We have spent time exploring some of the theoretical aspects of NP-completeness, as well as a sampling of NP-complete problems. On this assignment, you will explore the more practical side of NP-completeness: what can you do when you really, actually need to solve an NP-complete problem?

It turns out that for quite a few natural NP-complete problems, even though all known algorithms take exponential time *in the worst case*, the problems can be solved reasonably quickly *on the particular kinds of inputs that arise in practice*. A paradigmatic example is the SAT problem which we explored in class. In theory, no one knows any way to solve SAT instances which is asymptotically faster than just trying all possible assignments; in practice, however, there are algorithms for solving SAT which seem to be quite fast on the kinds of SAT instances which actually turn up in practice, even with up to tens of thousands of variables. These algorithms are implemented in software known as *SAT solvers*; developing and improving SAT solvers is an active area at the intersection of theory and practice.

In practice, then, if you need to solve an NP-complete problem, one good strategy is to reduce your problem to SAT, and then give the resulting SAT instance to an existing SAT solver. On this assignment you will carry out this kind of practical reduction strategy for a particular NP-complete problem known as GRAPH COLORING.

## *What to turn in*

You should turn in two files:

- `certificate.txt`, described in the section "Coloring your graph" below.
- Whatever program(s) you used to help you generate `certificate.txt`.

## *Background: graph coloring*

Suppose we have an undirected graph $G = (V, E)$. Given some set of colors $C$, a *coloring* of $G$ is a function $\chi : V \to C$, that is, an assignment of a color to each vertex. A *valid* coloring is one for which no two adjacent vertices have the same color; that is, $\chi(u) \neq \chi(v)$ for every $(u, v) \in E$.

For example, Figure 1 shows a valid coloring of a graph with 7 vertices. You can check that no two adjacent vertices have the same color.

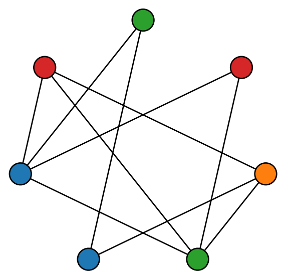
Figure 1: A valid 4-coloring of a graph with 7 vertices

A $k$*-coloring* is a valid coloring of a graph which uses at most $k$ colors. The example shown in Figure 1 is a 4-coloring (and also a 5-coloring, and a 26-coloring, and so on). A graph $G$ is called $k$*-colorable* if it has a valid $k$-coloring. Finally, the *chromatic number* of a graph $G$ is the *smallest* $k$ for which $G$ is $k$-colorable. The graph shown in Figure 1 actually has a chromatic number of 3—can you find a valid 3-coloring (and prove that no 2-coloring exists)?

Given a graph $G$ and a number $k$, the GRAPH COLORING problem asks: is the graph $k$-colorable? There are $k^{|V|}$ possible colorings to check, so a brute force algorithm is definitely not going to work. In fact, this problem is NP-complete for $k \geq 3$, which can be proved by reduction from 3-SAT. The proof has a similar flavor to the reduction 3-SAT $\leq_P$ INDEPENDENT SET we did in class: given a 3-SAT instance, one builds a particular graph out of "gadgets" which encodes the constraints given by the 3-SAT clauses, in such a way that a 3-coloring of the graph corresponds to a valid truth assignment. For details of the proof, see, for example, `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/sat.pdf`.

For $k = 2$, however, it can be solved efficiently—can you figure out how?

## *Defining your graph*

For this assignment you will work with your own personal graph. Here is how to determine your graph:

1. Pick a specific string to represent your name. For example, mine might be `"Brent Yorgey"` or `"Brent A. Yorgey"` or `"Dr. Yorgey"`—it does not matter what string you pick, as long as you specify it exactly.

