Complexity Classes $\mathcal{P}$ and $\mathcal{NP}$

Michael Levet

December 25, 2014

Introduction
In this tutorial, I will introduce the complexity classes $\mathcal{P}$ and $\mathcal{NP}$. Complexity theory is the study of how efficiently a problem can be solved. While there are many more complexity classes beyond $\mathcal{P}$ and $\mathcal{NP}$, these are the two most commonly studied. This tutorial will address three points of interest: defining the classes $\mathcal{P}$ and $\mathcal{NP}$, formalizing the notion of what constitutes a hard problem, and proving a problem is in one of these classes or hard for one of these classes.

While a deep understanding of computability theory is not necessary, some familiarity with the concept of a Turing Machine is useful, as the complexity classes are defined in terms of Turing Machines. This tutorial provides more information on how Turing Machines work, from a practical standpoint. I will gloss over details of Turing Machines as appropriate in this tutorial, so one will be able to follow along without much background on Turing Machines.

Defining the Classes $\mathcal{P}$ and $\mathcal{NP}$
If you are familiar with the problem Does $\mathcal{P} = \mathcal{NP}$?, then it may be motivating to study this topic for your chance to win $1 Million. For those not familiar with this problem, we will explore what it means and seek to better understand it in this section. Let's start with formal definitions of the classes $\mathcal{P}$ and $\mathcal{NP}$.

Definition 1. *$\mathcal{P}$: A language L is in the complexity class $\mathcal{P}$ if and only if L can be decided by a deterministic Turing Machine in polynomial time on the size of the input.*

Definition 2. *$\mathcal{NP}$: A language L is in the complexity class $\mathcal{NP}$ if and only if for any string $\omega \in L$, there exists a non-deterministic Turing Machine that can verify $\omega \in L$ in polynomial time on $|\omega|$. Note that there will be no false-positives. That is, if $\omega \notin L$, the Turing Machine will never accept ω.*

These definitions are somewhat clunky and may not make a lot of sense right now. So let's unpack them and develop some intuition. The Church-Turing Thesis conjectures that there is no model of computation more powerful than the Turing Machine. So we equate the concept of an algorithm with the formalization of a Turing Machine. That is, determining if a Turing Machine exists can be answered by providing an algorithm.

Notice that in both $\mathcal{P}$ and $\mathcal{NP}$, the problem is string membership in a language (ie., is "word" in the English language?). That is, we are asking the question: is $\omega \in L$? This is a yes or no question. So our Turing Machine (read- algorithm) takes the string as an input. If the Turing Machine halts on ω and accepts it, then $\omega \in L$. If the Turing Machine halts on ω and rejects it, we conclude $\omega \notin L$. There is a third option as well: the Turing Machine may not halt on ω. This will only happen if $\omega \notin L$, but we cannot tell when an arbitrary Turing Machine is not halting. So this third option prohibits us from making a conclusion.

In the context of the Turing Machine model, languages are the problems we seek to solve. So for example, consider the problem of determining whether a graph has a Hamiltonian circuit. The corresponding language would then be: $L_{HC} = \{G : G \text{ is a graph that has a Hamiltonian Circuit}\}$. We would then ask if a graph H is in L_{HC}. In other words, does H have a Hamiltonian Circuit?

So $\mathcal{P}$ and $\mathcal{NP}$ are sets of decision problems. The problems in $\mathcal{P}$ can all be answered correctly in polynomial time, while the problems in $\mathcal{NP}$ can only have correct answers of "yes" verified in polynomial time. So if $L_{HC} \in NP$, and a graph $G \in L$, then an algorithm can verify that G has a Hamiltonian Circuit provided said algorithm is given both G and the Hamiltonian circuit in G.

The Hamiltonian Circuit problem is actually in the class $\mathcal{NP}$. Some other problems in $\mathcal{NP}$ include the Knapsack problem, the Minimum Spanning Tree problem, and the Shortest Path problem. Incidentally, the Minimum Spanning Tree and Shortest Path problems are also in $\mathcal{P}$, as we have polynomial time algorithms to solve them. From the definition of $\mathcal{P}$ (and the examples provided), we can see that $\mathcal{P} \subset \mathcal{NP}$. We don't know if $\mathcal{NP} \subset \mathcal{P}$, though. So while we don't know if $\mathcal{P} = \mathcal{NP}$, it is widely believed that $\mathcal{P} \neq \mathcal{NP}$. The importance of the $\mathcal{P} = \mathcal{NP}$ problem will become clearer in the next section, where the concept of a computationally hard problem is formalized.

