18.415/6.854 Advanced Algorithms October 24, 2001

Lecture 12

Lecturer: Michel X. Goemans *Scribe: David Woodruff and Xiaowen Xin*

In this lecture we continue the presentation of the Cancel and Tighten Algorithm for solving the minimum cost circulation problem. In our initial analysis, we will determine the algorithm's running time to be $O(\min(mn^2 \log(nC), m^2 n \log(2n)))$. The latter half of today's lecture will be devoted to splay trees, a data structure which will help reduce this running time to $O(\min(mn \log(nC) \log n, m^2 n \log^2 n))$.

1 Cancel and Tighten

1.1 The Algorithm

Definition 1 *The admissible edges of a residual network G_f are defined to be the set of edges $\{(v,w) \in E_f : c_p(v,w) < 0\}$. An admissible cycle of the residual graph G_f is a cycle consisting solely of admissible edges. The admissible graph is the subgraph of G_f consisting solely of the admissible edges of E_f.*

Cancel and Tighten Algorithm:

1. Initialize f, p, ϵ.
2. While f is not optimum, i.e., G_f contains a negative cost cycle, do:
   (a) Cancel: While G_f contains an admissible cycle Γ, push as much flow as possible along Γ.
   (b) Tighten: Find p', ϵ', such that f is ϵ'-optimum with respect to p' and $\epsilon' \leq (1 - 1/n)\epsilon$.

At any point in the algorithm we will have a circulation f, a potential function p, and an ϵ such that f is ϵ-optimum with respect to p. Recall that f is said to be ϵ-optimum with respect to p if $\forall (v,w) \in E_f, c_p(v,w) \geq -\epsilon$. To initialize the algorithm, we will assume that $f = 0$ satisfies the capacity constraints (i.e., $f = 0$ is a circulation). If this is not the case, then a circulation can be obtained by solving one maximum flow problem or by modifying the instance so that a circulation can easily be found. Our initial potential function p will be the zero function. Setting

$$\epsilon = \max_{(v,w) \in E} |c(v,w)|,$$

we surely have that f is ϵ-optimal, i.e., that

$$c_p(v,w) = c(v,w) \geq - \max_{(v,w) \in E} |c(v,w)|.$$

The existence of p', ϵ', in the tighten step follows from a lemma shown last time or from Theorem 10 in the notes.

1.2 Analysis

Each time we complete a tighten step, we reduce ϵ by a factor of $1 - 1/n$. Starting with

$$\epsilon = C = \max_{(v,w) \in E} |c(v,w)|,$$

a simple argument given last time shows that ϵ will be less than $1/n$ after $O(n\log(nC))$ tighten steps, which will then imply that f is optimal. Alternatively, after $n\log(2n)$ iterations of tighten, one additional edge is ϵ-fixed, and that edge remains fixed because $\epsilon(f)$ does not increase as the algorithm progresses. Using this alternative analysis, we obtain the strongly polynomial bound of $O(mn\log(2n))$ iterations of the tighten step. To determine the overall running time of the Cancel and Tighten Algorithm we need to determine the time spent for each cancel step and for each tighten step.

Implementing the Tighten Step: To implement the tighten step, we could use an extended Bellman-Ford algorithm to find the minimum mean cost cycle in $O(mn)$ time. This would be sufficient for our purposes, but is not necessary. We need only find a p', ϵ', such that f is ϵ'-optimum with respect to p' and $\epsilon' \leq (1-1/n)\epsilon$. After the cancel step completes, we know there are no cycles in the admissible graph. Therefore we can take a topological ordering of the admissible graph to rank the vertices so that for every edge (v,w) in the admissible graph, $rank(v) < rank(w)$. This ranking of the vertices can be done in $O(m)$ time using a standard topological sort algorithm. Letting $l(v) : V \to \{0,1,2,...,n-1\}$ denote the rank function on the vertices V of G_f, we have $l(w) > l(v)$ if (v,w) is an admissible edge of E_f. For each vertex v of V, we define our new potential function p' by $p'(v) = p(v) - l(v)\epsilon/n$.

