Introduction to Algorithms December 2, 2002
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 28

Problem Set ∞ Solutions

Problem ∞-1. Optimal scheduling

(a) For (1) the schedule in which task 1 runs first, followed by task 2:
$c_1 = p_1 = 3$ and $c_2 = p_1 + p_2 = 8$
average completion time $= (c_1 + c_2)/2 = 11/2 = 5.5$
For (2) the schedule in which task 2 runs first, followed by task 1:
$c_2 = p_2 = 5$ and $c_1 = p_2 + p_1 = 8$
average completion time $= (c_1 + c_2)/2 = 13/2 = 6.5$

(b) Run tasks in increasing order of processing time. This can be done by sorting the elements using heap sort or merge sort and then scheduling them in the order of increasing processing times. This algorithm takes $O(n \lg n)$.

This algorithm uses a greedy strategy. It is shown to be optimal as follows: $c_{avg} = \frac{c_1+c_2+c_3+\cdots+c_n}{n}$ This cost can also be expressed as $[p_1 + (p_1 + p_2) + (p_1 + p_2 + p_3) + \cdots + (p_1 + p_2 + \cdots + p_n)]\frac{1}{n}$. Note that p_1 is included n times, p_2 appears $n - 1$ times, etc. As a result, p_1 should have the shortest processing time, then p_2, etc. Otherwise, you could cut and paste in a shorter processing time and produce a faster algorithm. As a result, the greedy property holds and our algorithm is correct.

(c) This problem also exhibits the greedy property which can be exploited by running tasks in increasing order of remaining processing time. Use a priority queue which prioritizes based on the amount of time remaining. Each time a new task comes up, insert it into the queue and if it would take less time to do that task then the one you are on, do the shorter task. Each time you finish a task, the next task you should do is the one with the least remaining time until completion. The priority queue can be maintained in $O(n \lg n)$ time.

This algorithm minimizes the average completion time and the proof is similar to the one in the previous part. If we do not schedule using the greedy algorithm based on remaining processing time, then we will be able to swap two time slots which would then improve the sum of the completion times and thus result in a contradiction. For example, assume you have two tasks at time t, where task i has x processing time remaining and j has y processing time remaining where $x > y$. Assume for the purposes of contradicction that the optimal answer has task i running before task j. If i is done before j then $c_i = t + x$ and $c_j = t + x + y$. The average completion time is $\frac{2t+2x+y}{2}$. However if j were done before i, then $c_i = t + y + x$ and $c_j = t + y$. The average completion time is now $\frac{2t+2y+x}{2}$ which is less the average completion time for the "optimal" solution since $x > y$. As a result, the task with the lowest time remaining should be done first.

Problem ∞-2. Paintball routes

We assume the probabilities that each bridge will fail are independent. The probability P_S that a path with edges of failure probabilities $p_1, p_2, \cdots, p_k$ will *not* fail is $P_S = (1-p_1)(1-p_2)\cdots(1-p_k)$. Therefore, the probability of failure P_F is $1 - P_S$. We want to minimize P_F for the nodes in the graph, so we can do something akin to a shortest paths algorithm, which attempts to minimize the failure probability. In fact, we can reduce the problem to a classical shortest path problem with nonnegative weights by observing that minimizing P_F is equivalent to minimizing $-\log(P_S) = \sum_{i=1}^{k} -\log(1-p_i)$. Setting the weight of an edge to be $w_e = -\log p_e \geq 0$, we observe that a path of minimum failure probability is a path of minimum total weight (with respect to the weights w_e). We could thus simply apply Dijkstra with the weights w_e. Since we haven't really discussed whether our computational model allows the computation of logarithms we can simply modify Dijkstra to work directly on the p_e's instead of the w_e's but perform exactly the same updates as Dijkstra's algorithm would do on the w_e's. Correctness will simply follow from Dijkstra's correctness. Our variant of Dijkstra's algorithm will maintain estimates on the failure probabilities which we will relax. Here is the pseudocode.

```
NOT-FALL(G, w, s)
1  Initialize-NF-Single-Source(G, s)
2  S ← NIL
3  Q ← V[G]
4  while Q ≠ NIL
5       do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8               do NF-RELAX(u, v, w)
```

```
INITIALIZE-NF-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V[G]
2       do d[v] ← 1
3          pi[v] ← NIL
4  d[s] ← 0
```

```
NF-RELAX(u, v, w)
1  if d[v] > 1 − (1 − d[u])(1 − w(u, v))
2     then d[v] ← 1 − (1 − d[u])(1 − w(u, v))
3          π[v] ← u
```

