CS273 Foundations of Parallel and Network Computing **Spring 2003**

Lecture 12: March 5, 2003

Lecturer: Christos Papadimitriou *Scribe: Brighten Godfrey*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

12.1 Motivation and definitions

We've been discussing techniques to optimally parallelize problems: solve them in time $O(\log n)$ with an unlimited number of processors, while doing no more total work than the best sequential algorithm. For some very simple problems, we were able to accomplish this, using different techniques than the classic single-processor algorithms.

But for other problems, we could get only poly-logarithmic time: $O(\log^c n)$ for some constant c. To capture the notion that this is "good enough," we introduce a parallel version of the well-known and much-beloved complexity class **P**.

Definition 12.1 *A language L is in **NC** ("Nick's Class") iff there exists an algorithm that solves L using $O(n^c)$ processors and $O(\log^c n)$ time, for some constant c.*

We'd like **NC** to be to **P** as **P** is to **NP**. Thus, we also have a notion of **P**-completeness. **P**-complete problems are the most inherently sequential problems in **P**: if we could parallelize any one of them, then we would be able to parallelize all problems in **P**.

Definition 12.2 *A language L is* **P**-hard *iff for all $L' \in \mathbf{P}$, L' reduces to L through some **NC** algorithm. L is* **P**-complete *iff $L \in \mathbf{P}$ and L is **P**-hard.*

Some **P**-complete problems include the Monotone Circuit Value Problem (MCVP), most yes/no questions regarding depth-first search, Greedy Independent Set, and Max Flow/Linear Programming. In the following sections, we look at several of these in detail.

12.2 The Monotone Circuit Value Problem

The Monotone Circuit Value Problem is defined as follows. We are given a circuit composed of gates $g_1, \ldots g_n$, where each gate g_i is specified as

- $g_i =$ **true**, or
- $g_i =$ **false**, or
- $g_i = g_j \wedge g_k$ for some $j, k < i$, or
- $g_i = g_j \vee g_k$ for some $j, k < i$.

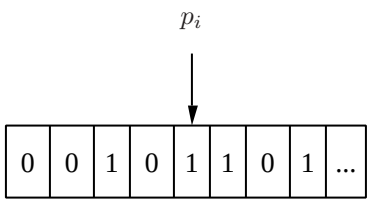
Our task is to compute the truth value of the output gate g_n.

The circuit is *monotone* because it does not contain NOT gates. Note also that the circuit has no free variables; this isn't like SAT where we try to reason about possible inputs.

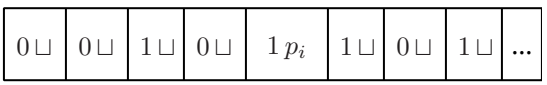
Theorem 12.3 *MCVP is **P**-complete.*

Proof: First note that MCVP can easily be computed in polynomial time. To show **P**-hardness, consider any language $L \in \mathbf{P}$. Let A be a polynomial time algorithm that decides L. We'll design an **NC** algorithm which, given some input x, constructs an instance of MCVP whose output gate takes on the same value as $A(x)$.

Recall the Turing machine model of computation: a computer consists of a tape of binary memory cells and a tape read/write head. At each step a program reads the contents of the cell over which the head is currently located and based on the result (1) updates the contents of the cell, (2) moves the tape head left or right, and (3) moves to a new program instruction. Thus, at any step i in A's computation, its current state s_i is completely described by the current tape contents, head location ($\downarrow$), and program instruction ("program counter") p_i:



Our goal is to create a layered circuit such that the output of the gates at level i represents s_i. To do this we first modify the above diagram of the state to make it more convenient. In our new representation, the entry $s_{i,j}$ includes (1) the contents of tape cell j at time i and (2) the program instruction p_i if the tape head is in position j, or the blank symbol $\sqcup$ otherwise, as shown below.



