Learning Objectives:

- 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.

15

Polynomial-time reductions

Let us consider several of the problems we have encountered before:

- Finding the longest path in a graph
- Finding the maximum cut in a graph
- The 3SAT problem: deciding whether a given 3CNF formula has a satisfying assignment.
- Solving quadratic equations over n variables $x_0, \ldots, x_{n-1} \in \mathbb{R}$.

All of these 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 problems 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 trivial exponential time algorithms that involve enumerating all possible solutions.
- At the moment, for all these problems the best known algorithms are not much better than the trivial one in the worst case.

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

15.0.1 Decision problems

For reasons of technical conditions rather than anything substantial, we will concern ourselves with *decision problems* (i.e., Yes/No questions) or in other words *Boolean* (i.e., one-bit output) functions. Thus, we will model all the problems as functions mapping $\{0,1\}^*$ to $\{0,1\}$:

- The *3SAT* problem can be phrased as the function $3SAT : \{0,1\}^* \to \{0,1\}$ that maps a 3CNF formula φ to 1 if there exists some assignment x that satisfies it, and to 0 otherwise.[1]
- The *quadratic equations* problem corresponds to the function $QUADEQ : \{0,1\}^* \to \{0,1\}$ that maps a set of quadratic equations E to 1 if there is an assignment x that satisfies all equations, and to 0 otherwise.
- The *longest path* problem corresponds to the function $LONGPATH : \{0,1\}^* \to \{0,1\}$ that maps a graph G and a number k to 1 if there is a simple[2] path in G of length at least k, and maps (G,k) to 0 otherwise. The longest path problem is a generalization of the well-known Hamiltonian Path Problem of determining whether a path of length n exists in a given n vertex graph.
- The *maximum cut* problem corresponds to the function $MAXCUT : \{0,1\}^* \to \{0,1\}$ that maps a graph G and a number k to 1 if there is a cut in G that cuts at least k edges, and maps (G,k) to 0 otherwise.

[1] We assume some representation of formulas as strings, and define the function to output 0 if its input is not a valid representation. We will use the same convention for all the other functions below.

[2] Recall that a *simple* path in a graph is one that does not visit any vertex more than once. For the *shortest path problem* we can assume that a path is simple without loss of generality since removing a loop (a portion of the path that starts from the same vertex and returns to it) only makes the path shorter. For the *longest path problem* we need to make this restriction to avoid "degenerate" paths such as paths that repeat endlessly the same loop.

15.1 Reductions

Suppose that $F, G : \{0,1\}^* \to \{0,1\}$ are two functions. How can we show that they are "computationally equivalent"? The idea is that we show that an efficient algorithm for F would imply an efficient algorithm for G and vice versa. The key to this is the notion of a *reduction*:[3]

Definition 15.1 — Reductions. Let $F, G : \{0,1\}^* \to \{0,1\}^*$. We say that F *reduces to* G, denoted by $F \leq_p G$ if there is a polynomial-time computable $R : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$,

$$F(x) = G(R(x)) \,. \tag{15.1}$$

We say that F and G have *equivalent complexity* if $F \leq_p G$ and $G \leq_p$

[3] Several notions of reductions are defined in the literature. The notion defined in Definition 15.1 is often known as a *mapping reduction*, *many to one reduction* or a *Karp reduction*.

F.

If $F \leq_p G$ and G is computable in polynomial time (i.e., $G \in \mathbf{P}$), then F is computable in polynomial time as well. Indeed, Eq. (15.1) shows how to compute F by applying the polynomial-time reduction R and then the polynomial-time algorithm for computing F. One can think of $F \leq_p G$ as saying that (as far as polynomial-time computation is concerned) F is "easier or equal in difficulty to" G. With this interpretation, we would expect that if $F \leq_p G$ and $G \leq_p H$, then it would hold that $F \leq_p H$; and indeed this is the case:

Lemma 15.2 For every $F, G, H : \{0,1\}^* \to \{0,1\}$, if $F \leq_p G$ and $G \leq_p H$ then $F \leq_p H$.

