CSE 542 – Algorithms and Programs

Final Exam Solutions

Jonathan Turner *12/22/05*

1. (15 points) Suppose you're implementing a graph algorithm that uses a heap as one of its primary data structures. The algorithm does at least n *insert* and n *deletemin* operations and at most $m^{2/3}$ *insert* and $m^{2/3}$ *deletemin* operations, when $m^{2/3}>n$. It also does at most m *changekey* operations, all of which reduce the key value. If you implemented this algorithm using d-heaps, what value of d would you use to get the lowest asymptotic running time. (Hint: break this into two cases; one for smaller values of m, one for larger values.)

   For $m<n^{3/2}$, we have $m^{2/3}<n$, so the best choice is the same as for Prim's algorithm. So I would use $d=2+m/n$ in this case.

   For $m>n^{3/2}$, the running time is $O((m+m^{2/3})\log_d n + m^{2/3} d \log_d n)$. The asymptotic running time is dominated by the larger of the two terms, so we can minimize it by choosing a value of d that makes the two terms roughly equal. Choosing $d=m^{1/3}$ will work.

   What is the resulting asymptotic running time?

   $O(m \log_{2+m/n} n)$ for $m<n^{3/2}$, $O(m)$ for $m \geq n^{3/2}$.

   What is the resulting asymptotic running time if you used a Fibonacci heap, instead of a d-heap?

   $O(m + n \log n)$ for $m<n^{3/2}$, $O(m+m^{2/3} \log n)=O(m)$ for $m \geq n^{3/2}$.

   Which type of heap would you choose for this application? Why?

   Fibonacci heaps have the best overall asymptotic complexity, but they only have an advantage for $m<n \log n$. On the other hand, they use considerably more space than d-heaps (5 values per stored item vs. 3) and are more complex to implement. So, I would most likely use a d-heap, unless I knew that the input graphs were going to be very sparse and that the running time of the heap operations dominated the overall running time.

2. (15 points) In the Fibonacci heaps data structure, a cut between a vertex u and its parent v causes a cascading cut at v if v has already lost a child since it last became a child of some other vertex. Suppose we change this, so that a cascading cut is performed at v only if v has already lost *two* children. How does this change alter the lemma shown below (this lemma is from the analysis of the running time of Fibonacci heaps)? Explain your answer.

> **Lemma**. Let x be any node in an F-heap. Let $y_1, \ldots, y_r$ be the children of x, in order of time in which they were linked to x (earliest to latest). Then, $rank(y_i) \geq i-2$ for all i.

The inequality in the lemma becomes $rank(y_i) \geq i-3$. Since y_i had the same rank as x when it became a child of x and x must have had at least $i-1$ children at that time, y_i must have had rank of at least $i-1$ when it became a child of x. Since it still is a child of x, it can have lost at most two children since that time, so its rank must be at least $i-3$.

Let S_k be the smallest possible number of descendants that a node of rank k has, in our modified version of Fibonacci heaps. Give a recursive lower bound on S_k. That is, give an inequality of the form $S_k \geq f(S_0, S_1, \ldots, S_{k-1})$ where f is some function of the S_i's for $i<k$.

Clearly $S_0=1$, $S_1=2$ and $S_2=3$. For $k>2$, we can use the modified lemma to conclude that $S_k \geq 3 + S_0 + S_1 + \cdots + S_{k-3}$. Note that the difference between the bounds for S_k and for S_{k-1} is S_{k-3}.

Use this to give a lower bound on the smallest number of descendants that a node with rank 7 can have.

