead of sorting on the least significant bit/letter, we can follow a similar paradigm but sort using the most significant bit/letter so that we don't need to add padding to the string and the algorithm will simply ignore a string if it reaches the end of that string. The algorithm is called ***MSD radix sort***.

MSD radix sort can be implemented in a divide-and-conquer fashion. For each position, the MSD radix sort partitions the list based on the current letter and performs radix sort within each partition. We repeat this recursively for each position as necessary, until each string in the original array is in its own partition.

```
MSD-Radix-Sort(A, p)
1  sorted = {s ∈ A | |s| = p − 1}
2  A = A \ sorted
3  B = Counting-Sort(A) using letter at position p as primary key
4  𝒜 = {A_1, ..., A_k} = partition B based on letter at position p
5  for i = 1 to k
6      A_i = MSD-Radix-Sort(A_i, p + 1)
7  return Concatenate(sorted, A_1, A_2, ..., A_k)
```

During the first call, *sorted* is empty, and A is sorted by the letter at position 1 (from the left) and partitioned based on the letter at position 1. We then recurse on each partition. In all subsequent recursive call to MSD-RADIX-SORT$(A_i, p+1)$, all elements in A_i share the same prefix of length p, and this prefix itself, if present in A_i, has the lowest lexicographic ordering within A_i. We exclude the prefix from A_i because its relative position within A_i is determined. We then call RADIX-SORT to sort A_i by its $(p+1)$th letter, or equivalently, by the prefix of length $p+1$. We continue to recursively call MSD-RADIX-SORT with increasing prefix length. Upon returning from the recursive calls, the procedure returns the concatenation of *sorted* and all the subpartitions. Note that *sorted* always goes first because it contains the proper prefix to all the remaining elements in the partition A_i. See figure below for an example.

	$p=1$	$p=2$	$p=3$	$p=4$
za	ab	ab	ab	ab
z	abc	abc	abb	abb
ab	abb	abb	abc	abc
bbc	abz	abz	abz	abz
abc	bbc	bac	bac	bac
abb	bac	bbc	bbc	bbc
bac	za	z	z	z
abz	z	za	za	za

Figure 6.2: Example run of recursive MSD radix sort. Strings that are colored red are in *sorted* and will not be further sorted or partitioned. Strings in *sorted* will be prepended to the beginning of each sorted partition because it contains the proper prefix of strings in that partition.

This recursive implementation of MSD radix sort runs in $O(N+\sigma m)$ time where $N = \sum_{s\in A}|s|$, σ is the size of the alphabet, and m is the length of the longest string in A. An issue with this recursive implementation is the large overhead associated with having a lot of small partitions, and this can cause bad memory locality. We now present an iterative implementation that runs in $O(N+\sigma)$ time with a smaller memory overhead. The idea of this iterative approach is to use the block pointer array introduced when we discussed counting sort as well as the notion of *partition refinement*.

Let m be the longest string in the input array A. Let A^p be the set defined as $\{A[k][1\ldots p] \mid k \in \{1,\ldots,n\}\}$ for $p = 1,\ldots,m$. We define $A[k][1\ldots p]$ to be $A[k]$ if $p > |A[k]|$. Every prefix $P \in A^p$ maps to a contiguous interval in the sorted list A^* that contains all strings that start with P. For each prefix P, we denote the interval induced by the prefix in the sorted list I_P, and we denote the start and end index of this interval in A^* with $I_P.start$ and $I_P.end$. Further, we can partition A^* with

$$\mathcal{I}_p = \{I_P \mid P \in A^p\}.$$

Note that $I_{P\cdot a} \subseteq I_P$ for all $a \in \Sigma$. The partition induced by $p+1$ is finer than the partition induced by p, so we call $\mathcal{I}_{p+1}$ a *refinement* of $\mathcal{I}_p$. The idea of the algorithm is to transform A to A^* by iteratively assigning each string to a finer partition with increasing p until every string is assigned its own partition.

The algorithm maintains a list of L of triplets $(pos, char, idx)$ where $pos \in \{1,\ldots,m\}$, $idx \in \{1,\ldots,n\}$, and $char = A[idx][pos]$. We sort L using pos as primary key and $char$ as secondary key. Partition the sorted L into $\{L_1,\ldots,L_m\}$ based on pos. By the way we sorted L, each L_i is futher partitioned by the secondary key $char$. Furthermore, the algorithm keeps four arrays Q, S, T, and B.

- Q stores the current position inside an interval: $Q[I_P.start] = Q[I_{P\cdot a}.start]$
- S stores the current size of each interval: $S[I_P.start] = |I_P|$
- T stores the current character corresponding to each interval: $C[I_P.start] = a$
- B stores the offset of each interval in the final sorted order.

We now present a pseudocode for this iterative implementation.

```
MSD-Radix-Sort(A, n, m)
 1  L = []
 2  for i = 1 to n
 3      p = 1 to m
 4          if |A[i]| ≤ p
 5              L.Append((pos = p, char = A[i][p], idx = i))
 6  sort L using Counting-Sort with pos as primary key and idx as seconday key
 7  𝓛 = {L_1, ..., L_m} = Partition(L, key = pos)
 8  for k = 1 to |L_1|
 9      B[k] = |{h : S[h][1] < S[k][1]}| + 1
10  for p = 2 to m
11      for k = 1 to |L_p|
12          T = S[L_p[k].idx][1 ... p − 1]
13          I_T.start = B[L_p[k].idx]
14          Q[I_T.start] = 0
15      for k = 1 to |L_p|
16          T = S[L_p[k].idx][1 ... p − 1]
17          if Q[I_T.start] == 0
18              Q[I_T.start] = I_T.start
19              C[I_T.start] = L_p[k].char
20      for k = 1 to |L_p|
21          T = S[L_p[k].idx][1 ... p]
22          if C[I_T.start] == L_p[k].char
23              B[L_p[k].idx] = Q[I_T.start]
24              S[I_T.start] = S[I_T.start] + 1
25          else
26              Q[I_T.start] = Q[I_T.start] + S[I_T.start]
27              S[I_T.start] = 0
28              C[I_T.start] = L_p[k].char
29              B[L_p[k].idx] = Q[I_T.start]
30  for i = 1 to n
31      A*[B[i]] = A[i]
32  return A*
```

For correctness, we note that at the beginning of each iteration of the loop on Line 10, $B[k]$ contains the starting index of the interval $I_{S[k][1...p-1]} \in \mathcal{I}_{p-1}$ in S^*. Every interval of $\mathcal{I}_p$ is completely contained within every interval of $\mathcal{I}_{p-1}$. Existence of interval $I_T \in \mathcal{I}_{p-1}$ implies the existence of an interval $I_{T \cdot a} \in \mathcal{I}_p$ where $I_{T \cdot a}.start = I_T.start$ and $a \in \Sigma$ is the lexicographically smallest character at position p that is preceeded by T. Therefore, it is correct to set $I_T.start$ to $B[L_p[k].idx]$ on Line 13. We initially set all Q values to 0 and we only update them for the starting positions. We also only update C only for each starting position once. It is set to the lexicographically smallest character following any occurrence of T as a prefix. This ensures us to detect any possible refinements within each partition I_T where $C[I_T.start] \neq L_p[k].char$. When that happens, we create a new partition and update the offset of the new partition in B. The algorithm will terminate when every distinct string is assigned its own partition. The offset at which each refined interval is given by B. Finally, we can construct a sorted list by assigning elements to its corresponding interval and at the given offset. The pseudocode as given above does not handle the case where there are duplicate elements (Line 31 will attempt to assign them to the same position), but it can be easily modified to break ties arbitrarily in case of duplicates.

The algorithm runs in $O(N + \sigma)$ time because there is only one round of counting sort when we sort the triplets in L. As for space complexity, L, Q, S, C, and B all contributes to the space complexity, but since

sorting is done in place, there is no additional overhead other than creating the result array. The space complexity is also $O(N + \sigma)$.

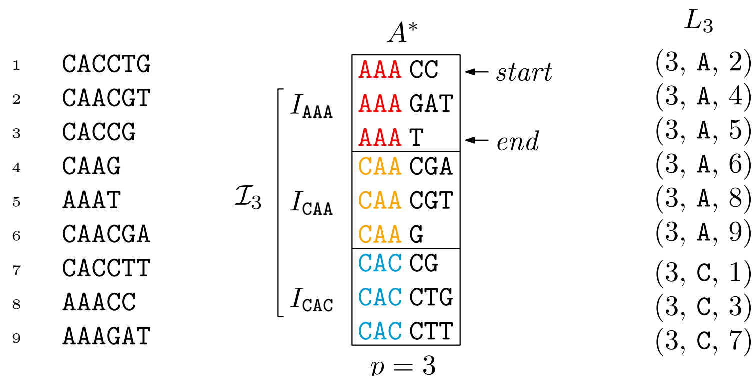


Figure 6.3: From left to right: (a) The input array A; (b) A partition $\mathcal{I}_3$ of A^*; (c) The list of triplets L_3; L is sorted using counting sort and partitioned into $L_1, \ldots, L_m$.

6.1.3 Suffix Sorting Using Prefix Doubling

In this subsection, we will introduce the first suffix array construction algorithm that makes use of the linear time sorting techniques introduced in the last two subsections. It is a conceptually simple algorithm that achieves $O(n \log n)$ time for a string of length n.

Let T_i^ℓ be the substring $T[i \ldots \min\{i+\ell, n\}]$, that is, T_i^ℓ is the substring starting at i and of length ℓ except when the $i + \ell > n$, in which case, it is just the suffix starting at i. T_i^ℓ is the *prefix of the suffix* T_i of length ℓ, or the suffix T_i itself if $|T_i| < \ell$. The idea of the prefix doubling algorithm is to sort the sets $T^\ell = \{T_i^\ell \mid i \in \{1, \ldots, n\}\}$ for every increasing value of ℓ. The set T^ℓ includes a prefix of length ℓ of every suffix T_i of T. To achieve $O(n \log n)$ runtime, we first sort T^1, which is equivalent to sorting each individual characters. As we have seen earlier, this can be done in $O(n)$ time. Then, for $\ell = 1, 2, 4, 8, \ldots$, use the sorted set T^ℓ to sort the set $T^{2\ell}$ in $O(n)$ time. After $O(\log n)$ iterations, $\ell > n$ and $T^\ell = T$ and at this point we would have sorted all suffixes.

To see how to use the sorting of T^ℓ to sort $T^{2\ell}$, we introduce the notion of *order preserving names*. For $i \in \{1, \ldots, n\}$, let N_i^ℓ be an integer in the range $\{1, \ldots, n\}$ such that for all $i, j \in \{1, \ldots, n\}$,

$$N_i^\ell \leq N_j^\ell \text{ iff } T_i^\ell \leq T_j^\ell.$$

Then, for $\ell > n$, $N^\ell[i] = \mathrm{SA}^{-1}[i]$ where SA^{-1} is the inverse suffix array. A simple choice of N is the rank

$$N_i^\ell = |\{j \in \{1, \ldots, n\} \mid T_j^\ell < T_i^\ell\}|.$$

For the pseudocode below, note that we begin by sorting L, which is just an array of all characters in the string T. We then go through the sorted L and compute the rank of each character and store them in R. At the end of the first iteration of the outer while loop, $R[i]$ contains the relative rank of T_i^1. Continuing inductively, in subsequent iterations, we use a pair of ranks $(R[i], R[i+\ell])$ to represent T_i^ℓ. Sorting the pairs is equivalent to sorting T_i^ℓ because rank is an order preserving name for the prefixes, and at the end of each iteration of the while loop, $R[i]$ contains the starting position of the interval of string $T[i \ldots i + 2^p]$ in the suffix array. Each iteration runs in $O(n)$ time and there are $O(\log n)$ iterations so the algorithm runs in $O(n \log n)$ time.

```
PREFIX-DOUBLING-SORT(T = t_1 t_2 ··· t_n)
 1  R = [0, ...] (array of length 2n)
 2  L = (p = t_1, s = t_1, v = 1), (t_2, t_2, 2), ..., (t_n, t_n, n)
 3  p = 0
 4  while 2^p < n
 5      L' = COUNTING-SORT(L, primary = p, secondary = s)
 6      for k = 1 to |L|
 7          if L'[k].p == L'[k-1].p and L'[k].s == L'[k-1].s
 8              R[L'[k].v] = R[L'[k-1].v]
 9          else
10              R[L'[k].v] = R[L'[k-1].v] + 1
11      L = (R[1], R[1 + 2^p], 1), (R[2], R[2 + 2^p]), ..., (R[n], R[n + 2^p])
12      p = p + 1
13  for i = 1 to n
14      SA_T[R[k]] = k
```

R^1	L	T^1	R^2	L	T^2	R^4	L	T^4	R^8	L	T^8
4	b	b	4	(4,1)	ba	4	(4,5)	bana	4	(4,5)	banana$
1	a	a	2	(1,5)	an	3	(2,5)	anan	3	(3,1)	anana$
5	n	n	5	(5,1)	na	6	(5,5)	nana	6	(6,0)	nana$
1	a	a	2	(1,5)	an	2	(2,1)	ana$	2	(2,0)	ana$
5	n	n	5	(5,1)	na	5	(5,0)	na$	5	(5,0)	na$
1	a	a	1	(1,0)	a$	1	(1,0)	a$	1	(1,0)	a$
0	$	$	0	(0,0)	$	0	(0,0)	$	0	(0,0)	$
0			0			0			0		
0			0			0			0		
0			0			0			0		
0			0			0			0		
0			0			0			0		
0			0			0			0		
0			0			0			0		

Figure 6.4: Example run of the prefix doubling algorithm. Initially, L contains individual letters from T^1. In the subsequent iterations, L is updated to a list of triplets containing the relative rank $(R[i], R[i+\ell])$ where p is the rank of T_i^ℓ and $R[i+\ell]$ is the rank of $T_{i+\ell}^\ell$ (the prefix of the suffix that is ℓ letters apart from i). The rank tuple is used as a surrogate of the prefix of length 2ℓ of the suffix T_i. We iteratively sort L and update R. In the end, we have an invserse suffix array, which can be easily turned into a suffix array. R is zero filled to avoid out-of-range errors and ensure that suffixes ending with the terminator symbol $ is sorted properly.

6.2 Naive Construction From Suffix Tree

Given a suffix tree, a suffix array can be trivially constructed through a lexicographic depth first search (that is, at each internal node, decide which path to recurse on based on the alphabetical order of the first character of each path label).

As we have seen, a suffix tree can be constructed in $O(n)$ time. It follows that a suffix array can also be constructed in $O(n)$ time using this method. However, a major issue with this approach is that we must build an intermediate suffix tree, which may require significantly more memory, and this defeats the purpose of having a suffix array in the first place, which is to have a more compact representation of the suffixes of a string.

6.3 A Divide-and-Conquer Approach

The next natural approach one might consider when presented with a problem like constructing a suffix array is divide and conquer. We are all familiar with merge sort, which runs in $O(n \log n)$ time. Constructing a suffix array similarly involves sorting the suffixes.

Let T be the text for which we want to construct a suffix array. Consider the following divide-and-conquer approach:

1. Divide the suffix positions into $A \subset [0 \ldots n]$ and $\overline{A} = [0 \ldots n] \setminus A$
2. Construct a suffix array for T_A (suffixes that start at positions in A) recursively
3. Construct a suffix array for $T_{\overline{A}}$ based on the suffix array for T_A
4. Merge the two suffix arrays

The most straightforward way to divide is to divide the positions by parity (even/odd). This gives an algorithm whose runtime is given by the recurrence $T(n) = T(\lceil n/2 \rceil) + T_{merge}(n)$.

Everything seems good so far, but there are two important problems: (1) how do we construct the second suffix array non-recursively, and (2) how to merge in linear time? For a long time, finding a way to merge two suffix arrays in linear time remained an open question. The most obvious way to merge takes $O(n^2)$ time. Researchers came up with clever tricks but still only got an $O(n \log n)$ time bound. Because of that, until the early 2000s, the best known algorithm for constructing a suffix array only ran in $O(n \log n)$ time.

In 2003, Juha Kärkkäinen and Peter Sanders published their seminal paper (along with some other researchers who independently published similar results around the same time), which proposed one of the first linear time algorithms for constructing a suffix array. It uses the same divide-and-conquer framework, with a little twist.

6.4 Kärkkäinen-Sanders Algorithm

Kärkkäinen and Sanders' algorithm uses the same divide-and-conquer approach, but instead of dividing the positions into even and odd positions like previous researchers have done, they divided the positions i's into those with $i \bmod 3 \neq 0$ and $i \bmod 3 = 0$. This, along with a neat trick during merging, is enough to give us an $O(n)$ time algorithm for construct a suffix array.

Let's first recall the general framework for constructing suffix array using divide-and-conquer

```
KÄRKKÄINEN-SANDERS
1  construct suffix array for suffixes starting at positions i mod 3 ≠ 0
   recursively // T(2/3n)
2  construct suffix array for suffixes starting at positions i mod 3 = 0
   using results from step 1 // O(n)
3  merge the two suffix arrays // O(n)
```

We will see how to perform each step within the given time. We call the suffixes starting at positions $i \bmod 3 \neq 0$ the ***sample suffixes***, and the suffixes starting at positions $i \bmod 3 = 0$ the ***non-sample suffixes***.

6.4.1 Sorting The Sample Suffixes, Recursively

Given a string T of length n, we define $T[j]$ for all $j > n$ to be equal to `$`, so $T[j] =$ `$` for all positions beyond n. This is just to avoid having to deal with the edge cases.

Let t_0 be the set of **triples** (not suffixes) starting at position $i \bmod 3 = 0$, so $t_0 = \{T[i \ldots i+2] \mid i \bmod 3 = 0,\ i \leq n\}$. Similarly, let t_1 and t_2 be the sets of triples starting at position $i \bmod 3 = 1$ and 2, respectively. For example, suppose $T =$ `dadbcddadbcd$` with the delimiter $ in the end, we will have

$$t_1 = \{\texttt{dad, bcd, dad, bcd, \$\$\$}\}$$

and

$$t_2 = \{\texttt{adb, cdd, adb, cd\$}\}$$

To sort the suffixes starting at positions $i \bmod 3 \neq 0$, we first sort the triples in $t_1 \cup t_2$. This can be done in $\Theta(n)$ time using **radix sort**. For $x \in t_1 \cup t_2$, we define the **rank** $rank(x)$ to be the order of the triple x in the sorted list of $t_1 \cup t_2$. If two triples have the same order in the sorted list, they will have the same rank. Further, for a set of triples X, we define $Rank(X)$ to be the list of ranks for each triple in the sorted order. That is, the ith element of $Rank(X)$ will be $rank(X[i])$. Using the same example as above where $T =$ `dadbcddadbcd$`, we have

pos	=	13	2	8	4	10	11	5	1	7
$Sorted(t_{1,2})$	=	`$$$`	`adb`	`adb`	`bcd`	`bcd`	`cd$`	`cdd`	`dad`	`dad`
$Rank(t_{1,2})$	=	1	2	2	3	3	4	5	6	6

We also record pos, the starting position of each of the triple in the original string.

Now, let us go back to the original set of triples, t_1 and t_2. We create a new string t' equals to $t_1 \cdot t_2$ (t_1 concatenated with t_2) with each triple **mapped to its rank**.

pos	=	1	4	7	10	13	2	5	8	11
$t_1 \cdot t_2$	=	`dad`	`bcd`	`dad`	`bcd`	`$$$`	`adb`	`cdd`	`adb`	`cd$`
t'	=	6	3	6	3	1	2	5	2	4

Next, we **recursively find the suffix array** for t'. We claim that the suffix array for t' specifies the suffix array for S restricted to the suffixes starting at positions $i \bmod 3 \neq 0$.

Wait! But Why?

Claim. Suffix array for t' specifies the suffix array for S restricted to the suffixes starting at positions $i \bmod 3 \neq 0$

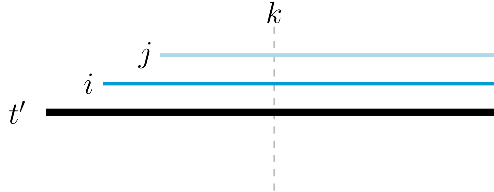
To see why this is true, we first prove this lemma.

Lemma 6.2. *Let t'_i and t'_j be two suffixes of t' starting at position i and j, respectively. If $s'_i \prec_{lex} s'_j$ (if s'_i is lexicographically less than s'_j), then the suffix of the original string T starting at position $pos[i]$ is also lexicographically smaller than the suffix of T starting at position $pos[j]$.*

Proof. Recall that t' can be divided into two parts. The first half contains the ranks of triples whose first character starts at position $i \bmod 3 = 1$. The second half contains the ranks of triples whose first character

starts at position $i \bmod 3 = 2$. To prove the lemma, we consider the following cases regarding the positions of i and j in t'.

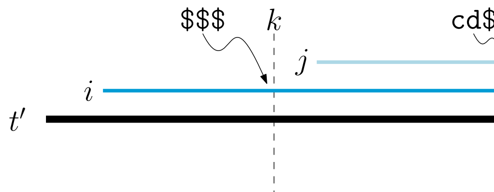
Case 1: Both i and j are in the first half. Then, $pos[i] = pos[j] = 1 \mod 3$. We first observe that the comparison of the two suffixes starting at i and j **will not go beyond the boundary** between the first and the second half. More formally, let k be the position such that $pos[k] \bmod 3 = 1$ but $pos[k+1] \bmod 3 = 2$.



Then, during the comparison of the two suffixes of t' starting at i and j, at most k characters are compared. This is because the triple at position $pos[k]$ will contain the unique **null terminator** that is lexicographically smaller than any character in the alphabet, thus giving the triple at $pos[k]$ a **unique rank**.

Moreover, each symbol in t' represents the **rank of a triple** starting at that position in T. By assumption, the suffix of t' starting at i is lexicographically smaller than the suffix of t' starting at j. Then, there must exists some c such that $t'[i+c] < t'[j+c]$. This implies that the triple starting at position $pos[j+c]$ that is lexicographically larger than the triple starting at position $pos[i+c]$ because t' represents the ranks of the triples. The presence of this **lexicographically larger triple** makes the suffix of the original string at $pos[i]$ lexicographically smaller than the suffix at $pos[j]$.

Case 2: Both i and j are in the second half. This case follows from a similar argument as Case 1.



Case 3: i is in the first half and j is in the second half. As in the first two cases, the comparison will **never cross the boundary**. This, again, is due to the distinct null terminator symbol. In particular, the triple starting at $pos[k]$ (recall that k is the boundary between the two parts) contains at least one more/fewer null terminator symbol compared to the triple at $pos[|t'|]$. Hence, the triple at $pos[k]$ will have a **unique rank**, which helps us **break the tie** after the comparison at position k. We can always determine the lexicographic order of the two suffixes without crossing the boundary.

Now, going back to t', if we have the suffix of t' starting at i being lexicographically smaller than the suffix of t' starting at j, we know that there must be some c such that $t'[i+c] < t'[j+c]$, which implies the rank of some triple at $pos[i+c]$ is lexicographically smaller than that at $pos[j+c]$. Since we never cross the boundary when comparing the suffixes of t' starting at i and j, we are always comparing the rank of a contiguous and non-overlapping substring of T starting at $pos[i]$ with the rank of some other contiguous and non-overlapping substring of T at $pos[j]$ without ever going backward in the comparison (because we don't cross the boundary). Then, it follows that the lexicographic ordering of the ranks in t' implies the ordering in T.

Case 4: i is in the second half and j is in the first half. This follows from a similar argument as Case 3.

In all cases, the implication holds, so the lemma holds. □

One important takeaway from the proof of this lemma is that the null terminator symbol $ is a tie-breaker, giving us unique ranks for triples at the end of the first and second half so that we never cross the boundary between the two halves. This unique rank, in turn, allows us to use the string of ranks to implicitly sort the suffixes in the original string T.

Using our previous example with

i		1	2	3	4	5	6	7	8	9
pos	=	1	4	7	10	13	2	5	8	11
$t_1 \cdot t_2$	=	dad	bcd	dad	bcd	$$$	adb	cdd	adb	cd$
t'	=	6	3	6	3	1	2	5	2	4

we have

SA for t'	=	5	8	6	4	2	9	7	3	1
SA_{12} for T	=	13	8	2	10	4	11	5	7	1

The ith entry in the SA for T is $SA_{12}(T)[i] = pos[SA(T')[i]]$. Here, $SA_{12}(T)$ refers to the suffix array for the original string T but only considering the suffixes at positions 1 or 2 mod 3.

6.4.2 Sorting the Non-Sample Suffixes

There is an easy way to sort the non-sample suffixes. Those are the suffixes that start at positions $i \bmod 3 = 0$. Again, we begin by considering the triples starting at these positions. Each of such positions is followed by two positions with $i \bmod 3 \neq 0$. Ordering of the triples starting at one and two positions after those with $i \bmod 3 = 0$ have already been determined recursively as discussed in the previous subsection. We can then use the information we know about the sample suffixes to sort the non-sample suffixes in linear time, non-recursively.

To this end, we construct a list t'' that contains all characters at positions $i \bmod 3 = 0$ with each character followed by the rank of the suffixes starting at position immediately after i (which can be determined from $SA[t']$ that we have constructed in the previous step).

Slightly more formally, the ith element of t'' will be

$$t''[i] = T[3i] \cdot SA_{12}(T)[3i+1]$$

In our example, $T =$ `dadbcddadbcd$` and

i		1	2	3	4	5	6	7	8	9
SA for t'	=	5	8	6	4	2	9	7	3	1
SA_{12} for T	=	13	8	2	10	4	11	5	7	1

so

triple	=	dbc	dda	dbc	d$$
t''	=	d5	d8	d4	d1
pos	=	3	6	9	12

We sort t'' using radix sort in $\Theta(n)$ time. For our example, this gives us

t''	=	d1	d4	d5	d8
pos	=	12	9	3	6

The corresponding positions in the sorted t'' is the suffix array for the suffixes starting at $i \bmod 3 = 0$, so we have $SA_3(T)$ as well.

The correctness of this step is trivial from the correctness of radix sort and the fact that the entries in SA_{12} are unique (so there won't be tie).

6.4.3 Merging the Two Suffix Arrays

The final punchline. We will merge $SA_{12}(T)$ and $SA_3(T)$ into one suffix array in linear time.

Recall that in the $O(n^2)$ algorithm for constructing a suffix array, the merging is done in $O(n^2)$ time using the naive method. The naive method keeps two pointers to each of the restricted suffix arrays SA_{12} and SA_3. It then compare the suffixes explicitly in worst-case $O(n)$ time. We do this for all the $O(n)$ pairs of positions, giving us an $O(n^2)$ time algorithm.

1 $i, j = 1, 1$
2 **while** $i \le |SA_{12}(T)|$ and $j \le |SA_3(T)|$
3 compare suffixes $T[SA_{12}(T)[i] \ldots]$ and $T[SA_{12}(T)[j] \ldots]$
4 update i, j accordingly

However, with the suffixes arrays SA_{12} and SA_3, we can actually do the comparison in constant time. For each arbitrary pair of positions i, j, we only need at most 3 explicit character comparisons before we reach a position i', j' such that $i' = j' \mod 3$, at which point the lexicographic order of the two suffixes can be determined using an $O(1)$ **lookup** in the appropriate restricted suffix array.

