ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

CAO THỤC TUYẾT TRINH

NGHIÊN CỨU PHƯƠNG PHÁP NÉN DỮ LIỆU ĐỂ TĂNG HIỆU QUẢ LƯU TRỮ CHUỖI DNA

LUẬN VĂN THẠC SĨ HỆ THỐNG THÔNG TIN

HÀ NỘI – 2016

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

CAO THỤC TUYẾT TRINH

NGHIÊN CỨU PHƯƠNG PHÁP NÉN DỮ LIỆU ĐỂ TĂNG HIỆU QUẢ LƯU TRỮ CHUỖI DNA

Ngành: Hệ thống thông tin
Chuyên ngành: Hệ thống thông tin
Mã số: 60 48 01 04

LUẬN VĂN THẠC SĨ HỆ THỐNG THÔNG TIN

NGƯỜI HƯỚNG DẪN KHOA HỌC: Tiến sĩ Nguyễn Thị Hậu

HÀ NỘI – 2016

LỜI CAM ĐOAN

Tôi xin cam đoan nội dung của luận văn ***"Nghiên cứu phương pháp nén dữ liệu để tăng hiệu quả lưu trữ chuỗi DNA"*** là sản phẩm do tôi thực hiện dưới sự hướng dẫn của TS. Nguyễn Thị Hậu. Trong toàn bộ nội dung của luận văn, những điều được trình bày hoặc là của cá nhân hoặc là được tổnghợp từ nhiều nguồn tài liệu. Tất cả các tài liệu tham khảo đều có xuất xứ rõ ràng và được trích dẫn hợp pháp.

Tôi xin hoàn toàn chịu trách nhiệm và chịu mọi hình thức kỷ luật theo quy định cho lời cam đoan của mình.

Hà Nội, ngày 20 tháng 5 năm 2016

TÁC GIẢ

Cao Thục Tuyết Trinh

LỜI CẢM ƠN

Trước tiên tôi xin gửi lời cảm ơn chân thành tới tập thể các các thầy cô giáo trong Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội đã giúp đỡ tận tình và chu đáo để tôi có môi trường tốt học tập và nghiên cứu.

Đặc biệt, tôi xin bày tỏ lòng biết ơn sâu sắc tới TS. Nguyễn Thị Hậu, người trực tiếp đã hướng dẫn, chỉ bảo tôi tận tình trong suốt quá trình nghiên cứu và hoàn thiện luận văn này.

Một lần nữa tôi xin được gửi lời cảm ơn đến tất cả các thầy cô giáo, bạn bè và gia đình đã giúp đỡ tôi trong thời gian vừa qua. Tôi xin kính chúc các thầy cô giáo, các anh chị và các bạn mạnh khỏe và hạnh phúc.

Hà Nội, ngày 20 tháng 5 năm 2016

TÁC GIẢ

Cao Thục Tuyết Trinh

MỤC LỤC

DANH MỤC KÍ HIỆU VÀ CHỮ VIẾT TẮT

Kí hiệu	Tiếng Anh	Tiếng Việt
DNA	Deoxyribonucleic acid	Phân tử mang cấu trúc gen di truyền
NST	Chromosome	Nhiễm sắc thể
A	Adenine	
T	Thymine	
G	Guanine	
C	Cytosine	
SNP	Single nucleotide polymorphisms	Tính đa hình của phân tử nucleotit. Mỗi SNP biểu diễn một biến đổi trong một khối chuỗi DNA
CPU	Cental processing unit	Khối xử lý trung tâm
RAM	Random access memory	Bộ nhớ truy cập ngẫu nhiên
FRESCO	Framework for REferential Sequence Compresion	Khung nén tham chiếu FRESCO
2D	Differential Direct coding	Mã hóa trực tiếp phần khác biệt
XM	eXpert Markov	Thuật toán Markov
GRS	Genome ReSequencing	Thuật toán sắp xếp chuỗi gen GRS
RLZ	Relative Lempel-Ziv	Thuật toán Lempel Ziv RLZ
GDC	Genome Differential Compressor	Bộ nén chuỗi gen GDC
HTS	High – Throughput Sequencing	Sắp xếp chuỗi đa lượng

GIỚI THIỆU

Những tiến bộ kỹ thuật trong việc sắp xếp các chuỗi đa lượng (high-throughput sequencing) đã và đang tạo ra một khối lượng khổng lồ dữ liệu các chuỗi gen phục vụ cho y sinh học hiện đại. Kích thước dữ liệu ngày càng tăng đặt ra vấn đề về chi phí cho không gian lưu trữ và tốc độ truy cập, truyền tải.

Bộ gen của con người gồm khoảng 3 tỉ đặc trưng trên 23 cặp nhiễm sắc thể (NST). Cơ sở dữ liệu hệ gen là vô cùng lớn và phức tạp. Để lưu trữ, truy cập và xử lý dữ liệu này một cách hiệu quả là một nhiệm vụ rất khó khăn. Do vậy cần một thuật toán nén hiệu quả để lưu trữ khối lượng dữ liệu khổng lồ này. DNA (Deoxyribonucleic Acid) là tên hóa học chỉ các phân tử mang cấu trúc gen trong tất cả các thực thể sống. DNA gồm một chuỗi được tạo nên từ 4 loại đơn vị nucleotide, mỗi loại gồm: 1 đơn vị đường carbon 5 (2'-deoxyribose), 1 nhóm phốt phát (phosphate) và 1 trong 4 thành phần cơ bản adenine, cystosine, guanine và thymine gọi là các bazơ. Mỗi phân tử đường được gắn với ¼ thành phần cơ bản. Dạng đơn giản nhất của DNA trong 1 tế bào là 1 cấu trúc dây xoắn đôi, trong đó 2 sợi DNA đơn xoắn quanh nhau theo hình xoắn ốc thuận tay phải. Do chuỗi DNA gồm 4 thành phần A, T, G, C nên cách hiệu quả nhất để biểu diễn chúng là sử dụng 2 bits cho mỗi kí hiệu. Tuy nhiên, nếu ứng dụng phần mềm nén tiêu chuẩn như "Unix\compress and \compact" hoặc chương trình nén "MS-DOS \pkzip and \arj" thì các tệp sẽ bị mở rộng ra hơn 2 bit trên mỗi thành phần cho dù những phần mềm nén này là những thuật toán nén cơ bản. Những phần mềm này được thiết kế để nén văn bản, trong khi đó những quy tắc trong chuỗi DNA thì lại phức tạp hơn. Mã hóa 2 bit là cách hiệu quả nếu các bazơ xuất hiện ngẫu nhiên trong chuỗi. Nhưng cuộc sống của một sinh vật là không ngẫu nhiên, do đó chuỗi DNA xuất hiện trong 1 sinh vật là không ngẫu nhiên và có một số ràng buộc. Nén chuỗi DNA là một nhiệm vụ rất thách thức. Đặc trưng phức tạp của một chuỗi DNA nằm ở chỗ đó là một chuỗi các chỉ số độ dài khác nhau biểu diễn một phạm vi có thể dự đoán được của các thành phần cơ bản cấu tạo nên DNA. Những đặc trưng phức tạp này cho phép tìm kiếm những cấu trúc lặp bên trong một nhiễm sắc thể hoặc qua nhiều nhiễm sắc thể. Và cũng chính những đặc trưng này được sử dụng để tìm ra khoảng cách tiến hóa và cấu trúc nên cây phát sinh loài. Do sự cấu tạo phức tạp này mà có thể thấy là trong thực tế không có 1 chương trình nén tệp thông thường nào có thể nén chuẩn được chuỗi DNA. Nhiều thuật toán nén dành riêng cho chuỗi DNA đã được phát triển từ khoảng 10 năm trước. Sự thật là nén chuỗi DNA là một việc khó đối với các thuật toán nén cơ bản, nhưng từ quan điểm của lý thuyết nén thì nó là một đề tài

thú vị cho việc tìm hiểu thuộc tính của nhiều thuật toán nén. Ở đây chúng ta nói về phương pháp luận của các phương pháp nén một cách ngắn gọn.

Hiện nay, kỹ thuật nén dữ liệu chuỗi gen được sử dụng rộng rãi trong lưu trữ dữ liệu sinh học. Có hàng trăm thuật toán đã được đề xuất cho nén dữ liệu DNA nhưng nhìn chung các thuật toán nén được chia thành một số cách tiếp cận như sau: (1) mã hóa bit (naive bit manipulation), (2) nén dựa trên bộ từ điển (dictionary-based), (3) nén thống kê (statistical), và (4) nén tham chiếu (reference-based) [1,2]. Trong khuôn khổ luận văn, người viết chỉ trình bày một số thuật toán tiêu biểu cho từng phương pháp đã nêu và hầu hết các phương pháp đều nhằm hai mục đích chính: đạt được tỉ lệ nén cao nhất có thể để tiết kiệm không gian lưu trữ và đạt được tốc độ nén/giải nén cũng như truy cập thông tin nhanh chóng.

- *Thuật toán mã hóa bit*: sử dụng mã hóa độ dài cố định hai hoặc nhiều kí tự trên một byte đơn [38].
- *Thuật toán nén dựa trên bộ từ điển*: hay còn gọi là thuật toán thay thế, thuật toán thay thể các chuỗi lặp bằng việc tham chiếu tới một từ điển (một tập các chuỗi đã có hoặc được xác định trước), từ điển này được xây dựng trong thời gian chạy (runtime) hoặc ngoại tuyến (offline) [39, 40].
- *Thuật toán nén thống kê:* hay còn gọi là thuật toán mã hóa entropy, bắt nguồn từ một mô hình lấy xác suất dữ liệu đầu vào. Dựa trên các chuỗi khớp từng phần của tập con đầu vào, mô hình dự đoán kí tự tiếp theo trong chuỗi. Tỉ lệ nén cao có thể đạt được nếu mô hình luôn chỉ ra được xác suất cao cho kí tự tiếp theo, nghĩa là dự đoán đáng tin cậy [15, 41].
- *Thuật toán nén tham chiếu:* tương tự nén dựa trên bộ từ điển, thuật toán thay thế các chuỗi con dài của đầu vào với tham chiếu tới chuỗi khác. Tuy nhiên, tham chiếu này trỏ tới các chuỗi bên ngoài mà không phải là một phần của dữ liệu nén. Hơn nữa, tham chiếu thường là tĩnh còn từ điển thì được mở rộng trong pha nén.

Trung bình thuật toán mã hóa bit đạt tỉ lệ 4:1, thuật toán nén dựa trên bộ từ điển đạt 4:1 đến 6:1, thuật toán xác suất đạt 4:1 tới 8:1, riêng thuật toán nén tham chiếu có thể đạt tới tỉ lệ 400:1 [2] hoặc có thể cao hơn với điều kiện lý tưởng về chuỗi tham chiếu và chỉ số nén.

Thuật toán nén tham chiếu mang tới một tiềm năng lớn cho nén chuỗi đa lượng, điển hình là chuỗi DNA. Tương tự như thuật toán nén dựa trên bộ từ điển nhưng do các chuỗi mã hóa tham chiếu tới tập hợp chuỗi tham chiếu bên ngoài nên tốc độ nén cao hơn và giải mã cũng thuận lợi hơn. Các chuỗi DNA được nén

tham chiếu bao gồm các phần khớp nhau về khoảng và có thể đạt tới tốc độ nén cao nhất đối với nén trong cùng loài. Tuy vẫn còn một số bất lợi cho nén các hệ gen khác loài nhưng nén tham chiếu rõ ràng đã cho thấy lợi thế về tỉ lệ nén và tốc độ nén nếu đạt được một số điều kiện lý tưởng. Vì việc tìm ra chuỗi tham chiếu phù hợp là điều khá khó khăn do các chuỗi gen nghiên cứu là các mẫu được lấy ngẫu nhiên từ một tập hợp lớn các loài. Bên cạnh việc tìm kiếm chuỗi khớp xác định thì việc khớp giữa đầu vào và chuỗi tham chiếu cũng khá là phức tạp. Tuy nhiên, phương pháp tìm kiếm một chuỗi tham chiếu tốt có thể dựa trên *băm k-mer*. Sự tương đồng cao của *k*-mers đưa ra một tiềm năng lớn cho việc nén dựa trên tham chiếu. Có nhiều khung nén đã được phát triển dựa trên thuật toán nén tham chiếu. Qua thời gian, mỗi phương pháp nén dựa trên tham chiếu đều được cải tiến về phương thức lưu trữ dữ liệu, đánh chỉ số chuỗi gen, thuật toán tìm kiếm chuỗi tham chiếu tốt nhất hay viết lại tham chiếu và tìm kiếm chuỗi khớp tối ưu. Tất cả những cải tiến này đều cho thấy những hiệu quả khả quan đạt được về tỉ lệ cũng như tốc độ nén/giải nén chuỗi gen của thuật toán nén dựa trên tham chiếu. Đây cũng chính là lý do mà trong luận văn này, người viết tập trung nghiên cứu, thực nghiệm so sánh kết quả nén chuỗi đa lượng DNA dựa trên thuật toán nén tham chiếu với thuật toán nén tiêu biểu là JDNA, phát triển dựa trên thuật toán được sử dụng bởi FRESCO [25], được tối ưu với 3 phương pháp cải tiến là lựa chọn tham chiếu, viết lại tham chiếu và nén thứ tự hai. Ngoài ra JDNA còn thêm hai cải tiến để tối ưu về tỉ lệ nén và thời gian nén/giải nén là (1) sử dụng tính tương đương và (2) thay thế chỉ số tham chiếu hoàn toàn bằng một phương thức chỉ số theo yêu cầu. Những cải tiến này cho kết quả rất tốt về tỉ lệ nén, có thể đạt tỉ lệ nén tới 1000:1 với điều kiện lý tưởng. Người viết cũng thực hiện thực nghiệm bổ sung so sánh thuật toán tham chiếu JDNA với thuật toán nén dựa trên phương thức khác là Lempel-Ziv, nén dựa trên bộ từ điển và Huffman, nén dựa trên xác suất thống kê để thấy rõ được tính ưu việt của thuật toán tham chiếu này về cải thiện tỉ lệ nén, tốc độ giải nén và dung lượng lưu trữ. Tuy kết quả đạt được về tỉ lệ nén và thời gian nén của thực nghiệm bổ sung chưa đạt được tỉ lệ mong đợi cao nhất của thuật toán nén tham chiếu do còn hạn chế về môi trường thực nghiệm nhưng đã góp phần chứng minh những nhận định về hiệu quả của thuật toán nén tham chiếu đối với việc nén chuỗi gen mà người viết đã nghiên cứu.

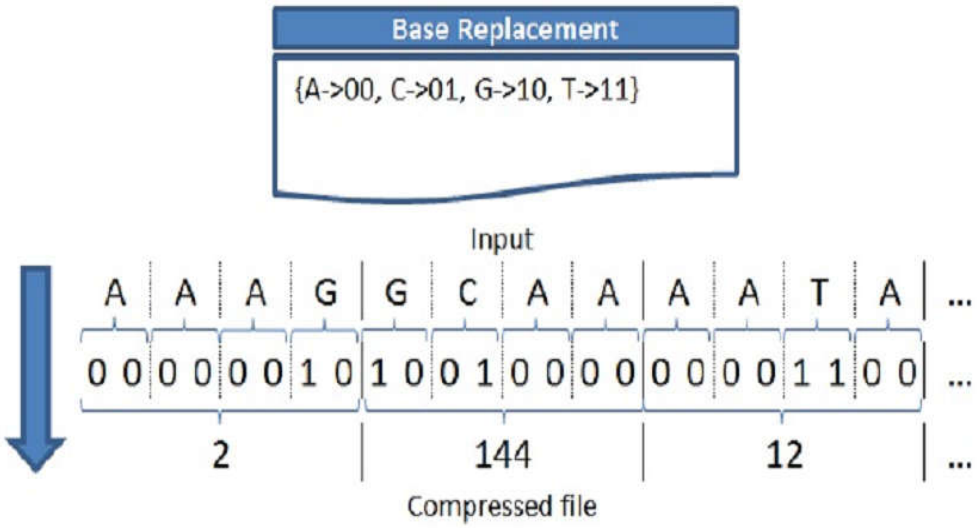
Bố cục luận văn được chia thành 3 chương. Chương 1 trình bày về tổng quan các phương thức nén dữ liệu sử dụng cho nén chuỗi DNA. Thuật toán nén tham chiếu cụ thể mà người viết luận văn tập trung nghiên cứu, thuật toán nén tham chiếu JDNA, được trình bày ở chương 2. Chương 3 của luận văn mô tả

môi trường thực nghiệm so sánh thuật toán nén tham chiếu JDNA với hai thuật toán thuộc phương thức nén khác và một số phân tích đánh giá của người viết về kết quả đạt được. Cuối cùng là kết luận về hiệu quả cũng như hạn chế còn tồn tại và hướng phát triển trong tương lai.

CHƯƠNG 1 – TỔNG QUAN VỀ THUẬT TOÁN NÉN DỮ LIỆU

1.1. Thuật toán mã hóa bit (Naïve Bit)

Thuật toán mã hóa bit sử dụng các bit trạng thái để biểu diễn dữ liệu nén. 4 bazơ đặc trưng của DNA được mã hóa bởi 2 bit (4 trạng thái). Kỹ thuật nén thẳng dữ liệu chuỗi DNA là mã hóa 4 bazơ trong một byte theo mã hóa bit. Hình 1.1 [2] cho thấy một ví dụ về nén mã hóa bit



Hình 1.1. Ví dụ mã hóa bit

Mỗi kí tự ở đầu vào được thay thế bởi 2 bit sử dụng phép thay thế {A = 00, C = 01, G = 10, T = 11}.

Những cấu trúc hiện tại cung cấp các phép toán bit tốt hơn, về cơ bản cho phép một mã hóa của dữ liệu chuỗi DNA với 2 bit. Mã hóa này ảnh hưởng tới khả năng đọc dữ liệu đáng kể vì cần một bảng tìm kiếm để dịch dữ liệu nén. Do biểu diễn 4 bazơ vừa đủ chính xác trong 8 bit nên nếu xảy ra thêm giá trị biên thì sẽ phá hỏng cấu trúc này.

Mã hóa sẽ trở nên phức tạp nếu thêm một hoặc nhiều phần bù ví dụ như *N* vào chuỗi. Một phương pháp dùng để mã hóa 5 kí tự *A, C, G, T, N* là đặt 3 bazơ liên tiếp vào 1 byte. 7 bit có thể mã hóa 128 trạng thái và vì $5^3 < 128$. Tuy nhiên, việc tăng kích thước các kí tự (nhiều kí tự được thêm vào chuỗi) sẽ khiến cho việc biểu diễn kí tự trở nên khó khăn hơn.

Tỉ lệ nén của thuật toán mã hóa bit là 4:1 nếu kích thước của chuỗi kí tự đầu vào là 4 hoặc ít hơn 4:1 nếu nhiều hơn 4 kí tự [2].

Có nhiều thuật toán được xây dựng dựa trên phương thức mã hóa bit như thuật toán mã hóa trực tiếp phần khác biệt (thuật toán 2D), thuật toán này có thể xử lý các chuỗi đầu vào ở bất kỳ định dạng nào. Với 5 kí tự thông thường của

DNA (A, C, G, T, N), một mã hóa 7bit cho 3 kí tự liên tiếp được sử dụng. Theo cách này thì có tới 128 kí tự bổ sung sẽ được mã hóa. Tiếp theo là Genbit compress (GBC), một công cụ nén chuỗi viết bằng ngôn ngữ java, sử dụng mã hóa độ dài (run-length encoding) thực hiện trên 2 bit (naïve 2bit) [3]. [4] cũng đưa ra một phương thức nén các nhiễm sắc thể tương đồng, mã hóa 3 bazơ sử dụng 1 byte. Tuy nhiên, trong thuật toán này kết hợp những xử lý phức tạp cho phần lặp *N*, sau đó nén mã hóa đạt được bằng LZ77. Một phương thức khác thuộc lớp thuật toán này được xây dựng trên cơ sở dữ liệu Oracle [5]. Và [6] kết hợp một thuật toán bổ sung cho việc tìm kiếm nhiều đoạn trong dữ liệu nén. Sau cùng là một thuật toán tập trung vào việc phân tích cách thức lưu trữ các phần lặp với những mã hóa có kích thước biến đổi, thuật toán DNABit [7].

Do tính đặc trưng của thuật toán mã hóa bit được thể hiện khá rõ nét trong 2 thuật toán mã hóa trực tiếp phần khác biệt (2D) và DNABit nên sau đây người viết luận văn sẽ trình bày chi tiết hai thuật toán này.

1.1.1. Mã hóa trực tiếp phần khác biệt (thuật toán 2D)

Với sự phát triển ngày càng mạnh về các tập dữ liệu gen khổng lồ, nhiều phương pháp nén đã và đang được phát triển để đáp ứng khối lượng lớn gen gồm nhiều chuỗi và phần bù lớn hơn (như đầu chuỗi). Các giao thức nén phát triển riêng cho dữ liệu chuỗi thì thường có tỉ lệ nén tốt nhưng hiệu suất thấp trên tập dữ liệu lớn mà gồm nhiều dữ liệu phụ trợ (phần bù). Để so sánh thì những ứng dụng nén thông thường có thể dễ dàng nén các tệp dữ liệu lớn không đồng nhất nhưng lại bị hạn chế đối với dải dữ liệu kí tự trong dữ liệu chuỗi. Bởi vậy, thuật toán 2D được thiết kế để cung cấp một giao thức nén chuỗi nucleotit thông thường. Giao thức này có thể phân biệt dữ liệu chuỗi và dữ liệu phần bù, từ đó đưa ra sự điều chỉnh phù hợp giữa nén dữ liệu chung chung và cụ thể. Thuật toán 2D có những mục tiêu như sau [43]:

- Thời gian thực hiện tuyến tính cho việc hỗ trợ các tập dữ liệu lớn: cả hai quá trình nén và giải nén đều phải hỗ trợ thực hiện đối với độ phức tạp thời gian thực hiện O(n).
- Hỗ trợ bao gồm cả những kí tự phụ mà không phải thành phần của tập bazơ nucleotit mong đợi: các kí tự bổ sung có thể được sử dụng để biểu diễn thông tin tự do, dữ liệu chú thích hoặc các chuỗi con đặc biệt như miền chức năng hoặc các chuỗi lặp đặc biệt.
- Mã hóa trực tiếp pha đơn: Pha nén yêu cầu chỉ một chiều đơn mà không có pha loại bỏ những thông tin dư thừa và không lưu trữ dữ liệu vào các tệp

cấu trúc phụ hoặc trung gian tạm thời. Tương tự, việc không lưu trữ dữ liệu phụ phải cho phép khôi phục một chiều đơn đối với pha giải nén.

- Nén không mất dữ liệu: Chuỗi gốc phải được khôi phục hoàn toàn sau quá trình giải nén. Việc này có thể được thực hiện chỉ dựa trên chuỗi thẳng mà không quan tâm tới định dạng hay bị ngắt dòng, hoặc dựa trên bố cục từng dòng của dữ liệu chuỗi gốc.
- Không phân biệt loại chuỗi: Nén và giải nén không ưu tiên hay phân biệt chuỗi là DNA hay mRNA.
- Giải nén chuỗi polipeptit (mỗi peptit gồm 10 tới 100 amino axit): Có thể lựa chọn khôi phục chuỗi nén nucleotit trực tiếp tới một chuỗi polipeptit dựa trên khung đọc xác định.
- Sử dụng được cùng với phương pháp nén khác: Một chuỗi mã hóa 2D có thể nén được bằng những ứng dụng nén khác để đưa ra khả năng nén chuỗi gốc trong tương lai.

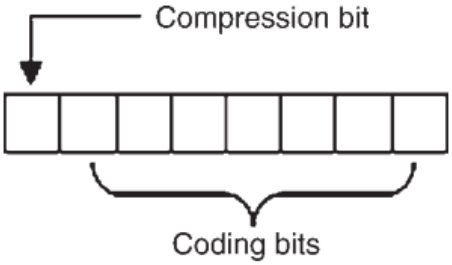
(a) Mô hình

Để cho thời gian thực hiện tuyến tính, 2D sử dụng một mô hình tĩnh cho việc mã hóa dữ liệu chuỗi cùng với bất kỳ thông tin nào mà có thể được bao gồm trong đầu vào. 2D cho rằng DNA gồm {A, C, G, T} và mRNA gồm {A, C, G, U} . Đồng nhất hai tập trên, tập kí tự cho mô hình 2D gồm {A, C, G, T, U}. Việc này giúp khai báo rõ ràng loại chuỗi. Trong trường hợp kí tự không phải nucleotit, 2D hỗ trợ tập giá trị ASCII truyền thống gồm 0 tới 127.

Để hoàn thành nén, 2D cần biểu diễn nhiều bazơ sử dụng một byte đơn như khung 2-bit-mỗi-bazơ. 2D sử dụng mã hóa trực tiếp trên một bộ ba (3 bazơ nucleotit liên tiếp) vì những lý do sau. Đầu tiên, việc này cho phép 3 bazơ nucleotit hợp lại trong một byte đơn mà không phải là nhiều byte. Thứ hai, bằng việc nén bộ ba (thay vì bộ hai) thì các kí tự không mong muốn có thể được mã hóa trực tiếp. Do đó giúp bỏ được pha loại các kí tự thừa và lưu trữ dữ liệu thừa trong cấu trúc thứ cấp. Điều này có lợi cho cả thời gian nén và giải nén. Sau cùng, biểu diễn theo bộ ba giúp 2D giải nén các chuỗi polipeptit bằng cách biên dịch bộ ba như các codon (chuỗi liên kết 3 nucleotit của DNA hoặc RNA).

(b) Mã hóa

Ở mức thấp nhất, 2D sử dụng một byte được gán có dải giá trị từ -128 tới 127. Về mặt khái niệm, 7 bit của mỗi byte được sử dụng cho mã hóa và bít quan trọng nhất được sử dụng như một cờ nén. Khung nén này được mô tả ở Hình 1.2



Hình 1.2. Khung mã hóa byte 2D. Ít nhất 7 bit được sử dụng để mã hóa dữ liệu. Bit quan trọng nhất được sử dụng như cờ để cho biết ngữ cảnh của byte là dữ liệu nén hay giải nén.

Các kí tự được chuyển thành các bộ ba liên tiếp nếu mỗi thành phần là một bazơ nucleotit hợp lệ. Một bộ ba hợp lệ được gán một giá trị đơn trong dải từ 1 tới 125 và cờ nén là một tập hợp ngang hàng với giá trị gán trong khoảng -1 và -125. 2D sẽ phân biệt dữ liệu chuỗi và các kí tự khác, nếu một giá trị không mong muốn xảy ra thì nó có thể được biên dịch như một giá trị ASCII trong dải từ 0 tới 127, sau đó giá trị này sẽ được lưu nguyên bản và không được gán cờ nén. Trong trường hợp xảy ra giá trị không mong muốn thì các thành phần khác của bộ ba hiện tại vẫn được mã hóa và giải nén độc lập dù có bazơ nucleotit hay không, việc này để duy trì khung đọc hiện tại cho việc hỗ trợ biên dịch một polipeptit chính xác. Mặc định là sự thực hiện có thể giả sử một khung đọc mong muốn bắt đầu cùng với phần bắt đầu của chuỗi. Tuy nhiên, nhiều khung đọc cũng được hỗ trợ dễ dàng bằng việc mã hóa một hoặc hai kí tự đầu tiên khi dữ liệu chưa được nén và sau đó mới bắt đầu thực hiện 2D. Sau cùng, trong trường hợp kí tự không-biết, 2D biểu diễn nó bằng việc lưu ở dạng chưa nén với giá trị byte được gán nhỏ nhất -128.

(c) Thuật toán

Đoạn mã giả sau đây mô tả lõi của thuật toán nén 2D, nhận một chuỗi đầu vào và trả về mã hóa 2D dưới dạng mảng byte.

```
begin
  byte list = new List
      char triplet = new Array
      int baseCount = 0
      int nonCompressCount = 0
      foreach character c in input string
          if nonCompressCount = 0 then
              if c is a nucleotide base then
                  triplet at position baseCount = c
                  baseCount = baseCount + 1
```

```
            if baseCount = 3 then
              convert triplet to byte b and add b to list
              reset triplet
              baseCount = 0
          else
              foreach character t in triplet
                  convert t to byte b and add b to list
              endfor
              convert c to byte b and add b to list
              reset triplet
              nonCompressCount = 2 - baseCount
              baseCount = 0
        else
            convert c to byte b and add b to list
            nonCompressCount = nonCompressCount - 1
    endfor
  return list as byte Array
end
```

Dữ liệu giải mã được khôi phục theo dòng với độ dài chia hết cho 3. Ví dụ, nếu chuỗi trong tệp nguồn được chia thành dòng, mỗi dòng 70 kí tự thì chuỗi trong tệp khôi phục sẽ có độ dài dòng là 69, 69, 72, 69, 69, 72… . Việc này thực hiện để tăng tính nén toàn bộ mà vẫn duy trì được khả năng đọc. Tuy nhiên, nếu yêu cầu thì có thể thực hiện được phiên bản đọc từng dòng chính xác nhưng sẽ bị giảm tính nén toàn bộ.

(d) Tỉ lệ nén

Do 2D sử dụng khung mã hóa trực tiếp nên tỉ lệ nén của nó được xác định bởi kích thước ban đầu trên kích thước đã mã hóa, có thể được tính xấp xỉ theo một công thức chung. Giả sử yêu cầu 1 byte biểu diễn một kí tự chưa nén. Nếu chuỗi ban đầu chỉ gồm các bazơ nucleotit có độ dài *L,* kích thước *L* byte thì kích thước mã hóa của nó sẽ là *(L/3 + L mod 3)* byte. Như vậy các kí tự bổ sung cũng dường như xảy ra ở một số tần suất xấp xỉ. Do việc xảy ra một hoặc nhiều kí tự như vậy trong bộ ba đưa ra sẽ dẫn đến việc mã hóa gồm các kí tự thừa. Bởi vậy, 2 byte phải được thêm vào kích thước mã hóa cho mỗi khi xảy ra một kí tự thừa *[aL]*, trong đó *a* là tần suất các kí tự thừa và các kí tự thừa này được phân bố ngẫu nhiên mà không phải là gói lại cùng nhau. Do đó kích thước chuỗi mã hóa 2D được tính xấp xỉ theo công thức:

$$Encoded\ size\ \approx \left(\frac{L}{3} + L\ mod\ 3 + 2\ aL\ \right) bytes$$

Công thức này có thể được thay thế trong định nghĩa ban đầu về tỉ lệ nén để đưa ra một công thức chung tính tỉ lệ nén 2D:

$$Compression\ ratio\ \approx L\ bytes / \left(\frac{L}{3} + L\ mod\ 3 + 2\ aL\ \right) bytes$$

So sánh hiệu quả nén của thuật toán 2D với một số thuật toán khác như GenCompress, gzip hay kết hợp 2D và gzip sử dụng hệ gen vi khuẩn *Bacillus subtilisand*, *Escherichia coli K12 MG1655* và *Mycoplasma genitalium*. Ta có kết quả như ở bảng 1.1.

