CSCI 2300 — Introduction to Algorithms
Homework 7 (document version 1.2) — DUE April 29, 2020
P, NP, and NP-Complete

- This homework is due by 11:59PM on the due date above and must be submitted in Submitty as a single PDF called `upload.pdf`.
- Just like the labs, please avoid using Google to find suggestions or solutions. You are allowed to use the textbooks and class notes as references. We strongly encourage you to use the office hours, labs/recitations, and Discussion Forum to successfully complete this assignment.
- Also, you are allowed to consult with your classmates, collaborating in line with the policies set forth in the syllabus. You must write up your own separate solution; do not copy-and-paste anything.
- Remember the goal is to use your own brain to work these problems out, which will help you develop the skills to do well on the exams and, more importantly, become a substantially better computer scientist.
- For your solutions and proofs, please write clearly and concisely, and use rigorous formal arguments and pseudocode that mimics the style used in the Dasgupta textbook. Be consistent.
- In general, if you are asked to provide an algorithm, the best thing to do is to provide a clear description in English (e.g., "Use BFS, but with the following change..." or "Run BFS two times as follows...").
- Be as concise as you can in your answers. Long answers are difficult to grade.

Warm-Up Problems (do not hand in for grading)

Work out the problems below as practice problems before going on to the next section. Note that warm-up problems might sometimes show up on an exam!

1. DPV Problem 8.1 – **SOLUTIONS:**

   In this problem, we see the difference between an optimization problem and a search problem. A search problem has a polynomial-time algorithm to check any proposed solution and therefore shows that the search problem is in class NP-complete.

   To show that if TSP can be solved in polynomial time then $TSP - OPT$ can also be solved in polynomial time, we need to define a polynomial-time algorithm in which $TSP - OPT$ uses TSP.

   Let us assume that each edge weight $w_i \geq 1$.

   One initial approach that will *not* work here is to have $TSP{-}OPT$ call TSP with a budget $b = 1$, then a budget $b = 2$, and so on until a solution is found. This will result in an exponential number of times we would need to call TSP.

   Instead, we can define an upper bound on the length of the optimal tour and use binary search. So, define upper bound w_{max} here as simply the sum of all edge weights w_i. Using that upper bound, we can repeatedly call a binary search function that calls TSP with budget w_{max}. The binary search approach will repeatedly cut this budget in half.

   By using a binary search algorithm here, with w_{max} representing the upper bound on the length of the optimal tour, we know that $\log w_{max}$ calls to TSP are required. We can ensure this is polynomial time in the input size by noting that say e_{max} is the largest edge weight in the given graph. If all edges were e_{max} then we will need $|E| \log e_{max}$ bits to represent the case for w_{max}, which is polynomial in the input size.

2. DPV Problem 8.10 (parts (b) and (e) only) – show a reduction for each (not just a generalization), and for part (e), consider how the Sparse Subgraph problem can relate to the Independent Set problem

   – **SOLUTIONS:**

   (b) The Longest Path problem is as follows: Given graph G and integer g, find in G a simple path of length g. (Here, *simple* means we visit each vertex exactly once.)

   We reduce the Rudrata Path problem to the Longest Path problem. Given instance $I(G, n)$ of the Rudrata Path problem for graph G with n vertices, let $g = n - 1$. Then $f(I) = (G, n - 1)$ is an instance of the Longest Path problem.

   Given solution S to the Longest Path problem for instance $f(I)$, we define algorithm $h(S) = S$, i.e., solution S is also a solution to the Rudrata Path problem.

   Note that a simple path of length $n-1$ must contain exactly n vertices. Also, any Rudrata path is of length $n - 1$. Therefore, by definition, we must have a Rudrata path.

   Further, for any graph with n vertices, instance $(G, n - 1)$ is precisely asking for the Rudrata path.

   (e) The Sparse Subgraph problem is as follows: Given graph G and two integers a and b, find a set of a vertices of G such that there are at most b edges between them.

   We can reduce the Independent Set problem to the Sparse Subgraph problem.

   Given instance $I(G, k)$ for the Independent Set problem, with k vertices as our goal, let $a = k$ and $b = 0$. Then $f(I) = (G, a, b) = (G, k, 0)$ is an instance of the Sparse Subgraph problem.

   Next, given solution S to the Sparse Subgraph problem, we define algorithm $h(S) = S$, i.e., solution S is also a solution to the Independent Set problem.

   Here, any subgraph of G with k vertices and zero edges must be an independent set of size k.

3. DPV Problem 8.11 (all parts) – **HINTS:**

   (a) To reduce the Directed Rudrata Path problem for graph $G = (V, E)$ to the (undirected) Rudrata Path problem, consider how we can create undirected graph G' from given directed graph G.

   (b) To reduce the Directed Rudrata Path problem to the (undirected) Rudrata (s,t)-Path problem, we can use the solution to part (a) as long as we define what s and t map to in that solution.

