Assignment 4: Reductions (10%)

Winter 2024 – CPSC 413

Due at 23:59, Apr. 10 on D2L

Assignment policy:

- This is an **individual** assignment, so the work you hand in must be your own. Any external sources used must be properly cited (see below).
- Submit your answers to the written questions as a **single PDF** to the assignment on Gradescope. Code files for any programming questions will need to be uploaded separately to the corresponding category on Gradescope.
- Extensions will not be granted to individual students. Requests on behalf of the entire class will only be considered if made more than 24h before the original deadline.
- All submissions must be clearly legible. Handwritten assignments will be accepted, but marks may deducted if the TAs cannot read your writing. It is recommended to use LaTeX to typeset your work. An assignment template is available on D2L.
- Some tips to avoid plagiarism in your assignments:
    1. Cite all sources for material you hand in that is not your original work. This applies to both proof-based and coding questions. You can put the citation into the submitted PDF or as a comment in your code. For example, if you find and use code found on a website, include a comment that says, for example:

       ```
       # the following code is from https://www.quackit.com/python/tutorial/python_
       hello_world.cfm.
       ```

       Use the complete URL so that the marker can check the source.
    2. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. However, you may still get a low grade if you submit work that is not primarily your own.
    3. Discuss and share ideas with other students as much as you like, but make sure that when you write your assignment, it is your own work. A good rule of thumb is to wait 20 minutes after talking with somebody before writing it down. If you exchange written material with another student, take notes while discussing with a fellow student, or copy from another person's screen, then this work is not yours.
    4. We will be looking for plagiarism in all submissions, possibly using automated software designed for the task.
    5. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help than it is to plagiarize.

Questions:

1. Consider the following two problems:

   k-Vertex Cover: Given a graph $G = (V, E)$, where $n = |V|$, does there exist a vertex cover $C \subseteq V$ of G such that $|C| \leq k$?
   Minimum Vertex Cover: Given a graph $G = (V, E)$, where $n = |V|$, what is the minimum size of a vertex cover $C \subseteq V$ for G?

   Observe that k-Vertex Cover is a decision problem, as its output is either "yes" or "no". As discussed in lecture, k-Vertex Cover is an example of a problem that is NP-complete, so finding a polynomial time algorithm that solves it would be a proof that **P = NP**.

   (a) (1.5 marks) Suppose you managed to find an algorithm that solves k-Vertex Cover in polynomial time (in n) and won the million dollars and everlasting fame. How could you use this to solve Minimum Vertex Cover, which is not a decision problem, in polynomial time (in n)?

   (b) (0.5 marks) Is this a reduction from k-Vertex Cover to Minimum Vertex Cover, or a reduction from Minimum Vertex Cover to k-Vertex Cover?

   Note that clearly, if you can solve Minimum Vertex Cover in polynomial time, then you can use that to solve k-Vertex Cover in polynomial time (the reduction itself is constant time). These two problems are therefore considered "equivalent" in the sense that each is *reducible* to the other in polynomial time. The purpose of this question is to justify the statement that Minimum Vertex Cover is NP-complete, which is a slight abuse of notation because NP is by definition a class of decision problems.

2. Below is the decision version of the Knapsack problem. Recall that we gave a dynamic programming algorithm for the maximization version of this problem in class.

   T-Knapsack: Given a list of n item types with weights w_i and values v_i, a capacity K, and a target value T, is there a selection of items with total weight at most K such that the combined value of the items is at least T?

   For the following problems, recall that a "polynomial time" algorithm means that the runtime is a polynomial expression in the size of the input. Here, as usual, the input size refers to the **number of bits** needed to represent it. This will be important as you write your solutions below, so discuss with your TA if you need clarification.

   (a) (1 mark) Like the decision version of 0-1 Knapsack, which you will discuss in tutorial, T-Knapsack is **NP**-hard. To show that T-Knapsack is **NP**-complete, you also need to show it is in **NP**. Prove that T-Knapsack is in **NP** by giving a 1-certificate and describing an algorithm that can verify it in polynomial time.

(b) (2 marks) Now, recall the standard Knapsack problem, which we covered in the lectures on dynamic programming.

> **Knapsack:** Given a list of n item types with weights w_i and values v_i, and a capacity K, what is the largest total value of items you can put in the knapsack without the total weight exceeding the capacity?

Suppose you managed to find an algorithm that solves T-Knapsack in polynomial time. How could you use this to solve Knapsack, which is not a decision problem, in polynomial time? As with the previous question, the goal is to justify the informal statement that Knapsack is **NP**-complete.