P We leave the proof of Lemma 15.2 as Exercise 15.2. Pausing now and doing this exercise is an excellent way to verify that you understood the definition of reductions.

R **Polynomial reductions** We have seen reductions before in the context of proving the uncomputability of problems such as $HALTONZERO$ and others. The most crucial difference between the notion in Definition 15.1 and previously occuring notions is that in the context of relating the time complexity of problems, we need the reduction to be computable in *polynomial time*, as opposed to merely computable. Definition 15.1 also restricts reductions to have a very specific format. That is, to show that $F \leq_p G$, rather than allowing a general algorithm for F that uses a "magic box" that computes G, we only allow an algorithm that computes $F(x)$ by outputting $G(R(x))$. This restricted form is convenient for us, but people have defined and used more general reductions as well.

Since both F and G are Boolean functions, the condition $F(x) = G(R(x))$ in Eq. (15.1) is equivalent to the following two impliciations: **(i)** if $F(x) = 1$ then $G(R(x)) = 1$, and **(ii)** if $G(R(x)) = 1$ then $F(x) = 1$. Traditionally, condition **(i)** is often known as *completness* and condition **(ii)** is often known as *soundness*. We can think of this as saying that the reduction R is *complete* if every 1-input of F (i.e. x such that $F(x) = 1$) is mapped by R to a 1-input of G, and that it is *sound* if no 0-input of F will ever be mapped to a 1-input of G. As we will see below, it is often the case that establishing **(ii)** is the more challenging part.

15.2 Some example reductions

We will now use reductions to relate the computational complexity of the problems mentioned above – 3SAT, Quadratic Equations, Maximum Cut, and Longest Path. We start by reducing 3SAT to the latter three problems, demonstrating that solving any one of them will solve 3SAT. Along the way we will introduce one more problem: the *independent set* problem. Like the others, it shares the characteristics that it is an important and well-motivated computational problem, and that the best known algorithm for it takes exponential time. In the next lecture we will show the other direction: reducing each one of these problems to 3SAT in one fell swoop.

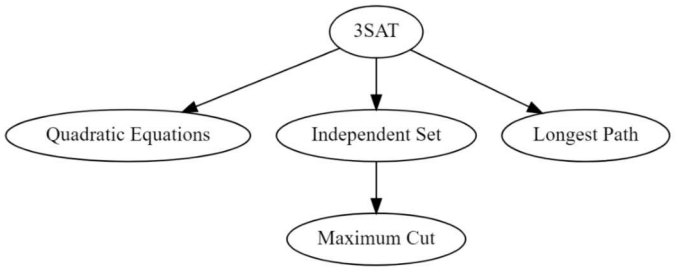

Figure 15.1: Our first stage in showing equivalence is to reduce 3SAT to the three other problems

15.2.1 Reducing 3SAT to quadratic equations

Let us now see our first example of a reduction. Recall that in the *quadratic equation* problem, the input is a list of n-variate polynomials $p_0, \ldots, p_{m-1} : \mathbb{R}^n \to \mathbb{R}$ that are all of degree at most two (i.e., they are *quadratic*) and with integer coefficients.[4] The task is to find out whether there is a solution $x \in \mathbb{R}^n$ to the equations $p_0(x) = 0$, $p_1(x) = 0, \ldots, p_{m-1}(x) = 0$.

[4] The latter condition is for convenience and can be achieved by scaling.

For example, the following is a set of quadratic equations over the variables x_0, x_1, x_2:

$$\begin{aligned} x_0^2 - x_0 &= 0 \\ x_1^2 - x_1 &= 0 \\ x_2^2 - x_2 &= 0 \\ 1 - x_0 - x_1 + x_0 x_1 &= 0 \end{aligned} \tag{15.2}$$

You can verify that $x \in \mathbb{R}^3$ satisfies this set of equations if and only if $x \in \{0,1\}^3$ and $x_0 \vee x_1 = 1$.

We will show how to reduce 3SAT to the problem of Quadratic Equations.

