《程序设计艺术与方法》课程实验报告

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | 实验二 搜索算法的实现 | | | | | | |
| 姓 名 | 党存远 |  | 计算机与信息学院 | 班 级 | 物联网工程22-2 | 学 号 | 2022217587 |
| 实验日期 | 2023.6.6 | | 指导教师 | 曹力 | | 成 绩 |  |
| **一、实验目的和要求**  1.掌握宽度优先搜索算法。  2.掌握深度优先搜索算法。 | | | | | | | |
| **二、实验预习内容**  1．预习实验指导书中的搜索算法部分内容  2. 理解什么是深度优先搜索和广度优先搜索。 | | | | | | | |
| **三、实验项目摘要**  1. 八皇后问题： 在一个国际象棋棋盘上放八个皇后，使得任何两个皇后之间不相互攻击，求出所有 的布棋方法。上机运行并检验结果。 思考：将此题推广到 N 皇后的情况，检验在 N 比较大的情况下，比方说 N=16 的时 候，你的程序能否快速的求出结果，如果不能，思考有什么方法能够优化算法。  2. 倒水问题：给定2 个没有刻度容器，对于任意给定的容积，求出如何只用两个瓶装出L 升  的水，如果可以，输出步骤，如果不可以，请输出No Solution。 | | | | | | | |
| **四、实验结果与分析（复杂度分析、伪代码、测试样例及相关说明）**  **1、**八皇后问题： 在一个国际象棋棋盘上放八个皇后，使得任何两个皇后之间不相互攻击，求出所有的布棋方法。上机运行并检验结果。思考：将此题推广到 N 皇后的情况，检验在 N 比较大的情况下，比方说 N=16 的时 候，你的程序能否快速的求出结果，如果不能，思考有什么方法能够优化算法。  根据题目分析可知，在8\*8的棋盘上需要摆放8个皇后并且使其之间不可进行相互攻击，即任意两个皇后不可再同一行同一列或者同一斜线上  解决思路:  首先考虑4皇后，在4\*4的棋盘上放置4个皇后，满足上述规则。  声明一个表示棋盘的4\*4数组board,在棋盘中用0，1填充棋盘。0位置表示此位置不可放置皇后，1位置表示该位置可以放置皇后。初始将数组全部初始化为1【即每个位置均可以存放皇后】，每当将一个皇后放在位置(i,j)之后，将board[i][j]位置设置为0，同时，通过一个函数找到该位置所在的行，该位置所在的列，以及该位置所在的斜线，将第i行和第j列以及与(i，j)在同一条斜线上的所有位置。  算法的时间复杂度: O(n!)，其中n为棋盘大小(n\*n)。这是因为在每一行中，都需要枚举每一列，判断是否可以放置皇后。由于每一行都有n种可能性，所以总共有n种可能性  **伪代码：**  // 定义一个叫做dfs的函数，它接受一个整数参数叫做row  函数 dfs(row)  // 检查row是否大于n，这意味着我们找到了一个解  如果 row > n 那么  // 增加全局变量ans的值，它记录了解的个数  ans = ans + 1  // 从函数中返回  返回  结束如果  // 从1到n循环，其中n是棋盘的大小  对于 i = 1 到 n 做  // 检查第i列和通过(row, i)的两条对角线是否没有被任何皇后占用  如果 col[i] == 0 并且 d1[row + i] == 0 并且 d2[row - i + n] == 0 那么  // 把第i列和两条对角线标记为占用，把它们设为1  col[i] = 1  d1[row + i] = 1  d2[row - i + n] = 1  // 递归地调用dfs函数，把下一行作为参数  dfs(row + 1)  // 把第i列和两条对角线标记为未占用，把它们设回0  col[i] = 0  d1[row + i] = 0  d2[row - i + n] = 0   1. 样例输出与相关说明:     2、倒水问题:给定2 个没有刻度容器，对于任意给定的容积，求出如何只用两个瓶装出L 升  的水，如果可以，输出步骤，如果不可以，请输出No Solution。  根据题目分析可知：需要设置两个容器分别为A或B，每次只能对这个瓶子内的水进行倒满或者清空操作，倒满A或B，清空A或B，将A中的水倒入B，直到A为空或B满，将B中的水倒入A，直到B为空或A满求出最少的操作次数，以及对应的操作序列。如果无法达到目标水量，输出No Solution。  **解决思路:**  首先，定义了一些常量和变量。`N` 是状态数组的大小，`a`、`b` 和 `c` 分别表示两个容器的容量和目标容量。`st` 数组用来记录状态是否出现过，避免重复搜索。接下来定义了一个结构体 `Node`，表示搜索过程中的状态。其中 `x` 和 `y` 分别表示两个容器中的水量，`path` 表示从初始状态到达当前状态的操作序列。在 `bfs()` 函数中，使用队列来实现广度优先搜索。初始时将两个容器都为空的状态加入队列，并标记为已访问。在搜索过程中，每次取出队首元素，并判断是否满足条件。