# Question (1)

## Part (a)

Part (i) Metric TSP(D): Given graph G, can we find a path with distance <= k s.t. d(A, C) <= d(A, B) + d(B, C) for all nodes A, B, C?

Part (ii) Need to show MTSP(D) is in NP and is NP-hard.

To show it is in NP, guess a path from start node x that goes through all nodes exactly once and returns to the start. Can check this occurs and that this happens in distance <= k, where the metric inequality holds. Takes p-time to sum up weights along the path => MTSP(D) is in NP.

To show it is NP-complete, we show HC <= MTSP(D) (so G has an HC iff f(G) satisfies MTSP(D))

Function f can add edges to the graph such that new edges have weight 2 and old edges have weight 1. We still have the metric inequality as the direct path will have at worst weight 2 whereas the indirect path has at least weight 2. We add weights in p-time (no more than O(n^2) to add all possible combinations and assign weights to existing edges) so f is in p-time.

=>: Assume G is in HC. Then we have a path from x going through all nodes and returning to x. Apply f to G and so we have a path that has distance <= n (as otherwise we take an edge with weight 2 that wasn’t in the graph). So f(G) is in MTSP(D).

<=: Assume f(G) is in MTSP(D). Then we have a path from start node x passing through all nodes and returning to x, so G has an HC.

Part (iii) p121 slides

Part (iv) Yes as we we used weights of 1 and 2 in reduction from HC and still had that MTSP was NP-complete.

Part (b) As above need to show in NP and NP-hard.

For showing NP, guess colouring of G and check colouring is valid (no two adj nodes have same colour) and num nodes with same colour is equal => O(n^2) time to check all edges (where we compare nodes on each side of the edge) and O(n) to count all colourings for n nodes=> takes p-time and so FAIR3COL is in NP.

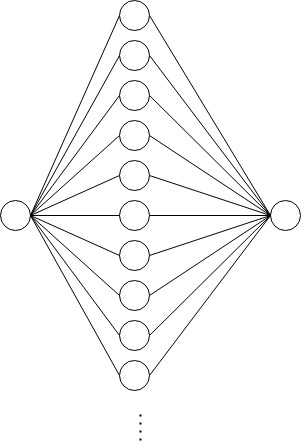
For showing NP-hard we reduce 3COL to FAIR3COL, so G is in 3COL iff f(G) is in FAIR3COL.

Let f add at nodes to G so that we have size n’ = 3 \* n (original size) which is clearly divisible by 3. Nodes are given colours that are used the least (we count in O(n)) and we either connect them to single New nodes in G with different colours to not affect the result, or don’t connect them at all. f clearly takes p-time.

=> : Assume G is 3COL. Apply f to G and now we have a 3 colouring with the same number of nodes with each colour. Then f(G) is in FAIR3COL.

<=: Clearly G is in 3COL if f(G) is in FAIR3COL as the new nodes added do not affect the original result.

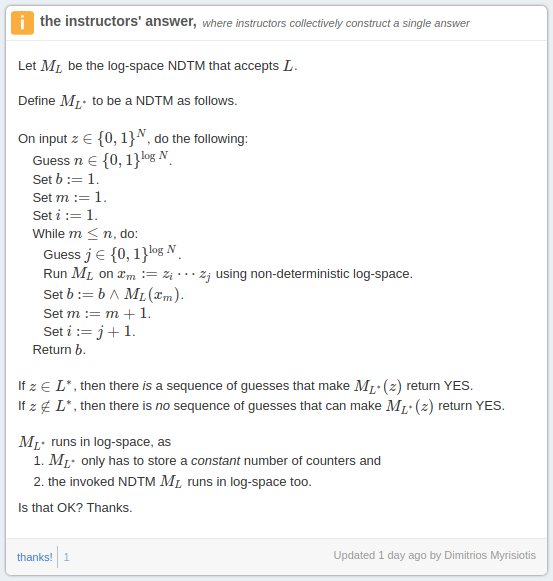
Counterexample to a 3 colouring normally using same number of each colour in the colouring (e.g. trying to colour the outer nodes first):



# Question 2

Part (a) Can iterate over all nodes where at each node we count num edges linked to other nodes (only need to track current node and degree with counters which takes logspace and we can read edges from input) => ED is in L.

