Practice Quiz 2 Solutions

Problem Practice-1. LRU data structure

Managing the primary memory of a virtual-memory system with LRU page replacement can be viewed as a dynamic data structure problem. The data structure manages a set $S = \{1, \ldots s\}$ of s fixed-size slots of primary memory. The virtual memory can be viewed as a set $P = \{1, \ldots, p\}$ of virtual-memory pages. At any time a subset of P having size at most s is resident in the s slots of primary memory. The job of the data structure is to maintain a dynamic mapping so that the system can identify whether a user's reference to a page resides in a slot of primary memory, and if not, drop the least-recently used page and replace it with the referenced page.

The LRU data structure must therefore support the following operations:

- INSERT(q, t): Insert virtual page $q \in P$ into slot $t \in S$ of primary memory. The slot t must be empty.
- USE(q): Return the slot $t \in S$ containing virtual page $q \in P$, or else NIL if q does not reside in primary memory.
- DROP$()$: Remove the least-recently used page $q \in P$ from its slot $t \in S$, and return the now-empty slot t.

Describe briefly why these operations suffice to implement the LRU page replacement policy. Give an efficient implementation of this dynamic set.

Solution: This problem had a very simple $\Theta(1)$ time implementation without making any simplifying assumptions. It should be assumed that for a paging system on any real computer, anything less than a $\Theta(1)$ time for any of those operations would simply be unacceptable.

We implement LRU by the use of

- A doubly linked list, call it L.
- A hash table, call it H.

The two data structures contain a structure holding the virtual page number $q \in P$ and the corresponding physical page slot it occupies $t \in S$ for all pages that are currently occupying physical memory slots. The hash table entries also hold a pointer to the linked list node corresponding to the virtual page q. The space requirement for these is therefore $O(s)$.

In order to implement LRU, we need a complete ordering in terms of last access for every page currently occupying a physical memory slot. As was pointed out in lecture, a doubly-linked list coupled with a move to front heuristic provides exactly this. Doubly-linked lists are convenient as cutting and pasting an element can be done in $\Theta(1)$ time, if we already have pointers to the element to cut and the element before or after were we will paste. Thus in order to support the DROP and USE operations in constant time, we use such a list, and *arbitrarily* define ordering to be from most recently used page at the head to least recently used at the tail. We thus also need to keep track of the head and tail of the doubly linked list so as to know where to cut and paste.

However, apart from the LRU information, we need to the ability to lookup a certain virtual page and quickly extract the physical page slot it occupies. The INSERT procedure inserts this mapping into the data structure, whereas the USE procedure returns the mapping. This is an instance of the dictionary problem, which we know how to efficiently solve using a hash table. We use a chained hash table as described in chapter 12 of CLR. The average access time in the table is $\Theta(1+\alpha)$, where α is the load factor defined as $\frac{n}{m}$ where n is the number of things to store and m is the number of slots. We therefore decide to use a table of size $\Theta(|S|)$ to ensure that the load factor remains small. As for a hash function, we can simply use a random hash function drawn from a collection of universal hash functions, or any of the other hash functions presented in CLR.

Assuming that we have implemented such a list and hash table, the DROP operation can be implemented as follows:

$procDrop()$
$q \leftarrow tail[L].t$
if $q = NIL$
  return NIL
  else $tail[L] \leftarrow tail[L].previous$
  $tail[L].previous = NIL$

The USE is then simply (assuming that the hash function HASH returns the hash key of its argument):

USE(q)
ref $\leftarrow$ CHAINED-HASH-SEARCH$(H, \text{HASH}(q))$
if *ref* $= NIL$
  error "Page not in primary memory"
  else *ref*$.node.next.previous \leftarrow$ *ref*$.node.previous$
    ref$.node.previous.next \leftarrow$ *ref*$.node.next$
    ref$.node.previous \leftarrow NIL$
    ref$.node.next \leftarrow head[L]$
    $head[L] \leftarrow$ *ref*
    return *ref*$.t$

Finally, the INSERT becomes:

INSERT(q, t)
$node \leftarrow$ NEW$(ListNode)$
$node.next \leftarrow head[L]$
$node.previous \leftarrow NIL$
ref $\leftarrow$ NEW$(HashNode)$
ref$.node \leftarrow node$
ref$.t \leftarrow t$
ref$.q \leftarrow q$

The three functions that we asked you to implement, would be provided as a front end to your system, therefore, it is conceivable that DROP could be called an arbitrary number of times, without being immediately followed by an INSERT. With this in mind, the above functions *do not* suffice for a correct implementation of LRU. We need some way to first look if there is an empty slot of phyical memory and only if one does not exist should we drop the tail of the list. Modifying the DROP and INSERT procedures to do this is easy, by using a direct mapping table and doubly linked list of free physical memory slots, exactly as we do above for the LRU. Since we only have $|S|$ physical memory slots, we do not need to use a hashtable. Then, the DROP and INSERT procedures operate on all four data structures.