Bảng 1.1. So sánh nén hệ gen sử dụng GenCompress, 2D, gzip và 2D + gzip.

Compression method	Source genome								
	Bacillus subtilis			*Escherichia coli K12 MG1655*			*Mycoplasma genitalium*		
	Size (bytes)	Ratio	Time (ms)	Size (bytes)	Ratio	Time (ms)	Size (bytes)	Ratio	Time (ms)
None	4 274 929	1.000	N/A	4 706 046	1.000	N/A	588 437	1.000	N/A
GenCompress	0	Ø	58 363 756	0	Ø	27 887 599	0	Ø	8 127 438
2D	1 465 177	2.918	717.5	1 612 930	2.918	788.9	201 721	2.917	100.5
gzip	1 300 308	3.288	1671.3	1 431 844	3.287	1819.4	174 398	3.374	254.5
2D + gzip	1 093 657	3.909	824.9	1 214 444	3.875	891.3	145 727	4.038	182.8

Trong đó, kích thước tệp, tỉ lệ nén và thời gian thực hiện được đưa ra cho mỗi thuật toán dựa trên mỗi hệ gen. Thời gian thực hiện là kết quả trung bình từ 100 thử nghiệm, ngoại trừ GenCompress đạt được thời gian thực hiện ngắn nhất sau 3 lần thất bại liên tiếp.

Bảng 1.2. Kết quả giải nén sử dụng hệ gen nén bởi 2D

Source genome	File size (bytes)			File inflation		Decomp. time (ms)
	Normal	2D Comp.	2D Decomp.	Bytes	Lines	
Bacillus subtilis	4 274 929	1 465 177	4 274 930	1	1	923.9
Escherichia coli K12 MG1655	4 706 046	1 612 930	4 706 047	1	1	1042.3
Mycoplasma genitalium	588 437	201 721	588 438	1	1	116.2

2D cung cấp một giao thức nén chuỗi nucleotit thông thường, giao thức này có thể phân biệt dữ liệu chuỗi và dữ liệu phụ. Từ đó đưa ra phương thức điều chỉnh, kết hợp việc nén cho mục đích thông thường hoặc các chuỗi đặc biệt. Tuy thuật toán 2D phù hợp cho nén hầu hết các dữ liệu dạng chuỗi, bao gồm cả những tập dữ liệu lớn như metagenomes. Nhưng do 2D nén toàn bộ hệ gen, kể cả các dữ liệu phụ, bổ trợ hoặc dư thừa ngay cả khi những dữ liệu dư thừa đó không bao giờ được dùng đến. Điều này tiêu tốn về không gian lưu trữ cũng như hiệu quả sử dụng thông tin.

1.1.2. Thuật toán nén DNABIT

Thuật toán DNABIT nén các chuỗi DNA của toàn bộ hệ gen (cả các chuỗi lặp và không lặp) theo 2 pha. Hai pha lần lượt sử dụng 5 kỹ thuật được gọi là *kỹ thuật 2 Bit, 3, 5, 7* và *9 Bit.*

2 pha: (1) Kỹ thuật Bit chẵn: kỹ thuật 2Bit; (2) Kỹ thuật Bit lẻ: kỹ thuật 3Bit, kỹ thuật 5Bit, kỹ thuật 7Bit, kỹ thuật 9Bit.

- **Kỹ thuật Bit chẵn:**

Chuỗi DNA được gán 2 bit cho mỗi bazơ đơn. Các bazơ đó là {A, C, G, T}. 2 bit được gán tới các bazơ của vùng không lặp. Các bit được gán tới các bazơ đơn như ở bảng 1.3.

Bảng 1.3. Các bazơ và bit nhị phân biểu diễn

BAZƠ	CÁC BIT NHỊ PHÂN
A	00
G	01
C	10
T	11

- **Kỹ thuật Bit lẻ**

Kỹ thuật Bit lẻ sử dụng 4 kỹ thuật đó là Kỹ thuật 3Bit, Kỹ thuật 5Bit, Kỹ thuật 7Bit và Kỹ thuật 9Bit.

(a) <u>Kỹ thuật 3Bit</u>

Trong chuỗi DNA nếu tồn tại 2 hoặc 3 bazơ giống nhau liền kề thì kỹ thuật 3Bit được ứng dụng. Chuỗi mã hóa được biểu diễn dạng mã 3Bit. Trong mã 3Bit, bit đầu tiên bên trái được đặt là "0" hoặc "1". Bit đầu tiên là "0" nếu bazơ nhắc lại 2 lần và "1" nếu bazơ nhắc lại 3 lần. Mã 3Bit được thể hiện trong bảng 1.4 và 1.5

Bảng 1.4. Mã 3Bit cho các bazơ nhắc lại chính xác 2 lần.

Bazơ	Mã 3Bit
AA	000
GG	010
TT	011
CC	001

Bảng 1.5. Mã 3Bit cho các bazơ nhắc lại chính xác 3 lần

Bazơ	Mã 3Bit
AAA	100
GGG	110

TTT	111
CCC	101

(b) <u>Kỹ thuật 5Bit</u>

Trong chuỗi DNA nếu tồn tại nhiều hơn 3 bazơ lặp, lên tới 8 (4,5,6,7,8) tức là có từ trên 3 tới 8 bazơ giống nhau liền kề thì kỹ thuật 5Bit được ứng dụng. Chuỗi mã hóa được biểu diễn ở dạng mã 5Bit như ở bảng 1.6 đến bảng 1.10

Bảng 1.6. Mã 5Bit cho lặp 4

Bazơ	**Mã 5Bit**
AAAA	01100
GGGG	01110
TTTT	01111
CCCC	01101

Bảng 1.7. Mã 5Bit cho lặp 5

Bazơ	**Mã 5Bit**
Aaaaa	10000
Ggggg	10010
Ttttt	10011
Ccccc	10001

Bảng 1.8. Mã 5Bit cho lặp 6

Bazơ	**Mã 5Bit**
aaaaaa	10100
gggggg	10110
tttttt	10111
cccccc	10101

3 bit đầu tiên biểu diễn mã cho (lặp) "6". [6 = 101]

Bảng 1.9. Mã 5Bit cho lặp 7

Bazơ	**Mã 5Bit**
aaaaaaa	11000
ggggggg	11010
ttttttt	11011
ccccccc	11001

3 bit đầu tiên biểu diễn mã cho (lặp) "7". [7 = 110]

Bảng 1.10. Mã 5Bit cho lặp 8

Bazơ	**Mã 5Bit**

aaaaaaa	11000
ggggggg	11010
ttttttt	11011
ccccccc	11001

3 bit đầu tiên biểu diễn mã cho (lặp) "8". [8 = 111]

(c) Kỹ thuật 7Bit

Mã 7Bit cho lặp chính xác 2 bazơ:

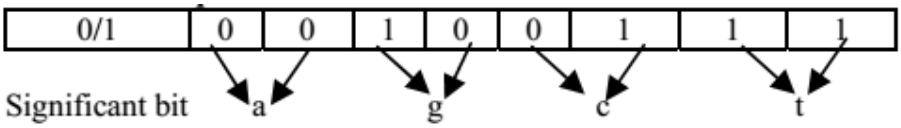
Trong chuỗi nếu có 2 kí tự lặp lại từ hơn 1 lần tới 8 lần thì chuỗi 7 bit được sử dụng. Trong mã 7Bit này, 3 bit đầu tiên biểu diễn số lặp của kí tự đó. 4 bit còn lại biểu diễn mã cho kí tự (xem bảng 1.11)

Bảng 1.11. Kỹ thuật 7Bit

Số lần lặp	**Mã 3Bit**
2	001
3	010
4	011
5	100
6	101
7	110
8	111

(d) Kỹ thuật 9Bit

Trong mã 9Bit có hai kỹ thuật. Nếu 4 bazơ liên tiếp giống nhau thì chuỗi mã hóa lấy mã 9Bit. Bit đầu tiên hoặc là "0" hoặc là "1". "0" chỉ ra phép lặp là lặp chính xác, "1" chỉ ra phép lặp là lặp nghịch đảo (xem hình 1.3).



Hình 1.3. Kỹ thuật 9Bit

Trong 9Bit thì bit đầu tiên biểu diễn ý nghĩa là giống nhau hoặc nghịch đảo. 8 bit còn lại biểu diễn mã cho mỗi bazơ.

- **Tính tỉ lệ nén**

Cho = Tổng số bazơ trong chuỗi

Tỉ lệ nén = Số Bit/ Tổng số bazơ

μ = Số mã 3Bit * 3

β = Số mã 5Bit * 5

δ = Số mã 7Bit * 7

γ = Số mã 9Bit * 9

þ = Số mã Bit chẵn * 2 (chuỗi bazơ đơn)

Tổng số Bit mã hóa nhị phân = (μ + β + δ + γ+ þ)

Tỉ lệ Ř = (μ + β + δ + γ+ þ) / Bits/Bases.

Ví dụ: Gggggagctacgagctagctacaccacatatatatatatatata

Mã hóa: 1000100011011001001000011011100011000001111011111100

Tỉ lệ: **Ř** = 2*3 + 5*1 +7*1 + 9*2 +2*8 / 50

= 52/50

= 1.04 bits/bazơ.

Trường hợp tốt nhất**:** Hiệu quả tốt nhất được chứng minh trong thuật toán nén DNABIT mới vì tỉ lệ nén tốt nhất của nó là R = 1,04 và xấu nhất là R = 1.58. Trường hợp xấu nhất được xác định là khi số lặp trong chuỗi DNA là không đáng kể.

- **Thuật toán mã hóa nén DNABIT**

Đầu vào: Chuỗi đầu vào (INSTRING) gồm A, T, G, C

Đầu ra: Chuỗi đã mã hóa (OUTSTRING)

Thủ tục mã hóa:

Begin

1: Chia chuỗi DNA thành các đoạn, trong đó mỗi đoạn gồm 2 kí tự, 4 kí tự.

2: Tạo tất cả các liên kết có thể trong chuỗi DNA (A, C, G, T)

3: Áp dụng Kỹ thuật Bit chẵn nếu các bazơ đồng thời không khớp với các bazơ khác (chuỗi DNA được gán 2 bit cho mỗi bazơ đơn thuộc vùng không lặp)

4: Nếu tồn tại 2 hoặc 3 bazơ liền kề giống nhau thì áp dụng kỹ thuật 3Bit

5: Nếu tồn tại hơn lặp 3 tới lặp 8 (4,5,6,7,8) thì áp dụng kỹ thuật 5Bit. Chuỗi mã hóa được biểu diễn ở dạng mã 5Bit

6: Nếu chuỗi có 2 kí tự lặp hơn 1 lần tới 8 lần thì áp dụng kỹ thuật 7Bit. Trong mã 7Bit, 3 bit đầu biểu diễn số lần lặp của các kí tự (4 bit khác biểu diễn mã cho mỗi kí tự)

7: Nếu 4 bazơ liên tiếp giống nhau thì chuỗi mã hóa sử dụng mã 9Bit. Bit đầu tiêu hoặc "0" hoặc "1". "0" chỉ ra lặp chính xác, "1" chỉ ra là lặp nghịch đảo.

8: Truyền các bit nhị phân tới chuỗi đầu ra (OUTSTRING)

End.

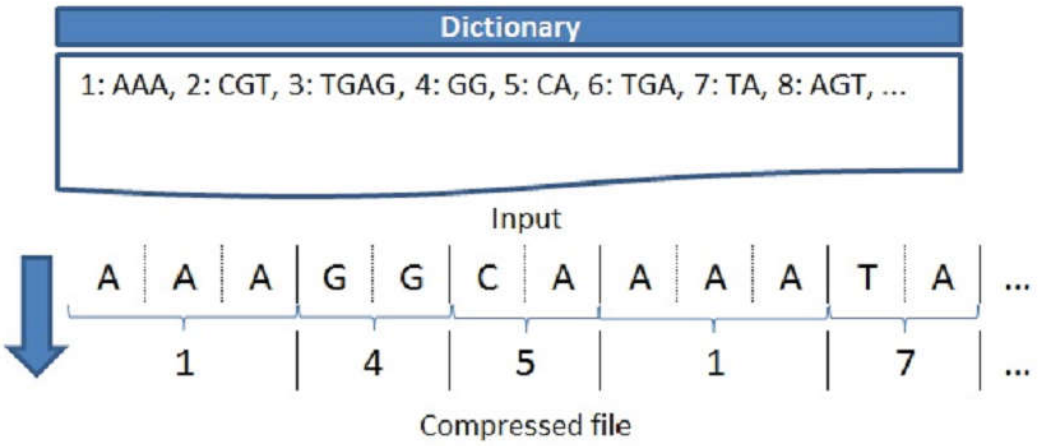
Thuật toán giải mã cũng có thủ tục giống mã hóa nhưng ở dạng nghịch đảo.

Tuy nhiên, thuật toán DNABIT chỉ quan tâm tới nén các chuỗi lặp chính xác hoặc nghịch đảo và không lặp mà chưa tính tới các trường hợp có phần bù hoặc kí tự đặc biệt. Giả sử có một kí tự bù như N cho bazơ không phân biệt

được thì việc mã hóa sẽ trở nên phức tạp. Ngoài ra sử dụng các bit nhị phân để mã hóa các bazơ như kí tự trong văn bản cũng khiến lãng phí không gian.

1.2. Thuật toán nén dựa trên bộ từ điển

Nén dựa trên bộ từ điển là một khung nén độc lập với các thuộc tính cụ thể của dữ liệu đầu vào. Phương thức chính của thuật toán là thay thế các phần tử dữ liệu lặp (các chuỗi con DNA) của đầu vào với tham chiếu tới một bộ từ điển. Những phần lặp thường được phát hiện bằng cách lưu lại các chuỗi xuất hiện trước đó. Bộ từ điển sẽ được tái cấu trúc theo nhiều cách khi thực hiện trong quá trình giải nén. Điều này có nghĩa là không cần phải lưu bộ từ điển cùng với dữ liệu nén. Hình 1.4 cho thấy một ví dụ về nén dựa trên bộ từ điển [2].



Hình 1.4. Nén dựa trên bộ từ điển

Vấn đề cần quan tâm khi thực hiện thuật toán này là độ dài trung bình của chuỗi lặp lại được mã hóa và đánh chỉ số tham chiếu ở từ điển. Thuật toán này đạt được tỉ lệ nén từ 4:1 đến 6:1 tùy theo tần số phần tử lặp lại trong hệ gen được nén [2]. Thuật toán Comrad [8] thực hiện nhiều vòng trên dữ liệu đầu vào. Mỗi vòng đều tìm được chuỗi khớp dài hơn và bổ sung vào bộ từ điển. Thuật toán chạy cho đến khi từ điển không còn thay đổi nào nữa hoặc khi đạt tới một tần số giới hạn nào đó. Một thuật toán khác cũng được đề cập tới như là thuật toán cơ sở từ điển đó là thuật toán dựa trên một cây mở rộng [9]. Cây mở rộng này là một cây tìm kiếm nhị phân tự điều chỉnh, trong đó sẽ truy cập phần tử gần nhất. Thuộc tính này có thể cải thiện được hiệu suất khi gặp phải những tần số bazơ biến đổi. Sự mở rộng ở đây là quá trình sắp xếp hoặc xoay cây khi một phần tử cụ thể được đặt vào gốc của cây đó. Tương tự như cây Huffman, các kí tự càng gần gốc thì mã hóa càng ngắn. Tuy các thuật toán đưa ra đã phần nào đạt được lợi ích từ việc xây dựng bộ từ điển và hiệu suất nén chuỗi DNA tương đối, nhưng thuật toán nén Lempel-Ziv (LZ77 và LZ78) vẫn là một trong những thuật toán nén dựa trên từ điển tiêu biểu nhất. Lempel-Ziv là một thuật toán nén

không mất dữ liệu. Thuật toán này là một nhánh của 2 thuật toán được đề xuất bởi Jacob Ziv và Abraham Lempel trong bài báo tiêu biểu của họ năm 1977, 1978.

1.2.1. LZ77

Thuật toán LZ77 là một thuật toán được sử dụng phổ biến, trong đó chương trình như PkZip sử dụng nền tảng của họ cùng với một số thuật toán khác. LZ77 tận dụng thành phần là các từ hoặc cụm từ lặp trong một tệp văn bản. Khi có sự lặp xảy ra, chúng có thể được mã hóa như một con trỏ để chỉ tới một phần xảy ra sớm hơn, con trỏ được theo sau bởi số các kí tự được khớp. LZ77 là một thuật toán rất đơn giản, nó không yêu cầu phải biết trước về nguồn hay các thuộc tính của nguồn.

Trong thuật toán LZ77, từ điển là phần mã hóa chuỗi trước tiên. Bộ mã hóa sẽ kiểm tra chuỗi đầu vào bằng cách nhấn vào dịch vụ cửa sổ trượt gồm 2 phần: bộ đệm tìm kiếm và bộ đệm xem thẳng. Một bộ đệm tìm kiếm gồm một phần chuỗi mới được mã hóa và bộ đệm xem thẳng gồm phần tiếp theo của chuỗi sẽ được mã hóa.

Thuật toán tìm kiếm cửa sổ trượt cho chuỗi khớp dài nhất mà bắt đầu ở bộ đệm xem thẳng và cho kết quả là một con trỏ tới phần khớp đó. Có thể sẽ không có chuỗi khớp nào, do đó đầu ra không thể chỉ gồm các con trỏ. Trong LZ77, chuỗi được mã hóa có dạng bộ ba *(o, l, c)*, trong đó *"o"* biểu diễn phần bù offset của chuỗi khớp, *"l"* biểu diễn độ dài chuỗi khớp và *"c"* là kí tự tiếp theo được mã hóa. Một con trỏ trống được tạo ra trong trường hợp không có chuỗi khớp nào (cả offset và độ dài chuỗi khớp đều bằng 0) và là kí tự đầu tiên trong bộ đệm xem thẳng, nghĩa là (0, 0, "kí tự"). Các giá trị của một offset cho 1 chuỗi khớp và độ dài phải được giới hạn tới một hằng số lớn nhất. Hơn nữa, hiệu suất của LZ77 phụ thuộc chủ yếu vào những giá trị này [15].

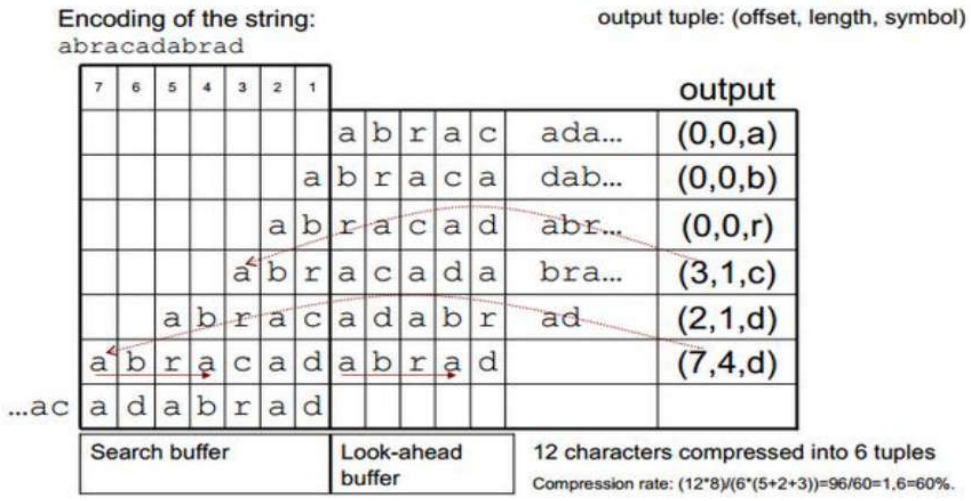
- ***Thuật toán***

```
While (bộ đệm xem thẳng không trống)
{ Nhận một con trỏ (vị trí, độ dài) cho chuỗi khớp dài nhất;
If (độ dài >0)
  { Đầu ra (vị trí, độ dài chuỗi khớp dài nhất, kí tự tiếp theo);
    Thay cửa sổ bằng (độ dài + 1) vị trí và tiếp tục;
  }
Else
{Đầu ra (0, 0, kí tự đầu tiên trong bộ đệm xem thẳng);
 Thay cửa sổ bằng 1 kí tự tiếp theo;
}
```

}

Ví dụ [6]**:**

	search buffer								look-ahead buffer									
…a	c	C	a	b	r	a	c	a	d	a	b	r	a	r	r	a	r	r	a	c…



Hình 1.5. Ví dụ về thuật toán LZ77

- ***Một số hạn chế của thuật toán LZ77***

- Trong thuật toán LZ77 gốc, Lempel và Ziv đã đề xuất toàn bộ chuỗi được mã hóa là độ dài và phần bù (offset), thậm chí cả chuỗi tìm thấy mà không khớp.
- Trong LZ77, bộ đệm tìm kiếm dài hàng nghìn bytes, trong khi bộ đệm xem thẳng chỉ 10 bytes
- Quá trình mã hóa tiêu tốn thời gian do phải thực hiện số lượng so sánh lớn để tìm mẫu khớp.
- LZ77 không có từ điển ngoài nên tạo ra vấn đề trong khi giải mã trên máy khác. Ở thuật toán này, cứ khi nào mà không có mẫu khớp nào thì nó sẽ mã hóa chuỗi đó như là độ dài và phần bù, điều này sẽ làm tốn không gian và bước không cần thiết này cũng làm tăng thời gian thực hiện thuật toán.

1.2.2. LZ78

LZ78 là một thuật toán nén dựa trên từ điển mà duy trì một từ điển rõ ràng. Đầu ra được mã hóa gồm hai thành phần: một chỉ số (index) tham chiếu tới từ điển mẫu khớp đầu vào và kí tự không khớp đầu tiên. Thuật toán cũng bổ sung cặp chỉ số và kí tự cho từ điển. Với phương pháp này, thuật toán tạo nên từ điển.

Thuật toán LZ78 có khả năng bắt được các mẫu và giữ chúng xác định nhưng nó cũng có một hạn chế nghiêm trọng. Từ điển cứ lớn dần lên mãi mà

không có điểm dừng. Có nhiều phương thức để hạn chế kích thước từ điển. Cách dễ nhất là dừng bổ sung đầu vào và tiếp tục sử dụng như một bộ mã từ điển tĩnh hoặc vứt từ điển cũ đi và bắt đầu lại từ đầu sau khi đã đạt được một kết quả nào đó.

- ***Thuật toán***

```
w :=NIL;
While (có đầu vào) {
  K := kí tự tiếp theo đầu vào;
     If (wK tồn tại trong từ điển) {
  w :=wK;
  }
       Else {
  Đầu ra (chỉ số (w),K);
  Thêm wK vào từ điển;
  w :=NIL;
  }
}
```

Ta có bảng 1.12 so sánh thuật toán LZ77 và LZ78 như sau:

Bảng 1.12. So sánh thuật toán LZ77 và LZ78

LZ77	**LZ78**
1. Thuật toán LZ77 thực hiện trên dữ liệu cũ 2. LZ77 chậm hơn 3. Định dạng đầu ra của LZ77 là bộ ba <o, l, c>, trong đó o=offset (phần bù), l= length (độ dài chuỗi khớp), c = continue (kí tự nén tiếp theo) 4. Ứng dụng: Thuật toán này là mã nguồn mở và được biết đến như ZIP, các định dạng như PNG, TIFF, PDF… LZ77 được sử dụng trong gzip, Squeeze, LHA, PKZIP và ZOO	1. Thuật toán LZ78 thực hiện trên dữ liệu mới 2. LZ78 nhanh hơn LZ77 3. Định dạng đầu ra của LZ78 là cặp <i, c>. Trong đó i = index (chỉ số) và c = continue (kí tự tiếp theo) 4. Ứng dụng: LZ78 có nhiều ứng dụng trong lĩnh vực lý thuyết thông tin như tạo số ngẫu nhiên, kiểm thử giả thuyết, phân tích cú pháp chuỗi… LZ78 được sử dụng trong GIF, CCITT (modems), ARC, PAK

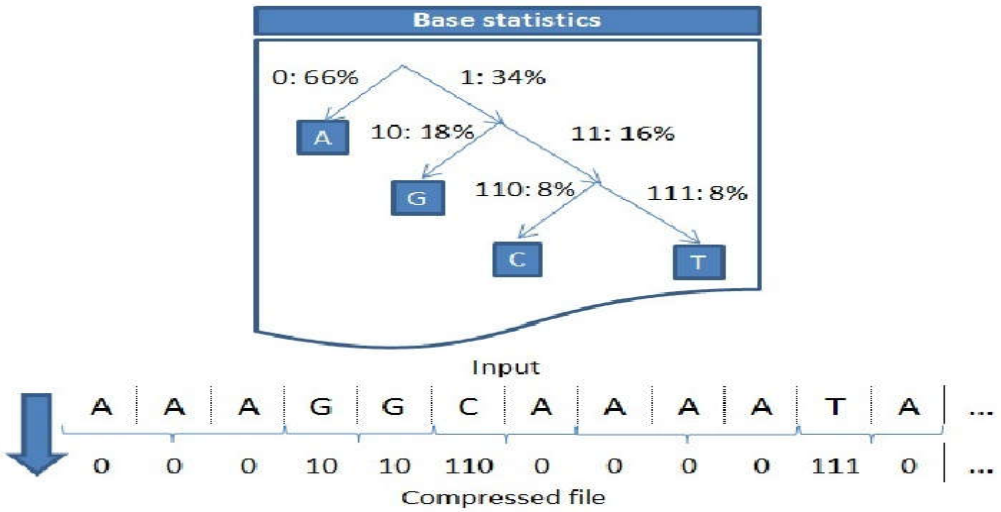
LZ77 và LZ78 có tốc độ nén chậm nhưng giải nén rất nhanh. LZ78 nhanh hơn LZ77 nhưng không phải lúc nào cũng đạt được tỉ lệ nén tốt như LZ77. L78 vượt trội hơn LZ77 ở việc giảm được số chuỗi so sánh trong mỗi bước mã hóa

Có thể nói thuật toán nén Lempel Ziv (LZ77, LZ78) dựa trên từ điển là một thuật toán nén không mất dữ liệu, ứng dụng tốt trong nén văn bản và các chuỗi khớp ngẫu nhiên. Nhưng để nén chuỗi DNA thì thuật toán này không thỏa mãn và không phù hợp.

1.3. Thuật toán nén xác suất thống kê

Thuật toán thống kê tạo một mô hình thống kê dữ liệu đầu vào mà trong hầu hết các trường hợp được biểu diễn như một cấu trúc dữ liệu cây tiền tố hoặc xác suất. Các chuỗi con với tần suất cao trong gen được biểu diễn với mã ngắn hơn. Do đó, nén thống kê có thể được xem như một dạng biến đổi của nén dựa trên từ điển mà giải quyết được các vấn đề về phát hiện lặp và mã hóa tham chiếu trong một thuật toán đơn. Tỉ lệ nén phụ thuộc chất lượng của mô hình cũng như sự tồn tại của những mẫu có thể phát hiện được ở đầu vào.

Một trong những mã hóa tiêu biểu của nén thống kê là mã hóa Huffman. Mã hóa này sử dụng một bảng mã độ dài biến đổi lấy từ đánh giá thống kê về các trường hợp xảy ra của mỗi kí tự. Một cây nhị phân được tạo ra, trong đó nút lá biểu diễn kí tự, trên cạnh ghi xác suất và mã đạt được. Kết quả bảng mã Huffman được lưu bổ sung vào chuỗi nén và bởi vậy được đưa vào việc tính tỉ lệ nén. Chia sẻ cùng một bảng mã trên nhiều chuỗi có thể giảm được không gian lưu trữ. Mã hóa Huffman sử dụng bảng kí tự lớn cùng với sự phân bố không đồng đều các kí tự sử dụng. Do đó thuật toán này không phải là lý tưởng cho nén chuỗi DNA hiệu quả. Hình 1.6 mô tả một ví dụ thuật toán thống kê [2].



Hình 1.6. Ví dụ thuật toán xác suất thống kê

Ở ví dụ này, bazơ *A* xảy ra thường xuyên được gán một mã ngắn (0), T ít xuất hiện hơn được gán mã dài hơn (111).

Trong khi mã Huffman dựa trên việc tìm kiếm mã ngắn nhất cho các kí tự đơn thì mã hóa số học lại mã hóa các chuỗi dài hơn hoặc thậm chí là toàn bộ chuỗi đầu vào như một số đơn giữa 0 và 1. Việc này giúp cho tỉ lệ nén cao hơn với số ít kí tự.

Ngoài ra, phương pháp nén dựa trên mô hình Markov ẩn cũng được đề xuất cho nén chuỗi DNA, giả sử rằng dữ liệu chuỗi DNA có thể được tính xấp xỉ bằng mô hình Markov ẩn. Người ta dùng thứ tự mô hình để phân biệt các phương pháp dựa trên mô hình Markov. Ví dụ một mô hình Markov thứ tự 2 sẽ đưa ra ngữ cảnh của hai kí tự cuối để dự đoán xác suất của các kí tự tiếp theo.

Gene-Compressor cũng được đề xuất như một phương pháp mã hóa những phần không lặp của chuỗi DNA [10]. Ở lần chạy đầu tiên, xác xuất các kí tự được đánh giá và tương đương với mã hóa Huffman đã chọn. Sau đó, đầu ra mã hóa được chia thành các khối và mỗi khối sẽ được tái cấu trúc để mã hóa độ dài hiệu quả ở bước cuối cùng.

Một mô hình khác sử dụng mô hình chuẩn hóa các chuỗi tương tự được xây dựng dựa trên mô hình Markov thứ tự 1 và mã hóa bit để biểu diễn các bazơ, ở mô hình này các chuỗi đầu vào được phân mảnh thành những khối không trùng lắp [11]. Mô hình cũng sử dụng một bộ tìm kiếm phần lặp xấp xỉ và chỉ bị ngắt bởi một vài SNP. Hệ chuyên gia cũng loại bỏ các đoạn mã ngắn và đầu ra được nén bởi nén số học. Nhưng phương pháp này chỉ có thể xử lý 4 kí tự bazơ ở chuỗi đầu vào. Do đó cũng như thuật toán mã hóa bit, chưa đạt được hiệu quả nén DNA khi có xuất hiện những bazơ bổ sung khác. Tiếp tục bổ sung và phát triển mô hình nén xác suất, Kalyan Kumar Kaipa, Kyusang Lee, Taejin Ahn cùng đồng nghiệp đã đề xuất phương pháp nén truy cập ngẫu nhiên chuỗi DNA [12]. Trong đó mã hóa các cùng không lặp cũng như các chuỗi không khớp ở phần lặp với một bộ mã hóa số học dựa trên mô hình Markov. Kết quả cho các chuỗi được phân chia thành các khối cho phép truy cập ngẫu nhiên.

Tỉ lệ nén của thuật toán nén xác suất thường là trong khoảng 4:1 và 8:1. Tỉ lệ nén phụ thuộc chủ yếu vào sự phân bố các kí tự đầu vào và bộ nhớ sẵn có cho cấu trúc phân bố tần suất.

