**算法设计与分析 实验报告**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 宋致远 | | 院系 | 软件学院 | | 学号 | 2201110126 |
| 任课教师 | | 王金宝 | | | 指导教师 | 王金宝 | |
| 实验地点 | | 研究院中507 | | | 实验时间 | 2022年4 月 25 日 | |
| 实验名称 | | 树搜索算法设计与实现 | | | | | |
| 同 组 人 | | 我 | | | | | |
| 实验内容 | | | | | | | |
| **1、问题**   * **活动选择问题**   1. 输入：*S={1, 2, …, n}*, *F*={ [*si，fi*] }，*n≥i≥1*   输出：*S*中的最大相容活动集合  要求:  （1）使用动态规划技术求解活动选择问题. 请写出你使用的递归方程，说明其中各项的含义; 请说明优化子结构以及问题是否具有子问题重叠性; 实现你设计的算法.）  （2）使用贪心算法求解活动选择问题；（实现课件上的算法即可）  （3）使用树搜索算法求解活动选择问题；（说明如何将问题转化为树搜索问题；说明数中每个节点的含义；说明你使用的分支界限搜索算法）  （随机生成若干活动，并使用它测试动态规划算法、贪心算法和树搜索算法的运行时间和活动数量）  **2、基于动态规划技术的算法**  递归方程：  c[i][j]=  c[i][j]表示为sij中最大兼容子集中的活动数目。  当sij非空时，若ak 在sij的最大兼容子集中被使用，则sik和skj的最大兼容子集也被使用故c[i][j]=c[i][k]+c[k][j]+1  子问题重叠性：  由于每次计算预选活动集P时都要获取以之前活动为结尾的最大相容活动集，所以具有子问题重叠性。  代码  void dynamic\_activity\_selector(int \*s,int \*f,int c[N+1][N+1],int ret[N+1][N+1]);  void trace\_route(int ret[N+1][N+1],int i,int j);    int main()  {  int s[N+1] = {-1,1,3,0,5,3,5,6,8,8,2,12};  int f[N+1] = {-1,4,5,6,7,8,9,10,11,12,13,14};  int c[N+1][N+1]={0};  int ret[N+1][N+1]={0};  int i,j;  dynamic\_activity\_selector(s,f,c,ret);  printf("c[i][j]的值如下所示：\n");  for(i=1;i<=N;i++)  {  for(j=1;j<=N;j++)  printf("%d ",c[i][j]);  printf("\n");  }  //包括第一个和最后一个元素  printf("最大子集的个数为: %d\n",c[1][N]+2);  return 0;  }  void dynamic\_activity\_selector(int \*s,int \*f,int c[N+1][N+1],int ret[N+1][N+1])  {  int i,j,k;  int temp;  //当i>=j时候，子问题的解为空，即c[i][j]=0  for(j=1;j<=N;j++)  for(i=j;i<=N;i++)  c[i][j] = 0;  //当i>j时，需要寻找子问题的最优解，找到一个k使得将问题分成两部分  for(j=2;j<=N;j++)  for(i=1;i<j;i++)  {  //寻找k，将问题分成两个子问题c[i][k]、c[k][j]  for(k=i+1;k<j;k++)  if(s[k] >= f[i] && f[k] <= s[j]) //判断k活动是否满足兼容性  {  temp = c[i][k]+c[k][j]+1;  if(c[i][j] < temp)  {  c[i][j] =temp;  ret[i][j] = k;  }  }  }  }  void Recursive\_Activity\_Selector(vector<int>\* A, int\* s, int\* f, int k, int n) {    int m = k + 1;  while (m <= n && s[m] < f[k]) {  m++;  }    if (m <= n) {  A->push\_back(m);  Recursive\_Activity\_Selector(A, s, f, m, n);  }  }  #ifndef \_\_TREE\_H\_\_  #define \_\_TREE\_H\_\_  // Tree node  typedef struct TreeNode{  struct TreeNode\* parent;  struct TreeNode\* leftChild;  struct TreeNode\* rightChild;  int border;  int extra;  int actiId;  #endif  #include <stdio.h>  #include <stdlib.h>  #include "bool.h"  #include "node.h"  #include "tree.h"  #define SIZE 10  #define RANGE\_MIN 0  #define RANGE\_MAX 20  Node S[SIZE];  TreeNode \*root;  TreeNode \*lastActi;  int treeNodeNum = 0;  void tree\_search(){  printf("\nThis is tree search\n");    randomArr(S, SIZE, RANGE\_MIN, RANGE\_MAX);    printf("random array:\n");  \_printActi();    mergeSort(S, SIZE, false, true);    