

Introduction to Algorithms and Data Structures

IT University of Copenhagen

June 4, 2003

This exam consists of 3 exercises containing in total 13 subexercises. Each of these 13 subexercises is given the same weight in the evaluation. The exam consists of 6 pages. You have 4 hours to complete the exam. Remember to number the pages and write your name and CPR number on every page.

CLRS refers to “Introduction to Algorithms” by Cormen, Leiserson, Rivest and Stein, Second Edition, 2001. For exercises in which algorithms must be specified, the asymptotic time complexity of the specified solution will be taken into account when grading. Exercises asking for time complexity must be answered using O -notation. It is weighted in the evaluation that growth rates in the O -notation are expressed with least possible asymptotic growth.

Exercise 1

This exercise is about rooted binary trees and heaps. The representation and notation for binary trees is similar to section 10.4 pp. 214-215 in CLRS.

We assume that all nodes x in the trees we consider have a field $key[x]$ which contains an integer. Furthermore, A is an n -element max-heap as defined in Chapter 6 in CLRS, initially empty before the call to `TREETOHEAP`. The procedure uses the operation `MAX-HEAP-INSERT` as defined on page 140 in CLRS. Consider the following procedure, which as input takes the root x of a binary tree with n nodes.

```
TREETOHEAP( $x$ )
1  if  $x \neq \text{NIL}$ 
2      then MAX-HEAP-INSERT( $A, key[x]$ )
3          TREETOHEAP( $right[x]$ )
4          TREETOHEAP( $left[x]$ )
```

a) Give the time complexity of the procedure `TREETOHEAP(x)` in O -notation.

Answer a) $O(n \log n)$, there is n heap operations each taking $O(\log n)$.

In the subexercises b) and c) below we assume that the tree with root x satisfy the *binary-search-property*.

b) Describe an efficient algorithm for finding the closest pair of nodes in a tree T with root x , *i.e.*, return two nodes y and z in T such that the difference between $key[y]$ and $key[z]$ is smallest possible among all pairs of nodes in T . Give the time complexity of your algorithm and justify your answer.

Answer b) Since the tree satisfy binary-search-property, we can put all element in T in an array in sorted order in time $O(n)$ using an in-order traversal. Then we can go through this array left to right and checking each pair, remembering the smallest which is the closest pair to be reported. Time clearly in total $O(n)$.

c) Describe an efficient algorithm for a procedure `SEARCHTREETOHEAP(A, x)` that like `TREETOHEAP` inserts all keys from the tree to heap A using the `MAX-HEAP-INSERT` operation n times. Give the time complexity of your algorithm and justify your answer.

Answer c) As before an in-order traversal of T gives the elements in sorted order. Hence we can insert the element in decreasing order. Since each element is smaller than any other element in the heap during inserts in decreasing order, this implies that HEAPIFY operation does not make any swap, *i.e.*, MAX-HEAP-INSERT takes time $O(1)$, giving a total cost of $O(n)$ for this algorithm.

Consider the data structure S that supports the following three operations.

INSERT(x) inserts element x into S with key $key[x]$.

DELETE(x) removes element x from S .

CLOSESTPAIR() returns two different elements $x, y \in S$ such that $key[x] - key[y]$ is least possible.

d) Make a data structure that supports the above three operations in worst-case time $O(\log n)$ per operation, where n denotes the actual size of S . Justify your answer.

Answer d) Maintain the element in S in a red-black search tree ordered after the key values. Secondly we maintain a priority queue of each of the $n - 1$ pair of elements being neighbour pairs in a sorted sequence. This priority queue has as key for each pair the difference between the two elements keys. Hence this priority queue satisfies the invariant that the minimum pair is the closest pair. The priority queue may be implemented by a dynamic heap or a red-black search tree in time $O(\log n)$ per operation (INSERT, DELETE or MIN). The above two data structures are maintained for an insert as follows. An insert of element x deletes the pair (y, z) where y is the predecessor to x and z is the successor to x . Hence, we start by finding the predecessor y and successor z to x in the red-black search tree in time $O(\log n)$. Then, we can find this pair (y, z) in the priority queue using a dictionary (implemented by yet another search tree) in time $O(\log n)$, and then delete the pair from the priority queue in time $O(\log n)$. Second, we insert the two new pairs (y, x) and (x, z) into the priority queue and a dictionary enabling the previously mentioned lookup. Time cost in total $O(\log n)$ for all these operations (using say red-black search for all the data structures).