Do những đặc trưng và lợi ích sử dụng cũng như những phát triển của các thuật toán xác suất khác dựa trên thuật toán mã Huffman và hệ chuyên gia mô hình Markov mà trong khuôn khổ luận văn, ở chương 1, mục 1.3, người viết xin trình bày chi tiết thuật toán nén HuffBit sử dụng cây nhị phân mở rộng với mã

Huffman và thuật toán mô hình chuyên gia Markov (Expert Markov) để có cái nhìn tổng quan hơn về thuật toán nén xác xuất thống kê đối với các chuỗi DNA.

1.3.1. Thuật toán nén HuffBit sử dụng cây nhị phân mở rộng với mã Huffman

Nén Huffbit sử dụng khái niệm cây nhị phân mở rộng để nén; gán 0 và 1 cho các cây con trái và phải. Huffbit là một thuật toán xử lý 2 chiều; tạo ra 1 cây nhị phân mở rộng ở pha khởi tạo. Trong trường hợp tốt nhất thì thuật toán sẽ đạt tỉ lệ nén là 1.006 bit trên mỗi bazơ. Thuật toán đơn giản nên tỉ lệ nén đạt được không thỏa mãn và nó cần quét toàn bộ chuỗi gen để đạt được tần suất của các bazơ đơn trước khi bắt đầu trình tự nén.

Thuật toán đưa ra là một quá trình xử lý 2 chiều:

(a) Thực hiện cấu trúc cây nhị phân mở rộng

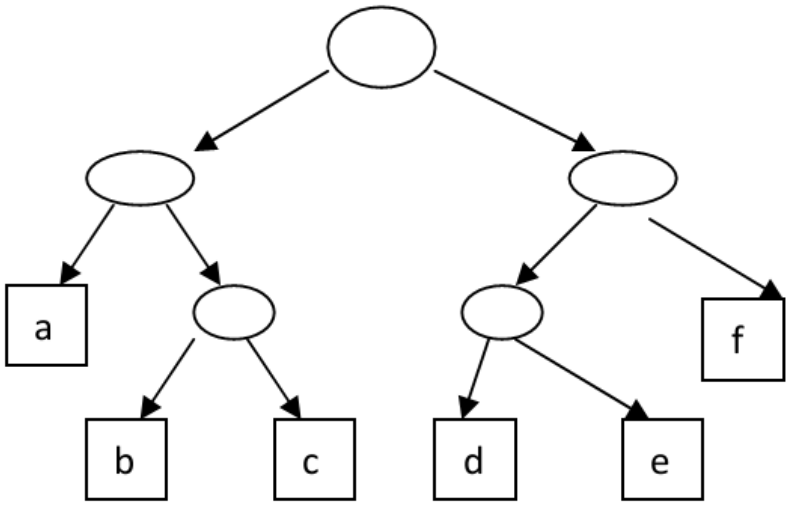
(b) Mã hóa Huffman từ cây nhị phân mở rộng được sử dụng để tính số bit của độ dài chuỗi mã hóa. Tỉ lệ nén là kích thước đã nén trên kích thước chưa nén.

- **Cây nhị phân mở rộng**

Cây nhị phân mở rộng là một dạng chuyển đổi của bất kỳ cây nhị phân nào sang một cây nhị phân hoàn toàn.

Dạng chuyển đổi này bao gồm thay thế mọi cây con rỗng của cây gốc bằng "nút đặc biệt".

Nút từ cây gốc được gọi là nút trong (internal node). Nút đặc biệt được gọi là nút ngoài (external node). Cây sau đây là cây nhị phân mở rộng. Vòng tròn trống biểu diễn nút trong và hộp vuông biểu diễn nút ngoài. Mỗi nút trong gồm chính xác 2 con và mỗi nút ngoài là 1 lá (xem hình 1.7)



Hình 1.7. Cây nhị phân mở rộng

Cây nhị phân mở rộng cho chuỗi DNA {A, C, G, T}

Coi $\partial(x)$ là độ dài của phần ngắn nhất từ nút *x* tới một nút ngoài trong cây con của nó. Giả sử nếu *x* là một nút ngoài, *x* được cho là "0"

Nếu *x* là một nút trong, giá trị ∂ là:

Nút trong ∂ = Min $\{ \partial[\tau], \partial[R]\} + 1$

Where τ = con trái của *x*.

And *R* = con phải của *x*.

Ví dụ:

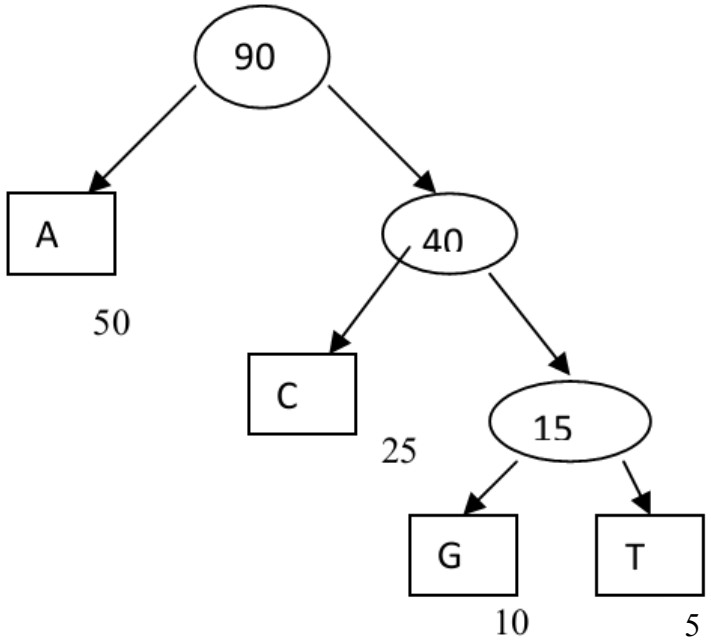
Cho chuỗi = a,c,g,t.

Độ dài = 4.

Số nút trong = 3

Số nút ngoài = 4.

Cây nhị phân mở rộng cho các bazơ A, C, G, T của DNA như hình 1.8.

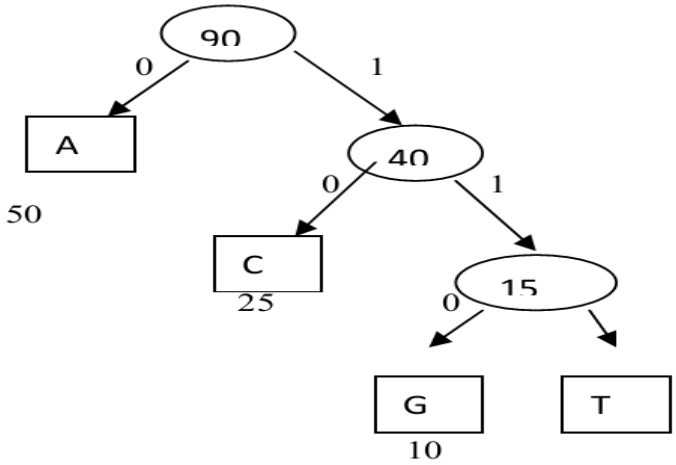


Hình 1.8. Cây nhị phân mở rộng cho các bazơ DNA {A, C, G, T}

- **Mã hóa Huffman**

Ý tưởng cơ bản của mã hóa Huffman rất đơn giản. Chúng ta lấy một số dữ liệu muốn nén, đưa ra một danh sách các kí tự 8 bit. Sau đó ta tạo một bảng giá trị mà trong đó sắp xếp các kí tự theo tần suất. Nếu ta không biết trước danh sách kí tự của ta sẽ trông như thế nào thì ta có thể sắp xếp các kí tự theo xác xuất xuất hiện của nó trong chuỗi. Tiếp đó, ta gán các từ mã cho bảng giá trị, trong đó gán các từ mã ngắn cho hầu hết các giá trị có thể xuất hiện. Một từ mã đơn giản là một số nguyên *n-bit* được thiết kế sao cho không nhập nhằng hoặc xung đột

với các từ mã ngắn hơn. Dùng cây nhị phân mở rộng a, c, g, t như hình 1.8, quy ước bit ‘0’ biểu diễn nhánh con trái khi vẽ đường đi và bit ‘1’ biểu diễn nhanh con phải. Ta có cây huffman cho các bazơ như hình 1.9.



Hình 1.9. Các bazơ với mã Huffman

Mã Huffman cho A,C,G,T đi từ gốc tới nút ngoài của chúng.

A = 0

C = 10

G = 110

T = 111

Đặt “s” là một chuỗi các bazơ

F(x) là tần suất của các bazơ, khi đó $x = \epsilon$ {a,c,g,t}

Đường đi có trọng số mở rộng = $\sum_{i=1}^{n} \lambda(I) \quad F(I)$

Trong đó

$\lambda(I)$ = độ dài của đường đi (số cạnh trên đường) từ gốc tới nút ngoài có nhãn *i*

F(I) = Tần suất xuất hiện của các base (a,c,g,t).

Tính số bit trong chuỗi mã hóa = 1*f(a) + 2*f(c) + 3*f(g) + 3*f(t)

Trong đó λ(I) for base a= 1 (since A=0)

λ(I) for base c= 2 (since C=10)

λ(I) for base g= 3 (since G=110)

λ(I) for base t= 3 (since T=111)

- **Thuật toán mã hóa**
  - Xác định các base (a,c,g,t) khác nhau trong chuỗi DNA đưa ra và tính tần suất của chúng.
  - Cấu trúc một cây nhị phân với đường đi có trọng số nhỏ nhất (cây Huffman)
  - Các nút ngoài của cây được gán nhãn là các base trong chuỗi.

- Trọng số của mỗi nút ngoài là tần số của base và là các nhãn {A,C,G,T} của nó.
- Đi qua các nút từ nút gốc tới nút ngoài và nhận được mã.
- Thay thế các kí hiệu trong chuỗi bằng các mã của nó (mã Huffman)

- **Thuật toán giải mã**

Giải mã là quá trình nghịch đảo của mã hóa. Nếu chúng ta có một chuỗi bit 00011111010, ta đọc các bit cho đến khi có một bit khớp trong bảng. Chuỗi ví dụ trên giải mã thành AAATGC. Bảng từ mã được thiết kế để không có xung đột nào xảy ra. Nếu bảng đọc A=0, C=10, G=110, T=111, và ta gặp bit 0 trong một bảng thì sẽ không có cách nào chúng ta có thể nhận được một C vì A sẽ luôn khớp với bit 0 đó.

Phương thức tiêu chuẩn của giải mã một chuỗi mã hóa Huffman là đi qua một cây nhị phân được tạo ra từ bảng từ mã. Khi gặp một bit 0 thì chuyển nó sang bên trái cây và chuyển sang phải nếu gặp bit 1. Đây là phương pháp đơn giản được sử dụng trong bộ giải mã.

- Giải mã các bit nhị phân sẽ cho ra chuỗi bazơ gốc
- Các mã (0,10,110,111) là duy nhất và không có mã nào là tiền tố của các mã còn lại.
- Kết quả khi kiểm tra mã từ trái qua phải sẽ được một phần khớp với một mã chính xác, do đó giải mã có thể sử dụng mã Huffman.
- Không có đường đi nào từ gốc tới ngoài là một tiền tố của đường đi khác.
- Không có mã đường đi nào là tiền tố của mã đường đi khác.

- **Hạn chế**

Kiểm tra trên các chuỗi sinh học thực, thực hiện với công cụ hoặc truyền nhận thức tới các tác vụ tương tự là không thể.

Thuật toán chưa kiểm tra được trên hệ gen lớn.

1.3.2. Thuật toán Expert Markov (XM)

Thuật toán mã hóa mỗi kí tự bằng cách đánh giá xác suất dựa trên thông tin có được từ kí tự trước nó. Nếu kí tự là một phần của chuỗi lặp thì thông tin từ một hoặc nhiều chuỗi trước nó được sử dụng. Khi đã xác định được phân bố xác suất của kí tự thì nó sẽ được mã hóa bởi một thuật toán nén sơ cấp như mã hóa số học.

Là một phương pháp thống kê, thuật toán XM nén mỗi kí tự bằng cách xác định phân bố xác suất cho kí tự và sau đó sử dụng một khung nén sơ cấp để mã hóa nó [15]. Phân bố xác suất tại một vị trí dựa trên kí tự nhìn thấy trước nó.

Tương ứng với nó, bộ giải mã cũng tìm tất cả các kí tự đã giải mã trước nó để có thể tính phân bố xác suất đồng nhất và có thể khôi phục lại kí tự tại vị trí đó.

Để định hình được phân bố xác suất của một kí tự, thuật toán duy trì một tập chuyên gia mà những dự đoán về kí tự của họ được kết hợp trong một phân bố xác suất đơn. Một chuyên gia là một thực thể mà có thể cung cấp một phân bố xác suất tại một vị trí. Các ý kiến chuyên gia về một kí tự được pha trộn để đưa ra một dự đoán kết hợp về kí tự đó.

Thống kê về các kí tự có thể thay đổi trên chuỗi. Một chuyên gia có thể thực hiện tốt trên một số vùng nhưng cũng có thể đưa ra lời khuyên không tốt trên những vùng khác. Một kí tự có thể có những thuộc tính thống kê giống nhau với phạm vi phụ cận, cụ thể là kí tự đứng trước nó. Độ tin cậy của một chuyên gia được đánh giá từ những dự đoán gần đây của nó. Một chuyên gia đáng tin cậy có trọng số cao đối với dự đoán kết hợp còn chuyên gia không đáng tin thì sẽ có ít ảnh hưởng tới dự đoán cuối cùng hoặc bị bỏ qua.

- **Các loại chuyên gia**

Một chuyên gia có thể là bất kỳ thứ gì mà đưa ra được một phân bố xác suất hợp lý cho một vị trí trong chuỗi. Một chuyên gia đơn giản có thể là một mô hình Markov (*Markov expert*). Một chuyên gia Markov thứ tự *k* sẽ đưa ra xác suất của một kí tự ở một vị trí kí tự trước *k.* Đầu tiên, *Markov expert* không có bất kỳ một nhận thức nào về chuỗi và bởi vậy nó đưa ra phân bố đồng nhất cho một kí tự. Phân bố xác suất thích hợp khi thu thập mã hóa. Về cơ bản, *Markov expert* cung cấp phân bố xác suất cơ sở của các kí tự trên chuỗi. Ở đây chúng ta sử dụng *Markov expert* thứ tự 2 cho DNA và thứ tự 1 cho protein.

Những vùng khác nhau của một chuỗi DNA có thể có những chức năng khác nhau và bởi vậy có thể có những phân bố kí tự khác nhau. Một loại chuyên gia khác đó là chuyên gia Markov ngữ cảnh (*context Markov expert)*, phân bố xác suất của chuyên gia này không dựa trên toàn bộ lịch sử của chuỗi mà dựa trên ngữ cảnh hạn chế trước nó. Nói cách khác, *context Markov expert* dựa trên dự đoán của nó về thống kê cục bộ. *Context Markov expert* hiện tại được XM sử dụng là thứ tự 1 với ngữ cảnh là 512 kí tự trước nó.

Khả năng nén các chuỗi sinh học xuất phát từ các chuỗi con lặp. Bởi vậy, các chuyên gia có thể sử dụng được đặc tính này là rất quan trọng. XM sử dụng một sao chép chuyên gia (*copy expert)* mà coi kí tự tiếp theo như một phần của vùng sao chép từ một phần bù cụ thể. Một *copy expert* với phần bù *f* gợi ý kí tự tại vị trí *i* có thể giống như kí tự tại vị trí *i – f.*

Một *copy expert* sẽ không đưa ra một xác suất mù cho vị trí mà nó gợi ý. Nó sử dụng một mã phù hợp trên một số lịch sử gần đây cho những dự đoán

đúng/không đúng. *Copy expert* đưa ra xác suất cho kí tự dự đoán của nó theo công thức sau:

$$p = \frac{r+1}{w+2}$$

Trong đó, *w* là kích thước cửa sổ mà trên đó chuyên gia xem xét hiệu suất của nó và *r* là số dự đoán đúng mà chuyên gia đưa ra. Xác suất phần bù, *1 – p* được phân bố đều tới các kí tự khác trong bảng chữ cái.

Đối với các lặp nghịch đảo phần bù thì một chuyên gia nghịch đảo (*reverse expert)* tương tự được sử dụng. Chuyên gia này làm việc chính xác là giống như *copy expert*, ngoại trừ việc nó gợi ý kí tự phần bù từ trường hợp sớm hơn và tiếp tục theo hướng nghịch đảo.

- **Kết hợp các dự đoán chuyên gia**

Phần lõi của thuật toán XM là việc đánh giá và kết hợp các dự đoán chuyên gia. Giả sử một bảng các chuyên gia *E* sẵn có cho bộ mã hóa. Chuyên gia θ_k đưa ra dự đoán $P(x_{n+1}|\theta_k, x_{1..n})$ của kí tự x_{n+1} dựa trên quan sát *n* kí tự trước nó. Một cách kết hợp các dự đoán chuyên gia dễ thấy là dựa trên trung bình Bayesian:

$$\begin{aligned} P(x_{n+1}|x_{1..n}) &= \sum_{k\in E} P(x_{n+1}|\theta_k, x_{1..n}) w_{\theta_k,n} \\ &= \sum_{k\in E} P(x_{n+1}|\theta_k, x_{1..n}) P(\theta_k|x_{1..n}) \end{aligned} \tag{2}$$

Nói cách khác, trọng số $w_{\theta_k,n}$ của chuyên gia θ_k cho mã hóa x_{n+1} là xác suất hậu nghiệm $P(\theta_k, x_{1..n})$ của θ_k sau khi mã hóa *n* kí tự. $w_{\theta_k,n}$ có thể được đánh giá bằng định lý Bayes:

$$\begin{aligned} w_{\theta_k,n} &= P(\theta_k|x_{1..n}) \\ &= \frac{\prod_{i=1}^{n} P(x_i|\theta_k, x_{1..i-1}) P(\theta_k)}{\prod_{i=1}^{n} P(x_i|x_{1..i-1})} \end{aligned} \tag{3}$$

Nếu giả sử mỗi chuyên gia đều có xác suất tiên nghiệm $P(\theta_k)$, sau đó chuẩn hóa phương trình (3) bằng một hệ số *M*. Ta có:

$$w_{\theta_k,n} = \frac{1}{M} \prod_{i=1}^{n} P(x_i|\theta_k, x_{1..i-1}) \tag{4}$$

Chuẩn hóa hệ số *M*, thực tế không vấn đề gì khi phương trình (2) có thể được chuẩn hóa lại để có $\sum P(x_{n+1}|x_{1..n}) = 1$. Lấy loga phủ định phương trình (4) và bỏ qua giới hạn hằng số:

$$-log_2(w_{\theta_k,n}) \sim -\sum_{i=1}^{n} log_2 P(x_i|\theta_k, x_{1..i-1}) \quad (5)$$

Vì $\log_2 P(x_i|\theta_k, x_{1..i-1})$ là giá trị của kí tự mã hóa x_i bởi chuyên gia θ_k, vế phải của (5) là độ dài của mã hóa chuỗi con $x_{1..n}$ bởi chuyên gia θ_k. Khi ta muốn đánh giá các chuyên gia dựa trên lịch sử kích thước *w* hiện tại, chỉ độ dài thông tin các kí tự mã hóa $x_{n-w+1..n}$ được sử dụng để xác định các trọng số chuyên gia. Thuật toán thực hiện tốt nhất khi phủ định loga cơ số 2 của trọng số chuyên gia biến thiên bằng ba lần độ dài mã trung bình trên cửa sổ kích thước *w = 20:*

$$\begin{aligned} -log_2(w_{\theta_k,n}) &\sim -\frac{3}{w}\sum_{i=n-w+1}^{n} log_2 P(x_i|\theta_k, x_{1..i-1}) \\ &= 3AveMsgLen(x_{n-w+1..n}|\theta_k) \end{aligned} \quad (6)$$

Hoặc

$$w_{\theta_k,n} \propto 2^{-3AveMsgLen(x_{n-w+1..n}|\theta_k)} \quad (7)$$

Giả sử có 3 giả thuyết về cách mà một kí tự được tạo ra: bằng sự phân bố hệ gen các loài; bằng sự phân bố các chuỗi con hiện tại; hoặc bằng cách lặp chuỗi con trước nó. Bởi vậy ta có 3 chuyên gia cho các giả thuyết này: (i) *Markov expert* cho sự phân bố hệ gen các loài, (ii) *context Markov expert* cho phân bố cục bộ, và (iii) *repeat expert* kết hợp sao chép và *reverse expert* cho giả thuyết thứ 3. Các dự đoán của chuyên gia được kết hợp như ở phương trình (2) và (7).

Nếu một kí tự là một phần lặp chính thì *copy* hoặc *reverse expert* của phép lặp phải dự đoán tốt hơn một cách đáng kể so với dự đoán thông thường như từ *Markov expert*. Định nghĩa một giới hạn nghe T để xác định độ tin cậy của một chuyên gia sao chép (*copy expert)* hoặc nghịch đảo *(reverse expert)*. Một chuyên gia sao chép hoặc nghịch đảo được coi là đáng tin cậy nếu độ dài từ mã trung bình của nó nhỏ hơn C_{mk} T bit, trong đó C_{mk} là từ mã trung bình của *Markov expert*. *T* là một tham số của thuật toán.

Thuật toán có thể được sử dụng như một bộ ước lượng entropy hoặc một bộ nén cho chuỗi sinh học. Nội dung thông tin của mỗi kí tự đơn được đánh giá bởi loga phủ định của xác suất của nó. Để nén chuỗi, sử dụng mã hóa số học để mã hóa mỗi kí tự dựa trên phân bố xác suất kết hợp từ các chuyên gia.

XM là một thuật toán nén đơn giản và hiệu quả cho cả DNA và protein. Thuật toán sử dụng các phép lặp xấp xỉ và thuộc tính thống kê của chuỗi sinh học cho việc nén. Như một phương thức nén thống kê, XM có thể tính toán nội

dung thông tin của mỗi kí tự trong chuỗi, điều này rất hữu ích cho việc tìm hiểu về DNA.

1.4. Thuật toán nén tham chiếu

Cũng như kỹ thuật nén dựa trên từ điển, thuật toán này thay thế các chuỗi con dài của đầu vào được nén với tham chiếu tới chuỗi khác. Tuy nhiên, tham chiếu này trỏ tới các chuỗi bên ngoài, không phải là một phần của dữ liệu đầu vào được nén. Ngoài ra, tham chiếu của thuật toán này là tĩnh, còn tham chiếu của thuật toán cơ sở từ điển được mở rộng trong quá trình nén dữ liệu. Do các chuỗi mã hóa tham chiếu tới tập hợp chuỗi tham chiếu bên ngoài nên tốc độ nén cao hơn và giải mã cũng thuận lợi hơn. Các chuỗi DNA được nén tham chiếu bao gồm các phần khớp nhau về khoảng và có thể đạt tới tốc độ nén cao nhất đối với nén trong cùng loài. Tuy nhiên, nếu nén hệ gen khác loài thì sẽ có tỉ lệ sai khác đáng kể và việc mã hóa sẽ tốn dung lượng hơn. Nhìn chung, việc tìm ra chuỗi tham chiếu phù hợp là điều khá khó khăn do các chuỗi gen nghiên cứu là các mẫu được lấy ngẫu nhiên từ một tập hợp lớn các loài. Phương pháp tìm kiếm một chuỗi tham chiếu tốt có thể dựa trên *băm k-mer*. Sự tương đồng cao của *k*-mers đưa ra một tiềm năng lớn cho việc nén dựa trên tham chiếu.

1.4.1. Đặc trưng thuật toán tham chiếu

Thuật toán tham chiếu đề xuất một phương thức nén gen mục tiêu dựa trên một gen tham chiếu đã biết. Đầu tiên thuật toán tạo ra một ánh xạ từ hệ gen tham chiếu tới hệ gen mục tiêu, sau đó nén ánh xạ này bằng bộ mã hóa entropy. Ứng dụng thuật toán này vào hệ gen của James Watson với *hg18* làm tham chiếu, có thể giảm 2991 MB hệ gen xuống còn 6.99 MB, trong khi nén Gzip là 834.8 MB [13].

Để tạo được một ánh từ gen tham chiếu tới gen mục tiêu chính xác có thể coi là một thử thách. Cho một gen tham chiếu và một gen mục tiêu, 2 tệp riêng biệt được tạo ra: một tệp gồm các chuỗi nucleotit đơn (SNPs) và một tệp gồm các phép chèn, xóa nhiều nucleotit (indels). Do đó, ánh xạ giữa các gen được biểu diễn như một kết hợp giữa các phép chèn, xóa và thay thế.

Trên thực tế, các SNPs và tệp indels (các phép chèn, xóa, thay thế) có thể không sẵn có và tạo ra chúng là việc đầu tiên, đặc biệt khi không xác định tập khác biệt nhỏ nhất giữa các gen một cách cần thiết. Bất lợi này thúc đẩy việc phát triển một chương trình nén end-to-end, chương trình sẽ tìm kiếm để tạo ra một ánh xạ hiệu quả giữa gen tham chiếu và gen mục tiêu, sau đó mô tả ánh xạ này theo một cách hiệu quả.

Xem xét vấn đề nén gen mục tiêu X^N dựa trên gen tham chiếu Y^M sẵn có cho cả hai bên mã hóa và giải mã. Ta có X=Y. Mã hóa được biểu diễn bởi ánh xạ $f(X^N, Y^M)$, hàm này chỉ ra phiên bản nén X^N dựa trên Y^M. Giải mã được biểu diễn bởi ánh xạ $g(f(X^N,Y^M), Y^M)$. Do đó nền tảng ở đây là nén không mất dữ liệu, cặp mã hóa – giải mã thỏa mãn

$$X^N = g(f(X^N, Y^M), Y^M).$$

Mục đích là tạo nên một chương trình mã hóa/giải mã tối giản độ dài của ánh xạ $f(-,-)$, trong khi đảm bảo một tái cấu trúc chuỗi X^N hoàn hảo ở giải mã.

Thuật toán mã hóa có thể được chia thành 2 giai đoạn. Đầu tiên là tạo một ánh xạ từ gen tham chiếu Y^M tới gen mục tiêu. Trong giai đoạn 2, ánh xạ này được nén nguyên vẹn sử dụng bộ mã entropy. Sau đó, bộ giải mã sẽ lấy đại diện đã nén của ánh xạ này và giải nén nó để đạt được ánh xạ từ Y^M tới X^N. Do Y^M sẵn có tại bộ giải mã như thông tin phụ. Nó có thể khôi phục X^N một cách hoàn hảo bằng cách nghịch đảo ánh xạ ban đầu.

(a) <u>Tạo ánh xạ</u>

Việc tạo ánh xạ có động cơ là từ thuật toán cửa sổ trượt Lempel-Zip LZ77. Trong LZ77, dữ liệu mới được nén theo kiểu dữ liệu chuyển đổi trước đó. Để khai thác sự giống nhau giữa chuỗi gen tham chiếu và chuỗi gen mục tiêu, ta nén các chuỗi trong chuỗi mục tiêu theo dạng khớp chuỗi đáp ứng trong chuỗi tham chiếu.

Áp dụng phương pháp sử dụng sự xuất hiện của một mảng những thay thế lớn hơn. Mảng thay thế này cùng với hệ gen tham chiếu và hệ gen mục tiêu kết hợp động, cho phép đạt được sự cải thiện đáng kể, trong khi một chương trình chuyển đổi dựa trên LZ77 cũng được sử dụng. Bắt đầu bằng việc mô tả cửa sổ động dựa trên thuật toán khớp chuỗi sau khi đã phân đoạn trong các phần thay thế và dạng xóa/chèn xác định giúp dễ dàng cho việc nén dữ liệu.

Định nghĩa $y_{W_k - L_k}^{W_k + R_k}$ là cửa sổ tại thời điểm k. Sau đó chọn tham số (W_k, L_k, R_k). Ý tưởng cơ bản của thuật toán có thể được tóm tắt như sau: Giả sử $x_1^{n_k}$ đã được mã hóa cho một số n_k. Sau đó, tìm độ dài lớn nhất l_k sao cho $x_{n_k+1}^{n_k+l_k} = y_i^{i+l_k-1}$ và $x_{n_k+l_k+1} \neq y_{i+l_k}$, với $W_k - L_k \leq i \leq W_k + R_k$, mã hóa i, l_k và kí tự mới $x_{n_k+l_k+1}$. Đặt $n_{k+1} = n_k + l_k + 1$ và lặp lại các thủ tục. Do đó, bắt đầu với vị trí trong một cửa sổ cố định trong chuỗi tham chiếu, mã hóa vị trí và độ dài của chuỗi khớp trong tham chiếu cũng như kí tự mới không khớp đầu tiên. Ở cuối giai đoạn này của thuật toán sẽ thu được một tập cấu trúc {F} đủ để tái tạo lại hệ gen mục tiêu dựa trên hệ gen tham chiếu.

Một cách lý tưởng, ta sẽ cho phép mã hóa để tìm kiếm một chuỗi khớp trong toàn bộ chuỗi gen tham chiếu Y^M để tìm chuỗi khớp tốt nhất. Tuy nhiên, về khả năng tính toán thì thuật toán bị hạn chế khi tìm kiếm một cửa sổ xác định, trong khi cập nhật động vị trí và chiều rộng của nó theo hệ gen tham chiếu để xác định hiệu quả việc đồng bộ trong quá trình chuyển đổi giữa hai chuỗi. Quá trình này được mô tả chi tiết như sau:

Cửa sổ tại thời điểm k được biểu diễn bởi các tham số *(W_k, L_k, R_k)*. Theo trực giác thì trung tâm của cửa sổ W_k là sự đánh giá thô về nơi mà mong là tìm thấy một phần khớp trong chuỗi tham chiếu khi tính toán những thay đổi trong việc đồng bộ theo các phép chèn và xóa. Chọn (W_{n_k},L_{n_k},R_{n_k}) là một hàm của (W_{n_i},L_{n_i},R_{n_i}), l_i và p_i (trong đó $i<k$). Lưu ý là không xác định tham số cửa sổ cho tất cả *n*, mà cho hằng số thời gian n_k cụ thể một cách đệ quy. Trên thực tế, sử dụng ý tưởng này để tìm W_{n_k} tối ưu và sử dụng *(L_k, R_k)* cố định.

Ý tưởng chính như sau: Đầu tiên, lưu *M,* l_i và p_i và tăng W_{n_k} bởi l_k. Sau đó nếu p_i tăng (hoặc giảm) quá nhiều thì thay đổi W_{n_k} để giữ p_i gần với trung tâm cửa sổ để giữ cho kích thước của cửa sổ nhỏ. Chính xác hơn là kiểm tra các phần khớp M trước và thay đổi W_{n_k} bằng trung bình của những p_i này. Ở đây lấy trung bình của p_i vì muốn bảo toàn và tránh những thay đổi quá mức, và làm cho thuật toán bật ra các phép chèn/xóa mà không biết. Để giữ thuật toán thực tế, ta thực hiện chỉnh sửa mọi chuỗi khớp M. Đây là sự cân bằng giữa thời gian chạy hợp lý và sự đồng bộ cửa sổ được cải thiện. Ý tưởng này đánh giá tỉ lệ chèn và xóa tương đương tại mỗi thời gian cập nhật dựa trên một số chuỗi khớp trước nhất định [13].

