Yap Dian Hao A0184679H Tutorial 7

CS3230 Assignment 3.

Date No.

1 Show that PLUGIT is NP-hard.

Reduction:

(1) Each variable in 3-SAT can be mapped to a powered socket and a broadcast node in PLUGIT. Since each variable X_i or its negation $\neg X_i$ will appear at least one, this instance will always exist. For example, say variable X_1:

X_1: ⏚ △

$\neg X_1$: ⏚ △

(2) Since each variable ~~must~~ exist at least once, a valid 3-SAT will assign a fixed value (True or False) to a variable X_i, and $\neg X_i$ is its negation. Similarly, if 3-SAT assigns X_i to be true, the powered socket mapped by X_i is going to power the broadcast node next to it. For example, X_1 in 3-SAT is true:

X_i: ⏚ ⟶ ⚠

$\neg X_1$: ⏚ △

This step is to prevent two values mapped to the same value, which will lead to an invalid instance of 3-SAT.

(3) The distance between both broadcast nodes are far enough so that one does not power another.

(4) Next, we are going to represent clauses in PLUGIT. We model each variable in a clause to be a broadcast node, and an unpowered socket. Furthermore, the broadcast node and the socket are close enough to be able to power it. Besides, all the variables are aligned in a column. For example, a clause with $(X_1 \vee X_2 \vee X_3)$ will look like this:

Date No.

$(x_1 \vee x_2 \vee x_3)$

x_1: △ ○

$\neg x_1$: △ ← Here we do not include a socket because $\neg x_1$ is not in the clause.

x_2: △ ○

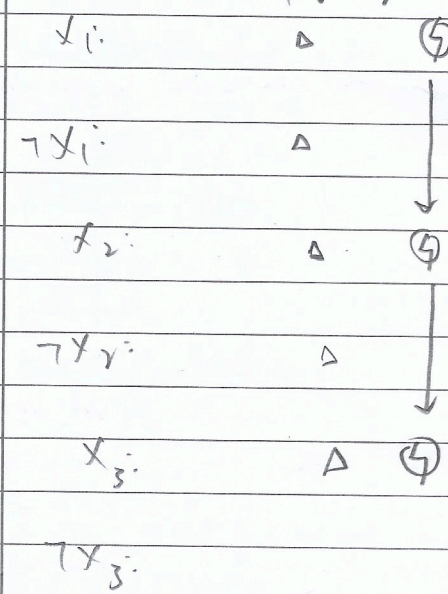
$\neg x_2$: △

x_3: △ ○

$\neg x_3$: △

Note that if one of these sockets are powered, all the sockets in that column is powered. Even if it is the middle socket that is powered, we can do a simple reordering to observe the rules of no two wires can be drawn, but that is not a concern of this mapping. For example:

$(x_1 \vee x_2 \vee x_3)$



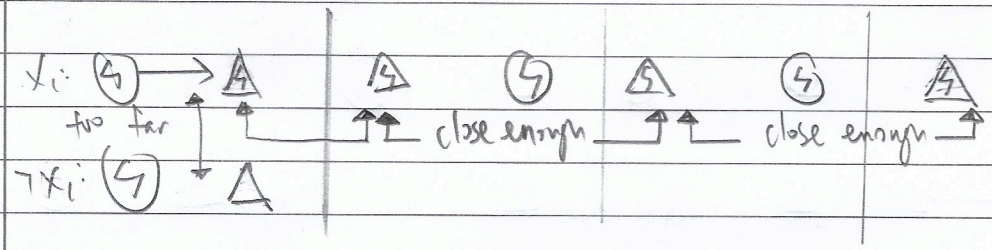
If x_1 socket is powered, a wire can be drawn to a socket below it and this will hold for subsequent sockets underneath it. Therefore, $(x_1 \vee x_2 \vee x_3)$ is true if any one literal is true, and this holds for PLUGIT

Date No.

