# COMP 531 – Advanced Theory of Computation
# Assignment #2

Guillaume Labranche (260585371)

due on 15 February 2016

## 1 Logspace closed under composition

Here we assume that by log-space computable functions, we mean functions which can be computed by a log-space transducer, that is a TM with a read-only input tape, a log-space bounded work tape and a write-only output tape.

**Theorem 1.** *If both $f$ and $g$ are computable in logarithmic space, $f \circ g$ is also computable in logarithmic space.*

*Proof.* In the trivial solution, for any input $x$ to $f \circ g$, we first compute $g(x)$ and then run $f$ on $g(x)$'s output tape. Let $n = |x|$. While $g(x)$ requires logarithmic work space, its output may be polynomial (by lemma 1). Let $n' = |g(x)| = O(n^k)$. Then $f(g(x))$ requires $O(\log(n^k)) = O(k \log n) = O(\log n)$ work space. Although individually $f$ and $g$ only require logarithmic space, storing $g(x)$ requires polynomial space. Therefore we devise a trick that enables us to compute $f(g(x))$ without storing $g(x)$.

We modify the transducer $F$ for computing $f$ as follows. Whenever $F$ reads the $i$th symbol of its input tape (which contains $g(x)$), it runs the transducer $G$ for $g$ on $x$ until the $i$th symbol is outputted (transducers have a write-only output tape). This can be done using two counters of size $O(\log(n^k)) = O(\log n)$ each, one for storing the position on $F$'s input tape and one for keeping track of the number of symbols outputted by $G$. As we can see, $f \circ g$ is still computable in logarithmic space because we do not store $g(x)$, only one character at a time and two counters of logarithmic size. □

**Lemma 1.** *The output of a log-space computable function $f$ is polynomial in the size of its input.*

*Proof.* Assuming that the log-space transducer outputs 1 symbol at every step of its runtime (without loss of generality), by first obtaining a bound on its runtime we can then bound the size of its output. The number of possible configurations is $|Q| \cdot (c \cdot \log_2 n) \cdot 2^{c \cdot \log_2 n} = O(2^{c \cdot \log_2 n}) = O(n^c)$. Since we know that $f$ is computable, the log-space transducer will terminate in finite time. The maximum number of configurations it can be in without looping forever is polynomial, and therefore so is the maximum number of symbols it can write to its output tape. □

## 2 Cycle searching principle

The key intuition here is to realize that the son's future **and** past behaviour is entirely determined by the current edge (and direction) that he is following. With this insight, we can show that he cannot enter a cycle and get stuck in it unless he was spawned in it by his father. This is then used to show that the son always returns to his initial spawn vertex in finite time. Termination follows from that naturally while correctness requires some more detailed case analysis.

**Lemma 2.** *Let $(v, w) \in E$, $(u_1, v, w)$ and $(u_2, v, w)$ be two paths followed by the son at some point during the execution of the algorithm. We claim that $u_1 = u_2$.*

*Proof.* Let $(v, w)$ be $v$'s $i$th edge. Therefore by the algorithm behaviour, $(u_1, v)$ is $v$'s $(i-1)$th edge. But $(u_2, v)$ is also $v$'s $(i-1)$th edge. Therefore it must be that $u_1 = u_2$. □

**Definition.** We define a **son-loop** as a sequence of vertices $(u, v, \ldots, u, v)$ such that when the son goes from $u$ to $v$, he will always come back to $u$ then $v$ and would loop forever unless his father is somewhere in the loop to stop him.

**Lemma 3.** *If the son is outside of any son-loop, he cannot enter one.*

*Proof.* Consider a son-loop $L = (u, v, \ldots, w, u)$. Assume the son enters the cycle through vertex $u$ and then follows $(u, v)$ without loss of generality. By lemma 2, the son was on vertex $w$ before entering the son-loop, which leads to a contradiction since $w \in L$. Either he was in the sun-loop all along, or he did not enter it. □

**Lemma 4.** *After being brought to vertex $u$ by his father, the son always returns to $u$ in finite time.*

*Proof.* (by contradiction).