Xác định tập *S (n,W, L, R,* x^∞, y^∞*) = {l:* $x_{n+1}^{n+l} = y_i^{i+l-1}$ *và* $x_{n+l+1} \neq y_{i+l}$, *với* $W - L \leq i \leq W + R$*}*. Thủ tục có thể được mô tả như sau:

- **Thuật toán tạo ánh xạ**

  Khởi tạo: $n_1 = 0$ *và* $k = 1$

  Thực hiện:

  1. *Tính* $S_k = S(n_k, W_k, L_k, R_k, x^\infty, y^\infty)$
  2. *Biểu diễn cụm thứ k bởi* $x_{n_k+1}^{n_k+l_k+1}$*, trong đó* $l_k = max\ S_k$
  3. *Đặt* $p_k = arg\ min_i\ S_k$ *và* $z_k = x_{n_k+l_k+1}$
  4. *Lưu* $F_k = (p_k, l_k, z_k)$
  5. *Đặt* $n_{k+1} = n_k + l_k + 1$, $k = k+1$ *và cập nhật tham số cửa sổ* (W_k, L_k, R_k)
  6. *Lặp lại quá trình cho đến khi chuỗi* x_1^N *trống.*

Ví dụ 1.

Xem xét ví dụ về gen mục tiêu và tham chiếu sau:

Mục tiêu: AATGCAGGTACTATAAGNAANT GC ...

Tham chiếu: AATGTAGGTACATAAGATGCNNNN...

Trong trường hợp này, tập cấu trúc {F} được viết như sau:

F_1: $(p_1, l_1, z_1) = (1, 4, C)$

F_2: $(p_2, l_2, z_2) = (6, 6, T)$

F_3: $(p_3, l_3, z_3) = (12, 5, N)$

F_4: $(p_4, l_4, z_4) = (14, 2, N)$

...

Tập cấu trúc {F} ở trên mới sử dụng một phần gen tham chiếu đủ để cấu trúc lại gen mục tiêu. Thực hiện một bước tiếp theo để xác định rõ ràng các phép chèn, xóa và thay thế từ gen tham chiếu tới gen mục tiêu. Việc này được thực hiện trong trường hợp hầu hết sự khác nhau giữa hai gen là do các phép thay thế, bước này là để giảm số lượng phần tử lưu trữ khi nén gen.

Nói cách khác, mục đích là để giảm kích thước {F} bằng cách xen lẫn các cấu trúc và tạo ra các cấu trúc mới theo dạng thay thế, chèn, xóa lần lượt được lưu trong các tập {S}, {I}, {D}. Với phép thay thế và chèn, chỉ cần lưu vị trí mà tại đó các phép này xảy ra trên gen mục tiêu và kí tự mới. Với phép xóa, cần lưu vị trí của trường hợp trước nó và độ dài của nó, độ dài này không được vượt quá giá trị lớn nhất L_{max} đã xác định trước đó. Sau đây là cách phân loại phép sửa trong những loại này.

Để xác định các phép thay thế, kiểm tra điều kiện giữa F_k và F_{k+1} sao cho:

$$p_k + l_k + 1 = p_{k+1} \quad (1)$$

Nếu (1) xảy ra, ta có

$$x_{n_k+1}^{n_{k+1}+l_{k+1} \setminus n_{k+1}} = y_{p_k}^{p_{k+1}+l_{k+1}-1 \setminus p_{k+1}-1} \quad (2)$$

Và

$$x_{n_{k+1}} \neq y_{p_{k+1}-1} \quad (3)$$

Biểu diễn một thay thế tại $x_{n_{k+1}}$ từ $y_{p_{k+1}-1}$ tới z_k . Sau đó thay thế cấu trúc F_k và F_{k+1} bởi một cấu trúc duy nhất $(p_k, l_k + l_{k+1} + 1, z_{k+1})$, và thêm thay thế mới vào tập {S} khi

$$(p^{(s)}, z^{(s)}) = (n_{k+1}, z_k)$$

Trong đó $p^{(s)}$ và $z^{(s)}$ lần lượt cho biết vị trí tuyệt đối ở gen mục tiêu và kí tự mới. Phép chèn 1 độ dài xảy ra nếu điều kiện sau xảy ra:

$$p_k + l_k = p_{k+1} \quad (4)$$

Trong trường hợp này, ta có:

$$x_{n_k}^{n_{k+1}+l_{k+1}} = y_{p_k}^{p_{k+1}+l_{k+1}-1} \quad (5)$$

Với phép chèn z_k tại vị trí $x_{n_{k+1}}$. Như ở trên, nếu điều kiện này xảy ra thì sẽ thay thế 2 cấu trúc gốc bằng $(p_k, l_k + l_{k+1}, z_{k+1})$ và thêm phép chèn mới

$$(p^{(i)}, z^{(i)}) = (n_{k+1} - 1, z_k)$$

vào tập {I}

Sau cùng, các phép xóa được tìm thấy bằng việc kiểm tra thỏa mãn

$$2 \leq p_{k+1} \quad (p_k + l_k + 1) \leq L_{max} \quad (6)$$

Và

$$z_k = y_{p_{k+1}-1} \quad (7)$$

Nghĩa là có thể cấu trúc lại $x_{n_k+1}^{n_{k+1}+l_{k+1}}$ từ $y_{p_k}^{p_{k+1}+l_{k+1}-1}$ bằng việc xóa $y_{p_k+l_k}^{p_{k+1}-2}$. Do đó, hai cấu trúc sẽ thành $p_k, p_{k+1} + l_{k+1} \quad p_k, z_{k+1}$ và thêm phép xóa

$$\left(p^{(d)}, l^{(d)}\right) = (n_k + l_k, p_{k+1} \quad 1 \quad p_k \quad l_k$$

Vào tập {D}, trong đó $p^{(d)}$ và $l^{(d)}$ lần lượt biểu diễn vị trí trong gen mục tiêu mà tại đó phép xóa xảy ra và độ dài của nó.

Phương thức mô tả ở trên có thể áp dụng cho một số bất kỳ các cấu trúc liên tiếp mà thỏa mãn một điều kiện cụ thể. Không khó để thấy được cách mà các cấu trúc mới được tạo ra trong trường hợp đó.

(b) <u>Mã hóa Entropy</u>

Sử dụng mã hóa entropy để nén và mô tả hiệu quả các tập *{F}, {S}, {I}* và *{D}*. Cụ thể là cần lưu toàn bộ các kí tự và số nguyên xuất hiện trong các tập này, mỗi tập được xử lý riêng biệt.

Mỗi cấu trúc trong tập {F} có 2 số nguyên, để biểu diễn vị trí và độ dài. Với các số nguyên biểu diễn vị trí, thực hiện mã hóa delta, nghĩa là với mỗi vị trí sẽ mã hóa sự khác nhau giữa vị trí hiện tại và vị trí trước đó. Ngoài ra, vì chúng không thể xuất hiện theo thứ tự tăng dần nên ta giữ 1 bit cho mỗi số nguyên để chỉ ra kí hiệu. Đối với độ dài, không thực hiện mã hóa delta. Mặt khác, các số nguyên trong các tập {S, I} xuất hiện theo thứ tự tăng dần, bởi vậy mã hóa delta được thực hiện mà không giữ lại bit kí hiệu. Cuối cùng, với mỗi đầu vào của tập {D} sẽ có 2 số nguyên lần lượt mô tả vị trí và độ dài của mỗi phép xóa. Danh sách số nguyên đầu tiên được sắp xếp và bởi vậy mà thực hiện mã hóa delta. Sau đó thêm vào độ dài phép xóa để trả ra danh sách kết quả. Sử dụng phương pháp này có thể tạo ra một danh sách toàn bộ các số nguyên cần để mô tả cho bộ giải mã. Sau cùng nén danh sách này sử dụng mã hóa Huffman.

1.4.2. Các thuật toán nén tham chiếu

Có nhiều khung nén tham chiếu đã được phát triển dựa trên thuật toán nén tham chiếu. Khung nén [32] đề xuất chỉ lưu sự khác nhau giữa chuỗi đầu vào nén và chuỗi tham chiếu. Thuật toán xem xét 3 loại thay đổi bazơ đơn: chèn, xóa và thay thế. Kết quả chính đạt được là sự phân tích về cách mã hóa các số nguyên cho vị trí tham chiếu tương đối và tuyệt đối. Cụ thể là thuật toán so sánh các định dạng mã hóa entropy cố định (Golomb, Elias) và biến đổi (Huffman) và đưa ra kết quả mã hóa Huffman thực hiện tốt hơn Golomb và Elias không đáng kể. Tuy nhiên, thuật toán này nhấn mạnh việc lựa chọn chuỗi tham chiếu có ảnh hưởng tới tỉ lệ nén nhiều hơn khung mã hóa số nguyên trên thực tế.

Tương tự, [33] trình bày một thuật toán nén tham chiếu mà chỉ xem xét các SNPs và các phép INDELs (các phép chèn, xóa, thay thế) nhiều bazơ giữa đầu vào và các chuỗi tham chiếu. Mỗi đầu vào nén gồm một tham chiếu vị trí và dữ liệu bổ sung như độ dài chuỗi khớp hoặc chuỗi bazơ thô. Các số nguyên độ dài biến đổi được sử dụng để mã hóa vị trí nơi mà bit cuối cùng trong một byte được sử dụng như một bit dừng. Mã hóa Huffman được sử dụng để nén *k-mers* thông thường. Thuật toán đưa ra cách tối ưu để cải thiện tỉ lệ nén. Ngoài chuỗi tham chiếu thì thuật toán cần một ánh xạ kích thước SNP tham chiếu khoảng 1GB.

GRS [20] là một công cụ nén tham chiếu khác dựa trên chương trình Unix *diff,* tìm kiếm chuỗi con dài nhất trong hai chuỗi đầu vào. Trong GRS, *diff* được sử dụng để tính độ tương đồng giữa nhiễm sắc thể đầu vào và nhiễm sắc thể tham chiếu. Nếu sự tương đồng vượt quá một giới hạn đưa ra thì sự khác nhau giữa chuỗi đầu vào và chuỗi tham chiếu được nén sử dụng mã hóa Huffman. Ngược lại, nhiễm sắc thể đầu vào và tham chiếu được tách thành các khối (block) nhỏ hơn và việc tính toán được bắt đầu lại trên từng cặp khối.

RLZ [30] mô tả một phương pháp dựa trên việc tự đánh chỉ số. Thuật toán hoạt động như sau: thuật toán nén chuỗi đầu vào với mã hóa LZ77 liên quan tới mảng hậu tố của chuỗi tham chiếu. Không lưu các chuỗi thô kể cả đó có là chuỗi khớp ngắn với tham chiếu được mã hóa. Thuật toán coi việc xem xét các chuỗi tham chiếu là vấn đề sống còn. RLZopt [24] được mô tả là một mở rộng của RLZ. Hướng chính là tính toán chuỗi con dài nhất mà cho phép mã hóa vị trí hiệu quả. RLZopt hỗ trợ truy vấn ngẫu nhiên.

GreEn [31] một khung nén tham chiếu dựa trên hệ chuyên gia mới được đề xuất gần đây. Lấy ý tưởng từ khung nén XM (eXpert Markov) không tham chiếu, GreEn sao chép hệ chuyên gia tìm kiếm chuỗi khớp *k-mers* giữa đầu vào và tham chiếu. Các kí tự thô ở dạng ASCII được mã hóa với mã hóa số học. Thuật toán phân biệt các trường hợp đặt biệt, trong đó chuỗi đầu vào và tham

chiếu có độ dài bằng nhau. Trong trường hợp này, GreEn giả sử rằng các chuỗi đã được sắp xếp và chỉ mã hóa SNPs.

Ngoài ra còn có các phương pháp nén tham chiếu khác như: Hệ thống dựa trên web [34], khung nén LZ77-style với truy cập ngẫu nhiên [35], các phương pháp dựa trên cấu trúc chỉ số cố định [36, 37].

Vấn đề chính của thuật toán nén tham chiếu là tìm được chuỗi khớp dài và chuỗi tham chiếu hiệu quả. Nếu đạt được những điều kiện này thì hiệu quả của thuật toán nén tham chiếu có thể nói là tốt hơn các thuật toán khác một bậc. Vượt lên trên nhiều khung nén tham chiếu với nhiều cải thiện đạt được tỉ lệ nén và không gian lưu trữ, JDNA được biết đến là một khung nén tham chiếu hiệu quả, xây dựng dựa trên một thuật toán nén tham chiếu nhanh và mã nguồn được mở để cộng đồng cùng sử dụng và cải tiến. JDNA sử dụng một bảng *k-mer* để đánh chỉ số cho chuỗi tham chiếu, JDNA tốn ít thời gian nén, phần lớn thời gian thực hiện là dùng cho việc đánh chỉ số. JDNA sử dụng thư viện mã nguồn mở được dùng bởi FRESCO [21] đáp ứng được 4 thiết kế chính khi thực hiện một thuật toán nén tham chiếu đó là: (1) định dạng đầu vào, (2) cấu trúc chỉ số tham chiếu, (3) thuật toán nén và (4) định dạng thứ tự chuỗi cho các tệp nén [25]. Bên cạnh đó JDNA còn thêm hai cải tiến để tối ưu về thời gian nén/giải nén và dung lượng lưu trữ là (1) sử dụng tính tương đương và (2) thay thế chỉ số tham chiếu hoàn toàn bằng một phương thức chỉ số theo yêu cầu. Giao diện chuỗi của khung nén cho phép tải chuỗi từ tệp và viết chuỗi vào tệp với định dạng dữ liệu RAW hoặc FASTA. Một chỉ số *index* dựa trên chỉ số băm k-mer được sử dụng để tìm chuỗi khớp cho chuỗi nén dựa trên tham chiếu. Tiếp theo giao diện nén sẽ cung cấp sự thực hiện hai hàm, một cho nén chuỗi vào danh sách các đầu vào khớp tham chiếu và một hàm khác cho giải nén. Cuối cùng là thực hiện chuỗi hóa, sắp xếp một danh sách khớp tham chiếu vào một tệp với xử lý 3 chuẩn: (1) định dạng ASCII, (2) mã hóa DELTA, và (3) dạng nhị phân rút gọn (COMPACT). Có thể nói JDNA đạt được hiệu quả nén chuỗi đa lượng đáng mong đợi và có tiềm năng mở rộng, cải tiến khá lớn. Ở chương 2, người viết sẽ tập trung trình bày chi tiết về khung nén JDNA, cách mà khung nén này kế thừa những đặc trưng của FRESCO và những cải tiến mà thuật toán đã thực hiện để mang lại hiệu quả về tỉ lệ nén cũng như dung lượng lưu trữ khi nén chuỗi gen.

CHƯƠNG 2 – THUẬT TOÁN NÉN THAM CHIẾU JDNA

Kể từ khi phát hành hệ gen người đầu tiên [17], chi phí cho việc sắp xếp chuỗi gen đã giảm nhanh chóng. Ngày nay, giá xấp xỉ 2000 USD mỗi hệ gen và mong là sẽ giảm trong tương lai khi mà kỹ thuật sắp xếp chuỗi gen thế hệ thứ ba trở nên phổ biến [18]. Ngược lại với những năm trước khi mà chỉ có một cá thể của loài được sắp xếp chuỗi gen (như người, chuột, vi khuẩn E.coli…) thì chi phí giảm khiến cho việc sắp xếp các mẫu lớn trở nên phổ biến. Những nghiên cứu về sắp xếp gen, đặc biệt là hệ gen người đã và đang rất được quan tâm từ nhiều góc độ, như sự biến đổi gen ở các loài gây nên nhiều bệnh tật, đo liều lượng thuốc hay đơn giản là để hiểu rõ hơn về mối liên hệ giữa kiểu gen và hiện tượng gen. Dữ liệu về gen ngày càng tăng khiến cho việc quản lý, lưu trữ và phân tích ngày càng trở nên phức tạp.

Để lưu toàn bộ một hệ gen người cần khoảng 3GB không gian lưu trữ, sử dụng 1 byte cho mỗi nucleotit. Sử dụng 8 bit có thể mã hóa 256 kí tự khác nhau, không gian này có thể được giảm bằng cách mã hóa mỗi nucleotit với ít hơn 8 bit. Thuật toán nén thay thế hoặc thống kê có thể giảm không gian lưu trữ tới 6:1 (mỗi bazơ được mã hóa với 1,3 bit). Tuy nhiên, trong nhiều dự án chỉ các gen trong cùng loài được xem xét. Điều này có nghĩa là các dự án thường xử lý hàng trăm gen có độ tương đồng cao; ví dụ hai gen người lựa chọn ngẫu nhiên được ước lượng là có sự tương đồng tới 99,9%. Sự tương đồng giữa các chuỗi có thể được khai thác sử dụng thuật toán nén gọi là nén tham chiếu, phương pháp nén mà mã hóa những phần khác nhau của chuỗi đầu vào dựa trên một chuỗi tham chiếu. Sử dụng mã hóa những khác biệt hiệu quả về không gian và các thuật toán thông minh để tìm các chuỗi DNA dài mà không có khác biệt, thuật toán nén tham chiếu tốt nhất hiện nay được biết là có tỉ lệ nén trong khoảng 500:1 tới 1000:1 đối với hệ gen người [19].

Do những lợi ích và hiệu quả mà JDNA đạt được đối với nén chuỗi gen theo phương pháp nén tham chiếu mà ở chương này, người viết luận văn lựa chọn trình bày thuật toán nén tham chiếu JDNA, một khung nén tham chiếu xây dựng trên thuật toán nén tham chiếu nhanh và mã nguồn mở của FRESCO được phát hành miễn phí cho cộng đồng sử dụng và mở rộng. Hiệu quả thực hiện đạt được tỉ lệ nén cao hơn các thuật toán thuộc các phương thức khác và cao hơn cả FRESCO một bậc. Thêm vào đó, người viết sẽ trình bày những đặc trưng của FRESCO mà JDNA đã kế thừa, đồng thời trình bày những đặc trưng mà JDNA đã cải tiến và mang lại hiệu quả thực sự về tỉ lệ nén và dung lượng lưu trữ cũng như tốc độ nén chuỗi gen.

Thuật toán được đánh giá trên tập dữ liệu từ 3 loài: 1092 gen người, 180 gen loài cỏ Arabidopsis thaliana và 38 gen khuẩn men.

2.1. THUẬT TOÁN JDNA - Nén tham chiếu các chuỗi gen đã sắp xếp

Như đã trình bày ở phần trước, bước đánh chỉ số chiếm phần lớn thời gian thực hiện. Hình 2.3 cho thấy trên 95% thời gian thực hiện là đánh chỉ số, trong khi chỉ khoảng 2% cho nén và 3% cho các bước còn lại [21]. Từ quan sát này có thể thấy được hai cơ hội tối ưu: (1) sử dụng tính tương đương và (2) thay thế chỉ số tham chiếu hoàn toàn bằng một phương thức chỉ số theo yêu cầu. Ở giải pháp thứ nhất, việc tạo ra cấu trúc chỉ số sẽ được thực hiện song song sử dụng cơ chế đồng bộ khi lưu trữ các chỉ số và phần băm. Tuy nhiên sẽ bị hạn chế bởi dung lượng máy và có thể không cần toàn bộ cấu trúc chỉ số. Trong trường hợp thứ hai, thay vì đánh chỉ số tham chiếu đầy đủ khi bắt đầu thực hiện chương trình thì phương thức so sánh chuỗi đơn giản sẽ được sử dụng để tìm ra chuỗi khớp. Phương thức thứ hai được lựa chọn để cải thiện thuật toán, gọi là *đánh chỉ số theo yêu cầu*, tuy nhiên cả hai phương pháp là bổ sung cho nhau.

Phương pháp đánh chỉ số theo yêu cầu được phát triển và cung cấp một công cụ nén dùng ngôn ngữ Java, gọi là JDNA cho nén tham chiếu sử dụng phương pháp này. JDNA được thiết kế để áp dụng nén tham chiếu cho các tệp gen. Có bốn loại biến gen được JDNA hỗ trợ là: SNPs, thay thế, chèn và xóa.

- SNP: một sai khác cặp bazơ đơn ở cùng vị trí từ các gen khác nhau.
- Thay thế (*Substitution)*: một chuỗi các cặp bazơ mà khác nhau ở cùng vùng của hai gen.
- Chèn (*Insertion*): một chuỗi các cặp bazơ mà được thêm vào một vùng gen. Do đó, trong cùng vùng của DNA khác thì chuỗi cặp bazơ đó không được biểu diễn.
- Xóa (*Deletion)*: một chuỗi cặp bazơ không được biểu diễn trong một vùng gen nhưng được biểu diễn trong phần chính của gen.

Một phần của SNPs khác với các cặp bazơ có các biến gen có thể thay đổi về kích thước. Định nghĩa một giá trị θ sử dụng để phân biệt một biến là nhỏ hay lớn. Ở luận văn này, giá trị θ là 300, đoạn biến kích thước là nhỏ (*SD*) nếu nó nhỏ hơn θ và là lớn (*BD*) nếu nó lớn hơn θ. Chỉ số theo yêu cầu dựa trên thực tế là các gen từ cùng một loài thì có sự tương đồng về di truyền cao, vì vậy hầu hết các biến là *SD* và có thể dễ dàng được tìm thấy bằng cách sử dụng phương pháp so sánh đơn giản.

2.1.1. Thuật toán nén

Nén thực hiện ở các khối tệp hoặc với toàn bộ tệp trong bộ nhớ. JDNA được thiết kế để tải tệp lên tới 250MB lên bộ nhớ với các tệp lớn hơn tải vào các khối 250MB. Cả tệp tham chiếu và đầu vào đều được tải theo cách này. Mỗi khối được nén riêng biệt, bởi vậy không có sự kết nối nào giữa các khối nén thậm chí chúng ở cùng một tệp. Nếu nén khối được sử dụng thì chỉ một tệp nén được tạo ra gồm các khối được nén.

Trong JDNA, chuỗi khớp được tìm thấy trong 3 bước:

1. Kiểm tra xác định có một SNP hay không.
2. Nếu nó không phải một SNP thì thực hiện tìm kiếm cục bộ khoảng cặp bazơ cho một chuỗi khớp – khoảng này gồm hầu hết SDs; là 6 lần K, trong đó K là kích thước K-mer.
3. Nếu không tìm thấy chuỗi khớp nào thì đánh chỉ số một phần tham chiếu trong bảng K-mer và tìm kiếm vị trí hiện tại của đầu vào trong bảng.

Nếu sau các bước trên mà vẫn không tìm thấy chuỗi khớp nào thì ghi lại một cặp bazơ của đầu vào và lặp lại quá trình. Có thể thấy rằng mỗi lần đánh chỉ số tham chiếu một chút sẽ tốt hơn vì nếu có một đoạn chèn lớn ở tệp đầu vào thì có thể sẽ đánh chỉ số nhiều tham chiếu mà không để làm gì cả. Nếu một chuỗi khớp được tìm thấy (bất kì loại nào) thì sẽ tìm kiếm xác định chuỗi khớp đó dài bao nhiêu để nén và sau đó ghi lại chuỗi khớp.

Tệp đầu vào được xử lý tìm kiếm chuỗi khớp trong tham chiếu cho đến khi toàn bộ đầu vào đã được so sánh.

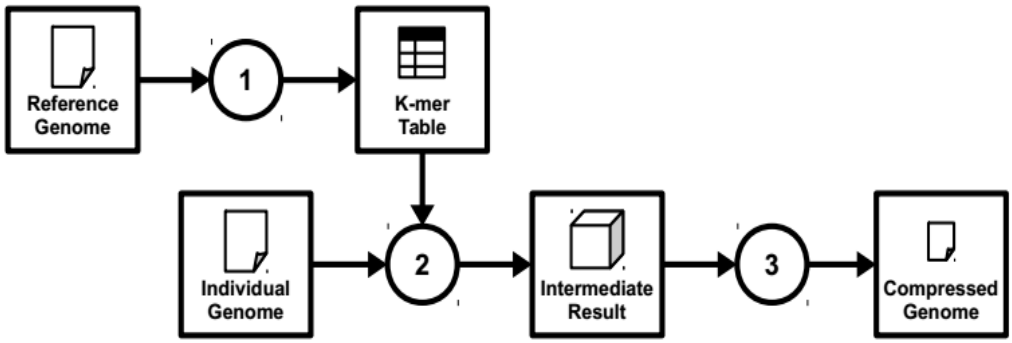
Lợi ích chính của cấu trúc thuật toán này là sử dụng chỉ số theo yêu cầu. Vì các gen giống nhau tới 99.5% nên không cần một cấu trúc chỉ số để tìm phần chính của chuỗi khớp; chỉ khi hầu hết các bước trực tiếp bị thất bại (bước 1 và 2) thì cần phải đánh chỉ số.

Mã hóa Huffman. Sử dụng mã hóa Huffman [45] khi ghi lại chuỗi khớp để tăng nén bằng cách giảm kích thước của những phần ghi vào đĩa. Ví dụ, thay vì ghi một số nguyên (32 bits) thì ghi một số bít mô tả số nguyên. Sau khi sử dụng mã hóa đã nói thì kết quả được nén kỹ hơn bằng GZIP.

2.1.2. Thư viện FRESCO

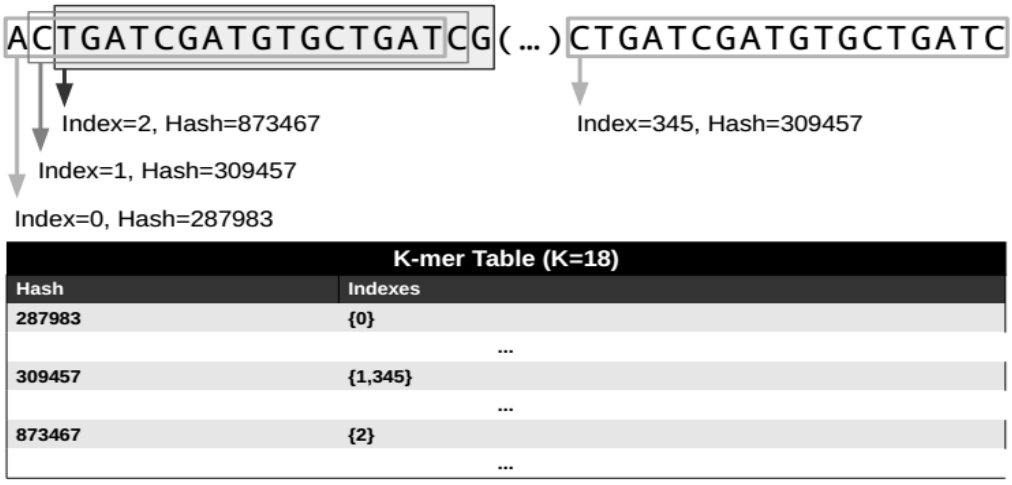
Như đã nói ở đầu chương, JDNA phát triển và cải tiến dựa trên thuật toán tham chiếu và mã nguồn mở FRESCO nên thư viện FRESCO cũng được JDNA sử dụng như thư viện chính cho chương trình. FRESCO là một thư viện nén tham chiếu không mất dữ liệu cho các tệp gen được sắp xếp, lưu ở định dạng RAW hoặc FASTA. Được viết bằng C++ và là mã nguồn mở [25]. FRESCO gồm những biến phát sinh sau: *single nucleotide polymorphisms* (SNPs), phép

chèn, phép xóa và phép thay thế. Nó thực hiện ba bước bên trong (hình 2.1 [21]): (1) đánh chỉ số; (2) nén chính nó và (3) mã hóa.



Hình 2.1. Mô hình các bước thực hiện của FRESCO.
(1) Đánh chỉ số gen tham chiếu. (2) Nén gen đầu vào, sử dụng tham chiếu đã đánh chỉ số. (3) Mã hóa các kết quả sơ bộ để tạo ra tệp cuối cùng

(1) Đánh chỉ số. FRESCO sử dụng một cấu trúc dữ liệu gọi là *bảng K-mer* [26], để đánh chỉ số gen tham chiếu hoàn chỉnh. Bảng này giống như một bảng băm, khác ở chỗ nó lưu nhiều giá trị trên mỗi khóa, như được chỉ ra ở hình 2.2 [21]. Việc băm mỗi đoạn với kích thước K được tính toán và lập chỉ số khi mà đoạn được tìm thấy trong tham chiếu được lưu trong danh sách được đánh chỉ số bởi bảng băm này. Khi tất cả các đoạn kích thước K từ tham chiếu được xử lý thì cấu trúc có thể được sử dụng để tìm chuỗi khớp giữa tham chiếu và các tệp đầu vào. Chỉ số tham chiếu hoàn chỉnh cung cấp một thuộc tính tìm kiếm mang tính quyết định vì có thể chắc chắn được đoạn K có hay không có trong tham chiếu.

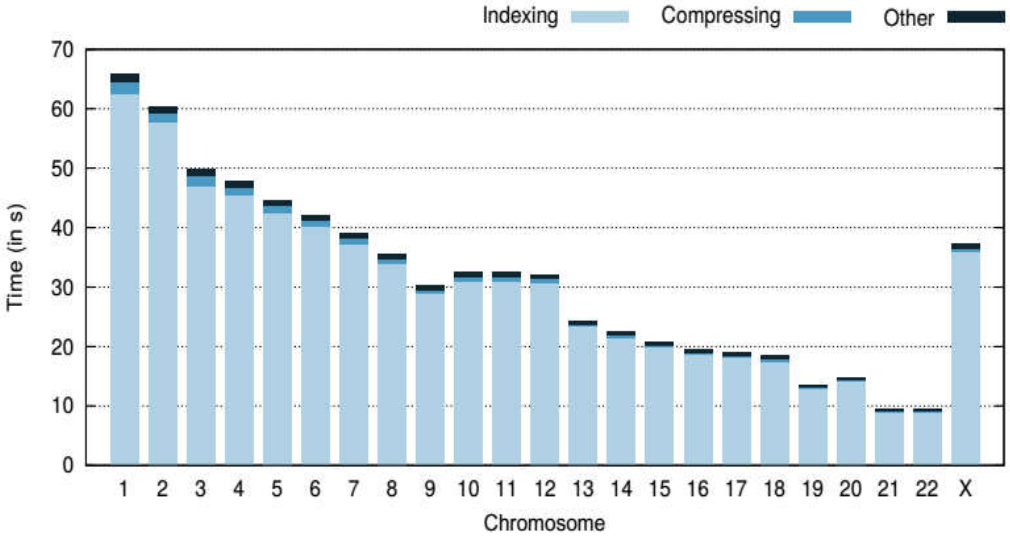


Hình 2.2. Mô hình chỉ số sử dụng bảng K-mer.
Cột đầu tiên lưu phần băm tương ứng với một hoặc nhiều chỉ số và những chỉ số này như được lưu ở cột thứ hai

(2) Nén. Pha nén sử dụng cấu trúc chỉ số chỉ dẫn và gen sẽ-được-nén. Một đoạn đầu vào kích thước K được băm và được tìm kiếm trong bảng K-mer. Tìm kiếm thành công trả ra kết quả một tập các chỉ số, nơi mà các đoạn có thể được tìm thấy trong tham chiếu. Chỉ số tạo ra chuỗi khớp dài nhất được chọn, sau đó chuỗi khớp được mở rộng. Khi chuỗi khớp kết thúc thì một đầu vào mới được tạo ra ở kết quả trung gian. Đầu vào này được tạo nên bởi kích thước chuỗi khớp mà nó bắt đầu và khác các cặp bazơ cho đến chuỗi khớp mới. Sau đó, một chuỗi khớp mới được tìm kiếm trong bảng, phương pháp này nhắc lại cho đến khi toàn bộ tệp đầu vào được xử lý.

