I/O-efficient Symbolic Model Checking

Steffan Christ Sølvsten

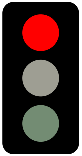
15th of May 2025

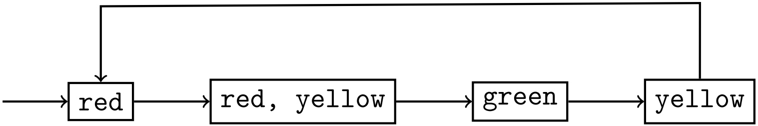
AARHUS
UNIVERSITY

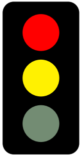
defer (difœ').

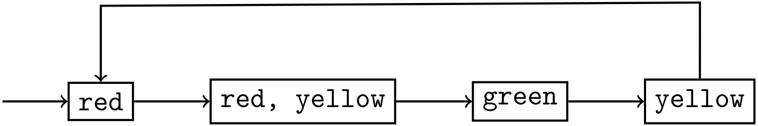
defer (difœ'). *v.t.* to put off; to postpone.

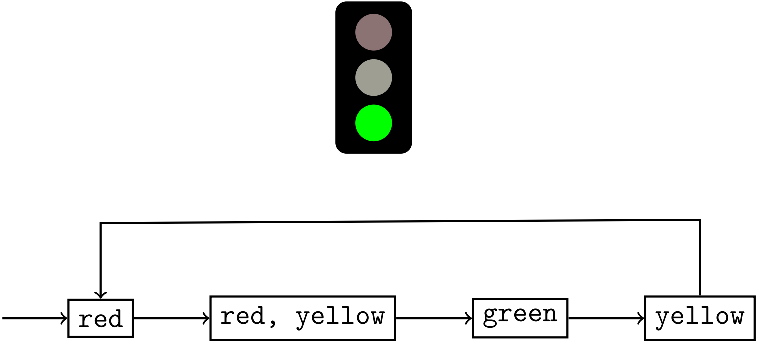
[. . .]

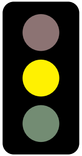


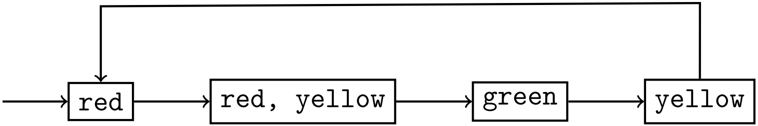

red
red, yellow
green
yellow

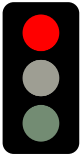


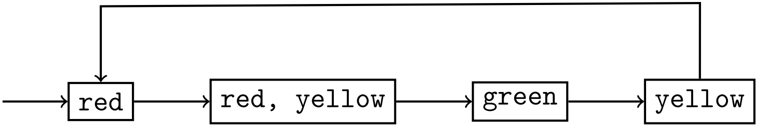

red
red, yellow
green
yellow

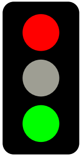

red
red, yellow
green
yellow

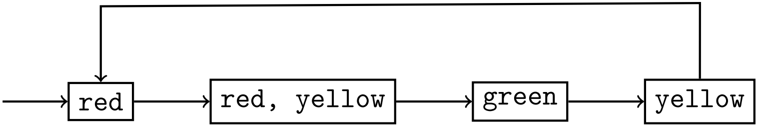



red
red, yellow
green
yellow



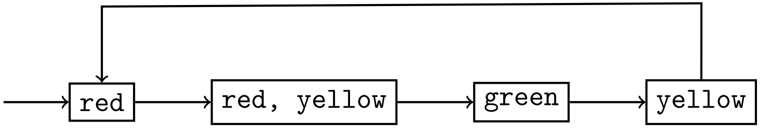

red
red, yellow
green
yellow




red
red, yellow
green
yellow

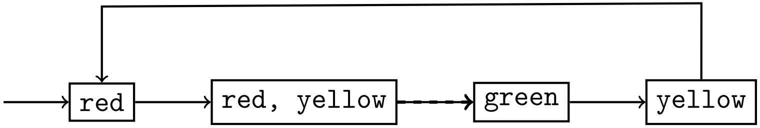
"*It is either red and/or ($\vee$) yellow or it is exclusively ($\oplus$) green.*"

$$(\texttt{red} \vee \texttt{yellow}) \oplus \texttt{green}$$

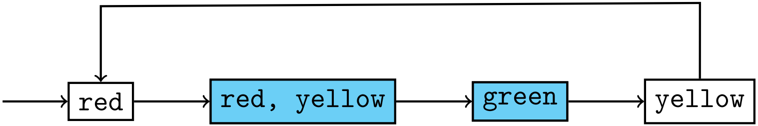


"When red and yellow, it turns exclusively green."

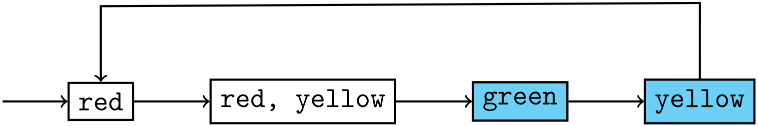
$$\texttt{red} \wedge \texttt{yellow} \rightarrow \neg\texttt{red}' \wedge \neg\texttt{yellow}' \wedge \texttt{green}'$$



$$RelProd(S_{\vec{x}}, T_{\vec{x},\vec{x}'}) \triangleq (\exists \vec{x} \,.\, S_{\vec{x}} \wedge T_{\vec{x},\vec{x}'})[\vec{x}' \mapsto \vec{x}]$$

