单位代码 10006
学　　号 14021194
分 类 号 TP301

北京航空航天大學
BEIHANG UNIVERSITY

毕业设计（论文）

大规模人类基因数据的快速查询方法研究

学院名称 电子信息工程学院
专业名称 电子信息工程
学生姓名 李松晨
指导教师 Sebastian Wandelt

2018 年 6 月

大规模人类基因数据的快速查询方法研究

李松晨

北京航空航天大学

本科生毕业设计（论文）任务书

I、毕业设计（论文）题目：

大规模人类基因数据的快速查询方法研究

II、毕业设计（论文）使用的原始资料（数据）及设计要求

通过对于基因型压缩器(GenoType Compressor)的算法进行分析和实验，测试其在大规模人类基因数据中的查询速度，并尝试对其做出改进。所使用的原始数据为千人基因组计划中第三阶段第 22 号染色体数据。

III、毕业设计（论文）工作内容

论文通过模拟出的样本数量庞大的基因数据，来检验基因型压缩器在压缩过的大规模基因数据中查询的能力。通过实验和分析来判断其查询算法的时间复杂度，并通过优化算法实现方式以及为可移动列数受限制的重排序问题设计新的解决方法，来提出进一步加快查询速度的方案。

IV、主要参考资料

“GTC: an attempt to maintenance of huge genome collections compressed”, Agnieszka Danek, Sebastian Deorowicz

“Reproducible Simulations of Realistic Samples for Next-Generation Sequencing Studies Using Variant Simulation Tools”, Bo Peng

电子信息工程 学院 电子信息工程 专业类 140228 班

学生 李松晨

毕业设计（论文）时间：2018 年 1 月 2 日至 2018 年 6 月 1 日

答辩时间： 2018 年 6 月 7 日

成绩：

指导教师：

兼职教师或答疑教师（并指出所负责的部分）：

系（教研室）主任（签字）

本人声明

我声明，本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

作者： 李松晨

签字：

时间：2018 年 6 月

大规模人类基因数据的快速查询方法研究

学　　生：李松晨

指导教师：Sebastian Wandelt

摘要

随着科技的发展，采集人类基因序列的资金成本和时间成本都在降低，基因数据量的增长速度也日益加快。在此背景下，将大规模的人类基因数据压缩存储后，能够快速的从中查询所需要的信息便十分重要。因此，在世界上已有众多学者针对基因数据压缩和查询的算法展开了多种研究。由于现阶段不易获取大规模人类基因数据，我们选择了使用自己模拟出的基因数据来进行实验。在验证了模拟出的基因数据的可靠性后，利用最大规模为 50 万人的基因数据，我们测试了基因型压缩器在处理大规模基因数据时的查询速度，得出了其在查询 VCF 文件中某段位置上全部记录全部样本的基因型时，平均查询每条记录的时间随样本数量的增长呈现出略大于线性的增长趋势，证明了其有能力从压缩后的大规模人类基因数据中进行查询。随后，我们通过修改查询算法的实现方式，使得基因型压缩器的查询速度提升了约 10%，并在接下来的研究中，提出了一种可移动列数受限时通过重排序来减小矩阵相邻列间总汉明距离的算法，理论上可以在不过多损失压缩率的情况下，进一步提升查询基因记录的速度。

关键词：大规模人类基因数据，压缩，快速查询，汉明距离

Fast Query Method for Large Population DNA Data

Author: Li Song-chen

Tutor: Sebastian Wandelt

Abstract

With the development of science and technology, the capital and time cost of collecting human DNA sequences are decreasing, which leads to the accelerating growth rate of the total amount of DNA data. In this context, it is very important to be able to quickly query for the required information, from where large population DNA data is compressed and stored. Therefore, many researchers in the world have carried out various researches on the algorithm of DNA data compression and query. Since it is not easy to obtain large population human DNA data at this stage, we chose to use our simulated DNA data for experiments. After verifying the reliability of simulated DNA data, the query speed of GenoType Compressor was tested while the population was rising, using the simulated DNA data with a maximum size of 500,000 people. When querying genotypes of all samples and all records on a range of positions, the average runtime of querying one record grows slightly faster than linearly with respect to the number of samples, proving its ability to fast query from large population DNA data. At the same time, by modifying the implementation of the query algorithm, the query speed of GenoType Compressor was increased by about 10%. Furthermore, we proposed an algorithm for reducing the total Hamming distance between adjacent columns of a matrix by reordering a limited number of columns. Theoretically, further improvement of the query speed can be achieved without excessive loss of compression ratio, using the algorithm.

Key words: Large Population DNA Data, Compression, Fast Query, Hamming Distance

目 录

1 绪论

1.1 课题背景

人类的基因数据对于社会发展的很多方面有着巨大的作用，如诊断遗传病，进行生物医学研究，改善社会治安等。相应的例子如，国际刑警组织利用 DNA 数据库在犯罪嫌疑人频繁大范围移动的情况下对其进行追踪[1]。Doleac, Jennifer L 在[2]中指出，DNA 数据库可以遏制有记录的犯罪分子犯罪，减少犯罪率，并且比传统的执法工具更节约成本。随着千人基因组计划（1000 Genomes Project）的进行，在 2012 年，全球范围内有超过 1000 人的约 3 千 8 百万单核苷酸多态性（SNPs）被采集[3]，到 2015 年，更是有超过 2500 人的约 8 千 470 万 SNPs 被采集[4]。更进一步的，随着获取人类基因序列技术的不断成熟，成本不断降低，人类基因数据量的增长速度十分惊人。根据 Zachary D. Stephens, Skylar Y. Lee 总结的数据，在从 2005 至 2015 的十年左右时间，几乎每隔七个月，所获得的基因序列数据便会翻上一番。根据 Omicsmap 的报告，在 2015 年全世界就有多于 2500 家高产量的机构在收集基因数据，这些机构由不同等级的公司运营，分布在全世界 55 个国家的近 1000 个采样中心[5]。

在这样的情况下，未来会收集到规模巨大的基因数据是可以预料的。由于数据规模的庞大，对数据的压缩是必要的，然而对于压缩后的基因数据，在分析时需要从其中查询，也就是解压。压缩的数据规模极为庞大时，如何快速的从压缩后的数据中查询到所需要的数据便非常重要。快速的查询方法，意味着可以更高效的从采集到的大量基因序列中挖掘和分析信息，让基因数据物尽其用。

1.1.1 VCF 文件格式

VCF（Variant Call Format）是一种文本文件格式，常用来存储基因数据文件。它可以用来存储 DNA 多态性数据，如单核苷酸多态性(single nucleotide polymorphism,SNP)、染色体突增(Insertion)、缺失(deletions)和结构变异，通常还会记录有大量的标注数据[6]。这种格式最先是为了千人基因组计划而开发的[7]，在后来被广泛地用于记录基因数据。它通常包含一些信息行，用于记录一些相关的信息，如文件格式的版本，数据被记录的日期，使用的参考基因等等。VCF 文件还会包含一个头行，相当于一个表头，在头行下面，便是一行一行的记录。每个记录中，会记录这个记录记载的是哪个染色体（CHROM）

上，哪个位置(POS)的信息，同时会记录这条记录上的参考基因(REF)，以及因为变异或其他原因而产生的非参考基因(ALT)。一条记录上，只会出现一个参考基因，但是可能出现多个非参考基因。记录的最右侧记载的便是在这个文件中所统计的个体的基因型（Genotype）。一个正式的 VCF 文件中，除了上述内容中提到的信息外，还会记录其他的信息，如基因型质量（Genotype Quality）、单倍型质量（Haplotype Quality）等。图 1.1 是一个化简后的 VCF 文件示例[6]。下面对于示例中一些有助于理解 VCF 文件结构的关键信息做以说明。如示例中所展示的，5 条记录都来自第 20 号染色体，记录了 5 个不同的位置，其中第 4 条记录没有非参考基因，第 3 条记录所对应的 S2 的基因型为 T|G，分别对应第 2 个非参考基因和第 1 个非参考基因。第 5 条记录上，两个非参考基因，第一个对应为缺失了两个基因 T 和 C，第二个对应为插入了一个 T 基因。

##fileformat=VCFv4.2

##fileDate=20090805

##source=myImputationProgramV3.1

##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta

#CHROM	POS	REF	ALT	FORMAT	S1	S2	S3
20	14370	G	A	GT	0|0	1|0	1|1
20	17330	T	A	GT	0|0	0|1	0|0
20	1110696	A	G,T	GT	1|2	2|1	2|2
20	1230237	T	.	GT	0|0	0|0	0|0
20	1234567	GTC	G,GTCT	GT	0|1	0|2	1|1

图 1.1 一个简化的 VCF 文件示例

1.1.2 常用工具及压缩方法

VCF 格式的文件通常会以压缩文件的方式存储，因为压缩后的文件体积远小于原始的 VCF 文件，且由常用的压缩方式压缩的文件，有对应的工具可以方便地进行操作和分析。比较常见的方式有通过 tabix 工具进行 bgzip 压缩[8]，压缩时使用与 zlib 兼容的 BGZP 库[9]，压缩后的文件后缀名为.vcf.gz，通常会配有一个后缀名为.tbi 的对应的索引

文件。通过索引文件，能够在 bgzip 压缩的状态下直接索引所需要的信息。bgzip 的压缩效果非常明显，如对于千人基因组计划第 22 号染色体的数据文件，其 VCF 格式的文件大小为 10.4GB，而 bgzip 压缩后的文件大小只有 214.5MB，由此可见压缩的效率和意义。另外一种常见的存储 VCF 文件的方式为对其压缩的二进制版本 BCF2 文件进行存储，对应的文件格式后缀名为.bcf[6]。

VCFtools[7]和 BCFtools[6]是两种常用的查看和处理上述基因数据文件的工具，通常情况下，BCFtools 的速度要快于 VCFtools。

1.2 研究目的及意义

如上文中所述，随着人类基因数据量的不断增大，尤其是随着采集基因的成本降低，更多人的基因数据会被收集到，相比于只有几千几万人的基因数据，不久之后需要处理的很可能就是几十万甚至上百万人的基因数据。本次毕业设计的目的是，先从现有的表现较好的基因数据压缩与查询的方法入手，测试其在人口规模庞大的数据集上的表现，证明其是否已经为即将来到的大量人类基因数据做好了准备，并尝试对其做出改进，或提出新的方法来解决如何在如此大量的数据中进行快速查询的问题。

1.3 国内外研究现状

Layer R M, Kindlon N, Karczewski K J, et al.在 2016 年提出了基因型查询器（Genotype Query Tools，GQT），GQT 在查询个体基因型、表现型和关系时有很好的表现，然而 GQT 无法进行对于整条记录的查询[10]。Li H.提出了 BGT[11]，一种基于 positional Burrows-Wheeler 变换[12]的紧凑的数据格式，取得了比 GQT 更好的压缩效果，且既支持查询变异记录，又支持查询个体的全部基因型。Zheng X, Gogarten S M, Lawrence M, et al.研发了 SeqArray[13]，一种基于 R 语言的可以高效存储和查询人类基因数据的程序，在程序中可以使用 LZMA 算法[14]对基因数据进行压缩。Agnieszka Danek 和 Sebastian Deorowicz 在 2017 年提出了基因型压缩器（GenoTpye Compressor）[15]，通过将 Lempel-Ziv 压缩[16]、游程编码[17]和霍夫曼编码[18]等方法混合使用的压缩算法，对人类基因数据进行了更高压缩率的压缩，支持个体基因型、变异记录等多种方式的查询，并且可以较快地进行对于整条记录和样本全部基因型的查询。

