Midterm Examination II

March 04, 2015

Name: _____
LASTNAME, Firstname(s)

Student Number: _____

Duration: 90 minutes
Aids allowed: one double-sided, 8.5" $\times$ 11" sheet of *handwritten* notes

Keep the exam booklet closed until the beginning of the examination. Make sure that your booklet has 8 pages (including this one). Write your answers in the spaces provided. *Please write legibly*.

Answer questions 1, 2 and 3, and *any one* of questions 4 and 5 (leaving the remaining question BLANK)

Take your time. Read each question carefully and try to avoid making problems more difficult than they really are.
Good luck!

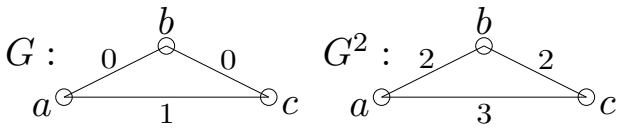
1	2	3	4	5	total
/26	/19	/20	/20	/20	/85

1. (5+5+8+8 marks)

Let $G = (V, E)$ be a graph, where every edge $e \in E$ has an associated, possibly negative, cost $c(e)$. Let $\Delta > 0$ be any constant and let G^{Δ} be the same as G, except each edge $e \in E$ has been given the modified cost $c(e)+\Delta$.

(a) Prove or disprove: for every pair $u, v \in V$, if P is a minimum-cost path from u to v in the graph G^{Δ}, then P is also a minimum-cost path in G.

This is easily disproved by exhibiting a counterexample to the assertion. In the example below, the shortest path from a to c in G is the path $a - b - c$. However the shortest path fro a to c in G^2 is $a - c$.



We introduced another general scheme for reweighting the edges of a graph $G = (V, E)$:
if $z : V \to \Re$ is any real-valued function defined on the vertices of G, then the new cost $\hat{c}((u, v))$ of edge (u, v) is defined as:

$$\hat{c}((u, v)) = c((u, v)) - z(u) + z(v)$$

for all $(u, v) \in E$, where $c(u, v)$ is the old cost.

(b) What property of path costs is *unchanged* by this reweighting? Explain briefly why this is true.

The relative cost (cost difference) of any two paths with the same start and end points is unchanged by the reweighting. This is true because, for any path $P = v_1, v_2, \ldots, v_n$:

$$\begin{aligned}
\hat{c}(P) &= \sum_{i>1} \hat{c}((v_{i-1}, v_i)) \\
&= \sum_{v>1} [c((v_{i-1}, v_i)) - z(v_{i-1}) + z(v_i)] \\
&= [\sum_{v>1} c((v_{i-1}, v_i))] - z(v_1) + z(v_n) \\
&= c(P) - z(v_1) + z(v_n)
\end{aligned}$$

So if paths P and P' both have start point v_1 and end point v_n then

$$\hat{c}(P') - \hat{c}(P) = c(P') - c(P).$$

(c) Johnson's algorithm for computing minimum-cost paths, for all vertex pairs, makes use of the reweighting scheme in part (b) to ensure that the reweighted edges all have non-negative cost. Explain how Johnson's algorithm determined the values for the "lifting" function z.

Johnson's algorithm:
(i) adds a new vertex v^* and edges (v, v^*), with cost 0, for all $v \in V$, to form a new graph G^*;
(ii) computes $\delta(v, v^*)$, the minimum cost path from v to v^*, for all vertices v, using Bellman-Ford (viewed as a single-destination algorithm); then
(iii) assigns $z(v)$ to be $\delta(v, v^*)$.

(d) Edge reweighing is also used to make Dijkstra's algorithm, interpreted as a single-source-single destination minimum-cost path algorithm, *goal directed*. Explain briefly what it means to be goal-directed, and how edge reweighing helps to achieve this.

"Goal directed" means that the (approximate) distance (cost) remaining to the destination is taken into account to help bias the search for the minimum cost path.

By reweighing edges, choosing $z(v)$ to be an estimate (consistent underestimate) of the distance remaining to the destination, Dijkstra's algorithm now chooses vertices in order in increasing values of: (distance from the source) + (estimated distance to the destination). This corresponds to what is commonly known as the A^* heuristic.

2. (7+5+7 marks)

We studied examples of *adaptive* (or *self-organizing*) search structure, based on lists and trees.

(a) Describe a situation in which splay trees, which know nothing of key access frequencies, would outperform a static binary search tree optimized to exploit known access frequencies..

Such a situation occurs when accesses, despite the fact that they adhere to some overall frequency, are nevertheless clustered in time. For example, if all accesses have the same total frequency but those of one key appear predominantly at the start, those of another predominantly at the end, etc.

(b) The Access Theorem for splay trees asserts that the amortized cost of accessing a node x in a splay tree with root node r is proportional to the difference of the rank of r and the rank of x, where the rank of any node z is (essentially) the logarithm of the total weight of the nodes in the subtree rooted at z.
How does the splay-to-root operation, that restructures the tree when a node x is accessed, depend on the weights of the nodes on the access path to x?

The operation itself does not depend on the weights at all. The weights are only used in the cost analysis.