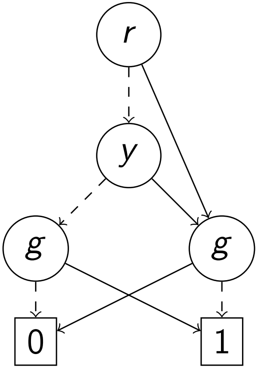


$$RelProd(S_{\vec{x}}, T_{\vec{x},\vec{x}'}) \triangleq (\exists \vec{x} \;.\; S_{\vec{x}} \wedge T_{\vec{x},\vec{x}'})[\vec{x}' \mapsto \vec{x}]$$



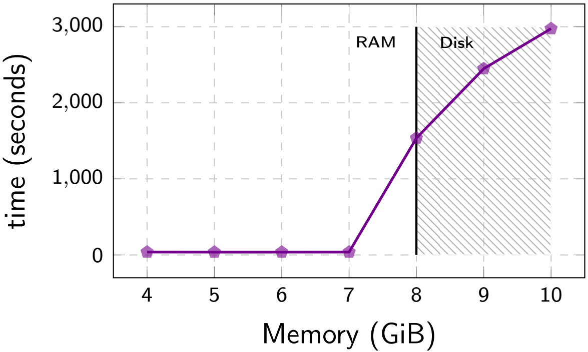
Binary Decision Diagram (BDD)

$(\texttt{red} \vee \texttt{yellow}) \oplus \texttt{green}$



Usually BDDs are implemented by means of:

- Depth-first recursion
- Hash Tables



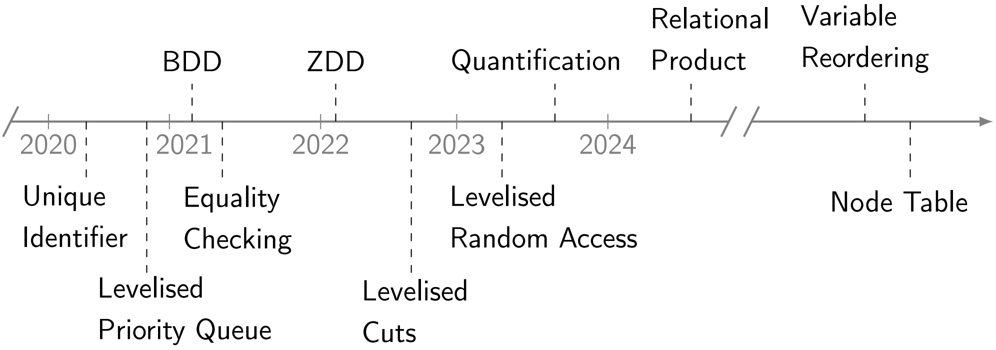
`BuDDy` BDD Library

- **Formal Methods**:

  Motivation and applications of algorithms.
- **Algorithmics**:

  Theoretical tools for the design and analysis of algorithms.
- **Algorithm Engineering**:

  Experimental evaluation and design for real-life computers.


BDD
ZDD
Quantification
Relational Product
Variable Reordering
2020
2021
2022
2023
2024
Unique Identifier
Levelised Priority Queue
Equality Checking
Levelised Cuts
Levelised Random Access
Node Table

The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation

Lars Arge

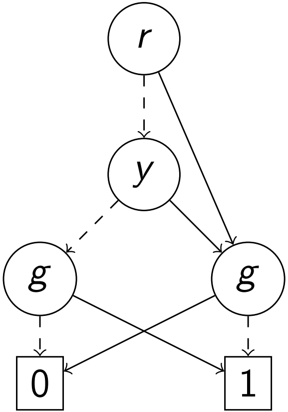
Department of Computer Science

University of Aarhus

August 1996

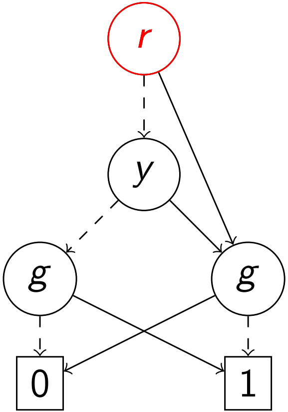
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



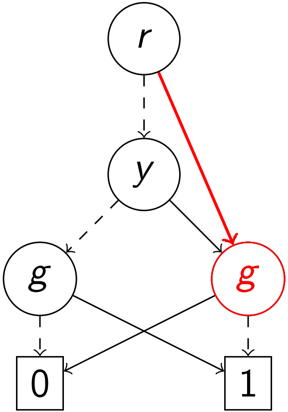
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



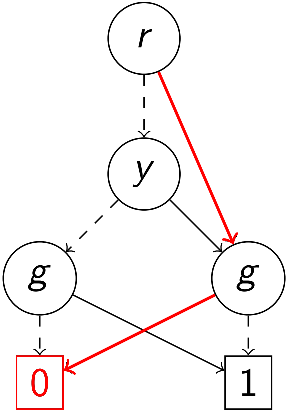
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



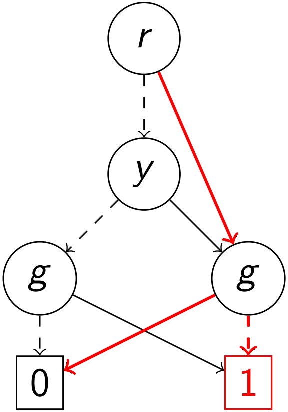
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



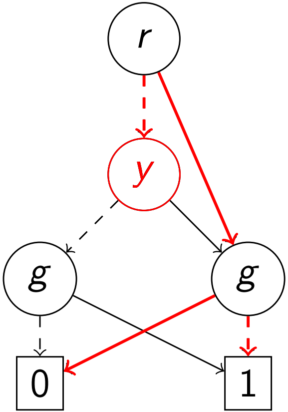
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



