York University
Department of Computer Science and Engineering

COSC 3101
Summer 2005

Course instructor:
Periklis Papakonstantinou

Assignment 2

Print this `COVER PAGE` and complete the information as indicated. Attach this page to your report. Note that without this page (completed and signed) your assignment will not be marked.

Student 1
Name (First, Last): ______________________

Student number: ______________________

E-mail: ______________________

Student 2
Name (First, Last): ______________________

Student number: ______________________

E-mail: ______________________

Collaborators: ______________________
(If you did not discuss this assignment with others then write Collaborators: none)

References: ______________________
(Cite any sources you have used - apart from the textbook)

We certify that this assignment is our own work, and that we have acknowledged every person we discussed it with and we have cited every source of information used in its completion.

Signatures: ______________________

Problem	1	2	3	4	Total
Points:	20	60	50	50	180
Score:					

Assignment 2

Due date: FRIDAY June 10, 2005

Problem 1 (20 points)

Half of chapter 24 (or 25 if you use the 1st edition) of your textbook is devoted to Dijkstra's algorithm for Single Source Shortest Path (SSSP). In class we gave a simplified proof of correctness for a simpler problem, which we called *SSSP-Value*. A description of Dijkstra's algorithm is given below. Apart from lines 4, 17 the algorithm is the same as the one presented in class and proved to be correct for *SSSP-Value*. The purpose of this question is to do a slight extension to our proof so as to get a simplified proof of correctness for Dijkstra's algorithm. Lets repeat the definitions of the problems:

Problem: SSSP-Value

Input: A directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{Q}^{\geq 0}$ and a designated vertex $s \in V$.
Output: The *value* (i.e. total weight) of the shortest paths from s to every other vertex of G.

Problem: SSSP

Input: A directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{Q}^{\geq 0}$ and a designated vertex $s \in V$.
Output: The shortest paths from s to every other vertex of G.

We showed that Dijkstra's algorithm excluding lines 4, 17 is correct for *SSSP-value*. Now, suppose that we add lines 4, 17. Our goal is to show that this algorithm is correct for *SSSP*.

In the algorithm (given below) lines 1-7 correspond to the "initialization phase", where lines 12-17 correspond to the "relaxation phase". Intuitively $\pi[u]$ corresponds to the predecessor of u. That is, in a path from s to u $(\pi[u], u)$ is the last edge (pay attention that the edge is from $\pi[u]$ to u).

Of course the purpose of this question is not to copy (almost 10 pages) from your textbook. In the following proof you will take for granted what we have proved in class (i.e. that DIJKSTRA is correct for *SSSP-Value*).

(a) To be precise, DIJKSTRA does not exactly output shortest paths. It outputs all vertices together with their predecessors. How do we find a path from s to u from the output of DIJKSTRA? Suppose that in the output there exist a path from s to u. We can find the sequence of vertices of the path in *reverse order* as follows: the last vertex is u. The vertex before the last vertex is $\pi[u]$. The vertex before $\pi[u]$ is $\pi\big[\pi[u]\big]$ and so on. When have we reached s? This happens when $\pi[\pi[\ldots\pi[u]\ldots]] = NIL$. We shall call this procedure where given the list of predecessors we construct a path in this way as "starting from u we follow backwards up to s the list of predecessors".

If in the output there is no path from s to u then $\pi[u] = NIL$.

Side note: It is clear that there is a straightforward algorithm that runs in linear time in the number of vertices, where given the outputted list π and a vertex u it determines the path from s to u.

Pay attention to the fact that in the above discussion no assertion is made whether the outputted paths are actual shortest paths or even if they are paths in the given graph. This is exactly what you have to show (proof of correctness).

We wish to show that when DIJKSTRA terminates then for every vertex u by following backwards the predecessor list (up until π is NIL) we get a shortest path from s to u.

i. (5 points) Show that for every vertex u, when in every iteration the algorithm goes into the relaxation phase (i.e. when $d[u]$ and $\pi[u]$ are updated) $d[u]$ is the length of a path from s to u of length $d[u]$ and $(\pi[u], u)$ is its last edge.

