4 Design of Algorithms: By Greedy Method

4.1 Greedy algorithms

Reading: CLRS 16.2

- Making change: How to pay x to a customer using the smallest possible number of coins, given the coinage, $c_1 < c_2 < \cdots < c_n$, and a unlimited supply of coins of each denomination?

  A greedy algorithm: Starting with nothing, at every stage we add to the coins already chosen a coin of the largest value available that does not take us past the amount to be paid.

  Consider the American coinage. If $x = 94$, then the change includes 3 quarters, 1 dime, and 1 nickel and 4 pennies. A proof, though surprisingly hard, exists that the proposed greedy algorithm always gives the optimal (minimum) solution for the American coinage.

  The algorithm may not be optimal for other coinages. For example, let $c_1 = 1$, $c_2 = 9$, and $c_3 = 15$. Assume $x = 19$. The greedy algorithm gives a solution with 5 coins. However, the optimal solution has only 3 coins.

- Greedy algorithms are suitable for solving optimization problems with solutions consisting of a sequence of choices.

  Greedy algorithms makes the choice that looks the best at the moment (short-sighted), and never changes the choices made (stubborn). So they may not always give optimal solutions.

  Greedy(C: set) //C is the set of all choices

  $S \leftarrow \emptyset$ //S is the solution set

  while S is not a solution and $C \neq \emptyset$

  $x \leftarrow$ an element in C that optimizes $Select(x)$ //You pick $Select(x)$ in advance

  $C \leftarrow C - \{x\}$

  if $S \cup \{x\}$ is a feasible solution then $S \leftarrow S \cup \{x\}$

  if S is a solution then return S

  else return "No solution found"

- Examples of greedy algorithms:

  Kruskal's algorithm for finding the minimum spanning tree of an undirected edge-weighted graph.

  Dijkstra's algorithm for finding the shortest paths from a node to all other nodes in a directed, edge-weighted graph.

  Approximation algorithms for bin packing (e.g., First-Fit, Best-Fit, First-Fit-Decreasing, and Best-Fit-Decreasing).

  Algorithms for the knapsack problem:

  - 0-1 knapsack problem: All greedy algorithms (e.g., sort by decreasing value and sort by decreasing value per unit weight) are approximate as the problem is NP-complete.
  - Fractional knapsack problem: The greedy algorithm that takes items in the order of the greatest value per unit weight yields an optimal solution

4.2 Scheduling with deadlines

Reading: CLRS 16.5

- Given a single machine and n unit-time jobs, $J_1, \ldots, J_n$. Assume that J_j has deadline d_j (integer) and penalty p_j (integer) which occurs when the deadline is missed. How can we schedule the jobs on the machine so that the total penalty is minimized?

- A greedy algorithm: Schedule jobs with the largest penalty in the latest available time slot that incurs the smallest penalty.

  For the following instance, the greedy algorithm defines a schedule $J_4, J_2, J_3, J_1, J_7, J_6, J_5$ with total penalty of 50.

J_j	J_1	J_2	J_3	J_4	J_5	J_6	J_7
d_j	4	2	4	3	1	4	6
p_j	70	60	50	40	30	20	10

The following pseudocode describes the algorithm:

WLOG, assume $p_1 \geq p_2 \geq \cdots \geq p_n$.
$P \leftarrow 0$ //total penalty so far
for $i \leftarrow 1$ to n $S[i] \leftarrow 0$ //S is the schedule
for $j \leftarrow 1$ to n
    if $\exists k$ such that $S[k] = 0$ and $k \leq d_j$
    then $i \leftarrow \max\{k|S[k] = 0 \text{ and } k \leq d_j\}$
    else $i \leftarrow \max\{k|S[k] = 0\}$
        $P \leftarrow P + p_j$
    $S[i] \leftarrow j$

The worst-case time complexity of the greedy algorithm is $\Theta(n^2)$. However, from the recent homework we know that the algorithm can be implemented using the union-find operations to achieve near-linear time complexity.

- Proof of optimality

  Assume that the optimal schedule looks different from the greedy schedule. We wish to prove that the former can be transformed to the latter without changing the total penalty. We use c_j to denote the completion time of J_j in the current schedule, which is initially the optimal schedule. Assume $p_1 \geq p_2 \geq \cdots \geq p_n$. For any J_j, it is either an early job or a late job in the current schedule. Consider the following procedure:

  for $j \leftarrow 1$ to n
      if J_j is an early job in the current schedule ($c_j \leq d_j$)
      then $T \leftarrow \{J_i|J_i \in [c_j, d_j] \text{ and } i > j\}$
          if $T \neq \emptyset$
          then let J_i be the job in T with the largest c_i
              swap J_i and J_j //step 1
      else //J_j is a late job ($c_j > d_j$)
          let J_i be the latest scheduled job with $i > j$
          swap J_i and J_j //step 2

  Step 1 does not increase the total penalty. This is because after the swap J_j remains to be an early job and J_i is moved to an earlier slot. Step 2 also does not increase the total penalty since J_j remains to be a late job and J_i is moved to an earlier slot. So after applying the above procedure, the optimal schedule is transformed to a greedy schedule while the total penalty is not increased. Since it is impossible to have a decreased total penalty, we have a greedy schedule with the same total penalty as the optimal schedule. Therefore, the greedy schedule is optimal.