Hint: you will need to find an upper bound for the largest possible value that can be returned by Knapsack. You can assume for simplicity that w_i and v_i are positive 32-bit integers, so the input size is determined only by n and K.

3. (3 marks) Recall the two lilypad-hopping problems you have worked on so far: one on the midterm, and one on your previous assignment. We will consider the versions of these two problems where the lilypads can be any integer (not just ± 1). The problem statements are given below as a reminder.

Problem A (varied-length hops)

> **Input:** An array of n integers, $x_1, x_2, \ldots x_n$, where x_i represents the value of the i^{th} lilypad.
> **Output:** The maximum total score that Fred can accumulate by jumping across the row of lilypads, where Fred can jump from the i^{th} lilypad to either the $(i+1)^{th}$ or the $(i+2)^{th}$ lilypad at each step.

Problem B (coloured lilypad hops)

> **Input:** Three arrays of n integers each, denoted $r_1, r_2, \ldots r_n$, $g_1, g_2, \ldots g_n$, and $b_1, b_2, \ldots b_n$, where r_i, g_i and b_i represent the values of the red, green, and blue lilypads in the i^{th} cluster.
> **Output:** The maximum total score that Fred can accumulate by jumping across the row of lilypads, where Fred always jumps from the i^{th} cluster to the $(i+1)^{th}$ cluster and cannot land on the same colour of lilypad twice in a row.

In both problems, Fred starts on the ground, so his first jump lands him on either the first or second lilypad in the problem A, or any of the three lilypads in the first cluster in problem B. He must also finish on the ground on the opposite side (so he can't stop partway through the row, and in problem A he doesn't necessarily need to include the last lilypad).

Give an $O(n)$ reduction from A to B such that the output of the black box/oracle for B is returned **unchanged**[1]. Describe your algorithm for the reduction and justify why it is correct (this should include translating a sample input for clarity). You should also justify in a sentence or two why your reduction runs in $O(n)$.

Hint: notice that in the first problem, Fred cannot skip two lilypads in a row (his jump size is at most two, so if he misses a lilypad he will have to land on the next one). You'll have to find a way to capture this constraint using the constraints iin the second problem, i.e. that Fred must change the colour of his lilypad with each jump.

4. (2 marks) In class, we have discussed how SAT can be used to model a wide variety of problems involving constraints. In addition, it is such a standard problem that there exist libraries for many languages that are optimized for solving SAT. These libraries are known as **SAT solvers**, and this question will give you some practice with using them. In particular, your job will be to build a function that translates an input for graph 3-colorability into an input for SAT, which can then be run through the PySAT solver.

   Write a Python 3 function that, given an input graph, returns a Boolean value indicating whether the input graph is 3-colourable. Your script needs to use the reduction discussed in lecture to translate the input graph into a boolean formula that can be solved using PySAT[2].

   Your function should be contained in the Python 3 starter class provided on D2L. This class contains a single `solve()` function with an argument named `graph`, which will be an adjacency matrix. Your goal is to convert this to a boolean formula (i.e. an array of clauses). You **must** implement the reduction presented in class to receive marks for this component.

   Upload `colourSAT.py` to the correct location on Gradescope. Make sure your code is clean and well-commented.

   Finally, it is strongly recommended that you attend your tutorials to see the TA demos for how to use PySAT. They will be covering similar examples, so it will make your life easier when solving this question. These tutorials will happen on April 3–5.

[1]Note that both algorithms are solvable in $O(n)$ with dynamic programming, but this requirement means that you can't simply solve problem A, ignore the oracle output, and return the solution you computed for A directly. You must somehow find a way to translate inputs for A into "equivalent" inputs for B.

[2]`https://pysathq.github.io/`

Autograder notes:

Your code will be graded by an autograder on Gradescope. For your code to work with the autograder, you will need to use the template class `colourSAT.py` provided on D2L. You should write your code to fill in the `solve()` function **without changing the arguments**, and you must return (not print) a boolean value indicating whether the input graph is 3-colourable. You do not need to change the rest of the class. Do not change the filename, and do not upload any files other than your `colourSAT.py`.

You can test your code with the autograder as many times as you like before submitting. It is also recommended that you add print statements for debugging during your own initial development. There is a set of test cases included in `a4test.py`, provided on D2L, that you can use to test your code locally if you prefer.

Note that marks will be allocated to your code structure and commenting, so pay attention to this too.