(3) Hệ thống hóa. Sau cùng, những kết quả trung gian được nén qua mã hóa kí tự. FRESCO sử dụng GZIP để nén các kết quả trung gian, tạo ra kết quả cuối cùng là tệp đầu ra.

Hình 2.3 [21] biểu diễn một phân tích về thời gian thực hiện của FRESCO đối với các nhiễm sắc thể khác nhau của một hệ gen. Mỗi thanh của đồ thị được tạo nên bởi 3 bước chi tiết trước đó. Các thành phần khác được tạo nên bởi một vài chức năng nhỏ, ví dụ đọc tệp, thời gian viết, mã hóa và các điều khiển khác như khởi tạo biến. Hình 2.3 cho thấy hầu hết thời gian thực hiện của FRESCO được sử dụng cho pha đánh chỉ số. Các bước khác thể hiện phần nhỏ hơn so với đánh chỉ số.



Hình 2.3. Thời gian thực hiện của FRESCO cho các bước

Do chỉ số thể hiện trên FRESCO là khá lớn, nên một tối ưu được đề xuất để giảm tổng thời gian thực hiện. Lưu ý là tốc độ của FRESCO nhanh hơn nhiều so với những đối thủ của nó và còn có thể được cải thiện.

Cuối cùng là trong quá trình giải nén, hệ gen gốc được khôi phục lại từ tệp nén và tham chiếu cùng một hệ gen mà nó sử dụng trong quá trình nén. Tệp nén

ban đầu được tạo bằng cách trỏ tới những phần tham chiếu mà khớp với tệp đầu vào và các cặp bazơ đáp ứng với phần khác nhau giữa các tệp. Chuỗi khớp và các cặp bazơ khác nhau được xen kẽ. Với mỗi phần tạo nên bởi một chuỗi khớp và một biến đổi thì chuỗi khớp được viết, một chuỗi từ tham chiếu được viết ở đầu ra và sau đó các cặp bazơ khác nhau được viết trực tiếp ở đầu ra. Ở đây sự tối ưu hóa không tập trung vào quá trình giải nén.

- **FRESCO – Mã nguồn mở**

FRESCO – Framework for REferential Sequence Compresion, là tên một mã nguồn mở. Phần mềm có tại https://github.com/hubsw/FRESCO.git. FRESCO viết bằng C++, sử dụng thư viện BOOST, CST[91] và libz. FRESCO được thiết kế theo một module để dễ thay thế các phần của thuật toán nén, ví dụ cấu trúc chỉ số với các thực nghiệm khác nhau. Những lựa chọn thiết kế chính khi thực hiện một thuật toán nén tham chiếu là 1) định dạng đầu vào, 2) cấu trúc chỉ số cho tham chiếu, 3) thuật toán nén, ví dụ: tham ăn (greedy) và 4) định dạng thứ tự chuỗi cho các tệp nén, nghĩa là mã hóa thực tế các chuỗi khớp. Với các thuật toán nén đã có, người viết luận văn khi lập trình đã lựa chọn các tiêu chí này trong lúc thiết kế. FRESCO gồm giao diện cho mỗi phần trong 4 phần trên và cho phép sử dụng những thực hiện thay thế khác nhau và thêm vào các thuật toán mới, chuyên dụng. Sau đây, mỗi giao diện và sự thực hiện tiêu chuẩn trong FRESCO sẽ được mô tả chi tiết.

Giao diện chuỗi định nghĩa hai chức năng: một cho việc tải chuỗi từ tệp và một cho việc viết chuỗi vào tệp. FRESCO cung cấp sự thực hiện cho việc xử lý tệp thô (một byte cho một kí tự) và FASTA file.

Một chỉ số *index* được sử dụng để tìm chuỗi khớp cho chuỗi nén dựa trên tham chiếu. Chỉ số được khởi tạo từ một chuỗi tham chiếu cho trước, ví dụ tải từ một FASTA file. Giao diện khai báo một hàm tìm kiếm khớp tiền tố dài nhất của một chuỗi đầu vào dựa trên tham chiếu được đánh chỉ số. Trong FRESCO, chúng ta cung cấp một thực hiện tiêu chuẩn dựa trên một chỉ số băm k-mer, nghĩa là với mỗi k-mer ta lưu tất cả các trường hợp xảy ra trong chuỗi tham chiếu. khi cần một chuỗi khớp cho một chuỗi thành phần thì tiền tố k-mer của chuỗi thành phần được sử dụng để tìm chuỗi khớp dài nhất trong chuỗi tham chiếu. Sự thực hiện khác có thể sử dụng mảng hậu tố như trong [24].

Giao diện nén định nghĩa hai hàm: một cho nén chuỗi vào danh sách các đầu vào khớp tham chiếu và hàm khác để giải nén đầu vào khớp tham chiếu trở lại thành chuỗi. FRESCO cung cấp 3 thuật toán nén: 1) tham lam (BAS) – luôn tìm chuỗi khớp dài nhất có thể; 2) một tối ưu cho tìm kiếm chuỗi khớp cục bộ mà không cần tìm chỉ số (LO); và 3) một tối ưu ưu tiên ngắn nhưng khớp cục bộ

trên các chuỗi khớp dài hơn xa khỏi chuỗi khớp trước đó (LO_MD), phương pháp được đưa ra ở [19].

Chuỗi hóa sắp xếp một danh sách khớp tham chiếu tới/từ một tệp. FRESCO có 3 xử lý tiêu chuẩn: 1) định dạng ASCII đơn giản (PLAIN); 2) mã hóa đơn giản với những vị trí được mã hóa có liên quan tới các chuỗi khớp trước (DELTA) [19]; 3) mã hóa nhị phân rút gọn (COMPACT).

2.1.3. Bảng K-mer

Việc nén dữ liệu cần một cấu trúc đánh chỉ số. Đầu tiên, JDNA sử dụng đối tượng cổ điển (như HashMap) để đánh chỉ số. Tuy nhiên, bộ nhớ sử dụng cho các đối tượng phức tạp là rất lớn. Khai báo một *HashMap* 50000 ngăn và cung cấp cấu trúc tương ứng làm cho việc sử dụng bộ nhớ tăng quá 20GB. Cần giữ bộ nhớ sử dụng nhiều nhất là 2GB-2.5GB để JDNA cạnh tranh được với FRESCO ở mọi khía cạnh.

Một bảng băm gọi là bảng K-mer được sử dụng để lưu thông tin liên quan tới thuật toán nén. Bảng này giống với bảng FRESCO minh họa ở hình 2.2. Bảng K-mer giống với một cấu trúc *MultiValueMap,* lưu trữ nhiều giá trị trên mỗi khóa thay vì chỉ một giá trị. Bảng K-mer là một bảng *table* ma trận số nguyên và một mảng đếm *integer* số nguyên. Trong Java, một ma trận số nguyên là một mảng sơ cấp của nhiều mảng thứ cấp số nguyên. Mỗi vị trí trong mảng sơ cấp bắt đầu với giá trị *null.* Sau đó, dựa theo một *ArrayList* trong mỗi vị trí mảng sơ cấp để chèn giá trị vào *table* bằng cách tạo ra một mảng kích thước cố định và sử dụng biến đếm *counter* để điều khiển số phần tử ở mỗi vị trí sơ cấp. Mảng *counter* được sử dụng để truy cập các giá trị được lưu trữ và để biết khi nào mở rộng mảng.

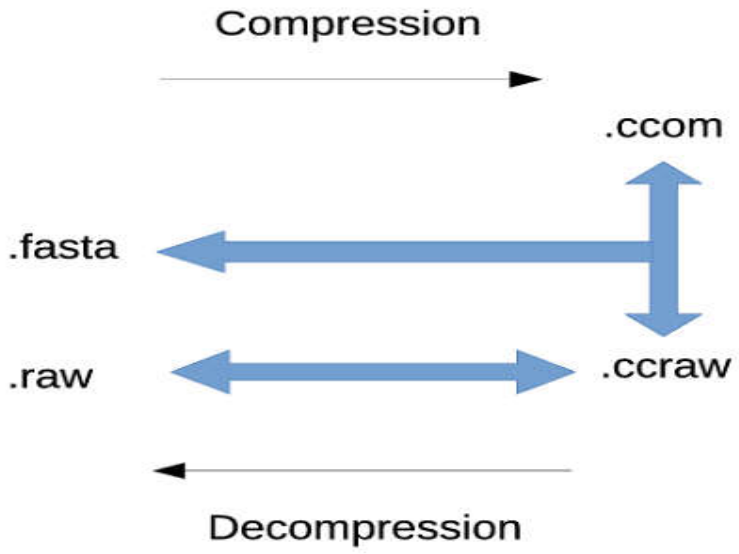
Việc tạo *MultiValueMap* này hiệu quả cho việc tạo đối tượng nhỏ nhất. Sự thực hiện này mang lại hai lợi ích. Một là giảm sử dụng bộ nhớ sử dụng một cấu trúc dữ liệu hiệu quả. Hai là khả năng tái sử dụng. Bảng K-mer có thể được tái sử dụng dễ dàng bằng việc đặt lại toàn bộ giá trị trong *counter* về 0.

Điều khiển *put* và *get* nhận một chỉ số *I* trên tham chiếu hoặc đầu vào để tạo ra một băm *H. H* đạt được bởi băm một số các cặp bazơ bằng K (K là kích thước K-mer) bắt đầu tại chỉ số đã cho. Sau đó *H* được sử dụng như một khóa để lưu *I* hoặc để nhận toàn bộ các giá trị biểu diễn cho khóa *H.* Mỗi khóa *H* lưu một tập các chỉ số với băm *H.* Tất cả băm là 32 bit.

2.1.4. Định dạng tệp

JDNA chấp nhận hai loại tệp cho đầu vào nén: RAW hoặc FASTA. Nếu tệp là RAW thì một tệp CRAW (cho RAW nén) được tạo ra; nếu tệp là FASTA

thì một CRAW và một CCOM được tạo ra. Tệp CCOM có chú thích trong tệp FASTA và số dòng của những chú thích này. Với giải nén, một tệp CRAW được gán như đầu vào. JDNA sẽ tìm kiếm một tệp CCOM cùng tên với tệp CRAW. Nếu tìm thấy CCOM thì FASTA được tạo ra, nếu không thì RAW được tạo. Thông tin này được tóm tắt trong hình 2.4.



Hình 2.4. Định dạng file đầu vào, đầu ra của JDNA

2.2. Đánh giá

Ở phần đánh giá này, người viết trình bày kết quả nén đạt được về tỉ lệ nén, thời gian nén và vùng nhớ để so sánh hiệu suất của thuật toán tham chiếu FRESCO với các thuật toán cùng loại khác như GDC và RLZ. Đồng thời so sánh hiệu suất của JDNA với Fresco để thấy được những cải tiến của JDNA đã thực sự mang lại hiệu quả về tỉ lệ nén và dung lượng lưu trữ. Do luận văn tập trung nghiên cứu chính là hiệu quả về tỉ lệ nén nên sau đây người viết sẽ tập trung mô tả cách thức và những cải thiện đạt được về tỉ lệ nén của các thuật toán. Hiệu quả về thời gian và dung lượng lưu trữ cũng được đưa ra như một kết quả của việc nghiên cứu. Mỗi kiểm tra được thực hiện 10 lần và kết quả thể hiện giá trị trung bình.

2.2.1. Cải thiện tỉ lệ nén

Chuỗi tham chiếu là nhân tố chính xác định tỉ lệ nén, cho một mã hóa đầu vào khớp tham chiếu cố định. Ví dụ, nếu một hệ gen người được nén tham chiếu dựa trên hệ gen chuột thì đầu ra được nén thực tế sẽ lớn hơn hệ gen đầu vào. Điều này xảy ra do có nhiều chuỗi đầu vào khớp tham chiếu rất ngắn (khoảng 12 bazơ); với mỗi đầu vào đều phải mã hóa vị trí, độ dài và kí tự không khớp.

Ngay cả khi trong cùng một loài thì chuỗi tham chiếu cũng có ảnh hưởng quan trọng tới tỉ lệ nén, ví dụ, nếu tham chiếu và đầu vào nén có quan hệ gần như mối quan hệ di truyền. Với việc làm tăng sự giống nhau giữa tham chiếu và chuỗi nén thì đầu vào khớp tham chiếu dài hơn có thể được tìm thấy và tỉ lệ nén sẽ được tăng lên.

Định nghĩa 1. Đặt *sersize(s,ref)* là kích thước chuỗi *comp(s,ref)*. Cho một tập $S=\{s_1,...,s_n\}$, đặt $sersize(S,ref) = \sum_{i<n} sersize(s_i,ref)$. Bài toán tìm tham chiếu tối ưu cho *S* được xác định như sau: Tìm một tham chiếu *ref1* để không tồn tại một tham chiếu *ref2* mà *sersize(S,ref2) < sersize(S,ref1)*.

Xác định kích thước chuỗi mở: có thể là đầu vào khớp tham chiếu hoặc số byte cần cho việc lưu trữ. Tìm một tham chiếu tối ưu cho một tập các chuỗi là một bài toán khó: Có 4^n(5^n, gồm *N*) khả năng tham chiếu độ dài *n*. Do độ dài của một nhiễm sắc thể lên tới vài trăm megabytes, việc liệt kê hết được toàn bộ các chuỗi tham chiếu là điều không thể. Sau đây người viết xin trình bày hai phương pháp cho bài toán tìm tham chiếu tối ưu. Phương pháp thứ nhất, lựa chọn tham chiếu, hạn chế tập tham chiếu ứng cử. Phương pháp thứ 2, viết lại tham chiếu, cải thiện một tham chiếu đã tồn tại bằng cách viết lại nó dựa trên các chuỗi nén.

- **<u>Lựa chọn một tham chiếu tốt</u>**

Đầu tiên, nói về sự lựa chọn một chuỗi tham chiếu tốt nhất cho một chuỗi nén đơn.

Định nghĩa 2. Cho một chuỗi *s* và một tập các tham chiếu ứng cử $\{ref_1,...,ref_m\}$, ref_i được gọi là một tham chiếu tốt nhất nếu không tồn tại $j \neq i$ mà $|comp(s,ref_j)| < |comp(s,ref_i)|$, trong đó $|X|$ là biểu diễn kích thước của chuỗi được nén tham chiếu *X*.

Có thể tồn tại hơn một tham chiếu tốt nhất, trong trường hợp đó một tham chiếu sẽ được chọn ngẫu nhiên. Theo thực nghiệm thì trường hợp này không bao giờ xảy ra.

Một phương thức đơn giản để tìm chuỗi tham chiếu tốt nhất là nén tất cả các chuỗi dựa trên tất cả chuỗi tham chiếu có thể và chọn ra tham chiếu mà cho ra số đầu vào khớp tham chiếu ít nhất, gọi là Rsbest. Nếu các chuỗi dài như trong trường hợp đưa ra thì sẽ tốn khá nhiều thời gian để tính toán nén tham chiếu *n * m,* trong đó *m* là số chuỗi tham chiếu ứng cử và *n* là số chuỗi nén. Nếu muốn nén 1000 chuỗi mà chọn chuỗi tham chiếu tốt nhất theo phương thức này thì sẽ tốn vài tuần; tuy nhiên, ta sẽ sử dụng phương thức này trên một mẫu để đánh giá các phương pháp được mô tả tiếp theo.

Cách thức giải quyết bài toán như sau: Thay vì nén một chuỗi dựa trên các tham chiếu ứng cử thì nén tham chiếu của chuỗi được so sánh với nén tham

chiếu của ứng cử tham chiếu dựa trên một tham chiếu đầu tiên được lựa chọn ngẫu nhiên. Cách thức này chỉ cần nén mỗi chuỗi một lần dựa trên tham chiếu đầu tiên, độc lập với số các tham chiếu ứng cử. Các tham chiếu ứng cử được lựa chọn ngẫu nhiên. Trước khi giới thiệu chi tiết về các phương thức lựa chọn, xác định sự giống nhau giữa hai phương pháp nén tham chiếu. Ý tưởng là hai nén tham chiếu được xác định là giống nhau hơn nếu chúng chia sẻ nhiều đầu vào khớp tham chiếu hơn.

Định nghĩa 3. Sự giống nhau giữa hai nén tham chiếu rc_1 và rc_2, kí hiệu là $rsim(rc_1,rc_2)$, được xác định như sau $rsim(rc_1,rc_2) = |rc_1 \cup rc_2| - |rc_1 \cap rc_2|$

Giá trị *rsim* càng nhỏ thì sự giống nhau càng lớn. Hai nén tham chiếu đồng nhất sẽ có giá trị *rsim* là 0. Một phương pháp lựa chọn tham chiếu được đề xuất là RsbitX như ở thuật toán 3 (xem hình 2.5 [21]).

Algorithm 3 Reference Selection RSbitX

Input: set of to-be-compressed sequences $s_1, ..., s_n$, set of candidate reference sequences $ref_1, ..., ref_m$, a base reference sequence ref_{base}, and a speedup value X
Output: index b for best reference

Compute $comp(ref_i, ref_{base})$ for all $1 \leq i \leq m$
for $1 \leq j \leq n$ **do**
    Split s_j into 1000 blocks $b_1, ..., b_{1000}$ of equal length
    Let sx_j be the concatenation of each X-th block of $b_1, ..., b_{1000}$[2]
end for
Compute $comp(sx_j, ref_{base})$ for all $1 \leq j \leq n$
for $1 \leq i \leq m$ **do**
    Let $val_i = 0$
    for $1 \leq j \leq n$ **do**
        $val_i = val_i + |rsim(comp(sx_j, ref_{base}), comp(ref_i, ref_{base}))|$
    end for
end for
Find the smallest val_{min} from $val_1, ..., val_m$ and let $b = min$

Hình 2.5. Lựa chọn tham chiếu RSbitX

Phương pháp theo cùng mẫu như RSbest, với hai khác biệt:

1. Nén các chuỗi đầu vào nén không chỉ dựa trên mỗi tham chiếu ứng cử mà còn dựa trên chuỗi tham chiếu cơ sở được chọn ref_{base}. Bởi vậy, nén tham chiếu được sử dụng ở trong vòng lặp cho tính toán *rsim*, nghĩa là $comp(s_j,ref_{base})$ và $comp(ref_i,ref_{base})$ không phải tính toán lại trên mỗi lần lặp.

2. Chỉ nén từng phần mỗi chuỗi, mong sự giống nhau của các chuỗi thành phần là biểu diễn cho các chuỗi hoàn thành. X xác định có bao nhiêu chuỗi được sử dụng cho nén từng phần. Mỗi chuỗi được chia thành 1000 khối có độ dài bằng nhau và sau đó $1/x$ khối được sử dụng cho nén từng phần (tất cả các khối được lấy trong trường hợp $X = 1$). Phân bố các khối cho nén từng phần bằng nhau trên toàn bộ chuỗi đầu vào.

Trong khi RSbest cần tính $m * n$ nén tham chiếu thì RSbitX chỉ cần tính $m + n$ nén tham chiếu, và nếu $X > 1$ thì (tính thô) chỉ cần tính $m + \frac{n}{X}$ nén tham

chiếu. Thời gian giảm được theo hệ số $\frac{m \quad n}{m+\frac{n}{X}}$ so với lựa chọn tham chiếu tốt nhất. Phương pháp này giả sử quá trình nén một chuỗi có độ phức tạp thời gian tuyến tính và có thể có lỗi hoặc bỏ sót khi cài đặt cấu trúc dữ liệu cho việc nén chuỗi.

Thực nghiệm trên số các khối khác nhau và đạt được các kết quả rất giống nhau. Nếu kích thước khối là nhỏ (nhỏ hơn 10000 byte) thì với hệ gen người, lựa chọn tham chiếu sẽ cho các kết quả giống nhau giống như phương thức lựa chọn ngẫu nhiên. Kết quả trên được cho là do indels (các phép chèn, xóa bazơ trong DNA) lớn hơn trong tập dữ liệu (những vùng giống nhau giữa hai chuỗi không kết thúc trong cùng một khối). Nếu số khối là nhỏ hơn 1000 thì tốc độ nén đạt được sẽ bị mất. Tập dữ liệu 1000 khối đã đạt được kết quả trung bình khá.

- **<u>Viết lại tham chiếu</u>**

Một phương thức khác là viết lại chuỗi tham chiếu theo cách mà nó biểu diễn một đường đi (chuỗi hành động) hầu như giống nhau qua tất cả các chuỗi trong tập hợp chuỗi nén. Trong phương thức này, số các chuỗi tham chiếu ứng cử được cố định là một. Viết lại chuỗi có một động lực sinh học: các SNP khác nhau thường xảy ra với tần suất khác nhau. Bằng việc viết lại tham chiếu để xác định và gắn các SNP thường xuyên nhất tới tham chiếu.

Ví dụ 1.

Nén tham chiếu các chuỗi

s_1 = *AAAACGGACAATCTGA*

s_2 = *AAAACGGACAATCTGT*

s_3 = *AAAACGACAATCTGT*

dựa trên tham chiếu AAAACGCACAATCTGC, ta có 3 nén tham chiếu sau:

rc_1 = *{(0,6,G), (7,8,A)}*

rc_2 = *{(0,6,G), (7,8,T)}*

rc_3 = *{(0,6,A), (8,7,T)}*

Nếu vị trí thứ 7 của chuỗi tham chiếu gồm một G thay cho một C thì có thể nén rc_1 *và* rc_2 *sử dụng chỉ một đầu vào mỗi:* rc_1^{new} = *{(0,15,A)},* rc_2^{new} = *{(0,15,T)}.*

Từ ví dụ 1 có thể thấy rằng việc viết lại chuỗi tham chiếu để giảm số đầu vào khớp tham chiếu và do vậy mà tăng được tỉ lệ nén. Các bước viết lại cần được xem xét cẩn thận. Với một tập các chuỗi lớn thì không chắc là tất cả các chuỗi sẽ nghiêng về các phép chèn/xóa/thay thế bazơ đặc thù dựa trên một tham chiếu. Tuy nhiên, ngay cả khi phần lớn các chuỗi chia sẻ cùng độ lệch cơ sở so với tham chiếu thì tỉ lệ nén vẫn có thể được tăng lên. Ví dụ 1 còn cho thấy là

không thể viết mù lại tham chiếu vì không phải tất cả chuỗi đều nằm trên vị trí thứ 7.

Sau đây là một phương pháp viết lại các chuỗi tham chiếu. Những đánh giá chỉ ra rằng việc viết lại này có thể thực sự tiết kiệm lên tới 20% không gian trên các chuỗi sống thực. Xác định một tập các ứng cử thay thế từ một (tập hợp) chuỗi nén cho trước. Trong phần còn lại của mục này, người viết sẽ tập trung vào việc viết lại bazơ mà hoặc là thay thế, chèn hoặc xóa bazơ; trong tương lai sẽ nghiên cứu những thay đổi xa hơn. Do đầu vào khớp tham chiếu có lưu cả phần không khớp dựa trên tham chiếu nên dễ dàng tìm được các ứng cử thay thế. Tiêu chí hình thức cho một ứng cử thay thế là tồn tại hai chuỗi đầu vào khớp tham chiếu liên tiếp, như *(0,6,C)* và (*7,8,A)* ở ví dụ 1, để một thay thế với kí tự không khớp trong tham chiếu sẽ đạt được một khoảng dài liên kết thay cho hai khoảng ngắn. Hình 2.6 mô tả thuật toán viết lại tham chiếu [21].

Algorithm 4 Reference Rewriting Algorithm

Input: set of referential compressions $S = \{rc_1, ..., rc_n\}$, a reference string ref, and a threshold t
Output: rewritten reference $result$

Let $result$ be an empty string
for $1 \leq p \leq |ref|$ **do**
  if there exists a most frequent rewrite candidate (X, p, c) for p in S, with $freq((X, p, c), S) \geq t$ **then**
    if X=REPL **then**
      Append c to $result$
    else if X=INS **then**
      Append c to $result$
      Append $ref(p)$ to $result$
    else if X=DEL **then**
      do nothing
    end if
  else
    Append $ref(p)$ to $result$
  end if
end for

Hình 2.6. Thuật toán viết lại tham chiếu

Định nghĩa 4. Một phép thay thế cho một nén tham chiếu *rc* được gọi là *(repl, p, c)*, nếu tồn tại hai RME liên tiếp $[(p_1, l_1, c), (p_2, l_2, c_2)] \in rc$ với $p_1 + l_1 + 1 = p_2$ và $p = p_1 + l_1$. Một phép chèn cho nén tham chiếu *rc* được gọi là *(ins, p, c)*, nếu tồn tại hai RME liên tiếp $[(p_1, l_1, c), (p_2, l_2, c_2)] \in rc$ với $p_1 + l_1 = p_2$ và $p = p_1 + l_1$. Một phép xóa cho nén tham chiếu *rc* được gọi là *(del, p, _)*, nếu tồn tại hai RME liên tiếp $[(p_1, l_1, c), (p_2, l_2, c_2)] \in rc$ với $p_1 + l_1 + 2 = p_2$ và $p = p_1 + l_1$. Các phép viết lại của nén tham chiếu dựa trên tham chiếu *ref*, kí hiệu *rewr(rc)* là tập hợp tất cả các phép thay thế, chèn, xóa của *rc.*

Định nghĩa 5. Cho một tập nén tham chiếu $S = \{rc_1, ..., rc_n\}$ dựa trên tham chiếu *ref,* tần suất tương đối của một phép viết lại được xác định như sau:

$$freg\big((X, p, c), S\big) = \frac{|\{rc_i | rc_i \in S \quad (X, p, c) \in rewr(rc_i)\}|}{|S|}$$

Cho vị trí *p,* phép viết lại xảy ra nhiều nhất cho *p* trong *S* là *(X,p,c)*, nếu không tồn tại một *X* ∈ {repl, ins, del} và c* với freq ((X*,p,c*), S) > freq((X,p,c), S).* Trong trường hợp hai phép viết lại có tần suất bằng nhau thì chọn một phép ngẫu nhiên.

Ví dụ 2. Trong ví dụ 1, ta có $rewr(rc_1) = \{(repl, 6, G)\}$, $rewr(rc_2) = \{(repl, 6, G)\}$ và $rewr(rc_3) = \{(del, 6, \_)$

Tần suất của *(repl, 6, G)* là *2/3*, nghĩa là sự thay thế xảy ra trong 2 trên 3 chuỗi nén. Tần suất của *(del, 6, _) là 1/3*. Do đó, phép viết lại cho vị trí 6 có tần suất cao nhất là *(repl, 6, G).*

Phép viết lại cho mỗi vị trí có tần suất cao nhất trong tham chiếu được sử dụng để viết lại chuỗi tham chiếu. Thuật toán viết lại tham chiếu được chỉ ra trong thuật toán 4. Đầu vào của thuật toán là một tập nén tham chiếu *S*, một chuỗi tham chiếu sẽ-được-viết-lại *ref* và một giới hạn *t.* Giới hạn được sử dụng chỉ để chọn ra phép viết lại mà có ít nhất một tần suất tương đối trong *S.* Thuật toán lặp lại trên chuỗi tham chiếu và kiểm tra mỗi vị trí trong tham chiếu, để xác định nếu một phép viết lại có tần suất cao nhất tồn tại thì tần suất đó phải cao hơn giới hạn *t.* Nếu tồn tại một phép viết lại như vậy thì các kí tự được thêm vào đầu ra *result* của thuật toán tùy thuộc vào loại viết lại (thay thế, chèn, xóa). Nếu không tồn tại phép viết lại đó cho vị trí *p* thì thuật toán chỉ gắn bazơ gốc từ vị trí *p* của tham chiếu tới *result.* Sau khi thực hiện thuật toán, *result* sẽ bao gồm chuỗi tham chiếu được viết lại. Theo thực nghiệm thì việc lựa chọn chuỗi tham chiếu ban đầu chỉ có một tác động nhỏ tới tỉ lệ nén. Hơn nữa, tính toán lại mỗi nén tham chiếu đối với chuỗi tham chiếu được viết lại đã được thực nghiệm. Việc cập nhật nén tham chiếu để phản ánh những thay đổi trong tham chiếu viết lại mà không cần phải nén lại sẽ là một hướng đáng quan tâm trong tương lai.

Ví dụ 3. Nếu áp dụng thuật toán 4 (hình 2.6) vào ví dụ 2 với giới hạn *t=0.6,* ta đạt được tần suất tham chiếu viết lại AAAACGCACAATCTGC, do chỉ tồn tại một phép viết lại với tần suất tương đối lớn hơn 0.6 nên ta có phép viết lại *(repl,6,C).* Nếu đặt *t=0.8* thì thuật toán sẽ không thay đổi chuỗi tham chiếu. Một chú ý là phép viết lại *(del,6,_)* sẽ không bao giờ được sử dụng trong quá trình thực hiện thuật toán, độc lập với giới hạn, do *(del,6,_)* chịu ảnh hưởng bởi *(repl,6,C)* cho vị trí 6.

Có thể thấy từ ví dụ 3 là việc lựa chọn giới hạn *t* có ảnh hưởng to lớn đến đầu ra của thuật toán viết lại: giới hạn quá lớn sẽ bỏ qua các phép viết lại có tần suất tương đối ngang bằng mà được chia sẻ bởi nhiều nén tham chiếu.

Độ phức tạp cho tính toán viết lại là tuyến tính theo số chuỗi và độ dài của chuỗi. Thuật toán phải tìm kiếm mỗi cặp RMEs liên tiếp và kiểm tra, kể cả nó có

là một phép viết lại cho vị trí *p* hay không. Nếu có, thì thêm một chú thích đầu vào vị trí *p* trong chuỗi tham chiếu. Sau cùng, tìm mỗi vị trí của tham chiếu trong trường hợp tần số phép viết lại là trên giới hạn *t*. Do vậy, việc phân tích tất cả các chuỗi sẽ mất khoảng thời gian tuyến tính và viết lại thực cũng có thể được thực hiện trong thời gian tuyến tính. Hướng quan tâm cho việc nghiên cứu trong tương lai đó là viết lại các chuỗi dài hơn, nghĩa là xác định các phép sửa (indels) tần suất dựa trên tham chiếu.