1.4 研究内容及论文构成

1.4.1 论文研究内容

本篇论文的研究从实验出发，先模拟出适用于实验的大规模人类基因数据，然后用得到的数据，验证基因型压缩器是否能够处理数据量日益庞大的人类基因数据。在验证过程中，从理论和实验结果上对其查询算法的时间复杂度进行分析，并尝试进行改进。在改进的过程中，遇到了一种较为新颖的问题，在可移动列数受限制的情况下减小矩阵中相邻列间汉明距离的总和，并为这种问题设计了算法，以在不过多损失压缩率的基础上，提高对于大规模基因数据的查询速度。

1.4.2 论文章节安排

本篇论文的章节结构安排如下：

第二章介绍了对于大规模人类基因数据的收集。列举了现存的已经经过收集整理的人类基因数据集，然后介绍了本篇论文中主要使用的基因数据模拟器变异模拟工具（Variant Simulation Tools，VST）[19]的使用方法，并用实验选择了合适的用于模拟人类基因的参数，再根据参数制定出了模拟大规模人类基因数据的方案。

第三章介绍了对于基因型压缩器查询速度的测试，通过对实验数据进行拟合的方式，对于其查询过程的时间复杂度进行了分析，并且对于程序各部分的运行时间做了测试，确定了提高查询速度的突破口。

第四章详细介绍了基因型压缩器压缩和查询部分的算法，并结合第三章中得到的分析结果，对查询算法最耗时部分的时间复杂度、重要性皆做了论述。在这一章中对这部分程序进行了优化，提高了查询速度，并且提出了一种新的算法，来在只移动少量列的情况下，降低被压缩矩阵相邻列间的总汉明距离，为了进一步提高查询速度而不过多损失压缩率。

2 大规模人类基因数据收集

2.1 收集现有的人类基因数据

考虑到课题的目的是研究对于大规模人口的基因数据的查询方法，为了评估现有的人类基因数据压缩与查询方法，以及对未来的研究实验进行验证，收集到大量的人类基因数据就成为了不可缺少的工作。经过对于现有文献的查询整理，目前已经存在的人类基因数据集有千人基因组计划所收集的 2504 个人的基因数据，人类单倍型参考基因联盟(Haplotype Reference Consortium, HRC)收集的 64796 个单倍型数据[20]和外显子组聚合联盟（Exome Aggregation Consortium, ExAC）所收集到的 60706 个人的基因数据[21]等。后两个数据集的规模较大，无疑更适合用来作为课题的研究对象，然而在进行本次毕业设计时，我们无法获得后两个数据集的访问权，因而可以使用的只有千人基因组计划的数据集。

2.2 模拟基因数据

千人基因组计划的数据集只收集了 2504 个人的基因数据，因此无法满足实验对于人口规模的要求，又因为无法获取更多的真实基因数据，所以实验将使用模拟出的基因数据对基因型压缩器的查询速度进行评估。

2.2.1 选定模拟基因数据的模型

为了保证模拟出的基因数据的合理性，尤其是数据在被压缩后，查询时应具有与真实基因数据相类似的速度，需要仔细的选择用来模拟基因数据的工具和模型。本实验中使用了变异模拟工具来进行基因数据的模拟。

变异模拟工具通过一个流程文件来控制模拟基因数据时所使用的模型、原始数据集、基因位点选取区域、变异频率、人口繁衍代数以及最终的人口数。实验中选取的原始基因数据为千人基因组计划数据集，第二十二号染色体上，从位置 16050075 到位置 1800000 这一段数据。采用的模型是变异模拟工具中，Peng2014_ex1 中的 resample 模型[19]。这个模型所设计的本意就是从千人基因组计划中抽取一段数据并按照时间推演的方式演化出更多的人口，故而适合用来产生本课题所需要的数据集。

2.2.2 确定模拟基因数据的参数

在 resample 模型中，控制基因模拟的参数主要有规模(scale)，繁衍代数(T)和最终人口数(NT)。其中规模参数的作用是在模拟基因数据的过程中增加变异、自然选择和基因重组的概率。而对于存储基因数据的 VCF 文件，规模参数的提高就意味着在同样人口数的情况下，记录数量的增加。由于在对于基因进行模拟的过程中，在进行一代一代的繁衍时，需要将上一代的数据存储在内存中以供程序推演下一代的基因型，故而程序对于内存的消耗非常大。经过实验，发现对于实验所使用的拥有 8GB 内存的电脑，模拟从位置 16050075 到位置 18000000 一共约二百万个基因位点时，无法一次模拟出所需要的全部数据，所以需要采取分批模拟，并融合成不同规模基因数据的方式来产生进行实验所需的全部数据。

由于实验将使用模拟出的数据对于基因型压缩器的查询速度进行评估，为了保证实验的可靠性，对于模拟基因时所采用参数的选取就十分重要。经过实验观察，当把利用变异模拟工具产生的不同人口规模的数据融合为一个基因数据文件以便用于测试基因型压缩器时，文件中所含记录的数量会产生超出正常规律的增长。比如，经过实验统计，在原本的千人基因组计划数据中，在选取的数据段，记录数量随人口数量增长的趋势为

$$r = 516.1 \times n^{0.578} \tag{2.1}$$

由公式(2.1)推导，当人口增长到 5000 时，应有约 7.1 万条记录，人口增长至 20000 时，应有约 15.8 万条记录，然而，当通过调整变异模拟工具的参数，使得分为 4 次产生的，每次人口数量为 5000 的基因数据中分别含有 69676、70194、70474、70350 条记录，符合增长规律，且略低于依照增长趋势所推算的 7.1 万条记录时，在将这 4 个 5000 人口的基因数据文件融合成一个人口数量为 20000 的基因数据文件时，发现文件中包含约 17.7 万条记录，反而超过了记录数量的增长规律。而实际上，随着统计的人口数量的增加，记录数量的增加速度应该和从千人基因组计划数据中统计得到的增加速度一致甚至略低，考虑到同一地区人类基因数据间的相似性，故而可见通过变异模拟工具先分批产生再融合成一个数据文件的方式得到的记录数量是过高的。因此，在用于测试基因型压缩器的查询速度时，对于固定的基因位点区间，用查询的总时间去衡量它适用于大规模人类基因数据的能力就不太合适了。所以在本次毕业设计中，使用基因型压缩器在查询一个较大范围（一百万个基因位置）内的记录时，查询每条记录所花费的平均时间作为衡量其查询速度的标准，相当于把查询时间对于所查询的记录数目做归一化。由此，测量结果便只受到人口数量增长的影响。

由于用于测试的数据集规模较小，基因型压缩器的查询时间很短，所以即使是较小的误差也可能引起参数选择的不准确。为了保证能够尽量科学的选择模拟基因数据的参数，在测量查询基因数据所花时间的时候需要确保没有其余的进程跑在同一个处理器线程上。为此，在本次毕业设计中所进行的所有测量时间的实验中，均使得计算机在开机时便保留一个处理器单元，共两个线程，第 6 和第 7 号线程，不被其他应用所使用，并在进行查询时，指定基因型压缩器只使用第 6 号线程来进行查询。当然，在限定基因型压缩器只能使用指定线程进行查询之前，已经在允许其使用任意线程的情况下，确认了其查询命令不支持多线程运行，所以限定其使用的线程不会减慢程序的查询速度。在确定模拟基因数据时所用参数的实验中用于测量基因型压缩器查询时间的指令是 Linux 系统的 time 指令，选择获取的用户(user)与系统(sys)时间之和作为确定模拟基因数据参数的依据。选择使用用户与系统时间之和的原因是，尽管真实(real)时间是程序运行的实际时间，也就是影响程序应用效果的时间，但是考虑到校验数据集的规模较小，查询时间本就很小，而真实时间在重复 20 次相同的查询指令时有高达的 4.72%标准差，而用户与系统时间之和的标准差只有 0.684%，故而选取实验结果较为稳定的用户与系统时间之和作为衡量的标准。通过多次实验，并结合考虑实验设备的内存大小，最后选定产生模拟基因数据的参数为：规模=13，繁衍代数=4，最终人口数=10000。通过将实验产生的一万人口的数据分割成 4 份 2500 人的数据，获得了和千人基因组数据规模相似的数据，可以看到对于基因型压缩器查询时间测试的结果如表 2.1。

表 2.1 用于验证模拟基因数据时所使用参数的数据

实验数据集	样本数量（人）	查询一百万个位置上每条记录的平均时间（毫秒）
千人基因组计划第二十二条染色体	2504	0.00606
模拟基因数据	2500	0.00616

由表 2.1 中的实验结果可以看到，对于模拟出的基因数据和真实的千人基因组数据，基因型压缩器有着极其相似的对于每条记录的查询时间，故而可以认为这组参数是合适的。

2.2.3 基因数据模拟方案

通过用上述实验测定的参数，每次模拟 1 万人口的基因数据，总共用 100 次模拟出了 100 万人口的基因数据。变异模拟工具在模拟基因数据时，为了达到模拟数据的可复现性，允许指定一个种子，只要两次模拟的所有参数都一样，种子也一样，便可以重复出完全相同的模拟。实验中分别使用了 1…100 作为模拟的种子。模拟时，采取先产生完 100 个小规模数据文件，再融合成不同规模数据文件的方式来进行，最终产生了人口规模为从 1 变化至 50 万，一共 13 个大规模的基因数据文件。实验中最终没有将所有人口融合至 50 万以上，是因为在利用 50 万人口的文件进行基因型模拟器的压缩时，压缩程序便已经几乎占用了电脑全部的 8GB 内存加 8GB 交换空间，所以可知利用现有设备无法再进行更大规模的数据压缩，故而没有继续完成更大规模的融合。

2.3 小结

通过尽量精确地控制实验的方式，验证了模拟的基因数据满足对实验的要求，并选定了用于模拟基因数据的参数，产生了足够大规模的基因数据文件以便用于后面的查询速度测试。

3 对基因型压缩器的测试与分析

3.1 对于查询速度的测试

基因型压缩器支持多种基因数据的查询方式，包括对一条记录进行查询，对一段位置区间内的记录进行查询，也可以直接解压整个文件，以及对于单个个体的基因数据进行查询等。在实验中，对基因型压缩器解压整个基因数据文件（包含从位置 16050075 到位置 18000000 共约二百万个位置）、查询连续一百万个位置（从位置 16050075 到位置 17050075）上的基因记录的时间进行了测试。通过将上述两种测试结果分别除以整个基因数据文件的记录数、以及相应位置区间上的记录数，来得到查询每条记录所用的平均时间。在查询时，将得到的数据以 VCF 文件的二进制形式输出，并且不对数据进行任何压缩，同时将解压出的数据直接删除，不保留在硬盘，以消除压缩算法和硬盘读写速度对于实验的影响。

