**CS120: Intro. to Algorithms and their Limitations** Prof. Salil Vadhan

# Problem Set 5

*Harvard SEAS - Fall 2021* *Due: Wed Oct. 19, 2022 (11:59pm)*

**Your name: Nicole Chen**
**Collaborators: Jayden Personatt, Prin Pulkes**
**No. of late days used on previous psets: 3**
**No. of late days used after including this pset: 3**

1. (Exponential-Time Coloring) In the Github repository for PS5, we have given you basic data structures for graphs (in adjacency list representation) and colorings, an implementation of the Exhaustive-Search $k$-Coloring algorithm, and a variety of test cases (graphs) for coloring algorithms.

   (a) Implement the $O(n+m)$-time algorithm for 2-coloring that we covered in class in the function `bfs_2_coloring`, verifying its correctness by running `python3 -m ps5_tests 2`.

   (b) Implement `is_independent_set`, which checks if a given subset of nodes is an independent set.

   (c) Implement the $O(1.89^n)$-time algorithm for 3-coloring (ISET + BFS) that you studied in the SRE in the function `iset_bfs_3_coloring`, also verifying its correctness by running `python3 -m ps5_tests 3`.

   (d) Compare the efficiency of Exhaustive-Search 3-coloring and the $O(1.89^n)$-time algorithm. Specifically, identify the largest instance each algorithm is able to solve (within a time limit you specify, e.g. 10 seconds) and the smallest instance each algorithm is unable to solve (again within that same time limit).

   In addition to these numeric values, please provide a brief explanation of why these results make sense, based on your knowledge of how each algorithm finds a valid coloring. For this part, there is no need to go through every combination of parameters; feel free to give just the largest and smallest instances each algorithm can solve and speak generally as to why one algorithm performs better than the other. More instructions can be found in `ps5_experiments`.

   (Solution) For all values below, the numbers were determined through the parameters:

   ```
   subgraph_line_parameter_range = (100, 300, 100)
   cluster_graph_p_parameter_range = (0.2, 0.95, 0.15)
   cluster_graph_cluster_size_parameter_range = (2, 26, 8)
   cluster_graph_cluster_quantity_parameter_range = (2, 5, 1)
   ```

i. For the Line of Rings graphs, I identified that the largest instance ISET BFS Coloring is able solve within a time of 20 seconds was a size of ring 4 with $n = 800$ and $m = 999$. For the Line of Rings graphs, I identified that the smallest instance ISET BFS Coloring is unable to solve within a time of 20 seconds was a size of ring 3 with $n = 300$ and $m = 399$.

ii. For the Line of Rings graphs, there were no graphs for which Exhaustive Search Coloring was able to solve within the time of 20 seconds. The smallest instance Exhaustive Search was unable to solve within a time of 20 seconds was a size of ring 3 with $n = 300$ and $m = 399$.

iii. For Randomized Cluster Connections, I identified that the largest ISET BFS Coloring was able to solve within a time of 20 seconds was an 18-size of cluster (with 2 clusters) where $n = 36$ and $m = 268$ at probability 0.8. The smallest instance ISET BFS Coloring was unable to solve within a time of 20 seconds was at $n = 40$ and $m = 125$ for a 10-size cluster (with 4 clusters) at probability 0.2.

iv. For Randomized Cluster Connections, I identified that the largest Exhaustive Search Coloring is able to solve within a time of 20 seconds was 10-size cluster (with 2 clusters) where $n = 20$ and $m = 79$ at probability 0.8. The smallest instance Exhaustive Search Coloring is unable to solve within a time of 20 seconds was 10-size cluster (with 3 clusters) where $n = 30$ and $m = 60$ at probability 0.2.