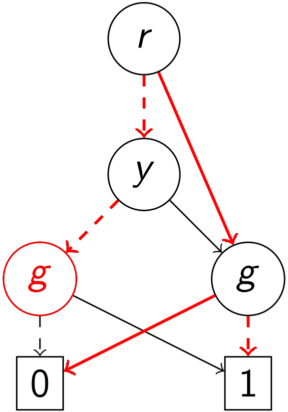
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



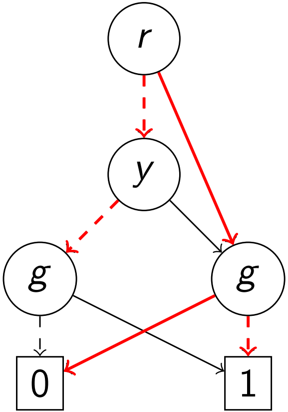
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)



Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)

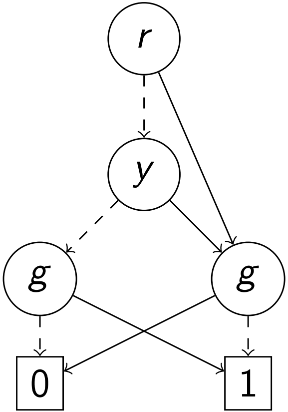


Processing Order

- **Depth-first**:

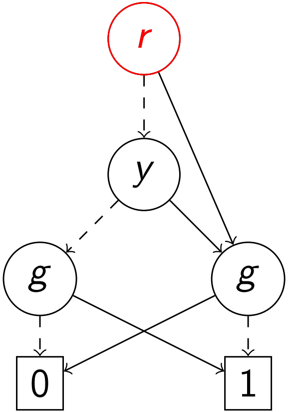
  Last in, first out Queue (Stack)
- **Breadth-first**:

  First in, first out Queue



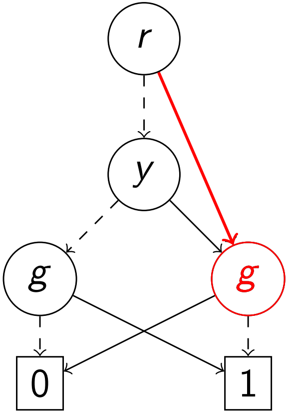
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue



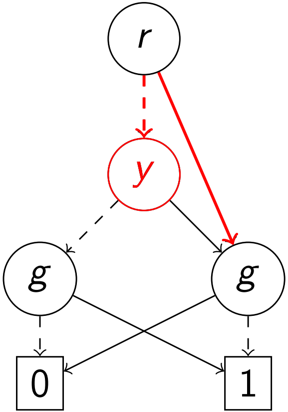
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue



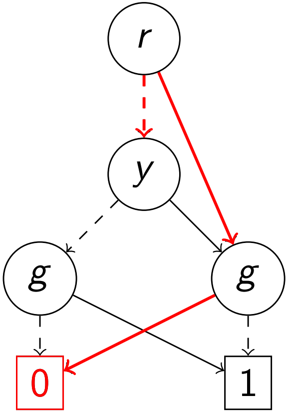
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue



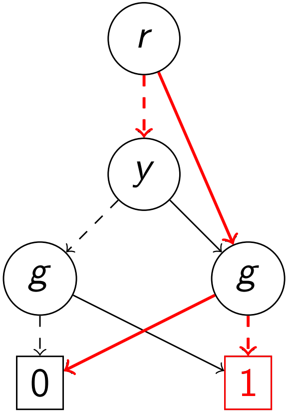
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue



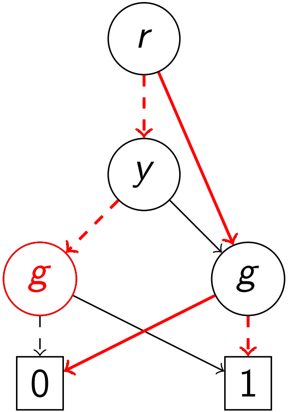
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue



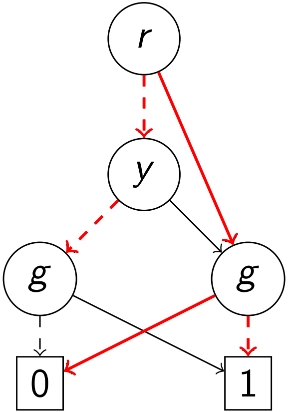
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue



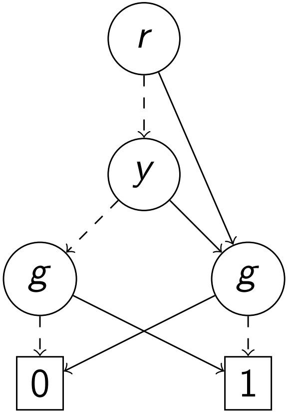
Processing Order

- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue

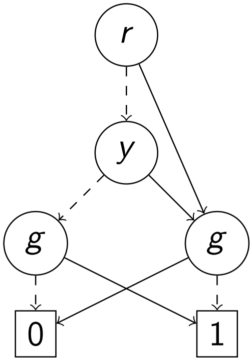


Processing Order

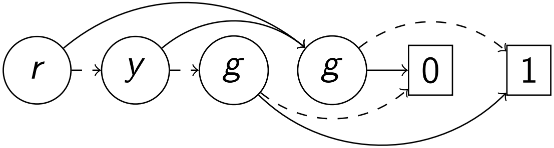
- **Depth-first**:
  Last in, first out Queue (Stack)
- **Breadth-first**:
  First in, first out Queue
- **Time-forward Processing**:
  Priority Queue



