Nondeterministic Circuits

by

Marty J. Wolf

Computer Sciences Technical Report #698
August 1987

Nondeterministic Circuits

Marty J. Wolf [1]

Computer Sciences Department
University of Wisconsin

Abstract:

Nondeterministic circuits are introduced and defined. By varying the number of nondeterministic gates in a circuit, different complexity classes are developed. Specifically, if NC circuits are allowed a polynomial number of nondeterministic gates, the class is equivalent to NP. If the number of nondeterministic gates is limited to O(log n), the class is equivalent to NC. A form of the quasigroup isomorphism problem is shown to be in the class obtained by allowing a polylog number of nondeterministic gates; this last problem is not known to be in either P or NC.

INTRODUCTION

Often in complexity theory different models of computation are used to define an abstract entity such as a complexity class, with each of them offering a slightly different view of the structure and make up of the entity. In the same way that partial recursive functions, random access machines, Turing machines, and Markov algorithms each characterize what is computable and offer insight into the structure and relationship between what is computable and what is not computable, a collection of different models for a complexity class can help us understand the structure of the class as well as the structure of the sets within the class.

With this in mind, we define a new collection complexity classes using nondeterministic circuits. Here a nondeterministic circuit is a Boolean circuit with ordinary gates, some nondeterministic gates, and one gate designated as the output gate. A *nondeterministic gate* takes no inputs and nondeterministically produces exactly one bit of output. Nondeterministic gates will also be called *guess gates*. The nondeterministic class NNC(f(n)) is defined to be the class of sets accepted by logspace *uniform* families of nondeterministic circuits with polylog(n) depth and a polynomial number of gates where at most O(f(n)) of those gates are nondeterministic gates, and n is the length of the input. (See Cook [Co] for a complete description of uniform circuit families.) In other words, a set in NNC(f(n)) is accepted by an NC circuit with O(f(n)) guess gates. Thus f(n) can be thought of as a bound on the amount of nondeterminism allowed or the maximum number of guess bits allowed in the computation on inputs of length n.

[1]Research supported in part by NSF Grant No. DCR85-04485

We will often abuse notation and write NNC(*class*) where *class* is a class of functions. For example, NNC(poly) is defined below.

To understand computation on an NNC circuit we can think of it proceeding in a manner similar to that of a nondeterministic Turing machine even though the following description is somewhat different than the actual model. The circuit makes a copy of itself for each of the possible outputs of the nondeterministic gates. In each copy, the nondeterministic gates are replaced by a particular value. Computation then proceeds as usual. At the end the output of each of the copies of the circuit are fed into a giant "or" gate, and the circuit accepts if and only if some combination of the choices made by the guess gates caused some copy of the circuit to accept.

The central problem becomes choosing f(n). We would like to find functions f such that NC $\subsetneq$ NNC(f) $\subsetneq$ NP. We begin by considering the class NNC(poly) = $\bigcup_{k\geq 1} NNC(n^k)$. We will show that this class is too powerful, since NNC(poly) = NP. This is not too surprising since Fortune and Wyllie [FW] using a PRAM model for NC show a similar result. Dymond [Dy] using a nondeterministic version of a hardware modification machine also shows NP and nondeterministic NC are equivalent. By using circuits as the model to study nondeterministic NC, we are given an intuitive means to quantify the amount of nondeterminism used in computation. Taking advantage of this feature, we see that if f(n) = log n, then NNC(f(n)) becomes too weak. It is not hard to show that NNC(log n) = NC. The case that appears to be most interesting is when f(n) is polylog in n. We are able to show a form of quasi-group isomorphism is in NNC(polylog), but it is not known to be in P. (Here NNC(polylog) = $\bigcup_{k\geq 1} NNC(log^k\ n)$.) In Section 1, we present a proof that NNC(poly) is another characterization of nondeterministic polynomial time. The proof is similar to Cook's proof that Satisfiability of Boolean Formulas is NP-complete (see [HU]). A brief proof that NNC(log n) = NC is also given. Section 2 gives results showing that quasigroup isomorphism is in NNC(polylog) when the quasigroups are input as multiplication tables. Section 3 presents some open problems.