(5) Now we are going to show how a true literal will power a socket. Recall the literal instance and clause instance of PLUGIT; and if we place them adjacent to each other:

Literal	$C_1 = (x_1 \vee x_2 \vee x_3)$	$C_2 = (x_1 \vee \neg x_2 \vee x_3)$	$C_3 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$
x_i: ϟ △	△ ○	△ ○	△
$\neg x_i$: ϟ △	△	△	△ ○

Note that the column distance between each broadcast node is far enough, but the row broadcast node's distances are close enough to power all the nodes in the same row. For example, x_1 is true:

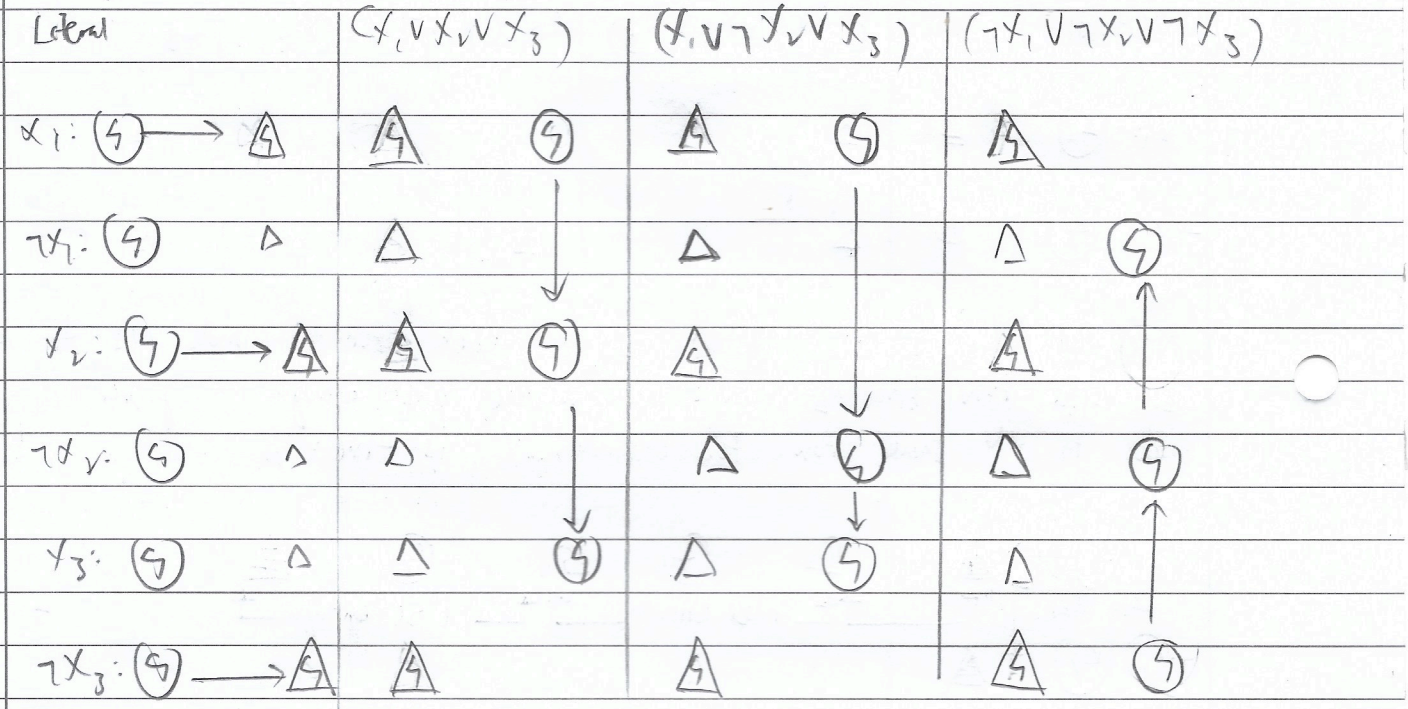


(6) Next, we are going to show the 3-SAT instance maps to a PLUGIT instance. For example, the 3-SAT is $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$

Literal	$(x_1 \vee x_2 \vee x_3)$	$(x_1 \vee \neg x_2 \vee x_3)$	$(\neg x_1 \vee \neg x_2 \vee \neg x_3)$
x_1: ϟ △	△ ○	△ ○	△
$\neg x_1$: ϟ △	△	△	△ ○
x_2: ϟ △	△ ○	△	△
$\neg x_2$: ϟ △	△	△ ○	△ ○
x_3: ϟ △	△ ○	△ ○	△
$\neg x_3$: ϟ △	△	△	△ ○

Date No.

