Polynomial time reductions

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Polynomial-time reductions

### 

* Introduce the notion of *polynomial-time reductions* as a way to relate the complexity of problems to one another.
* See several examples of such reductions.
* 3SAT as a basic starting point for reductions.

Consider some of the problems we have encountered in chapefficient:

1. The *3SAT* problem: deciding whether a given 3CNF formula has a satisfying assignment.
2. Finding the *longest path* in a graph.
3. Finding the *maximum cut* in a graph.
4. Solving *quadratic equations* over variables .

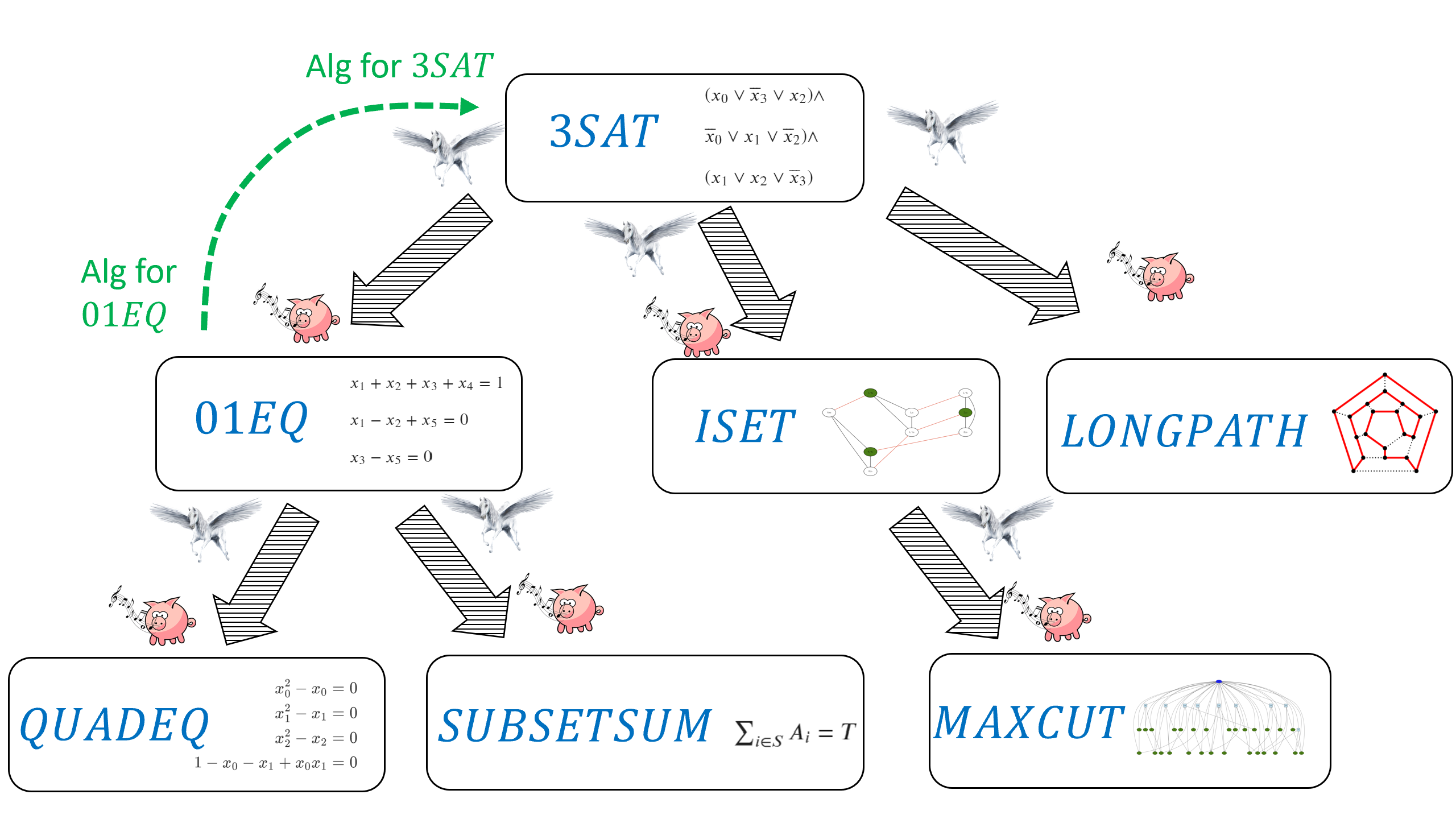
All of these problems have the following properties:

* These are important problems, and people have spent significant effort on trying to find better algorithms for them.
* Each one of these is a *search* problem, whereby we search for a solution that is “good” in some easy to define sense (e.g., a long path, a satisfying assignment, etc.).
* Each of these problems has a trivial exponential time algorithm that involve enumerating all possible solutions.
* At the moment, for all these problems the best known algorithm is not much faster than the trivial one in the worst case.

In this chapter and in cooklevinchap we will see that, despite their apparent differences, we can relate the computational complexity of these and many other problems. In fact, it turns out that the problems above are *computationally equivalent*, in the sense that solving one of them immediately implies solving the others. This phenomenon, known as  *completeness*, is one of the surprising discoveries of theoretical computer science, and we will see that it has far-reaching ramifications.

This chapter introduces the concept of a *polynomial time reduction* which is a central object in computational complexity and this book in particular. A polynomial-time reduction is a way to *reduce* the task of solving one problem to another. The way we use reductions in complexity is to argue that if the first problem is hard to solve efficiently, then the second must also be hard. We see several examples for reductions in this chapter, and reductions will be the basis for the theory of  *completeness* that we will develop in cooklevinchap.

All the code for the reductions described in this chapter is available on the [following Jupyter notebook](https://colab.research.google.com/github/boazbk/tcscode/blob/master/Chap_13_reductions.ipynb).



In this chapter we show that if the problem cannot be solved in polynomial time, then neither can the , , and problems. We do this by using the *reduction paradigm* showing for example “if pigs could whistle” (i.e., if we had an efficient algorithm for ) then “horses could fly” (i.e., we would have an efficient algorithm for .)

In this chapter we will see that for each one of the problems of finding a longest path in a graph, solving quadratic equations, and finding the maximum cut, if there exists a polynomial-time algorithm for this problem then there exists a polynomial-time algorithm for the 3SAT problem as well. In other words, we will *reduce* the task of solving 3SAT to each one of the above tasks. Another way to interpret these results is that if there *does not exist* a polynomial-time algorithm for 3SAT then there does not exist a polynomial-time algorithm for these other problems as well. In cooklevinchap we will see evidence (though no proof!) that all of the above problems do not have polynomial-time algorithms and hence are *inherently intractable*.

## Formal definitions of problems

For reasons of technical convenience rather than anything substantial, we concern ourselves with *decision problems* (i.e., Yes/No questions) or in other words *Boolean* (i.e., one-bit output) functions. We model the problems above as functions mapping to in the following way:

**3SAT.** The *3SAT problem* can be phrased as the function that takes as input a 3CNF formula (i.e., a formula of the form where each is the OR of three variables or their negation) and maps to if there exists some assignment to the variables of that causes it to evalute to *true*, and to otherwise. For example

$$3SAT\left("(x\_0 \vee \overline{x}\_1 \vee x\_2) \wedge (x\_1 \vee x\_2 \vee \overline{x\_3}) \wedge (\overline{x}\_0 \vee \overline{x}\_2 \vee x\_3)" \right) = 1$$

since the assignment satisfies the input formula. In the above we assume some representation of formulas as strings, and define the function to output if its input is not a valid representation; we use the same convention for all the other functions below.

**Quadratic equations.** The *quadratic equations problem* corresponds to the function that maps a set of quadratic equations to if there is an assignment that satisfies all equations, and to otherwise.

**Longest path.** The *longest path problem* corresponds to the function that maps a graph and a number to if there is a simple path in of length at least , and maps to otherwise. The longest path problem is a generalization of the well-known [Hamiltonian Path Problem](https://en.wikipedia.org/wiki/Hamiltonian_path_problem) of determining whether a path of length exists in a given vertex graph.

**Maximum cut.** The *maximum cut problem* corresponds to the function that maps a graph and a number to if there is a cut in that cuts at least edges, and maps to otherwise.

All of the problems above are in but it is not known whether or not they are in . However, we will see in this chapter that if either , or are in , then so is .

