Assignment 7 — Reductions

Due: Friday, November 22nd

Credit: This is a lab by Christopher Siu.

We can show that a problem B is at least as hard as a problem A by showing that an algorithm that solves B can be used to solve A. In this assignment, you'll develop programs that use each other to solve $\mathcal{NP}$-Complete problems.

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/m2Xn4Y9f`

Required Files: `compile.sh`, `run.sh`

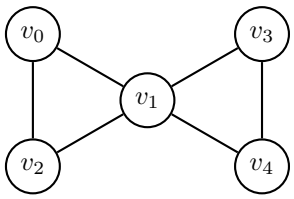
Optional Files: `*.py`, `*.java`, `*.clj`, `*.kt`, `*.js`, `*.sh`

Part 1: Subgraph Isomorphism

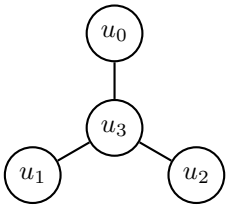
Recall that two graphs, $G = (V_G, E_G)$ and $H = (V_H, E_H)$, are *isomorphic* if there exists a bijection $f : V_G \to V_H$ such that two vertices $u, v \in V_G$ are adjacent if and only if the corresponding vertices $f(u), f(v) \in V_H$ are also adjacent. In other words, two graphs are isomorphic if they have the same shape[1], even if their vertices are named differently.

The Subgraph Isomorphism problem asks whether, given two graphs G and H, G contains a subgraph isomorphic to H. For example, given:

G:



H:



Here, G contains a subgraph isomorphic to H. Note that the reverse is not necessarily true: H does not contain a subgraph isomorphic to G. Further note that every graph contains a subgraph isomorphic to itself.

The Subgraph Isomorphism problem is known to be $\mathcal{NP}$-Complete; it is among the hardest problems to solve for which a solution can still be verified efficiently. The given `subgraph_isomorphism.py` implements a naïve, brute force algorithm[2] to solve this problem.

This implementation can be invoked from the command line, given two files containing edge lists:

```
>$ python3 subgraph_isomorphism.py graph_g.txt graph_h.txt
Isomorphic vertices:
0, 1, 2, 3
>$ echo $?
0
```

…if G contains a subgraph isomorphic to H, `subgraph_isomorphism.py` prints the vertices of that subgraph and terminates with exit status `0`. If not, it terminates with exit status `1`:

```
>$ python3 subgraph_isomorphism.py graph_g.txt graph_h.txt
No isomorphic vertices.
>$ echo $?
1
```

[1] "isomorphic", from Greek, "*ἰσόμορφος*", meaning "of equal form"

[2] It's not the most efficient known approach, but it is straightforward to read.

Alternatively, both the Subgraph Isomorphism implementation and its associated graph data structure can be imported like any other Python module:

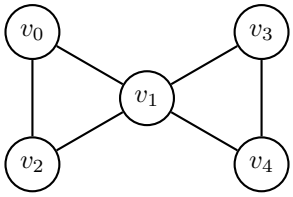
```
import graph
import subgraph_isomorphism as si

graph_g = graph.Graph()
graph_h = graph.Graph()

subgraph = si.isomorphic_subgraph(graph_g, graph_h)
```

…if G contains a subgraph isomorphic to H, `subgraph_isomorphism.isomorphic_subgraph` returns such a subgraph. If not, it returns `None`.

Part 2: Clique to Subgraph Isomorphism

A *clique* is a subgraph within which every vertex is connected to every other vertex. The Clique problem asks whether, given a graph G and an integer k, G contains a clique on k vertices.

For example, recall the following graph from above:



This graph contains cliques on $k = 1$, $k = 2$, and $k = 3$ vertices, but not any cliques on $k = 4$ vertices.

In your programming assignment of choice per Assignment 1, implement an algorithm that, given a graph G and a natural number k, uses the given Subgraph Isomorphism implementation to determine whether or not G contains a clique on k vertices.

The specific input and output format of this implementation is up to you, as your code is the only code that will make use of it. The only requirement is that you *must* use the given Subgraph Isomorphism implementation; that is, your pseudocode should look something like:

CLIQUE$(G \leftarrow (V, E), k)$

Input: A graph G and a natural number k
Output: Whether or not G contains a clique on k vertices

$\vdots$

SUBGRAPHISOMORPHISM$(\ldots)$

$\vdots$

…thereby reducing Clique to Subgraph Isomorphism, demonstrating that Subgraph Isomorphism is at least as hard as Clique is.

Part 3: 3-SAT to Clique

The *boolean satisfiability problem*, or "SAT", asks whether, given a proposition, there exists an assignment of truth values to propositional variables that satisfies the proposition. The variation known as "3-SAT" restricts the given propositions to conjunctive normal form with exactly 3 literals per clause.

