CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 17 : Polynomial time reductions

We study polynomial time reductions, which would allow us to compare the difficulty of different problems.

Polynomial Time Reductions

Once we have learned more and more algorithms, they become our building blocks and we may not need to design algorithms for new problems from scratch every time.

So it becomes more and more important to be able to use existing algorithms to solve new problems.

We have already seen some reductions.

For instances, we have reduced subset-sum to knapsack, longest increasing subsequence to longest common subsequence, and baseball / basketball league winner to maximum bipartite matching, etc.

In general, if there is an efficient reduction from problem A to problem B and there is an efficient algorithm to solve problem B, then we have an efficient algorithm to solve problem A.

Decision Problems

To formalize the notion of a reduction, it is more convenient to restrict our attention to decision problems, for which the output is just YES or NO, so that every problem has the same output format.

For example, instead of finding a maximum matching, we consider the decision version of the problem "Does G have a matching of size at least k?".

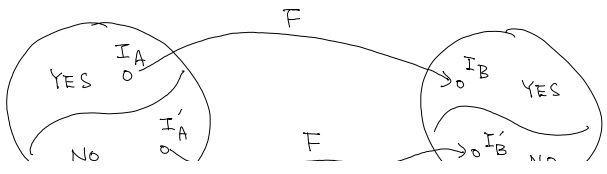
As we will discuss later, for all the problems that we will consider, if we know how to solve the decision version of our problem in polynomial time, then we can use the decision algorithm as a blackbox / subroutine to solve the search version of our problem in polynomial time.

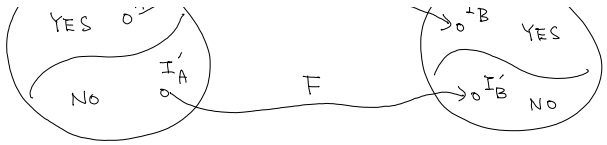
Definition (polynomial time reductions)

We say a decision problem A is polynomial time reducible to a decision problem B if there is a polynomial time algorithm F that maps / transforms any instance I_A of A into an instance I_B of B (that is, $F(I_A) = I_B$) such that

I_A is a YES instance of problem A if and only if I_B is a YES instance of problem B.

We use the notation $A \leq_p B$ to denote that such a reduction exists, intuitively saying that problem A is not more difficult than B in terms of polynomial time solvability. □





Now, suppose we have such a polynomial time reduction algorithm F and a polynomial time algorithm ALG_B to solve problem B, then we have the following polynomial time algorithm to solve problem A

Algorithm (solving problem A by reduction)

Input: an instance I_A of problem A

Output: whether I_A is a YES-instance

1. Use the reduction algorithm F to map/transform I_A into $I_B = F(I_A)$ of problem B.
2. Return $ALG_B(I_B)$.

Correctness follows from the property of the reduction algorithm F that I_A is a YES-instance of problem A iff I_B is a YES-instance of problem B, and the correctness of ALG_B to solve problem B.

Time Complexity Suppose F has time complexity $p(n)$ for an instance I_A of size n where $p(n)$ is a polynomial in n, and ALG_B has time complexity $q(m)$ for an instance I_B of size m where $q(m)$ is a polynomial in m.

Then the above algorithm has time complexity $q(p(n))$, a polynomial in the input size n.

Proving Hardness using Reductions

So far it is all familiar: We reduce problem A to problem B efficiently, and use an efficient algorithm for problem B to obtain an efficient algorithm to solve problem A.

Now, we explore the other implication of the inequality $A \leq_p B$.

Suppose problem A is known to be impossible to be solved in polynomial time.

Then $A \leq_p B$ implies that B cannot be solved in polynomial time either, as otherwise we can solve problem A in polynomial time using the reduction algorithm proven above.

Therefore, if A is computationally hard and $A \leq_p B$, then B is also computationally hard.

By our current knowledge, however, we know almost nothing about proving a problem cannot be solved in polynomial time, and so we could not draw such a strong conclusion from $A \leq_p B$.

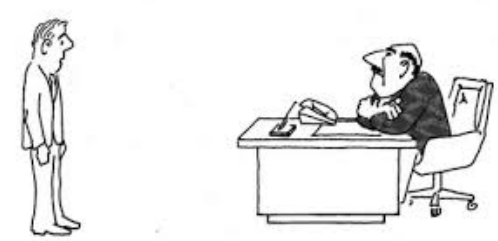
But suppose there is a problem, say the traveling salesman problem in L14, which is very famous and

has attracted many brilliant researchers to solve it in polynomial but without any success.

