MIT OpenCourseWare
http://ocw.mit.edu

6.854J / 18.415J Advanced Algorithms
Fall 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

18.415/6.854 Advanced Algorithms September 17, 2008

Lecture 5

Lecturer: Michel X. Goemans

Today, we continue the discussion of the minimum cost circulation problem. We first review the Goldberg-Tarjan algorithm, and improve it by allowing more flexibility in the selection of cycles. This gives the Cancel-and-Tighten algorithm. We also introduce splay trees, a data structure which we will use to create another data structure, dynamic trees, that will further improve the running time of the algorithm.

1 Review of the Goldberg-Tarjan Algorithm

Recall the algorithm of Golberg and Tarjan for solving the minimum cost circulation problem:

1. Initialize the flow with $f = 0$.
2. Repeatedly push flow along the minimum mean cost cycle Γ in the residual graph G_f, until no negative cycles exist.

We used the notation

$$\mu(f) = \min_{\text{cycle } \Gamma \subseteq E_f} \frac{c(\Gamma)}{|\Gamma|}$$

to denote the minimum mean cost of a cycle in the residual graph G_f. In each iteration of the algorithm, we push as much flow as possible along the minimum mean cost cycle, until $\mu(f) \geq 0$.

We used $\epsilon(f)$ to denote the minimum ϵ such that f is ϵ-optimal. In other words

$$\epsilon(f) = \min\{\epsilon : \exists \text{ potential } p : V \to \mathbb{R} \text{ such that } c_p(v,w) \geq -\epsilon \text{ for all edges } (v,w) \in E_f\}.$$

We proved that for all circulations f,

$$\epsilon(f) = -\mu(f).$$

A consequence of this equality is that there exists a potential p such that any minimum mean cost cycle Γ satisfies $c_p(v,w) = -\epsilon(f) = \mu(f)$ for all $(v,w) \in \Gamma$, since the cost of each edge is bounded below by mean cost of the cycle.

1.1 Analysis of Goldberg-Tarjan

Let us recall the analysis of the above algorithm. This will help us to improve the algorithm in order to achieve a better running time. Please refer to the previous lecture for the details of the analysis.

We used $\epsilon(f)$ as an indication of how close we are to the optimal solution. We showed that $\epsilon(f)$ is a non-increasing quantity, that is, if f' is obtained by f after a single iteration, then $\epsilon(f') \leq \epsilon(f)$. It remains to show that $\epsilon(f)$ decreases "significantly" after several iterations.

Lemma 1 *Let f be any circulation, and f' be the circulation obtained after m iterations of the Goldberg-Tarjan algorithm. Then*

$$\epsilon(f') \leq \left(1 - \frac{1}{n}\right)\epsilon(f).$$

We showed that if the costs are all integer valued, then we are done as soon as we reach $\epsilon(f) < \frac{1}{n}$. Using these two facts, we showed that the number of iterations of the above algorithm is at most $O(mn\log(nC))$. An alternative analysis using ϵ-fixed edges provides a strongly polynomial bound of $O(m^2 n \log n)$ iterations. Finally, the running time per a single iteration is $O(mn)$ using a variant of Bellman-Ford (see problem set).

1.2 Towards a faster algorithm

In the above algorithm, a significant amount of time is used to compute the minimum cost cycle. This is unnecessary, as our goal is simply to cancel enough edges in order to achieve a "significant" improvement in ϵ once every several iterations.

We can improve the algorithm by using a more flexible selection of cycles to cancel. The idea of the Cancel-and-Tighten algorithm is to push flows along cycles consisting entirely of negative cost edges. For a given potential p, we push as much flow as possible along cycles of this form, until no more such cycles exist, at which point we update p and repeat.