Unfortunately, a number of people assumed that it was possible to keep an array of size $\Theta(|P)$ with an entry for each virtual page and store all information needed, therefore obtaining an $O(1)$ time. This solution however bears little physical realism. Most modern computer systems have a 64 bit virtual address space, and 4KB pages. Therefore, a direct mapping lookup table for each virtual page would require $\frac{2^{64}}{2^{12}} = 2^{52}$ entries. If only one integer of information was kept in each table entry, one would need 16384TB...

A number of other people used some form of balanced tree to do lookups, leading to logarithmic running times for some or all of the procedures. However, again, such a solution is not desirable, since a VM system is one of the most heavilly loaded components of the operating system.

Some people claimed that the move to front heuristic combined with the locality that program memory references exhibit, allows us to use the LRU linked list to implement the USE procedure by a linear scan. However, an operating system runs a number of programs at the same time, therefore the system as a whole exhibits less locality. Furthermore, (also for the people who claimed USE might not be used too much) this proceudre is probably the most used of the system for obvious (!) reasons. All these solutions might work for Windowz, but not on a usable OS.

Other sources for loss of points were forgetting to update the LRU information when USE was called. LRU does indeed stand for least recently *used*.

Problem Practice-2. Random-number generation

The array $A[1 \, .. \, n]$ contains a probability distribution over the set $\{1, 2, \ldots, n\}$; that is, we have

$A[i] \geq 0$ and $\sum_{i=1}^{n} A[i] = 1$. We wish to generate a random integer X in the range $1 \leq X \leq n$ such that

$$\Pr\{X = i\} = A[i] .$$

A uniform random-number generator UNIFORM() is available which generates in constant time a real number y uniformly in the range $0 \leq y < 1$. Using UNIFORM() as a subroutine, devise an efficient algorithm to generate a random integer according to the distribution specified by A.

Your algorithm may include an initialization phase to preprocess the array A. After the preprocessing phase, the user can make any number of calls to your random-number generator, each of which should return a random integer according to the distribution A. The highest priority in your design is to make your random-number generator run as fast as possible, but your preprocessing phase should be efficient as well. Analyze both the time for preprocessing and the time for actual random-number generation.

Solution:

First, we present an algorithm based on generating random variates from a given probability distribution by inverting the cumulative distribution function. This algorithm uses $\Theta(n)$ preprocessing time and then takes $\Theta(\lg n)$ time for actual generation of a random number according to the distribution A.

```
RAND-BINARY(A)
 1. C[1] ← A[1]
 2. for i = 2 to n
 3.     C[i] ← C[i − 1] + A[i]
 4. u ← UNIFORM()
 5. return BINARYSEARCH(C, 1, n, u)
 6. end
```

BINARYSEARCH in line 5 returns r such that $C[r-1] \leq u < C[r]$.

This implies $\Pr\{X = r\} = \Pr\{C[r-1] \leq u < C[r]\} = C[r] - C[r-1] = A[r]$, which establishes correctness. RAND-BINARY takes $\Theta(n)$ preprocessing time (lines $1-3$), while lines $4-5$ take $\Theta(\lg n)$ time.

Our next algorithm is based on expressing a given probability distribution as a weighted sum of some other probability mass functions that are easy to generate from (using the inversion technique mentioned above), and then selecting one of these probability functions according to the weights followed by generating (easily) from it. We give an algorithm that performs the preprocessing in time $\Theta(n^2)$, but generates random numbers from the distribution A in time $\Theta(1)$. The idea is that if we write a probability mass function as an equally weighted mixture of $n-1$ probability mass functions Q_k, e.g. $A = \frac{1}{n-1}\sum_{k=1}^{n-1} Q_k$, with the special property that each Q_k assigns positive probabilities to at most two components, then in order to sample from A, all we need to do is to generate a uniform random variate to select a distribution Q_j, and then to generate another uniform random variable to select the correct positive component of Q_j. In fact we will use the fact that for a uniformly generated random variate u, $(n-1)u - \lfloor (n-1)u \rfloor$ is independent of u and thus we only need to generate one uniform random variate to sample from distribution A.