Assume the son never returns to $u$. By the pigeonhole principle, since there are only $2|E|$ possible ways to follow any edge ($|E|$ edges and 2 directions), the son must be following the same edge $v$ in the same direction at least twice. By the deterministic nature of the algorithm, he will then follow the exact same path when leaving $v$ and end up in a sun-loop. In the case that he was spawned in that sun-loop (on vertex $u$), since any sun-loop must have finite length, the son will return to $u$, which contradicts our assumption. It must then be the case that the son was spawned outside of the sun-loop and later entered it. But lemma 3 shows that this is not possible. Therefore our assumption cannot be true and the son will return to $u$. □

**Theorem 2.** *This algorithm terminates in finite time.*

*Proof.* $\forall v \in V, \forall (v, u) \in E$, the son follows $(v, u)$ and by lemma 4 comes back to $v$ in finite time. Since $|V|$ and $|E|$ are both finite, the algorithm terminates in finite time.

□

**Theorem 3.** *This algorithm detects a cycle $\iff$ there is a cycle.*

*Proof.* ( $\implies$ ) Consider the only way that the algorithm detects a cycle: When the son is placed at a vertex $v$, leaves through an edge $(v, u)$ and comes back to $v$ through a different edge $(w, v)$ where $u \neq w$. In this case, there is a path from $v$ to $u$, from $u$ to $w$ and from $w$ to $v$. This forms a cycle.

($\Longleftarrow$) Suppose there is a cycle $C$ containing the edge $(u, v)$. Then at some point the father will bring the son to $u$ and send it along the edge of $v$. By lemma 4, the son will always return to $u$. Now consider two cases:

- If the son does not come back through $(u, v)$, the cycle is detected.
- If the son does come back through $(u, v)$, then there must be a vertex $w \in C$ such that the son did not enter the successor of $w$ in $C$ (in other words he must have left $C$ at some vertex $w$). Since $w$ is connected to its successor, its predecessor and some other vertices, its fanout is at least 3. Assume that the predecessor and successors of $w$ are accessible through the $i$th and $j$th edges of $w$ respectively. Then the son must leave $C$ at $w$ through the $k$th edge of $w$ where $i < k < j$ otherwise he would have reached the successor of $w$ first and by definition it does not happen. Now consider 3 subcases:
  - If the son leaves the cycle from $w$ and comes back to $w$ through a different edge, then when the father brings him to $w$ he will detect a cycle.
  - If the son comes back to $w$ through the same edge, then he will eventually reach the vertex following $w$ in $C$. But that contradicts our definition of $w$ which is supposedly the furthest vertex reached in $C$. Therefore this case is not possible.
  - If the son does not come back to $w$, then he must eventually re-enter $C$ through one of the vertices he already visited. Let's call that vertex $z$. When later in the algorithm the father brings the son to $z$ and sends him to the successor of $z$ in $C$ called $z'$, due to the deterministic nature of the algorithm, the son will follow the same path as just described and not return to $z$ through the same edge (since we defined $z$ as the vertex where the son re-enters $C$). At that time, the father will notice the different return edge and detect a cycle.

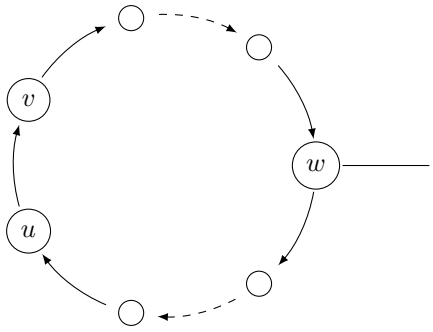



Figure 1: Diagram illustrating a cycle $C$ in $G$

□

The algorithm only needs to store a constant amount of vertex indices:

- which vertex the father is visiting
- which vertex the son visits initially
- which vertex the son last visited while traversing the graph

Therefore the space complexity is $O(\log |V|)$. Also worth noting is that this algorithm works for both connected and disconnected graphs.

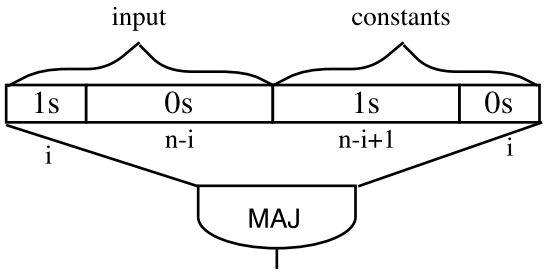
# 3 Boolean symmetric functions $\rightarrow$ MAJ $\circ$ MAJ

First we note that the description of a symmetric boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is basically a mapping $m : \{0,1,\ldots,n\} \rightarrow \{0,1\}$.