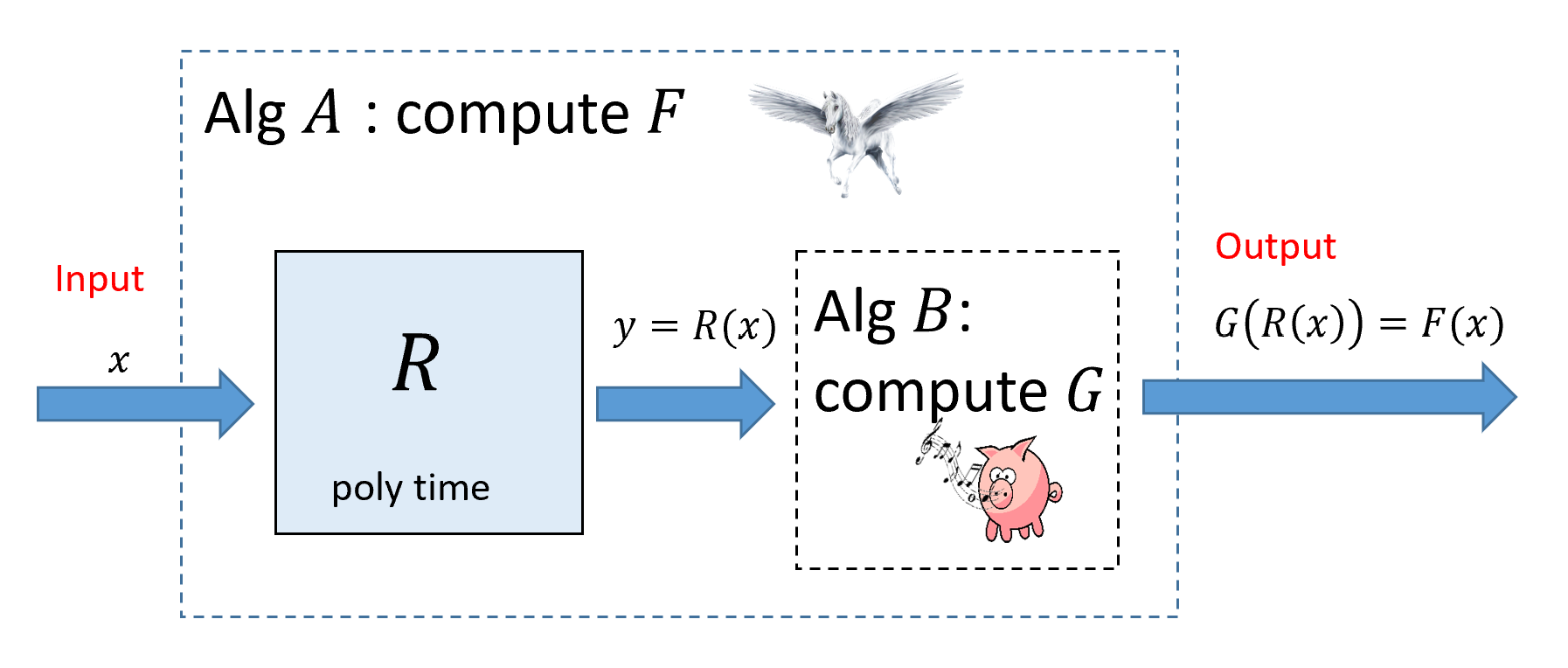
## Polynomial-time reductions

Suppose that are two Boolean functions. A *polynomial-time reduction* (or sometimes just *“reduction”* for short) from to is a way to show that is “no harder” than , in the sense that a polynomial-time algorithm for implies a polynomial-time algorithm for .

### 

Let . We say that  *reduces to* , denoted by if there is a polynomial-time computable such that for every ,

We say that and have *equivalent complexity* if and .



If then we can transform a polynomial-time algorithm that computes into a polynomial-time algorithm that computes . To compute we can run the reduction guaranteed by the fact that to obtain and then run our algorithm for to compute .

The following exercise justifies our intuition that signifies that “ is no harder than ”.

Prove that if and then .

As usual, solving this exercise on your own is an excellent way to make sure you understand reduction-def.

Suppose there is an algorithm that computes in time where is its input size. Then, eq:reduction directly gives an algorithm to compute (see reductionsfig). Indeed, on input , Algorithm will run the polynomial-time reduction to obtain and then return . By eq:reduction, and hence Algorithm will indeed compute .

We now show that runs in polynomial time. By assumption, can be computed in time for some polynomial . In particular, this means that (as just writing down takes steps). Computing will take at most steps. Thus the total running time of on inputs of length is at most the time to compute , which is bounded by , and the time to compute , which is bounded by , and since the composition of two polynomials is a polynomial, runs in polynomial time.

A *reduction* shows that is “no harder than ” or equivalently that is “no easier than ”.

### Whistling pigs and flying horses

A reduction from to can be used for two purposes:

* If we already know an algorithm for and then we can use the reduction to obtain an algorithm for . This is a widely used tool in algorithm design. For example in linerprogsec we saw how the *Min-Cut Max-Flow* theorem allows to reduce the task of computing a minimum cut in a graph to the task of computing a maximum flow in it.
* If we have proven (or have evidence) that there exists *no polynomial-time algorithm* for and then the existence of this reduction allows us to conclude that there exists no polynomial-time algorithm for . This is the “if pigs could whistle then horses could fly” interpretation we’ve seen in reductionsuncompsec. We show that if there was an hypothetical efficient algorithm for (a “whistling pig”) then since then there would be an efficient algorithm for (a “flying horse”). In this book we often use reductions for this second purpose, although the lines between the two is sometimes blurry (see the bibliographical notes in reductionsbibnotes).

The most crucial difference between the notion in reduction-def and the reductions we saw in the context of *uncomputability* (e.g., in reductionsuncompsec) is that for relating time complexity of problems, we need the reduction to be computable in *polynomial time*, as opposed to merely computable. reduction-def also restricts reductions to have a very specific format. That is, to show that , rather than allowing a general algorithm for that uses a “magic box” that computes , we only allow an algorithm that computes by outputting . This restricted form is convenient for us, but people have defined and used more general reductions as well (see reductionsbibnotes).

In this chapter we use reductions to relate the computational complexity of the problems mentioned above: 3SAT, Quadratic Equations, Maximum Cut, and Longest Path, as well as a few others. We will reduce 3SAT to the latter problems, demonstrating that solving any one of them efficiently will result in an efficient algorithm for 3SAT. In cooklevinchap we show the other direction: reducing each one of these problems to 3SAT in one fell swoop.

**Transitivity of reductions.** Since we think of as saying that (as far as polynomial-time computation is concerned) is “easier or equal in difficulty to” , we would expect that if and , then it would hold that . Indeed this is the case:

For every , if and then .

If and then there exist polynomial-time computable functions and mapping to such that for every , and for every , . Combining these two equalities, we see that for every , and so to show that , it is sufficient to show that the map is computable in polynomial time. But if there are some constants such that is computable in time and is computable in time then is computable in time which is polynomial.

## Reducing 3SAT to zero one and quadratic equations

We now show our first example of a reduction. The *Zero-One Linear Equations problem* corresponds to the function whose input is a collection of linear equations in variables , and the output is iff there is an assignment of values to the variables that satisfies all the equations. For example, if the input is a string encoding the set of equations

then since the assignment satisfies all three equations. We specifically restrict attention to linear equations in variables in which every equation has the form where and .[[1]](#footnote-37)

If we asked the question of whether there is a solution of *real numbers* to , then this can be solved using the famous *Gaussian elimination* algorithm in polynomial time. However, there is no known efficient algorithm to solve . Indeed, such an algorithm would imply an algorithm for as shown by the following theorem:

### 

### 

A constraint can be written as . This is a linear *inequality* but since the sum on the left-hand side is at most three, we can also turn it into an *equality* by adding two new variables and writing it as . (We will use fresh variables for every constraint.) Finally, for every variable we can add a variable corresponding to its negation by adding the equation , hence mapping the original constraint to . The main **takeaway technique** from this reduction is the idea of adding *auxiliary variables* to replace an equation such as that is not quite in the form we want with the equivalent (for valued variables) equation which is in the form we want.



Left: Python code implementing the reduction of to . Right: Example output of the reduction. Code is in our [repository](https://github.com/boazbk/tcscode).

To prove the theorem we need to:

1. Describe an algorithm for mapping an input for into an input for .
2. Prove that the algorithm runs in polynomial time.
3. Prove that for every 3CNF formula .

We now proceed to do just that. Since this is our first reduction, we will spell out this proof in detail. However it straightforwardly follows the proof idea.

INPUT: 3CNF formula $\varphi$ with $n$ variables $x\_0,\ldots,x\_{n-1}$ and $m$ clauses.  
  
OUTPUT: Set $E$ of linear equations over $0/1$ such that $3SAT(\varphi)=1$ iff $01EQ(E)=1$.  
  
Let $E$'s variables be $x\_0,\ldots,x\_{n-1}$, $x'\_0,\ldots,x'\_{n-1}$, $y\_0,\ldots,y\_{m-1}$, $z\_0,\ldots,z\_{m-1}$.  
For{$i \in [n]$}  
 add to $E$ the equation $x\_i + x'\_i = 1$  
endfor  
For{$j\in [m]$}  
 Let $j$-th clause be $w\_0 \vee w\_1 \vee w\_2$ where $w\_0,w\_1,w\_2$ are literals.  
 For{$a\in[3]$}  
 If{$w\_a$ is variable $x\_i$}  
 set $t\_a \leftarrow x\_i$  
 endif  
 If{$w\_a$ is negation $\neg x\_i$}  
 set $t\_a \leftarrow x'\_i$  
 endif  
 endfor  
 Add to $E$ the equation $t\_0 + t\_1 + t\_2 + y\_j + z\_j = 3$.  
endfor  
return $E$

The reduction is described in zerooneeqreduction, see also threesat2zoeqreductionfig. If the input formula has variables and clauses, zerooneeqreduction creates a set of equations over variables. zerooneeqreduction makes an initial loop of steps (each taking constant time) and then another loop of steps (each taking constant time) to create the equations, and hence it runs in polynomial time.

Let be the function computed by zerooneeqreduction. The heart of the proof is to show that for every 3CNF , . We split the proof into two parts. The first part, traditionally known as the **completeness** property, is to show that if then . The second part, traditionally known as the **soundness** property, is to show that if then . (The names “completeness” and “soundness” derive viewing a solution to as a “proof” that is satisfiable, in which case these conditions corresponds to completeness and soundness as defined in #godelproofsystemssec. However, if you find the names confusing you can simply think of completeness as the “-instance maps to -instance” property and soundness as the “-instance maps to -instance” property.)

We complete the proof by showing both parts:

* **Completeness:** Suppose that , which means that there is an assignment that satisfies . If we use the assignment and for the first variables of then we will satisfy all equations of the form . Moreover, for every , if is the equation arising from the th clause of (with being variables of the form or depending on the literals of the clause) then our assignment to the first variables ensures that (since satisfied ) and hence we can assign values to and that will ensure that the equation is satisfied. Hence in this case is satisfied, meaning that .
* **Soundness:** Suppose that , which means that the set of equations has a satisfying assignment , , , . Then, since the equations contain the condition , for every , is the negation of , and morover, for every , if has the form and is the -th clause of , then the corresponding assignment will ensure that , implying that is satisfied. Hence in this case .

### Quadratic equations

Now that we reduced to , we can use this to reduce to the *quadratic equations* problem. This is the function in which the input is a list of -variate polynomials that are all of [degree](https://en.wikipedia.org/wiki/Degree_of_a_polynomial) at most two (i.e., they are *quadratic*) and with integer coefficients. (The latter condition is for convenience and can be achieved by scaling.) We define to equal if and only if there is a solution to the equations , , , .

For example, the following is a set of quadratic equations over the variables :

You can verify that satisfies this set of equations if and only if and .

### 

### 

Using the transitivity of reductions (transitiveex), it is enough to show that , but this follows since we can phrase the equation as the quadratic constraint . The **takeaway technique** of this reduction is that we can use *non-linearity* to force continuous variables (e.g., variables taking values in ) to be discrete (e.g., take values in ).

