基于压缩后缀数组的短读序列比对算法

李双江

June 2014

1 引言

随着生物学，医学等相关科学的发展，新的 DNA 测序技术不断涌现，其中，以 Illumina/Solexa 为代表的 NGS(NEXT-GENERATION SEQUENCING DAT) 技术以其低廉的测序成本成为当前的主流 DNA 测序技术。NGS 测序的每一次测序实验都会得到大量的短读 (short reads) 序列 (5 亿到 20 亿个)。在对测序得到的数据做下一步分析研究前，首先要对短读进行重组 (assembly) 或者比对 (alignment)，这也对应着从头测序和重测序两种常见的测序。重测序实验中得到的短读中每一个短读一般不超过 1000 个碱基，，但数量巨大。短读比对的的目标是把测序实验中得到的大梁短读映射到一个参考序列上去，参考序列一般选择已经测序的基因组，如人类基因组等。映射的过程是对每一个短读在参考序列上查找一个合适的位置，使得短读在该位置和参考序列尽量匹配。

综上所述，短读序列的映射问题可以抽象为一个模式查找问题：给定一个共有 m 个模式的模式集合 $P = \{_1, P_2 \dots P_m\}$，每个模式的长度已知分别为 $l_1, l_2 \dots l_m$，已知一个长为 n 的参考序列 T，求得一个集合 $S = \{s_1, s_2 \dots s_n\}$ 使得 $P_i = T_{s_i \dots s_i + l_i - 1}$。这个查找的过程即为短读到参考序列的比对映射。其中参考序列 T 和短读序列 P_i 都是由 DNA 测序中常用的碱基字符 $\{A, T, C, G, U, N\}$ 构成的。

为实现快速且准确的短读序列映射，近年来出现了很多比对算法。所有这些算法都可以分为两类，一类是通过对短读序列使用散列表等方法建立短读序列的索引，之后遍历整个参考序列。另一类是为参考序列建立索引，之后再对每个短读进行独立的比对。

第一类比对算法的代表是 MAQ，ZOOM，SHRiMP 等。MAQ[9] 基于散列技术，结合短读中每一个核苷酸的测序质量分数，实现了无空位 (ungapped) 比对。ZOOM[13] 使用了 space seeds 技术，提高了比对的精确率。而 SHRiMP[16] 则结合 space seeds" 和 smith-waterman 算法得到了更高的精确率。

第二类算法为参考序列建立索引，通过索引后的数据可以实现快速的比对。如 SOAP,WHAM,BFAST 等。SOAP[10] 使用 seeds 技术和一个散列查询表加速比对，且可以处理较少的空位比对。WHAM[12] 对参考序列建立散列表，先通过散列表查找潜在的比对位置，再进一步比对确定最终结果。BFAST[4] 则通过为参考序列建立多个索引来提高精确度。这几种方法使用的索引方法都需要很大的内存空间，所以比对时空间需求很大，尤其是在用类基因组这样的较大序列作为参考序列时。在第二类方法中以 SOAP2,Bowtie,BWA 为代表的基于 BW 变换 (Burrows-Wheeler transform,BWT)[2] 来创建参考序列索引的方法具有很大的空间优势。Bowtie[6] 使用 BWT 建立索引，采用回溯递归的搜索方法，再结合双

端搜索实现了高速，空间高效的比对，是目前最快的比对软件之一，但缺陷是不能实现空位 (gap) 比对。BWA[8] 也是基于 BWT 的一种比对算法，比对速度较 Bowtie 慢，但可实现空位比对。SOAP2[11] 使用了 bidirectional BWT 来建立参考序列的索引，比对速度和 Bowtie 相当。基于 BWT 的这些方法都使用了后向搜索方法 [14] 来加速查询。后向搜索可以在 $O(m)$ 时间内实现长为 m 的字符串的计数查询，以及 $O(m \log n)$ 时间复杂度的 query 查询。利用后向搜索的性质，Bowtie 实现了基于回溯法的非精确匹配算法，而 BWA 则采用前缀树搜索的方法实现非精确匹配。在实现非精确匹配的基础上，加上一些打分机制，既实现了短读序列到参考序列的匹配。

本文提出的 CSAA(csa alignment) 采用压缩后缀数组 (Compressed Suffix Array,CSA)[3] 索引参考序列，利用后向搜索的方法实现短读比对算法。这一算法的核心是用 CSA 的后向搜索结合优先队列实现近似搜索，从而支持短读比对，且支持 gap 比对。CSAA 中后向搜索实现快速搜索，优先队列保存所有的匹配位置，并为每个匹配位置打分，在匹配过程中，通过分支限界抛弃所有低分搜索方向，降低搜索空间，同时保证匹配结果最优。按照上一节中对短读比对算法的分类，该算法属于对参考序列进行索引的比对算法。

2 Definitions and notations

2.1 后缀数组和压缩后缀数组

压缩后缀数组 (CSA) 是由 Grossi 和 Vitter[3] 最早提出的第一种实现全文索引的压缩索引数据结构，是对后缀数组 (SA) [15] 占用空间过大的改进，并且实现了自索引特性。

设长为 n 的文本序列 T，字符集为 Σ，本文中将假设 T 有一个特殊的结尾符号 \$，\$ 不在 Σ 中并且字典序小于 Σ 中的所有符号。假设 T 存储在一个数组 $T[0 \ldots n-1]$ 中。对任何的整数 i，假设

- $T[i]$ 为 T 中从左往右 0 开始的第 i 个字符;
- T_i 为 T 的第 i 个后缀，即 $T_i = T[i]T[i+1] \ldots T[n-1]$。

的后缀数组 $SA[0 \ldots n-1]$ 定义为 T 的 n 个后缀按字典序排序后的序列，由 $\{0, 1, \ldots, n-1\}$ 的一个排列构成，满足 $T_{SA[0]} < T_{SA[1]} < \ldots < T_{SA[n-1]}$。即 $SA[i]$ 表示 T 的 n 个后缀中第 i 小的后缀的开始位置。如表1所示。后缀数组占用空间 $n \log n$，给定文本 T 和其后缀数组 $SA[0 \ldots n-1]$，T 中的任何模式 P 可以在 $O(|p| \log n + occ)$ 时间复杂度内求出其出现位置 [15]，并且不需要读原文本 T。其中 occ 是模式的出现次数。

对于任意的整数 $i \in [0 \ldots n-1]$，定义 $SA^{-1}[i] = j$ 使得 $SA[j] = i$，很明显 $SA^{-1}[i]$ 为 T_i 在 T 的所有后缀中的排名，即 T 的后缀中比 T_i 小的后缀的数量。

$$\Phi[i] = j \qquad if\ SA[j] = (SA[i] + 1) \mod n[5] \tag{1}$$

序列 T 的压缩后缀数组 (Compressed Suffix Array, CSA) 是对后缀数组 (SA) 空间复杂度过大的一个改进。其本身也是一个包含 n 个整数与后缀数组 SA 大小相同且由 SA 的近邻函数变换而来的数组 Φ。近邻函数定义如1，由于 $T[n-1] = \$$，所以 $\Phi[0] = SA^{-1}[0]$。另一个角度来看，若后缀 T_k 在 T 的后缀中排名为 i，则 $\Phi[i]$ 为后缀 T_{k+1} 在 T 的后缀中的排名。

