15-451 Assignment 7

Kenny Lu krlu@andrew.cmu.edu
Mike Wehar mwehar@andrew.cmu.edu
Shashank Singh sss1@andrew.cmu.edu

May 4, 2012

Parallel 2-Dimensional Linear Programming

(a) Given an array of n elements, Algorithm will proceed as follows:

(a) Spliting the array:

(i) Split array down the middle, dividing into subarrays of roughly equal size

(ii) Recursively split each subarray with step 1, using separate processors for each subarray. Stop when the subarray is a singleton. Final result should be a delimited list of singletons

(b) Finding max/min element in result by comparisons:

(i) Merge adjacent singletons computed, and output the maximum of the two as a new singleton.

(ii) For each pair of singletons, use separate processors for each pair of elements to perform step3, once again, we recursively compute the maximum of two elements and stop when array of comparables is a singleton.

Amount of time is $O(\log n)$ since as each level of the recursion, we are doing constant work, and the depth of the recursion tree is $O(\log n)$.
Amoung of work done is O n) since then number of split and the number of comparisons are both at most n.

For solving 1-D LP problems, we want to maximize $Cx, C \in \mathbb{R}, x \in \mathbb{R}^n$, subject to the constraints:
$a_i x \leq b_i$ for each $i \in \{1, ...n\}$ and $x \geq 0$.
The constraints imply $x \leq \frac{b_i}{a_i}$, in which case we want to find $i \in \{1, ...n\}$ such that $\frac{b_i}{a_i}$, we can do this using the above algorithm, in O(logn) time and O(n) work. All that remains is to consider the following cases:
if($\frac{b_i}{a_i}$) the result is in the infeasible, hence no solution.
else if($c > 0$) $x = \frac{b_i}{a_i}$
else if($c < 0$) $x = 0$
■

(b) Consider a point P in our bounding box. For point P to be an optimal point for our LP solution, it must be on the intersection of two half planes.

For the k-th half plane to have resulted in an unsatisfied constraint, the k-th cut must intersect point P, because that would mean point P was originally on the corner of a different region defined by the intersection of the previous k-1 half-planes.

Following directly from the LP construction of P's position on the corner of the feasible region, the probability for any single cut to intersect P is $1/k$. Thus for a point P on the intersection of 2 half planes, it is $2/k$, since only 1 needs to have given an unsatisfied constraint.

We then take the sum $\sum_{k=0}^{n} \frac{2}{k} = 2(k\text{th harmonic number}) = \text{O(2*logn)} \in \text{O(logn)}$.

■

(c) Given a list of n half-planes and a point P (as in part (b)). Suppose we have evaluated the intersection of the first k half planes (and their associated cuts).
We take the remaining $n-k$ half planes, and we cut the subarray in half, with the two resultant subarrays have start and end indices $(n-k, \lfloor n - \frac{n-k}{2}) \rfloor$ and $(\lfloor n - \frac{n-k}{2}, n) \rfloor$. We then use separate processors for each recursive split on those subarrays. We stop splititng until we have singleton arrays (similar to the splitting in part (a)).
We then evaluate the singletons in parallel using the processors we've called during the recursive step. For each pair of singletons, we pass up to the next step the lowest indexed (supposing we have list ranking or an indexed array) cut that results an unsatisfied constraint, and recursively compare them each above step until there is only 1 singleton S left. If P does not violate its constaint, then there are no half-planes whose constraints are unsatisfied. Otherwise, if P does violate constraint S, then index of S be the lowest of all unsatisfied cuts in subarray $(n-k+1, n)$, making the half-plane of S the first for which P violates its contraints.

(d) After building our list of n half-planes, we must fine each one that results. Letting P be a point within our bounding box, we observe that P will be outside the feasible region O(log n) times given n cuts across the bounding box, by part (b).
Thus for each cut that does this, we search our list using the algorithm in part(d), and then, use the algorithm in part(a) to solve the resulting 1-D LP problem.