如果满足条件，则返回操作序列。接下来枚举所有可能的操作，并判断是否可以进行该操作。如果可以，则更新状态并将新状态加入队列。  **伪代码:**  // 定义两个水壶的最大容量和目标水量  a, b, c = 输入()  // 定义一个布尔矩阵来存储已经访问过的状态  st = 创建一个大小为 (a+1) x (b+1) 的矩阵，并用 false 填充  // 定义一个结构体来存储水壶的状态和操作路径  结构体 Node {  x, y // 每个水壶中的水量  path // 一个操作序列  }  // 定义一个队列来存储待探索的状态  q = 创建一个空队列  // 把初始状态 (0, 0) 和一个空路径放入队列  q.入队(Node(0, 0, ""))  // 把初始状态标记为已访问  st[0][0] = true  // 循环直到队列为空或找到解决方案  当 q 不为空时 {  // 从队列中弹出队首的状态  t = q.出队()  // 从状态中获取每个水壶中的水量  x = t.x  y = t.y  // 检查是否有任何一个水壶达到了目标水量  如果 x == c 或者 y == c {  // 返回状态中的操作路径  返回 t.path  }  // 尝试对水壶进行所有可能的操作  // 把 A 水壶装满水  如果 st[a][y] 是 false {  // 把新状态标记为已访问  st[a][y] = true  // 把新状态和更新后的路径放入队列  q.入队(Node(a, y, t.path + "FILL(A)\n"))  }  // 把 B 水壶装满水  如果 st[x][b] 是 false {  // 把新状态标记为已访问  st[x][b] = true  // 把新状态和更新后的路径放入队列  q.入队(Node(x, b, t.path + "FILL(B)\n"))  }  // 把 A 水壶倒空  如果 st[0][y] 是 false {  // 把新状态标记为已访问  st[0][y] = true  // 把新状态和更新后的路径放入队列  q.入队(Node(0, y, t.path + "DROP(A)\n"))  }  // 把 B 水壶倒空  如果 st[x][0] 是 false {  // 把新状态标记为已访问  st[x][0] = true  // 把新状态和更新后的路径放入队列  q.入队(Node(x, 0, t.path + "DROP(B)\n"))  }  // 把 A 水壶中的水倒入 B 水壶  k = min(x, b - y) // 计算可以倒入的水量  如果 st[x - k][y + k] 是 false {  // 把新状态标记为已访问  st[x - k][y + k] = true  // 把新状态和更新后的路径放入队列  q.入队(Node(x - k, y + k, t.path + "POUR(A,B)\n"))  }  // 把 B 水壶中的水倒入 A 水壶  k = min(y, a - x) // 计算可以倒入的水量  如果 st[x + k][y - k] 是 false {  // 把新状态标记为已访问  st[x + k][y - k] = true  // 把新状态和更新后的路径放入队列  q.入队(Node(x + k, y - k, t.path + "POUR(B,A)\n"))  }  }  // 如果在探索所有状态后没有找到解决方案，返回 "No solution"  返回 "No solution"  样例输出及相关说明:      算法的时间复杂度为:O(n^2)【采用了广度优先搜索遍历】  《程序设计艺术与方法》课程实验报告   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 实验名称 | 实验四 动态规划算法的实现 | | | | | | | | 姓 名 | 党存远 |  | 计算机与信息学院 | 班 级 | 物联网工程22-2 | 学 号 | 2022217587 | | 实验日期 | 2023.6.6 | | 指导教师 | 曹力 | | 成 绩 |  | | **一、实验目的和要求**  (1) 理解动态规划的基本思想、动态规划算法的基本步骤。  (2) 掌握动态规划算法实际步骤。 | | | | | | | | | **二、实验预习内容**  (1) 求两个字符串的最长公共子序列。X 的一个子序列是相应于 X 下标序列{1, 2, …, m}的一个子序列，求解两个序列的所有 子序列中长度最大的，例如输入：pear, peach 输出：pea。  (2) 给定两个字符串 a 和 b，现将串 a 通过变换变为串 b，可用的操作为，删除串 a 中的一 个字符；在串 a 的某个位置插入一个元素；将串 a 中的某个字母换为另一个字母。对于 任意的串 a 和串 b，输出最少多少次能够将串变为串 b。 思考：输出变换的步骤。 | | | | | | | | | **三、实验项目摘要**  1、求两个字符串的最长公共子序列。X 的一个子序列是相应于 X 下标序列{1, 2, …, m}的一个子序列，求解两个序列的所有 子序列中长度最大的，例如输入：pear, peach 输出：pea。  2、给定两个字符串 a 和 b，现将串 a 通过变换变为串 b，可用的操作为，删除串 a 中的一 个字符；在串 a 的某个位置插入一个元素；将串 a 中的某个字母换为另一个字母。对于 任意的串 a 和串 b，输出最少多少次能够将串变为串 b。 思考：输出变换的步骤。 | | | | | | | | | **四、实验结果与分析（复杂度分析、伪代码、测试样例及相关说明）**  **1、**求两个字符串的最长公共子序列。X 的一个子序列是相应于 X 下标序列{1, 2, …, m}的一个子序列，求解两个序列的所有子序列中长度最大的，例如输入：pear, peach 输出：pea。  解决思路:使用动态规划方法来进行求解，状态f[i][j]表示a字符串的前i个字符和b字符串的前j个字符中最长子序列的长度。  **伪代码:**  首先输入两个字符串a和b  设n为a的长度，m为b的长度  创建一个n+1行m+1列的二维数组f，初始化为0  遍历i从1到n  遍历j从1到m  f[i][j] = max(f[i-1][j], f[i][j-1]) //计算a字符串前i-1个字符与b字符串前j-1个字符之间最长公共串的长度，  如果a[i-1] == b[j-1]，对下一个字符串求出最大相同的长度  f[i][j] = max(f[i][j], f[i-1][j-1] + 1) // 选a[i-1]和b[j-1]  测试样例及相关说明:      **2、**给定两个字符串 a 和 b，现将串 a 通过变换变为串 b，可用的操作为，删除串 a 中的一 个字符；在串 a 的某个位置插入一个元素；将串 a 中的某个字母换为另一个字母。对于 任意的串 a 和串 b，输出最少多少次能够将串变为串 b。  **解决思路:**  首先，定义了一些常量和变量。N 是状态数组的大小，f 数组用来记录状态。  在 main() 函数中，首先读入两个字符串 a 和 b，并计算它们的长度。  接下来初始化状态数组。当 j = 0 时，表示将 a 变为空串，需要删除 i 次；当 i = 0 时，表示将空串变为 b，需要插入 j 次。  接下来使用动态规划来求解。状态表示为 f[i][j]，表示将 a 的前 i 个字符变为 b 的前 j 个字符所需的最少操作次数。  状态转移分为三种情况：删除、插入和替换。如果删除，则状态转移方程为 f[i][j] = f[i - 1][j]+1；如果插入，则状态转移方程为  f[i][j] = f[i][j - 1] + 1；  如果替换，则需要满足 a[i - 1] != b[j - 1]，此时状态转移方程为  f[i][j] = f[i - 1][j - 1] + 1。如果不需要替换，则状态转移方程为 f[i][j] = f[i - 1][j - 1]。  最后，在循环结束后，输出结果即可。  **//伪代码:**  // 定义一个叫做printSteps的函数，它接受两个整数参数i和j  函数 printSteps(i, j)  // 如果i和j都等于0，那么直接返回  如果 i == 0 并且 j == 0 那么  返回  结束如果  // 如果i大于0，并且dp[i][j]等于dp[i-1][j]加1，那么递归地调用printSteps(i-1, j)，并输出从A中删除a[i-1]的操作  如果 i > 0 并且 dp[i][j] == dp[i - 1][j] + 1 那么  printSteps(i - 1, j)  输出 "从A中删除" + a[i - 1]  结束如果  // 如果j大于0，并且dp[i][j]等于dp[i][j-1]加1，那么递归地调用printSteps(i, j-1)，并输出在A中插入b[j-1]的操作  否则如果 j > 0 并且 dp[i][j] == dp[i][j - 1] + 1 那么  printSteps(i, j - 1)  输出 "在A中插入" + b[j - 1]  结束如果  // 如果i和j都大于0，并且dp[i][j]等于dp[i-1][j-1]加上a[i-1]和b[j-1]是否不相等的值，那么递归地调用printSteps(i-1, j-1)，并输出如果需要的话，把a[i-1]替换成b[j-1]的操作  否则如果 i > 0 并且 j > 0 并且 dp[i][j] == dp[i - 1][j - 1] + (a[i - 1] != b[j - 1]) 那么  printSteps(i - 1, j - 1)  如果 a[i - 1] != b[j - 1] 那么  输出 "把" + a[i - 1] + "替换成" + b[j - 1]  结束如果  结束如果  结束函数  // 定义一个叫做main的函数，它没有参数  