One way to implement this idea is to first create $n-1$ buckets such that each bucket contains exactly two elements from $\{1, 2, \ldots, n\}$ and the total mass according to the distribution A in each bucket is $\frac{1}{n-1}$.

PREPROCESS(A)

1. $A' \leftarrow [(\sigma(1), A[\sigma(1)]), (\sigma(2), A[\sigma(2)]), \ldots, (\sigma(n), A[\sigma(n)])]$, where $A[\sigma(1)] \leq \cdots \leq A[\sigma(n)]$ and σ is a permutation of $\{1, 2, \ldots, n\}$.
2. for $i \leftarrow 1$ to $n-1$
3. $\quad B[i][1].element \leftarrow$ FIRST$(A').element$
4. $\quad B[i][1].mass \leftarrow$ FIRST$(A').mass$
5. $\quad slack \leftarrow \frac{1}{n-1} -$ FIRST$(A').mass$
6. $\quad B[i][2].element \leftarrow$ LAST$(A').element$
7. $\quad B[i][2].mass \leftarrow slack$
8. $\quad massLast \leftarrow$ LAST$(A').mass - slack$
9. $\quad$ REMOVE$(A',$ FIRST$(A'))$
10. $\quad$ REMOVE$(A',$ LAST$(A'))$
11. $\quad$ ADDINORDER$(A', (B[i][2].element, massLast))$

Line 1 creates a list A' of tuples $(element, mass)$ from the elements of the array A, where $element \in \{1, 2, \ldots, n\}$, $mass = A[element]$, and the tuples in A' are sorted on mass. This operation takes time $\Theta(n \lg n)$. FIRST takes a list as argument and returns the element at the head of the list in time $\Theta(1)$. LAST likewise returns the last element in the list in time $\Theta(1)$. REMOVE takes as argument a list and an element in the list and removes the element from the list. This takes time $\Theta(n)$ if done using a linear scan. ADDINORDER inserts the given element in the given list such that the new list is still sorted on mass. This also takes time $\Theta(n)$. Hence, in all, the preprocessing stage takes $\Theta(n^2)$ time.

RAND-CONSTANT(A)

1. PREPROCESS(A)
2. $u \leftarrow$ UNIFORM$()$
3. $bucket \leftarrow 1 + \lfloor (n-1)u \rfloor$
4. $index \leftarrow u - \frac{bucket-1}{n-1}$
5. if $B[bucket][1].mass > index$ then
6. $\quad$ return $B[bucket][1].element$
7. else return $B[bucket][2].element$

The preprocessing time in the above algorithm can be improved to $\Theta(n \lg n)$ using a min-heap and a max-heap, though a slight modification is needed to the data structures.

Correctness follows by observing that the sum of masses associated with element $i \in \{1, 2, \ldots, n\}$ over all the buckets is $A[i]$ and $\Pr\{\cup(u \in I_m)\} = \sum \Pr\{u \in I_m\}$, if no two distinct intervals I_n, I_p overlap.

Comments on solutions: Almost every student used an approach similar to the first one described above, and computed the cumulative distribution function. Most got the idea of using binary search or a balanced binary search tree and devised $\Theta(\lg n)$ algorithms for the actual generation. However full credit was not given to these solutions. Some students linearly scanned the cumulative distribution array and thus lost some points. Some folks lost credit by making an assumption about a small number of significant figures, though essentially they needed an arbitrarily large amount of storage space to implement their algorithm to work for the problem as specified. A very few students got near optimal performance. Some needlessly used red-black trees and got worse performance bounds than efficiently implemented simpler data structures would have.

Problem Practice-3. Music recognition

A ***recorder*** is a simple blown musical instrument that sounds much like a flute. Prof. Cary Oki has recently programmed his computer to listen to a stream of music from a recorder and convert it into a set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ of (possibly overlapping) time intervals, where each interval corresponds to the duration of a note transcribed by the system. Moreover, the professor has developed a heuristic that gives for each interval $T_i \in \mathcal{T}$, a metric m_i indicating how likely it is that note i was played by the recorder. The larger m_i, the greater the confidence that note i was played by the recorder.