SECTION 1. NNC(poly) = NP.

This section is concerned with the relationship between NP and NNC(poly). It is not hard to see that NNC(poly) $\subseteq$ NP.

Lemma 1.1: NNC(poly) $\subseteq$ NP.

Proof. Assume the set A is in NNC(poly). Then given x, an input of length n, and the NNC circuit to accept words in A of length n, it is easy to construct an NP-machine to correctly simulate the NNC circuit. First the NP

machine guesses the outputs of the guess gates, and then since there are only a polynomial number of gates in the circuit, the NP-machine can easily compute the output of each of the gates including the output gate in polynomial time. □

The next goal is to show NP $\subseteq$ NNC(poly), which implies NNC(poly) = NP. The idea of the proof is this. Let M be a nondeterministic one-tape Turing machine that accepts a set A in NP. On input x of length n, the NNC(poly) circuit "guesses" an instantaneous descriptor (ID) for each move of an accepting computation of M on input x. An ID is a string of bits representing the contents of the tape, the position of the tape head on the tape (written in binary), and the current state of machine M. Then for each pair of adjacent ID's, ID_{i-1} and ID_i, there is a small circuit that verifies that ID_i follows from ID_{i-1} by a legal move of machine M, based on the head position and the state of M. If every local checking circuit says everything is acceptable, the entire circuit accepts. The diagram on the following page illustrates the circuit.

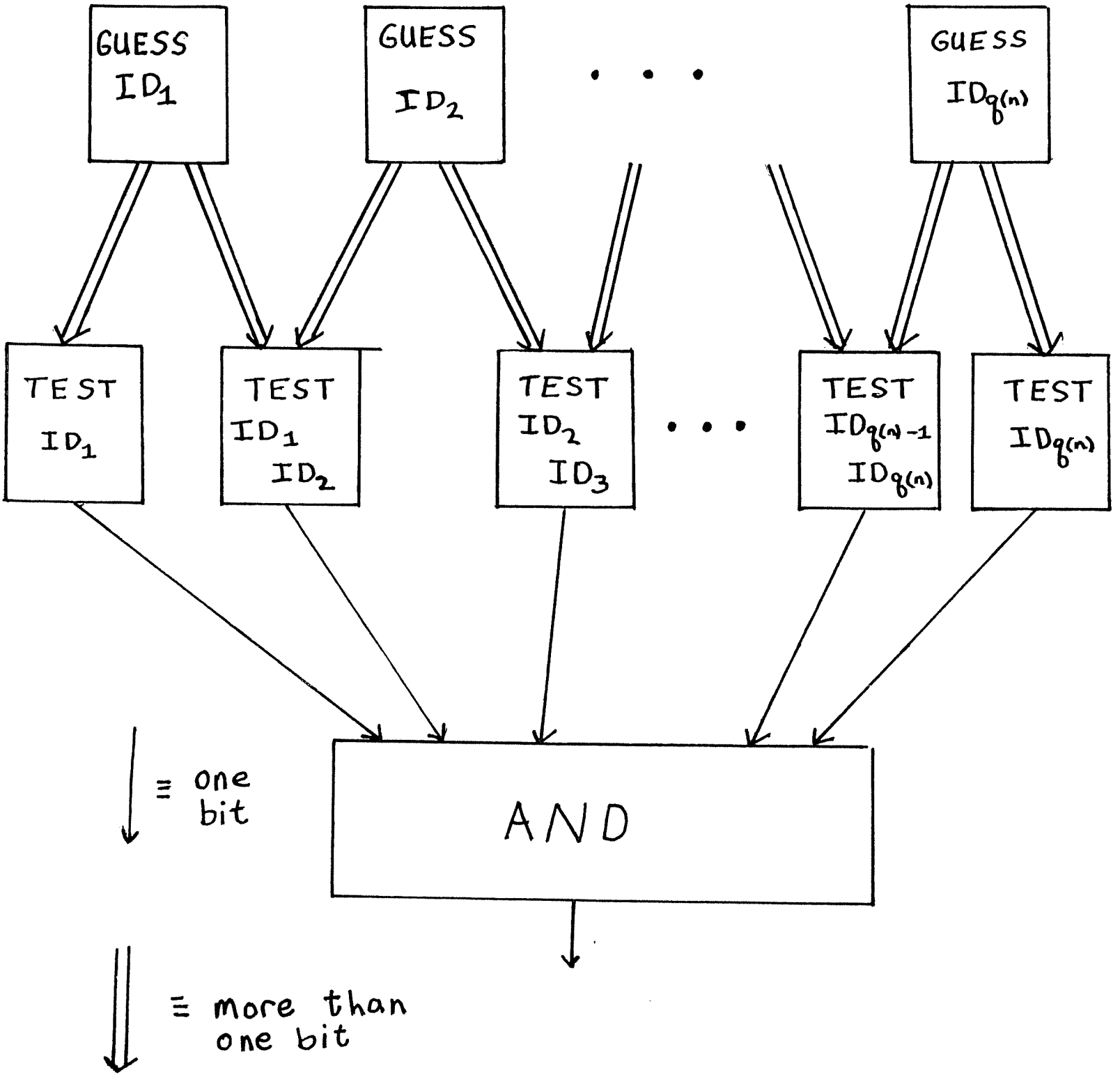
Theorem 1.2: NP = NNC(poly)