Theorem 15.3 — Hardness of quadratic equations.

$$3SAT \leq_p QUADEQ \tag{15.3}$$

where $3SAT$ is the function that maps a 3SAT formula φ to 1 if it is satisfiable and to 0 otherwise, and $QUADEQ$ is the function that maps a set E of quadratic equations over $\{0,1\}^n$ to 1 it has a solution and to 0 otherwise.

Proof Idea: At the end of the day, a 3SAT formula can be thought of as a list of equations on some variables $x_0, \ldots, x_{n-1}$. Namely, the equations are that each of the x_i's should be equal to either 0 or 1, and that the variables should satisfy some set of constraints which corresponds to the OR of three variables or their negation. To show that $3SAT \leq_p QUADEQ$ we need to give a polynomial-time reduction that maps a 3SAT formula φ into a set of quadratic equations E such that E has a solution if and only if φ is satisfiable. The idea is that we can transform a 3SAT formula φ first to a set of *cubic* equations by mapping every constraint of the form $(x_{12} \vee \overline{x}_{15} \vee x_{24})$ into an equation of the form $(1-x_{12})x_{15}(1-x_{24}) = 0$. We can then turn this into a *quadratic equation* by mapping any cubic equation of the form $x_i x_j x_k = 0$ into the two quadratic equations $y_{i,j} = x_i x_j$ and $y_{i,j} x_k = 0$.

Proof of Theorem 15.3. To prove Theorem 15.3 we need to give a polynomial-time transformation of every 3SAT formula φ into a set of quadratic equations E, and prove that $3SAT(\varphi) = QUADEQ(E)$.

We now describe the transformation of a formula φ to equations E and show the completeness and soundness conditions. Recall that a *3SAT formula* φ is a formula such as $(x_{17} \vee \overline{x}_{101} \vee x_{57}) \wedge (x_{18} \vee \overline{x}_{19} \vee \overline{x}_{101}) \wedge \cdots$. That is, φ is composed of the AND of m *3SAT clauses* where a 3SAT clause is the OR of three variables or their negation. A *quadratic equations* instance E is composed of a list of equations, each of involving a sum of variables or their products, such as $x_{19}x_{52} - x_{12} + 2x_{33} = 2$, etc.. We will include the constraints $x_i^2 - x_i = 0$ for every $i \in [n]$ in our equations, which means that we can restrict attention to assignments where $x_i \in \{0,1\}$ for every i.

There is a natural way to map a 3SAT instance into a set of *cubic* equations, and that is to map a clause such as $(x_{17} \vee \overline{x}_{101} \vee x_{57})$ (which is equivalent to the negation of $\overline{x}_{17} \wedge x_{101} \wedge \overline{x}_{57}$) to the equation $(1-x_{17})x_{101}(1-x_{57}) = 0$. We can map a formula φ with m clauses into a set E of m such equations such that there is an x with $\varphi(x) = 1$

if and only if there is an assignment to the variables that satisfies all the equations of E. To make the equations *quadratic* we introduce for every $i, j \in [n]$ a variable $y_{i,j}$ and include the constraint $y_{i,j} - x_i x_j = 0$ in the equations. This is a quadratic equation that ensures that $y_{i,j} = x_i x_j$ for every $i, j \in [n]$. Now we can turn any cubic equation in the x's into a quadratic equation in the x and y variables. For example, we can "open up the parentheses" of an equation such as $(1 - x_{17})x_{101}(1 - x_{57}) = 0$ to $x_{101} - x_{17}x_{101} - x_{101}x_{57} + x_{17}x_{101}x_{57} = 0$. We can now replace the cubic term $x_{17}x_{101}x_{57}$ with the quadratic term $y_{17,101}x_{57}$. This can be done for every cubic equation in the same way, replacing any cubic term $x_i x_j x_k$ with the term $y_{i,j}x_k$. The bottom line is that we get a set E of quadratic equations in the variables $x_0, \ldots, x_{n-1}, y_{0,0}, \ldots, y_{n-1,n-1}$ such that the 3SAT formula φ is satisfiable if and only if the equations E have a solution. ■

15.3 The independent set problem

For a graph $G = (V, E)$, an independent set (also known as a *stable set*) is a subset $S \subseteq V$ such that there are no edges with both endpoints in S (in other words, $E(S, S) = \emptyset$). 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.[5] 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. But it also arises in very different settings, including trying to find structure in protein-protein interaction graphs.[6]

To phrase independent set as a decision problem, we think of it as a function $ISET : \{0,1\}^* \to \{0,1\}$ that on input a graph G and a number k outputs 1 if and only if the graph G contains an independent set of size at least k. We will now reduce 3SAT to Independent set.