The above results show that ISET BFS is far more efficient than Exhaustive Search Coloring. In fact, for the Line of Rings graph, there were no graphs for which Exhaustive Search Coloring was able to solve that ISET BFS could not also solve. In fact, Exhaustive Search Coloring could not solve any graphs while ISET BFS Coloring was able to solve all graphs of size of ring 4. Similarly, for Randomized Cluster Connections, every graph solvable by Exhaustive Search was also solvable by ISET BFS Coloring. This is shown by the fact that the largest instance ISET BFS Coloring could solve is significantly larger than that of Exhaustive Search Coloring and the fact that the smallest instance Exhaustive Search Coloring was unable to solve is significantly smaller than the smallest that ISET BFS Coloring couldn't solve. The fact that ISET BFS runs faster than Exhaustive Search aligns with the fact that the efficiency for Exhaustive Search 3-coloring is an $O(3^n)$ algorithm while ISET BFS Coloring is an $O(1.89^n)$ algorithm. More specifically, ISET BFS Coloring runs Exhaustive Search on a smaller subset such that the algorithm only cares about subsets of size at most $\frac{n}{3}$ while Exhaustive Search cares about all subsets of size at most $n$. This is because ISET-BFS has an independent set checker and a 2-coloring algorithm that it calls for its 3-coloring. Therefore, ISET BFS runs through $\begin{pmatrix} n \\ \frac{n}{3} \end{pmatrix}$ combinations of color-combos for the graph while Exhaustive Search runs through $3^n$ combinations for the graph. $\begin{pmatrix} n \\ \frac{n}{3} \end{pmatrix}$ reduces down to 1.89; hence, ISET BFS runs $O(1.89^n)$.

2. (Reductions Between Variants of IndependentSet) Consider the following three variants of the IndependentSet problem:

- IndependentSet-OptimizationSearch: given a graph $G$, find the largest independent set in $G$.

- IndependentSet-ThresholdSearch: given a graph $G$ and a number $k \in \mathbb{N}$, find an independent set of size at least $k$ in $G$ (if one exists).
- IndependentSet-ThresholdDecision: given a graph $G$ and a number $k \in \mathbb{N}$, decide (by outputting YES or NO) whether or not there is an independent set of size at least $k$ in $G$.

For each part below, be sure to both prove correctness and analyze runtime for the algorithms you provide.

(a) Suppose that there is an algorithm running in time $T(n, m) \geq n + m$ that solves IndependentSet-OptimizationSearch on graphs with at most $n$ vertices and at most $m$ edges. Prove that there is an algorithm running in time $O(T(n, m))$ that solves IndependentSet-ThresholdDecision.

(Solution) We first show our algorithm.

i. Find the largest independent set in $G$ through a reduction from IndependentSet-ThresholdDecision to IndependentSet-OptimizationSearch. Store the largest independent set in $G$ as $G_n$
ii. Given $G_n$, iterate through its set of vertices to determine the size of the vertex set and store in variable $size_n$. Essentially, we are solving for the size of $G_n$.
iii. Compare $size_n$ to $k$. If $size_n \geq k$ then IndependentSetThresholdDecision outputs "YES." If not, then IndependentSetThresholdDecision outputs "NO."

We now prove correctness.

i. If the inputted graph $G$ has an independent set of at least $k$ then the largest independent set of $G$ must have size greater than or equal to $k$. Because our algorithm's call to IndependentSet-OptimizationSearch correctly finds the largest independent set of $G$, our algorithm can then correctly determine the size of the largest independent set in $G$ and compare whether or not that set is greater than or equal to $k$.
ii. If $G$'s largest independent set's size is not greater than $k$, then no other independent set in $G$ is greater than $k$ and therefore our algorithm successfully returns "NO".
iii. However, if $G$'s largest independent set's size is greater than or equal to $k$, then we know there is at least one independent set that is at least size $k$, and therefore our algorithm successfully returns "YES."

We lastly prove run time.

i. A call to IndependentSet-OptimizationSearch takes $O(T(n, m))$. Storing the maximum set $G_n$ that IndependentSet-OptimizationSearch returns takes $O(1)$
ii. Iterating through $G_n$ to find its size $size_n$ takes $O(n)$ time (if all $n$ vertices in $G$ are independent)
iii. Checking $size_n \geq k$ takes $O(1)$ time
iv. We therefore have a $O(T(n, m))$ algorithm for IndependentSet-ThresholdDecision that makes one "oracle call" to IndependentSet-OptimizationSearch on graph $G$ and we conclude:

$$Time_{IndependentSet-ThresholdDecision}(n,m) \leq O(n) + O(1) + T(n,m) = O(T(n,m))$$

(b) Suppose that there is an algorithm running in time $T(n,m) \geq n+m$ that solves IndependentSet-ThresholdSearch on graphs with at most $n$ vertices and at most $m$ edges. Prove that there is an algorithm running in time $O((\log n) \cdot T(n,m))$ that solves IndependentSet-OptimizationSearch. (Hint: Come up with a reduction that makes at most $\log n$ oracle calls. You might find it useful to first find one that makes at most $n$ oracle calls.)