Để tính toán nén tham chiếu dựa trên tham chiếu viết lại thì phải nén lại tất cả các chuỗi từ phần thô hỗn hợp. Với thời gian nén nhanh như FRESCO, trong hầu hết trường hợp thì nén lại là có thể chịu được. Tuy nhiên, đối với các tập chuỗi thay đổi thường xuyên thì nên tránh việc nén lại.

(1) <u>So sánh FRESCO với hai thuật toán cùng loại GDC và RLZ</u>

So sánh sự thực hiện của các thuật toán nén tham chiếu với FRESCO. Hai đối thủ của FRESCO là GDC [19] và RLZ [24]. Có thể thấy RLZ là ngang hàng trong nén tham chiếu, trong khi GDC là chương trình tốt nhất khi so về tốc độ nén và tỉ lệ nén.

So sánh đầu tiên như sau: với mỗi loài và mỗi nhiễm sắc thể, lựa chọn ngẫu nhiên 10 chuỗi và áp dụng vào mỗi thuật toán nén tham chiếu. GDC áp dụng một loại lựa chọn trước tham chiếu cho một tập các chuỗi đầu vào. Thời gian lựa chọn tham chiếu không bao gồm trong phép đo: chỉ tính toán thời gian nén. RLZ sử dụng các mảng hậu tố cho chuỗi tham chiếu. Thời gian xây dựng mảng hậu tố không bao gồm trong phép đo (xây dựng mảng hậu tố cho tham chiếu của HG-1 mất khoảng 2 phút). FRESO sử dụng một chỉ số k-mer (với k=34) cho chuỗi tham chiếu và lựa chọn LO_MD và COMPACT. Lựa chọn *k* có một ảnh hưởng lớn lên tốc độ nén, nhưng hầu như không ảnh hưởng tới tỉ lệ nén. Với một giá trị *k* nhỏ hơn 14, nén được xác nhận là chậm hơn, do FRESCO phải kiểm tra nhiều chuỗi khớp giả mà không liên quan tới nén tham chiếu bởi vì chúng không đạt được chuỗi khớp dài. Với giá trị *k* trong khoảng giữa 14 và 34, tốc độ nén tăng đáng kể (theo hệ số 2-3), trong khi tỉ lệ nén được xác nhận là không thay đổi. Tăng giá trị *k* lớn hơn 34 không làm thay đổi tốc độ nén. Thời gian tạo chỉ số *k*-mer cho mỗi chuỗi tham chiếu là khoảng 1 phút đối với chuỗi lớn nhất và không bao gồm trong các phép tính toán. Kết quả nén 10 chuỗi được chỉ ra ở Hình 2.7.

	Compressed size (in MB)			Runtime (in s)			Compression factor			Compression speed (MB/s)		
Dataset	GDC	RLZ	FRESCO	GDC	RLZ	FRESCO	GDC	RLZ	FRESCO	GDC	RLZ	FRESCO
H-1	**3.7**	15.5	4.2	495.2	224.0	**20.0**	**680.0**	160.8	590.6	5.0	11.1	**124.3**
H-2	**3.9**	15.9	4.5	454.9	199.4	**19.4**	**625.5**	152.9	542.8	5.3	12.2	**125.5**
H-3	**3.3**	13.4	3.8	314.6	165.5	**14.9**	**593.6**	147.5	513.9	6.3	11.9	**132.4**
H-4	**3.5**	13.8	4.1	247.0	159.4	**15.0**	**543.8**	138.4	466.1	7.7	12.0	**127.1**
H-5	**3.0**	12.0	3.4	243.4	144.0	**13.9**	**608.2**	150.6	526.3	7.4	12.6	**130.2**
H-6	**3.0**	11.9	3.6	248.0	143.8	**15.3**	**566.1**	143.7	475.1	6.9	11.9	**112.0**
H-7	**2.7**	10.7	3.1	403.1	121.1	**12.8**	**591.2**	148.7	508.8	3.9	13.1	**124.7**
H-8	**2.5**	10.1	2.9	171.8	122.9	**11.6**	**577.5**	144.8	500.5	8.5	11.9	**126.3**
H-9	**2.0**	8.4	2.3	130.0	102.2	**11.0**	**714.3**	168.0	618.2	10.9	13.8	**128.8**
H-10	**2.4**	9.4	2.7	183.6	109.8	**10.9**	**572.2**	144.1	493.4	7.4	12.3	**124.7**
H-11	**2.5**	9.6	2.8	153.6	118.3	**11.0**	**548.3**	140.5	474.3	8.8	11.4	**122.2**
H-12	**2.3**	8.9	2.6	199.2	113.5	**10.0**	**593.0**	150.4	514.1	6.7	11.8	**133.5**
H-13	**1.9**	7.5	2.2	65.5	90.9	**9.2**	**602.5**	153.4	532.2	17.6	12.7	**124.5**
H-14	**1.6**	6.4	1.8	68.5	77.0	**8.6**	**664.7**	167.6	591.1	15.7	13.9	**124.2**
H-15	**1.4**	5.9	1.6	72.2	70.7	**8.1**	**710.1**	173.7	636.9	14.2	14.5	**126.9**
H-16	**1.4**	5.4	1.6	103.1	68.9	**6.9**	**638.5**	167.1	552.5	8.8	13.1	**131.4**
H-17	**1.3**	5.1	1.5	140.3	68.9	**6.5**	**635.3**	159.1	552.8	5.8	11.8	**125.4**
H-18	**1.4**	4.8	1.6	44.6	66.7	**6.6**	**565.2**	162.5	487.0	17.5	11.7	**118.3**
H-19	**1.1**	4.0	1.3	116.8	50.8	**5.3**	**546.7**	147.8	468.0	5.1	11.6	**111.1**
H-20	**1.0**	4.0	1.2	43.8	49.5	**4.5**	**623.7**	157.4	542.5	14.4	12.7	**139.3**
H-21	**0.7**	2.8	0.9	12.3	33.3	**3.5**	**684.3**	171.8	553.0	39.1	14.5	**138.2**
H-22	**0.6**	2.7	0.7	19.3	32.0	**3.7**	**816.9**	189.7	735.9	26.5	16.0	**137.1**
H-X	**1.7**	7.7	2.0	168.2	96.3	**12.1**	**903.6**	201.6	789.0	9.2	16.1	**128.0**
AT-1	**2.0**	6.5	2.3	8.3	41.3	**2.5**	**154.2**	105.3	133.2	36.7	7.4	**123.1**
AT-2	**1.4**	4.5	1.7	4.2	25.4	**1.4**	**145.0**	98.5	119.0	46.9	7.8	**136.8**
AT-3	**1.7**	5.5	2.0	5.5	32.1	**1.6**	**139.8**	96.0	117.2	42.7	7.3	**145.1**
AT-4	**1.3**	4.3	1.6	3.7	24.4	**1.5**	**139.5**	97.2	116.7	50.2	7.6	**126.5**
AT-5	**1.9**	6.1	2.2	6.3	37.5	**1.9**	**144.6**	99.5	121.3	42.8	7.2	**141.2**
Y-WG	**1.0**	86.8	1.4	2.8	47.6	**1.0**	**127.3**	1.4	89.0	44.5	2.6	**124.7**
AVG	**2.0**	10.7	2.3	142.4	90.9	**8.6**	**532.9**	142.8	460.7	18.0	11.5	**128.0**

Hình 2.7. Thống kê nén 10 chuỗi ngẫu nhiên dựa trên một tham chiếu cố định (kết quả tốt nhất được bôi đậm)

GDC đạt được kết quả nén tốt nhất cho mỗi tập dữ liệu dùng để đánh giá (trung bình 2.0MB cho 10 chuỗi). Điều này được dự đoán là phụ thuộc kỹ thuật mã hóa đối với định dạng chuỗi và cơ chế lựa chọn tham chiếu. GDC cũng cố gắng tìm và mã hóa chuỗi khớp xấp xỉ trong tham chiếu. Ý tưởng này dường như hoạt động tốt đối với các loài khác nhau cao. FRESCO đạt được hiệu quả nén tốt thứ hai (trung bình 2.3MB cho 10 chuỗi), trong khi RLZ cần hầu hết không gian cho mỗi tập dữ liệu (hơn 5 lần so với GDC). RLZ đạt hệ số nén thấp đối với Y-WG dường như là do kỹ thuật tối ưu hạn chế trong nó (đặc biệt là đối với chuỗi khớp ngắn). Hệ số nén trung bình đối với H-* là: GDC = 635, RLZ = 158 và FRESCO = 551. Hệ số nén cho AT-* và Y-WG được xem là thấp hơn do đặc điểm giống nhau giảm giữa các chuỗi trong các tập hợp.

FRESCO có thời gian nén ngắn nhất (trung bình 8.6 giây cho 10 chuỗi), trong khi RLZ chậm hơn khoảng 10 lần và GDC chậm hơn khoảng 16 lần. Tốc độ nén cho H-* như sau: GDC=11.2 MB/s, RLZ=12.8 MB/s, FRESCO=126.8 MB/s. Tốc độ nén trung bình của GDC cho tất cả các loài là 18.0 MB/s. Dường như GDC là tối ưu cao cho nén các chuỗi ngắn (hay cụ thể là các loài khuẩn men): Tốc độ nén của GDC cho AT-* và Y-WG hầu hết là cao hơn 5 lần so với cho H-*. FRESCO được cho là nhanh hơn GDC vì ba lý do. Đầu tiên, GDC cố gắng mở rộng chuỗi tham chiếu với các phần tham chiếu nhỏ bổ sung trong suốt

quá trình nén, trong khi FRESCO sử dụng một tham chiếu cố định cho nén đầu tiên. Lưu giữ cấu trúc các chỉ mục bổ sung (hoặc cập nhật chúng thường xuyên) tiêu tốn khá nhiều chi phí. Thứ hai, GDC đã mã hóa chuỗi khớp xấp xỉ. Trong khi việc này cho ra tỉ lệ nén cao hơn FRESCO cơ sở, thì nó dường như đắt về mặt tính toán để xác định các chuỗi khớp này với các lỗi nhỏ. Thứ 3, sử dụng một chỉ mục k-mer nhanh mà sử dụng nhiều bộ nhớ hơn GDC, nhưng cho phép tìm kiếm nhanh hơn.

Tốc độ nén trung bình của RLZ là 11.5 MB/s, FRESCO là một hằng số ước chừng giữa các loài như nhau: 128.0 MB/s. Cả RLZ và FRESCO đều chậm hơn một chút cho Y-WG hơn cho các loại khác. Có thể thấy là cả 3 chương trình đều có một tốc độ nén ổn định (ngoại trừ GDC thì còn có thể liên quan tới loại chứ không phải tới độ dài của chuỗi).

Chạy thực nghiệm với GReEn [31] và một mẫu 10 chuỗi của H-1. GReEn cần 183 giây thời gian nén chỉ cho cả 10 chuỗi (mà không tạo cấu trúc chỉ số cho tham chiếu). Việc này chậm hơn gần 10 lần so với FRESCO. Tỉ lệ nén vào khoảng 250:1. FRESCO- cơ sở (590:1) và GDC (680:1) đạt được tỉ lệ nén ít nhất gấp đôi. Sau cùng, kết quả nén của GReEn rất giống với kết quả đạt được bởi RLZ.

Lưu ý là tốc độ đọc cao nhất của đĩa cứng ở thực nghiệm là khoảng 145 MB/s. Nén với FRESCO dường như có giới hạn vào/ra: thực hiện các thực nghiệm bổ sung với các chuỗi trong bộ nhớ chính. Đối với H-*, đạt được tốc độ nén trung bình là 729 MB/s và một tốc độ nén lớn nhất là 1 GB/s với FRESCO. Tốc độ này lớn hơn hai bậc so với các phương thức nén đang tồn tại. Với hai loài khác, tốc độ nén bộ nhớ chính không được ghi nhận là cao hơn so với ổ cứng ngoài. Trong các kiểm tra, các tệp nén tham chiếu có thể được giải nén với tốc độ khoảng 500 MB/s với bộ nhớ chính.

Bộ nhớ chính sử dụng cho FRESCO là khoảng 8 – 10 lần kích thước chuỗi tham chiếu, dành cho việc biểu diễn chỉ số k-mer trong bộ nhớ chính. Trong thực nghiệm, với cây hậu tố được nén, việc tiêu tốn bộ nhớ chính có thể giảm được tới 2 lần kích thước tham chiếu cộng với kích thước của chuỗi nén, trong khi thời gian nén bị tăng một chút (thêm 30% cho H-*).

Có một điều thú vị là xếp hạng của 3 chương trình thực sự là nhất quán, không chỉ với các nhiễm sắc thể khác nhau mà còn với cả các loài khác nhau dựa trên hai tiêu chí đánh giá. Tóm lại, GDC luôn đạt được kết quả nén tốt nhất, trong khi FRESCO thì đạt tốc độ nén nhanh hơn RLZ và GDC. Hình 2.8 tóm tắt kết quả của nén tham chiếu FRESCO so với GDC và RLZ.

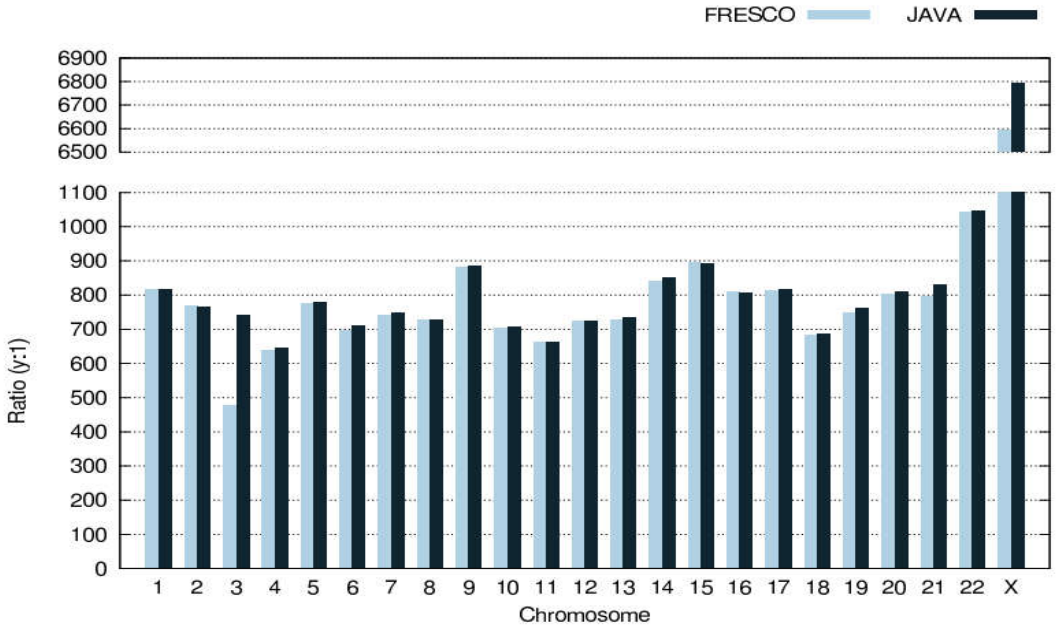
	GDC		RLZ		FRESCO		FRESCO (reference selection)		FRESCO (reference rewriting)		FRESCO (second-order compression)	
	CF	C.Speed	CF	C.Speed	CF	C.Speed	CF	C.Speed	CF	C.Speed	CF	C.Speed
H-*	635.0	11.2	158.4	12.8	550.7	126.8	594.7	81.2	742.4	90.6	3,057.2	58.4
AT-*	144.6	43.9	99.3	7.5	121.5	134.5	126.9	56.3	123.6	60.6	407.7	53.7
Y-WG	127.3	44.5	1.4	2.6	89.0	124.7	89.0	21.1	91.9	21.5	712.8	41.4
AVERAGE	302.3	33.2	86.4	7.6	253.7	128.7	270.2	52.8	319.3	57.5	1,392.6	51.1

Hình 2.8. Tóm tắt kết quả nén đạt được từ FRESCO, GDC và RLZ (CF: hệ số nén, C.speed: tốc độ nén MB/s)

Từ kết quả thực nghiệm trên cho thấy FRESCO đạt hiệu quả tốt hơn hẳn so với các thuật toán nén GDC và RLZ.

(2) <u>So sánh hiệu quả của JDNA và Fresco</u>

Tỉ lệ nén *(y:1)* nghĩa là kích thước của một tệp nén là *y*x nhỏ hơn kích thước ban đầu. Bảng 2.1 thể hiện so sánh tỉ lệ nén giữa hai công cụ và hình 2.9 biểu diễn tỉ lệ nén đạt được [21]. Tỉ lệ nén hầu hết là xác định giữa JDNA và FRESCO. Có hai lý do cho những khác nhau nhỏ thấy được với các nhiễm sắc thể. Thứ nhất là thuật toán mã hóa sử dụng ở mỗi giải pháp. Một bit khác nhau trong pha mã hóa tạo nên sự thay đổi đáng kể về tỉ lệ nén do nó bị khuếch đại bởi hàng ngàn chuỗi khớp. JDNA sử dụng một phiên bản mã hóa Huffman chỉnh sửa và Gzip, trong khi FRESCO chỉ sử dụng Gzip để nén mỗi chuỗi khớp. Lý do thứ hai là chỉ số hoàn toàn được thực hiện bởi FRESCO. Việc tìm kiếm xác định đảm bảo một chuỗi khớp có tồn tại hay không, không như JDNA, một chuỗi khớp được tìm thấy chỉ bởi những tìm kiếm xác định và thao tác chỉ số nhỏ.



Hình 2.9. So sánh tỉ lệ nén

Bảng 2.1. Bảng so sánh JDNA/FRESCO

Chr	Original Size (MB)	Compression JDNA (KB)	Compression FRESCO (KB)	Index Time JDNA (s)	Index Time FRESCO (s)	Indexes K	%
1	238	306	306	0.02	62.56	40.48	1.6
2	232	319	318	0.01	57.6	33.55	1.4
3	189	269	406	0.01	47.02	34.99	1.8
4	182	296	299	0.03	45.5	63.79	3.3
5	173	234	235	0.02	42.58	38	2.1
6	163	243	249	0.05	40.03	150.23	8.8
7	152	215	217	0.02	37.2	41.96	2.6
8	140	202	202	0.01	33.89	26.87	1.8
9	135	161	162	0.03	28.97	94.12	6.7
10	129	192	194	0.01	30.94	15.5	1.1
11	129	204	204	0.02	30.92	30.89	2.3
12	128	186	186	0.02	30.65	34.81	2.6
13	110	157	159	0.01	23.3	15.55	1.4
14	102	126	128	0.01	21.44	19.96	1.9
15	98	115	115	0.01	19.91	10.13	1
16	86	113	112	0.01	18.63	15.91	1.8
17	77	100	100	0.01	18.01	16.13	2
18	74	114	115	0.01	17.56	7.74	1
19	56	79	80	0.01	12.81	26.87	4.5
20	60	79	79	0.01	14.06	22.16	3.5
21	46	58	61	0.01	8.93	25.11	5.2
22	49	49	50	0.01	8.89	24.28	4.7
X	148	109	109	0.01	35.93	18.23	1.2

2.2.2. Cải thiện thời gian

Thực nghiệm ở trên đã chứng minh Fresco hiệu quả hơn các thuật toán cùng loại và cũng đã cho thấy sự vượt trội của JDNA so với Fresco. Ở phần tiếp theo, người viết sẽ chỉ trình bày so sánh hiệu quả về thời gian và vùng nhớ của JDNA so với Fresco.

Thực nghiệm này so sánh hiệu quả về thời gian của cả hai công cụ cho việc nén và giải nén hệ gen người. Việc đánh giá thời gian được chia thành 4 phần:

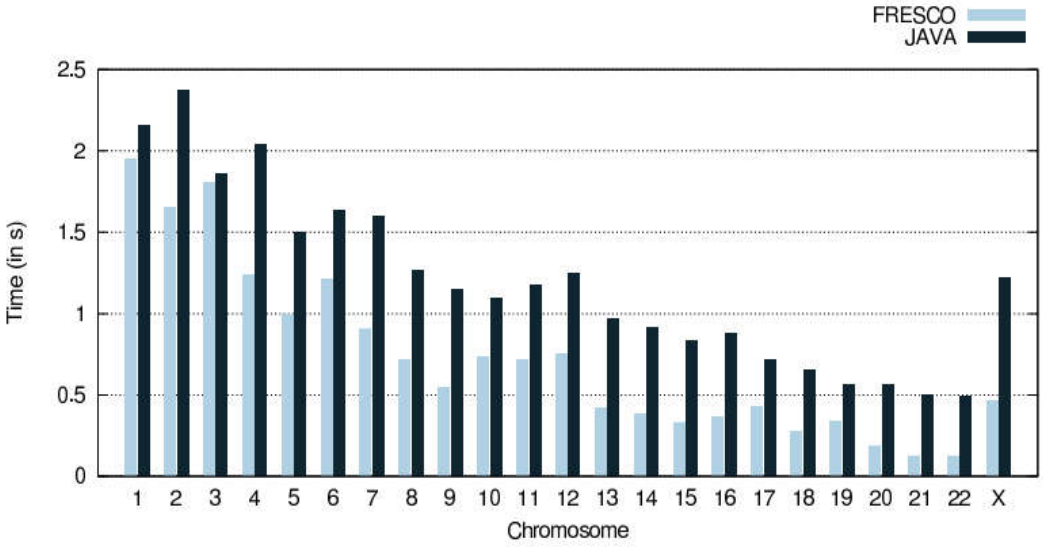
Nén đầy đủ: Đây là sự thực hiện đầy đủ của thư viện; gồm thời gian bắt đầu, đọc tệp, sắp xếp bộ nhớ, đánh chỉ số tham chiếu, nén và ghi tệp.

Đánh chỉ số thời gian: Vì JDNA đưa ra đánh chỉ số theo yêu cầu nên ta chỉ so sánh thời gian đánh chỉ số.

Thời gian nén: Trong phạm vi luận văn, người viết đánh giá hiệu suất của hai phương pháp chỉ trên việc nén, bằng cách đo thời gian cho việc nén thực sự.

Thời gian giải nén: Ở đây đánh giá hiệu suất giải nén cả hai thư viện, đo thời gian thực hiện toàn bộ.

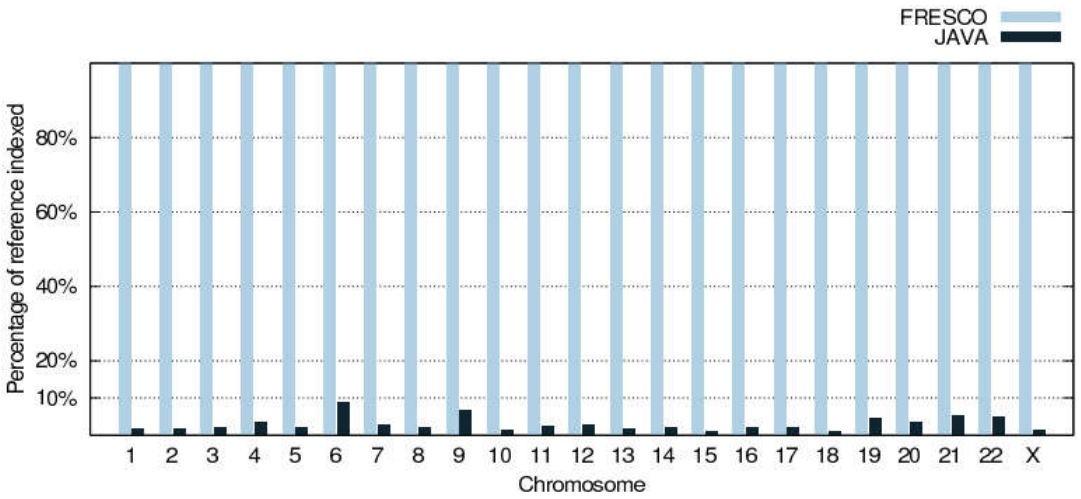
Thực nghiệm đo cả hai thời gian bắt đầu, JVM với cấu hình cho JDNA trung bình mất 0.1 giây để bắt đầu, FRESCO mất 0.04 giây. Thời gian nén đầy đủ đo được sử dụng dòng lệnh *time*, kết quả có thể thấy ở hình 2.10.



Hình 2.10. Thời gian nén

Như đã mô tả từ trước, JDNA tránh đánh chỉ số, điều này tạo nên sự khác biệt lớn về thời gian nén. Những giá trị này có thể thấy ở bảng 2.1. Những tệp lớn (ví dụ nhiễm sắc thể 1) mất khoảng 5 giây để nén với cấu trúc thuật toán này, đây là một khác biệt lớn so với FRESCO mất gần cả phút để nén cùng số tệp. Thời gian nén của JDNA gần như là cố định (khoảng 3 giây), còn khoảng 2 giây với những nhiễm sắc thể nhỏ. Kết quả này nhanh hơn khoảng 5 đến 12 lần so với những gì ta thấy ở FRESCO. Sự khác biệt này là do đánh chỉ số theo yêu cầu. Vì không đánh chỉ số toàn bộ tham chiếu, JDNA không mất thời gian đánh chỉ số khi bắt đầu thực hiện.

Thời gian mỗi thư viện dùng để đánh chỉ số gen tham chiếu được đo trong quá trình thực hiện chương trình, kết quả có thể thấy ở bảng 2.1. JDNA hầu như không tốn thời gian đánh chỉ số, đặc biệt là so với thời gian đánh chỉ số luôn lớn hơn ở FRESCO. JDNA dùng thời gian cho nén đầy đủ với bất kỳ nhiễm sắc thể nào. Một phần trăm nhỏ các cặp bazơ được đánh chỉ số, có thể thấy ở bảng 2.1 và hình 2.11. FRESCO luôn đánh chỉ số 100% các tham chiếu gen.

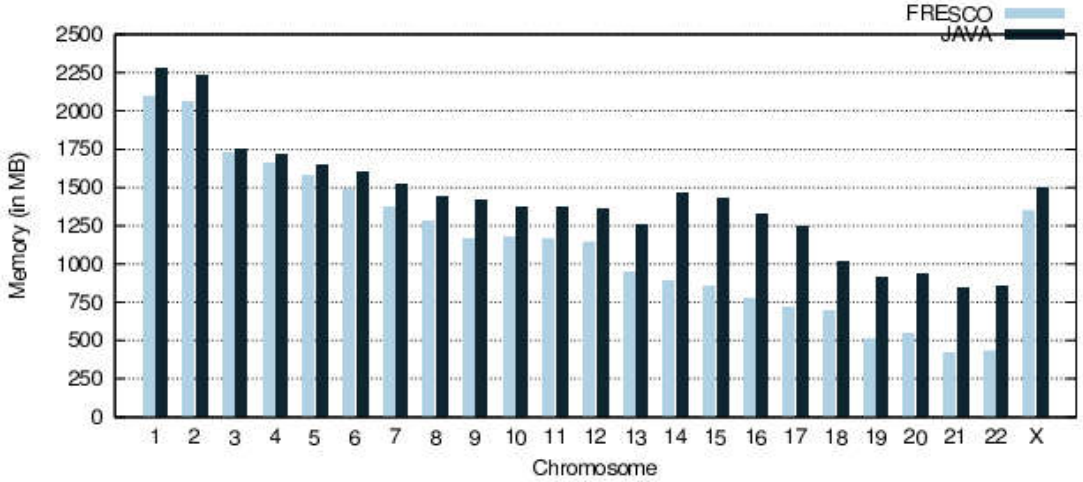


Hình 2.11. Phần trăm đánh chỉ số tham chiếu ở mỗi công cụ

Kết quả đo thời gian nén của chương trình được thể hiện ở hình 2.10. JDNA tốn thời gian gần như FRESCO cho bước nén, hai phương thức này khác nhau về thời gian thực hiện chủ yếu là ở bước đánh chỉ số.

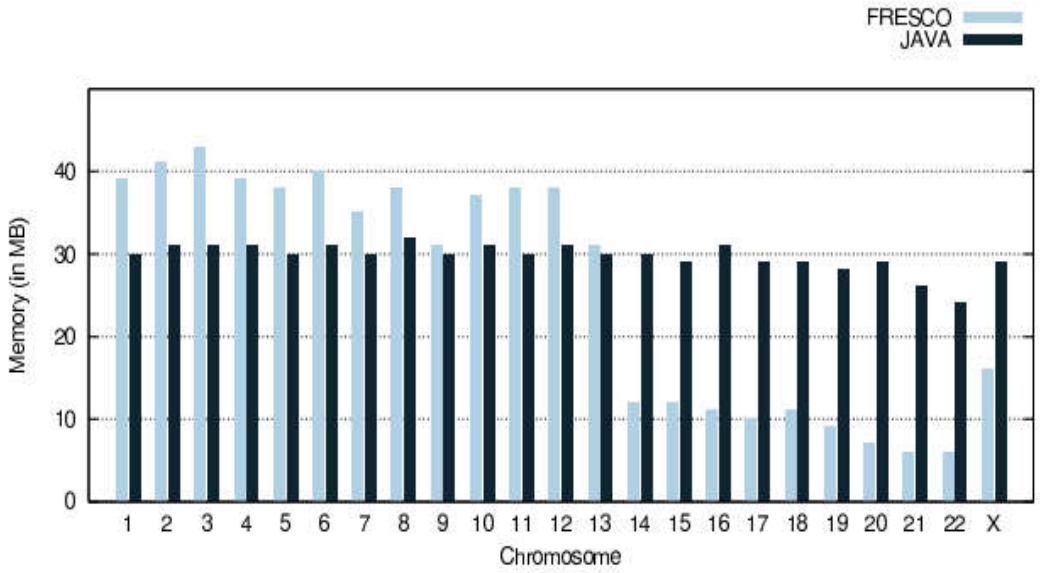
2.2.3. Cải thiện vùng nhớ

Vùng nhớ nén. Một công cụ ngoài được sử dụng để đo việc sử dụng bộ nhớ lớn nhất của hai phương pháp. Hình 2.17 cho thấy việc sử dụng bộ nhớ của JDNA và FRESCO. JDNA thực hiện cơ chế tái sử dụng đối tượng và giảm việc tạo ra đối tượng. Tuy nhiên, JDNA và FRESCO sử dụng vùng nhớ tương tự nhau, ngay cả sau khi đã nỗ lực giảm sử dụng vùng nhớ đáng kể. Việc sử dụng bộ nhớ trong JDNA phụ thuộc bảng K-mer. Mặc dù JDNA đã giảm đánh chỉ số và bảng K-mer chỉ là một ma trận số nguyên, do mỗi dòng ma trận là một đối tượng mới nên bộ nhớ sử dụng vẫn lớn so với FRESCO, phương thức mà đánh chỉ số toàn bộ tham chiếu.



Hình 2.12. So sánh vùng nhớ nén.

Vùng nhớ giải nén. Giải nén sử dụng một lượng vùng nhớ cố định cho tham chiếu, kết quả trong một hằng số sử dụng vùng nhớ (xem hình 2.13).



Hình 2.13. So sánh vùng nhớ giải nén.

Điểm tương đồng giữa gen tham chiếu và gen đầu vào sẽ quyết định kết quả của FRESCO và JDNA, trong đó sự tương đồng càng lớn thì tỉ lệ nén càng cao. Các kết quả được trình bày là những giá trị trung bình. Nén toàn bộ một hệ gen người cho kết quả trong một tệp kích thước từ 4 tới 10MB.

