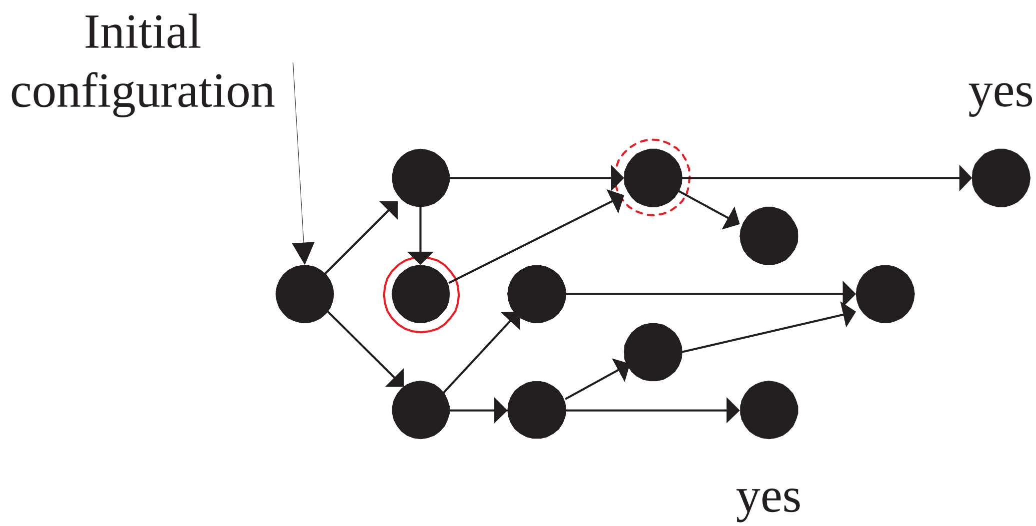
The Reachability Method

- The computation of a time-bounded TM can be represented by a directed graph.
- The TM's configurations constitute the nodes.
- Two nodes are connected by a directed edge if one yields the other in one step.
- The start node representing the initial configuration has zero in degree.

The Reachability Method (concluded)

- When the TM is nondeterministic, a node may have an out degree greater than one.
  - The graph is the same as the computation tree earlier except that identical configuration nodes are merged into one node.
- So M accepts the input if and only if there is a path from the start node to a node with a "yes" state.
- It is the reachability problem.

Illustration of the Reachability Method



Relations between Complexity Classes

Theorem 23 *Suppose $f(n)$ is proper. Then*

1. $\text{SPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$,
   $\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n))$.
2. $\text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n))$.
3. $\text{NSPACE}(f(n)) \subseteq \text{TIME}(k^{\log n + f(n)})$.

- Proof of 2:
  - Explore the computation *tree* of the NTM for "yes."
  - Specifically, generate an $f(n)$-bit sequence denoting the nondeterministic choices over $f(n)$ steps.

Proof of Theorem 23(2)

- (continued)
  - Simulate the NTM based on the choices.
  - Recycle the space and repeat the above steps.
  - Halt with "yes" when a "yes" is encountered or "no" if the tree is exhausted.
  - Each path simulation consumes at most $O(f(n))$ space because it takes $O(f(n))$ time.
  - The total space is $O(f(n))$ because space is recycled.

Proof of Theorem 23(3)

- Let k-string NTM

$$M = (K, \Sigma, \Delta, s)$$

with input and output decide $L \in \text{NSPACE}(f(n))$.

- Use the reachability method on the configuration graph of M on input x of length n.
- A configuration is a $(2k+1)$-tuple

$$(q, w_1, u_1, w_2, u_2, \ldots, w_k, u_k).$$

Proof of Theorem 23(3) (continued)

- We only care about

$$(q, i, w_2, u_2, \ldots, w_{k-1}, u_{k-1}),$$

  where i is an integer between 0 and n for the position of the first cursor.

- The number of configurations is therefore at most

$$|K| \times (n+1) \times |\Sigma|^{(2k-4) f(n)} = O(c_1^{\log n + f(n)}) \tag{1}$$

  for some c_1, which depends on M.