(Solution) We first show our algorithm.

i. Find the largest $k$ in graph $G$ such that graph $G$ has an independent set of size at least $k$. We do so through a reduction from IndependentSet-OptimizationSearch to IndependentSet-ThresholdSearch.

ii. More specifically, we set $k = \frac{n}{2}$ and in a binary search manner, recursively call IndependentSet-ThresholdSearch, adjusting our $k$ depending on whether the prior call to IndependentSet-ThresholdSearch returns an independent set of size at least $k$ or not. We do so by a binary search algorithm that recursively segments our $k$ by halves based on the prior IndependentSet-ThresholdSearch call. Thus, if we start off at $\frac{n}{2}$, we will try to "divide" or "segment" our next $k$'s by half of the range that we started with in the direction that our prior call suggests. For example, if $k = \frac{n}{2}$ and our call to IndependentSet-ThresholdSearch produces no independent set, then we set our $k$ to $\frac{n}{4}$. Similarily, if at $k = \frac{n}{2}$ and our call to IndependentSet-ThresholdSearch produces an independent set, we reset our $k$ to $\frac{3n}{4}$. This continues until we have found a "cutoff" point of our $k$ such that that at $k$, IndependentSet-ThresholdSearch produces an independent set of at least $k$, but at $k+1$, IndependentSet-ThresholdSearch produces no independent set.

iii. Let's call the independent set that is produced at the "cutoff" point $G_n$. Our largest independent set is therefore $G_n$ and we return it.

iv. More concretely, we can think of having three pointers $l$ for lower bound, $u$ for upper bound, and $c$ for current index. We initially set $l = 0$, $u = n+1$, and $c = \frac{u-l}{2}$. Consider each pointer to be pointing to possible values of $k$. Given $c$, the algorithm makes an oracle call to IndependentSet-ThresholdSearch. If the call returns an independent set, we adjust $l = c$ and reassign $c$ accordingly to the value considered to be the median between $u$ and $l$ (we can do this through setting $c = l + \frac{u-l}{2}$ If the call does not return an independent set, we adjust $u = c$ and reassign $c$ accordingly: $c = l + \frac{u-l}{2}$. Repeat this until $u + 1 = l$. We then return the independent set.

We now prove correctness.

i. If graph $G$ has an independent set with size $s$ such that all other independent sets in $G$ have size less or equal to $s$, then we can say that graph $G$ has an independent set of at least $k = s$. However, we cannot say that graph $G$ has an independent set of at least $k = s+1$ as our largest independent set has size $s$.

ii. We perform a binary search on possible values of $k \in 0, ...n-1$ where $n$ is the size of the graph $G$ to find $k = s$. Our binary search is based on comparisons to calls of IndependentSet-ThresholdSearch. Since we compare whether at the call's select

$k$ there is an independent set of at least size $k$. We want to maximize $k$ and thus perform calls to IndependentSet-ThresholdSearch until $k = s$. Unless there are no independent sets in $G$, we will always get $k = s$ by the correctness of the binary search algorithm.

iii. More specifically, at any iteration, given a lower bound $l$ and upper bound $u$ and current pointer $c$, if the oracle call to IndependentSet-ThresholdSearch for $k = c$ returns $\perp$, there exists no independent set in $G$ of at least size $k = c$. As a result, we adjust our lower and upper bound so that the upper bound becomes $c$ and we can gradually filter accordingly. This is because our prior oracle call shows us that there is no independent set in $G$ of at least size $k = c$, so we can disregard values of $k > c$ (hence, upper bound is set to $c$. Similarly, if the oracle call to IndependentSet-ThresholdSearch for $k = c$ does not return $\perp$, we know there is a valid independent set in $G$ of at least size $k = c$ and set our lower bound to $c$ as we know that our "bottom threshold" is $c$. Because our search continues until $u = l + 1$, we know that $k = l = s$ has the largest $k$ value for $G$ (we don't use $u$ because its value may be outside the number of total vertices of $G$). Thus, we have found our largest independent set's size.

iv. Our algorithm discovers $k = s$, and we thus return the independent set in $G$ with size of at least $k = s$ through our call to IndependentSet-ThresholdSearch. Since $s$ denotes the size of the largest independent set in $G$, we have appropriate returned the largest independent set.