Kết quả chỉ ra ở phần đánh giá chứng minh rằng thuật toán đánh chỉ số theo yêu cầu có thể được sử dụng để xây dựng một công cụ có thể so sánh với các công cụ khác mà đánh chỉ số tham chiếu hoàn toàn. Các kết quả có tính cạnh tranh cho những thuộc tính được kiểm thử và cho thấy sự cải thiện về tổng thời gian thực hiện và tỉ lệ nén. JDNA đã kế thừa và những cải tiến cho thấy thuật toán đã đạt được hiệu quả khả quan trong việc nén chuỗi gen và cả hệ gen.

Thuật toán nén tham chiếu dù chỉ mới phát triển gần đây và được biết đến như một loại thuật toán thứ tư cho nén chuỗi đa lượng nhưng đã cho thấy hiệu quả vượt trội hơn hẳn so với ba loại thuật toán nén được biết đến trước đó là (1) thuật toán nén mã hóa bit, (2) thuật toán nén dựa trên bộ từ điển và (3) thuật toán nén xác suất thống kê. Trong luận văn này, người viết thực hiện thực nghiệm bổ sung so sánh JDNA với thuật toán thuộc phương thức xác suất thống kê Huffman và thuật toán nén dựa trên bộ từ điển Lempel-Ziv để làm rõ hơn tính ưu việt của thuật toán nén tham chiếu như đã nhận định. Chi tiết thực nghiệm so sánh sẽ được trình bày ở chương 3 của luận văn.

CHƯƠNG 3 – THỰC NGHIỆM SO SÁNH THUẬT TOÁN JDNA VỚI THUẬT TOÁN MÃ HÓA HUFFMAN VÀ LEMPEL - ZIV

Ở chương này, người viết trình bày thực nghiệm bổ sung để minh họa thêm về tính hiệu quả của thuật toán nén tham chiếu đối với nén chuỗi gen DNA mà tiêu biểu là thuật toán JDNA so với hai thuật toán thuộc loại khác là Lempel-Ziv, thuật toán nén dựa trên từ điển và Huffman, thuật toán nén dựa trên xác suất thống kê. Như đã trình bày ở chương 1, có 4 loại thuật toán được sử dụng cho nén chuỗi gen. Thuật toán mã hóa bit dùng phương pháp mã hóa hai hoặc nhiều kí tự trong một byte với độ dài mã hóa cố định, ở trường hợp này nén chuỗi gen với 4 bazơ đặc trưng sẽ cho tỉ lệ nén cố định là 4:1. Thuật toán nén cơ sở từ điển cho tỉ lệ nén tốt hơn với phương pháp thay thế các chuỗi lặp bằng tham chiếu tới một từ điển được xác định trước và có thể mở rộng trong quá trình thực hiện. Lempel-Ziv là một thuật toán tiêu biểu của phương thức này đạt được tỉ lệ nén trong khoảng 4:1 tới 6:1 tùy thuộc tần suất lặp trong chuỗi gen được nén. Thuật toán nén hiệu quả thứ 3 là thuật toán nén xác suất thống kê, xuất phát từ việc sử dụng mô hình xác suất. Dựa trên các chuỗi khớp từng phần của đầu vào mà dự đoán các kí tự tiếp theo trong chuỗi, tỉ lệ nén đạt được là cao nếu dự đoán là đáng tin cậy. Một trong những thuật toán mã hóa xác suất tốt nhất được sử dụng là mã hóa Huffman. Tỉ lệ nén của thuật toán xác suất thường trong khoảng từ 4:1 tới 8:1. Thuật toán nén tham chiếu gần đây được biết đến như là loại thuật toán thứ 4 dùng cho nén chuỗi gen nhưng đã thể hiện được tính ưu việt về tốc độ nén, tỉ lệ nén và không gian lưu trữ. Thuật toán nén JDNA đã được người viết trình bày ở chương 2 là một thuật toán nén tham chiếu dựa trên thư viện và mã nguồn mở của FRESCO với những cải tiến mang lại hiệu quả vượt trội về tỉ lệ nén và dung lượng lưu trữ. Sau đây, người viết trình bày về thực nghiệm mà người viết đã thực hiện để làm rõ hơn nhận định về tính hiệu quả mà thuật toán nén tham chiếu, điển hình là JDNA đã mang lại cho việc nén chuỗi gen.

3.1. Môi trường thực nghiệm

Tất cả thực nghiệm được thực hiện trên máy tính cá nhân Dell Latitude E6420 với cấu hình như sau:

- CPU: Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz / L2 cache
- Bộ nhớ: 6GB RAM (1x4GB, 1x2GB)/ DIMM
- Dung lượng: 250GB/ SCSI/ Disk drives WDC WD2500BEKT-75PVMT0

Phần mềm sử dụng: Các chương trình được chạy trên nền Linux kernel (64-bit). JDNA mã nguồn mở được viết và chỉnh sửa bằng ngôn ngữ Java sử dụng

Oracle Java 7 JVM (build 1.7.0 40-b43). Huffman và Lempel Ziv (LZW) được viết và chỉnh sửa bằng ngôn ngữ C++.

Các kích thước đo bằng byte, ví dụ 1MB có nghĩa là 1000000 byte. Thuật ngữ "hệ số nén" được sử dụng để biểu diễn nghịch đảo của tỉ lệ nén, ví dụ một hệ số nén 10 nghĩa là tỉ lệ nén là 10:1.

Các tập dữ liệu thực nghiệm: Người viết thực hiện so sánh ba thuật toán nén trên ba tập dữ liệu sinh học: (1) tập hợp gen người, (2) tập hợp gen từ cây Arabidopsis thaliana và (3) tập hợp gen khuẩn men.

(1) Tập dữ liệu đầu tiên là gen người được lấy từ genBank dùng cho nghiên cứu. Trích rút ra một chuỗi liên ứng mỗi loại cho các gen. Sử dụng H-# để biểu diễn tập tất cả chuỗi cho nhiễm sắc thể người #, ví dụ H-1 biểu diễn nhiễm sắc thể người 1. Các chuỗi lấy từ cùng nhiễm sắc thể sẽ có độ tương đồng cao hơn các chuỗi lấy từ các nhiễm sắc thể khác nhau. Tập tất cả 23 tập dữ liệu gen người (H-1 tới H-22, H-X) được kí hiệu là H-*. Tập dữ liệu gen người lớn nhất là H-1 với 65631142 byte (62.6MB), tập dữ liệu nhỏ nhất là H-22 với 9953567 byte (9.5MB) và kích thước H-* khoảng 50000000 byte (5Gb).

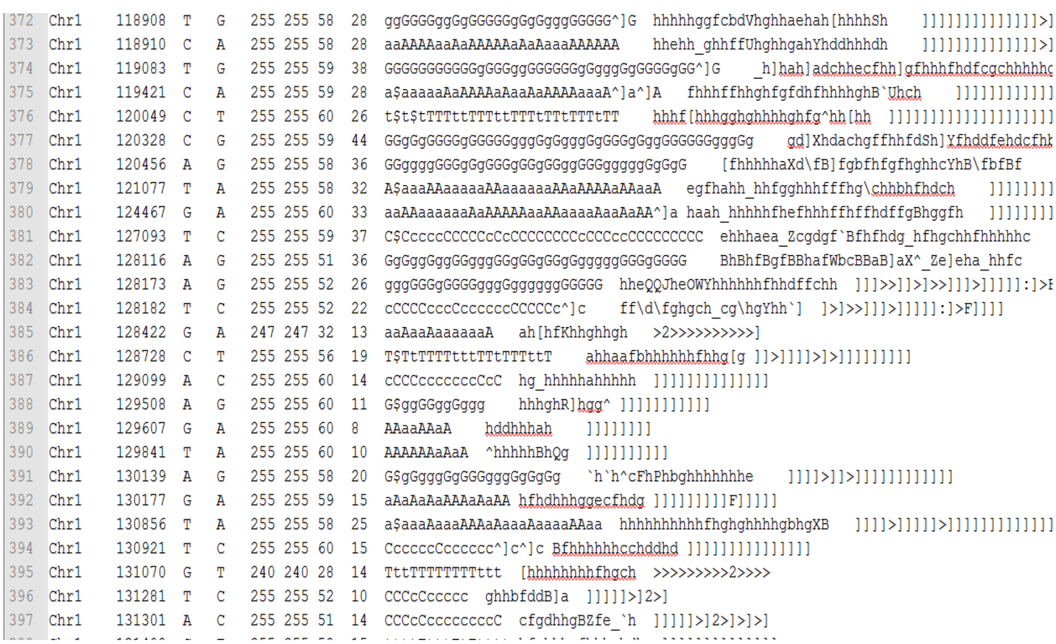
(2) Các tập dữ liệu Arabidopsis thaliana được lấy từ dự án 1001 gen xuất bản tại GMINordborg2010. Tập hợp tất cả tập dữ liệu Arabidopsis thaliana được kí hiệu là AT-*. Các chuỗi được lưu trong tệp SNPs tương ứng tham chiếu TAIR9. Tập dữ liệu Arabidopsis thaliana nhỏ nhất là AT_Bil-5 với 34110000 byte (34.1MB). Tập lớn nhất là AT_Aedal-1 với 70976000 byte (70.9MB) và kích thước AT-* vào khoảng 362500000 byte (2.9Gb).

(3) Tập dữ liệu sau cùng là tập hợp các gen khuẩn men. Tổng cộng đã tải xuống 16 chuỗi khuẩn men, mỗi chuỗi được cung cấp theo định dạng FASTA. Tập dữ liệu khuẩn men được kí hiệu là Y-WG kích thước khoảng 25000000 byte (0.2Gb).

Dữ liệu trong tệp gen nén có dạng chuỗi. Các hình 3.1, 3.2 và 3.3 dưới đây thể hiện định dạng chuỗi gen trong các tập dữ liệu thực nghiệm.

```
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNCCACTTACAGATCCTTCAAAAAGAGTGTTTCAAAACTGCTCTATGAAAAGGAATGTTCAACTCT
GTGAGTTAAATAAAAGCATCAAAAAAAAGTTTCTGAGAATGCTTCTGTCTAGTTTTTATGTGAAGATATT
TCCATTTTCTCTATAAGCCTCAAAGCTGTCCAAATGTCCACTTGCAGATACTACAAAAAGAGTGTTTCAA
AAGTGCTCAATGAAAAGGAATGTTCAGCTCTGTGAGTTAAATGCAAACATCACAAATAAGTTTCTGAGAA
TGCTTCTGTCTAGTTTTTATGGGAAGATAATTCCGTGTCCAGCGAAGGCTTCAAAGCTTTCAAAATATCC
ACTTGCAAATTCTACAAAAAGAGTGTTTCAAAGCTGCTTTATCAAAAGAAAGTTTCAACTCTGTGAGTTG
AATGTGCACATCACAAAGAAGTTTCTGAGAATGCCTTCAGTCTGGTTTTTATGTGAAGATATTCCCTTTT
CCAACGAAAGCCTCGAAGCTGTCCAAATATCCACTTGTAAGTGCTGCAAAAAGAGTGTTTCAAAACTGCT
ACAGCAAAAGAAAGGTTTATCTCTGTGAGTTGAGTAGACACATCAAGAAGAAATTTCTGAGAATGCTTCT
GTCTAGTTTTTATGTGAAGATATTTCCTTTGTCACCATAGGCCTCCAAGCCCTCCAAATGTCCACTTGCA
GATGCTACAAAAAGAGTGTTTCAAAACTGCTGTATGAAAAGAAATGCTCAAATCTGTGAGATAAATGCAT
ACATCACAAAGAAGTCTTTGAGAATGCTTCTGTCTAGTTTTTATGTTAAGATATTTCCTATTTCACCATA
CGTCTCAACGCACACAAAATGTACACTTGCAGATGCTACAAAGAGAGTGTTTCAAAACTTGTAGATCAAA
ACAAGTGTTCAACTTTGTGAGTTGAGGACACACATCTGAAAGAAGTTTCTGAGAATGCTTCTGTCTAGTT
TTTATGTGAAGATATTCCCGTTTCCAGCGAAAGCCCCAAAACTATCCAAATATCCACTTGCACATTCTAC
AAAAAGAGTGTTTCAAATCTGCTCTATCAAAATAAAGGTTCAACTCTGTGAGTTGACTACACACATCACA
AAGAAGTTTCTAAGAATGCTTCTGTCTGGTTTTTATGGGAAGATATTTCCTTTTTCAACATAGGCCTTGC
AGCATCTACAAAAAGAGTTTTTCAAAACTCCTCTAAGAAAAGGAATGTTCAACTCCATGAGTTTAATGCA
```

Hình 3.1. Định dạng tệp dữ liệu gen người H-22



```
Chr1    118908  T   G   255 255 58  28  ggGGGGggGgGGGGGgGgGgggGGGGG^]G  hhhhhggfcbdVhghhaehah[hhhhSh    ]]]]]]]]]]]]]]>]
Chr1    118910  C   A   255 255 58  28  aaAAAAaaAaAAAAAaAaAaaaAAAAAA    hhehh_ghhffUhghhgahYhddhhhdh    ]]]]]]]]]]]]]]>]
Chr1    119083  T   G   255 255 59  38  GGGGGGGGGGGgGGGggGGGGGGgGgggGgGGGGgGG^]G    _h]hah]adchhecfhh]gfhhhfhdfcgchhhhhc
Chr1    119421  C   A   255 255 59  28  a$aaaaaAaAAAAaAaaAaAAAAaaaA^]a^]A   fhhhffhhghfgfdhfhhhhghB`Uhch    ]]]]]]]]]]]
Chr1    120049  C   T   255 255 60  26  t$t$tTTTttTTTttTTTtTTtTTTtTT    hhhf[hhhgghghhhhghfg^hh[hh  ]]]]]]]]]]]]]]]]]]]
Chr1    120328  C   G   255 255 59  44  GGgGgGGGGgGGGGgggGgGgggGgGGGgGggGGGGGgggGg  gd]XhdachgffhhfdSh]Yfhddfehdcfhh
Chr1    120456  A   G   255 255 58  36  GGggggGGGgGgGGGgGGgGGggGGGgggggGgGgG    [fhhhhhaXd\fB]fgbfhfgfhghhcYhB\fbfBf
Chr1    121077  T   A   255 255 58  32  A$aaaAAaaaaaAAaaaaaaAAaAAAAaAAaaA   egfhahh_hhfgghhhfffhg\chhbhfhdch    ]]]]]]]
Chr1    124467  G   A   255 255 60  33  aaAAaaaaaaAaAAAAAaaAAaaaaAaaAaAA^]a haah_hhhhhfhefhhhffhffhdffgBhggfh    ]]]]]]]
Chr1    127093  T   C   255 255 59  37  C$CccccCCCCCcCcCCCCCCCCcCCCccCCCCCCCCC  ehhhaea_Zcgdgf`Bfhfhdg_hfhgchhfhhhhhc
Chr1    128116  A   G   255 255 51  36  GgGggGggGGgggGGgGGgGGgGggggGGGgGGGG    BhBhfBgfBBhafWbcBBaB]aX^_Ze]eha_hhfc
Chr1    128173  A   G   255 255 52  26  gggGGGgGGGGgggGggggggGGGGG  hheQQJheOWYhhhhhhfhhdffchh  ]]]>>]]>]>>]]]>]]]]]:]>F
Chr1    128182  T   C   255 255 52  22  cCCCCcccCcccccccCCCCCc^]c   ff\d\fghgch_cg\hgYhh`]  ]>]>>]]]>]]]]]:]>F]]]]
Chr1    128422  G   A   247 247 32  13  aaAaaAaaaaaaA   ah[hfKhhghhgh   >2>>>>>>>>>>]
Chr1    128728  C   T   255 255 56  19  T$TtTTTTtttTTtTTTttT    ahhaafbhhhhhhfhhg[g ]]>]]]]>]>]]]]]]]]
Chr1    129099  A   C   255 255 60  14  cCCCcccccccCcC  hg_hhhhhahhhhh  ]]]]]]]]]]]]]]
Chr1    129508  A   G   255 255 60  11  G$ggGGggGggg    hhhghR]hgg^ ]]]]]]]]]]]
Chr1    129607  G   A   255 255 60  8   AAaaAAaA    hddhhhah    ]]]]]]]]
Chr1    129841  T   A   255 255 60  10  AAAAAAaAaA  ^hhhhhBhQg  ]]]]]]]]]]
Chr1    130139  A   G   255 255 58  20  G$gGgggGgGGGgggGgGgGg   `h`h^cFhPhbghhhhhhhe    ]]]]>]]>]]]]]]]]]]]]
Chr1    130177  G   A   255 255 59  15  aAaAaAaAAAaAaAA hfhdhhhggecfhdg ]]]]]]]]]F]]]]]
Chr1    130856  T   A   255 255 58  25  a$aaaAaaaAAAaAaaaAaaaaAAaa  hhhhhhhhhhfhghghhhhgbhgXB   ]]]]>]]]]]>]]]]]]]]]]]]]
Chr1    130921  T   C   255 255 60  15  CccccCCcccccc^]c^]c Bfhhhhhhcchddhd ]]]]]]]]]]]]]]]
Chr1    131070  G   T   240 240 28  14  TttTTTTTTTTttt  [hhhhhhhhfhgch  >>>>>>>>>2>>>>
Chr1    131281  T   C   255 255 52  10  CCCcCccccc  ghhbfddB]a  ]]]]]>]2>]
Chr1    131301  A   C   255 255 51  14  CCCcCcccccccccC cfgdhhgBZfe_`h  ]]]]]>]2>]>]>]
```

Hình 3.2. Định dạng tệp dữ liệu gen Arabidopsis thaliana AT-1

```
>tpg|BK006935.2| [organism=Saccharomyces cerevisiae S288c] [strain=S288c] [moltype=genomic] [chromosome=I] [note=R64-1-1
CCACACCACACCCACACACCCACACACCACACCACACCACACCCACACACACA
CATCCTAACACTACCCTAACACAGCCCTAATCTAACCCTGGCCAACCTGTCTCTCAACTT
ACCCTCCATTACCCTGCCTCCACTCGTTACCCTGTCCCATTCAACCATACCACTCCGAAC
CACCATCCATCCCTCTACTTACTACCACTCACCCACCGTTACCCTCCAATTACCCATATC
CAACCCACTGCCACTTACCCTACCATTACCCTACCATCCACCATGACCTACTCACCATAC
TGTTCTTCTACCCACCATATTGAAACGCTAACAAATGATCGTAAATAACACACACGTGCT
TACCCTACCACTTTATACCACCACCACATGCCATACTCACCCTCACTTGTATACTGATTT
TACGTACGCACACGGATGCTACAGTATATACCATCTCAAACTTACCCTACTCTCAGATTC
CACTTCACTCCATGGCCCATCTCTCACTGAATCAGTACCAAATGCACTCACATCATTATG
CACGGCACTTGCCTCAGCGGTCTATACCCTGTGCCATTTACCCATAACGCCCATCATTAT
CCACATTTTGATATCTATATCTCATTCGGCGGTCCCAAATATTGTATAACTGCCCTTAAT
ACATACGTTATACCACTTTTGCACCATATACTTACCACTCCATTTATATACACTTATGTC
AATATTACAGAAAAATCCCCACAAAAATCACCTAAACATAAAAATATTCTACTTTTCAAC
AATAATACATAAACATATTGGCTTGTGGTAGCAACACTATCATGGTATCACTAACGTAAA
AGTTCCTCAATATTGCAATTTGCTTGAACGGATGCTATTTCAGAATATTTCGTACTTACA
CAGGCCATACATTAGAATAATATGTCACATCACTGTCGTAACACTCTTTATTCACCGAGC
AATAATACGGTAGTGGCTCAAACTCATGCGGGTGCTATGATACAATTATATCTTATTTCC
ATTCCCATATGCTAACCGCAATATCCTAAAAGCATAACTGATGCATCTTTAATCTTGTAT
GTGACACTACTCATACGAAGGGACTATATCTAGTCAAGACGATACTGTGATAGGTACGTT
```

Hình 3.3. Định dạng tệp dữ liệu gen khuẩn men Y-WG

3.2. Thực nghiệm so sánh JDNA với Mã hóa Huffman và Lempel – Ziv

So sánh sự thực hiện của các thuật toán mã hóa Huffman và Lempel-Ziv với nén tham chiếu JDNA. Kết quả cho thấy tốc độ nén của Huffman khá tốt, trong khi JDNA đạt được hiệu quả vượt trội về hệ số nén và kích thước tệp nén.

So sánh đầu tiên như sau: với mỗi loài và mỗi nhiễm sắc thể, lựa chọn ngẫu nhiên một số chuỗi và áp dụng mỗi thuật toán lựa chọn cho các chuỗi ngẫu nhiên đó. Kết quả được thống kê và so sánh về kích thước gen sau khi nén, thời gian nén và hệ số nén của từng thuật toán cho một hoặc nhiều chuỗi gen cụ thể.

Các chương trình thuật toán được chạy trên máy ảo Linux bằng các dòng lệnh tương ứng.

(1) Lệnh nén chuỗi DNA sử dụng mã hóa Huffman:

echo 'hs_ref_GRCh38.p2_chr22.fa'

echo 'hs_ref_GRCh38.p2_chr22.fa' >> /vagrant/HuffmanArchiver-master/Result/timespan1.txt;

START=$(date +%s);

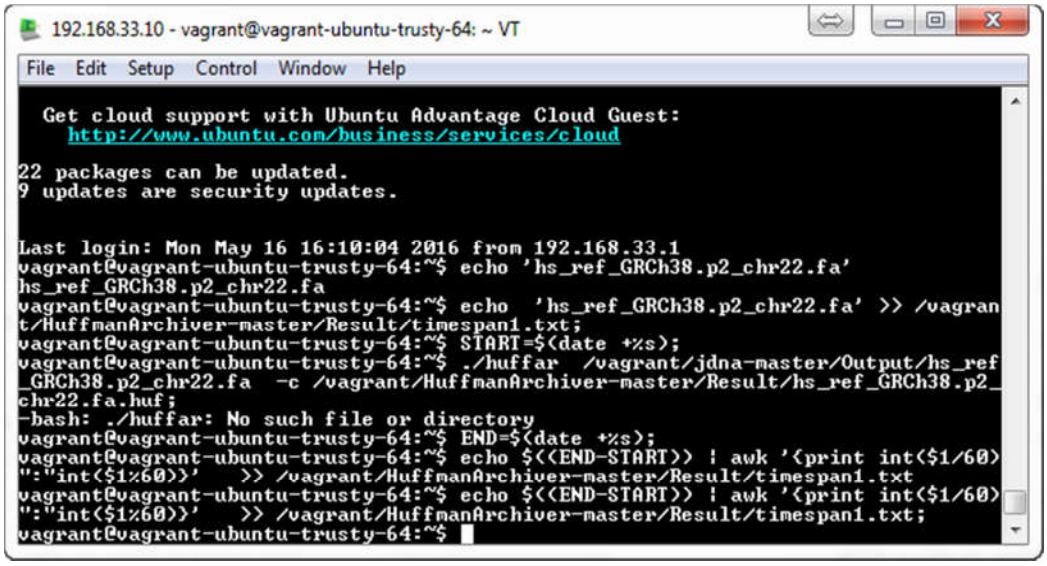
./huffar /vagrant/jdna-master/Output/hs_ref_GRCh38.p2_chr22.fa -c /vagrant/HuffmanArchiver-master/Result/hs_ref_GRCh38.p2_chr22.fa.huf;

END=$(date +%s);

echo $((END-START)) | awk '{print int($1/60)":"int($1%60)}' >> /vagrant/HuffmanArchiver-master/Result/timespan1.txt;

Trong đó, // *hs_ref_GRCh38.p2_chr22.fa* là tệp đầu vào và //*hs_ref_GRCh38.p2_chr22.fa.huf* là tệp nén đầu ra của thuật toán. Tệp

//timespan1.txt hiển thị thời gian nén. Hình 3.4 dưới đây thể hiện màn hình thực hiện chương trình thuật toán mã hóa Huffman.



Hình 3.4. Chương trình thuật toán mã hóa Huffman

(2) Lệnh nén chuỗi DNA sử dụng thuật toán LZW

echo 'hs_ref_GRCh38.p2_chr22.fa'

echo 'hs_ref_GRCh38.p2_chr22.fa' >> /vagrant/LZW/LZW-master/Result/timespan1.txt;

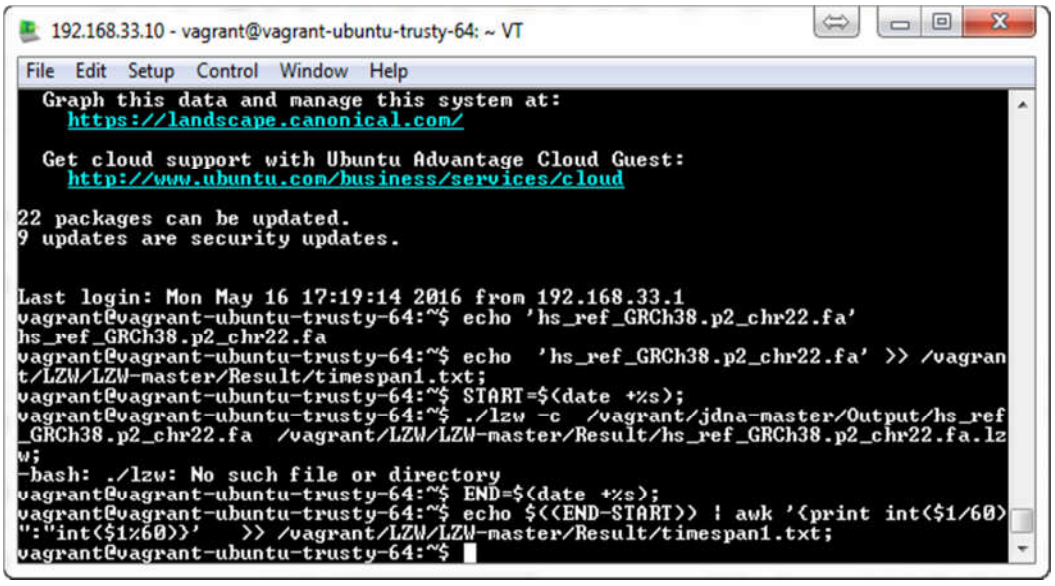
START=$(date +%s);

./lzw -c /vagrant/jdna-master/Output/hs_ref_GRCh38.p2_chr22.fa /vagrant/LZW/LZW-master/Result/hs_ref_GRCh38.p2_chr22.fa.lzw;

END=$(date +%s);

echo $((END-START)) | awk '{print int($1/60)":"int($1%60)}' >> /vagrant/LZW/LZW-master/Result/timespan1.txt;

Trong đó, // *hs_ref_GRCh38.p2_chr22.fa* là tệp đầu vào và //*hs_ref_GRCh38.p2_chr22.fa.lzw* là tệp nén đầu ra của thuật toán. Tệp *//timespan1.txt* hiển thị thời gian nén. Hình 3.5 thể hiện màn hình thực hiện chương trình thuật toán mã hóa Lempel-Ziv.



Hình 3.5. Chương trình thuật toán Lempel-Ziv (LZW)

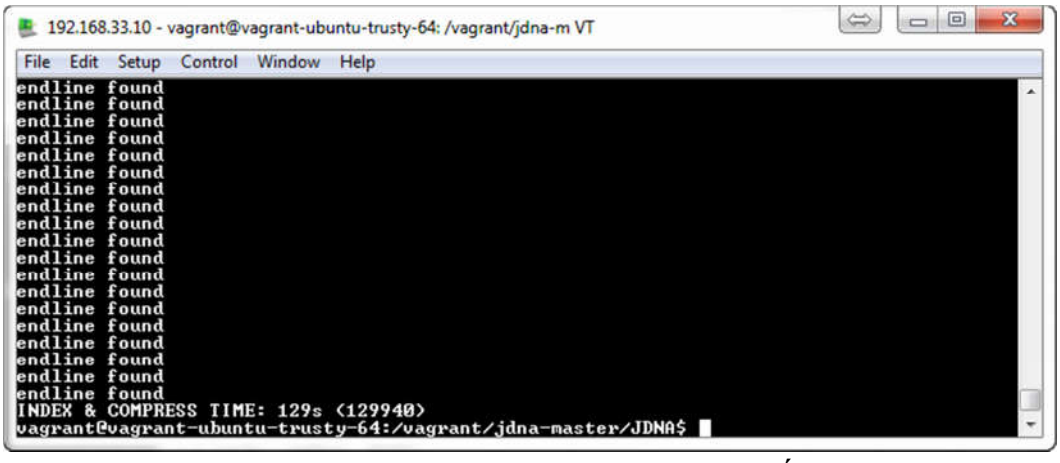
(3) Lệnh nén chuỗi DNA sử dụng thuật toán JDNA

alias java='java -Xmx4096M'

export _JAVA_OPTIONS="-Xmx4096M"

java -jar JDNA.jar COMPRESS ref_ex.raw hs_alt_CHM1_1.1_chr21.fa hs_alt_CHM1_1.1_chr21.fa.cmp

Trong đó, // *hs_alt_CHM1_1.1_chr21.fa* là tệp đầu vào và // *hs_alt_CHM1_1.1_chr21.fa.cmp* là tệp nén đầu ra của thuật toán. Hình 3.6 thể hiện màn hình thực hiện chương trình thuật toán JDNA.



Hình 3.6. Chương trình thuật toán tham chiếu JDNA

Trong nhiều trường hợp, việc nén dữ liệu thành công không đồng nghĩa với việc giải nén cũng thành công và đạt hiệu quả tốt như mong đợi. Vì lý do này mà trong khuôn khổ thực nghiệm so sánh bổ sung, người viết cũng đã chỉnh sửa

chương trình và thực hiện giải nén các chuỗi gen đã được nén. Ở phần nén thuật toán JDNA đã đạt được hiệu quả tốt hơn các thuật toán thuộc loại khác, kết quả sẽ được phân tích ở phần 3.3 dưới đây nên khi thực hiện so sánh hiệu quả giải nén, người viết sẽ chỉ thống kê kết quả về thời gian thực hiện để chứng minh tính ưu việt về thời gian giải nén của thuật toán JDNA so với hai thuật toán được lựa chọn để so sánh. Việc giải nén cũng được thực hiện bằng các dòng lệnh tương ứng chạy trên nền Linux.

3.3. Phân tích và đánh giá kết quả thực nghiệm

Bộ dữ liệu được tải về khá lớn, tổng cộng gần 100GB nhưng do môi trường thực nghiệm có hạn nên người viết chỉ lựa chọn ra một số chuỗi với dung lượng phù hợp để thực hiện quá trình nén, giải nén và so sánh. Hình 3.7 thể hiện bảng thống kê kết quả đạt được khi nén các tập dữ liệu sử dụng thuật toán nén Huffman, Lempel-Ziv và JDNA.