- Remark: When multiple optimal solutions exist for some inputs, we may use "prove by transformation" to prove optimality of a greedy algorithm. Consider an optimal solution for some instance. Assume it looks different from the greedy solution for the same instance. Show that the optimal solution can be transformed step by step to the greedy solution without increasing the minimized value of the objective function or without decreasing the maximized value of the objective function. Thus, we have proved the optimality of the greedy algorithm.

4.3 Single machine scheduling to minimize average completion time

- Given a single machine and a set of n jobs labeled with 1 through n. Also given is the processing time of each job, $p_1, \ldots, p_n$, To schedule a job j on the machine, an algorithm has to determine the start time s_j of the job. Keep in mind that the machine can run only one job at any time. Obviously, once s_j is picked, the completion time c_j is just $c_j = s_j + p_j$. The goal of the problem is to schedule all jobs on the single machine to minimize the average completion time $\frac{1}{n}\sum_{j=1}^{n} c_j$, which is equivalent to minimizing the sum of completion time $\sum_{j=1}^{n} c_j$.

- A greedy algorithm: Shortest-Job-First (SJF)

  Sort the jobs in increasing processing times and schedule in the sorted order.

- Proof of optimality of SJF

  Proof: Assume that for any instance, the OPT schedule is different from the SJF schedule. Then there must be two jobs, J_a and J_b with $p_a > p_b$, such that J_a is scheduled earlier than J_b in the OPT schedule. Now swap these two jobs while maintaining the order of the remaining jobs. It can be proved easily that the new schedule has a sum of completion time less than that in the OPT schedule. Thus a contradiction to that OPT is optimal.

- Remark: When the optimal solution for any input is unique, we may use "prove by contradiction" to prove optimality. Again, assume that the optimal solution is different from the greedy solution. Then show that by making some change to the optimal solution there is a better solution than the optimal. Thus, a contradiction.

4.4 A parallel scheduling problem

- Motivation: In a parallel system, a job can be assigned to an arbitrary number of processors to execute. The more processors assigned to the job, the less the computation time. This is the linear speedup in computation. However, having multiple processors working on a job will incur an overhead cost such as communication and synchronization. The more processors, the more the overhead cost. This is the linear slowdown in overhead.

- A mathematical model: Assume job J_j has processing requirement of p_j. This is given. If in scheduling J_j, k_j processors are assigned, then the execution time $t_j = p_j/k_j + (k_j - 1)c$, where the term p_j/k_j is the computation time and the term $(k_j - 1)c$ is the overhead with c being the constant overhead per processor. Note that if $k_j = 1$, $t_j = p_j$. This is the scheduling of sequential jobs.

- Problem formulation:

  Input: A parallel system of m identical processors and a sequence of n independent jobs $1, \ldots, n$ where p_j is the processing requirement of job j.

  Output: A schedule with the minimum makespan (i.e., $C = \max_j\{c_j\}$).

- An online scheduling algorithm is required to process the jobs in the order of 1 through n and for each job j, it chooses k_j, the number of processors to execute j simultaneously, and s_j, the start time of j on some k_j processors.

- A greedy algorithm: Shortest-Execution-Time (SET)

  For job j, define function $t_j(k) = p_j/k + (k-1)c$. It is easy to verify that $t_j(k)$ is minimized at $k = \sqrt{p_j/c}$. Since k_j has to be an integer in $[1, m]$, let $k_j = \min\{m, \lfloor\sqrt{p_j/c}\rfloor\}$ if $t_j(\lfloor\sqrt{p_j/c}\rfloor) \leq t_j(\lceil\sqrt{p_j/c}\rceil)$ and $k_j = \min\{m, \lceil\sqrt{p_j/c}\rceil\}$ otherwise. Once the k_j that minimizes t_j is computed, the job is assigned to the k_j processors that give the job the earliest start time.

  An example: $m = 3$, $n = 6$, and $c = 1$. (In the final schedule, $C = 14$.)

j	1	2	3	4	5	6
p_j	4	1	4	1	9	4
k_j	2	1	2	1	3	2
t_j	3	1	3	1	5	3
s_j	0	0	3	1	6	11

- Another greedy algorithm: Earliest-Completion-Time (ECT)

  For job j, choose k_j processors and the start time s_j such that the completion time $c_j = s_j + t_j = s_j + p_j/k_j + (k_j - 1)c$ is the earliest. For the above example, $k_1 = 2$, $s_1 = 0$, $k_2 = 1$, $s_2 = 0$, $k_3 = 1$, $s_3 = 1$, $k_4 = 1$, $s_4 = 3$, $k_5 = 2$, $s_5 = 4$, $k_6 = 1$, $s_6 = 5$, with $C = 9.5$. This shows that SET is not optimal.

  Is ECT optimal? No.