COMP 314 Class 22: Polynomial time reductions

1 Definition of polytime reductions

Recall that earlier in the semester, we defined reductions as follows:

P reduces to Q	means (informally)	If you can solve Q, you can solve P

The above definition ignores efficiency, and considers only computability. For that reason, this type of reduction is sometimes called a *Turing reduction*. We are now going to give a similar definition that takes efficiency into account:

P polyreduces to Q	means (informally)	If you can solve Q in polynomial time, you can solve P in polynomial time

We will use the phrases "polynomial time reduction", "polytime reduction", and "polyreduction" interchangeably. The same goes for "reduce in polynomial time" and "polyreduce".

Recall that the definition of Turing reductions is rather flexible. If we are asked to prove that P reduces to Q, we need to write a program `P.py` that solves P. At any point in the program, we can import and use a function `Q` from `Q.py`, which is assumed to solve problem Q. (That is, `Q` terminates with correct output on any input.) We are free to invoke the function `Q` as often as we like, even inside a loop that might be executed exponentially often. And we can pass parameters to `Q` which might be exponentially longer than the original input to `P.py`.

Once we begin considering efficiency, and switch our attention to polytime reductions, there is much less flexibility in the appropriate ways to use function `Q`. This time, we can certainly import and use `Q` from `Q.py`. And we can assume that `Q` runs in polynomial time, as a function of the length of its input parameters. But beyond this, we would have to be rather careful how we use `Q`. If we invoke it too often, or send it parameters that are too long, the total running time of `P.py` might not be polynomial anymore.

For this reason, the formal definition of polytime reductions is much more restrictive. We insist that `Q` is invoked exactly once (with parameters whose length is polynomial, as a function of the length of the original input to `P.py`). To make this work, we need to transform the original input (an instance of problem P) into some new input (an instance of problem Q). We then run `Q` once on this new input, producing some output (a solution to Q). Finally, the output from `Q` must be transformed back into a solution to P. Both transformations (P-input to Q-input, and Q-output to P-output) must also run in polynomial time.

Figure 1 shows an example of this in action. It uses the problems PARTITION and SUBSETSUM, and shows how we can polyreduce PARTITION to SUBSETSUM. PARTITION is a problem that we haven't seen before. The input is a set of integers, and the objective is to partition the set into two parts that have equal weight. The output consists of the two partitioned sets separated by a semicolon (or "`no`", if no partition is possible). For example, on input "`3 5 8 6`", one of the possible outputs is "`5 6; 3 8`", since $5 + 6 = 3 + 8$.

Recall that SUBSETSUM also takes a set of integers as input, but it receives one additional integer which is the target value. The output is a subset that sums to the target value, or "`no`" if this is impossible.

So, how do we polyreduce PARTITION to SUBSETSUM? The idea is simple: to create the desired partition, just set the target value of SUBSETSUM to be half of the sum of the inputs. More precisely, if the inputs to PARTITION are the weights $w_1, w_2, \ldots, w_m$, we first calculate target $T = \sum_{k=1}^{m} w_k/2$, then send the input $w_1, w_2, \ldots, w_m, T$ to SUBSETSUM. Once the output to SUBSETSUM is obtained, some trivial reformatting produces the correct solution to the original PARTITION instance.