函数 main()  // 输入两个字符串a和b  输入 a 和 b  // 获取a和b的长度，分别赋值给len1和len2  len1 = a的长度()  len2 = b的长度()  // 循环从0到len1，把dp数组的第一列设为i的值  对于 i = 0 到 len1 做  dp[i][0] = i  结束对于  // 循环从0到len2，把dp数组的第一行设为j的值  对于 j = 0 到 len2 做  dp[0][j] = j  结束对于  // 循环从1到len1，再循环从1到len2，计算dp数组的其他元素的值，根据a和b的字符是否相等来决定是否需要编辑操作，并取最小值作为结果  对于 i = 1 到 len1 做  对于 j = 1 到 len2 做  如果 a[i - 1] == b[j - 1] 那么  dp[i][j] = dp[i - 1][j - 1]  否则  dp[i][j] = 最小值(dp[i - 1][j - 1], 最小值(dp[i - 1][j], dp[i][j - 1])) + 1  结束如果  结束对于  结束对于  // 输出最小编辑距离是dp[len1][len2]的值  输出 "最小编辑距离是" + dp[len1][len2]  // 输出编辑的步骤，调用printSteps函数，传入len1和len2作为参数  输出 "编辑的步骤是："  printSteps(len1, len2)  // 返回0，表示程序正常结束  返回 0  测试样例及相关说明: | | | | | | | |   《程序设计艺术与方法》课程实验报告   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 实验名称 | 实验六：字符串/组合数学类问题的建模与实现 | | | | | | | | 姓 名 | 党存远 |  | 计算机与信息学院 | 班 级 | 物联网工程22-2 | 学 号 | 2022217587 | | 实验日期 | 2023.6.15 | | 指导教师 | 曹力 | | 成 绩 |  | | **一、实验目的和要求**  (1) 理解字符串处理问题的建模与算法实现。  (2) 掌握组合数学类问题的建模与算法实现。 | | | | | | | | | **二、实验预习内容**  1、自学 KMP 算法，并上机练习。  2、有袋子里均匀地装着 c 种颜色的巧克力，每种巧克力均有无限多。每次从袋子里拿一块 放在桌子上，如果桌子上已经有一块颜色相同的巧克力，就把两块巧克力都吃掉。一共 取出了 n 块巧克力，问：最后桌子上有 m 块的概率为多大。例如 c=5,n=100,m=2 时，概率为0.625 | | | | | | | | | **三、实验项目摘要**  1、自学 KMP 算法，并上机练习。  2、有袋子里均匀地装着 c 种颜色的巧克力，每种巧克力均有无限多。每次从袋子里拿一块 放在桌子上，如果桌子上已经有一块颜色相同的巧克力，就把两块巧克力都吃掉。一共 取出了 n 块巧克力，问：最后桌子上有 m 块的概率为多大。例如 c=5,n=100,m=2 时，概率为0.625 | | | | | | | | | **四、实验结果与分析（复杂度分析、伪代码、测试样例及相关说明）**  **1、**自学 KMP 算法，并上机练习。  **Kmp算法的伪代码:**  // 构造next数组，记录模式串中每个前缀的最长相等前后缀长度  function get\_next(pattern):  // 初始化next数组，长度和模式串相同  next = new array of size pattern.length  // 第一个元素设为0，表示没有相等的前后缀  next[0] = 0  // 定义两个指针i和j，分别指向模式串的当前字符和最长相等前后缀的末尾字符  i = 1 // 从第二个字符开始  j = 0 // 初始时没有相等的前后缀  // 遍历模式串，直到i到达末尾  while i < pattern.length:  // 如果当前字符和最长相等前后缀的下一个字符相同，说明最长相等前后缀长度增加了1  if pattern[i] == pattern[j]:  // 更新j指针  j = j + 1  // 将j的值赋给next[i]，表示pattern[0..i]的最长相等前后缀长度为j  next[i] = j  // 更新i指针，继续遍历下一个字符  i = i + 1  // 否则，如果当前字符和最长相等前后缀的下一个字符不同，说明需要缩短最长相等前后缀的长度，找到更短的相等前后缀  else:  // 如果j已经为0，说明没有更短的相等前后缀了，那么next[i]就为0，表示pattern[0..i]没有相等的前后缀  if j == 0:  next[i] = 0  // 更新i指针，继续遍历下一个字符  i = i + 1  // 否则，如果j不为0，说明还有可能找到更短的相等前后缀，那么就根据next数组回溯j指针，找到上一个相等前后缀的末尾字符位置  