Next, we introduce a new type of gate $\text{MAJ}_{\geq i}$ constructed from a single MAJ gate that returns 1 when $i$ or more of its $n$ input wires are 1, and 0 otherwise. To construct it, we feed all $n$ inputs into a MAJ gate. We also add $i$ 0-constant inputs and $n-i+1$ 1-constant inputs for a total of $2n+1$ inputs. Therefore the MAJ gate will output 1 when:

$$\begin{aligned}
\#_1 &> \frac{2n+1}{2} \\
\#_1 &> n + 1/2 \\
\#_1 &\geq n+1 \\
\#_1 &\geq i + (n-i+1 \text{ 1-constants})
\end{aligned}$$

Therefore when ignoring the constants, our $\text{MAJ}_{\geq i}$ will return 1 when $i$ or more of its $n$ inputs are 1. The following diagram illustrates the strategy visually:




We also introduce a similar gate $\text{MAJ}_{\leq i}$ that returns 1 if $i$ or less inputs are 1 and 0 otherwise. Its construction is very similar to $\text{MAJ}_{\geq i}$ but with negated inputs. Checking whether there are at most $i$ 1s is equivalent to checking whether there are more than $n-i$ 0s. Thus we negate all $n$ inputs and feed them into a MAJ gate. We then add $n-i$ 0-constant inputs and $i+1$ 1-constant inputs so that the MAJ gate has fanin $2n+1$. Therefore the MAJ gate will output 1 when:

$$\begin{aligned}
\#_1 &> \frac{2n+1}{2} \\
\#_1 &> n + 1/2 \\
\#_1 &> n \\
\#_0 &\leq n \\
\#_0 &\leq i + (n-i \text{ 0-constants})
\end{aligned}$$

Since we negated the $n$ inputs, this is equivalent to checking that

$$\#_1(\text{original input}) \leq i$$

We then introduce a new type of gate $\text{EQ}_i : \{0,1\}^n \rightarrow \{0,1\} \times \{0,1\}$ composed of a $\text{MAJ}_{\geq i}$ and a $\text{MAJ}_{\leq i}$. On input $w$, it will output $(1,1)$ if $\#_1(w) = i$, and $(1,0)$ (or $(0,1)$) if $\#_1(w) \neq i$.

Now to construct a circuit to compute $f$, for all $i$ such that $f(i) = 1$, add an $\text{EQ}_i$ gate with the same inputs as $f$ at level 1. At level 2, add a single MAJ gate taking all our level 1 gates as input.

In order to show that this indeed computes $f$, let's perform a case analysis on $f(j)$ for all $0 \leq j \leq n$:

- $f(j) = 1$: This will make all the $\text{EQ}_{i \neq j}$ return $(1, 0)$ or $(0, 1)$ except the gate $\text{EQ}_{i=j}$ which will return $(1, 1)$. There will be more 1s than 0s feeding into the level 2 MAJ gate and the circuit will output 1.
- $f(j) = 0$: This will result in all EQ gates returning $(1, 0)$ or $(0, 1)$ since no $\text{EQ}_{i=j}$ gate has been added to the circuit. Therefore the number of 0s and 1s will be equal and the level 2 MAJ gate will output 0.

Note that this also creates a circuit of linear size, since $f(i) = 1$ for at most $n$ values and that in each of those cases we add 2 MAJ gates. This completes the construction of a linear-size MAJ $\circ$ MAJ circuit computing a symmetric boolean function.

# 4 Polysize circuits $\rightarrow$ $\mathbf{NC^1}$

In order to show that the boolean function induced from a polysize tree-shaped circuit is actually in $NC^1$, we describe a transformation of such circuits to equivalent polysize circuits of logarithmic depth.

Let $F(x_1, \ldots, x_n)$ be the boolean formula induced from a circuit $C_n \in C$ with root node $u$ and size $s = O(n^k)$. Starting a $u$, find a subtree of size $t$ where $\frac{1}{3}s \leq t \leq \frac{2}{3}s$ using the strategy described in lemma 5 and call its root $v$. Let $F'(x_1, \ldots, x_n)$ be the boolean formula represented by this subcircuit. See figure 2 for an illustration.