2 Cancel-and-Tighten

2.1 Description of the Algorithm

Definition 1 *An edge is* admissible *with respect to a potential p if $c_p(v, w) < 0$. A cycle Γ is* admissible *if all the edges of Γ are admissible.*

Cancel and Tighten Algorithm (Goldberg and Tarjan):

1. Initialization: $f \leftarrow 0$, $p \leftarrow 0$, $\epsilon \leftarrow \max_{(v,w)\in E} c(v, w)$, so that f is ϵ-optimal respect to p.
2. While f is not optimum, i.e., G_f contains a negative cost cycle, do:
   (a) Cancel: While G_f contains a cycle Γ which is admissible with respect to p, push as much flow as possible along Γ.
   (b) Tighten: Update p to p' and ϵ to ϵ', where p' and ϵ' are chosen such that $c_{p'}(v, w) \geq -\epsilon'$ for all edges $(v, w) \in E_f$ and $\epsilon' \leq \left(1 - \frac{1}{n}\right)\epsilon$.

Remark 1 *We do not update the potential p every time we push a flow. The potential p gets updated in the tighten step after possibly several flows are pushed through in the Cancel step.*

Remark 2 *In the tighten step, we do not need to find p' and ϵ' such that ϵ' is as small as possible; it is only necessary to decrease ϵ by a factor of at least $1 - \frac{1}{n}$. However, in practice, one tries to decrease ϵ by a smaller factor in order to obtain a better running time.*

Why is it always possible to obtain improvement factor of $1 - \frac{1}{n}$ in each iteration? This is guaranteed by the following result, whose proof is similar to the proof used in the analysis during the previous lecture.

Lemma 2 *Let f be a circulation and f' be the circulation obtained by performing the Cancel step. Then we cancel at most m cycles, and*

$$\epsilon(f') \leq \left(1 - \frac{1}{n}\right)\epsilon(f).$$

Proof: Since we only cancel admissible edges, after any cycle is canceled in the Cancel step:

- All new edges in the residual graph are non-admissible, since the edge costs are skew-symmetric;
- At least one admissible edge is removed from the residual graph, since we push the maximum possible amount of flow through the cycle.

Since we begin with at most m admissible edges, we cannot cancel more than m cycles, as each cycle canceling reduces the number of admissible edges by at least one.

After the cancel step, every cycle Γ contains at least one non-admissible edge, say $(u_1, v_1) \in \Gamma$ with $c_p(u_1, v_1) \geq 0$. Then the mean cost of Γ is

$$\frac{c(\Gamma)}{|\Gamma|} \geq \frac{1}{|\Gamma|} \sum_{(u_1,v_1)\neq(u,v)\in\Gamma} c_p(u,v) \geq \frac{-(|\Gamma|-1)}{|\Gamma|}\epsilon(f) = -\left(1 - \frac{1}{|\Gamma|}\right)\epsilon(f) \geq -\left(1 - \frac{1}{n}\right)\epsilon(f).$$

Therefore, $\epsilon(f') = -\mu(f') \leq \left(1 - \frac{1}{n}\right)\epsilon(f)$. □

2.2 Implementation and Analysis of Running Time

2.2.1 Tighten Step

We first discuss the Tighten step of the Cancel-and-Tighten algorithm. In this step, we wish to find a new potential function p' and a constant ϵ' such that $c_{p'}(v,w) \geq -\epsilon'$ for all edges $(v,w) \in E_f$ and $\epsilon' \leq \left(1 - \frac{1}{n}\right)\epsilon$. We can find the smallest possible ϵ' in $O(mn)$ time by using a variant of the Bellman-Ford algorithm. However, since we do not actually need to find the best possible ϵ', it is possible to vastly reduce the running time of the Tighten step to $O(n)$, as follows.