else:  j = next[j - 1]  // 返回构造好的next数组  return next  // 使用next数组进行字符串匹配，返回模式串在文本串中出现的位置（如果有多个，返回第一个）  function kmp\_search(text, pattern):  // 获取next数组  next = get\_next(pattern)  // 定义两个指针i和j，分别指向文本串和模式串的当前字符  i = 0 // 文本串从第一个字符开始匹配  j = 0 // 模式串从第一个字符开始匹配  // 遍历文本串，直到i到达末尾或者找到匹配位置  while i < text.length:  // 如果当前字符匹配成功，说明文本串和模式串的当前部分是相同的  if text[i] == pattern[j]:  // 更新i和j指针，继续匹配下一个字符  i = i + 1  j = j + 1  // 如果j到达模式串的末尾，说明模式串完全匹配了文本串的一个子串，返回匹配位置为i - j（即文本串中模式串的起始下标）  if j == pattern.length:  return i - j  // 否则，如果当前字符匹配失败，说明文本串和模式串的当前部分有不同的字符  else:  // 如果j已经为0，说明模式串的第一个字符就和文本串的当前字符不同，那么就将i指针后移一位，继续匹配文本串的下一个字符  if j == 0:  i = i + 1  // 否则，如果j不为0，说明模式串的前j个字符和文本串的前j个字符是相同的，那么就根据next数组回溯j指针，找到下一个可能匹配的位置  else:  j = next[j - 1]  // 如果遍历完文本串都没有找到匹配位置，说明模式串在文本串中不存在，返回-1  return -1  测试样例及相关说明:    有袋子里均匀地装着 c 种颜色的巧克力，每种巧克力均有无限多。每次从袋子里拿一块 放在桌子上，如果桌子上已经有一块颜色相同的巧克力，就把两块巧克力都吃掉。一共 取出了 n 块巧克力，问：最后桌子上有 m 块的概率为多大。  例如 c=5,n=100,m=2 时，概率为0.625  **解题思路:**  设置一个变量j 代表桌子上巧克力的数量。当 j 等于 0 时，意味着桌子上没有巧克力。  在 else 语句中，我们计算 f[i][j] 的值。这个值表示取出 i 块巧克力后，桌子上有 j 块巧克力的概率。我们可以通过两种方式来得到这种情况：  当前取出的巧克力与桌子上的巧克力颜色不同。此时，桌子上的巧克力数量会增加 1。这种情况的概率为 (c-j+1)/c，因为有 c-j+1 种颜色与桌子上的巧克力颜色不同。  当前取出的巧克力与桌子上的巧克力颜色相同。此时，桌子上的巧克力数量会减少 1。这种情况的概率为 j/c，因为有 j 种颜色与桌子上的巧克力颜色相同。  综上所述，我们可以得到如下公式来计算 f[i][j] 的值：  f[i][j] = f[i-1][j-1] \* (c-j+1)/c + f[i-1][j+1] \* j/c  **//伪代码:**  // 输入c, n, m  input c, n, m  // 初始化f数组，大小为(n+1) x (n+1)，初始值为0  f = new array of size (n+1) x (n+1), initialized with 0  // 边界条件：进行0次操作后，还剩下0个巧克力的概率为1  f[0][0] = 1  // 遍历每一次操作，从1到n  for i from 1 to n:  // 遍历每一种可能的剩余巧克力数，从0到i  for j from 0 to i:  // 如果剩余巧克力数为0，说明上一次操作吃掉了一个巧克力，那么这一次操作的概率等于上一次操作剩余一个巧克力的概率乘以吃掉的概率（即1/c）  if j == 0:  f[i][j] = f[i-1][j+1] \* (j+1) / c  // 如果剩余巧克力数等于操作次数，说明每一次操作都没有吃掉巧克力，那么这一次操作的概率等于上一次操作剩余同样数量巧克力的概率乘以不吃掉的概率（即(c-j)/c）  else if j == i:  f[i][j] = f[i-1][j-1] \* (c-j+1) / c  // 如果剩余巧克力数在0到i之间，说明有些操作吃掉了巧克力，有些没有，那么这一次操作的概率等于上一次操作剩余比当前多一个或者少一个巧克力的概率乘以相应的吃掉或者不吃掉的概率（即(j+1)/c或者(c-j+1)/c）  else:  f[i][j] = f[i-1][j-1] \* (c-j+1) / c + f[i-1][j+1] \* (j+1) / c  // 输出最终结果，即进行n次操作后，还剩下m个巧克力的概率  output f[n][m]  测试样例及相关说明 | | | | | | | | |  | | | | | | | | | | | | | | | |