(N)P-Hard and (N)P-Complete Problems

In this section, the notion of a computationally hard problem will be formalized, providing the context for why the $\mathcal{P} = \mathcal{NP}$ problem is important.

Consider two problems H, J. These need not necessarily be decision problems. Problem J is harder than problem H if any algorithm to solve J can also be used to solve H. We denote this by saying $H \leq J$. Notice that this defines an ordering on problems (called a partial ordering). This also indicates that H can be reduced to J. When looking at a particular complexity class, such a reduction usually has constraints such as being computed in polynomial time or a logarithmic amount of space.

Such a reduction would be a function $f : H \to J$, which transforms each instance of H into an instance of J. The transformation procedure also must satisfy the constraints imposed by the complexity class.

Now let's define the hardest problems in $\mathcal{P}$ and $\mathcal{NP}$.

Definition 3. *$\mathcal{NP}$-Hard A problem Q is $\mathcal{NP}$-Hard if every $X \in \mathcal{NP}$ is reducible to Q in polynomial time. We denote this as $X \leq_p Q$. That is, given such a transformation $f : X \to Q$, for every $x \in X$, $f(x)$ is computable in polynomial time with respect to the size of the input.*

Similarly, we define P-Hard problems:

Definition 4. *$\mathcal{P}$-Hard A problem Q is $\mathcal{P}$-Hard if for every $X \in \mathcal{P}$, X is reducible to Q in log-space. We denote this as $X \leq_l Q$. That is, given such a transformation $f : X \to Q$, for every $x \in X$, $f(x)$ is computable using a logarithmic amount of space with respect to the size of the input.*

Notice that $\mathcal{P}$-Hard and $\mathcal{NP}$-Hard problems need not necessarily be in the classes $\mathcal{P}$ and $\mathcal{NP}$. A good example is the Traveling Salesman optimization problem (TSP_{opt}), which is $\mathcal{NP}$-Hard. As the class $\mathcal{NP}$ consists of only decision problems, $TSP_{opt} \notin \mathcal{NP}$.

There are $\mathcal{P}$-Hard and $\mathcal{NP}$-Hard problems in the classes $\mathcal{P}$ and $\mathcal{NP}$ respectively, though. We call such problems $\mathcal{P}$-Complete and $\mathcal{NP}$-Complete. So an $\mathcal{NP}$-Complete problem has three characteristics:

- It is a decision problem.
- A correct answer of "YES" can be verified in polynomial time, given a certificate.
- All other problems in $\mathcal{NP}$ are reducible to our $\mathcal{NP}$-Complete problem in polynomial time.

It follows that all $\mathcal{NP}$-Complete problems are reducible to each other in polynomial time. So all $\mathcal{NP}$-Complete Problems are equally as hard.

Similarly, $\mathcal{P}$-Complete problems have three characteristics:

- It is a decision problem.
- It can be solved in polynomial time.
- All other problems in $\mathcal{P}$ are reducible to our $\mathcal{P}$-Complete problem in log-space.

So what is the relation between a hard problem for a complexity class and computational intractability? A problem is efficiently computable if it can be solved in polynomial time. That is, the problem is in $\mathcal{P}$. Problems that are $\mathcal{NP}$-Complete are believed to be computationally intractable, but we do not know this for certain. In other words, it is unknown if an $\mathcal{NP}$-Complete problem is in $\mathcal{P}$. If this were the case, then $\mathcal{P} = \mathcal{NP}$ and any problem in $\mathcal{NP}$ could be solved in polynomial time. However, this is not believed to be the case, so $\mathcal{NP}$-Complete and $\mathcal{NP}$-Hard problems are generally believed to be computationally intractable.