由于通过这两种查询方式最终得到的对于每条记录的平均查询时间几乎相同，又考虑到在将记录数量非常庞大的数据，如千人基因组计划中整条染色体上的数据压缩后，通常会只解压出其中需要的一段进行分析，本文中只列出查询整个基因文件中连续一百万个位置上基因记录的实验结果，如表 3.1。通过将表 3.1 中的数据可视化，并对数据使用方程式进行拟合，可以画出平均每条记录的查询时间随人口数量增长的关系曲线，结果如图 3.1 所示。从图 3.1 中可以看到，拟合出的曲线与真实数据几乎重合，拟合曲线对应方程如公式(3.1)。

$$t = 1.322 \times 10^{-2} \times n^{1.222} + 2.571 \times 10^{-2} \tag{3.1}$$

公式(3.1)中，n为人口数量，单位为万人，t为平均查询一条记录所需的时间，单位为毫秒。从公式中我们可以看到，查询一条记录的时间随这条记录中所含人口数量的增长呈现出略大于线性的增长趋势，这表明基因型压缩器已经具有了较好的适用于大规模基因数据的能力。然而，美中不足的是，尽管略大于线性的时间增长已经十分出色，但当人口数量继续增加时，还是会较大的增加查询时间，所以下面的工作便是尝试对于查询速度提升。

3.2 确定最消耗时间的索引步骤

只有对于整个查询过程的时间统计，并不足以分析出如何提升基因型压缩器的查询

表 3.1 对从位置 16050075 到位置 17050075 这一区间上的基因记录进行查询所花的时间

人口规模（万）	记录数量	总查询时间（秒）	平均每条记录的查询时间（毫秒）
1	18844	0.7311	0.0388
2	25670	1.412	0.05501
3	31273	2.243	0.07172
5	40851	5.081	0.1244
10	64016	15.96	0.2493
15	86097	33.36	0.3875
20	107554	57.71	0.5366
25	128427	89.07	0.6935
30	148863	133.6	0.8975
35	168563	174.4	1.035
40	187821	227.6	1.212
45	206522	287.6	1.393
50	224651	363.4	1.618

注：平均每条记录的查询时间是由总查询时间除以记录数量得到的，查询时总是查询全部样本的基因型。

速度，所以在接下来的实验中，我们将整个查询过程分解，并测量统计了各部分的运行时间，以查看应该着重提高哪部分程序对应的算法效率或运行效率。

在基因型压缩器所对应的解压程序中，首先在不同位置把程序分成了 5 段，并测量了每段的运行时间，测量结果经过处理后，如图 3.2 所示。通过实验结果，可以看出随着人口数量的增长，第三段程序占用了大多数的运行时间。再仿照上述步骤，将第三段中的程序继续分解，最终确定了是程序中的一个名为 decode_perm_rev 的函数占用了最多的运行时间。单独测量这个函数的运行时间，可以看到它占整个查询程序运行时间的比例如图 3.3 变化。当人口数超过 20 万时，这个函数所占用的运行时间达到了查询整条记录的时间的 75%，并稳定在这一数值。所以，下面将以这个函数作为切入点来尝试进行对于基因型压缩器查询速度的提高。

3.3 小结

通过上述的实验，最终将研究的重点确定在了 decode_perm_rev 这一函数上。下面将通过多种方法来尝试提高对于压缩后的基因数据的查询速度。

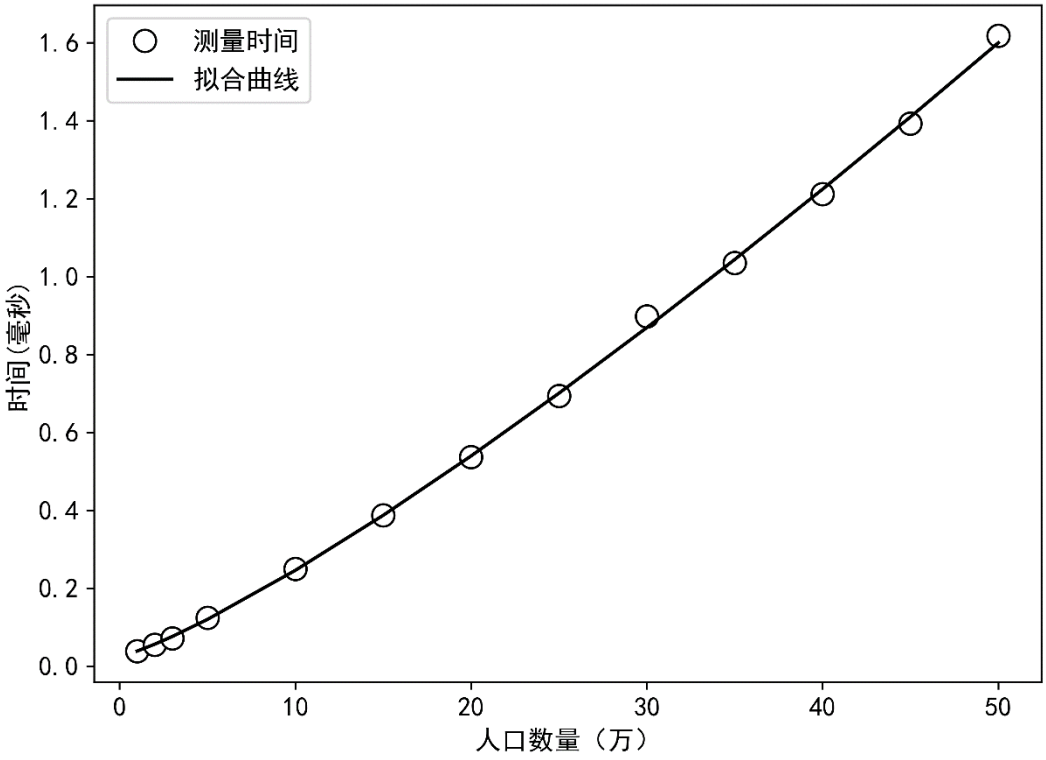


图 3.1 基因型压缩器查询每条记录的平均时间随记录中所含人口数量变化的关系曲线

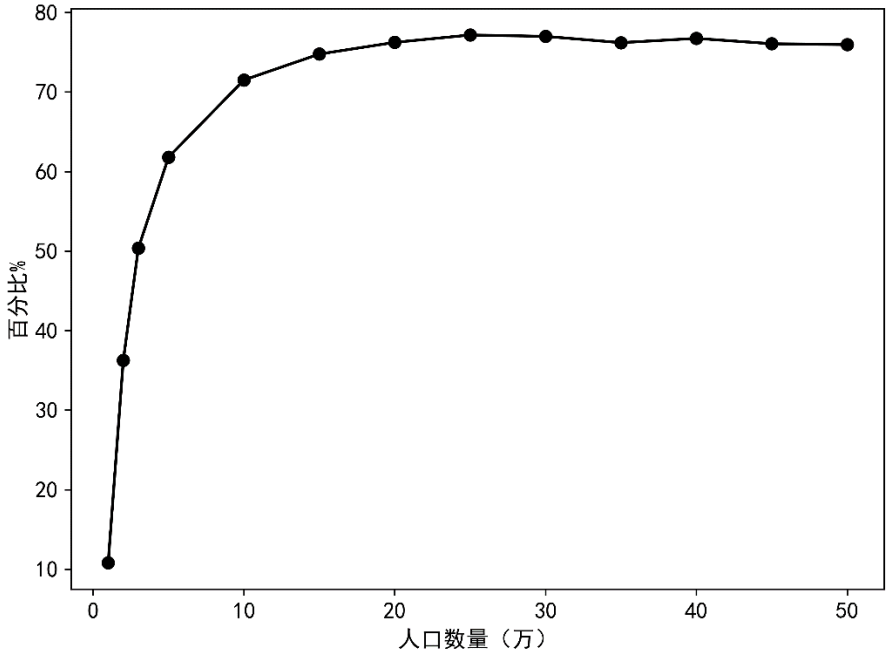
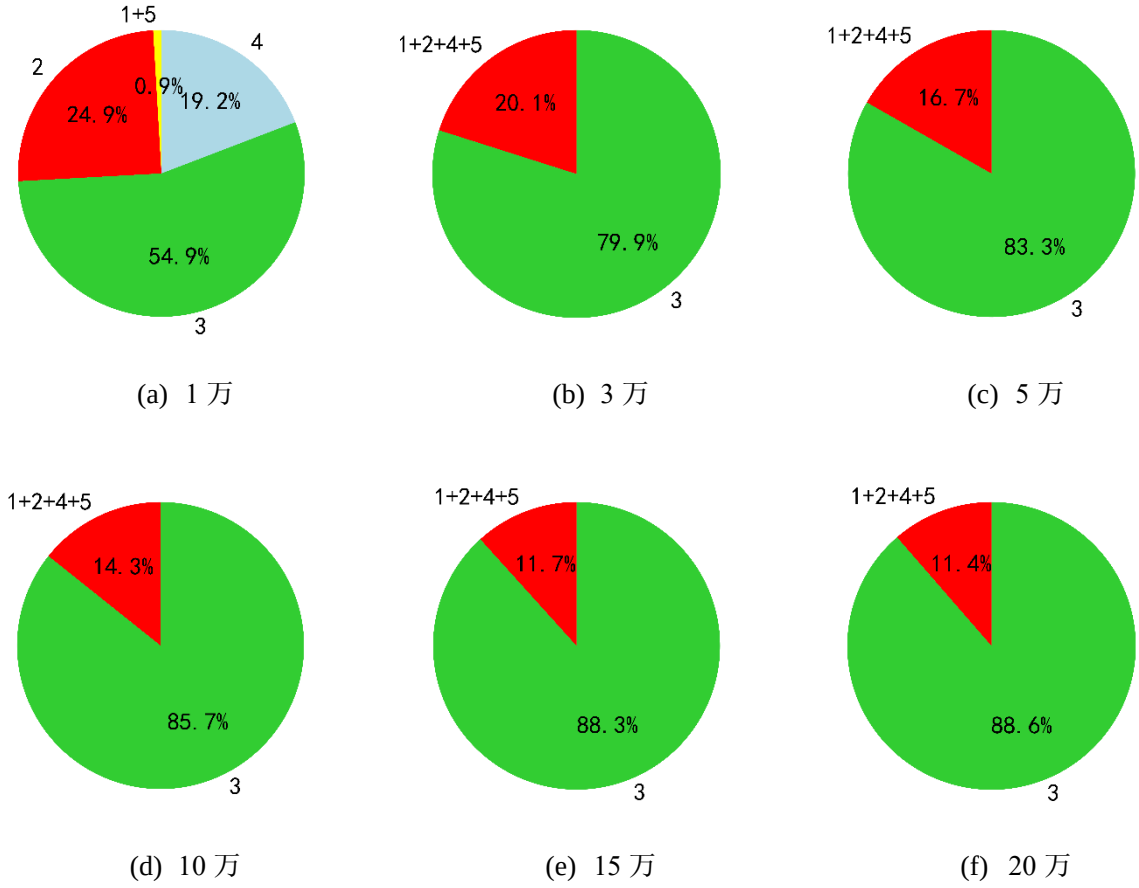


