**《算法设计与分析》实验报告**

实验名称 ： 实验10 算法比较

实验日期 ： 　 2025/6/3

姓 名 ： 　　　高心阳

学 号 ： 　 084623237

班 级 ： 　 计算机232

成 绩 ：

**人工智能与信息技术学院**

**南京中医药大学**

|  |
| --- |
| **实验目的：** |
| 1. 深入了解不同算法的思想 2. 比较不同算法应用的区别和联系 |
| **实验内容和要求** |
| 1、背包问题在不同算法中出现了，请思考并查阅该问题在哪些算法中应用，解决相应问题有什么不同或相同之处。请根据不同算法举例，并给出相应的文字说明。 |
| **运行结果（写清题号）** |
| |  | | --- | | 动态规划 | | #include <iostream>  #include <vector>  #include <algorithm>  int knapsack\_01\_dp(int W,std::vector<int>& wt,std::vector<int>& val, int n) {  std::vector<std::vector<int>> dp(n + 1, std::vector<int>(W + 1, 0));  for (int i = 0; i <= n; ++i) {  for (int j = 0; j <= W; ++j) {  if (i == 0 || j == 0) {  dp[i][j] = 0;  } else if (wt[i - 1] <= j) {  dp[i][j] = std::max(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);  } else {  dp[i][j] = dp[i - 1][j];  }  }  }  return dp[n][W];  } |  |  | | --- | | 贪心算法 | | #include <iostream>  #include <vector>  #include <algorithm>  struct Item {  int id;  int value;  int weight;  double density;  Item(int i, int v, int w) : id(i), value(v), weight(w) {  if (weight > 0) {  density = static\_cast<double>(value) / weight;  } else {  density = 0;  }  }  };  bool compareItems(const Item& a, const Item& b) {  return a.density > b.density;  }  double fractional\_knapsack\_greedy(int W, std::vector<Item>& items\_vec) {  std::sort(items\_vec.begin(), items\_vec.end(), compareItems);  double total\_value\_in\_knapsack = 0.0;  int current\_weight\_in\_knapsack = 0;  for (const auto& item : items\_vec) {  if (current\_weight\_in\_knapsack == W) {  break;  }  if (item.weight <= (W - current\_weight\_in\_knapsack)) {  current\_weight\_in\_knapsack += item.weight;  total\_value\_in\_knapsack += item.value;  } else {  int remaining\_capacity = W - current\_weight\_in\_knapsack;  double fraction\_to\_take = static\_cast<double>(remaining\_capacity) / item.weight;  total\_value\_in\_knapsack += item.value \* fraction\_to\_take;  current\_weight\_in\_knapsack += remaining\_capacity;  break;  }  }  return total\_value\_in\_knapsack;  } |  |  | | --- | | 递归回溯法 | | #include<iostream>  #include<vector>  #include<algorithm>  int knapsack\_01\_recursive\_internal(intW, conststd::vector<int>&wt, conststd::vector<int>&val, intcurrent\_idx, inttotal\_items) {  if (W == 0 || current\_idx == total\_items) {  return 0;  }  if (wt[current\_idx] > W) {  return knapsack\_01\_recursive\_internal(W, wt, val, current\_idx + 1, total\_items);  } else {  intvalue\_without\_current = knapsack\_01\_recursive\_internal(W, wt, val, current\_idx + 1, total\_items);  intvalue\_with\_current = val[current\_idx] + knapsack\_01\_recursive\_internal(W - wt[current\_idx], wt, val, current\_idx + 1, total\_items);  returnstd::max(value\_with\_current, value\_without\_current);  }  }  int knapsack\_01\_recursive(intW, conststd::vector<int>&wt, conststd::vector<int>&val) {  if (wt.empty() || val.empty() || wt.size() != val.size()) {  return 0;  }  return knapsack\_01\_recursive\_internal(W, wt, val, 0, wt.size());  } |  |  | | --- | | 分支限界法 | | #include <iostream>  #include <vector>  #include <queue>  #include <algorithm>  struct Item\_BnB {  int id;  int value;  int weight;  double density;  Item\_BnB(int i, int v, int w) : id(i), value(v), weight(w) {  if (weight > 0) {  density = static\_cast<double>(value) / weight;  } else {  density = 0;  }  }  };  bool compareItems\_BnB(const Item\_BnB& a, const Item\_BnB& b) {  return a.density > b.density;  }  struct Node\_BnB {  int level;  int current\_value;  int current\_weight;  double upper\_bound;  bool operator<(const Node\_BnB& other) const {  