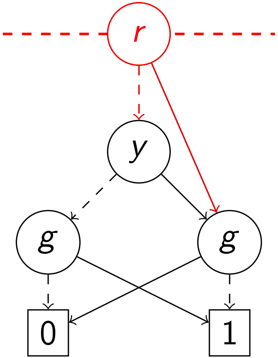
Time-forward Processing



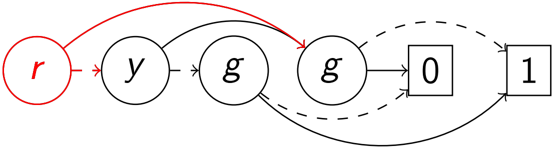
$\Longrightarrow$



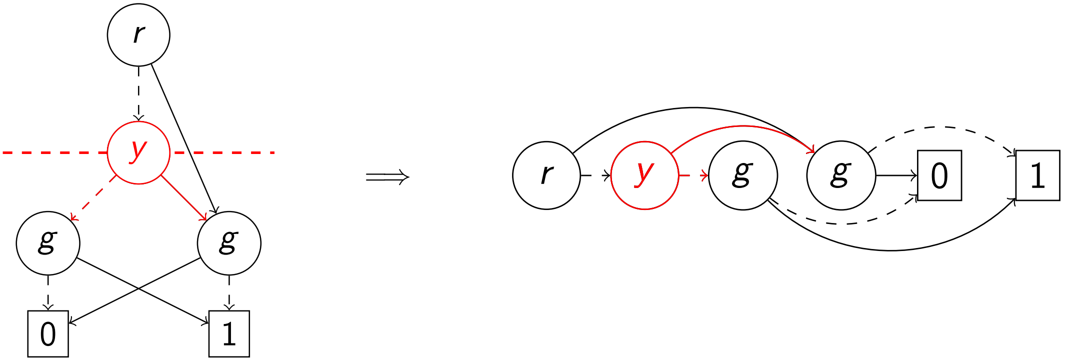
Time-forward Processing



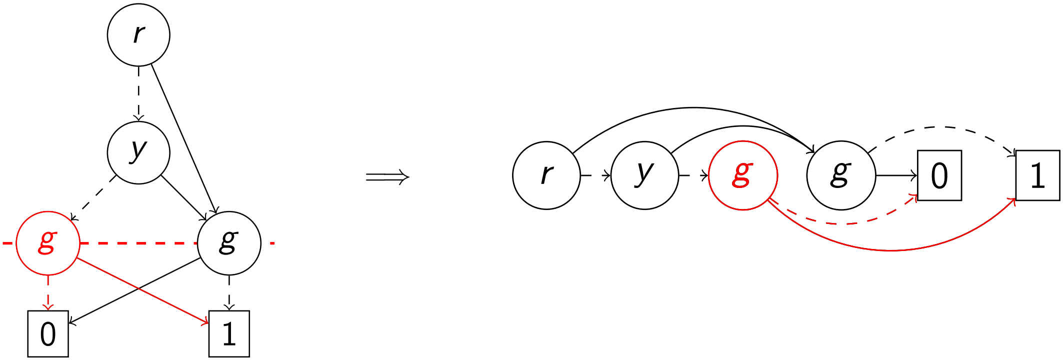
$\Longrightarrow$



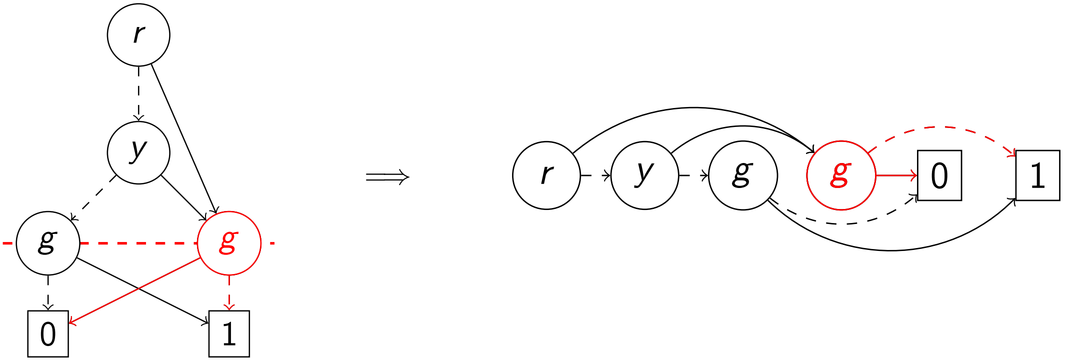
Time-forward Processing



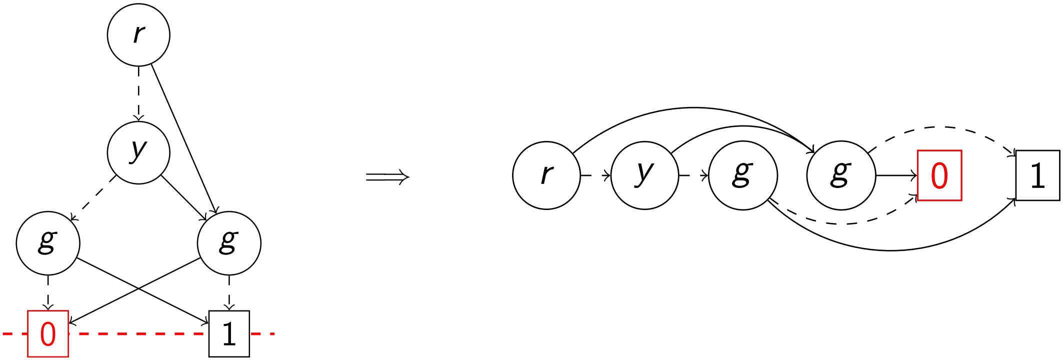
Time-forward Processing



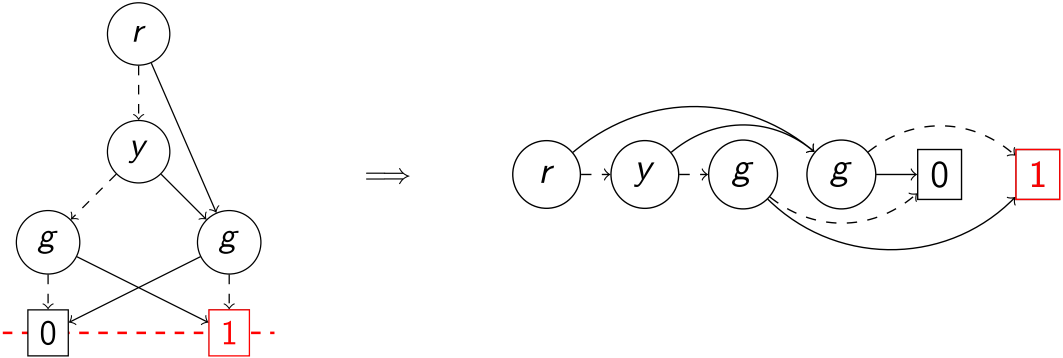
Time-forward Processing