Claim 1 *p' is such that f is ϵ'-optimal with respect to p', where $\epsilon' \leq (1-1/n)\epsilon$.*

Proof of claim 1: Suppose $(v,w) \in E_f$. Then,

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

Case1: (v,w) was admissible:

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

Case2: (v,w) was not admissible:

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

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

Implementing the Cancel Step: We now shift our focus to the implementation of the cancel step. The goal in the cancel step is to saturate at least one edge in every cycle in the admissible graph $G = (V, A)$. We will implement this with a Depth-First-Search (DFS), pushing flow to saturate and remove edges every time we encounter a cycle, and removing edges whenever we determine that they cannot be part of any cycle. The algorithm is:

Cancel Step algorithm($G = (V, A)$)**:**

1. Choose v in V. Begin a DFS rooted at v.
   (a) If we encounter a vertex w that we have previously encountered on our path from v, we have found a cycle Γ. In this case, we let $\delta = \min_{(v,w)\in\Gamma} u(v,w)$. We then increase the flow along each of the edges of Γ by δ, saturating at least one edge. We remove the saturated edges from A, and recurse with $G' = (V, A')$, where A' denotes the edges of A with the saturated edges removed.
   (b) If we encounter a vertex w such that there are no edges emanating from w in G, we backtrack until we find an ancestor r of w for which there is another child to explore. As we backtrack, we remove the edges we backtrack along from A until we encounter r. We continue the DFS by exploring paths emanating at r.

Every edge $e \in A$ that is not part of any cycle is visited at most two times, and hence the running time to remove edges that are not part of any cycle is $O(m)$. For each cycle that we cancel, we need to determine

$$u = \min_{(v,w)\in\Gamma} u(v,w)$$

and update the flow along each of the edges of the cycle. Since there can be at most $n-1$ edges in a cycle, we spend $O(n)$ time per cycle cancelled. Since we saturate and remove at least one edge from A every time we cancel a cycle, we can cancel at most m cycles. Hence, the overall running time of the cancel step is $O(m + nm) = O(mn)$.

Overall Running Time: Note that the cancel step requires $O(mn)$ time per iteration, whereas the tighten step only requires $O(m)$ time. Hence, the cancel step is the bottleneck of the Cancel and Tighten Algorithm. Earlier we determined that the number of iterations of the Cancel and Tighten Algorithm is $O(\min(n\log(nC), mn\log(2n)))$. Hence the overall running time is $O(\min(mn^2\log(nC), m^2n^2\log(2n)))$.

In the following sections we will create data structures which will reduce the running time of a single cancel step from $O(mn)$ to $O(m\log n)$. We will make use of dynamic trees which will enable us to spend an amortized cost of $O(\log n)$ per cycle cancelled. The overall running time will then be $O(\min(mn\log(nC)\log n, m^2n\log^2 n))$.

2 Binary Search Trees

A Binary Search Tree (BST) is a data structure that stores a dictionary. A dictionary is a collection of objects with ordered (and we'll assume unique) keys.

2.1 Structure of a BST

A BST stores objects as nodes in a binary tree such that $key[x] \leq key[y]$ if and only if x is to the left of y (see Figure 1).

2.2 Operations on a BST

Here are some operations that a typical BST supports:

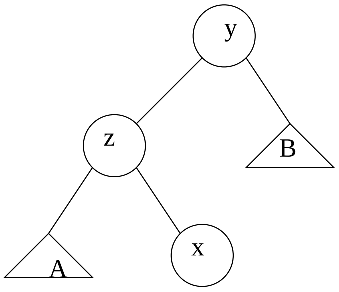

Figure 1: A BST where x is to the left of y.

1. $find(x)$: returns true if x is in the tree, otherwise returns false. This is implemented by traversing the tree: when you encounter an element y, branch left if $key[y] > key[x]$, branch right if $key[y] < key[x]$, and return true if $key[y] = key[x]$. If you cannot traverse any more, then return false. This runs in $O(h)$ time, where h is the height of the BST.

