McGill University COMP531 Winter 2010 **Instructor: Phuong Nguyen**

Assignment 2 solution

Question 1 (5pt) First, suppose that there is a p-bounded QBF proof system F. Then there is a polynomial p so that for every valid QBF formula A there is an F-proof π of A of size $|\pi| \leq p(|A|)$. So we have

$$A \in \text{QBF-TAUT} \quad \text{iff} \quad \exists \pi, |\pi| \leq p(|A|), F(\pi) = A$$

The relation $F(\pi) = A$ is a polytime relation, so this definition of QBF-TAUT shows that it is in **NP**, hence **PSPACE** $\subseteq$ **NP**. Since **NP** $\subseteq$ **PSPACE**, we have **PSPACE** $=$ **NP**.

Now we prove the other direction. Suppose that **PSPACE** $=$ **NP**, we will show that there is a p-bounded QBF proof system. By the hypothesis, there is a polytime relation R and a polynomial t so that for all A:

$$A \in \text{QBF-TAUT} \quad \text{iff} \quad \exists y, |y| \leq t(|A|), R(A, y)$$

Define a polytime function F as follows:

$$F(A, y) = \begin{cases} A & \text{if} \quad |y| \leq t(|A|) \wedge R(A, y) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since R is a polytime relation, the function F is computable in polytime. Furthermore, if A is a valid QBF formula then by the **NP**-definition of QBF-TAUT there is y of length at most $t(|A|)$ that makes $R(A, y)$ true. For this choice of y, $F(A, y) = A$, so F is an onto function. Also, for each A in QBF-TAUT, the F-proof (A, y) of A has length at most $|A| + t(|A|)$ which is a polynomial in $|A|$. So F is a p-bounded QBF proof system.

Question 2 (5pt) First suppose that $\mathbf{NP}^A = \mathbf{NP}$. Clearly both A and its complement A^c belong to $\mathbf{NP}^A$:

- The oracle Turing machine for A works as follows: on input x it asks the oracle the membership of x in A, and accepts if and only if the answer is YES.
- Similarly, the oracle Turing machine for A^c works as follows: on input x it asks the oracle the membership of x in A, and accepts if and only if the answer is NO.

By the assumption that $\mathbf{NP}^A = \mathbf{NP}$, we have both A and A^c belong to **NP**, hence $A \in \mathbf{NP} \cap co\text{-}\mathbf{NP}$.

Now for the other direction, suppose that $A \in \mathbf{NP} \cap co\text{-}\mathbf{NP}$. This means that there are polytime nondeterministic Turing machines M_1 and M_2 that accepts A and A^c. To show that $\mathbf{NP}^A \subseteq \mathbf{NP}$, let M be an polytime oracle nondeterministic Turing machine with access to A. We show that M can be simulated by nondeterministic Turing machine M' in polytime.

The machine M' works as follows. On input x it simulates M, and every time M as a query of the form $y \in A$? M' runs both M_1 and M_2 on y, Exactly one of these will accept, and M' continues to simulate M with oracle answer YES if M_1 accepts and with oracle answer NO if M_2 accepts. Since M, M_1, M_2 all run in polytime, M' also runs in polytime time.

Question 3 (Exercise 5.13(a) in the text) (5pt) We give a formula defining membership in VC-DIMENSION that has the right alternation of bounded quantifiers. By definition: (C, k) is in VC-DIMENSION iff

$$\begin{gathered}\text{there is a set } X \subseteq \{0,1\}^n,\ |X| = k \text{ such that}\\ \text{for all } X' \subseteq X,\\ \text{there is some } i,\ 1 \le i \le 2^l \text{ such that } X' = X \cap \{x \ :\ C(i,x) = 1\}\end{gathered}$$

This statement already has the required form of quantifier alternation (i.e., $\exists\forall\exists$). We need to argue that the quantified variables are of bounded length, and the relation $X' = X \cap \{x \ :\ C(i,x) = 1\}$ is computable in polytime.