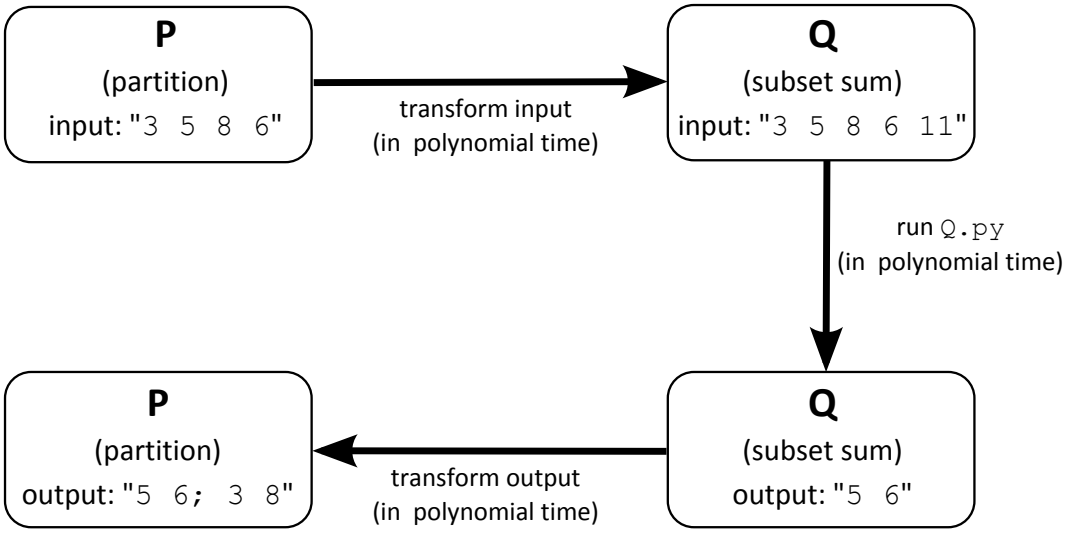


Figure 1: **An example of a polynomial time reduction.** Problem P (PARTITION) has its input transformed into an equivalent input for problem Q (SUBSETSUM). Problem Q is solved using an existing program, and its output is converted back into an equivalent output for P. Each step runs in polynomial time. More precisely, the transformations of input and output must run in polynomial time, and the program `Q.py` is assumed to exist and to run in polynomial time.

Let's examine the details of the concrete example in Figure 1. The PARTITION input is "`3 5 8 6`", which gets transformed to the SUBSETSUM input "`3 5 8 6 11`". This SUBSETSUM instance is solved by running the polynomial-time program `Q.py`. (Of course,

`Q.py` need not really exist. It is *assumed* to exist for the purposes of proving the polytime reduction.) There are several possible correct outputs, but the program happens to produce the output "`5 6`". Combining this with the original PARTITION input, we see that the other partition must consist of 3 and 8. So the SUBSETSUM output is easily transformed, in polynomial time, to "`5 6; 3 8`", which is the correct solution for the original PARTITION instance.

With this background, we are ready for a formal definition of polynomial time reductions:

Definition (polyreduction): Problem P polyreduces to problem Q if there exist:

- a polytime program `C` (for Convert) that converts instances of P into instances of Q, and
- a polytime program `R` (for Revert) that converts solutions of Q into solutions of P (possibly also by inspecting the original instance of P)

such that, for any input string I,

$$\texttt{R}(\texttt{Q}(\texttt{C}(I)), I) \qquad (*)$$

is a solution to P on input I. A few remarks well help to explain the formula (*) above:

- `Q` represents a program that solves the problem Q. Program `Q` need not actually exist; it is assumed to exist for the purposes of the definition.
- Program `R` actually receives two inputs: the output of `Q` (e.g. In Figure 1 this is "`5 6`"), and the original input I (e.g "`3 5 8 6`"). The reason for this is that sometimes, we need to inspect P's original input before reconstructing P's output. Figure 1 is a good example of this: given only the output "`5 6`", it is impossible to reconstruct the PARTITION solution "`5 6; 3 8`", because of the silly reason that we don't know what other elements were in the original set. Once the original input ("`3 5 8 6`") is provided, the reconstruction becomes trivial.

2 The meaning of polynomial-time reductions

Recall that for a given Turing reduction, there are three main consequences:

- If P reduces to Q, then P is "easier" than Q. (Or, more accurately, P is "no harder" than Q.) This is sometimes written $P \leqslant Q$, or $P \leqslant_{\text{Turing}} Q$.
- If P reduces to Q, and Q is computable, then P is computable.
- If P reduces to Q, and P is uncomputable, then Q is uncomputable.

For polytime reductions, there are some similar consequences:

1. If P polyreduces to Q, then P can be solved "faster" than Q. (Or, more accurately, P can be solved "as fast as" Q, if we neglect any polynomial slowdown factor.) This is sometimes written $P \leqslant_{\text{Poly}} Q$.

2. If P polyreduces to Q, and Q is in Poly, then P is in Poly.

3. If P polyreduces to Q, and P requires exponential time, then Q requires exponential time.

These facts follow easily from the definition of polytime reductions. Fact 1 is just a restatement of Fact 2, so let's prove Fact 2. To solve P in polynomial time, we just follow the arrows in Figure 1. Each step takes only polynomial time, provided `Q.py` actually exists and runs in polynomial time. But the statement of Fact 2 guarantees that Q has a polytime solution, and this completes the proof of Fact 2.

Fact 3 is easiest to prove by contradiction. Suppose Q can be solved in subexponential time. Then by following the arrows in Figure 1, we have a way of solving P in subexponential time, and this contradicts the fact that P requires exponential time.