Proof. Lemma 1.1 shows one direction of the theorem. We now give more details of the proof that NP $\subseteq$ NNC(poly).

Let M be a nondeterministic polynomial time bounded Turing Machine accepting a language L in NP. Without loss of generality it can be assumed M has only one tape. Let q(n) be the polynomial that bounds the computation time of M on inputs of length n, and it can also be assumed that every computation on inputs of length n takes q(n) steps.

Now consider the computation of M on input x of length n to be a sequence of instantaneous descriptors of the form $\alpha\#\beta\#\gamma$, where α is a binary representation of the string currently on the work tape, β is the position of the read/write head on the tape written in binary, and γ is the state of the machine (also in binary). Thus the computation of M on input x can be viewed as a table of ID's, where the first ID is of the form $x\#0^{q(n)}\#<q_0>$, assuming q_0 is the start state of M. ($<q_0>$ is the binary encoding of q_0.) Also for the table, ID_i must follow from ID_{i-1} via a transition rule of M, and if $ID_{q(n)}$ is y#i#$<q_j>$, q_j is a final state of M if and only if x $\in L$.

The behavior of the circuit to simulate the computation of M on x is to first guess every ID of $M's$ computation. There are are only O(q(n)^2)) bits to guess since the length of each ID is O(q(n)), and the number of ID's is q(n). Then for each pair of ID's there is a small circuit for testing whether ID_i follows from ID_{i-1} via a legal move



NNC(poly) circuit for M

of M. Most of the work is done by the *test* subcircuits. If the first ID is correct, each tester is satisfied, and the state of the last ID is a final state the circuit accepts, otherwise it rejects. Since the final AND-circuit has q(n) inputs, it is easy to build an NC circuit to do this computation.

The *test* circuit is also an NC circuit. In parallel, the circuit verifies that all the bits have remained the same from ID_i to ID_{i+1} except those in the area of the read/write head. The bit under the read/write head is extracted by having a small circuit for each bit that tests whether the read/write head is at that position. This can easily be done in NC since the position of the bit can be wired into the small circuit. Each of these small circuits sends a bit to an OR-circuit; it sends its bit if the read/write head is at that position and a '0' otherwise. Verifying that the move that took place between ID_i and ID_{i+1} is a legal move of M can be done in constant depth with a table look up.

There are two last concerns. The first ID tester must verify that ID_1 represents the input and the last ID tester must verify that the state of $ID_{q(n)}$ is a final state. The first of these tasks can be done easily in NC, and since the number of the states is a constant with respect to M, the second can be tested with a constant number of gates. The detailed construction of these circuits is left to the reader.

Thus for each set in NP there is a NNC(poly) family of uniform circuits that accepts it. □

Theorem 1.3: NNC(log n) = NC

Sketch of proof. The proof of this theorem is not difficult. It is obvious that NC $\subseteq$ NNC(log n). To show the other direction is not much more difficult. Since there are only $2^{O(\log n)}$ (which is at most a polynomial in n) possible different guesses that can be made by the guessing gates, enumerating all of them would not greatly increase the size or the the depth of the circuit. The details of this proof are left to the reader. □

SECTION 2. Quasigroup isomorphism is in NNC(polylog).