- Add edges to the configuration graph based on M's transition function.

Proof of Theorem 23(3) (concluded)

- $x \in L \Leftrightarrow$ there is a path in the configuration graph from the initial configuration to a configuration of the form ("yes", $i, \ldots$).[a]
- This is REACHABILITY on a graph with $O(c_1^{\log n + f(n)})$ nodes.
- It is in TIME$(c^{\log n + f(n)})$ for some c because REACHABILITY $\in$ TIME(n^j) for some j and

$$\left[c_1^{\log n + f(n)} \right]^j = (c_1^j)^{\log n + f(n)}.$$

---

[a]There may be many of them.

Space-Bounded Computation and Proper Functions

- In the definition of *space-bounded* computations earlier (p. 95), the TMs are not required to halt at all.
- When the space is bounded by a proper function f, computations can be assumed to halt:
  - Run the TM associated with f to produce a quasi-blank output of length $f(n)$ first.
  - The space-bounded computation must repeat a configuration if it runs for more than $c^{\log n + f(n)}$ steps for some c (p. 225).

Space-Bounded Computation and Proper Functions (concluded)

- (continued)
  - So we can prevent infinite loops during simulation by pruning any path longer than $c^{\log n + f(n)}$.
  - In other words, we only simulate $c^{\log n + f(n)}$ time steps per computation path.

A Grand Chain of Inclusions[a]

- It is an easy application of Theorem 23 (p. 222) that

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP.$$

- By Corollary 20 (p. 217), we know $L \subsetneq PSPACE$.
- So the chain must break somewhere between L and EXP.
- It is suspected that all four inclusions are proper.
- But there are no proofs yet.

---

[a]With input from Mr. Chin-Luei Chang (`R93922004`, `D95922007`) on October 22, 2004.

Nondeterministic Space and Deterministic Space

- By Theorem 4 (p. 101),

$$\text{NTIME}(f(n)) \subseteq \text{TIME}(c^{f(n)}),$$

an exponential gap.

- There is no proof yet that the exponential gap is inherent.
- How about NSPACE vs. SPACE?
- Surprisingly, the relation is only quadratic—a polynomial—by Savitch's theorem.

Savitch's Theorem

Theorem 24 (Savitch (1970))

$$\text{REACHABILITY} \in \text{SPACE}(\log^2 n).$$

- Let $G(V, E)$ be a graph with n nodes.
- For $i \geq 0$, let
$$\text{PATH}(x, y, i)$$
mean there is a path from node x to node y of length at most 2^i.
- There is a path from x to y if and only if
$$\text{PATH}(x, y, \lceil \log n \rceil)$$
holds.

The Proof (continued)

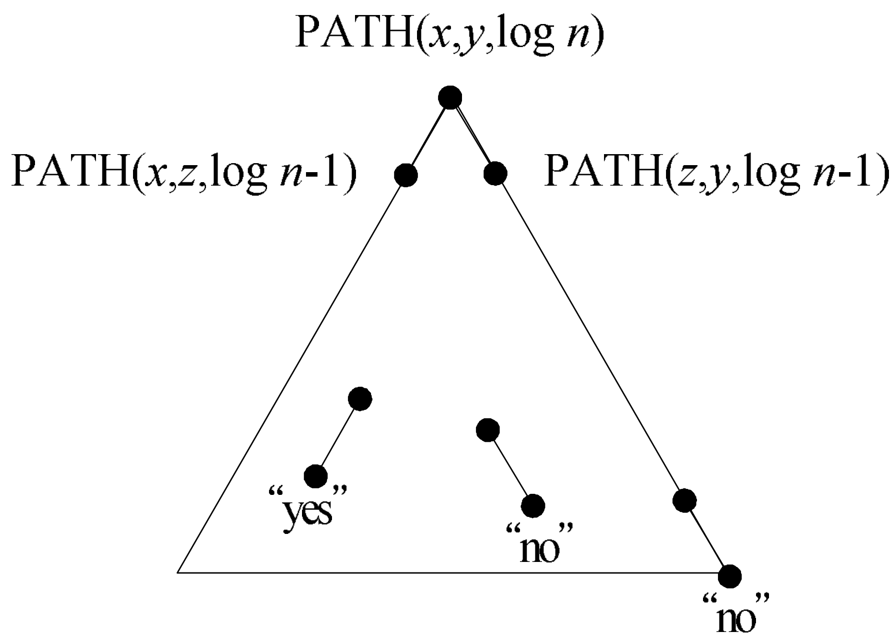
- For $i > 0$, PATH(x, y, i) if and only if there exists a z such that PATH$(x, z, i-1)$ and PATH$(z, y, i-1)$.
- For PATH$(x, y, 0)$, check the input graph or if $x = y$.
- Compute PATH$(x, y, \lceil \log n \rceil)$ with a depth-first search on a graph with nodes (x, y, z, i)s (see next page).[a]
- Like stacks in recursive calls, we keep only the current path of (x, y, i)s.
- The space requirement is proportional to the depth of the tree: $\lceil \log n \rceil$.

