CS 5800: Algorithms K.Lieberherr

Due: March. 24, 2014

Homework Module 9

1 Submission Rules

`http://www.ccs.neu.edu/home/lieber/courses/algorithms/cs5800/sp14/homeworks/submission-rules.pdf`

2 CLRS

Solve CLRS exercise 34.4-4.

3 LeafCovering

You study a practical problem appearing in the implementation of Programming Languages. It is about statically checking a multiple-dispatch program guaranteeing that certain run-time errors cannot appear. It is similar to the transition from dynamic to static checking for HSR.

The general problem is co-NP-hard problem but fortunately, there are many polynomial-time special cases. You develop fast algorithms for those.

`Reading:`
`http://mathworld.wolfram.com/GraphCartesianProduct.html`
`http://en.wikipedia.org/wiki/Function_problem`

We investigate the following theme: We have a computational problem which is hard to solve in general. Fortunately, we have only inputs that satisfy a specific property. This allows us to speed up the algorithm for the subset that satisfies the property.

We have seen this before: k-coloring a graph is NP-hard but 2-coloring a graph is polynomial. Finding a satisfying assignment for a CNF is NP-hard but is polynomial if the clauses are of length 2.

We first introduce the LeafCovering problem which is computationally hard and then we look at a special case for which we want to find a polynomial-time algorithm.

According to Wikipedia, there are 5 kinds of computational problems: decision, search, counting, optimization and function problems. The LeafCovering problem is a function problem that uses a counting problem in the solution approach.

Terminology: $T_i = (V_i, E_i)$ is a tree with nodes V_i and edges E_i. DAG = directed acyclic graph.

Definition: Graph Cartesian product (GCP). Given n directed trees $T_1, ..., T_n$, where $T_i = (V_i, E_i)$, the graph Cartesian product is a DAG $GCP(T_1, ..., T_n)$ that has as nodes the tuples $(v_1, ..., v_n)$ where v_i in V_i and the following edges: Node $(v_1, ..., v_i, ..., v_n)$ has as

children all nodes $({}_v1, ..., v_{i'}, ..., v_n)$, where $v_{i'}$ is a child of $v_i \in T_i$, and all other components are the same.

Note that GCP is a DAG (directed acyclic graph), so leaves don't repeat. Each leaf appears exactly once.

3.1 The LeafCovering Problem

- INSTANCE Given n directed trees $T_1, ..., T_n$ defining the graph Cartesian product $GCP(T_1, ..., T_n)$, and a set of nodes, M, in $GCP(T_1, ..., T_n)$.
- SOLUTION Does each leaf in $GCP(T_1, ..., T_n)$ have some ancestor in M? The answer is either "yes" (all leaves are covered) or "no". If the answer is "no", you also provide a non-covered leaf, called a witness.

Claim: LeafCovering is co-NP hard. Compare with CLRS exercise 34.4-4: Whether a boolean formula is a tautology is complete for co-NP.

3.2 Examples of Graph Cartesian Product

```
Tree T1
Baum : Node | Leaf.

Tree T2:
Exp : Lit | Bin.
Bin : Add | Sub.

GCP(T1,T2):
http://www.ccs.neu.edu/home/lieber/courses/algorithms/cs5800/sp14/
homeworks/m9/graphCartesianproduct1.png

Another example:
Tree T1,T2:
T:L|R.

GCP(T1,T2):
TT : LT | RT | TL | TR
LT : LL | LR
RT : RL | RR
TL : LL | RL
TR : LR | RR
```

There are a total of 9 nodes. The leaves are: LL,RL,LR,RR.

If, for example, M = (TL,TR), then there is no uncovered node while if M = (RT,TL) then LR is a witness (the only one). You can think of this problem as a cube covering problem (here we cover the 2x2 cube).

This looks like a rather abstract problem but it has a very useful application: It is a type checking problem that multiple dispatch programming systems must answer. There are many more applications.

3.3 Brute Force

There is a straight-forward brute-force algorithm: Explore all leaves of the GCP and check whether each is covered by at least one element of M. Because there are exponentially many leaves, the brute-force algorithm is not practical.

Can you avoid an explicit representation of the graph Cartesian product which would clearly lead to an exponential algorithm?

