Braylee Cumro

CS 397 Hw 07

1. Big-Oh

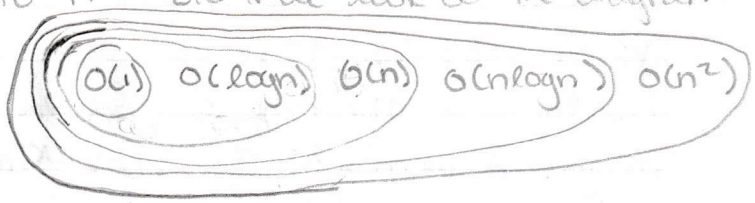
1.1 Big-Oh

Show that $n^3 + n^2 \log n + n^2 + 10$ is $O(n^3)$. Make sure to include an appropriate c & n_0 w/ justification for how they are chosen

$$\begin{aligned} f(n) &= n^3 + n^2 \log n + n^2 + 10 \\ &\leq n^3 + n^3 + n^2 + 10 && \text{for } n \geq 1 \\ &= 2n^3 + n^2 + 10 \\ &\leq 2n^3 + n^3 + 10 && \text{for } n \geq 1 \\ &= 3n^3 + 10 \\ &\leq 3n^3 + 10n^3 && \text{for } n \geq 1 \\ &= 13n^3 \end{aligned}$$

Let $c = 13$ & $n_0 = 1$

We can let $c = 13$ because we took each term of $f(n)$ and bounded it by a constant k times n^3. Since all terms can be upper bounded by a kn^3 then if we add all kn^3 terms we can get an appropriate constant c that upper bounds the whole of $f(n)$. Note we can upper bound $n^2 \log n$ to n^3 b/c if we look at the diagram made in class $O(n \log n)$ is upper bounded by $O(n^2)$ which in turn has an upper bound of $O(n^3)$.



As for no being $n \geq 1$ this number is choosen b/c zero would not hold for all the cases when we use the $\leq$ method. This is the same for negative numbers which may do weird things when we are looking for $\leq$. 1 is the 1st number that works for all the terms starting with the expression $\leq$. This holds as no becomes larger then one.

1.2 big-oh

Show that $15n\log n + n + 27$ is $O(n^2)$. Make sure to include an appropriate c & n_0 w/ justification for how they are choosen

$$
\begin{aligned}
f(n) &= 15n\log n + n + 27 \\
&\leq 15n^2 + n + 27 && \text{for } n \geq 1 \\
&= 15n^2 + n + 27 \\
&\leq 15n^2 + n^2 + 27 && \text{for } n \geq 1 \\
&= 16n^2 + 27 \\
&\leq 16n^2 + 27n^2 && \text{for } n \geq 1 \\
&= 43n^2
\end{aligned}
$$

16
27
43

For $C = 43$ and $n_0 = 1$

I got C of 43 by using the $\leq$ method and upperbording each term of the original $f(n)$ by a $O(n^2)$ counter part. For the n and 27 terms it is easy to see how n^2 and $27n^2$ are the corresponding upperbounds. As for $15n\log n$ we can treat the 15 as a coefficient to our new term and then turn the $n\log n$ into n^2 by the diagram shown in the last problem. $15 + 1 + 27$ is 43.

As for n_0 we choose 1 in the same fashion as the previous problem. At the steps that begin with $\leq$ we "test" values to determine which ones make that statement true. In this case as we moved down the steps to solve this problem these statements were all made true when $n_0 \geq 1$.

2. P

2.1 SPATH (Problem 7.21 sipser) 2et . . .

SPATH = $\{(G, s, t, k) \mid G$ has a path of at most length k from s to $t\}$

Show that SPATH $\in P$

Idea run a breadth first search to get the shortest path in graph G that goes from s to t. If that path is $<= k$ accept. Otherwise reject

Algorith:

BFS
- ① Start at node s and mark it
- ② mark all children node as unvisited
- ③ for all nodes marked unvisited in order they were marked
  - ④ mark node as visited
  - ⑤ add its children to the unvisted list maintaining order
  - ⑥ continue until you hit node t or all nodes visited
    - ⑦ return path to t bfs found

O(1) ⑧ test length of return path if length $\leq k$ accept otherwise reject

Time complexity: $O(V + E)$ where V is the number of vertices/nodes and E is the number of edges in G. (lines 1-7). line 8 is a comparison between the length of the path generated w/ bfs and k which can run in O(1) time.

SPATH $\in P$ b/c there is a polynomial time algorithm to solve it. We know from the church-turing thesis that if we can write an algorithm for a problem there is an equivalent turing machine for that algorithm.

[1, 2, 3, 4, 5, 6, 7]
x y z

2.2 Triangle

Let

Triangle = {<G> | G is a graph that contains a triangle}

where a triangle is a 3-clique

Show that Triangle ∈ P

3clique: 

Idea: find all combinations of 3 nodes in G. Test if edges exist b/t any three if so accept else reject

algorithm

V = "on input <G,K>:

O(|V³|)

① for int i = 0 to # of nodes - 2

② for int j = 1 to #nodes - 1

③ for int x = 2 to # nodes

④ node 1 = nodes[i] node 2 = nodes[j]

node 3 = nodes[x]

O(|E|)

⑤ for edge in G:

⑥ if edge (node 1, node 3) ∈ G and

edge (node 1, node 2) ∈ G and

edge (node 3, node 2) ∈ G

accept

⑦ If you make it through all combinations of 3 nodes w/out finding a triangle reject

Time complexity: To generate all combinations of three nodes (steps 1-3) the time is O(|V³|) where V is the number of nodes in G. Then for each combination we need to check the edge list in O(|E|) time where E is the number of edges in G.

overall O(|V³||E|)

① Show polytime verifier exists
or
② show polytime nondeterministic TM exists

3 NP

A cut in a graph is a seperation of the vertices into 2 disjoint subsets S & T. The size of a cut is the number of edges w/ one endpoint in S & the other in T. let

MAXCUT = {<G,k> | G has a cut of k or more}

Show MAXCUT ∈ NP.