Theorem 15.4 — Hardness of Independent Set. $3SAT \leq_p ISET$.

Proof Idea: 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 "$x_{17} = 0$" and "$x_{17} = 1$" as two conflicting events, and of the constraints $x_{17} \vee \overline{x}_5 \vee x_9$ as creating a conflict between the events "$x_{17} = 0$", "$x_5 = 1$" and "$x_9 = 0$", saying

[5] While we will not consider it here, people have also looked at the *maximal* (as opposed to *maximum*) independent set, which is the task of finding a "local maximum" of an independent set: an independent set S such that one cannot add a vertex to it without losing the independence property (such a set is known as a *vertex cover*). 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.

[6] In the molecular biology literature, people often refer to the computationally equivalent clique problem.

that these three cannot simultaneosly 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.

Proof of Theorem 15.4. Given a 3SAT formula φ on n variables and with m clauses, we will create a graph G with $2n + 3m$ vertices as follows: (see Fig. 15.2 for an example)

- For every variable x_i of φ, we create a pair of vertices that are labeled "$x_i = 0$" and "$x_i = 1$", and put an edge between them. Note that this means that every independent set S in the graph can contain at most one of those vertices.

- For every clause in φ involving the variables x_i, x_j, x_k, note that the clause is the OR of these three variables or their negations and so there are some $a, b, c \in \{0,1\}$ such that the clause is not satisfied if and only if $x_i = a$, $x_j = b$, and $x_k = c$. We add three vertices to the graph with the labels "$x_i \neq a$", "$x_j \neq b$" and "$x_k \neq c$" and connect them with a triangle. This means that every independent set S in the graph can contain at most one of the members of this triangle. Moreover, we put an edge between the vertex labeled "$x_i \neq a$" and the vertex we labeled "$x_i = a$" which means that no independent set in the graph can contain both of them, and add analogous edges connecting "$x_j \neq b$" with "$x_j = b$" and "$x_k \neq c$" with "$x_k = c$".

The construction of G based on φ can clearly be carried out in polynomial time. Hence to prove the theorem we need to show that φ is satisfiable if and only if G contains an independent set of $n + m$ vertices. We now show both directions of this equivalence:

- **Completeness:** The "completeness" direction is to show that if φ has a satisfying assignment x^*, then G has an independent set S^* of $n + m$ vertices. Let us now show this. Indeed, suppose that φ has a satisfying assignment $x^* \in \{0,1\}^n$. Then there exists an $n + m$-vertex independent set S^* in G that is constructed as follows: for every i, we include in S^* the vertex labeled "$x_i = x^*_i$" and for every clause we choose one of the vertices in the clause whose label agrees with x^* (i.e., a vertex of the form $x_j \neq b$ where $b \neq x^*_j$) and add it to S^*. There must exist such a vertex as otherwise it would hold that x^* does not satisfy this clause (can you see why?). Moreover, such a vertex would not have a neighbor in S^* since we don't add any other vertex from the triangle and in the case above the vertex "$x_j = b$" is not a member of S^* since $b \neq x^*$. Since we added one vertex per variable and one vertex per clause, we

get that S^* has $n+m$ vertices, and by the reasoning above it is an independent set.