[a]Contributed by Mr. Chuan-Yao Tan on October 11, 2011.

The Proof (continued): Algorithm for PATH(x, y, i)

```
if i = 0 then
    if x = y or (x, y) ∈ E then
        return true;
    else
        return false;
    end if
else
    for z = 1, 2, ..., n do
        if PATH(x, z, i − 1) and PATH(z, y, i − 1) then
            return true;
        end if
    end for
    return false;
end if
```

The Proof (concluded)



- Depth is $\lceil \log n \rceil$, and each node (x, y, z, i) needs space $O(\log n)$.
- The total space is $O(\log^2 n)$.

The Relation between Nondeterministic Space and Deterministic Space Only Quadratic

Corollary 25 *Let $f(n) \geq \log n$ be proper. Then*

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

- Apply Savitch's proof to the configuration graph of the NTM on the input.
- From p. 225, the configuration graph has $O(c^{f(n)})$ nodes; hence each node takes space $O(f(n))$.
- But if we construct *explicitly* the whole graph before applying Savitch's theorem, we get $O(c^{f(n)})$ space!

The Proof (continued)

- The way out is *not* to generate the graph at all.
- Instead, keep the graph implicit.
- In fact, we check node connectedness only when $i = 0$ on p. 233, by examining the input string G.
- There, given configurations x and y, we go over the Turing machine's program to determine if there is an instruction that can turn x into y in one step.[a]

[a]Thanks to a lively class discussion on October 15, 2003.

The Proof (concluded)

- The z variable in the algorithm on p. 233 simply runs through all possible valid configurations.
  - Let $z = 0, 1, \ldots, O(c^{f(n)})$.
  - Make sure z is a valid configuration before using it in the recursive calls.[a]
- Each z has length $O(f(n))$ by Eq. (1) on p. 225.
- So each node needs space $O(f(n))$.
- As the depth of the recursive call on p. 233 is $O(\log c^{f(n)})$, the total space is therefore $O(f^2(n))$.

[a]Thanks to a lively class discussion on October 13, 2004.

Implications of Savitch's Theorem

- PSPACE = NPSPACE.
- Nondeterminism is less powerful with respect to space.
- Nondeterminism may be very powerful with respect to time as it is not known if P = NP.

Nondeterministic Space Is Closed under Complement

- Closure under complement is trivially true for deterministic complexity classes (p. 210).
- It is known that[a]

$$\text{coNSPACE}(f(n)) = \text{NSPACE}(f(n)). \tag{2}$$

- So

$$\begin{aligned} \text{coNL} &= \text{NL}, \\ \text{coNPSPACE} &= \text{NPSPACE}. \end{aligned}$$

- But it is not known whether coNP = NP.

[a]Szelepscényi (1987) and Immerman (1988).