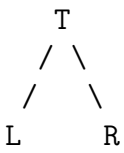
3.4 Polynomial Special Case

Now we turn our attention to the special case LeafCovering(2), where $|M| = 2$. In general, LeafCovering(k) is the restricted problem where M contains only k elements.

Can you propose a polynomial-time algorithm for LeafCovering(2)?

3.5 DEBATES

We will explore the LeafCovering(2) problem in a debate called called LEAF(2) on Piazza and in your homework. For the debate, we simplify the problem further by assuming that all trees are of this form:



(all edges are directed downward) I.e., all trees have one root and two children.

Therefore, an instance now consists of: n: the number of directed trees. M: a set of n tuples, e.g., if n=3, M=(L,T,T), (T,R,L).

With this simplification it is much easier to define inputs and the problem is still interesting and a solution will generalize to the original problem. We could use the term cube covering problem instead of leaf covering problem. We try to cover the n-dimensional cube.

3.5.1 Representing M

Because we often have many more Ts than Ls and Rs in the nodes in M, we choose a more economical representation:

(!v3,v9) is a convenient way to represent (T,T,L,T,T,T,T,T,R). So we represent a node in GCP as a set of literals. The first variable is v1.

During the debates on Piazza you don't reveal your algorithms: they are your trade secret until after the due date. On Piazza you will use instances where a brute-force algorithm would fail to find a solution in a reasonable amount of time. If we assume the set-up of the textbook, and choose n to be 50 or larger, it takes 36 years with brute-force. Therefore, when you attack a claim use an instance where $n \geq 50$.

In LEAF(2) the claims are of the form: LEAF-Claim(2,r,s,t) meaning:

r is a constant factor. s is the exponent of n. t is the cross-over point (we used to call this n0 in the Landau notation). We are ready with an algorithm for LeafCovering(2) that uses at most $r * (n^s)$ milliseconds if $n \geq t$. n is the number of input trees. Your objective is to keep r and s as small as possible. Highest priority is to have a small s and then to have a small r. t should be at most 10. Notice that we not only strive for good asymptotic run-time but also for good practical run-time with small constants. The 2 appears in the claim because $|M| = 2$.

LEAF-Claim(2,r,s,t) says: There exists an algorithm LC for LeafCovering(2) so that for all instances inp(n) of LeafCovering(2) with $n \geq t$ and for which there exists an uncovered leaf, LC correctly solves inp(n) in at most $r * (n^s)$ milliseconds.

You make this prediction based on the analysis of your algorithm and by running your algorithm on various test cases and using the data to determine the constants r, s, t.

For the purpose of the debates, it is important that the validity of the solution can be checked quickly. Therefore, we require that the instance given has a non-covered leaf. We use again the honor system: when you try to refute a claim you must give an instance which has a non-covered leaf. After the due date you may be asked to reveal an uncovered leaf for some of the instances that you provided. In other words, as falsifier you never reveal the uncovered leaf unless asked by the teaching staff.

Note that the verifier might find a different uncovered leaf than the secret one the opposer has.

Note that we can efficiently check whether the leaf is uncovered: Check for both elements of M whether there is a path to the leaf. This can be done efficiently.

We use the honor system: you must report the run-time faithfully. What you report can be checked after you have publicized your algorithm after the due date. You must follow the scientific method and make your run-times reproducible.

Claim LEAF-Claim(2,r1,s1,t1) is stronger than claim LEAF-Claim(2,r2,s2,t2) if $s1 < s2$ or (if $s1 = s2$ and $r1 < r2$)

Some slides about visualizing the notation used in LEAF(2) see: LeafCovering.ppt in the m9 directory.

3.5.2 JSON language

We represent an n-tuple of L,R,T by numbering (starting from 1) the elements of the tuple from 1 to n. If L is at position j, we use -j. If R is at position j, we use j. T is not mentioned. All the missing numbers represent Ts.

