# 附录实验报告

离散实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 组员姓名 | 陈新宇，常瀚堃，田皓莘，王云峰，马梓灿，谢峰 | 实验时间 | 2024年12月1日 |
| 实验题目 | 等价类的应用-有限状态机 | | |
| 实验过程中遇到的主要问题 | Q：在这个程序中，如何表示归类的中止  A：在Hopcroft算法的程序实现中，归类的中止条件是指算法已经不能再进一步分割状态集合，即所有的状态集合都已经是等价类，不能再被进一步细化。在程序中，这个中止条件通常是通过检查状态集合的数量是否在迭代过程中增加来判断的。具体来说，在程序中的Hopcroft函数里，会有一个循环，这个循环会不断地对每个输入字符和每个状态集合调用split函数。每次调用split函数都可能增加新的等价状态集合到eq中。如果在某次迭代中，没有新的等价状态集合被添加，即t（状态集合的数量）的值没有增加，那么算法就可以停止了，因为这意味着所有的状态集合都已经是最小的等价类，不能再被进一步分割。  程序片段：  void Hopcroft() {  // ... 初始化代码 ...  while (1) {  for (i = 0; i < zf; i++) { // 对于每个输入字符  l = t; // 记录当前集合的数量  for (j = 0; j < l; j++)  split(j, zfj[i]); // 分割状态集合  }  if (l == t) break; // 如果集合数量没有变化，则算法结束  }  } | | |
| 实验小结 | 通过实验，深入理解了Hopcroft算法的工作原理，包括状态集合的分类和分割，以及如何通过等价类来最小化DFA。 | | |
| 数据结构  （自定义数据类型） | zf：整型变量，用于存储输入字母表（即DFA的状态转移可以响应的输入符号集合）的大小。这个大小告诉我们有多少不同的输入符号需要考虑。  zfj：字符数组，具体存储了输入字母表中的每个字符。由于zf存储了字母表的大小，zfj数组的大小被设置为101，以确保有足够的空间存储所有的字符，包括可能的额外字符。  V ：Node结构体的向量，用于存储DFA的所有节点。每个Node对象包含两个属性：s（状态编号）和flag（标记）。s用于标识DFA中的状态编号，而flag是一个布尔值，用来标记该状态是否为接受状态（如果flag为true，则该状态是接受状态）。  Edge：二维数组，其中每个元素都是一个Edge结构体的向量。这个数组以状态编号为索引，每个元素存储了从该状态出发的所有边。每条边由Edge结构体表示，包含两个属性：to（边的目标状态）和c（边对应的字符）。这样，edge数组就详细描述了DFA的状态转移关系。  v：整型变量，存储DFA中节点（状态）的数量。  e：整型变量，存储DFA中边的数量。  t：整型变量，初始时设置为2，表示算法开始时有两个状态集合：N（非接受状态集合）和A（接受状态集合）。随着算法的进行，t的值会增加，以反映新发现的等价状态集合的数量。  eq：整型向量，用于存储等价关系集合。每个元素是一个整数，表示一个状态集合。这些集合将最终形成最小DFA的状态。eq中的每个整数都是一个位掩码，用于表示属于同一集合的所有状态。  Ed： Edge\_dfa结构体的集合，用于存储最小化DFA的边集。Edge\_dfa结构体包含三个属性：from（边的起始状态）、to（边的目标状态）和c（边对应的字符）。这个集合最终将包含最小DFA的所有状态转移。  p：整型数组，用于记录每个状态属于哪个集合。由于eq中的每个元素都是一个状态集合，p数组的每个元素指向eq中的一个索引，表示相应状态所属的集合。 | | |