Reductions and Completeness

It is unworthy of excellent men
to lose hours like slaves in the labor of
computation.
— Gottfried Wilhelm von Leibniz (1646–1716)

Degrees of Difficulty

- When is a problem more difficult than another?
- B **reduces to** A if there is a transformation R which for every input x of B yields an input $R(x)$ of A.[a]
  - The answer to x for B is the same as the answer to $R(x)$ for A.
  - R is easy to compute.
- We say problem A is at least as hard as problem B if B reduces to A.

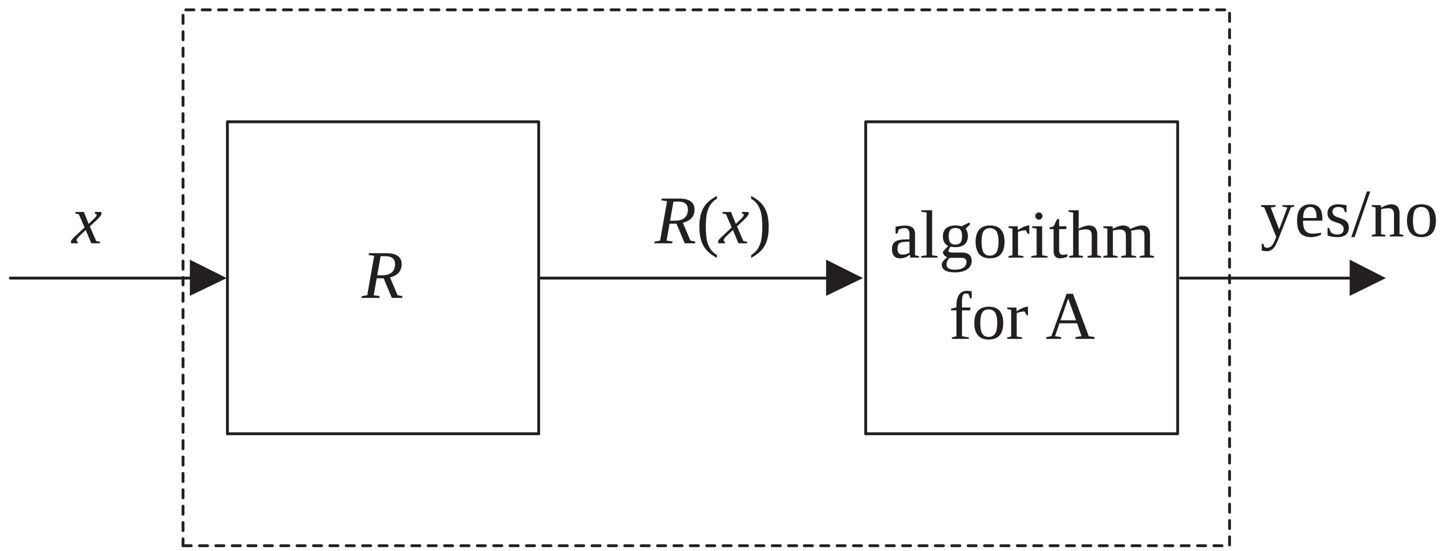
[a]See also p. 148.

Degrees of Difficulty (concluded)

- This makes intuitive sense: If A is able to solve your problem B after only a little bit of work of R, then A must be at least as hard.
  - If A is easy to solve, it combined with R (which is also easy) would make B easy to solve, too.[a]
  - So if B is hard to solve, A must be hard (if not harder), too!

[a] Thanks to a lively class discussion on October 13, 2009.

Reduction



Solving problem B by calling the algorithm for problem A *once* and *without* further processing its answer.

Comments[a]

- Suppose B reduces to A via a transformation R.
- The input x is an instance of B.
- The output $R(x)$ is an instance of A.
- $R(x)$ may not span all possible instances of A.[b]
  - Some instances of A may never appear in the range of R.
- But x must be a general instance for B.

---

[a] Contributed by Mr. Ming-Feng Tsai (D92922003) on October 29, 2003.

[b] $R(x)$ may not be onto; Mr. Alexandr Simak (D98922040) on October 13, 2009.

