Notes for CS 321

Paul Cull

Department of Computer Science
Oregon State University

January 6, 2016

Chapter 4

Reductions, or *When is one problem no harder than another?*

4.1 Ordering Problems by Hardness

When is one problem harder than another? We want to say that problem B is harder than problem A when we have to use more resources (like time or space) to solve B then we need to use to solve A. Perhaps I have an $\Theta(n^3)$ algorithm for B and an $\Theta(n^2)$ algorithm for A. Am I justified in claiming that problem B is harder than problem A? Let's be definite: NO, I'm not justified.
All I know is that one algorithm for B takes more time than some other algorithm for A. If I'm sensible, and try to look up algorithms for B, perhaps I'll find an $\Theta(n \log n)$ algorithm. Now I have an $\Theta(n \log n)$ algorithm for B and an $\Theta(n^2)$ algorithm for A. Should I change my mind and now claim that "A is harder than B"? In our everyday sense of "hard" this might be reasonable. That is, the hardness or easiness of a problem might depend on my state of knowledge. When I know more about a problem, I may be willing to say that it is easier than I thought. While such locutions might make sense in casual conversation, in theory we would like the hardness of a problem to be independent of our feelings or state of knowledge. Our theory should be Platonic in the sense that hardness exists in the eternal abstract nonmaterial mathematical universe. Our theory should not depend on our feelings or the limitations of our knowledge about problems. [1]

As we have said, we may not and usually do not know the BEST algorithm

[1] Various philosophers might argue that no such abstract universe exists and that real thought only exists in our real material bodies and brains. To them we reply that the assumption of the Platonic universe of mathematical objects has proven very effective in practice, and so even if the abstract universe does not exist, it may be practically useful to adopt this fiction in our reasoning.

for a problem. Even if we know the BEST algorithm, we may not know that it is best. To skirt around this difficulty of lack of knowledge, we can try to relate one problem to another. This is the general approach with the unfortunate (regrettable) name – REDUCTION. We will eventually talk about problem A being *reduced* to problem B.

The problem with this terminology is that we are trying to say that A may be easier than B, or as I prefer to say

$$A \text{ is no harder than } B.$$

We can symbolize this relationship by the usual "less than" symbol

$$A \leq B.$$

Reading this as A *is no harder than* B has the potentially easier problem A and the potentially harder problem in the "right" places. Reading this as A *reduces to* B tends to cause the confusion that we usually reduce something harder to something easier and so it seems that A is the harder problem and B is the easier problem.

4.2 Partial Orders

We want to discuss an ordering on problems which is analogous to the less than ordering on numbers. But, we expect our problem ordering to have some different properties than our number ordering. Let us recall some facts and definitions about orderings (or orders). A *relation* on a set S is a set of ordered pairs of elements from S. [2] An *ordered pair* has a first element and a second element, and which is first and which is second matters. So, for example, (1,3) is an ordered pair of integers which is different from (3,1) because the first of these pairs has 1 as its first element and the second of these pairs has 3 as its first element. Two ordered pairs are the same if they both have the same first element and they also both have the same second element.

Some simple examples of relations are:

$$R1 = \{ (1,2),\ (3,3),\ (1027,8) \}$$

$$R2 = \{ (44,43),\ (42,41),\ (0,0),\ (41,42) \}$$

$$R3 = \{ (1789,1776) \}$$

$$R4 = \{ (APPLE, ORANGE),\ (SPINACH, STEAK) \}$$

$$R5 = \{ (x,y) \mid y = x^2 \}.$$

In these examples, we did not specify the set S, because S is usually clear from the context in which a relation is defined. $R1$, $R2$, $R3$ and $R5$ presumably have

[2] Actually this is a *binary relation* since it only involves two elements at a time. We could have a more general view of relations which would involve k elements at a time. Since, we'll be using only binary relations, we'll use the term relation without the modifier "binary".

S's which contain numbers. I would guess that $S1$, $S2$, $S3$ and $S5$ contain the natural numbers, but they might contain many more numbers or many fewer numbers. As $R4$ points out, relations are not restricted to numbers, and an S could contain both numbers and non-numbers. Although in $R3$ the elements appear to be numbers, they could instead be dates, e.g. 1789 (the year of the French revolution) is related to 1776 (the year of the American revolution). $R1$, $R2$, $R3$ and $R4$ are finite relations because they contain only a finite number of ordered pairs. Such finite relations can often be specified by listing their ordered pairs. $R5$ may be a finite or infinite relation depending on $S5$. If $S5$ were the natural numbers, then $R5$, which is the relation between a number and its square, would be an infinite relation. If $S5 = \{0, 1\}$, then $R5$ would be the finite relation $\{(0,0), (1,1)\}$. It's also possible that $S5$ could contain some elements for which the squaring relation is not defined, and such elements could not appear as the first elements of any pairs in $R5$. The style of $R5$, representing a relation by a property is often used even for finite relations because there may be too many pairs to write down conveniently.

We would like our relations to have certain extra properties. Often we want relations to be *reflexive*, that is, if

$$\forall x \in S \quad (x, x) \in R$$

then R is a **reflexive** relation. For orderings, the most important property is *transitivity*. A relation R is **transitive** if

$$\forall x, y, z \in S \quad (x, y) \in R \text{ and } (y, z) \in R \quad \text{implies} \quad (x, z) \in R.$$

A **partial order** is both **reflexive** and **transitive**. [3]

An additional property of many relations is symmetry. A relation, R, is **symmetric** if

$$\forall x, y \in S \quad (x, y) \in R \quad \text{implies} \quad (y, x) \in R.$$

For a symmetric relation we can forget about the pairs being ordered, and just say that a symmetric relation is a set of (unordered) pairs.

A relation which is **reflexive**, **symmetric**, and **transitive** is called an **equivalence relation.** For example, the usual equality relation on natural numbers is an equivalence relation because it consists only of the pairs (x, x) for every nartural number x. Obviously, this equality relation, $EQUAL$, is reflexive and symmetric. It is also transitive because

$$(x, x) \in EQUAL \text{ and } (x, x) \in EQUAL \quad \text{implies} \quad (x, x) \in EQUAL,$$

and there are no other pairs which have x as a second element. Many other relations are also equivalence relations. For example, $MOD2$ on natural numbers is the relation which says that $(x, y) \in MOD2$ iff x and y leave the same remainder (either 0 or 1) when divided by 2. Here, again reflexive and symmetric

[3] With only these two properties, some authors call the relation a pre-order and reserve the term partial order for a pre-order with one additional property.

are obvious. $MOD2$ is also transitive, because if x and y leave remainder r, and y and z leave remainder r', then $r = r'$ and so x and z both leave remainder r.

In attempting to make partial order similar in definition to equivalence relation, some authors add the anti-symmetric property to partial orders. A relation, R, is **anti-symmetric** if

$$\forall x, y \in S \quad (x, y) \in R \text{ and } (y, x) \in R \quad \text{implies} \quad x = y.$$

The usual less than relation on integers has this anti-symmetric property. That is,

$$x \leq y \text{ and } y \leq x \quad \text{implies} \quad x = y.$$

(Notice that we have changed the way we are writing relations. Instead of writing $(x, y) \in R$, we write $x \; R \; y$ where R is the name of the relation or a symbol representing the relation.)

What bothers me about this definition is that a relation can be *both* symmetric and anti-symmetric. For example, the usual equality relation is both symmetric and anti-symmetric. That is,

if $x = y$ then $y = x$ (symmetric),

but also, if $x = y$ and $y = x$ then $y = x$ (anti-symmetric).

Further, we run into some relations which I would like to call partial orders which are not anti-symmetric. For example, consider a directed graph in which we have $x \to y$ iff there is a directed edge from x to y. I would like to say $x \leq y$ if there is a chain of (zero or more) directed edges so that

$$x \to y_1 \to y_2 \ldots \to z.$$

While this $\leq$ is reflexive (using zero edges) and transitive by construction, it may not be anti-symmetric because there may be directed cycles in the graph. In the special case, when there are no directed cycles in the graph, we call the graph a **DAG**, a directed acyclic graph, and $\leq$ on a DAG is anti-symmetric. But, if there are directed cycles, say $x \to y$, $y \to z$, and $z \to x$, then $x \leq z$ and $z \leq x$ but $x \neq z$, and $\leq$ is not anti-symmetric.