A third attempt at finding an interesting nondeterministic analog for NC can be made by allowing a polylog number of guess gates. This class seems nontrivial and potentially of considerable interest. Since Miller [Mi] has shown the quasigroup isomorphism problem has an $O(n^{\log n})$ sequential algorithm and since $n^{\log n} = 2^{\log^2 n}$, this problem is a natural candidate for being in NNC(polylog).

Quasigroups here are thought of as Cayley tables. That is, if G is a quasigroup of order n, we view it as a binary function on the integers $\{1,...,n\}$, which is specified by a multiplication table. For review, definitions of

group and quasigroup are given.

Definition: A *Group* is a binary operation * on a set G satisfying the following properties:

1. There is a unique x such that $a*b$ = x.
2. There is a unique x such that $a*x$ = b.
3. There is a unique x such that $x*a$ = b.
4. If $a, b, c \in G$, then (a*b)*c = a*(b*c).

A quasigroup is more general than a group since a quasigroup is a binary relation satisfying 1, 2, and 3. A Latin square is the multiplication table of a quasigroup. Miller [Mi] shows that quasigroups of order n are generated by at most log n elements. We take advantage of this fact to develop the circuit for quasigroup isomorphism testing. Since quasigroups are more general than groups, the following construction also shows Cayley table group isomorphism is in NNC(polylog).

Theorem 2.1: Given two multiplication tables M_1 and M_2, representing the quasigroups H_1 and H_2, respectively, the set $\{(H_1,H_2) : H_1$ and H_2 are isomorphic$\}$ is in NNC($\log^2 n$), where n is the order of the two quasigroups.

Proof. The circuit to test isomorphism begins by guessing two sets of generators G_{H_1} for H_1 and G_{H_2} for H_2 in parallel. Then, in parallel it verifies that G_{H_1} generates H_1, G_{H_2} generates H_2, and that G_1 and G_2 are isomorphic. The circuits that perform these tasks are similar.

Consider the following family of uniform NNC circuits. There are $2\log^2 n$ guessing gates. Since each generator is log n bits long and there are log n generators, half of the guessing gates are used to guess G_{H_1}, the set of generators for H_1, and the rest are used to guess G_{H_2}, the set of generators for H_2.

To verify that G_{H_1} does generate H_1, we use the following subcircuit. Imagine 2log n copies of the multiplication table of H_1 stacked on top of each other with each copy having n sentinels, one for each element of H_1. Each sentinel heads its respective row and column. In the first copy of the multiplication table the i^{th} sentinel determines whether one of the guess gates chose h_i. If an element has been chosen, the sentinel for the element tells each element in its row and column that the sentinel is in the quasigroup. Each element in the interior of the table waits for information from its row sentinel and its column sentinel. If both are in the quasigroup, then the element knows it is in the quasigroup. In the mean time, each sentinel passes the information it has onto the corresponding sentinel in the next copy of the multiplication table. Each element in the interior that now knows it is in the quasigroup informs the sentinels at the next copy of the multiplication table. The sentinels that currently do not know if they are in the quasigroup check this new information. The sentinels that were in the quasigroup at the previous level as well as

