CS120: Intro. to Algorithms and their Limitations Hesterberg & Anshu

Problem Set 5

Harvard SEAS - Fall 2023 *Due: Wed Oct. 25, 2023 (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`.

   Done

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

   Done

   (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`.

   Done

   (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`.

   Timing:
   I ran the `ps5_experiments` and observed that the independent set + BFS algorithm was never beaten by the exhaustive algorithm. However, there were also a ton of trials, and many of them featured both algorithms failing to pass the timeout. Exhaustive failed every ring trial, the independent set algorithm pretty handily outperformed on the cluster tests (with many ties as well). Since

this was a bit difficult to analyze, I also wrote my own testing function to get a simpler but more consistent set of measurements (code at the bottom).

Running the tests on five separate random graphs with edge probability of 7.5 percent, I found that the exhaustive algorithm was typically capable of handling a graph of about 20 nodes in 10 seconds. Under the same constraints, the Independent Set / BFS algorithm could manage a graph of about 40 nodes in 10 seconds. Note, however, the randomness of the input edges also played a huge role. Often, of the five trials, one would take more time than the other four combined. Considering lower bounds, for very unfortunate graphs, the exhaustive solution was once unable to solve a graph of 12 nodes in 10 seconds, and the BFS algorithm had a similar worst case run through 30 nodes over 10 seconds. That seems to be the worst-case complexity showing itself.

The immense complexity of a worst case solution for the exhaustive algorithm is understandable. The code enumerates every possible color combination of nodes in the graph and checks each independent set. Often enough, it will get lucky and find an independent set early on. However, if every combination must be generated and evaluated, this algorithm spirals out of control quickly. For example, if the algorithm tests 20 nodes in set combinations of cardinality zero through 20, I believe the summation of all those combinations makes for $1,048,576$ possible combinations to check, again just for 20 nodes.

The BFS solution is meaningfully better. The most pronounced factor here is that we can check sets of cardinality zero through at most $n/3$, clipping much of the tail off the worst case runtime.

Testing function:

```python
if __name__ == "__main__":
    num_nodes = 40
    prob_edge = 0.075

    total_exhaustive = 0
    total_iset = 0

    num_trials = 5

    for ct in range(num_trials):
        print(f"trial {ct}")
        G = generate_random_graph(Graph, num_nodes, prob_edge)

        start_exhaustive = time()
        exhaustive_search_coloring(G)
        diff = time() - start_exhaustive
```

```
            total_exhaustive += diff

            start_iset = time()
            iset_bfs_3_coloring(G)
            diff = time() - start_iset
            total_iset += diff

        print(f"exhaustive: {total_exhaustive / num_trials}")
        print(f"iset: {total_iset / num_trials}")
```

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.

   To solve IndependentSet-ThresholdDecision, an algorithm must accept a graph G and a natural number K and decide whether or not an independent set of size greater than or equal to k exists in the graph. Consider the algorithm below:

```
from typing import Set, Tuple, Any

Node = Any
Edge = Tuple[Node, Node]
Graph = Tuple[Set[Node], Set[Edge]]

def independent_set_optimization_search(graph: Graph) -> Set[Node]:
    # ORACLE
    pass

def independent_set_threshold_decision(graph: Graph, k: int) -> str:
    largest = independent_set_optimization_search(graph)
    if len(largest) >= k:
        return "YES"
    return "NO"
```

Correctness:
Given an input graph, the oracle IndependentSet-OptimizationSearch will correctly return the largest independent set in the graph. Check the cardinality of this set. If it is greater than or equal to k, then there is an independent set of size at least k in G. In this case, the algorithm returns YES. If the greatest independent set in the graph is still smaller than k, the algorithm returns NO. So, for any valid input graph, a correct response of either YES or NO is returned given correct oracle to IndependentSet-ThresholdDecision.

Runtime:
Assert that set cardinality is a constant time operation. So, the algorithm runs in time $O(1)$ with a reduction to IndependentSet-ThresholdSearch. The question gives that the oracle for IndependentSet-ThresholdSearch runs in time $T(n, m) \geq n + m$. The complexity of this oracle call dominates the function, so the total time complexity of IndependentSet-ThresholdDecision is that of IndependentSet-OptimizationSearch, which 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.)

In any graph, the minimum number size of the independent set is 0 (empty graph), and the maximum size is the number of nodes total. Use these two bounds to binary search for the largest independent set. Consider this algorithm:

```
from typing import Set, Tuple, Any

Node = Any
Edge = Tuple[Node, Node]
Graph = Tuple[Set[Node], Set[Edge]]

def independent_set_threshold_search(graph: Graph, k: int) -> Set[Node] | None:
    """Find an independent set of size at least k in G"""
    # ORACLE
    pass

def independent_set_optimization_search(graph: Graph) -> Set[Node]:
    """Find the largest independent set in G"""
    lower_bound = 0
    upper_bound = len(graph[0])
    largest: Set[Node] = set()

    while lower_bound <= upper_bound:
        mid = (lower_bound + upper_bound) // 2
        set_geq_mid = independent_set_threshold_search(graph, mid)
        if set_geq_mid is None:
            upper_bound = mid - 1
        else:
            largest = set_geq_mid
            lower_bound = mid + 1

    return largest
```

Correctness:
Given any input graph, the binary search in the algorithm is capable of reaching any arbitrarily large or small independent set in the all-encompassing range of possibilities between the empty set and every node in the graph. If threshold search returns nothing, the binary search checks lower values of k. If threshold search returns a set, that set is saved and is returned unless a greater set is found later. Because threshold search returns sets greater than or equal to size k, binary search always receives a sufficiently large set and checks for sets even greater if any exist in the graph. In this way, the variable *largest* in the code above will always be the largest independent set in a well-formed input graph when it is returned. Because of this, the algorithm must be correct.