We lastly prove run time.

i. To find $k = s$, we perform $O(\log(n))$ calls to IndependentSet-ThresholdSearch by the basis of binary search, which takes $O(\log(n))$. More specifically, iterating through all $n$ vertices to find $n$ takes $O(n)$ time. Setting $l$, $u$, and $c$ initially take constant time. At each iteration, we are gradually dividing the list of possible $k$ values by half. Therefore, as the range of sizes is repeatedly halved until $u = l + 1$, the loop will run $O(\log(n))$ time. Each re-assignment of $u$, $c$, and $l$ takes $O(1)$ time.

ii. We store $G_s$, which is the returned independent set of the last call to IndependentSet-ThresholdSearch, in time $O(1)$. $G_s$ therefore represents the independent set of $G$ with size at least $k = s$.

iii. Returning $G_s$ takes $O(1)$ time.

iv. We therefore have a $O(\log(n) \cdot T(n, m))$ algorithm for IndependentSet-OptimizationSearch that makes $O(\log(n))$ "oracle calls" to IndependentSet-ThresholdSearch on graph $G$ and we conclude:

$$Time_{IndependentSet-OptimizationSearch}(n, m) \leq O(n) + O(\log(n) \cdot T(n, m)) + O(1) = O(\log(n) \cdot T(n, m))$$

(c) Suppose that there is an algorithm running in time $T(n, m) \geq n + m$ that solves IndependentSet-ThresholdDecision. Prove that there is an algorithm running in time $O(n \cdot T(n, m))$ that solves IndependentSet-ThresholdSearch. (Hint: Show that $G$ has an independent set of size at least $k$ containing vertex $v$ iff $G - N(v)$ has an independent set of size at least $k - 1$, where $G - N(v)$ denotes the graph obtained by removing $v$ and all of its neighbors from $G$. Use this fact and the oracle to determine for each vertex

$v$, whether or not to include $v$ in the independent set. Be sure to update the graph appropriately after each decision.)

(Solution) We first show our algorithm

i. Find an independent set of size at least $k$ in $G$ (if one exists) through a reduction from IndependentSet-ThresholdSearch to IndependentSet-ThresholdDecision.

ii. More specifically, our algorithm makes an initial call to IndependentSet-ThresholdDecision. If the call returns "YES" we proceed with our algorithm. Otherwise, return $\perp$.

iii. We have now established that our graph $G$ has an independent set of size at least $k$. Our algorithm then initializes an empty independent set called $G_e$ and an empty set *visited* that represents all the vertices we have visited.

iv. Call IndependentSet-ThresholdDecision for each vertex $v$ that is not in the *visited*. Each call to IndependentSet-ThresholdDecision should be on graph $G-N(v)$, which represents the graph obtained by removing $v$ and all of its neighbors from $G$. We can remove obtain $G - N(v)$ by removing $v$ from $G$, iterating through $v$'s edge list to remove its neighbors, and also iterating through the edge lists of all the vertices in $G$ and deleting occurences of $v$.

v. For each $v$ in $G$, if the call to IndependentSet-ThresholdDecision on $G - N(v)$ with $k - 1$ returns "YES," we add that vertex $v$ to $G_e$. We then set our graph $G$ to $G - N(v)$ plus the vertex $v$ without its neighbors and its original edges. This is a key step: since $v$ is in our independent set $G_e$, we know that none of $v$'s neighbors are in $G_e$ and can therefore be removed. If the call to IndependentSet-ThresholdDecision on $G-N(v)$ with $k-1$ returns "NO," then we move on to the next node. For both possibilities, we update $v$ to our *visited* set.

vi. Once we have iterated through all nodes in $G$ (this is the updated $G$), we return our set of independent nodes $G_e$ which should have size $k$.

We now prove correctness

i. If there is an independent set of size $k$ in our graph $G$, our call to IndependentSet-ThresholdDecision will output "YES." Otherwise, IndependentSet-ThresholdDecision will output "NO," and our algorithm will appropriately return $\perp$ as there is no independent set of size $k$ in our graph $G$.

ii. We now move to prove the correctness of the biconditional that states "$G$ has an independent set of size at least $k$ containing vertex $v$ iff $G-N(v)$ has an independent set of size at least $k-1$, where $G-N(v)$ denotes the graph obtained by removing $v$ an all of its neighbors from $G$." First, if $G$ has an independent set of size at least $k$ containing vertex $v$ then removing that vertex and its subsequent neighbors will result in the independent set shrinking only by a size of 1 because by definition, an independent set contains vertices that are not neighbors of $v$. Thus, if we remove $v$ and its neighbors, the independent set will only shrink by one because $v$'s neighbors have no impact on the independent set. Therefore, $G - N(v)$ has an independent set of size at least $k - 1$. We now prove the other direction: if $G - N(v)$ has an independent set of size at least $k-1$ then $G$ has an independent set of size at least $k$ containing vertex $v$. This is because in $G-N(v)$, there is currently no vertex that is a neighbor of $v$, as we removed $v$ and all of its neighbors in $G-N(v)$. Since $G-N(v)$