Sneak preview of the real reason for studying polytime reductions

Polynomial time reductions are interesting in their own right, but the most important reason for studying them is that they lie at the heart of an important topic called *NP-completeness*. Therefore, here's a sneak preview of the *real* reason we're studying polytime reductions. In the next few lectures, we will find out about two more types of problems: *NP-hard* problems, and *NP-complete* problems. For the moment, we can think of both "NP-hard" and "NP-complete" as meaning "very hard—probably requires exponential time". With this vocabulary, we can add some other consequences to the concept of polytime reductions:

- If P polyreduces to Q, and P is NP-complete, then Q is NP-hard.

3 Examples of polytime reductions using Hamiltonian circuits

The problems HamiltonianCircuit (also known as UndirectedHamiltonianCircuit) and DirectedHamiltonianCircuit provide some good examples of polynomial reductions. Figures 2 and 3 describe these problems in detail. The basic idea is, given a graph that could be directed or undirected, find a cycle that visits every vertex exactly once. Please study Figures 2 and 3 carefully before reading on.

Problem HamiltonianCircuit

- **Input:** A graph (specifically, an undirected, unweighted graph). The input format is just a list of the edges in the graph separated by semicolons. For example, the graph below would be input as "`v1 v2; v2 v3; v3 v1; v2 v4; v1 v4`".
- **Output:** A Hamiltonian circuit for the input graph if one exists, or "`no`" otherwise. A Hamiltonian circuit is a cycle that contains each vertex exactly once. For example, one possible output for the graph below is "`v1 v4 v2 v3`".

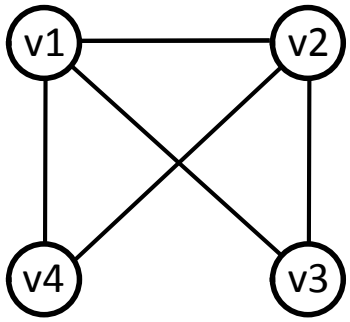


Figure 2: Description of the computational problem HamiltonianCircuit, also known as UndirectedHamiltonianCircuit

Problem DirectedHamiltonianCircuit

- **Input:** A directed graph. The input format is just a list of the directed edges in the graph separated by semicolons. For example, the graph below would be input as "`v1 v2; v2 v1; v3 v2; v1 v3; v2 v4; v4 v1`".
- **Output:** A directed Hamiltonian circuit for the input graph if one exists, or "`no`" otherwise. A directed Hamiltonian circuit is a directed cycle that contains each vertex exactly once. For example, one possible output for the graph below is "`v1 v3 v2 v4`".

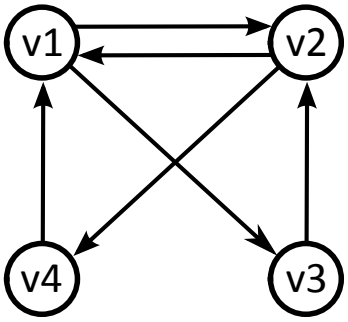
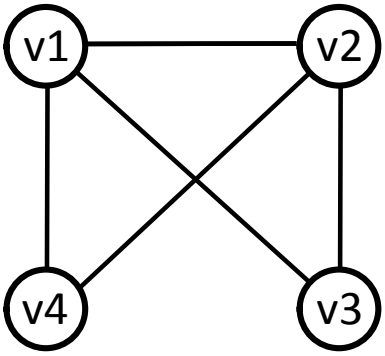
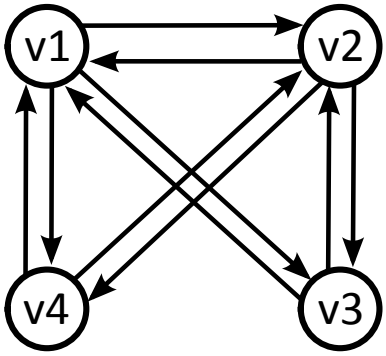


Figure 3: Description of the computational problem DirectedHamiltonianCircuit.



G, the original input to UndirectedHamiltonian-Circuit.



G', the transformed input to DirectedHamiltonian-Circuit.

Figure 4: An example of reducing UndirectedHamiltonianCircuit to Directed-HamiltonianCircuit.

3.1 A polyreduction from UndirectedHamiltonianCircuit to DirectedHamiltonianCircuit

So, can we find a polytime reduction from UndirectedHamiltonianCircuit to DirectedHamiltonianCircuit? The answer is yes, there's a surprisingly simple reduction that achieves this. Let's abbreviate the problem names to UHC and DHC respectively, and examine a specific example first. We are given a UHC input string I (such as "`v1 v2; v2 v3; v3 v1; v2 v4; v1 v4`") describing an undirected graph G (as in the left panel of Figure 4—this G is the same as in Figure 2). We need to transform G into a new, *directed* graph G', then somehow use DHC to solve our original problem. The key trick is to change each edge of G into two edges of G'—one in each direction. In our specific example, this results in the G' in the right panel of Figure 4. Specified as an ASCII string, this G' would be "`v1 v2; v2 v1; v2 v3; v3 v2; v3 v1; v1 v3; v2 v4; v4 v2; v1 v4; v4 v1`".