4. DPV Problem 8.12 (all parts) – **SOLUTIONS:**

   (a) **SOLUTION:** To show that the k-Spanning Tree problem is a search problem, we need to provide a polynomial-time algorithm that can verify a given proposed solution. In other words, given a spanning tree T for graph G, we need to verify that: (1) T is indeed a tree; (2) each edge in T is also in G; (3) each vertex in G is also in T; and (4) each vertex in T has degree $k \leq 2$. All four of these can be achieved in polynomial time using DFS.

   (b) **HINT:** To show that the k-Spanning Tree problem is in class NP-complete, we first need to complete part (a) above, then we need to come up with a reduction between the k-Spanning Tree problem and, as the hint suggests, the Rudrata Path problem. Start by reducing the 2-Spanning Tree problem to the Rudrata path problem. Next, for $k > 2$, consider how you could construct new graph G' from G.

Required Problems (hand in for grading)

Work out the problems below and submit for grading before the deadline.

1. DPV Problem 8.2 – **SOLUTIONS:**

   → 3pts for correct use of a `hasRudrataPath()` function
   → 3pts for solution running in polynomial time

   Given a hypothetical polynomial-time procedure that determines whether a graph has a Rudrata path, we can use it to develop a polynomial-time algorithm for the Rudrata Path problem itself.

   Call this procedure `hasRudrataPath()` and note that it returns true or false, not the actual path. Given graph $G = (V, E)$, assume that `hasRudrataPath()` returns true. We can arbitrarily order the edges in E, removing them one by one. If the graph obtained by removing edge e_i still has a Rudrata path (i.e., `hasRudrataPath()` still returns true), then keep removing edges.

   If the graph obtained by removing edge e_i does *not* have a Rudrata path (i.e., `hasRudrataPath()` returns false), then add edge e_i back to the graph. This graph is then a solution to the Rudrata Path problem for the original graph.

2. DPV Problem 8.4 (parts (a) and (b) only) – **SOLUTIONS:**

   (a) → 2pts for a correct checking algorithm
   → 2pts for checking algorithm running in polynomial time

   To prove that a problem is in class NP, we need to identify a polynomial-time algorithm to check whether a proposed solution is correct or not. Given a clique in the graph, we can easily verify in polynomial time that there is an edge between every pair of vertices. To do so, we can use a brute force approach and systematically check every pair of vertices, which is $O(n^2)$.

   (b) → 3pts for stating what is wrong
   → 1pt for stating how to correct this

   The reduction is in the wrong direction. We must reduce the Clique Problem to the Clique-3 Problem if we intend to show that the Clique-3 Problem is at least as hard as the Clique Problem.

   Note that it turns out that the Clique-3 Problem is not in class NP-complete. (In fact, we can come up with a polynomial-time algorithm for the Clique-3 Problem.)

3. DPV Problem 8.13 (parts (a), (b), and (c) only) – consider how the given problems could relate to the Rudrata Path problem – **SOLUTIONS:**

   (a) $\rightarrow$ 3pts for a polynomial-time algorithm

   This problem can be solved in polynomial time. Given a set of nodes $L \subseteq V$, the goal is to find a spanning tree such that its set of leaves includes set L. This implies there might be other leaves in the resulting tree.

   Delete all vertices in set L from the given graph G and find a spanning tree T for the remaining graph. Next, for each vertex $u \in L$, add u to any of its neighbors in T.

   Tree T, if it is possible to construct, must have all vertices in L as leaves. If the graph becomes unconnected after removing L or some vertex in L has no neighbors in G L (i.e., graph G without vertices from L), then no spanning tree exists with all vertices in L as leaves.

   (b) $\rightarrow$ 3pts for showing a reduction (NP-complete)

   This problem cannot be solved in polynomial time and is in class NP-complete. Given a set of nodes $L \subseteq V$, the goal is to find a spanning tree such that its set of leaves is precisely set L. In other words, no other leaves may exist in the resulting tree.

   To show this is in class NP-complete, we need to define a reduction from an NP-complete problem to this problem. We can reduce the (undirected) (s,t)-Rudrata Path problem to this current problem.

   Given graph $G = (V, E)$ and two vertices s and t in V, we set $L = \{s, t\}$. Next, we claim that the tree must be a path between s and t. It cannot branch out anywhere because each branch must end at a leaf defined in L and the only options are vertices s and t. Further, since it must be a spanning tree, the path must include all vertices of G.

   (c) $\rightarrow$ 3pts for showing a reduction (NP-complete)

   Like in part (b), this problem cannot be solved in polynomial time and is in class NP-complete. Given a set of nodes $L \subseteq V$, the goal is to find a spanning tree such that its set of leaves is included in set L. In other words, all leaves of the spanning tree must be in L.

   To show this is in class NP-complete, we can define the same reduction as that shown in part (b) above. Note that a tree must have at least two leaves, so for set $L = \{s, t\}$, the set of leaves in resultant tree T must be exactly those two vertices.