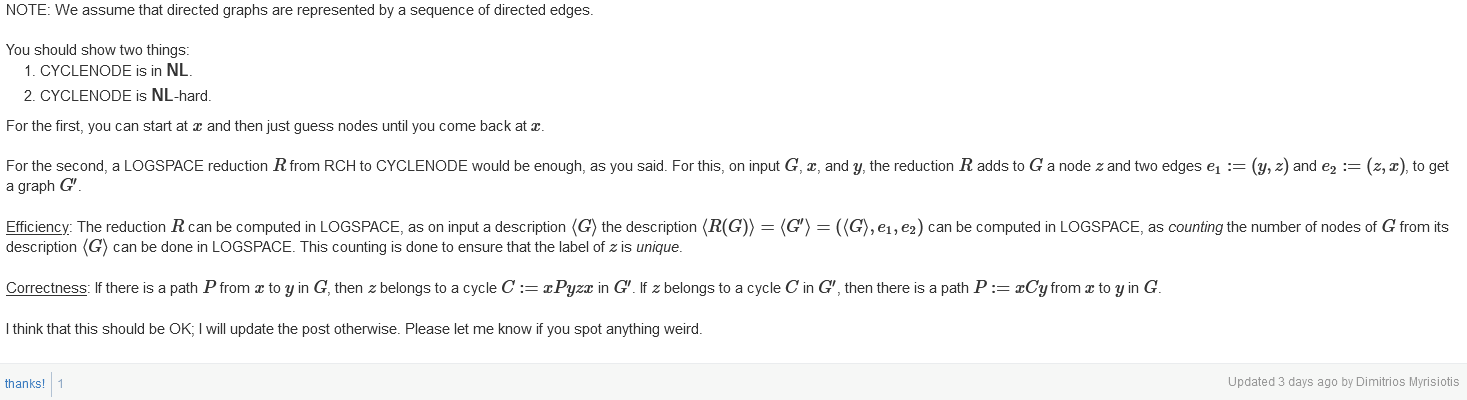
## Part (b)



Described (note a guess and check won’t work here, as the guess would need to include each xis guess):

Non-deterministically pick a split point, with the word between 0 and that point, check it is in L. Repeat: non-deterministically pick a split point (now greater than the previous), with the word between the last split point and this one, check it is in L. Repeat until point beyond concatenation word length. Since this is non-deterministically, it will be able to guess the correct splitting. And each word check is NL, so in total NL.

Part (c) Need to check in NP and is NP-hard. In NP: Guess a path and check it is a cycle and x is in the cycle. NP-hard: reduce RCH to CYCLENODE. Function f creates an edge (y,x) for every (x,y) and if we assume G is in RCH then we have a path from x to y so now we have a path from x to x in f(G, x, y) (we track points x and y). Similarly if we assume f(G, x, y) is in CYCLENODE then we have a path from x to x through some y and so we have a path from x to y.



Part (d) From CW q2b.

# Question 3

## Part (a)

Part (i) p188 slides

Part (ii) p204 slides (with I believe NC0 = PT/WK (1, n^k))

## Part (b)

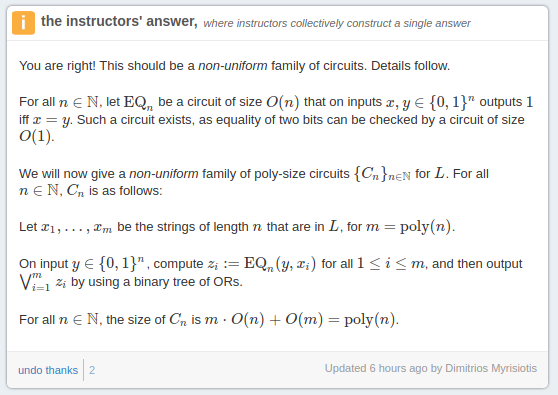
**Not too sure about this one.**

Part (i) Take arbitrary L and assume L is unary. Then we can construct a circuit such that Cn(x) = tt for x in L (can have n nodes corresponding to value 1?) This takes p-time to construct, p(n) and the size is bounded by p(n). Therefore L has polynomial circuits.

Alternative:

Part (i) This question is really badly worded but it means show there's polynomial circuits for a language L like {11, 1111, 11111} . In this case the circuits would be: C2(x) = tt, C4(x) = tt, C5(x) = tt. and otherwise Cn(x) = ff i.e. we can decide purely based on the length. More generally Cn(x) = tt if 1^n is in L, otherwise Cn(x) = ff. This is obviously uniform.

### Part (ii)



Idea: build a worst-case, brute-force poly circuit design, so that it is general enough to represent any sparse language, then show it is a (uniform) poly circuit.

For any fixed n, let's design $$C\_{n}(x)$$: we know the number of strings is less than or equal to a poly of the length $$n$$ (the input word length), so the idea is we could just test all strings.

To this end, for any given string in the language, compare, using XOR, the input characters to the series of constant 0’s and 1's representing each string in the language exactly (i.e. character-wise comparison). The output of this will be all 1s if the input matches this string exactly. Then use an AND tournament to combine the result. Repeat this for every single word in the language (this character-wise XOR, then AND tournament).