Similarly the delete operation can be maintained by deletion of two pairs in the priority queue and insertion of a single new pair, using the predecessor/successor queries in the red-black search tree, a pair of dictionary lookups. Finally the query CLOSESTPAIR is by the above invariant simply a call to MIN of the priority queue.

Exercise 2

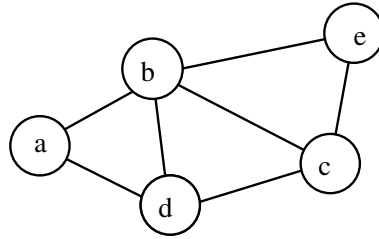
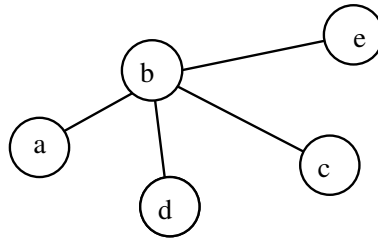


Figure 1: The graph H

a) Draw a spanning tree of the above graph H .

Answer a)



In subexercises b), c) and d) below we consider a graph G with n fixed nodes $V = \{1, 2, \dots, n\}$, and initially without any edges. This graph is maintained by a number of operations given below. Furthermore, we assume that each edge in G is colored either blue or white. Consider the following four operations.

$\text{INSERTEDGE}(u, v)$ inserts a *white* edge between node u and v .

$\text{COLORAROUNDNODE}(v)$ colors all white edges incident to v *blue*.

$\text{COLOREGE}(u, v)$ colors the edge (u, v) *blue* (keep it blue if it already is blue).

$\text{BLUE}(u, v)$ returns true if and only if the edge (u, v) is *blue*.

b) Give an efficient algorithm that supports the above four operations aiming at fastest possible amortised time per operation. Give both worst-case and amortised time complexity of your algorithm and justify your answer.

Answer b) We maintain the graph using two adjacency list representations for the blue and the white edges. Each edge insert is done by appending the edge to the white lists of the corresponding nodes. Furthermore we maintain a single bit for each edge indicating its color. Finally all edges are put into a hash table such that we determine their location (if existing) in the adjacency lists (blue or white). When an edge is colored blue it is moved from the white to the blue lists. An accounting argument shows that if each white edge has one dollar when inserted, this can pay for the subsequent single move to the blue list. Hence `COLORAROUNDNODE` are trivially implemented by moving all edges incident to v in the white adjacency list for v into the blue list for v , one by one, flipping the color bit. It takes $O(n)$ time worst-case, while the accounting argument implies $O(1)$ amortised cost. Clearly, the `BLUE` query is a single hash lookup in constant time with check of the color bit, *i.e.*, worst-case (and thus also amortised) time $O(1)$.

We say that two nodes u and v in G are *connected by a blue path* if there is a path between u and v with only blue edges. Furthermore, the *blue component of v* is the set of nodes which are connected to v by a blue path including v . Consider the following additional operation.

`BLUEPATH(x, y)`. Return *true* if and only if x and y are connected by a blue path.

c) Give an efficient algorithm that supports all of the above five operations aiming at fast amortised time per operation. Give both worst-case and amortised time complexity of your algorithm and justify your answer.

Answer c) Extend the data structure in b) by maintaining each node in a Disjoint Set union-find data structure. Each time an edge is colored blue we union the two sets containing the two end-nodes of the edge. Then there is a blue path between a pair of nodes iff they are in the same set, so two find-set operations decide `BluePath`. Time bounds are as above in b) except for the additional cost of union-find operations. These take amortised time $O(\alpha(m))$ per operation for m operations where α is the inverse Ackermann function. Worst case, the time cost of union-find is $O(\log n)$, but the structure above already takes time $O(n)$ worst-case, so this is the worst-case bound.

Consider the following additional operation.

REMOVEBLUECLOUD(x). Delete all *blue* edges between pairs of nodes in the blue component of x .

d) Give an efficient algorithm that supports all of the above six operations aiming at fast amortised time per operation for operation sequences of length $m \leq n$. Give both worst-case and amortised time complexity of your algorithm and justify your answer.

Answer d) Extend c) where RemoveBlueCloud is as follows: A call for node x traverses the blue component with x using the blue adjacency lists, and by a bread-first search. This takes time linear in the number of blue edges in this component. During this traversal, the set containing all the nodes from the union-find data structure is deleted and new singleton sets are made. All edges are clored white and moved to white adj. lists. Time cost linear time in the number of nodes and edges. Hence using the accounting method, we can put one dollar on each edge and its end-nodes when it is colored blue, which will pay for the above linear time costs, that makes the white again. Hence amortised time cost as in c), and worst-case time cost is $O(m)$ where m is number of blue edges.