Writing Complexity Proofs- $\mathcal{NP}$
In this section, we look at writing $\mathcal{NP}$-Completeness proofs and $\mathcal{NP}$-Hardness proofs. These proofs can be a bit lengthy and hard to follow. So the goal is to really break down the proof structure to understand what needs to be shown and how to show each part.

Recall that the definition of $\mathcal{NP}$-Complete states that an $\mathcal{NP}$-Complete problem is in $\mathcal{NP}$ and is $\mathcal{NP}$-Hard. To show that a problem is in $\mathcal{NP}$, it suffices to exhibit a polynomial time verification algorithm. That is, the algorithm accepts some certificate and checks it against the given instance. If the certificate is valid, then the algorithm runs in polynomial time and recognizes the certificate as valid. Looking back at the Hamiltonian Circuit problem, an example of a certificate would be a Hamiltonian circuit contained in the graph. It would then be sufficient to construct an algorithm to verify that the provided certificate was indeed a Hamiltonian cycle of the graph.

The second part of an $\mathcal{NP}$-Completeness proof is to prove that the problem is $\mathcal{NP}$-Hard. That is, it is necessary to show that every problem in $\mathcal{NP}$ is reducible in polynomial time to the $\mathcal{NP}$-Hard problem. This is done in one of two ways. The first is by explicitly showing the existence of a reduction for every problem in $\mathcal{NP}$ to the problem in question. The second way is by taking an existing $\mathcal{NP}$-Complete or $\mathcal{NP}$-Hard problem and providing a polynomial time reduction to the problem in question. The composition of polynomial time reductions is computable in polynomial time, so this is a valid and much easier approach.

In order to be able to use a reduction, it is necessary first to have an $\mathcal{NP}$-Complete problem. Stephen Cook proved the Boolean Satisfiability problem (SAT) to be $\mathcal{NP}$-Complete, and Richard Karp used this result to prove 21 other problems as $\mathcal{NP}$-Complete. The proof of SAT being $\mathcal{NP}$-Complete is quite intense and will not be covered in this tutorial. Instead, we demonstrate the reduction technique.

We begin with some conventions. Recall that $\mathcal{NP}$-Complete problems are decision problems. Each problem has an instance, and a decision question. Think of the instance like a function parameter

in computer programming. The problem name is also written in all capital letters. Let's look at the HAMILTONIAN CIRCUIT problem as an example:

Definition 5. *HAMILTONIAN CIRCUIT*

- ***INSTANCE*** *Let $G(V, E)$ be a simple graph.*
- ***DECISION*** *Does there exist a Hamiltonian Circuit in G?*

We note that HAMILTONIAN CIRCUIT is $\mathcal{NP}$-Complete, and we will use this to prove that the HAMILTONIAN PATH problem is $\mathcal{NP}$-Complete. The HAMILTONIAN PATH problem is defined similarly as HAMILTONIAN CIRCUIT:

Definition 6. *HAMILTONIAN PATH*

- ***INSTANCE*** *Let $G(V, E)$ be a simple graph.*
- ***DECISION*** *Does there exist a Hamiltonian Path in G?*

For convenience, I will abbreviate HAMILTONIAN CIRCUIT as HC, and HAMILTONIAN PATH as HP.

Theorem 1. *HAMILTONIAN PATH is $\mathcal{NP}$-Complete*

Proof. In order to show that HP is $\mathcal{NP}$-Complete, it will be shown that HP $\in \mathcal{NP}$, and HC $\leq_p$ HP.

Claim 1. *HP is in $\mathcal{NP}$.*

In order to show that HP is in $\mathcal{NP}$, a polynomial time verification algorithm will be exhibited. Let G be the graph associated with the instance of HP, and let P be the path in G to serve as the certificate. That is, let P be the Hamiltonian Path in G. The algorithm must verify that P is in fact a Hamiltonian Path. To check that P is a Hamiltonian Path, it suffices to check that it is connected and all but two vertices, v_1 and v_n, have degree 2. The vertices v_1 and v_n must have degree 1. This can be checked using a breadth-first traversal of P, which can be done in polynomial time. And so HP is in $\mathcal{NP}$.