For a more detailed procedure for merging, consider the following cases:

Case 1: Compare two suffixes starting at i and j where $i \bmod 3 = 2$ and $j \bmod 3 = 0$. If the encounter a character such that $T[i] \neq T[j]$, then we are done. Otherwise, continue comparing $T[i]$ with $T[j]$ and updating i and j. After at most 2 comparisons, $i \bmod 3 = 1$ and $j \bmod 3 = 2$. We can determine the ordering of the two suffixes by comparing the locations of i and j in SA_{12} in $O(1)$ time.

Case 2: Compare two suffixes starting at i and j where $i \bmod 3 = 1$ and $j \bmod 3 = 0$. If we encounter a character such that $T[i] \neq T[j]$, then we are done. Otherwise, the problem reduces to Case 1, and we can determine the lexicographic ordering of the suffixes starting at i and j with at most 3 explicit comparisons.

Case 3: $i = j \mod 3$. This case is trivial through a constant-time lookup in SA_{12} if $i \bmod 3 = j \bmod 3 \neq 0$ or in SA_3 if $i \bmod 3 = j \bmod 3 = 0$.

In all three cases, we can determine the lexicographic ordering of the two suffixes within $O(1)$ comparisons. We repeat this for all $|T|$ positions, giving us an $O(n)$ time algorithm for merging.

6.4.4 Wrapping It Up

And here we have it, the linear-time algorithm for constructing a suffix array. At first glance, it appears to be quite a sophisticated algorithm, but the ideas behind it are actually quite fundamental. It based on the same divide-and-conquer approach that previous $O(n^2)$ and $O(n \log n)$ time algorithms have used, but with a few ingenious improvements that allow us to do the merging in $O(n)$ time. Note that the linear-time merging is not possible if we divide the suffixes up into positions 0 or 1 mod 2 since we are not guaranteed to be at a position $i = j \mod 2$ after just a constant number of comparisons.

As we mentioned at the beginning, this algorithm is due to Kärkkäinen and Sanders. It is often referred to as the ***Kärkkäinen-Sanders (KS) algorithm*** or the ***DC3 algorithm*** since it is a divide-and-conquer algorithm that divides the positions based on their values modulo 3.

To wrap this section up, let us prove that the KS algorithm indeed runs in linear time.

Theorem 6.3. *The suffix array for text T of length n can be computed in time $O(n)$.*

Proof. We use the Kärkkäinen-Sanders' algorithm. The correctness of the algorithm is argued as we introduce the algorithm. Now, we consider the runtime of the algorithm.

Sorting of the triples takes $O(n)$ time using radix sort, and so does the computation of the SA for the non-sample suffixes at position $i \bmod 3 = 0$. At each level of the recursion, the suffix array that we recursively construct is of size $\lceil 2/3n \rceil$. Finally, merging takes $O(n)$ time. Hence, the overall runtime is given by the recurrence

$$\begin{aligned} T(n) &= T(\lceil 2/3n \rceil) + 3O(n) \\ &= T(\lceil 2/3n \rceil) + O(n) \\ &\leq n \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i \\ &\in O(n) \end{aligned}$$

The same recurrence can also be solved using the Master's theorem. □

Bibliography

Counting sort is a folklore linear time sorting algorithm, but the presentation is a combination of materials from CLRS and the course String Processing Algorithms at the University of Helsinki [6]. Kärkkäinen-Sanders's algorithm was introduced in [12]. Another similar algorithm was discovered by Kim et al. around the same time [14]. The presentation of Kärkkäinen-Sanders's algorithm is based on the book Genome-Scale Algorithm Design [18].

Chapter 7

Burrows-Wheeler Transform and FM Index

7.1 Burrows-Wheeler Transform

Burrows-Wheeler transform is an algorithm initially designed by Michael Burrows and David Wheeler for document compression. Using Burrows-Wheeler transform, we can design space-efficient and versatile index data structure for counting and finding patterns in large strings. In this chapter, we will formally define Burrows-Wheeler transform, both in its original cyclic rotation definition and an alternative definition based on suffix array. After that, we will discuss the construction of index data structures using Burrows-Wheeler transform such as FM index and the powerful bidirectional BWT index.

7.1.1 Cyclic Rotation

The Burrows-Wheeler transform is defined in terms of cyclic shifts of a string $T = t_1 \cdots t_n$, Assume that we build the n cyclic shifts of T in a matrix

$$\begin{bmatrix} t_1 & t_2 & \cdots & t_{n-1} & t_n \\ t_2 & t_3 & \cdots & t_n & t_1 \\ t_3 & t_4 & \cdots & t_1 & t_2 \\ \vdots & & \ddots & & \vdots \\ t_n & t_1 & \cdots & t_{n-2} & t_{n-1} \end{bmatrix}$$

We can then sort the rows of the matrix in lexicographic order to obtain the Burrows-Wheeler matrix. The Burrows-Wheeler transform BWT(T) is defined to be the last column of the BTW matrix. To ensure every row is unique in the BWT matrix, we add a terminator symbol \$ to the end of the T. We name the first column F and the last column L. F contains characters of T in sorted order. L contains the BWT of T.

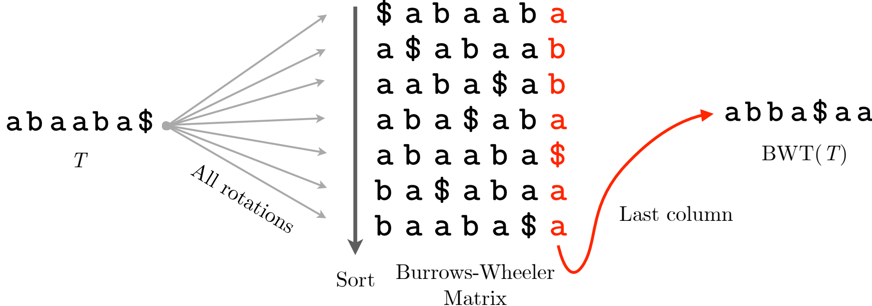


Figure 7.1: Burrows-Wheeler Transform as cyclic rotation.

BWT is useful for compression because the characters of a BWT are sorted by their right-context. This provides additional structure to the BWT, making is useful for run-length encoding compression. The procedure for compressing a string using BWT is as follows:

1. Compute BWT(T)

2. Partition BWT(T) by k-context (rotational context)
3. Compute H_0 encoding on partitions

This procedure can be repeated by increasing k to create higher-order empirical entropy encoder.

7.1.2 BWT and Suffix Array

Given a string T, consider its BWT matrix and the suffixes corresponding to its suffix array. If we look at them side-by-side closely, we will notice that the BWT matrix bears a resemblance to the suffix array. In particular, the columns of the sorted rotations share the same sorted order as the suffixes. The ith suffix in sorted order is a prefix of the ith rotation in sorted order.

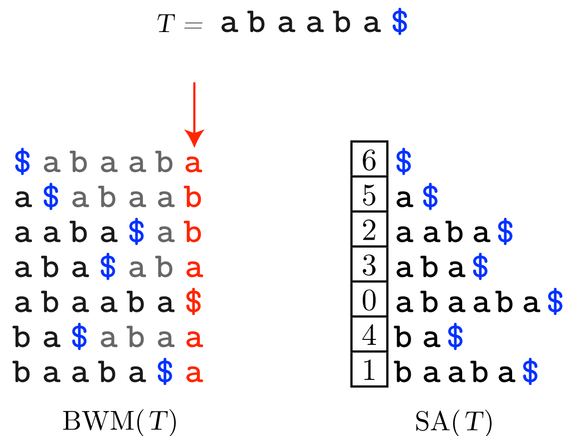


Figure 7.2: Relationship between the BWT (highlighted red) and the suffix array of the string T. The ith character of BWT(T) is the character before the starting position of the ith suffix in the sorted order.

Thus, we have this following definition of BWT of T:

$$\mathrm{BWT}(T)[i] = \begin{cases} T[\mathrm{SA}_T[i] - 1] & \text{if } \mathrm{SA}_T[i] > 0 \\ \$ & \text{if } \mathrm{SA}_T[i] = 0. \end{cases}$$

This also gives us a way to efficiently compute the BWT using the linear-time SA construction algorithm without explicitly constructing the BWT matrix.

7.1.3 Inverse BWT and LF-Mapping

A transformation like BWT would be rather useless if we cannot invert it as we would often like to recover the original string from a compressed one, or extract useful information from an index. Luckily, the BWT matrix has a property known as ***LF mapping*** that will help us derive a way to invert the BWT. Recall that we call the first column of the BWT matrix F and the last column L, so an LF mapping is just a mapping from L to F.

Consider the text T. For clearity, we assign each character t in T a rank that is equal to the number t occurred previously in T.

$$T = \mathtt{a}_0\, \mathtt{b}_0\, \mathtt{a}_1\, \mathtt{a}_2\, \mathtt{b}_1\, \mathtt{a}_3\, \$$$

$$
\begin{array}{ccccccc}
F &&&&&& L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}
\qquad
\begin{array}{ccccccc}
F &&&&&& L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}
$$

Figure 7.3: The BWT matrix for $T = a_0\, b_0\, a_1\, a_2\, b_1\, a_3\, \$$ with ranks shown as subscripts. Note that in the first column F and last column L, the relative order of the rank for each character is preserved.

We now look at the BWT matrix with ranks. As we can see in the figure, the first column F is all characters of T sorted in lexicographic order. Because of this, same character appears in the same chunk, that is, F is partitioned based on characters. Notably, the relative order of the ranks for each character is preserved as we go from F to L. In Figure 7.3, the relative order of the a's in F is a_3, a_1, a_2, a_0, and the realtive order is the same in L. We call this the LF mapping property:

> The ith occurrence of a character c in L and the ith occurrence of c in F correspond to the same occurrence of c in T.

Intuitively, the LF mapping property holds because for each c, the occurrences of c in F are sorted by its right-context; similarly, occurrences of c in L is sorted by their right rotational context. Moreover, we remark that the "rank" that we assign to each character is completely arbitrary. We can assign the characters arbitrary ranks and the LF mapping property would still hold. Thus, instead of defining rank to be the number of occurrences of a character preceeding the current occurrence, we can define a "*pseudo-rank*" so that the ranks are in *ascending order* as we traverse down the F and L columns.

$$
\begin{array}{ccccccc}
F &&&&&& L \\
\$ & a_3 & b_1 & a_1 & a_2 & b_0 & a_0 \\
a_0 & \$ & a_3 & b_1 & a_1 & a_2 & b_0 \\
a_1 & a_2 & b_0 & a_3 & \$ & a_3 & b_1 \\
a_2 & b_0 & a_0 & \$ & a_3 & b_1 & a_1 \\
a_3 & b_1 & a_1 & a_2 & b_0 & a_0 & \$ \\
b_0 & a_0 & \$ & a_3 & b_1 & a_1 & a_2 \\
b_1 & a_1 & a_2 & b_0 & a_0 & \$ & a_3
\end{array}
\quad \text{Ascending pseudorank}
$$

Figure 7.4: The BWT marix with ranks replaced by a pseudorank.

Now, F has a very predictable structure: a \$, followed by blocks of characters in lexicographic order where the characters in each block has ascending pseudorank. Following this, we can design an algorithm that computes F given L. The algorithm simply runs *radix sort* on L using the character as primary key and the rank as secondary key. The resulting sorted list is F.

Why is it useful to have F as well as L? It is useful to reconstruct the original text T because the character in each row of L appears immediately before the corresponding character in F in the text T. This property follows from the fact that the column F is the right 1-context of L. With this insight, we can design an algorithm that recovers the original text T from F and L.

INVERSE-BWT(F, L)

```
1  i = 0
2  T = $
3  while L[i] ≠ $
4      c = L[i]
5      T = c · T
6      i = F[c][rank(i)]
7  return T
```

The function *rank* computes the rank of each character. $rank(i)$ is the (pseudo)rank of the ith character in L. It can be easily computed by keeping a running counter of occurrence of each character in L. As we traverse through L and F, we concatenate each character that we encounter in L to the front of our running result. After this, we jump to the next position in F, which is given by the character c and the offset given by the current rank of c. F is partitioned based on the character, that is, $F = \{F_x \mid x \in F\}$ where F_x is a block of letter $x \in \Sigma$ ordered in ascending rank, so $F[c][rank(i)]$ returns a pointer to the character c with rank given by $rank(i)$.

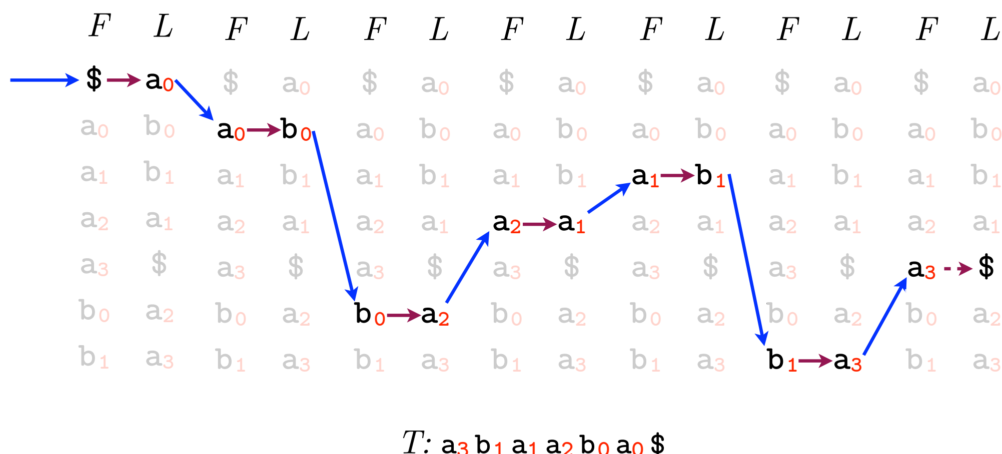


Figure 7.5: A visualization of the algorithm INVERSE-BWT as it traverses between L and F to recover the original text T.

7.2 FM Index

FM index is a compressive self-index. It compresses the data and indexes it at the same time. As we have seen earlier in the chapter, we can apply empirical entropy encoders like Huffman to compress the BWT. In this section, we will focus on how to construct a space-efficient index data structure using the Burrows-Wheeler transform. Recall when we discussed LF-mapping, we made use of the concept of a rank. We did not provide an efficient implementation of the function *rank* which computes the rank of a character at a given position in the BWT. We suggested an implementation that simply counts the number of occurrence of a given letter by traversing L. However, this is inefficient in practice. We need a quicker way to compute rank.

7.2.1 Rank-Select-Access Query

This subsection is devoted to succinct data structures to answer the following types of queries:

1. RANK(A, c, i): Compute the rank of character c at position i of the vector A. That is, the number of occurrences of c up to position i in the vector A.

$$\text{RANK}(A, c, i) = |\{i' \mid 1 \le i' \le i,\ A[i'] = c\}|$$

2. SELECT(A, c, i): Return a pointer to the ith occurrence of c in A.
3. ACCESS(A, i): Return a pointer to the character at position i in A.

We will first look at a data structure for solving RANK and SELECT queries on bit vectors. In the following subsection, we will introduce wavelet tree, which combines the idea from bit vector rank-select and the idea of recursive decomposition of the alphabet.

Jacobson's Rank

Consider a bit vector A. We divide A into chunks each of $\lg^2 n$ bits long. There are $\frac{n}{\lg^2 n}$ such chunks. For each chunk, we store the cumulative rank up to that chunk (the number of 1-bits from the beginning of the bit vector to the start of the chunk). The space required to store all the cumulative rank is $O(\lg n \cdot \frac{n}{\lg^2 n}) \in O(\frac{n}{\lg n}) \in o(n)$.

We further divide each chunk into $2 \lg n$ subchunks, each of $\frac{1}{2} \lg n$ bits. For each subchunk, we store a relative cumulative rank that is the number of 1's from the beginning of the chunk to the start of the subchunk. This takes in total $O(n \cdot \frac{\lg \lg n}{\lg n}) \in o(n)$ bits.

Finally, we store a lookup table that allows us to answer query within the subchunk. The lookup table stores all possible bit vectors of the length of the subchunk, as well as the rank at each possible position. For example, if the subchunk size is 3 bits. Then, we construct the lookup table as follows.

bitvector \ position	0	1	2
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	0	0
101	1	1	2
110	1	2	2
111	1	2	3

Each subchunk represents a bit vector of length $\frac{1}{2} \lg n$. There are $2^{\frac{1}{2} \lg n}$ possible bit vectors of length $\frac{1}{2} \lg n$. For each bit vector, there are $\frac{1}{2} \lg n$ possible offsets within the bit vector. Finally, each entry in the lookup table takes $\lg(\frac{1}{2} \lg n)$ bits to store. This gives us a total size of $O(2^{1/2 \lg n} \cdot \frac{1}{2} \lg n \cdot \lg(\frac{1}{2} \lg n)) = O(\sqrt{n} \lg n \lg \lg n) \in o(n)$ bits. In total, our data structure takes $o(n)$ bits and answering the rank query can be done in $O(1)$ time by adding up the results in the three-level index.

$$\text{RANK}(A, 1, i) = \textit{first}\left[\frac{i}{\lceil \lg^2 n \rceil}\right] + \textit{second}\left[\frac{i}{\lceil \frac{1}{2} \lg n \rceil}\right] + \textit{third}[c_i][(i \bmod \lceil 1/2 \lg n \rceil) - 1]$$

where c_i represent the bit vector $A[k \cdot (i/k) + 1 \ldots (i/k + 1) - 1]$ for $k = \lceil \frac{1}{2} \lg n \rceil$.

Clark's Select

Next, we look at how to implement select using sublinear space and constant time. The algorithm will roughly follow this outline:

```
 1  locate the chunk the ith 1-bit is in
 2  if chunk is sparse
 3      look up answer in sparse offset table
 4  elseif chunk is dense
 5      look up chunk offset
 6      locate the subchunk it is in
 7      look up relative offset
 8      if subchunk is sparse
 9          look up answer in sparse subchunk table
10          return results from Line 4 + Line 6 + Line 9
11      elseif subchunk is dense
12          look up answer in dense lookup table
13          return results from Line 4 + Line 6 + Line 12
```

We first divide the bit vector into chunks so that each chunk contains $\lg^2 n$ 1-bits. Note this is different than how we divide the chunks in Jacobson's select. The chunk here may have different sizes, but they all have the same number of 1-bits in them. Since each chunk has different size, we need a lookup table to locate the start of each chunk. This takes $O(\frac{n}{\lg^2 n}\lg n) = O(\frac{n}{\lg n}) \in o(n)$ because in the worst case, every bit in the vector can be a 1-bit.

Once we arrive at a chunk, we need to handle two cases. We say a chunk is *sparse* if it has length of at least $\lg^4 n$ bits. Otherwise, we say the chunk is *dense*. Intuitively, a chunk is sparse if the fraction of 1-bits is less than square root of the total number of bits in that chunk. If the chunk is sparse, it turns out we can simply store the answer to the select query. We have chosen the cut-off for sparsity so that the chunk is sparse enough that the results of select query can fit in $o(n)$ bit of memory. There can be at most $\frac{n}{\lg^4 n}$ sparse chunks in total. It takes $\lg n$ bits to store a single answer, and there are $\lg^2 n$ 1-bits per chunk whose answers need to be stored, giving us a total of $O(\frac{n}{\lg^4 n} \cdot \lg n \lg^2 n) = O(\frac{n}{\lg n}) = o(n)$ bits to store the select query results to all bits inside sparse chunks.

In the case where the chunk is dense, we need some additional structures to support constant-time query without exceeding the space bound of $o(n)$. Recall that a chunk is dense if the length is $< \lg^4 n$. We can split a dense chunk into subchunks so that each subchunk contains $\sqrt{\lg n}$ 1-bits. Similar to the chunks, we need to store a relative offset to the start of each subchunk within a given chunk. Each relative rank takes $O(\lg \lg^4 n) = O(\lg \lg n)$ bits to store. Overall, since there can be at most $\frac{n}{\sqrt{\lg n}}$ subchunks, it takes $O(\frac{n \lg \lg n}{\sqrt{\lg n}}) \in o(n)$ bits to store all the relative ranks. We say a *subchunk* is *sparse* if its length is $\geq \frac{1}{2}\lg n$. In the case that a subchunk is sparse, we can store the realtive offset for every 1-bit in that subchunk. It takes $O(\frac{n}{\frac{1}{2}\lg n} \cdot \lg \lg n) = O(\frac{n \lg \lg n}{\sqrt{\lg n}}) = o(n)$. If the subchunk is dense, then it is short enough that we can store the answer for all possible chunks just like the lookup table in Jacbson's rank. There are $2^{\frac{1}{2}\lg n}$ possible bitvectors of length less than $\frac{1}{2}\lg n$. Each subchunk can contain up to $\sqrt{\lg n}$ 1-bits and it takes $\lg \lg n$ bits to store an answer for each one of them. This takes in total $O(\sqrt{n \lg n} \lg \lg n) \in o(n)$ bits.

Overall, each step of the select query is a constant time operation, and each data structure we keep to answer the select query can be stored in $o(n)$ bits and there are only constant many such data structures (1 chunk offset table, 1 sparse chunk lookup table, 1 subchunk relative offset table, 1 sparse subchunk lookup table, 1 dense subchunk lookup table), so the overall space usage is still in $o(n)$.

7.2.2 Wavelet Tree

Wavelet tree builds upon the results from our constant-time and sublinear space rank-select data structures and support fast rank-select-access query for a string with arbitrarily large alphabet Σ where $\sigma = |\Sigma|$.

Consider a perfectly balanced binary tree where each node corresponds to a subset of the alphabet Σ. The children of each node partition the node subset into two. A bit vector B_v at node v indicates which children each sequence position belongs. Each children handles the subsequence of the parent's sequence corresponding to its alphabet subset. The root of the tree handles the sequence $T[1 \dots n]$. The leaves each represent a single character of the alphabet and is not stored explicitly.

More concretely, suppose $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. We first divide the alphabet into to subsets $\{\texttt{A}, \texttt{C}\}$ and $\{\texttt{G}, \texttt{T}\}$. In the bit vector at the root, we store 0 at position i if $T[i] \in \{\texttt{A}, \texttt{C}\}$ and 1 if $T[i] = \{\texttt{G}, \texttt{T}\}$. The left child of the root represents the subset $\{\texttt{A}, \texttt{C}\}$ and the right child represents $\{\texttt{G}, \texttt{T}\}$. At the left child, we further split $\{\texttt{A}, \texttt{C}\}$ into $\{\texttt{A}\}$ and $\{\texttt{C}\}$. And similarly, at the right child, we split $\{\texttt{G}, \texttt{T}\}$ into $\{\texttt{G}\}$ and $\{\texttt{T}\}$. Note at the child node, we don't store the bit vector corresponding to the entire string, but only parts of the parent string with characters in the child's alphabet subset. So in our example, we only store the fraction of the string that has A and C in the left child of the root.

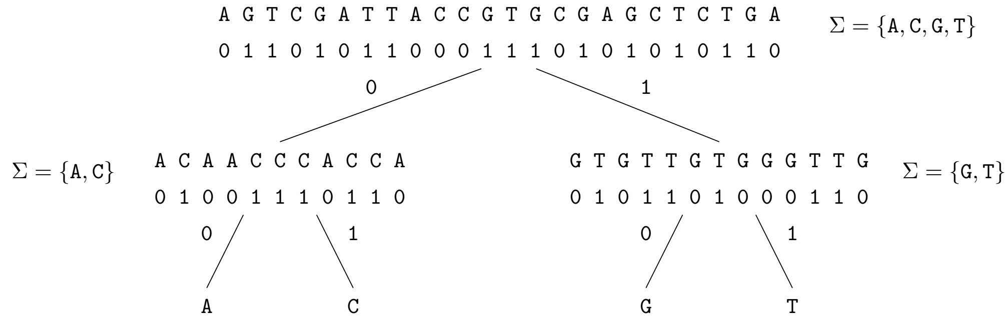


Figure 7.6: Example of a wavelet tree representing the string $T = \texttt{AGTCGATTACCGTGCGAGCTCTGA}$. We divide the alphabet as described in the example.

We want to implement three operations: Access, Rank, and Select. As we have seen, rank-select-access queries can all be done on a bit vector in $O(1)$ time, so all of the algorithms below run in $O(\log \sigma)$ time.

A wavelet tree can be constructed in $O(n \log \sigma)$ time and takes $n \log \sigma (1 + o(1))$ bits of space.

```
Access(T, i)
1  N = T.root
2  while N is not a leaf
3        B = N.bitvector
4        b = B.Access(i)
5        N = N.child[b]
6        i = B.Rank(b, i)
7  return N
```

```
Rank(T, c, i)
1  N = T.root
2  while N is not a leaf
3        B = N.bitvector
4        Σ = N.alphabet
5        if Σ.IndexOf(c) ≤ |Σ|/2
6              b = 0
7        else
8              b = 1
9        N = N.child[b]
10       i = B.Rank(b, i)
11 return i
```

```
Select(T, c, i)
1  N = T.leaf(c)
2  while N is not a leaf
3        N = N.parent()
4        B = N.bitvector
5        Σ = N.alphabet
6        if Σ.IndexOf(c) ≤ |Σ|/2
7              b = 0
8        else
9              b = 1
10       i = B.Select(b, i)
11 return i
```

7.2.3 Efficient LF Mapping

The main reason that we went on a detour about succinct data structures is so that we can to rank-select queries more efficiently on a BWT. It will be crucial to speeding up operations on our FM index.

An FM index for a text T consists of the BWT of T BWT(T) stored in a wavelet tree; and an integer array C which we call the *skip-amount array.* $C[c]$ stores the number of characters alphabetically smaller than c in T. Now, given a c at position i in L, we can compute the corresponding c in F using

$$\text{BWT}(T).\text{RANK}(c, i) + C[c].$$

This is correct because $C[c]$ gives us the offset where the c-block starts in F, and the rank query gives us the relative offset within the c-block because the L column and the F column have the same relative order of ranks. This means we can now compute the inverse BWT in $O(n \lg \sigma)$ time.

7.2.4 Count

To implement COUNT which outputs the number of occurrences of a given pattern P, we first make some important observations. We first note that every substring is a prefix of some suffix. Rows with the same prefix are consecutive in the BWT matrix, and characters in the last column (L) are those preceeding the prefixes in T. The idea of our counting algorithm is to start with the shortest suffix and try to match the pattern to successively longer suffixes.

We start from the last character p_m of the pattern and find a prefix of suffix that starts with p_m. Given a prefix of a suffix, we can peek at the previous character immediately preceeding the start of the prefix by looking at the corresponding character in the L column. We then perform an LF mapping to go to the previous character that matches with p_{m-1}. We slowly narrow our range, specified by the pointers *sp* and *ep* until we either reach the end of the pattern or when the size of our search range becomes 0.

```
COUNT(L, C, T, P)
  // Precondition: L is a BWT of T stored in a wavelet tree;
  // C is the skip-amount array; T is null-terminated with a $
  // P is the pattern that we are counting

 1  i = |P|
 2  (sp, ep) = (1, n)
 3  while sp ≤ ep and i > 1
 4      c = P[i]
 5      sp = C[c] + L.RANK(c, sp − 1)
 6      ep = C[c] + L.RANK(c, ep)
 7      i = i − 1
 8  if ep < sp
 9      return 0
10  else
11      return ep − sp + 1
```

See Figure 7.7 for an example of a run of COUNT. COUNT runs in $O(m \lg \sigma)$ time where $m = |P|$.