There is a standard method to skirt this problem. We first define an equivalence relation on elements, then form equivalence classes of elements, and then put a partial order on these equivalence classes rather than on the original elements. Continuing with our digraph example, we define the equivalence relation $\equiv$ by $x \equiv y$ iff there is a directed cycle which contains both x and y. This relation is reflexive using zero length cycles. The relation is symmetric because its definition is symmetric in x and y. Finally, this relation is transitive because if there a directed cycle containing both x and y then there is a directed path from x to y, and similarly if there a directed cycle containing both y and z then there is a directed path from y to z and putting these two paths together gives a directed path from x to z. Turning this argument around, there is also a directed path from z to x and so there is a directed cycle which contains both x and z. (Notice that the paths and cycles we are using do not have to be simple – we allow vertices and/or edges to be used more than once.) The equivalence

classes for $\equiv$ are sets of vertices. Two vertices, x and y, are in the same equivalence class exactly when there is a directed path from x to y and a directed path from y to x. (In graph theory, these equivalence classes are called the **strongly connected components** of the graph.) Now consider the digraph which has these equivalence classes as its vertices, and has a directed edge from equivalence class E_1 to equivalence class E_2 when there is some (original) vertex in E_1 which has a directed edge to some (original) vertex in E_2. If D is the original digraph and $\equiv$ is this equivalence relation, then we can use the notation $D/\equiv$ to refer to this new directed graph on equivalence classes. The point of this whole construction now becomes clear: $D/\equiv$ is a DAG and the $\leq$ on $D/\equiv$ is anti-symmetric. Specifically, if $E_i \rightarrow E_j$ and $E_j \rightarrow E_i$ then $E_i = E_j$. Or considering the original vertices within the E's, if $E_i \rightarrow E_j$ there is a vertex x in E_i which has a directed edge to some vertex y in E_j, and if $E_j \rightarrow E_i$ there is a vertex w in E_i which has a directed edge to some vertex z in E_i. But from the definition of $\equiv$, since y and w are both in E_j there is a directed path from y to w and thence to z and back to x, and so $x \equiv y$. This argument can be extended to show that for each $x_i \in E_i$ and $y_j \in E_j$, there is a directed path from x_i to y_j and a directed path from y_j to x_i, and so every element of E_i is equivalent to every element of E_j. Since these are equivalence classes, they are identical, and anti-symmetry is verified. While this construction always works, it gives partial orders defined on equivalence classes of elements rather than on the original elements themselves. To avoid this construction, I would prefer to redefine "anti-symmetric". First, I say that

$$\text{if } x \leq y \text{ and } y \leq x \quad \text{then} \quad x \equiv y.$$

(Here, I'm sneaking in the equivalence relation.)
Then, I say that $\leq$ is *anti-symmetric*

$$\text{if } x \leq y \text{ and } y \leq x \quad \text{implies} \quad x \equiv y.$$

So, by my definition the original relation on elements is automatically anti-symmetric if the relation is reflexive and transitive. Thus, a partial ordering need only be reflexive and transitive.

4.3 The General Idea of Reductions

At its core most modern science is reductive in the sense that it solves problems (or answers questions) by reducing them to problems or questions about smaller or simpler objects. In computer programming, in a similar way, we reduce problems to very simple problems by building complex systems out of a handful of simple primitives. The whole system works correctly if the primitives work correctly and if the reduction (our program) correctly reduces the problem to the primitives. In the overwhelming majority of cases, failures of systems are the result of programming errors rather than failure of the primitives to be correct. (The Pentium debacle of a decade ago is one of the few examples in which a

primitive (a multiplier) was incorrect.) While the process of creating reductions is neither trivial nor well-understood, we still manage to get some programs to work correctly.

We are now going to take a step upward in the design process. Let us assume that we have correctly working programs for some problems. We might have designed these programs, or they might have been given to us by a wise guru, or we might have purchased them from a (we hope) trusted source, or we might have received these program as a divine gift, or (and this is usually the case when we are arguing complexity results) we might assume that these programs exist without actually having them or knowing how to construct them. We'd like to use these assumed correct programs to create programs for other problems.

The idea of reductions has two sources. One, which we have discussed in Chapter 1 is the classification of problems. There we hinted at the idea that the halting problem was, in some sense, the hardest problem in **RE**. The second source is traditional design of algorithms in which we solve one problem by solving another problem. For example, we can find the smallest element in an array by sorting the array and reporting the first element in the sorted array. Here we can say that **Find-Smallest** is no harder than **Sort** and use the usual less than or equal to notation:

$$\textbf{Find-Smallest} \quad \leq \quad \textbf{Sort} \ .$$

Turing introduced the idea of an ordering on problems in the context of showing that a seemingly harder problem was no harder than a seemingly simpler problem. For example, Turing argued that **HALT** was no harder than **HALT-ON-ZERO**, but since **HALT-ON-ZERO** is a special case of **HALT**, and therefore simpler than **HALT**, he referred to his argument as a *reduction*. While later authors have used the ordering notation:

$$\textbf{HALT} \quad \leq \quad \textbf{HALT-ON-ZERO} \ ,$$

they have stuck with the word "reduction" and read this formula as

$$\textbf{HALT} \ \text{ is reducible to } \ \textbf{HALT-ON-ZERO} \ ,$$

or as

$$\textbf{HALT} \ \text{ reduces to } \ \textbf{HALT-ON-ZERO} \ .$$

I think that it is easier to read this as

$$\textbf{HALT} \ \text{ is no harder than } \ \textbf{HALT-ON-ZERO} \ ,$$

because this helps keep the harder problem on the right side of the $\leq$ sign, and when using "reduces" it is too easy to mistakenly put the easier problem on the right of the $\leq$ sign.

While it's obvious that

$$\textbf{Find-Smallest} \quad \leq \quad \textbf{Sort} \ ,$$

it takes a little effort to show that

$$\textbf{HALT} \quad \leq \quad \textbf{HALT-ON-ZERO} \ .$$

We begin by stating these two problems as **YES/NO** problems:

HALT
INPUT: A pair (M, t) where M is a program and t is an input for M.
QUESTION: Does the program M eventually halt when given the input t?

HALT-ON-ZERO
INPUT: A program M.
QUESTION: Does M halt when given input 0?

We want to take the input (M, t) for **HALT** and convert it into another program, say M_t, so that M_t halts on 0 exactly when M halts given t. We build M_t by starting with some code which looks at the input and if the input is 0, then this code overwrites the input with the input t. We add to this code the code for M, so that, M_t given 0 will overwrite 0 with t and then will behave like M given t. In particular, if M_t given 0 halts, then M given t will also halt. Conversely, if M_t given 0 does not halt, then M given t will not halt. Hence if we could answer the question: Does M_t halt when given 0 ?, we would also have the answer to the question: Does M eventually halt when given the input t?

As we said above, the word "reduction" seems to be appropriate here because **HALT-ON-ZERO** is a special case of **HALT**. That is, we could decide if M_t given 0 halts by correctly deciding whether $(M_t, 0)$ is a **YES** instance of **HALT.** So here in a very reasonable sense we have both

$$\textbf{HALT} \quad \leq \quad \textbf{HALT-ON-ZERO} \ ,$$

and

$$\textbf{HALT-ON-ZERO} \quad \leq \quad \textbf{HALT} \ .$$

In at least partial distinction from this, we saw that

$$\textbf{Find-Smallest} \quad \leq \quad \textbf{Sort} \ ,$$

but we would probably be willing to say that

$$\textbf{Sort} \quad \not\leq \quad \textbf{Find-Smallest}$$

because we can find the smallest element without sorting the array.