contains no vertices that are neighbors of $v$, the independent set of $G - N(v)$ will also not contain neibhros of $v$. Therefore, $v$ is independent from any of the vertices in the independent set, and so by adding $v$, the independent set's size increases from $k-1$ to $k$. Therefore, by adding $v$ and its neighbors to $G - N(v)$, we can add 1 more vertex to the independent set. Therefore, we can say that $G$, which includes $v$ and its neighbors has an independent set of size at least $k$.

iii. Now that we have established the biconditional, we look more broadly at the algorithm. After establishing that $G$ has an independent set of size at least $k$ (established in (i)), we want to find the vertices that compose of this independent set. As a result, we iterate through each vertex $v$ in $G$, calling IndependentSet-ThresholdDecision on $G - N(v)$ for $k-1$. If the $v$ that we remove is part of the independent set, then by (ii), the size of the independent set will decrease to $k-1$. Therefore, if our call to IndependentSet-ThresholdDecision on $G - N(v)$ for $k-1$ produces a "YES," we know that our vertex $v$ is part of the independent set and can therefore be added to $G_e$. Once we know that $v$ is part of the independent set, we know that any neighbors of $v$ are not. Therefore, we reset $G$ to be equal to $G - N(v)$ plus our vertex $v$ without its neighbors. On the other hand, if the $v$ that we remove is not part of the independent set, then our call to IndependentSet-ThresholdDecision on $G - N(v)$ for $k-1$ produces a "NO." This is because $N(v)$ removes not only $v$ but also its neighbors. Since $v$ is not part of the independent set, we know some of its neighbors are part of the independent set, and therefore, removing $v$ and its neighbors will cause the size of the independent set to reduce below $k-1$. Thus, when IndependentSet-ThresholdDecision on $G - N(v)$ for $k-1$ produces a "NO," we do not add the vertex $v$ to $G_e$ because we know it should not be part of the independent set.

iv. We have therefore proven that after iterating through all $n$ vertices in $G$, the graph returns an independent set of size at least $k$ if there is one. If not, our algorithm returns $\perp$ appropriately.

We lastly prove run time

i. Our initial oracle call to IndependentSet-ThresholdDecision to determine whether $G$ has an independent set of size at least $k$ takes $O(T(n,m))$ time. If there is no independent set, we return $\perp$ in constant time.

ii. Once we have determined that there is an independent set of size at least $k$, we initialize $G_e$ and *visited* as empty sets, which take $O(1)$ time.

iii. Our algorithm then checks for every node in $G$ whether removing the node and its neighbors produces a graph $G - N(v)$ with an independent size of at least $k-1$. In the worst-case scenario, it can take $O(n+m)$ to construct $G-N(v)$ because we would need to iterate through all $n$ vertices and $m$ edges to find deletions through BFS. We then determine whether $v$ is part of the independent set or not through an oracle call to IndependentSet-ThresholdDecision which takes time $O(T(n,m))$. Within every call to IndependentSet-ThresholdDecision, we check to see if a "YES" or "NO" is returned. This takes constant time. If a "YES" is returned, we add $v$ to $G_e$ in constant time and set $G$ equal to $G - N(v)$ plus $v$ without its neighbors. This also takes constant time. As we are making $n$ calls to IndependentSet-ThresholdDecision, these steps takes $O(n) \cdot O(T(n,m) + O(n+m) + O(1)) = O(n \cdot T(n,m))$.

iv. We therefore have a $O(n \cdot T(n,m))$ algorithm for IndependentSet-ThresholdSearch that makes $n$ "oracle calls" to IndependentSet-ThresholdDecision, and we conclude:

$$Time_{IndependentSet-ThresholdSearch}(n,m) \leq O(n \cdot T(n,m)) + O(1) = O(n \cdot T(n,m))$$

We remark (but you don't need to submit anything) that the combination of the three previous problem parts means that for every constant $c \in [1,2]$, if there is an algorithm solving any one of the three problems in time $(n+m)^{O(1)} \cdot c^n$, there are algorithms solving the other two problems in $(n+m)^{O(1)} \cdot c^n$ time. The best known algorithm (by Xiao and Nagamochi, 2013) has $c \approx 1.1996$.