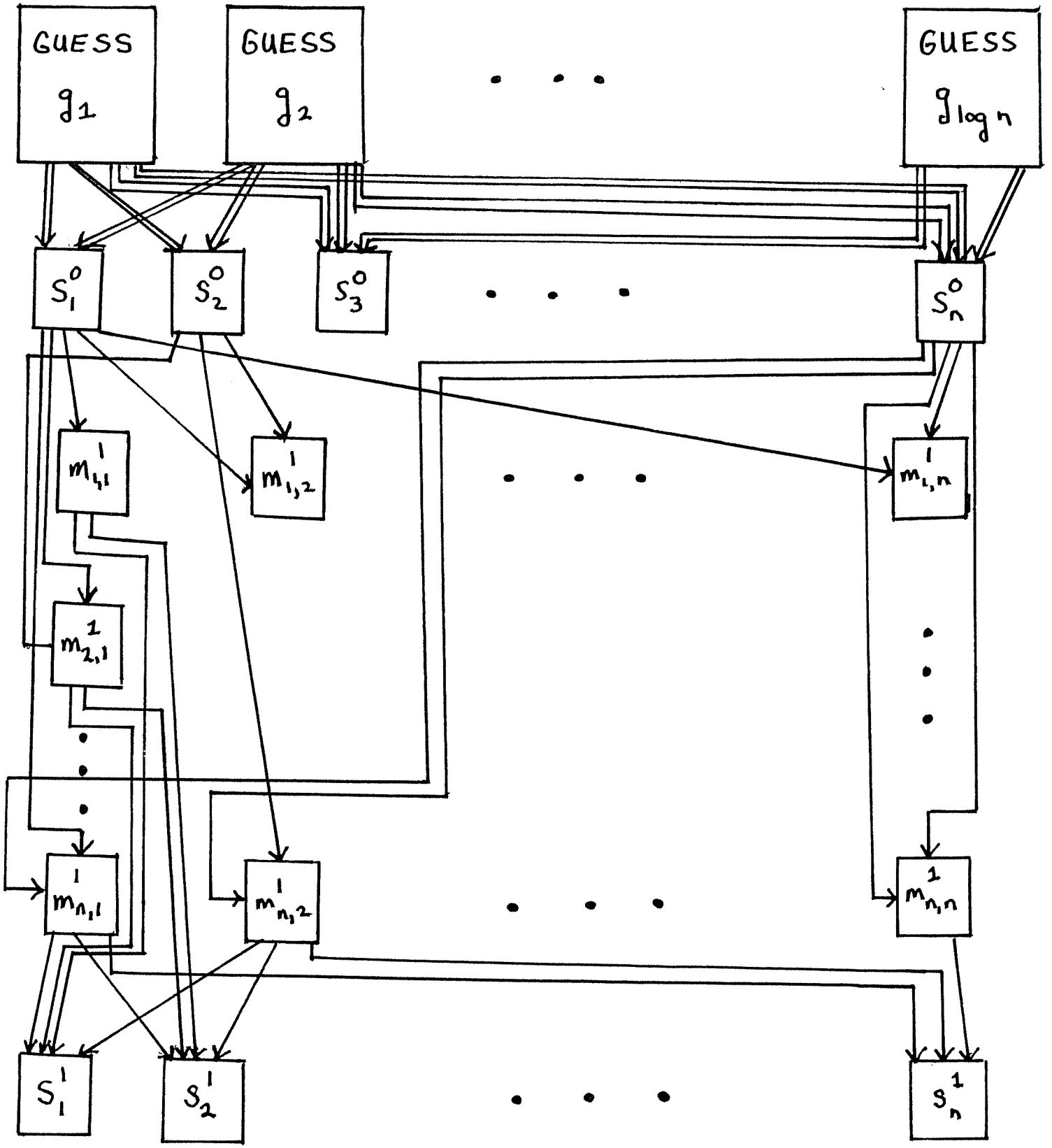
the ones added at the current level pass their information along to the interior of the table, and the process repeats until the bottom of the stack of tables is reached. After the information is passed to the sentinels for the last time, each sentinel that has been generated sends a 1 to a final AND-circuit. If the output of the final AND is 1, then G_{H_1} generates H_1. An identical circuit verifies G_{H_2} generates H_2.

The first level of the circuit to test if G_{H_1} generates H_1 looks like the figure on the next page, where s_i^k is the subcircuit for the i^{th} sentinel on the k^{th} level multiplication table, $m_{i,j}^1$ is the subcircuit needed look up the i,j element of the first level multiplication table, although each s_i^k and $m_{i,j}^1$ also requires wires from the input. The same general construction is repeated $2\log n$ times, with the outputs from each $s_i^{2\log n}$ being fed into a final AND-circuit. The construction of each s_i^k and $m_{i,j}^k$ and the rest of the levels is left to the reader.