(a) The professor would like to determine which notes are played by the recorder. Since the recorder can produce only one note at a time (when properly played), if two intervals have a common intersection, one of the notes must be spurious (produced by background noise). Give an efficient algorithm to determine a set $\mathcal{S} \subseteq \mathcal{T}$ of nonoverlapping intervals (ostensibly corresponding to the notes played by the recorder) that maximizes

$$\sum_{T_i \in S} m_i \ .$$

Solution: We will solve this using dynamic programming. Let $start(T_j)$ and $end(T_j)$ denote the start and end time, respectively, of interval T_j. First, sort the intervals in increasing order of start time. This takes $O(n \lg n)$ time. Of course, we can avoid this step if the intervals are already sorted in this order. This seems reasonable since the recorder encounters the time intervals in increasing order of their start times.

Our subproblems have the following form: find a non-overlapping set of intervals $S \subseteq \{T_j, T_{j+1}, \ldots, T_n\}$ such that the quantity (henceforth called the objective function)

$$\sum_{T_i \in S} m_i$$

is maximized. Thus, there are n subproblems, one for each $1 \leq j \leq n$. Let n_j be the smallest value of j' such that $start(T_{j'}) > end(T_j)$. That is, if the instrument is played during interval T_j, then the earliest interval that the instrument can be played next is T_{n_j}. Let ϕ_j be the maximum value of the objective function for the subproblem $\{T_j, T_{j+1}, \ldots, T_n\}$. The answer to the original problem is ϕ_1.

In any optimal subset of non-overlapping intervals for the subproblem $\{T_j, T_{j+1}, \ldots, T_n\}$, there are two possible cases for time interval T_j. It is easy to observe the optimal substructure in both these cases.

Case 1: The instrument is not played during interval T_j. Then, the value of the objective function for the optimal subset is equal to ϕ_{j+1}.

Case 2: The instrument is played during interval T_j. Then, the earliest interval during which it can be played next is T_{n_j}. In this case, the value of the objective function for the optimal subset is $m_j + \phi_{n_j}$.

Combining the above two cases, we have

$$\phi_j = \max(\phi_{j+1}, m_j + \phi_{n_j})$$

The boundary condition for this dynamic program is given by $\phi_n = m_n$. The table can be filled up in a bottom-up fashion in order of decreasing value of j. Computation of each table entry takes $O(1)$ time if the n_j's are precomputed. Hence, the time to fill up the table is $O(n)$. Precomputing the n_j's for all j takes $O(n \lg n$ time (try to figure out how to do this using binary search !), so that the overall running time is $O(n \lg n)$.

(b) The professor now wishes to extend his algorithm to recorder quartets, which consist of soprano, alto, tenor, and bass instruments. He upgrades his heuristic to give for each interval $T_i \in \mathcal{T}$, a metric m_{ik} indicating how likely it is that note i was played by the recorder k for $k = 1, 2, 3, 4$. Give an efficient algorithm to determine four disjoint sets $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4 \subseteq \mathcal{T}$, where each S_k contains nonoverlapping intervals (ostensibly corresponding to the notes played by the four respective recorders), that maximizes

$$\sum_{k=1}^{4} \sum_{T_i \in S_k} m_{ik} .$$

Solution:

Observe that if we allow two or more instruments to play during the same time interval T_j, then the optimal solution is obtained by solving separately for each instrument, using the algorithm for part (a).

The problem is more complicated when two instruments cannot play during the same time interval. However, note that this allows two instruments to play during the same *time instant*, e.g., when two instruments play during two overlapping time intervals. We will solve this by generalizing the dynamic program from part (a).

Let $\phi_{j_1,j_2,j_3,j_4,j}$ be the maximum value of the objective function for the subproblem $\{T_j, T_{j+1}, \ldots, T_n\}$ under the restriction that the earliest interval during which instrument k *can play* is j_k for $1 \leq k \leq 4$. Note that j_k is at least j for each $1 \leq k \leq 4$ and the answer to the original problem is $\phi_{1,1,1,1,1}$. We will divide the number of table entries into $2^4 = 16$ categories depending on whether j_k is equal to or strictly greater than j for $1 \leq k \leq 4$. We give the dynamic programming equations for the case when $j_1 = j_2 = j$ and $j_3, j_4 > j$. You are encouraged to work out the equations for the other cases.

