CS 491G Combinatorial Optimization
Midterm

Solutions

July 16, 2001

1 *MinMax Spanning Tree* (2.17)

First of all, we must show that the optimal solution is the smallest cost c_e such that the set $\{f : f \in E, c_f \leq c_e\}$ contains all the edges of a spanning tree.

Suppose the set $\{f : f \in E, c_f \leq c_e\}$ contains a full spanning tree made up of edges whose cost values are all (strictly) less than c_e. If this were true, c_e would clearly not be the optimal solution to the *MinMax Spanning Tree* problem. But it is stated above that c_e is as small as possible. To make c_e as small as possible we can lower it until it is equal to the maximum cost of any edge in the spanning tree, throwing out all edges whose cost exceeds c_e (we can throw them out because they are not in the spanning tree). So when c_e is as small as possible, it is (strictly) equal to the maximum cost of any edge in the spanning tree, and is therefore also the optimal solution to *MinMax Spanning Tree*.

From problem 2.12 we know that an $O(m)$ algorithm exists to find some spanning tree of a graph. This leads us to the following $O(m^2)$ *MinMax Spanning Tree* algorithm:

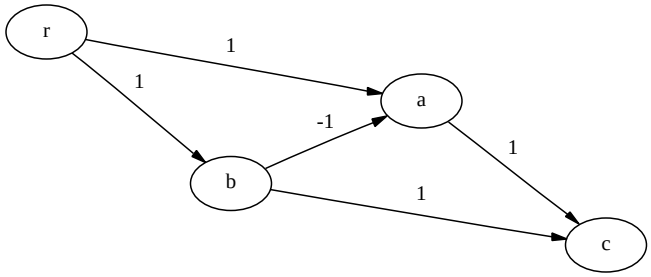
1. Sort edges by cost into a set E'.
2. Verify that a spanning tree made up of edges from E' covers all the graph's vertices.
3. Remove from E' all edges whose cost is equal to the maximum edge cost, and loop back to the previous step.

The algorithm stops as soon as it is no longer possible to make a spanning tree from the edges in E'. The solution is the cost of an edge from the last set of edges removed. The first step requires $O(m \log m)$ time to sort the edges. The "loop" (steps 2 and 3) requires $O(m)$ time and will be executed at most m times (at least one edge is removed every time). Therefore the whole algorithm requires $O(m \log m + m^2) = O(m^2)$.

We can improve the algorithm to $O(m \log m)$ by using a "divide and conquer" strategy: instead of removing the maximum cost edges, we would pick a cost equal to 1/2 the maximum and throw out all edges with cost between that and the maximum. If a spanning tree could still be made with the lower-1/2-cost edges, we would throw out the upper half of these; otherwise we would try the lower 3/4, etc. In this way we reach the solution with at most $\log m$ iterations through the loop.

2 Dijkstra's Algorithm (2.36)

The following graph, which includes negative costs, may not always be solved correctly by Dijkstra's Algorithm:



1. Initialize:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	∞	-1
b	∞	-1
c	∞	-1

queue $= r \to a \to b \to c$

2. Scan r:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	1	r
c	∞	-1

queue $= a \to b \to c$

3. Scan a:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	1	r
c	2	a

queue $= b \to c$

4. Scan b:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	0	b
b	1	r
c	2	a

queue $= c$

5. Scan c (no changes).

But the shortest path to c should be from r to b to a to c, with length $= 1$. So Dijkstra's Algorithm will not always work for this graph.

3 *MinMax Path* (2.39)

For 2.17 we took from 2.12 the fact that an $O(m)$ algorithm exists to find some spanning tree. In this case we need the fact that it is possible to find out whether there is a dipath between two vertices in $O(m+n)$ time (using *Breadth-* or *Depth-First Search*).

Our algorithm is like that of 2.17:

1. Sort edges by cost into a set E'.
2. Verify that a dipath exists from r to s, made up of only edges from E'.
3. Remove from E' all edges whose cost is equal to the maximum edge cost, and loop back to the previous step.

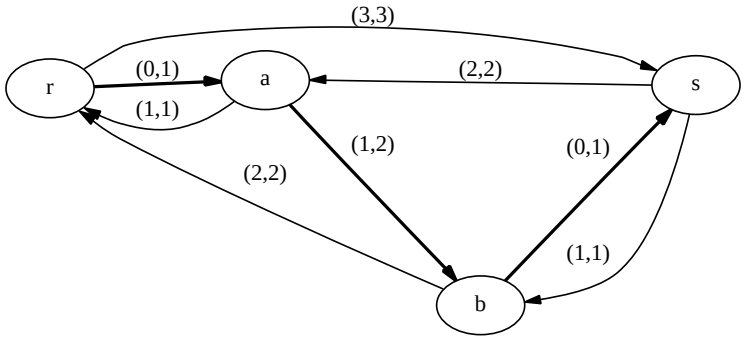
The algorithm stops as soon as it cannot find a dipath between r and s. The solution is the cost of an edge from the last set removed from E'.

Sorting requires $O(m \log m)$ time; the looping steps require $O(m+n)$ time and may be executed at most m times. Therefore the algorithm requires $O(m \log m + m(m+n)) = O(m(m+n))$ time. As before we can improve the running time using a "divide and conquer" strategy, so that the looping steps may be executed at most $\log m$ times. The improved algorithm would require $O((m+n) \log m)$ time.

4 Maximum $\vec{x}$-Widths (3.10)

Suppose an augmentation has just been performed on the $\vec{x}$-augmenting path of maximum $\vec{x}$-width. The following example shows that the maximum $\vec{x}$-width with respect to the new flow may indeed be larger.

Before the augmentation, the maximum $\vec{x}$-width = 1:



After augmenting the path in bold above (from r to a to b to s), the maximum $\vec{x}$-width with respect to the new flow = 2 (the path with maximum $\vec{x}$-width, with reverse edges from r to b to a to s, is shown bold below):