4.4 Some Examles

In spite of our contention that

$$\text{SORT} \quad \not\leq \quad \text{FIND-SMALLEST},$$

we could still use a program for FIND-SMALLEST to carry out a SORT. Our procedure would find the smallest, remove it from the set and put it in a queue. Then we would repeat the procedure on the remaining set, and so forth. At the end of this calculation, the set will be empty, and the sorted version of the set will be in the queue. So, we could say that

$$\text{SORT} \quad \leq \quad \text{FIND-SMALLEST},$$

but our notion of $\leq$ has changed. Whereas in arguing that

$$\text{FIND-SMALLEST} \quad \leq \quad \text{SORT},$$

we only used SORT once, in doing SORT via FIND-SMALLEST we used the FIND-SMALLEST routine n times, where n was the number of elements in the set. In terms of the time needed to solve these problems, we have that $T_{\text{FIND-SMALLEST}} \leq T_{\text{SORT}}$ but $T_{\text{SORT}} \leq n\, T_{\text{FIND-SMALLEST}}$.

As another example, we can look at multiplication and division, and ask which one is harder. But, using our idea of transforming one problem to another we may find that both problems are essentially the same.

Let's start by assuming that we have routines for addition, subtraction, and reciprocal. The reciprocal of A is $1/A$.
(If we were worrying about all the details, we might have to decide if we're talking about floating point numbers or appropriately scaled integers, and what $1/A$ would mean in each case.)
The easy to check formula

$$\frac{1}{\frac{1}{A} - \frac{1}{A+1}} - A \quad = \quad A^2$$

tells us that the square can be computed use 3 reciprocals and a few adds and subtracts. Further the formula

$$(x + y)^2 - (x - y)^2 \quad = \quad 4\,x\,y$$

tells us that the product $x\,y$ can be computed using two squarings and a couple of adds and subtracts.
(It seems that we also have to divide by 4, but if we're working in binary, this is a simple two bit shift.)
So, we can compute the product $x\,y$ using 6 reciprocals and a few adds, subtracts, and shifts. Hence we could write that

$$\text{MULTIPLY} \quad \leq \quad \textit{RECIPROCAL}.$$

On the other hand, the Newton iteration

$$\hat{x} \;=\; x\,(\,2 \,-\, A\,x\,)$$

allows us to compute $1/A$. It's easy to show that if $\hat{x} = x$ then $x = 1/A$. A little more analysis (see *Difference Equations* book) shows that this iteration

converges quickly and that $1/A$ correct to n bits can be computed in a constant times the time to multiply n bit numbers. So, we can now write

$$\text{RECIPROCAL} \quad \leq \quad \text{MULTIPLY}.$$

This example suggests that there may be *constellations* of problems, so that if one could discover a faster method for one of the problems in the constellation, then one would also have faster methods for all of the other problems in the constellation.

Since we have that $A \leq B$ and $B \leq A$ for any two problems in the constellation, we could write $A \equiv B$ and say that A and B are in the same equivalence class of problems.

The MULTIPLY constellation which includes SQUARING, RECIPROCAL, and DIVISION, is not the only constellation. For example, there are a variety of problems including TRANSITIVE CLOSURE and MEMBERSHIP in a CONTEXT FREE language that can be shown to be in the constellation of MATRIX MULTIPLICATION.

In these examples, we are saying that problems A and B are equivalent if

$$T_A(n) = O(T_B(n)) \quad \text{and} \quad T_B(n) = O(T_A(n))$$

that is the time to compute the answer to a size n instance of A, symbolized by $T_A(n)$ is bounded above by a constant time the time to solve a size n instance of B, and vice-versa. But we are saying more. If C and D are some totally unrelated problems so that T_C and T_D happen to be (essentially) the same (using the methods we currently know), we don't want to say that C and D are equivalent, unless we can also show that speeding up the solution time for one would also speed up the solution time for the other.

While finding these constellations is important, we may also want to find the *hardest* problem among a group of problems, so that a faster method for this hardest problem would also imply a faster method for all the problems in the group, in spite of the fact that we may already know methods for some of the problems in the group which are faster than the methods we know for the hardest problem.

Our example here is SORT and FIND-SMALLEST. If we could find a faster method for SORT then we could speed up the method for FIND-SMALLEST which uses SORT, but we already know a method for FIND-SMALLEST which is faster than any method we know for SORT.
(In fact, it can be shown, at least in the *comparison model of computation*, that FIND-SMALLEST can be computer faster than ANY possible method for SORT.)

4.5 Turing Reductions

A problem A is Turing reducible to a problem B, symbolized as

$$A \leq_{\mathbf{T}} B$$

when an algorithm for B can be used as a subroutine to create an algorithm for A.

This notion is quite general and we may need to add some extra stipulations to conform to common usage.

(a) Total Functions: If A and B are total functions then we say $A \leq_{\mathbf{T}} B$ to mean that if had a routine that could (totally) compute B, then there would be a program (possibly using the routine for B as a subroutine) that would (totally) compute A.
(This is the commonly used meaning of Turing reducible for functions.)

(b) Partial Functions: If A and B are partial functions then we say $A \leq_{\mathbf{T}} B$ to mean that if had a routine that could compute B, then there would be a program (possibly using the routine for B as a subroutine) that would compute A.
(Here B or A could be total. When B is only partial, things may be a little dicey. We might want to call B's routine only with inputs that cause the routine to halt but we can allow non-returning calls to B's routine on inputs to A's routine that should not halt. Since halting problems are generally unsolvable, we will probably have difficulty in telling which calls to allow and which calls to prevent.)

(c) Sets (Recognizers): $A \leq_{\mathbf{T}} B$
if we had a recognizer for B, then there would be a recognizer for A which may use B's recognizer as a subroutine.

(d) Sets (Acceptors): $A \leq_{\mathbf{T}} B$
if we had a acceptor for B, then there would be a acceptor for A which may use B's acceptor as a subroutine.

(e) Sets (Rejectors): $A \leq_{\mathbf{T}} B$
if we had a rejector for B, then there would be a rejector for A which may use B's rejector as a subroutine.

(In Chapter 1, we talked about oracles. Here we can say that $A \leq_{\mathbf{T}} B$ if we can use an oracle for B to build an algorithm for A.)

4.6 Many-one Reductions

A set A is many-one reducible to a set B in symbols
$A \leq_{\mathbf{m}} B$ means there exists a computable function f,
so that $x \in A$ **iff** $f(x) \in B$.
(This is sometimes called "transformation" because x, potentially in A is transformed to $f(x)$ which may be in B.)
(These are called **many-one** reductions because several (*many*) x's may be transformed to a single (*one*) $f(x)$. In many common cases, f is in fact one-to-one. Later, we will consider possible restrictions on the complexity of f.)

Many-one reduction seems more natural then Turing reduction for many problems, because after the transformation only one call to the oracle for B is used, whereas in the Turing reduction there is no bound on the number of calls used and the number of calls may depend on x.

Usually, Turing reductions use recognizers, so if for sets A and B, $A \leq_{\mathbf{T}} B$ then also $\overline{A} \leq_{\mathbf{T}} B$, where $\overline{A}$ is the complement of A because the recognizer for B is also a recognizer for $\overline{B}$ by interchanging **YES** and **NO**. So, for example, $\leq_{\mathbf{T}}$ would lump together all of the sets in **RE** and **coRE**. In contrast to Turing reduction, many-one reduction does not allow the use of negation (or "not in") and thus allows for a distinction between **RE** and **coRE**.

This observation suggests the easy to prove result

$$A \leq_{\mathbf{m}} B \Rightarrow A \leq_{\mathbf{T}} B$$

which could be symbolized as $\leq_{\mathbf{m}} \Rightarrow \leq_{\mathbf{T}}$ and read as "the order $\leq_{\mathbf{m}}$ is a refinement of the order $\leq_{\mathbf{T}}$".

The "naturalness" of $\leq_{\mathbf{m}}$ for **RE** is captured in the following theorem.

Theorem 1. (Closure of RE) *Any set many-one reducible to an RE set is an RE set:* $A \in$ **RE iff** $\exists B \in$ **RE** *so that* $A \leq_{\mathbf{m}} B$.
In particular any set many-one reducible to HALT is an RE set:
$A \in$ **RE iff** $A \leq_{\mathbf{m}}$ **HALT**.

Proof. If $A \in$ **RE**, then we can use can use the identity transformation (which is clearly computable) to give $A \leq_{\mathbf{m}} A$. If $A \leq_{\mathbf{m}} B$ and $B \in$ **RE**, then B has an acceptor, and we have an acceptor for A by $\mathbf{ACC}_A(x) = \mathbf{ACC}_B(f(x))$ and so $A \in$ **RE**.
Finally, if $A \in$ **RE** then $x \in A$ **iff** $(\mathbf{ACC}_A, x) \in$ **HALT** and taking $f(x) = (\mathbf{ACC}_A, x)$ gives the many-one reduction from A to **HALT**. □

There is some structure to the ordering created by many-one reduction. Specifically, every pair of **RE** sets has a *least upper bound* under this ordering.

