**深 圳 大 学 实 验 报 告**

**课程名称： 算法设计与分析**

**实验名称： 动态规划—代码查重问题**

**学院： 计算机与软件学院 专业： 软件工程**

**报告人： 蔡晓鑫 学号： 2017151019 班级： 01**

**同组人： 无**

**指导教师： 杨煊**

**实验时间： 2019年4月28日——2019年5月27日**

**实验报告提交时间： 2019年5月15日星期三**

**教务处制**

**一、实验目的：**

（1） 掌握动态规划算法设计思想。

（2） 掌握代码查重问题的动态规划解法。

**二、内容：**

1、给定两个代码行长度分别为*m*与*n*的代码源文件“*A*.cpp”与“*B*.cpp”，其中“*A*.cpp”为模板代码文件，“*B*.cpp”为测试代码文件。如果文件“*A*.cpp”在第*i*行代码与文件“*B*.cpp”在第*j*行代码相似程度超过事先给定的重复比率阈值*r*, 则认定文件“*B*.cpp”的第*j*行与文件“*A*.cpp”的第*i*行代码相重。其中，这两行代码的相似度S left parenthesis i comma j right parenthesis可以计算为：

S left parenthesis i comma j right parenthesis equals fraction numerator L C S left parenthesis A left parenthesis i right parenthesis comma B left parenthesis j right parenthesis right parenthesis over denominator m i n left parenthesis L subscript i left parenthesis A right parenthesis comma L subscript j left parenthesis B right parenthesis right parenthesis end fraction          ，

其中*A*(*i*)与*B*(*j*)为文件“*A*.cpp”与“*B*.cpp”分别在其第*i*行与第*j*行的代码，L C S left parenthesis A left parenthesis i right parenthesis comma B left parenthesis j right parenthesis right parenthesis为文件“*A*.cpp”在其第*i*行代码与文件“*B*.cpp”第*j*行代码的最长相同子代码模块，L subscript i left parenthesis A right parenthesis与L subscript j left parenthesis B right parenthesis为文件“*A*.cpp”与“*B*.cpp”分别在第*i*行与第*j*行代码的长度。根据，计算

D left parenthesis i comma j right parenthesis equals open curly brackets table attributes columnalign left columnspacing 1.4ex end attributes row 1 cell i f S left parenthesis i comma j right parenthesis greater than r end cell row 0 cell o t h e r w i s e end cell end table close

其中，D left parenthesis i comma j right parenthesis=1表示文件“*B*.cpp”的第*j*行与文件“*A*.cpp”的第*i*行代码重复。

其相关定义如下：

**定义1：行代码模块及行代码模块长度**

行代码模块是由被排成一行的元素（包括字符、标点符号、空格）构成。在行代码模块中，每个元素不是在其他元素之前，就是在其他元素之后。

行代码模块长度*k*是序列中元素的个数。

例如：行代码模块*X* = (int *a*, *b*)，其中第二个元素*n*在第一个元素*i*之后，并在第三个元素*t*之前。其*X*的行代码模块长度*k*=8（注意括弧不是代码）。

**定义2：子代码模块**

子代码模块是从给定行代码模块中去除一些元素，不改变其他元素之间相对位置而得到。

    例如：(in), (int), (nt *a*)等都是行代码模块*X*的子代码模块。

**定义3：相同子代码模块**

任意给定行代码模块 *X*, *Y*，如果另一代码模块*Z*是所有行代码模块 *X*, *Y*的子代码模块，则称代码模块*Z*是给定行代码模块*X*, *Y*的相同子代码模块。

**定义4：最长相同子代码模块**

相同子代码模块中长度最长的子代码模块称为最长相同子代码模块。

问题：根据给定重复比率阈值*r*与模板代码文件“*A*.cpp”*，*计算测试代码文件“*B*.cpp”的最多代码重复行数？

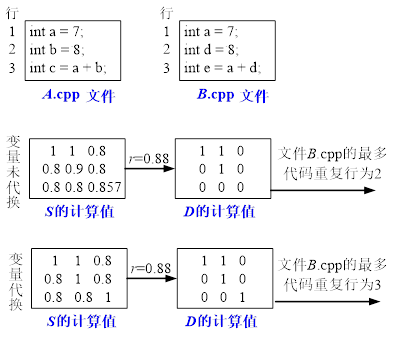
1）  设计动态规划算法求解文件“*A*.cpp”在第*i* 行代码与文件“*B*.cpp”在第*j*行代码的最长相同子代码模块问题，并写出求得最长相同子代码模块的递推公式。