图 3.3 函数 decode_perm_rev 运行时间占整个查询程序运行时间的比例随人口数量的变化



(a) 1 万 (b) 3 万 (c) 5 万

(d) 10 万 (e) 15 万 (f) 20 万

图 3.2 各段程序占查询每条记录的平均时间的百分比随人口数量增长的变化

4 对于基因型压缩器的改进尝试

4.1 算法分析

想要对 decode_perm_rev 函数进行改进，就需要先细致的分析整个压缩与解压算法，分析出这个函数在整个算法中的作用，再尝试进行改进。

4.1.1 压缩算法

1、预处理

基因型压缩器首先对输入的 VCF 文件进行预处理，将每条拥有多种非参考基因的记录，分解成只拥有一种非参考基因的多条记录。在分解前，对于一条拥有多种非参考基因的记录，需要用不同的数字来表示，也就意味着需要用不同数量的比特来表示，比如表示第 4 种参考基因的话，就需要用 100，共 3 比特。而分解后，每条记录上只有一种参考基因，便可以用 2 比特的空间来完整的表示每条记录上基因型的信息：00 表示参考基因，01 表示这条记录上的非参考基因，11 表示另一条非参考基因，10 表示未知基因。然后将每个 2 比特的数的高低位分开，便可将原本由字符矩阵表示的 VCF 文件转化为由 0、1 矩阵表示的基因数据。这里将矩阵的第 1 行编号为 0，作为偶数行计算，其余行按照 1,2,3...依次编号。矩阵中偶数行为原来 2 位二进制数的高位，奇数行为低位。将基因型数据转化为 0、1 矩阵后，沿纵向也就是记录条数增长的方向分块，分块大小可在压缩时由参数指定，算法的默认参数为 3584 条分解后的记录为一块，也就是将矩阵的 7168 行分为一块。分块后，先对每块分别进行压缩，再将压缩后得到的数据整理成最终的压缩文件。图 4.1 是一个对于预处理步骤的简单示意图，图中 S1 和 S2 表示样本 1 和样本 2。图 4.1(d)中，按照 2 条分解后的记录为一块，沿虚线将 0、1 矩阵分为了 2 块。

2、分块处理

分块后，在对每块矩阵进行处理时，使用启发式最近邻居算法(Nearest Neighbor Heuristic) **Error! Reference source not found.**来对每一列进行重排序，使得排序后从第一列到最后一列的总汉明距离(Hamming Distance) [22]尽可能的小。启发式最近邻居算法是一种可以用于解决旅行推销员问题(Travelling Salesman Problem, TSP) [22]的算法。启发式最近邻居算法可以简单的描述如下：以矩阵第一列为起点，选择剩余列中与其汉明距离最近的一列，

放在第二列的

REF	ALT	S1	S2
A	C	0|1	.|.
G	A,T,CT	2|3	0|1

(a) 简化的 VCF 文件记录

REF	ALT	S1	S2
A	C	0|1	2|2
G	A	3|3	0|1
G	T	1|3	0|3
G	CT	3|1	0|3

(b) 分解后的记录

REF	ALT	S1	S2
A	C	00|01	10|10
G	A	11|11	00|01
G	T	01|11	00|11
G	CT	11|01	00|11

(c) 以二进制数表示的记录

0	0	1	1
0	1	0	0
1	1	0	0
1	1	0	1
0	1	0	1
1	1	0	1
1	0	0	1
1	1	0	1

(d) 由样本基因型转化成的 0、1 矩阵

图 4.1 基因型压缩器对 VCF 文件进行预处理的示意图

0	0	1	1
0	1	0	0
1	1	0	0
1	1	0	1
0	1	2	3

(a) 排序前

	0	0	1	1
	0	1	0	0
	1	1	0	0
	1	1	1	0
P^i	0	1	3	2

(b) 排序后

图 4.2 启发式最近邻居算法的示例

位置上，然后再从剩下的列中选择与第二列汉明距离最近的作为第三列，以此往复，直到最后一列。这种算法并不是这一类问题的最优解，但是这类问题在实际工作中往往无法得到最优解，尤其是对于这种问题规模在万以上的情况，故而采取这种启发式的算法来求解。图 4.2 是使用这种算法对图 4.1(d)中分出的第一块矩阵进行排序的示意图。矩

阵中深色行为每一列对应的序号。图中排序后，使用一个数组P^i来存储每列排序后数据原来的位置。对于这样将矩阵进行重排序的原因，将在下文进行讨论。

在将矩阵进行排序后，便可以开始一行一行的处理矩阵。首先，如果遇到全部为零的行，便将其序号记录下来，之后查询这些行时便可直接输出全零数据，我们称这样的行为全零行；如果遇到从来没有编码过的行，便对其进行编码，编码的方式会在后文介绍，我们称这样的行为新行；如果遇到和之前编码过的行相同的行，便将其作为复制行记录下来，具体为记录这一行的序号以及前面与其相同的新行的序号，我们称这样的行为复制行。如此，便可完成对于一个矩阵块的处理。

对于每一个新行，对其按字节进行编码，将长于两个或两个以上为 0 或 255 的字节（全 0 或全 1）直接编码为连续 0 和连续 1，记录其位置和长度。这里使用了游程编码的思想来实现数据的压缩。对剩余的字节，首先向上在编码过的新行中寻找有没有长度超过 5 个字节的相同的部分，如果有，则记录与其包含相同连续字节的行的序号，并记录下相同部分的长度。这一步模仿了 Lempel-Ziv 系列压缩算法，只不过由向前寻找相同字符串变为了向上寻找相同字节。可以看到，在处理一行数据的过程中，基因型压缩器在利用连续 0 和连续 1 进行横向编码的同时，还在纵向进行编码，进一步提高了压缩比率。从上述的编码步骤可看出之前对于矩阵进行重排序以降低相邻列间总汉明距离的原因：利用人类基因数据的相似性，通过重排序，增加了寻找到连续 0、连续 1、和向上寻找到连续相同字节的可能性，从而提高压缩率。对于无法向上寻找到相同连续字节的部分，则将其逐字进行编码。最终所有新行被编码成一个一个的元组，一行由一个或多个元组组成。在编码的过程中还有一些其他细节，在此不做过多叙述。

3、矩阵块融合

当把整个矩阵块编码后，便可以将所有的矩阵块融合到一起，将所有存储有单倍型基因型原始位置的数组P^i融合成一个数组P, 把所有为空和为复制的行的序号用 4 个数组$E_{even}, E_{odd}, C_{even}, C_{odd}$存储，这 4 个数组的长度与预处理时分解后的记录数相同，下标为$even$的数组对应 0、1 矩阵的偶数行，下标为odd的数组对应 0、1 矩阵的奇数行，E, C数组中相应的位置为 1 分别表示这一行为全零行或是复制行。另外用一个单独的数组C_{origin}来存储复制行所复制的是哪个新行，将新行的序号存入数组。新行的序号在融合时会根据据其所在的矩阵块而做出调整。所有新行调整后的序号还会以能够使用的最

少的比特数存储到一个数组V_{id}中。然后将之前分块处理时得到的用于表示新行的元组进行霍夫曼编码。每个元组中，第一个要被编码的部分是标签，用来表明其是哪种元组，并在解码时指定对接下来的比特流应该用哪个对应的编码集进行解码。不同种类的元组中有不同的部分需要分组编码，在此不一一写明。编码后的二进制数据被存放在一个数组U中。同时，编码后，每个被编码的行在U中的起始位置，被单独存储在一个U_{pos}矩阵中，存储时分段使用了增量编码[24]。

经由上述的步骤，便完成了对于基因数据的压缩。上文只讨论了压缩过程中的主体部分，尤其是对查询速度造成较大影响的部分，还有其他一些细节，暂且不做讨论。

4.1.2 查询算法

基因型压缩器的查询算法主要有两种方式构成，按照记录查询和按照个体查询。本文主要讨论的是按照记录查询的方法。

在按照记录查询时，首先指定要查询的位置，然后基因型压缩器从压缩时只记录了原始记录位置和分解后记录位置的 BCF 文件中，读取需要解压的矩阵范围。之后，便开始按照压缩时的分块，一块一块的解压矩阵。解压时，依据行的编号，先查询E_{even}, E_{odd}数组中对应位置是否为 1。若为 1，则这一行为全零行；若不为空，再检查C_{even}, C_{odd}中对应位置是否为 1，若为 1，则这一行为复制行，通过计算数组C_{odd}, C_{even}中，截止至相应位置的秩，来寻找其复制的是哪行。对于一个由 0、1 组成的数组，其秩计算方法的定义见公式(4.1)。

$$rank(C[I]) = \sum_{i=0}^{I-1} C[i] \tag{4.1}$$

由公式(4.1)可知，一个数组I位置处的秩，便等于这个数组从起始位置到$I-1$位置处所有为 1 的元素的数量。对于储在I位置处的复制行，其所对应的新行在数组C_{origin}中的位置J可由公式(4.2)求得。

$$J = rank(C_{even}[I]) + (C_{odd}[I]) \tag{4.2}$$

$C_{origin}[J]$便是其所复制的新行的序号。对于在$E_{even}, E_{odd}, C_{even}, C_{odd}$中对应位置都不为 1 的行，说明这既不是全零行，也不是复制行，而是一个新行，其序号在数组V_{id}中的位置可由公式(4.3)计算。

$$J = I - rank(E_{even}[I]) - rank(E_{odd}[I]) - rank(C_{even}[I]) - rank(C_{odd}[I]) \tag{4.3}$$

之后便可通过$V_{id}[J]$来得到这一新行的序号。随后，根据新行的序号，从之前存储的U_{pos}数组中找到此行编码后在数组U中的存储位置，接下来便可以从这个位置开始进行霍夫曼解码。解码时，首先使用标签码集进行霍夫曼解码，得到标签后，根据标签选择下一次解码使用的码集。以对连续 1 字节的解码为例，当解码得到标签为f_{one_run}时，下一次解码使用连续为 1 的字节的长度这个码集，解码得到长度后，对于这个元组的解码就完成了，然后开始对于下一个标签的解码。同时，解码时使用一个变量来记录在这一行中当前解码到的位置，因为在遇到向上相同的字节串时，需要向上回溯到另一行，并利用记录下的位置信息，尽可能地避免重新解码，尽快跳至当前解码的位置，直接解码出所需要的字节。

当把一个矩阵块内的 0、1 矩阵全都解码完成后，便需要按照之前记录的数组P中的信息，来恢复每一列的位置。恢复位置后，便可先恢复成分解后的记录形式，再将被分解的记录融合成写有多个非参考基因的原始 VCF 文件记录格式。

4.2 程序优化

前文所提的占用查询时间最长的函数 decode_perm_rev 其作用是将之前使用启发式最近邻居算法排序的列恢复到原来的位置，具体来说，由于查询时是一条一条记录进行查询的，每次执行 decode_perm_rev 时，会将霍夫曼解码得到的 0、1 矩阵中的一行恢复至使用启发式最近邻居算法排序前的顺序。下面首先分析其时间复杂度和作用，然后再进行优化。

4.2.1 时间复杂度

恢复原始顺序的具体算法是：首先创建一个空的输出数组，相当于一个全为 0 的比特向量。然后以字节为单位遍历解码出的 0、1 矩阵中的一行，遇到值不为 0 的字节，就再遍历这个字节的每一位比特。将每一位比特所对应的在P_i矩阵中原始的位置取出，如果这个比特为 1，就将用于存放输出数组的空数组中对应原始位置的比特置 1。这样一直处理完整个矩阵中的一行。可见，如果设一行中的元素数量为n，则这一函数的时间复杂度为约为$O(n)$,并且，由于矩阵中每一列的位置都有可能发生改变，所以在恢复时需要遍历其中的全部元素，这个复杂度是很难降低的。

