CS60001 Advances in Algorithms, Autumn 2008– 09

End-Semester Test

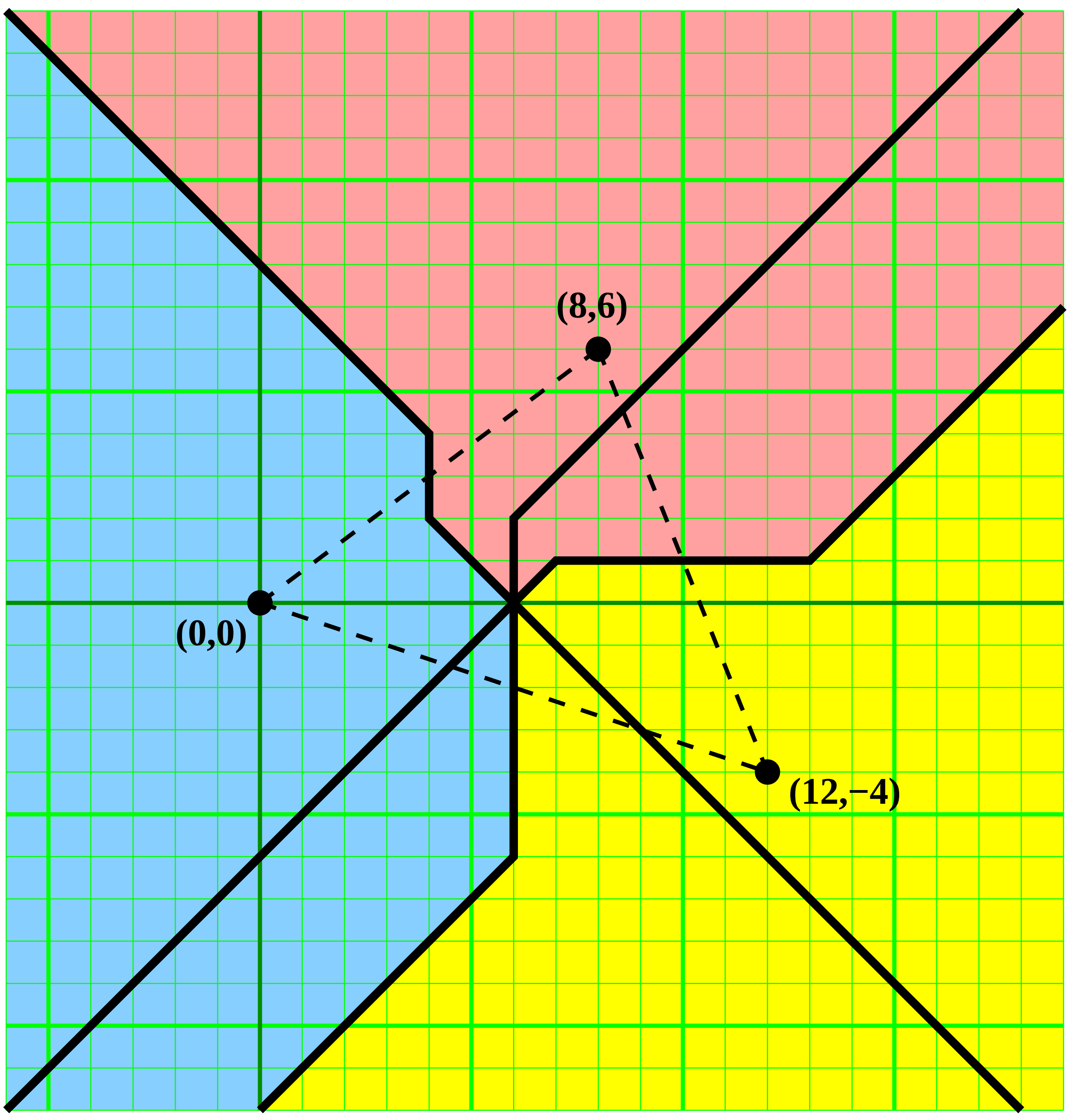
Maximum marks: 60 | November 22, 2008 | Total time: 3 hours