By tsattozoeqthm and transitiveex, it is sufficient to prove that . Let be an instance of with variables . We map to the set of quadratic equations that is obtained by taking the linear equations in and adding to them the quadratic equations for all . (See zeroonetoquadreductionalg.) The map can be computed in polynomial time. We claim that if and only if . Indeed, the only difference between the two instances is that:

* In the instance , the equations are over variables in .
* In the instance , the equations are over variables but we have the extra constraints for all .

Since for every , if and only if , the two sets of equations are equivalent and which is what we wanted to prove.

INPUT: Set $E$ of linear equations over $n$ variables $x\_0,\ldots,x\_{n-1}$.  
  
OUTPUT: Set $E'$ of quadratic equations over $m$ variables $w\_0,\ldots,w\_{m-1}$ such that there is an $0/1$ assignment $x\in \{0,1\}^n$  
satisfying the equations of $E$ iff there is an assignment $w \in \R^m$ satisfying the equations of $E'$.  
That is, $01EQ(E) = QUADEQ(E')$.  
  
Let $m \leftarrow n$.  
Variables of $E'$ are set to be same variable $x\_0,\ldots, x\_{n-1}$ as $E$.  
For{every equation $e\in E$}  
 Add $e$ to $E'$  
endfor  
For{$i\in [n]$}  
 Add to $E'$ the equation $x\_i^2 - x\_i = 0$.  
endfor  
return $E'$

## The subset sum problem

As another consequence of the reduction of to , we can also show that (through ) reduces to the *subset sum* problem (also known as the *knapsack* problem). In the *subset sum* problem, we are given a list of integers and an integer . We need to determine whether or not there exists some set of the integers that sums up to . That is, for , if and only if there exists such that . Note that the input length for the subset sum problem is the length of string needed to encode all the numbers, which will be approximately , since encoding an integer using the binary representation requires bits.

### 

### 

We reduce from . The intuition is the following. Consider an instance of with variables and equations . Recall that each equation in has the form (potentially with more or less than three variables summed up on the left-hand side of the equation). For every variable , we can define a vector where if the variable appears in the equation and otherwise. Then there is a solution to the set of equations if and only if there is some set (corresponding to the ’s such that ) such that where is the vector of right hand sides of the equations (i.e., is the value on the righthand side of the -th equation). Now if we could interpret the vectors and as *numbers* then we could think of this as a subset sum instance. The key insight is that we can in fact think of vectors as numbers by thinking of the -th coordinate of the vector as the -th digit. Since the vectors are in , the natural choice is to use the binary basis, but this turns out to cause issues with “carries” when we add them up. Hence we use a larger basis , see proof below.

For a given set of on variables, we note that the right hand side can never be larger than (since the sum of at most variables in is at most ). More concretely, if the instance has such an equation then we can know for sure that the answer is (and in the context of a reduction map it into some trivial instance of subset sum that doesn’t have a solution such as and ).

Our reduction is described in zeroonetossumnalg. On input an instance of over variables , we output an instance computed as follows:

* where equals if the variable appears in the equation and equals otherwise. The number is set to be (any numb er larger than would work.)
* where is the integer on the right-hand side of the equation .

In other words, and are the integers such that, written in the -ary basis, the -th digit of is iff appears in , and the -th digit of is the right-hand side of .

The following claim will imply the correctness of the reduction:

**Claim:** For every , if then satisfies the equations of if and only if .

**Proof:** Key to the proof is the following simple property of gradeschool addition: when adding at most numbers in the -ary basis, if all the numbers have all their digits either or , and , then for every , the -th digit of the sum is the sum of the -th digits of the numbers. This is a simple consequence of the fact that there is no “carry” in the addition. Since in our case the numbers satisfy this property in the -ary basis, and , we get that for every and every digit , the -th digit of the sum is simply the sum of the -th digit, which would correspond to the sum over for all ’s that participate in the -th equation. This sum would equal the -th digit of if and only if that equation is satisfied.

The claim shows that which is what we needed to prove.

INPUT: Set $E = \{ e\_t \}\_{t\in [m]}$ of $m$ linear equations over $n$ variables $x\_0,\ldots,x\_{n-1}$.  
  
OUTPUT: Numbers $y\_0,\ldots,y\_{n-1},T \in \mathbb{Z}$ such that there is an $0/1$ assignment $x\in \{0,1\}^n$ satisfying the equations of $E$ iff there is $S \subseteq [n]$ such that $\sum\_{i\in S}y\_i = T$.  
  
For{every equation $e\_t\in E$}  
 Let $A \subseteq [n]$ and $b\in \mathbb{Z}$ be such that $e\_t$ has the form $\sum\_{i\in A} x\_i = b$  
 Let $v\_i^t \leftarrow 1$ -if $i\in A$ and $v\_i^t \leftarrow 0$ otherwise.  
 Let $b\_t \leftarrow b$.   
endfor  
Set $B \leftarrow 2n$  
For{$i\in [n]$}  
 Let $y\_i \leftarrow \sum\_{t=1}^m B^t v\_i^t$.   
endfor  
Let $T \leftarrow \sum\_{t=1}^T B^t b\_t$  
return $y\_0,\ldots,y\_{n-1},T$

## The independent set problem

For a graph , an [independent set](https://en.wikipedia.org/wiki/Independent_set_(graph_theory)) (also known as a *stable set*) is a subset such that there are no edges with both endpoints in (in other words, ). Every “singleton” (set consisting of a single vertex) is trivially an independent set, but finding larger independent sets can be challenging. The *maximum independent set* problem (henceforth simply “independent set”) is the task of finding the largest independent set in the graph. The independent set problem is naturally related to *scheduling problems*: if we put an edge between two conflicting tasks, then an independent set corresponds to a set of tasks that can all be scheduled together without conflicts. The independent set problem has been studied in a variety of settings, including for example in the case of algorithms for finding structure in [protein-protein interaction graphs](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3919085/).

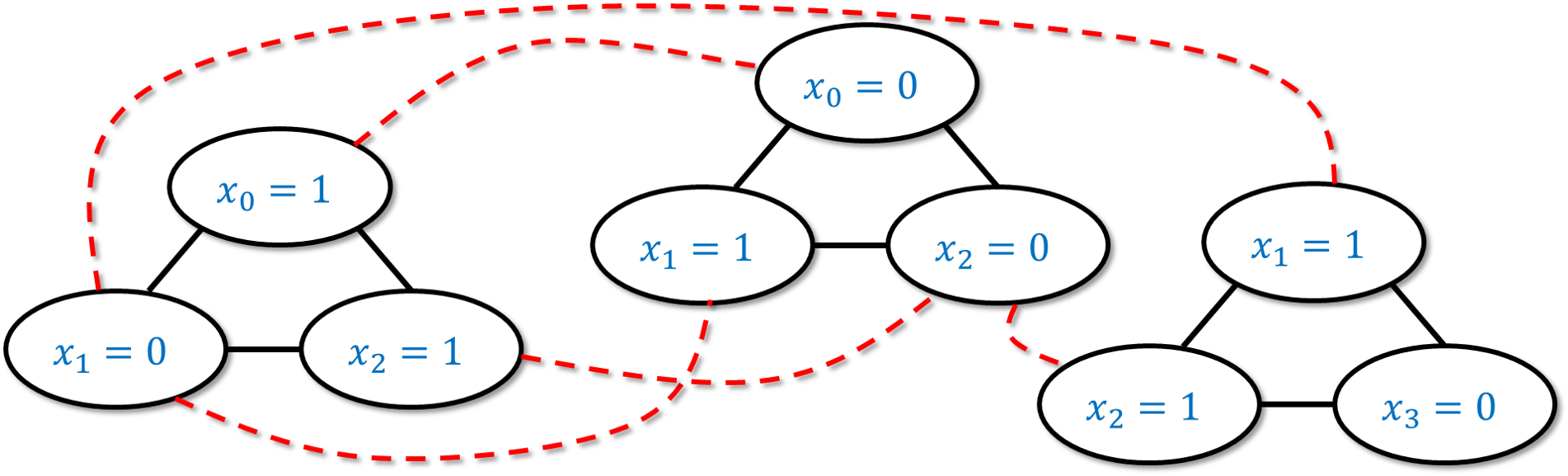
As mentioned in formaldefdecisionexamplessec, we think of the independent set problem as the function that on input a graph and a number outputs if and only if the graph contains an independent set of size at least . We now reduce 3SAT to Independent set.

### 

.

### 

The idea is that finding a satisfying assignment to a 3SAT formula corresponds to satisfying many local constraints without creating any conflicts. One can think of “” and “” as two conflicting events, and of the constraints as creating a conflict between the events “”, “” and “”, saying that these three cannot simultaneously co-occur. Using these ideas, we can we can think of solving a 3SAT problem as trying to schedule non-conflicting events, though the devil is, as usual, in the details. The **takeaway technique** here is to map each clause of the original formula into a *gadget* which is a small subgraph (or more generally “subinstance”) satisfying some convenient properties. We will see these “gadgets” used time and again in the construction of polynomial-time reductions.



An example of the reduction of to for the case the original input formula is . We map each clause of to a triangle of three vertices, each tagged above with “” or “” depending on the value of that would satisfy the particular literal. We put an edge between every two literals that are *conflicting* (i.e., tagged with “” and “” respectively).

INPUT: $3SAT$ formula $\varphi$ with $n$ variables and $m$ clauses.  
  
OUTPUT: Graph $G=(V,E)$ and number $k$, such that $G$ has an independent set of size $k$ iff $\varphi$ has a satisfying assignment.  
That is, $3SAT(\varphi) = ISET(G,k)$,   
  
Initialize $V \leftarrow \emptyset, E \leftarrow \emptyset$  
For {every clause $C = y \vee y' \vee y''$ of $\varphi$}  
 Add three vertices $(C,y),(C,y'),(C,y'')$ to $V$  
 Add edges $\{ (C,y), (C,y') \}$, $\{(C,y'),(C,y'') \}$, $\{ (C,y''), (C,y) \}$ to $E$.  
endfor  
for {every distinct clauses $C,C'$ in $\varphi$}  
 for {every $i\in [n]$}  
 if{$C$ contains literal $x\_i$ and $C'$ contains literal $\overline{x}\_i$}  
 Add edge $\{ (C,x\_i), (C',\overline{x}\_i) \}$ to $E$  
 endif  
 endfor  
endfor  
return $(G=(V,E), m)$

Given a 3SAT formula on variables and with clauses, we will create a graph with vertices as follows. (See threesattoisetreductionalg, see also example3sat2isetfig for an example and threesattoisfig for Python code.)