This format is convenient because to compute $s_{i,j}$, we need know only the corresponding entry and neighbors from the previous timestep: $s_{i-1,j-1}$, $s_{i-1,j}$, and $s_{i-1,j+1}$. Why? At time i, the jth tape cell is modified only if the head was at position j at time $i-1$; we can compute the new value knowing only the old value and the program instruction, both of which are included in our representation of $s_{i-1,j}$. Furthermore, the head could move into position j only if it were in position $j-1$ or $j+1$ at time $i-1$. To summarize, there is a function f_A such that for all i, $s_{i,j} = f_A(s_{i-1,j-1}, s_{i-1,j}, s_{i-1,j+1})$. We lay this out in a circuit as shown in Figure 12.1.

One important detail remains: when implementing f_A as a circuit, we may need to use NOT gates, which violates the monotonicity property of MCVP. Fortunately, in any circuit whose inputs are fixed, we can (in

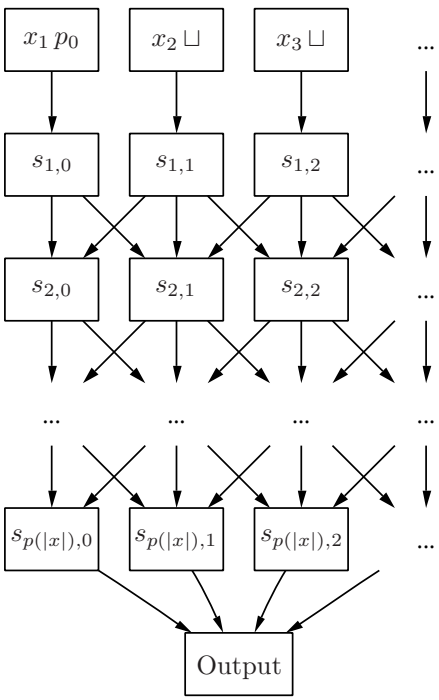


Figure 12.1: Reducing the computation of algorithm A on input $x = x_1, x_2, \ldots$ to MCVP. Each box is encoded as a small circuit. The first row's boxes are fixed input gates determined by the starting state of the algorithm. The computation of each $s_{i,j}$ is determined by f_A. $p(|x|)$ is a polynomial bound on the runtime of A on inputs of size $|x|$. The "Output" circuit examines the entries in the last timestep to determine whether the program halted on an **accept** instruction or on a **reject** instruction.

NC) eliminate NOTs while at most doubling the size of the circuit.[1] We do not give details here, but the intuition is that we "push down" the NOTs ($\neg(a \wedge b)$ becomes $(\neg a) \vee (\neg b)$; $\neg(a \vee b)$ becomes $(\neg a) \wedge (\neg b)$) until we reach the bottom level of fixed inputs, where we replace $\neg$**false** with **true** and $\neg$**true** with **false**.

Finally, to construct this circuit in parallel, we assign one processor to wire up each $s_{i,j}$ circuit, which is easily accomplished in logarithmic time on each processor since each $s_{i,j}$ has a constant number of gates, inputs, and outputs. How many processors are used? A runs in polynomial time, so both the number of timesteps and the number of tape cells used by A are bounded by some polynomial $p(|x|)$. Thus there are polynomially many $s_{i,j}$s to construct. The Output circuit, which has $p(|x|)$ inputs, can be decomposed into a tree of $O(p(|x|))$ circuits, each with a constant number of inputs; we assign a processor to each of these as well. Thus in total we use only logarithmic time on $O(p(|x|))$ processors, so our reduction is in **NC**. ■

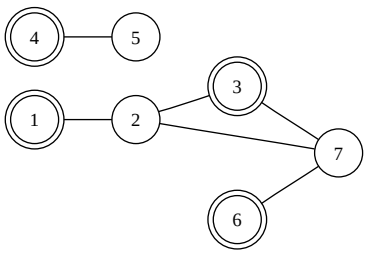
12.3 Greedy Independent Set

Greedy Independent Set (GIS) is so named because its solution is the output of the greedy algorithm for finding a large independent set[2] of nodes in a graph: consider the nodes in some order, picking each node

[1] Note since the circuit has fixed inputs, its output is also fixed at either **true** or **false**, so there is a trivial *one*-gate circuit with the same output value as the original. The trickiness comes only because we must construct the equivalent circuit in **NC**.