Now our boss gives us a problem C, and we couldn't solve it in polynomial time.

Instead of just saying that we couldn't solve it in polynomial time, it would be much more convincing if we could prove that $TSP \leq_p C$.

The following is a cartoon from the classical book about NP-completeness by Garey and Johnson.



"I can't find an efficient algorithm, I guess I'm just too dumb."

"I can't find an efficient algorithm, but neither can all these famous people."

This is what we will be doing in these few lectures!

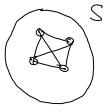
Simple Reductions

We will show that the following three problems are equivalent in terms of polynomial-time solvability. either they all can be solved in polynomial time or they all cannot be solved in polynomial time.

Maximum Clique (Clique) A subset of vertices $S \subseteq V$ is a clique if $uv \in E \ \forall u,v \in S$.

Input: Graph $G = (V, E)$, an integer k.

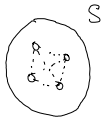
Output: Is there a clique in G with at least k vertices?



Maximum Independent Set (IS) A subset of vertices $S \subseteq V$ is an independent set if $uv \notin E \ \forall u,v \in S$.

Input: Graph $G = (V, E)$, an integer k.

Output: Is there an independent set in G with at least k vertices?



Minimum Vertex Cover (VC) A subset of vertices $S \subseteq V$ is a vertex cover if $\{u,v\} \cap S \neq \emptyset \ \forall uv \in E$.

Input: Graph $G = (V, E)$, an integer k.

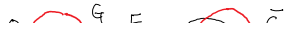
Output: Is there a vertex cover in G with at most k vertices?

Proposition Clique $\leq_p$ IS and IS $\leq_p$ Clique.

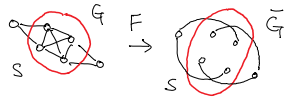
Proof Clique and IS are very similar, one wants to find at least k vertices with all edges in between and one wants to find at least k vertices with no edges in between.

So, to reduce Clique to IS, we just need to change edges to non-edges and vice-versa.

More formally, to solve Clique on $G = (V,E)$, the reduction algorithm F would construct the



More formally, to solve Clique on $G=(V,E)$, the reduction algorithm F would construct the complement graph $\bar{G}=(V,\bar{E})$, where $uv \in E$ if and only if $uv \notin E$.



Clearly, the reduction algorithm runs in polynomial time, actually in linear time.

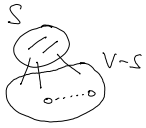
It should also be clear from the construction that S is a clique in G iff S is an independent set in $\bar{G}$.

Therefore, $\{G,k\}$ is an YES-instance for Clique iff $\{\bar{G},k\}$ is an YES-instance for IS. □

To see the connection between independent sets and vertex covers, we need the following observation.

Observation In $G=(V,E)$, $S \subseteq V$ is a vertex cover of G iff $V-S$ is an independent set in G.

Proof If S is a vertex cover, then $V-S$ is an independent set, because if there is an edge e between two vertices in $V-S$, then $S \cap e = \phi$ and S is not a vertex cover.



Similarly, if $V-S$ is an independent set, then all the edges in G have at least one endpoint in S, and thus S is a vertex cover. □

Proposition $VC \leq_p IS$ and $IS \leq_p VC$.

Proof The observation above states that G has a vertex cover of size at most k if and only if G has an independent set of size at least $n-k$.

So, the reduction algorithm simply maps $\{G,k\}$ for VC to $\{G,n-k\}$ for IS, and clearly it runs in polynomial time and it maps YES/NO instances for VC to YES/NO instances for IS respectively. □

Note that polynomial time reductions are transitive.

Lemma If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Proof If there exists a polynomial time algorithm F that maps instances of A to that of B, and a polytime algorithm H that maps B to C, then $H \circ F$ maps A to C in polynomial time.

Also, $H \circ F$ has the property that it maps YES/NO-instances of A to YES/NO-instances of C respectively. □

Therefore, Clique, IS and VC are equivalent in polynomial time solvability.

More Simple Reductions

Hamiltonian Cycle (HC) A cycle is a Hamiltonian cycle if it touches every vertex exactly once.

Input: Undirected graph $G=(V,E)$.

Output: Does G have a Hamiltonian cycle?