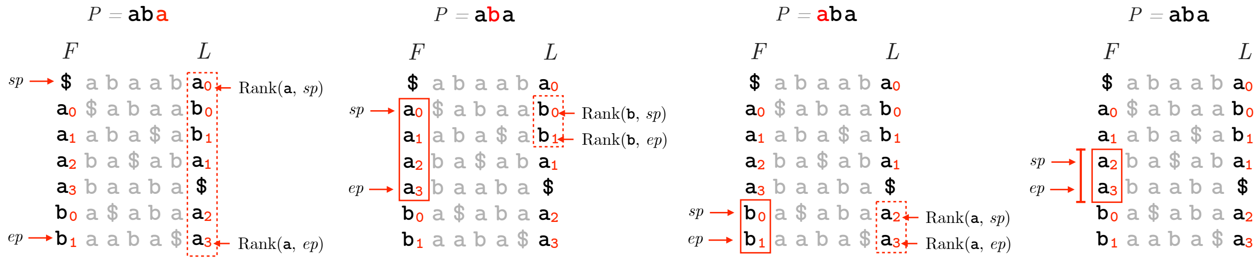


Figure 7.7: COUNT ran on $T =$ abaaba and $P =$ aba.

7.2.5 Locate

For LOCATE, the goal is to find all offsets where a given pattern P occurs as a substring of T. It is easy to implement LOCATE with the help of a suffix array because we can simply search through the suffix array and return the offsets at which P occurs as a prefix of the suffix. But a suffix array would be too big to store and it defeats the purpose of an FM index.

We instead stores a *sample suffix array* or *succinct suffix array*. Let r be the sample rate. Specifically, we keep the value of the suffix array at i if $\mathrm{SA}[i] = rk$ for $0 \leq k \leq \frac{n}{r}$ where n is the length of the string. In addition to the sample suffix array, we store a bit vector B such that $B[i] = 1$ iff $\mathrm{SA}[i] = rk$, that is, if the position i in the suffix array is sampled.

If $B[i] = 1$, we can access $\mathrm{SA}[i]$ by doing a rank query on the bit vector. So $\mathrm{SA}[i] = \mathrm{SSA}[B.\mathrm{RANK}(1, i)]$. Otherwise, if $B[i] = 0$, we can set an auxillary variable j to i, and then iteratively update j to $\mathrm{LF}(j)$ until $B[j] = 1$. We count the number of LF mappings required to get a 1-bit in the bit vector. Let d be the count. Then, $\mathrm{SA}[i] = \mathrm{SA}[j] + d = \mathrm{SSA}[B.\mathrm{RANK}(1, j)] + d$. In the pseudocode below, let (L, C) be the FM index for the text T where L is the BWT and C is the skip-amount array.

```
SA(i, SSA, B, L)
1  if B[i] == 1
2      return SSA[B.Rank(1, i)]
3  else
4      j = i
5      d = 0
6      while B[j] == 0
7          j = C[L[j]] + L.Rank(L[j], j)
8          d = d + 1
9      return SSA[B.Rank(1, j)] + d
```

Computing the value at i of the suffix array using a sample suffix array takes $O(r \lg \sigma)$ time where r is the sample rate since each of the $d \leq r$ steps requires one rank query which is done in $O(\lg \sigma)$ time on a wavelet tree. By setting $r = \lg^{1+\epsilon} n / \lg \sigma$, we can fit the entire sample suffix array in $o(n \lg \sigma)$ bits and the computation of suffix array entries takes $O(\lg^{1+\epsilon} n)$ time. Having implemented this, we can implement LOCATE quite easily without much space overhead.

7.3 Bidirectional BWT Index

Given a null-terminated string T and its reversal T^R, let $I(W,T)$ be the function that returns the interval in BWT(T) of the suffixes of T that is prefixed by string W. The interval $I(W,T)$ in the suffix array of T contain all the starting positions of W in T. We consider a data structure that supports the following operations:

1. Is-Left-Maximal(i,j): returns 1 iff substring BWT$(T)[i \ldots j]$ contains at least two distinct characters
2. Is-Right-Maximal(i,j): returns 1 iff substring BWT$(T^R)[i \ldots j]$ contains at least distinct characters
3. Enumerate-Left(i,j): returns all distinct characters that appear in BWT$(T)[i \ldots j]$
4. Enumerate-Right(i,j): returns all distinct characters that appear in BWT$(T^R)[i \ldots j]$
5. Extend-Left$(c, I(W,T), I(W^R,T^R))$: returns the pair $(I(cW,T), I(W^R cT^R))$
6. Extend-Right: returns the pair $(I(Wc,T), I(cW^R T^R))$

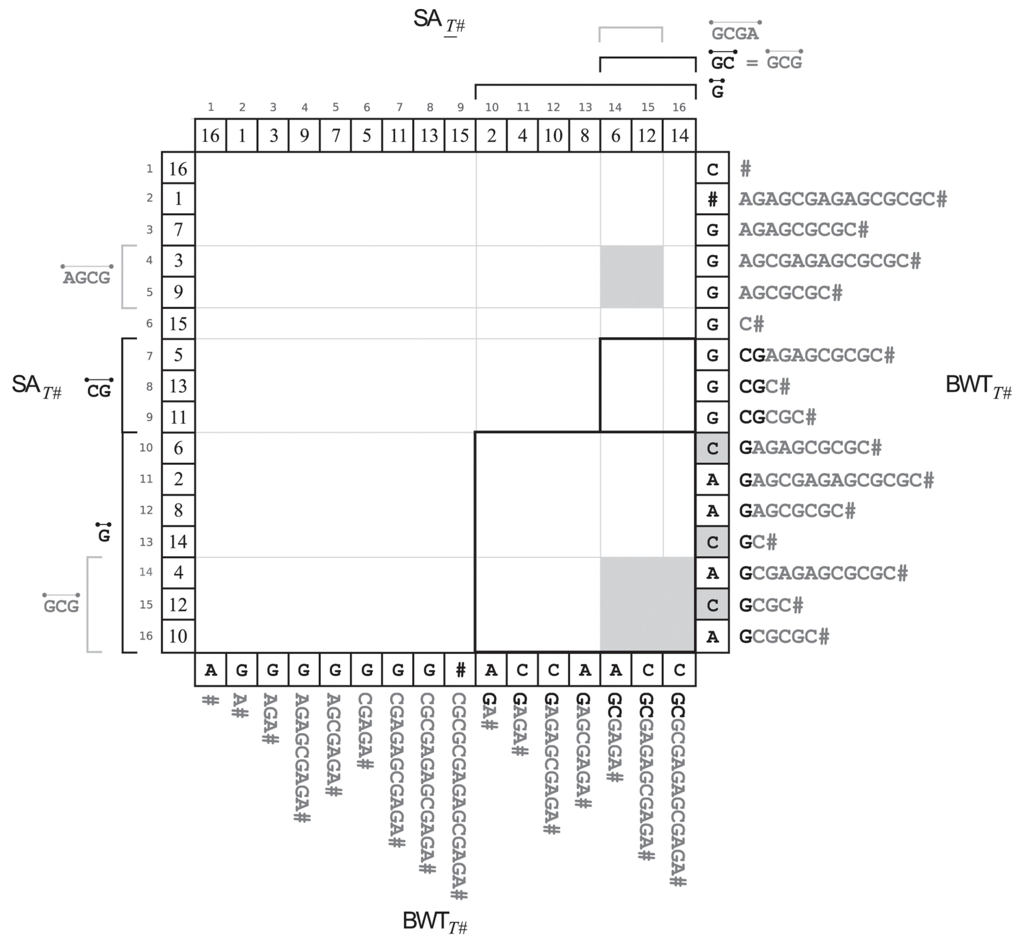


Figure 7.8: Example of a bidirectional BWT index for $T =$ `AGAGCGAGAGCGCGC`. The squares represent the interval pairs $(I(W,T), I(W^R,T^R))$.

Although we will not give the algorithm here, we note that a bidirectional BWT index allows for the traversal

of all nodes of the suffix tree in $O(n \log \sigma)$ time and $O(\sigma \log^2 n)$ space. We only discuss the high level idea of a bidirectional BWT and will mainly use it as a blackbox for implementing other algorithms in the subsequent chapters.

Theorem 7.1 (Belazzougui et al., 2019)**.** *We can find all intervals of the suffix array of T that corresponds to internal nodes of the suffix tree of T, as well as the length of the label of such nodes in $O(n \log \sigma)$ time and $O(\sigma \log^2 n)$ space.*

Theorem 7.2 (Belazzougui et al., 2019)**.** *Given a string $T \in [1 \dots \sigma]^n$, there is a representation of the bidirectional BWT index of T that uses $2n \lg \sigma (1 + o(1))$ bits of space and supports all operations except* Enumerate-Left *and* Enumerate-Left *in $O(\log \sigma)$ time.* Enumerate-Left *and* Enumerate-Left *can be supported in $O(d \log(\sigma/d))$ time where d is the output size.*

Bibliography

Burrows-Wheeler transform is due to Burrows and Wheeler [4]. It was originally introduced as a compression algorithm. FM index is developed by Ferragina and Manzini [7]. It is further improved to use wavelet tree. Jacobson's rank and Clark's select discussed in this chapter are from two PhD theses [10] [5]. Wavelet tree itself was developed by Grossi *et al.* [8]. Finally, bidirectional BWT index was developed by Belazzougui *et al.* [1]. The overall presentation of this chapter is based on Ben Langmead's lecture slides.

Part IV

Data in High-Dimensional Space

Chapter 8
Geometry of High-Dimensional Objects

Everything we looked at seems quite far removed from the biological application that we are interested. In this chapter and the subsequent chapters, we will slowly piece together the mathematical concepts and see how they affect and inspire the development of algorithms in computational biology.

For Part IV, we will look at is the dimensionality of the data that we will be dealing with. A lot of biological data originates from genome sequencing. Sequencing is the process in which biological sample (in this case, DNA or RNA) is read into a human readable string. There are a lot of intermediate steps involved between putting a sample into the sequencer and getting a string output, and we will cover some of the algorithmic aspects of this process in subsequent chapters. In this part, we will focus more on the dimensionality of our output. With the development of next-generation Sequencing (NGS) technologies, especially PacBio HiFi and Oxford Nanopore sequencing, we are now able to obtain raw sequencing reads of greater length. This also presents the problem of how to analyze long sequencing reads and even genomic data in general. In this chapter, we will be looking at some of the properties of data in high-dimensional space and how this may limit our ability to analyze these data.

8.1 Most Volume of High-dimensional Objects is Near the Surface

Most volume of high-dimensional objects is near the surface. As an example, we consider the unit ball.

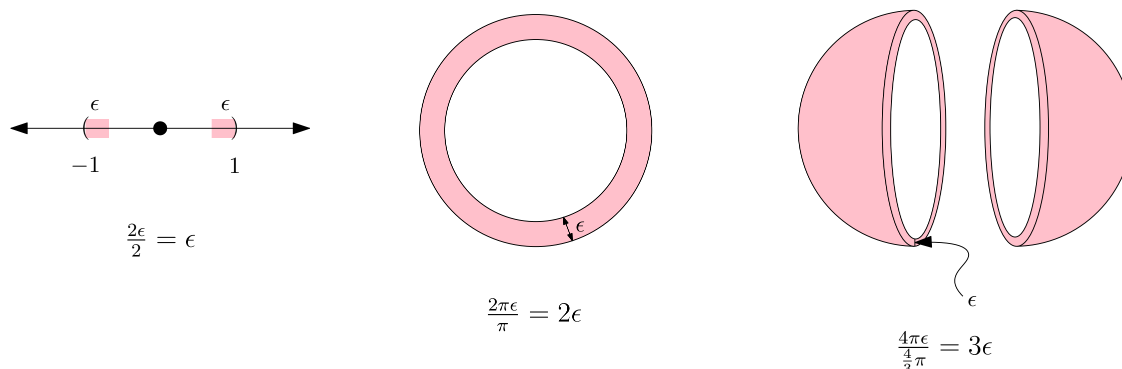


Figure 8.1: From left to right: 1D ball (interval); 2D ball (circle); 3D ball (sphere) and the corresponding approximate fraction of the volume ϵ distance away from the surface.

Theorem 8.1. *More rigorously, consider any object $A \subset \mathbb{R}^d$. Shrink A by a factor of ϵ to produce*

$$(1-\epsilon)A = \{(1-\epsilon)x \mid x \in A\}$$

Then,

$$V((1-\epsilon)A) = (1-\epsilon)^d V(A)$$

Proof. Partition A into infinitestimal hypercubes. Then, $(1-\epsilon)A$ is the union of the set of set of cubes

obtained by shrinking the cubes by a factor of $(1-\epsilon)$. Shrinking the side-lengths by $(1-\epsilon)$ implies that the volume of each hypercube is shrunk by $(1-\epsilon)^d$. □

Note that $1 - x \leq e^{-x}$, so for any $A \subset \mathbb{R}^d$,

$$\frac{V((1-\epsilon)A)}{V(A)} = (1-\epsilon)^d \leq e^{-\epsilon d} \to 0$$

as $d \to \infty$. Thus, most volume is not in $(1-\epsilon)A$.

Going back to the unit ball $B_d \subseteq \mathbb{R}^d$, we have

$$\frac{V((1-\epsilon)B_d)}{V(B_d)} \leq e^{-\epsilon d}$$

so

$$V(B_d \setminus (1-\epsilon)B_d) \geq (1 - e^{-\epsilon d})V(B_d).$$

Let $\epsilon = 1/d$. Then,

$$V(B_d \setminus (1-1/d)B_d) \geq (1-e^{-1})V(B_d) \approx 0.632V(B_d)$$

Most volume is contained in an annulus of width $1/d$ near the boundary. In other words, if $\epsilon = 1/d$, over half of the points in the hypersphere B_d is contained in the ϵ-annulus.

8.2 Most Points in a Unit Ball Are Nearly Orthogonal

Let us first define rigorously what it means for points to be "neraly orthogonal". Recall the dot product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{d} a_i b_i = \|\mathbf{a}\|\|\mathbf{b}\| \cos\theta_{\mathbf{ab}}.$$

So $\mathbf{a} \cdot \mathbf{b}$ is small (without having $\|\mathbf{a}\|$ and $\|\mathbf{b}\|$ being too small) implies $\mathbf{a}$ and $\mathbf{b}$ are "**nearly orthogonal**".

WLOG, fix $\mathbf{e}_1$, the first unit vector, as "North". Then, $\mathbf{e}_1 \cdot \mathbf{x} = x_1$. Then, by showing most points are near the equator, we can prove that most points are nearly orthogonal. Since the north pole is arbitrarily chosen, the result holds for any pair of points.

More formally, we show that most of the volume of the unit ball have $|x_1| \leq O(1/\sqrt{d})$.

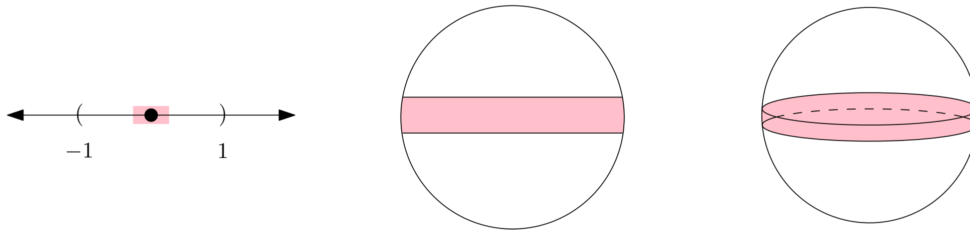


Figure 8.2: Most volume of a high-dimensional hypersphere is near the equator

Theorem 8.2. *For $c \geq 1$ and $d \geq 3$, at least*

$$1 - \frac{2}{c}e^{-\frac{c^2}{2}}$$

fraction of the volume of B_d has $|x_1| \leq \frac{c}{\sqrt{d-1}}$ for $\mathbf{x} \in B_d$.

Proof. By symmetry, we only consider the top hemisphere of the unit hypersphere. Let

$$A = \{\mathbf{x} \in B_d \mid x_1 \geq \frac{c}{\sqrt{d-1}}\}$$

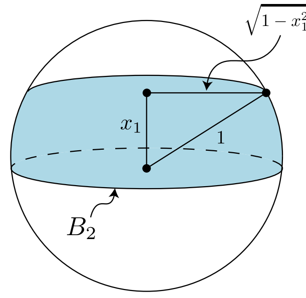
and let

$$H = \{\mathbf{x} \in B_d \mid x_1 \geq 0\}$$

be the upper hemisphere. Note that the volume of A, $V(A)$, can be expressed as an integral

$$V(A) = \int_{\frac{c}{\sqrt{d-1}}}^{1} \underbrace{(1-x_1^2)^{\frac{d-1}{2}}}_{\text{scale down}} \underbrace{V(B_{d-1})dx_1}_{\substack{\text{vol of unit radius} \\ \text{ball of dim } d-1}}$$

Pictorially,

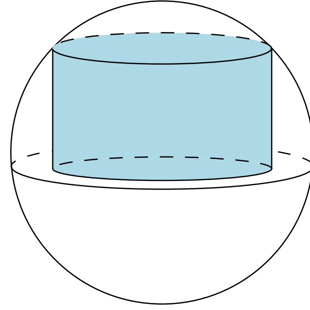


Since $1 - x \leq e^{-x}$ and $\frac{x_1\sqrt{d-1}}{c} \geq 1$ in A,

$$\begin{aligned}
\int_{\frac{c}{\sqrt{d-1}}}^{1} (1-x_1^2)^{\frac{d-1}{2}} V(B_{d-1})dx_1 &\leq \int_{\frac{c}{\sqrt{d-1}}}^{\infty} e^{-\frac{d-1}{2}x_1^2} V(B_{d-1})dx_1 \\
&\leq \int_{\frac{c}{\sqrt{d-1}}}^{\infty} \frac{x_1\sqrt{d-1}}{c} \cdot e^{-\frac{d-1}{2}x_1^2} \cdot V(B_{d-1})dx_1 \\
&= V(B_{d-1}) \cdot \frac{\sqrt{d-1}}{c} \int_{\frac{c}{\sqrt{d-1}}}^{\infty} x_1 e^{-\frac{d-1}{2}x_1^2} dx_1 \\
&= V(B_{d-1}) \cdot \frac{\sqrt{d-1}}{c} \cdot \left[-\frac{1}{d-1} e^{-\frac{d-1}{2}x_1^2} \right]_{\frac{c}{\sqrt{d-1}}}^{\infty} \\
&= \frac{V(B_{d-1})}{c\sqrt{d-1}} e^{-\frac{c^2}{2}}.
\end{aligned}$$

This gives us an upper bound on the volume of A. Next, we derive a lower bound on $V(H)$ using the fact $(1-x)^a > 1 - ax$ for $a \geq 1$.

$$\begin{aligned}
V(H) &\geq V(\{\mathbf{x} \in H \mid x_1 \leq \frac{1}{\sqrt{d-1}}\}) \\
&\geq \underbrace{V(B_{d-1}) \cdot \left(1 - \frac{1}{d-1}\right)^{\frac{d-1}{2}} \cdot \frac{1}{\sqrt{d-1}}}_{\text{vol of cylinder}} \\
&\geq \frac{V(B_{d-1})}{2\sqrt{d-1}}
\end{aligned}$$



Then,

$$\frac{V(A)}{V(H)} \leq \frac{2}{c} e^{-\frac{c^2}{2}}.$$

□

The theorem tells us most points in a high-dimensional sphere is near the equator, which is orthogonal to the arbitrarily chosen north pole. It follows that most pair of points are orthogonal.

Theorem 8.3. *Consider drawing n points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in B_d$ uniformly at random. Then, with probability $1 - O\left(\frac{1}{n}\right)$,*

1. $\|\mathbf{x}_i\| \geq 1 - \frac{2 \ln n}{8}$ *for all i, and*
2. $\|\mathbf{x}_i \cdot \mathbf{x}_j\| \leq \frac{\sqrt{6 \ln n}}{\sqrt{d-1}}$ *for all $i \neq j$.*

This theorem further formalizes the notion that most points are near the surface and most pairs of points are nearly orthogonal in a high-dimensional hypersphere.

Proof.

(1): For any fixed i,

$$\Pr(\|\mathbf{x}_i\| < 1 - \epsilon) \leq e^{-\epsilon d}$$

by Theorem 8.1. Then,

$$\Pr\left(\|\mathbf{x}_i\| < 1 - \frac{2 \ln n}{d}\right) \leq e^{-2 \ln n} = \frac{1}{n^2}.$$

By the union bound,

$$\Pr\left(\exists i,\ \|\mathbf{x}_i\| < 1 - \frac{2 \ln n}{d}\right) \leq n \cdot \Pr\left(\|\mathbf{x}_i\| < 1 - \frac{2 \ln n}{d}\right) = \frac{1}{n}.$$

(2): By Theorem 8.2, for fixed i,

$$\Pr\left(\|\mathbf{x}_i \cdot \mathbf{e}_1\| > \frac{c}{\sqrt{d-1}}\right) \leq \frac{2}{c} e^{-\frac{c^2}{2}}$$

This implies that

$$\Pr\left(\|\mathbf{x}_i \cdot \mathbf{e}_1\| > \frac{\sqrt{6 \ln n}}{\sqrt{d-1}}\right) \leq \frac{2}{\sqrt{6 \ln n}} e^{-3 \ln n} = \frac{2}{\sqrt{6 \ln n}} \cdot \frac{1}{n^3} \in O\left(\frac{1}{n^3}\right).$$

Note that this holds for any arbitrary North $\mathbf{e}_1$. There are $\binom{n}{2}$ pairs i and j, so for each pair, we define $\mathbf{x}_j/\|\mathbf{x}_j\|$ as the North. By union bound for all $i \neq j$, the dot product condition fails with probability at most $O\left(\binom{n}{2}n^{-3}\right) = O(1/n)$. □

Why does this matter? It turns out because most pairs of points are orthogonal, the volume of the unit sphere approaches 0 as the dimension goes to infinity. Because of this, it is hard to sample points randomly from a unit sphere. This result is formalized in the following corollary.

Corollary 8.4.

$$\lim_{d\to\infty} V(B_d) = 0$$

This should not be obvious, at least in low dimensions. In 1D, the volume is 2; in 2D, the volume is π; and in 3D, the volume is $\frac{4}{3}\pi$. This result can be shown directly from computing the volume (by integrating in polar coordinate), which gives us

$$V_d(r) = \begin{cases} 1 & n = 0 \\ 2r & n = 1 \\ \frac{2\pi}{n} r^2 \cdot V_{d-2}(r) \end{cases}$$

and taking the limit as $d \to \infty$ gives us $V_d \to 0$. However, this can also be proved as a corollary of Theorem 8.2.

Proof. Let $c = 2\sqrt{\ln d}$. By Theorem 8.2, at least a

$$1 - \frac{1}{\sqrt{\ln d}} e^{-2\ln d} = 1 - \frac{1}{d^2\sqrt{\ln d}}$$

fraction of the volume has $|x_1| \leq \frac{2\sqrt{\ln d}}{\sqrt{d-1}}$. This implies that for $\mathbf{x} \in B_d$ randomly drawn,

$$\Pr\left(|x_1| > \frac{2\sqrt{\ln d}}{\sqrt{d-1}}\right) < \frac{1}{d^2\sqrt{\ln d}} < \frac{1}{d^2}.$$

Let C be a box/hypercube with side length $\frac{4\sqrt{\ln d}}{\sqrt{d-1}}$ centered at the origin. This is a box covering the equator. If $\mathbf{z} \in C \cap B_d$, then

$$|z_i| \leq \frac{2\sqrt{d}}{\sqrt{d-1}}$$

for all $i \in \{1, \ldots, d\}$. By the union bound,

$$\Pr\left(\forall i,\, |x_i| > \frac{2\sqrt{\ln d}}{\sqrt{d-1}}\right) < \frac{1}{d} \leq \frac{1}{2}$$

for $d \geq 2$. Hence,

$$V(C \cap B_d) \geq \frac{1}{2} V(B_d),$$

but $V(C) = \left(\frac{4\sqrt{\ln d}}{\sqrt{d-1}}\right)^d = \left(\frac{16\ln d}{d-1}\right)^{d/2} \to 0$ as $d \to \infty$. This implies at least half the volume of the unit ball B_d approaches 0 as $d \to \infty$. It follows that $V(B_d) \to 0$ as $d \to \infty$. □

As we alluded to earlier, this fact actually makes it hard to sample directions uniformly from a ball. Sampling from a cube and then projecting the points to the ball inscribed within does not work because when projected to the ball, the resulting sampling would be biased towards the corners of the cube and thus not uniform.

And this not only affects random sampling. It also affects us when we want to separate or classify points in high-dimensional space, a common task in computational biology. Due to the previous results, one need exponentially (to the dimension) many points before getting one point that has a close neighbor in high-dimensional space. This means methods like k-nearest neighbor may fail to classify data points in high-dimensional space correctly, and canonical distance measure may fail to give us a reasonable approximation to the similarity or dissimilarity between two data points – both crucial to applications involving biological data. In the next chapter, we will start by looking at some biological applications involving points in high dimensional space.

Bibliography

Contents of this chapter roughly follows the second chapter of the book Foundations of Data Science [2].

Chapter 9
Comparing Data in High Dimension

In this chapter, we will consider the problem of comparing data in high dimensions. As we have seen in the previous chapter, as the dimension increases, it will become harder to separate or classify points in high-dimensional space. We will discuss two main topics in this chapter: dimensionality reduction, and a practical application in bioinformatics where we compare sequences without aligning them. Alignment-free sequence comparison is often used as an alternative to alignment-based comparison using dynamic programming.

9.1 Johnson-Lindenstrauss Lemma and Random Projection

9.1.1 Gaussian Annulus Theorem

Recall that most points in a d-dimensional hyperball are tightly concentrated near the surface. The question now is whether we can make similar conclusions about spherical Gaussians. Now, consider a d-dimensional vector $\mathbf{x} = (x_1, \ldots, x_d)$ where each $x_i \sim \mathcal{N}(0, 1)$ i.i.d. $\mathbf{x}$ is a **spherical Gaussian** random variable centered at the origin with unit variance in every direction.

Gaussian do not have a boundary like hyperballs, but we have

$$\mathbb{E}(\|\mathbf{x}\|^2) = \sum_{i=1}^{d} \mathbb{E}(x_i^2) = d\,\mathbb{E}(x_1^2) = d[\mathbb{E}(x_1^2) - \mathbb{E}(x_1)^2] = d.$$

We call $\sqrt{d}$ the radius of the spherical Gaussian. The Gaussian Annulus Theorem states that the points distributed according to spherical Gaussian are tightly concentrated in a thin annulus of with $O(1)$ at radius $\sqrt{d}$ just like points in hyperballs.