Time-forward Processing

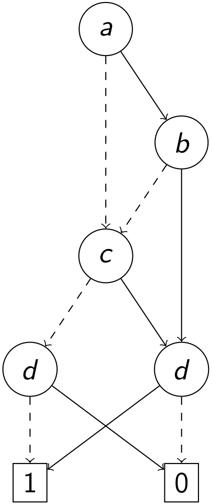


Time-forward Processing

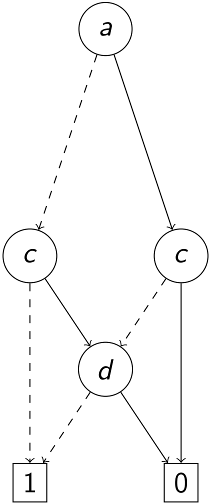


$$\phi \wedge \psi$$

$\phi \wedge \psi$

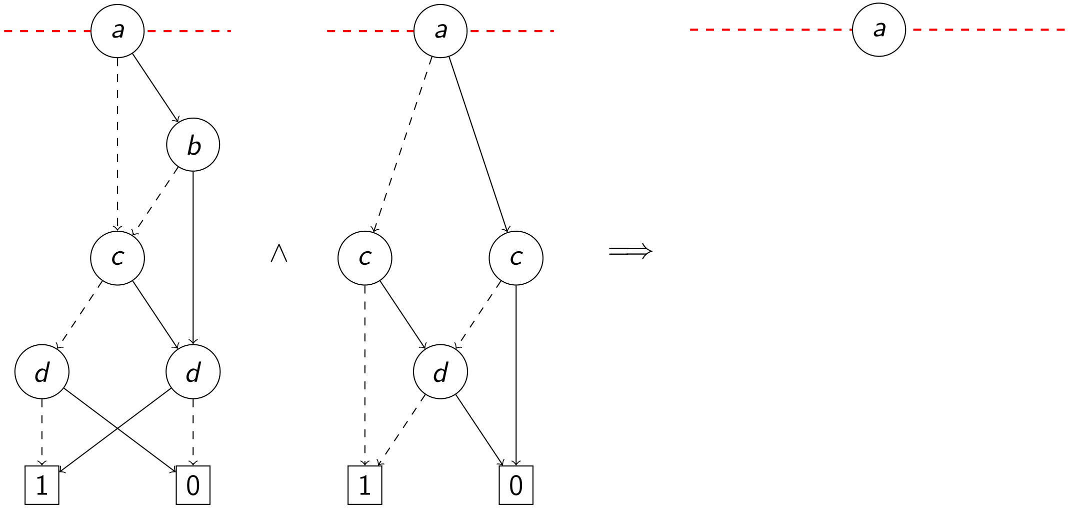


$\wedge$

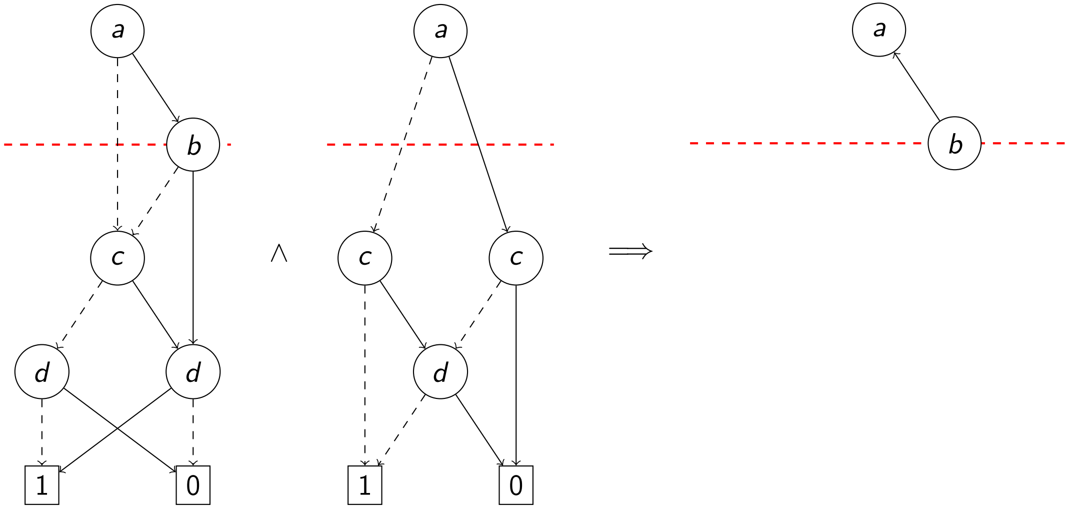


$\Longrightarrow$

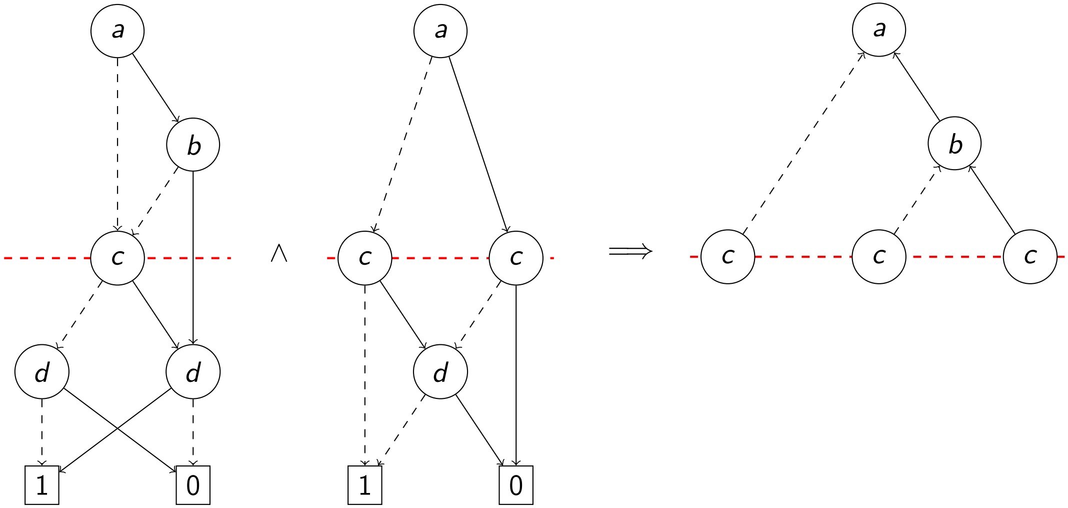
$\phi \wedge \psi$



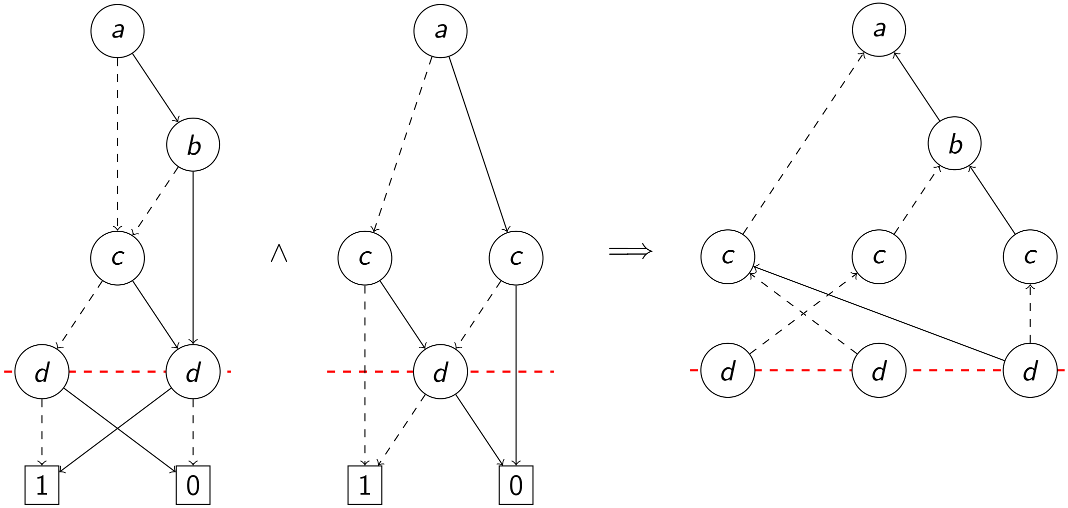
$\phi \wedge \psi$



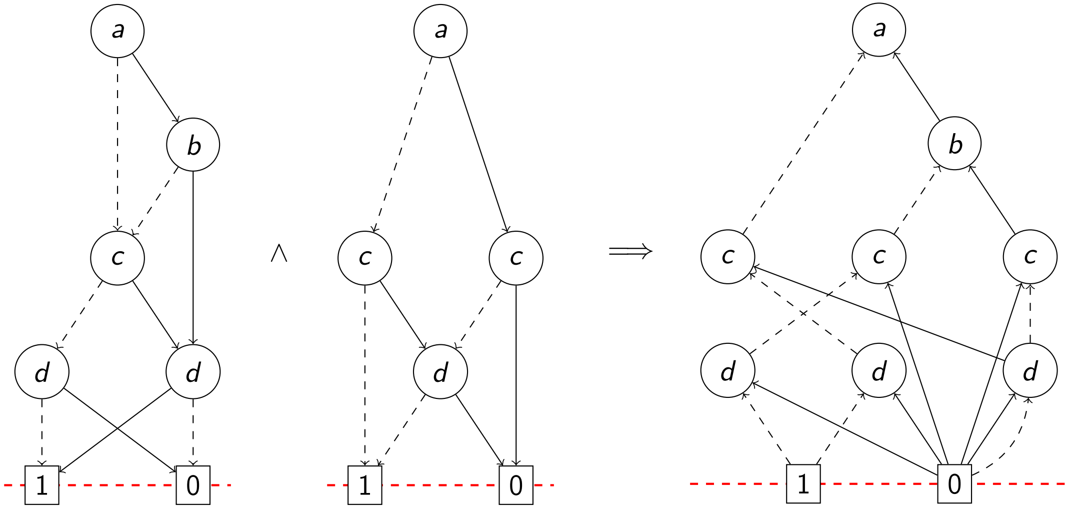
$\phi \wedge \psi$