表 1: *acaaccg*\$ 的后缀数组和 Φ 数组

i	$T[i]$	T_i	$SA[i]$	$T_{SA[i]}$	$\Phi[i]$	$T[SA[i]]$
0	a	acaaccg\$	7	\$	2	\$
1	c	caaccg\$	2	aaccg\$	3	a
2	a	aaccg\$	0	acaaccg\$	4	a
3	a	accg\$	3	accg\$	5	a
4	c	ccg\$	1	caaccg\$	1	c
5	c	cg\$	4	ccg\$	6	c
6	g	g\$	5	cg\$	7	c
7	a	\$	6	g\$	0	g

同时，可以看到 $SA^{-1}[1] = SA^{-1}[SA[SA^{-1}[0]+1] = \Phi[\Phi[SA[SA^{-1}[0]]]] = \Phi[\Phi[0]]$，同理可以得到 $SA^{-1}[2] = \Phi[\Phi[\Phi[0]]]$。以此类推，即可根据 $\Phi[0\ldots n-1]$ 迭代求出 $SA^{-1}[0\ldots n-1]$，由 $SA^{-1}[0\ldots n-1]$ 可快速求出 $SA[0\ldots n-1]$。由此可得出，从后缀数组 $SA[0\ldots n-1]$ 到数组 $\Phi[0\ldots n-1]$ 的变换是可逆的。

$\Phi[0\ldots n-1]$ 包含 n 个整数，显示存储时，也需要 $n\lceil\log n\rceil$ 位的存储空间，同后缀数组 SA 相同。然而，观察表1 可以发现 $\Sigma[1\ldots n-1]$ 可以分解为 $|\Sigma|$ 个严格递增的序列，这使得压缩后缀数组可以用简明数据结构存储。而 $\Sigma[1\ldots n-1]$ 的递增属性则是基于以下引理。

引理 2.1. *对于任意的整数 $i<j$, 若 $T[SA[i]] = T[SA[j]]$，则 $\Phi[i] < \Phi[j]$。*

证明. 当 $i<j$ 时，则 $T_{SA[i]} < T_{SA[j]}$ 一定成立，反之亦然。这等价于当 $T[SA[i]] = T[SA[j]]$ 时，$T_{SA[i]+1} < T_{SA[j]+1}$，即 $T_{SA[\phi[i]]} < T_{SA[\Phi[j]]}$，所以可以得到 $\Phi[i] < \Phi[j]$。即引理2.1成立。 □

对任意一个 Σ 中的字符 c，定义 $\alpha(c)$ 为 T 的后缀中首字符小于 c 的后缀的数目，定义 $\beta(c)$ 为 T 的后缀中首字符为 c 的后缀的数目。则有以下结论：

推论 2.1. *对于 Σ 中的任意一个字符 c，$\Phi[\alpha(c)], \Phi[\alpha(c)+1]\ldots\Phi[\alpha(c)+\beta(c)-1]$ 是一个严格递增序列。*

证明. 对于任意的字符 c，$T[SA[\alpha(c)]] = T[SA[\alpha(c)+1]] = \ldots = T[SA[\alpha(c)+\beta(c)-1]] = c$，由引理2.1可知，$\Phi$ 在 $\Phi[\alpha(c)\ldots\alpha(c)+\beta(c)-1]$ 上严格递增。 □

根据以上结论，Φ 可以划分为 $|\Sigma|$ 个递增序列，这个递增性质是压缩后缀数组可压缩性的本质保证。Grossi 和 Vitter[3] 据此提出了一种压缩模式来存储 Φ，使得可以在 $O(n(H_0+1))$ 位的空间内存储 Φ 数组，其中 $H_0 \le \log|\Sigma|$，是文本 T 的 0 阶经验熵。

2.2 后向搜索

压缩后缀数组上的后向搜索模式匹配算法是由 Sadakan 最早提出来的 [17]。如上文中所述，我们用 (l,r) 表示一个模式串在 T 上的后缀位置，模式匹配的目的正是要找到 P 的

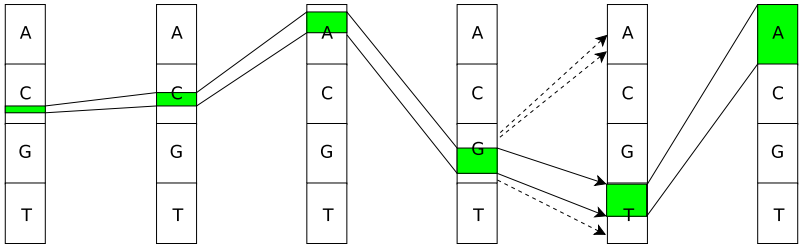

图 1: CSA 上的后向搜索

(l,r)。首先，计算 $P[m]$ 的后缀位置，很明显 $(\alpha(P[m]),\alpha(P[m])+\beta(P[m])-1)$ 就是。假设一般情况，我们已经知道 $P[i+1\ldots m-1]$ 的后缀数组位置为 $(l_{P_{i+1}},r_{P_{i+1}})$，那么 $P[i\ldots,m-1]$ 的后缀数组位置 (l_{P_i},r_{P_i}) 必定是由 $P[i]$ 的后缀数组位置的一部分组成的，设 $P[i]$ 的后缀数组位置为 $(l_{P[i]},r_{P_i})$，那么对于 $l_{P[i]\le k\le r_{P_i}}$ 只要 $SA^{-1}SA[k]+1$ 在 $(l_{P_{i+1}},r_{P_{i+1}})$ 中，k 一定在 (l_{P_i},r_{P_i}) 中。也就是说 $P[i\ldots n-1]$ 的出现位置一定是 $P[i]$ 的出现位置后面紧跟着 $P[i+1\ldots m-1]$ 的出现位置。加上 $\Phi[i]=SA^{-1}[SA[i]+1]$，所以我们可以得到公式2。

$$k\in(l_{P_i},r_{P_i})\iff k\in(l_{P[i]},r_{P[i]})\wedge\Phi[k]\in(l_{P_{i+1}},r_{P_{i+1}}) \tag{2}$$

由于在简明数据结构存储下 Φ 可以在常数时间内获得，并且其值在 $(l_{P_{i+1}},r_{P_{i+1}})$ 上是递增的，所以，可以用二分搜索来搜索 $\Phi[k]$ 是否在 $(l_{P_{i+1}},r_{P_{i+1}})$ 上，时间复杂度是 $O(\log n)$。重复上述过程 m 次，即可在 CSA 上用 $O(m\log n)$ 时间复杂度找到模式 $P[0\ldots m-1]$ 的后缀数组范围 (l,r)

算法1给出了 CSA 上的后向搜索的伪代码。而图1给出了一个采用算法1 的例子，搜索模式 $P=CCAGTA$。每一个方块儿对应一个字符的后缀数组区域，灰色的区域表示对应 P 的一个后缀在 T 的后缀数组上的位置范围。在右起第二步，计算新的区域时，根据下一个字符 G 的后缀范围，计算其 Φ 值，查看这个 Φ 是否落在当前模式 TA 的后缀范围内，由于 Φ 的递增特性，很容易用二分搜索方法找到新的区域的上下界。

算法 1 后向搜索

Input: P,Φ,α,β

Output: (l,r)

1: **function** BACKWARDSEARCH(P,Φ,α,β)
2: $l_m\leftarrow 0; r_m\leftarrow n-1$
3: **for** $i\leftarrow m-1$ down to 0 **do**
4: $l_i\leftarrow\min\{k\in[\alpha(P[i]),\alpha(P[i])+\beta(P[i])-1],\Phi[k]\in[l_{i+1},r_{i+1}]\}$
5: $r_i\leftarrow\max\{k\in[\alpha(P[i]),\alpha(P[i])+\beta(P[i])-1],\Phi[k]\in[l_{i+1},r_{i+1}]\}$
6: **if** $l>r$ **then**
7: **return** $\varnothing$
8: **return** (l_0,r_0)