Definition 4.6.1. For any two sets A and B and an ordering $\leq$, the set C is a least upper bound for A and B with respect to $\leq$,

$$\textbf{iff} \qquad A \leq C, \qquad B \leq C$$

$$\textbf{and} \qquad \forall D \qquad A \leq D \quad \text{and} \quad B \leq D \qquad \Longrightarrow \qquad C \leq D.$$

This is symbolized by $C = \mathbf{LUB}(A, B)$.

If we assume, reasonably, that our sets of strings are defined over an alphabet with at least two characters, then we can actually describe these **LUB**'s.

Theorem 2. (LUB Theorem)
RE *under* $\leq_{\mathbf{m}}$ *has the* **LUB** *property and the* **LUB***'s are in* **RE**.

Proof. Given any two **RE** sets A and B define C by

$$C \;=\; \{\, z \,|\, z \;=\; ax \;\text{ with } x \;\in\; A \;\;\text{ or }\;\; z \;=\; by \,\text{ with } y \;\in\; B \,\}.$$

Assume that there is a D so that $A \;\leq_{\mathbf{m}}\; D$ and $B \;\leq_{\mathbf{m}}\; D$, so there are two computable functions $f(x)$ and $g(x)$ which respectively map strings from A into D and strings from B into D. From these we create a new function $h(z)$ defined by

$$h(\,z\,) \;=\; \begin{cases} f(x) & \text{if} \quad z \;=\; ax \\ g(y) & \text{if} \quad z \;=\; by \\ d_0 & \textbf{ELSE}\;\{\text{where } d_0 \;\notin\; D\} \end{cases} .$$

Clearly $h(\,z\,)$ defines a many-one reduction from C to D, i.e. if $z \;\in\; C$ then it starts with an a or b and using the mappings $f(x)$ and $g(y)$, $h(\,z\,)$ maps z to an element of D, if $z \;\notin\; C$ the third line of $h(\,z\,)$ maps z to something outside of D. Finally, note that C is in **RE** because an acceptor for C follows from acceptors for A and B, i.e.

$$\mathbf{ACC}_C(z) \;=\; \begin{cases} \mathbf{ACC}_A(x) & \text{if} \quad z \;=\; ax \\ \mathbf{ACC}_B(y) & \text{if} \quad z \;=\; by \\ \text{DOES NOT HALT} & \text{if } z \text{ does not start with } a \text{ or } b \end{cases}$$

□

The **LUB** construction and theorem work for many variations on many-one reduction, e.g. putting many reasonable bounds on computing the transforming function, but for very restrictive transformations like homomorphism and finite state reduction the construction fails.

4.7 Complete Problems

Here we want to use the ideas of reduction to show that some classes of problems have *hardest* problems. The idea is that the *hardest* problems should be in the class and that all problems in the class are reducible to (*no harder than*) the hardest problem. We will call such problems **complete problems**. One caveat, it is traditional to say that problem B is $\mathcal{C}$-complete to mean that B is hardest problem in the class $\mathcal{C}$, but the notion of reduction being used is usually *implied* rather than stated.

4.7.1 HALT is RE-Complete.

The best know **RE**-Complete set is the **HALT** set, which is the set

$$\{(\,M,\, x\,) \mid M \text{ is an acceptor, } x \text{ is an input, and } M \text{ accepts } x\}.$$

HALT is **RE**-Complete with respect to $\leq_{\mathbf{m}}$ because for any set $A \;\in\; \mathbf{RE}$ then there is a nondeterministic acceptor $\mathrm{ACC_A}$ for A, so that, $\mathrm{ACC_A}(x) \;=\; \mathbf{YES}$

iff $x \in A$. For our transforming function $f(x)$, we use $f(x) = (\,\mathrm{ACC_A},\, x\,)$. Clearly, $f(x) \in$ **HALT** implies that $\mathrm{ACC_A}$ accepts x and hence $x \in A$. Conversely, if $x \in A$ then $\mathrm{ACC_A}$ accepts x, and $f(x) \in$ **HALT**. This $f(x)$ is clearly a computable function. In fact, it is linear time computable because writing out $\mathrm{ACC_A}$ takes *constant* time, that is independent of the length of x, and writing out a copy of x takes only $O(\,|x|\,)$ time.

4.7.2 LINEAR–HALT is $\mathcal{NP}$-Complete.

Here we want to show that there is an $\mathcal{NP}$-Complete problem. That is, we will give a problem B and show that it is the hardest problem in $\mathcal{NP}$. As we mentioned above, we have to decide which notion of ordering among problems we should use. For our notion of "no harder than" we use polynomial time many-one reduction. That is, $A \leq^{\mathrm{p}}_{\mathbf{m}} B$ **iff** there is a polynomial time computabel function $f(x)$ so that $x \in A$ iff $f(x) \in B$.

Using this ordering, $\mathcal{NP}$-HALT is $\mathcal{NP}$-Complete. $\mathcal{NP}$-HALT is the set of triples $(M,\ 1^t,\ x\ )$ where M is a non-deterministic Turing machine, which accepts the input string x in at most t steps. (Here, 1^t is a string of t 1's, i.e. t written out in unary.) If $A \in \mathcal{NP}$ then there is a polynomial $p(n)$ and a nondeterministic acceptor $\mathrm{ACC_A}$ for A, so that, $\mathrm{ACC_A}(x) =$ **YES** in time at most $p(\,|x|\,)$ iff $x \in A$. for our polynomial time transforming function $f(x)$, we use $f(x) = (\,\mathrm{ACC_A},\, 1^{p(\,|x|\,)},\, x\,)$. Clearly, $f(x) \in \mathcal{NP}$-HALT implies that $\mathrm{ACC_A}$ accepts x in at most $p(\,|x|\,)$ steps and hence $x \in A$. Conversely, if $x \in A$ then $\mathrm{ACC_A}$ accepts x in at most $p(\,|x|\,)$ steps, and $f(x) \in \mathcal{NP}$-HALT. This is a polynomial time transformation because writing out $\mathrm{ACC_A}$ takes *constant* time, that is independent of the length of x, further given x, one can quickly compute $p(\,|x|\,)$ and write out $1^{p(\,|x|\,)}$, and finally writing out a copy of x takes only $O(\,|x|\,)$ time.

4.7.3 Greibach's $\mathcal{L}_0$ is CFL-Complete

Theorem 3. *(Greibach,1973) There is a HARDEST context free language, $\mathcal{L}_0$, in the sense that every* **CFL** *can be reduced to $\mathcal{L}_0$ using homomorphisms.*

For this theorem the reduction is still many-one but with the restriction that the transforming function is a homomorphism, that is it takes each character of the input string to a string over the output alphabet and the resulting transformed string is the concatenation of these output strings. For example, if $f(a) = 3\#\#\#$ and $f(b) = \#2\#$ then $f(ab) = 3\#\#\#\#2\#$. Obviously, a homomorphism could be computed by a finite state machine which reads an input symbol and outputs a string. In fact, for a homomorphism the finite state machine would only need one state.

Although $\mathcal{L}_0$ is **CFL-**Complete under these weak notions of reductions, a slightly stronger notion will permit better analogies among **RE**, $\mathcal{NP}$, and **CFL**. In this stronger reduction, the transformation is allowed to be computed by a deterministic pushdown machine (DPDA). This machine has a stack which is

an indefinite memory and the DPDA can use this memory to compute functions which cannot be computed by finite state machines.

Since obviously, homomorphisms can be computed by DPDA's, $\mathcal{L}_0$ is also a complete set for **CFL** under DPDA reduction.

