Pset 6

Darwin Do

April 17, 2022

(a) Darwin Do

(b) 919941748

(c) Collaborators:

(d) I have followed the academic integrity and collaboration policy

(e) Hours:

1 Minimum Spanning Tree

(a) Starting from a random vertex v in S, use DFS to `Explore` and count the number of reachable nodes from v. If the number of nodes in this component-from-v is not equal to the total number of vertices in S, we know S is not connected. Otherwise, all vertices in S are reachable from v and S is connected. This takes $O(|E| + |V|)$ time through DFS.

(b) Starting from a random vertex v in S, use DFS to `Explore` S and keep track of the nodes visited. If there is a back-edge detected where a node is a neighbor to an already-visited node that isn't the node used to get to that current node, we know S contains a cycle. Otherwise, if we manage to DFS through all of S and detect no back-edge, S should be cycle-free. This takes $O(|E| + |V|)$ time through DFS.

(c) Starting from a random vertex v in S, use DFS to `Explore` all of S. If we fail to encounter a node in S that exists in V, we know that S does not have vertex set V. Otherwise if all the nodes in V are present in S, we have vertex set V. This takes $O(|E| + |V|)$ time through DFS.

(d) First we check that S is connected, cycle-free, and has vertex set V from the steps outlined in a - c. If any of these conditions fail then S is not a spanning tree. We run Prim's algorithm on G to get $G' = (V', E')$, the MST of G. We DFS on the G' and sum all the edge weights to $w_{g'}$. We then DFS on S and sum the edge weights to w_s. If $w_s > w_{g'}$ then S is not a MST, otherwise if $w_s = w_{g'}$, S is a MST. We run Prim's once and DFS twice. The runtime of Prim is $O((|V| + |E|) \log |V|)$ which dominates the linear runtime of DFS/BFS.

The dominating runtime of all the substeps to check if S is a MST is $O((|V| + |E|) \log |V|)$. That means we can verify whether a candidate is a MST in polynomial time, so the `MST` problem is in **NP**.

2 SAT Variations

(a) Since each clause has to be full of either all `True` or all `False` terms, we can exploit this and turn all 3-term clauses into 2-term clauses. Let $c = (t_a \vee t_b \vee t_c)$ be a generic clause in this CNF. We want either the terms to be either all `True` or `False` :

$$(t_a \wedge t_b \wedge t_c) \vee (\bar{t}_a \wedge \bar{t}_b \wedge \bar{t}_c)$$

We can distribute this and simplify to get the following expression:

$$(t_a \vee \bar{t}_b) \wedge (t_a \vee \bar{t}_c) \wedge (t_b \vee \bar{t}_a) \wedge (t_b \vee \bar{t}_c) \wedge (t_c \vee \bar{t}_a) \wedge (t_c \vee \bar{t}_b)$$

We can do this process for every clause in the CNF to obtain a `2SAT` problem. We have reduced `Prob1` to `2SAT`. We know that `2SAT` is solvable in polynomial time through the technique discussed in lecture of creating a node for each variable and its negation, and adding edges between vertices that are constrained. We then check the graph to see if a variable is neighbors with its negated edge. If so, we know there is no assignment possible.

(b) Let f be a valid input to the `3SAT` problem. By construction, f already has 3 literals in each clause. We just need to ensure that no variable appears in more than 3 clauses. Let A be the set of variables that appear in more than 3 clauses. For every $x \in A$ in n different clauses, we replace every x with a separate set of variables $y_1 ... y_n$ and ensure that the $y_1 ... y_n$ variables have the same value. This is a biconditional around all of the y variables. i.e: $y_1 \implies y_2 \implies ... \implies y_n \implies y_1$. An implication of $y_1 \implies y_2$ is logically equivalent to $\bar{y}_1 \vee y_2$. That means we can append $(\bar{y}_1 \vee y_2) \wedge (\bar{y}_2 \vee y_3) \wedge ... \wedge (\bar{y}_n \vee y_1)$ to the CNF to ensure this conditional property. The ensurance of the property only uses 2 instances of each y_i, leaving the third y_i to replace the $x \in A$. Therefore no y_i will exceed three clauses.

At this point we have an instance of `Prob2` as each clause has at most 3 literals and each literal appears in at most 3 clauses.