   Hint: You can do induction on the iterations of the main loop. Be careful on the choice of your induction predicate. If you do induction then your induction predicate could be of the form "If Q then W". Clearly, if Q is not true then the inductive predicate is true.

ii. (5 points) Relying on the correctness of DIJKSTRA for *SSSP-Value*, prove that when DIJKSTRA terminates then for every vertex u: (i) if u is reachable from s then by following backwards the predecessor list (up until π is NIL) we get a shortest path from s to u. (ii) If u is not reachable from s then $\pi[u] = NIL$.

   Hint: Again you can do induction, but on what? Think of a "clean" way to argue.

```
DIJKSTRA[$G = (V, E), w, s$]
1  for every $u \in V$
2        do
3            $d[u] \leftarrow \infty$
4            $\pi[u] \leftarrow NIL$
5  $d[s] \leftarrow 0$
6  $S \leftarrow \emptyset$
7  $Q \leftarrow V$ (Comment: $Q$ is a priority queue)
8  while $Q \neq \emptyset$
9        do
10           $u \leftarrow$ EXTRACT-MIN($Q$)
11           $S \leftarrow S \cup \{u\}$
12           for every $v$ adjacent to $u$
13               do
14                   if $d[v] > d[u] + w(u, v)$
15                       then
16                           $d[v] \leftarrow d[u] + w(u, v)$
17                           $\pi[v] \leftarrow u$
18 Output $d$ and $\pi$
```

(b) The list π in the output has an interesting underlying structure. By the proof of correctness of the previous part, every vertex not reachable from s has no predecessors.

Focus on the component consisting of all vertices reachable from s (a vertex u is reachable from s iff there exists a directed path from s to u).

We define a *directed tree T rooted at s* as a directed graph where every vertex in T is reachable from s and if we ignore the direction of the edges then it is a tree.

We wish to show that in the output of Dijkstra the vertices of $G = (V, E)$ which are reachable from s form a directed tree.

i. (5 points) Give an example of (at least) 5 vertices and write the output of list π of DIJKSTRA. Then, draw the directed tree for your example (to help the marker make your directed tree weighted with the corresponding weights from the input graph).

ii. (5 points) Observe that in every iteration of DIJKSTRA every vertex has at most one predecessor. (how many predecessors does s have?) Conclude (using the correctness of DIJKSTRA - showed in part (a)) that the output regarding the vertices reachable from s is a directed tree.

Hints: Argue by the way of contradiction. Suppose that it is not a directed tree. Then, clearly the only problem is that there exists a cycle when ignoring the edge directions on the subgraph containing the vertices reachable from s. See what happens when a cycle is constructed. Where s should be? How about the vertices belonging to the cycle? When replacing the directions how the directed cycle looks like?

Problem 2 (60 points)

In class we often omit, around reals x, the ceiling ($\lceil x \rceil$) and the floor ($\lfloor x \rfloor$) for the sake of presentational simplicity. As far as it concerns the worst-case running time of algorithms asymptotically there is no problem in omitting them. As far as it concerns descriptions of algorithms, in most cases there is a quite important reason for treating these roundings carefully. In this question, among others you are asked to do some work related to the importance of correct roundings.

In section 2.3 of your textbook (or section 1.3 if you are using the 1st edition) there is a description of Merge-Sort. In the same section there is an informal description of Merge.

(a) i. (5 points) Give a clear description of Merge.

ii. (5 points) Give an example of an input sequence of (at least) 5 elements and present the steps of Merge-Sort.

(b) (10 points) Show that Merge is correct. To show this

i. you first have to clearly formulate a problem associated with Merge. Obviously, the definition of this problem *must* be helpful for the correctness of Merge-Sort.

ii. Then show that Merge terminates and

iii. that under Merge (defined in (a)) the preconditions (i.e. if your input is in the correct form) of your problem's formulation imply the postconditions (i.e. then your output is in the correct form).

(c) (5 points) Show that Merge-Sort terminates.

(d) (10 points) Give a proof of correctness for Merge-Sort.

(e) (10 points) Show that if we modify Merge-Sort such that in the 2nd line we have $q \leftarrow \lceil (p+r)/2 \rceil$, then modified Merge-Sort is not correct.