- **Soundness:** The "soundness" direction is to show that if G has an independent set S^* of $n+m$ vertices, then φ has a satisfying assignment $x^* \in \{0,1\}^n$. Let us now show this. Indeed, suppose that G has an independent set S^* with $n+m$ vertices. Out of the $2n$ vertices corresponding to variables and their negation, S^* can contain at most n, since otherwise it would contain a neighboring pair of vertices of the form "$x_i = 0$" and "$x_i = 1$". Out of the $3m$ vertices corresponding to clauses, S^* can contain at most m since otherwise it would contain a pair of vertices inside the same triangle. Hence the only way S^* has $n+m$ vertices is if it contains exactly n vertices out of those correpsonding to variables and exactly m vertices out of those corresponding to clauses. Now define $x^* \in \{0,1\}^n$ as follows: for every $i \in [n]$, we set $x^*_i = a$ if the vertex "$x_i = a$" is in S^* (since S^* is an independent set and contains n of these vertices, it will contain exactly one vertex of this form). We claim that x^* is a satisfying assignment for φ. Indeed, suppose not, and for every clause of φ, let $x_i = a, x_j = b, x_k = c$ be the assignment that is "forbidden" by this clause. Then since S^* contains one vertex out of "$x_i \neq a$" , "$x_j \neq b$" and "$x_k \neq c$" it must be that x^* does not match this assignment, as otherwise S^* would not be an independent set.

  This completes the proof of Theorem 15.4 ■

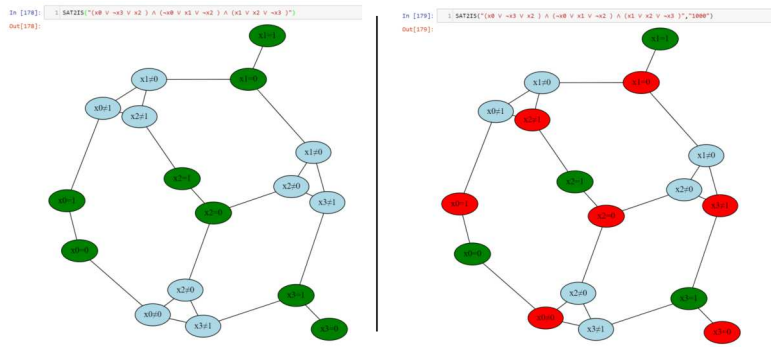

Figure 15.2: An example of the reduction of $3SAT$ to IS. We reduce the formula of 4 variables and 3 clauses into a graph of $8 + 3 \cdot 3 = 17$ vertices. On the lefthand side is the resulting graph, where the green vertices correspond to the variable and the light blue vertices correspond to the clauses. On the righthand side is the 7 vertex independent set corresponding to a particular satisfying assignment. This example is taken from a jupyter notebook, the code of which is available.

15.4 *Reducing Independent Set to Maximum Cut*

Theorem 15.5 — Hardness of Max Cut. $ISET \leq_p MAXCUT$

Proof Idea: We will map a graph G into a graph H such that a large independent set in G becomes a partition cutting many edges in H. We can think of a cut in H as coloring each vertex either "blue" or "red". We will add a special "source" vertex s^*, 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 s^* we cut. Now, for every edge u, v in the original graph G we will add a special "gadget" which will be a small subgraph that involves u,v, the source s^*, and two other additional vertices. We design the gadget in a way so that if the red vertices are not an independent set in G then the corresponding cut in H 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.

Proof of Theorem 15.5. We will transform a graph G of n vertices and m edges into a graph H of $n + 1 + 2m$ vertices and $n + 5m$ edges in the following way: the graph H will contain all vertices of G (though not the edges between them!) and in addition to that will contain:

* A special vertex s^* that is connected to all the vertices of G

* For every edge $e = \{u, v\} \in E(G)$, two vertices e_0, e_1 such that e_0 is connected to u and e_1 is connected to v, and moreover we add the edges $\{e_0, e_1\}, \{e_0, s^*\}, \{e_1, s^*\}$ to H.