printf("sorted array:\n");  \_printActi();    \_initRoot();    \_treeFirstResult();  \_treeSearchPre(root); \_treePrintResult();  }  // 初始化根节点  void \_initRoot(){  root = (TreeNode\*)malloc(sizeof(TreeNode));  root->border = 0;  root->extra = 0;  root->actiId = -1;  root->parent = NULL;  root->leftChild = NULL;  root->rightChild = NULL;  }  TreeNode\* \_createTreeNode(TreeNode\* parent, int border, int extra, int actiId, bool leftChild){  TreeNode\* node;  if(leftChild){  parent->leftChild = (TreeNode\*)malloc(sizeof(TreeNode));  node = parent->leftChild;  } else {  parent->rightChild = (TreeNode\*)malloc(sizeof(TreeNode));  node = parent->rightChild;  }  node->border = border;  node->extra = extra;  node->actiId = actiId;  node->parent = parent;  node->rightChild = NULL;  node->leftChild = NULL;  return node;  }  void \_treeFirstResult(){  TreeNode\* node = root;  for(int i = 0; i < SIZE; i++){  if(node->parent != NULL){  //printf("%d, %d\n",node->parent->actiId, node->parent->border);  }  if(S[i].x >= node->border){  node = \_createTreeNode(node, S[i].y, node->extra + 1, i, true);  } else {  node = \_createTreeNode(node, node->border, node->extra, i, false);  }  }  lastActi = node;  }  int \_getPossibleMaxActiNum(int actiId){  int max = 0;  for(int i = actiId + 1; i < SIZE; i++){  if(S[i].x >= S[actiId].y){  max++;  }  }  return max;  }  void \_treeSearchPre(TreeNode\* node){  treeNodeNum++;  if(node == NULL){  return;  } else {  // 树的深度不能超过活动数量上限  if(node->actiId >= SIZE - 1){  return;  }  // 判断是否有打到最大相容活动数的可能性  if(lastActi->extra < node->extra + \_getPossibleMaxActiNum(node->actiId)){  // 不管怎么样，右子节点一定是可以有的  if(node->rightChild == NULL){ // 该子节点已经有了的话就不创建了  \_createTreeNode(node, node->border, node->extra, node->actiId + 1, false);  }  if(node->leftChild == NULL && S[node->actiId + 1].x >= node->border){ // 该子节点已经有了的话就不创建了  \_createTreeNode(node, S[node->actiId + 1].y, node->extra + 1, node->actiId + 1, true);    if(node->leftChild->extra > lastActi->extra){  lastActi = node->leftChild;  }  }  // 继续遍历  \_treeSearchPre(node->leftChild);  \_treeSearchPre(node->rightChild);  } else {  // 没有的话直接结束（枝剪）  return;  }  }  }  void \_treePrintResult(){  printf("there are %d tree nodes, max acti num -> %d, they are:\n", treeNodeNum, lastActi->extra);  \_treePrintActis(lastActi);  }  void \_treePrintActis(TreeNode\* node){  // 如果是根节点，停止递归  if(node->parent == NULL){  return;  }  \_treePrintActis(node->parent);  // 如果是左节点，说明该活动在结果集中  if(node == node->parent->leftChild){  printf("(%d, %d)\n", S[node->actiId].x, S[node->actiId].y);  }  } | | | | | | | |
| 实验结论（结果分析、遇到的困难和解决方法等） | | | | | | 备注 |  |
| 数搜索算法得出的实验结果与贪心算法和动态规划算法得出的结论相符 | | | | | | | |