Completeness

William Shoaff

A PDF version of this set of notes

Reductions

We want a concepts that defines *at least as hard*, in that if we could solve problem *A* we could solve problem *B*, so *A* is at least as hard as problem *B*. This is the concept of *reduction*. So far we have used computable reductions: ones that could be implemented by some terminating Turing machine. Now we want to limit the resources used in a reduction. That is, we need to consider the efficiency of the reduction.

Polynomial time bounded reductions We say language L_1 reduces to language L_2 if there is a function

$$R : \Sigma_1 \rightarrow \Sigma_2$$

that is computable on a deterministic Turing machine in time $O(n^k)$and $w \in L_1$ if and only if $R(w) \in L_2$.

Logarithmic space bounded reductions We say language L_1 reduces to language L_2 if there is a function

$$R : \Sigma_1 \rightarrow \Sigma_2$$

that is computable on a deterministic Turing machine in space $O(\log n)$ and $w \in L_1$ if and only if $R(w) \in L_2$.

Note that logarithmic space reductions imply polynomial time reductions. There are at most $O(nc^{\log n})$ configurations on input *w*where |*w*|=*n*. Since the machine is deterministic, no configuration can repeat. Thus the computation has length at most $O(n^k)$ for some *k*.

See examples of reductions in the text [1].

HAMILTON PATH can be reduced to SAT: Given a graph *G* we construct a Boolean expression *R*(*G*) such that *G* has a Hamilton path if and only if *R*(*G*) is satisfiable.. Suppose *G* has *n* vertices: 1, 2, ..., *n*; *R*(*G*) will have n^2 variables

$$x_{ij}, 1 \leq i,j \leq n$$

where x_{ij} will represent that ``node j is the i node of a Hamilton path'' (which may be true or false). $R(G)$ is made of clauses:

The first series of clauses is true when each node j in the graph lies on a Hamilton path:

$$(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj}), \quad j = 1, \ldots, n$$

node j is either the first, second, ...nth node on such a path.

The second series of clauses is true when node j is not both the ith and kth node on such a path:

$$(\neg x_{ij} \vee \neg x_{kj}), \quad j = 1, \ldots, n, \quad i \neq k$$

Next some vertex must be the ith node on the path:

$$(x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}), \quad i = 1, \ldots, n$$

And no two vertices can be the ith node on the path:

$$(\neg x_{ij} \vee \neg x_{ik}), \quad i = 1, \ldots, n, \quad j \neq k$$

Finally, vertex j can come right after i only if (i, j) is an edge in G, so for each pair (i, j) that is not an edge in Gwe include:

$$(\neg x_{ki} \vee \neg x_{k+1,j}), \quad k = 1, \ldots, n-1.$$

The expression $R(G)$ is the conjunction of all these clauses.

To show that R is a reduction from HAMILTON PATH to SAT we must show: (1) for any graph G, expression $R(G)$ has a satisfying truth assignment if and only if G has a Hamilton path, and (2) R can be computed in logarithmic space.

Suppose $R(G)$ has a truth assignment T.

For each j there is a unique i such that $T(x_{ij})$ = **true**otherwise the clauses

$$(\neg x_{ij} \vee \neg x_{kj}), \quad \text{and}(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj})$$

cannot all be satisfied. Similarly for each i there is a unique j such that $T(x_{ij})$=**true**, or not all of

$$(\neg x_{ij} \vee \neg x_{ik}), \quad \text{and}(x_{i1} \vee x_{i2} \vee \cdots \vee x_{in})$$

can all be satisfied.

Hence T can be thought of as a permutation of the vertices of G $\pi(i) = j$ where $T(x_{ij})$=**true**

Moreover, the clauses

$$(\neg x_{ki} \vee \neg x_{k+1,j})$$

where (i,j) is not an edge guarantee that for all k $(\pi(i), \pi(j))$ is an edge of G and thus

$$(\pi(1), \pi(2), \ldots \pi(n))$$

is a Hamilton path.

Conversely, if G has a Hamilton path

$$(\pi(1), \pi(2), \ldots \pi(n))$$

then the truth assignment $T(x_{ij})$ = **true** if $\pi(i) = j$ and $T(x_{ij})$ = **false** if $\pi(i) \neq j$ satisfies the clauses of G.

Now let's show we use only logarithmic space in the computation of R. Given G a Turing machine M outputs $R(G)$ as follows:

1. Write n the number of vertices in G (in binary [logarithmic space])
2. Generate on the output tape the clauses that do not depend on G (the first four sets of clauses) Here, M just needs three counters for i, j, and k
3. For the remaining clauses describing edges M generates one by one all clauses of the form:

$$(\neg x_{ki} \vee \neg x_{k+1,j}), \quad k = 1, \ldots, n-1$$

   M then looks at its input to see if (i,j) is an edge of G and if not outputs the Boolean clause. Again the counters i, j, and k are sufficient to complete this computation.

REACHABILITY can be reduced to CIRCUIT VALUE.