2.3 CSA 索引

压缩后缀数组有两个基本特征:Φ 数组的递增特性以及算法1中的后向搜索。前者保证了通过特定的编码可以实现压缩后缀数组的压缩性质，后者则保证了压缩后缀数组对文本的检索能力。本文中使用的压缩后缀数组索引是用 gap 编码实现压缩存储，使用后向搜索做匹配搜索的一种全文压缩索引。索引只需一次建立，即无需再保存庞大的参考序列数据，并且一次建立可以多次使用，而且空间利用率最高。

3 CSA 短读比对算法

CSA 短读比对算法 (CSAA) 采用 CSA 索引参考序列，利用 CSA 的后向搜索完成匹配过程。CSA 的后向搜索是一种精确匹配算法，由于 DNA 序列存在变异，以及测序技术有一定的错误率，所以不能简单的使用精确搜索算法。本文提出了两种策略来实现短读比对中的非精确匹配要求。一是在后向搜索过程中引入了搜索树，使得在搜索过程中可以随时对短读序列进行插入，删除，替换等操作。另一种策略是采用类似优先队列的数据结构，这一数据结构结合打分机制保证在后向搜索的每一步中都是沿着最优的搜索方向前进，并且采用分支限界的策略适时淘汰很差的搜索方向。

3.1 精确匹配

精确匹配本质上就是是后向搜索。根据算法1，可知给定一个长为 n 的参考序列 T，P 为测序得到的短读序列中的一个短读，假设已经获取到短读的后缀 $P[i+1\ldots m-1]$ 的后缀数组位置 $(l_{P_{i+1}}, r_{P_{i+1}})$，那么可以跟据公式3 确定后缀 $P[i\ldots m-1]$ 的后缀数组位置 (l_{P_i}, r_{P_i})。

$$\begin{aligned} l_{P_i} &\leftarrow \min\{k \in [\alpha(P[i]), \alpha(P[i]) + \beta(P[i]) - 1], \Phi[k] \in [l_{P_{i+1}}, r_{P_{i+1}}]\} \\ r_{P_i} &\leftarrow \max\{k \in [\alpha(P[i]), \alpha(P[i]) + \beta(P[i]) - 1], \Phi[k] \in [l_{P_{i+1}}, r_{P_{i+1}}]\} \end{aligned} \tag{3}$$

由于 $\Phi[\alpha(P[i])\ldots\alpha(P[i])+\beta(P[i])-1]$ 是严格递增序列，所以公式3可以采用二分搜索实现。搜索 $[\alpha(P[i], \alpha(P[i])+\beta(P[i])-1]$ 的左边界和右边界在 $\Phi[\alpha(P[i])\ldots\alpha(P[i])+\beta(P[i])-1]$ 上是否出现，即可得 $P[i\ldots n-1]$ 的后缀数组位置 (l_{P_i}, r_{P_i})。

综上所述，可以得到以下精确匹配算法2，该算法采用后向搜索的方法，根据一个给定的模式 X 的后缀数组 (l, r) 和字符 c 计算出新模式 cX 的后缀数组位置。根据这个算法，加上一定策略，即可得到我们下文将提出的近似匹配算法。

3.2 近似匹配

根据上一小节所述的精确匹配算法思想，模式匹配就只有两种结果，要么匹配上，要么没匹配上这种完全的匹配方式并不适合大多数存在变异和测序误差的短读序列。本文提出的 CSAA 算法是建立在非精确匹配的基础上的。

由于同一物种之间存在的个体差异 (具体在 DNA 上表现为单核苷酸变异 SNP)，以及测序技术存在的误差，Resequencing 技术得到的短读序列和该物种的标准参考序列之间哪怕是同一基因片段也必然存在着一些不同，这会造成即使同一种基因序列也无法完全比对映射到

算法 2 精确匹配

Input: $\Phi, \alpha, \beta, l_{old}, r_{old}, c$

Output: (l, r)

```
1: function ExactMatch(Φ, α, β, l_old, r_old, c)
2:     l_c ← α(c)
3:     r_c ← α(c) + β(c) − 1
4:     (l, r) ←binarySearch(Φ[l_c ... r_c], l_old, r_old)
5:     if l > r then
6:         return ∅
7:     else
8:         return (l, r)
```

参考序列上。所以对短读和参考序列之间的比对仅仅精确比对映射是远远不够的，这会造成大量的短读因为和参考序列相差一两个核苷酸不同而不能映射到参考序列上，而这一两个不同的核苷酸很可能是测序误差或者生物个体之间的 SNP 造成的，不应当认为二者是不能匹配的。所以，采用合适的非精确比对算法是必须的。本文提出的非精确匹配算法中，支持对短读进行替换 (substitude)，插入 (insert)，删除 (delete) 三种变换操作，通过这三种操作可以保证变异的或者发生测序错误的短读序列依然能正确匹配到参考序列的合适位置上。

考虑一个给定的长为 m 的短读序列 P，采用后向搜索，当搜索到第 $i+1$ 个位置时，得到序列 P_{i+1} 的后缀数组位置 (l_{i+1}, r_{i+1})，依照精确匹配的算法下一步应当在 (l_{i+1}, r_{i+1}) 的基础上搜索字符 $P[i]$，从而得到一个新的后缀数组位置 (l_i, r_i)。在此，如果我们不搜索 $P[i]$，而是搜索另一个符号 $c \neq P[i]$，得到另一个后缀数组位置 (l_i', r_i')。接着在 (l_i', r_i') 基础上继续搜索 $P[0 \ldots i-1]$，那么最终得到后缀数组位置 (l_0, r_0) 将不再是序列 P 的一个后缀数组位置了，而是将 $P[i]$ 替换为 c 后的新字符串的后缀数组位置。称这个新位置为 P 的一个替换近似串的后缀数组位置，计为 $(l_0, r_0, m, P(S_i))$，即长为 m 的序列 P 在 i 位置进行一次替换后可以映射到参考序列的后缀数组位置 (l_0, r_0)。

图2给出了把短读序列中的一个符号 T 替换为 A 时的搜索过程，实线为精确匹配的过程，虚线是替换后的搜索过程。

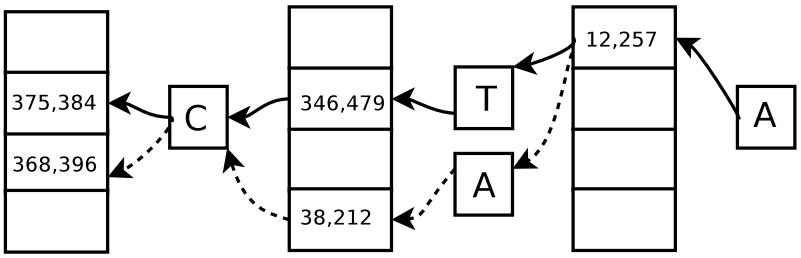

图 2: 替换操作示例

不同于替换操作，如果在搜索到 $P[i]$ 时，放弃搜索 $P[i]$ 符号，直接在 (l_{i+1}, r_{i+1}) 的基础上搜索序列 $P[0 \ldots i-1]$，那么最终得到的后缀数组位置 (l_0, r_0) 将是 P 删除 $P[i]$ 后的近似串在参考序列上的后缀数组位置，计为 $(l_0, r_0, m, P(D_i))$,，即长为 m 的序列 P 在 i 位置进行一次删除后可以映射到参考序列的后缀数组位置 (l_0, r_0)。