In the case when $j_1 = j_2 = j$ and $j_3, j_4 > j$, there are three possible cases for time interval T_j. It is easy to observe the optimal substructure in each of these three cases.

Case 1: No instrument is played during interval T_j. Then, the value of the objective function in this case is equal to $\phi_{j_1+1,j_2+1,j_3,j_4,j+1}$.

Case 2: Instrument 1 is played during interval T_j. Then, the earliest interval during which it can be played next is $T_{n_{j_1}}$. In this case, the value of the objective function is equal to $\phi_{n_{j_1},j_2+1,j_3,j_4,j+1}$.

Case 3: Instrument 2 is played during interval T_j. Then, the earliest interval during which it can be played next is $T_{n_{j_2}}$. In this case, the value of the objective function is equal to $\phi_{j_1+1,n_{j_2},j_3,j_4,j+1}$.

As before, $\phi_{j_1,j_2,j_3,j_4,j}$ will be the maximum of the objective function values in each of the above cases. The boundary condition is given by $\phi_{n,n,n,n,n} = \max\limits_{1 \leq k \leq 4} m_{nk}$.

Note that there can be at most $4 + 1 = 5$ cases for each table entry (the largest number of cases occurs when $j_1 = j_2 = j_3 = j_4 = j$). Thus, computation of each table entry takes $O(1)$ time. The number of table entries is at most n^5, so that the table filling time is $O(n^5)$ (try to figure out the order in which you need to fill up the table in a bottom-up approach !). Note that this dominates the $O(n \lg n)$ time for sorting of the intervals and precomputation of the n_j's. Thus, the overall running time is $O(n^5)$.

Remark: The coordinate j in the above dynamic program can be removed without changing the essential structure of the dynamic programming equations. Then, the running time becomes $O(n^4)$.

Problem Practice-4. Party, party, party!

Students at the Monotonic Institute of Technology are reluctant to go to a party if they don't know many people there. Moreover, cycles of indecision lead to situations where Alice will go to a party if Bob goes and Bob will go if Alice goes, but neither ends up going, since neither knows the other's conditions for attending.

To encourage more social behavior, the Student Invitational Party Board (SIPB) is developing a web service to help organize party going. For a given party, each student registers if he definitely wishes to attend or if he conditionally wishes to attend depending on whether a sufficient quorum of friends also attends. Specifically, the student indicates his condition on a SIPB web form by giving a threshold $t \geq 0$ and a list L of t or more other students such that he agrees to attend if at least t of the students on L also attend. Some of the students on L may not register, in which case we assume that they will not attend. At some designated time before the party, the SIPB service emails a message to each registered student whether the student should attend. The service guarantees that if all students who are emailed positive responses attend, then all the attendees' conditions are satisifed. We assume that a student is honor-bound to attend if his condition is satisfied, and that he doesn't register for conflicting parties.

The SIPB party service wishes to process the database of conditions so that as many people go to a given party as possible. Thus, in the Alice and Bob example, both should be sent a positive response. Model the problem formally, and give an efficient algorithm to select as many party-goers as possible subject to the students' conditions. For bonus points, devise a more general set of conditions that can be efficiently processed by a similar algorithm.

Solution:

1 Model

Let L_i and t_i be the list and threshold submitted by student i.

Algorithm PARTY accepts as input a directed graph, $G(V, E)$. G is represented as an array of adjacency lists. That is `Adj`$[i]$ lists neighbors of i. In addition to a name, each vertex has attributes `Slack` and `Attend`. There is an edge $e_{ij} \in E$ (from v_i to v_j) if and only if student i appears in list L_j. That is student j's attendance may depend on student i's attendance. We define `Slack`$[i]$ = $|L_i| - t_i$. We initialize, `Attend`$[i]$ $=$ `true`. We use one key additional data structure. Queue, Q, lists indices "recently" discovered *not* to be attending the party. Initially Q is empty.

2 PARTY (G)

1. For each vertex $v \in V$, if `Slack`$[v] < 0$, then assign `Attend`$[v]$ $=$ `false` and enqueue v on Q.
2. While Q is not empty do the following. Dequeue the head index i. For each student j listed in `Adj`$[i]$, decrement `Slack`$[j]$. If `Attend`$[j]$ $=$ `true` and `Slack`$[j] < 0$ enqueue j on Q.
3. For each vertex $v \in V$, if `Attend`$[v]$ $=$ `true`, add v to the solution set of students to be informed to attend.