Remark: Showing that HC $\leq_p$ HP is more involved. We have three main parts of the proof: constructing the reduction, arguing that the reduction can be computed in polynomial time, and arguing the validity of the reduction. Constructing the reduction is done by transforming the instance of HC into an instance of HP. To prove validity, we argue that there is an answer of YES to an instance of HC if and only if there is an answer of YES to the corresponding instance of HP (via the transformation). Let's now construct the polynomial time reduction from HC to HP.

Claim 2. *HC is polynomial time reducible to HP.*

Let G be the graph associated with the instance of HC. That is, G is a graph with a Hamiltonian Circuit. From G, we construct a graph G' such that G has a Hamiltonian Circuit if and only if G' has a Hamiltonian path. If G is a single vertex or edge, then $G = G'$ and we are done. Otherwise, let x be a vertex of G. Let x' be a copy of x, and let all edges incident to x be copied and incident to x'. That is, if $\{x, v\}$ is an edge in G, $\{x', v\}$ is in G'. Let G' contain all vertices and edges of G, as well as x' and all incident edges to x'. Finally, add new vertices v, v' and edges $\{v, x\}, \{v', x'\}$ to G'. As G is simple, x can have at most $|V(G)| - 1$ adjacencies, so this operation takes $\mathcal{O}(|V(G)|)$ time. Thus, the

reduction is polynomial in time.

The validity of the reduction will now be shown. This is done by showing that G has a Hamiltonian Circuit if and only if G' has a Hamiltonian Path. Suppose G has a Hamiltonian Circuit. We show that G' has a Hamiltonian Path. Start at v on G' and move to x. Then follow the Hamiltonian Circuit from G on G' (this exists, as G is a subgraph of G'). Rather than returning to x, though, move to x'. Then from x', move to v'. This completes the Hamiltonian Path on G'. Conversely, suppose G' has a Hamiltonian Path. Removing v, v', and x' renders the Hamiltonian Path as a Hamiltonian Circuit. Hence, G has a Hamiltonian Circuit if and only if G' has a Hamiltonian Path. And so HC $\leq_p$ HP, and HP is $\mathcal{NP}$-Complete. QED.

□

So when we have two decision problems, the reduction is quite straight-forward. I offer a second proof reducing HC to TSP_{OPT}, the optimization version of the Traveling Salesman Problem. We define TSP_{OPT} as follows.

Definition 7. *TSP_{OPT}*

- ***INSTANCE*** *Let $G(V, E)$ be a simple graph.*
- ***PROBLEM*** *Find the minimum cost Hamiltonian Circuit in G.*

Notice that TSP_{OPT} is not a decision problem, and so it is not $\mathcal{NP}$. To show TSP_{OPT} is $\mathcal{NP}$-Hard, it suffices to show a polynomial time reduction from an existing $\mathcal{NP}$-Complete problem to TSP_{OPT}.

Claim 3. *TSP_{OPT} is $\mathcal{NP}$-Hard.*

Proof. Let $G(V, E)$ be a simple graph with at least 3 vertices, and that G is associated with an instance of HC. So G has a Hamiltonian Circuit. Let $n = |V(G)|$, and construct a weighted complete graph on n vertices K_n, as follows. The vertices of K_n will be those of G. For each $\{v_i, v_j\} \in E(G)$, the edge $\{v_i, v_j\}$ is weighted 1 in K_n. For each $\{v_i, v_j\} \notin E(G)$, $\{v_i, v_j\}$ is weighted 2 in K_n. This construction examines $\binom{n}{2}$ edges (where $\binom{n}{2}$ represents the binomial coefficient), examining each edge one at a time. So the reduction is polynomial in time. As the minimum edge weight is 1 and n edges are required for a Hamiltonian Circuit, the optimal solution to the TSP_{OPT} problem is at least n.