[2] Recall that an *independent set* is a set of nodes $S \subseteq V$ such that there is no edge between any pair of nodes in S.

if it has no edges to the nodes we've already picked. Formally, the problem is defined as follows. Given a graph $G = (V, E)$ and a bijective numbering $\text{num}(\cdot) : V \to \{1, \ldots, |V|\}$ on the nodes, find the (unique) set of nodes $S \subseteq V$ such that for all $v \in V$, $v \in S$ iff for all $v' \in S$ with $\text{num}(v') \leq \text{num}(v)$, there is no edge between v and v'. For example, in the graph below, S is the set of circled nodes.



For the purposes of proving **P**-completeness, we consider a decision version of GIS in which we must decide whether the highest-numbered node is in S. Clearly, if we cannot accomplish this, we also can't solve the general version, in which we output the entire set S.[3]

Theorem 12.4 *Greedy Independent Set is **P**-complete.*

Proof: First note that GIS is obviously in **P**. To show **P**-hardness, we reduce MCVP to the decision version of GIS. We have a monotone, fixed-input circuit C whose gates are $G = \{g_1, \ldots, g_n\}$. We must compute the output of C. For each $g_i \in G$, we create two nodes T_{g_i} and F_{g_i} representing the truth value of g_i. Exactly one of T_{g_i} and F_{g_i} will end up being in the Greedy Independent Set solution S, according to the truth value of g_i. We accomplish this through choice of node numbers and edges, as described below and as illustrated in Figure 12.2.

- For *every* $g_i \in G$, we create edge (T_{g_i}, F_{g_i}), which ensures that at most one of T_{g_i} and F_{g_i} will be in S.
- For each fixed input gate $g_i \in G$, if $g_i =$ **false**, we let $\text{num}(F_{g_i}) = 2i - 1$ and $\text{num}(T_{g_i}) = 2i$, so F_{g_i} will be chosen first; otherwise, $g_i =$ **true** and we flip the numbering so T_{g_i} will be chosen first.
- For each OR gate $g_i \in G$, let I be the set of inputs of g_i. For each $g_j \in I$ we create an edge (T_{g_j}, F_{g_i}) and we let $\text{num}(F_{g_i}) = 2i - 1$ and $\text{num}(T_{g_i}) = 2i$. Through this ordering, F_{g_i} is considered first, and through our choice of edges, F_{g_i} is picked when all $g_j \in I$ are **false**; otherwise T_{g_i} will be picked.
- Similarly, for each AND gate $g_i \in G$, let I be the set of inputs of g_i. For each $g_j \in I$ we create an edge (F_{g_j}, T_{g_i}) and we let $\text{num}(T_{g_i}) = 2i - 1$ and $\text{num}(F_{g_i}) = 2i$. Through this ordering, T_{g_i} is considered first, and through our choice of edges, T_{g_i} is picked when all $g_j \in I$ are **true**; otherwise F_{g_i} will be picked instead.

  If g_i is the output gate (i.e. $i = n$), we create a special output node v_{out}, assign it a number higher than any other in the graph, and add the edge $(F_{g_i}, v_{\text{out}})$. This final step ensures that the highest numbered node is in S if and only if the output gate is **true**, as required by the decision version of GIS which we are using. (Note that this is already satisfied if g_i is an OR gate, which is why we don't create v_{out} in that case.)

It is not hard to see through induction that this construction results in S containing the correct truth nodes with respect to the gates' truth values. Furthermore, we can easily perform this construction in **NC** by assigning one processor to construct the nodes and incident edges corresponding to each gate. ■

[3]Interestingly, solving the general search version of a **P**-complete problem given an algorithm for the decision version is often non-trivial in **NC**, whereas in **P** it is easy.