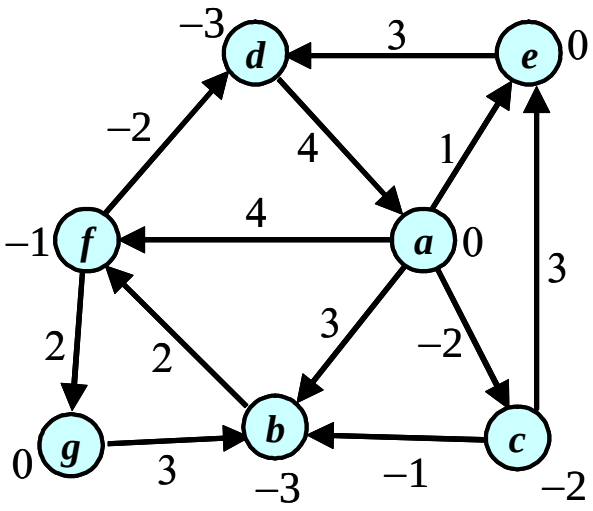
From the above, we have $S_3 \geq 3+S_0=4$, $S_4 \geq 4+S_1=6$, $S_5 \geq 6+S_2=9$, $S_6 \geq 9+S_3 \geq 13$, $S_7 \geq 13+S_4 \geq 19$.

3. (15 points) Suppose you are asked to write a program that responds to a series of queries about a given graph of the form "Will the graph have a negative cycle, if the weight of edge (u,v) is changed to x?" You may assume that the graph has no negative cycles, although it does have negative edges. Describe (in words) an algorithm that responds to such queries in $O(m+n \log n)$ time. Your algorithm may pre-compute certain information before the first query is received, but the amount of additional information is limited to $O(n)$ new values. How much time does your algorithm need to initialize this extra information?

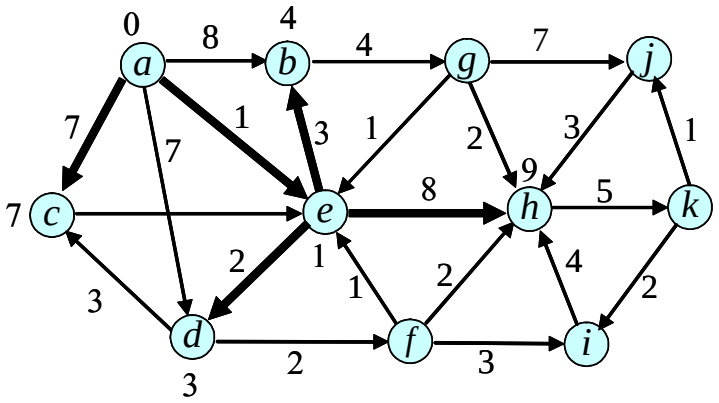
We start by computing distance labels, so that we can transform the edge costs to make them non-negative. This is done by using the breadth-first scanning algorithm to compute a shortest path tree from a new start vertex in an augmented version of the graph. We let d(z) be the shortest path distance to vertex x from the start vertex in this shortest path tree. This takes O(mn) time.

Given the values d(z), we can respond to a query about edge (u,v) using Dijkstra's algorithm (with Fibonacci heaps) to compute a shortest path tree with root v. In this computation, we use the transformed edge costs cost(x,y)+(d(x)–d(y)), where cost(x,y) is the original cost of edge (x,y). If the cost of the shortest path from v to u (applying the reverse transform to get the original cost of the shortest path) plus the proposed new cost of the edge (u,v) is negative, then we respond true, else false.

Show the auxiliary information your method would compute for the graph shown below.



4. (10 points). The figure below shows an intermediate state in the execution of Dijkstra's algorithm. The bold edges in the graph are the edges defined by the parent pointers, and the numbers next to the vertices are the current distance values. Fill in the blanks (as appropriate) in the arrays that implement the d-heap (assume d=2).



	1	2	3	4	5	6	7	8	9	10	11	12
h	*d*	*b*	*c*	*h*								

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*
key		4	7	3				9				
h^{-1}		2	3	1				4				

Show how the heap content changes after the next iteration.