3 Analysis

The loop in step 1 does constant work for each vertex in V. Step 1 running time: $\Theta(|V|)$.

In step 2, each vertex is added to Q at most once. To see this, note that an index i is added to Q only if $\texttt{Attend}[i] = \texttt{true}$, $\texttt{Attend}[i]$ is assigned `false` before enqueing, and $\texttt{Attend}[i]$ never transitions from `false` to `true`. Thus, each adjacency list is traversed at most once. Therefore, the while loop does constant work for each edge in E. Step 2 running time: $\Theta(|E|)$.

Tallying the solution set takes takes time proportional to the number of students. Step 3 running time: $\Theta(|V|)$.

Thus, the running time for PARTY: $\Theta(|V| + |E|)$. (*Not* the same as $\Theta(|V|^2)$.

4 Correctness

To show correctness, we show that every solution is *feasible* and *optimal*.

At any point a student i is *viable* if there exist t_i members, j, of L_i such that $\texttt{Attend}[j] = \texttt{true}$. A solution is feasible if every student in the solution set is viable. To show feasibility it is sufficient to prove the folowing loop invariant $\mathcal{I}$.

$$\mathcal{I} : \forall v \in V, (\texttt{Attend}[v] = \texttt{true}) \Rightarrow (v \text{ is viable})$$

This can be shown by induction on the iteration of step 2. Step 1 sets up the base case by enqueuing every non-viable i. Each iteration of the while loop explicitly sets $\texttt{Attend}[j] = \texttt{false}$ if viability is violated.

A solution is optimal if the solution set is as large as any other feasible solution set. To show optimality, it is sufficient to show that a node i is enqueued only if i is not viable. Again we show this by induction on the iterations of step 2. Step 1 sets up the base case by enqueuing every non-viable i. Similarly, step 2 only enqueues i if viability is violated.

5 Building the graph

Many students worried about transforming lists of student names into a graph. Using a hash function, this operation takes time proportional to the total size of the lists submitted.

First, assume every student is uniquely named (by e-mail address, for example). For each student mentioned in a list, hash the name. If the student is in the table, replace the name on the list with the index found in the hash table. If not, insert the student with a new index and slack -1. For each student that registers, hash the name. If the student is in the table, update the slack. If not, insert the student with a new index and the correctly calculated slack. Insert the list of indices of friends in the appropriate slot in the array of adjacency lists.

Now, we have a graph with edges from students to their friends. To get our input graph, transpose this graph. Transposing a graph takes time $\Theta(|V| + |E|)$.

6 Comments

Many students submitted $\Theta(|V|^2)$ and $\Theta(|V|^3)$ solutions. Some students forgot to check for the knock-on effects of removing a node. Some students tried to calculate students that should go rather than students that should not. Those attempts did not succesfully handle cycles.

The standard (slow) solution was not to transpose the graph and then just iterate until no change happened. Each iteration checks that a quorum of friends is still in the graph. If not, the node is removed. Checking the lists of friends for each node takes time $\Theta(|V|)$ on this graph representation. There are $\Theta(|V|)$ nodes to be checked on each iteration. Since only one node is removed per iteration, there are $\Theta(|V|)$ iterations. Giving the $\Theta(|V|^3)$ result. With work this time can get even worse if some measure of the slack is not memoized.

Many students knocked off a factor of $|V|$ either by using a queue or otherwise ensuring a student is only removed once. Some students transposed the graph but used an adjacency matrix rather than adjacency lists and so were still stuck with a $\Theta(|V|^2)$ result.

A few creative students represented the problem as a system of linear constraint equations. Let x_i be 1 if student i attends. 0 otherwise. Now the problem is to maximize $\Sigma_{i=1}^{n} x_i$ subject to the constraints of the form $t_i < \Sigma_{j \in L_i} x_j$.

Problem Practice-5. Reliable distribution

A communication network consists of a set V of nodes and a set $E \subset V \times V$ of directed edges (communication links). Each edge $e \in E$ has a ***weight*** $w(e) \geq 0$ representing the cost of using e. A ***distribution*** from a given source $s \in V$ is a set of directed $|V| - 1$ paths from s to each of the other $|V| - 1$ vertices in $V - \{s\}$. The ***cost*** of a distribution is the sum of the weights of its constituent paths. (Thus, some edges may be counted more than once in the cost of the distribution.)