* A clause in has the form where are *literals* (variables or their negation). For each such clause , we will add three vertices to , and label them , , and respectively. We will also add the three edges between all pairs of these vertices, so they form a *triangle*. Since there are clauses in , the graph will have vertices.
* In addition to the above edges, we also add an edge between every pair of vertices of the form and where and are *conflicting* literals. That is, we add an edge between and if there is an such that and or vice versa.

The algorithm constructing based on takes polynomial time since it involves two loops, the first taking steps and the second taking steps (see threesattoisetreductionalg). Hence to prove the theorem we need to show that is satisfiable if and only if contains an independent set of vertices. We now show both directions of this equivalence:

**Part 1: Completeness.** The “completeness” direction is to show that if has a satisfying assignment , then has an independent set of vertices. Let us now show this.

Indeed, suppose that has a satisfying assignment . Then for every clause of , one of the literals must evaluate to *true* under the assignment (as otherwise it would not satisfy ). We let be a set of vertices that is obtained by choosing for every clause one vertex of the form such that evaluates to true under . (If there is more than one such vertex for the same , we arbitrarily choose one of them.)

We claim that is an independent set. Indeed, suppose otherwise that there was a pair of vertices and in that have an edge between them. Since we picked one vertex out of each triangle corresponding to a clause, it must be that . Hence the only way that there is an edge between and is if and are conflicting literals (i.e.  and for some ). But then they can’t both evaluate to *true* under the assignment , which contradicts the way we constructed the set . This completes the proof of the completeness condition.

**Part 2: Soundness.** The “soundness” direction is to show that if has an independent set of vertices, then has a satisfying assignment . Let us now show this.

Indeed, suppose that has an independent set with vertices. We will define an assignment for the variables of as follows. For every , we set according to the following rules:

* If contains a vertex of the form then we set .
* If contains a vertex of the form then we set .
* If does not contain a vertex of either of these forms, then it does not matter which value we give to , but for concreteness we’ll set .

The first observation is that is indeed well defined, in the sense that the rules above do not conflict with one another, and ask to set to be both and . This follows from the fact that is an *independent set* and hence if it contains a vertex of the form then it cannot contain a vertex of the form .

We now claim that is a satisfying assignment for . Indeed, since is an independent set, it cannot have more than one vertex inside each one of the triangles corresponding to a clause of . Hence since , it must have exactly one vertex in each such triangle. For every clause of , if is the vertex in in the triangle corresponding to , then by the way we defined , the literal must evaluate to *true*, which means that satisfies this clause. Therefore satisfies all clauses of , which is the definition of a satisfying assignment.

This completes the proof of isetnpc



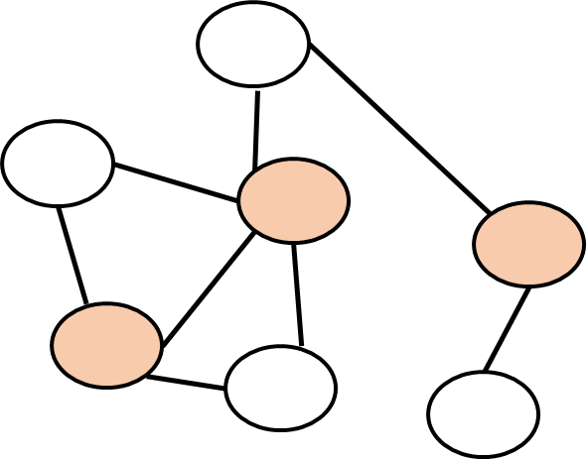
The reduction of 3SAT to Independent Set. On the right-hand side is *Python* code that implements this reduction. On the left-hand side is a sample output of the reduction. We use black for the “triangle edges” and red for the “conflict edges”. Note that the satisfying assignment corresponds to the independent set , , .

## Some exercises and anatomy of a reduction.

Reductions can be confusing and working out exercises is a great way to gain more comfort with them. Here is one such example. As usual, I recommend you try it out yourself before looking at the solution.

A *vertex cover* in a graph is a subset of vertices such that every edge touches at least one vertex of (see smallvertexcoverfig). The *vertex cover problem* is the task to determine, given a graph and a number , whether there exists a vertex cover in the graph with at most vertices. Formally, this is the function such that for every and , if and only if there exists a vertex cover such that .

Prove that .



A *vertex cover* in a graph is a subset of vertices that touches all edges. In this -vertex graph, the filled vertices are a vertex cover.

The key observation is that if is a vertex cover that touches all vertices, then there is no edge such that both ’s endpoints are in the set , and vice versa. In other words, is a vertex cover if and only if is an independent set. Since the size of is , we see that the polynomial-time map (where is the number of vertices of ) satisfies that which means that it is a reduction from independent set to vertex cover.

The [maximum clique problem](https://en.wikipedia.org/wiki/Clique_problem) corresponds to the function such that for a graph and a number , iff there is a subset of vertices such that for *every* distinct , the edge is in . Such a set is known as a *clique*.

Prove that and .

If is a graph, we denote by its *complement* which is the graph on the same vertices and such that for every distinct , the edge is present in if and only if this edge is *not* present in .

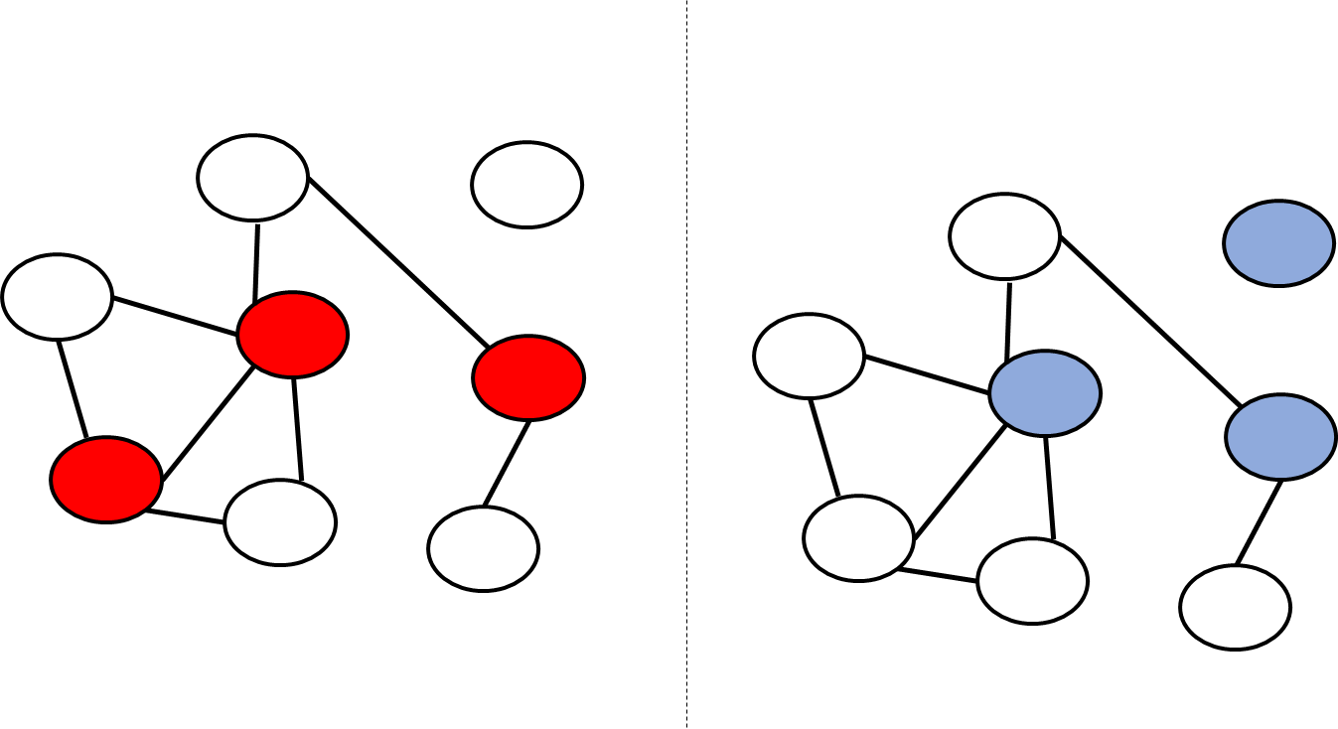
This means that for every set , is an independent set in if and only if is a *clique* in . Therefore for every , . Since the map can be computed efficiently, this yields a reduction . Moreover, since this yields a reduction in the other direction as well.

### Dominating set

In the two examples above, the reduction was almost “trivial”: the reduction from independent set to vertex cover merely changes the number to , and the reduction from independent set to clique flips edges to non-edges and vice versa. The following exercise requires a somewhat more interesting reduction.

A *dominating set* in a graph is a subset of vertices such that every is a neighbor in of some (see dominatingvertexcover). The *dominating set problem* is the task, given a graph and number , of determining whether there exists a dominating set with . Formally, this is the function such that iff there is a dominating set in of at most vertices.

Prove that .



A dominating set is a subset of vertices such that every vertex in the graph is either in or a neighbor of . The figure above are two copies of the same graph. The red vertices on the left are a vertex cover that is not a dominating set. The blue vertices on the right are a dominating set that is not a vertex cover.

Since we know that , using transitivity, it is enough to show that . As dominatingvertexcover shows, a dominating set is not the same thing as a vertex cover. However, we can still relate the two problems. The idea is to map a graph into a graph such that a vertex cover in would translate into a dominating set in and vice versa. We do so by including in all the vertices and edges of , but for every edge of we also add to a new vertex and connect it to both and . Let be the number of isolated vertices in . The idea behind the proof is that we can transform a vertex cover of vertices in into a dominating set of vertices in by adding to all the isolated vertices, and moreover we can transform every -sized dominating set in into a vertex cover in . We now give the details.