(3) Now we are going to show a YES-instance in 3-SAT will correspond to a YES-instance in PLUGIT. In the previous example, the 3-SAT will satisfy if x_1 = True, x_2 = True y_3 = False.



Here we see that all the sockets are powered, therefore a yes-instance of 3-SAT maps to a yes-instance of plugit.

Date No.

2. Proving in NP.

Assume that the certificate given follows the rules of PLUGIT, which it is a valid-instance.

The certificate takes the form of a list l, where the elements in l are the powered sockets.

Now let U be the set of all the sockets. The verification algorithm loops through the powered socket list l, for each powered socket find all the other sockets that can be powered via this current socket, either by direct wire connection or powering of nearby broadcast nodes. Let S call this set of derived powered socket S. Then, remove S from U. When the algorithm terminates, simply check if U is empty, i.e. no unpowered sockets left, and return yes and no accordingly.

It is clear that the looping of sockets take poly time, and the finding of removing of sockets take $O(|U|)$ worst. Since encoding issues and $\log n$ factors of inputs are not a concern of PLUGIT, the verification algorithm runs in polynomial time. The certificate is in poly size too, since it is just a list of unpowered sockets.

With question 1 proving PLUGIT is NP-hard and question 2 giving a polynomial time verification algorithm, and polynomial sized certificate, it is shown that PLUGIT is NP-complete.

Date No.

3. To show the problem is NP-complete, I will first show a reduction from Partition to Subset-sum, and after showing Subset-Sum is NP-complete, this problem can be shown to be NP-complete.

Let k to be the maximum of n, number of people who owe money, and m, number of people who are owed. To show that subset-sum is NP-complete, for a given target sum X and an input set $S = \{a_1, a_2, a_3 \ldots a_k\}$ simply partition S into two sets S_1 and S_2 and set X to be $\frac{1}{2}\sum_{i=1}^{k} a_i$. It is clear that no pre-processing for S is needed, therefore the reduction runs in polynomial time, and shown that subset-Sum is NP-complete.

To prove the problem is NP-complete, I will now show a reduction from subset Sum. Set k as the max of (n,m). Select the set of smaller size between A and B, which is the people who owe money and the people who are owed. For each value in the set, select the target sum as the value, and determine if there exist a subset Sum in the other set. This is done at most k times, and a yes-instance of the problem is returned if all the subset sums are yes instance. The preprocessing runs in poly time, therefore the reduction is complete, and the problem is NP-complete.

(1) Choose smaller set out of A, B
(2) For each value out of chosen set, select subset-sum target as the value, find if there exist a subset-sum in the set not chosen.
(3) Runs at most |size of chosen set| times.

Shown that Partition $\leq_p$ subset-sum $\leq_p$ pay money, therefore this problem is NP-complete

Date No.

4. To find the approximation algorithm,
(1) sort both sets from greatest to lowest value.
(2) select the maximum number in set A, which is the people who owe money, and pay it to the people with maximum money owed in set B.
(3) Keep iterating until the remaining value in both sets are 0.

Running time: Let k be the larger size of both sets, which is $\max(n,m)$.
The running time is $k(\log k)$, for sorting.

Approximation ratio:
The optimal solution is k transactions, with $k = \max(n,m)$. this is because every people who owe money has to pay at least once, and every people who are owed should receive at least once.
The greedy algorithm sorts both sets from largest to smallest. At the first iteration, the people who pay the most is selected, and pay the people who owed the most. 3 cases are possible:
(1) He fully settled his debt, but the people who are owed still need more money before he fully claim his money. He claim from others.
(2) He paid the other people, the other people fully claimed, but the people who is paying is still left with some money. He pay others.
(3) Both sides fully settled their debts, and one has nothing left to pay, and the other has nothing left to claim. This is the optimal case.