(!v3,v9) is a convenient way to represent (T,T,L,T,T,T,T,T,R). We write this now as: [-3,9]. [-1,2,3,4,5,6,7,-8] for n=8 represents the leaf: [L,R,R,R,R,R,R,L]

```
{"instance": "LeafCovering", "n": 3, "M": [[-1,-2,3],[1,3]]
}

Solution
{"solution": "LeafCovering", "answer": "covered"}
{"solution": "LeafCovering", "answer": "uncovered",
  "uncovered_leaf": [-1,-2,3] }


With GSON we can use the following classes in Java:
class Instance{
String instance;
int n;
int [][] M;
}

class Solution{
String solution;
String answer;
int[] uncovered_leaf;
}
```

3.6 What to turn in

- Debates For LEAF(2): Pointers to your Piazza debates in your team. Find a fast algorithm through interacting with your partners.
- Fast Algorithm for LEAF(2) Turn in your algorithm with an argument of correctness and describe how you tested the algorithm.
- FAST Algorithm for LEAF(3) There is a lab LEAF(k) for each k. Describe how your algorithm generalizes to k=3.

Remember the hint I gave earlier: use a counting problem for the solution. You heard about the counting problem most likely in Discrete Structures.

4 Debate Evaluation Algorithms [40 points]

Based on Ahmed Abdelmeged's Dissertation.

Algorithm families play an important role in computer science. Consider the family P-ALGS of all polynomial-time algorithms. We would like to know whether P-ALGS contains an algorithm for the Satisfiability problem.

Here we deal with a more tractable family of algorithms. In this homework we study debate evaluation algorithms at a higher level. We consider all ranking algorithms that map a labeled graph representing a table about debate outcomes into a ranking. But we restrict ourselves to ranking algorithms which satisfy three properties (or axioms) that you are already familiar with:

- Non-Negative Effect for Winning (NNEW)
- Non-Positive Effect for Losing (NPEL)
- Collusion-Resilience (CR)

Can you say something interesting about any algorithm mapping labeled graphs to rankings and which satisfies the three properties: NNEW, NPEL and CR.

Let's restate NNEW and NPEL from homework 7. It is unacceptable for a ranking function to reward losing or to penalize winning. That is, a player's rank cannot be worsened by an extra win nor can it be improved by an extra loss.

4.1 NNEW

If $x \leq^T y$ for table T then $x \leq^{T'} y$ for table T' where T' contains an additional row where x won. We call this property Non-Negative Effect for Winning I (NNEW.I).

If $x \leq^T y$ for table T then $x \leq T''y$ for table T'' where T'' contains one fewer row where y won. We call this property Non-Negative Effect for Winning II (NNEW.II).

NNEW = NNEW.I and NNEW.II.

4.2 NPEL

If $x \leq^T y$ for table T then $x \leq T'y$ for table T' where T' contains an additional row where y lost. We call this property Non-Positive Effect for Losing (NPEL.I).

If $x \leq Ty$ for table T then $x \leq T''y$ for table T'' where T'' contains one fewer row where x lost. We call this property Non-Positive Effect for Losing (NPEL.II).

NPEL = NPEL.I and NPEL.II.

4.3 Background Recall

Here comes further background from hw6 and the midterm. You have been playing debates (semantic games) and you have recorded your debate results in tables such as the following one. You have column headers **PW**, **PL**, **Forced**. There is an additional column *Fault* which is computed from the first three. Each row of the table represents a debate g. **PW** and **PL** are the columns for the two participants. Because there is always a winner (there

are no draws) we choose the **PW** column to show the winner. The **Forced** column shows which participant is forced. Remember that at most one is forced. We use entry "none" indicating that none of the two participants was forced. The *Fault* column depends on the columns **PL** and **Forced** and indicates whether the participant in column **PL** had a fault (lost in non-forced position).

We use the following Boolean functions for a debate g: g.participates(p) = p is a participant in debate g (either the winner or the loser). g.wins(p) = g.participates(p) and participant p wins in debate g. g.loses(p) = g.participates(p) and participant p loses in debate g. g.forced(p) = g.participates(p) and participant p is forced in debate g. g.fault(p) = g.participates(p) and participant p makes a fault in debate g. Note that !g.wins(p) = !g.participates(p) or g.loses(p).

Definition of fault: g.fault(p) = g.participates(p) and g.loses(p) and !g.forced(p).

Similar to the fault concept there is the control concept.

Definition of control: g.control(p) = g.participates(p) and (g.wins(p) or !g.forced(p)). Informally, a participant p is not in control in a debate if p is not involved in the debate or p loses while forced. Again informally we say that a participant p controls a debate if p either wins or p had a guaranteed opportunity to win (was non-forced). In a debate there is at least one of the two participants in control: the winner.