In total, we end up with a runtime of $O(\log n \cdot (\log n + \log n)) = O(\log^2 n)$ in parallel, and work of $O(logn \cdot (n+n)) = O(n \log n)$ operations.

Maximum Independent Set in a Tree

(a) Let S_1 be an independent set in T, and suppose v is a leaf in T with $v \notin S_1$. Let $S_2 = (S_1 \cup \{v\})\backslash(N(v))$, where $N(v)$ denotes the set of neighbors of v. Note that, since v is a leaf, $|N(v)| = 1$, so that, since $v \notin S_1$, $|S_2| = |S_1|$. Since S_2 contains no neighbors of v and $S_2\backslash\{v\} \subseteq S_1$ S_2 is independent. Assuming T has more than two vertices, and is connected, the neighbor of v cannot itself be a leaf. Thus, for any maximum independent set of T not containing all the leaves of T, there is a maximum independent of T containing strictly more leaves of T, so that there exists a maximum independent set containing all leaves of T. ■

(b) Beginning with the lowest level of the tree, iteratively perform the following:

   (a) Add any leaves in the level to the independent set.

   (b) For each vertex v that is not a leaf in the current level, if v has no child in I, add v to I.

   (c) Repeat steps 1 and 2 on the level of the parents of the vertices in the current level unless the current level is the root (in which case, terminate).

   We claim, I will be a maximum independent set. That I is independent is clear. To see that I is maximum, consider a maximum independent set S containing all the leaves of T (such an S much exist by the result of part (a)). If $S \neq I$, then consider a lowest (farthest from the root) vertex v such that $v \in S$ but $v \notin I$. Since $v \notin I$, one of its children u must be in I. By choice of v, however, $u \in I$, contradicting the fact that S is independent. Therefore, no such v can exist so that $S \subseteq I$, and $|S| \leq |I|$. ■

   By implementing the algorithm as a post-order traversal, where constant time (the time to check whether any child of the current vertex is in v) is spent at each vertex, the algorithm runs in linear time, as desired. ■

(c) We give a constant-time linear-work reduction to a similar problem P concerning arithmetic expression trees presented in lecture. Label each leaf in the tree with a 1, and every other vertex as follows: if the vertex has one child, assign it the expression $(1 - c_1)$, where c_1 is the value of that child; if the vertex has two children, assign it the expression $(1-c_1)(1-c_2)$, where c_1 and c_2 are the the values of its children. Evaluate the expression tree using tree contraction as in lecture. Then, include a vertex in the maximum independent set I if and only if the value at that vertex is a 1.

   Since $(1 - c_1)(1 - c_2)$ (or $(1 - c_1)$) in the case of one child node) is 1 if and only if c_1 and c_2 are 0, a vertex is in I if and only if neither of its children is. Thus, the algorithm returns the same tree as that returned by the algorithm given in part (b) above, so that the same proof of correctness applies. Since the reduction is constant-time linear-work, and the solution to P runs in $O(\log n)$ time with $O(n)$ work, the algorithm runs in $O(1 + \log n) = O(\log n)$ with $O(n + n) = O(n)$ work, as desired. ■

NP-Completeness

(a) Given a 3-CNF fomula $F = C_1 \wedge ... \wedge C_n$ and an assignment A, if A is a $\neq$-assignment for F then $\neg A$ is a $\neq$-assignment for F.

Proof. If A is a $\neq$-assignment for F, then for each i, A assigns true to at least one literal in C_i and false to at least one literal in C_i. Therefore, for each i, $\neg A$ assignes true to at least one literal in C_i and false to at least one literal in C_i. Hence, $\neg A$ is a $\neq$-assignment for F.

(b) Claim: $\neq$-SAT is NP-$Complete$.

Proof. $\neq$-SAT is in NP since given an assignment A, we can check if A is a $\neq$-assignment in linear time.