4.2.2 函数作用分析

对于基因型压缩器来说，这个函数的存在是必要的。因为在压缩时，将一个 0、1 矩阵分成了不同的区块，并且每个区块分别进行重排序，所以在查询时，就必须把每个矩阵的顺序恢复一致，以保证能合理地恢复一个人在查询范围内全部的基因型。而压缩时的重排，通过使相邻列之间的汉明距离尽可能地小，来提高相邻列间的相似性。通过分析压缩算法，可以发现重排序对于全零行以及复制行的压缩并没有影响，其实际上影响的是在将新行编码为元组时，可以编码成的连续 0、连续 1 以及向上可以寻找到相同连续字节的概率和长度。故而可见重排序会提高压缩的效率，所以，对重排序后的恢复，就是不可避免的了。

4.2.3 函数优化

上文分析了函数的必要性，以及函数在时间复杂度上难以降低的原因。但由于函数执行时间占整个查询时间的比重较大，为了加快查询的速度，需要尽可能地缩短函数的运行时间。由上文对于函数时间复杂度的分析可知，函数在执行时并不是简单的将每一个解码出为 1 的比特放置回原来的位置，其原因是在 C++中编程时，难以对每个比特进行单独赋值，故而函数书写时选择了通过对于一个字节进行赋值的方式来实现对于一个比特的赋值。而这种方式则会向函数中多引入三次运算，假设需要将一个比特向量中的位置j（位置编号从 0 开始）置 1，则这三次运算如公式(4.4)、（4.5）、（4.6）所示。

$$byte = int\left(\frac{j}{8}\right) \tag{4.4}$$

$$bit = j\%8 \tag{4.5}$$

$$\Delta = 128 \gg bit \tag{4.6}$$

经如上 3 步运算后，将比特向量中的第$byte$个字节加上Δ，就实现了将位置j置 1 的操作。在原程序中，函数会在每次寻找到包含非零位的字节时，就对字节中的每一位进行上述的运算。然而，对于一个 8 位的字节，若其中只有 1 位为 1，也就是说只需要对输出数组中的 1 位置 1 的话，那其中的 7 次运算便都没有意义。更进一步，查询程序对于每一个矩阵块中的内容是按行处理的，然而一个矩阵块内的所有行都共用一个恢复重排的矩阵P^i，所以对于一个矩阵内每一个需要处理的行，理论上只需要对其中的每个位置进行一遍上述的三步运算，便可获得对于每个输出位置赋值时所需要的变量。

调整后的查询程序结构为，在处理一个矩阵块内的每一行记录前，就先计算出将每个位置x的原始位置j置为 1 所需要的变量，并存储在数组$revPermByte$和$revPermBit$中。其中数组$revPermByte$存储的是每个位置x的原始位置j在输出数组中所在的字节，相当于公式(4.4)中的$byte$，数组$revPermBit$中存储的是，将输出数组中位置j置 1 时所需的与位置j所在字节进行相加运算的字节，也就是公式(4.6)中的Δ。恢复矩阵顺序的时候，如果解码出的比特向量中的位置x为 1，需要将输出中x对应的原始位置j置 1，就可以直接利用x索引两个数组，利用公式(4.7)

$$decomp\_data\left[revPermByte[x]\right] += revPermBit[x] \tag{4.7}$$

来完成赋值。

通过使用修改后的程序来测试查询时间，将数据记录到表 4.1，可以观察到其查询速度有了一定的提升。由表 4.1 中数据可知，当人口数上升到一定数量时，所作优化对查询时间的提升稳定在 10%左右。

表 4.1 程序优化后对查询速度的提升效果

人口数量（万）	优化前查询时间（毫秒）	优化后查询时间（毫秒）	查询时间变化量（毫秒）	查询时间变化百分比
1	0.03572	0.03695	-0.001221	-3.42%
2	0.05033	0.04807	0.002259	4.49%
3	0.06862	0.06578	0.002846	4.15%
5	0.1154	0.1046	0.01077	9.34%
10	0.2473	0.2271	0.02015	8.15%
15	0.3835	0.3452	0.03833	9.99%
20	0.5338	0.4784	0.05541	10.38%
25	0.6992	0.6123	0.0869	12.43%
30	0.917	0.8001	0.1169	12.75%
35	1.047	0.9385	0.1086	10.37%
40	1.206	1.106	0.1001	8.3%
45	1.402	1.283	0.1191	8.49%
50	1.612	1.444	0.1683	10.44%

注: 表中显示时间为，用优化前和优化后的程序从压缩后的基因数据中，查询一百万个位置上的记录时，平均每条记录所花费的时间。

若将优化前的查询时间和优化后的查询时间作出曲线，则可以较为直观地看出提升效果，如图 4.3。在修改基因型压缩器的查询程序后，已将查询出的基因数据与修改前的查询结果进行了比对，二者相符，证明对于程序的优化没有影响程序查询时的正确性。查询程序优化后的部分见附录 A。

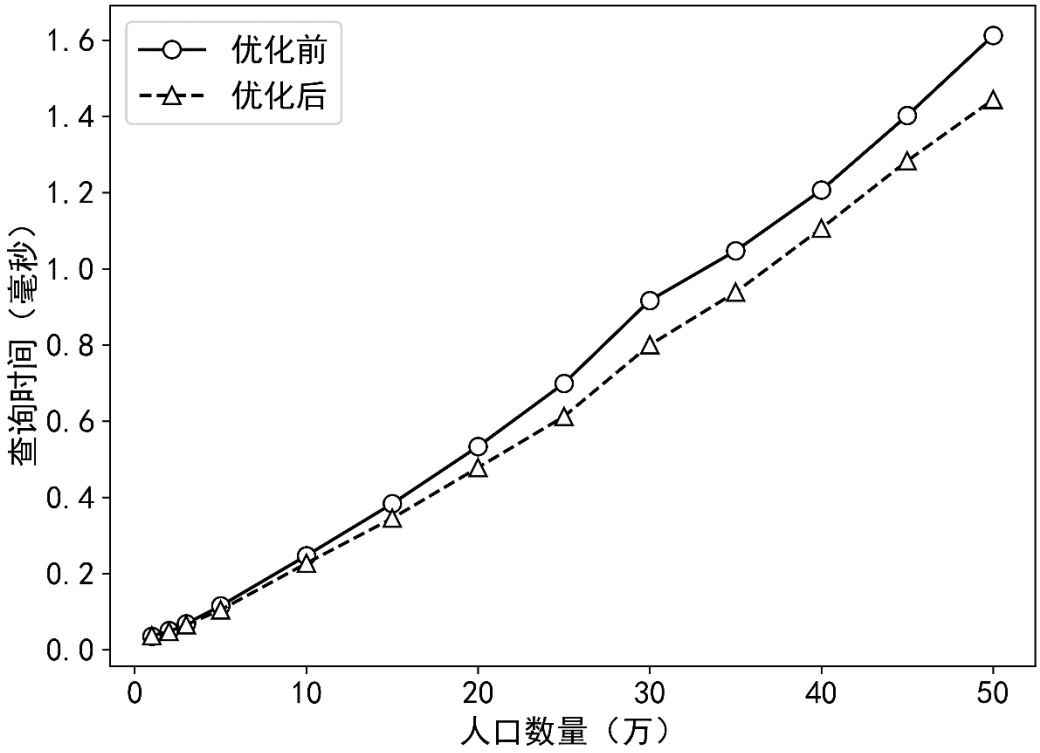


图 4.3 程序优化后查询速度的提升效果

4.3 一种可移动列数受限条件下的重排序算法

如上文分析，查询时间的大部分都花费在了恢复矩阵的原始排序上。基因型压缩器通过在压缩时对矩阵块中的每一列使用启发式最近邻居算法排序，以减小整个矩阵块的相邻汉明距离总和，使得相似的列可以互相聚拢，以便在后续的压缩中寻找到更多的连续 0、连续 1 和纵向的连续相同字节。这意味着在查询时需要将矩阵块中的每一列都遍历一遍，以恢复其原始位置。若能通过一种算法，在只挪动少部分列位置的约束条件下，使得矩阵块整体的汉明距离总和有较大的下降，便可能在不过多损失压缩率的前提下，提升查询速度。

4.3.1 受限重排序算法的可能性简单示例

图 4.4 是一个通过移动较少的列来降低矩阵相邻列间总汉明距离的示例。矩阵中深色行为每一列对应的序号。从图可以看出，通过启发式最近邻居算法排序的矩阵，其除起始列外，其余所有列的位置都发生了变化，从第一列到最后一列的总汉明距离从 24 降

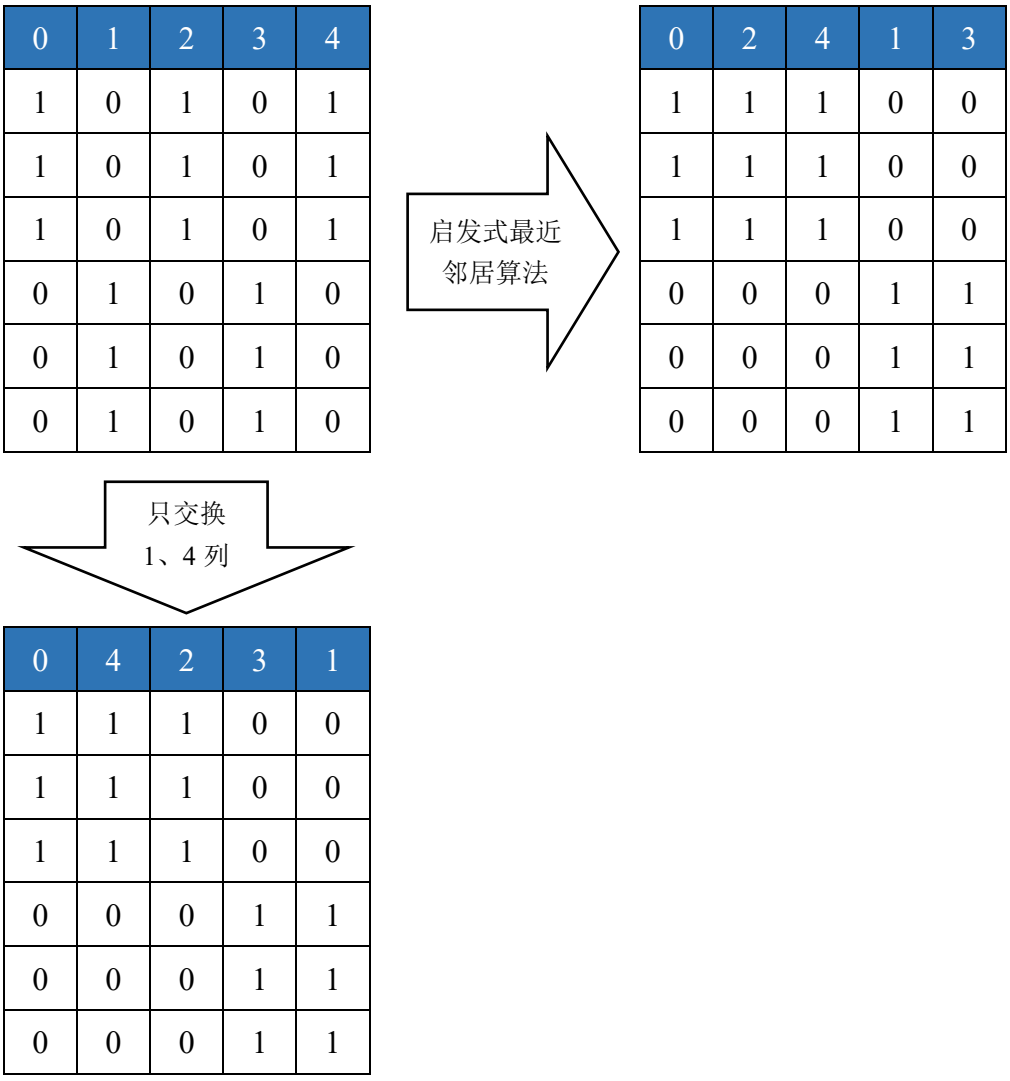


图 4.4 对可移动列数受限制的情况下通过重排序以减小总汉明距离的算法的可行性示例

到了 6。而通过只交换 1、4 列的位置，同样可以使从第一列到最后一列的总汉明距离从 24 降到 6。由此可见，对于某些类型的 0、1 矩阵，只交换少量的列，是有可能达到降低总汉明距离的效果的。需要注意的是，上述例子只是一个用于证明约束条件下解法存在可能性的简单特殊情况，并不保证任何情况下限制可移动的列数后，依然能有和任意重排序时效果相似的解。而通过后续的实验，也确实发现限制可移动的列数后，难以达