It suffices to show that G has a Hamiltonian Circuit if and only if the optimal solution of the TSP tour in G' is n. Suppose G has a Hamiltonian circuit. By construction, each edge in G is contained in the weighted K_n with each edge weighted 1. It suffices to trace along the Hamiltonian Circuit from G in K_n, providing a TSP tour of weight n, which is the optimal solution. Conversely, suppose that the optimal TSP tour has weight n. As the tour is a cycle on n vertices, there are n edges. The minimum edge weight on the K_n is 1, and so each edge in the tour must be weighted 1. By construction, each such edge is in G. Thus, tracing the TSP tour produces a Hamiltonian Circuit in G. Thus, HC $\leq_p TSP_{OPT}$. And so TSP_{OPT} is $\mathcal{NP}$-Hard. QED. □

Writing Complexity Proofs- $\mathcal{P}$

The purpose of this section is to better understand how to construct complexity proofs related to the class $\mathcal{P}$. Recall that $\mathcal{P}$ is the class of decision problems which can be correctly answered in polynomial time. We examine two proofs in this section. The first is to demonstrate a reduction showing a problem is in $\mathcal{P}$. The second proof demonstrates the procedure for showing a problem is $\mathcal{P}$-Complete.

Let's first examine how to show a problem is in $\mathcal{P}$. There are two approaches to accomplish this. The first is to explicitly demonstrate a polynomial time algorithm to decide the problem. The other approach is to provide a polynomial time reduction from the given decision problem to an existing problem in $\mathcal{P}$. I will not focus on algorithm correctness, as this topic is too broad for this tutorial. Instead, I will exhibit a polynomial time reduction between two problems in $\mathcal{P}$.

The problem we will examine is 2-SAT, which is defined as follows.

Definition 8. *2-SAT*

- ***INSTANCE****: Let $x \in \{0,1\}^n$ and consider conjunction C of clauses chosen from the components of x and their negations. Each clause is restricted to contain exactly two components of x (or their negations).*
- ***DECISION****: Is there a configuration $x \in \{0,1\}^n$ such that the expression $C(x)$ evaluates to true?*

Note: Observe that 2-SAT is a subset of the general SAT problem, which is $\mathcal{NP}$-Complete. So there do exist subsets of $\mathcal{NP}$-Complete problems that are decidable in polynomial time (ie., are in $\mathcal{P}$). This is important to note and remember when working with classes of $\mathcal{NP}$-Complete problems.
In order to prove that 2-SAT is in P, we reduce it to the STRONGLY CONNECTED COMPONENT (SCC) problem defined as follows:

Definition 9. *STRONGLY CONNECTED COMPONENT (SCC)*

- ***INSTANCE*** *Let $G(V, A)$ be a directed graph.*
- ***DECISION*** *Do there exist $i, j \in V(G)$ such that there is a directed $i \to j$ path and a directed $j \to i$ path?*

This problem can be decided in polynomial time using Tarjan's algorithm, and so SCC is in $\mathcal{P}$.

Claim 4. *2-SAT is in $\mathcal{P}$.*

Proof. The proof is by reduction to SCC. We begin by constructing the implication graph G, which is a directed graph. The vertices of G are the components of x and their negations, yielding $2n$ vertices in total. For each clause in C $(x_i \vee x_j)$, add directed edges $(\neg x_i, x_j), (\neg x_j, x_i)$. (So for example, if a clause contained $(\neg x_2 \vee x_3)$, the edges added to G would be $(x_2, x_3), (\neg x_3, \neg x_2)$.) The reduction to SCC looks at the implication graph to determine if there is a component x_i such that there is a directed path x_i to $\neg x_i$, and a directed path $\neg x_i to x_i$. Notice that there are at most $\binom{n}{2}$ clauses to examine and so at most $\binom{2n}{2}$ edges to add to G, so the reduction is polynomial in time.

So we need to prove a couple facts:

- If there exists an $a \to b$ directed path in G, then there exists a directed $\neg b \to \neg a$ path in G. We will use this fact to justify the existence of a strongly connected component if there is a directed $x_i \to \neg x_i$.
- C is satisfiable if and only if there is no strongly connected component in G containing both a variable and its negation. This will substantiate the validity of the reduction.

Claim 1: If there exists a directed $a \to b$ path in G, then there exists a directed $\neg b \to \neg a$ path in G.

Suppose there exists an $a \to b$ directed path in G. By construction, for each edge $(c, d) \in G$, there exists an edge (d, c). So given the $a \to b$ directed path: $a \to p_1 \to ... \to p_k \to b$, there exist directed edges in G: $(\neg b, \neg p_k), ..., (\neg p_1, \neg a)$, yielding a directed $\neg b \to \neg a$ path, as claimed.