When the Cancel step terminates, there are no cycles in the admissible graph $G_a = (V, A)$, the subgraph of the residual graph with only the admissible edges. This implies that there exists a topological sort of the admissible graph. Recall that a topological sort of a directed acyclic graph is a linear ordering $l : V \to \{1, \ldots, n\}$ of its vertices such that $l(v) < l(w)$ if (v,w) is an edge of the graph; it can be achieved in $O(m)$ time using a standard topological sort algorithm (see, e.g., CLRS page 550). This linear ordering enables us to define a new potential function p' by the equation $p'(v) = p(v) - l(v)\epsilon/n$. We claim that this potential function satisfies our desired properties.

Claim 3 *The new potential function $p'(v) = p(v) - l(v)\epsilon/n$ satisfies the property that f is ϵ'-optimal with respect to p' for some constant $\epsilon' \leq (1 - 1/n)\epsilon$.*

Proof: Let $(v,w) \in E_f$, then

$$\begin{aligned} c_{p'}(v,w) &= c(v,w) + p'(v) - p'(w) \\ &= c(v,w) + p(v) - l(v)\epsilon/n - p(w) + l(w)\epsilon/n \\ &= c_p(v,w) + (l(w) - l(v))\epsilon/n. \end{aligned}$$

We consider two cases, depending on whether or not $l(v) < l(w)$.

Case 1: $l(v) < l(w)$. Then

$$\begin{aligned} c_{p'}(v,w) &= c_p(v,w) + (l(w) - l(v))\epsilon/n \\ &\geq -\epsilon + \epsilon/n \\ &= -(1 - 1/n)\epsilon. \end{aligned}$$

Case 2: $l(v) > l(w)$, so that (v,w) is not an admissible edge. Then

$$\begin{aligned} c_{p'}(v,w) &= c_p(v,w) + (l(w) - l(v))\epsilon/n \\ &\geq 0 - (n-1)\epsilon/n \\ &= -(1 - 1/n)\epsilon. \end{aligned}$$

In either case, we see that f is ϵ'-optimal with respect to p', where $\epsilon' \leq (1 - 1/n)\epsilon$. □

2.2.2 Cancel Step

We now shift our attention to the implementation and analysis of the Cancel step. Naïvely, it takes $O(m)$ time to find a cycle in the admissible graph $G_a = (V, A)$ (e.g., using Depth-First Search) and push flow along it. Using a more careful implementation of the Cancel step, we shall show that each cycle in the admissible graph can be found in an "amortized" time of $O(n)$.

We use a Depth-First Search (DFS) approach, pushing as much flow as possible along an admissible cycle and removing saturated edges, as well as removing edges from the admissible graph whenever we determine that they are not part of any cycle. Our algorithm is as follows:

Cancel($G_a = (V, A)$): Choose an arbitrary vertex $u \in V$, and begin a DFS rooted at u.

1. If we reach a vertex v that has no outgoing edges, then we backtrack, deleting from A the edges that we backtrack along, until we find an ancestor r of v for which there is another child to explore. (Notice that every edge we backtrack along cannot be part of any cycle.) Continue the DFS by exploring paths outgoing from r.

2. If we find a cycle Γ, then we push the maximum possible flow through it. This causes at least one edge along Γ to be saturated. We remove the saturated edges from A, and start the depth-first-search from scratch using $G'_a = (V, A')$, where A' denotes A with the saturated edges removed.

Every edge that is not part of any cycle is visited at most twice (since it is removed from the admissible graph the second time), so the time taken to remove edges that are not part of any cycle is $O(m)$. Since there are n vertices in the graph, it takes $O(n)$ time to find a cycle (excluding the time taken to traverse edges that are not part of any cycle), determine the maximum flow that we can push through it, and update the flow in each of its edges. Since at least one edge of A is saturated and removed every time we find a cycle, it follows that we find at most m cycles. Hence, the total running time of the Cancel step is $O(m + mn) = O(mn)$.

2.2.3 Overall Running Time

From the above analysis, we see that the Cancel step requires $O(mn)$ time per iteration, whereas the Tighten step only requires $O(m)$ time per iteration. In the previous lecture, we determined that the Cancel-and-Tighten algorithm requires $O(\min(n \log(nC), mn \log n))$ iterations. Hence the overall running time is $O(\min(mn^2 \log(nC), m^2n^2 \log n))$.