Now, it remains to show that $\neq$-SAT is NP-$Hard$. We will do this by reducing 3-SAT to $\neq$-SAT. Let a 3-CNF formula $F = C_1 \wedge ... \wedge C_n$ be given. Define $F' = D_1 \wedge ... \wedge D_n$ such that $D_i = (y_1 \vee y_2 \vee z_i) \wedge (\bar{z_i} \vee y_3 \vee b)$ where $C_i = (y_1 \vee y_2 \vee y_3)$ for each i.

Subclaim: F has a satisfying assignment iff F' has a $\neq$-assignment.

$\rightarrow$: Suppose that F has a satisfying assignment A. We will now construct a $\neq$-assignment A' for F'. (1) A' will assign the same values to variables in F as A. (2) A' assigns false to b. (3) For each i, we will split into cases to define z_i. Let y_1, y_2, and y_3 so that $D_i = (y_1 \vee y_2 \vee z_i) \wedge (\bar{z_i} \vee y_3 \vee b)$. We assign true to z_i if A assigns true to y_3 and false to y_1 and y_2. Otherwise, we assign false to z_i. Therefore, for each clause in F', A' assigns true to at least one literal and false to at least one literal. Hence, A' is a $\neq$-assignment for F'.

$\leftarrow$: Suppose that F' has a $\neq$-assignment A'. By part (a), $\neg A'$ is a $\neq$-assignment for F'. Hence, either A' or $\neg A'$ is a $\neq$-assignment that assigns false to b. WLOG A' assigns false to b. Define an assignment A for F such that A agrees with A' on variables of F. Now, I will show that A is a satisfying assignment for F. To do this, we just need to show that C_i is satisfied for each i. Let y_1, y_2, and y_3 so that $D_i = (y_1 \vee y_2 \vee z_i) \wedge (\bar{z_i} \vee y_3 \vee b)$ and $C_i = (y_1 \vee y_2 \vee y_3)$. Since A' is a $\neq$-assignment for F', it assigns true to D_i. And, since b is assigned false, either y_1, y_2, or y_3 is assigned true. Therefore, A assigns true to C_i. It follows that A is a satisfying assignment for F.

(c) Claim: $MAXCUT$ is NP-$Complete$.

$MAXCUT$ is in NP since given a cut of a graph G and given a number k, we can check if k or more vertices are in the cut in quadratic time.

Now, it remains to show that $MAXCUT$ is $NP\text{-}Hard$. We will do this by reducing $\neq\text{-}SAT$ to $MAXCUT$. Let a 3-CNF formula F be given. Construct the graph G that is described in the problem hint.

Subclaim: F has a $\neq$-assignment iff $(G, r) \in MAXCUT$ where $r := 2k + 9k^2n$, k is the number of clauses, and n is the number of variables.

$\rightarrow$: Suppose that F has a $\neq$-assignment A. Consider the cut associated with A, where each literal that is assigned true is in one set and each literal that is assigned false is in the other. In total, there are $3k+9k^2n$ edges where $9k^2n$ edges are adjacent to variables and their negations, and $3k$ edges are associated with clauses. The cut associated with A has the $9k^2n$ edges since each variable and its negation are separated. The cut has 2k additional edges because there is one literal from each clause triange in each set. (Each clause triangle adds 2 edges.) Therefore, the cut has r edges and hence $(G, r) \in MAXCUT$.

$\leftarrow$: Suppose that $(G, r) \in MAXCUT$. Choose a cut (S, T) with at least r edges. This cut must include the $9k^2n$ edges that are adjacent to variables and their negations. Otherwise, it would have at most $9k^2(n-1) + 3k$ edges which is less than r. Therefore, this cut separates variables and their negations and hence is associated with an assignment A. Since the cut can have at most 2 edges per clause triangle and the cut has $2k$ more edges, there must be exactly two edges from each clause triange in the cut. Therefore, for each clause, A assigns true to at least one literal and false to at least one literal. It follows that A is a $\neq$-assignment for F.