如图3所示为删除短读序列中的一个符号后的搜索过程。

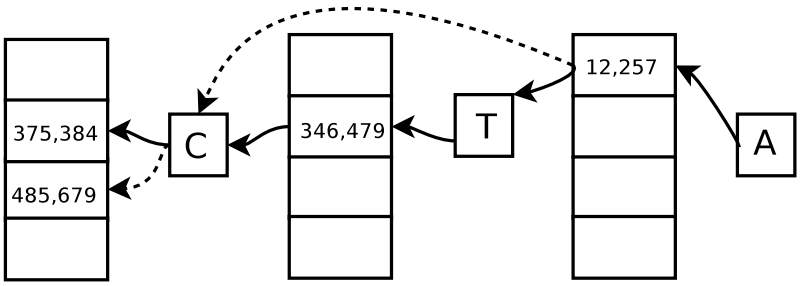

图 3: 删除操作示例

对于插入操作，在搜索到 $P[i]$ 时，不直接搜索 $P[i]$，而是在 l_{i-1}, r_{i-1} 的基础上搜索符号 c, 得到 $(l_i^{'}, r_i^{'})$，接着在此基础上搜索序列 $P[0 \dots i]$，最终得到的后缀数组位置 (l_0, r_0) 将是 P 在 i 位置插入符号 c 后的近似序列的后缀数组位置。计为 $(l_0, r_0, m, P(Ii)$，即长为 m 的序列 P 在 i 位置插入一个符号后可以映射到参考序列的后缀数组位置 (l_0, r_0)。

如图4所示为插入一个符号'A' 前后的短读序列中的一个符号后的搜索过程。

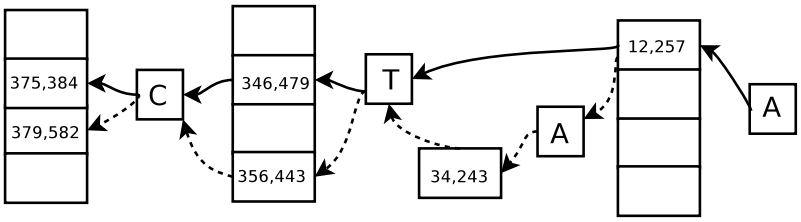

图 4: 插入操作示例

综上所述，通过在搜索的过程中加入替换，删除，插入三种操作，可以扩展后向搜索的搜索方向，达到近似匹配的目的。如果已知模式串的某个字符是不可靠的，那么我们可以通过在这个位置加入上述三种操作，实现非精确匹配，查找到这个模式串的可能匹配位置。而大多数情况下，我们是不知道具体哪一个字符不可靠的，所以只能试探每一个字符，对每一个字符都做删除，替换，插入操作，从而找到一个合适的位置。据此，我们提出非精确匹配算法3，这个算法递归的实现近似匹配，对 P 的每一个位置都做了三种操作，所以这个算法的本质是对长为 m 的序列 P 做了所有可能的近似串的搜索，即做了 $|\Sigma|^m$ 个序列的匹配。因为采用了后向搜索的过程，每一次递归实际都是产生 Σ 次替换递归，Σ 次插入递归，和 1 次删除递归，所以实际的时间复杂度满足递归式4。解递归式4可得 $T(m) = (2\Sigma + 1)^{m-1} + \Theta(m \log n)$。

$$T(m) = \begin{cases} \Theta(1) & \text{if } m = 1 \\ 2(\Sigma + 1)T(m-1) + \Theta(\log n) & \text{if } m > 1 \end{cases} \tag{4}$$

3.3 搜索树

根据上一小节中描述的方法，用后向搜索实现替换，删除，插入操作的方法，实现短读序列的近似匹配。在不知道具体的替换，删除以及插入位置时，需要在每一次后向搜索一个符号时都分别做一次替换，删除，插入操作。而每一次操作实际上都导致了一个新的近似序列的产生，在未完成整个序列的搜索时，最终哪一个近似序列能够较好的映射到参考序列上依然是未知的，这就要求我们在搜索过程中保留每一个可能的近似序列。

算法 3 近似匹配

Input: $\Phi, \alpha, \beta, l, r, P, i$

Output: (l, r)

function INEXACTMATCH($\Phi, \alpha, \beta, l, r, P, i$)
  if $i < 0$ **then**
    return (l, r)
  $S \leftarrow \varnothing$
  $S \leftarrow S \cup$ INEXACTMATCH($\Phi, \alpha, \beta, l, r, P, i-1$) ▷ 删除 $P[i]$ 操作
  for all $c \in \Sigma$ **do**
    $(l, r) \leftarrow$EXACMATCH($\Phi, \alpha, \beta, l, r, c$) ▷ 调用算法2
    if $l \leq r$ **then**
      $S \leftarrow S \cup$ INEXACTMATCH($\Phi, \alpha, \beta, l, r, P, i-1$) ▷ 替换 $P[i]$ 为 c
      $S \leftarrow S \cup$ INEXACTMATCH($\Phi, \alpha, \beta, l, r, P, i$) ▷ 在 $P[i]$ 后插入 c
  return S

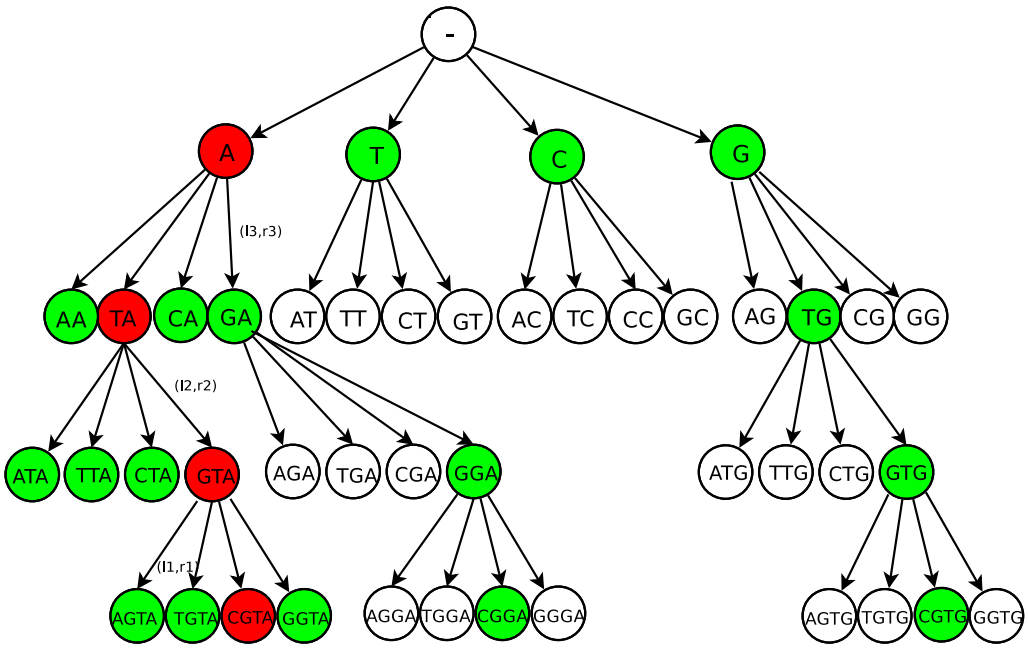

图 5: 近似匹配示例

实际上，每一次的替换，删除或者插入操作导致的新的近似序列可以看作一个树上的遍历过程。如图5所示，对于一个长为 4 的短读 P =“CGTA” 的搜索过程，简化起见，省略了删除和插入操作。首先通过 CSA 查询 $P[3]$ =‘A’ 在参考序列上的后缀数组位置。接着把 $P[3]$ =‘A’ 分别替换为 ‘T’，‘C’，‘G’，在 CSA 上查找替换后的后缀数组位置。图中红色路径标志的是到当前搜索深度没有替换操作的近似串，而绿色标记的则是到当前深度有一次替换操作的近似串，无色的是有两次以上替换操作的近似串搜索路径。可以看到，随着搜索深度的增加，搜索的方向急剧扩大，加上还有未画出的删除和插入操作，可能的搜索方向会更多。