	1	2	3	4	5	6	7	8	9	10	11	12
h	*b*	*f*	*c*	*h*								

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*
key		4	6			5		9				
h^{-1}		1	3			2		4				

5. (15 points) Consider a binary search tree in which each vertex has an associated key *and* a cost. The vertices are ordered by the keys in the usual way (so the keys of the vertices in the left subtree of a given vertex *x* are strictly less than the key of *x,* and so forth). The costs are represented using the differential representation we used for representing path sets. Complete the recursive function `search,` shown below, so that it returns the node with the smallest key from among those vertices with *costs* less than or equal to a given bound. The structure of the tree nodes is shown below also.

```
class twoWayTrees {
   int n;                      // trees defined on items {1,...,n}
   struct node {
      int k, Dc, Dm;           // key and differential cost fields
      int lc, rc;              // indices of left and right children
   } *vec;
   . . .
}
#define left(x) (vec[x].lc) // you may assume similar declarations
                            // for right, key, Dcost, Dmin
int search(int t, int costBound, int dmSum) {
// Return the index of the leftmost node in the subtree with
// root t that has cost <= costBound. The variable dmSum is
// the sum of the Dmin values for the proper ancestors of t.
// Return Null, if there is no node with cost less than costbound.

   int mincost;

   if (t == Null) return Null;
   mincost = dmSum + Dmin(t);
   if (mincost > costBound) return Null;

   int y = search(left(t), costBound, mincost);
   if (y != Null) return y;
   if (mincost + Dcost(t) <= costBound) return t;
   return search(right(t),costBound,mincost);
}
```

6. (15 points) In this problem, you are to show that the general preflow-push algorithm takes $O(mn)$ time to find a maximum flow in a graph in which all edges have capacity 1. (Recall that the bound for general graphs is $O(mn^2)$.) How many steps add flow to an edge in the residual graph without saturating it? Why?

   Zero. If all edges have capacity 1, any time we add flow to an edge, it becomes saturated.

   Explain why the time spent on relabeling steps is $O(mn)$.

   The maximum label for a vertex is $<2n$, so each vertex is relabeled at most n times. If vertex u_i has m_i edges incident to it, we spend $O(m_i)$ time on each relabeling, giving $O(m_i n)$ overall. Since $\sum_i m_i n = 2mn$ it follows that the overall running time is O(mn).

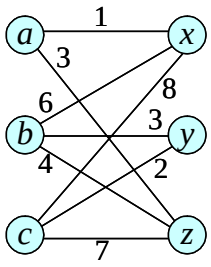
   Explain why the time spent on steps that saturate an edge in the residual graph is $O(mn)$.

   When we saturate an edge (u,v), any new flow must be added in the other direction. To add flow from v, we must first relabel v in order to make (v,u) admissible. So between any two successive saturations of an edge, one of its endpoints must be relabeled. Since each edge has two endpoints and these endpoints are relabeled $<2n$ times, the total number of times we add flow to an edge and saturate it is O(mn). Since each of these steps takes constant time, the time spent on these steps is also O(mn).

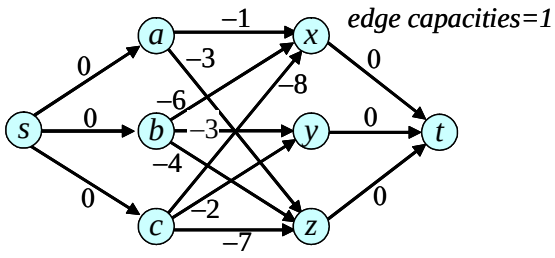
   Explain why the time spent finding admissible edges (using the *nextedge* pointers) is $O(mn)$.

   For each vertex, we make one pass through its adjacency list between successive relabelings. So, for a vertex u_i with m_i edges incident to it, we spend $O(m_i)$ time on each pass through the adjacency list and $O(m_i n)$ overall, since there are $<2n$ relabelings. Since $\sum_i m_i n = 2mn$ it follows that the overall time spent finding new admissible edges is O(mn).

7. (10 points) Given a bipartite, graph with edge weights.

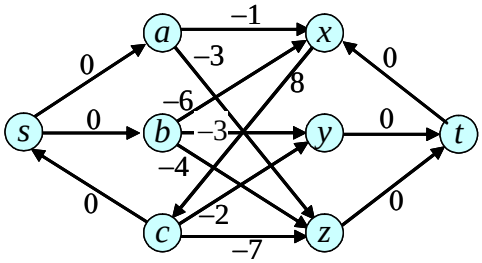


Draw a picture of the min-cost flow graph that can be used to find a maximum matching in this graph.



