算法设计与分析补充

1. NP问题

(1) NP问题: 能在多项式时间内检验一个解是否正确的问题, 与是否存在多项式时间的算法无关.

(2) NPC问题: 所有NP问题都可规约为NPC问题, 它是NP问题中最难的一部分问题, 目前无多项式时间算法.

(3) NP-Hard问题: 不能在多项式时间内检验一个解是否正确的问题. 所有的NP问题都可规约为NP-Hard问题.

例子:

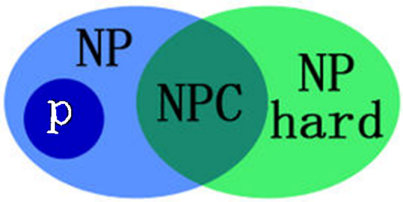
(1) NP问题: 密码破译.

(2) NPC问题: Hamilton回路、图染色、蛋白质折叠问题.

(3) NP-Hard问题:

P、NP、NPC、NP-Hard的关系:

难度: $P \leq NP \leq NPC \leq NP-Hard$.



2. 散列表与Hash函数

散列表冲突的原因: 散列地址不同的结点争夺同一个后继散列地址的现象称为聚集或堆积. 这将造成不是同义词的结点也处在同一个探查序列之中, 从而增加了探查序列的长度, 即增加了查找时间. 若散列函数不好或装填因子过大, 都会使堆积现象加剧.

Hash函数冲突的原因: 哈希冲突的根本原因是哈希函数的设计不够好, 导致计算得到的哈希值不够随机, 出现了大量的哈希冲突.

3. DP

分治与DP的关系;

(1) 共同点:

① 都要求子问题和原问题有相同的性质, 只是量级不同.

② 子问题都具有最优子结构, 即在当前量级, 返回的是最好的输出.

(2) 不同点:

① 分治法通常用递归.

② 动态规划自底向上用迭代, 自顶向下用递归.

③ 分治法将分解后的子问题相互独立; 动态规划分解后的子问题为相互间有联系，且需要从这些子问题中归纳出原问题的解.

回溯与DP的关系:

(1) 不同点:

① DP依赖于最优性原理. 最优性原则指出: 决定或选择每个子序列的最优序列也必须是最优的. 回溯问题通常不是最优方法, 它们只能应用于允许部分候选解决方案概念的问题.

② DP通过将其分解为子问题来解决大型的计算密集型问题, 这些子问题的解决方案只需了解当前的先验解决方案. 在修剪解决方案树的情况下, 回溯更复杂, 因为已知特定路径不会产生最佳结果. 因此, 由于DP假定执行了所有计算, 回溯可以优化内存, 然后该算法返回到成本最低的节点.

③ DP是一种解决优化问题的策略. 优化问题是关于最小或最大结果(单个结果). 但是在回溯中, 用蛮力法, 而不是针对优化问题, 它用于有多个结果且需全部或部分结果时.

④ 具有边界功能的状态空间树的深度第一节点生成称为回溯, 其中, 当前节点取决于生成它的节点. 具有存储功能的状态空间树的深度优先节点生成称为自顶向下DP, 其中当前节点取决于它生成的节点.

最优性原理: 多阶段决策过程的最优决策序列有性质: 不论初始状态和初始决策如何, 对于前面决策所造成的某一状态而言, 其后各阶段的决策序列构成最优策略.

动态规划解决的问题: 多阶段决策过程最优化问题.

适合用动态规划求解的最优化问题应有两个要素: 最优子结构、子问题重叠.

(1) 最优子结构:

① 最优子结构: 若一个问题的最优解包含子问题的最优解, 则称此问题有最优子结构.

② 有最优子结构的问题, 求解子问题的最优解才有意义. (贪心算法也要求最优子结构).

(2) 子问题重叠:

① 若递归算法反复求解相同的子问题, 则称最优化问题具有重叠子问题. 与之相对的, 适合分治方法求解的问题通常在每一步都会生成一个全新的子问题, 此处与分治法的子问题不相交不同.

② 具有子问题重叠的问题, 用自顶向上的动态规划才能减少计算量.

[**例**] 证明边权为正的图上的最短路问题有最优子结构.

[**解**] 记节点 u 到节点 v 的最短路为 $P_{u,v}$, w 为 $P_{u,v}$ 上异于 u 和 v 的任一节点.

下证 $P_{u,w}$ 和 $P_{w,v}$ 分别为 u 到 w 和 w 到 v 的最短路.

若不然, 则存在路径 $P'_{u,w}$ 比 $P_{u,w}$ 更短, 有如下两种情况:

(1) 若 $P'_{u,w}$ 与 $P_{u,w}$ 无除 w 外的公共节点, 则路径 $(P'_{u,w}+P_{w,v})$ 是一条比 $P_{u,v}$ 更短的路径, 矛盾.

(2) 若 $P'_{u,w}$ 与 $P_{u,w}$ 除 w 外还有公共节点 x , 注意到 $P'_{u,w}=P'_{u,x}+P'_{x,w}$, $P_{w,v}=P'_{w,x}+P'_{x,v}$,

则 $P'_{u,v}=P'_{u,x}+P'_{x,v}$ 是一条比 $P_{u,v}$ 更短的路径, 矛盾.

故 $P_{u,w}$ 是 u 到 v 的最短路, 同理 $P_{w,v}$ 是 w 到 v 的最短路.

4. 堆

4.1 堆的时间复杂度

(1) 维护堆: $O(\log n)$.

(2) 无序数组建堆: $O(n)$.

(3) 堆排序: $O(n \log n)$.

4.2 堆的实现

将序列非升序排列, 建小根堆:

```
const int MAXN = 1e3 + 5;
struct HeapSort {
  int n;
  int heap[MAXN];

  HeapSort(int _n) :n(_n) {
    for (int i = 0; i < n; i++) cin >> heap[i];

    init();
    heapSort();
  }

  void sift(int u, int length) {
    int ls = u * 2 + 1, rs = u * 2 + 2;
    if (ls >= length) return;  // 不存在左儿子

    if (rs < length) {  // 存在右儿子
      if (heap[ls] < heap[rs]) {
        if (heap[ls] < heap[u]) {
          swap(heap[ls], heap[u]);
          sift(ls, length);
        }
      }
      else {
        if (heap[rs] < heap[u]) {
          swap(heap[rs], heap[u]);
          sift(rs, length);
        }
      }
    }
    else {  // 只有左儿子
      if (heap[ls] < heap[u]) {
        swap(heap[ls], heap[u]);
        sift(ls, length);
      }
    }
  }

  void print() {
    cout << n << ' ';
```

```
    for (int i = 0; i < n; i++)
      cout << heap[i] << " \n"[i == n - 1];
  }

  void init() {  // 调整为初始堆
    for (int i = n / 2; i >= 0; i--) sift(i, n);
    print();
  }

  void heapSort() {
    for (int i = n - 1; i >= 1; i--) {
      swap(heap[0], heap[i]);
      sift(0, i);
      print();
    }
  }
};
```