return upper\_bound < other.upper\_bound;  }  };  double calculate\_upper\_bound(const std::vector<Item\_BnB>& items\_sorted, int W\_capacity, const Node\_BnB& node) {  if (node.current\_weight > W\_capacity) {  return 0;  }  double bound = node.current\_value;  int weight\_so\_far = node.current\_weight;  int num\_items = items\_sorted.size();  for (int i = node.level; i < num\_items; ++i) {  if (weight\_so\_far + items\_sorted[i].weight <= W\_capacity) {  weight\_so\_far += items\_sorted[i].weight;  bound += items\_sorted[i].value;  } else {  int remaining\_capacity = W\_capacity - weight\_so\_far;  bound += items\_sorted[i].density \* remaining\_capacity;  break;  }  }  return bound;  }  int knapsack\_01\_branch\_and\_bound(int W\_capacity, std::vector<Item\_BnB>& items\_vec) {  if (items\_vec.empty()) return 0;  std::sort(items\_vec.begin(), items\_vec.end(), compareItems\_BnB);  std::priority\_queue<Node\_BnB> pq;  Node\_BnB root\_node = {0, 0, 0, 0.0};  root\_node.upper\_bound = calculate\_upper\_bound(items\_vec, W\_capacity, root\_node);  pq.push(root\_node);  int max\_profit\_found = 0;  int num\_items = items\_vec.size();  while (!pq.empty()) {  Node\_BnB current\_node = pq.top();  pq.pop();  if (current\_node.upper\_bound <= max\_profit\_found) {  continue;  }  if (current\_node.level == num\_items) {  if (current\_node.current\_value > max\_profit\_found) {  max\_profit\_found = current\_node.current\_value;  }  continue;  }  const Item\_BnB& current\_item = items\_vec[current\_node.level];  Node\_BnB node\_with\_item = current\_node;  node\_with\_item.level++;  if (current\_node.current\_weight + current\_item.weight <= W\_capacity) {  node\_with\_item.current\_weight += current\_item.weight;  node\_with\_item.current\_value += current\_item.value;  if (node\_with\_item.current\_value > max\_profit\_found) {  max\_profit\_found = node\_with\_item.current\_value;  }  node\_with\_item.upper\_bound = calculate\_upper\_bound(items\_vec, W\_capacity, node\_with\_item);  if (node\_with\_item.upper\_bound > max\_profit\_found) {  pq.push(node\_with\_item);  }  }  Node\_BnB node\_without\_item = current\_node;  node\_without\_item.level++;  node\_without\_item.upper\_bound = calculate\_upper\_bound(items\_vec, W\_capacity, node\_without\_item);  if (node\_without\_item.upper\_bound > max\_profit\_found) {  pq.push(node\_without\_item);  }  }  return max\_profit\_found;  }  int main() {  std::vector<Item\_BnB> items;  items.emplace\_back(1, 60, 10);  items.emplace\_back(2, 100, 20);  items.emplace\_back(3, 120, 30);  int W = 50;  std::cout << knapsack\_01\_branch\_and\_bound(W, items) << std::endl;  return 0;  } |   **相同之处**  **优化目标**：根本目标都是在满足背包容量约束的前提下，最大化所选物品的总价值。  **最优解保证**：动态规划、递归回溯以及分支限界法，在正确实现的情况下，都能保证为0/1背包问题和完全背包问题找到全局最优解。  **不同之处**  **决策过程与搜索方式：**   * + - * 贪心算法：进行一系列单向的、不可撤销的局部最优决策。       * 动态规划：通过状态转移方程，隐式地考虑了所有可能的决策组合，并从中选择最优的子路径，最终构建全局最优解。       * 递归回溯：显式地通过递归调用探索决策树的每一个分支，尝试“选”与“不选”的每一种可能。       * 分支限界法：基于决策树的搜索，但通过限界函数和剪枝策略，主动地避免了对不含最优解的子树的探索。   **对物品的处理：**   * + - * 贪心算法在解决分数背包问题时，能够利用物品可分割的特性。       * 其他三种算法在处理0/1背包或完全背包时，则必须考虑物品的不可分割性   **最优性保证的范围：**   * + - * 贪心算法仅对分数背包问题能保证得到最优解。       * 动态规划、递归回溯、分支限界法对0/1背包和完全背包问题均能保证最优解。   **时空复杂度综合对比表**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 算法名称 | 主要问题类型 | 时间复杂度 | 空间复杂度 | 最优性 | | 动态规划 | 0/1背包,完全背包 | O(nW) | O(nW) | 是 | | 贪心算法 | 0/1背包,完全背包 | O(nlogn) | O(n) | 否 | | 递归回溯 | 0/1背包,完全背包 | O(2n) | O(n) | 是 | | 分支限界法 | 0/1背包,完全背包 | 最坏O(2n),平均情况通常更好 | 最坏可达O(2n)或O(n) | 是 | |
| **实验的体会与建议** |
| 对比了四大算法对于01背包和分数背包的应用，了解了各个算法的优劣以及适配的问题，对背包问题和四大算法有了更加深入的理解。 |