Consider algorithms that translate a debate table into a ranking of the participants. What is a ranking? A reflexive, transitive and complete binary relation.

Informally, our goal is to find the "best" participants based on the debates. We consider two ranking algorithms.

The first ranking algorithm, called $\leq_{WC}$, is based on win counting while the second, called $\leq_{FC}$ is based on a special kind of loss counting, called fault counting.

We fix a set of debates D. $x \leq^D y$ means that participant x is ranked weakly better than participant y for D. Sometimes we omit the superscript D when it is clear from context. Informally, weakly better means that either x is better then y or x and y have the same rank.

You might need some of the following in your answer: $wins(x)$ is the number of wins of participant x over all debates in D. Formally, $wins(x)$ is the number of rows $g \in D$ which satisfy $g.wins(x)$. $faults(x)$ is the number of faults that x makes over all debates in D. Formally, $faults(x)$ is the number of rows g in D which satisfy $g.fault(x)$.

In principle, all the functions defined above would require a superscript for D. For example, $wins^D(x)$ is the number of wins of participant x over all debates in D. However, we omit it in order not to clutter the definitions.

4.4 CR

Definition of collusion resilience: A ranking $\leq$ of participants is said to be collusion-resilient if for all debate tables D and for any two participants x and y, the property $x \leq^D y$ implies $x \leq^E y$, where E is D with added debates g that x cannot control, i.e., for which !g.control(x).

Informally, if $x \leq^D y$ and more debates that x cannot control are added to D resulting in E, then $x \leq^E y$ holds.

CR prevents debaters from gaming the debates by preventing another debater to be top ranked through intentional losses.

4.5 Example of Algorithms that satisfy NNEW and NPEL and CR

We show examples of ranking algorithms which have the desirable properties. The goal of the homework question is to figure out a property that is common to all of the algorithms.

4.5.1 Fault Counting

$\leq_{FC}$ is **defined** as: $x \leq_{FC} y$ if $faults(x) \leq faults(y)$.

NNEW and NPEL you have shown in hw 7. CR you have shown in the midterm.

4.5.2 Weighted Fault Counting

There are two kinds of faults that happen during a debate: (1) The debaters have chosen their prefered sides which are opposite to each other. (2) The debaters have chosen the same side.

If a fault happens with context (1), we count the fault with weight w_{opp}. With context (2) the weight is w_{same}.

$wfaults(x)$ is the number of weighted faults that x makes over all debates in D. Formally, $wfaults(x)$ is the number of weighted rows g in D which satisfy $g.fault(x)$.

$\leq_{WFC}$ is **defined** as: $x \leq_{WFC} y$ if $wfaults(x) \leq wfaults(y)$.

With the weights we can influence the competitiveness of the debaters. When w_{opp} is considerably less than w_{same} the debaters will avoid losing against a forced debater. On the other hand, the forced debater will be strongly motivated to try to find weaknesses in the opponent.

4.5.3 All Equal

$\leq_=$ is **defined** as: $x \leq_= y$ if true.

All debaters are ranked the same. (Everybody gets an A independent of performance.)

4.6 Make a claim and defend it

4.6.1 3 properties

Make a claim about the family of ranking algorithms that have all three properties NNEW, NPEL and CR.

4.6.2 NNEW and CR (2 properties)

Make a claim about the family of ranking algorithms that have two properties NNEW and CR.

Justify your claims by providing a proof. But the focus is on expressing the correct claims. Informal arguments for correctness are acceptable.

4.6.3 Hint

A ranking algorithm takes as input a table and maps it to a ranking relation for the debaters. We consider filters on the table which keep the ranking relation obtained invariant. The filter eliminates some rows from the table which we cannot use because of the CR and NNEW properties that we want to hold.

For any pair (x,y) of debaters, the relative rank that $\leq$ assigns to x with respect to y ($x \leq^D y$) for a table D satisfies: $x \leq^D y \iff x \leq^{f(D,x,y)} y$ where $f(D, x, y)$ is suitably defined.

f expresses which games are important to compute the ranking if we want CR and NNEW to hold.

Can you propose a suitable f?