到和任意重排序时相同的降低总汉明距离的效果。

4.3.2 算法设计

由于缩短总旅行距离这类问题通常是以访问不同地点为背景的，如旅行推销员问题，很少遇到必须在特定次序访问特定地点这种限制，故而关于这类算法的资料较少，在查找文献的过程中也确实没有找到类似的资料。因此，我们自己设计了一种算法，来试图寻找这种问题的较优解。如果想要寻找这类问题的最优解，规定矩阵一共有N列，最多可以挪动其中的L列，可以较直观地推断出需要计算的可能性数量为:

$$T = \frac{N!}{(N-L)!} \tag{4.8}$$

即从N列中选出L列的全排列。这样庞大的可能性数量使得在实际应用中很难寻找到这类问题的最优解，因此设计算法时，同样采用了启发式的思想来搜索较优解。

算法一开始，会从所有的列中寻找出和其两边邻居汉明距离之和最大的列，选为想要和其他列的位置交换的列，简称其为不和列。然后在剩余的列中寻找一个最佳的交换方案。由于在排序时有了对于移动列数的限制，故而需要在判定排序方案的好坏时，考虑进这次交换所造成的不在自己原本位置的列数。这个算法通过收益和成本间的比值，也就是收益率来衡量某一种交换方案是否理想。收益定义为：在执行这次交换后，从第一列到最后一列总相邻汉明距离的减小量。成本定义为：在执行这次交换后，不处在原始位置的列的增加量。只有当收益率高于某一阈值，这次交换才可能发生。每次交换发生后，重新选择与两边邻居汉明距离之和最大列作为不和列，然后继续和上述过程相似的搜索。如果对于某一个不和列，无法找到一个合适的交换方案，则放弃这一列，而选择与两边邻居汉明距离之和第二大的列作为不和列，继续搜索，以此往复。

以上就是算法大致的搜索思路。在具体实现时，对搜索方案进行了一定规划，以尽可能地加快搜索的速度。首先，对所有列的位置按照其和两边邻居汉明距离之和的大小从左至右，由大到小排序，搜索时，从左向右搜索，由于两个位置上的列交换时具有相互性，所以每个位置只向右搜索，以避免重复的计算和比较。如果第一个位置无法向后搜索到一个合适的交换方案，那么第二个位置搜索时便不再考虑第一个位置。同样，为了加快搜索的速度，降低需要进行的计算总量，每一次做过的计算的信息会被尽可能地保留下来。目前算法中使用的方法是，对每一个位置的所有交换方案定义 3 种状态：激活，休眠，抛弃。对每一个被选为不和列的位置，在搜索其合适的交换方案时，会只在

处于激活状态的交换方案中搜索。对于一种方案，如果其收益小于零，则直接将其抛弃；如果收益等于零，成本大于等于零，这意味着这次交换毫无用处，则也将其抛弃；如果收益和成本均大于零，则计算其收益率，如果收益率小于等于当前阈值，则将其根据其收益率将这中交换方案置为不同深度等级的休眠状态。对于其他情况，包括成本小于零且收益不为负、收益率高于当前阈值，这些属于交换可以发生的情况，则保留这种方案的激活状态，并记录下需要做交换的位置。

设立休眠状态以及不同休眠等级，其目的其实就是制作一个数据库，将之前计算过的不同的交换方案的收益率记录下来。因为在进行搜索的过程中，当经过一定次数的搜索失败后，算法会降低收益率的阈值，从而允许更多交换方案的发生。休眠等级的定义是和不同的阈值相对应的，每一次阈值降低后，便将对应休眠等级的交换方案激活。这里需要说明，目前这种休眠等级的设定实际上是一种对于曾经计算出的收益率的模糊，会降低存储的总信息量。最理想的存储方式，应该是将每一种交换方案的收益率都储存起来，当阈值发生变化后，直接根据之前计算的结果来选择合适的交换方案。但是精确地存储所有的交换方案会使得算法占用的内存增大，并且可能需要不同的实现方式，故而现在没有采用这种存储方式。对于休眠等级，等级划分越细，也就越接近精确存储每种交换方案收益率的效果，每次降低阈值后，重新激活的交换方案也更少，这意味着需要搜索的交换方案也更少，因而可以起到加速搜索的效果。

算法中所存在的三种状态皆是动态的，由于每一次交换都会改变 2 个位置上的列，所以，在每次交换后，交换位置本身，以及这两个位置的邻居所涉及的所有交换方案都会被重新激活，不管他们之前处于休眠或是抛弃状态。

算法中，存在多个可以调整的参数，通常执行算法时会设定休眠阈值为初始阈值，搜索深度为初始深度，设定阈值衰减量，深度增长量和最大深度。算法的执行流程如算法 1 所示。

4.3.3 算法效果

通过在千人基因组计划，第 22 号染色体上的位置 16050075-16500000 这一段区间上进行实验，对算法的效果进行了检验。检验算法时所使用的参数为，初始阈值=100，阈值衰减量=5，初始深度=200，深度增加量=100。实验数据被转化为了一个有 6139 行，5008 列的矩阵，其原始状态下，从第一列到最后一列的汉明距离总和为 570689；通过

启发式最近邻居算法重排列后，从第一列到最后一列的汉明距离总和为 340203；通过上述限制可移动的列的数量的重排序算法，在只改变 1002 列的位置的情况下，将从第一

算法 1：可移动列数受限的重排序算法

循环 1:当移动的总列数小于限制：

将所有列根据与邻居的汉明距离和由大到小排列;

i = 1;

循环 2:取排序后的第 i 个位置为不和列：

循环 3:从第 i 个位置向右搜索交换方案：

若交换方案收益<0，抛弃方案，跳至循环 3 末尾；

否则，若交换方案成本<0，记录这个方案的交换位置，跳出循环 3；

否则，若交换方案收益==0，抛弃方案，跳至循环 3 末尾；

否则，若方案收益率>当前阈值，记录这个方案的交换位置，将当前阈值设为这个方案的收益率，跳至循环 3 末尾；

否则，若方案收益率>休眠阈值，跳至循环 3 末尾；

否则，根据方案收益率，将方案放入相应等级的休眠区；

如果有被标记的方案：

执行方案所对应的交换；

总移动列数 += 成本

将交换的位置和其邻居所涉及的所有交换方案激活；

当前阈值=休眠阈值

跳出循环 2；

如果没有被标记的方案：

i += 1;

如果 i >搜索深度：

如果搜索深度>最大深度：

调整参数以尽快结束搜索，跳出循环 2；

休眠阈值 -= 阈值衰减量；

搜索深度 += 深度增长量；

激活对应休眠等级内所有的交换方案；

跳出循环 2；

列到最后一列的总汉明距离降低到了 483039，对总汉明距离的减小量为启发式最近邻居算法的 38.5%，却只需移动约$\frac{1}{5}$的列的位置。这意味着在按照记录查询基因数据时，需要遍历以恢复其原始位置的列的数量为之前的$\frac{1}{5}$，根据恢复重排序的时间占全部查询时间的比重，将需要恢复位置的行的数量减少$\frac{4}{5}$，理论上可以很大程度上的加快对于基因数据的查询速度。对算法的具体实现见附录 B。