Hamiltonian Path (HP) A path is a Hamiltonian path if it touches every vertex exactly once.

Hamiltonian Path (HP) A path is a Hamiltonian path if it touches every vertex exactly once.

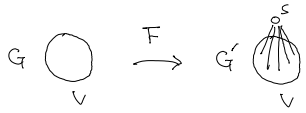
Input: Undirected graph $G = (V, E)$.

Output: Does G have a Hamiltonian path?

It is not surprising that these two problems are equivalent in terms of polynomial time computation, but it is a good exercise to work out the reductions.

Proposition $HP \leq_p HC$.

Proof Given $G = (V, E)$ for HP, we construct $G' = (V+s, E')$ by copying G and adding a new vertex s that connects to every vertex in V.



Clearly, the reduction runs in polynomial time.

We claim that G has a Hamiltonian path iff G' has a Hamiltonian cycle.

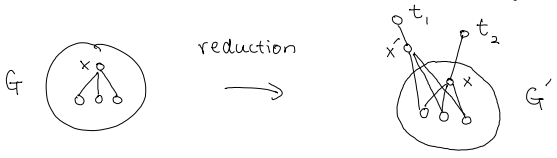
$\Rightarrow$) If G has a Hamiltonian path P with endpoints a and b, then $P+sa+sb$ is a Hamiltonian cycle in G'.

$\Leftarrow$) If G' has a Hamiltonian cycle C, then there are two edges incident on s, call them sa and sb, then $C-sa-sb$ is a Hamiltonian path in G. □

The other direction is slightly more interesting.

Proposition $HC \leq_p HP$.

Proof Given $G = (V, E)$ for HC, we pick an arbitrary vertex $x \in V$, and construct G' by adding a duplicate x' of x and two new degree one vertex t_1 and t_2 as shown in the picture

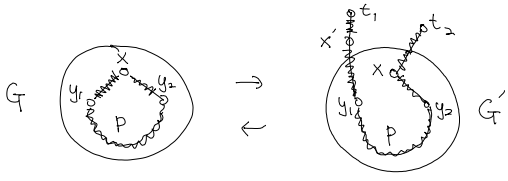


Clearly the reduction runs in polynomial time.

We need to prove that G has a Hamiltonian cycle iff G' has a Hamiltonian path.

$\Rightarrow$) If there is a Hamiltonian cycle in G, then there is a Hamiltonian path in G' by replacing xy_1 by $x'y_1$ and $x't_1$ and also adding xt_2.

$\Leftarrow$) If there is a Hamiltonian path in G', then the two endpoints must be t_1 and t_2 since they are degree one vertices.



Then, t_1x' must be in the path, and call the other neighbor of x' in the path be y_1.

Since x' is a duplicate of x, we know that x has an edge to y_1 in G.

We also know that t_2x must be in the Hamiltonian path, since t_2 is a degree one vertex.

So, by replacing t_1x', $x'y_1$ and t_2x in G' by xy in G, we have a Hamiltonian cycle in G. □

A common technique to do reduction is to show that one problem is a special case of another problem. We call this technique <u>specialization</u>. The following is an example.

<u>Proposition</u> $HC \leq_p TSP$.

<u>Proof</u> Given $G=(V,E)$ for HC, we construct $G'=(V,E')$ for TSP such that if $uv \in E$, then we set its weight in G' to be 1, and if $uv \notin E$, then we set its weight in G' to be 2.

Then G has a Hamiltonian cycle if and only if G' has a TSP tour with cost n. □

G G'

An important problem and a nontrivial reduction

First, we introduce the 3-SAT problem, which will be important in the theory of NP-completeness.

In this problem, we are given n boolean <u>variables</u> $x_1, x_2, \ldots, x_n$, each can either be set to True or False.

We are also given a formula in conjunctive normal form (CNF), where it is an AND of the <u>clauses</u>, and each clause is an OR of some <u>literals</u>, where a literal is either x_i or $\bar{x}_i$.

For example, in $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_3)$, there are 4 clauses, each clause has at most 3 literals, and the formula has only 3 variables x_1, x_2, x_3.

<u>3-Satisfiability (3-SAT)</u>

Input: A CNF-formula in which each clause has at most three literals.

Output: Is there a truth assignment to the variables that satisfies all the clauses?

In the above example, setting $x_1=T$, $x_2=F$, $x_3=T$ will satisfy all the clauses, and hence the formula.