That there is a hardest context-free language was show by Greibach(1973). Here we give a slight variant due to G. Révész (1983):

$$\{\, X_1 c Y_1 c Z_1 d \cdots X_n c Y_n c Z_n d \,\}$$

where $n \geq 1$ and

$$Y_1,\, Y_2,\, \cdots,\, Y_n$$

each begin with a d which is followed by a string of balanced parantheses of two kinds in which e_1 and e_2 are the "open" and $\overline{e_1}$ and $\overline{e_2}$ are the "close" parentheses, and

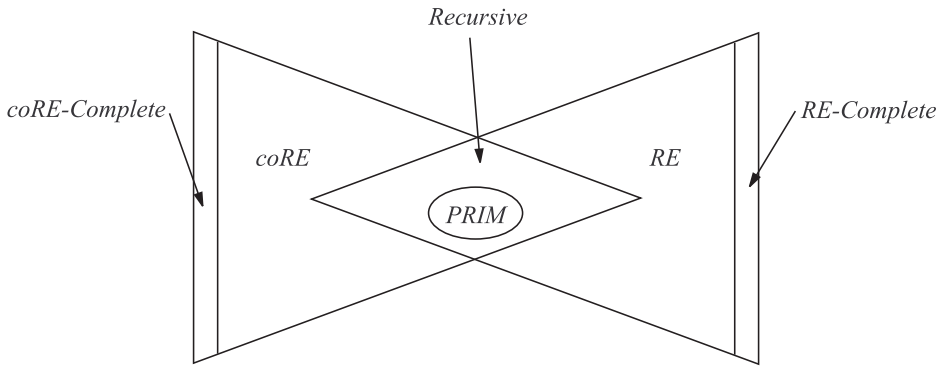
$$X_1,\, X_2,\, \cdots, X_n \text{ and } Z_1,\, Z_2,\, \cdots, Z_n$$

are strings over $\{\, b,\, c,\, e_1,\, e_2,\, \overline{e_1},\, \overline{e_2} \,\}$.

We will not prove that this is the hardest context-free language. The idea of the proof is to show that this language will contain a string $h(x)$ exactly when another context-free language, L, contains the string x. Of course for each L, there will be a different $h(\,)$. The argument proceeds by assuning that L is represented by a grammar in Greibach form (which we have not explained) and showing how to code each production in L's grammar within the above allowed alphabet and then showing that $h(x)$ is an encoding of a derivation of x.

There are some details about the null string. As given above the hardest language does not contain the null string, but the grammar given in Exercise 4.7 does generate the null string. Which of these to use depends on whether or not L contains the null string.

4.8 The World of RE



The pleasant and amazing result is that one can prove all of the inclusions depicted in the above diagram. The diagram shows containments for various classes of sets, but there is also an implied notion of *harder* which radiates out from the center, with the easier sets at the center and the harder sets at the right and left edges. For this notion of hardness, we want to use some version of many-one reduction, that is

$$A \leq B \quad \textbf{iff} \quad \text{there is an } f \text{ so that } x \in A \text{ exactly when} f(x) \in B.$$

The most obvious version of this reduction would allow f to be any computable function. The problem with using this reduction is that it would hide some of the inclusions in the diagram. In particular, under this reduction all recursive sets would be mutually reducible, that is

$$A \in \textbf{Rec} \quad \text{and} \quad B \in \textbf{Rec} \quad \Longrightarrow \quad A \leq B \text{ and } B \leq A$$

and so using this reduction the whole inner *diamond* could be reduced to a single point indicating that all the recursive sets are equivalent under this reduction. A more reasonable choice would be to limit f to primitive recursive functions, since then not all recursive sets would be equivalent. But, even with this restriction on f, the inner circle (for **PRIM**) would reduce to a single point. To get the full picture, we could restrict f to be polynomial time computable functions. With polynomial time many-one reduction not all **PRIM** sets are equivalent, but our picture does not show the fine structure of these relationships, instead it merely indicates that these sets are not all mutually reducible.

4.8.1 PRIM $\neq$ Rec

There are sets which are recursive but are not primitive recursive. This can be show by a simple diagonal argument (see Chapter 1). From any computable

listing of the **PRIM** sets, this argument constructs a **Rec** set which is not in **PRIM**. The computable listing exists because there is a programming language which exactly specifies the **PRIM** sets.

4.8.2 RE $\neq$ Rec

There are **RE** sets which are not in **Rec**. Here we use the HALT set which has the Universal Turing Machine as its acceptor and therfore HALT $\in$ **RE**. But, as we saw in Chapter 1, HALT cannot have a recognizer and hence HALT $\notin$ **Rec**.

4.8.3 RE $\neq$ coRE

Again we use the HALT set. The HALT set cannot have a rejector because then it would have a recognizer. So HALT $\in$ **RE** but HALT $\notin$ **coRE**.

4.8.4 RE-complete Sets

The outer borders of the diagam indicate that there are hardest sets (both for **RE** and for **coRE**). Here, we mention a few of these complete sets for **RE**. The complements of these sets are **coRE**-complete.

(a) HALT set
the set of all Turing machine and tape pairs, so that the Turing machine halts when given the tape.

(b) HALT-ZERO
the set of all Turing machines which halt when given the blank tape.

(c) DIOPHANTINE
The set of polynomial equations in several variables with integer coefficients which have an integer solution.
E.G. $p(x_1, x_2, \ldots, x_n) = 0$
where p is a polynomial with integer coefficients in n variables, and $p(x_1, x_2, \ldots, x_n) = 0$ has a solution in which x_1 and x_2 and $\ldots$ x_n are all integers.
Even if n is restricted to some small value, say $n \leq 10$, this set remains **RE**-complete. (As we have seen, when $n = 1$, this set is in **Rec** and hence is not **RE-**complete.)

(d) Predicate Logic
The set of TRUE formulas in Predicate Logic.
A formula has the form

$$\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_n \; F(x_1, \, x_2 \, \ldots, x_n, \, \ldots \, x_r)$$

where F is a logical formula containing variables, the logical connectives (e.g. and , or, not), and unspecified propositional function names (e.g. $g(x_{17}, x_{35}, x_{64})$ with no other knowledge about what g is, except that g

can only return TRUE or FALSE).
For the formula to be TRUE, it must evaluate to TRUE for all values of the free variables x_{n+1} through x_r and for ALL choices for the unspecified functions, the g's.
Gödel proved that this set has an acceptor, that is he showed that every TRUE formula in Predicate Logic has a proof. (Gödel's Completeness Theorem).

(e) Post Correspondence:
Given a set of equations

$$a_{i1}\, a_{i2}\, \ldots\, a_{ik_i} \;=\; b_{i1}\, b_{i2}\, \ldots\, b_{ir_i}$$

(there will be several of these equations and usually the number of characters on either side will be different) is there a string $X = x_1\, x_2\, \ldots\, x_n$ which can parsed in two different ways so that the first parse gives X as the left parts (a parts) of some of the equations and the second parse gives X as the right parts (b parts) of the corresponding equations? The string on one side of an equation is said to *correspond* to the string on the other side of the equation.
The set of sets of equations for which the desired X can be found is an **RE**-complete set.

(f) Languages:
The set of pairs G, x consisting of a formal grammar G and a string x so that x is in the language specified by G.

4.8.5 Structure of RE

RE contains sets which are neither **RE-complete** nor in **Rec**. In fact, the structure of **RE** is so highly branched that the following theorem can be proven. (We will not prove this theorem because the proof is rather complicated.)

Theorem 4. *Let D be any directed acyclic graph (i.e. a set of vertices with a set of directed edges (arrows) between some of the vertices so that starting at any vertex and following the arrows one cannot return to the starting vertex) then it is possible to assign* **RE** *sets to the vertices so that if $A \longrightarrow B$ is an edge then $A \leq B$ and if there is no edge between A and B then neither $A \leq B$ nor $B \leq A$ except when a relation is implied by transitivity (e.g. if $A \longrightarrow B$ and $B \longrightarrow C$, then $A \leq C$).*

4.8.6 RE-hard

A set is sometimes said to be **RE**-hard if the set is Turing equivalent to an **RE**-complete set. So, all **coRE**-complete sets are **RE**-Hard. In fact, if Turing reduction rather than many-one reduction were used, the diagram should be modified because the distinction between **RE** and **coRE** would vanish, each **coRE** set would be equivalent to an **RE** set and vice-versa.