(a) Give an efficient algorithm to determine the cheapest distribution from a given source $s \in V$. You may assume all nodes in V are reachable from s.

(b) One of the edges in the communication network may fail, but we don't know which one. Give an efficient algorithm to determine the maximum amount by which the cost of the cheapest distribution from s might increase if an adversary removes an edge from E. (The cost is infinite if the adversary can make a vertex unreachable from s.)

Solution:

1. This problem can be modelled as a single-source shortest paths problem. A distribution is a tree of paths from s to every other vertex in the graph. Since the cost of a distribution is the sum of the lengths of its paths, a minimum cost distribution is a set of shortest paths. Since the edge weights are non-negative, we can use Dijkstra's algorithm. The running time of Dijkstra's algorithm can be improved from $O(|V|^2)$ to $O((|V| + |E|) \lg |V|)$ using a binary heap. Since we assumed that the graph is connected, the running time is $O(|E| \lg |V|)$. If we use a Fibonacci heap, the running time is $O(|V| \lg |V| + |E|)$.

   In addition, we need to return the minimum cost distribution. Dijkstra's algorithm as given in CLR computes backpointers $\pi[v]$ to represent the shortest paths. We can use these to represent the distribution; when asked for the shortest path from s to v, we trace the backpointers from v to s and return the traversed edges (in reverse order).

   Notes:

   Some folks stated that the min-cost distribution problem is equivalent to the single-source shortest path problem but failed to explain why. Others used Dijkstra without explaining that this is correct because all weights are non-negative. Still others did not present the most efficient version possible (using Fibonacci heaps). Then there were those that used Bellman-Ford or all-pairs shortest paths, or invented their own algorithms.

   When returning the distribution, some folks returned a set of $|V| - 1$ shortest paths, one for each vertex. This is less efficient than the shortest paths tree, but they were not penalized harshly for this.

2. If an edge is removed from the graph, it is possible that the cost of the minimum cost distribution on the resulting graph may be more than the cost of original chepeast distribution. That is, let D be the minimum cost distribution found in part (a), and let $C(D)$ be its cost. If we remove edge e from the graph, let $C(e)$ denote the cost of the minimum cost distribution for the new graph (with edge set $E - \{e\}$). We need to compute

$$\max_{e \in E} C(e) - C(D).$$

   First recall that cost of a distribution is $\sum_{v \in V} d[v]$, where the $d[v]$ are the distance values returned by Dijkstra.

   The straightforward brute-force approach to solve this problem is compute $C(e)$ by deleting e from the graph and rerunning the algorithm from (a). However, note that if the deleted edge $e \notin D$ then $C(e) = C(D)$, since the removal of e does not affect the distribution D. So we only need to find

$$\max_{e \in D} C(e) - C(D).$$

   Since the edges in D are a set of shortest paths, they form a tree, and a tree has $|V| - 1$ edges.

   To compute $C(e)$ for an edge $e \in D$, we can delete e and then rerun Dijkstra's algorithm on the resulting graph. It is important to note that removing e may make some vertices unreachable from s. To check this, we remove e, rerun Dijkstra, and then check if any of the $d'[v]$ distances are ∞. If so, then $C(e) = \infty$ and we should halt the algorithm and return ∞ as the maximum possible increase. If not, then $C(e) = \sum_v d'[v]$.

The running time of this solution is the cost of $|V| - 1$ calls to Dijkstra's algorithm. Using Fibonacci heaps, this is $O(|V|^2 \lg |V| + |V||E|)$.

Notes:

Many folks got the idea of running $|E|$ Dijkstras, and quite a few optimized to $|V| - 1$ Dijkstras.

Some forgot to check the case when the graph becomes disconnected by the removal of an edge. Even if they assumed that operations with ∞ are well-defined (a poor assumption but accepted if clearly stated), they still should have explained why their code worked correctly for this case.

Some tried to fix up the original distribution in the following way: if the edge from u to v was deleted, they found the second best path from u to v and added it to the distribution. This is not optimal, because the new shortest path to v may no longer go through u.

Others misinterpreted the question, thinking that the edge deleted by the adversary was input to the problem.