2. Find the MD5 hash of your string, which should yield a 16-byte hash value. For example, in Python 2.x, given your chosen string stored in a variable `name`, one can write

```
import md5

hash_bytes = md5.new(name).digest()
```

   to compute the MD5 hash. In Python 3.x:

```
import hashlib

hash_bytes = hashlib.md5(b"Your Name").digest()
```

   In Java, one can write something like

```
import java.security.*;

byte[] bytesOfMessage = name.getBytes("UTF-8");

MessageDigest md = MessageDigest.getInstance("MD5");
byte[] hash_bytes = md.digest(bytesOfMessage);
```

3. The resulting 16-byte hash can of course be thought of as a sequence of 128 bits. We can use a sequence of bits to encode an undirected, unweighted graph as follows. Each bit records the presence or absence of a single edge: a 1 bit means the edge is present, and 0 means it is absent. We assume that the vertices of the graph are numbered from 0 to $n-1$.

   - The index-0 bit, that is, the *least significant* or *last* bit, corresponds to the edge $(1,0)$.
   - The bit at index 1, i.e. the second-to-last bit, corresponds to $(2,0)$, and the bit at index 2 corresponds to $(2,1)$.
   - The next three bits correspond to $(3,0)$, $(3,1)$, and $(3,2)$.

   The general pattern is that the first $1+2+3+\cdots+k = k(k+1)/2$ bits (counting from the *end* of the bit string) correspond to all the possible edges involving vertices numbered 0 through $k$; the next $k+1$ bits after that correspond to edges from vertex $k+1$ to all the smaller vertices, starting with vertex 0.

   For example, we need $(4\cdot 5)/2 = 10$ bits to encode a graph with 5 vertices; the graph shown in Figure 2 is encoded by 0010001101. The last bit (a 1) indicates that there is an edge from vertex 1 to vertex 0; the second-to-last bit (a 0) indicates that there is no edge from 2 to 0; and so on.

   In general, to determine whether vertices $i$ and $j$ are connected, first sort them so $i > j$, and then look at the bit with index $i(i-1)/2 + j$.

   Since $16(16-1)/2 = 120 < 128$ (but $17(17-1)/2 = 136 > 128$), your 128-bit hash value defines a particular undirected graph with 16 vertices. (The remaining 8 bits of your hash value are not used.)

   Note that there is an annoying mismatch between the way we are numbering the bits (starting from the right) and the way the individual bytes in an array or string are numbered (starting from the left). To make things easier on yourself, after computing the MD5 hash you may want to reverse the list/array of bytes, so that the last byte will be at index 0, the second-to-last at index 1, and so on. Then bit $j$ will be the $(j \bmod 8)$th bit in the $\lfloor j/8 \rfloor$th byte. Alternately, you can just turn your list of bytes into a list of bits, and then reverse the entire list.

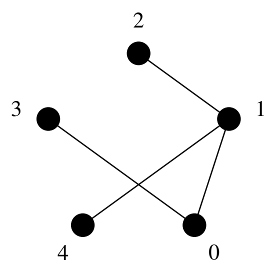

Figure 2: The graph encoded by 0010001101.

As an example, my graph (generated from the string `Brent Yorgey`) is shown in Figure 3. For example, the final 10 in the hexadecimal representation of the hash corresponds to the bits 00010000, which correspond to edges in the graph as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| (4,1) | (4,0) | (3,2) | (3,1) | (3,0) | (2,1) | (2,0) | (1,0) |

As you can see, there is indeed an edge between vertices 1 and 3 in the graph, and no edges between any of the other indicated pairs of vertices. To

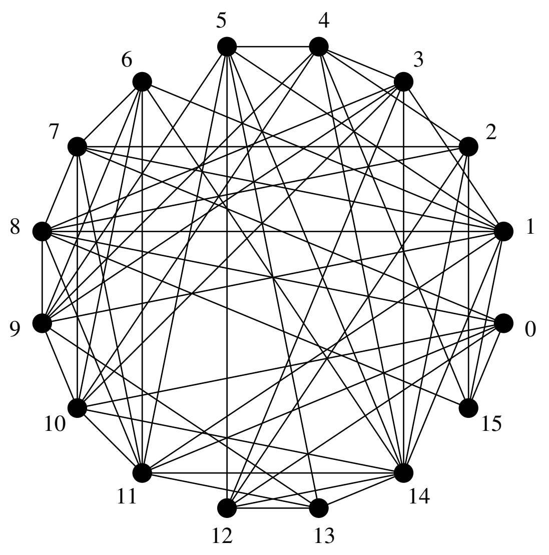

Figure 3: The graph generated by "Brent Yorgey", corresponding to the hash `ad 02 2f e3 f6 88 00 b6 f1 db 37 a8 f8 e1 4b 10`

make sure you understand how a hash value determines a graph, you can spot-check other edges for yourself: pick a pair of vertices and check that the corresponding bit in the hash value is what you expect.