Roll no: ____________________ **Name:** ____

- *Write your answers in the question paper itself. Be brief and precise. Answer all questions.*
- *Avoid untidiness in the answer script.*

1. The *Manhattan distance* between two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ in the plane is defined as $d_\infty(P_1, P_2) = \max(|x_1 - x_2|, |y_1 - y_2|)$. Your task is to compute the Voronoi diagram of the three points $P_1 = (0, 0)$, $P_2 = (8, 6)$ and $P_3 = (12, -4)$ with respect to the Manhattan distance. The locus of the points P equidistant from P_1 and P_2 (that is, $d_\infty(P, P_1) = d_\infty(P, P_2)$) is provided. Draw the similar loci for the two other pairs (P_2, P_3) and (P_1, P_3). Then, mark the Voronoi cells of the three points P_1, P_2 and P_3. **(3+3+4)**

Solution



2. An *interval* $[a, b]$ refers to the set of real numbers x satisfying $a \leqslant x \leqslant b$. Two intervals I_1 and I_2 are called overlapping if $I_1 \cap I_2 \neq \emptyset$.

You are given n intervals $I_1 = [a_1, b_1]$, $I_2 = [a_2, b_2]$, $\ldots$, $I_n = [a_n, b_n]$, each specified by its two endpoints. Your task is to find all the pairs (I_i, I_j) of overlapping intervals. Describe an $O(n \log n + h)$-time algorithm to solve this problem, where h is the number of overlapping pairs. You must supply an argument corroborating that your program achieves this running time. You may assume that the endpoints of the input intervals are in general position (no repetitions). **(6+4)**

Solution I propose a sweep algorithm for solving this problem. Here, a point sweeps from $-\infty$ to $+\infty$. An events occurs when the sweep point meets an endpoint of an interval. The events are processed one by one with increasing value. For a particular position of the sweep point, the *active intervals* are those intervals that contain the sweep point. We maintain a list L of active intervals at each position of the sweep point.

Initialization: Store the endpoints $a_1, b_1, a_2, b_2, \ldots, a_n, b_n$ in a min-priority queue Q. Initialize L to empty.

Event loop: As long as Q is not empty, consider the next event (endpoint), handle it, and dequeue this event from Q. We have only two types of events.

Enter interval I: At this point, I becomes active and so overlaps with all other active segments at this point. Output (I, J) for all intervals J in L. Insert I in L.

Leave interval I: After this point, I becomes inactive, so delete I from L.

Running time: We realize Q as a min-heap. Initializing the heap by $2n$ points requires $O(n)$ time. The total number of events that occur is exactly $2n$, that is, we need to call EXTRACT-MIN on Q exactly $2n$ times. This calls for a total of $O(n \log n)$ time to manipulate the queue.

We maintain L as a doubly connected linked list. We also maintain a list of pointers indexed by $i = 1, 2, \ldots, n$. The i-th pointer points to the node in the linked list, storing the i-th interval I_i, provided that I_i is active. If not, I_i is not present in L, and the corresponding pointer may be set to NULL. Each insertion of an interval in L during an enter interval event or each deletion of an interval from L during a leave interval event can be handled in $O(1)$ time. Since each interval is inserted only once in L and deleted only once from L, the total cost associated with the manipulation of L is $O(n)$.

Finally, identifying and reporting h overlapping pairs of intervals take a total of $O(h)$ time.

3. Let $G = (V, E)$ be a simple directed graph with each edge carrying a positive cost. Suppose that each vertex in G also carries a positive cost. You may visualize G as a computer network in which the cost associated with the edge (v, w) stands for the cost of transferring a packet from v to w. On the other hand, the cost associated with a vertex v is the cost of relaying a packet at v. Relaying costs are not applicable at the source and the destination. To sum up, the total cost of the path $v_1, v_2, \ldots, v_k$ is $c(v_1, v_2) + c(v_2) + c(v_2, v_3) + c(v_3) + \cdots + c(v_{k-1}) + c(v_{k-1}, v_k)$.