The subcircuit to verify H_1 and H_2 are isomorphic is similar in structure to the one described above. When G_{H_1} and G_{H_2} are guessed, a mapping between the generators is also guessed, i.e. the generators are guessed in pairs, with the first one belonging to G_{H_1} and the second belonging to G_{H_2}. Again, there are $2\log n$ levels of multiplication tables. Each level takes input from the previous level as above, except this time the input comes as pairs of elements, the first element is from H_1 and the second from H_2. Again there is a group of n sentinels, one for each member of H_1. Each sentinel looks to see if its name is the first element of one of the pairs. If it is, it remembers its image in H_2. If it already knows its image in H_2, it verifies that the two images are the same. Now the sentinel tells each element in the row and column it heads what the sentinel's image is. If an element in the interior of the table hears from both its row sentinel and its column sentinel, it determines the product of the two images it received by looking it up in the multiplication table of H_2. It then passes its identity and its image on to the group of sentinels waiting at the next level. After the the last level the sentinels verify that each has exactly one image, and that once a sentinel has received an image all subsequent images received by that sentinel are identical.

In the more detailed description of the isomorphism tester described below, we see the general structure and relationship of the multipliers and the sentinels is the same as before, although the task of testing whether the mapping is an isomorphism is somewhat more complex than testing whether the generators did generate the quasigroup. For the isomorphism testing phase the sentinels will be called t_i^j and the multipliers will be called $iso_{i,j}^k$.

The general sentinel is more complex than the sentinel at the zeroth level, so only the general sentinel will be described. Each sentinel circuit t_i^j takes as input h_i, the image of h_i (if it is known), the product generated by each $iso_{k,l}^j$, and the images associated with each of those products. The sentinel tests to see if h_i is equal to any of the n^2



First Level of Circuit to Test Whether G_{H_1} generates H_1

products. Since one log n bit number is tested against n^2 log n bit numbers, this can be done in parallel with a depth of O(loglog n). If h_i equals one of the products, the image of h_i and the image of the product are tested, if they are equal the sentinel outputs the image of h_i to all multipliers on the next level $iso_{i,k}^{j+1}$ and $iso_{k,i}^{j+1}$ for all k and outputs a 1 to a special verifier subcircuit, otherwise it sends a zero to the verifier subcircuit. This can be implemented in O(log n) depth. The sentinel also sends a bit along indicating whether it knows the image of h_i. The special verifier subcircuit takes as input a single bit from each of the $n(1 + 2\log n)$ sentinels (n from each level and there are $1 + 2\log n$ levels) and AND's them all together. If one of them is a zero, this shows the mapping was not well defined.

Multiplier $iso_{i,j}^{k}$ gets input from sentinels t_i^{k-1} and t_j^{k-1}. From each it receives a bit indicating if the sentinel knows its image as well as the image of the element the sentinel represents (if it is known). From the input $iso_{i,j}^{k}$ knows which element in the multiplication table for H_2 it represents, and it also receives the entire multiplication table for H_2. If one of the sentinels does not know its image, $iso_{i,j}^{k}$ does nothing, otherwise it finds the product of the two images it received in the multiplication table for H_2. The table can be fed to $iso_{i,j}^{k}$ as ordered triples $<h_r, h_s, h_r * h_s>$, then using an equality tester, the product $h_r * h_s$ can be extracted from the table in depth O(log n). Next $iso_{i,j}^{k}$ passes the product and the image of the product on to the sentinels on level k. This requires depth O(log n) to make the n copies of the product and its image needed at the next level. Again the construction of the multipliers and sentinels is left to the reader.

If it is verified that the generators guessed do generate H_1 and H_2, we are guaranteed in the isomorphism testing phase that image of every element of H_1 will be tested, and the special verifier subcircuit is used to insure the mapping is well defined. We are left to show that if the guessed set of generators does generate the quasigroup, then the first subcircuit will verify this. Note that every word in a quasigroup can be represented as a binary tree where the internal nodes represent multiplications and the leaves are labeled with generators. The next lemma shows every word in a quasigroup can be represented by a tree of small depth.