**Warning**: be careful that you don't accidentally drop leading zeros when computing the hash! As a test you can hash my name and make sure that the hexadecimal `02` and `00` are handled properly.

## *Coloring your graph*

The question you now need to answer is: what is the chromatic number of your graph? Most randomly chosen graphs with 16 vertices seem to have chromatic number 5, so that wouldn't be a bad guess, but of course it could in theory be as small as 1 (if your graph has no edges) or as large as 16 (if your graph includes every possible edge). Empirically, it seems likely that at least one of you will have a graph with chromatic number 6. In any case, guessing the right chromatic number would not be enough anyway: in addition to determining the chromatic number $k$, you should construct an actual valid $k$-coloring, which can serve as an efficiently-checkable *certificate* that your graph is indeed $k$-colorable.

On the other hand, coming up with an efficiently-checkable certificate proving that your graph is *not* $(k-1)$-colorable is quite a different story, and is likely to be impossible, though no one knows for sure.

Ultimately, you should turn in a file named `certificate.txt` containing three lines:

1. The first line should contain the exact string you chose to generate your graph.
2. The second line should contain a single number, the chromatic number $k$ of your graph.
3. The third line should contain 16 characters specifying a valid coloring for the vertices $0 \ldots 15$ in your graph. It does not matter which characters you

use, but there should be exactly $k$ different characters, one standing for each color.

For example, my `certificate.txt` looks like this:

```
Brent Yorgey
5
eeedcddbabaccebd
```

You can check that assigning the same color to the first three vertices, a different color to the next vertex, and so on, is indeed a valid 5-coloring of the graph generated by the string `Brent Yorgey`, as shown in Figure 4.

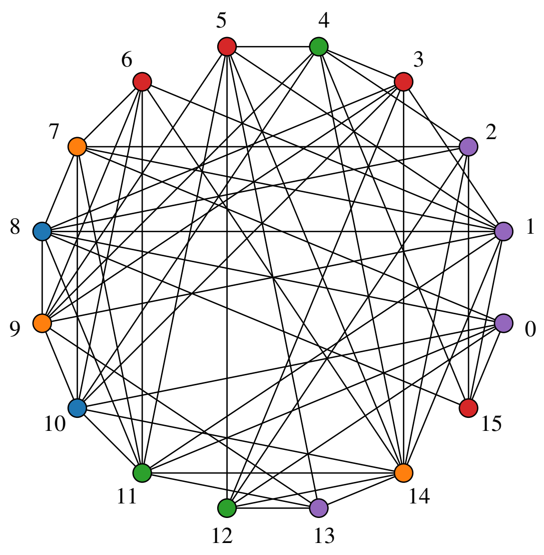

Figure 4: A 5-coloring of the "Brent Yorgey" graph

At this point you might wonder how you are supposed to come up with a coloring for your graph. Even with the relatively small value $n = 16$, a brute force algorithm might have to check, for example, up to $5^{16} = 152587890625$ potential 5-colorings. One might try a greedy strategy: for each vertex from $0 \dots n-1$, pick the smallest color that is unused by any of the previous vertices it is connected to. But as you can see if you try it, this does not work. For example, this greedy algorithm on my graph produces the (non-optimal) 6-coloring shown in Figure 5. Notice how vertex 14 is given a sixth color (brown), since by the time it is reached, it has at least one neighbor with each of 5 different colors.

In fact, we will use a tried-and-true technique for attacking computationally hard problems like this: reduce the problem to SAT, and hand it off to a *SAT solver*, a program specifically designed to solve SAT instances. Even though all such programs (that we know of) take exponential time in the *worst* case, they are astonishingly efficient in the general case.