**Description of the algorithm.** Given an instance for the vertex cover problem, we will map into an instance for the dominating set problem as follows (see vctodsreductionfig for Python implementation):

INPUT: Graph $G=(V,E)$ and number $k$.  
  
OUTPUT: Graph $H=(V',E')$ and number $k'$, such that $G$ has a vertex cover of size $k$ iff $H$ has a dominating set of size $k'$, that is, $DS(H,k') = VC(G,k)$.  
  
Initialize $V' \leftarrow V, E' \leftarrow E$  
For {every edge $\{u,v\} \in E$}  
 Add vertex $w\_{u,v}$ to $V'$  
 Add edges $\{ u, w\_{u,v} \}$, $\{ v, w\_{u,v} \}$ to $E'$.  
endfor  
Let $\ell \leftarrow$ number of isolated vertices in $G$  
return $( H=(V',E') \;,\; k+\ell)$

independentsettodsredalg runs in polynomial time, since the loop takes steps where is the number of edges, with each step can be implemented in constant or at most linear time (depending on the representation of the graph ). Counting the number of isolated vertices in an vertex graph can be done in time if is represented in the adjacency matrix representation and time if it is represented in the adjacency list representation. Regardless the algorithm runs in polynomial time.

To complete the proof we need to prove that for every , if is the output of independentsettodsredalg on input , then . We split the proof into two parts. The *completeness* part is that if then . The *soundness* part is that if then .

**Completeness.** Suppose that . Then there is a vertex cover of at most vertices. Let be the set of isolated vertices in and be their number. Then . We claim that is a dominating set in . Indeed for every vertex of there are three cases:

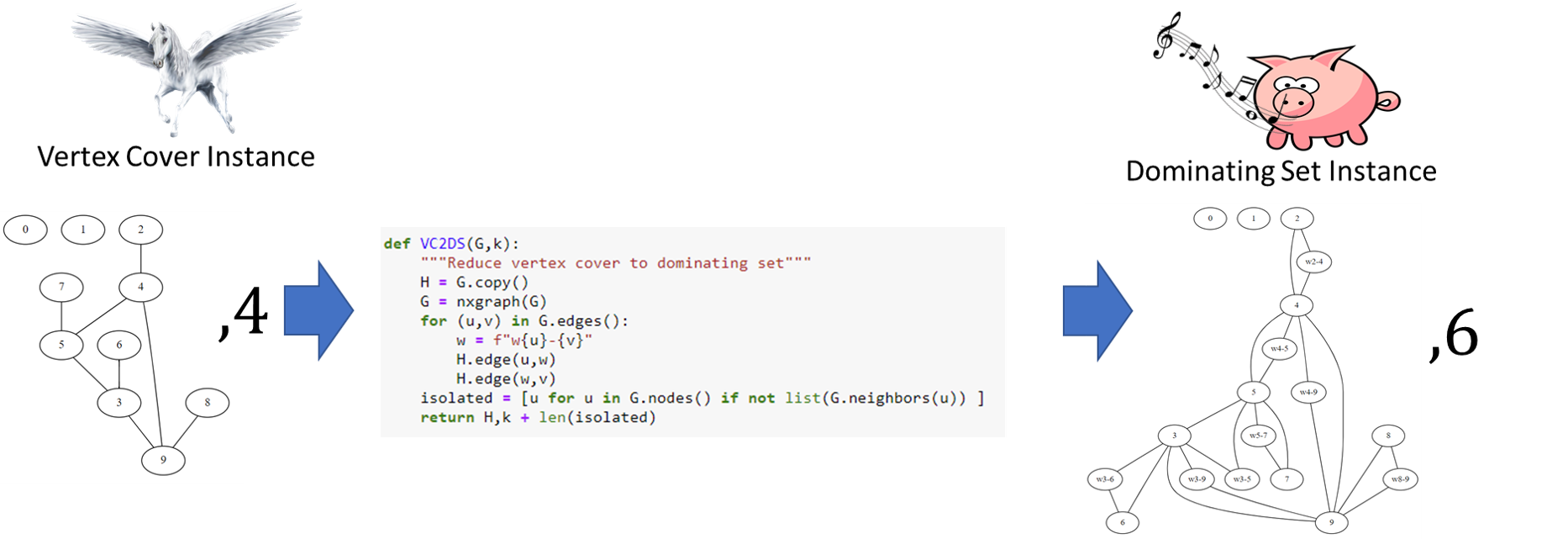
* **Case 1:** is an isolated vertex of . In this case is in .
* **Case 2:** is a non-isolated vertex of and hence there is an edge of for some . In this case since is a vertex cover, one of has to be in , and hence either or a neighbor of has to be in .
* **Case 3:** is of the form for some two neighbors in . But then since is a vertex cover, one of has to be in and hence contains a neighbor of .

We conclude that is a dominating set of size at most in and hence under the assumption that , .

**Soundness.** Suppose that . Then there is a dominating set of size at most in . For every edge in the graph , if contains the vertex then we remove this vertex and add in its place. The only two neighbors of are and , and since is a neighbor of both and of , replacing with maintains the property that it is a dominating set. Moreover, this change cannot increase the size of . Thus following this modification, we can assume that is a dominating set of at most vertices that does not contain any vertices of the form .

Let be the set of isolated vertices in . These vertices are also isolated in and hence must be included in (an isolated vertex must be in any dominating set, since it has no neighbors). We let . Then . We claim that is a vertex cover in . Indeed, for every edge of , either the vertex or one of its neighbors must be in by the dominating set property. But since we ensured doesn’t contain any of the vertices of the form , it must be the case that either or is in . This shows that is a vertex cover of of size at most , hence proving that .

A corollary of independentsettodsredalg and the other reduction we have seen so far is that if (i.e., dominating set has a polynomial-time algorithm) then (i.e., has a polynomial-time algorithm). By the contra-positive, if does *not* have a polynomial-time algorithm then neither does dominating set.



Python implementation of the reduction from vertex cover to dominating set, together with an example of an input graph and the resulting output graph. This reduction allows to transform a hypothetical polynomial-time algorithm for dominating set (a “whistling pig”) into a hypothetical polynomial-time algorithm for vertex-cover (a “flying horse”).

### Anatomy of a reduction



The four components of a reduction, illustrated for the particular reduction of vertex cover to dominating set. A reduction from problem to problem is an algorithm that maps an input for into an input for . To show that the reduction is correct we need to show the properties of *efficiency*: algorithm runs in polynomial time, *completeness*: if then , and *soundness*: if then .

The reduction of dominatingsetex gives a good illustration of the anatomy of a reduction. A reduction consists of four parts:

* **Algorithm description:** This is the description of *how* the algorithm maps an input into the output. For example, in dominatingsetex this is the description of how we map an instance of the *vertex cover* problem into an instance of the *dominating set* problem.
* **Algorithm analysis:** It is not enough to describe *how* the algorithm works but we need to also explain *why* it works. In particular we need to provide an *analysis* explaining why the reduction is both *efficient* (i.e., runs in polynomial time) and *correct* (satisfies that for every ). Specifically, the components of analysis of a reduction include:
  + **Efficiency:** We need to show that runs in polynomial time. In most reductions we encounter this part is straightforward, as the reductions we typically use involve a constant number of nested loops, each involving a constant number of operations. For example, the reduction of dominatingsetex just enumerates over the edges and vertices of the input graph.
  + **Completeness:** In a reduction demonstrating , the *completeness* condition is the condition that for every , if then . Typically we construct the reduction to ensure that this holds, by giving a way to map a “certificate/solution” certifying that into a solution certifying that . For example, in dominatingsetex we constructed the graph such that for every vertex cover in , the set (where is the isolated vertices) would be a dominating set in .
  + **Soundness:** This is the condition that if then or (taking the contrapositive) if then . This is sometimes straightforward but can often be harder to show than the completeness condition, and in more advanced reductions (such as the reduction of isetnpc) demonstrating soundness is the main part of the analysis. For example, in dominatingsetex to show soundness we needed to show that for *every* dominating set in the graph , there exists a vertex cover of size at most in the graph (where is the number of isolated vertices). This was challenging since the dominating set might not be necessarily the one we “had in mind”. In particular, in the proof above we needed to modify to ensure that it does not contain vertices of the form , and it was important to show that this modification still maintains the property that is a dominating set, and also does not make it bigger.

Whenever you need to provide a reduction, you should make sure that your description has all these components. While it is sometimes tempting to weave together the description of the reduction and its analysis, it is usually clearer if you separate the two, and also break down the analysis to its three components of efficiency, completeness, and soundness.

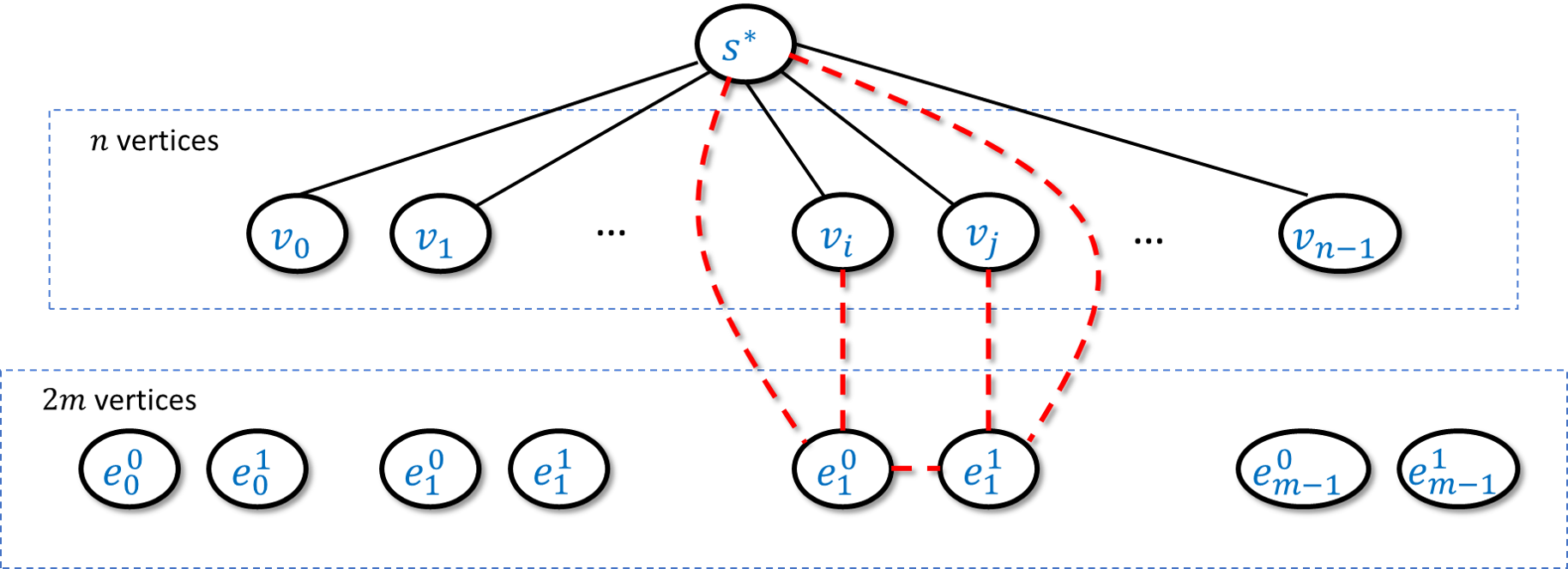
## Reducing Independent Set to Maximum Cut