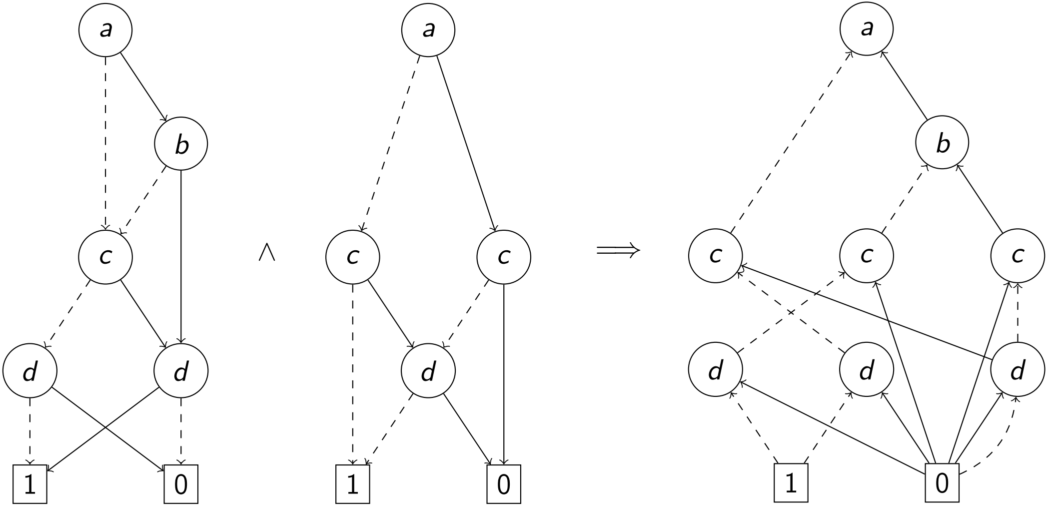
$\phi \wedge \psi$



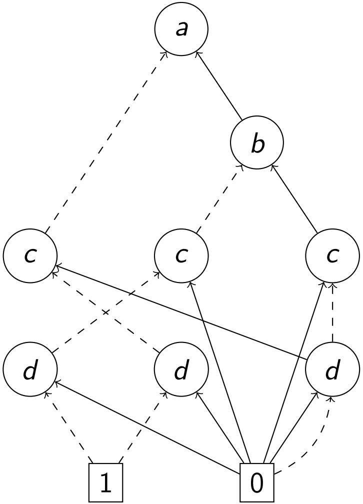
$\phi \wedge \psi$



$\phi \wedge \psi$

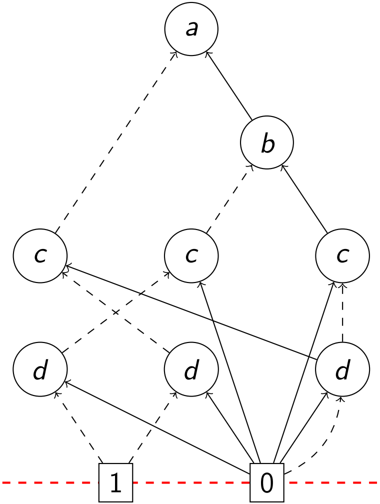


$\phi \wedge \psi$



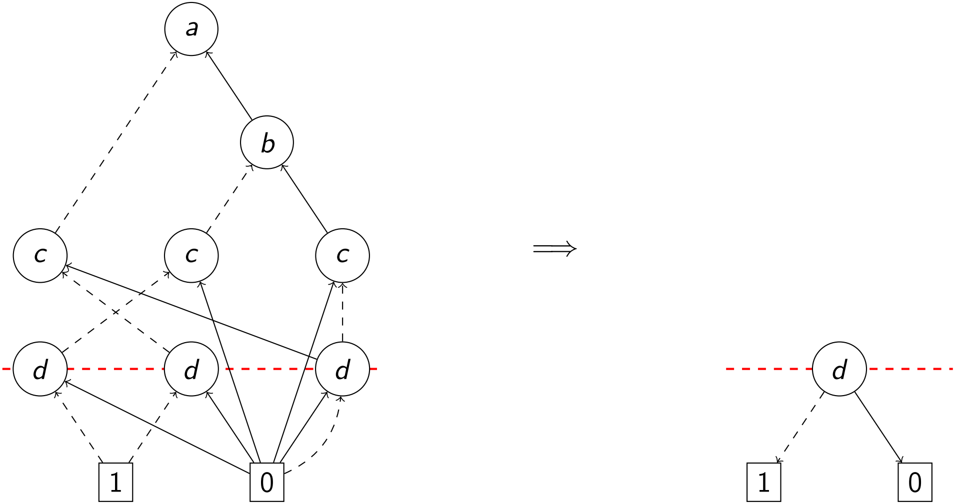
$\Longrightarrow$

$\phi \wedge \psi$

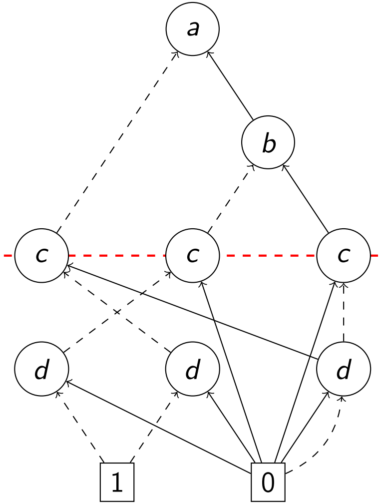


$\Longrightarrow$

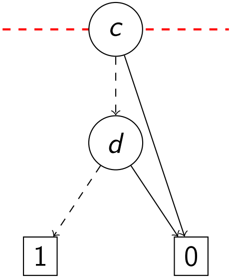
$\phi \wedge \psi$



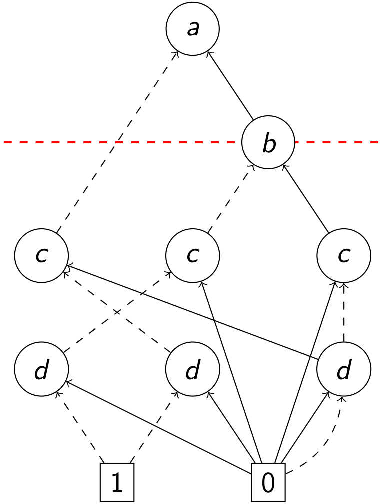
$\phi \wedge \psi$