Now take the subtree rooted at $v$ out of our main circuit and replace it with a variable called $z$. We now have two trees rooted at $u$ and $v$, each of size at most $\frac{2}{3}s$. Let $\hat{F}(x_1, \ldots, x_n, z)$ be the formula represented by the circuit rooted at $u$ after removing the $u$ subtree.

We now construct a new circuit equivalent to our original as shown in figure 3. In order for our new circuit to return 1, we join two cases by an $\vee$ gate.

- In the first case we assume the circuit rooted at $v$ returns 1. In order to not modify the output of the circuit, we join the following two conditions by an $\wedge$ gate:
  - Our original circuit must return 1 when $z = 1$.
  - The subcircuit at $v$ must also return 1 since we set $z = 1$.
- The second case is when the subcircuit returns 0. We join the following two conditions by an $\wedge$ gate:
  - Our original circuit must return 1 when $z = 0$.
  - The subcircuit at $v$ must return 0 since we set $z = 0$. In order to do that, we first negate the boolean formula: $\neg F'(x_1, \ldots, x_n)$. We then use De Morgan's laws to push the negation all the way down to the inputs and negate the ones that are negated in the formula. We therefore do not need to add extra gates.

The result of this transformation is shown in figure 3. Once this is done, we recursively apply this transformation on all the subtrees rooted at level 3 nodes of our new circuit. The recursion stops once we reach subtrees of size 1.

In order to show that this indeed creates a tree of depth $O(\log n)$, realize that one recursion step only adds a constant (2) depth to the newly created tree so far. Also note that the recursion is done on subtrees that are at most $2/3$ the size of the original tree. Therefore in the worst case, we have $\log_{3/2}(s) = O(\log n^k) = O(k \log(n))$. Therefore the depth of our final tree has depth logarithmic in the number of inputs.

In order for the transformed circuit to be in $NC^1$ we must also show that its size is polynomial. At every step of the recursion, we add a constant amount of gates and we also duplicate the subtrees without altering their number of gates. We therefore double the amount of gates at each step. Since there are $O(\log n)$ steps, we multiply the number of gates by a factor of $2^{\log n} = n$. Therefore this only increases the degree of the polynomial by 1 which is still a polynomial amount of gates in terms of the number of inputs $n$.

**Lemma 5.** *Given a tree of size $s$ with root node $u$, there is a node $v$ whose subtree is of size $t$ where $\frac{1}{3}s \leq t \leq \frac{2}{3}s$*

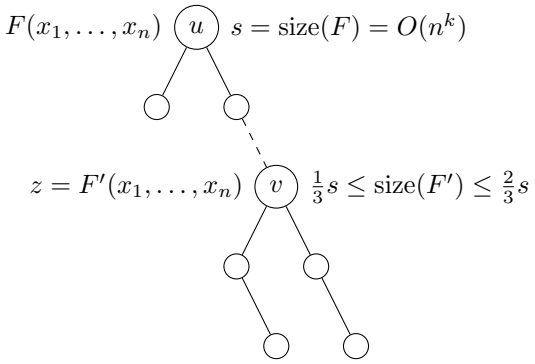



Figure 2: $C_n$ before the transformation

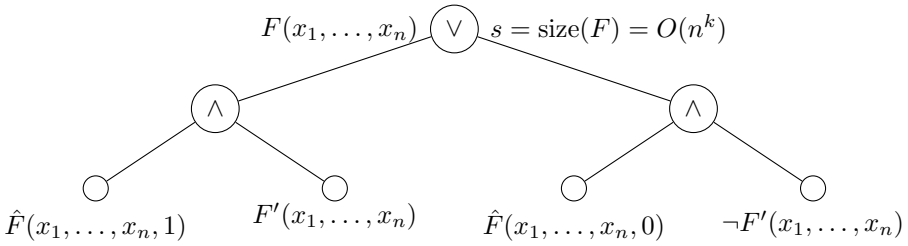



Figure 3: $C_n$ after one recursion step

*Proof.* If either the immediate left or right child of $u$ is a subtree size $t$, then we have found $v$. Otherwise, recurse on the child whose subtree is bigger. Because each recursion step reduces the size of the tree by at least one, we are guaranteed to find such $v$. □

# 5 $w$-parity

Left blank due to lack of time.