Lemma 2.2: If H is a quasigroup of order n and $g_1, g_2, ..., g_k$ generate H, then for every $h \in H$ there is a tree of depth $\leq 2\log n$ that represents it.

Proof. Let $depth_k = \{h \in H \mid$ the shallowest tree representing h has depth $\geq k\}$. Note that $depth_0 = H$. Let $Q^{(k)} = \{(i,j) \in H \times H \mid i \text{ or } j \in depth_{k-1}\}$. Since H is finite, there is a d such that $depth_{d+i}$ is empty for all $i>0$. Let $P^{(d)} = H \times H$, and let $P^{(k)} = P^{(k+1)} - Q^{(k+1)}$ for $k<d$. Thus a pair (i,j) is in $P^{(k)}$ if and only if both i and j have

shallowest tree depth less than k. A pair (i,j) is said to *represent* an element h if $i*j=h$. We show by reverse induction that $|P^{(d-1)}| \leq \frac{n^2}{4}$, $|P^{(d-3)}| \leq \frac{|P^{(d-1)}|}{4}$, $|P^{(d-5)}| \leq \frac{|P^{(d-3)}|}{4}$, ..., and that the sets $depth_{d-1}$, $depth_{d-2}$, $depth_{d-3}$, ..., $depth_1$ are nonempty.

The basis is $k = d$. Let $h \in depth_d$. Let $D^{(d)} = \{(i,j) \mid i*j = h\}$. For each $1 \leq i \leq n$ there is exactly one pair (i,j) that represents h, thus $D^{(d)}$ has exactly n members. Since at least one element of every pair in $D^{(d)}$ must be in $depth_{d-1}$, there are at least $\frac{n}{2}$ elements in $depth_{d-1}$. Each element in $depth_{d-1}$ appears as the first element of n pairs, and as the second element in n pairs and there are $\frac{n}{2}*\frac{n}{2}$ pairs with both elements from $depth_{d-1}$, thus there are at least $\frac{3n^2}{4}$ pairs in $Q^{(d)}$, and no more than $\frac{n^2}{4}$ pairs in $P^{(d-1)}$.

Now there is at least one element, k, in $depth_{d-1}$ and not in $depth_d$ such that there is a pair in $P^{(d-1)}$ that represents it. If such a k does not exist then the elements of $depth_{d-1}$ could not be generated since all pairs (g_i, g_j) where g_i and g_j are generators are in $P^{(d-1)}$

For the inductive step, assume $k \geq 2$, let $k \in depth_l$ and $k \notin depth_{l+1}$ and $P^{(l)}$ be given inductively. Let m be the number of elements in H that are not in $depth_l$. Thus there are m^2 in $P^{(l)}$. We will show there exists a $k' \in depth_{l-2}$ such that k' is represented by a pair (i,j) not in $P^{(l-2)}$ which implies either i or j is in $depth_{l-3}$ and not in $depth_{l-2}$.

For the following sublemmas, recall that $H = \{1,2,3, ..., n\}$, and since we can assume that the elements of H are sorted by depth, $depth_l = \{m+1, m+2, ..., n\}$.

Lemma 2.2.1: For every $i \leq m$, there is a pair $(i,j) \in P^{(l)}$ such that $i*j \in depth_{l-1}$.

Proof. Since $k \in depth_l$ and $k \notin depth_{l+1}$, and since there is a pair $(k',k'') \in P^{(l)}$ such that $k'*k''=k$, either k' or k'' is in $depth_{l-1}$ and not in $depth_l$. Without loss of generality assume it is k'. Assume the lemma does not hold. Let i' be such that for all pairs $(i',j) \in P^{(l)}$, $i'*j \notin depth_{l-1}$. Now each element of $depth_l$ as well as k' must be represented by a pair of the form (i',j). Since there are only $n - m$ pairs of this type that are not in $P^{(l)}$, at least one pair of the form (i',j) is in $P^{(l)}$. This is a contradiction. □

Lemma 2.2.2: For every $j \leq m$, there is a pair $(i,j) \in P^{(l)}$ such that $i*j \in depth_{l-1}$.