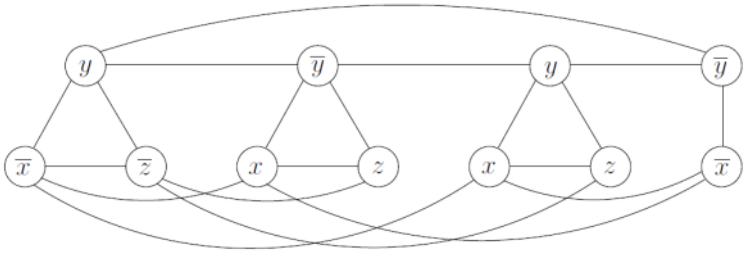
This problem looks quite different than other problems that we have seen.

Doing a reduction between two seemingly different problems usually requires some new ideas.

<u>Theorem</u> $3\text{-SAT} \leq_p IS$.

<u>Proof</u>: Given a 3SAT formula, we would like to construct a graph G so that the formula is satisfiable if and only if the graph G has an independent set of certain size.

Figure 8.8 The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$.



[DPV]

Reduction : For each literal in the formula, we create one vertex in the graph.

We would like a satisfying assignment of the formula corresponds to an independent set, and vice versa.

To satisfy the formula, we need to set at least one literal for each clause to be True.

It is easy to do so if we satisfy each clause separately.

The important point is to be consistent over the choices, i.e. if we set x to be True to satisfy some clause, then we can't set x to be False to satisfy some other clauses.

To enfore consistency, for each variable x_i, we add an edge between x_i and $\overline{x_i}$, for each appearance of x_i and $\overline{x_i}$, so that x_i and $\overline{x_i}$ won't both belong to an independent set.

We also add edges between literals of the same clause, to ensure that we only choose one literal in each clause to the independent set.

So, there are two types of edges.

One type are edges within each clause (i.e. one triangle for each clause of three literals).

Another type are edges between each appearance of x_i and $\overline{x_i}$ for $1 \le i \le n$.

Clearly, the construction of G can be done in polynomial time in the size of the formula.

The following claim will complete the proof of the theorem.

Claim Suppose the formula has k clauses. Then the formula is satisfiable if and only if there is an independent set of size k in the graph G.

Proof $\Rightarrow$) If there is a satisfying assignment, then we choose one literal that is set to True in each clause (e.g. if x_2=F in the satisfying assignment, then the literal $\overline{x_2}$ is True), and put the corresponding vertex to be in the independent set.

Since the assignment is satisfiable, there is at least one True literal in each clause, and so the set has at least k vertices.

These k vertices form an independent set because there are no edges of the first type between them as we choose only one literal vertex in each clause, and also there are no edges of

the second type as we won't choose both x_i and $\bar{x}_i$ as the satisfying assignment is consistent.

$\Leftarrow$) Suppose there is an independent set of size k in G.

Since there are edges (of the first type) between literals in the same clause, any independent set can only pick at most one vertex in each clause.

Since there are only k clauses, an independent set of size k must choose exactly one vertex from each clause.

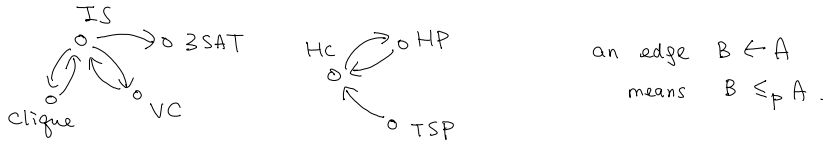
Also, because of the consistency edges (i.e. edges of the second type), each variable we choose at most one literal (note that we could choose none of the literals).

So, if we choose x_i in the independent set, then we set x_i to be True; otherwise, we set x_i to be False.

By the consistency edges, this assignment is well-defined, and since there is a vertex in each clause, this assignment satisfies every clause. □ □

Concluding Remarks

We have introduced the notion of a polynomial time reduction, and used it to establish relations between different problems, and so far we have



an edge $B \leftarrow A$ means $B \leq_p A$.

In principle, we can add new problems and relate to these problems, and slowly build a big web of all computational problems.

As $\leq_p$ is transitive, any strongly connected component in this web forms an equivalent class of problems in terms of polynomial time solvability.

Is there a better way to do it than to consider the problem one by one?

References: [KT 8.1, 8.2]