First observe that a set X of k elements has precisely 2^k subsets, so if $\mathcal{S} = \{S_1, S_2, \ldots, S_{2^\ell}\}$ shatters X, $\mathcal{S}$ must have at least 2^k elements. It follows that $k \le \ell$. Also, it is obvious that $n, \ell \le |C|$. So in the following, it suffices to give the upper bounds on the lengths of the quantified variables as polynomials in n, k, ℓ.

The set X can be encoded simply as a sequence of k n-bit strings (members of X), and this is a string of length nk. Then X' can be easily encoded as a binary string of length k whose bits indicate membership in X' for each element in X. The last quantified variable, i, can be written as a binary string of length ℓ.

Finally, the relation $X' = X \cap \{x \ :\ C(i,x) = 1\}$ can be verified in time polynomial in nk by verifying that for each element x of X, $C(i,x) = 1$ iff $x \in X'$. This requires running through all elements of X (thus k loops), for each element x computing $C(i,x)$ (taking time polynomial in the size of $|C|$) and verifying that the output of C agrees with the membership of x in X'. All these operations take polynomial time in $|C|$.

Question 4 (10pt) Following the "elementary school algorithm" for multiplication, we compute $MULT(x,y)$ by writing down the following table and then computing the sum of all rows. Let $|y| = n$ and y be the binary string $y_{n-1}y_{n-2}\ldots y_2y_1y_0$. The table has n rows; the i-th row is either (i) all 0 string if $y_i = 0$, or (ii) the string x shifted left by i bits, if $y_i = 1$. In other words:

$$MULT(x,y) = \sum_{i=1}^{n} x^i$$

and

$$x^i = \begin{cases} 0 & \text{if } y_i = 0 \\ x00\ldots0 \ \ (x \text{ padded with } i \text{ 0's}) & \text{if } y_i = 1 \end{cases}$$

We will argue that this algorithm can be carried out in $\mathcal{O}(\log(n))$ space.

First, of course we don't have enough space to write down the above table, but this is not a problem, because the j-th least significant bit of the i-th row is

$$\begin{cases} 0 & \text{if } j < i \text{ or } y_i = 0 \\ x_{j-i} & \text{otherwise} \end{cases}$$

So we will compute this bit only when we need (and then discard the result).

We will sum up these rows and output the bits of the sum starting from the least significant bit. This means that for each column we compute the sum of all bits in the column together with the carry from the previous columns, and write down the last bit of the sum. For this we need to maintain the current carry over from the previous column (initially this carry over is 0). The next Claim shows that the carry over can be stored in $\mathcal{O}(\log(n))$ space.

Claim: the value of the carry over is always $\le n$.

We prove the Claim by induction on j that the carry over from column j is always $\leq n$. For the base case, the carry for the left-most column is 0. For the induction step, assume that the carry over for column j is at most n. Then the sum of all bits in column $j+1$ and the carry over from column j is at most $n + n = 2n$. There the carry over for column $j+1$ is at most $2n/2 = n$. This proves the Claim.

Since the values of the carries over are $\leq n$, they can be maintained using $\lceil \log(n) \rceil$ bits. We will also need to maintain an index for the current column, and for summing the bits in the column an index for the rows. In total, we need $\mathcal{O}(\log(n))$ space.

Question 5 (10pt) **First we show that HornSAT is in P**. We will give a polytime algorithm for HornSAT. Given a Horn formula A, our algorithm actually searches for a satisfying truth assignment for A, and output NO if no such truth assignment is found. It is based on the following observation.

Claim: If A is a Horn formula all of whose clauses have at least one negative literal, then A is satisfiable by the truth assignment that assigns FALSE to every variable.

The proof of the Claim is straightforward: Suppose that every clause in A has at least one negative literal, then the truth assignment that assigns FALSE to every variable satisfies every clause, and hence A as well.

Thus let S be the set of clauses in A. The goal is to search for a truth assignment that satisfies all clauses in S. By the Claim, we know that if all clauses in S have at least one negative literal, we can conclude that A is satisfiable by the truth assignment that make every variable FALSE. Otherwise, S must contain a clause whose only literal is a positive literal p. Any truth assignment that satisfies A must set p to TRUE. We can now simplify the clauses in S: if a clause C contains p, it is already true because we have set p to TRUE, and hence can be eliminated from S. If a clause C contains $\neg p$ then we can eliminate $\neg p$ from it.