2）对于任意给定的 *i*, *j*, 求出*S*(*i*,*j*)。根据给定重复比率阈值*r，*求出*D*(*i*,*j*)。

3）根据求出的D(i,j)，利用动态规划算法求出测试代码文件“B.cpp”的最多代码重复行数？写出求得最多行重复模板代码的递推公式。

4）设计变量名代换的不同程序代码查重（此问题为加分选项）。

求解问题例子如下所示：



**三、实验要求**

1. 在blackboard提交电子版实验报告。

2. 源代码作为实验报告附件上传。

3. 在实验完成之后，将进行一次PPT介绍。

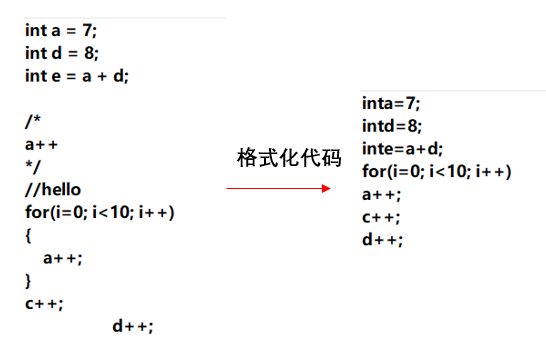
**四、实验成绩**

实验成绩的给分标准是实验报告50%，PPT汇报50%。

1. **代码预处理**
2. **思路：**

由于每个人的代码书写习惯不同，不同IDE对代码格式化的策略不同，例如VS2017是自动为代码通过空格美化的。空格、换行、tab、注释、花括号是抄袭代码中可人为添加的干扰查重因素，除去这些特殊标记有助于更好对比两个代码文件在结构及文本上的相似度

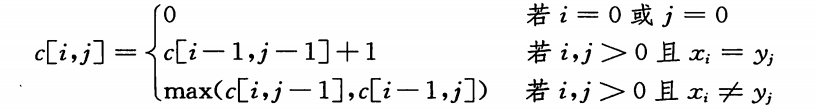
1. **预处理效果**



1. **求解相同最长子模块问题**
2. **最长公共子序列问题（LCS问题）**
3. 概念：

给定两个序列 X = (x1,x2, ..., xm)和Y = (y1,y2, ..., yn),如果Z既是序列 X 的子序列，又是序列Y的子序列，而且Z是这些公共子序列中最长的，则称Z为X和Y的最长公共子序列。

1. 递推式求解：利用LCS的最优子结构特征
2. 如果x[i] = y[j]，可得z[k]=x[i]=y[j]，并且Zk-1是Xi-1和Yj-1的一个LCS
3. 如果x[i]≠y[j]，可得z[k]≠x[i]，可得Zk-1是Xi-1和Yj的一个LCS
4. 如果x[i]≠y[j]，可得z[k]≠y[j]，可得Zk-1是Xi和Yj-1的一个LCS



1. 应用：借助LCS，可以求得文件“A.cpp”在第i 行代码与文件“B.cpp”在第j行代码在文本上的重复率，当重复率大于某个设定的阈值时，认定为这两行代码存在抄袭情况，相似度计算：

S left parenthesis i comma j right parenthesis equals fraction numerator L C S left parenthesis A left parenthesis i right parenthesis comma B left parenthesis j right parenthesis right parenthesis over denominator m i n left parenthesis L subscript i left parenthesis A right parenthesis comma L subscript j left parenthesis B right parenthesis right parenthesis end fraction

1. 伪代码：

**LCS(X ,Y)**

**row = Length(X)**

**column = Length(Y)**

**for i = 1 to row**

**for j = 1 to column**

**if X[i-1] = Y[j-1]**

**DP[i][j] = DP[i-1][j-1] + 1**

**else**

**DP[i][j] = MAX(DP[i-1][j] , DP[i][j-1]**

**return DP[i][j]**

1. **Levenshtein距离**
2. 概念：

Levenshtein距离是一种计算两个字符串间的差异程度的字符串度量（string metric）。我们可以认为Levenshtein距离就是从一个字符串修改到另一个字符串时，其中编辑单个字符（比如替换、插入、删除）所需要的最少次数。

