

CS386 Assignment 3

Due: Friday 10/9 10PM ET

1. Do (g) and two more:

- (a) Construct your own instance (3 players is enough) and a priority order such that running the DRAFT protocol from class produces an allocation that is EF1 but fails EFX. State the allocation, and identify exactly which envy relationship breaks EFX and which item is the problem.
- (b) In class we argued that the DRAFT protocol is EF1. We didn't really formalize the proof - do so here.
- (c) In class we argued that the CUT & CHOOSE protocol is EFX. We didn't really formalize the proof - do so here.
- (d) Does EF1 imply proportionality? Does proportionality imply EF1? For each direction, either prove it or provide a counterexample.
- (e) Does MMS imply proportionality? Does proportionality imply MMS? For each direction, either prove it or provide a counterexample.
- (f) Consider the following languages:

$\text{EF-DECISION} = \{ \langle G \rangle \mid G \text{ is an instance of indivisible items with given valuation functions for each player where an EF allocation exists} \}$

$\text{MMS-DECISION} = \{ \langle G, k_1, \dots, k_n \rangle \mid G \text{ is an instance of indivisible items with given valuation functions for each player where the MMS of player } i \text{ is at least } k_i \}$

$\text{MNW-DECISION} = \{ \langle G, k \rangle \mid G \text{ is an instance of indivisible items with given valuation functions for each player where the MNW of the instance is at least } k \}$

Pick one and show it's **NP**-Complete (by means of reduction from PARTITION).

- (g) i. Show that SUBSETSUM is **NP**-Complete by means of reduction from 3SAT. We didn't write it out on the board for everything, so I want you to formally show each step. Be explicit about the algorithm that takes as input an instance of 3SAT and outputs an instance of SUBSETSUM.
- ii. Determine the exact numbers created from the following instances of 3SAT:
 - $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$
 $\wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$

$$\bullet (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \\ \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

- iii. Notice that the second instance has every way of creating a clause from 3 variables along with their negations without creating a clause with a variable and its negation. Explain why that immediately means that no assignment will work. Additionally, explain why the exclusion of one of those clauses (as in the case of the first instance) makes exactly one assignment work.
- iv. For the first instance, identify the numbers created in the SUBSETSUM instance that sum to the target value. Show that the same numbers do not sum to the target value in the second instance.

2. This section previews a distinction you'll want next Tuesday.

- (a) Let $\text{KNAPSACK} = \{ \langle (w_1, v_1), \dots, (w_n, v_n), W, V \rangle \mid \exists S \subset [n] \text{ such that } \sum_{i \in S} w_i \leq W \text{ and } \sum_{i \in S} v_i \geq V \}$.

Show that KNAPSACK is **NP-Complete** (by means of reductions from SUBSETSUM). The reduction should be very easy - don't overcomplicate it. (*The w_i s are supposed to denote weight and the v_i s are supposed to denote value - basically, you're trying to fill a bag that can hold at most W weight with at least V value*)

- (b) Design an algorithm that decides KNAPSACK in $O(nW)$ time. You can assume the w_i s and v_i s are positive integers if you'd like, but technically this assumption is unnecessary. If you have taken CS375 and know how Dynamic Programming works, this is easy. Otherwise, I recommend assuming the w_i s are positive integers and considering the following: suppose we want to construct a 2D array A of size $(n+1) \times (W+1)$ where $A[i, j]$ will denote the *best* value I can get from the first i items using at most j weight. Then clearly

$$A[0, j] = 0 \text{ for all } j, A[1, j] = \begin{cases} 0 & j < w_1 \\ v_1 & w_1 \leq j, \end{cases} \text{ and we can extend this item by item for}$$

a bit but obviously the formula gets harder to write down. What we would really like is a *simple* recursive formula for $A[i, j]$. Note that whatever the right answer is, it either includes item i , or it doesn't. If the best solution over the first i items of weight at most j **doesn't** include item i , then surely it is also the best solution over the first $i-1$ of weight at most j ! If it **does** include item i , then its value is $v_i + A[\text{____}, \text{____}]$. Since it either includes item i or doesn't, $A[i, j]$ is just the maximum of the above two cases. Use these ideas to fill the whole array in $O(nW)$ time. Which entry tells me whether I can get V value in under W weight?

- (c) After doing the above problem, you should be concerned that it sure looks like you've just solved **P** vs **NP** - $O(nW)$ sure *looks* polynomial. Explain carefully why the runtime of this algorithm is not necessarily polynomial **in the size of the input**. (The key quantity to think about: how many *bits* does it take to write down the input, versus how large is W itself? Your algorithm from part (b) is polynomial in W , but is it polynomial in the input's bit-length?)