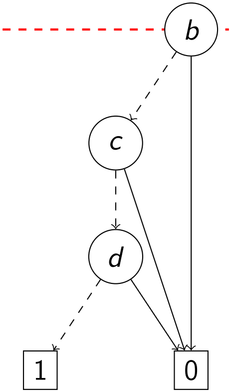
$\Longrightarrow$



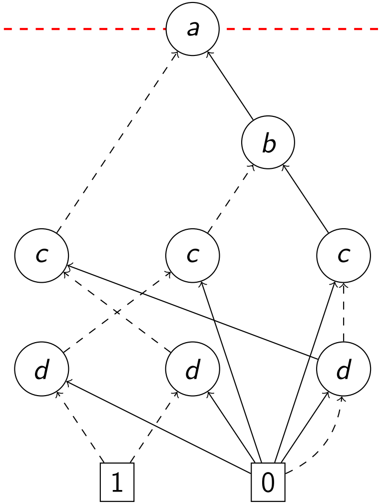
$\phi \land \psi$



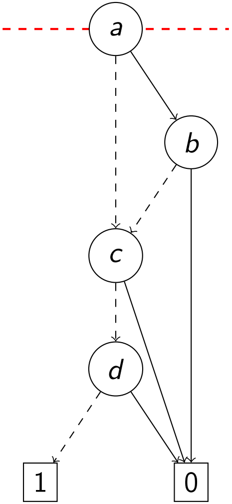
$\Longrightarrow$



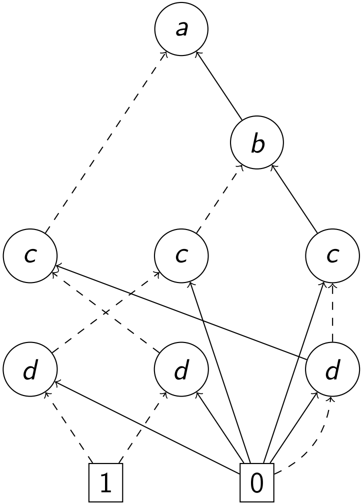
$\phi \wedge \psi$



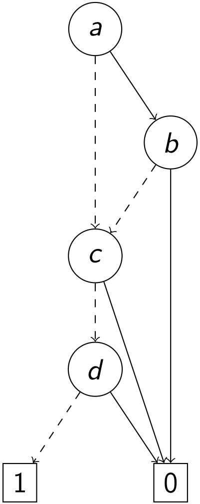
$\Longrightarrow$



$\phi \wedge \psi$



$\Longrightarrow$



Depth-first

$$\mathcal{O}(N + T)$$

but not I/O efficient!

Time-forward Processing

$$\mathcal{O}((N + T)\log(N + T))$$