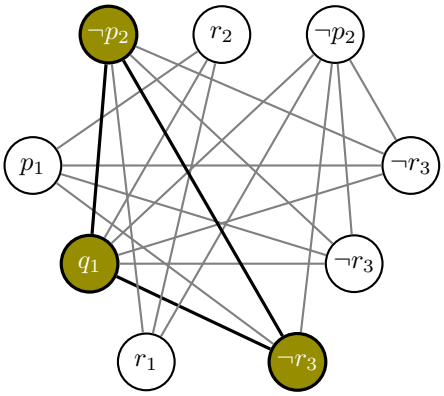
3-SAT can be reduced to Clique by representing the proposition as a graph, where:

- A vertex p_i represents a literal p in clause i.
- Two vertices m_i and n_j are connected by an edge if and only if $i \neq j$ and $m_i \not\equiv \neg n_j$ (m and n may or may not be the same literal).

In other words, the edges of this graph indicate potential non-conflicting assignments of truth values between different clauses. For example, given:

$$(p \vee q \vee r) \wedge (\neg p \vee r \vee \neg p) \wedge (\neg r \vee \neg r \vee \neg r)$$

…this proposition is represented by:



Since edges represent non-conflicting, inter-clause assignments, and we must make a set of assignments such that at least one literal in every clause is true, and none of our assignments can conflict with each other, we need to find a clique on k vertices within this graph, where k is the number of clauses in the given proposition.

The vertices of that clique then indicate the satisfying assignments; if no such clique can be found, then no satisfying assignment exists. In the example above, finding a clique on $k = 3$ vertices yields that $p \equiv F$, $q \equiv T$, and $r \equiv F$ will satisfy the given proposition, a fact that is verified by the corresponding truth table:

p	q	r	$(p \vee q \vee r)$	$(\neg p \vee r \vee \neg p)$	$(\neg r \vee \neg r \vee \neg r)$	*conjunction*
T	T	T	T	T	F	**F**
T	T	F	T	F	T	**F**
T	F	T	T	T	F	**F**
T	F	F	T	F	T	**F**
F	T	T	T	T	F	**F**
F	T	F	T	T	T	**T**
F	F	T	T	T	F	**F**
F	F	F	F	T	T	**F**

In your programming language of choice per Assignment 1, implement an algorithm that, given a proposition in CNF with exactly 3 literals per clause, uses your Clique implementation to determine whether or not the proposition is satisfiable.

You *must* make use of your Clique implementation to solve 3-SAT, and, by extension, you *must* make use of the given Subgraph Isomorphism implementation. By doing so, you have demonstrated that:

- Subgraph Isomorphism is at least as hard as Clique.
- Clique is at least as hard as 3-SAT.

Additionally, we have that:

- 3-SAT is known to be $\mathcal{NP}$-Complete, and both Clique and Subgraph Isomorphism are in $\mathcal{NP}$.

…therefore, both Subgraph Isomorphism and Clique are also $\mathcal{NP}$-Complete.

You may assume that the proposition will be well-formed, using '`~`' to indicate negation, '`&`' to indicate conjunction, and '`|`' to indicate disjunction[3]. You may also assume that all propositional variables will be one of the 26 lowercase English letters, and that a single space will appear between all literals and operators.

For example, the above proposition would be represented as:

```
( p | q | r ) & ( ~p | r | ~p ) & ( ~r | ~r | ~r )
```

Your program must accept as a command line argument the name of a file containing a proposition as described above, then print to `stdout` its satisfiability according to the following format:

- If the proposition is satisfiable, the satisfying assignments must appear on a single line, sorted in alphabetical order by their propositional variables.
- If the proposition is unsatisfiable, a message must indicate as such.

For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
Satisfying assignment:
~p, q, ~r
```

You may further assume that the satisfying assignment, if one exists, will be unique[4]. Your program will be tested using `diff`, so its printed output must match *exactly*.

Part 4: Submission

The following items must be demonstrated in lab on the day of the deadline:

- A working program to solve Clique using the given program that solves Subgraph Isomorphism, as specified — be prepared to show your source code.

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `compile.sh` — A Bash script to compile your submission (even if it does nothing), as specified.
- `run.sh` — A Bash script to run your submission, as specified.

The following files are optional:

- `*.py`, `*.java`, `*.clj`, `*.kt`, `*.js`, `*.sh` — The Python, Java, Clojure, Kotlin, Node.js, or Bash source code files of a working program to solve 3-SAT using the given program that solves Subgraph Isomorphism, as specified.

Any files other than these will be ignored.

Additionally:

- There will be two test runs, one after 8pm the date before the due date and one at the end of lab (10am) on the due date. A feedback containing only what your program scores (on the same test cases that will be used to grade it). I may be able to give you an idea of what your program is failing on (if you have made an effort to test it), so please take advantage of these runs.
- Late submissions made within 24 hours of the deadline receive a .7 multiplier. If you decide to make a late submission, please notify me by sending an email to vnguy143@calpoly.eduwith both the subject and the body in the format: "CSC349,late,asgn<i>", i being the assignment number.

[3] That is, the standard bitwise operators.

[4] There are, in fact, 6 cliques on $k = 3$ vertices in the example above. However, this is due to the duplicate literals in the latter clauses, and all of the cliques yield the same satisfying assignment.