We now show that the independent set problem reduces to the *maximum cut* (or “max cut”) problem, modeled as the function that on input a pair outputs iff contains a cut of at least edges. Since both are graph problems, a reduction from independent set to max cut maps one graph into the other, but as we will see the output graph does not have to have the same vertices or edges as the input graph.

### 

### 

We will map a graph into a graph such that a large independent set in becomes a partition cutting many edges in . We can think of a cut in as coloring each vertex either “blue” or “red”. We will add a special “source” vertex , connect it to all other vertices, and assume without loss of generality that it is colored blue. Hence the more vertices we color red, the more edges from we cut. Now, for every edge in the original graph we will add a special “gadget” which will be a small subgraph that involves ,, the source , and two other additional vertices. We design the gadget in a way so that if the red vertices are not an independent set in then the corresponding cut in will be “penalized” in the sense that it would not cut as many edges. Once we set for ourselves this objective, it is not hard to find a gadget that achieves it see the proof below. Once again the **takeaway technique** is to use (this time a slightly more clever) gadget.



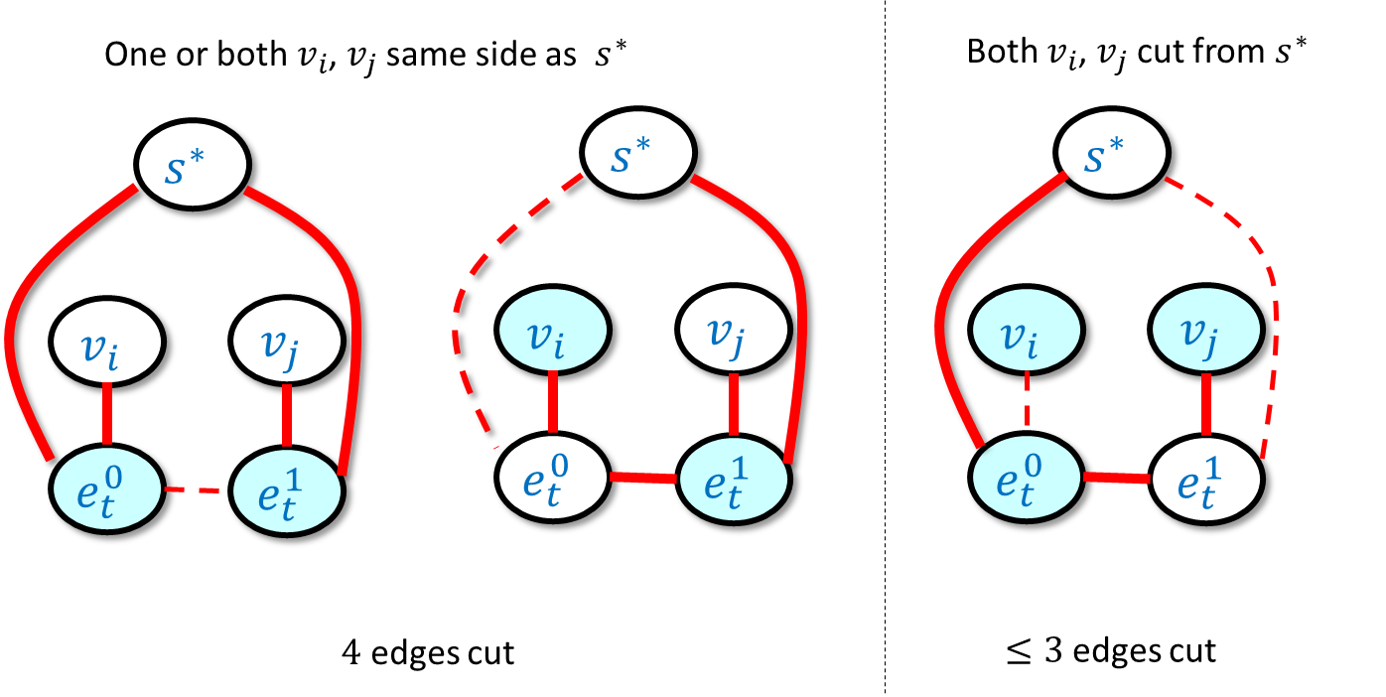
In the reduction of to we map an -vertex -edge graph into the vertex and edge graph as follows. The graph contains a special “source” vertex , vertices , and vertices with each pair corresponding to an edge of . We put an edge between and for every , and if the -th edge of was then we add the five edges . The intent is that if we cut at most one of from then we’ll be able to cut out of these five edges, while if we cut both and from then we’ll be able to cut at most three of them.

We will transform a graph of vertices and edges into a graph of vertices and edges in the following way (see also iset2maxcutoverviewfig). The graph contains all vertices of (though not the edges between them!) and in addition also has:  
\* A special vertex that is connected to all the vertices of   
\* For every edge , two vertices such that is connected to and is connected to , and moreover we add the edges to .

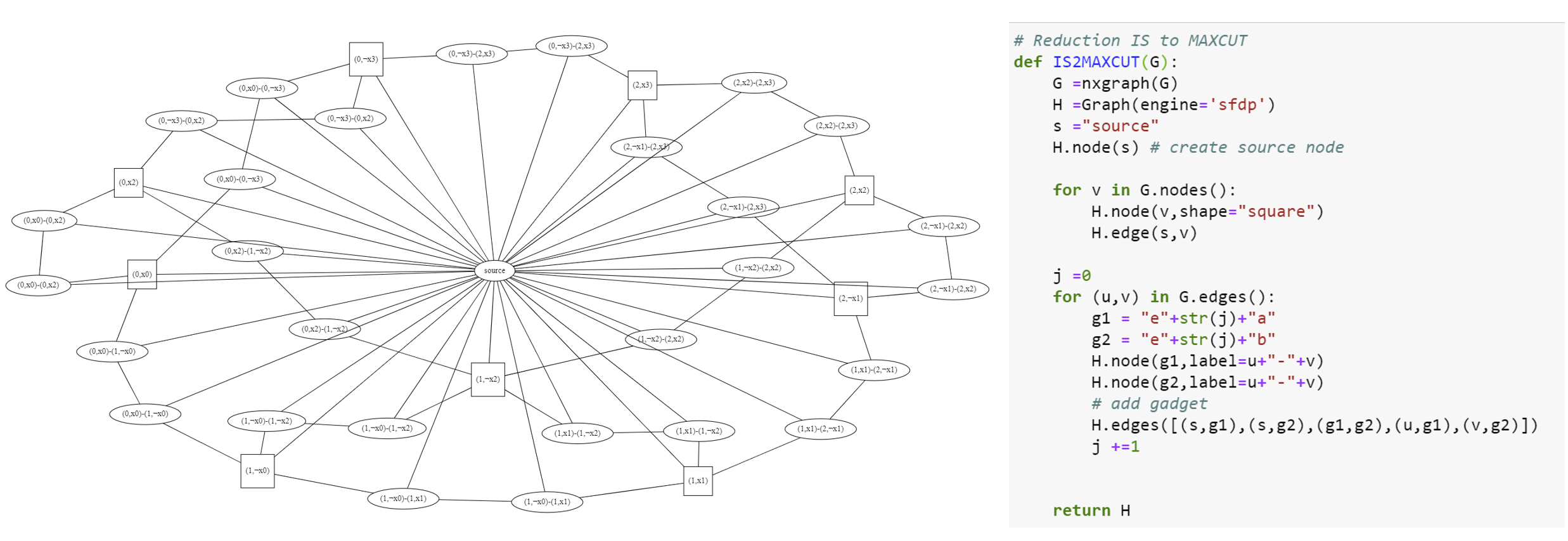
isettomaxcut will follow by showing that contains an independent set of size at least if and only if has a cut cutting at least edges. We now prove both directions of this equivalence:

**Part 1: Completeness.** If is an independent -sized set in , then we can define to be a cut in of the following form: we let contain all the vertices of and for every edge , if and then we add to ; if and then we add to ; and if and then we add both and to . (We don’t need to worry about the case that both and are in since it is an independent set.) We can verify that in all cases the number of edges from to its complement in the gadget corresponding to will be four (see ISETtoMAXCUTfig). Since is not in , we also have edges from to , for a total of edges.

**Part 2: Soundness.** Suppose that is a cut in that cuts at least edges. We can assume that is not in (otherwise we can “flip” to its complement , since this does not change the size of the cut). Now let be the set of vertices in that correspond to the original vertices of . If was an independent set of size then we would be done. This might not always be the case but we will see that if is not an independent set then it’s also larger than . Specifically, we define be the set of edges in that are contained in and let (i.e., if is an independent set then and ). By the properties of our gadget we know that for every edge of , we can cut at most three edges when both and are in , and at most four edges otherwise. Hence the number of edges cut by satisfies . Since we get that . Now we can transform into an independent set by going over every one of the edges that are inside and removing one of the endpoints of the edge from it. The resulting set is an independent set in the graph of size and so this concludes the proof of the soundness condition.