1. 递推式求解：利用最优子结构特征
2. 删除：如果将Xi-1修改成Yj需要的操作数为ops，可得只需删除x[i]便可将Xi修改为Yj，所需操作数为ops+1
3. 插入：如果将Xi修改成Yj-1需要的操作数为ops，可得只需将y[i]插入到x[i]和x[i+1]之间便可以将Xi修改为Yj，所需操作数为ops+1
4. 替换：如果将Xi-1修改为Yj-1需要的操作数为ops，那么只需把x[i]替换成y[j]便可将Xi修改为Yj，所需操作数为ops+1
5. 应用：通过比较两行代码的修改程度，从而计算相似度，也是检测两行代码之间是否具有抄袭情况的一种方法，相似度计算：
6. 伪代码：  
   **Levenshtein(X,Y)**

**row = Length(X)**

**column = Length(Y)**

**for i = 1 to row**

**for j = 1 to column**

**if X[i-1] = Y[j-1]**

**op = 0**

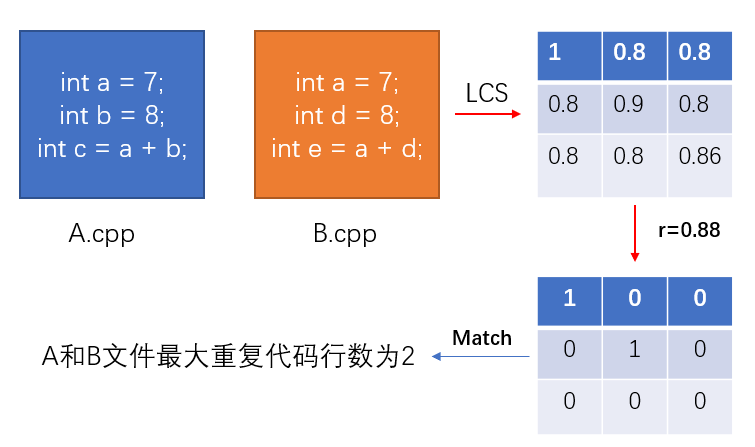
**else op = 1**

**DP[i][j] = MIN(DP[i-1][j-1]+op,DP[i][j-1]+1,DP[i-1][j]+1)**

**return DP[i][j]**

1. **求解最多重复代码行数**
2. **求解思路**

通过求解任意两行代码的相似度，设置阈值判定是否抄袭，就可以得到最终的相似度矩阵，反映出两个代码文件的相似情况。然后通过文本匹配方法根据相似度矩阵计算出两个代码文件中存在的最多重复代码行数，其中，查重率计算：



1. **将匹配问题转化为最长公共子序列问题**
2. 相似度矩阵反映的是两个代码文件之间的抄袭情况，通过LCS算法可以求解两个代码文件的序列相似度
3. 伪代码：  
   **LCS\_Match(Matrix,row,column)**

**for i = 1 to row**

**for j = 1 to column**

**if Matrix[i][j]=1**

**DP[i][j] = DP[i-1][j-1] + 1**

**else**

**DP[i][j] = MAX(DP[i-1][j] , DP[i][j-1]**

**return DP[i][j]**

1. **将匹配问题转化为Levenshtein文本距离问题**
2. 通过计算代码文件A通过替换、删除、插入方式转化为代码文件B所需的最小编辑次数，从而反映两个文件之间的相似度，相似度的计算：
3. 伪代码

