Greedy Algorithms

Ming-Hwa Wang, Ph.D.
COEN 279/AMTH 377 Design and Analysis of Algorithms
Department of Computer Engineering
Santa Clara University

Greedy algorithms

- Greedy algorithm works in phases. In each phase, a decision is make that appears to be good (local optimum), without regard for future consequences. When the algorithm terminates, hope that the local optimum is equal to the global optimum. Otherwise, a suboptimal solution is produced. Used to generate approximate answers, rather than exact one which need more complicated algorithms.
- Applications: core-changing problem, traffic problem
- Matroid - The theoretical foundations

Activity-Selection Problem

- Scheduling a resource among several competing activities and try to select a maximum-size set of mutually compatible activities. Activities i and j are compatible if the intervals of start time and finish time \([s_i, f_i]\) and \([s_j, f_j]\) do not overlap.
- Sort the input activities in increasing finishing time by \(O(n \log n)\). The greedy choice is the one that maximize the amount of unscheduled time remaining in \(O(n)\) and always find the optimal solution.

Knapsack Problem

- Fractional knapsack problem
  Sort the value per weight for each item in \(O(n \log n)\) and then taking as much as possible. Always give optimal solution.
- 0/1 knapsack problem
  Not always give optimal solution.

Unit-time Task with Deadlines and Penalties

- Using early-first form, in which the early tasks precede the late tasks, and the early tasks are scheduled in order of non-decreasing deadlines.
- Sort the tasks in non-decreasing order of penalties, For \(t = 1, 2, \ldots, n\), let \(N_t(A)\) denote the number of tasks in \(A\) whose deadline is \(t\) or earlier. if \(N_t(A) \leq t\), then no task is late.

Scheduling Problem

- Given jobs \(j_1, j_2, \ldots, j_N\) all with running times \(t_1, t_2, \ldots, t_N\). What is the best way to schedule these jobs in order to minimize average completion time?
- Virtually all scheduling problems are either NP-complete or are solvable by a greedy algorithm.
- Single processor non-preemptive scheduling: by shortest job first always yield an optimal schedule.
- Multiple processors non-preemptive scheduling: start jobs in order, cycling through processors. Optimal.
- Minimizing the final completion time: NP-complete.

Huffman Codes

- Huffman Codes for file compression
  - If the size of the character set is \(C\), then \([\log C]\) bits are needed in a standard encoding.
  - Real files can be large, usually a big disparity between the most frequent and least frequent characters.
  - Allow the code length to vary from character to character and to ensure that frequently occurring characters have short codes.
  - If the characters are placed only at the leaves, any sequence of bits can always be decoded unambiguously.
  - Trie: binary tree with data only at the leaves, using a 0 to indicate the left branch and a 1 for right. If character \(c_i\) is at depth \(d_i\) and occurs \(f_i\) times, then the cost of the code is equal to \(\sum df_i\).
  - Full tree: all nodes either are leaves or have two children, nodes with only one child can move up a level.
  - Prefix code: If the character codes are different length, as long as no characters code is a prefix of another character code.
  - Huffman code finds the full binary tree of minimum total cost. There are many optimal codes by swapping children in the encoding tree
  - Huffman algorithm: maintain a forest, the weight of a tree is equal to the sum of the frequencies of its leaves. Initially, each node forms a tree. Select the two trees of smallest weight, breaking ties arbitrarily, and form a new tree with the two selected trees as subtrees. \(O(N \log N)\) using heap.
  - First pass collects the frequency data, and the second pass does the encoding. The encoding information must be transmitted at the start of the compressed file, since otherwise it will be impossible to decode. For large files, this is not significant.
Bin Packing

- Given $N$ items of sizes $s_1, s_2, \ldots, s_N$. All sizes satisfy $0 < s_i \leq 1$. The problem is to pack these items in the fewest number of bins (each bin has unit capacity).
- On-line algorithm: no optimal algorithm, use at least $4/3$ the optimal number of bins. Three simple algorithms that guarantee that the number of bins used is no more than twice optimal.
- Next fit: when processing any item, we check to see if it fits in the same bin as the last item. If it does, it is placed there; otherwise, a new bin is created. $O(N)$.
- First fit: scan the bins in order and place the new item in the first bin that is large enough to hold it. $O(N^2) \rightarrow O(N \log N)$
- Best fit: place a new item in the tightest spot among all bins. $O(N \log N)$

- Off-line algorithm (no guarantee to get optimal solution): allow to view the entire item list before producing an answer. By sorting the items and placing the largest items first.
  - First fit decreasing & best fit decreasing for distinct items, first fit non-increasing and best fit non-increasing for non-distinct items.