Problem Set 10

Due date: Electronic submission of this homework is due on **4/21/2023** on canvas. In order to receive any credit, you need to typeset your homework in LaTeX and submit the resulting pdf file. Any submission that cannot be checked for plagiarism will not be graded.

Name: Ayushri Jain

Resources.

Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, 3rd edition, The MIT Press, 2009 (or 4th edition)

https://www.geeksforgeeks.org/maximal-independent-set-in-an-undirected-graph/

On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment. Furthermore, I have disclosed all resources (people, books, web sites, etc.) that have been used to prepare this homework.

Signature: 

Problem 1. (20 points)

Solution. Let's consider the base case as $i = 1$. We are given that $P = NP$. So, we can say Σ_1 =NP=P. If L is the class of languages that belong to NP, i.e. $L \in NP$, then we know that Π_1=co-NP is the complement of L, i.e. $L^c \in NP$. But we have been given that NP=P, so we can compute complement of L, i.e. L^c in polynomial time. Hence, we can also say that we can compute L in polynomial time. So, $\Pi_1 = P$.

Inductive Step : Let's assume that PH=P is true for some i which means that $\Sigma_i, \Pi_i = P$. Now, we want to prove that $\Sigma_{i+1}, \Pi_{i+1} = P$ is also true. We know that a language L is in Σ_{i+1} if there exists a language L' which belongs to Π_{i+1} such that for all x in language L there exists some $y : x \in L \iff \exists y, |y| \leq |x|^c, (x, y) \in L'$ for $c > 0$. Also, $L' \in P$ due to induction hypothesis which means there exists a polynomial time turing machine to compute it. But $L \in NP$ and $NP = P$, so $\Sigma_{i+1} = P$. We can prove that $\Pi_{i+1} = P$ in similar way. Hence, we have proved that by induction if $P = NP$, then $PH = P$.

Problem 2. (30 points)

Solution. (a) Every vertex in the cycle graph C_n is adjacent to exactly two other vertices. It means that if we choose a vertex u for our independent set, then we cannot choose its two neighbors. So, maximum independent set in C_n can contain at most one vertex from every pair of adjacent vertices. When n is even, we can select one vertex from every pair of adjacent vertices, which gives us a maximum independent set of size $\frac{n}{2}$. Since $\frac{n}{2}$ is an integer, we can use the floor function to obtain this value as $\lfloor \frac{n}{2} \rfloor$. When n is odd, we will have one vertex left out and we can select one vertex from every other pair of adjacent vertices. This gives us a maximum independent set of size $\lfloor \frac{n}{2} \rfloor$. For example, if we have 5 vertices (1,2,3,4,5) and edges (1-2, 2-3, 3-4, 4-5, 5-1) then we can select 1,3 or 1,4 or 2,5 or 2,4 or 3,5 (2 vertices in all cases). If we try to add any more vertex, then the set will not remain independent, hence maximal independent set for this graph is $\lfloor \frac{5}{2} \rfloor = 2$. In conclusion, answer is $\lfloor \frac{n}{2} \rfloor$ for any n.

(b) The approximation algorithm APPROX-VERTEX-COVER given in textbook repeatedly picks a random edge and adds both its vertices to the vertex cover and removes all other edges incident on these two vertices. This algorithm always produces a vertex cover that is at most twice as large as the minimum vertex cover. As we saw above, the size of a maximum independent set in a cycle graph of size n is$\lfloor \frac{n}{2} \rfloor$ which means that the minimum vertex cover is of size $n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil$. Therefore, the approximation algorithm APPROX-VERTEX-COVER can give an optimal result for a cycle graph of size n if and only if $\lceil \frac{n}{2} \rceil \leq 2 \lfloor \frac{n}{2} \rfloor$, which is true only if n is even. If n is odd, the minimum vertex cover is of size $\lceil \frac{n}{2} \rceil + 1$ but APPROX-VERTEX-COVER will always produce a vertex cover that is of size at most $2(\lceil \frac{n}{2} \rceil + 1) = n + 2$ which is not an optimal result.

Problem 3. (30 points)

Solution. (a) To find an optimal vertex cover for a tree, we can use greedy approach. If there is only one node in the tree, we will simply return empty vertex cover. For other cases, we can start with an empty vertex cover and repeatedly find a leaf vertex, add its parent vertex to vertex cover, remove all the edges incident to the parent vertex from tree, remove leaf vertex and parent vertex from tree until there are no leaves left in the tree and finally return the vertex cover. For this to work in linear time, we can use depth-first search and maintain a list of leaf nodes.

```
Algorithm : OptimalVertexCoverGreedy(G)
#given graph G = (V,E) : V is set of vertices, E is set of edges
#empty vertex cover
vertexcover = {}
#if degree of any node in tree is 1 then it is a leaf node
find list of all leaves in G using DFS, denote by L
while V is not empty:
  if length of V is 1 or 0:
    return V

  let v be the first leaf in list L and its parent is u #edge is : {v,u}
  remove v from V
  add u to vertexcover

  for each edge {u,w} in E incident to u:
    if degree(w) is 1: #means leaf node
      append w to L
    remove {u,w} from E

  remove u from V
  remove v from L
return vertexcover
```

(b) The above algorithm works because if v is a leaf vertex, then there is an optimal vertex cover for the tree which does not contain v because it will contain its parent u and all other vertices that have edges incidental to parent u are also covered by parent u so no need to add them in vertex cover. We also add maintain list of current leaf vertices, so in the algorithm, a vertex either start as a leaf or it becomes a leaf (after removing edge). If we keep on doing this, then we always get a vertex cover which covers all edges in the tree. Clearly, the algorithm has linear run-time in terms of number of vertices assuming we have adjacency list representation of graph. We found initial list of leaf nodes using depth first traversal which takes linear time. After that, we are going through each vertex in graph and then going through edges of its parent vertex so each vertex and each edge is visited once overall. All the remove, append, condition check, add steps take constant time, hence overall runtime is linear.

Problem 4. (20 points)

Solution. It is not true that the given relationship imply that there is a polynomial time approximation algorithm for clique problem with a constant ratio. Let's assume we have a graph G with V vertices and smallest vertex cover size for G is k. If we found a vertex cover of size $2k$ then we know that we have a clique of size $V - 2k$ in its complement graph. But since smallest vertex cover size is k, then largest clique size in complement graph is $V - k$. If there is a constant ratio present, then we need to have a constant factor approximation which means that we need to always have $V - 2k \geq c(V - k)$ for some constant value of c. However, the inequality will not hold when k is close to $V/2$. Let $k = V/2 - \epsilon$ and substitute in inequality : $V - 2(V/2 - \epsilon) \geq c(V - (V/2 - \epsilon))$. This gives us $2\epsilon \geq c(V/2 + \epsilon)$. Since we assumed that k is close to $V/2$, ϵ value can be made very small even when we increase the number of vertices in graph. So, the inequality will not hold and hence there is constant approximation ratio polynomial time algorithm for clique problem.