You are given a distinguished vertex $u \in V(G)$ (the source). Your task is to compute the cheapest paths from u to all vertices in G. Modify Dijkstra's SSSP algorithm to solve this problem. (Mention only the steps you have modified. Do not write the entire algorithm.) Your modification should have the same running time as Dijkstra's algorithm (supply an argument). Also prove the correctness of your modification. **(6+2+2)**

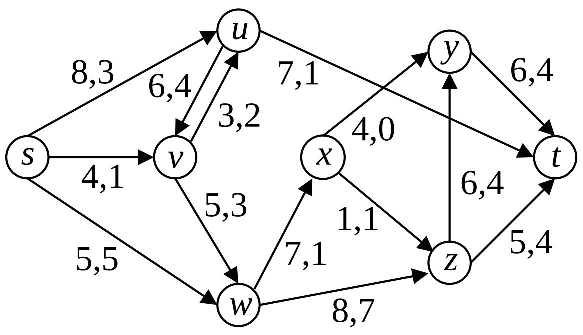
Solution The only modification is during the updating of the current shortest distance $D[v]$. When w is moved from Q to P, we check whether $D[w] + c(w) + c(w, v) < D[v]$. If so, we replace $D[v]$ by $D[w] + c(w) + c(w, v)$.

Since only the distances are calculated in a new way ($D[w] + c(w) + c(w, v)$ instead of $D[w] + c(w, v)$) and the modification entails no further change in Dijkstra's algorithm, the modified algorithm continues to have the same running time as Dijkstra's original SSSP algorithm.

The proof of correctness remains exactly the same as is given in connection with Dijkstra's SSSP algorithm.

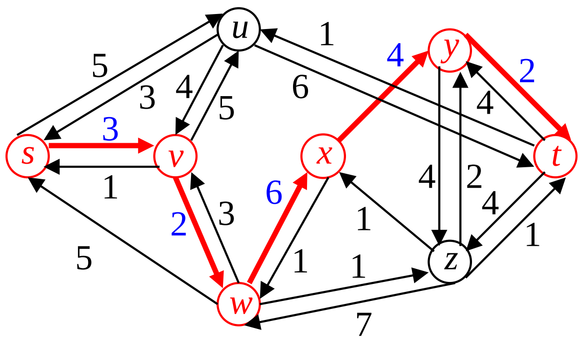
Note: There is an alternative way to look at the problem, which also helps in an understanding of the proof of correctness of the above solution. We convert the given graph G to a graph G' as follows. We split every vertex v of G into two vertices v_{in} and v_{out} connected by the (directed) edge $(v_{\text{in}}, v_{\text{out}})$. For every edge (v, w) in G, we add the edge $(v_{\text{out}}, w_{\text{in}})$ with cost $c(v, w)$. For $v \neq u$, we define the cost of $(v_{\text{in}}, v_{\text{out}})$ to be the relaying cost $c(v)$. Finally, we take the cost of $(u_{\text{in}}, u_{\text{out}})$ as 0. We run Dijkstra's original SSSP algorithm on the converted graph G' with source u_{out}. For $v \neq u$, the cheapest u, v path in G corresponds to the cheapest $(u_{\text{out}}, v_{\text{in}})$ path in G'. The conversion of G to G' runs in $O(|E| + |V|)$ time. The running time of Dijkstra's SSSP algorithm on G' is $O\left((|E| + |V|) \log(2|V|)\right)$. If we make the assumption that every node $v \in V$ is reachable from u, we have $|E| \geqslant |V| - 1$. In this case, the running time of this alternative solution is again $O(|E| \log |V|)$. However, if $|E| = o(|V|)$, then this alternative solution leads to an increase in the running time.

4. Consider the network flow shown in the following figure. Here, s is the source, and t is the sink. The capacity $c(e)$ and the current flow amount $f(e)$ are shown against the edge e as $c(e), f(e)$.