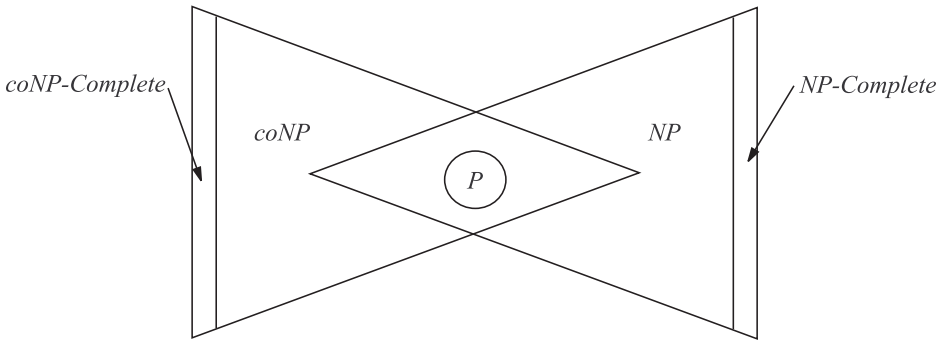
4.8.7 Hard and Soft Boundaries

In the diagram some of the boundaries are hard in the sense that they can be defined by a programming language, while others are soft in that they can not be specified by a programming language. Perhaps the hard boundaries should be drawn as solid lines and the soft boundaries should be drawn as dotted lines. For example, the *circle* enclosing **PRIM** is a hard boundary because we have programming languages which specify exactly the **PRIM** sets. The outer boundary of the *butterfly* enclosing **RE** and **coRE** is also hard because any general purpose programming language can only specify sets in these classes and any set in these classes can be specified by a program. On the other hand the *diamond* boundary separating **Rec** is a soft boundary because as we saw in Chapter 1, there is no programming language that can exactly specify the recursive sets. The boundary separating the *complete* sets from their respective classes is also a soft boundary, but we have not proved this.

4.8.8 Oracle Worlds

All of the results for **RE** *relativize*, that is, if we consider **RE** to be the class of sets which have acceptors without oracles or if we consider **RE** to be the class of sets which have acceptors with oracles, the same constructions and proofs can be carried out. So the picture of the world **RE** will also be the picture of the world of $\mathbf{RE}^k$ for the classes of sets with type k oracles.

4.9 The World of $\mathcal{NP}$



4.9.1 $\mathcal{P} \neq \mathcal{NP}$?????

$\mathcal{P}$ is the class of languages that have deterministic polynomial time acceptors. A deterministic polynomial acceptor is a program together with a polynomial $p(n)$ so that the program accepts all strings of length n in the language using at most $p(n)$ steps. Obviously, running the acceptor on a string of length n for $p(n)$ steps and not getting an acceptance means that the string is not in the language, and hence the acceptor is also a polynomial time recognizer.

$\mathcal{NP}$ is the class of languages that have *non*deterministic polynomial time acceptors. A nondeterministic polynomial acceptor is a nondeterministic program together with a polynomial $p(n)$ so that the program accepts all strings of length n in the language using at most $p(n)$ steps. But since, the acceptor is nondeterministic there seems to be no reason why it should also be a polynomial time recognizer.

Unfortunately, no one has been able to prove that there is an $\mathcal{NP}$ set which is not a $\mathcal{P}$ set. This has been an outstanding problem for about 50 years, and is considered the biggest unsolved problem in theoretical computer science. A few years ago, a prize of $1,000,000 was offered for a solution to this problem. Although many have looked at this problem, no real progress has been made, and it remains a large challenge for future researchers.

4.9.2 $\mathcal{NP}$ and co$\mathcal{NP}$

Most of the sets we know are in $\mathcal{P}$ or are $\mathcal{NP}$-complete. The following two problems were candidates $\mathcal{NP} \cap co\mathcal{NP}$ but have since been shown to be in $\mathcal{P}$.

LIN-INEQ:
INPUT: A set of linear inequalities

$$a_{11}\,x_1 \;+\; \cdots \;+\; ain\,x_n \;\geq\; b_1$$

$$\vdots$$

$$a_{r1}\,x_1 \;+\; \cdots \;+\; arn\,x_n \;\geq\; b_r$$

where the a's are integers and usually $r > n$.
QUESTION: Can you assign rational numbers to the x's so that all of the inequalities are satisfied?

Clearly this has an acceptor, simply guess the values for the x's. Some effort is required to show that the guessed x's don't have to be too big and so the acceptor can be a polynomial time acceptor. There is also a rejector. Guess nonnegative numbers c_1, c_2, $\cdots$ c_r so that

$$\sum c_j\,a_{j1} \;=\; \sum c_j\,a_{j2} \;=\; \cdots \;=\; \sum c_j\,a_{jn} \;=\; 0 \;\text{ and }\; \sum c_j\,b_j \;>\; 0.$$

Again some effort is required to show that the guessed c's don't have to be too big and so the rejector can be a polynomial time rejector.

About 30 years ago, it was show that there is a polynomial time recognizer for the sets of inequalities which can be satisfied. More strongly, if the system has a solution, then the solution can be found in polynomial time. In the ensuing years much effort has been expended on making really efficient algorithms for this problem because it allows one to solve linear programming problems which are the basis for many economic decisions.

PRIMES:
The set of natural numbers which have no non-trivial divisors, i.e. no divisors between 2 and $p-1$.

It is obvious that there is a polynomial time nondeterministic rejector for **PRIMES**. Guess a factor and then perform division and show that the guessed factor really is a factor. A nondeterministic acceptor is not quite so obvious. It turns out that a number p is prime iff it has a primitive root, that is, a number w so that every number between 1 and $p-1$ can be written as a power of w when the calculation is carried out mod p. This fact can be used to build an acceptor, but more work is required.

Within the last 10 years, a professor and two undergraduates in India showed that there is a polynomial time recognizer for **PRIMES**. At the moment their method does not give a really efficient algorithm, but one expects that efficient algorithms will appear shortly. It is important to note that their method does **not** produce a factor for a non-prime number, and so the encryption methods which depend on factoring for their security are still safe.

4.9.3 $\mathcal{NP}$-complete

One of the strongest lines of support for the supposition that $\mathcal{P} \neq \mathcal{NP}$ is that there is a long list of $\mathcal{NP}$-complete problems and NO polynomial time algorithms (recognizers) are known for any of them. Below we list a few of these complete problems. They are, of course, complete with respect to polynomial time many-one reduction.

(a) **SAT**
The set of Boolean expressions which can be *satisfied* in the sense that there is some assignment of TRUE and FALSE to the variables of the expresssion which makes the expression evaluate to TRUE.

(b) **HAM-PATH**
The set of graphs which have Hamiltonian paths. A graph G has a Hamiltonian path when one can start at some vertex and follow edges of G to visit every vertex so that each vertex is visited exactly once.

(c) **P-1-ECC**
The set of graphs which support perfect one-error correcting codes (also called perfect domination). G has a P-1-ECC if there is a subset C of the vertices so that no two vertices in C are adjacent and each vertex not in C is adjacent to exactly one vertex in C.

(d) **PARTITION:** The sets of natural numbers which can be split into two subsets $S1$ and $S2$ so that the sum of numbers in $S1$ is exactly equal to the sum of the numbers in $S2$.

(e) **NP-HALT**
The triples of a nondeterministic acceptor M, an input string x, and a time bound t given in unary, so that M accepts x in time at most t.

4.9.4 $\mathcal{NP}$-Hard Problems.

A problem $\mathcal{C}$ is called *$\mathcal{NP}$-hard* if an $\mathcal{NP}$-complete problem $\mathcal{B}$ is *polynomial time Turing reducible* to $\mathcal{C}$.

Often, there is some implication that $\mathcal{C}$ is is not *too* hard, meaning that $\mathcal{C}$ can be solved by a polynomial time non-deterministic algorithm, even though $\mathcal{C}$ may not be a **Yes/No**-problem. If this is the case, some authors call $\mathcal{C}$ an $\mathcal{NP}$-complete problem. For these authors a problem $\mathcal{C}$ is *$\mathcal{NP}$-hard* if it is *polynomial time Turing equivalent* to an $\mathcal{NP}$-complete problem.

As an example, we'll consider the Traveling Salesperson Problem (**TSP**).