(c) We discussed several corollaries of the Access Theorem that described different properties of splay trees based on different choices for the weights assigned to individual nodes. One of attributes of splay trees that remains only a conjecture is the so-call *Dynamic Optimality Conjecture*. Dynamic optimality has been established for list-structured dictionaries, using the move-to-front restructuring operation. Briefly describe what it means for the move-to-front restructuring operation to be dynamically optimal.

Dynamic optimality refers to the ability to compete, in amortized cost, to within a constant factor with any dynamic scheme dealing with the same sequence of operations (accesses), even one that knows the access sequence in advance. Thus the dynamic optimality of move-to-front means that the amortized cost of any sufficiently long sequence of accesses under move-to front is no worse than a constant factor more than the cost of any other restructuring algorithm, on the same sequence, even one that knows (or guesses) the access sequence in advance.

3. (10+10 marks)

In Assignment 5, you were invited to *consider* two different problems associated with a given a directed graph $G = (V, E)$ each of whose edges is coloured by one of χ distinct colours.

The first problem, the *minimum colour-transition path problem* asks for a path from a specified vertex s to another specified vertex t that minimizes the number of colour changes (that is the number of transitions between different coloured sub-networks). This problem was solved by reduction to the standard minimum-cost path problem in a graph whose edges have associated weights in $\{0, 1, \infty\}$.

(a) Let G be any n-vertex m-edge graph G whose edges have associated weights in $\{0, 1, \ldots k\}$, for some fixed integer k. We want to show that, in this situation, Dijkstra's algorithm can be implemented in such a way that it computes $\delta(s, v)$ for all $v \in V$ in $O(n + m)$ time in total. The idea is to exploit the fact that every simple path in G has has an integer-valued weight between 0 and kn. Thus we can maintain the priority queue, in Dijkstra's algorithm, consisting of the d_S-values associated with elements of $V - S$, as a list structure $L[0 \ : \ kn+1]$, where $L[i]$ points to a doubly-connected list containing all vertices $v \in V - S$ with $d_S[v] = i$, and $L[kn + 1]$ points to a doubly-connected list of vertices $v \in V - S$ with $d_S[v] > kn$.

Using the fact that the d_S-values extracted from the priority queue increase monotonically over time, show that the EXTRACT-MIN operation on this priority queue can be implemented to run in *amortized* $O(1)$ time.

The key idea here is that the structure described for maintaining the priority queue (d_S-values) can simply keep track of the list that contained the most recently extracted element (initially $L[0]$). When the list becomes empty, EXTRACT-MIN simply advances to the next non-empty list. (This may entail looking at a number of lists but the total amortized cost is $O(1)$ since once a list becomes empty nothing will ever be added to it, since the d_S values that are extracted increase monotonically.) If a list is non-empty we can simply extract its first element, which takes $O(1)$ time.

(Question 3, cont.) The second problem, the *minimum colour path problem* asks for an s, t-path that uses the smallest total number of distinct edge colours.

The (apparently unrelated) *vertex cover* problem takes as input an undirected graph $G = (V_G, E_G)$ and an integer k and asks if there exists a subset V' of V_G of size k such that every edge $(u, v) \in E_G$ is *covered* by V', that is either $u \in V'$ or $v \in V'$ (or both).

Suppose that $V_G = \{v_1, v_2, \ldots, v_n\}$ and $E_G = \{e_1, e_2, \ldots, e_m\}$. Consider the edge-coloured directed graph $H = (V_H, E_H)$, where $V_H = \{u_0, u_1, \ldots u_m\}$ and E_H contains an edge from u_{i-1} to u_i, with colour c_j exactly when vertex v_j is an endpoint of edge e_i in G.

(b) Prove that if the graph $H = (V_H, E_H)$ has a path from u_0 to u_m that uses k colours then the graph G has a vertex cover of size k. (Hint: draw a picture of the graph H.)

Note that ALL paths in H go through progressively higher numbered vertices. Suppose that H has a path $P = u_0, u_1, \ldots u_m$ that uses edges with k different colours in total. Let c_{j_i} be the colour of the edge in P from u_{i-1} to u_i. Then by construction the set of vertices $\{v_{j_1}, \ldots, v_{j_m}\}$ (i) has size k and (ii) covers all edges (edge e_i is covered with vertex v_{j_i}).

4. (10+10 marks)

Recall that in homework Assignment 5 , we studied a data structure, called a *block-sorted array* (or B-S array for short), that was designed to combine some of the advantages of ordered arrays (for fast MEMBER queries, using binary search) and unordered arrays (for fast dynamic operations, like INSERT and DELETE).

Specifically, suppose that at any moment in time set S contains n elements. Let $n = \sum_{j=0}^{k} b_j 2^j$, where $k = \lfloor \lg n \rfloor$ (i.e. $\langle b_k \dots b_1 b_0 \rangle$ is the binary representation of n.) A B-S array representation of S is an array A of size $2^{k+1} - 1$, divided into $k + 1$ *blocks*, where the i-th block, $0 \leq i \leq k$, is the subarray $A[2^i : 2^{i+1} - 1]$. The invariant maintained by the B-S array representation is that (i) blocks whose index i corresponds to a 0-bit in the binary representation of n contain a special sentinel ∞ only; (ii) all other blocks contain keys of S, without duplication; and (iii) the elements in each block are sorted in increasing order. Note that no assumption is made about the relationship between elements in different blocks.