搜索树的本质是对每一个可能的近似序列都进行搜索，假设一个短读序列的长度为 m，DNA 序列字符集为 4，根据上一小节，总的时间复杂度为 $9^{m-1} + \Theta(m \log n)$。考虑到 m 一般在 20 到 70 之间，如果直接采用搜索树来进行近似匹配，搜索规模会非常大而难以在有效时间内实现。实际上也无需比对所有的近似串，对于某些替换，删除或者插入操作过多的近似串，应该提前抛弃掉，即采用分支限界法来对搜索树进行剪枝，去除掉无效的搜索方向，降低搜索空间。

3.4 分支限界

根据上一小节的分析，搜索树的搜索空间复杂度是随短读序列长度指数级增长的，因此必须进行适当的剪枝，抛弃一些没有价值的搜索方向。传统的 DNA 序列比对中用到了编辑距离 (Edit Distance)，汉明距离 (Hamming Distance) 等来度量两个序列的相似度。在此我们也可以使用类似的方法做分支限界，比如采用编辑距离，可通过给定一个允许的最大编辑距离 $maxDistance$，当在短读序列 P 上后向搜索进行到第 i 步时，在某个搜索方向上的一个近似序列为 P'，计算 P 和 P' 之间的编辑距离，若距离大于预先定义的 $maxDistance$，则抛弃这个搜索方向，否则，保持这个搜索方向。汉明距离做分支限界的方法和编辑距离方法类似。

采用编辑距离和汉明距离的方法必须对每一个得到的近似序列和短读序列计算距离，这一方面会导致算法效率的降低，另一方面由于我们在做短读比对时，会有大量的短读需要比对，这些短读的长度不一定都相同，指定唯一的 $maxDistance$ 是不合适的。对此，在论文 [8] 中，作者提出了一种类似于编辑距离的算法，称之为 $difference$ 数组，该算法很好的实现了自适应的类编辑距离的最大限制距离。因此本文提出的 CSAA 比对算法也使用 $difference$ 数组作为分支限界的一个条件。$difference$ 数组是利用 CSA 上的精确匹配算法，首先对需要比对的短读做预处理，处理过程如算法4所示。处理过的 $difference[i]$ 反映了在搜索 $P[i \ldots n-1]$ 时可以允许的最大替换，插入，删除操作的次数。由于在序列比对中我们采用的是后向搜索的过程，所以在计算 $difference$ 数组时，也需要从后往前算，使得 $difference$ 是一个递减的序列，也即在从后往前比对短读 $P[0 \ldots n-1]$ 时，越往前，允许的替换，插入，删除数量越多。

$difference$ 数组的作用，是结合每一个近似序列的一个辅助值 z 来分支限界的。开始比对时，序列 P 的 z 值是 0，之后每做一次插入，删除或者替换，则 z 增加一次，而在该操作位置的最大允许删除，插入，替换操作数是 $difference[i]$。比较两者，若 $z > difference[i]$ 则证明该搜索方向做了过多的替换，删除，插入操作，短读已经和参考序列相似度太小，应当删除该搜索方向，考虑搜索树上的其他方向。

除了 $difference$ 距离，另一可用的分支限界策略是罚分机制 [7]。在搜索树上向前搜索时，每一次替换，删除或者插入操作都会产生一个新的近似序列，而每增加一次这样的操作，都会导致这个方向上的近似序列和参考序列的相似度下降。基于此，我们可以预定义一个罚

算法 4 计算 $difference$ 数组

Input: Φ, α, β, P

Output: $difference[0 \ldots |P|-1]$

```
1: function CalculateDifference(Φ, α, β, P)
2:     z ← 0
3:     l ← 0
4:     r ← |P| − 1
5:     for i ← |P| − 1 downto 0 do
6:         (l, r) ← ExactMatch(Φ, α, β, l, r, P[i])          ▷ 调用算法2
7:         if l > r then          ▷ 失配，difference[i] 增加一次, 否则 difference[i] 不变
8:             l ← 0
9:             r ← |P| − 1
10:            z ← z + 1
11:        difference[i] ← z
```

分上限 $maxPenalty$。同时为替换，插入，删除三种操作分别定义各自的罚分，当这个短读比对过程中每做一次上述操作时，都会对这个近似序列增加相应的罚分，当罚分达到预定义的最大罚分 $maxPenalty$ 时，即认为这个近似序列已经和短读序列相似度太小，没有必要再继续搜索这个方向。通过这样的罚分机制，可以去除较差的近似序列，达到分支限界的目的。

在生物信息学领域，每一个删除或者插入统称为一个 indel，而每一个 indel 都会在短读序列上造成一个空位 (gap)，对应的罚分称为空位罚分。gap 通常分为两种，一种是单独出现在序列中，称为 gap open；另一种是连续的 gap，称为 gap extension。gap open 的罚分是各个 gap 罚分的线性叠加和，而通常 gap extension 的罚分不能简单的做线性增加来处理。Affine gap penalties[1] 是生物信息学领域常用的一种罚分机制，假设有一个 gap extension 长为 x，则这个 gap 的罚分为 $-(\rho + \sigma x)$，其中 $\rho > 0$，σ 是每一个 indel 操作的罚分。

有了 $difference$ 距离和 gap 罚分两种分支限界的策略，可以大大减少搜索的规模，通过控制合适的 $maxPenalty$ 可以实现在比对时间和比对精度之间的平衡。然而如果直接依赖算法3实现比对算法也是不合适的。因为算法3是一个递归的算法，递归算法本身在实际实现时过于低效，我们的 short read 一般都长 30bp 以上，这样的深度是不能用递归的。另一方面，递归比对的本质在搜索树上表现为一个深度优先的搜索，每一次递归都会把一个方向上的所有方向做一次尝试，直到遇到分支限界条件不合适才会回溯到上一层。这一特性会引起过度回溯的问题。这一问题的本质是每次进入一个新的搜索方向时没有做出正确的选择，整体的搜索方向是随机的，算法大多数的递归选择都会进入不是最好的方向，从而最终不得不回溯，这显著增大了时间开销。据此本文提出一种优先堆结构，把搜索树上的深度优先搜索改为广度优先搜索。每一层上都把所有的搜索方向排序放在一个优先堆里，算法每进入下一个搜索方向都在当前优先堆中选择最优的即罚分最低的搜索方向进入。借助这一优先堆结构，可以把搜索方向限制在只沿着最优的搜索方向前进，从而实现最优比对。同时，还可以限定优先堆的大小，抛弃掉虽然符合分支限界条件但相对于其他搜索方向过差的搜索方向。

优先队列使用了最大堆来实现，每一项保存一个近似匹配的序列相关匹配信息，以每个近似序列的罚分和 z 值作为优先堆的 key 值。本文中使用的优先堆支持如下操作:

- INIT-HEAP($heap$)：初始化优先堆 $heap$ 为空。

- INSERT-HEAP($heap, x$)：把元素 x 插入优先堆 $heap$ 中，并按照罚分排序。
- EXTRACT-MIN($heap$)：去掉并返回 $heap$ 中罚分最小的元素。
- HEAP-SIZE($heap$)：返回 $heap$ 中元素的数量。
- HEAP-DROP-MAX($heap$)：删除 $heap$ 中罚分最大的算素。