**Levenshtein\_Match(Matrix,row,column)**

**for i = 1 to row**

**for j = 1 to column**

**if Matrix[i][j]=1**

**op = 0**

**else op = 1**

**DP[i][j] = MIN(DP[i-1][j-1]+op,DP[i][j-1]+1,DP[i-1][j]+1)**

**return DP[i][j]**

1. **对比两种匹配算法**
2. 采用LCS算法求解最长代码子模块，r=0.88，测试程序A0、B0的查重率

已知A0和B0是抄袭代码，在控制变量的条件下，LCS的文本匹配效果明显优于Levenshtein

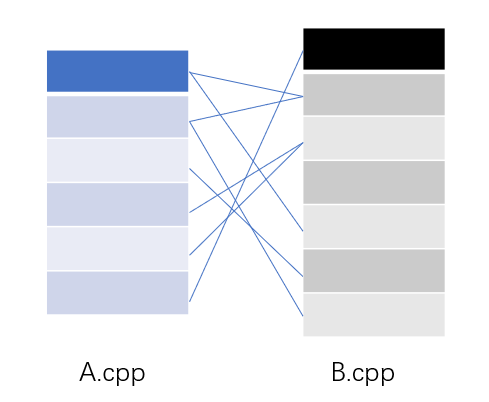
1. **r值调整及文本匹配算法改进**
2. **r值调整**

阈值r是检测两行代码是否存在抄袭情况的下界，所以r的控制十分重要，太低会导致文本过度匹配，太高会使出现很大的匹配误差。

1. **在使用LCS求解最多代码重复行数的前提下，分别测试A1，B1代码使用LCS，Levenshtein求解最长代码子模块，在r值为0.80-0.95时查重率的变化情况**
2. 在已知A1，B1存在抄袭的情况下，Levenshtein算法的查重效果逊于LCS，从算法策略上，Levenshtein更注重从修改的角度去推测两者的相似度，例如a和array，对于Levenshtein来说两者达到近似需要的修改成本很高，但是对于LCS来说这两者是没有差别的。在不考虑变量代换及结合代码抄袭习惯，LCS更加适合用于代码查重。
3. 由图可得，随着r值的增大，查重率会缓慢降低，在r=0.9时趋于平缓，符合预期猜想。趋于平缓表示具有抄袭嫌疑的代码的相似度界限可模糊确定，**由此，可以选择r=0.9作为不进行变量代换处理的代码查重阈值。**
4. **文本匹配算法的改进**
5. 在（三）中我采用LCS和levenshtein策略求解最多行代码重复问题，在基于B0，A0代码位置相对一致的条件下得到较好的查重效果，但分析算法的策略，发现这两种算法并不能解决代码位置变动，例如函数位置互换，打乱变量声明位置等避免查重的技巧。
6. 将A1代码函数顺序打乱，在r=0.9的情况下与B1进行查重比较

结论：符合预期猜想，当代码模块顺序打乱后，这两种代码行匹配策略效果很差，非常容易被避免查重的技巧欺骗。

1. 转化为二分图最大匹配问题：  
   下图为代码查重的结构特征，A中每行代码都与B中的若干行代码具有抄袭嫌疑，那求解最多重复代码行数问题就可以转化为让A的代码尽可以一对一匹配到B中，这种策略称为二分图最大匹配，下面将利用匈牙利算法、网络流算法求解该问题。



1. **匈牙利（****Hungarian）算法**
2. 算法简介：

匈牙利算法是由匈牙利数学家Edmonds于1965年提出，因而得名。匈牙利算法是基于Hall定理中充分性证明的思想，它是部图匹配最常见的算法，该算法的核心就是寻找增广路径，它是一种用增广路径求二分图最大匹配的算法。

1. 伪代码

**//N:左侧结点数，M：右侧结点数**

**//Visit:数组，标记右侧访问节点**

**//Match:数组，记录两侧的匹配关系**

**//Matrix:匹配关系邻接表**

**Find(x)**

**for u in Matrix[x]**

**if Visit[u] = False**

**Visit[u] = True**

**if Match[u] no match or Find(u)**

**Match[u] = x**

**return True**

**return False**

**Hungarain()**

**cnt = 0**

**for x = 1 to N**

**Init Visit with False**

**if Find(x)**

**cnt = cnt + 1**

**return cnt**

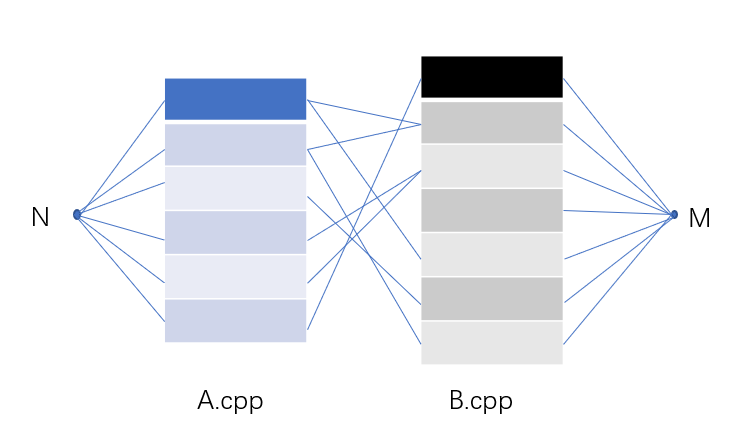
1. 算法分析
2. 时间复杂度：-V顶点数、E边数
3. 空间复杂度：
4. **网络流-最大流问题(Dinic)**
5. 网络流-最大流介绍：