| 主要算法  （或算法说明） | **第一步：初始化状态集合**  程序首先通过读取输入来初始化DFA的状态集合。这包括状态的数量、每个状态是否为接受状态、以及状态之间的转移关系。这些信息存储在全局变量V（DFA节点集）和edge（DFA边集）中。  **第二步：状态分类**  当我们考虑状态集s中的状态对输入字符c的响应时，此步骤的目标是对吃了字符c之后到达的状态进行分类。核心在于找出状态转移的相似性和差异性，以此来判断哪些状态可能属于同一等价类。 我们使用了一种基于状态集的分类方法，即根据状态在转移后所属的现有状态集进行分类。假设整个状态空间被划分为多个状态集，对于状态集s中的每个状态，当它们吃了字符c并发生状态转移后，会到达不同的状态。我们需要明确这些到达状态所属的状态集。这里使用a和 b数组辅助完成这个任务。a数组存储的是吃字符c后到达的状态集，而b数组存储的是这些到达状态在吃字符c前的原状态集。 通过这种分类方式，我们能够观察到对于同一字符c，哪些状态会转移到相同的状态集。这是很重要的，因为如果不同状态在接受相同字符时最终到达的状态集相同，它们可能具有相似的行为，更可能属于同一等价类；反之，若它们到达不同的状态集，可能需要将它们区分开。 具体来说，当我们遍历状态集s中的状态并考虑其对字符c的转移时，将到达的状态根据其所属的状态集进行归类。这样做的好处是可以从宏观上观察到状态转移的整体趋势，即哪些部分的状态在接受c时表现出相似的行为，为后续的集合分割提供依据。这有助于我们发现状态集中可能隐藏的等价类的边界，将具有相似转移行为的状态聚集在一起，为状态集的细化做准备。  在split函数中，程序实现了状态分类的功能。对于给定的状态集合x和输入字符c，程序会遍历集合中的状态，并跟踪它们在吃掉字符c后到达的状态。这通过遍历edge数组来完成，其中存储了每个状态的出边信息。程序使用两个辅助数组a和b来记录状态转移后到达的状态集和这些状态在转移前的原状态集。   * a数组：存储吃掉字符c后到达的状态集。 * b数组：存储这些到达状态在吃字符c前的原状态集。   通过这种分类，程序能够识别出对于同一字符c，哪些状态会转移到相同的状态集，哪些会转移到不同的状态集。  **第三步：分割状态集合**  若在第 2 步中发现对于字符c，状态集s中的部分状态在吃了c之后到达的状态集与其他部分不同，那么这些状态不应属于同一等价类，需要将它们从原状态集s中分割出来。 这一步的操作依据是，如果我们在第2步的分类结果中发现某些状态在接受字符c后的转移行为与其他状态不一致（表现为到达不同的状态集），那么它们应该被区分开。通过添加新的状态集，我们可以将这些具有不同行为的状态从原集合中分离。 具体实现上，我们会根据b数组中的信息来判断是否需要分割。如果b[i]不为零且不等于原集合eq[x]，说明存在一组状态在吃了字符c后到达了不同的状态集，需要将其分离。将b[i]作为一个新的状态集添加到eq中，对于其中的状态更新其所属的状态集编号，同时从原集合 eq[x]中移除这组状态。 这种分割操作是不断迭代的，通过对不同字符和不同状态集的多次处理，我们可以逐步细化状态集的划分。每次分割都是基于状态对不同字符的转移行为，随着分割的进行，我们不断将具有不同转移行为的状态分离，从而使等价类更加精确。 例如，对于某个状态集，在处理不同字符时，可能会发现某些状态在某些字符输入下的转移行为与其他状态不同，通过这种分割操作，这些状态将被分离到新的状态集中。最终，经过多次迭代和对不同字符的处理，状态集将被不断细化，最终形成等价类，实现 DFA 的最小化。 这种分割操作的原理确保了状态集根据状态转移行为被精确划分，使具有相同转移行为的状态保持在同一等价类中，而具有不同转移行为的状态被分离，最终达到 DFA 最小化的目的。通过不断细化状态集，我们能够将 DFA 中的冗余状态去除，得到一个状态数最少但功能等价的最小化 DFA，提高 DFA 的性能和简洁性。  具体来讲程序会检查b数组中的每个元素，如果发现某个状态集合i在吃掉字符c后与原集合eq[x]不同，且不为零，那么程序会将这个状态集合i作为一个新的状态集合添加到等价关系集合eq中，并更新状态集合映射p。  这个过程是迭代的，程序会不断地对每个输入字符和每个状态集合重复这一过程，直到不能再分割为止，即当迭代过程中集合的数量t不再增加时。  **第四步：输出等价类和最小化DFA**  在main函数的最后部分，程序输出了等价状态集合的信息，并根据等价类合并了DFA的边集，生成了最小化DFA的边集Ed。程序还使用了一个type数组来记录每种状态转移是否已经输出过，以避免重复输出相同的转移。 | | |