Claim 2: C is satisfiable if and only if there is no strongly connected component in G.

It will first be shown that if C is satisfiable, then there is no strongly connected component in G. This will be done by contradiction. Let x be a satisfying configuration of C, and let x_i such that there is a strongly connected component including x_i and $\neg x_i$. Let $x_i \to p_1 \to ... \to p_n \to \neg x_i$ be the directed $x_i \to \neg x_i$ path in G. Suppose first $x_i = TRUE$. By construction, the edges $(\neg x_i, p_1), (\neg p_i, p_{i+1})$ for each $i = 1, ..., n-1$; and $(\neg p_n, \neg x_n)$ are in G. And so for each $i = 1, ..., n$, p_i must be true to satisfy the corresponding clause in C. However, since $\neg x_i$ is false, p_n must be false to satisfy $(\neg p_n, \neg x_n)$, a contradiction. By similar analysis, x_i cannot be FALSE either. And so C is unsatisfiable. Thus, if C is satisfiable, there is no strongly connected component containing both x_i and $\neg x_i$.

Now suppose there is no strongly connected component in G. It will be shown that C is satisfiable by contradiction. Suppose there are no $x_i \in x$ such that there exists a directed $x_i \to \neg x_i$ path. For each unmarked vertex $v \in V(G)$ such that no $v \to \neg v$ path exists, mark v as TRUE. Now mark each neighbor of v as TRUE, and the negations of each marked variable as FALSE. Repeat this process until all vertices have been marked. By finiteness of the graph, this process terminates. Since there are no strongly connected components in G, all vertices will be marked. As C is unsatisfiable, let $i, j \in \{1, ..., n\}$ such $i \neq j$ and that there exists directed $x_i \to x_j$ and $x_i \to \neg x_j$ paths. So x_i implies both x_j and $\neg x_j$, which is a fallacy. By construction of G, there must exist a $\neg x_j \to \neg x_i$ directed path in G, which implies that G has a strongly connected component. However, G has no strongly connected component by assumption, so a contradiction has been reached.

Thus, we conclude C is satisfiable if and only if there exists no strongly connected component in G, proving Claim 2.

As the polynomial time reduction to SCC is valid, it follows that 2-SAT is in $\mathcal{P}$. QED. □

The last problem we will explore is the MONOTONE CIRCUIT VALUE (MCV) problem. MCV is $\mathcal{P}$-Complete. To prove it takes a bit of work, and a few prior results are needed. In fact, many $\mathcal{P}$-Completeness proofs are either ciruit-related or rely on reductions from circuit-related problems. We start with a few facts.

Fact 1: All Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$ can be computed using the operations AND, OR, and NOT.

Fact 2: All Boolean functions can be computed using operations equivalent to AND, OR, and NOT.

Fact 3: All logical circuits can be written as straight-line programs. That is, we have only variable assignments and arithmetic being performed. There are no loops, conditionals, or control structures of any kind.

Now let's define MCV:

Definition 10. *MONOTONE CIRCUIT VALUE (MCV)*

- ***INSTANCE**: Let C be a circuit over the monotone basis $\{AND, OR\}$ (only AND, OR operations are used), and let $x \in \{0,1\}^n$ be a fixed configuration.*

  ***DECISION**: Does x satisfy C?*

Notice that MCV is different from SAT. In SAT, we are given a Boolean expression and are trying to determine if there is a satisfying configuration. In MCV, we are given the circuit and trying to determine if a particular input configuration will satisfy it. In other words, MCV deals with a "here, try this one" for a single input configuration.

An example instance of MCV is $x = (0, 1)$ with $C = x_1 \wedge x_2$. Notice that x takes on specific values here.