Theorem 15.5 will follow by showing that G contains an independent set of size at least k if and only if H has a cut cutting at least $k + 4m$ edges. We now prove both directions of this equivalence:

- **Completeness:** If I is an independent k-sized set in G, then we can define S to be a cut in H of the following form: we let S contain all the vertices of I and for every edge $e = \{u, v\} \in E(G)$, if $u \in I$ and $v \notin I$ then we add e_1 to S; if $u \notin I$ and $v \in I$ then we add e_0 to S; and if $u \notin I$ and $v \notin I$ then we add both e_0 and e_1 to S. (We don't need to worry about the case that both u and v are in I since it is an independent set.) We can verify that in all cases the number of edges from S to its complement in the gadget corresponding to e will be four (see Fig. 15.3). Since s^* is not in S, we also have k edges from s^* to I, for a total of $k + 4m$ edges.
- **Soundness:** Suppose that S is a cut in H that cuts at least $C = k + 4m$ edges. We can assume that s^* is not in S (otherwise we can "flip" S to its complement $\overline{S}$, since this does not change the size

of the cut). Now let I be the set of vertices in S that correspond to the original vertices of G. If I was an independent set of size k then would be done. This might not always be the case but we will see that if I is not an independent set then its also larger than k. Specifically, we define $m_{in} = |E(I,I)|$ be the set of edges in G that are contained in I and let $m_{out} = m - m_{in}$ (i.e., if I is an independent set then $m_{in} = 0$ and $m_{out} = m$). By the properties of our gadget we know that for every edge $\{u,v\}$ of G, we can cut at most three edges when both u and v are in S, and at most four edges otherwise. Hence the number C of edges cut by S satisfies $C \leq |I| + 3m_{in} + 4m_{out} = |I| + 3m_{in} + 4(m - m_{in}) = |I| + 4m - m_{in}$. Since $C = k + 4m$ we get that $|I| - m_{in} \geq k$. Now we can transform I into an independent set I' by going over every one of the m_{in} edges that are inside I and removing one of the endpoints of the edge from it. The resulting set I' is an indepenent set in the graph G of size $|I| - m_{in} \geq k$ and so this concludes the proof of the soundness condition.

■

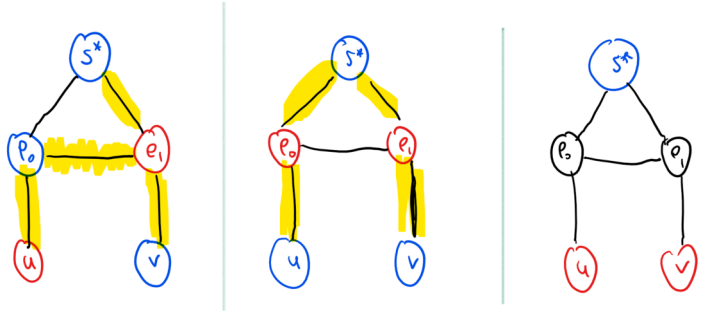

Figure 15.3: In the reduction of independent set to max cut, we have a "gadget" corresponding to every edge $e = \{u,v\}$ in the original graph. If we think of the side of the cut containing the special source vertex s^* as "blue" and the other side as "red", then the leftmost and center figures show that if u and v are not both red then we can cut four edges from the gadget. In contrast, by enumerating all possiblities one can verify that if both u and v are red, then no matter how we color the intermediate vertices e_0, e_1, we will cut at most three edges from the gadget.

15.5 *Reducing 3SAT to Longest Path*

7

[7] This section is still a little messy, feel free to skip it or just read it without going into the proof details

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.

Theorem 15.6 — Hardness of longest path.

$$3SAT \leq_p LONGPATH \tag{15.4}$$

Proof Idea: To prove Theorem 15.6 need to show how to transform a 3CNF formula φ into a graph G and two vertices s, t such that G has a path of length at least k if and only if φ is satisfiable. The idea of the reduction is sketched in Fig. 15.4 and Fig. 15.5. 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 φ in a way that we would only be able to use the gadgets if we have a satisfying assignment.

Proof of Theorem 15.6. We build a graph G that "snakes" from s to t as follows. After s we add a sequence of n 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 s from t.