In the reduction of independent set to max cut, for every , we have a “gadget” corresponding to the -th edge in the original graph. If we think of the side of the cut containing the special source vertex as “white” and the other side as “blue”, then the leftmost and center figures show that if and are not both blue then we can cut four edges from the gadget. In contrast, by enumerating all possibilities one can verify that if both and are blue, then no matter how we color the intermediate vertices , we will cut at most three edges from the gadget. The figure above contains only the gadget edges and ignores the edges connecting to the vertices .



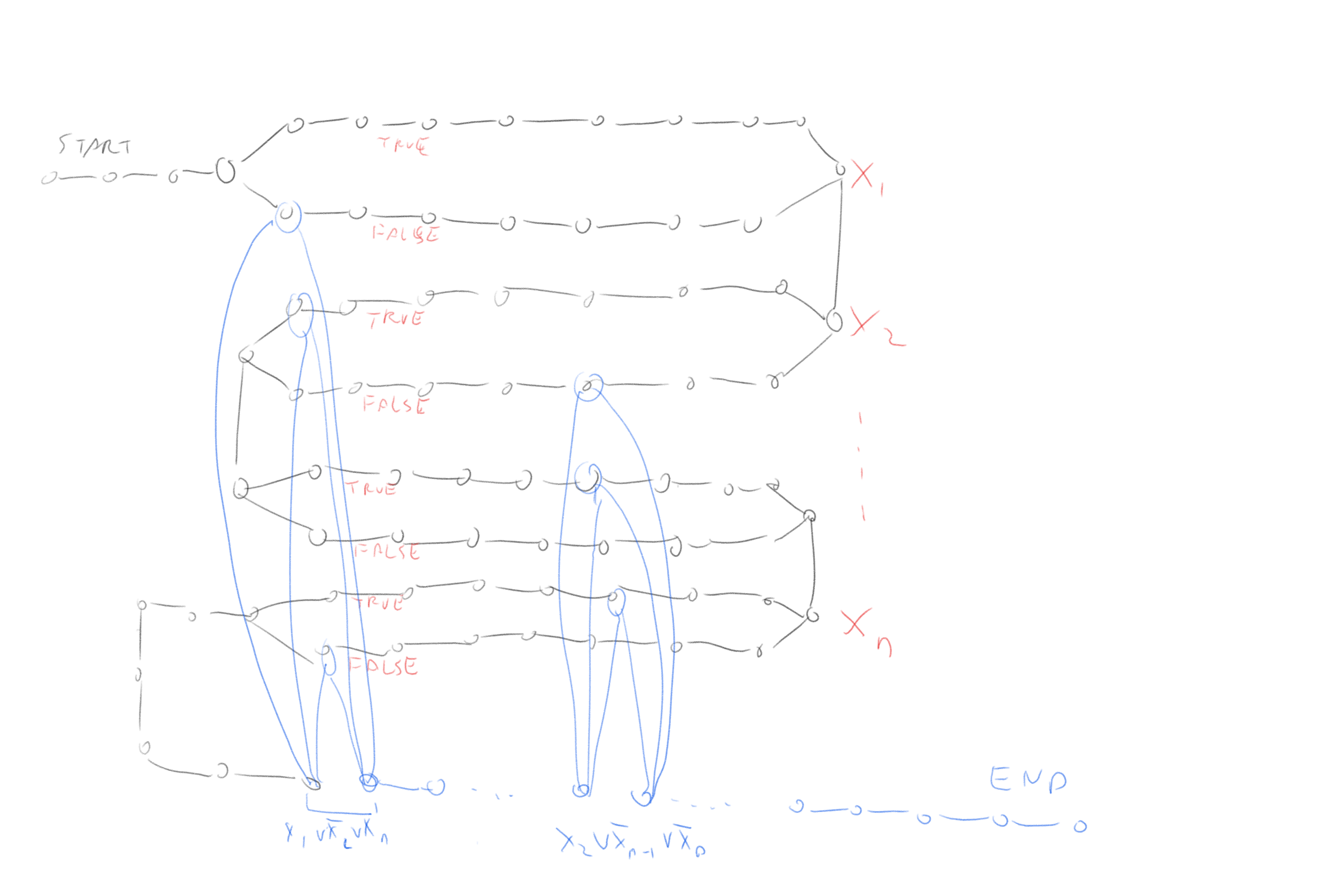
The reduction of independent set to max cut. On the right-hand side is Python code implementing the reduction. On the left-hand side is an example output of the reduction where we apply it to the independent set instance that is obtained by running the reduction of isetnpc on the 3CNF formula .

## Reducing 3SAT to Longest Path

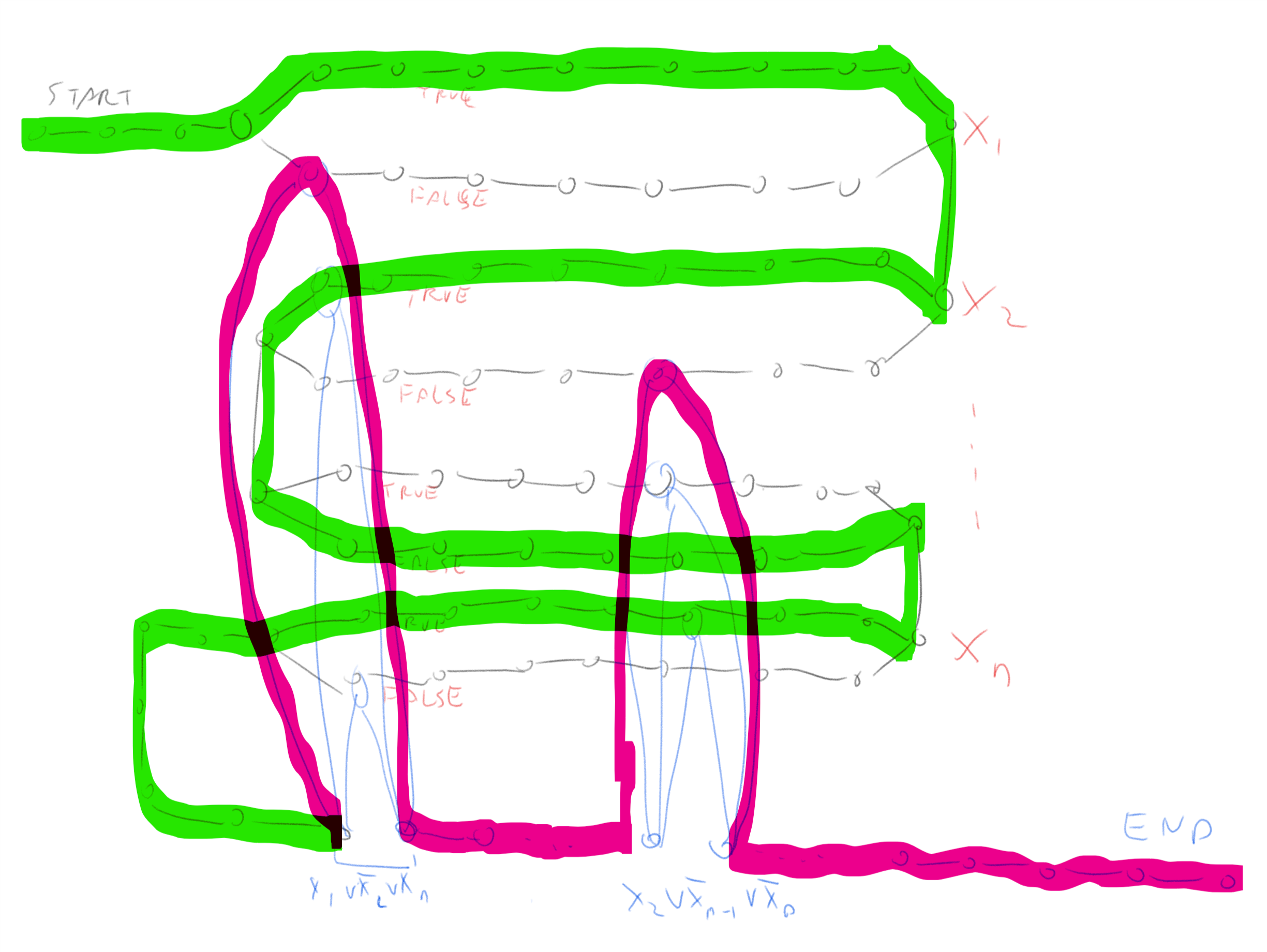
**Note:** This section is still a little messy; feel free to skip it or just read it without going into the proof details. The proof appears in Section 7.5 in Sipser’s book.

One of the most basic algorithms in Computer Science is Dijkstra’s algorithm to find the *shortest path* between two vertices. We now show that in contrast, an efficient algorithm for the *longest path* problem would imply a polynomial-time algorithm for 3SAT.

### 



We can transform a 3SAT formula into a graph such that the longest path in the graph would correspond to a satisfying assignment in . In this graph, the black colored part corresponds to the variables of and the blue colored part corresponds to the vertices. A sufficiently long path would have to first “snake” through the black part, for each variable choosing either the “upper path” (corresponding to assigning it the value True) or the “lower path” (corresponding to assigning it the value False). Then to achieve maximum length the path would traverse through the blue part, where to go between two vertices corresponding to a clause such as , the corresponding vertices would have to have been not traversed before.



The graph above with the longest path marked on it, the part of the path corresponding to variables is in green and part corresponding to the clauses is in pink.