Proof. The proof is the same as Lemma 2.2.1. □

Lemmas 2.2.1 and 2.2.2 imply at least one element of every pair in $D^{(l)} = \{(i,j) \mid i*j \in depth_{l-1}\}$ is in $depth_{l-2}$. Then using arguments similar to those used in the base case we find $depth_{l-2}$ has at least $\frac{m}{2}$ elements that are not in $depth_l$, and $Q^{(l-2)}$ has at least $\frac{3m^2}{4}$ pairs, and there are no more than $\frac{m^2}{4}$ pairs in $P^{(l-2)}$. Therefore, as before, there must exist a $k' \in depth_{l-2}$ such that $k' \notin depth_{l-1}$ and there is a pair in $P^{(l-3)}$ that represents it.

Considering the sequence of sets $depth_{d-1}$, $depth_{d-3}$, ..., $depth_0$, we find that if $d \geq 2\log n$, $depth_{d+1}$ is empty. □

Lemma 2.3: If H is a quasigroup of order n and $g_1, g_2, \ldots, g_k$ generate H, then the circuit defined above will generate every possible tree of depth less than or equal to m in these generators after the m^{th} level of the circuit.

Proof. The proof is by induction on m. The base case is when $m = 0$, namely after the generators have been guessed. Certainly all the trees of depth one have been generated.

Now assume that after the $(m-1)^{st}$ level all the trees of depth less than or equal to $m-1$ have been generated. Let w be a tree of depth less than or equal to m. Certainly w can be composed of two trees of depth less than or equal to $m-1$, and the elements represented by these trees, by induction, will be available at level m - 1. Hence w will appear at level m. □

Combining Lemma 2.2 and Lemma 2.3, with $k = \log n$ and $m = 2\log n$, we are guaranteed that if the guessed set of generators does generate the quasigroup, then the first subcircuit will verify it. Thus quasigroup isomorphism is in NNC(polylog). □

SECTION 3. Conclusion and open questions.

In the development of traditional complexity classes, time and space are the resources used quantitatively to define specific classes. By looking at various values for f(n), NNC(f(n)) offers another resource, nondeterminism, to use quantitatively to study the relationship between problems in NP. The results above suggest that the class NNC(polylog) may offer a hierarchy to classify the problems in NP that are not known to be in P and are not known to be NP complete. The hierarchy can be subdivided on either the degree of the log term, the depth of the NC circuit or on a combination of the two. For example, closer inspection of the circuit used in Theorem 2.1 shows it is an

NC^3 circuit with $O(\log^2 n)$ guess gates, thus quasigroup isomorphism is in $NNC^3(log^2\, n)$. Since quasigroup isomorphism has a subexponential algorithm, it seems reasonable that other problems with this property may be contained in $NNC^k(log^j\, n)$ for some j and k.

The following is a list of open questions that are related to these results and to the idea of nondeterministic circuits in general.

(1) Are there other natural problems in NNC(polylog)? Where in the hierarchy do they sit?

(2) What is the relationship between P and NNC(polylog)? Is P contained in NNC(polylog)?

(3) What is the relationship between RNC and NNC(polylog)? Can the nondeterminism used in the NNC circuit be used to simulate the randomness of the RNC circuit?

Acknowledgements:

I extend a special thanks to Eric Bach for suggesting the study of this area to me and for all the insightful discussions and guidance.

REFERENCES

[Co] Cook, S.A., "The Classification of Problems which have Fast Parallel Algorithms," *Lecture Notes in Computer Science*, V. 158, Springer-Verlag, New York, (1983), 78-93.

[Dy] Dymond, P.W., "On Nondeterminism in Parallel Computation," *Theoretical Computer Science 47*, (1986), 111-120.

[FW] Fortune, S. and Wyllie, J., "Parallelism in Random Access Machines," *Proceedings of the Tenth ACM Symposium on the Theory of Computing* (1978), 114-118.

[HU] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass. (1979).

[Mi] Miller, G.L., "On the $n^{\log n}$ Isomorphism Technique," *Proceedings of the Tenth ACM Symposium on the Theory of Computing* (1978), 51-58.

[Ru] Ruzzo, W.L., "On uniform circuit complexity," *Journal of Computer and System Sciences 22*, 3(June 1981), 365-383.