(f) (10 points) Which is the exact point in your argument in your correctness proof (parts c or d) that collapses when the modification of part (e) is applied.

(g) (5 points) Up to now we were only concerned with the time complexity of algorithms. We define the space complexity of an algorithm as the maximum number of *different* memory registers it uses (i.e. when we use the same register more than once this register adds only one to the space complexity). For simplicity suppose that every register can store only one number and that your description of Merge (part (a)) is reasonable (i.e. it does not encode in one number a list of numbers). In class we said that Merge-Sort is a not an *in-place* sorting algorithm. What is (asymptotically) the worst-case space complexity of Merge-Sort.

Hints: (i) The proof of correctness of MERGE depends on what algorithm gave in (a). Some reasonable implementations of MERGE might involve more than one non-nested loops. It seems easier to prove correctness by breaking your correctness argument into parts (one for each loop).

(ii) To show termination associate the length of the biggest subarray that it is passed as an argument in the recursive call with a strictly decreasing sequence. Important detail (might also be useful in subsequent parts of this question): show that when the maximum length is 2 then no more subsequent calls are made after the last two ones.

Problem 3 (50 points)

Consider a meta-search WWW engine. This engine queries the same query to a number of WWW search engines and gets several rankings for the results. A fundamental problem in these types of applications is to compare different rankings. Lets focus on the problem of comparing two different rankings. For simplicity suppose that every search engine returns the same results (set of objects). Fix a specific query and an order among the returned objects $\langle obj_1, obj_2, \ldots, obj_n \rangle$. Say that one engine ranks the objects $obj_1, obj_2, \ldots, obj_n$ as $1, 2, \ldots, n$ respectively, according to some scoring criterion. Now suppose that another engine ranks the same objects as $a_1, a_2, \ldots, a_n$ (where $a_1, a_2, \ldots, a_n$ are all distinct and $a_i \in \{1, \ldots, n\}$) respecting the position in $\langle obj_1, obj_2, \ldots, obj_n \rangle$. We wish to have a natural measure which tells us how much the two orderings differ. If $a_1 < a_2 < \ldots < a_n$ then the two orderings are the same and a natural measure should tell us 0. It seems that a natural measure would be to count the number of inversions in the sequence $\langle a_1, a_2, \ldots, a_n \rangle$. For example, if this sequence is $\langle 5, 10, 7, 6 \rangle$ there are 3 inversions (since $10 < 7$, $10 < 6$, $7 < 6$).

Given a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of distinct integers we say that two indices $i < j$ form an inversion iff $a_i > a_j$. Our goal is to count the number of inversions.

(a) (Warm-up part)

i. (5 points) Consider the following Brute-Force algorithm: Given a sequence of integers as an input check all possible pairs of integers to see if they form an inversion. What is the (worst-case) running time of this algorithm?

ii. (5 points) What is the best lower-bound on the number of inversions for a sequence of n integers? Is the above running time (a-i) a tight bound for the worst-case running time? (i.e. does it match the worst-case upper bound?)

(b) From the previous part it should be clear that the Brute-Force algorithm takes as much time as the (worse) number of inversions on a given sequence. Therefore, we have to devise an algorithm that counts the number of inversions that does not even check every inverted pair. We will use a Divide & Conquer approach. Divide the sequence into two parts of (almost-i.e. up to rounding) equal size. Count the number of inversions in each part. The only tricky thing is the "combine-step" of your Divide and Conquer algorithm. That is, to determine the number of inversions that occur between elements belonging to different parts. To get a faster than $O(n^2)$ algorithm we have to "combine" within linear time. The trick is to do some extra work in the algorithm and keep the two parts sorted. Now, the "combine-step" can be done in linear time with respect to the size of the parts.

i. (10 points) Modify the MERGE-SORT so that it counts the number of inversions for the given sequence of n integers in the input.

ii. (10 points) Present how your algorithm works on an input example of a sequence of (at least) 5 elements.

iii. (10 points) Prove that your algorithm takes $O(n \log n)$ steps in the worst case. Sketch the proof of correctness of your algorithm. When sketching the proof you may refer to the correctness of MERGE-SORT (see Problem 2).