Theorem 9.1 (Gaussian Annulus Theorem)**.** *Let $\mathbf{x} = (x_1, \ldots, x_d)$ where $x_i \sim \mathcal{N}(0, 1)$ i.i.d. Let $r = \|\mathbf{x}\|$. Then for all $\beta \leq \sqrt{d}$, there exists some constant c, such that*

$$\Pr(|r - \sqrt{d}| \geq \beta) \leq 3e^{-c\beta^2}.$$

Proof. If $|r - \sqrt{d}| \geq \beta$, then $|r^2 - d| \geq \beta(r + \sqrt{d}) \geq \beta\sqrt{d}$. Thus, $\Pr(|r - \sqrt{d}| \geq \beta) \leq \Pr(|r^2 - d| \geq \beta\sqrt{d})$. Note

$$r^2 - d = (x_1^2 + x_2^2 + x_3^3 + \cdots + x_d^2) - d = (x_1^2 - 1) + \cdots + (x_d^2 - 1).$$

Let $y_i = x_i^2 - 1$. Then, $\mathbb{E}[y_i] = \mathbb{E}[x_i^2] - 1 = 0$. To apply the Master Tail Bound theorem, we first compute the moments of the variable y_i. Since for $|x_i| \leq 1$, $|y_i|^s \leq 1$ and for $|x_i| \geq 1$, $|y_i|^s \leq |x_i|^{2s}$, then

$$|\,\mathbb{E}[y_i^s]| \leq \mathbb{E}[|y_i|^s] \leq \mathbb{E}(1 + x_i^{2s}) = 1 + \mathbb{E}[x_i^{2s}]$$

which is equal to the even moments of Gaussian. Thus, by the Gamma integral,

$$\mathbb{E}[y_i^s] \leq 1 + \sqrt{\frac{2}{\pi}} \int_0^\infty x^{2s} e^{-\frac{x^2}{2}}\, dx \leq 2^s s!.$$

It follows that $\mathrm{Var}(y_i) = \mathbb{E}[y_i^2] - \mathbb{E}[y_i]^2 = \mathbb{E}[y_i^2] \leq 2^2 \cdot 2 = 8$. Unfortunately, we do not have $|\,\mathbb{E}[y_i^s]| \leq 8s!$ as required by the Master Tail Bound Theorem. To fix this, we let $w_i = y_i/2$. Then, $\mathrm{Var}(w_i) \leq 2$ and

$|\mathbb{E}[w_i^s]| \le 2s!$. We can now bound the probability that $|w_1 + \cdots + w_d| \ge \frac{\beta\sqrt{d}}{2}$ using the Master Tail Bound Theorem. Applying Theorem 2.18 with $\sigma^2 = 2$ and $n = d$, we have

$$\Pr(|r - \sqrt{d}| \ge \beta) \le \Pr\left(|w_1 + \cdots + w_d| \ge \frac{\beta\sqrt{d}}{2}\right) \le 3e^{-\frac{\beta^2 d}{48 \cdot d \cdot 2}} = 3e^{\frac{-\beta^2}{96}}.$$

The theorem holds by taking $c = \frac{1}{96}$. □

9.1.2 Random Projection

Random projection tackles the problem of finding nearest neighbor of a given point $\mathbf{v}$. More formally, given $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and $\mathbf{v} \in \mathbb{R}^d$, we would like to find $\text{argmax}_i \|\mathbf{x}_i - \mathbf{v}\|$. The naive approach is to compare the distance between $\mathbf{v}$ and every one of the n points. This takes $O(n)$ comparisons and each comparison takes $O(d)$ time to compute $\|\mathbf{x}_i - \mathbf{v}\|$, giving us an overall runtime of $O(dn)$. This is acceptable for low dimensional data, but it can be infeasible for high dimensional data. There are many different ways to solve this problem, including some techniques that we will cover in the next part on randomization. In this chapter, we will talk about a technique that makes use of properties of random Gaussians.

Definition 9.2 (Random Projection). *Pick k Gaussian vectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ in $\mathbb{R}^d$ with unit-variance coordinates. For any vector $\mathbf{v}$, define the projection $f(\mathbf{v})$ by*

$$f(\mathbf{v}) = (\mathbf{u}_1 \cdot \mathbf{v},\ \mathbf{u}_2 \cdot \mathbf{v},\ \ldots,\ \mathbf{u}_k \cdot \mathbf{v}).$$

Alternatively, one can think of $f(\mathbf{v})$ as the product of the vector $\mathbf{v}$ and a random matrix where each entry is i.i.d. Gaussian. That is, $f(\mathbf{v}) = \mathbf{A}\mathbf{v}$ where $a_{ij} \sim \mathcal{N}(0, 1)$ i.i.d.

Theorem 9.3 (Random Projection Theorem). *Let $\mathbf{v}$ be a fixed vector in $\mathbb{R}^d$ and let f be a projection function as defined in Definition 9.2. There exists constant $c > 0$ such that for $\epsilon \in (0, 1)$,*

$$\Pr\left(\Big|\|f(\mathbf{v})\| - \sqrt{k}\|\mathbf{v}\|\Big| \ge \epsilon\sqrt{k}\|\mathbf{v}\|\right) \le 3e^{-ck\epsilon^2}.$$

Proof. Without loss of generality, assume $\|\mathbf{v}\| = 1$; if not, we can rescale the inequality by a factor of $\|\mathbf{v}\|$. The sum of independent Gaussians is still Gaussian where the expectation and variance are the sums of the individual expectations and variances. Thus

$$\mathbb{E}[\mathbf{u}_i \cdot \mathbf{v}] = 0 \qquad \text{Var}(\mathbf{u}_i \cdot \mathbf{v}) = \text{Var}\left(\sum_{j=1}^{d} u_{ij} v_j\right) = \sum_{j=1}^{d} v_j^2 \text{Var}(u_{ij}) = \sum_{j=1}^{d} v_j^2 = 1.$$

Let $\mathbf{w} = (\mathbf{u}_1 \cdot \mathbf{v},\ \mathbf{u}_2 \cdot \mathbf{v}, \ldots, \mathbf{u}_k \cdot \mathbf{v})$ is a k-dimensional spherical Gaussian with unit variance in each coordinate. Then, the theorem follows from the Gaussian Annulus Theorem. □

The random projection theorem establishes that the probability of the length of the projection of a single vector differing significantly from its expected value is exponentially small in k, which is the dimension of the target space.

Next, we will prove the famous Johnson-Lindenstrauss Lemma, which says for a pair of n vectors, the pairwise distance is also preserved by this dimensionality reduction using random projection. It is an immediate consequence of the random projection theorem.

Lemma 9.4 (Johnson-Lindenstrauss Lemma)**.** *For any $0 < \epsilon < 1$ and any integer n, let $k \geq \frac{3}{c\epsilon^2} \ln n$ with c as in Theorem 9.1. For any set of n points in $\mathbb{R}^d$, the random projection $f : \mathbb{R}^d \to \mathbb{R}^k$ has the property that for all pair of points $\mathbf{v}_i$ and $\mathbf{v}_j$, with probability of at least $1 - \frac{3}{2n}$,*

$$(1-\epsilon)\sqrt{k}\|\mathbf{v}_i - \mathbf{v}_j\| \leq \|f(\mathbf{v}_i) - f(\mathbf{v}_j)\| \leq (1+\epsilon)\sqrt{k}\|\mathbf{v}_i - \mathbf{v}_j\|.$$

Proof. By the random projection theorem, for any fixed $\mathbf{v}_i$ and $\mathbf{v}_j$,

$$\Pr\left(\left|\|f(\mathbf{v})\| - \sqrt{k}\|\mathbf{v}\|\right| \geq \epsilon\sqrt{k}\|\mathbf{v}\|\right) \leq 3e^{-ck\epsilon^2} \leq \frac{3}{n^3}$$

for $k \geq \frac{3 \ln n}{c\epsilon^2}$. Since there are $\binom{n}{2} < n^2/2$ pairs of points, by the union bound, the probability that any pair has a large pairwise distance is bounded upper bounded by $\frac{3}{2n}$. □

Note that the Johnson-Lindenstrauss lemma holds for every pair of points in the set. Further, the number dimensions in the projection only needs to be logarithmically on n. Since k is often much less than d, it shows that random projection is a valid method for dimensionality reduction.

9.2 Alignment-Free Sequence Comparison

There are some immediate applications of random projection, including nearest neighbor search – the problem of finding the closest point to a given query point. Naive methods often suffer from the curse of dimensionality as we proved in the previous chapter. We can map the high dimensional data points to a lower dimension. We will discuss more on this topic in Chapter 12 where we cover locality sensitive hashing. In this section, we will discuss alignment-free sequence comparison, which is a technique used to compare biological sequences without explicitly aligning them.

Alignment-free comparison usually relies on the use of composition vectors. We represent the sequences S and T as vectors $\mathbf{s}$ and $\mathbf{t}$. The vectors encode some substructures of a specific type (e.g. k-mers), and values assigned to each entry of $\mathbf{s}$ and $\mathbf{t}$ usually correspond to the frequency of a given structure. Because of this, $\mathbf{s}$ and $\mathbf{t}$ are often called ***composition vectors***. In addition to a composition vector, we also need a distance metric to score the similarity and dissimilarity between vectors. In this section, we will cover the construction of the composition vector, different choice of distance metric, and finally, we will briefly touch upon the algorithms and data structures for computing the distance without explicitly constructing the composition vectors in cases where explicitly constructing the vectors might be computationally prohibitive.

The frequencies encoded in a composition vector captures genome-wide compositional biases and thus allows one to estimate similarity between two evolutionary distant species, a task not suitable for alignment based methods due to the large dissimilarity.

9.2.1 Distance Metric

Let S and T be two strings over the alphabet $\Sigma = \{1, \ldots, \sigma\}$. A substring $W \in \{1, \ldots, \sigma\}^k$ of a fixed length $k > 0$ is called a k-mer.

We denote by $f_S(W)$ the frequence (number of occurrences) of string W in S and $p_S(W)$ be the ***empirical probability*** of W occurring in S. $p_S(W)$ is normalized so that it satisfies the property of being a probability distribution.

$$p_S(W) = \frac{f_S(W)}{|S| - k + 1}.$$

The distance that we want to compute given **s** and **t** is the cosine similarity. We are using the term distance very loosely here since the functions we discuss may or may not be an actual distance function in $\mathbb{R}^n$.

Definition 9.5. *Given* **s** *and* **t***, we define the* ***cosine similarity*** *to be*

$$\kappa(\mathbf{s},\mathbf{t}) = \frac{\sum_W s_W t_W}{\sqrt{\left(\sum_W s_W^2\right)\left(\sum_W t_W^2\right)}}. \tag{9.1}$$

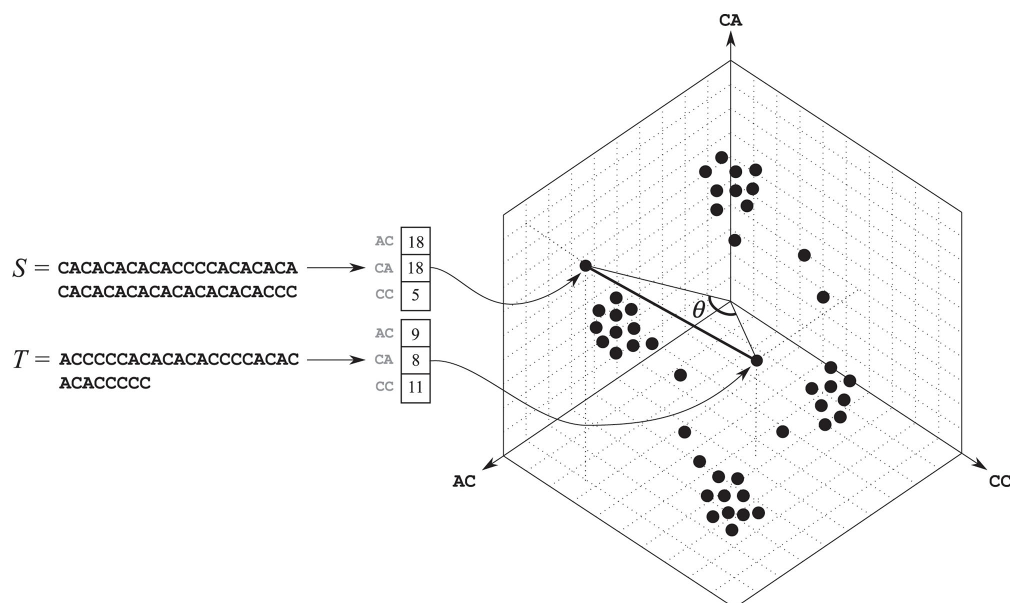


Figure 9.1: The k-mer kernel for $k = 2$. The distance is given by the Euclidean (ℓ_2) distance between **s** and **t**.

The cosine difference can be converted into another alternative measure of dissimilarity.

$$d(\mathbf{s},\mathbf{t}) = \frac{1-\kappa(\mathbf{st})}{2}.$$

Note that d is not a true distance function because it does not satisfy the Cauchy-Schwarz inequality. Recall the definition of the p-norm and ∞-norm.

$$\|\mathbf{s}-\mathbf{t}\|_p = \left(\sum_W |s_W - t_W|^p\right)^{1/p},$$

$$\|\mathbf{s}-\mathbf{t}\|_\infty = \max_W\{|s_W - t_W\}.$$

Cosine similarity is closely related to the Euclidean (ℓ_2) distance as it can be computed by dividing the dot product of two vectors by their Euclidean norm. Thus, we will consider instead the problem of creating a low-dimensional embedding that preserves the Euclidean distance between vectors using the random projection method introduced earlier.

9.2.2 Closest Pair of Sequences Using Random Projection

One can easily compute the Euclidean distance between two vectors in $O(d)$ time. However, this can often be time consuming for high-dimensional vectors. If one directly convert the sequence into a vector and attempt to compute the norm of such vector, it will take time proportional to the length of the sequence. One simple way to solve this problem is to use composition vectors. Composition vectors provides a low-dimensional summary of the substructures in the sequence. However, the shortcoming of using composition vectors is that it does not account for the relative orders of the substructures (k-mers, for example), nor does it consider structural differences caused by large indels. In this subsection, we cover an alternative way to reduce the dimension of sequences using random projection and see how this method can be used to find the closest pair of sequences given a set of sequences.

Consider the following algorithm. In the pseudocode, we use $position(s)$ to denote the encoding of the position of a substring s relative to a set of strings C. More concretely, let $position(s)$ output a tuple (i, j) where i is the index of the substring s in some $c \in C$, and j is the index of c relative to the set C. $position(s).src$ denotes the index of the sequence $c \in C$ from which the substring s is obtained. m is the number of random projections to perform, and d is the maximum number of substitutions allowed. The algorithm outputs a set of all pairs of ungapped k-mers with at most d substitutions.

```
Closest-Pair-Sequence(C, ℓ, d, m, k)
 1  A = ∅
 2  repeat m times
 3      f = Random-Positions(ℓ, k)
 4      Φ = 0
 5      for c ∈ C
 6          for 1 ≤ j ≤ |c| − ℓ + 1
 7              s = c[j ... j + ℓ − 1]
 8              Φ = Φ ∪ {(f(s), position(s))}
 9      𝒞 = {C_1, C_2, ...} = Partition(Φ)
10      for C_q ∈ 𝒞
11          for (f(s_i), position(s_i)), (f(s_j), position(s_j)) ∈ C_q × C_q
12              if position(s_i).src ≠ position(s_j).src and Count-Substitutions(s_i, s_j) ≤ d
13                  A = A ∪ {(position(s_i), position(s_j))}
14  return A
```

The subroutine Random-Position(ℓ, k) samples k elements from $\{1, \ldots, \ell\}$ uniformly with replacement. Partition(Φ) partitions Φ into classes with the same projection value. That is, it partitions Φ into $\mathcal{C} = \{C_1, \ldots\}$ where $\forall (f(s_1), p_1), (f(s_2), p_2) \in C_i \times C_i,\ f(s_1) = f(s_2)$.

We first analyze the runtime of this algorithm. For a set of N sequences each of max length n, creating the tuples takes $O(knN)$ time and partitioning also takes $O(knN)$ using hash table or radix sort. For checking the number of substitutions, it takes at most $\sum_q O(|C_q|^2)$ steps. In the subsequent analysis, we will show how to obtain a bound on m and k so that the algorithm is guaranteed to return the correct result with high probability. A concrete choice of m and k will also gives us a bound on the runtime for checking substitution.

Let s_1, s_2 be two sequences that differ in only d positions. A single randomly chosen projection sampling k positions from ℓ positions will project s_1 and s_2 to the same result with probability at least $(1-d/\ell)^k$. Hence, the probability that s_1 and s_2 are never projected together with m independent projections is bounded by

$$\left[1 - \left(1 - \frac{d}{\ell}\right)^k\right]^m .$$

This gives us a bound on the probability of a false negative (a pair of actually close sequences not included in the result). To achieve a false negative probability of at most θ, we take

$$m \geq \frac{\log \theta}{\log \left(1 - \left(1 - \frac{d}{\ell}\right)^k\right)}.$$

Next, for false positive, we assume a i.i.d. background sequence model. The probability that two independent random ℓ-mers differ by exactly t substitutions is give by the binomial distribution.

$$\beta_{1-\phi,\ell}(t) = \binom{\ell}{t}(1-\phi)^t \phi^{\ell-t}$$

where ϕ is the probability that two bases match. We take $\phi = 0.25$ for an equal distribution of bases in DNA sequence but the value can be modified for biased genomic sequences. The chance that two not related ℓ-mers projecting to the same value in a single projection is $(1 - t/\ell)^k$. Summing over all $t > d$, we have

$$FP = \sum_{t=d+1}^{\ell} \beta_{1-\phi,\ell}(t) \left(1 - \frac{t}{\ell}\right)^k.$$

For m reptitions, the overall false positive probability is $m \cdot FP$. With this bound on false positive probability, we can also bound the runtime for checking each pair of sequences in $C_q \times C_q$. The checking steps takes $O(m \cdot FP \cdot N^2 n^2)$.

9.2.3 String Kernel Methods

The algorithm we discussed in the previous subsection, while has a nice theoretical guarantee, does not run particularly fast and is not much better asymptotically than performing pairwise alignment for every pair of sequences.

Now that we have explored the methods for creating a low-dimensional embedding of the sequences, we consider a slightly different and more challenging problem – computing the distance between strings without explicitly constructing the composition vectors. This is called a *string kernel method.*

Definition 9.6 (String Kernel)**.** *Given two strings S and T, a **string kernel** is a function that simultaneously converts S and T into vectors $\mathbf{s}, \mathbf{t} \in \mathbb{R}^n$ for some $n > 0$ and computes a similarity or **distance** measure between $\mathbf{s}$ and $\mathbf{t}$ without building and storing $\mathbf{s}$ and $\mathbf{t}$ directly.*

We highlight that part of the definition that says "without building and storing $\mathbf{s}$ and $\mathbf{t}$ directly". Often time, it would be too costly to store and perform computation on such big vectors. Plus, we will encounter issues discussed in the previous chapter when dealing with vectors in high-dimensional space. In other words, the string kernel must be able to compute the similarity score *without ever constructing the vectors.*

The algorithms and kernels discussed in this chapter work even for non-standard norms:

$$N = \bigoplus_W g_1(s_W, s_T) \tag{9.2}$$

$$D_S = \bigotimes_W g_2(s_W) \tag{9.3}$$

$$D_T = \bigodot_W g_3(t_W) \tag{9.4}$$

where $\bigoplus, \bigotimes, \bigodot$ are *associative* and *communtative* operators and g_1, g_2, g_3 are arbitrary functions.

Substring and k-mer Kernels

The simplest way to represent a string as a composition vector is to count the frequency of all its distinct substrings of a fixed length (k-mers).

Definition 9.7 (k-mer Spectrum and Complexity)**.** *Given a string $S \in \{1, \ldots, \sigma\}^+$ and length $k > 0$, let vector $\mathbf{s}_k$ be such that $\mathbf{s}_k[w] = f_S(W)$ for every $W \in \{1, \ldots, \sigma\}^k$. The k-mer complexity $C(S, k)$ is the number of non-zero entries of $\mathbf{s}_k$.*

The ***k-mer kernel*** is then the function that computes Equation 9.5 evaluated on $\mathbf{s}_k$ and $\mathbf{t}_k$. If we remove the constraint that all substrings must be of the same length $k > 0$, we get the ***substring kernel***.

Consider a suffix tree of string S and recall that every substring is a prefix of a suffix. It follows that each k-mer (substring of length k) in S corresponds to some node v in the suffix tree such that the root-to-node label length $\ell(v) = k$. We can then compute the k-mer complexity as follows.

1. Initialize the count $C(S, k)$ to $|S| + 1 - k$. This is the number of leaves that correspond to suffixes of S of length at least k.
2. For each node v of the suffix tree, let $\ell(v)$ be the label on the path from root to v. If $|\ell(v)| < k$, do nothing. Otherwise, we increment $C(S, k)$ by 1 and decrement it by the number of children of v. This is correct because if $|\ell(v)| \geq k$, its children must have root-to-node label of length strictly greater than k so they cannot be a k-mer locus. We add 1 because v itself could potentially be a k-mer locus.

This is called the *telescoping technique*. We claim that at the end of this algorithm, $C(S, k)$ is equal to the number of unique k-mers in S. To see why, we need to show that it does not overcount nor undercount. Every node v that is at depth at least k and that is not a locus of a k-mer is both added to $C(S, k)$ when we visit them, and removed from $C(S, k)$ when we visit its parent. If its parent is visited, that means all its children cannot be a k-mer locus. All leaves at depth at least k is added at initialization, but subtracted if when its parent is visited. To show it also does not undercount, we note that every k-mer locus v is added to the count but never subtracted because its parent will have $|\ell(v.parent)| < k$.

This can be computed efficiently using bidirectional BWT introduced in Chapter 7. In particular, Theorem 7.1 tells us that we can traverse the vertices of a suffix tree efficiently, and Theorem 7.2 tells us that Enumerate-Right can be implemented correctly, which can be used to calculate $|\ell(v)|$ for every vertex v.

The cosine similarity between two k-mer spectra can also be calculated similarly using a telescoping technique like this.

9.2.4 Compression Distance

Finally, we briefly discuss another alternative method for comparing sequences without alignment. This does not suffer as much from the usual curse of dimensionality because it does not use a Euclidean metric but instead relies on information theory.

Recall the LZ complexity, defined in Chapter 3. We defined the normal compression distance with respect to LZ compression as

$$d(S, Q) = \frac{c(SQ) - \min\{c(S), c(Q)\}}{\max\{c(S), c(Q)\}}.$$

The can be generalized to every compressor satisfying a set of properties.

Definition 9.8 (Normal Compressor)**.** *A compressor is normal if it satisfies the following properties for all strings S, T, and U, up to an additive factor of $O(\log n)$ term, where n is the input size in bits:*

- *idempotency:* $C(SS) = C(S)$
- *monotonicity:* $C(ST) \geq C(S)$
- *symmetry:* $C(ST) = C(TS)$
- *distributivity:* $C(ST) + C(U) \leq C(SU) + C(TU)$
- *subadditivity:* $C(ST) \leq C(S) + C(T)$

For completeness, we prove that the normal compression distance is a distance metric (i.e. satisfies non-negativity, identity, symmetry, and the triangle inequality).

Theorem 9.9. *The normal compression distance is a distance metric.*

Proof. Assume C is a normal compressor.

Then, $d(S,T) \geq 0$ by the monotonicity and symmetry of C. $d(S,S) = 0$ by the idempotency of C. $d(S,T) = d(T,S)$ by the symmetry of C.

It remains to show that d satisfies the triangle inequality. Without loss of generality, suppose that $C(S) \leq C(T) \leq C(U)$. From symmetry, we know that it suffices to prove the triangle inequality for $d(S,T), d(T,U)$ and $d(S,U)$. That is, we prove that $d(S,T) + d(T,U) \leq d(S,U)$. By distributivity of C, we have

$$C(ST) + C(U) \leq C(SU) + C(TU)$$

and by symmetry

$$C(ST) + C(U) \leq C(SU) + C(UT).$$

Subtracting $C(S)$ from both sides of the inequality gives us

$$C(ST) - C(S) \leq C(SU) - C(S) + C(UT) - C(U).$$

Dividing both sides by $C(T)$ we have

$$\frac{C(ST) - C(S)}{C(T)} \leq \frac{C(SU) - C(S) + C(UT) - C(U)}{C(T)}.$$

By subadditivity, the LHS of this inequality is at most 1. If the RHS is at most one, then adding any positive number to both the numerator and denominator can only increase the ratio. If the RHS is greater than one, then adding a positive number to the numerator and denominator decreases the ratio, but the ratio is still greater than one. But the LHS is still at most one. Thus, by adding a positive number to both the numerator and denominator of the fraction on the RHS, the RHS remains greater than or equal to the LHS. It follows that by adding a δ such that $\delta = C(U) - C(T)$, we have

$$\frac{C(ST) - C(S)}{C(T)} \leq \frac{C(SU) - C(S)}{C(U)} + \frac{C(UT) - C(T)}{C(U)}.$$

□

Bibliography

Proof and presentation of Johnson-Lindenstrauss lemma and the Gaussian annulus theorem roughly follows the second chapter of the book Foundations of Data Science [2]. The idea of using of random projection for sequence comparison is based on the PhD thesis by Jeremy Buhler [3]. Rest of the discussion on alignment-free sequence comparison is based on [18].

Part V

Randomness and Randomization

Chapter 10

Markov Chain and Random Process

Markov chain is a useful tool for modeling random processes such as random walks. More specifically, Markov chains are often used to model situations where all the information of the system necessary to predict the future can be encoded in the current state. In this chapter, we will look at two important applications of Markov chains. The first application is the Markov Chain Monte Carlo (MCMC) method, widely used to sample a large space according to some probability distribution p. This is extremely useful when the sample space is very large. We can design a Markov chain where the states corresponds to the elements of the space and a useful property of Markov chain guarantee that by designing the Markov chain in a certain way, we will eventually converge to a stationary distribution.

The other application is Hidden Markov Models (HMMs), where we fit a Markov chain to model a random processes with unobservable states. This is used to solve the segmentation problem where we we would like to find a segmentation of the sequence S using information from other sequences with known segmentation.

10.1 Definitions

Definition 10.1 (Markov Chain)**.** *A* ***Markov chain*** *is a random process generating a sequence of states* $S = s_1 s_2 \cdots s_n$ *such that the probability of emitting each state* $s_i \in \Omega$ *is fixed and depends only on the previous states. This is called the* ***transition probability****, and we denote it* $\Pr(s_i \mid s_{i-1})$*. This property is referred to as* ***memorylessness****.*

A Markov chain can be described equivalently as a finite-state machine or direct graph.

Definition 10.2 (Markov Chain)**.** *Consider the graph* $G = (\Omega, E)$ *where each edge* $(x, y) \in E$ *has weight* $P_{x,y}$*.* P *is a matrix where all entries are non-negative and the sum of entries in every row equals 1.* P *is called the* ***transition matrix****. A* ***Markov chain*** *is a random walk on the graph* $(X_0, X_1, \ldots)$ *defined by* $X_0 = x_0$ *with*

$$\Pr(X_t = y \mid X_{t-1} = x, X_{t-2} = x_{t-2}, \ldots, X_0 = x_0) = \Pr(X_t = y \mid X_{t-1} = x) = P_{x,y}.$$

The vertex set Ω *is called the* ***state space*** *and the vertices are called* ***states****.*

More generally, the starting state does not have to be fixed, and can be random and given by a vector p, with entries indexed by Ω where $\Pr(X_0 = x) = p_x$. Further, if we remove the memorylessness requirement, we obtain a ***Markov chain with finite memory*** where

$$\Pr(X_t = y \mid X_{t-1} = x, X_{t-2} = x_{t-2}, \ldots, X_0 = x_0) = \Pr(X_t = y \mid X_{t-1} = x, \ldots, X_{t-m} = x_{t-m})$$

with $t > m$. m is the the memory span of the Markov chain and is called the ***order*** of the Markov chain.