Over the course of the next few lectures, we will develop data structures that will enable us to reduce the running time of a single Cancel step from $O(mn)$ to $O(m \log n)$. Using dynamic trees, we can reduce the running time of the Cancel step to an amortized time of $O(\log n)$ per cycle canceled. This will reduce the overall running time to $O(\min(mn \log(nC) \log n, m^2 n \log^2 n))$.

3 Binary Search Trees

In this section, we review some of the basic properties of binary search trees and the operations they support, before introducing splay trees. A Binary Search Tree (BST) is a data structure that maintains a dictionary. It stores a collection of objects with ordered keys. For an object (or node) x, we use $key[x]$ to denote the key of x.

Property of a BST. The following invariant must always be satisfied in a BST:

- If y lies in the left subtree of x, then $key[y] \leq key[x]$
- If z lies in the right subtree of x, then $key[z] \geq key[x]$

Operations on a BST. Here are some operations typically supported by a BST:

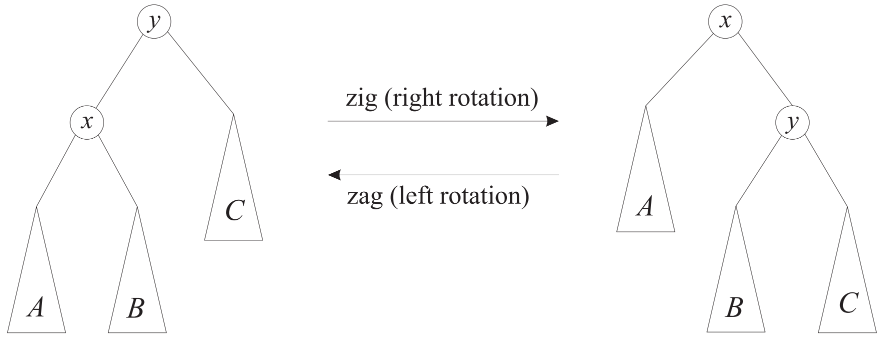
- Find(k): Determines whether the BST contains an object x with $key[x] = k$; if so, returns the object, and if not, returns false.
- Insert(x): Inserts a new node x into the tree.
- Delete(x): Deletes x from the tree.
- Min: Finds the node with the minimum key from the tree.
- Max: Finds the node with the minimum key from the tree.
- Successor(x): Find the node with the smallest key greater than $key[x]$.
- Predecessor(x): Find the node with the greatest key less than $key[x]$.
- Split(x): Returns two BSTs: one containing all the nodes y where $key[y] < key[x]$, and the other containing all the nodes z where $key[z] \geq key[x]$.
- Join(T_1, x, T_2): Given two BSTs T_1 and T_2, where all the keys in T_1 are at most $key[x]$, and all the keys in T_2 are at least $key[x]$, returns a BST containing T_1, x and T_2.

For example, the procedure Find(k) can be implemented by traversing through the tree, and branching to the left (resp. right) if the current node has key greater than (resp. less than) k. The running time for many of these operations is linear in the height of the tree, which can be as high as $O(n)$ in the worst case, where n is the number of nodes in the tree.

A *balanced BST* is a BST whose height is maintained at $O(\log n)$, so that the above operations can be run in $O(\log n)$ time. Examples of BSTs include Red-Black trees, AVL trees, and B-trees.

In the next lecture, we will discuss a data structure called *splay trees*, which is a self-balancing BST with amortized cost of $O(\log n)$ per operation. The idea is that every time a node is accessed, it gets pushed up to the root of the tree.

The basic operations of a splay tree are *rotations*. They are illustrated the following diagram.


y
x
A
B
C
zig (right rotation)
zag (left rotation)
x
y
A
B
C