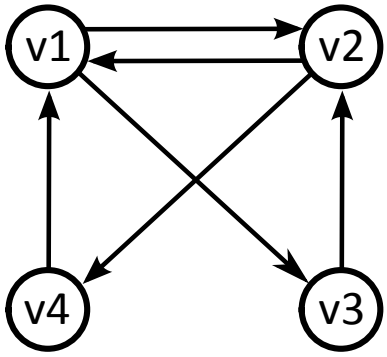
This is where a crucial part of the definition of polytime reductions comes into play: we are allowed to *assume* the existence of a polytime solution to problem Q (the problem we are trying to reduce *to*). Therefore, we can assume that by running some polytime program, we can find a directed Hamiltonian circuit for G'. Suppose the output in this case is "`v1 v4 v2 v3`".

The last part of the polyreduction is to convert the DHC output back to a corresponding solution for UHC. In this case, it is particularly easy: the DHC solution *is* a UHC solution, so no transformation is required.

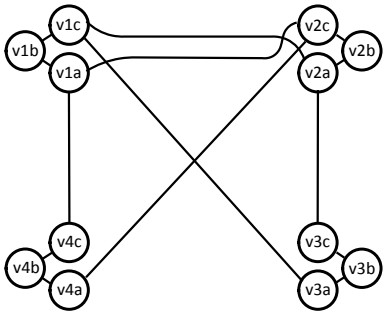
Our proof that UHC polyreduces to DHC isn't quite complete. We still need to show that both transformations (input→input and output→output) run in polynomial time. Well, the output transformation takes no time at all, so that part is certainly easy. The input transformation is also easy to deal with. Note that the ASCII form of G' is twice as long as G, since it contains twice as many edges. And we can compute G' from G from a single scan of G's description, copying and reversing each edge that we encounter. If the description of G has length n, this transformation requires only $O(n)$ time, which is certainly polynomial.

Compulsory exercise 1. Which, if any, of the following statements can be inferred directly from the fact that UHC polyreduces to DHC?

1. If UHC $\in$ Poly, then DHC $\in$ Poly.
2. If DHC $\in$ Poly, then UHC $\in$ Poly.
3. UHC and DHC are both members of Poly.
4. If UHC requires exponential time, then DHC requires exponential time.
5. If DHC requires exponential time, then UHC requires exponential time.
6. UHC and DHC both require exponential time.



G, the original input to DirectedHamiltonian-Circuit.



G', the transformed input to UndirectedHamiltonian-Circuit.

Figure 5: An example of reducing DirectedHamiltonianCircuit to Undirected-HamiltonianCircuit.

3.2 A polyreduction from DirectedHamiltonianCircuit to UndirectedHamiltonianCircuit

Interestingly, we can go back the other way: there is a polytime reduction from DHC to UHC too. The reduction in this direction is a little more elaborate, but it involves a beautiful idea that should be seen by all computer science students at least once in their careers.

This time we are given a *directed* graph G, such as the one on the left panel of Figure 5. (This G is the same as the graph in Figure 3). The string description of G is "`v1 v2; v2 v1; v3 v2; v1 v3; v2 v4; v4 v1`". We will now convert G to an undirected graph G', as shown in the right panel of Figure 5. Each vertex of G becomes three vertices in G': `v1` becomes `v1a`,`v1b`,`v1c`, and similarly for the other vertices. Let's call `v1a`,`v1b`,`v1c` the *children* of `v1` (and similarly for the other vertices). Each triplet of children with the same parent is connected by an edge from the `a`-child to the `b`-child, and from the `b`-child to the `c`-child. And every edge in G is transformed into a corresponding edge in G' from a `c`-child to an `a`-child. For example, the edge "`v2 v4`" becomes the edge "`v2c v4a`".

It's not too hard to convince yourself that any undirected Hamiltonian circuit C in G' corresponds to a directed Hamiltonian circuit C' in G. In the particular example we are working with, one possible value of C is "`v1a v1b v1c v3a v3b v3c v2a v2b v2c v4a v4b v4c`". By coalescing each triplet of children back to its parent, we can convert to a solution to our original directed input, in this case yielding "`v1 v3 v2 v4`".