10.1.1 Fundamental Theorem of Markov Chain

For each time step t, let us define a *row* vector $\mathbf{p}(t)$ with non-negative entries summing up to 1 and $p(t)_x = \Pr(X_t = x)$. Then, by definition of a transition matrix,

$$\mathbf{p}(t+1) = \mathbf{p}(t)P$$

and for $t > 0$

$$\mathbf{p}(t) = \mathbf{p}(0)P^t.$$

The goal of this section is to prove the Fundamental Theorem of Markov Chain that asserts a Markov chain satisfying a certain property that is truely memoryless has a stationary distribution as $t \to \infty$. We first formalize the two properties that will be important to proving the fundamental theorem.

Definition 10.3 (Irreducibility)**.** *A Markov chain with transition matrix P is called **irreducible** if for all states $x, y \in \Omega$, there exists a t such that $\Pr(X_t = y \mid X_0 = x) = (P^t)_{x,y} > 0$. Equivalently, the directed graph G represented by P is strongly connected.*

Theorem 10.4 (Aperiodicity)**.** *A Markov chain with transition matrix P is called **aperiodic** if for all states $x \in \Omega$, we have that the greatest common divisor of the set $\{t \geq 1 \mid \Pr(X_t = x \mid X_0 = x) = (P^t)_{x,x} > 0\}$ is equal to 1. That is, for any state $v \in \Omega$, $\gcd\{|c| \mid c \in C_v\} = 1$ where C_v is the set of all directed cycles that contains v. Equivalently, the undirected graph represented by P is bipartite.*

There are some graph properties that provide some sufficient condition for an aperiodic Markov chain.

Proposition 10.5. *Suppose that P represents an irreducible Markov chain with a graph G with has at least one self-loop. Then, the Markov chain is aperiodic.*

Proposition 10.6. *Suppose that the graph G represented by P is symmetric (i.e. if $(x, y) \in E$, then $(y, x) \in E$). Then, if G is also connected and contains an odd cycle, then the Markov chain is aperiodic.*

In addition to these two proposition, there is another important property of irreducible and aperiodic Markov chain that will be used in the proof of the fundamental theorem of Markov chain.

Lemma 10.7. *If X is a irreducible and aperiodic Markov chain with transition matrix P, then there exists a positive integer t_0 such that for all $t \geq t_0$ and all $x, y \in \Omega$, $(P^t)_{x,y} > 0$.*

Proof. For the proof of this lemma, we invoke the following result from elementary number theory.

> Let $\{c_1, c_2, \dots, c_N\}$ be a group of positive integers whose gcd is 1. Then, there exists a positive integer t_0 such that all integers $t \geq t_0$ can be written as $t = \sum_{i=1}^N a_i s_i$ for some nonnegative integers a_i.

We omit the proof of this result but it can be proved using Bézout's identity and induction on N. Continuing with the proof of the lemma, we note that irreducibility implies $\forall x, y \in \Omega, \exists t \in \mathbb{N}, (P^t)_{x,y} > 0$. Fix some $x \in \Omega$. Suppose that there are N cycles of lengths $c_1, c_2, \dots, c_N$ starting and ending at state x. By definition of aperiodicity,

$$\gcd(c_1, \dots, c_N) = 1.$$

By the earlier result from number theory, there exists a positive integer $t_0(x)$ such that for all $t \geq t_0(x)$, $t = \sum_{i=1}^N a_i c_i$ for nonnegative a_i's. Thus, for all $t \geq t_0(x)$, there is a cycle of length t from x to x obtained by traversing the ith cycle of length a_i times.

Now, by taking $t_0' = \max_{x \in \Omega}\{t_0(x)\}$, we have for all $t \geq t_0$ and $x \in \Omega$, $(P^t)_{x,x} > 0$. It remains to be shown that there exists t_0 such that for all $t \geq t_0$ and $x, y \in \Omega$, $(P^t)_{x,y} > 0$. Again, by irreducibility, for every $x, y \in \Omega$, there exists $t_{x,y}$ such that $(P^{t_{x,y}})_{x,y} > 0$. Fix a pair of $x, y \in \Omega$ and take $t_{x,y}$. We claim that there is a path of length $t_0' + t_{x,y}$ from x to y by traversing a thourgh cycle of length t_0' along the path from x to y. By construction of t_0', for every $x, y \in \Omega$, there exists $t_{x,y}$ such that for all $t \geq t_0' + t_{x,y}$, $(P^{t_0' + t_{x,y}})_{x,y} > 0$.

The lemma holds by taking $t_0 = t_0' + \max_{x,y \in \Omega}\{t_{x,y}\}$ because $t_0 \geq t_0' + t_{x,y}$ for all $x, y \in \Omega$. □

We now prove the Fundamental Theorem of Markov Chain.

10.2 Fundamental Theorem of Markov Chain

Definition 10.8 (Stationary Distribution)**.** *A probability distribution π is* ***stationary*** *for a Markov chain with transition matrix P if $\pi P = \pi$. In other words, π is stationary if, when the distribution of X_0 is given by $\Pr(X_0 = x) = \pi_x$, then the distribution of X_1 is also given by $\Pr(X_1 = x) = \pi_x$.*

Theorem 10.9 (Fundamental Theorem of Markov Chain)**.** *For every irreducible and aperiodic Markov chain with transition matrix P, there exists a unique stationary distribution π. Moreover, for all $x, y \in \Omega$, $(P^t)_{x,y} \to \pi_y$ as $t \to \infty$. Equivalently, for every starting state $X_0 = x$, $\Pr(X_t = y \mid X_0 = x) \to \pi_y$ as $t \to \infty$.*

10.2.1 Existence of Stationary Distribution

We show that for every finite Markov chain P, there exists some row vector π such that $\pi P = \pi$. This is equivalent to $(\pi P)^\top = P^\top \pi^\top = \pi^\top$ so 1 is an eigenvalue of $P^\top$ with non-negative eigenvector $\pi^\top$. We will prove this using a theorem from linear algebra known the Perron-Frobenius Theorem.

Theorem 10.10 (Perron-Frobenius Theorem)**.** *Each nonnegative matrix A has a nonnegative real eigenvector $\mathbf{x}$ with eigenvalue $\lambda = \max\{|\lambda_i|\}$, where $\{\lambda_1, \ldots, \lambda_n\}$ are eigenvalues of A.*

Since P is a transition matrix, we have $P \cdot 1 = 1$ by definition. Thus, P has an eigenvalue 1. Since every eigenvalue of P is no large than the row sum which is 1, 1 is the largest eigenvalue. Further, we notice that $\det(P^\top - \lambda I)$ has the same solution as $\det(P - \lambda I)$. Thus, the maximum eigenvalue of $P^\top$ is also 1. By Perron-Frobenius theorem, there exists a nonnegative eigenvector π such that

$$P^\top \pi = \pi \quad \Longleftrightarrow \quad \pi^\top P = \pi^\top.$$

It follows that the normalized vector $\pi/\|\pi\|$ is a stationary distribution of P.

10.2.2 Total Variation Distance

For the proof of the fundamental theorem, we use the total variation distance to show that the distribution converges to the stationary distribution π. We prove convergence by showing that the total variation distance approaches zero.

Definition 10.11 (Total Variation Distance)**.** *The* ***total variation distance*** *between two distributions μ and ν on a countable state space is*

$$d_{\mathrm{TV}} = \frac{1}{2} \sum_{x \in \Omega} |\mu(x) - \nu(x)|.$$

Visually, the total variation distance is half of the area enclosed by the two curves of the probability distribution functions.

The following lemma provide an equivalent characterization of the total variation distance that will be more useful in our later proofs.

Lemma 10.12. *Let μ and ν be two probability over a finite sample space Ω. Then,*

$$d_{\mathrm{TV}}(\mu, \nu) = \max_{S \subseteq \Omega} \left| \Pr_{X \sim \mu}(X \in S) - \Pr_{Y \sim \nu}(Y \in S) \right|.$$

For notation simplicity, we will write $\Pr_{X\sim\mu}(X \in S)$ and $\Pr_{Y\sim\nu}(Y \in S)$ as $\mu(S) = \sum_{s \in S} \mu(s)$ and $\nu(S) = \sum_{s\in S} \nu(s)$.

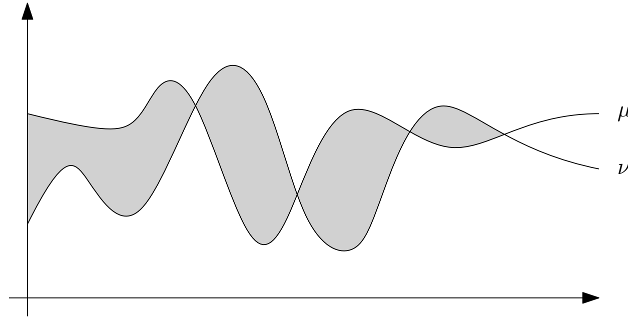


Figure 10.1: The area between the graph of the two distribution μ and ν.

Proof. Let A be the subset of those elements x in Ω for which $\mu(x) \geq \nu(x)$. Then,

$$\begin{aligned} d_{\text{TM}} &= \frac{1}{2} \sum_{x \in \Omega} |\mu(x) - \nu(x)| \\ &= \frac{1}{2} \left(\sum_{x \in A} (\mu(x) - \nu(x)) + \sum_{x \in \Omega \setminus A} (\nu(x) - \mu(x)) \right) \\ &= \frac{1}{2} (\mu(A) - \nu(A) + \nu(\Omega \setminus A) - \mu(\Omega \setminus A)) \\ &= \mu(A) - \nu(A) \\ &= \max_{S \subseteq \Omega} |\mu(S) - \nu(S)|. \end{aligned}$$

The second to last inequality holds because $\mu(A)+\mu(\Omega\setminus A) = 1 = \nu(A)+\nu(\Omega\setminus A)$, which implies $\mu(A)-\nu(A) = \nu(\Omega \setminus A) - \mu(\Omega \setminus A)$. ☐

10.2.3 Coupling

Another important concept that will be needed for the proof of the fundamental theorem and the bound on mixing time is the notion of coupling. The coupling of two distributions is simply a joint distribution of them.

Definition 10.13 (Coupling)**.** *Let μ and ν be two distributions on the same space Ω. Let ω be a distribution over the space $\Omega \times \Omega$ If $(X, Y) \sim \omega$ with $X \sim \mu$ and $Y \sim \nu$, then ω is called a* ***coupling*** *of μ and ν.*

For an example of couplings, consider the scenario where we have two coins: one is fair and the other one is loaded so that $\Pr(H) = \frac{1}{3}$ and $\Pr(T) = \frac{2}{3}$. The following two tables define two different couplings of the distribution of the coins.

coin 1 \ coin 2	Head	Tail	Pr(coin 1)
Head	1/3	1/6	1/2
Tail	0	1/2	1/2
Pr(coin 2)	1/3	2/3	

coin 1 \ coin 2	Head	Tail	Pr(coin 1)
Head	1/6	1/3	1/2
Tail	1/6	1/3	1/2
Pr(coin 2)	1/3	2/3	

The table defines a joint distribution and the sum of a certain row or column equal to the corresponding marginal probability. Among all the possible couplings, sometimes we are interested in the one who is "mostly coupled". We will formalize the notion of an "optimal" coupling in the subsequent subsections.

Lemma 10.14 (Coupling Lemma)**.** *Let μ and ν be two distributions on a sample space Ω. Then, for any coupling ω of μ and ν,*

$$\Pr_{(X,Y)\sim\omega}(X \neq Y) \geq d_{\mathrm{TV}}(\mu,\nu).$$

Moreover, there exists a coupling ω^* of μ and ν such that the inequality is tight. That is,

$$\Pr_{(X,Y)\sim\omega^*}(X \neq Y) = d_{\mathrm{TV}}(\mu,\nu).$$

Proof. Clearly,

$$\Pr_{(X,Y)\sim\omega}(X = Y) = \sum_{x\in\Omega}\Pr(X = Y = t) \leq \sum_{x\in\Omega}\min\{\mu(x),\nu(x)\}.$$

Thus,

$$\begin{aligned}\Pr_{(X,Y)\sim\omega}(X \neq Y) &\geq 1 - \sum_{x\in\Omega}\min\{\mu(x),\nu(x)\}\\ &= \sum_{x\in\Omega}(\mu(x) - \min\{\mu(x),\nu(x)\})\\ &= \max_{S\subseteq\Omega}\{\mu(S) - \nu(S)\}\\ &= d_{\mathrm{TV}}(\mu,\nu).\end{aligned}$$

For the optimal coupling, we construct ω^* such that for each $(x,y)\in\Omega\times\Omega$,

$$\omega^* = \begin{cases}\min\{\mu(x),\nu(y)\} & \text{if } x = y\\ \dfrac{\max\{\mu(x)-\nu(x),\,0\}\cdot\max\{\nu(y)-\mu(y),\,0\}}{d_{\mathrm{TM}(\mu,\nu)}} & \text{otherwise.}\end{cases}$$

We leave verifying that ω^* is a coupling as an exercise to the reader. It suffices to show that for $(X,Y)\sim\omega^*$,the marginal for X and Y are indeed μ and ν, respectively. Finally, if we let $\Omega^+ = \{x\in\Omega \mid \mu(x)\geq\nu(x)\}$ and $\Omega^- = \Omega\setminus\Omega^+ = \{x\in\Omega \mid \mu(x) < \nu(x)\}$, then

$$\begin{aligned}\Pr_{(X,Y)\sim\omega^*}(X = Y) &= \sum_{x\in\Omega}\omega^*(x,x)\\ &= \sum_{x\in\Omega}\min\{\mu(x),\nu(x)\}\\ &= \sum_{x\in\Omega^+}\nu(x) + \sum_{x\in\Omega^-}\mu(x)\\ &= \nu(\Omega^+) - \mu(\Omega^-)\\ &= 1 - (\mu(\Omega^+) - \nu(\Omega^+))\\ &= 1 - d_{\mathrm{TV}}(\mu,\nu).\end{aligned}$$

So, $\Pr_{(X,Y)\sim\omega^*}(X \neq Y) = d_{\mathrm{TV}}(\mu,\nu)$, as desired. □

10.2.4 Proof of the Fundamental Theorem of Markov Chain

We are finally ready to prove the fundamental theorem of Markov chain. The proof consists of three main parts:

1. Show that there exists a stationary distribution. This is done in Subsection 10.2.1;

2. Use Step 1 and aperiodicity to show that $\lim_{t\to\infty}(P^t)_{x,y} = \pi$ for all $x, y \in \Omega$. We do this by showing that for any two $x, y \in \Omega$, $d_{\mathrm{TV}}(p_x(t), p_y(t)) \to 0$ as $t \to \infty$ and $d_{\mathrm{TV}}(p_x(t), \pi) \to 0$ as well;
3. Use Step 1 and 2 to show the uniqueness of the stationary distribution π constructed in the proof of Lemma 10.14.

Recall that we denote $\Pr(X_t = x)$ as $p_x(t)$. This notation is introduced in the first paragraph of Subsection 10.1.1.

Proof (convergence). Define $\Delta(t) = \max_{x\in\Omega} d_{\mathrm{TV}}(p_x(t) - \pi)$. We will show that $\lim_{t\to\infty}\Delta(t) = 0$. First, fix two arbitrary elements $x, y \in \Omega$. We argue that $p_x(t)$ and $p_y(t)$ converge to the same distribution that is the stationary distribution π.

Let X_t and Y_t be two copies of the same Markov chain such that initially, $X_0 = x$ is independent and $Y_0 \sim \pi$ is distributed according to the stationary distribution π, and X_t and Y_t stay independent, and evolve according to the Markov chain, until the first time T where $X_T = Y_T$. Then, for any $t \geq T$, X_t and Y_t stick together. This is a coupling of X and Y. Formally, $(X_t, Y_t)_t$ is a Markov chain on $\Omega \times \Omega$ with a transition probability given by

$$Q((x_1, y_1), (x_2, y_2)) = \begin{cases} P(x_1, x_2) \cdot P(y_1, y_2) & \text{if } x_1 \neq y_1 \\ P(x_1, x_2) & \text{if } x_1 = y_1 \text{ and } x_2 = y_2 \\ 0 & \text{if } x_1 = y_1 \text{ and } x_2 \neq y_2. \end{cases}$$

$$\begin{array}{ccccccccc} \mu_0 & & \mu_1 & & & & \pi & & \pi & & \\ \wr & & \wr & & & & \wr & & \wr & & \\ X_0 & \to & X_2 & \to & \cdots & & X_T & \to & X_{T+1} & \to & \cdots \\ Y_0 & \to & Y_2 & \to & \cdots & & Y_T & \to & Y_{T+1} & \to & \cdots \\ \wr & & \wr & & & & \wr & & \wr & & \\ \pi & & \pi & & & & \pi & & \pi & & \end{array}$$

Let $T = \min\{t : X_t = Y_t\}$ be a random variable for the earliest time X_t and Y_t meet. By the coupling lemma, for all t,

$$d_{\mathrm{TV}}(p_x(t), p_y(t)) \leq \Pr(X_t \neq Y_t) = \Pr(T > t).$$

By Lemma 10.7, there exists some time step τ such that, for every two elements $z_1, z_2 \in \Omega$, $(P^\tau)_{z_1,z_2} > 0$. Let $C = \min_{z_1,z_2\in\Omega}\{(P^\tau)_{z_1,z_2}\} > 0$, and it follows that $(P^\tau)_{x_1,z} \cdot (P^\tau)_{y_1,z} \geq C^2$ for every $x_1, y_1, z \in \Omega$. This tells us that after τ time steps, X_t and Y_t will meet with probability of at least C^2. Thus, using X_t and Y_t to denote the state of the Markov chain at step t, we have

$$\Pr(X_{k\tau} \neq Y_{k\tau}) \leq (1 - C^2)^k,$$

which approaches 0 as $t = k\tau \to \infty$. Therefore, $d_{\mathrm{TV}}(p_x(t), p_x(t)) \to 0$ as $t \to \infty$. □

Proof (uniqueness). Suppose for contradiction that there exists some other stationary distribution π'. But, by convergence and assumption that π' is stationary,

$$\pi' = \lim_{t\to\infty} \pi'(P^t)_{x,y} = \lim_{t\infty\infty}(P^t)_{x,y} = \pi.$$

This contradicts our assumption that $\pi \neq \pi'$. □

This concludes the proof of the Fundamental Theorem of Markov Chain.

10.3 The Metropolis-Hastings Algorithm

Having proven the Fundamental Theorem of Markov Chain, we will consider an application of it. The problem we would like to solve in this section is described as follows. Suppose we have a state space Ω and a vector of positive weights $\mathbf{w}$, indexed by Ω. The weights define a probability distribution π, given by

$$\pi_x = \frac{w_x}{\sum_{y\in\Omega} w_y}.$$

We want to sample a random variable variable X, taking values in Ω. so that for every $x \in \Omega$, $\Pr(X = x) = \pi_x$. In practice, $|\Omega|$ can be very large and we would like the sampling to be much faster than $O(|\Omega|)$. To solve this problem, we construct a Markov chain with its stationary distribution as π. One very general method to do this is known as the Metropolis-Hastings algorithm.

The Metropolis-Hasting algorithm is a general method to design a Markov chain whose stationary distribution is a given target distribution $\mathbf{p}$. Suppose that Ω is some arbitrary state space, and $G = (\Omega, E)$ is a connected graph such that if $(x, y) \in E$ and $(y, x) \in E$. Let d be an upper bound on the maximum out-degree in G for all state $x \in \Omega$. Let $\mathbf{w}$ be the vector of positive weights that we will use as our stationary distribution. A single step on this Markov chain is as follows.

Metropolis-Step(X_t)
1 $N(X_t) = \{y \in \Omega \mid (X_t, y) \in E\}$
2 pick y so that for any $y \in N(X_t)$, $\Pr(Y = y) = 1/d$ and $\Pr(Y = \perp) = 1 - \frac{|N(X_t)|}{d}$
3 **if** $Y == \perp$
4 $\quad X_{t+1} = X_t$
5 **else**
6 $\quad X_{t+1} = \begin{cases} Y & \text{with probability } \frac{1}{2}\min\{1, \frac{w_Y}{w_{X_t}}\} \\ X_t & \text{with probability } 1 - \frac{1}{2}\min\{1, \frac{w_Y}{w_{X_t}}\} \end{cases}$

The following theorem proves the correctness of the Metropolis-Hastings algorithm.

Theorem 10.15. *The Markov chain constructed by* Metropolis-Step(X_t) *is irreducible, aperiodic, and has stationary distribution* $\pi = \frac{w_x}{\sum_{y\in\Omega} w_y}$.

Proof. The Markov chain is irreducible because we assumed that G is connected and symmetric, so it is strongly connected. Moreover, it is aperiodic because it is irreducible with at least one self-loop (as indicated by Line 4 of the algorithm). For any $x \neq y$ such that $(x, y) \in E$, the transition probability is

$$P_{x,y} = \frac{1}{d} \cdot \frac{1}{2} \min\left\{1, \frac{w_y}{w_x}\right\}.$$

Moreover, by symmetry,

$$P_{y,x} = \frac{1}{d} \cdot \frac{1}{2} \min\left\{1, \frac{w_x}{w_y}\right\}.$$

It follows that

$$\pi_x P_{x,y} = \frac{1}{2d\sum_{z\in\Omega} w_z} \cdot \min\{w_x, w_y\} = \pi_y P_{y,x}.$$

Therefore, the Markov chain is time reversible, and thus has a stationary distribution π that is uniform over Ω according to Lemma 10.17. Uniformity is from the fact that at each step, we have a 1/2 probability of moving to a new state and any neighboring states is equally likely. $\square$

We will also prove the non-trivial fact we used at the end of the proof.

Definition 10.16 (Time Reversible Markov Chain)**.** *A Markov chain with transition matrix P is* ***reversible*** *if there exists a probability distribution over Ω given by a vector π such that*

$$\pi_x P_{x,y} = \pi_y P_{y,x}.$$

Lemma 10.17. *If P defines a time-reversible Markov chain, and π satisfies that $\pi_x P_{x,y} = \pi_y P_{y,x}$, then π is a stationary distribution.*

Proof. By definition of time-reversible Markov chain,

$$(\pi P)_y = \sum_{x\in\Omega} \pi_x P_{x,y} = \sum_{x\in\Omega} \pi_y P_{y,x} = \pi_y \sum_{x\in\Omega} P_{y,x} = \pi_y.$$

□

The Metropolis-Hastings algorithm falls under a general class algorithms known as Markov Chain Monte Carlo (MCMC). These algorithms roughly follows this paradigm: We construct a sequence of random variables that converges to the target probability distribution π. By the Fundamental Theorem of Markov Chain, we should converge to this distribution regardless of our starting point as long as our chain is irreducible and aperiodic.

10.4 Gibbs Sampling

The Gibbs sampling algorithm is another example of MCMC algorithms. Gibbs sampling can be used when we are given the conditional probabilities of the parameters of interest, and we are interested in finding their joint probabilities. To generate samples of $\mathbf{x} = (x_1, \ldots, x_d)$ from a target distribution $p(\mathbf{x})$, we repeat the following steps:

1. Choose a variable x_i to be updated
2. Update x_i to a new value sampled based on the marginal probability of x_i with other variables fixed.

Let $\mathbf{x}, \mathbf{y}$ be two states that differ in only one coordinate. Without loss of generality, assume the first coordinate is different. Then,

$$P_{\mathbf{x},\mathbf{y}} = \frac{1}{d} p(y_1 \mid x_2, x_3, \ldots, x_d).$$

Similarly,

$$P_{\mathbf{y},\mathbf{x}} = \frac{1}{d} p(x_1 \mid y_2, y_3, \ldots, y_d) = \frac{1}{d} p(x_1 \mid x_2, x_3, \ldots, x_d).$$

By definition of conditional probability,

$$\begin{aligned} P_{\mathbf{x},\mathbf{y}} = \frac{1}{d} p(y_1 \mid x_2, x_3, \ldots, x_d) &= \frac{1}{d} \frac{p(y_1 \mid x_2, \ldots, x_d) \cdot p(x_2, \ldots, x_d)}{p(x_2, \ldots, x_d)} \\ &= \frac{1}{d} \frac{p(y_1, x_2, \ldots, x_d)}{p(x_2, \ldots, x_d)} \\ &= \frac{1}{d} \frac{p(\mathbf{y})}{p(x_2, \ldots, x_d)}. \end{aligned}$$

Similarly,

$$P_{\mathbf{y},\mathbf{x}} = \frac{1}{d} \frac{p(\mathbf{x})}{p(x_2, \ldots, x_d)}.$$

It follows that $p(\mathbf{x})P_{x,y} = p(\mathbf{y})P_{y,x}$. By Lemma 10.17, this Markov chain has a stationary distribution that is $\mathbf{p}$.

Gibbs sampling is applied in bioinformatics to identify repeating motifs in DNA sequence. We describe one step of this iterative algorithm. In this example, we set the k-mer size $k = 7$.

1. Choose a sequence for samping.

```
ACCATGACAG
GAGTATACCT
CATGCTTACT
CGGAATGCAT
```

2. Choose a random motif position for each sequence that is not selected for sampling.

```
ACCATGACAG
GAGTATACCT
CATGCTTACT
CGGAATGCAT
```

3. Construct count table for each position of the selected k-mer. Position 0 represents the portion of the string not covered by the selected k-mer.

	0	1	2	3	4	5	6	7
A	3	0	1	1	2	1	0	0
C	2	0	1	0	0	1	2	1
G	2	2	1	0	0	0	1	0
T	2	1	0	2	1	1	0	2

4. Convert count table to frequency table according to the following formula. $c_{i,j}$ represents the (i, j)-entry of the count table and b_i and B are pseudocount adjustments so that we don't end up with a zero entry in the final table. N is the number of sequences.

$$q_{i,j} = \frac{c_{i,j} + b_i}{N - 1 + B} \qquad q_{j,0} = \frac{c_{i,0} + b_i}{\sum_{k=1}^{i} c_{k,0} + B}$$

	0	1	2	3	4	5	6	7
A	0.31	0.1	0.3	0.3	0.5	0.3	0.1	0.1
C	0.23	0.1	0.3	0.1	0.1	0.3	0.5	0.3
G	0.23	0.5	0.3	0.1	0.1	0.1	0.3	0.1
T	0.23	0.3	0.1	0.5	0.3	0.3	0.1	0.5

5. Calculate weight for each possible motif position (offset) in the chosen sequence. The weight is calculated using the following formula, for sequence $S =$ `ACCATGACAG`. In the formula, k is the length of the k-mer.

$$w_i = \frac{\prod_{j=0}^{k-1} q_{S[i+j],j+1}}{\prod_{j=0}^{k-1} q_{S[i+j],0}}$$

   Normalize w_i so that $\mathbf{w}$ is a probability distribution. This gives us the probability that the k-mer starting at position i is generated by the profile.

