Max-Profit Scheduling: Insights into Proofs and DP Methods

- **Problem 15-7 (p 369).** Scheduling to maximize profit

  Given a set $1,2,...n$ of jobs, each requiring processing time $t_i > 0$ and will receive a payoff $p_i > 0$ if finished by deadline d_i, select a subset to maximize total payoff
- consider an example, and whether to include last job or not
- in DP, we max/minimize some cost property (possibly with constraints) of the solution, as well as construct the solution itself
- assuming a DP approach is possible:
  - what assumptions in the ordering of the jobs can be useful?
    - why is an ordering necessary for DP?
    - possibilities: order by deadline, payoff, value (=payoff/time)
    - what (implied) constraints are there?
  - will the chosen ordering always lead to an optimal solution?

Deadline-Driven Scheduling

- idea: assume jobs are ordered in increasing deadline, i.e. $1 \leq d_1 \leq d_2 \leq \ldots \leq d_n$,

  then the max-profit scheduling problem can be expressed as:

  find a sub-sequence $(i_1, i_2, \ldots i_m)$, where $1 \leq i_1 < i_2 < \ldots < i_m \leq n$ s.t.
  the payoff $\Sigma_{j=1}^{m} p_{i_j}$ is maximized
  with the constraint that $\Sigma_{j=1}^{k} t_j \leq d_k$, for each $1 \leq k \leq m$
- will this idea work?
- try to construct a sequence of jobs which could be satisfied in non-deadline order, but not in deadline order

t:	2	2	1	1
d:	2	4	4	11

(deadline order $(1, 2, 3, 4)$ fails)

- non-deadline orderings $(3, 1, 2, 4)$ and $(4, 3, 2, 1)$ also fail

Proving the Procrastination Lemma

- statement of the Procrastination Lemma:

  given a permutation $(i_1, i_2, \ldots, i_m)$ of $(1, 2, \ldots, m)$, and $0 < d_1 \leq d_2 \leq \ldots \leq d_m$:

  if $\Sigma_{j=1}^{k} t_{i_j} \leq d_{i_k}$ for each $1 \leq k \leq m$, then: $\Sigma_{j=1}^{k} t_j \leq d_k$ for each $1 \leq k \leq m$
- prove the converse (why? cf. prof by contradiction)

  assume there is a k s.t. $\Sigma_{j=1}^{k} t_j > d_k$; show $\Sigma_{j=1}^{K} t_{i_j} > d_{i_K}$ for some K
- approaches:
  - how can we use the assumption? i.e. $(\Sigma?) \geq \Sigma_{j=1}^{k} t_j > d_k \geq (?)$
  - clues from the previous example?
- proof:
  - (for later)

Proof of the Procrastination Lemma

- assume there is a k s.t. $\Sigma_{j=1}^{k} t_j > d_k$; show $\Sigma_{j=1}^{K} t_{i_j} > d_{i_K}$ for some K
- idea: the non-ordered sequence must fail when it has just covered all the jobs in $(1, 2, \ldots, k)$

  i.e. choose K s.t. $\{i_1, i_2, \ldots, i_K\} \supseteq \{1, 2, \ldots, k\}$ and $1 \le i_K \le k$
- then the non-ordered sub-sequence fails to meet the deadline at job i_K:

$$\begin{array}{rcll} \Sigma_{j=1}^{K} t_{i_j} & \ge & \Sigma_{j=1}^{k} t_j & \text{, as } \{i_1, i_2, \ldots, i_K\} \supseteq \{1, 2, \ldots, k\} \\ & > & d_k & \text{, by assumption} \\ & \ge & d_{i_K} & \text{, as } i_K \le k \end{array}$$

DP Solution to Max-profit Scheduling: 1st Attempt

- idea: similar approach to the maximum decreasing subsequence problem
- let P_i denote the maximum accumulated payoff for any sub-sequence of jobs 1 to i (including i)
  - $P_i = P_j + p_i$, where $0 \leq j < i$ maximizes $P_j + p_i$ under constraint $T_j + t_i \leq d_i$
  - T_j is accumulated time associated with P_j (similarly defined as $T_i = T_j + t_j$)
  - $P_i = T_i = 0$ if no such j exists, or $i = 0$
- best solution is the max. of $\{P_1, P_2, \ldots, P_n\}$
- a counterexample was found! (lesson: always (exhaustively) test a solution!)

d:	4	5	5	6	7	17	19
t:	2	3	2	1	3	5	2
p:	2	10	2	5	4	9	4

  - DP algorithm: payoff 30, solution vector 6b $(0,1,3,5,6)$
  - Brute Force algorithm: payoff 32, solution vector 7a $(1,3,4,5,6)$
    - iterates over all possible 2^n solution vectors!
- optimal sub-structure property does not hold (consider 1st 2 jobs)

The ‘Potential’ of Sub-solutions in Dynamic Programming

- in the presence of constraints, as well as its value, a sub-solution may have other attributes determining its *potential* to form a larger solution
  - maximum decreasing subsequence problem, what attributes of L_i (longest dec. subsequence ending at position i) have?
  - what about P_i?
- DP must be extended to include the best sub-solutions for each possible potential
- reconsider the sub-solutions up to job 2 in the previous example:

d:	4	5	5	6	7	17	19
t:	2	3	2	1	3	5	2
p:	2	10	2	5	4	9	4

Extended DP Solution to Max-profit Scheduling

- let $P_{i,T}$ denote the maximum accumulated payoff for any sub-sequence of jobs 1 to i (including i) having an accumulated time of T
  - assuming the job times are bounded ($t_i \leq t_{\max}$), then $T \leq nt_{\max}$, i.e. $T = O(n)$
- then $P_{i,T+t_i} = P_{j,T} + p_i$, where $0 \leq j < i$ is chosen to maximize $P_{j,T} + p_i$, under constraint $T + t_i \leq d_i$
  - does the optimal sub-structure property hold for this definition of a solution?
- total solution is maximum of $P_{i,T}$ over $1 \leq i \leq n$ and $0 \leq T < nt_{\max}$
- an implementation using 2-D arrays is in smp.c
  - what is the complexity of this algorithm?
- what data structures could improve its efficiency?
  - would this reduce its complexity?

Data Structures in Practice

Ref: `man(3C++)`

- the C++ Standard Template Library (STL) provides generic definitions of:
  - priority_queue: priority queue implemented using a heap
  - map: associative array, implemented using a sorted (key,value) array
  - hash_map (not standard): associative array, implemented using hash tables with chaining
    - requires a equality, rather than a less than, comparison method
    - when would you use this over map?
    - in `G++ 3.4.4`, default integer hash function appears to use the division method $h(k) = k \bmod m$ where m is taken from a table of chosen primes
    - default string hash function repeats h = 5∗h + s[i]; over each string element
    - the table gets automatically resized when $>$ 75% full (why?)
- Java has similar generic class definitions
- when should you use standard data structure libraries instead of implementing your own ?

Cdt: A Container Data Type Library

Ref: Cdt web page

- has C/C++ interface with a cleaner abstraction of the underlying data structure
- DtSet corresponds to hash_map
  - also uses chaining and automatic resizing
    - chaining is augmented with a move-to-front heuristic for often-searched keys
  - default string hash function repeats h = h + (s[i] + s[i−1]<<8) ∗ 17109811; over each pair of string elements
  - claims $O(1)$ access time for a good hash function
- claims a reduction in the number of comparisons of $6 - 10\times$, with an overall speedup of $2 - 3\times$ over hash_map
  - the same hash function was used
- why aren't better methods (e.g. double hashing) used?
- and the more sophisticated data structures?