综上所述，我们提出本文的核心比对算法6，该算法中使用了算法5，后者是在给定的位置上做替换，插入，删除操作的过程。比对算法6可以指定 $maxPenalty$ 和 $maxHeapSize$ 两个参数来控制比对的精度和比对时间之间的平衡。

算法 5 处理替换，删除，插入

Input: $P, \Phi, \alpha, \beta, x, heap$

Output: $heap$

```
1: function ProcessIndel(P, Φ, α, β, x, heap)
2:     y.l ← x.l; y.r ← x.r; y.i ← x.i − 1                          ▷ 删除 P[i]
3:     y.penalty ← x.penalty + delPenalty
4:     y.z ← x.z + 1
5:     Insert-Heap(heap,y)
6:     for all c ∈ {A, C, G, T} do
7:         (l, r) ←ExactMatch(Φ, α, β, x.l, x.r, c)
8:         if l ≤ r then
9:             y.l ← l; y.r ← r; y.i ← x.i                          ▷ 在 P[i] 之后插入 c
10:            y.penalty ← x.penalty + insPenalty;
11:            y.z ← x.z + 1
12:            Insert-Heap(heap, y)
13:            if P[i] = c then                                      ▷ match
14:                y.l ← l; y.r ← r; y.i ← x.i − 1
15:                y.penalty ← x.penalty;
16:                y.z ← x.z
17:                Insert-Heap(heap, y)
18:            else                                                  ▷ 替换 P[i]
19:                y.l ← l; y.r ← r; y.i ← x.i − 1
20:                y.penalty ← x.penalty + subPenalty
21:                y.z ← x.z + 1
22:                Insert-Heap(heap, y)
23:    return heap
```

4 CSAA 的实现

上文中给出了 CSAA 的算法过程，实际实现中 CSAA 采用了一些优化措施，进一步提高程序的效率。这些措施主要是从提高执行效率和准确度两个方面来考虑的。CSAA 主要包

算法 6 CSA 比对算法

Input: $\Phi, \alpha, \beta, P, maxPenalty, maxHeapSize$

Output: $result$ ▷ 输出为符合比对条件的所有结果组成的优先堆

```
1: function CSAAlignment(Φ, α, β, P, maxPenalty, maxHeapSize)
2:     difference ←CalculateDifference(Φ, α, β, P)
3:     heap ← Init-Heap(heap)
4:     result ←Init-Heap(result)
5:     x.l ← 0; x.r ← |P| − 1; x.i ← |P| − 1
6:     x.penalty ← 0; x.z ← 0
7:     heap ← ProcessIndel(P, Φ, α, β, x.heap)                    ▷ 处理 P[m − 1]
8:     while x ←Extract-Min(heap) do
9:         if x.i < 0 then
10:            Insert(result, x)                                  ▷ 获得一个较好的比对结果
11:            continue
12:        if x.z > difference[x.i] or x.penalty > maxPenalty then  ▷ 剪枝
13:            continue
14:        heap ←ProcessIndel(P, Φ, α, β, x, heap)                ▷ 处理 P[0 . . . m − 2]
15:        while Heap-Size(heap)>maxHeapSize do                   ▷ 去除较差的比对方向
16:            Heap-Drop-Max(heap)
17:    return result                                              ▷ 返回所有符合条件的比对结果
```

括三个子程序:build index,alignment,output。build index 子程序完成对参考序列 T 建立压缩后缀数组索引，对同一个参考序列只需建立一次索引即可。alignment 子程序是 CSAA 的核心，完成对短读序列到参考序列 T 的映射，输出 CSAA 自定义的一种 SAI 格式文件。最终 SAI 格式文件通过 output 子程序转为标准比对结果文件 SAM(Sequence Alignment/Map format)。

4.1 使用 seed 提高比对速度

在 DNA 测序技术中，单端测序和双端测序都存在测序质量的问题。离引物结合位点越近的位置，测序结果准确率越高，而越远的位置，测序准确率越低。根据这一现象，关于 Bowtie 实现的论文 [6] 中提出了 seed 的概念，该论文认为针对短读序列中测序可信度较高的部分采用较高的限制，在较低的地方做较低的限制可以取得更快的比对速度和更好的比对结果。如图6所示，read 前面绿色部分是可信度比较高的 seed 部分，后面是可信度比较低的非 seed 部分。

在算法6中，我们当某个近似序列的 $x.penalty > maxPenalty$ 时，会直接剪枝。CSAA 中实际上是分为两部分来剪枝的，seed 部有一个 $maxPenalty$，非 seed 部分有另一个 $maxPenalty$，前者较小，后者较大。通过这样的策略，可以控制搜索结果中的待选近似序列中的替换删除，插入等操作的绝大部分发生在非 seed 部分，即 read 的不可靠部分，从而有效提高比对的精度。至于 seed 的具体长度，可以根据不同的测序技术来给定。默认情况下，Bowtie 的 seed 长度是 28，随本文发布的 CSAA 中 seed 的默认长度是 32。

因为 seed 的限制条件较高，所以如果能首先对 seed 进行匹配，可以尽早抛弃掉一些不

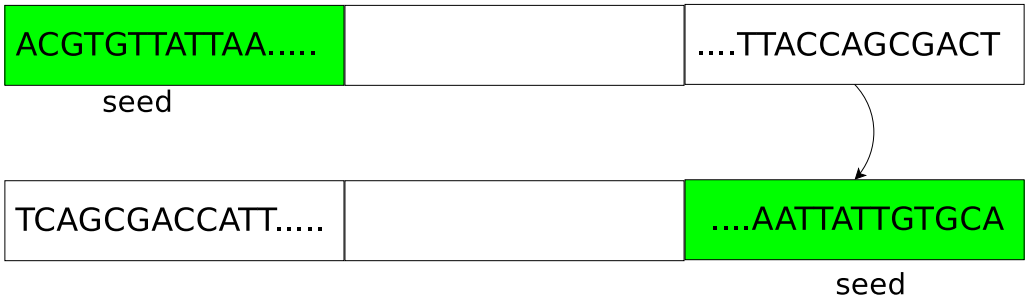

图 6: seed 示例

满足 seed 部分匹配要求，从而不用搜索非 seed 部分，从而提高搜索速度。但因为 seed 是在短读序列的前缀部分，而我们使用 CSA 索引的后向搜索时是从后往前搜索的，最后才搜索 seed 部分。所以在 CSAA 中无论是建立索引还是短读匹配都是对短读的逆序序列进行的。如图6所示，通过逆序，把 read 的 seed 部分转到后向搜索的首先搜索的部分。

在图7中，对 seed 的长度对比对速度和比对精度做了对比测试，测试中的数据同图8使用的模拟数据相同，都是 100 万条 hg19 上的仿真数据，indel 概率为 0.01%，SNP 概率为 0.09%，不同的是本次实验中只测试了长为 70bp 的短读。可以发现，seed 的选取对比对速度有着明显的影响，seed 选取的越长，比对的速度越快，几乎成线性下降的趋势。seed 的长度对 recall rate 和 accuracy 同样有影响，seed 选取的越长，recall rate 有明显下降，这是因为如果 seed 越长，seed 部分不匹配的可能越大，造成更多的短读不能匹配到参考序列上，而 accuracy 有所提高则是因为较长的 seed 选取，抛弃了更多的可能错误匹配，保证了更少的匹配错误。