A problem like this that is **NP-Complete**, but solvable in time polynomial in the numeric values involved (just not in the bit-length) is called **weakly NP-complete** (or **NP-Complete** "in the ordinary sense"). A problem that stays **NP-Complete** even when every number in the instance is bounded by a polynomial in the input length (so writing numbers in unary wouldn't even help) is called **strongly NP-complete**. (3-PARTITION, which is the same idea as PARTITION, but splitting into 3 equal sets instead of 2, is the standard

example of the strong kind; no algorithm like part (b)'s seems likely to exist for it, under standard assumptions.)

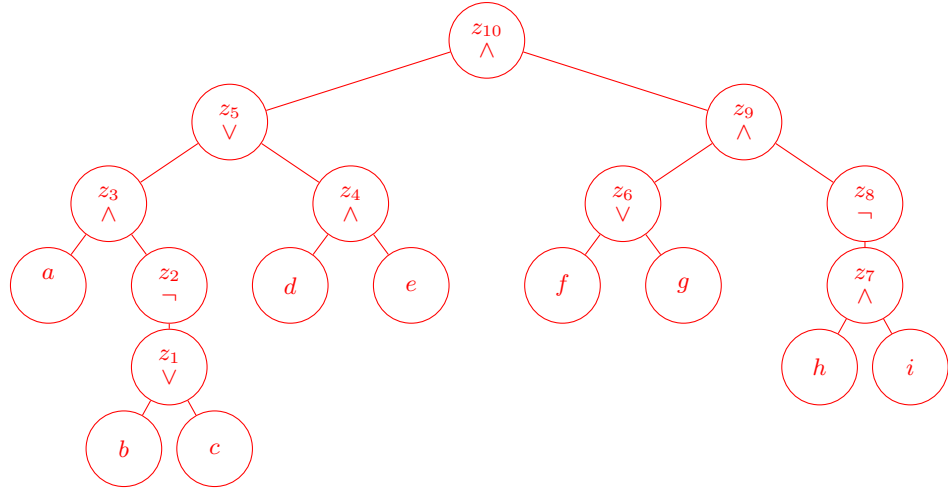
3. In class, we said that you can show that $\text{SAT} \leq_p 3\text{SAT}$. There are two steps to that:

- (a) Convert the given formula into Conjunctive-Normal-Form (CNF). This first step is called a Tseitin transformation. It's not that hard, so let's do it! A formula is in CNF if it is just a conjunction (AND) of a bunch of clauses that are each a disjunction (OR) of a bunch of literals. So it looks like 3SAT instances, except that the clauses can be any number of literals.

- i. First, suppose we drew our boolean formula as a tree. So, given the formula

$$((a \wedge \neg(b \vee c)) \vee (d \wedge e)) \wedge ((f \vee g) \wedge \neg(h \wedge i))$$

we could imagine it as:



Because I'm a nice guy, I'll even give you the algorithm for the transformation (see Algorithm 1 on the next page). First, apply the algorithm to the tree and determine the formula that gets created. Your formula should have 29 clauses.

- ii. Next, let's prove the individual 'gadgets' work
- Think about lines 5 - 10: let's say we're looking at some subtree rooted at v , a $\neg$ node. Think about the subtree rooted at its child u . Suppose by induction that z_u is the resulting evaluation of the tree rooted at u . Why does $(z_v \vee z_u) \wedge (\neg z_v \vee \neg z_u)$ enforce that $z_v = \neg z_u$?
 - Think about lines 11 - 17: let's say we're looking at some subtree rooted at v , a $\wedge$ node. Think about the subtrees rooted at its children u, w . Suppose by induction that z_u and z_w are the resulting evaluation of the tree rooted at u and w . Why does $(\neg z_v \vee z_u) \wedge (\neg z_v \vee z_w) \wedge (z_v \vee \neg z_u \vee \neg z_w)$ enforce that $z_v = z_u \wedge z_w$?
 - Think about lines 18 - 24: let's say we're looking at some subtree rooted at v , a $\vee$ node. Think about the subtrees rooted at its children u, w . Suppose by induction that z_u and z_w are the resulting evaluation of the tree rooted at u and w . Why does $(z_v \vee \neg z_u) \wedge (z_v \vee \neg z_w) \wedge (\neg z_v \vee z_u \vee z_w)$ enforce that $z_v = z_u \vee z_w$?