(a) Draw the residual graph for this flow. **(5)**

Solution

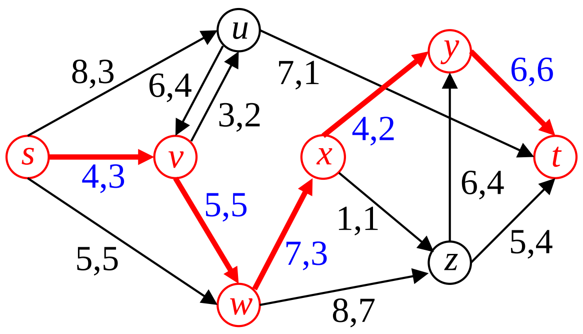


(b) Mention an s, t path in the residual graph. **(1)**

Solution s, v, w, x, y, t

(c) Redraw the network after augmenting the flow along the path of Part (b). **(4)**

Solution



5. Two computational problems P_1 and P_2 are called *polynomial-time equivalent* if there exist polynomial-time reductions $P_1 \leqslant P_2$ and $P_2 \leqslant P_1$. Prove or disprove: Every two NP-Complete problems are polynomial-time equivalent. **(5)**

Solution **True.** Let P_1 and P_2 be two NP-Complete problems. By definition, $P_1, P_2 \in \mathrm{NP}$. Since P_2 is NP-Complete, there exists a polynomial-time reduction $P_1 \leqslant P_2$. Moreover, since P_1 is NP-Complete, there exists a polynomial-time reduction $P_2 \leqslant P_1$. Therefore, P_1 and P_2 are polynomial-time equivalent.

6. A Boolean formula is said to be in the *disjunctive normal form* (or *DNF* or the *sum-of-products form*) if it is the disjunction (OR) of conjunctions (AND) of literals. For example, $(x_1 \wedge \bar{x}_3) \vee (\bar{x}_2 \wedge \bar{x}_3 \wedge x_4) \vee (\bar{x}_2)$ is in the DNF. By DNF-SAT, we refer to the computational problem of deciding whether a Boolean formula in the DNF is satisfiable. Prove that DNF-SAT $\in \mathrm{P}$. **(5)**

Solution Let $\phi = P_1 \vee P_2 \vee \cdots \vee P_l$, where each P_i is a conjunction of literals. It is evident that ϕ is satisfiable if and only if some P_i is satisfiable. Let $P_i = y_1 \wedge y_2 \wedge \cdots \wedge y_r$, where each y_i is a variable or its complement. P_i is satisfiable if and only if it is consistent to write $y_1 = y_2 = \cdots = y_r = 1$, that is, if and only if P_i does not contain contradictory literals (x and $\bar{x}$) simultaneously.

7. Two (simple undirected) graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be *isomorphic* if there exists a bijection $f : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$.

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ with $V' \subseteq V$ and with $(u, v) \in E'$ whenever $(u, v) \in E$ (for $u, v \in V'$). In other words, H is a subgraph of G if its vertex set is a subset of the vertex set of G and if all edges of G with both vertices in $V(H)$ are also edges of H.

Let SUBGRAPH-ISOMORPHISM denote the computational problem to decide, for the input of two graphs G and G', whether G contains a subgraph isomorphic to G'. Prove that SUBGRAPH-ISOMORPHISM is NP-Complete. **(10)**

Solution First, I show SUBGRAPH-ISOMORPHISM $\in \mathrm{NP}$. For an input (G, H) of SUBGRAPH-ISOMORPHISM, guess a subset $V' \subseteq V(G)$ together with a bijective function $f : V' \to V(H)$. Then, check whether f preserves the adjacency relations of V' in $V(H)$. $\boxed{3}$