After this simplifying procedure, what we have left is a set S' of simplified clauses that contains at most one clause less than S, and S' doesn't contain p. We can now recursively search for a satisfying truth assignment for S'.

Formally the algorithm is as follows: Let S be the set of clauses of A.

HornSAT-SEARCH(S):

```
1. $\tau \leftarrow \varnothing$ (the current truth assignment)
2. while $S$ is not empty do
3.     if all clauses in $S$ contains at least a negative literal:
4.         append $\tau$ with the truth assignment that assigns FALSE to every variables in $S$
5.         output $\tau$
6.     else
7.         if there is an empty clause in $S$, output NO;
8.         else
9.             let $p$ be a clause in $S$ that contains only a positive literal
```

```
10.             $\tau \leftarrow \tau \cup \{p = TRUE\}$
11.             simplify $S$ as discussed above
12.         end if
13.     end if
14. end while
15. Output $\tau$
```

The proof that the algorithm is correct is based on the Claim above. First we show that if that if there is a truth assignment that satisfies all clauses in S then the algorithm outputs one such truth assignment. This is proved by proving by induction on the size of the set S. The base case is obvious. For the induction step, consider two cases:

- All clauses in S contains at least one negative literal. Then by the Claim S is satisfiable and the algorithm outputs one satisfying truth assignment.
- There is a clause in S that contains only a positive literal. Then any satisfying truth assignment for S must set this literal to TRUE, and the set S' obtained from S after the simplification is also satisfiable. Now we can apply the induction hypothesis.

Next we show that if S is unsatisfiable then the algorithm will output NO. So suppose that S is unsatisfiable. If S contains an empty clause then we output NO in line 7. Otherwise, by the Claim, S must contain a clause with only a positive literal. The set S' obtained from S by the simplification procedure is also unsatisfiable, so it contains either an empty clause, or a clause with only a positive literal, etc. This process of applying the simplification procedure must stop at some time, and at that point we must arrive at a set that contains an empty clause, and the algorithm will output NO.

Now we show that HornSAT is P-hard. This is done by reducing the Circuit Value Problem (CVP) to HornSAT. Thus, given a Boolean circuit C that takes only constant (0,1) inputs (i.e., there is no variable input), we want to construct a Horn formula A so that C outputs 1 iff A is satisfiable.

The reduction: Let the gates in C be $g_1, g_2, \ldots, g_n$ where g_n is the output gate, so that each gate g_i is either a constant 0 or 1, or for some $j, k < i$:

$$g_i = g_j \wedge g_k \text{ or } g_i = g_j \vee g_k \text{ or } g_i = \neg g_j$$

Our reduction uses the double-rail logic: we introduce for each gate g_i two variables p_i and q_i with the intended meanings $p_i = g_i$ and $q_i = \neg g_i$. The formula A is the conjunction of the following clauses: for each i we have two clauses below that say that $p_i = \neg q_i$:

$$p_i \vee \neg q_i \text{ and } \neg p_i \vee q_i$$

and

- if g_i is a constant gate 0, then we have two clauses $\neg p_i$ and q_i;
- if g_i is a constant gate 1, then we have two clauses p_i and $\neg q_i$;

- if g_i is an $\wedge$-gate with inputs from gates g_j and g_k then we have the following clauses that say that $p_i \leftrightarrow p_j \wedge p_k$:
$$\neg p_i \vee p_j, \quad \neg p_i \vee p_k, \quad \neg p_j \vee \neg p_k \vee p_i$$
- if g_i is an $\vee$-gate with inputs from gates g_j and g_k then we have the following clauses that say that $p_i \leftrightarrow p_j \vee p_k$:
$$\neg p_i \vee \neg q_j \vee \neg q_k, \quad \neg p_j \vee p_i, \quad \neg p_k \vee p_i$$
- if g_i is a $\neg$-gate with input from gate g_j then we have the following clauses that say that $p_i \leftrightarrow \neg p_j$:
$$\neg p_i \vee \neg p_j, \quad \neg q_j \vee p_i$$

Finally, we have clause g_n which says that the output of the circuit is 1.