形象地说，在一张复杂的自来水网中，每根管道都有相应的最大通量，求解从n到m，m节点所能得到的最大流量

1. 二分图最大匹配转换成最大流问题

增加两个端点N和M，将两个分图的所有节点分别连向N和M，设置每条边的最大通量为1，等价于每条边只有通或不通的状态，求解N到M，M节点获得的最大流量，该流量等于最大代码行匹配数

1. 算法复杂度：
2. -V顶点数、E边数



1. 伪代码

**Dinic()**

**cnt = 0**

**while BFS:为每一个点分配深度**

**while now=DFS:寻找增广路**

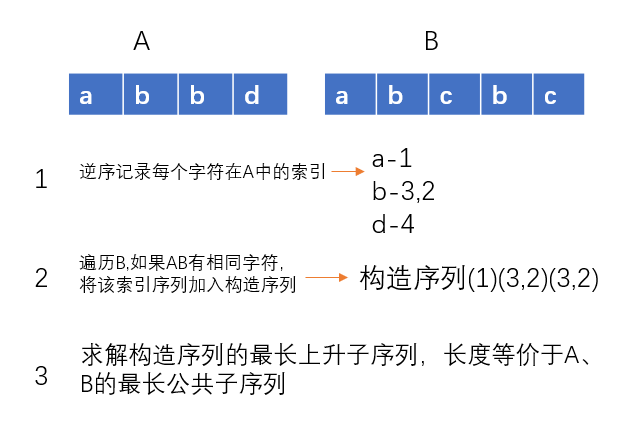
**cnt = cnt + now**

**return cnt**

1. **算法测试**
2. 在3.基础上测试Hungarian和Dinic的匹配效果
3. Hungarian和Dinic算法查同一份代码查重率相同，并且两个算法的最终目的一致，验证了两个算法的正确性。
4. 比较代码打乱顺序前，在已知A1B1是抄袭的情况下，Hungarian和Dinic算法查重的效果优于LCS和Levenshtein，原因也是这两种算法可以解决以交换部分代码逃避查重的问题
5. 比较代码打乱顺序后，Hungarian和Dinic算法的查重率与打乱顺序前一致，完美解决代码位置改变引起查重失效的问题
6. **代码查重优化**
7. **时间优化**
8. 在实验过程中，发现代码运行时间略长，而查重算法分为相似矩阵求解及代码行匹配及代码行匹配两部分，分别计算两部分运行时间：

可以明显看到，算法的运行时间主要体现在生成相似矩阵上，因为生成相似矩阵需要对每两行代码求解一次LCS或Levenshtein,鉴于前面的算法比较结果，Levenshtein效果不佳，于是主要是对LCS进行优化

1. LCS转LIS（最长上升子序列）问题



1. 算法分析：
2. 空间复杂度：
3. 时间复杂度：-遍历n，二分查找logn
4. 伪代码：

**//map(key,vector):记录X中每个字符的索引**

**//Array:构造数组**

**//Res:最长上升子序列，cnt:序列长度**

**LCS\_to\_LIS(X,Y)**

**map(key,vector)**

**for i = Length(X) downto 0**

**i ADD TO map[X[i]]**

**for j = 1 to Length(Y)**

**if Y[j] in map**

**Array = Array + map[Y[j]]**

**for i = 1 to Length(Array)**

**if Array[i] > Res[cnt]**

**Res[++cnt] = Array[i]**

**else**

**idx = lower\_bound(Array,Array[i])**

**Res[idx] = Array[i]**

**return cnt + 1**

1. 效率分析

在和1）相同条件下对比两种方法的运行时间：运行效率上大幅提升

1. **空间优化：针对求解最大匹配行数问题**
2. LCS空间优化
3. LCS的空间复杂度为O(MN),对于代码查重这个问题来说，并不需要一个动态规划矩阵存储全部状态，因为当前状态dp[i][j]只取决于dp[i-1][j-1],dp[i][j-1],dp[i-1][j]三个值，通过记录上一次dp[i-1][j-1]的值，可以将复杂度降至O(min(M,N))
4. 伪代码