	Compressed size (in MB)			Runtime (in s)			Compression factor			Compression speed (in MB/s)		
Dataset	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**
hs_alt_CHM1_1.1_chr1.fa	79.62	61.90	**24.12**	8	118	**10518**	3.1166	4.0088	**10.2880**	9.9525	0.5246	**0.0023**
hs_ref_GRCh38.p2_chr21.fa	14.62	13.02	**11.11**	1	18	**59**	3.1646	3.5535	**4.1644**	14.6200	0.7233	**0.1883**
hs_ref_GRCh38.p2_chr22.fa	16.22	72.80	**10.75**	2	26	**68**	3.0216	0.6732	**4.5591**	8.1100	2.7999	**0.1581**
hs_ref_GRCh38.p2_chrY.fa	14.58	8.32	**7.36**	2	24	**178**	3.8879	6.8131	**7.7018**	7.2900	0.3467	**0.0413**
hs_alt_CHM1_1.1_chr22.mfa	19.69	60.80	**5.57**	2	29	**2935**	2.5796	0.8352	**9.1167**	9.8425	2.0966	**0.0019**
hs_ref_GRCh38.p2_chr21.mfa	19.00	14.45	**6.03**	2	18	**3524**	2.4357	3.2019	**7.6779**	9.4975	0.8028	**0.0017**
hs_ref_GRCh38.p2_chrY.mfa	17.58	9.03	**3.60**	2	24	**2913**	3.2242	6.2802	**15.7415**	8.7905	0.3761	**0.0012**
hs_alt_CHM1_1.1_chrX.mfa	62.16	45.75	**10.26**	6	77	**12783**	2.4729	3.3598	**14.9888**	10.3595	0.5942	**0.0008**
hs_alt_CHM1_1.1_chr15.mfa	41.41	28.97	**12.82**	5	72	**5027**	2.4492	3.5001	**7.9122**	8.2810	0.4024	**0.0025**
hs_alt_CHM1_1.1_chr16.mfa	37.83	23.72	**12.42**	3	63	**6290**	2.4027	3.8320	**7.3173**	12.6100	0.3765	**0.0020**
hs_ref_GRCh38.p2_chr19.mfa	22.83	16.97	**7.54**	2	40	**6752**	2.5432	3.4214	**7.7056**	11.4150	0.4243	**0.0011**
AT_Aedal-1	41.30	25.90	**7.33**	12	55	**10805**	1.7167	2.7375	**9.6829**	3.4417	0.4709	**0.0007**
AT_Aedal-3	33.20	21.00	**5.90**	9	49	**6742**	1.7078	2.7000	**9.6178**	3.6889	0.4286	**0.0009**
AT_Ale-Stenar-44-4	19.50	16.80	**4.40**	7	31	**5835**	2.1744	2.5238	**9.6455**	2.7857	0.5419	**0.0008**
AT_Algutsrum	27.90	15.90	**5.14**	8	37	**6790**	1.7814	3.1258	**9.6654**	3.4875	0.4297	**0.0008**
AT_App1-12	25.40	15.80	**4.57**	7	32	**10825**	1.6850	2.7089	**9.3681**	3.6286	0.4938	**0.0004**
AT_Baa1-2	22.90	15.30	**4.09**	6	29	**6850**	1.6419	2.4575	**9.1988**	3.8167	0.5276	**0.0006**
AT_Bil-5	20.00	13.70	**3.60**	5	32	**5750**	1.6300	2.3796	**9.0608**	4.0000	0.4281	**0.0006**
AT_Broet1-6	23.10	15.70	**4.08**	6	50	**6850**	1.6190	2.3822	**9.1871**	3.8500	0.3140	**0.0006**
AT_Doer-10	19.90	12.90	**3.52**	5	46	**4792**	1.7136	2.6434	**9.6958**	3.9800	0.2804	**0.0007**
AT_Dra2-1	30.10	19.20	**5.35**	9	88	**8515**	1.6711	2.6198	**9.4208**	3.3444	0.2182	**0.0006**
AT_Eden-1	22.10	14.50	**3.86**	6	22	**6850**	1.6878	2.5724	**9.6843**	3.6833	0.6591	**0.0006**
AT_Eds-1	21.50	13.50	**3.78**	6	20	**6665**	1.7023	2.7111	**9.6699**	3.5833	0.6750	**0.0006**
AT_Faeb-2	20.30	13.90	**3.67**	6	19	**6164**	1.6256	2.3741	**9.0106**	3.3833	0.7316	**0.0006**
AT_Gro-3	29.30	17.90	**5.24**	9	30	**7360**	1.7611	2.8827	**9.8633**	3.2556	0.5967	**0.0007**
Yst-1	32.60	20.60	**5.76**	10	32	**10200**	1.7117	2.7087	**9.8000**	3.2600	0.6438	**0.0006**
Y-WG4	0.44	0.42	**0.22**	1	1	**5**	3.5320	3.6746	**7.1620**	0.4380	0.4210	**0.0432**
AVERAGE	**27.22**	**22.55**	**6.74**	**5.44**	**40.07**	**6001.67**	**2.25**	**3.06**	**9.14**	**6.09**	**0.64**	**0.02**

Hình 3.7. Thống kê kết quả nén của các thuật toán Huffman, Lempel-Ziv và JDNA.

JDNA đạt hiệu quả về kích thước tệp nén (trung bình 6.74MB cho 27 tập dữ liệu) và hệ số nén tốt nhất 9.14 tức là tỉ lệ nén khoảng 9:1 cho các tập dữ liệu thực nghiệm. Lempel-Ziv đạt hiệu quả nén tốt thứ hai (trung bình 22.55MB cho 27 tập dữ liệu) và hệ số nén trung bình là 3.06. Huffman đạt được tốc độ nén khá tốt (trung bình 5.44 giây) nhưng lại chưa hiệu quả về kích thước tệp nén, hệ số nén cũng như không gian lưu trữ.

Thuật toán Huffman xử lý và mã hóa các chuỗi dựa trên xác suất xảy ra hay nói cách khác là tần số xuất hiện của các bazơ (A, C, G, T) nên với các chuỗi có mật độ các bazơ xảy ra cao thì kích thước nén sẽ được giảm đáng kể. Ngoài ra, thuật toán Huffman cũng là một điển hình của thuật toán "tham lam", thuật toán này tìm kiếm lựa chọn tối ưu địa phương ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục. Kết nối các nút gần nhau nhất để tạo ra một mã hóa dài hơn và cho kết quả mã hóa tối ưu về tổng thể. Tại mỗi bước của thuật toán, quy hoạch động đưa ra quyết định dựa trên các quyết định của bước trước, và có thể xét lại đường đi của bước trước hướng tới lời giải. Giải thuật tham lam quyết định sớm và thay đổi đường đi thuật toán theo quyết định đó, và không bao giờ xét lại các quyết định cũ. Đối với một số bài toán, đây có thể là một thuật toán không chính xác. Đây là lý do mà thuật toán Huffman tuy cho kết quả khá tốt về thời gian nén nhưng lại chưa tốt về tỉ lệ nén hay dung lượng tệp nén (trung bình kích thước tệp được nén là 27.22MB trên 27 chuỗi thực nghiệm).

JDNA sử dụng một chỉ số k-mer cho chuỗi tham chiếu. Lựa chọn *k* có một ảnh hưởng lớn lên tốc độ nén, nhưng hầu như không ảnh hưởng tới tỉ lệ nén. Với một giá trị *k* nhỏ thì nén được xác nhận là chậm hơn, do JDNA phải kiểm tra nhiều chuỗi khớp giả mà không liên quan tới nén tham chiếu bởi vì chúng không đạt được chuỗi khớp dài. Với giá trị *k* tương đối lớn thì tốc độ nén tăng đáng kể, và tỉ lệ nén được xác nhận là không thay đổi. Thời gian tạo chỉ số k-mer cho mỗi chuỗi tham chiếu là khoảng 1 phút đối với chuỗi lớn nhất và không bao gồm trong các phép tính toán. Ngoài ra JDNA còn được cải tiến để xử lý được các bazơ có độ tương đồng chưa cao, ngoài các bazơ đặc trưng A, T, G, C thì còn có các bazơ không xác định được đưa về dạng N. Điều này làm ảnh hưởng tương đối tới tốc độ nén của JDNA, có thể thấy trong thực nghiệm tốc độ nén của JDNA (trung bình 6001.67 giây cho 27 tập dữ liệu thực nghiệm) chậm hơn hai thuật toán nén Huffman và Lempel-Ziv. Điểm tương đồng giữa gen tham chiếu và gen đầu vào sẽ quyết định kết quả của thuật toán JDNA, trong đó sự tương đồng càng lớn thì tỉ lệ nén càng cao và thời gian nén cũng sẽ được cải thiện khá nhiều. Tuy nhiên, thực nghiệm đã đạt được hiệu quả đáng mong đợi về tỉ lệ nén, kích thước tệp nén và dung lượng lưu trữ cho nén chuỗi gen.

Bên cạnh JDNA thì còn một thuật toán thực nghiệm khác cũng cho kết quả thời gian nén chưa cao là Lempel-Ziv (trung bình 40.07 giây). Ở thuật toán Lempel-Ziv, bộ mã hóa sẽ kiểm tra chuỗi đầu vào bằng cách nhấn vào dịch vụ cửa sổ trượt gồm 2 phần: bộ đệm tìm kiếm và bộ đệm xem thẳng. Một bộ đệm tìm kiếm gồm một phần chuỗi mới được mã hóa và bộ đệm xem thẳng gồm phần tiếp theo của chuỗi sẽ được mã hóa. Trong khi đó, Lempel-Ziv còn đề xuất

là mã hóa toàn bộ độ dài chuỗi và cả phần bù, thậm chí cả chuỗi tìm thấy mà không khớp, bộ đệm tìm kiếm dài hàng nghìn bytes, trong khi bộ đệm xem thẳng chỉ 10 bytes. Quá trình mã hóa tiêu tốn thời gian do phải thực hiện số lượng so sánh lớn để tìm mẫu khớp. Với những lý do như vậy mà Lempel-Ziv dù đạt được dung lượng nén và hệ số nén khá hơn mã hóa Huffman nhưng lại tốn thời gian nén hơn.

Trong quá trình thực nghiệm, người viết đã thấy rằng kết quả đạt được của 3 thuật toán là khá nhất quán với các nhiễm sắc thể khác nhau trong cùng loài, với các loài khác nhau và định dạng tệp dữ liệu khác nhau. Tóm lại, thuật toán nén tham chiếu JDNA luôn đạt được kết quả nén tốt nhất về tỉ lệ nén và giảm kích thước tệp nén đáng kể với hiệu quả về không gian lưu trữ tuy phải tốn khá nhiều thời gian cho xử lý những bazơ có độ tương đồng chưa cao. Thuật toán nén xác suất Huffman tuy đạt tốc độ nén cao nhất nhưng lại kém nhất về tỉ lệ nén và kích thước tệp nén. Thuật toán nén dựa trên bộ từ điển Lempel-Ziv ở giữa với tỉ lệ nén không bằng JDNA, tốc độ nén kém Huffman nhưng kích thước tệp nén và hệ số nén lại có phần nhỉnh hơn Huffman. Hình 3.8 tóm tắt kết quả nén trung bình của 3 thuật toán JDNA, Lempel-Ziv và Huffman cho tổng thể 3 tập dữ liệu thực nghiệm.

	Compressed size (in MB)			Runtime (in s)			Compression factor			Compression speed (in MB/s)		
Dataset	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**
H-*	31.41	32.34	**10.14**	3.18	46.27	**4640.64**	2.85	3.59	**8.83**	10.070	0.861	**0.036**
AT-*	25.46	16.57	**4.61**	7.21	38.57	**7199.50**	1.72	2.63	**9.48**	3.566	0.485	**0.001**
Y-*	16.52	10.51	**2.99**	5.50	16.50	**5102.50**	2.62	3.19	**8.48**	1.849	0.532	**0.022**
AVERAGE	**24.46**	**19.81**	**5.91**	**5.30**	**33.78**	**5647.55**	**2.40**	**3.14**	**8.93**	**5.16**	**0.63**	**0.02**

Hình 3.8. Tóm tắt kết quả nén đạt được từ các thuật toán JDNA, Lempel-Ziv và Huffman

Kết quả thực nghiệm đã cho thấy JDNA đạt hiệu quả tốt hơn hai thuật toán thuộc loại khác là nén dựa trên từ điển Lempel-Ziv và nén xác suất thống kê Huffman.

Như đã trình bày ở trên, JDNA không chỉ là thuật toán hiệu quả về tỉ lệ nén và tối ưu dung lượng lưu trữ mà còn rất hiệu quả về thời gian khi thực hiện giải nén. Hình 3.9 dưới đây thể hiện so sánh thời gian giải nén của JDNA so với Huffman và LZW. Kết quả cho thấy thời gian giải nén của JDNA nhanh hơn hai thuật toán Huffman và LZW một bậc. Trong khi thời gian giải nén trung bình của LZW là 15.3 giây cho các chuỗi gen thực nghiệm và thời gian giải nén trung bình của Huffman xếp thứ hai với 4.26 giây thì thời gian giải nén của JDNA cho các chuỗi lựa chọn chỉ mất 1.44 giây.

	Compressed size (in MB)			Decompresion Runtime (in s)		
Dataset	Huffman	LZW	**JDNA**	Huffman	LZW	**JDNA**
hs_alt_CHM1_1.1_chr1.fa	79.62	61.90	**24.12**	14	28	**2**
hs_ref_GRCh38.p2_chr21.fa	14.62	13.02	**11.11**	2	9	**2**
hs_ref_GRCh38.p2_chr22.fa	16.22	72.80	**10.75**	2	38	**2**
hs_ref_GRCh38.p2_chrY.fa	14.58	8.32	**7.36**	6	5	**1**
hs_alt_CHM1_1.1_chr22.mfa	19.69	60.80	**5.57**	4	33	**0.001**
hs_ref_GRCh38.p2_chr21.mfa	19.00	14.45	**6.03**	3	10	**1**
hs_ref_GRCh38.p2_chrY.mfa	17.58	9.03	**3.60**	9	6	**1**
hs_alt_CHM1_1.1_chrX.mfa	62.16	45.75	**10.26**	5	21	**0.001**
hs_alt_CHM1_1.1_chr15.mfa	41.41	28.97	**12.82**	5	41	**0.001**
hs_alt_CHM1_1.1_chr16.mfa	37.83	23.72	**12.42**	6	12	**2**
hs_ref_GRCh38.p2_chr19.mfa	22.83	16.97	**7.54**	2	8	**1**
AT_Aedal-1	41.30	25.90	**7.33**	3	14	**3**
AT_Aedal-3	33.20	21.00	**5.90**	2	11	**2**
AT_Ale-Stenar-44-4	19.50	16.80	**4.40**	3	11	**0.001**
AT_Algutsrum	27.90	15.90	**5.14**	3	10	**2**
AT_App1-12	25.40	15.80	**4.57**	2	11	**0.001**
AT_Baa1-2	22.90	15.30	**4.09**	2	9	**2**
AT_Bil-5	20.00	13.70	**3.60**	2	11	**0.001**
AT_Broet1-6	23.10	15.70	**4.08**	4	14	**3**
AT_Doer-10	19.90	12.90	**3.52**	3	12	**3**
AT_Dra2-1	30.10	19.20	**5.35**	4	14	**3**
AT_Eden-1	22.10	14.50	**3.86**	6	8	**1**
AT_Eds-1	21.50	13.50	**3.78**	6	20	**2**
AT_Faeb-2	20.30	13.90	**3.67**	6	19	**0.001**
AT_Gro-3	29.30	17.90	**5.24**	3	12	**3**
Yst-1	32.60	20.60	**5.76**	3	14	**3**
Y-WG4	0.44	0.42	**0.22**	5	12	**0.001**
AVERAGE	**27.22**	**22.55**	**6.74**	**4.26**	**15.30**	**1.44**

Hình 3.9. Thống kê kết quả giải nén của các thuật toán Huffman, Lempel-Ziv và JDNA.

Kết quả nén và giải nén đều cho thấy thuật toán Lempel-Ziv tốn thời gian nhiều nhất cho việc thực hiện. Điều này là do Lempel-Ziv trong quá trình nén phải tạo ra từ điển khi gặp các chuỗi không khớp dài về khoảng và khi giải nén thì không có từ điển ngoài nên tạo ra vấn đề trong khi giải mã trên máy khác. Ở thuật toán này, cứ khi nào mà không có mẫu khớp nào thì nó sẽ mã hóa chuỗi đó như là độ dài và phần bù, điều này sẽ làm tốn không gian và bước không cần thiết này cũng làm tăng thời gian thực hiện thuật toán. Thuật toán xác suất thống kê Huffman vẫn giữ kết quả về thời gian thực hiện khá tốt, đứng thứ 2 trong 3 thuật toán thực nghiệm. Thuật toán nén tham chiếu JDNA tuy phải dùng tới thời gian nén lâu hơn do phải xử lý các phần bù trong chuỗi gen (những thành phần không phải A, T, G, C) và cả những phần chưa tương đồng trong chuỗi gen nhưng đã cho thấy hiệu quả về thời gian khi giải nén là tốt hơn rất nhiều so với

hai thuật toán Lempel-Ziv và Huffman, trung bình chỉ mất 1.44 giây cho giải nén các tập dữ liệu thực nghiệm đã nén.

Như vậy sau quá trình thực hiện thực nghiệm, kết quả đã cho thấy thuật toán nén tham chiếu JDNA đạt được hiệu quả rất khả quan cho việc nén chuỗi gen. Không chỉ đạt được hiệu quả về tỉ lệ nén cao, kích thước tệp gen nén giảm rõ rệt, tiết kiệm dung lượng lưu trữ mà JDNA còn đạt được sự ưu việt về thời gian giải nén đáng mong đợi.

KẾT LUẬN

Mặc dù những thách thức đối với lưu trữ thông tin chuỗi DNA cho tới nay đã có thể kiểm soát phần nào nhưng việc cải tiến trong sắp xếp chuỗi đa lượng và phương pháp nén tốt hơn cho chuỗi DNA vẫn là một vấn đề quan trọng đối với cộng đồng sinh học, nhất là những tiềm năng trong việc kiểm soát việc mất thông tin trong hoặc sau quá trình nén/giải nén chuỗi gen.

Sắp xếp chuỗi đa lượng (HTS) tạo nên một cuộc cách mạng trong nghiên cứu sinh học phân tử [44]. Công nghệ cung cấp những phương thức nén hiệu quả cho tập dữ liệu DNA khổng lồ. Thêm vào đó là những thách thức trong việc hiểu cấu trúc, chức năng và tiến hóa của hệ gen, những phương pháp sắp xếp chuỗi đa lượng cũng đặt ra câu hỏi và tập trung vào việc biểu diễn, lưu trữ, truyền tải, truy vấn và bảo vệ thông tin chuỗi gen.

Trong luận văn này, người viết đã trình bày các phương thức và thuật toán nén tiêu biểu cho mỗi phương thức nén dữ liệu chuỗi DNA. Trong đó, người viết chọn phương thức nén tham chiếu và thuật toán nén tiêu biểu JDNA làm mục tiêu nghiên cứu chính vì những hiệu quả mà thuật toán này mang lại cho nén DNA như tiết kiệm không gian lưu trữ, tỉ lệ nén đạt được cao hơn các thuật toán nén loại khác một bậc. JDNA được phát triển dựa trên thuật toán được sử dụng bởi FRESCO [25]. Thuật toán đã đạt được hiệu quả trong việc tăng tỉ lệ nén chuỗi đa lượng bằng 3 phương pháp kế thừa: (1) lựa chọn tham chiếu, (2) viết lại tham chiếu và (3) nén thứ tự hai. Tỉ lệ nén có thể đạt 400:1 hoặc cao hơn với những kế thừa ở điều kiện lý tưởng về chuỗi tham chiếu lựa chọn phù hợp hay chuỗi gen cùng loài có độ tương đồng cao. Bên cạnh những đặc trưng kế thừa từ thuật toán nén tham chiếu Fresco, JDNA còn thực sự hiệu quả khi sử dụng phương pháp đánh chỉ số theo yêu cầu để tiết kiệm thời gian nén thực và tăng tỉ lệ nén đáng kể. Đóng góp chính của JDNA là sử dụng phương thức đánh chỉ số theo yêu cầu. Cơ chế này kết hợp được hai đặc tính tốt nhất đó là: một cấu trúc chỉ số khá đơn giản xử lý những khác biệt chính giữa các tệp gen và nén nhanh các chuỗi khớp trực tiếp.

Đạt được những ưu việt về tỉ lệ nén, thời gian giải nén và không gian lưu trữ. Đồng thời xử lý và nén được nhiều định dạng tệp gen. Nhưng JDNA lại gặp bất lợi về thời gian nén do phải xử lý những chuỗi gen có sự tương đồng chưa cao, gồm nhiều những kí tự khác các bazơ đặc trưng (A, T, G, C) và chỉ đạt được tỉ lệ nén cao với các chuỗi DNA đã được sắp xếp. JDNA cũng bị hạn chế hiệu suất bởi JVM, trong đó việc quản lý bộ nhớ phức tạp của JVM cũng làm tăng độ khó khăn trong việc tạo ra một ứng dụng bộ nhớ hiệu quả. Để nén toàn

bộ hệ gen, hiệu suất JDNA có thể được tăng nhờ cơ chế song song. JDNA nén các tệp lớn theo các khối độc lập mà được nén riêng biệt. Cơ chế song song có thể làm tăng việc sử dụng vùng nhớ nhưng sẽ giảm được thời gian nén đáng kể. Tuy gặp một số bất lợi về thời gian nén và dung lượng máy ảo JVM do sử dụng ngôn ngữ Java làm công cụ phát triển nhưng JDNA đã chứng minh được tính hiệu quả trong việc nén chuỗi gen của thuật toán nén tham chiếu. Trong tương lai JDNA có thể được tiếp tục cải tiến để đạt được tốc độ nén và hiệu suất lưu trữ đáng mong đợi.

Cùng với những nghiên cứu và nhận định đã trình bày, người viết cũng đã thực hiện thực nghiệm so sánh thuật toán tham chiếu JDNA với hai thuật toán nén thuộc phương thức khác là nén dựa trên bộ từ điển Lempel-Ziv và thuật toán nén xác suất thống kê Huffman để bổ sung cho kết quả nghiên cứu đạt được. Kết quả thực nghiệm tuy chưa đạt được tỉ lệ nén hay thời gian mong đợi cao nhất của thuật toán nén tham chiếu do một số hạn chế về môi trường thực nghiệm, nhưng đã bước đầu khẳng định được sự tối ưu của thuật toán nén tham chiếu mà tiêu biểu là JDNA cho nén chuỗi gen. Những kết quả thực nghiệm này sẽ là tiền đề để người viết tiếp tục những nghiên cứu và cải tiến cho việc nén chuỗi gen trong tương lai.

TÀI LIỆU THAM KHẢO

[1] Samantha Woodward BIOC 218. *A Critical Analysis of DNA Data Compression Methods, 2011.*

[2] Sebastian Wandelt, Marc Bux, and Ulf Leser. *Trends in Genome Compression, 2013.*

[3] P. Raja Rajeswari, Allam Apparo, and V. K. Kumar. *Genbit compress tool(gbc): A javabased tool to compress dna sequences and compute compression ratio(bits/base) of genomes.* CoRR, abs/1006.1193, 2010

[4] Rajendra Kumar Bharti, Archana Verma, and R.K. Singh. *A biological sequence compression based on cross chromosomal similarities using variable length lut.* International Journal of Biometrics and Bioinformatics, 4:217 – 223, 2011.

[5] Ateet Mehta and Bankim Patel. *Dna compression using hash based data structure.* International Journal of Information Technology & Knowledge Management, 3:383 – 386, 2010.

[6] Piyuan Lin, Shaopeng Liu, Lixia Zhang, et al. *Compressed pattern matching in dna sequences using multithreaded technology.* In 3rd International Conference on Bioinformatics and Biomedical Engineering, ICBBE'09, 2009.

[7] Pothuraju Rajarajeswari, Allam Apparao. *DNABIT Compress – Genome compression agorithm*, Journal on Bioinformation, Volume 5, Issue 8, January 2011.

[8] Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, et al. *Iterative dictionary construction for compression of large dna data sets.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 9(1):137 – 149, 2012

[9] Dimitris Antoniou, Evangelos Theodoridis, and Athanasios Tsakalidis. *Compressing biological sequences using self adjusting data structures.* In Information Technology and Applications in Biomedicine, 2010.

[10] K. R. Venugopal, K. G. Srinivasa, and Lalit Patnaik. *Probabilistic Approach for DNA Compression.* Chapter 14, pages 279 – 289. Springer, 2009.

[11] I.Tabus and G.Korodi. *Genome compression using normalized maximum likelihood models for constrained markov sources.* In Information Theory Workshop, 2008.

[12] Kalyan Kumar Kaipa, Kyusang Lee, Taejin Ahn, et al. *System for random access dna sequence compression.* In International Conference on Bioinformatics and Biomedicine Workshops, 2010.

[13] B. G. Chern, I. Ochoa, A. Manolakos, A. No, K. Venkat and T. Weissman,Department of Electrical Engineering, Stanford University, Stanford CA 94305. *Reference Based Genome Compression.*

[14] Suman M. Choudhary, Anjali S. Patel, Sonal J. Parmar. *Study of LZ77 and LZ78 Data Compression Techniques,* International Journal of Engineering Science and Innovative Technology (IJESIT), Volume 4, Issue 3, May 2015.

[15] M. D. Cao, T. Dix, L. Allison, and C. Mears. *A simple statistical algorithm for biological sequence compression*. In Data Compression Conference, 2007. DCC '07, pages 43 –52, march 2007.

[16] P.Raja Rajeswari, Dr. Allam Apparao, Dr. R.Kiran Kumar. *Huffbit Compress – Algorithm To Compress Dna Sequences Using Extended Binary Trees,* Journal of Theoretical & Applied Information Technology, Vol. 13 Issue 1/2, pages 101-106, 2010.

[17] I. H. G. S. Consortium. *Initial sequencing and analysis of the human genome*. Nature, 409(6822):860–921, February 2001.

[18] E. E. Schadt, S. Turner, and A. Kasarskis. *A window into third-generation sequencing*. Human molecular genetics, 19(R2):R227–R240, Oct. 2010.

[19] S. Deorowicz and S. Grabowski. *Robust relative compression of genomes with random access*. Bioinformatics, 27(21):2979–2986, 2011.

[20] C. Wang and D. Zhang. *A novel compression tool for efficient storage of genome resequencing data*. Nucleic Acids Research, 39(7):e45, Apr. 2011.

[21] Jim Dowling, KTH. *Reference Based Compression Algorithm*, Scalable, Secure Storage of Biobank Data, Work Package 2, pages 23 – 44, June 2014.

[22] M. Cohn and R. Khazan. *Parsing with prefix and suffix dictionaries*. In Data Compression Conference, pages 180–189, 1996.

[23] S. Grabowski and S. Deorowicz. *Engineering relative compression of genomes*. CoRR, abs/1103.2351, 2011.

[24] S. Kuruppu, S. J. Puglisi, and J. Zobel. *Optimized relative Lempel-Ziv compression of genomes*. In Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113, ACSC '11, pages 91–98, Darlinghurst, Australia, Australia, 2011.

[25] S.Wandelt and U.Leser. *Fresco: Referential compression of highly similar sequences*. Computational Biology and Bioinformatics, IEEE/ACM Transactions on, 10(5):1275–1288, Sept 2013.

[26] S.Kurtz, A.Narechania, J.Stein, and D.Ware. *A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes*. BMC Genomics, 9(1):517, 2008

[27] 1000 Genomes Project Consortium. *A map of human genome variation from populationscale sequencing. Nature*, 467(7319):1061–1073, October, 2010.

[28] P. Danecek, A. Auton, G. Abecasis, and 1000 Genomes Project Analysis Group. *The variant call format and VCFtools*. Bioinformatics, 27(15):2156–2158, August 2011.

[29] H.Mewes, K.Albermann, M.Bahr, D.Frishman, A.Gleissner, J.Hani, K.Heumann, K.Kleine, A.Maierl, S.Oliver, et al. *Overview of the yeast genome.* Nature, 387(6632):7–8, 1997

[30] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. *Relative lempel-ziv compression of genomes for large-scale storage and retrieval*. In

Proceedings of the 17th International Conference on String Processing and Information Retrieval, SPIRE'10, pages 201 – 206, 2010.

[31] A. J. Pinho, D. Pratas, and S. P. Garcia. *GReEn: a tool for efficient compression of genome resequencing data*. Nucleic Acids Research, December 2011.

[32] Marty C. Brandon, Douglas C. Wallace, and Pierre Baldi. *Data structures and compression algorithms for genomic sequence data*. Bioinformatics, 25(14):1731 – 1738, 2009.

[33] Scott Christley, Yiming Lu, Chen Li, et al. *Human genomes as email attachments*. Bioinformatics, 25(2):274 – 275, 2009.

[34] Hyoung Do Kim and Ju-Han Kim. *Dna data compression based on the whole genome sequence*. Journal of Convergence Information Technology, 4(3):82 – 85, 2009.

[35] Sebastian Kreft and Gonzalo Navarro. *Lz77-like compression with fast random access*. In Proceedings of the 2010 Conference on Data Compression, DCC'10, pages 239 – 248, 2010.

[36] Andrew Peel, Anthony Wirth, and Justin Zobel. *Collection-based compression using discovered long matching strings*. In Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM'11, pages 2361 – 2364, 2011.

[37] Pragya Pande and Dhruv Matani. *Compressing the human genome against a reference*. Technical report, Stony Brook University, 2011.

[38] Stéphane Grumbach and Fariza Tahi. *A new challenge for compression algorithms: genetic sequences.* Information Processing & Management, 30(6):875 – 886, 1994.

[39] Jesper Larsson and Alistair Mofat. *Offline dictionary-based compression*. In Proceedings of the 1999 Conference on Data Compression, DCC'99, pages 296 – 305, 1999.

[40] John G. Cleary, Ian, and Ian H. Witten. *Data compression using adaptive coding and partial string matching.* IEEE Transactions on Communications, 32:396 – 402, 1984.

[41] M. H. Fritz, R. Leinonen, G. Cochrane, et al. *Efficient storage of high throughput DNA sequencing data using reference-based compression*. Genome Research, 21(5):734–740, May 2011.

[42] Xin Chen, Sam Kwong, Ming Li. *A Compression Algorithm for DNA Sequences and Its Applications in Genome Comparison*, International Conference on Genome Informatics, 10:51-61, February 1999.

[43] Gregory Vey. *Differential direct coding - A compression algorithm for nucleotide sequence data*, Article ID bap013, June 2009.

[44] M. L. Metzker. *Sequencing technologies — the next generation*, Nat. Rev. Genet., 11(1):31–46, January 2010.

[45] M. R. Wick. *An object-oriented refactoring of Huffman encoding using the Java Collections Framework*. SIGCSE Bull., 35(1):283–287, January 2003.

[46] D. A. Huffman. *A method for the construction of minimum-redundancy codes*. Proceedings of the Institute of Radio Engineers, 40(9):1098–1101, September 1952.