(a) We described an implementation of INSERT that behaves a lot like incrementing a binary counter. In the worst case this INSERT operation into a B-S array with n elements could require $\Theta(n)$ time. However the *amortized* cost of INSERT (over a sequence of n INSERT operations into a B-S array of size at most n) is $O(\lg n)$. Explain the amortized analysis that gives this bound.

In the worst case we need to merge all of the elements into one new block. Since merging cost is linear in the total size of the blocks being merged, and the blocks double in size at each step, the total cost for merging is $\Theta(n)$ in the worst case.

It is straightforward to do an accounting-type amortized analysis to show that the amortized cost of INSERT (over a sequence of n INSERT operations, starting from the empty B-S array) is $O(\lg n)$. Simply assign each element $\lg n$ tokens, each of which pays for a unit of work in a single merge step. Note that a merge involving 2^i elements has a cost proportional to 2^i, so one token for each of the participants will pay for the full merge cost.) Since individual elements y can participate in at at most $\lg n$ merges in total (each merge doubles the size of the block containing y) these $\lg n$ tokens per element inserted suffice to pay for all of the required work.

(b) The operation INSERT can be viewed as a very special case of the UNION operation on B-S arrays (where one of the arrays has just one element). Describe how to implement a linear-time *general* UNION operation that takes two B-S arrays $A[1 : 2^{j+1} - 1]$ and $B[1 : 2^{k+1} - 1]$ and forms a new B-S array C representing the union of the elements represented in A and B. Explain your analysis.

The idea here is to observe that just as INSERT is similar to the INCREMENT operation on a binary counter, UNION is similar to binary ADDITION... As with binary addition, we work from low-order bits to high order bits. In general, when looking at the i-th position (bit or block, of size 2^i), we have a bit reflecting the status (empty or full) of the i-th block of A, a bit reflecting the status of the i-th block of B, and a carry bit, reflecting the status of a carry block (from the preceding step). As in binary addition we need to treat the various cases when 0, 1, 2 or 3 of these bits have value 1:
case 0: the i-th block of C is set to empty (sentinel value);
case 1: the sole non-empty block is copied into the i-th block of C;
case 2: the two non-empty blocks are merged (in linear time) into a carry block of size 2^{i+1}, and the i-th block of C is set to empty (sentinel value);
case 3: the current carry block is copied into the i-th block of C, and the i-th blocks of A and B are merged into a new carry block of size 2^{i+1}.
Since only a constant number of blocks (of size 2^i) are copied or merged in the i-th step, the total cost is $O(\sum_{0 \leq i \leq k} 2^i)$ which is $O(n)$.

5. (10+10 marks)

We have seen several examples of arguments that establish *lower bounds* on the cost of solving certain computational problems.

(a) Give a brief explanation of *information theoretic* and *adversary* arguments, sufficient to highlight their similarities and differences. Illustrate your answer with examples of these arguments that we have discussed.

information theoretic arguments exploit the fact that any process that distinguishes among n possibilities using two outcome questions, must use $\Omega(\lg n)$ questions on average. This is often formulated as a binary decision tree argument in which case it states that any tree (describing different computation paths) with n leaves (realizable outcomes) must have at least one path of length $\lg n$. This argument was used, for example, to show that sorting requires $\Omega(n \lg n)$ comparisons, or that locality of search cannot hope to do better than $\lg k$, on average, for keys located no further than k from the preceding query.

Adversary arguments on the other hand are more adaptive. The take the form of an anti-algorithm designed to respond to comparisons in such a way that the algorithm is forced into something resembling the worst case. A simple example is an anti-search adversary that forces any probing scheme, including binary search, to take at least $\lg n$ probes before locating the query value. In its simplest form an adversary simply chooses an input that is bad for any algorithm. We saw a more adaptive adversary in the first assignment where we proved a lower bound on finding the maximum and minimum elements in a set.

(b) Consider the set $\{0,1\}^n$ of all binary strings of length n. A *property* defined on $\{0,1\}^n$ (think of something like "palendrome" or "binary representation of a prime number") is said to be *odd* if an odd number of strings in $\{0,1\}^n$ satisfy the property. Argue, by describing a general *adversary strategy* that, for *every* odd property P, *every* algorithm that checks strings in $\{0,1\}^n$ for the property P can be forced to look at *every* bit in the string, in the worst case.

Suppose P is an odd property. If algorithm A asks about the i-th bit in the input string then the adversary checks the parity of the set of strings with the i-th bit set to 0 that satisfy P. If this set has odd parity then the adversary responds 0, otherwise it responds 1. In either case the set of strings consistent with the response that satisfy P remains odd. In this way, the adversary can continue to answer queries about individual bits, and as long as one or more bits remain un-probed the algorithm cannot be certain if the string being formed by the adversary does or does not satisfy the property P.