Is "Reduction" a Confusing Choice of Word?[a]

- If B reduces to A, doesn't that intuitively make A smaller and simpler?
  - Sometimes, we say, "B can be reduced to A."
- But our definition means just the opposite.
- Our definition says in this case B is a special case of A.
- Hence A is harder.

[a] Moore and Mertens (2011).

Reduction between Languages

- Language L_1 is **reducible to** L_2 if there is a function R computable by a deterministic TM in space $O(\log n)$.
- Furthermore, for all inputs x, $x \in L_1$ if and only if $R(x) \in L_2$.
- R is said to be a (**Karp**) **reduction** from L_1 to L_2.

Reduction between Languages (concluded)

- Note that by Theorem 23 (p. 222), R runs in polynomial time.
  - In most cases, a polynomial-time R suffices for proofs.[a]
- Suppose R is a reduction from L_1 to L_2.
- Then solving "$R(x) \in L_2$?" is an algorithm for solving "$x \in L_1$?"[b]

---

[a] In fact, unless stated otherwise, we will only require that the reduction R run in polynomial time.

[b] Of course, it may not be an optimal one.

A Paradox?

- Degree of difficulty is not defined in terms of *absolute* complexity.
- So a language $B \in \text{TIME}(n^{99})$ may be "easier" than a language $A \in \text{TIME}(n^3)$.
  - Again, this happens when B is reducible to A.
- But is this a contradiction if the best algorithm for B requires n^{99} steps?
- That is, how can a problem *requiring* n^{99} steps be reducible to a problem solvable in n^3 steps?

Paradox Resolved

- The so-called contradiction does not hold.
- Suppose we solve the problem "$x \in$ B?" via "$R(x) \in$ A?"
- We must consider the time spent by $R(x)$ and its length $|\, R(x) \,|$ because $R(x)$ (not x) is presented to A.

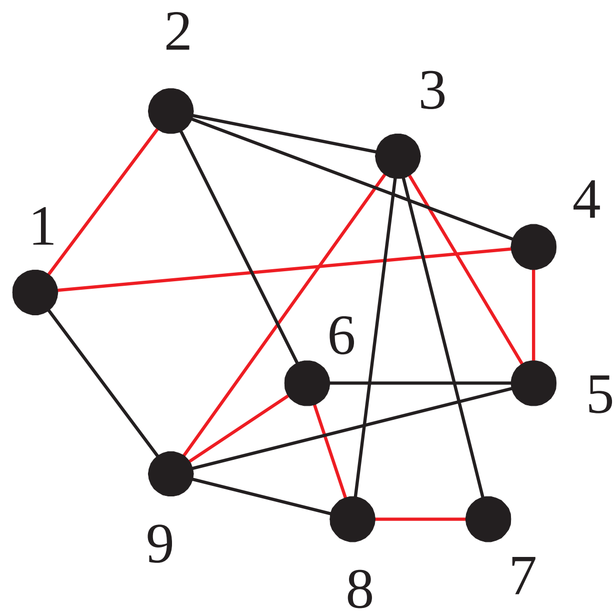
HAMILTONIAN PATH

- A **Hamiltonian path** of a graph is a path that visits every node of the graph exactly once.
- Suppose graph G has n nodes: $1, 2, \ldots, n$.
- A Hamiltonian path can be expressed as a permutation π of $\{1, 2, \ldots, n\}$ such that
  - $\pi(i) = j$ means the ith position is occupied by node j.
  - $(\pi(i), \pi(i+1)) \in G$ for $i = 1, 2, \ldots, n-1$.
- HAMILTONIAN PATH asks if a graph has a Hamiltonian path.

Reduction of HAMILTONIAN PATH to SAT

- Given a graph G, we shall construct a CNF $R(G)$ such that $R(G)$ is satisfiable iff G has a Hamiltonian path.
- $R(G)$ has n^2 boolean variables x_{ij}, $1 \leq i, j \leq n$.
- x_{ij} means

  "the ith position in the Hamiltonian path is occupied by node j."
- Our reduction will produce clauses.