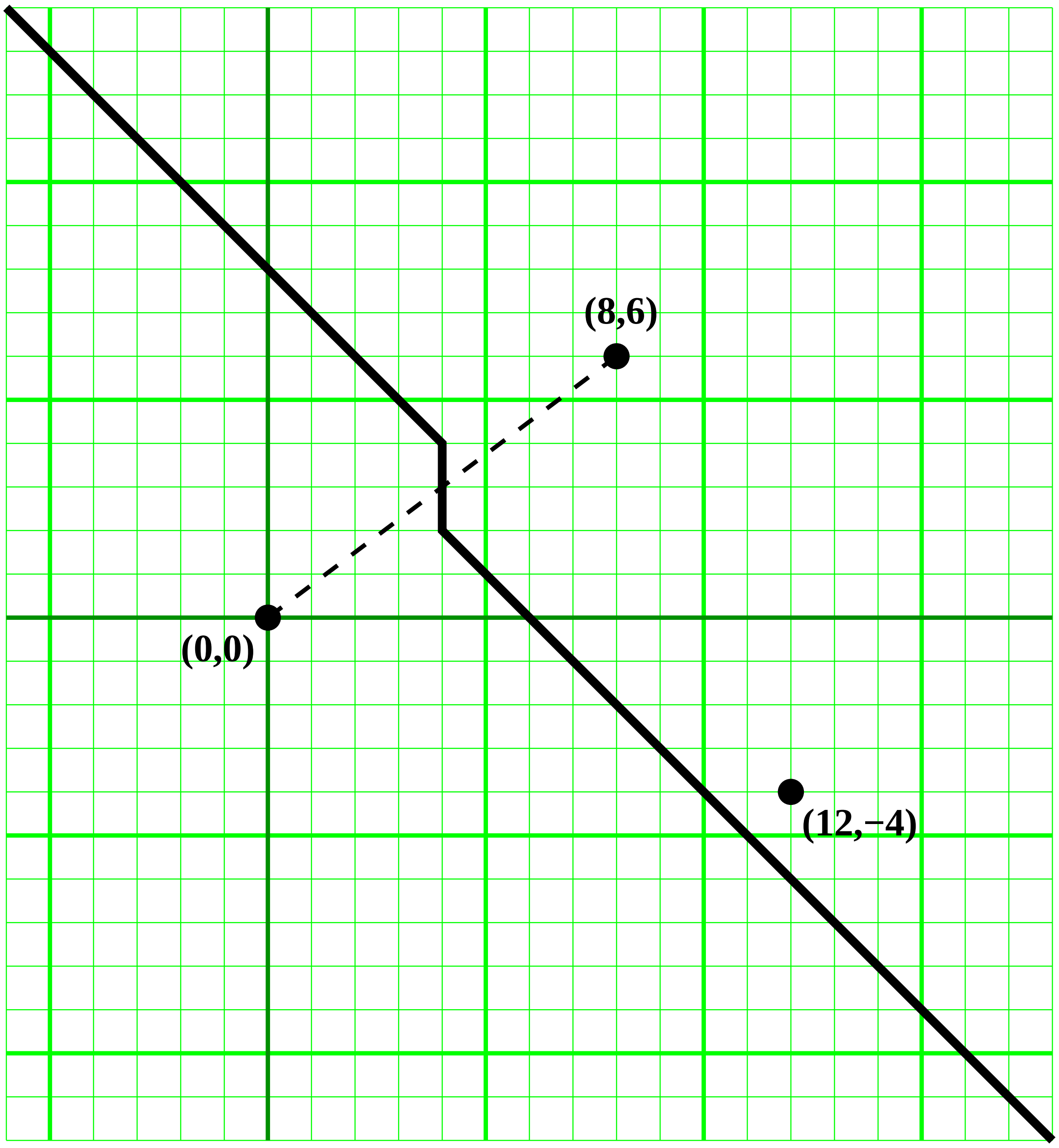
To show that SUBGRAPH-ISOMORPHISM is NP-Hard, I reduce CLIQUE to SUBGRAPH-ISOMORPHISM. Let (G, r) be an input for CLIQUE. We construct the input (G, K_r) for SUBGRAPH-ISOMORPHISM, where K_r is the complete graph on r vertices. G has an r-clique if and only if G has a subgraph isomorphic to K_r. $\boxed{6}$

This reduction evidently runs in polynomial time. $\boxed{1}$

ROUGH WORK

ROUGH WORK

ROUGH WORK


(8,6)
(0,0)
(12,−4)