TSP-LENGTH:
Traveling Salesperson Problem (**Length** of the solution)
INPUT: A complete weighted graph, W, i.e. a graph on n vertices with each of the possible $n(n-1)/2$ possible edges, and for each edge (v_i, v_j) a positive

integer the "weight" (or "distance").
OUTPUT: The minimum over all Hamiltonian circuits of W of the sum of the weights on the edges used in the circuit.

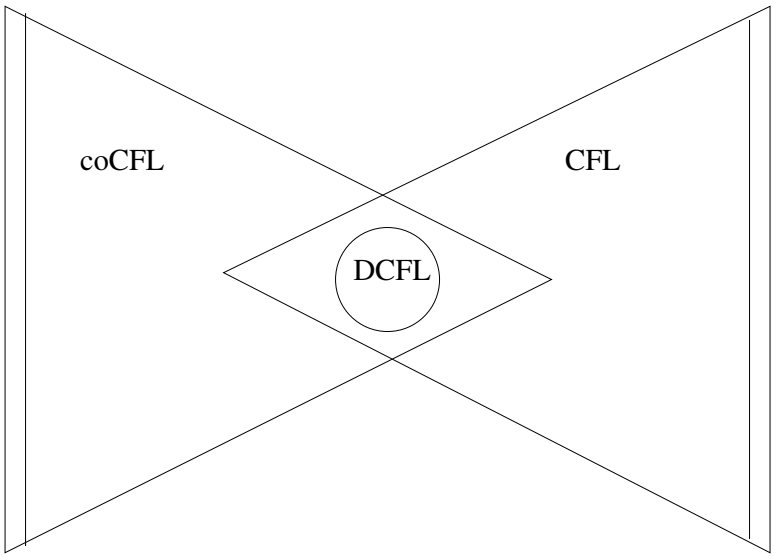
TSP-YES/NO:
Traveling Salesperson Problem (**YES/NO** version)
INPUT: A complete weighted graph, W, i.e. a graph on n vertices with each of the possible $n(n-1)/2$ possible edges, and for each edge (v_i, v_j) a positive integer the "weight" (or "distance") and a natural number B.
QUESTION: Does W have a Hamiltonian circuit in which the sum of the weights on the edges used in the circuit is at most B?

Obviously, **TSP-YES/NO** $\leq_{\mathbf{T}}$ **TSP-LENGTH** because if you knew the minimum length, you could simply compare it to B. On the other hand, if you had an upper bound, say B_0, then by putting W and B_0 into the YES/NO procedure, you could verify that B_0 is an upper bound. By binary search, i.e. successively halving B and testing this new B, you can arrive at the minimum length in about $\log B_0$ iterations. So if the YES/NO procedure takes polynomial time in the size of the input and you call it $\log B_0$ times, then the total time used will also be polynomial bounded.

Hence **TSP-LENGTH TSP-YES/NO** $\leq_{\mathbf{T}}$ **TSP-YES/NO**,
and so **TSP-LENGTH TSP-YES/NO** $\equiv_{\mathbf{T}}$ **TSP-YES/NO**.

4.10 The World of CFL

CFL is the class of Context Free Languages. Those languages which can be specified by contex free grammars, or equivalently those languages which have nondeterministic pushdown acceptors (NPDAs). Correspondingly, co**CFL** are those languages which have NPDA rejectors. The intersection of these two classes is **CFL** $\cap$ co**CFL** and languages in this class have NPDA recognizers.



DCFL are the languages which have deterministic PDA recognizers. As, we will show (and is well known) **DCFL** properly contains **Regular**. It is probably not as well know that **DCFL** is a proper subclass of **CFL** $\cap$ co**CFL**, but we will also show this.

Theorem 5.

$$\textbf{Regular} \subsetneq \textbf{DCFL} \subsetneq \textbf{CFL} \cap \textit{co}\textbf{CFL} \subsetneq \textbf{CFL}.$$

Since we want to display these inclusions in a diagram that gives us a hardness ordering on problems, we want to choose a notion of reduction that displays these inclusions but does not overly subdivide each of these classses. We will use transformations computable by deterministic pushdown machines which we will explain after we indicate where some languages are in these classes.

4.10.1 Regular and DCFL

There are some easy examples which distinguish theses classes.
Regular:
$\{ a^n \mid n > 0 \}$ That this set is regular is obvious and wll known.
DCFL $-$ **Regular:**

$\{ a^n b^n \mid n > 0 \}$ This is the traditional example of a **CFL** which can be shown not to be regular using the Pumping Lemma. It's obvious that a PDA can deterministically push a's and then pop them when matching b's are found, and can reject when the counts don't agree or if the a's and b's are not in order.

4.10.2 CFL $\cap$ coCFL $\neq$ DCFL

PALINDROMES is the set of strings that read the same forward and backward.

Obviously PALIDROMES $\in$ **CFL** by the familar context free grammar, or by the stack first half and compare it with the second half algorithm which can be implemented on a NDPDA which only has to *guess* where the middle is.

PALINDROMES $\in$ **coCFL** because the only way a string can fail to be a palindrome is that a character in the first half of the string does not match the corresponding character in the second half. So, a nondeterministic algorithm "guesses" i by remembering the first i characters on a stack, then it sweeps over the string until it "guesses" it is at the corresponding character, checks that this character is different from the character on top of the stack and then uses the the contents of the stack to count out the remaining characters in the string and verify that the "guessed" positions were, in fact, corresponding, and shows that the string is not a palindrome.

That PALINDROMES are not in **DCFL** can be shown by considering long enough palindromes which must be accepted. If a deterministic machine accepts one of these and ends with an empty stack, follow this palindrome with another palindrome. If this second palindrome is identical with the first then the whole string of 2 palindromes should be accepted, while if the second palindrome is not identical with the first the whole string should not be accepted. So, any deterministic machine must give an incorrect answer in one of these two cases.

4.10.3 CFL $\neq$ coCFL

CFL (and also other classes defined by nondeterministic acceptors) is closed under union. The idea is, if A is a **CFL** set and B is a **CFL** set then you have an acceptor, say ACC_A, for A and an acceptor, say ACC_B, for B. Then when given an input x, your acceptor for $A \cup B$ "guesses" whether to use ACC_A or ACC_B and can thus accept x, if $x \in A \cup B$, and obviously this procedure will never accept x if $x \notin A \cup B$.

So if **CFL** were closed under complement, **CFL** would also be closed under intersection by the De Morgan's law

$$A \cap B = \overline{\overline{A} \cup \overline{B}}.$$

But, **CFL** is not closed under intersection because $\{a^n\, b^n\, c^n\}$ is not a context-free language (see the chapter on Grammars), and this non-context-free language is the intersection of $\{a^n\, b^n\, c^j\}$ and $\{a^j\, b^n\, c^n\}$ which are both context-free languages. (In the above, these are infinite languages and the "exponents" range over the positive integers.)

A similar argument can be used to show that **CFL** $\neq$ **DCFL** because **CFL** is closed under union, but **DCFL** is not closed under union. As an example consider the two languages $\{ a^n b^n \}$ and $\{ a^n b^{2n} \}$. (Here, this notation indicates infinite sets where n ranges over the positive numbers.) Clearly both these languages are in **DCFL** because a deterministic machine can simply stack all the a's and then for the first language pop one a for each b, and for the second language a deterministic machine could stack all the a's and then pop one a for every two b's. But, no deterministic machine can accept the union of these two languages as can be shown by considering long enough strings and showing that the deterministic machine will either accept a string not in the union or fail to accept a string in the union.

4.10.4 DPDA Reduction

The appropriate reduction for the World of **CFL** seems to be DPDA TRANSDUCTION, that is the transformation from one set to another should be computed by a deterministic PDA. (The next subsection has an example.)

Since obviously, homomorphisms can be computed by DPDA's, Greibach's $\mathcal{L}_0$ which is complete for **CFL** with respect to reduction by homomorphisms is also a complete set for **CFL** under DPDA reduction.

Almost obviously,

$$A \in \textbf{DCFL} \text{ and } B \in \textbf{DCFL} \implies A \leq B \quad \text{and} \quad B \leq A.$$

The transformation for $A \leq B$ (with B non-trivial) is

$$f(x) = \begin{cases} b_{\text{YES}} & \text{if} \quad \text{DPDA}_A(x) = \text{YES} \\ b_{\text{NO}} & \text{if} \quad \text{DPDA}_A(x) = \text{NO}. \end{cases}$$

We should remark that B can even be taken as a (very simple) regular language, because all the "work" is being done by the transforming fuction, and so B can be very simple.