**LCS\_optimize(matrix,row,column)**

**for i = 1 to row**

**record = 0**

**for j = 1 to column**

**tmp = dp[j]**

**if matrix[i-1][j-1] = 0**

**dp[j] = record + 1**

**else**

**dp[j] = MAX(dp[j-1],dp[j])**

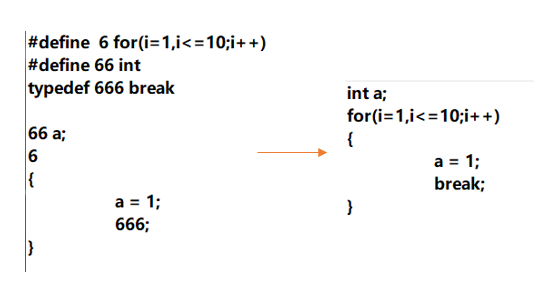
**record = tmp**

**return dp[column]**

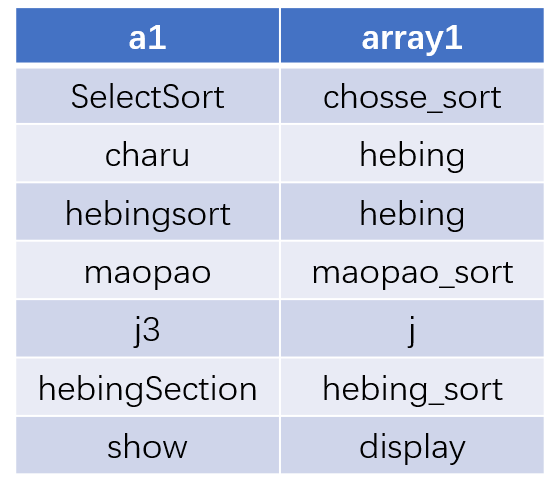
1. **针对Hungarian和Dinic的空间优化**
2. 对于这两个算法来说，采用邻接矩阵不仅可以降低空间复杂度，而且可以提高算法运行效率。在求解相似矩阵时，视大于阈值r的点为i，j通路，加入邻接矩阵，而对于小于阈值r的点，可弃
3. 空间复杂度分析

O(V+E)-V定点数、E边数

1. **变量代换**
2. **代换定义区变量**
3. 使用#define,typedef对某些语句、变量、常量进行同义转换，是避免查重的一种策略，并且是简单变量代换无法解决的问题，所以需要将定义区的原语句还原到代码当中，并删除定义区
4. 代换效果



