Assignment 7 — Reductions

Due: December 4th

A reduction uses an algorithm for a problem B to solve an instance of a problem A. In doing so, it demonstrates that B is at least as hard as A — if a faster algorithm existed for B, then it could also solve A via the reduction, thereby making A exactly as hard[1] as B.

Deliverables:

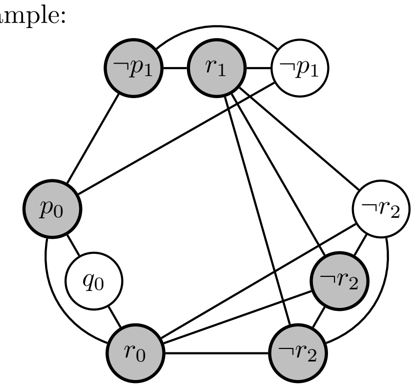
GitHub Classroom: `https://classroom.github.com/a/MJej6mih`

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

Optional Files: `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs`

Part 1: The Vertex Cover Problem

Given a graph $G = (V, E)$, a *vertex cover* is a subset of vertices $S \subseteq V$ such that every edge in E is incident to at least one vertex in S. For example:



Stated as a decision problem, the (Decision) Vertex Cover Problem asks for a vertex cover of cardinality k. In the above graph, $\{p_0, r_0, r_1, \neg p_1, \neg r_2, \neg r_2\}$ is a vertex cover, and proof that there exists a vertex cover of cardinality 6. There also exist vertex covers of cardinality ≥ 7, but none of cardinality ≤ 5.

The given `vertex_cover.py` implements a naïve, brute force algorithm[2] to solve this problem. It accepts as a command line argument the name of a file containing a threshold and an edge list:

```
>$ python3 vertex_cover.py tmp1.txt
Vertex cover on 6 vertices:
p00, r02, r11, ~p10, ~r20, ~r21
>$ echo $?
0
```

...if the graph contains a vertex cover of the desired cardinality, `vertex_cover.py` prints the covering vertices and terminates with exit status `0`. If not, it terminates with exit status `1`:

```
>$ python3 vertex_cover.py tmp3.txt
No vertex cover on 4 vertices.
>$ echo $?
1
```

[1]Assuming that the complexity of the reduction itself was negligible compared to that of the algorithm for B.
[2]It turns out that brute force, in this case, is also the best known approach.

Alternatively, both `vertex_cover` and the associated `graph` can be imported as Python[3] modules:

```
import graph, vertex_cover as vc

with open("tmp1.txt", "r") as graph_file:
    k = int(graph_file.readline())
    G = graph.read_graph(graph_file)
    S = vc.vertex_cover(G, k)
```

…if the given graph contains a vertex cover of the desired cardinality, `vertex_cover.vertex_cover` returns a tuple of the covering vertices. If not, it returns `None`.

Part 2: The 3-Satisfiability Problem

The 3-Satisfiability Problem, or "3-SAT", is known to be $\mathcal{NP}$-Hard. It asks whether there exists an assignment of truth values to propositional variables that satisfies a given proposition in conjunctive normal form with exactly 3 literals per clause. For example, the proposition $(p \vee q \vee r) \wedge (\neg p \vee r \vee \neg p) \wedge (\neg r \vee \neg r \vee \neg r)$ is satisfied by the assignments $p \equiv F$, $q \equiv T$, and $r \equiv F$:

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***

An instance of 3-SAT can be transformed into an equivalent instance of Vertex Cover as follows:

- A vertex p_i represents a literal p in clause i.
- Two vertices p_i and p_j are connected by an edge if and only if $i = j$ or $p_i \equiv \neg p_j$: if their literals are in the same clause or are contradictory.

For example, the graph in Part 1 corresponds to the above proposition, and the vertices $\{\neg p_1, q_0, \neg r_2\}$ are its satisfying assignment. Given a proposition with k clauses, the resulting graph $G = (V, E)$ contains a vertex cover S on $|V| - k$ vertices if and only if $V - S$ represents a satisfying assignment.

In your programming language of choice (see Assignment 1), design and implement an algorithm that determines whether or not such a proposition is satisfiable. You *must* make use of the given `vertex_cover` implementation[4], and the rest of your algorithm *must* have polynomial complexity[5].

Thus, you have demonstrated that an algorithm for the Vertex Cover Problem can solve the 3-Satisfiability Problem. Therefore, the Vertex Cover Problem is $\mathcal{NP}$-Hard.

Each input will consist of a proposition, where '`~`' indicates negation, '`&`', conjunction, and '`|`', disjunction. You may assume that the proposition will be well-formed and parenthesized, with single spaces between all literals and operators. You may further assume that all variables will be lowercase English letters.

[3]You are not required to use Python for this assignment, but you *must* make use of the given Python code, unmodified. If you choose to use your own Python graph implementation, it cannot be named "`graph.py`".

[4]In languages other than Python, the easiest way to pass information between your program and the given program is to split yours into two, then invoke the three programs in succession within your `run.sh` script, redirecting intermediate output to files.

[5]Furthermore, if you choose to implement a reduction other than the one described above, you may not create graphs with more vertices than there are literals, in order to allow for practical testing.

For example, the above proposition could 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` a satisfying assignment (if one exists) according to the following format:

- Each variable must appear as a literal: those assigned a value of T must appear unaltered; those assigned a value of F must appear negated.
- The satisfying assignment must appear as a single comma-separated line of literals, sorted in alphabetical order by their variables.

For example:

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

You may further assume that there will exist exactly zero or one satisfying assignments, and that no satisfying assignment will involve one literal that satisfies multiple clauses, such that output as described above will be unique. Your program will be tested using `diff`, so its printed output must match *exactly*.

Part 3: Submission

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

- `report.pdf` — Pseudocode for an algorithm to determine satisfiability by using a given algorithm to find vertex covers, along with a corresponding reduction diagram from 3-SAT to Vertex Cover.
- `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:

- `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs` — The source code of a working program to solve the 3-Satisfiability Problem, as specified.

Any files other than these will be ignored.