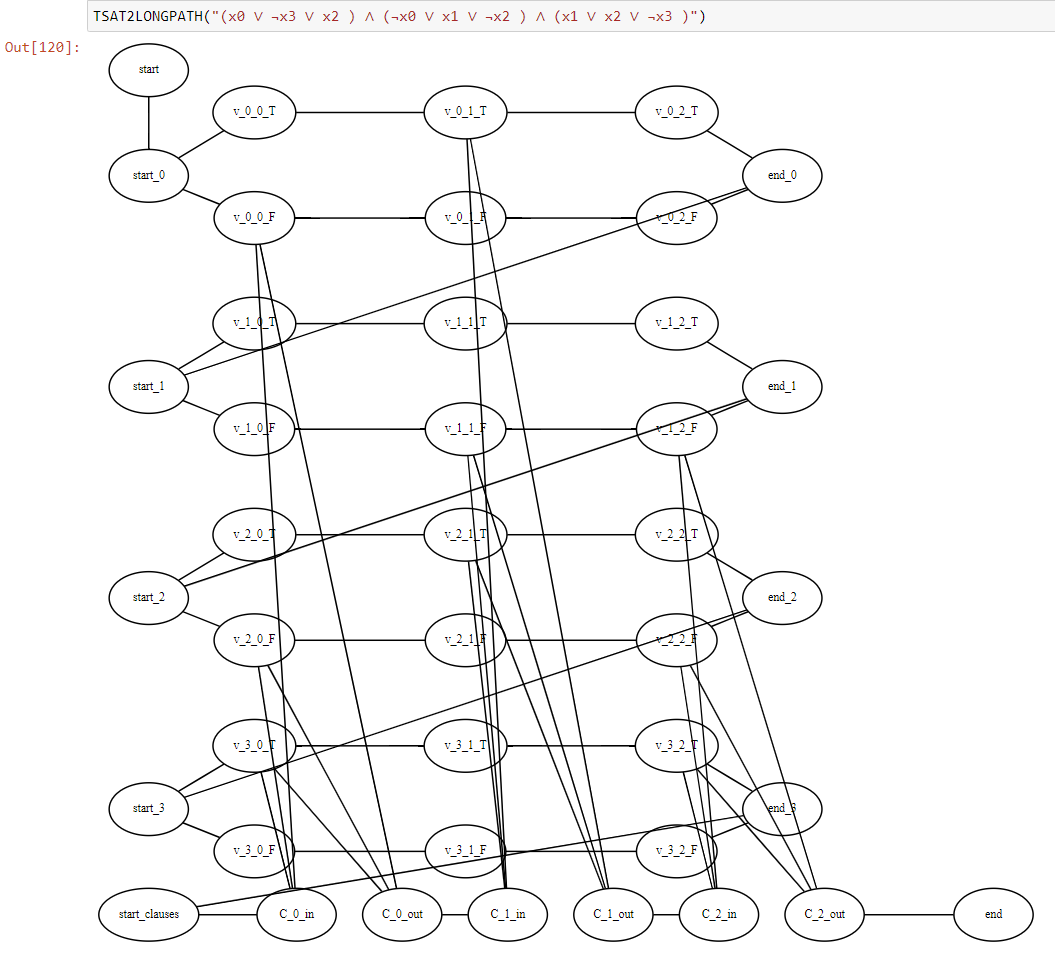
### 

To prove longpaththm need to show how to transform a 3CNF formula into a graph and two vertices such that has a path of length at least if and only if is satisfiable. The idea of the reduction is sketched in longpathfig and longpathfigtwo. We will construct a graph that contains a potentially long “snaking path” that corresponds to all variables in the formula. We will add a “gadget” corresponding to each clause of in a way that we would only be able to use the gadgets if we have a satisfying assignment.

def TSAT2LONGPATH(φ):  
 """Reduce 3SAT to LONGPATH"""  
 def var(v): # return variable and True/False depending if positive or negated  
 return int(v[2:]),False if v[0]=="¬" else int(v[1:]),True  
 n = numvars(φ)  
 clauses = getclauses(φ)  
 m = len(clauses)  
 G =Graph()   
 G.edge("start","start\_0")  
 for i in range(n): # add 2 length-m paths per variable  
 G.edge(f"start\_{i}",f"v\_{i}\_{0}\_T")  
 G.edge(f"start\_{i}",f"v\_{i}\_{0}\_F")  
 for j in range(m-1):   
 G.edge(f"v\_{i}\_{j}\_T",f"v\_{i}\_{j+1}\_T")  
 G.edge(f"v\_{i}\_{j}\_F",f"v\_{i}\_{j+1}\_F")  
 G.edge(f"v\_{i}\_{m-1}\_T",f"end\_{i}")  
 G.edge(f"v\_{i}\_{m-1}\_F",f"end\_{i}")  
 if i<n-1:  
 G.edge(f"end\_{i}",f"start\_{i+1}")  
 G.edge(f"end\_{n-1}","start\_clauses")  
 for j,C in enumerate(clauses): # add gadget for each clause  
 for v in enumerate(C):  
 i,sign = var(v[1])  
 s = "F" if sign else "T"  
 G.edge(f"C\_{j}\_in",f"v\_{i}\_{j}\_{s}")  
 G.edge(f"v\_{i}\_{j}\_{s}",f"C\_{j}\_out")  
 if j<m-1:  
 G.edge(f"C\_{j}\_out",f"C\_{j+1}\_in")  
 G.edge("start\_clauses","C\_0\_in")  
 G.edge(f"C\_{m-1}\_out","end")  
 return G, 1+n\*(m+1)+1+2\*m+1

We build a graph that “snakes” from to as follows. After we add a sequence of long loops. Each loop has an “upper path” and a “lower path”. A simple path cannot take both the upper path and the lower path, and so it will need to take exactly one of them to reach from .

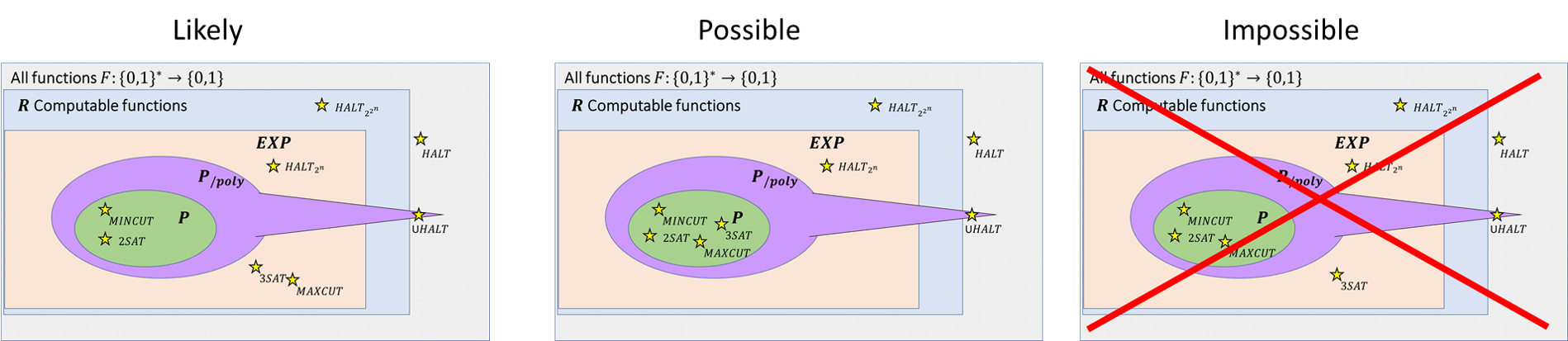
Our intention is that a path in the graph will correspond to an assignment in the sense that taking the upper path in the loop corresponds to assigning and taking the lower path corresponds to assigning . When we are done snaking through all the loops corresponding to the variables to reach we need to pass through “obstacles”: for each clause we will have a small gadget consisting of a pair of vertices that have three paths between them. For example, if the clause had the form then one path would go through a vertex in the lower loop corresponding to , one path would go through a vertex in the upper loop corresponding to and the third would go through the lower loop corresponding to . We see that if we went in the first stage according to a satisfying assignment then we will be able to find a free vertex to travel from to . We link to , to , etc and link to . Thus a satisfying assignment would correspond to a path from to that goes through one path in each loop corresponding to the variables, and one path in each loop corresponding to the clauses. We can make the loop corresponding to the variables long enough so that we must take the entire path in each loop in order to have a fighting chance of getting a path as long as the one corresponds to a satisfying assignment. But if we do that, then the only way if we are able to reach is if the paths we took corresponded to a satisfying assignment, since otherwise we will have one clause where we cannot reach from without using a vertex we already used before.



The result of applying the reduction of to to the formula .

### Summary of relations

We have shown that there are a number of functions for which we can prove a statement of the form “If then ”. Hence coming up with a polynomial-time algorithm for even one of these problems will entail a polynomial-time algorithm for (see for example reductiondiagramfig). In cooklevinchap we will show the inverse direction (“If then ”) for these functions, hence allowing us to conclude that they have *equivalent complexity* to .



So far we have shown that and that several problems we care about such as and are in but it is not known whether or not they are in . However, since we can rule out the possiblity that but . The relation of to the class is not known. We know that does not contain since the latter even contains uncomputable functions, but we do not know whether ot not (though it is believed that this is not the case and in particular that both and are not in ).

* The computational complexity of many seemingly unrelated computational problems can be related to one another through the use of *reductions*.
* If then a polynomial-time algorithm for can be transformed into a polynomial-time algorithm for .
* Equivalently, if and does *not* have a polynomial-time algorithm then neither does .
* We’ve developed many techniques to show that for interesting functions . Sometimes we can do so by using *transitivity* of reductions: if and then .

## Exercises

## Bibliographical notes

Several notions of reductions are defined in the literature. The notion defined in reduction-def is often known as a *mapping reduction*, *many to one reduction* or a *Karp reduction*.

The *maximal* (as opposed to *maximum*) independent set is the task of finding a “local maximum” of an independent set: an independent set such that one cannot add a vertex to it without losing the independence property (such a set is known as a *vertex cover*). Unlike finding a *maximum* independent set, finding a *maximal* independent set can be done efficiently by a greedy algorithm, but this local maximum can be much smaller than the global maximum.

Reduction of independent set to max cut taken from [these notes](https://people.engr.ncsu.edu/mfms/Teaching/CSC505/wrap/Lectures/week14.pdf). Image of Hamiltonian Path through Dodecahedron by [Christoph Sommer](https://commons.wikimedia.org/wiki/File:Hamiltonian_path.svg).

We have mentioned that the line between reductions used for algorithm design and showing hardness is sometimes blurry. An excellent example for this is the area of *SAT Solvers* (see [@gomes2008satisfiability]). In this field people use algorithms for SAT (that take exponential time in the worst case but often are much faster on many instances in practice) together with reductions of the form to derive algorithms for other functions of interest.

1. If you are familiar with matrix notation you may note that such equations can be written as where is an matrix with entries in and . [↑](#footnote-ref-37)