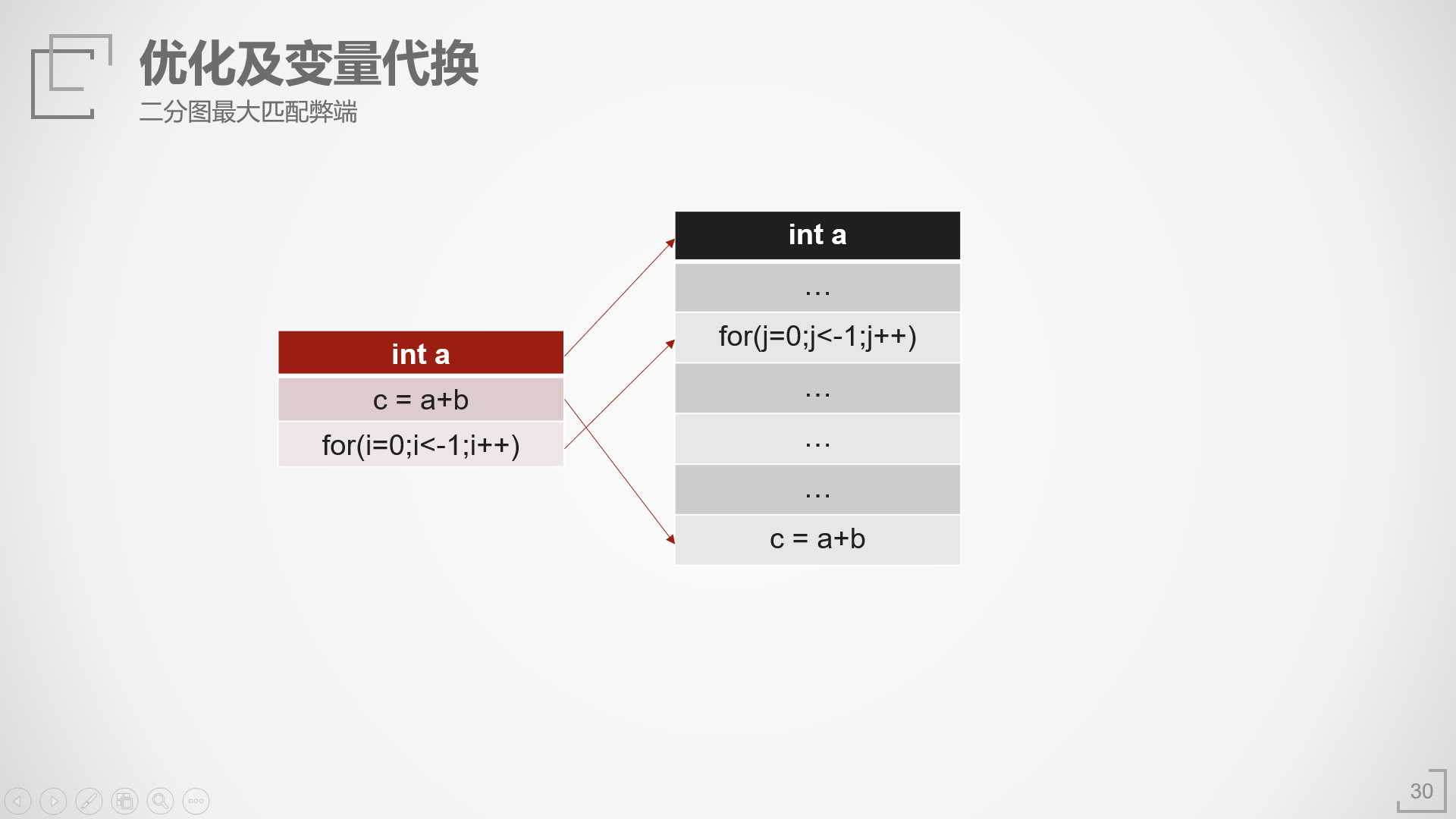
1. **变量名、函数名代换**
2. 策略：
3. 分别统计两个代码文件中**变量、结构体名、类名、函数名**的**类型、出现次数、变量名**
4. 计算变量名文本相似度S，采用Levenshtein（LCS经测试会过度拟合）
5. 计算变量名出现次数相似度
6. 计算变量类型相似度V，类型相同为1，相近为0.5，否则为0
7. 对三个影响因素进行加权得到总加权值，认为代码AB中变量间Weight值最大的两个变量是近似变量，并且将A中该变量替换成A在B中的近似变量
8. 经过不断调参，对三个参数分别设置为 **1.5，1，1.5**可得到最佳的匹配准确性
9. 对A2、B2进行变量代换匹配的部分结果，准确性非常高



1. **变量代换效果测试：r=0.9，LCS求解最长子模块**
2. 测试A0，B0
3. 测试A1、B1
4. 测试A2、B2
5. 运行结果分析：

在调参后，在控制其他条件不变的情况下测试变量代换效果，可以发现查重率均有较高的提升，说明变量代换效果良好

1. **算法总结**
2. 通过前面的数据分析，可以得到，使用LCS的优化算法求解最长子模块问题，在r=0.9的情况下构造相似矩阵，使用匈牙利算法或dinic算法进行代码行数最大匹配，可以得到在时间效率，查重效率上最优的查重策略
3. 二分图最大匹配查重效果良好的同时，可能会出现部分代码混乱组搭而导致查重率虚高，如图，在A中处于同一结构的代码，可能会被分配到B中的不同位置，但本身代码互不相关。所以可以通过r值调高的方法拉高相似代码匹配的门槛，但不能彻底解决该问题。



1. 变量名代换是充分利用抄袭心理去推测出两个文件见的变量映射关系，并且需要倾向数据类似相似、出现次数相似去查找变量相似关系。同时针对不同语言的使用习惯，比如C语言，预定义的使用很频繁，所以需要将预定义的变量、常量等先替换到代码正文。
2. 动态规划一般都会用到二维以上的空间，所以对于大数据运行的情况，需要根据实际的递推式判断实际规划时所需的空间，从而对dp空间进行降维。
3. 对于图的问题，尽可能使用邻接表可以很好地降低稀疏图的存储空间，同时也可以提高算法运行效率。