分析报告

17307130115 吴毓文

> `IDE: Visual Studio 2019`
>
> 操作系统：`windows`
>
> 编程语言:`C++`
>
> 实验结果：`INS`片段15个，`DEL`片段5个，`INV` `DUP` `TRA`片段各10个

项目结构：

> `main.cpp`：主函数入口，调用`select_sv()，read_file()`函数；
>
> `read_file.cpp`：读取序列数据以及编号并赋值给string对象`ref，sv,ID`；
>
> `write_file.cpp`：将输出写入结果文件`sv.bed`；
>
> `select_sv.cpp`：检测序列错误，并调用`write_file()`将结果写入`result.txt`文件；功能主体；
>
> `func.h`：保存函数和结构体的声明；

算法设计思路(`select_sv`)：

首先给出使用范围贯穿整个文件的字符串匹配算法`kmp():`

```
int* getNext(string p){...}
int kmp(string T, string p){...}
```

该算法时间复杂度和空间复杂度均为`O(n)`；具体实现见`select_sv.cpp`；（326行）

记原始(ID相同)DNA字符串序列为`ref`，包含错误信息的单链为`sv`，分别对`ref`和`sv`从下标`index1`和`index2`(初始化为0)开始每一个字符进行比较，相同则比较下一对字符，若不相同，则开始错误类型检测：

四种类型检测函数：

> `check_ins_dup()`:检测错误类型是否为INS或者`DUP`；
>
> `check_del()`：检测错误类型是否为`DEL`;
>
> `check_inv()`：检测错误类型是否为`INV`；
>
> `check_tra()`：检测错误类型是否为`TRA`；
>
> 以上四个函数返回值均为对应查找到的`SV`片段长度，同时根据类型相应地改变`index1`和`index2`。

> **检测`INS/DUP:`**
>
> 实现函数：`select_sv.cpp/check_ins_dup()`
>
> ```
> 
> ```

```
int check_ins_dup(string t, string p, int& ind1, int&
ind2,int&is_dup,int&pre)
{
    if (ind1 >= t.length() - 1000 || ind2 >= p.length() - 2000)
        return -1;
    string t_ref = t.substr(ind1, 20);
    string t_sv = p.substr(ind2, 1100);
    int del = kmp(t_sv, t_ref);
    if (del >= 0)
    {
        //ins? :yes
        //dup? : check
        for (int i = 0; i < 5; i++) {
            string dup = p.substr(ind2-i, del);
            string pre = p.substr(ind2-i-del, del);

            if (strcmp(dup.c_str(), pre.c_str()) == 0){
                pre = i;//偏差位数，记录起始位置需要往前移动i个单位
                is_dup = 1;
                ind2 += del;
                return del;
            }
            else if(i==4){
                ind2 += del;
                return del;
            }
        }
    }
    return -1;
}
```

思路：这里将`INS和DUP`类型一起检测是由于`DUP`可以看做INS的变种，即插入一段与前面内容相同的片段。检测的原理是将`ref`错误片段开始的长度为20的子序列`t_ref`和`sv`相同位置 开始长度为1100的子序列`t_sv`作为参数传递给`kmp()`，若匹配成功，返回值即为插入片段长度`len`。

接下来检测是否为`DUP`：取`sv`上`index2`开始长度为`len`的字符串`dup`和`index2-len`开始长度为`len`的字符串`pre`。

调用`strcmp()`比较这两个字符串，相等则错误类型`is_dup`设置为1。特别的，这里做了"偏差位数"的处理：因为重复子串`dup`也可能和ref中对应`dup`之后的序列在前几位相同，我们假设最多可能4个相同，因为此时概率为(1/4)^4=0.39%，再长的位数近乎不可能不以考虑。使用一个for循环检查是否存在偏差位数i，命中则返回对应的`POS`和`LEN`，以及偏差位数`pre(i)`并将`index2`加上LEN（`del`）以调整至下一轮检测位置。

检测`DEL:`

实现函数：`select_sv.cpp/check_del()`

思路：与INS检测相同，但是这里调用`kmp()`时，匹配字符串从`sv`中获取，目标字符串该从`ref`中获取，若命中并返回`len`，则将`index1`加上`len`，进入下一轮检查。具体代码见`select_sv.cpp`文件。