This transformation takes polynomial time as we spend a linear amount of time in terms of literals to create set A by iterating through all the literals. Then for each $x \in A$, we spend a linear amount of creating the y variables as we know the number of y variables will always be less than the total number of literals. Together, this makes for a polynomial runtime to do this transformation.

We have reduced from `3SAT` to `Prob2` so `3SAT` $\leq_p$ `Prob2`. Since `3SAT` is NP-hard as discussed in lecture, we know that `Prob2` is NP-hard as well.

(c) Let f be a valid input to the `SAT` problem (a boolean formula in CNF). Let f' be another CNF transformed from f to be of the form $f' = f \wedge C$ where C is a clause with the following unused-in-f literals: $(x_1 \vee x_2)$.

This is a valid reduction transformation. Say that f has a satisfying assignment so `SAT` is `Yes`. That means that f' is also `Yes` for `Prob3` as we can make the following assignments to create 3 distinct satisfying assignments:

(a) The assignments from f and $(x_1 = 1, x_2 = 0)$

(b) The assignments from f and $(x_1 = 0, x_2 = 1)$

(c) The assignments from f and $(x_1 = 1, x_2 = 1)$

Say that f does not have a satisfying assignment so `SAT` is `No`. Since there is no valid assignment for f, there can be no satisfying assignment for f' either since f' is composed of f appended with an independant part.

Say that f' has at least 3 satisfying assignments. That means

$$f' = f \wedge C = \texttt{True}$$

so f has to be `True` and have a satisfying assignment as well. Likewise, if f' does not have at least 3 satisfying assignments, then f has to be `False` and not have any satisfiable assignments as otherwise, the C clause will always add 3 combinations of valid assignments.

The creation of f' is polynomial as we just need to add a new clause with 2 literals which takes constant time.

This is a valid reduction from `SAT` to `Prob3` so `SAT` $\leq_p$ `Prob3`. Since `SAT` is NP-hard as discussed in lecture, we know that `Prob3` is NP-hard as well.

3 Graph Coloring

3.1 In NP

First we show that `4-COLORING` is in **NP**. Say we have graph $G = (V, E)$ with $k = 4$ colors and a $c(v)$ color assignment. We can itertate through all the nodes in G with DFS and check if any node has edges with the same color. If a node does, we know that this assignment $c(v)$ is not a valid `4-COLORING` assignment. Otherwise if we iterate through all the nodes and see that no node has two edges of the same color, the $c(v)$ candidate is a valid `4-COLORING` assignment.

This operation just uses DFS to explore the whole graph which is linear time. Therefore `4-COLORING` is in NP as this is also in polynomial time.

3.2 NP-Hard

Now we show `4-COLORING` is NP-hard. Say we have a graph $G = (V, E)$ with a `3-COLORING` input form of $k = 3$ colors and a $c(v)$ color assignment.

We can reduce this instance to an instance of `4-COLORING` with graph $G' = (V', E')$. We form G' by adding a new node v' with color $c(v') = 4$ to V and an edge $(v, v') \in E'$ to any $v \in V$.

The only difference between G and G' is this vertex v' with color $c(v') = 4$. Therefore if G is a valid `3-COLORING`, G' is a valid `4-COLORING` as no node $v \in V$ will break the color-neighbor property through the definition of `3-COLORING`. No neighbor of v' will break this property either as no other node will have color 4. Likewise, if G is not a valid `3-COLORING` then G' is not a valid `4-COLORING` as somewhere in V the color principle is already broken.

Assume that G' is a valid `4-COLORING`. By construction, we know that all nodes $v \in V'$ follow the color-neighbor `COLORING` principle. We also know that vertex v' is the only node with color 4 in G'. Therefore we also know that G is a valid `3-COLORING` because we can remove vertex v' to get a graph with only 3 colors where all nodes follow the color-neighbor principle. Likewise, if G' is not a valid `4-COLORING`, we know there are at least 2 nodes in G' that share the same color. It can't be v' as no other node has color 4. Therefore some pair of node neighbors $(u, v) \in V$ have the same color so `3-COLORING` is `No`.

Doing this transformation from G to G' is in polynomial time as it is just adding a single node and edge.

We have shown that `3-COLORING` $\leq_p$ `4-COLORING`. Since `3-COLORING` is NP-hard, `4-COLORING` is NP-hard as well.

Therefore, `3-COLORING` is NP-complete.