Correctness of reduction: Observe that each clause described above has at most one positive literal, so A is a Horn formula. It remains to show that A is satisfiable iff C outputs 1. First, suppose that A is satisfiable by a truth assignment τ. It can be shown by induction on i that $\tau(p_i)$ is precisely the value of the gate g_i and q_i is its negation. Hence the fact that $\tau(p_n) = 1$ implies that the output of C is 1. Next, suppose that C outputs 1. Then define a truth assignment τ to the variables p_i and q_i by letting $\tau(p_i)$ be the value of g_i and $\tau(q_i)$ be its negation. Then it can be shown that τ satisfies A.

Question 6 (10pt) **First we show that UDIST is in NL.** Notice that (G, s, t, d) is in UDIST iff (i) there is a path from s to t of length d, and (ii) there is no shorter path from s to t. (i) can be easily computable by an **NL** algorithm that guesses a path of length exactly d from s and t. By the same argument (ii) is computable by some algorithm in *co*-**NL**, but since *co*-**NL** = **NL** we also have (ii) can be done in **NL**. Thus UDIST is in **NL**.

Now we show that UDIST is NL-hard. We do this by showing that PATH is logspace reducible to UDIST.

The reduction: Given a directed graph H and two vertices x, y we want to construct an undirected graph G with two vertices s, t and a distance d so that

there is a path from x to y in H iff
the distance between s and t in G is exactly d.

Let n be the number of vertices in H and let $v_1, v_2, \ldots, v_n$ be all vertices of H, and without loss of generality suppose that $x = v_1, y = v_2$. The graph G has n copies of $v_1, v_2, \ldots, v_n$, the i-th copy are
$$u_{i,1}, u_{i,2}, \ldots, u_{i,n}$$
(for $1 \le i \le n$). The edges of G are as follows. For the i-th copy, there is an edge between $u_{i,\ell}$ and $u_{i,k}$ iff there is a (directed) edge in G from v_ℓ to v_k. In addition, between the i-the copy and the $(i+1)$-st copy there is an edge between $u_{i,\ell}$ and $u_{i+1,k}$ iff either $\ell = k$ or there is a (directed) edge in G from v_ℓ to v_k. Finally, take $s = u_{1,1}$ (i.e. x in the first copy of H) and $t = u_{n,2}$ (i.e. y in the last copy of H) and the distance $d = n$.

Correctness of the reduction: It can be seen that if there is a path from v_1 to v_2 in H, say
$$v_{j_1} = v_1, v_{j_2}, v_{j_3}, \ldots, v_{j_r - 1}, v_{j_r} = v_2$$

then there is a path of length exactly n between $u_{1,1}$ and $u_{n,2}$ in G:

$$u_{1,1}, u_{2,j_2}, u_{3,j_3}, \ldots, u_{r-1,j_r-1}, u_{r,2}, u_{r+1,2}, \ldots, u_{n,2}$$

Furthermore, since the edges go only from one copy to the next, and path between $u_{1,1}$ and $u_{n,2}$ must have length at least n. Thus the distance between s and t is n.

On the other hand, suppose that the distance between s and t in G is n. Then there is a path of length exactly n between $u_{1,1}$ and $u_{n,2}$. This path must be of the form

$$u_{1,j_1} = u_{1,1}, u_{2,j_2}, u_{3,j_3}, \ldots, u_{n-1,j_{n-1}}, u_{n,2} = u_{n,j_n}$$

By definition of G, for each i either $j_i = j_{i+1}$ or there is directed edge from v_{j_i} to $v_{j_{i+1}}$ in H. This shows that there is a path from v_1 to v_2 in H.

Complexity of the reduction: Given H the vertices and edges of G are easily computed in logspace. (In fact this is an $\mathbf{AC}^0$ reduction.)