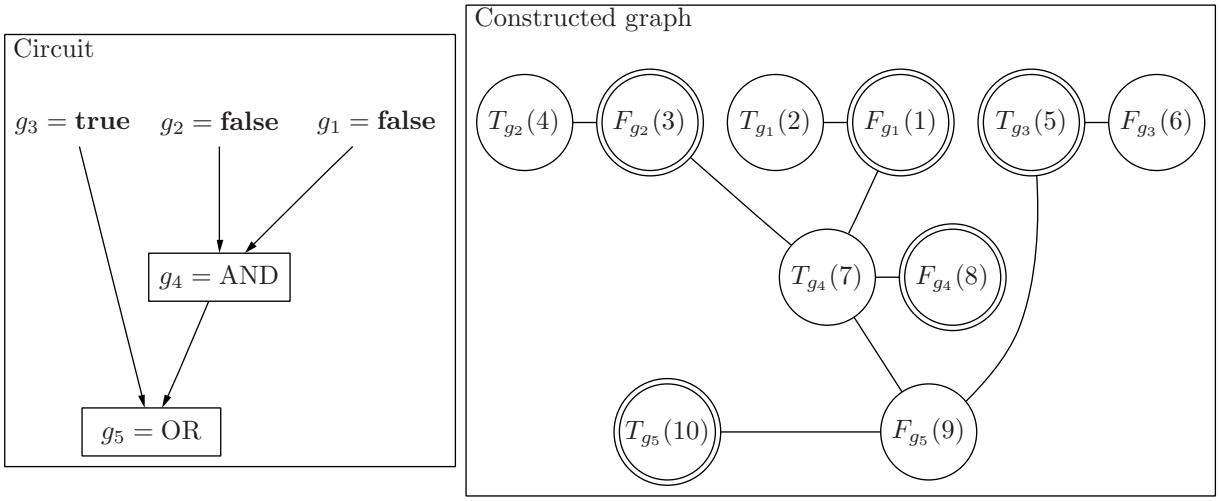


Figure 12.2: Construction in reduction from MCVP to Greedy Independent Set. The node numbering num(·) is in parentheses.

12.4 Max Flow

Another popular problem in **P** is Max Flow. We are given a directed graph $G = (V, E)$, a capacity function $c : E \to \mathbb{Z}$ on the edges, and two distinguished vertices $s, t \in V$. A *flow assignment* associates non-negative flows $f(e)$ along each edge $e \in E$ such that the edge's capacity is not exceeded ($f(e) \leq c(e)$), and at each node other than s and t, the total flow along incoming edges is equal to the total flow along outgoing edges (conservation of flow). We must compute the maximum flow leaving s (which will be the same as the maximum flow entering t) over all choices of flow assignment.

For the purposes of proving **P**-completeness, we consider a decision version of Max Flow in which we must decide whether the maximum flow is odd.

Theorem 12.5 *Max Flow is **P**-complete.*

Proof: First note that Max Flow is known to be in **P**. To show **P**-hardness, we once again reduce from MCVP. We are given a monotone circuit C whose gates $g_1, \ldots, g_n$ we assume have outdegree at most 2.[4] Our strategy will be to (1) construct a graph representing C, (2) construct a flow on that graph which sends odd flow to t iff C outputs **true**, and (3) prove that that flow is maximum.

The constructed graph has a node i for each gate g_i, with an edge from i to j iff g_i is an input of g_j. We label each such edge with capacity 2^{n-i}, so the edge capacities decrease as we traverse the graph from the fixed inputs to the circuit output. We also create a source s and sink t for our flow. For each fixed gate g_i, we create an edge $s \to i$ of capacity 0 if g_i is **false**, and of capacity equal to i's outgoing capacity if g_i is **true**. We add a single edge $g_n \to t$ of capacity 1.

To complete the construction, we add *balance edges* which equalize the in- and out-capacity of each node that corresponds to a gate (i.e. all nodes except s and t). Specifically, a node i might have more incoming than outgoing capacity (but not vice versa). Denote this extra capacity as e_i. For AND gates g_i, we add a

[4]This assumption can be easily removed.