2. $insert(x)$: inserts x into the tree. This is implemented by traversing the tree similar to $find(x)$ above until you reach the end, and inserting x as a leaf node. This runs in $O(h)$ time.

3. $delete(x)$: deletes x from the tree. This is implemented by first finding x. If x is a leaf node, delete it, otherwise swap x with its immediate successor. Since the immediate successor of x is guaranteed to be a leaf node, x becomes a leaf node, and we can delete it. This runs in $O(h)$ time.

4. min: find the minimum element in the entire tree.

5. max: find the maximum element in the entire tree.

6. $successor(x)$: find the element y with the smallest $key[y]$, where $key[y] > key[x]$.

7. $predecessor(x)$: find the element y with the greatest $key[y]$, where $key[y] < key[x]$.

8. $split(x)$: returns two dictionaries such that one of them contains elements y: $key[y] \geq key[x]$ and the other one contains elements z: $key[z] < key[x]$.

9. $join(T_1, x, T_2)$: given T_1 which contains elements z: $key[z] \leq key[x]$ and T_2 which contains elements y: $key[y] \geq key[x]$, return a BST that contains T_1, x, and T_2.

In the worse case, the height of the tree could be equal to the number of elements in the tree, so the running time of these operations is linear in the number of elements. A "balanced" BST is a BST such that the height is maintained at $O(\log n)$, where n is the number of nodes in the tree. With such a tree, above operations would run in $O(\log n)$. A balanced BST can be implemented using a Red-Black tree, AVL, or B-tree.

2.3 Rotations

In order to mutate the layout of a BST we can rotate parts of the tree to raise or lower nodes. Figure 2 shows a right rotation operation, known as a zig. A left rotation, called a zag, is the mirror image of this. In the following, we will see how we can use these operations to lower the *amortized cost* of the operations on a BST.

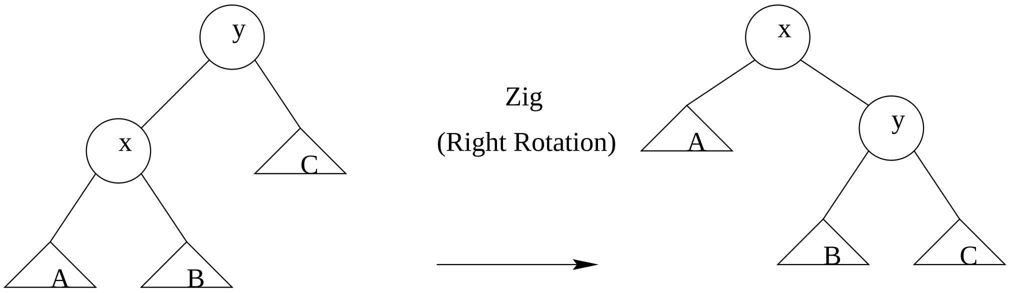

Figure 2: Zig (right rotation) operation.

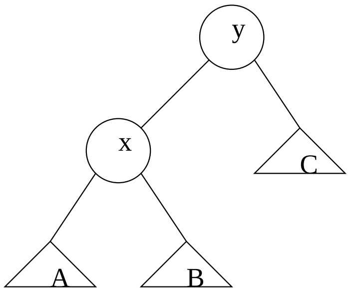

Figure 3: case 1.

3 Splay Tree

A Splay Tree is a BST which performs a splay operation every time an access is made. We will prove that *the amortized cost* of one operations on a splay tree is $O(\log n)$. We will see that this implies that for every k, the total running time of any sequence of k operations on the tree is $O((k+n)\log n)$.

3.1 Splay Step

$splay\_step(x)$ attempts to move x two levels higher if it's not already the root.

- **case 1:** $parent(x) = root$ (Figure 3): If x is the left child of its parent, then do a zig, otherwise, do a zag to make x the root.
- **case 2:** both x and $parent(x)$ are left children or they are both right children of their parents. Let $y = parent(x)$ and $z = parent(y)$.
  - If x and y are left children, then do a $zig(z)$ followed by a $zig(y)$. (See Figure 4).
  - If x and y are right children, then do a $zag(z)$ followed by a $zag(y)$.
- **case 3:** x is a right child while $parent(x)$ is a left child or vice versa. Again, let $y = parent(x)$ and $z = parent(y)$.
  - If x is a right child and y is a left child, then do a $zag(y)$ followed by a $zig(z)$. (See Figure 5).
  - If x is a left child and y is a right child, then do a $zig(y)$ followed by a $zig(z)$.