检测`INV:`

实现函数：`select_sv.cpp/check_inv()`

思路:同样是从不匹配的地方开始`(index1,index2)`，但对`ref`和`sv`同时选取从查错位开始的长度为1000的子串，对i=0→497，这里i为倒转序列(若有的话)的中间位置下标，这里使用了字符匹配函数`bool cmp(char,char)(select_sv.cpp298行)`，功能为若a,b两个字符在碱基对意义上匹配成功`（AT or GC）`返回true，否则返回false。以i为中心向两边延展比较，同时更新翻转子串的最大长度。这里也检测了子串长度的奇偶性以便返回正确的`sv`片段长度。这里给出主要代码，具体代码见`select_sv.cpp`文件：

```
int check_inv(string t, string p, int& ind1, int& ind2)
{
    if (ind1 >= t.length() - 1000 || ind2 >= p.length() - 1000)
        return -1;
    int len = 0;
    int ans = 0;
    bool even = false;
    string re = t.substr(ind1, 1000);
    string s = p.substr(ind2, 1000);
    const char* ref = re.c_str();
    const char* sv = s.c_str();
    while (len <= 497)
    {
        if (cmp(ref[len], sv[len]) )
        {
            for (int i = 1; i <= len; i++)
            {
                if (cmp(ref[len + i], sv[len - i]) == false ||
                    cmp(ref[len - i], sv[len + i]) == false)
                {
                    break;
                }
                if (i == len)
                    ans = len;
            }
        }
        len++;
    }
    len = 0;
 //偶数长度检测，代码长了不贴，思路和奇数位检测类似，并标记偶数标记位even
 //返回对应的长度
}
```

检测`TRA:`

实现函数：`select_sv.cpp/check_c_tra()`

```
int check_c_tra(int id, string t, string p, int& ind1, int& ind2,
vector<info_c_tra>& tra) {
    int len = 0;
    int max = 0;
    string subt = t.substr(ind1, 1010);
    string subp = p.substr(ind2, 1010);
    while (len < 1000) {
        if (subt[len] != subp[len]) {
            len++;
        }
```

```
        else {
            while (subt[len] == subp[len]) {
                max++;
                len++;
                if (max == 10)
                    break;
            }
            if (max < 10) {
                max = 0;
            }
            else {//find the tra
                tra.push_back({ id,ind1,len - 10,subt.substr(0,len -
10),subp.substr(0,len - 10) });
                ind1 += len - 10;
                ind2 += len - 10;
                return len - 10;
            }
        }
    }
    return -1;
}
```

思路：从查错位开始两边字符逐个比较直至`ref`和`sv`连续10个字符相同，即两边字符对上相同的片段，然后将`ref`上之前不相同的字符串存储在结构`info_c_tra`中（声明在`func.h`）,此时为待确认，需要在检测完所有`DNA`单链后对所有的可能的单链信息一一匹配，匹配成功说明该两个可能`tra`为互换的一对。

```
struct info_c_tra {//store the possible tras and check them later
    int id;
    int start;
    int len;
    string s1;
    string s2;
};
```

找完数据所有单链中可能为`tra`的片段后，两两比对，确认是否为互换的一对；实现该功能的函数为

`check_tra()`：算法复杂度`O(n^4)`

```
void check_tra(vector<info_tra> &t, vector<vector<info_c_tra>> tra_cop_All)
{

    //combine the exchanged tra
    for (int i = 0; i < tra_cop_All.size() - 1; i++)
    {
        for (int j = 0; j < tra_cop_All[i].size(); j++)
        {
            for (int k = i + 1; k < tra_cop_All.size(); k++)
            {
                for (int l = 0; l < tra_cop_All[k].size(); l++)
                {
                    if (tra_cop_All[i][j].len == tra_cop_All[k][l].len)
                        t.push_back({i,tra_cop_All[i]
[j].start,tra_cop_All[i][j].start + tra_cop_All[i][j].len, k, tra_cop_All[k]
[l].start,
```

```
                        tra_cop_All[k][l].start + tra_cop_All[k][l].len});
                }
            }
        }
    }
}
```

至此就找完了所有的 `sv` 片段，执行程序，读取数据文件，将结果写至 `sv.bed` 文件，写文件的函数 `write_file.cpp`，结果文件见 `sv.bed`。