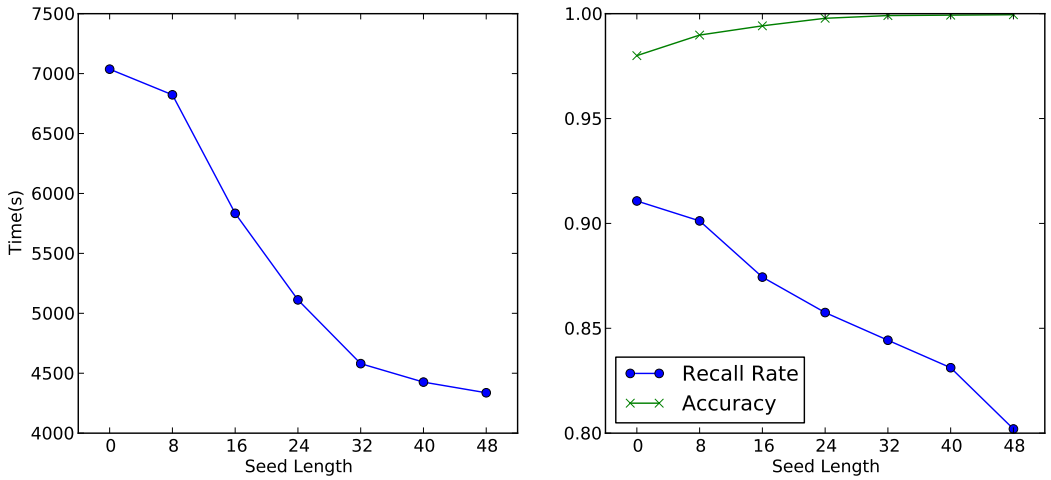

图 7: seed 长度对比对速度和比对精度的影响

本文中，比对精度包括两个测量指标，一个是 recall rate，一个是 mapping accuracy。recall rate 是正确比对的短读 (correctly mapped reads) 数占总的短读数 (all reads) 的比例，而 mapping accuracy 是正确比对的短读数 (correctly mapped reads) 占成功比对到参考序列上的短读 (mapped reads) 的比例。在序列比对中，对于实际测序实验中得到的短读数据，由

于预先并不知道短读的真正位置，所以正确比对的短读 (correctly mapped reads) 数量是未知的。只有用模拟数据才能获取短读的真正位置，再用比对软件对模拟生成的短读进行比对，把获取的比对结果和预知的模拟短读的真正位置做比较就能度量该比对软件的正确比对短读数目，从而计算出 mapping accuracy，来度量一个软件的比对的可信度。另一方面，因为比对软件在比对中设定的一些参数限制，如本文中的罚分，difference 距离等等，使得一些发生了较大的变异 (SNP,indels) 的短读不能映射到参考序列上，即该短读没有映射位置，从而造成最终在所有短读中只有一部分得以映射到参考序列上，这就是 mapped reads，而 recall rate 即反映了这一衡量短读比对性能的指标。在实际比对中，比对软件得到的参考序列上的比对位置和生成仿真数据时的实际位置并不需要完全比上，只要二者之间的距离在一定距离内即认为这个比对结果是正确的，本文中所有的实验中这一距离定义为 20。

4.2 比对过程并行化

CSAA 给参考序列建立索引需要很多时间，但我们只需要给一个参考序列建一次索引就可以了，其后的向同一参考序列的比对无需再建立一次索引。在比对阶段，由于一次实验的短读数量非常大，通常在几百万到数十亿条的规模，这使得比对过程通常耗时非常长。

观察比对的过程可以发现，每一条短读的比对都是独立进行的，之间互相只共享参考序列索引的数据和操作，且比对结果也是互相独立的一个优先堆。据此，我们可以很容易的设计一个并行计算的算法来完成比对。考虑到在计算过程中需要共享参考序列索引以及匹配算法的操作。所以 CSAA 选择了多线程编程模型作为并行方案。

csaa 在执行比对时，可以指定线程的数量，程序启动后，将建立多个线程，并把所有的短读数据分成多份，由每个线程单独执行一份，从而实现快速的比对。每个线程比对完成后都会把比对结果写如一个临时文件，待所有的线程执行完毕后再按顺序把比对结果输出到程序指定的文件。具体的并行算法如算法7所示。

算法 7 CSAA 比对并行算法

Input: $k, \Phi, \alpha, \beta, maxPenalty, maxHeapSize$

Output: $result$

1: **function** MATCH-MULT($k, \Phi, \alpha, \beta, maxPenalty, maxHeapSize$)
2: 分割 reads 文件为 k 个小文件 $\{read_1, read_2 \dots read_k\}$
3: 建立 k 个线程 $threads = \{thread_1, thread_2 \dots thread_k\}$
4: **for all** $thread_i \in threads$ **do**
5: 从文件 $read_i$ 读取一个短读 $read$
6: ▷ 调用算法6
7: $result \leftarrow$CSAALIGNMENT($\Phi, \alpha, \beta, read, maxPenalty, maxHeapSize$)
8: 输出 $result$ 到文件 $result_i$
9: 合并文件 $\{result_1, result_2 \dots result_k\}$ 到文件 $result$
10: 输出文件 $result$

通过增加比对时的线程数量可以快速减少比对的时间，如图8所示。实验中所用的参考序列为来自 NCBI 的人类基因组 hg19.fa，短读序列是用 wgsim 仿真程序生成的模拟短读数据，短读长分别为 35bp,70bp,125bp，0.09% 的 SNP 出现几率，0.01% 的 indel 出现几率，短读数量为 100 万个。可以看到，当由单线程变为双线程时，比对时间减少了一倍，这显示 CSAA

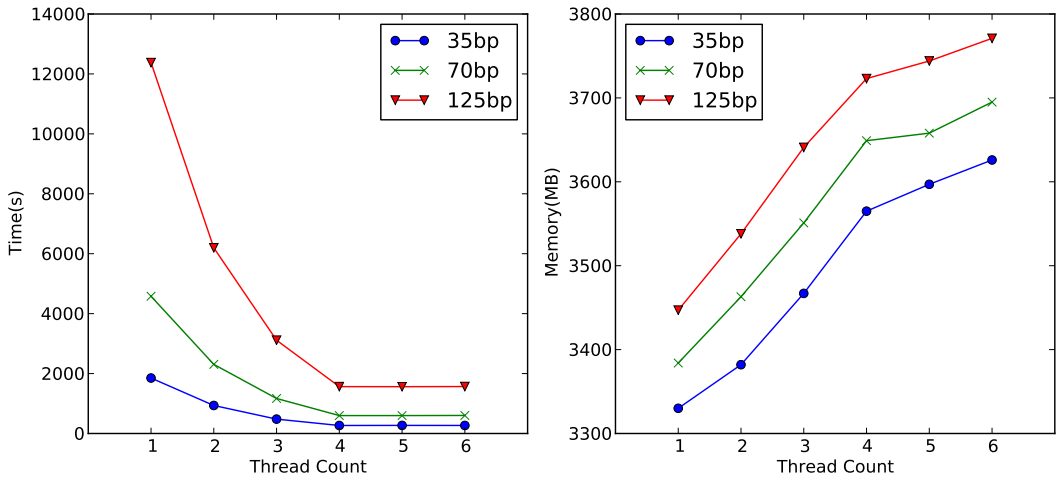

图 8: 多线程比对的时间空间效率

的多线程能明显提高比对的速度。之后再提高线程数不能再提高比对速度是因为测试机器是双核的，再多的线程已经不能有效提高 CSAA 的比对速度。对应的，当进程数增多时，比对所需要的内存也随之增多。图8中的曲线还显示，比对的速度以及内存消耗和短读长是密切相关的，短读越长，比对的时间也越长，所需内存也越多。

