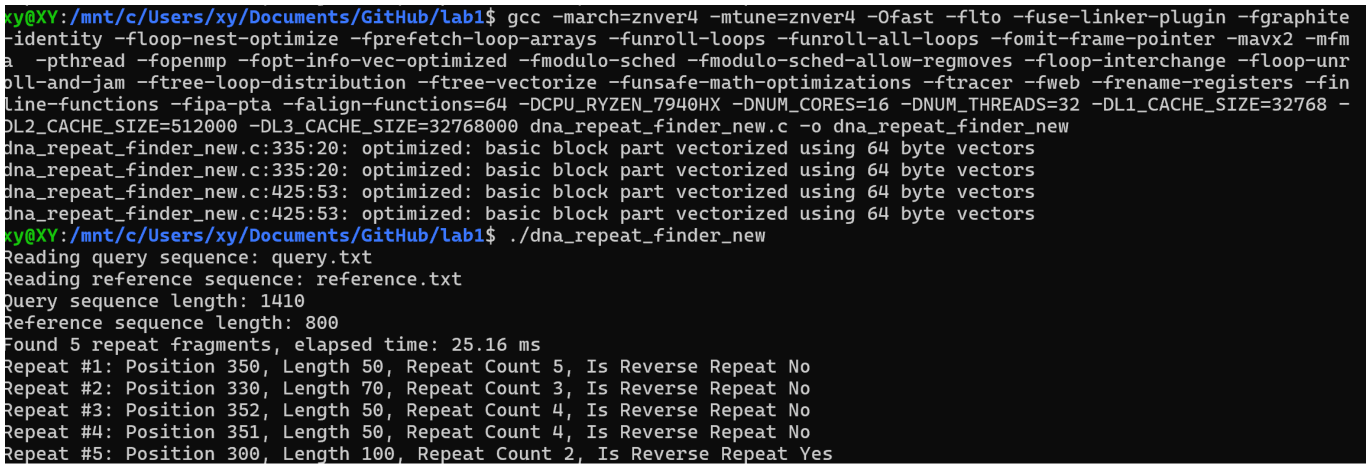
DNA重复序列查找器



```
xy@XY:/mnt/c/Users/xy/Documents/GitHub/lab1$ gcc -march=znver4 -mtune=znver4 -Ofast -flto -fuse-linker-plugin -fgraphite
-identity -floop-nest-optimize -fprefetch-loop-arrays -funroll-loops -funroll-all-loops -fomit-frame-pointer -mavx2 -mfm
a  -pthread -fopenmp -fopt-info-vec-optimized -fmodulo-sched -fmodulo-sched-allow-regmoves -floop-interchange -floop-unr
oll-and-jam -ftree-loop-distribution -ftree-vectorize -funsafe-math-optimizations -ftracer -fweb -frename-registers -fin
line-functions -fipa-pta -falign-functions=64 -DCPU_RYZEN_7940HX -DNUM_CORES=16 -DNUM_THREADS=32 -DL1_CACHE_SIZE=32768 -
DL2_CACHE_SIZE=512000 -DL3_CACHE_SIZE=32768000 dna_repeat_finder_new.c -o dna_repeat_finder_new
dna_repeat_finder_new.c:335:20: optimized: basic block part vectorized using 64 byte vectors
dna_repeat_finder_new.c:335:20: optimized: basic block part vectorized using 64 byte vectors
dna_repeat_finder_new.c:425:53: optimized: basic block part vectorized using 64 byte vectors
dna_repeat_finder_new.c:425:53: optimized: basic block part vectorized using 64 byte vectors
xy@XY:/mnt/c/Users/xy/Documents/GitHub/lab1$ ./dna_repeat_finder_new
Reading query sequence: query.txt
Reading reference sequence: reference.txt
Query sequence length: 1410
Reference sequence length: 800
Found 5 repeat fragments, elapsed time: 25.16 ms
Repeat #1: Position 350, Length 50, Repeat Count 5, Is Reverse Repeat No
Repeat #2: Position 330, Length 70, Repeat Count 3, Is Reverse Repeat No
Repeat #3: Position 352, Length 50, Repeat Count 4, Is Reverse Repeat No
Repeat #4: Position 351, Length 50, Repeat Count 4, Is Reverse Repeat No
Repeat #5: Position 300, Length 100, Repeat Count 2, Is Reverse Repeat Yes
```

编译

```
gcc -march=znver4 -mtune=znver4 -Ofast -flto -fuse-linker-plugin -fgraphite-identity -floop-nest-optimize -fprefetch-loop-arrays -funroll-loops -funroll-all-loops -fomit-frame-pointer -mavx2 -mfma  -pthread -fopenmp -fopt-info-vec-optimized -fmodulo-sched -fmodulo-sched-allow-regmoves -floop-interchange -floop-unroll-and-jam -ftree-loop-distribution -ftree-vectorize -funsafe-math-optimizations -ftracer -fweb -frename-registers -finline-functions -fipa-pta -falign-functions=64 -DCPU_RYZEN_7940HX -DNUM_CORES=16 -DNUM_THREADS=32 -DL1_CACHE_SIZE=32768 -DL2_CACHE_SIZE=512000 -DL3_CACHE_SIZE=32768000 dna_repeat_finder_new.c -o dna_repeat_finder_new
```