Runtime:
The runtime of the reduction is dominated by the binary search. In class, we've asserted that binary search has runtime $\log n$. That's also informally visible in the code, as the size of the range from *lower_bound* to *upper_bound* is cut in half at every step, and the size of the range initially is n (the number of nodes in the graph). So, the algorithm runs in time $O(\log n)$ with a reduction to $IndependentSet-ThresholdSearch$, which is called at most $\log n$ times. If the algorithm for $IndependentSet - ThresholdSearch$ runs in time $O(T(n+m))$, then the final runtime of $IndependentSet - OptimizationSearch$ is $O(\log n \cdot T(n,m))$.

(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.)

Consider this algorithm: First, intitialize a result set. Then, for each vertex in the graph, remove that vertex and all its neighbors from the graph. Also remove all edges referencing vertices no longer in the graph. Call IndependentSet-ThresholdDecision on the graph. If the response is YES, add the given vertex to the result set and leave it and all neighbors out of the graph. Decrement k by one. If the response is NO, leave the given vertex out of the graph, but add back all the neighbors that were removed (and any now-valid edges referencing those neighbors). By the end, if a large-enough set was found, return that. Otherwise, return $\perp$.

The algorithm is expressed more precisely in code below. Note the comment about edges.

```python
from typing import Set, Tuple, Any, Dict

Node = Any
Edge = Tuple[Node, Node]
Graph = Tuple[Set[Node], Set[Edge]]

def independent_set_threshold_decision(graph: Graph, k: int) -> str:
    # ORACLE
    pass

def index_edges(graph: Graph) -> Dict[Node, Node]:
    """Indexes edges by each endpoint"""
    indexed_edges: Dict[Node, Set[Node]] = {}
    [graph_nodes, graph_edges] = graph
    for node in graph_nodes:
        index_edges[node] = set()
    for edge in graph_edges:
        [node1, node2] = edge
        indexed_edges[node1].add(node2)
        indexed_edges[node2].add(node1)
    return indexed_edges

def independent_set_threshold_search(graph: Graph, k: int) -> Set[Node] | None:
    """Finds an independent set of size at least k in G"""
    [graph_nodes, graph_edges] = graph
    indexed_edges: Dict[Node, Set[Node]] = index_edges(graph)
    result_set: Set[Node] = set()

    for node in graph_nodes:
        graph_nodes.remove(node)
        for neighbor in indexed_edges[node]:
            graph_nodes.remove(neighbor)
        # !!! I'm leaving out edges to avoid code bloat.
        # Everywhere a node is removed from the graph, assume all edges
        # related to that node are also removed.
        # Similarly, when a node is re-added, assume its edges are also
        # re-added.
        part_of_res_set = independent_set_threshold_decision(graph, k - 1)
        if part_of_res_set == "YES":
            # keep neighbors out of the remaining graph
            result_set.add(node)
            k -= 1
            continue
        # keep node out, add neighbors back in
        for neighbor in indexed_edges[node]:
            graph_nodes.add(neighbor)

    if len(result_set) < k:
        return None
    return result_set
```

Correctness:
The algorithm accepts any valid graph and some k which the output cardinality must meet or exceed (if not none). Consider the loop evaluating every vertex in the input graph.

The vertex and its neighbors are removed from the input graph. If the vertex is part of the result set, then removing the neighbors will have no affect on the cardinatily of the remaining independent set. Removing the neighbors also guarantees that the removed vertex is capable of joining any remaining independent set. If such an independent set has size greater than or equal to $k-1$, then adding the removed vertex to that set leaves an independent set of size at least k, which is what the algorithm is looking for. This vertex is then added to the result set. Because this vertex has already been removed from the graph and added to the result set, k is decremented, and the algorithm continues searching for the remaining vertices in the independent set.

If no independent set of size greater than or equal to $k-1$ is found, then the current vertex cannot be in the independent set of size k, so it should stay removed from the graph. However, the vertex's neighbors might be in the target independent set, so they're added back to the graph.

In this way, every vertex is evaluated for potential membership in the result set and added if it belongs. Once every vertex has been considered, every vertex that belongs in the result set has been added to the result set, making it a complete independent set of vertices in the graph, which is returned. If no such set was found, then Null / $\perp$ is returned.

Thus, assuming the correctness of an oracle IndependentSet-ThresholdDecision, this algorithm returns a correct output for every valid input, making it correct.

Runtime:
My Python algorithm introduces a one time $O(n+m)$ loop to index the edges. This is somewhat implementation specific and is washed out by greater forces in the algorithm.

After that, the dominating complexity in the algorithm comes from the for loop, which iterates every node in the graph. So, the contents of the loop are run at most n times. For any given node, removing every neighboring edge and vertex may take at most $O(n+m)$ time. After IndependentSet-ThresholdDecision is called, another $O(n+m)$ steps may be introduced to re-add the neighboring edges and vertices to the graph (if NO was returned by the oracle call). Thus, when the oracle call is constant, the loop takes roughly $n+m$ time per iteration. However, the question also asserts that IndependentSet-ThresholdDecision has time complexity $T(n,m) \geq n+m$. Because it's explicitly stated that $T(n,m)$ is either equal to or greater than $n+m$, the complexity of the loop is best expressed by $T(n,m)$.

This produces runtime of $O(n \cdot T(n,m)+n+m$. Again, because $T(n,m)$ either meets or exceeds $n+m$, the complexity of processing the edges in the Python algorithm can also be dropped, leaving time complexity of $O(n \cdot T(n,m))$.

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 $c \approx 1.1996$.