$x_{12} = x_{21} = x_{34} = x_{45} = x_{53} = x_{69} = x_{76} = x_{88} = x_{97} = 1$;
$\pi(1) = 2, \pi(2) = 1, \pi(3) = 4, \pi(4) = 5, \pi(5) = 3, \pi(6) = 9, \pi(7) = 6, \pi(8) = 8, \pi(9) = 7$.

The Clauses of $R(G)$ and Their Intended Meanings

1. Each node j must appear in the path.
   - $x_{1j} \lor x_{2j} \lor \cdots \lor x_{nj}$ for each j.
2. No node j appears twice in the path.
   - $\neg x_{ij} \lor \neg x_{kj} (\equiv \neg (x_{ij} \land x_{kj}))$ for all i, j, k with $i \neq k$.
3. Every position i on the path must be occupied.
   - $x_{i1} \lor x_{i2} \lor \cdots \lor x_{in}$ for each i.
4. No two nodes j and k occupy the same position in the path.
   - $\neg x_{ij} \lor \neg x_{ik} (\equiv \neg (x_{ij} \land x_{ik}))$ for all i, j, k with $j \neq k$.
5. Nonadjacent nodes i and j cannot be adjacent in the path.
   - $\neg x_{ki} \lor \neg x_{k+1,j}$ for all $(i, j) \notin G$ and $k = 1, 2, \ldots, n-1$.

The Proof

- $R(G)$ contains $O(n^3)$ clauses.
- $R(G)$ can be computed efficiently (simple exercise).
- Suppose $T \models R(G)$.
- From the 1st and 2nd types of clauses, for each node j there is a unique position i such that $T \models x_{ij}$.
- From the 3rd and 4th types of clauses, for each position i there is a unique node j such that $T \models x_{ij}$.
- So there is a permutation π of the nodes such that $\pi(i) = j$ if and only if $T \models x_{ij}$.

The Proof (concluded)

- The 5th type of clauses furthermore guarantee that $(\pi(1), \pi(2), \ldots, \pi(n))$ is a Hamiltonian path.
- Conversely, suppose G has a Hamiltonian path

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

  where π is a permutation.
- Clearly, the truth assignment

$$T(x_{ij}) = \texttt{true} \text{ if and only if } \pi(i) = j$$

  satisfies all clauses of $R(G)$.

A Comment[a]

- An answer to "Is $R(G)$ satisfiable?" does answer "Is G Hamiltonian?"
- But a positive answer does not give a Hamiltonian path for G.
  - Providing a witness is not a requirement of reduction.
- A positive answer to "Is $R(G)$ satisfiable?" *plus a satisfying truth assignment* does provide us with a Hamiltonian path for G.

---

[a] Contributed by Ms. Amy Liu (`J94922016`) on May 29, 2006.

Reduction of REACHABILITY to CIRCUIT VALUE

- Note that both problems are in P.
- Given a graph $G = (V, E)$, we shall construct a *variable-free* circuit $R(G)$.
- The output of $R(G)$ is true if and only if there is a path from node 1 to node n in G.
- Idea: the Floyd-Warshall algorithm.

The Gates

- The gates are
  - g_{ijk} with $1 \leq i, j \leq n$ and $0 \leq k \leq n$.
  - h_{ijk} with $1 \leq i, j, k \leq n$.
- g_{ijk}: There is a path from node i to node j without passing through a node bigger than k.
- h_{ijk}: There is a path from node i to node j passing through k but not any node bigger than k.
- Input gate $g_{ij0} = \texttt{true}$ if and only if $i = j$ or $(i, j) \in E$.

The Construction

- h_{ijk} is an AND gate with predecessors $g_{i,k,k-1}$ and $g_{k,j,k-1}$, where $k = 1, 2, \ldots, n$.
- g_{ijk} is an OR gate with predecessors $g_{i,j,k-1}$ and $h_{i,j,k}$, where $k = 1, 2, \ldots, n$.
- g_{1nn} is the output gate.
- Interestingly, $R(G)$ uses no $\neg$ gates.
  - It is a **monotone circuit**.