Show a polytime verifier exists

- we need a certificate c. This certificate is a marking of the nodes in G that lets us know if the node is in subset S or subset t.

V = "on input <<G,k>, c>

O(|E|) ① for edge in G:

O(|V|) ② if (node 1 is marked S & node 2 marked T) OR (if node 1 is marked T and node 2 marked S)

③ mark that edge as cut

O(|E|) ④ loop through edges & count how many are marked cut

⑤ if #cut == k accept otherwise reject

Time complexity: Steps 1 & 4 have you loop through the number of edges in G O(|E|). line 2 would have you walk through the nodes to find the two associated w/ the edge O(|V|) where V is the # of vertices ∈ G. Overall O(|E| x |V|)

This is a polynomial time verifier thus MAXCUT ∈ NP

4 NP-complete

4.1 Dominating set

A subset of nodes of a graph G is a dominating set if every other node of G is adjacent to some node in the subset. Let

DOMSET = {<G,k> | G has a dominating set w/ k nodes}

Show that DOMSET is in NP-complete by showing its in NP & then showing its in NP-hard w/ a reduction from Vertex-cover

① In NP : C is a certificate w/ a set of nodes

V = On input <<G,k,c> :

① test whether c is a subset of G and has k number of nodes

② mark vertexes in C as being connected

③ for nodes not connected (those not in c)

④ loop through edges and find if an edge connects this node to a node in c

⑤ if no edge found, reject

⑥ If you make it through nodes not in C without rejecting, accept

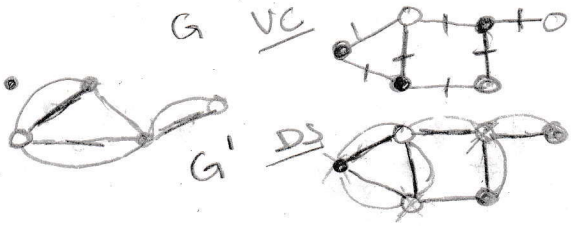
Time complexity: step 1 loop through nodes & count $O(|V|)$ where V is # of nodes $\in G$. 3rd $O(|V| \times |E|)$ loop through remaining nodes not in C and the edges looking for a connecting edge.

Total time $O(|V| \times |E|)$

Where V is # of nodes $\in G$ and E is the number of edges in G

② reduction from NP-complete

VERTEX-COVER = {<G,k> | G has a vertex cover of size k}



goal use an instance of vertex cover to solve an instance of dominating set

The original graph G is from our instance of vertex cover. G' will be this graph changed to an instance of dominating set.

Informal idea:

- find a way to change G to G' where we have a way to mark neighbor nodes in G
  - probably add edges to mark neighbor
- We would let a set S be a vertex cover in graph G of size k.
  - Since S is a vertex cover of G all edges in G are either in S or connected to a node in S
  - In G' you then need to show all the nodes in G' are either in S or adjacent to a node in S

- Some of my ideas for transformation drawn at top. Not sure how to build the proof but understand intuitively.

4.2 3 coloring

A coloring of a graph is an assignment of colors to its nodes such that no two adjacent nodes share a color

3COLOR = {<G> | G is colorable w/ 3 colors}

Show 3COLOR ∈ NP-Complete

① 3COLOR in NP

V = " on input <G, C> where C is a coloring of nodes

should check certificate has a coloring for all nodes in G

① for node x in C:

② for edge ε in G which has an endpoint of node x

③ if the other node in edge ε has the same color as x reject

④ accept if you loop through all nodes in C without rejecting

Time complexity: loop through node O(|V|) ①

loop through edges O(|E|) ②

overall O(|V| × |E|) where V is the number of nodes in G and E is the number of edges in G.

① Step 2 would be to show a reduction

Bonus Problem:

We can connect the idea of the church turing thesis to all of these topics.

Automata theory:

The church turing thesis helps us with showing that algorithms ≡ to modern computers. Any algorithm we can write has an equivalent turing machine model. This can also help us show the computability power bit models before the modern computers such as

NFA, REG, DFA, CFL, PDA, and many others.
An algorithm may be solvable on some of these simpler models, but the Church Turing Thesis at least proves it can be done on a turing machine.

Computability:

- Is the algorithm a decider. If you can write an algorithm that decides a problem there is a turing machine that can as well by the Church Turing thesis
- It would also go to show that problems that are undecidable on a turing machine do not have decidable algorithms (algorithms that always halt).
- Thus we can use these ideas to decide when an algorithm can solve a problem and when we cannot.

Complexity theory:

- Can find time either w/ turing machine model or algorithm pseudocode.
- Complexity theory helps us understand that there are cases where we could make a TM or algorithm but the runtime is unfeasable thus the answer is logically unreachable in our lifetimes.
- Just because there is an algorithm does not guarantee the answer will be given in a reasonable amount of time.