6. Randomly draw a sample from the distribution $\mathbf{w}$ and update the motif position of the chosen sequence to the newly sampled position.

To find the best motif position, we repeat the steps described above until the resulting motif position converges. For biased samples, Gibbs sampling can be modified to work with relative entropies instead of frequencies.

10.5 Hidden Markov Model

The previous few sections focused on using Markov chains for sampling, but they are equally as useful for optimization problems. Suppose we are given a set of sequence with annotation that tells us which region of each sequence is coding-region and which is not. We also have a new sequence without annotation. We would like to find an optimal segmentation of this novel sequence and annotate it (assign regions of this new sequence as either coding or non-coding regions). We can model the annotated set using a Markov chain. We omit the construction of this type of Markov chain, but we refer interested readers to the ***Baum-Welch training algorithm***. In subsequent discussion, we assume that the trained Markov model is given to us.

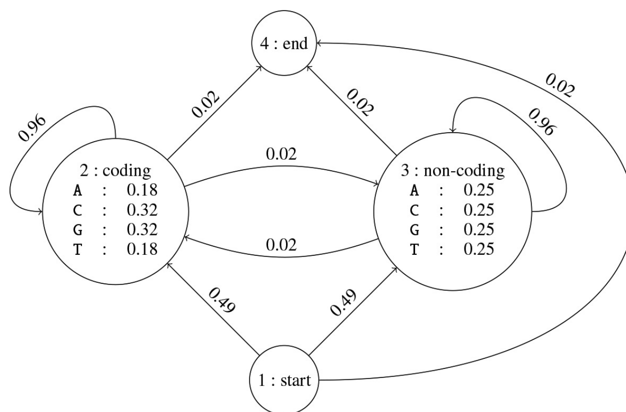


Figure 10.2: A trained Markov model that distinguishes coding and non-coding region by their GC-content.

The segmentation of a novel, unannotated sequence can be thought as a Markov chain in the trained model. This type of model is called the ***Hidden Markov Model*** (HMM). Unlike the Markov chains that we constructed earlier, in HMM, we don't know the sequence of states of our Markov chain. We only know the emitted string.

Formally,

Definition 10.18 (Hidden Markov Model)**.** *A **hidden Markov model** is a tuple $(H, \Sigma, T, E, \mathbb{P})$ where H is the set of hidden states, Σ is the set of symbols, $T \subseteq H \times H$ is the set of transitions, $E \subseteq H \times \Sigma$ is the set of emissions, and $\mathbb{P}$ is the probability function for elements of T and E, satisfying the following conditions:*

- *There is a single start state $h_{\text{start}} \in H$ with no incoming transitions and no emissions;*
- *There is a single end state $h_{\text{end}} \in H$ with no outgoing transitions and no emissions;*
- *Let $\mathbb{P}(h \mid h') = P_{h',h}$ denote the probability for the transition $(h', h) \in T$ and $\mathbb{P}(c \mid h)$ be the probability of an emission $(h, c) \in E$. It must hold that*

$$\sum_{h \in H} \mathbb{P}(h \mid h') = 1 \qquad \forall h' \in H \setminus \{q_{\text{end}}\}$$

and

$$\sum_{h \in H} \mathbb{P}(c \mid h) = 1 \qquad \forall h \in H \setminus \{q_{\text{end}}, q_{\text{end}}\}.$$

A path (chain) through an HMM is a sequence P of hidden states $P = p_0 p_1 p_2 \dots p_n p_{n+1}$ where $(p_i, p_{i+1}) \in T$.

The joint probability of P and a sequence $S = s_1 s_2 \dots s_n \in \Sigma^n$ is

$$\mathbb{P}(P, S) = \prod_{i=0}^{n} \mathbb{P}(p_{i+1} \mid p_i) \prod_{i=1}^{n} \mathbb{P}(s_i \mid p_i).$$

And our problem of finding an optimal segmentation can be reduced to finding an optimal path in the hidden Markov model.

Problem 10.19. Given an HMM M over alphabet Σ and a sequence $S = s_1 s_2 \dots s_n$ with each $s_i \in \Sigma$, find the path P^* in M having the highest probability of generating S, namely,

$$P^* = \arg\max_{P \in \mathcal{P}(n)} \mathbb{P}(P, S) = \arg\max_{P \in \mathcal{P}} \prod_{i=0}^{n} \mathbb{P}(p_{i+1} \mid p_i) \prod_{i=1}^{n} \mathbb{P}(s_i \mid p_i).$$

To simpify our notation a little bit, we give the following definition where we ignore the first and the last transitions, respectively. For path $P = p_0 p_1 \dots p_n$ through the HMM, we define

$$\mathbb{P}_{\text{prefix}}(P, S) = \prod_{i=0}^{n-1} \mathbb{P}(p_{i+1} \mid p_i) \prod_{i=1}^{n} \mathbb{P}(s_i \mid p_i).$$

Similarly, for path $P = p_1 \dots p_n p_{n+1}$, define

$$\mathbb{P}_{\text{suffix}}(P, S) = \prod_{i=1}^{n} \mathbb{P}(p_{i+1} \mid p_i) \prod_{i=1}^{n} \mathbb{P}(s_i \mid p_i).$$

10.6 The Viterbi Algorithm

The Vertibi algorithm solves the problem of finding the most probable path in an HMM. It is analogous to the Bellman-Ford algorithm for finding the shortest weighted path in a graph. For every $i \in [n]$ and $h \in H$,

$$v(i, h) = \max\{\mathbb{P}_{\text{prefix}}(P, s_1, \dots, s_i) : \ P = h_{\text{start}} p_1 \cdots p_{i-1} h\}.$$

$v(i, h)$ is the largest probability of a path starting from state h_{start} and ending in state h, given that the HMM generated the prefix $s_1 \cdots s_i$ of string S. v can be expressed equivalently as the following recurrence relation.

$$\begin{aligned} v(i, h) &= \max\{\mathbb{P}_{\text{prefix}}(h_{\text{start}} p_1 \cdots p_{i-1} h', s_1, \dots, s_{i-1}) \cdot \mathbb{P}(h \mid h') \cdot \mathbb{P}(s_i \mid h) : \ (h', h) \in T\} \\ &= \mathbb{P}(s_i \mid h) \cdot \max\{v(i-1, h) \cdot \mathbb{P}(h \mid h') : \ (h', h) \in T\}, \end{aligned}$$

where, by convention, $v(0, h_{\text{start}}) = 1$ and $v(0, h) = 0$ for all $h \neq h_{\text{start}}$. Finally, we need to find an optimal transition from the second-to-last state into the end state. The final solution is given by this equation.

$$\max_{P \in \mathcal{P}(n)} \mathbb{P}(P, S) = \max_{(h', h_{\text{end}}) \in T} \{v(n, h') \cdot \mathbb{P}(h_{\text{end}} \mid h')\}.$$

The value of v can be computed in $O(n|T|)$ time using a bottom-up DP approach. And to obtain the optimal path, one can store traceback pointers as we fill the DP array.

Bibliography

The proof of the fundamental theorem roughly follows the structure from CSC473 lecture notes by Aleksandar Nikolov at the University of Toronto. The Metropolis-Hastings algorithm is known to be first introduced in the paper Equation of state calculations by fast computing machines [21]. The discussion of Gibbs sampling is based on the book An Introduction to Bioinformatics Algorithms by Jones and Pevzner [11].

Chapter 11
Random Graph Theory

Definition 11.1 (Erdös-Rényi Random Graph)**.** Let $G(n, p)$ be a graph-valued random variable with vertices V and edges E such that $n = |V|$ and $p = \Pr(\{v_i, v_j\} \in E)$ for any $i \neq j$.

11.1 Bulk Properties of Random Graphs

Theorem 11.2. *Let $v_1 \in V$ be a vertex of a random graph $G(n, p)$. Let $\alpha \in (0, \sqrt{np})$. Then,*

$$\Pr(|(n-1)p - \deg(v_1)| \geq \alpha\sqrt{(n-1)p}) \leq 3e^{-\alpha^2/8}.$$

Proof. Note that

$$\deg(v_1) = \sum_{i=2}^{n} \mathbb{I}_{1,i}$$

where

$$\mathbb{I}_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

is an indicator random variable. Since the expected degree is just $(n-1)p$, the proof then follows from Chernoff bounds

$$\Pr[|\deg(v_1) - (n-1)p| \geq c(n-1)p] \leq 3e^{-(n-1)pc^2/8}.$$

The theorem follows by setting $c = \frac{a}{\sqrt{(n-1)p}}$. □

Corollary 11.3. *Suppose $\epsilon > 0$. If $p \geq \frac{9 \ln n}{(n-1)\epsilon^2}$, then with $1 - o(1)$ probability, for all i*

$$\deg(v_i) \in [(1-\epsilon)(n-1)p, (1+\epsilon)(n-1)p].$$

Proof. Let $\alpha = \epsilon\sqrt{(n-1)p}$ in the previous theorem. For a fixed i, the failure probability is $\leq 3e^{-\epsilon^2(n-1)p/8}$. By union bound, the failure probability for every i is $\leq 3ne^{-\epsilon^2(n-1)p/8}$. By assumption, $p \geq \frac{9 \ln n}{(n-1)\epsilon^2}$, so the failure probability for every i is upper bounded as follows.

$$\leq 3ne^{-\epsilon^2(n-1)p/8} \leq 3ne^{-9/8 \ln n} = 3n - n^{-9/8} = 3n^{-1/8} \in o(1).$$

□

This corollary essential tells us that if $p \in \Omega(\frac{\log n}{n})$, then with high probability, all vertices have tightly concentrated degree.

11.2 Structures in Random Graphs

Theorem 11.4. *For sufficiently large n, $G(n, \frac{d}{n})$ has in expectation, approximately $d^3/6$ triangles.*

We first give an intuitive justification of the theorem. Here, as n increases, the number of triples grows in order of n^3. But each pair of vertices has a d/n probability to be connected by an edge. So a pair of 3 vertices has an approximately d^3/n^3 probability to be adjacent to each other.

Proof. Let Δ_{ijk} be the indicator variable for the existence of a triangle with vertices $v_i, v_j, v_k \in V$. Then,

$$\begin{aligned}\mathbb{E}[\text{number of triangles}] &= \mathbb{E}\left[\sum_{i,j,k} \Delta_{ijk}\right] \\ &= \sum_{i,j,k} \mathbb{E}[\Delta_{ijk}] \\ &= \binom{n}{3}\left(\frac{d}{n}\right)^3 \\ &= \frac{n(n-1)(n-2)}{6} \cdot \frac{d^3}{n^3} \approx \frac{d^3}{6}.\end{aligned}$$

□

Let X be the random variable denoting the number of triangles. So $X = \sum_{i,j,k} \Delta_{ijk}$. So,

$$\mathbb{E}[X^2] = \mathbb{E}\left[\left(\sum_{i,j,k} \Delta_{ijk}\right)^2\right] = \mathbb{E}\left[\sum_{\substack{i,j,k \\ i',j',k'}} \Delta_{ijk}\Delta_{i'j'k'}\right].$$

But notice here, Δ_{ijk} and $\Delta_{i'j'k'}$ are not independent. We split the sum into 3 parts:

$$S_1 = \{i, j, k, i', j', k' \mid \Delta_{ijk} \text{ and } \Delta_{i'j'k'} \text{ share no edges}\}$$

$$S_2 = \{i, j, k, i', j', k' \mid \Delta_{ijk} \text{ and } \Delta_{i'j'k'} \text{ share exactly 1 edge}\}$$

$$S_3 = \{i, j, k, i', j', k' \mid \Delta_{ijk} = \Delta_{i'j'k'}\}.$$

Note for the last case, if two triangles share 2 or 3 edges, it implies that the two triangles are the same. So,

$$\mathbb{E}\left[\sum_{S_1} \Delta_{ijk}\Delta_{i'j'k'}\right] = \sum_{S_1} \mathbb{E}[\Delta_{ijk}]\,\mathbb{E}[\Delta_{i'j'k'}] \leq \left(\sum_{i,j,k'} \mathbb{E}[\Delta_{ijk}]\right) + \left(\sum_{i',j',k'} \mathbb{E}[\Delta_{i'j'k'}]\right).$$

The second case is tricky. There are $\binom{n}{4}$ ways to choose 4 vertices and $\binom{4}{2}$ ways to choose the two vertices forming the shared edge. Finally, we have p^5 probability that the remaining 5 edges are present to form two triangles. So,

$$\mathbb{E}\left[\sum_{S_2} \Delta_{ijk}\Delta_{i'j'k'}\right] = \binom{n}{4}\binom{4}{2}p^5 \approx \frac{n^4}{2^4} \cdot 6 \cdot p^5 = \frac{1}{4}n^4p^5 = \frac{1}{4}n^4\frac{d^5}{n^5} = \frac{1}{4}\frac{d^5}{n} \in o(1).$$

This tells us this case (two triangles sharing exactly one edge in a random graph) rarely happens.

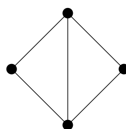


Figure 11.1: Two triangles sharing exactly one edge.

Finally, for the last case,

$$\mathbb{E}\left[\sum_{S_3} \Delta_{ijk}\Delta_{i'j'k'}\right] = \mathbb{E}\left[\sum_{S_3} \Delta_{ijk}\right] = \mathbb{E}[X].$$

Combining every case together, we have

$$\mathbb{E}[X^2] \leq \mathbb{E}[X]^2 + \mathbb{E}[X] + \epsilon$$

for some $\epsilon \in o(1)$. It follwos that

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2 \leq \mathbb{E}[X] + \epsilon$$

for $\epsilon \in o(1)$.

By Chebyshev's inequality, for sufficiently large n,

$$\Pr(X = 0) \leq \frac{\mathrm{Var}(X)}{\mathbb{E}[X]^2} \leq \frac{\mathbb{E}[X] + o(1)}{\mathbb{E}[X]^2} \leq \frac{6}{d^3} + o(1).$$

Note from the first section, for $p \geq \frac{9 \ln n}{(n-1)\epsilon^2}$, $\deg(v_i) \in [(1-\epsilon)(n-1)p,\ (1+\epsilon)(n-1)p]$ with high probaility. Then it follows that for a random graph $G(n, \frac{d}{n})$, the degree for each vertex will be near d with high probability. Then, for $d < \sqrt[3]{6}$, $\mathbb{E}[X] = d^3/6 < 1$, so in this case, there will not be many triangles. This should intuitively make sense because if most vertices have degree less than 2, then triangles should be rare in such graphs.

11.3 Phase Transitions in Random Graphs

Definition 11.5 (Phase Transition)**.** If there exists some function $p(n)$ such that $\lim_{n\to\infty} p_1(n)/p(n) = 0$ for some other function $p_1(n)$, $G(n, p_1(n))$ does not satisfy a property with high probability, but for another function p_2 where $\lim_{n\to\infty} p_2(n)/p(n) = \infty$, $G(n, p_2(n))$ satisfies the same property with high probability, then we say a ***phase transition*** occurs at the ***threshold*** $p(n)$.

Definition 11.6 (Sharp Threshold)**.** If for $cp(n)$ such that when $c < 1$, $G(n, cp(n))$ does not satisfy a property with high probability but for $c > 1$, $G(n, cp(n))$ satisfies the same property, then $p(n)$ is a ***sharp thershold***.

Below is a list of properties with phase transitions in an Erdös-Rényi random graph.

There are mainly two methods for analyzing phase transitions.

1. **First moment method**: Let $X(n)$ denote the occurrence of certain objects (structures, properties, etc.) in a random graph. If $\mathbb{E}[X(n)] \to 0$ as $n \to \infty$, then the graph almost surely has not occurrence of such object.

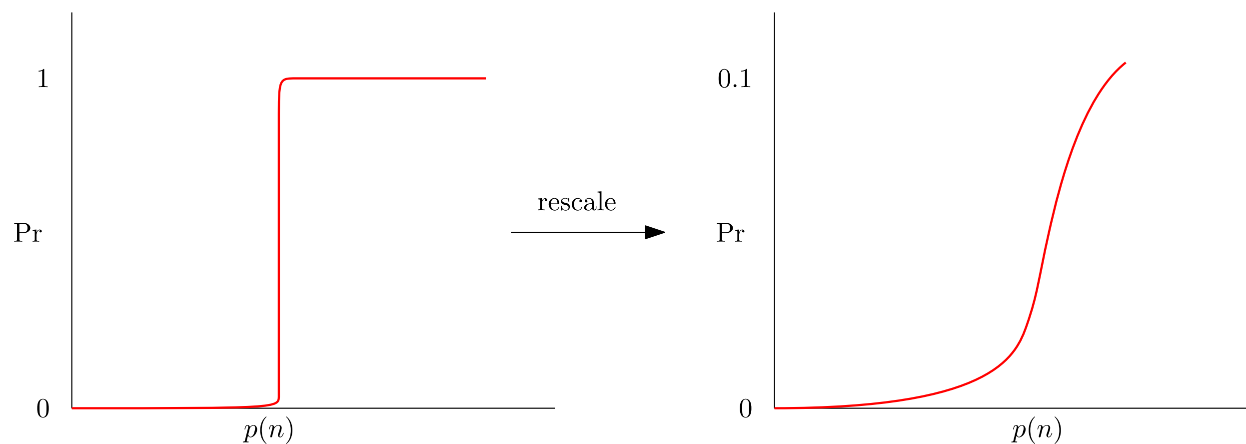


Figure 11.2: A graph illustration phase transition.

Probability	Property (Behavior)
$p \in o(\frac{1}{n})$	Graph is a forest of trees, component size bounded by $O(\log n)$
$p = \frac{d}{n}$, $d < 1$	Graph has some cycles, component size bounded by $O(\log n)$
$p = \frac{d}{n}$, $d = 1$	Component size bounded by $O(n^{2/3})$
$p = \frac{d}{n}$, $d > 1$	Giant component plus some components with size bounded by $O(\log n)$
$p = \frac{1}{2}\frac{\ln n}{n}$	Giant component plus isolated vertices
$p = \frac{\ln n}{n}$	No isolated vertices; existence of Hamiltonian circuits; graph diameter $O(\log n)$
$p = \sqrt{\frac{2 \ln n}{n}}$	Graph diameter is 2
$p = \frac{1}{2}$	Existence of clique of size $(2 - \epsilon) \ln n$

Proof. Since $X(n)$ is non-negative, we can apply Markov's inequality.

$$\Pr(X(n) \geq a) \leq \frac{\mathbb{E}[X(n)]}{a}$$

so if $\mathbb{E}[X(n)] \to 0$ as $n \to \infty$, then

$$\Pr(X(n) \geq 1) \leq \mathbb{E}[X(n)] \to 0.$$

□

2. **Second moment method**: Let $X(n)$ be a random variable with $\mathbb{E}[X(n)] > 0$. If $\mathrm{Var}(X) \in o(\mathbb{E}[X(n)]^2)$, then $X(n) > 0$ almost surely.

Proof. By Chebyshev,

$$\Pr(X \leq 0) \leq \Pr(|X - \mathbb{E}[X]| \geq \mathbb{E}[X]) \leq \frac{\mathrm{Var}(X)}{\mathbb{E}[X]^2} \to 0.$$

□

This is the technique used to prove the occurrence of triangles in random graphs.

11.4 Modeling Protein-Protein Interaction Network

Systems biology studies biological systems as a whole. This includes the study of interaction between proteins, genes, and the study of differential gene expression from data obtained via DNA microarray and RNASeq. Networks play a crucial role in arriving at and summing up the holistic picture and in understanding the emergent properties of the system. The volume of experimental data on protein-protein interactions is rapidly increasing thanks to high-throughput techniques which are able to produce large batches of PPIs. However, producing pairwise interaction data on a large set of potential interactors is often still infeasible both computationally and practically. This requires us to model the interaction network using a random graph model where the connectivity probability is trained to fit the experimental data.

Further, it is often hard to perform quantitative and qualitative analysis and comparisons on large-scale graphs and networks. In these cases, analyses are often performed on subgraphs and components of the graph. In this section, we will briefly discuss the ER random graph model and its application in PPI network as well as how the phase transition properties may affect our ability to compare PPI networks.

We first note that the probability of a given node in a random graph on n vertices having degree k is given by

$$\Pr(\deg(v) = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}.$$

If $n \gg kz$, the distribution becomes the Poisson distribution $p(k) = \frac{z^k e^{-x}}{k!}$ where z is the mean. As we have seen earlier, if $p \in \Omega(\frac{\log n}{n})$, then all vertices have tightly concentrated degree with high probability. Additionally, random graphs tend to have small diameters.

When using the ER random graph model to model PPI networks, we need two parameters n and p. n is the number of vertices, and p is calculated so that the expected number of the network equals to m.

Another important metric for a random graph is the cluster coefficient. Defined as follows:

$$C_i = \frac{|\{(v_j, v_k) \in E \mid v_j, v_k \in N(v_i)\}|}{\deg_-(v_i)(\deg_-(v_i) - 1)}$$

where $N(v_i)$ is the neighborhood of v_i and $\deg_-$ denotes the out-degree of a vertex. The overall clustering in a network can be mesured using the average clustering coefficient, defined as

$$\overline{C} = \frac{1}{n}\sum_{i=1}^{n} C_i.$$

In real-world networks like PPI networks, the clustering coefficient tends to be high, while Erdös-Rényi graphs do not. This is because in ER random graph, the probability of two vertices being adjacent is independent and the clustering coefficient for a vertex in an ER random graph is just p, the connection probability.

This is not the only issue with using ER graphs as models for PPI network. Due to the phase transition property of random graphs, it can be unreliable to compare networks using subgraphs. Commonly used metrics like the GDDA (Graphlet Degree Distribution Agreement) exhibits unstable behavior when the graph density is around the threshold region for the appearance of small subgraphs.

Bibliography

The last section of the chapter used idea and results from [22] while the rest of the material is based on Chapter 8 of [2].

Chapter 12
Hashing

12.1 Hash Functions

Intuitively, sometimes it could be helpful to randomly map numbers. Given a domain U, and a range $[m] = \{0, 1, \ldots, m-1\}$, a **truely random hash function** is a map $h : U \to [m]$ where each $h(x)$ is an i.i.d. uniform r.v. in $[m]$. h is a $|U|$-dimensional random variable chosen uniformly at random from $[m]^U$.

However, in practice, we need to store a hash function in order to use it again in the future. A truely random hash function is very expensive to store. For every element in the universe, we need to store the corresponding map under the hash function, taking $O(|U| \log_2 m)$ bits of space.

More generally,

Definition 12.1 (Hash Function)**.** *A* ***hash function*** $h : U \to [m]$ *is a random variable in the class of all functions* $U \to [m]$ *(not necessarily uniform).*

A trivial example of a hash function is the identity map. Another more practical example uses a prime field: Let $h : [p] \to [p]$ for a prime p given by $h(x) = (ax + b) \mod b$ where a, b are uniformly chosen in $[p]$.

We now formally define a desirable property of a hash function or family of hash functions. A fixed hash function can always suffer from bad worst-case performance against an adversary. In order to obtain provable results regarding hash functions, we need to introduce randomness and consider the expected performance.

Definition 12.2 (Universal Hash Family)**.** *A family* $\mathcal{H}$ *of hash functions* $h : U \to [m]$ *is* ***universal*** *if* $\forall x, y \in U.\, x \neq y$*, then*

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}.$$

Note that for truly random hash functions, this probability is equal to $\frac{1}{m}$. Furthermore, we say a family $\mathcal{H}$ of hash functions is c-***approximately universal*** if for all $x, y \in U$ such that $x \neq y$,

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{c}{m}$$

for some $c \in O(1)$.

If $|U| \leq m$, the identity function $h(x) = x$ is universal on $[|U|] \to [m]$. If $|U| \geq m$, the mod function $h(x) = x \mod m$ is not universal on $[|U|] \to [m]$ because 1 and $1 + m$ collide with probability 1.

12.2 Strong Universality (2-Independence)

Definition 12.3 (2-Independence)**.** *A random hash family* $\mathcal{H}$ *of* $h : U \to [m]$ *is* ***2-independent*** *or* ***strongly universal*** *if*

$$\Pr_{h \in \mathcal{H}}[h(i_1) = j_1 \wedge h(i_2) = j_2] = \frac{1}{m^2}$$

for all $i_1 \neq i_2$ *and* j_1, j_2*.*

Lemma 12.4. *Strong universality implies universality.*

Proof. Apply marginalization to sum over all possible choice of j_2. □

Theorem 12.5. *Strongly universality is equivalent to the statement that each key is hashed uniformly into $[m]$ and that every two keys are hashed independently.*

Proof.

$(\implies)$: Let $h: U \to [m]$ be strongly universal. Let $x \neq y \in U$. Clearly, for all $q \in [m]$,

$$\Pr_{h\in\mathcal{H}}[h(x) = q] = \sum_{r\in[m]} \Pr_{h\in\mathcal{H}}[h(x) = q \wedge h(y) = r] = \frac{m}{m^2} = \frac{1}{m}.$$

Hence, uniformity holds. Furthermore,

$$\Pr_{h\in\mathcal{H}}[h(x) = q \mid h(y) = r] = \frac{\Pr_{h\in\mathcal{H}}[h(x) = q \wedge h(y) = r]}{\Pr_{h\in\mathcal{H}}[h(y) = r]} = \frac{\frac{1}{m^2}}{\frac{1}{m}} = \frac{1}{m} = \Pr_{h\in\mathcal{H}}[h(x) = q]$$

so independence also holds.

$(\impliedby)$: If $h(x)$ and $h(y)$ are independent and uniform. Then,

$$\Pr[h(x) = q \wedge h(y) = r] = \Pr[h(x) = q] \cdot \Pr[h(y) = r] = \frac{1}{m^2}.$$

□

We can generalize the notion of **2-independence** to ***k*-independence**.

Definition 12.6 (k-independence)**.** *$\mathcal{H}$ is a k-independent family if for all distinct $i_1, i_2, \ldots, i_k \in U$ and for all $j_1, \ldots, j_k \in [m]$,*

$$\Pr_{h\in\mathcal{H}}[h(i_1) = j_1 \wedge \ldots \wedge h(i_k) = j_k] = \frac{1}{m^k}.$$

There are some trivial examples of k-independent hashing families.

Example 12.7. *The set $\mathcal{H}$ of all functions $[u] \to [m]$ is k-independent for all k. For this family, $|\mathcal{H}| = m^u$ so $h \in \mathcal{H}$ is representable in $u \lg m$ bits.*

Now we consider a non-trivial example of a k-independent hash family. Let $u = m = q$ where q is a prime power and $U = [u]$. Let $\mathcal{H}_{\textit{poly-k}}$ be the set of all polynomials of degree at most $k-1$ in $\mathbb{F}_q[x]$ (Galois field of order q).

Claim. $\mathcal{H}_{\textit{poly-k}}$ is k-independent.