Hint: It helps first to modify the MERGE-SORT given in your textbook so as it returns an array as a result (using programming terminology we would say that A is not viewed as a global variable). Then modify MERGE-SORT so that your algorithm returns not only the shorted sub-list but also the number of inversions.

(c) (10 points) Define a *major inversion* as follows: given a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of distinct integers we say that two indices $i < j$ form a major inversion iff $a_i > 2a_j$. Give an algorithm that counts the number of major inversions in (worst-case) running time $O(n \log n)$. Justify why the worst-case running time is $O(n \log n)$. Again you are not required to give a detailed proof of correctness. Explain which parts of the proof of correctness of the previous part and of the correctness of MERGE-SORT should be modified and how.

Problem 4 (50 points)

Consider undirected graphs that have weights on the vertices (not the on the edges). In some (mostly database) applications we may have available as an input a graph, but we do not have the weights of the vertices. Instead we can query a function (which we do not have its description) and learn the values of the weights of the vertices. Consider algorithms that take as inputs such an undirected graph and they are also provided with the ability to query a function which returns the value of the weight of a vertex. Suppose that the cost of querying the function is enormous compared to the cost of other operations that an algorithm might do. Therefore, the worst-case number of queries is a good estimate on the worst-case running time.

Given a graph $G = (V, E)$ with weights on its vertices $w : V \to \mathbb{Q}$ we say that a vertex $v \in V$ is a *local minimum* iff every adjacent to v vertex has weight greater than $w(v)$.

(a) (10 points) Say that $T = (V, E)$ is a complete binary tree (where $|V| = n = 2^h - 1$) rooted at $r \in V$ and $w : V \to \mathbb{Q}$ assigns *distinct* weights to vertices (i.e. each vertex gets a different weight). Say that $l \in V$ is a leaf of T. Consider the path $P = \langle r, u_1, u_2, \ldots, u_k, l \rangle$ (from r to l). Let T_{u_1} be the subtree of T with root u_1. If T_{u_1} does *not* contain a vertex which is a local minimum what is the relation between the weight of l ($w(l)$) and every other vertex in the path from u_1 to l? More generally, what is the relation between the weights of every two adjacent vertices in the path $P = \langle u_1, u_2, \ldots, u_k, l\rangle$? Prove your answer.

Hint: Show a stronger statement: Prove that your observation holds for every subtree T_u rooted at u (that goes down to the leaves of T) and every path from a leaf of T_u to u. Use induction on the height of the tree.

(b) (10 points) Under the conditions of part (a) local minima trivially exist. We want to show this fact by applying part (a). Say that $T = (V, E)$ is a complete binary tree rooted at $r \in V$ and $w : V \rightarrow \mathbb{Q}$ assigns *distinct* weights to vertices. Prove that T has (at least one) a local minimum.

Hint: Argue by the way of contradiction. Consider two disjoint paths from r to two leaves and use the stronger fact shown to prove (a).

(c) (20 points) If you used the hint to prove (b), although the argument goes by contradiction it is still a very constructive argument. You might have noticed that the way you argue regarding the weight of r "shows" you where you should look for a local minimum. Intuitively the way you argued reveals a nice "recursive structure" for a procedure that looks for a local minimum. Now, we make this intuition precise.

We are interested in the following problem: Given a complete binary tree $T = (V, E)$ with $|V| = n = 2^h - 1$ vertices (where h is the height of the tree). Consider a function $w : V \rightarrow \mathbb{Q}$ which assigns *distinct* weights to the vertices. The weights are given implicitly, i.e. every algorithm that wants to learn the weight of a vertex has to query the weight function. We want to find a vertex which is a *local minimum*. Describe an algorithm that solves this problem by querying (in the worst case) $O(\log n)$ vertices. You do not have to prove the worst case number of queries. Prove that your algorithm is correct.

(d) (10 points) Although, the properties of the complete binary tree suffice to formally show that the worst case number of queries is $O(\log n)$ you must show this as follows: Define a recurrence relation associating the number of queries and the input length. Use Master Theorem to show that the solution of this recurrence relation is $O(\log n)$.