Finally, we need to show that both transformations (input→input and output→output)

run in polynomial time. But it's easy to check that each transformation requires only linear time, so this requirement is certainly satisfied.

Compulsory exercise 2. Now that we know there are polytime reductions in *both* directions (UHC$\leqslant_{\text{Poly}}$DHC and DHC$\leqslant_{\text{Poly}}$UHC), which of the following statements can be inferred?

1. UHC and DHC are both members of Poly.
2. UHC and DHC both require exponential time.
3. Either both problems (UHC and DHC) are in Poly, or both require exponential time.
4. Either both problems (UHC and DHC) are in Poly, or both are outside Poly.

4 Polytime reductions between CircuitSAT, SAT, and 3-SAT

It turns out that there are reasonably simple polynomial time reductions in both directions between any two of the following problems: CircuitSAT, SAT, and 3-SAT. Some brief details of these reductions will be explained in class. The significance of these reductions will be seen when we begin our study of NP-completeness.

4.1 CircuitSAT polyreduces to SAT

Step 1: Rewrite the circuit as a Boolean formula. For example, the circuit shown in Figure 6 becomes

$$((x_1 \vee \neg x_2 \vee x_3) \wedge \neg(\neg x_3 \wedge x_4)) \vee x_5 \qquad (*)$$

Step 2: Convert the Boolean formula into CNF. We don't study the details of how to do this, but there is a simple polynomial time procedure for doing so. It is extremely similar to the procedure we all learned in high school for expanding any expression that uses additions and multiplications into a simple sum of products. For example, in high school we learn how to convert $x_1(x_2 + x_3(x_4 - x_5 - x_6))$ into $x_1x_2 + x_1x_3x_4 - x_1x_3x_5 - x_1x_3x_6$. Similarly, it's easy to convert the Boolean formula (*) into

$$(x_1 \vee \neg x_2 \vee x_3 \vee x_5) \wedge (x_3 \vee \neg x_4 \vee x_5)$$

By the way, you can also use Wolfram Alpha for this: enter the text

```
CNF ((x1 || ~x2 || x3) && ~(~x3 && x4)) || x5
```

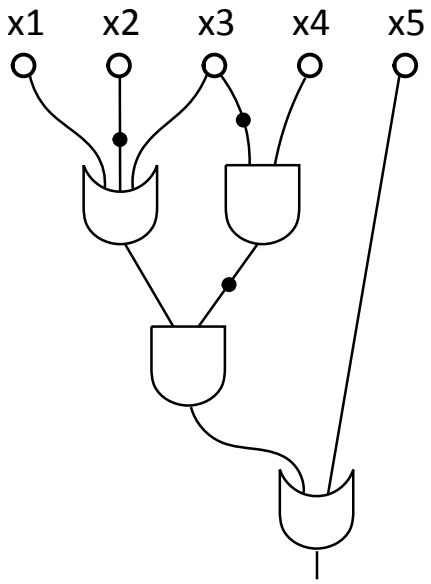


Figure 6: An example of a circuit. All circuits can easily be rewritten as Boolean formulas.

4.2 SAT polyreduces to CircuitSAT

This is easy—just rewrite a given Boolean formula as a circuit. It is a simple search-and-replace, substituting "`AND`" for $\wedge$, "`OR`" for $\vee$, and "`NOT`" for $\neg$.

4.3 3-SAT polyreduces to SAT

This is even more trivial—a 3-SAT formula *is* a SAT formula. So there's nothing to be done.

4.4 SAT polyreduces to 3-SAT

This time we have some work to do, but it turns out to be reasonably easy. We have to convert an arbitrary CNF formula into a CNF formula with at most 3 literals in each clause. The basic idea is to introduce dummy variables that break down any clause that is too big into two smaller clauses. For example, suppose we have a clause with four literals:

$$\ldots \wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \ldots$$

Introducing a new Boolean variable D (for Dummy), we can rewrite this as the equivalent formula:

$$\ldots \wedge (x_1 \vee x_2 \vee D) \wedge (\neg D \vee x_3 \vee x_4) \ldots$$

Now we just repeat this trick as many times as necessary, introducing dummy variables $D_2, D_3, \ldots$ until all the clauses have been reduced to 3 literals. It's easy to see that this transformation can be done in polynomial time.

4.5 An important conclusion

Because of the polytime reductions described above, we now know that: *either* all three problems (CircuitSAT, SAT, and 3-SAT) can be solved in polynomial time, *or* none of them can be solved in polynomial time.

Because no one has ever discovered a polynomial time algorithm for any of these three problems (and for additional reasons we will study soon), it is widely believed that none of these problems have polytime solutions.