Proof. If we know that $i_1, \ldots, i_k$ are distinct, then using the **Lagrange interpolation**, we have

$$p(x) = \sum_{r=1}^{k} \left(\frac{\prod_{y\in[k]\setminus\{r\}}(x - i_y)}{\prod_{y\in[k]\setminus\{r\}}(i_r - r_y)} \right) \cdot j_r$$

which satisfies $p(i_r) = j_r$ for all r. Note that $p(x)$ is a polynomial of degree $k-1$. Furthermore, $p(x)$ is the unique polynomial of degree of at most $k-1$, where $p(i_r) = j_r$. Thus,

$$p(x) = \alpha_{k-1}x^{k-1} + \cdots + \alpha_1 x + \alpha_0$$

so it is determined by k elements of $\mathbb{F}_q$. Then, $|\mathcal{H}_{poly\text{-}k}| = q^k$. It follows that

$$\Pr_{h \in \mathcal{H}_{poly\text{-}k}}[h(i_1) = j_1 \wedge \ldots \wedge h(i_k) = j_k] = \frac{1}{q^k}$$

since only one of the q^k polynomials of degree $k-1$ satisfies that $p(i_r) = j_r$ for all r. □

This hash family is easier to store and represent. Each $h \in \mathcal{H}_{poly\text{-}k}$ is representable using $k \lg q$ bits.

12.3 Finite Fields

Consider the Galois field $\mathbb{F}_q$ where $q = 2^w$ where w is the size of a word on a computer (usually 32 or 64). This turns out to be a reasonable choice for our hash family because 2^{64} is a prime power. Recall that $\mathbb{F}_{2^{64}} = \mathbb{F}_2[X]/(p)$ where p is an irreducible polynomial (i.e. cannot be factored) in $\mathbb{F}_2[X]$ of degree 64. An element $z \in \mathbb{F}_{2^{64}}$ can be written as $a_{n-1}x^{n-1} + \cdots a_1 x + a_0 \in \mathbb{F}_2[x]$ where each $a_i \in \{0,1\}$ and $(a_{n-1}, \ldots, a_1, a_0)$ can be encoded as a 64-bit binary number. Addition in this field is easy.

$$\begin{array}{cccc} & a_{n-1} & \cdots & a_0 \\ \text{XOR} & b_{n-1} & \cdots & b_0 \\ \hline & (a_{n-1} \oplus b_{n-1}) & \cdots & (a_0 \oplus b_0) \end{array}$$

However, multiplication requires Euclidean division by p, which is harder compared to some other choices of finite fields.

12.3.1 Prime Fields

Instead, we consider a better choice of a field for hashing. Let p be a large prime. Then $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ and multiplication is done modulo p. Recall that Mersenne primes are prime numbers of the form $2^n - 1$ which is relatively close to what we want (a power of 2).

Claim. If n is composite, so is $2^n - 1$.

Proof. Let $n = ab$. Then,

$$\begin{aligned} 2^{ab} - 1 &= (2^a - 1)(1 + 2^a + 2^{2a} + \cdots + 2^{(b-1)a}) \\ &= (2^b - 1)(1 + 2^b + 2^{2b} + \cdots + 2^{(a-1)b}). \end{aligned}$$

□

This tells us n being prime is a necessary condition for $2^n - 1$ to be prime. The following n's are valid Mersenne expononents so that $2^n - 1$ is prime: $n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127$ (OEIS A00043).

Furthermore, it turns out arithmetics is easy on a prime field.

Claim. If $p = 2^q - 1$ and p and q prime, then $x \equiv x \bmod 2^q + \left\lfloor \frac{x}{2^q} \right\rfloor \pmod{p}$.

Proof. Let $x = a2^q + b$ where $b < 2^q$. Then

$$\begin{aligned} x \bmod p &= (a \bmod p)(2^q \bmod q) + (b \bmod p) \\ &= (a \bmod p)(2^q \bmod 2^q - 1) + (b \bmod p) \\ &= (a + b) \bmod p. \end{aligned}$$

But $a = \left\lfloor \frac{x}{2^q} \right\rfloor$ is the upper bits and $b = x \bmod 2^q$ is the lower bits. □

It follows from this claim that this is easy to compute on a computer using bit shifts.

$$(x >> q) = \left\lfloor \frac{x}{2^q} \right\rfloor \qquad (x \& p) = x \bmod 2^q.$$

Then, $y = x \bmod p$ can be computed by

```
1  y = (x&p) + (x >> q)
2  if y ≥ p
3        y = y − p
```

12.4 Universal Hashing of Variable-Length Strings

Consider $x_0, x_1, \ldots, x_d$ where $x_i \in [u]$ and $U = [u]$. We would like to construct an approximately universal hash family to $[q]$. Let q be a prime number and consider the finite field $\mathbb{F}_q$. Let

$$p_{x_0,\ldots,x_d}(\alpha) = \sum_{i=0}^{d} x_i \alpha^i.$$

Let $h_a(x_0, \ldots, x_d) = p_{x_0,\ldots,x_d}(a)$ where $a \in \mathbb{F}_q$ uniformly drawn from the finite field.

Claim. If $y_0, \ldots, y_{d'}$ is some other string with $d' \leq d$, then

$$\Pr_{a \in \mathbb{F}_q}[h_a(x_0, \ldots, x_d) = h_a(y_0, \ldots, y_{d'})] \leq \frac{d}{q}.$$

Proof. Note that $h_a(x_0, \ldots, x_d) = h_a(y_0, \ldots, y_{d'})$ is equivalent to $p_{x_0,\ldots,x_d}(a) - p_{y_0,\ldots,y_{d'}}(a) = 0$. But then $p_{x_0,\ldots,x_d}(a) - p_{y_0,\ldots,y_{d'}}(a)$ is also a polynomial in $\mathbb{F}_q[x]$. By the fundamental theorem of algebra, the polynomial $p_{x_0,\ldots,x_d}(a) - p_{y_0,\ldots,y_{d'}}(a)$ has at most d distinct roots. So the probability that a random $a \in \mathbb{F}_q$ is the root is at most d/q. □

12.5 Applications of Hashing

Hashing has a wide range of applications in computer science and computational biology. It is used in the construction of many data structures to achieve good expected runtime as well as cryptographic algorithms that hash input string into a code or signature satisfying certain desirable cryptographic properties. Meanwhle, locality sensitive hashing is used to map data points that are close to each other (with respect to some distance function) in applications like data clustering and dimensionality reduction.

12.5.1 Hash Tables

Let S be a subset of the universe U where $|S| = n \leq m$. Consider the hash table constructed using m buckets with a randomly chosen universal hash function.

Now we use the definition of universality to prove that universal hash families perform well when used for a hash table. For each hash table bucket i, let S_i denote the set of all items $x \in S$ with $h(x) = i$. The

average-case performance of an insertion is the expected length of the chaining linked list at the bucket x hashed to.

Assume that h is chosen from a universal hash family, and that on query $h(x) = i$ such that $x \notin S_i$. Let $\mathbb{I}_x(y)$ be the indicator random variable that is equal to 1 when $h(x) = h(y)$. Then, by linearity of expectation and universality of $\mathcal{H}$,

$$\mathop{\mathbb{E}}_{h\in\mathcal{H}}\left[\sum_{y\in S_i}\mathbb{I}_x(y)\right] = \sum_{y\in S_i}\mathop{\mathbb{E}}_{h\in\mathcal{H}}[\mathbb{I}_x(y)] = \sum_{y\in S_i}\Pr_{h\in\mathcal{H}}[h(x)=h(y)] \le \frac{n}{m} \le 1.$$

That is, the average time complexity for insertion when using a universal hash family is $O(1)$.

12.5.2 Checksums

Suppose that Alice wants to send a file F to Bob. Bob receives F'. We would like to know if Bob received the correct file as sent by Alice. The idea of checksum using hashing is to check if $h(F) = h(F')$ for some hash function h. Ideally, if $h(F) = h(F')$, we can declare with high probability that $F = F'$.

Problem 12.8. *Assign a unique signature $s(x)$ for all $x \in S$ such that $|S| = n$. We want $s(x) \neq s(y)$ for all $x, y \in S$ where $x \neq y$.*

We can start by choosing a universal hash function $s : U \to [m]$. We need to know how large m needs to be in order to satisfy our requirements with high enough probability.

$$\begin{aligned}\Pr_{s\in\mathcal{H}}[\exists x,y\in S:\ s(x)=s(y),\ x\neq y] &\le \sum_{\substack{\{x,y\}\in S\\ x\neq y}}\Pr[s(x)=s(y)]\\ &\le \frac{\binom{n}{2}}{m}.\end{aligned}$$

Therefore, by choosing $m \ge n^2$, we can guarantee a $1 - \frac{1}{2n}$ probability of no collision.

12.5.3 Bloom Filters

A Bloom filter is a data structure that is similar to a hash table but for checking set membership. It is commonly used in genome assembly algorithms. It supports the following two operations:

1. Insert(x): insert x to the Bloom filter;
2. Check(x): returns true iff x is in the Bloom filter.

We would like both operations to be in constant time. We begin with a simple implementation: a bit vector hash table. Let h be a hash function from a universal hash family $\mathcal{H}$. Insert can be implemented very easily, and same for Check.

```
Insert(T, x)
1  T[h(x)] = 1
```

```
Check(T, x)
1  if T[h(x)] == 1
2      return True
3  else return False
```

Of course, an important issue with this is that there is a chance that CHECK will return the incorrect result. In particular, it is possible that x is not in the Bloom filter but its hash value collides with some other element, say y, that is in the Bloom filter. We can boost the success probability by making k copies of the Bloom filter, each with its own hash function h_i, sampled from a universal family. The implementation only needs to be slightly modified.

```
INSERT(T, x)
1  for i = 1 to k
2      T_i[h_i(x)] = 1
```

```
CHECK(T, x)
1  for i = 1 to k
2      if T_i(h_i(x)) ≠ 1
3          return False
4  return True
```

Assuming universality, the collision probability is at most $\frac{1}{m}$ where m is the size of each Bloom filter. The probability that a given bit is 0 after m elements are inserted is

$$\left(1 - \frac{1}{m}\right)^{nk}.$$

Then, the probability of a false positive is

$$\Pr(FP) = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k} = \left(1 - e^{\ln(1-\frac{1}{m})nk}\right)^{k}.$$

This is minimized when we take $k = \ln 2 \cdot \frac{m}{n}$.

12.6 Locality Sensitive Hashing

Definition 12.9 (Locality Sensitive Hash Family)**.** *A family $\mathcal{H}$ of hash functions is said to be (d_1, d_2, p_1, p_2)-sensitive with respect to some distance function $d(\cdot,\cdot)$ if for any $p, q \in P$ and any $h \in \mathcal{H}$,*

- *If $d(p, q) \leq d_1$, then $h(p) = h(q)$ with probability at least p_1*
- *If $d(p, q) \geq d_2$, then $h(p) = h(q)$ with probability at most p_2.*

It can also be defined equivalently for similarity $s(\cdot,\cdot)$.

- *If $s(p, q) \geq s_1$, then $h(p) = h(q)$ with probability at least p_1*
- *If $s(p, q) \leq s_2$, then $h(p) = h(q)$ with probability at most p_2.*

12.6.1 Hamming Distance and Nearest Neighbor Search

In this subsection, we will consider the locality sensitive hash family for Hamming distance and how to use this to implement a probabilistic data structure that allows for fast query of nearest neighbors.

Definition 12.10 (Hamming Distance)**.** *Let $\Sigma = \{0, 1 \ldots, k-1\}$ be an alphabet, and let $x, y \in \Sigma^d$. Then, the Hamming distance between x and y, $d_H(x, y)$ is defined as*

$$d_H(x, y) = |\{i \mid x_i \neq y_i\}|.$$

For any $i \in [d]$, $g: \{0,1\}^d \to \{0,1\}$ is defined by $g_i(x) = x_i$. Suppose i is picked from $[d]$ uniformly at random. Then,

$$\Pr_{i \sim [d]}(g_i(x) = g_i(y)) = \frac{|\{i \mid x_i = y_i\}|}{d} = 1 - \frac{|\{i \mid x_i \neq y_i\}|}{d} = \frac{1 - d_H(x,y)}{d}$$

And we can bound the collision probability:

1. If $d_H(x,y) \leq r$,
$$\Pr(g_i(x) = g_i(y)) \geq 1 - \frac{r}{d} = p_i$$

2. If $d_H(x,y) \geq Cr$,
$$\Pr(g_i(x) = g_i(y)) \leq 1 - \frac{Cr}{d} = p_2$$

We can define buckets using this LSH as follows.

$$\{x \in \{0,1\}^d \mid g_i(x) = 0\} \qquad \text{and} \qquad \{x \in \{0,1\}^d \mid g_i(x) = 1\}.$$

We can further amplify the probability gap by sampling more coordinates from the input point. For a sequence of indices from $I = (i_1, \ldots, i_k)$ from $[d]$, g_I is defined by

$$g_I(x) = (x_{i_1}, x_{i_2}, \ldots, x_{i_k}).$$

Here, k is a parameter to be decided later.

Example 12.11. *For example, given $x = (1,0,0,1,1,1,0)$ and $I = (3,1,7)$, we have $g_I(x) = (0,1,0)$.*

For I picked uniformly and independently from $[d]$,

$$\begin{aligned} \Pr(g_I(x) = g_I(y)) &= \Pr(x_{i_1} = y_{i_1}, \ldots, x_{i_k} = x_{i_k}) \\ &= \Pr(x_{i_1} = y_{i_1}) \cdots \Pr(x_{i_k} = y_{i,k}) \quad \text{independence} \\ &= \left(1 - \frac{d_H(x,y)}{d}\right)^k. \end{aligned}$$

so

1. If $d_H(x,y) \leq r$,
$$\Pr(g_I(x) = g_I(y)) \geq \left(1 - \frac{r}{d}\right)^k = p_i^k$$

2. If $d_H(x,y) \geq Cr$,
$$\Pr(g_i(x) = g_i(y)) \leq \left(1 - \frac{Cr}{d}\right)^k = p_2^k$$

We can then construct a data structure for near neighbor query using a two-level hashing scheme. The data structure consists of the following:

- L hash tables $T_1, \ldots, T_L$ with $m \geq n$ slots each
- L regular hash functions $h_1, \ldots, h_L : \{0,1\}^k \to [m]$, sampled from a universal family
- L locality sensitive hash function $g_{I_1}, \ldots, g_{I_L} : \{0,1\}^d \to \{0,1\}^k$.

Searching for near neighbor in this data structure can be done using the procedure describe in the pseudocode.

```
NEAR-NEIGHBOR(P, q)
 1  num-checked = 0
 2  for l = 1 to L
 3      i = h_l(g_{I_l}(q))
 4      x = T_l[i].head
 5      while x ≠ NIL
 6          if d(q, x) ≤ Cr
 7              return x
 8          num-checked = num-checked + 1
 9          if num-checked == 12L + 1
10              return FAIL
11          else
12              x = x.next
13  return FAIL
```

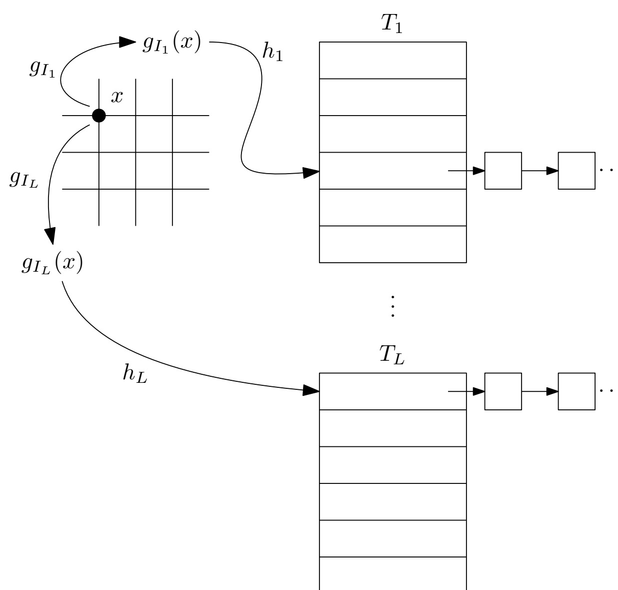


Figure 12.1: A point is first hashed with the locality sensitive hash function g before hashed a second time with h to determine the position in the hash table to insert. Resolve collision using chaining.

Locality sensitive hashing for Hamming distance and Euclidean distance by themselves are useful in applications such as nearest neighbor search and data clustering. However, there are also locality sensitive hash functions designed for distance metrics that are commonly used in biology to measure the similarity between sequences and gene sets. In the next few subsections, we will briefly talk about the techniques used to construct and analyze locality sensitive hash functions for other non-Euclidean distance metrics like the Jaccard and the edit distance.

12.6.2 Jaccard Index

Definition 12.12 (Jaccard Similarity)**.** *Let $A, B \subseteq U$ be two subsets of the universe U. Let $n = |A \cup B|$. Then, the **Jaccard similarity** is defined as*

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Jaccard similarity is a similarity measure for sets. It is used in bioinformatics to compare the similarity between, say two gene sets. For large sets, it is often intractable to compute the Jaccard index directly by taking the union and intersection of the sets as it would likely require storing the sets explicitly. We now describe a procedure known as ***Min Hash*** for estimating the Jaccard similarity in the streaming model. The goal of Min Hash is to compute the Jaccard index efficiently without explicitly computing the intersection and union.

Min-Hash-Jaccard(A, B)

1 $\mathcal{H} = \{\text{random hash function } h_i : U \to [q] \mid i \in [k]\}$
2 **for** $i = 1$ **to** k
3   **if** $\min_{a \in A} h_i(a) == \min_{b \in B} h_i(b)$
4         $\delta_i = 1$
5   **else**
6         $\delta_i = 0$
7 **return** $\widehat{J} = \frac{1}{k} \sum_{i=1}^{k} \delta_i$

Claim. If $q \geq \frac{kn^2}{\delta}$ and $k > \frac{2}{\epsilon^2 \delta}$, then

$$\Pr(|\widehat{J} - J| > \epsilon) < \delta.$$

This requires $O\left(\frac{1}{\epsilon^2 \delta} \log \frac{kn^2}{\delta}\right)$ space.

Proof. Recall that if h_i is a universal hash function, then all of $h_i(x)$ for $x \in A \cup B$ are distinct with probability at least $1 - \delta/2k$.

By the union bound, all the h_i's have no collisions with probability at least $1 - \delta/2$. Then, with probability at least $1 - \delta/2$, $h_i(a) = h_i(b)$ only if $a = b$. This implies

$$\min_{a \in A} h_i(a) = \min_{b \in B}(h_i(b)) \implies a = b.$$

Clearly, the converse is also true. Thus, $\mathbb{E}[\delta_i] = J$ and $\mathbb{E}[\widehat{J}] = J$. Further, $\mathrm{Var}(\delta_i) \leq J$. Then, by Chebyshev and the fact that $J \leq 1$,

$$\Pr(|\widehat{J} - J| \geq \epsilon) \leq \frac{\mathrm{Var}(\widehat{J})}{\epsilon^2} \leq \frac{\frac{J}{k}}{\epsilon^2} < \frac{\delta J}{2} \leq \frac{\delta}{2} \leq \delta.$$

□

This gives us an algorithm that gives us an estimate for the Jaccard index on expectation. It again used the non-trivial assumption that we have oracle access to a set of random hash functions. It turns out that k-independence is not sufficient for this purpose, and we need a ***minwise hash function***.

Definition 12.13 (Minwise Hashing)**.** *Given a random permutation π of the universe U, the **minwise hash function** is defined as*

$$h_\pi(S) = \min_{x \in S}(\pi(x)).$$

This is closely related to the previous algorithm that computes the Jaccard. When we take a random hash function and assume it has no collision, it is essentially a permutation of its domain. And taking the minimum hash value over all elements is really just the same as finding the smallest element in a random permutation. We will show that minwise hashing (a.k.a. MinHash) is *locality sensitive for Jaccard similarity.*

Proposition 12.14. *For all sets $S_1, S_2 \subseteq U$,* $\Pr(h_\pi(S_1) = h_\pi(S_2)) = J(S_1, S_2)$.

Proof. Let t be the element in $S_1 \cup S_2$ with the smallest hash value so

$$t = \operatorname*{argmin}_{i \in S_1 \cup S_2} h_\pi(i).$$

Then, S_1 and S_2 have the same hash value if and only if $t \in S_1 \cap S_2$. Since π is a random permutation, every element is equally likely to be t. Thus,

$$\Pr(h_\pi(S_1) = h_\pi(S_2)) = \Pr(t \in S_1 \cap S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}.$$

□

Example 12.15. Let $U = \{0, 1, 2, 3, 4\}$ with $S_1 = \{0, 3, 4\}$ and $S_2 = \{1, 2, 3\}$. Let $\pi = [3, 2, 0, 4, 1]$. That is, π maps $0, 1, 2, 3, 4$ to $3, 2, 0, 4, 1$, respectively. Then,

$$\pi(S_1) = \pi(\{0, 3, 4\}) = \{3, 4, 1\} \quad \Longrightarrow \quad h_\pi(S_1) = 1$$

$$\pi(S_1) = \pi(\{1, 2, 3\}) = \{2, 0, 4\} \quad \Longrightarrow \quad h_\pi(S_2) = 0.$$

In the next chapter, we will look at more applications of sketching algorithms and how to further improve the space complexity of our Jaccard estimator.

12.6.3 Edit Distance

Edit distance is a similarity/dissimilarity measure for strings. In essence, edit distance counts the number of edits (substitution, insertion, deletion) required to transform one string to another. In this section, we will present the high-level idea behind the locality sensitive hash function for edit distance but will not go into detail about the more rigorous analysis of our construction.

Issues With Jaccard and MinHash

Given a sequence A, let $\mathcal{K}(A)$ be the set of k-mers. You can read more on k-mer counting and other distance metrics that use k-mer sets in Chapter 9. One might attempt to use the Jaccard index of the k-mer sets of the two sequences as a proxy to approximate the edit distance. More specifically, we define the Jaccard distance between two sets as

$$d_J(S_1, S_2) = 1 - \frac{A \cap B}{A \cup B}.$$

Then, we can analogously define the Jaccard distance of two sequences A and B as the Jaccard distance between their corresponding k-mer sets.

$$d_J(A, B) = d_J(\mathcal{K}(A), \mathcal{K}(B)).$$

Indeed, if the edit distance is low, the Jaccard distance is also low. However, high edit distance does not necessarily imply high Jaccard distance because Jaccard distance ignored k-mer repetition. Say for example, let

$$A = \overbrace{\texttt{AAAAAAAAAAAAAAA}}^{n-k}\overbrace{\texttt{CCCCC}}^{k}$$
$$B = \underbrace{\texttt{AAAAA}}_{k}\underbrace{\texttt{CCCCCCCCCCCCCCC}}_{n-k}.$$

Clearly, the edit distance is high, whereas the Jaccard distance is 0 because they produce the same k-mer sets: {AAAAA, AAAAC, AAACC, AACCC, ACCCC, CCCCC}. To avoid this issue, one may consider using weighted Jaccard and multisets, defined as

$$J^w(A, B) = \frac{|A \cap B|}{|A| + |B|} \qquad \text{where } A \text{ and } B \text{ are multisets.}$$

However, we notice that although weighted Jaccard does not suffer from repetitive k-mers, it ignores the relative order of the k-mers. Therefore, we can conclude from our two observations that Jaccard distance is insensitive to either k-mer reptitions (e.g. repeated sequences, indels), relative positions of k-mers (e.g. translocation), or both.

We will use a modified version of MinHash to construct a locality sensitive hashing scheme for the edit distance. To rephrase it in LSH terminology, MinHash is a family of hash functions $\mathcal{H}_{\min}$ where

$$\mathcal{H}_{\min} = \left\{ h_\pi(A) = \min_{x \in A} \pi(x) : \ \pi \text{ is a permutation of } X \right\}.$$

$\mathcal{H}_{\min}$ is (s, s, s, s)-sensitive for any $s \in [0, 1]$ with respect to the Jaccard distance because $\Pr_{h \in \mathcal{H}_{\min}}(h(A) = h(B)) = J(A, B)$.

Order Min Hash (OMH)

We now introduce a locality-sensitive scheme for edit distance, built upon the idea of MinHash but with modifications to address the shortcomings discussed earlier. Similar to MinHash, a multiset of k-mers are selected at random using a random permutation, but additionally, we subsample l of the k-mers and record their relative order in the sequence. Moreover, the method handles repeated k-mers by appending to each k-mer in the multiset the number of times it has occurred so far in the sequence. Doing this gives us a unique representation of k-mers in each sequence.

For a string S of length $|S| = n$, consider the set $\mathcal{M}^w_k(S)$ of pairs of the k-mers and their occurrence number. If there are x copies of m in the sequence S, then the x pairs of $(m, 0), \ldots, (m, x-1)$ are in the set $\mathcal{M}^w_k(S)$. The ***occurrence number*** of m denotes the number of other copies of m left to this particular copy. That is, the occurrence number of m is

$$|\{j \in [i] \mid S[j : k] = m\}|.$$

A permutation π of $\Sigma^k \times [n]$ defines two function $h^w_{\ell,\pi}$ and $h_{\ell,\pi}$ where $h_{\ell,\pi}(S) = ((m_1, o_1), \ldots, (m_\ell, o_\ell))$ is a vector of length ℓ of elements of $\mathcal{M}^w_k(S)$ such that:

- the pairs (m_i, o_i) are the ℓ smallest elements of $\mathcal{M}^w_k(S)$ according to π;
- the pairs are listed in the vecotr in the order in which the k-mer appears in the sequence S. That is, if $i < j$, $m_i = S[x : k]$ and $m_j = S[y : k]$, then $x < y$.

and $h_{\ell,\pi} = (m_1, \ldots, m_\ell)$ contains only the k-mers from $h^w_{\ell,\pi}(S)$, in the same order. Then, the Order MinHash (OMH) is the set of hash functions

$$\mathcal{H}_{k,\ell} = \{h_{\ell,\pi} \mid \pi \text{ is a permutation of } \Sigma^k \times [n]\}.$$

In the extreme case where $\ell = n - k + 1$, the vector contains overlapping k-mers that cover the entire sequence S. In that case, equality of the hash values implies strict equality of the sequences. On the other hand, if $\ell = 1$, the vectors contain only k-mer and no relative order information is preserved. In this case, the vectors only measure similarity between k-mer contents.

In the paper by Marçais et al., the authors proved the following theorem about OMH.

Theorem 12.16. *For any $\ell = [2, n-k]$ and any $1 > s_1 \geq s_2 > 0$, there exist functions $p^1_{n,k,\ell}$ and $p^2_{n,k,\ell}$ such that OMH is $(s_1, s_2, p^1_{n,k,\ell}(s_1), p^2_{n,k,\ell}(s_2))$-sensitive for the edit distance.*

We omit the proof of the theorem. Interested readers should read the original paper by Marçais et al. for the proof of the main theorem.

Bibliography

OMH was introduced in the paper [19], and the discussion is based on the talk by the authors for the same paper.

Chapter 13
Probabilistic Sampling and Sketching

Having introduced the prerequisite concepts of hashing and probabilistic analysis, we now look at an example where we utilize randomness to compute information about a large set of data without explicitly storing all the data. The main model that we will consider in this chapter is the streaming model where the inputs come in as a stream. The goal of probabilistic streaming and sketching algorithms is to compute properties of the stream while using only a small amount of memory. These techniques are commonly used in bioinformatics, especially in the study of space efficient genomic analysis.

13.1 Frequency Moments

First, we look at how to use sampling to compute the frequency moment of a stream. Frequency moment provides us with useful insights into our data, ranging from distinct elements to variance.