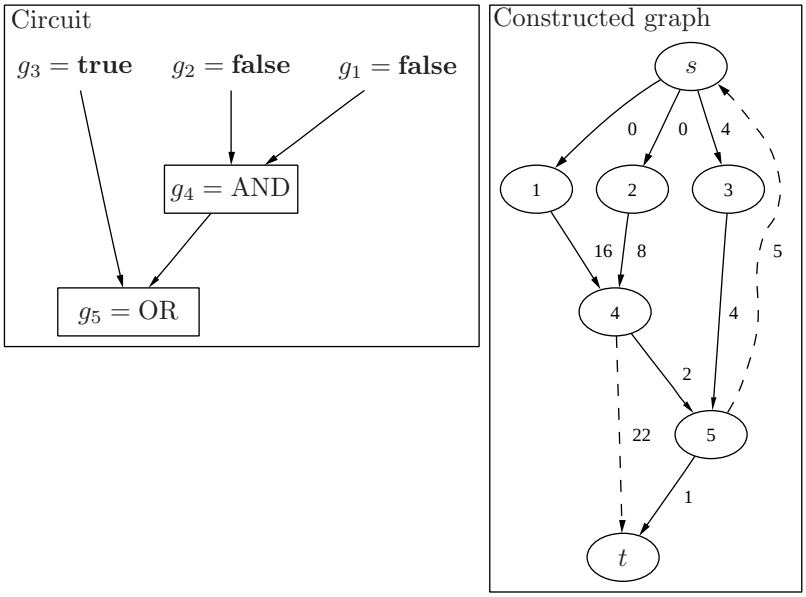


Figure 12.3: Construction in reduction from MCVP to Max Flow. The balance edges are dashed.

balance edge $i \to t$ of capacity e_i; for OR gates g_i, we add a balance edge $i \to s$ of capacity e_i. (The latter balance edges are not necessary, but simplify the analysis.)

An example of the construction is shown in Figure 12.3.

For the purposes of proving that the above reduction works, we construct a flow on this graph which fully saturates edges representing **true** wires, and leaves edges representing **false** wires empty.

We begin by saturating all the edges from s to gates fixed at **true**, and proceed by induction on the gate number i. Suppose we're at gate g_i; let's see what happens to the flow along the edges leaving node i. One of the following four cases happens.

- g_i is a **true** OR node. Then at least one input is true, providing it with at least 2^{n-i+1} inflow; it has at most two output gates, each with capacity 2^{n-i}, so it can saturate them and send any remaining flow along its balance edge $i \to s$.
- g_i is a **false** OR node. It has no inflow and thus no outflow.
- g_i is a **true** AND node. It has at least as much inflow as a true OR node, so it can saturate its outputs.
- g_i is a **false** AND node. It might have some inflow, but we don't want it to send that flow out edges representing wires in the circuit. Instead, it sends the flow along its balance edge, which by our construction has enough capacity to handle all of i's inflow as long as at least one of i's inputs is **false** (has zero flow).

Under this flow, the edge $g_n \to t$ is saturated when g_n is **true**, and has zero flow otherwise. But $g_n \to t$ has capacity 1, and is the only edge with odd capacity; thus, the $s \to t$ flow which we have constructed is odd iff g_n is **true**.

To see that the flow we've constructed is maximum, note that there is no path from s to t that could augment the flow. ■

Note that although Theorem 12.4 shows that we can't find the exact Max Flow in **NC** (unless $\mathbf{P} = \mathbf{NC}$), we do know how to approximate it in **NC**. Also note that the huge (exponentially-sized) capacities in the above proof are necessary: if a Max Flow problem instance involves only small capacities, it becomes equivalent to matching, a problem which – as we will see in the next lecture notes – we do know how to solve in **NC**.

12.5 References

The topics in this lecture are given treatment in

- C. Papadimitriou. *Computational Complexity.* Addison Wesley Longman, Reading, Massachusetts, 1994.