Identify a minimum cost augmenting path in the flow graph. What is its cost? Draw the residual graph that results from saturating this path.

The path s-c-x-t is a minimum cost augmenting path with cost -8.



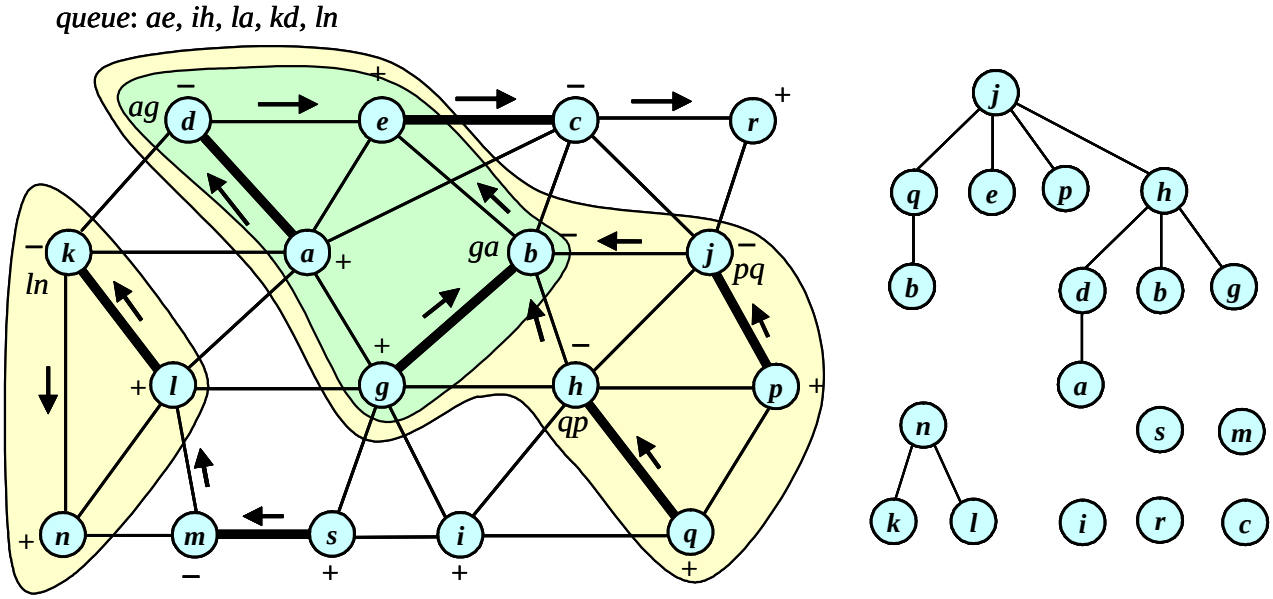
Identify a minimum cost augmenting path in the residual graph. What is its cost?

The path s-b-x-c-z-t is a minimum cost augmenting path, with cost -5.

What is the matching corresponding to the flow that exists after flow is added to this augmenting path?

{b,x}, {c,z}

8. (10 points) The figure below show a possible intermediate stage in the execution of the Edmonds-Karp algorithm for finding a maximum size matching. The states of the vertices are indicated by the plus and minus signs (for *even* and *odd*), the arrows represent the parent pointers and the edges adjacent to certain vertices correspond to their *bridge* values. The partition data structure is shown at right and the queue of edges to be processed is at the top. In the diagram at left, draw a closed curve around the sets of vertices that have been condensed into a single vertex in the current shrunken graph. If any of these sets contain subsets, corresponding to smaller blossoms, circle the vertices in those smaller blossoms, as well.



What happens when the next edge in the queue is processed by the algorithm? Explain.

Nothing. The edge ae joins two vertices that are contained in the same vertex in the current shrunken graph, so the algorithm just discards this edge.

When the next edge, [*i*,*h*] is processed, an augmenting path is found. What is that augmenting path?

i,h,q,p,j,b,g,a,d,e,c,r