Running time and correctness follow directly from Dijkstra's algorithm as stated earlier by using

the relationship between p_e and w_e.

Problem ∞-3. Computing partial mins

(a) We first note that if we could maintain a data structure that supports INSERT, DELETE and FIND-MIN in $O(\lg n)$ (n being the number of elements in the data structure), then we would be done. This is because we could then store all the elements visible through the window in this data-structure. *mins*$[i]$ can be computed by FIND-MIN, and sliding the window could be performed by a DELETE followed by INSERT.

These operations can be performed by a balanced BST like AVL-trees or Red-Black trees. INSERT and DELETE take $O(\lg n)$ time on a balanced BST. FIND-MIN can be performed in $O(\lg n)$ time on a BST by going along the left spline of the tree till we can go no more. To start with, the balanced BST can be built for the first position of the window in time $O(m \lg m)$. And then computing *mins*$[i]$ followed by a sliding involves the 3 operations FIND-MIN, DELETE and INSERT, all of which can be done in $O(\lg m)$ time. Hence the total time to compute the array *mins* in the manner specified is $O(m \lg m) + (n - m + 1)O(\lg m) = O(n \lg m)$

(b) The worst case time taken by SLIDE is when the outgoing element is green. In this case the sub-routine COLOR-RED is executed which takes $O(m)$ time, because it has to color all the m elements in the window red. The other operations in SLIDE take $O(1)$ time. Hence, the worst case time taken to compute the value of a single element of the array *mins* is $O(m)$.

(c) We observe that each element of the entire array A is colored green and then red at most once. Moreover, the COLOR-RED function takes time $O(1)$ for each element that it colors red. Since each element is colored red at most once, the sum of the times taken by all the calls to COLOR-RED is at most $O(n)$. Each time SLIDE is called, its operations (other than the call to COLOR-RED) take $O(1)$ time. It can easily be seen that all other operations performed by COMPUTE-MINS take only time $O(n)$ on the whole. Hence the total time taken by the entire algorithm is $O(n)$. Thus the amortized time taken to compute the minimum for each set of m consecutive elements is $\frac{O(n)}{n-m+1}$ which is $O(1)$ since $m < n/2$.

Problem ∞-4. Weird Multiplication

We build three tables $M(\sigma) = [m_{ij}(\sigma)]$ of boolean values, where $\sigma = a, b, c$ and $1 \leq i \leq j \leq n$. The value $m_{ij}(\sigma)$ will be 1 if it is possible to parenthesize $x_i \ldots x_j$ in such a way that the value of the resulting expression is σ, and 0 otherwise. The initial conditions are $m_{ii}(\sigma) = 1$ iff $x_i = \sigma$. The recurrence for, say, $m_{ij}(a)$, where $1 \leq i < j \leq n$, is obtained from the multiplication table. We note that there are three entries equal to a in the table; specifically $ac = a$, $bc = a$ and $ca = a$. So

$$m_{ij}(a) = \left(\bigvee_{k=i}^{j-1} m_{ik}(a) \wedge m_{k+1,j}(c) \right) \vee \left(\bigvee_{k=i}^{j-1} m_{ik}(b) \wedge m_{k+1,j}(c) \right) \vee \left(\bigvee_{k=i}^{j-1} m_{ik}(c) \wedge m_{k+1,j}(a) \right).$$

The recurrences for $m_{ij}(b)$ and $m_{ij}(c)$ are similar. It is easy to turn this into an $O(n^3)$ algorithm which computes the entire contents of the three tables. The value we are looking for is $m_{1n}(a)$.