Algorithm 1 Tseitin Transformation

```
1: procedure TSEITIN( $v$ )
2:   if  $v$  is a leaf labeled by variable  $x$  then
3:     return  $x$ 
4:   end if
5:   if  $v$  is a  $\neg$ -node with child  $u$  then
6:      $z_u \leftarrow \text{TSEITIN}(u)$ 
7:     Create a fresh variable  $z_v$ 
8:     Add clauses  $(z_v \vee z_u)$  and  $(\neg z_v \vee \neg z_u)$ 
9:     return  $z_v$ 
10:  end if
11:  if  $v$  is an  $\wedge$ -node with children  $u, w$  then
12:     $z_u \leftarrow \text{TSEITIN}(u)$ 
13:     $z_w \leftarrow \text{TSEITIN}(w)$ 
14:    Create a fresh variable  $z_v$ 
15:    Add clauses  $(\neg z_v \vee z_u)$ ,  $(\neg z_v \vee z_w)$ , and  $(z_v \vee \neg z_u \vee \neg z_w)$ 
16:    return  $z_v$ 
17:  end if
18:  if  $v$  is an  $\vee$ -node with children  $u, w$  then
19:     $z_u \leftarrow \text{TSEITIN}(u)$ 
20:     $z_w \leftarrow \text{TSEITIN}(w)$ 
21:    Create a fresh variable  $z_v$ 
22:    Add clauses  $(z_v \vee \neg z_u)$ ,  $(z_v \vee \neg z_w)$ , and  $(\neg z_v \vee z_u \vee z_w)$ 
23:    return  $z_v$ 
24:  end if
25: end procedure
26:
27:  $z \leftarrow \text{TSEITIN}(\text{root})$ 
28: Add the unit clause  $(z)$ 
29: return all clauses added
```

iii. Okay, so put it together. Explain why

- A. this algorithm *works*, ie show that the resultant clauses all and-ed together are satisfiable if and only if the original formula is satisfiable.
- B. this algorithm takes a polynomial amount of time

Great! So now we've shown that we can take any formula φ and compute in polynomial time an equivalent formula φ' that is in CNF (equivalent meaning that φ is satisfiable if and only if φ' is satisfiable).

(b) So now that we can assume we have a CNF formula, let's convert it into a 3SAT formula.

- i. Suppose that a given formula φ is in CNF and consider a clause $C = (x_1 \vee x_2 \vee \dots \vee x_k)$ with $k > 3$. Give a construction that replaces this clause with a collection of clauses $C' = (\dots) \wedge (\dots) \wedge \dots$, each containing exactly three literals, using additional variables if necessary. Show that your construction is equivalent - that is, show that an assignment satisfying C induces an assignment satisfying C' and vice-versa.
- ii. Now consider a CNF formula containing a clause with fewer than three literals. Give

constructions for replacing a 1-literal clause and a 2-literal clause with equivalent collections of 3-clauses, again using additional variables if necessary.

- iii. Put these parts together. Give a polynomial-time algorithm that takes an arbitrary Boolean formula φ in CNF as input and produces an equivalent 3SAT formula φ' . Prove that φ is satisfiable $\iff \varphi'$ is satisfiable and that the size of φ' is polynomial in the size of φ .

(c) Alright, we've got two big steps achieved. Put it together: explain how we have shown that $\text{SAT} \leq_p \text{3SAT}$.

4. Consider a fair-division instance with additive, nonnegative valuations, and let $A = (A_1, \dots, A_n)$ be an allocation maximizing Nash welfare. Assume that every player receives positive value in A .

(a) Show that A is Pareto Optimal. *Probably a proof by contradiction is easiest here.*

(b) Show that A is EF1. *Suppose otherwise, let i envy j by more than i 's value of any of j 's items. Let g^* be the item that minimizes $\frac{v_i(g)}{v_i(g)}$ (among all items g that i values positively, note that there must be at least one since we're supposing that i envies j). Let A' be the allocation identical to A except that we give g^* to i instead. Show that Nash Welfare is higher in A' than A .*

Note that this shows that maximizing Nash welfare is an extremely strong goal. Although I'm not asking you to prove it (I probably will next week), an MNW allocation also guarantees each player a π_n percentage of its MMS, where

$$\pi_n = \frac{2}{1 + \sqrt{4n - 3}}.$$

So for 3 players, it guarantees everyone gets $\frac{1}{2}$ their MMS. Obviously it gets worse as n increases, but that's a lot of nice guarantees. As a result, there's been a lot of work done in how to calculate MNW efficiently for 'nice' instances, or approximate it when it's too hard.