Given a graph G construct a variable free circuit $R(G)$with output **true** if and only if there is a path from node 1 to node n in G.

Gates of $R(G)$ are of the form

1. g_{ijk} where $1 \leq i,j \leq n$ and $0 \leq k \leq n$
2. h_{ijk} where $1 \leq i,j,k \leq n$

g_{ijk} will output **true** if and only if there is a path in G from i to j not using any intermediate node bigger than k.

h_{ijk} will output **true** if and only if there is a path in G from i to j not using any intermediate node bigger than k, but using k as an intermediate node.

For k=0, g_{ij0} are input gates that are **true** if i=$j$or (i,j) is an edge of G and **false** otherwise.

For $k = 1,\ldots n$, h_{ijk} is an **AND** gate with predecessors $g_{i,k,k\text{-}1}$ and $g_{k,j,k\text{-}1}$

For $k = 1,\ldots n$, g_{ijk} is an **OR** gate with predecessors $g_{i,j,k\text{-}1}$ and $h_{i,j,k}$

Finally g_{1nn} is the output gate.

We will show by induction on k that g_{ijk} will output **true** if and only if there is a path in G from i to j not using any intermediate node bigger than k and h_{ijk} will output **true** if and only if there is a path in G from i to j not using any intermediate node bigger than k, but using k as an intermediate node.

For k=0, g_{ij0} will output **true** if and only if there is a an path (edge) from i to j (this includes the empty path from $i$to i when j=i).

Suppose the outputs of g and h gates are as described up to k-1for some $k \geq 1$. Since

$h_{ijk} = (g_{i,k,k-1} \wedge g_{k,j,k-1})$ h_{ijk} will output **true** if and only if $g_{i,k,k\text{-}1}$ and $g_{k,j,k\text{-}1}$ are **true**.

And since $g_{ijk} = (g_{i,j,k-1} \vee h_{i,j,k})$ g_{ijk} will output **true** if and only if one of $g_{i,j,k\text{-}1}$ or $h_{i,j,k}$ is **true**.

In particular, g_{1nn} will be true if and only if there is a path from 1 to n in G.

Finally, $R(G)$ can be computed in logarithmic space by going over all vertices i, j and $k$and output the appropriate edges and gate types for the variables. (See the Floyd-Warshall algorithm from a text on algorithms).

Completeness

Definition Let *C* be a complexity class, and let $L \in C$. The language *L* is *C-complete* if any language $L' \in C$ can be reduced to *L*.

It is not clear that complete problems even exist. But they are a central concept, they capture the difficulty of a class (they are at least as hard as every other problem in the class). Being able to solve a complete problem implies we can solve (in theory at least) any problem in that class.

The existence of important, natural problems that are complete for a class tend to make a class significant; absence of such problems indicate the class may be artificial.

The most common use of completeness is to derive a *negative complexity result*: a complete problem *P*for class *C* is the least likely to belong to a subclass $C' \subseteq C$, provided the class *C*' is closed under reductions.

Definition: A class *C*' is closed under reductions if whenever *L* is reducible to *L*' and $L' \in C'$ then $L \in C'$.

Proposition: $\mathbb{P}$, $\mathbb{NP}$, $co\mathbb{NP}$, $\mathbb{L}$, $\mathbb{NL}$, $\mathbb{PSPACE}$, $\mathbb{EXP}$ are all closed under reductions.

For example, consider $\mathbb{P}$ and suppose *L* is reducible to $L' \in \mathbb{P}$, using logarithmic space reduction *R*. To decide: $w \in L$ construct the reduction *R*(*w*) (in logarithmic space, hence polynomial time) and then decide if $R(w) \in L$using the Turing machine *M* that decides *L* in polynomial time.

Complete problems are valuable since if we can show an $\mathbb{NP}$ complete problem is in $\mathbb{P}$then $\mathbb{P} = \mathbb{NP}$ (and so on for other complexity classes).

Theorem CIRCUIT VALUE is $\mathbb{P}$ complete.

AND, OR, and NOT can be used in an instance CIRCUIT VALUE. If we exclude NOT gates the problem remains $\mathbb{P}$ complete. Such (monotone) circuits are less expressive than general circuits, but by appling De Morgan's laws we can move all NOTs to the input and then change true to false and vice versa.

Cook's Theorem SAT is $\mathbb{NP}$ complete.

Theorem Let *L* be complete for class $\mathbb{C}$. If $L' \in \mathbb{C}$ and *L* reduces to *L*', then *L*' is complete for class $\mathbb{C}$.

Let *L*'' be any language in class $\mathbb{C}$. Since *L* is $\mathbb{C}$-complete there is a reduction *R*'' from *L*'' to *L*. Let *R*' be the reduction from *L* to *L*'. Then, since the composition of reductions is a reduction, $R = R'' \circ R'$ is a reduction from *L*'' to *L*'

Bibliography

1

C. H. PAPADIMITRIOU, *Computational Complexity*, Addison-Wesley, 1994.

William Shoaff
2001-03-14