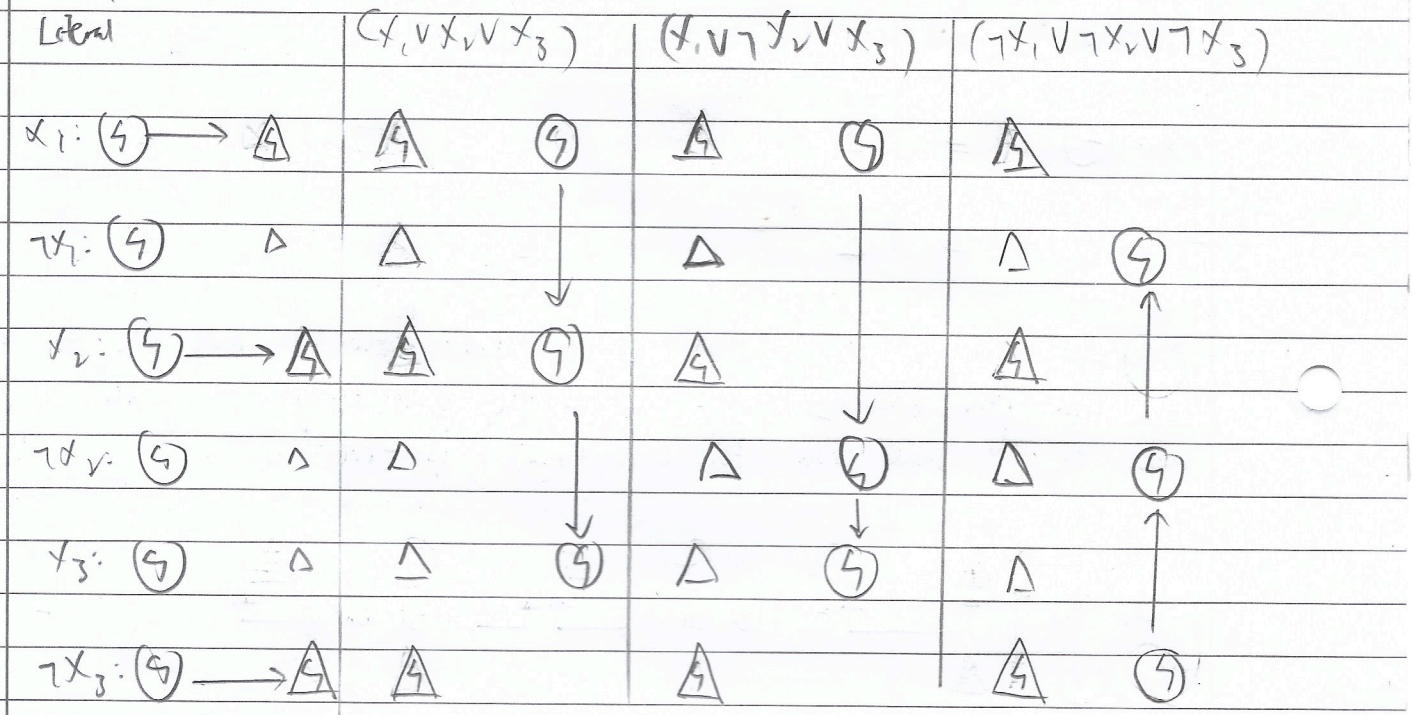
We can observe that in 3 cases, either one side or both sides are reduced to 0. the algorithm runs until all values are 0, which is at most $(n+m-1)$ times, which at the last iteration both values are reduced to 0.
Since k is the max of (n,m), 2 cases are possible:

$k=n, n>m$	$k=m, m>n$
$n+m-1$	$n+m-1$
$< n+n-1$	$< m+m-1$
$< 2n$	$< 2m$

∴ Therefore the greedy algorithm has approximation ratio of at most 2.

Date No.

(3) Now we are going to show a YES-instance in 3SAT will correspond to a YES-instance in PLUGIT. In the previous example, the 3SAT will satisfy if x_1 = True, x_2 = True x_3 = False.



Date No.

(3) Now we are going to show a YES-instance in 3SAT will correspond to a YES-instance in PLUGIT. In the previous example, the 3SAT will satisfy if x_1 = True, x_2 = True x_3 = False.