Consider a sequence $a_1, \ldots, a_n \in [m]$ where n and m are both large. For all $s \in [m]$, we call $f_s = |\{i \mid a_i = s\}|$ the frequency of s in the stream.

Definition 13.1. *For $p \in \mathbb{N}$, the pth* ***frequency moment*** *of the stream is*

$$F_p = \sum_{s=1}^{m} f_S^p.$$

For the purpose of this definition, we define $0^0 = 0$.

Remark: F_0 is the number of distinct elements; F_1 is the length of the stream; F_2 can be used to calculate the variance in the occurrence of elements.

$$\frac{1}{m}\sum_{s=1}^{m}\left(f_s - \frac{n}{m}\right)^2 = \frac{1}{m}\sum_{s=1}^{m}\left(f_s^2 - 2\frac{n}{m}f_s + \frac{n^2}{m^2}\right) = \left(\frac{1}{m}\sum_{s=1}^{m} f_s^2\right) - \frac{2n}{m^2}\sum_{s=1}^{m} f_s + \frac{n^2}{m^2}$$

which is equal to $\frac{F_2}{m} - \frac{n^2}{m^2}$.

Remark: The limit

$$\lim_{p\to\infty} F_p^{\frac{1}{p}} = \lim_{p\to\infty}\left(\sum_{s=1}^{m} f_s^p\right)^{\frac{1}{p}}$$

is equal to the frequency of the most frequent element(s).

The frequency moments can be very useful in bioinformatics. The most obvious and direct application is to estimate the number of distinct k-mers in a sequence or the distribution of k-mer frequencies.

13.2 Distinct Elements

In this section, we will cover sketching algorithms for computing the 0th frequency moment of a data set. Recall that the 0th moment is the number of distinct elements in the input stream. Because of this, the

algorithm is also referred to as count-distinct sketch. There are some simple and immediate algorithms for the distinct element problem:

- bit vector: $O(m)$ space
- list of items seen: $O(n \log m)$

13.2.1 A Lower Bound on Deterministic Algorithms

Although a simple problem, counting the number of distinct elements in an input stream is proven to be hard with limited memory space. This limits our ability to analyze large data sets with deterministic algorithms and will motivate the development of randomized algorithms.

Theorem 13.2. *Any exact deterministic algorithm solving the distinct element problem must use at least m bits of memory on some sequence of length $m+1$.*

Proof. Assume that some algorithm ALG uses less than m bits of memory on all such sequences. Recall that $|\mathcal{P}([m])| = 2^m$ and unique$\{a_1, \ldots, a_m\}$ can be any subset except the empty set $\emptyset$. Thus, there are $2^m - 1$ possible answers to the distinct element problem. However, we only have at most 2^{m-1} memory states. Then, by pigeonhole principle two different subsets $S_1, S_2 \in \mathcal{P}([m]) \setminus \emptyset$ where $S_1 \neq S_2$ must have the same memory states. The correctness of ALG implies that $|S_1| = |S_2|$ because otherwise ALG would be incorrect.

Let $b \in S_1$ and $b \notin S_2$. Then, $S_1 \cup \{b\} = S_1$ so $|S_1 \cup \{b\}| = |S_1|$ and $|S_2 \cup \{b\}| = |S_2| + 1$. Since ALG has the same memory state for S_1 and S_2, it should have the same memory state after adding b. However, this is not the case, which implies that ALG must be wrong on one of S_1 and S_2. □

13.2.2 Idealized Count-Distinct Sketch

Let $\sigma = a_1, \ldots, a_n$ be the stream of inputs, and let $d = F_0$ be the number of distinct elements in the stream. Consider the following algorithm:

```
Idealized-Distinct-Elements(σ)
1  h : [m] → [0,1] be a random hash function
2  z = 1
3  while σ is not empty
4        e = σ.Next()
5        z = min(z, h(e))
6  return 1/z − 1.
```

At the end of the loop, $z = \min_{i \in \sigma} h(a_i)$. We claim that the above algorithm gives us an estimation of the number of distinct elements. Let $\mathcal{S} = \text{unique}\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_d\}$. Since h is a random hash function and each element is hashed independently,

$$h(b_1), \ldots, h(b_d) = X_1, \ldots, X_d$$

are i.i.d. uniform r.v. over $[0, 1]$ and we can define the r.v. $Z = \min\{X_i\}_{i=1}^d$.

Lemma 13.3. *Let $X : \Omega \to [0, \infty)$ be a non-negative r.v. Then,*

$$\mathbb{E}[X] = \int_0^\infty \Pr(X > x) dx.$$

Next, we claim

$$\mathbb{E}[Z] = \frac{1}{d+1}.$$

Proof.

$$\begin{aligned}
\mathbb{E}[Z] &= \int_0^\infty \Pr(Z > \lambda) d\lambda \\
&= \int_0^1 \Pr(\forall i,\, X_i > \lambda) d\lambda \\
&= \int_0^1 \prod_{i=1}^{t} \Pr(X_i > \lambda) d\lambda \\
&= \int_0^1 (1-\lambda)^d d\lambda \\
&= \frac{1}{d+1}.
\end{aligned}$$

□

Thus, the expectation of our output value $1/z - 1$ is equal to $d = F_0$. To see how accurate our algorithm performs, we compute the variance of our result. After getting the variance, we can then bound the accuracy probability using Chebyshev's inequality.

Claim.

$$\mathbb{E}[Z^2] = \frac{2}{(d+1)(d+2)}.$$

Proof.

$$\begin{aligned}
\mathbb{E}[Z^2] &= \int_0^1 \Pr(Z^2 > \lambda) d\lambda \\
&= \int_0^1 \Pr(Z > \sqrt{\lambda}) d\lambda \\
&= \int_0^1 (1-\sqrt{\lambda})^d d\lambda \\
&= 2\int_0^1 u^t (u-1) du \qquad u = 1-\sqrt{\lambda} \\
&= \frac{2}{(d+1)(d+2)}.
\end{aligned}$$

□

Thus,

$$\mathrm{Var}(Z) = \mathbb{E}[Z^2] - (\mathbb{E}[Z])^2 = \frac{d}{(d+1)^2(d+2)} < \frac{1}{(d+1)^2}.$$

This algorithm is known as the idealized count-distinct sketch algorithm (ISA).

13.2.3 Averaging to Reduce Variance

We can further improve our estimate by repeating the ISA algorithm multiple times and taking the average.

1. run $q = \frac{1}{\epsilon^2 \eta}$ instances of the idealized count-distinct sketch algorithm in parallel
2. take the average over all the independent runs $\overline{z} = \frac{1}{q}\sum_{i=1}^{q} z_i$
3. output $1/\overline{z} - 1$.

We can calculate

$$\mathbb{E}[\overline{Z}] = \frac{1}{d+1} \qquad \text{and} \qquad \mathrm{Var}(\overline{Z}) = \frac{1}{q}\frac{d}{(d+1)^2(d+2)} < \frac{1}{q(d+1)^2}.$$

Note that for i.i.d. r.vs X_i where for each i, $\mathrm{Var}[X_i] = \sigma^2$, $\mathrm{Var}[\overline{X}] = \mathrm{Var}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n^2}\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}[X_i] = \frac{\sigma^2}{n}$.

By Chebyshev,

$$\Pr\left(|\overline{Z} - \frac{1}{d+1}| > \frac{\epsilon}{d+1}\right) < \frac{(d+1)^2}{\epsilon^2} \cdot \frac{1}{q(d+1)^2} = \eta.$$

Claim.

$$\Pr\left(\left|\left(\frac{1}{\overline{Z}} - 1\right) - d\right| > O(\epsilon d)\right) < \eta.$$

This is a non-trivial claim and needs to be proven because the variance can be different when we take the reciprocal of a random variable (e.g. $\overline{Z} \to 1/\overline{Z}$).

Proof. Recall

$$\Pr\left(|\overline{Z} - \frac{1}{d+1}| > \frac{\epsilon}{d+1}\right) < \eta.$$

The event $|\overline{Z} - \frac{1}{d+1}| > \frac{\epsilon}{d+1}$ is equivalent to $|d\overline{Z} + \overline{Z} - 1| > \epsilon$. Thus,

$$\Pr\left(|d\overline{Z} + \overline{Z} - 1| > \epsilon\right) < \eta.$$

This is, in turn, equivalent to

$$\Pr\left(\left|\frac{1}{\overline{Z}} - d - 1\right| > \frac{\epsilon}{|\overline{Z}|}\right) = \Pr\left(\left|d + 1 - \frac{1}{\overline{Z}}\right| > \frac{\epsilon}{|\overline{Z}|}\right) < \eta.$$

We know with probability $1-\eta$, $|\overline{Z}| \leq \frac{1+\epsilon}{d+1}$. Then, from the previous probability bound, we have

$$\Pr\left(\left|\frac{1}{\overline{Z}} - d - 1\right| > \frac{\epsilon(d+1)}{1+\epsilon}\right) < \eta.$$

Note that $\frac{\epsilon(d+1)}{1+\epsilon} = \epsilon(d+1)\cdot\frac{1}{1+\epsilon} \in O(\epsilon d)$ for small ϵ. So, with high probability, out estimate is within a factor $1 + O(\epsilon)$ of $1 + d$. □

The space complexity of the average algorithm is $O\left(\frac{1}{\epsilon^2\eta}\right)$, ignoring the space complexity of storing the random hash function. This, however, is a highly idealized assumption, and in practice, the space for storing a random hash function from $[m]$ to $[0, 1]$ is not trivial. Further, since the error parameters ϵ and η are in the denominator, the more accurate our estimate, the more space it will take.

13.2.4 Boosting Accuracy Using Median of Averages

1. instantiate $s = \left\lceil 36 \ln\left(\frac{2}{\delta}\right)\right\rceil$ independent instances of the average algorithm from the previous subsection with $q = 1/3$
2. output the median $\hat{d}$ of $\{1/\overline{z_j} - 1\}_{j=1}^{s}$ where $\overline{z}_j$ is the jth output of the average algorithm

Claim. For $s = \left\lceil 36 \ln\left(\frac{2}{\delta}\right)\right\rceil$,

$$\Pr\left(|\hat{d} - d| > \epsilon d\right) < \delta.$$

Proof. Let Y_j be the indicator variable for the event $|1/\overline{z_j} - d| > \epsilon d$. The median fails if at least half of the Y_j is 1. That is,

$$\sum_{j=1}^{s} Y_j > \frac{s}{2}.$$

Note

$$\Pr\left(\sum_{j=1}^{s} Y_j > \frac{s}{2}\right) = \Pr\left(\sum_{j=1}^{s} Y_j - \frac{s}{3} > \frac{s}{6}\right).$$

We will bound the probability on the RHS using Chernoff bound. First, let us simplify the problem by assuming the stronger assumption that $\mathbb{E}[Y_j] = \Pr(|1/\overline{z_j} - d| > \epsilon d) = 1/3$. Then, the RHS simplifies to

$$\Pr\left(\sum_{j=1}^{s} Y_j - \frac{s}{3} > \frac{s}{6}\right) = \Pr\left(\sum_{j=1}^{s} Y_j - \mathbb{E}\left[\sum_{j=1}^{s} Y_j\right] > \frac{1}{2}\,\mathbb{E}[Y_j]\right)$$

and by Chernoff ($\Pr(X - \mathbb{E}[X] \geq \delta\,\mathbb{E}[X]) \leq \exp(-\delta^2\,\mathbb{E}[X]/3)$ where X is the sum of n independent 0/1 r.v.s),

$$\Pr\left(\sum_{j=1}^{s} Y_j - \mathbb{E}\left[\sum_{j=1}^{s} Y_j\right] > \frac{1}{2}\,\mathbb{E}[Y_j]\right) < \exp\left(\frac{-(1/2)^2 \cdot s/3}{3}\right) < \delta.$$

□

The median of average algorithm has space complexity of $O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\right)$, again, ignoring the space for storing the random hash function.

13.2.5 Non-Idealized Distinct Element Sketch via Sampling

Recall that in last section, we discussed an idealized algorithm that solves the distinct element problem. Although having nice theoretical guarnatee, it makes a non-trivial assumption, which is the use of random hash functions that hash each element independently.

In this section, we will see how we can use 2-wise independent hash families and sampling to remove this assumption.

1. pick h from a 2-wise family $[n] \to [n]$ for some n that is a power of 2
2. maintain $X = \max_{a_i \in \sigma} \operatorname{lsb}(h(a_i))$ where lsb is the index of the least-significant bit of a number
3. output $\tilde{d} = 2^X$.

13.3 Second Frequency Moment Sketch

1. let $h : [m] \to \{-1, 1\}$ be a 4-wise independent hash function
2. let $X_s = h(s)$ be a Bernoulli r.v. with equal probability
3. output $a = \left(\sum_{i=1}^{n} h(a_i)\right)^2$

Lemma 13.4. $\mathbb{E}[a^2] \leq 3\,\mathbb{E}^2[a]$.

Proof. If any s, t, u, v are distinct, by 4-independence, the expectation is 0. So it suffices to consider only the cases where all 4 variables are the same or there are two pairs of variables.

$$\begin{aligned}
\mathbb{E}[a^2] = \mathbb{E}\left[\sum_{s=1}^{m} x_s f_s\right]^4 &= \mathbb{E}\left[\sum_{1\leq s,t,u,v\leq m} x_s x_t x_u x_v f_s f_t f_u f_v\right] \\
&= \binom{4}{2}\mathbb{E}\left[\sum_{s=1}^{m}\sum_{t=s+1}^{m} x_s^2 x_t^2 f_s^2 f_t^2\right] + \mathbb{E}\left[\sum_{s=1}^{m} x_s^4 f_s^4\right] \\
&= 6\sum_{s=1}^{m}\sum_{t=s+1}^{m} f_s^2 f_t^2 + \sum_{s=1}^{m} f_s^4 \\
&\leq 3\left(\sum_{s=1}^{m} f_s^2\right)^2 \\
&= 3\overset{2}{\mathbb{E}}[a].
\end{aligned}$$

□

13.4 Majority Element and Misra-Gries

13.4.1 Lower Bound on Deterministic Algorithms

Theorem 13.5. *Any deterministic streaming algorithm requires* $\Omega(\min(n, m))$ *space if we require the algorithm to output the majority element if there is one.*

Proof. Suppose n is even and the last $n/2$ elements are identical. Every possible set of unique $n/2$ first elements must have a different memory configuration. Otherwise, we can make the algorithm incorrect by choosing the second half to belong the one subset but not the other. If $n/2 \geq m$, then there are $2^m - 1$ subsets, which requires $\log(2^m - 1) \in \Omega(m)$ bits. If $n/2 \leq m$, then there are at least $\frac{m!}{(m-n/2)!}$ subsets, which requires $\log\left(\frac{m!}{(m-n/2)!}\right) \in \Omega(n)$ bits. □

13.4.2 Misra-Gries and Majority Algorithm

Problem (majority problem): Given a stream $\sigma = (i_1, \ldots, i_m)$ of updates in $[n] = \{1, \ldots, n\}$. If there exists an $i \in [n]$ such that more than half the updates in σ are equal to i, the algorithm should output i. If no such element exists, the algorithm outputs any arbitrary element.

The following algorithm by Boyer and Moore solves the problem using two words of memory.

```
MAJORITY(σ)
 1  element = i_1
 2  count = 1
 3  for t = 2 to m
 4      if element == i_t
 5          count = count + 1
 6      else
 7          if count > 0
 8              count = count − 1
 9          else
10              element = i_t
11              count = 1
12  return element
```

Intuitively, the algorithm keeps track of the head of a stack without actually storing the stack. Whenever the current element in the stream is not the stored majority element, we decrement the counter (decrementing the stack pointer, poping the top element). Whenever the current element in the stream is the majority element, we increment the counter (pushing the element into the stack). The correctness of the algorithm is captured by the following theorem.

Theorem 13.6. *If there exists an element i such that more than half of the updates in σ are equal to i, then* MAJORITY *outputs i. Moreover, at any point during the execution of the algorithm, $f_{element} \leq count + m/2$ where $f_{element}$ is the number of times element has occurred in the stream so far.*

Proof. The idea of the proof is to pair each update that leads to the decrement of *count* with a previous occurrence of the value currently stored as *element*. This formalizes our intuition that each encounter of an element that is not the stored majority element causes an element to be popped from the imaginary stack.

More specifically, we pair the updates as follows:

- Initialization: The first update i_1 is unpaired.

- Maintenance: If $i_t = element$, or $count = 0$, we leave i_t unpaired. If $i_t \neq element$ and $count > 0$, we pair i_t with some i_s where $s < t$ and $i_s = element$.

It can be shown using induction on t that at any time step t, *count* is equal to the number of unpaired updates equal to *element*. For all $j \neq element$, all updates equal to j are paired.

By our pairing procedure, for every pair (i_s, i_t), $i_s \neq i_t$. Since there are at most $m/2$ pairs, at most $m/2$ of the updates equal to i are paired. Letting f_i denote the number of occurrences of i so far, we have at least $f_i - m/2 > 0$ updates equal to i that are unpaired. *count* is equal to the $f_{element}$ minus the number of pairs (i_s, i_t) where $i_s = element$. If there is a majority element, i is returned when the algorithm terminates and we have $count \geq f_i - m/2$. If there is no majority element, the theorem holds trivially. □

We can generalize the majority element algorithm and obtain the *Misra-Gries algorithm* that finds all elements that appear in more than $1/k$ fraction of the updates.

```
Misra-Gries-Frequent(σ, k)
1  S =
2  for t = 1 to m
3      if ∃x ∈ S, x.element == i_t
4          x.count = x.count + 1
5      elseif |S| < k − 1
6          x = (element = i_t, count = 1)
7          S = S ∪ x
8      else
9          for x ∈ S
10             x.count = x.count − 1
11             if x.count == 0
12                 S = S \ {x}
13 return S
```

We have a similar theorem that proves the correctness of Misra-Gries.

Theorem 13.7. *The set S output by* Misra-Gries-Frequent *contains all $i \in [n]$ such that $f_i > m/k$. Moreover, for any $x \in S$, $f_{x.element} + m/k$.*

The proof is similar to the one for Majority with a few more cases for pairing. We omit the proof here.

13.5 CountMin Sketch

Recall the Bloom filter. Whenever a new element arrives, we insert it into the bit vector using k different hash functions. When we want to check for membership, we check if every k positions are all set. Bloom filter has a probability of returning a false positive.

Let $a_1, \ldots, a_n \in [m]$, and let $\mathbf{x} \in \mathbb{R}^m$ be the frequency vector containing the frequency of each item in $[m]$. We maintain a $t \times w$ matrix C. For each row, associate a hash function $h_j : [m] \to [w]$ from a 2-wise independent hash family. Insert i by incrementing all counters $C_{j,h_j(i)}$ for $j \in [t]$.

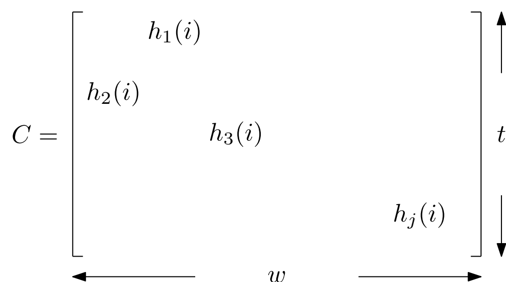


Figure 13.1: Example of a CountMin sketch matrix.

```
Insert(C)
1  for j = 1 to t
2      C[j][h_j(i)] = C[j][h_j(i)] + 1
```

We have another function Point-Query that outputs

$$\text{Point-Query}(i) = \min_{j \in [t]} \{C_{j,h_j(i)}\}.$$

Intuitively, taking the minimum should help balance out the potential over count resulted from hash collision. We have an overcount if some element other than the query point i, incremented some of the counters, but it may not have incremented all the same counters as i. This type of sketch is useful when estimating the k-mer counts in a large sequence. It provides more details about an individual set compared to Jaccard and MinHash which only gives us information about the set in relation to another set.

Theorem 13.8. *If $t > \lg(\frac{1}{\delta})$ and $w \geq \frac{2}{t}$, then*

$$\Pr\left(\text{Point-Query}(i) \in [x_i - \epsilon \|\mathbf{x}\|_1,\, x_i + \epsilon \|\mathbf{x}\|_1]\right) \geq 1 - \delta$$

where x_i is the true count of i.

Proof. For any $j \in [t]$,

$$C_{j,h_j(i)}(i) = x_i + \sum_{\substack{r \neq i \\ h_j(r) = h_j(i)}} x_r = x_i + \sum_{r \neq i} \delta_r x_r$$

where δ_r is the indicator variable $\mathbb{I}[h_j(r) = h_j(i)]$. Consider the expectation

$$\mathbb{E}\left[\sum_{r \neq i} \delta_r x_i\right] = \frac{1}{w} \sum_{r \neq i} x_r \leq \frac{\epsilon}{2} \|\mathbf{x}\|_1.$$

By Markov's inequality, since $x_i \geq 0$,

$$\Pr\left(\sum_{r \neq i} \delta_r x_r > \epsilon \|\mathbf{x}\|_1\right) \leq \frac{1}{2}.$$

Thus, $C_{j,h_j(i)}(i) \geq x_i$ and with probability more than $\frac{1}{2}$, $C_{j,h_j(i)}(i) \leq \epsilon \|\mathbf{x}\|_1$. Repeating this for t rows, we have

$$\Pr\left(\min_{j \in [t]} C_{j,h_j(i)} > x_i + \epsilon \|\mathbf{x}\|_1\right) \leq \frac{1}{2^t} < \delta.$$

□

Bibliography

[1] D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Algorithms–ESA 2013: 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings 21*, pages 133–144. Springer, 2013.

[2] A. Blum, J. Hopcroft, and R. Kannan. *Foundations of Data Science*. Cambridge University Press, jan 2020. doi: 10.1017/9781108755528. URL https://doi.org/10.1017%2F9781108755528.

[3] J. D. Buhler. *Search algorithms for biosequences using random projection*. PhD thesis, 2001.

[4] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *SRC Research Report, 124*, 1994.

[5] D. Clark. *Compact pat trees*. PhD thesis, 1997.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.

[8] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. 2003.

[9] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi: 10.1017/CBO9780511574931.

[10] G. J. Jacobson. *Succinct static data structures*. PhD thesis, 1988.

[11] N. C. Jones and P. A. Pevzner. *An introduction to bioinformatics algorithms*. MIT press, 2004.

[12] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[13] S. Kaur and M. Sukhjeet. Entropy coding and different coding technique. *Journal of Network Communication and Emerging Technologies*, 6:4–7, 2016.

[14] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings 14*, pages 186–199. Springer, 2003.

[15] L. G. Kraft. *A device for quantizing, grouping, and coding amplitude-modulated pulses*. PhD thesis, Massachusetts Institute of Technology, 1949.

[16] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, mar 1951. doi: 10.1214/aoms/1177729694. URL https://doi.org/10.1214%2Faoms%2F1177729694.

[17] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976. doi: 10.1109/TIT.1976.1055501.

[18] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, may 2015. doi: 10.1017/cbo9781139940023. URL https://doi.org/10.1017%2Fcbo9781139940023.

[19] G. Marçais, D. DeBlasio, P. Pandey, and C. Kingsford. Locality-sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

[20] B. McMillan. Two inequalities implied by unique decipherability. *IRE Transactions on Information Theory*, 2(4):115–116, 1956. doi: 10.1109/TIT.1956.1056818.

[21] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[22] T. Rito, Z. Wang, C. M. Deane, and G. Reinert. How threshold behaviour affects the use of subgraphs for network comparison. *Bioinformatics*, 26(18):i611–i617, sep 2010. doi: 10.1093/bioinformatics/btq386. URL https://doi.org/10.1093%2Fbioinformatics%2Fbtq386.

[23] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3): 379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[24] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013. ISBN 113318779X.

[25] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, sep 1995. doi: 10.1007/bf01206331. URL https://doi.org/10.1007%2Fbf01206331.

Index

2-independent, 119

aperiodic, 102
approximate near neighbor, 124

Bayes' rule, 23
bidirectional BWT index, 80
bijective, 18
binomial theorem, 18
bipartite graph, 11
Bloom filter, 123
Burrows-Wheeler transform, 71

Chebyshev's inequality, 26
checksum, 123
Chernoff bound, 28
chromatic number, 12
Clark's select, 76
codeword, 44
combination, 17
complete graph, 3
conditional expectation, 24
conditional probability, 23
connected component, 5
connected graph, 5
cosine similarity, 94
CountDistinct sketch, 131
counting sort, 59
CountMin sketch, 138
coupling, 104
coupling lemma, 105
covariance, 25
cycle graph, 3
cyclic shift, 71

DC3 algorithm, 65
degree, 4
directed graph, 3

edit distance, 128
empirical entropy, 44
entropy, 33
Erdös-Rényi random graph, 113
Eulerian circuit, 8
Eulerian graph, 8
expectation, 24

FM index, 74
frequency moment, 131
fundamental theorem of Markov chain, 103, 105

Galois field, 121
Gaussian Annulus Theorem, 91
Gibbs sampling, 108

Hall's theorem, 13
Hamiltonian graph, 9
Hamming distance, 124
Handshaking lemma, 4
hash function, 119
hashtable, 122
hidden Markov model (HMM), 110
Huffman's code, 46

independent, 24
independent set, 11
induced subgraph, 4
injective, 18
inverse BWT, 72
irreducible, 102
isomorphism, 4

Jaccard similarity, 127
Jacobson's rank, 75
Johnson-Lindenstrauss Lemma, 93

Kärkkäinen-Sanders' algorithm, 65
Kolmogorov complexity, 36
Kraft-McMillan inequality, 46
Kullback-Leibler divergence, 35

law of large numbers, 26
law of total expectation, 24
law of total probability, 23
law of total variance, 24
Lempel-Ziv complexity, 38
LF mapping, 72
locality sensitive hashing (LSH), 124

Markov chain, 101
Markov's inequality, 25
Master Tail Bounds Theorem, 27
median of averages, 135
Metropolis-Hastings algorithm, 107
MinHash, 127
Misra-Gries algorithm, 136
moment generating function, 28
MSD radix sort, 60

normal compression distance, 97
normal compressor, 97

order min hash (OMH), 129

path, 5
path graph, 3
permutation, 17
Perron-Frobenius theorem, 103
phase transition, 115
Pigeonhole Principle, 16
Poisson distribution, 117
poset, 15
prefix doubling algorithm, 63
prefix-free code, 45
Principle of Inclusion-Exclusion, 19
probability space, 23
proper coloring, 12
protein-protein interaction, 117

radix sort, 60
random projection, 92
random variable, 24

self-information, 33
sequence logo, 35
Shannon-Fano codes, 49
spanning subgraph, 7
spanning tree, 7
stationary distribution, 103
string kernel, 96
strong universality, 119
strongly connected, 5
subadditivity, 40
subgraph, 3
succinct suffix array, 79
suffix, 53
suffix array, 59
suffix link, 55
suffix trie, 53
surjective, 18
symbol code, 44

time-reversible, 108
total variation distance, 103
tree, 5
trie, 53

Ukkonen's algorithm, 55
undirected graph, 3
union bound, 23
uniquely decodable, 44
univeral hash family, 119
universality, 119

variance, 24
Viterbi algorithm, 111

walk, 5
wavelet tree, 77
weakly connected, 5
Well-Ordering Principle, 19
word RAM model, 58
worst-case entropy, 43