Exercise 3

This exercise is about weighted, directed graphs. We consider a special case of such graphs, which we will call *1-2 layered graphs*. A 1-2 layered graph is a directed graph $G = (V, E)$ where the nodes except two special nodes $s, t \in V$ can be partitioned into k disjoint sets L_1, L_2, \dots, L_k called *layers* such that the following holds. For $1 \leq i \leq k-1$, each node v in layer i has an edge to exactly two nodes in layer L_{i+1} , and there is no other edge from v to a node elsewhere. Furthermore, node s has an edge to all nodes in layer L_1 and no else, and all nodes in layer L_k have an edge to node t and no else. The graph in Figure 2 is thus an example of an 1-2 layered graph with three layers of nodes.

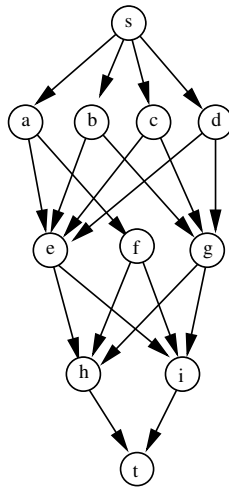


Figure 2: An example of an 1-2 layered graph H'

a) List the nodes in a sequence corresponding to a *depth-first* traversal of the H' starting from s .

Answer a) s a e h t j f b g c d

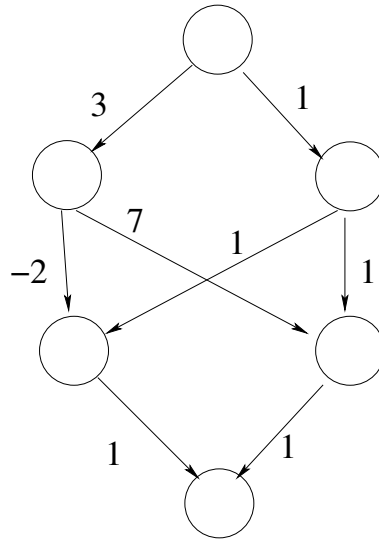
In subexercise b) and c) below, Dijkstra's algorithm refers to the version described in CLRS Chapter 24.3 pp. 595-599.

b) Give the time complexity for finding the shortest path between s and t in an 1-2 layered graph with n nodes using Dijkstra's algorithm assuming E only has positive edge weights.

Answer b) $O(n \log n)$

c) Give a counterexample of an 1-2 layered graph with both positive and negative edge weights, where Dijkstra's algorithm fails to find the shortest path.

Answer c)



d) Given a graph $G = (V, E)$, make an efficient algorithm that decides if G is an 1-2 layered graph. Give the time complexity of your algorithm and justify your answer.

Answer d) Find the node with out-degree zero which must be t – if there are more than one such node return false. Else we make a bread-first tree from t according to the graph where we have flipped all orientations of the edges (easy to precompute in linear time before this step). Each layer in this tree, ie depth 0, 1, 2 etc. from root t must be the layers of the 1-2 layered graph. Hence we mark each node with its layer being depth of tree minus depth of node. Then we go through all nodes and if node x is marked with layer k , it must have exactly two nodes with mark $k+1$ in its adjacency list. The only exception is a single node with layer 0, which is s and must contain all nodes of layer 1, and similiary the maximal layer only contain node t . All of this is checked in linear time, and if it holds true is returned. Total time $O(n)$.

e) Describe an efficient algorithm that finds a subset $E' \subseteq E$ of edges of an 1-2 layered graph $G = (V, E)$ forming a *shortest path tree* with root s such that the sum of edge weights in E' is smallest possible. Give the time complexity of your algorithm and justify your answer.

Answer e) I assume the graph is given such that nodes has associated their corresponding layer using a breadth-first traversal like the one given above, if this is not the case. We construct the minimal weight shortest path tree (MWSPT) inductively as follows. The root of MWSPT is clearly s , with distance 0 form s . Consider a node x at layer $k \geq 1$. Let y_1, \dots, y_k be the nodes at layer $k - 1$ with an edge to x , and assume inductively we have computed the shortest paths to these in MWSPT, and let $d(y_i)$ denote this distance for node y_i . Clearly, a shortest path to x is given by the i where $d(y_i) + w(y_i, x)$ is minimal. That is the parent of x has to be one of such nodes y_i in order to be a shortest path tree. Clearly, to minimize edge weight cost, we pick among these candiates the one with minimal edge weight $w(y_i, x)$ too (and solve ties arbitrarily). Time cost, straightforward $O(n)$.