5 实验测试

为测试 CSAA 的性能，本文同时测试了 Bowtie[6] 和 BWA[8] 这两个个比对工具的性能，以和 CSAA 进行比较。分别对比这几个工具在不同规模数据上建立索引时的时间，需要的内存；以及在模拟数据上和真实数据上的比对能力。这三个工具中，Bowtie 和 BWA 是用基于 BWT 的索引建立的比对工具，其中 Bowtie 采用了回溯法实现替换操作，但不支持 insert 和 delete 操作，即上文中所论述的 gap alignment。BWA 除了支持 insert，delete 之外，还加入了 Smith-Waterman 方法提高比对精度。此外 Bowtie 和 BWA 和我们的 CSAA 都支持多线程比对，但在比对测试中，我们的 csaa 和 bwa, bowtie 等都只使用单线程运行。

5.1 测试环境和数据

在对 CSAA 的对比测试中，我们使用 gcc 4.8 编译链接 CSAA，并且使用了 -O3 优化。系统环境为 64 位 redhat 5.0，测试机器有 64GB 内存和 24 核 CPU。测试使用的数据是标准的人类基因组序列，ucsc version 为 hg19，来自 1000 Genome Project 的名为 Genome Reference Consortium GRCh37 的参考序列。实验中所用的模拟数据也是在该数据上生成。

测试中，CSAA 和 BWA 在测试时均指定单线程，seed 长为 32。Bowtie 使用的是 Bowtie-v1，使用单线程，使用 –best -k 2 参数。Bowtie 使用长为 32 的 seed。模拟数据的获取和比对精度结果获得中使用了 SAMtools 系列工具中的 wgsim 和 wgsim-eval。

5.2 索引建立时间

表2中对比了 bowtie,BWA 和 CSAA 三种工具分别建立索引时所需要的时间和内存消耗对比情况，表中数据分别对应几个工具在索引 512M,1024M 和 2048M 的数据时所需要的时间和内存。CSAA 中使用经典方法构建压缩后缀数组。

表 2: 建立索引的时间空间对比

Program	Time(m)			Memory(MB)		
	512M	1024M	2048M	512M	1024M	2048M
Bowtie	21.85	45.33	93.02	987	1109	1210
BWA	7.85	16.33	34.72	1890	1902	2006
CSAA	6.27	14.0	29.87	2113	8680	19532

从表2中的测试结果可以看到，CSAA 相对其他两个比对工具在索引时间上具有较大优势。其中 Bowtie 建立索引时是分块建立索引的，最终再合并成一个索引，时间效率最差，但需用内存空间最小。

5.3 模拟数据测试

本文使用 SAMtools 工具包中的 wgsim 工具从人类基因组序列 hg19 中随机生成模拟的短读序列。然后，分别用四种比对工具对这些模拟的短读序列进行比对，最后测试比对速度以及比对精确度。因为这些模拟数据在参考序列上的映射位置是已知的，所以可以计算出各个工具的比对结果的 accuracy。本次实验中的精度测试工具使用 SAMtools 中的 wgsim_eval.pl 程序。

表3和4所示为四个测试工具在单端和双端模拟数据上的的比对结果展示，参考序列是人类基因组序列 hg19，模拟短读序列的长度为 70bp，总共有 100 万个短读。

表3和4中的实验的模拟数据使用 wgsim 程序在人类基因组上生成的短读长为 70 的模拟数据，生成过程中，单核苷酸变异 (SNP) 的概率是 0.09%，indel 变异的概率是 0.01%，生成双端模拟数据时的 fragment distance 的长度满足正太分布 $N(500, 50)$。对比结果中 Rec 是 recall rate，Acc 是 accuracy。意义如上文中所解释。从表中可以看出，CSAA 相对于 BWA 和 Bowtie 在查询时更节省内存，在 recall rate 和 accuracy 上和 BWA，Bowtie 基本持平。

表 3: 单端模拟数据比对测试

Program	Time	Memory(M)	Rec(%)	Acc(%)
Bowtie	24m38s	2923	80.21	98.64
BWA	39m57s	3506	89.32	99.91
CSAA	1h16m20s	3384	84.44	99.91

表 4: 双端模拟数据比对测试

Program	Time	Memory(M)	Rec(%)	Acc(%)
Bowtie	26m29s	2931	83.47	98.67
BWA	43m25s	4954	94.71	99.97
CSAA	1h17m55s	4023	88.57	99.97

5.4 真实数据测试

为测试 CSAA 在真实数据上的表现，本文从网络上下载了 12.2 million 个长为 51bp 的短读数据库。这些数据来自 European Read Archive (AC:ERR000589)，是 1000 Genomes Project 的一个名男性基因组测序，由 Illumina 测序技术完成测序。参考序列选的是人类基因组，测序编号 hg19。由于真实数据的比对位置并不能事先获取，所以在测试真实数据时使用 Mapped rate 来度量比对效果。Mapped rate 指比对程序在指定参数条件下能够给出比对结果的短读数目和所有测试数据中短读数目总和的比例。在本次比对中测试中，Bowtie,BWA 和 CSAA 使用默认的长为 32 的 seed 策略，Bowtie 使用 –best -k 2 参数。

表 5: 实际数据比对测试

Program	Time	Memory(MB)	Mapped(%)	Paired(%)
Bowtie	3h48m47s	2929	75.2	83.75
BWA	3h4m24s	4845	87.67	99.83
CSAA	6h49m17s	4008	83.18	99.75

表5中的测试结果显示，在实际数据中 CSAA 也具有较高的准确性，84% 的短读序列都能映射到参考序列上，基本性能和 BWA 很接近。在时间方面，CSAA，BWA，Bowtie 都使用索引参考序列再比对的方法，而 MAQ 则使用了索引短读序列的方法，所以 CSAA，BWA，Bowtie 的比对时间都比较接近，Bowtie 能更快一些。在比对使用的空间上，相对而言，CSAA 使用了更少的内存，空间效率优于其他三种发放，在常规 PC 机上具有优势。

6 总结

本文提出了一种基于压缩后缀数组的短读比对算法，利用 CSA 的后向搜索实现了近似匹配，结合搜索树和优先堆，最终提出了一种高效准确的短读比对算法。该算法和主流的短读比对算法相比，准确性更高，且使用更少的内存，在准确率和空间高效方面达到很好的权衡。

参考文献

[1] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[2] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[3] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[4] Nils Homer, Barry Merriman, and Stanley F Nelson. Bfast: an alignment tool for large scale genome resequencing. *PloS one*, 4(11):e7767, 2009.

[5] Hongwei Huo, Longgang Chen, Jeffrey Scott Vitter, and Yakov Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *Data Compression Conference (DCC), 2014*, pages 292–301. IEEE, 2014.

[6] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[7] Arthur M Lesk1e2, Michael Levitt, and Cyrus Chothia1a4. Alignment of the amino acid sequences of distantly related proteins using variable gap penalties. *Protein Engineering*, 1(1):77–78, 1986.

[8] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[9] Heng Li, Jue Ruan, and Richard Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

[10] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[11] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[12] Yinan Li, Jignesh M Patel, and Allison Terrell. Wham: a high-throughput sequence alignment method. *ACM Transactions on Database Systems (TODS)*, 37(4):28, 2012.

[13] Hao Lin, Zefeng Zhang, Michael Q Zhang, Bin Ma, and Ming Li. Zoom! zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.

[14] Ross A Lippert. Space-efficient whole genome comparisons with burrows-wheeler transforms. *Journal of Computational Biology*, 12(4):407–415, 2005.

[15] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[16] Stephen M Rumble, Phil Lacroute, Adrian V Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

[17] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 225–232. Society for Industrial and Applied Mathematics, 2002.