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:
Collaborators:
No. of late days used on previous psets:
No. of late days used after including this pset:

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`.

   Proof. We use the time limit of 4 seconds to get closer to the values used in parts a, b, c. There are also lots of probabilities given, so I choose three to examine.
   For the line of rings, the rings of size 3 fail with exhaustive coloring at 5 rings ($n+m=34$), and fail with ISET-BFS succeed with 7 but fail with 9 rings ($n+m=62$). For rings of size 4, the exhaustive coloring fails with 5 rings ($n+m=44$) and ISET-BFS succeeds with up to 4000 rings ($n+m=35999$), after which I stopped checking. For rings of size 5, the exhaustive coloring fails with rings of size 5 ($n+m=54$) and succeeds for rings of size 5 in ISET-BFS but fails with rings of size 7 ($n+m=77$).
   The reason this may be is that coloring with ISET-BFS is super easy for rings of size 4 - literally choose one node to be colored with color 2, and the rest can be 2-colored with

colors 0, 1 (by the structure of the shape). However, exhaustive coloring takes much longer to do because we manually check every single arrangement without the privilege of getting 2-coloring through BFS, which can be done super quickly. The cases of 3 and 5 are tougher because it's difficult to 2-color them, and you have to get to pretty large values of smallest set k (in fact, I believe it may just be $n/3$) for this to work.

For randomized cluster connections, we first examine the probability of keeping an edge between the sets at 0.2. Then when we have 2 clusters, the exhaustive coloring succeeds for clusters of size 10 ($n + m = 48$) but fails for clusters of size ($n + m = 96$), where ISET-BFS succeeds. For 3 clusters, if the clusters have size 10 then ISET-BFS succeeds but exhaustive coloring fails ($n + m = 83$), while ISET-BFS fails at clusters of size 18 ($n + m = 245$). For 4 clusters, we succeed for $n + m = 10$ for both ISET-BFS and exhaustive coloring, but only ISET-BFS succeeds at $n + m = 67$.

When the probability is 0.35, we succeed for ISET-BFS but fail for exhaustive coloring at $n + m = 142$ with 2 clusters. With 3 clusters, we succeed at $n + m = 11$ but fail at $n + m = 139$ for ISET-BFS and exhaustive colorings. With 4 clusters, we succeed at $n + m = 13$ for ISET-BFS and exhaustive colorings, but only ISET-BFS succeeds at $n + m = 96$.

When the probability is 0.8, we get similar results. With 2 clusters, $n + m = 294$ fails for exhaustive but succeeds for ISET-BFS. With 3 clusters, we succeed at $n + m = 12$ but fail at $n + m = 273$ for both exhaustive and ISET-BFS colorings. With 4 clusters, we we succeed at $n+m = 26$ but fail at $n+m = 516$ for both exhaustive and ISET-BFS colorings.

The results here are not *that* illuminating - we find that ISET-BFS outperforms exhaustive coloring generally, but large cases are not wonderful for either algorithm. This makes sense - there is no pattern to the arrangement, so we generally expect ISET-BFS to outperform exhaustive search pretty well, but we'd need more granular tests than what was given to get a strong idea of how much better it is. But this is the result we expect with pretty random graphs - exponential time is not a great runtime, even if the base of the exponent can be reduced (as we do with ISET-BFS). □

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.

Proof. OptimizationSearch finds the largest independent set in G. So to solve the ThresholdSearch problem with a threshold k, we can call the OptimizationSearch algorithm, which returns the set S in time $O(T(n,m))$. Then, count the length of set S, which just requires traversing an array, which can be done in time $O(n)$ (because there are fewer than n elements of the list). Finally, we compare the size of the list with k, returning S if $\text{len}(S) \geq k$ and $\perp$ otherwise, a step which takes $O(1)$ time (because all of this is basic computation. This solves the ThresholdSearch problem because we find a set which works if one exists, and if $\text{len}(S) < k$, we know there is no set which is large enough to work. Because $T(n,m) \geq n + m$, it dominates the $O(n)$ step, so the total runtime here is $O(T(n,m))$. □

(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.)

Proof. The ThresholdSearch algorithm on k returns an independent set of size k if one exists, and tells us if not. Then we can find the largest k for which an independent set exists by binary searching the range of 0 to the number of nodes n. Each step of the way, we binary search the remaining upper half if we find that there is an independent set of size at least k, and the remaining bottom half otherwise. This ensures that every time we get a set that works, it's the largest k that works so far. Then the algorithm terminates when we find some k for which Threshold Search returns a set on k, but not on $k+1$. This is unless $k = n$, in which case we return k.

We are guaranteed that a set will be returned, because for any nonempty collection G, any singleton set is independent. Clearly this set also works by definition. Then we also know that binary searching takes $\log n$ time, and the process of updating the next k takes time $O(1)$. Then because $T(n,m) \geq n + m > n > 1$, we find that we call the ThresholdSearch oracle $O(\log n)$, which means our algorithm happens in time $O((\log n)(T(n,m)))$. □

(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.)

Proof. We take the suggestion of the hint and prove the lemma listed there.

($\Longrightarrow$) Suppose G has an independent set of size at least k which contains v. Then v shares no edges with any of the other elements $v' \in G$. That means that we can remove v and all of its neighbors, none of which are $v' \neq v \in G$. Therefore there are still at least $k-1$ nodes in G without v and its neighbors, or $G - N(v)$, as we wanted to show.

($\Longleftarrow$) Suppose the graph $G - N(v)$ of G without some vertex v and all of its edges has an independent set of size at least $k-1$. Then if we add some node v in and a bunch of neighbors, and we know that v has no edges with nodes in $G - N(v)$, then we are guaranteed that we can add v to the independent set, which means that we have a graph of size at least k in G, which is what we wanted.

So now we have this neat lemma. Then we run the following algorithm: we iterate over v in the range from 0 to n run ThresholdDecision on the set $G - N(v)$, for some $v \in G$ (this is n nodes). If it returns YES, then by the lemma, we know that there exists some independent set of size at least k which contains v. Then consider the set $G - N(v)$ and increment v', checking to see if removing v' with value $k-1$ would return YES on ThresholdDecision. If it does, we update again and continue. Any time we get the response NO, we move on to the next node and do not remove that value of v' and its neighbors from the graph. At the end, the set of removed vertices forms an independent set of size at least k.

This algorithm is correct because at every step of the way, we know that an independent set of size at least k is achievable. If ThresholdSearch returns NO, then we know that the element we are examining cannot be in the set we are constructing. We are also assured that the element we are examining, should ThresholdSearch return YES, is independent from everything that has come so far, and that there is some subset of the remaining elements which completes an independent set of size at least k. So therefore we construct a working set in this case.

In our algorithm, we call the oracle ThresholdSearch at most n times, because we examine each node at most once. Nothing else we do in this process is intensive (the process of removing a node and edges takes time $O(1+m)$, which is dominated by $O(T(n,m)) \geq n+m$). Therefore the total time this takes can be described by calling the time $T(n,m)$ oracle n times, or $O(n(T(n,m)))$ as desired. □

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 $c \approx 1.1996$.