Reduction of CIRCUIT SAT to SAT

- Given a circuit C, we will construct a boolean expression $R(C)$ such that $R(C)$ is satisfiable iff C is.
  - $R(C)$ will turn out to be a CNF.
  - $R(C)$ is basically a depth-2 circuit; furthermore, each gate has out-degree 1.
- The variables of $R(C)$ are those of C plus g for each gate g of C.
  - The g's propagate the truth values for the CNF.
- Each gate of C will be turned into equivalent clauses.
- Recall that clauses are $\wedge$ed together by definition.

The Clauses of $R(C)$

g **is a variable gate** x**:** Add clauses $(\neg g \lor x)$ and $(g \lor \neg x)$.

- Meaning: $g \Leftrightarrow x$.

g **is a `true` gate:** Add clause (g).

- Meaning: g must be true to make $R(C)$ true.

g **is a `false` gate:** Add clause $(\neg g)$.

- Meaning: g must be false to make $R(C)$ true.

g **is a** $\neg$ **gate with predecessor gate** h**:** Add clauses $(\neg g \lor \neg h)$ and $(g \lor h)$.

- Meaning: $g \Leftrightarrow \neg h$.

The Clauses of $R(C)$ (concluded)

g **is a** $\vee$ **gate with predecessor gates** h **and** h'**:** Add clauses $(\neg h \vee g)$, $(\neg h' \vee g)$, and $(h \vee h' \vee \neg g)$.

- Meaning: $g \Leftrightarrow (h \vee h')$.

g **is a** $\wedge$ **gate with predecessor gates** h **and** h'**:** Add clauses $(\neg g \vee h)$, $(\neg g \vee h')$, and $(\neg h \vee \neg h' \vee g)$.

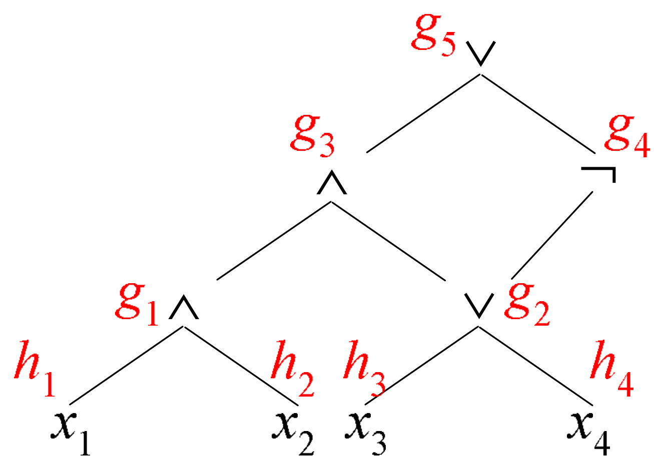
- Meaning: $g \Leftrightarrow (h \wedge h')$.

g **is the output gate:** Add clause (g).

- Meaning: g must be true to make $R(C)$ true.

Note: If gate g feeds gates $h_1, h_2, \ldots$, then variable g appears in the clauses for $h_1, h_2, \ldots$ in $R(C)$.

An Example



$$
\begin{array}{ll}
& (h_1 \Leftrightarrow x_1) \land (h_2 \Leftrightarrow x_2) \land (h_3 \Leftrightarrow x_3) \land (h_4 \Leftrightarrow x_4) \\
\land & [\, g_1 \Leftrightarrow (h_1 \land h_2) \,] \land [\, g_2 \Leftrightarrow (h_3 \lor h_4) \,] \\
\land & [\, g_3 \Leftrightarrow (g_1 \land g_2) \,] \land (g_4 \Leftrightarrow \neg g_2) \\
\land & [\, g_5 \Leftrightarrow (g_3 \lor g_4) \,] \land g_5 .
\end{array}
$$