Finally OR tournament the results of all of these. The output of this final tournament will be 1 iff the word is in the sparse language.

The sparsity definition allows us to show this circuit has poly number of gates. First an XOR can be built with 5 gates, so each character requires constant gates (w.r.t. input size $$n$$). So each word (of size $$n$$) requires $$O(n)$$ gates. The outputs of these are ANDed together: the tournament will require $$O(n)$$ gates (still $$O(n)$$ total per word now). There are at most poly number of words in the language (by def. of sparsity). So total: $$O(n) \times n^{k} = O(n^{k'})$$. ORing together each word just requires $$O(n^{k})$$ gates. So we have a grand total which is still poly.

Part (c) For L to be in NC1, L must take no more than O(log n) parallel time (and O(n^k) work done).

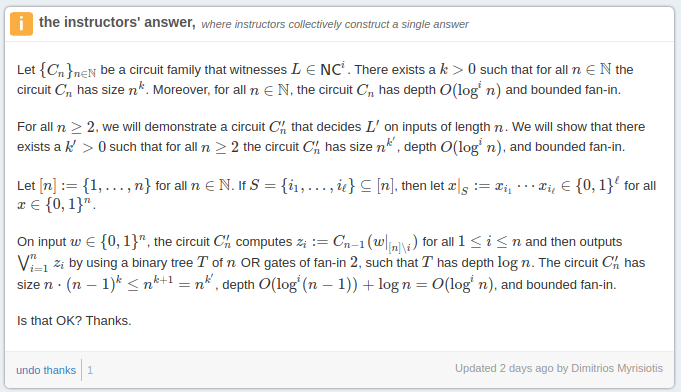
~~To check if x is in L we consider each digit xi separately and apply NOT to them, and add together (pairwise – tournament design) the values of the xi’s to see if we get an output >= 2. Takes ceil(log n) steps => O(log n) parallel time. Work done is O(n) as we have n digits => L is in NC1.~~

Alternative:

Firstly, negate all inputs. So we want to look for at least two occurrences of **1** in the output of this. Next, AND every single pair of input characters together (i.e. x\_i /\ x\_j for all i, j). This produces n(n-1) outputs. We can see that if even a single one of these ouputs is 1, there must have been a pair of 1s in the negated layer, so the input must have had at least a pair of 0s. So now we just want to reject iff any 1s in this layer.

So next we tournament (binary tree style thing) together all these previous layer outputs with OR gates. If there is even a single 1 for this (cause by at least pair of 0s in the input), the output will be 1. In total this outputs 1 iff at least a pair of 0s, 0 iff at less than a pair of 0s.

## Part (d)

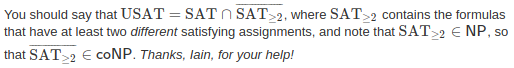


# Question 4

## Part (a)

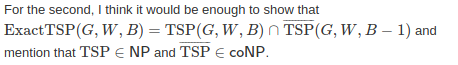
### Part (i)

From piazza:

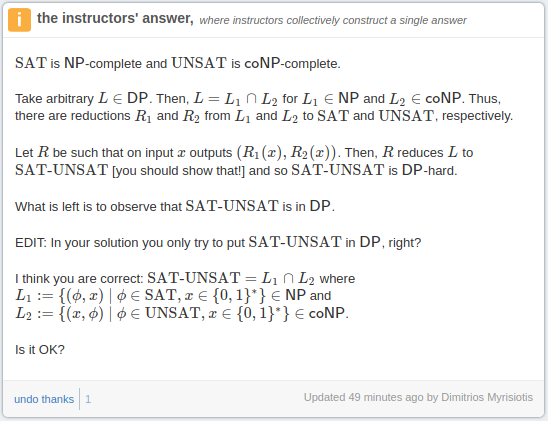


### Part (ii)

From piazza:



Part (b)



**Not sure about this one.**

Bounded below by complexity of SAT (NP-hard) which we use for SAT-UNSAT. Note SAT is in NP and in co-NP (as we invert the result) so we have that SAT-UNSAT is in DP, and as both sides are NP-hard we also have SAT-UNSAT is DP-hard and so is DP-complete.

Part (c) I think this is similar to tut 6 q10. Let our NP oracle be TSP(D), where we check increasing bounds B. We call TSP(D) with increasing B to find first B that we can use for our graph. We then apply a similar technique to the tutorial by setting individual edges to have weight B+1 and calling the oracle to see if this edge is in the path (if so then we get no from oracle and we revert weight, otherwise we leave it) until we find the edges in our path. Then (after tracking the path) we restore the weights and apply this same technique only to the weights in the path. If at any point we find when calling the oracle that we can find a new path without the edge then two paths must exist and we return no, otherwise we return yes at completion.