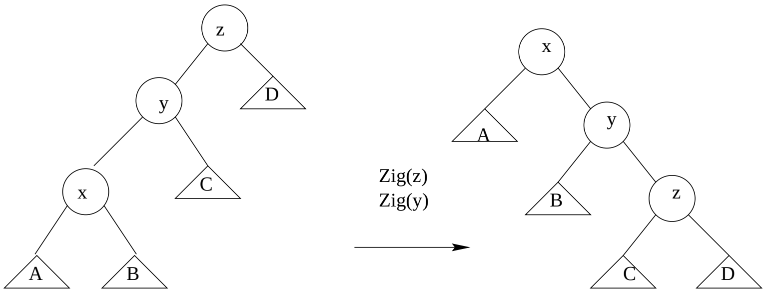

Figure 4: case 2.

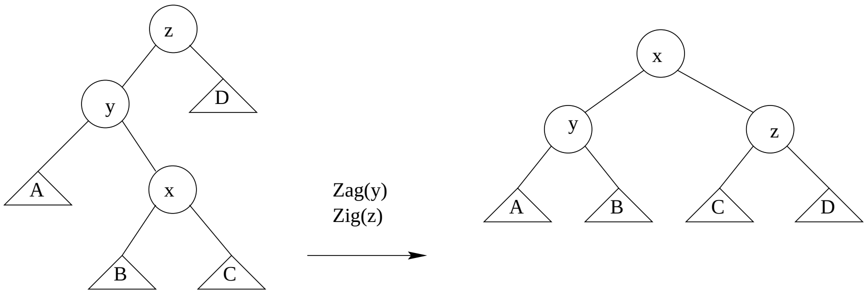

Figure 5: case 3.

3.2 Splay

The operation *splay*(x) moves x to the root by repeatedly calling *splay_step*(x). The Splay Tree data structure is defined by the following rule: When you access a Splay Tree, always call *splay*(x) where x is the last element you access. For instance, when calling $find(x)$, if you find x in the Splay Tree, you should then call *splay*(x). If you don't find x, but instead find the leaf y, you should call *splay*(y). For deletions, splay is called on the parent of the element x being deleted, if the element x is in fact found in the Splay Tree.

3.3 Amortized Analysis of Splay Tree

Assume *splay_step* takes 1 unit of time. Suppose for every node x of the tree we have assigned a weight $w(x) \geq 0$ (These weight are assigned only for the sake of analysis of the running time of the algorithm and do not appear in the actual implementation of the algorithm). We define:

$$\begin{aligned} S(x) &= \sum_{y \in \text{tree rooted at } x} w(y), \\ r(x) &= \lfloor \log_2(S(x)) \rfloor. \end{aligned}$$

We call $r(x)$ the *credit* of the vertex x. We define our *credit invariant* as the sum of $r(x)$ for all nodes x in the tree. Now, we can define the *amortized cost* of an operation as follows: If the operation has cost (i.e., running time) k, the amortized cost of the operation is $k + \sum_x r'(x) - r(x)$. We prove the following two lemmas in the next lecture.

Lemma 2 *The amortized cost of one splay_step(x) is at most $3(r'(x) - r(x)) + c$, where c is 1 if x is a parent of the root, and 0 otherwise.*

Lemma 3 *The amortized cost of splaying a tree of root v at node x is at most $3(r(v) - r(x)) + 1$.*

Notice that if we take $w(x) = 1$ for every vertex x of the tree, then Lemma 3 implies that the amortized cost of splaying a tree at any node is at most $3 \log n$ (this is because the credit of any node is at most $\log(n)$), even though the actual cost may be n. Intuitively, this means that the extra cost is paid by the credits in the tree.