4.3.4 算法分析

通过将上述实验中，每一次交换后的总汉明距离记录下来，可以观察到总汉明距离在交换过程中随交换次数的变化趋势，如图 4.5 所示。

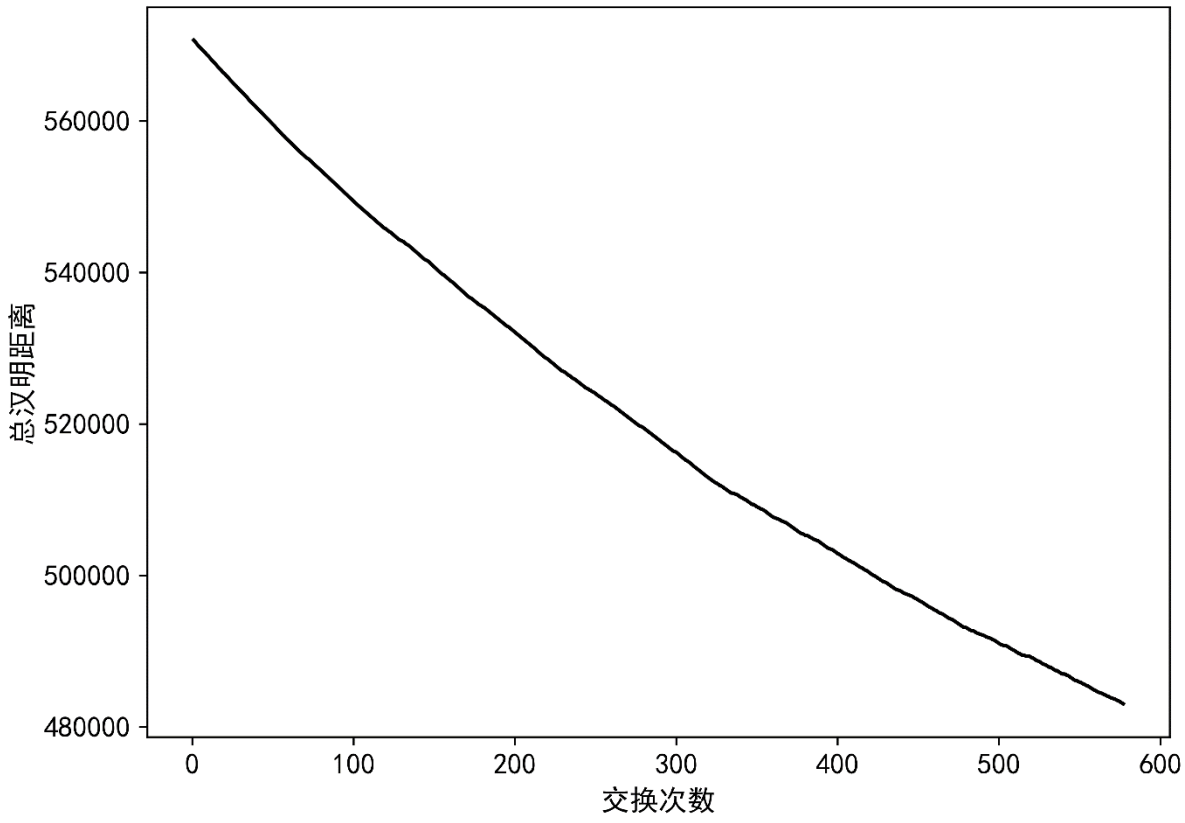


图 4.5 相邻列间总汉明距离随列的交换次数的变化

为了进一步分析算法，在实验中对总汉明距离随交换次数变化的曲线做了拟合，得到了公式(4.9)。

$$总汉明距离 = -908.8 \times s^{0.7284} + 574822 \tag{4.9}$$

从图中曲线以及公式(4.9)中s小于 1 的指数我们可以看出，随着交换次数的增加，总汉明距离的下降速度会越来越慢。

由之前的分析得出，使得矩阵所有列之间的总汉明距离减小，有利于寻找更多的连续 0 或连续 1 字节进行编码，也有利于通过纵向寻找相同的连续字节来进行编码。也就是说，通过受限制的重排序算法来减小矩阵相邻列间的总汉明距离，理论上虽然能带来查询速度的提升，但是也伴随着损失数据压缩率的风险。值得注意的是，在基因型压缩器压缩所得的文件中，用于恢复重排序的数组P占据了约 30.8%[15]的空间，通过减少数组P的大小，可以节省一部分空间，以用于平衡算法带来的压缩率的损失。

同时，可移动列数受限的重排序算法受到多个参数的控制，对于各个参数的选择方案，以及参数对于降低总汉明距离的效果、算法空间复杂度和时间复杂度的影响还有待进一步研究。

4.4 小结

这一章中，对于基因型压缩器的压缩和查询算法做了详细的分析，并优化了其查询过程的执行方式，取得了约 10%的查询速度提升。随后，为了解决恢复矩阵重排序所花费时间长的问题，设计了一种新的算法。算法通过搜索的方式，来寻找上述问题的较优解，并且取得了一定的成果。然而，考虑到这种问题应用场景的稀缺性，并没能找到足够多的参考资料或是前人经验来系统的研究这类问题的解决方式。在可移动的列数量受限的条件下，使得一个矩阵从第一列到最后一列的总汉明距离最小，这类问题在生活中是否还有其他的对应场景，这一点还值得探索。同样，对于这类问题的更优解法的挖掘也是值得期待的。对于实验中的基因矩阵，使用受限制的重排序算法进行排序后得到的结果，其总汉明距离要大于使用启发式最近邻居算法排序后得到的结果，而这种总汉明距离的提升对于压缩效率的影响也是需要进一步考虑的。

结论

本文先通过实验，选定了合适的模拟大规模人类基因数据的参数和方案，并生产了不同规模的人类基因数据集，最大的规模可达 50 万。随后利用模拟出的基因数据，对基因型压缩器(GenoType Compress)在不同人口规模的基因数据上的查询速度做了详细的测试，得到了其查询算法关于人口规模n的时间复杂度约为$O(n^{1.222})$,证明了其已经具有被应用于大规模人类基因数据压缩与查询工作的能力。同时，通过仔细的分析其压缩与查询算法，并逐层测量程序不同部分的运行时间，找到了能够进一步加快查询速度的突破口。利用对于其算法执行方式的优化，将其对于人口数量庞大的基因数据的查询速度提高了约 10%。从另一方面，为了减少矩阵中需要恢复位置的列的数量，本文提出了一种新的算法来在可移动的列数受限的情况下，尽可能地减小矩阵中相邻列间汉明距离的总和。通过在真实基因数据上的实验，验证了这种算法的可行性。对于实验数据集，算法通过移动 20%的列，取得了启发式最近邻居算法 38.5%的效果。

对于飞速增长的人类基因数据量，尽管基因型压缩器的表现已经足够出色，但是对于规模庞大的数据集，即使是$O(n)$的查询时间复杂度也可能无法满足使用要求，如何突破基因数据查询时间随人口数量线性增长的壁垒会是另一个难题。另一点值得注意的地方是，基因型压缩器在压缩实验中使用的人口规模为 50 万的数据时，占用了约 16GB 的内存并花费了长达 16 小时的时间。尽管对于基因数据的压缩属于一次性的工作，但是当数据规模非常庞大时，对于内存和时间的巨大消耗也是需要考虑的。在不断测试基因型查询速度的实验中，证实了其在查询时对于内存的消耗极低，这意味着它可以被广泛的在个人电脑上投入使用。同时，如果其可以支持图形界面，会给使用者带来更大的方便。

致谢

这次毕业设计是作者在大学四年中所完成的最具有挑战性的任务之一。因为对生物信息学并没有太多的了解，在项目开始初期经历了非常多的困难，才慢慢地对基因数据的操作熟悉起来。回首过往，不仅是这次毕业设计，还有大学四年的生活都历历在目。一路下来，有坎坷，有苦涩，也有欢笑与收获。

对于这次毕业设计，我首先要感谢的是我的指导老师 Sebastian Wandelt。他在我进行毕业设计的过程中，耐心地回答我的问题，并在我感到困惑时给予指点，向我推荐了可以用于操作基因数据的工具，能够借鉴及分析的已有研究工作，以及可以用来模拟基因数据的工具等。在对基因型压缩器的查询速度进行测试时，他也指点了我的测试方案和对测试结果该如何拟合。同时，我也要感谢孙小倩老师在毕业设计的过程中所给予我的帮助。

贤仪寒同学有着丰富的使用 Microsoft Word 编写学术论文的经验，在我的论文写作中为我提供了巨大的帮助。魏焕燊同学对于图论、网络等知识有着深入的了解，在我设计新算法的过程中也给了我灵感。我也要感谢 Variant Simulation Tools 的作者 Bo Peng，他在我对这个工具的使用遇到问题时，及时地在 GitHub 上回复了我，并起到了帮助。

最后，也是最重要的，我要感谢我的父母和家人，他们在我进行毕业设计的过程中始终支持我，理解我，包容我，是我能够克服困难，完成这次毕业设计的力量源泉。

参考文献

[1] Interpol DNA Monitoring Expert Group, Interpol Handbook on DNA Data Exchange and Practice[EB/OL]. France: OIPC-INTERPOL, 2009: 5-7.

[2] Doleac, Jennifer L. The Effects of DNA Databases on Crime[J]. American Economic Journal: Applied Economics, 2017, 9(1): 165-201.

[3] The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes[J]. Nature, 2012, 491: 56-65

[4] The 1000 Genomes Project Consortium. A global reference for human genetic variation[J]. Nature, 2015, 526: 68-74

[5] Stephens ZD, Lee SY, Faghri F, et al. Big data: astronomical or genomical?[J]. PLoS biology, 2015, 13(7): e1002195.

[6] Cristina Yenyxe Gonzalez Garcia, Louis Bergelson, Petr Danecek. The Variant Call Format (VCF) Version 4.2 Specification[EB/OL]. https://github.com/samtools/hts-specs/blob/master/VCFv4.2.pdf, 2017-09-25.

[7] Danecek P, Auton A, Abecasis G, et al. The variant call format and VCFtools[J]. Bioinformatics, 2011, 27(15): 2156-2158.

[8] Li H. Tabix: fast retrieval of sequence features from generic TAB-delimited files[J]. Bioinformatics, 2011, 27(5): 718-719.

[9] Li H, Handsaker B, Wysoker A, et al. The sequence alignment/map format and SAMtools[J]. Bioinformatics, 2009, 25(16): 2078-2079.

[10] Layer R M, Kindlon N, Karczewski K J, et al. Efficient genotype compression and analysis of large genetic-variation data sets[J]. Nature methods, 2016, 13(1): 63-65.

[11] Li H. BGT: efficient and flexible genotype query across many samples[J]. Bioinformatics, 2015, 32(4): 590-592.

[12] Durbin R. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT)[J]. Bioinformatics, 2014, 30(9): 1266-1272.

[13] Zheng X, Gogarten S M, Lawrence M, et al. SeqArray—a storage-efficient high-performance data format for WGS variant calls[J]. Bioinformatics, 2017, 33(15): 2251-2257.

[14] Salomon D, Motta G. Handbook of data compression[M]. Springer Science & Business Media, 2010: 411-416

[15] Danek A, Deorowicz S. GTC: how to maintain huge genotype collections in a compressed form[J]. Bioinformatics, 2018, 34(11): 1834-1840.

[16] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding[J]. IEEE transactions on Information Theory, 1978, 24(5): 530-536.

[17] Robinson A H, Cherry C. Results of a prototype television bandwidth compression scheme[J]. Proceedings of the IEEE, 1967, 55(3): 356-364.

[18] Huffman D A. A method for the construction of minimum-redundancy codes[J]. Proceedings of the IRE, 1952, 40(9): 1098-1101.

[19] Peng B. Reproducible Simulations of Realistic Samples for Next-Generation Sequencing Studies Using Variant Simulation Tools[J]. Genetic epidemiology, 2015, 39(1): 45-52.

[20] McCarthy S, Das S, Kretzschmar W, et al. A reference panel of 64,976 haplotypes for genotype imputation[J]. Nature genetics, 2016, 48(10): 1279-1283.

[21] Lek M, Karczewski K J, Minikel E V, et al. Analysis of protein-coding genetic variation in 60,706 humans[J]. Nature, 2016, 536(7616): 285-291.

[22] Aarts, Emile HL, and Jan Karel Lenstra, eds. Local search in combinatorial optimization[M]. Princeton University Press, 2003: 215-310.

[23] Hamming R W. Error detecting and error correcting codes[J]. Bell Labs Technical Journal, 1950, 29(2): 147-160.