In order to prove MCV is $\mathcal{P}$-Complete, we reduce from CIRCUIT VALUE (CV), which is $\mathcal{P}$-Complete. The difference between MCV and CV is that CV permits the NOT operation. So CV is defined as follows:

Definition 11. *CIRCUIT VALUE (CV)*

- ***INSTANCE**: Let C be a circuit over the monotone basis $\{AND, OR, NOT\}$ (only AND, OR, and NOT operations are used), and let $x \in \{0, 1\}^n$ be a fixed configuration.*

  ***DECISION**: Does x satisfy C?*

Since MCV is a subset of CV (we take circuits without the NOT operation, which are also instances of CV), MCV is in $\mathcal{P}$. To show MCV is $\mathcal{P}$-Hard, we reduce CV $\leq_l$ MCV. That is, for each instance of CV, a corresponding instance of MCV will be constructed. This is difficult though, as it is necessary to get rid of the NOT operations from CV. We do this by flushing the NOT operations down each layer of the circuit using DeMorgan's Law.

We then construct Dual-Rail Logic (DRL) circuits, where each variable x_i from x in the CV instance is represented as $(x_i, \neg x_i)$ in the MCV instance. So any negations we may want are constructed up-front, so the NOT operation becomes unnecessary. The DRL operations are defined as follows:

- DRL-AND: $(x, \neg x) \wedge (y, \neg y) = (x \wedge y, \neg(x \wedge y)) = (x \wedge y, \neg x \vee \neg y)$
- DRL-OR: $(x, \neg x) \vee (y, \neg y) = (x \vee y, \neg(x \vee y)) = (x \vee y, \neg x \wedge \neg y)$
- DRL-NOT: $\neg(x, \neg x)$ is given just by twisting the wires, sending x and $\neg x$ in opposite directions.

Since the NOT operation is given upfront in the variable declarations, the DRL operations are all realizable over the monotone basis of {AND, OR}. DRL is also equally as powerful as the basis {AND, OR, NOT}. So any Boolean function can be computed with DRL.

Now let's prove that MCV is $\mathcal{P}$-Complete.

Claim 5. *MCV is $\mathcal{P}$-Complete.*

Proof. In order to show MCV is $\mathcal{P}$-Complete, we show that MCV is in $\mathcal{P}$ and every problem in $\mathcal{P}$ is log-space reducible to MCV (ie., MCV is $\mathcal{P}$-Hard). As MCV is a subset of CV and CV is in $\mathcal{P}$, it follows that MCV is in $\mathcal{P}$.

To show MCV is $\mathcal{P}$-Hard, we show CV $\leq_l$ MCV. Let (x, C) be an instance of CV where x is the input sequence and C is the circuit over the basis {AND, OR, NOT}. We construct C' over the monotone basis {AND, OR} from C, by rewriting C as a dual-rail circuit. Let $P(C)$ be the straight-line program representing C. Let P' be the straight-line program used to construct C'. For each line n in $P(C)$, let this instruction be line $2n$ in P'. Line $2n + 1$ in P' corresponds to the negation of line n in $P(C)$.

The NOT operation in $P(C)$ is realized in P' by twisting the wires. That is, the step $(2k = \neg 2i)$ is realized by the steps $(2k = 2i + 1)$ and $(2k + 1 = 2i)$. The AND operation in $P(C)$ $(2k = 2i \wedge 2j)$ is replaced by the steps $(2k = 2i \wedge 2j)$ and $(2k + 1 = (2i + 1) \vee (2j + 1))$. Finally, the OR operation $(2k = 2i \vee 2j)$ is realized by $(2k = 2i \vee 2j)$ and $(2k + 1 = (2i + 1) \wedge (2j + 1))$. And so $P(C) = P'$ for all inputs. So $P(C) = 1$ if and only if $P' = 1$, and $P(C) = 0$ if and only if $P' = 0$. So the reduction is valid.

It now suffices to argue the reduction takes a logarithmic amount of space. Generating P' from $P(C)$ can be done using a counter variable. So for each step i in $P(C)$, we perform operations at lines $2i$ and $2i + 1$ in P'. So if there are n steps in $P(C)$, we need $log_2(\lceil 2n + 1 \rceil)$ bits, which grows asymptotically with $c(log_2(2) + log_2(n)) = c(1 + log_2(n))$ for some integer constant $c > 1$. So the amount of space required is $\mathcal{O}(log(n))$. And so we conclude that MCV is $\mathcal{P}$-Complete. QED. $\square$