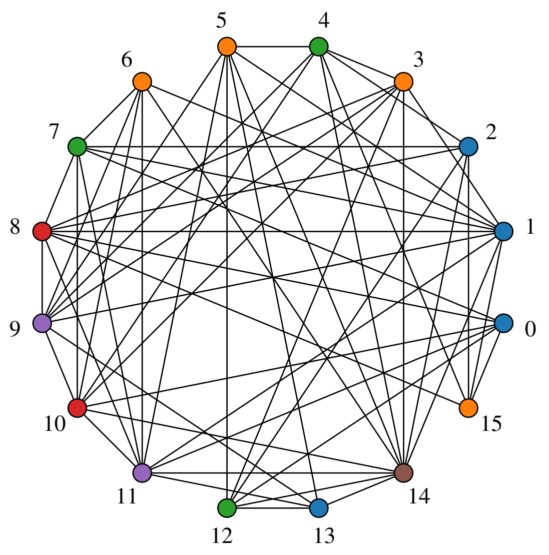

Figure 5: A non-optimal greedy coloring of the "Brent Yorgey" graph

## *Using the Yices SAT solver*

For this assignment, you will use a SAT solver called Yices, available at `http://yices.csl.sri.com/`. There are many solvers available; I chose Yices simply because it is freely available on multiple platforms and easy to get started with. (If you want to try a different one for some reason, you are welcome to.)

Download the latest version of Yices (2.5.2 as of this writing) for your operating system and unzip it somewhere. In the `bin/` folder you should find several executables, one of which is `yices-sat`. This is the SAT solver that comes with Yices. (In fact, Yices is not *just* a SAT solver, but actually something called an "SMT solver", which is much more general and powerful than a SAT solver; ask me if you are interested in the details.)

The format expected by `yices-sat` is described at `http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`. The short version is that the first line should contain

```
p cnf n k
```

where the text `p cnf` occurs literally, and `n` and `k` are replaced by the number of variables and the number of clauses, respectively. For example, `p cnf 7 20` would denote a SAT instance with 7 variables and 20 clauses.

`cnf` stands for *conjunctive normal form*.

The next `k` lines should describe the clauses. Each line consists of a list of integers, separated by spaces, with an extra 0 at the end. The integers indicate variables, numbered $1 \dots n$, with an integer $i$ corresponding to $x_i$ and $-i$ corresponding to $\overline{x_i}$. For example, to encode the clause $(x_1 \lor \overline{x_3} \lor x_7 \lor \overline{x_4})$ one would write

```
1 -3 7 -4 0
```

As a complete example, suppose we have 4 variables $x_1 \dots x_4$ and the 3 clauses

$$(x_1 \lor x_4), (x_2 \lor \overline{x_3} \lor \overline{x_4}), (\overline{x_1} \lor x_3 \lor x_4).$$

We would encode this as the file

```
p cnf 4 3
1 4 0
2 -3 -4 0
-1 3 4 0
```

Suppose this was saved in a file called `example.cnf`. Then we can give it as input to `yices-sat` just by giving the file name as an argument at the command line:

```
$ yices-sat example.cnf
sat
```

(Note that `$` indicates the command prompt, and should not be typed.) Yices returns immediately and tells us the clauses are satisfiable. To see the actual assignment it came up with, pass it the `-m` option:

```
$ yices-sat -m example.cnf
sat
-1 -2 -3 4 0
```

It tells us that a satisfying assignment consists of setting $x_1$, $x_2$, and $x_3$ to $F$, and $x_4$ to $T$.

On the other hand, given the input file

```
p cnf 2 4
1 2 0
1 -2 0
-1 2 0
-1 -2 0
```

`yices-sat` prints `unsat`, telling us that the set of clauses is not satisfiable (as you can easily verify).

## *Reducing to SAT*

You will need to write a program that reduces $k$-colorability of your graph to a SAT instance, and outputs an appropriate input file for `yices-sat`. You can then try each value of $k = 1, 2, \dots$ until you find the smallest one for which the corresponding instance is satisfiable. (In general one could of course use binary search, but for such small values of $k$ it is hardly worth the effort.) Then use the satisfying assignment generated by Yices to construct a valid $k$-coloring of your graph.

*Hint:*

*Hint*: I strongly suggest that you also write a program to *check* the validity of a certificate! This is much easier than the program you will need to write to find the certificate in the first place.