[24] Smith S W. The scientist and engineer's guide to digital signal processing[M]. 1997

附录 A 查询程序的优化部分

```
01. void inline Decompressor::reversePerm(uint32_t *perm, uint32_t *revPermByte, uchar_t *revPermBit,
02. int no_haplotypes)
03. {
04.     for(int i = 0; i < no_haplotypes; ++i)
05.     {
06.         revPermByte[perm[i]] = i/8; // restore which byte the original positions are in
07.         revPermBit[perm[i]] = 128 >> (i%8);
08.     }
09. }
```

```
01. void Decompressor::decodePermRev(int no_haplotypes, int vec2_start, uint32_t *revPermByte,
02. uchar_t *revPermBit, uchar_t *decomp_data_perm, uchar_t *decomp_data)
03. {
04.     uint32 x;
05.     for (x = 0; x + 8 < (uint32) no_haplotypes;)
06.     {
07.         int x8 = x / 8;
08.         int d_x8 = decomp_data_perm[x8];
09.         int d2_x8 = decomp_data_perm[vec2_start + x8];
10.
11.         if(!d_x8 && !d2_x8)
12.         {
13.             x += 8;
14.             continue;
15.         }
16.
17.         for(int x_p = 1 << 7; x_p>0; ++x, x_p >>= 1)
18.         {
19.             if (d_x8 & x_p)
20.             {
21.                 decomp_data[revPermByte[x]] += revPermBit[x];
22.             }
23.             if (d2_x8 & x_p)
24.             {
25.                 decomp_data[vec2_start + revPermByte[x]] += revPermBit[x];
26.             }
27.         }
28.     }
29.
30.     int x8 = x / 8;
31.     int d_x8 = decomp_data_perm[x8];
32.     int d2_x8 = decomp_data_perm[vec2_start + x8];
33.
34.     for (; x < (uint32) no_haplotypes; ++x)
35.     {
36.         uchar_t x_p = 128>>(x % 8);
37.
38.         if (d_x8 & x_p)
39.             decomp_data[revPermByte[x]] += revPermBit[x];
40.         if (d2_x8 & x_p)
41.             decomp_data[vec2_start + revPermByte[x]] += revPermBit[x];
42.     }
43. }
```

附录 B 对于可移动列数受限的重排序算法的实现

```
import vcf
import BitVector as bv
import numpy as np
import vcfpy as vp
import compressLib as cl
import time
import sys

startPos = 16050075
endPos = 16500000
bitVectors,POSs,refs,alts,sampleNames = cl.readVcfgzToBitVectors(filename
                                        ='gene1000chr22.vcf.gz',startPos=startPos,
                                        endPos=endPos)
colBitVecs = cl.columnToRow(bitVectors)
# Inputs bitVectors
bitVectors = colBitVecs
# Setup basic variables
amountBitVecs = len(bitVectors)
length = bitVectors[0].length()
limit = int(0.2*amountBitVecs) # Move at most 0.2 of all vectors
# The initial searching depth. After searching depth*limit without finding a good switch, the searching is reset and the threshold will be lower.
depth = 0.2
finalDepth = 1
depthGrowth = 0.1# How much the depth of searching grows, once the searching is reset
# An imperical setting, threshold for the profit rate, profit/cost
iniThresh = int(length/600)*10
# How much the threshold decreases, once the searching is reset
threshDecay = int(depthGrowth/2*iniThresh)
IDs = list(range(amountBitVecs)) # Initialize IDs
amountLevel = int((finalDepth-depth)/depthGrowth+1) # Amount of sleep levels
print(iniThresh, limit)

# Prepare for searching
moves = 0
# current threshold is the threshold hold used for this sleep level, which means it won't be changed if the fail counter is not reset.
currThresh = iniThresh
activePos = {}
sleepPos = [{} for i in range(amountLevel)]
sleepLevel = 0
```

```
# Setup the lowest threshold, so that once the profit rate is lower than this value, put it to the last sleep level.
lowestThresh = iniThresh - threshDecay*(amountLevel-1)
# Use a numpy array to store all the hamming distances between 2 bit vectors
allHamsNext = np.array([bitVectors[ID].hamming_distance(bitVectors[ID+1]) for ID in
                  IDs[0:-1]],dtype=np.int16)
totalHam = np.sum(allHamsNext)

# when the amount of moved bit vectors are still under the limit
while moves < limit:
    allHamsBoth = np.append(allHamsNext,0)
    allHamsBoth[1:] += allHamsNext
    # Rank all the positions according to their allHamsBoth, in an descending order. When search for a good switch, search from the left, assuming the unfitting one usually has a large Hamming distance from both sides.
```

```
    POSs = np.flip(allHamsBoth.argsort(),-1)
    # threshold will change during the switch searching for the same position, updating to the best profit rate, in order to find the best switch and be set back to the currThresh once a searching for the best switch for a position is finished.
    threshold = currThresh
```

```
    # FailCounter counter for how many times a switch can't be found, also use to mark where the search should begin
    failCounter = 0
    for unfitPos in POSs:
        # Add active postions
        if unfitPos not in activePos:
            activePos[unfitPos] = {a for a in POSs[failCounter+1:]}
        # Define the searching area, don't have to be copy, use copy to avoid potential problem
        searchArea = activePos[unfitPos].copy()
        found = False # A flag for whether find a good switch or not
        updateHams = []
        updatePos = [] # Use list because set has an order chaos
        for pos in searchArea:
            # Assign the left and right position
            if pos < unfitPos:
                leftPos,rightPos = pos,unfitPos
            else:
                leftPos,rightPos = unfitPos,pos

            # The if else branches below are used to calculate the whose Hamming distance to the next position need to be updated and update to what. The profit will be calculated at last.
            # If they are besides each other
            if rightPos - leftPos == 1:
                tempUpdatePos = {i for i in [leftPos-1,rightPos] if -1<i<amountBitVecs-1}
                if leftPos == 0:
                    tempUpdateHams = [(bitVectors[IDs[0]].hamming_distance(
                            bitVectors[IDs[2]]))]
                elif rightPos == amountBitVecs-1:
                    tempUpdateHams=[bitVectors[IDs[-3]].hamming_distance(bitVectors[IDs[-1]])]
                else:
                    tempUpdateHams = [bitVectors[IDs[leftPos-1]].hamming_distance(
                            bitVectors[IDs[rightPos]])] # left and right switch
                    tempUpdateHams.append(bitVectors[IDs[leftPos]].hamming_distance(
                            bitVectors[IDs[rightPos+1]]))

            # If they are separated by 1 position
            elif rightPos - leftPos == 2:
                tempUpdatePos = {i for i in list(range(leftPos-1,rightPos+1)) if
                                 -1<i<amountBitVecs-1}
                if leftPos == 0:
                    tempUpdateHams = [allHamsNext[1]]
                    tempUpdateHams.append(allHamsNext[0])
                    tempUpdateHams.append(bitVectors[IDs[0]].hamming_distance(
                            bitVectors[IDs[3]]))
                elif rightPos == amountBitVecs-1:
                    tempUpdateHams = [bitVectors[IDs[-4]].hamming_distance(
                            bitVectors[IDs[-1]])]
                    tempUpdateHams.append(allHamsNext[-1])
                    tempUpdateHams.append(allHamsNext[-2])
                else:
                    tempUpdateHams = [bitVectors[IDs[leftPos-1]].hamming_distance(
                            bitVectors[IDs[rightPos]])]
                    tempUpdateHams.append(allHamsNext[leftPos+1])
                    tempUpdateHams.append(allHamsNext[leftPos])
                    tempUpdateHams.append(bitVectors[IDs[leftPos]].hamming_distance(
                            bitVectors[IDs[rightPos+1]]))
            # If they are separated by more than 1 position
            else:
                tempUpdatePos = {i for i in [leftPos-1,leftPos,rightPos-1,rightPos] if
                                 -1<i<amountBitVecs-1}
                if leftPos == 0:
                    tempUpdateHams = [bitVectors[IDs[rightPos]].hamming_distance(
                            bitVectors[IDs[1]])]
                else:
                    tempUpdateHams = [bitVectors[IDs[leftPos-1]].hamming_distance(
                            bitVectors[IDs[rightPos]])]
                    tempUpdateHams.append(bitVectors[IDs[rightPos]].hamming_distance(
                            bitVectors[IDs[leftPos+1]]))
```

```
                if rightPos == amountBitVecs-1:
                    tempUpdateHams.append(bitVectors[IDs[-2]].hamming_distance(
                            bitVectors[IDs[leftPos]]))
                else:
                    tempUpdateHams.append(bitVectors[IDs[rightPos-1]].hamming_distance(
                            bitVectors[IDs[leftPos]]))
                    tempUpdateHams.append(bitVectors[IDs[leftPos]].hamming_distance(
                            bitVectors[IDs[rightPos+1]]))

            profit = np.sum(allHamsNext[list(tempUpdatePos)] - tempUpdateHams)
            # Deside switch or not according to profit and cost
            if profit < 0:
                activePos[unfitPos].remove(pos)
                continue
            cost = int(IDs[unfitPos]==unfitPos) + int(IDs[pos]==pos) - int(IDs[
                    unfitPos]==pos) - int(IDs[pos]==unfitPos)
            if cost < 0:
                # Select this switch as the best switch and do it anyway
                updateHams = tempUpdateHams
                updatePos = list(tempUpdatePos)
                fitPos = pos
                costSwitch = cost
                found = True
                break
            elif profit==0:
                activePos[unfitPos].remove(pos)
                continue
            elif cost==0: # profit > 0, no cost, do the switch
                updateHams = tempUpdateHams
                updatePos = list(tempUpdatePos)
                fitPos = pos
                costSwitch = cost
                found = True
                break
            else:
                rate = profit/cost
            if rate > threshold:
                updateHams = tempUpdateHams
                updatePos = list(tempUpdatePos)
                fitPos = pos
                threshold = rate
```

```
                costSwitch = cost
                found = True
                continue
            # The switch is even worse than the current threshold, put the switch to sleep, the
    sleep level is based on its profit/cost
            elif lowestThresh < rate <= currThresh:
                level = int((iniThresh-rate)/threshDecay)
                activePos[unfitPos].remove(pos)
                if unfitPos in sleepPos[level]:
                    sleepPos[level][unfitPos].add(pos)
                else:
                    sleepPos[level][unfitPos] = {pos}
                continue
            elif rate <= lowestThresh: # Rate is too low, put to the deepest sleep area
                activePos[unfitPos].remove(pos)
                if unfitPos in sleepPos[-1]:
                    sleepPos[-1][unfitPos].add(pos)
                else:
                    sleepPos[-1][unfitPos] = {pos}
                continue
            else: # The switch is not the best, but good enough to be active
                continue

```

```
184         if found:
185             #Switch
186             IDs[fitPos],IDs[unfitPos] = IDs[unfitPos],IDs[fitPos]
187             moves += costSwitch
188             updatePos.sort() # Because when convert from set, the order maybe weird
189             allHamsNext[updatePos] = updateHams
190 
191             # the positions that can be reactivated
192             refreshArea = {i for i in list(range(unfitPos-1,unfitPos+2))+list(
193                     range(fitPos-1,fitPos+2)) if -1 < i < amountBitVecs}
194             if abs(unfitPos - fitPos) == 2:
195                 refreshArea.remove((unfitPos+fitPos)/2)
196 
197             # Update active positions
198             for key in activePos:
199                 activePos[key].update(refreshArea)
200                 if key in refreshArea:
201                     activePos[key].remove(key)
202 
203             newTotalHam = np.sum(allHamsNext)
204             deltaHam = totalHam - newTotalHam
205             totalHam = newTotalHam
206 
207             print('Total Hamming Distance:',totalHam,' Moves:',moves,' profit:',deltaHam,'
    FailCounter:',failCounter,' Current Thresh:',currThresh)
208             break
209 
210         else:
211             failCounter += 1
212             if failCounter % (0.05*amountBitVecs) == 0:
213                 print('Still Running, failCounter:',failCounter)
214             if failCounter > depth*limit:
215                 failCounter = 0
216                 currThresh -= threshDecay
217                 depth += depthGrowth
218                 if depth >= 1: #  Searching is too difficult, just finish ASAP
219                     currThresh = 0
220                     continue
221 
222                 # Wake up the sleeping positions in this level
223                 for key in sleepPos[sleepLevel]:
224                     activePos[key].update(sleepPos[sleepLevel][key])
225                 sleepPos[sleepLevel].clear()
226                 sleepLevel += 1
227                 print('Bar too high, reset threshold. Sleep Level:',sleepLevel)
                    break
```