Our intention is that a path in the graph will correspond to an assignment $x \in \{0,1\}^n$ in the sense that taking the upper path in the i^{th} loop corresponds to assigning $x_i = 1$ and taking the lower path corresponds to assigning $x_i = 0$. When we are done snaking through all the n loops corresponding to the variables to reach t we need to pass through m "obstacles": for each clause j we will have a small gadget consisting of a pair of vertices s_j, t_j that have three paths between them. For example, if the j^{th} clause had the form $x_{17} \vee \overline{x}_{55} \vee x_{72}$ then one path would go through a vertex in the lower loop corresponding to x_{17}, one path would go through a vertex in the upper loop corresponding to x_{55} and the third would go through the lower loop corresponding to x_{72}. 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 s_j to t_j. We link t_1 to s_2, t_2 to s_3, etc and link t_m to t. Thus a satisfying assignment would correspond to a path from s to t 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 t is if the paths we took corresponded to a satisfying assignment, since otherwise we will have one clause j where we cannot reach t_j from s_j without using a vertex we already used before. ■

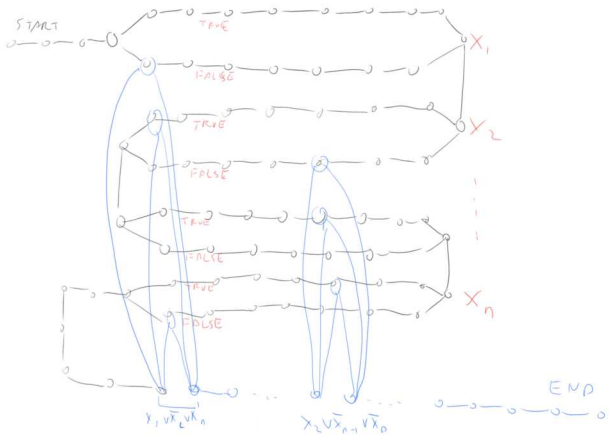

Figure 15.4: We can transform a 3SAT formula φ into a graph G such that the longest path in the graph G would correspond to a satisfying assignment in φ. In this graph, the black colored part corresponds to the variables of φ and the blue colored part corresponds to the vertices. A sufficiently long 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 $x_{17} \vee \overline{x}_{32} \vee x_{57}$, the corresponding vertices would have to have been not traversed before.

15.6 Exercises

8

Exercise 15.1 Prove **??** ■

Exercise 15.2 — Transitivity of reductions. Prove that if $F \leq_p G$ and $G \leq_p H$ then $F \leq_p H$. ■

[8] TODO: Maybe mention either in exercise or in body of the lecture some NP hard results motivated by science. For example, shortest superstring that is motivated by genome sequencing, protein folding, maybe others.

15.7 Bibliographical notes

Reduction of independent set to max cut taken from these notes.

15.8 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

15.9 Acknowledgements

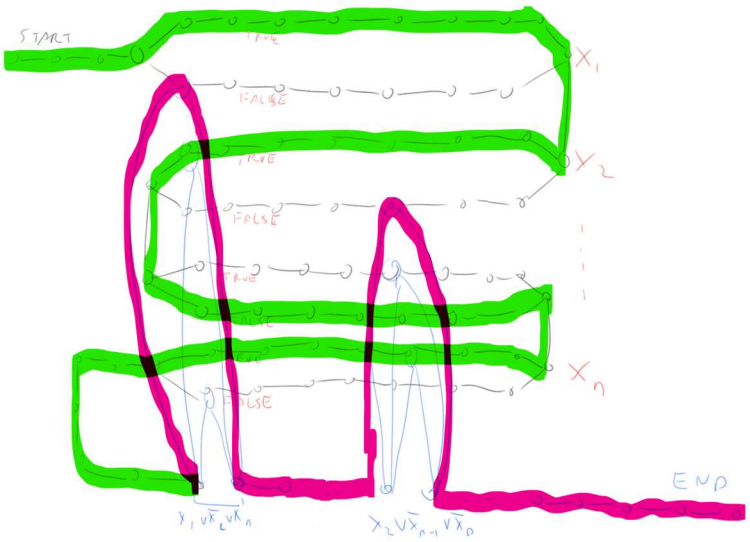

Figure 15.5: The graph above with the longest path marked on it, the part of the path corresponding to variables is in green and part corresponding to the clauses is in pink.