运行

```
 ./dna_repeat_finder_new
```

算法原理

该程序用于在查询DNA序列和参考DNA序列之间查找重复的DNA片段，包括正向重复和反向互补重复。

主要算法伪代码

```
FIND-REPEATS(query, reference)
  repeats ← 空数组
  repeat_count ← 0
  for length ← MIN_LENGTH to MAX_LENGTH
      window_positions ← 新建哈希表()

      // 为查询序列构建哈希索引
      for i ← 0 to |query| - length
```

```
        segment ← query[i..i+length-1]
        window_positions[segment] ← window_positions[segment] ∪ {i}

    // 检查参考序列中的片段
    for i ← 0 to |reference| - length
        segment ← reference[i..i+length-1]

        // 检查正向重复
        positions ← window_positions[segment]
        if |positions| ≥ 2
            groups ← FIND-CONSECUTIVE-GROUPS(positions, length)
            for each group in groups
                if repeat_count < MAX_REPEATS
                    repeats[repeat_count] ← {position: i, length: length,
                                            repeat_count: |group|, is_reverse: false,
                                            original_sequence: segment, query_position: group的起始位置}
                    repeat_count ← repeat_count + 1

        // 检查反向互补重复
        rev_comp ← GET-REVERSE-COMPLEMENT(segment)
        positions ← window_positions[rev_comp]
        if |positions| ≥ 2
            groups ← FIND-CONSECUTIVE-GROUPS(positions, length)
            for each group in groups
                if repeat_count < MAX_REPEATS
                    repeats[repeat_count] ← {position: i, length: length,
                                            repeat_count: |group|, is_reverse: true,
                                            original_sequence: segment, query_position: group的起始位置}
                    repeat_count ← repeat_count + 1

FILTER-NESTED-REPEATS(repeats, repeat_count)
SORT-REPEATS(repeats, repeat_count)
return repeats
```

查找连续组的伪代码

```
FIND-CONSECUTIVE-GROUPS(positions, length)
if |positions| < 2
    return 空数组

groups ← 空数组
current_count ← 1

for i ← 1 to |positions| - 1
    if positions[i] = positions[i-1] + length
        current_count ← current_count + 1
    else
```

```
        if current_count ≥ 2
            groups ← groups ∪ {current_count}
        current_count ← 1

if current_count ≥ 2
    groups ← groups ∪ {current_count}

return groups
```

过滤嵌套重复的伪代码

```
FILTER-NESTED-REPEATS(repeats, repeat_count)
if repeat_count ≤ 1
    return

to_remove ← 长度为repeat_count的布尔数组，初始化为false
new_count ← repeat_count

for i ← 0 to repeat_count - 1
    if to_remove[i] = false
        for j ← i + 1 to repeat_count - 1
            if to_remove[j] = false
                if repeats[i].position = repeats[j].position 且
                   repeats[i].is_reverse = repeats[j].is_reverse
                    if repeats[i].length > repeats[j].length
                        to_remove[j] ← true
                        new_count ← new_count - 1
                    else
                        to_remove[i] ← true
                        new_count ← new_count - 1
                        break

// 重新排列数组，移除被标记的重复
write ← 0
for read ← 0 to repeat_count - 1
    if to_remove[read] = false
        if write ≠ read
            repeats[write] ← repeats[read]
        write ← write + 1
    else
        释放repeats[read].original_sequence的内存

repeat_count ← new_count
```

排序重复序列的伪代码

```
QUICK-SORT-REPEATS(repeats, left, right)
if left < right
    pivot ← PARTITION(repeats, left, right)
```

```
3       QUICK-SORT-REPEATS(repeats, left, pivot - 1)
4       QUICK-SORT-REPEATS(repeats, pivot + 1, right)

PARTITION(repeats, left, right)
1   pivot_value ← repeats[right].length × repeats[right].repeat_count
2   i ← left - 1
3   for j ← left to right - 1
4       if repeats[j].length × repeats[j].repeat_count ≥ pivot_value
5           i ← i + 1
6           交换repeats[i]和repeats[j]
7   交换repeats[i + 1]和repeats[right]
8   return i + 1
```

算法复杂度分析

时间复杂度

- **总体算法**：O(L × (M + N))
    - L是检查的长度范围(MAX_LENGTH - MIN_LENGTH)
    - M是查询序列长度
    - N是参考序列长度
- **哈希表操作**：
    - 创建：O(1)
    - 插入：平均O(1)（最坏情况为O(n)，因为哈希碰撞）
    - 查找：平均O(1)（最坏情况为O(n)，因为哈希碰撞）
- **过滤嵌套重复**：$O(n^2)$, n为找到的重复数量
- **排序重复序列**：O(n log n)，使用快速排序，n为重复数量（最坏情况为$O(n^2)$)）

空间复杂度

- **哈希表**：O(Q) - 存储查询序列中所有子串的位置
- **重复数组**：O(min(MAX_REPEATS, R×L)) - 存储所有找到的重复
- **临时缓冲区**：O(L) - 用于存储片段和反向互补序列

主要优化策略

1. **哈希索引**：使用哈希表进行O(1)的序列位置查找
2. **特殊区域聚焦**：对生物学意义重要的区域进行更深入的分析
3. **基于长度的处理**：先处理较短的片段以过滤嵌套重复
4. **过滤算法**：移除嵌套重复以专注于最显著的模式