4.10.5 DPDA example

Here, we give an example of a simple computation carried out by a deterministic pushdown automaton (DPDA). We will use this kind of machine to compute the *difference* between two numbers.

Assume that two binary numbers x and y are to be fed into the machine as pairs of bits, i.e.

$$\binom{x}{y} = \binom{x_n \ \ldots \ x_0}{y_m \ \ldots \ y_0}.$$

We want the machine to output the binary number $|x - y|$ which is the difference between x and y.

We all know the usual subtraction algorithm which progresses from low order bits to high order bits and substracts y_0 from x_0 and if necessary propagates

a *borrow* to the next pair of bits. This algorithm can be carried out by a finite state machine which outputs the answer's bits one after the other. But, for a finite state machine to compute the *difference*, it seems that the machine would have to know whether $x \geq y$ or $x < y$.
(It seems that the FSM could make one nondeterministic "guess" as to the comparative sizes of x and y, and then correctly compute $|\, x - y \,|$. This argument seems to say that for transducers, nondeterminism can be more powerful, even though deterministic and nondeterministic finite state acceptors are equally powerful.)

Here, we want to show that difference can be computed by a deterministic PDA. The basic idea is that the PDA can compute the low order bits of *both* $x - y$ and $y - x$ and store these as pairs of bits on the stack, and then when the machine has determined whether $x \geq y$ or $x < y$, it can make appropriate use of these stored bits.

If $x \geq y$ then the machine can simply use the standard algorithm to compute and store the low bits of the difference, and when it has found that x is larger, it can continue propagating the *borrow* (if necessary) and end up with the correct answer on the stack, and then output this answer.

If $x < y$ the machine will need a "trick" to compute the appropriate low order bits. Say $y = 2^r y_1 + y_0$ and $x = x_0 < 2^r$, then the desired output is

$$\begin{aligned} y - x &= 2^r y_1 + (y_0 - x_0) \\ &= 2^r y_1 - (x_0 - y_0). \end{aligned}$$

The "trick" is that

$$-(x_0 - y_0) \equiv \overline{(x_0 - y_0)} + 1 \pmod{2^r}$$

where the overbar indicates bitwise complement, i.e. in a string of 0's and 1's all the 0's are replaced by 1's and all of the 1's are replaced by 0's. This follows because

$$(x_0 - y_0) + \overline{(x_0 - y_0)} = 11 \ldots 1 = 2^r - 1 \equiv -1 \pmod{2^r}.$$

So, the PDA computes and stacks in parallel the bits of $x - y$ and the bits of $\overline{(x - y)} + 1$.

Then, if it runs out of y's input bits before running out of x's input bits, it can stack the remaining input bits of x, and then output from the stack resulting in the high order bits of x followed by the bits of $x - y$. On the other hand, if it runs out of x's input bits before y's input bits, then it can stack the remaining bits of y, and then output the high order bits of y followed by the bits of $\overline{(x - y)} + 1$. In either case, the PDA will correctly compute the difference of x and y. Notice that he machine will only need one extra bit of *local* memory to remember which of bit of a pair to ouput.

4.11 Analogies

Somewhat surprisingly, the interrelations between classes around **CFL** seems to mirror those between the classes around **RE** and also the relations beween classes around $\mathcal{NP}$.

The Analogy between Various Classes

RE	$\leftrightarrow$	$\mathcal{NP}$	$\leftrightarrow$	**CFL**
co**RE**	$\leftrightarrow$	co$\mathcal{NP}$	$\leftrightarrow$	co**CFL**
REC	$\leftrightarrow$	$\mathcal{NP} \cap$ co$\mathcal{NP}$	$\leftrightarrow$	**CFL** $\cap$ co**CFL**
PRIM	$\leftrightarrow$	$\mathcal{P}$	$\leftrightarrow$	**DCFL**.

One might hope that these analogies might be made concrete by finding mappings between analogous classes. So that sets in one class would correspond in a one-to-one fashion with sets in an analogous class. But there are difficulties. For example, **PRIM** and $\mathcal{P}$ are both closed under union, and one might expect the mapping between these classes would respect union. On the other hand, **DCFL** is **not** closed under union, so a mapping between **DCFL** and **PRIM** or $\mathcal{P}$ should not respect unions.

4.12 Exercises

Ex 4.1. Show that **coRE** is closed under many-one reduction. That is, $A \in$ **coRE iff** $\exists B \in$ **coRE** so that $A \leq_{\mathbf{m}} B$.

Ex 4.2. Traveling Salesperson Problem (**FIND** the solution):
TSP-FIND:
INPUT: A complete weighted graph, W, i.e. a graph on n vertices with each of the possible $n(n-1)/2$ possible edges, and for each edge (v_i, v_i) a positive integer the "weight" (or "distance").
OUTPUT: The sequence of edges used in the Hamiltonian circuit of W which has the minimium sum of the weights on the edges used in the circuit.

Show that **TSP-FIND** is $\mathcal{NP}$-Hard by showing that this problem is polynomial time Turing equivalent to **TSP-YES/NO**.

Ex 4.3. (**SAT** and **3-SAT**)
SAT:
INPUT: A Boolean expression $B(x_1, x_2, \ldots, x_n)$ in clause form, i.e. and ANDing together of clauses which consist of an ORing together of some variables and some complemented variables.
QUESTION: Is there an assignment of TRUE and FALSE to the varaibles which makes the expression evaluate to TRUE?
3-SAT:
INPUT:A Boolean expression $B(x_1, x_2, \ldots, x_n)$ in clause form in which each clause has exactly 3 literals (a literal is a variable or a complemented variable).
QUESTION: Is there an assignment of TRUE and FALSE to the varaibles which makes the expression evaluate to TRUE?

Show that **SAT** $\leq_{\mathbf{m}}^{\text{poly}}$ **3-SAT**, and hence if **SAT** is $\mathcal{NP}$-complete then **3-SAT** is $\mathcal{NP}$-complete. (Note that we've stated this as problems rather than set. The set corresponding to a problem is the set of YES instances of the problem.)

Ex 4.4. Show that $\mathcal{NP}$ is closed under polynomial time many-one reduction, that is $A \in \mathcal{NP}$ **iff** $\exists B \in \mathcal{NP}$ so that $A \leq_{\mathbf{m}}^{\text{poly}} B$.

Ex 4.5. Show that $A \in \mathcal{NP}$ **iff** $A \leq_{\mathbf{m}}^{\text{poly}} \mathcal{NP} - HALT$. Generalize this to $A \in \mathcal{NP}$ **iff** $A \leq_{\mathbf{m}}^{\text{poly}} C$, where C is any $\mathcal{NP}$-complete problem.

Ex 4.6. Show that A is $\mathcal{NP}$-complete **iff**

(a) $A \in \mathcal{NP}$ and

(b) $\mathcal{NP} - HALT \leq_{\mathbf{m}}^{\text{poly}} A$.

Generalize this to A is $\mathcal{NP}$-complete **iff**

(a) $A \in \mathcal{NP}$ and

(b) $C \leq_{\mathbf{m}}^{\text{poly}} A$, where C is **any** $\mathcal{NP}$-complete problem.

Ex 4.7. Greibach Language (variant from G. Révész).
Show that the language given by the following grammar does generate the Greibach hardest contect-free language:

$$
\begin{array}{lll}
S \longrightarrow XbYcZd & Y \longrightarrow cZdXY & Y \longrightarrow YcZdX \\
Y \longrightarrow \lambda & X \longrightarrow Zc & Z \longrightarrow e_1 Z \\
Y \longrightarrow YY & Z \longrightarrow \lambda & Z \longrightarrow e_2 Z \\
Y \longrightarrow e_1 Y \overline{e_1} & Z \longrightarrow bZ & Z \longrightarrow \overline{e_1} Z \\
Y \longrightarrow e_2 Y \overline{e_2} & Z \longrightarrow cZ & Z \longrightarrow \overline{e_2} Z
\end{array}
$$