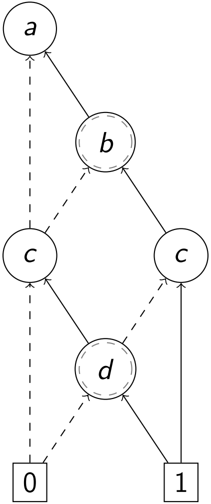
but I/O efficient!

N: Input Size T: (Unreduced) Output Size

$$\exists x : \phi(x)$$

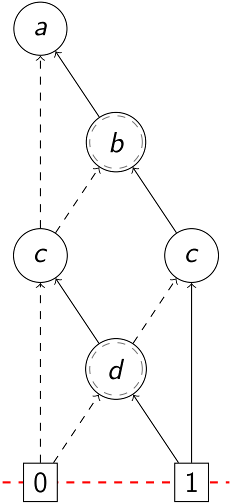
$$\exists x : \phi(x) \equiv \phi(0) \vee \phi(1)$$

$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



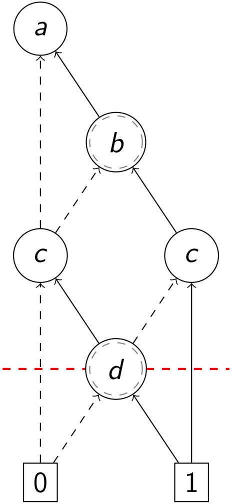
$\Longrightarrow$

$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$

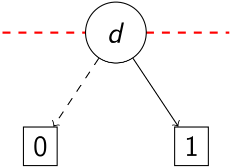


$\Longrightarrow$

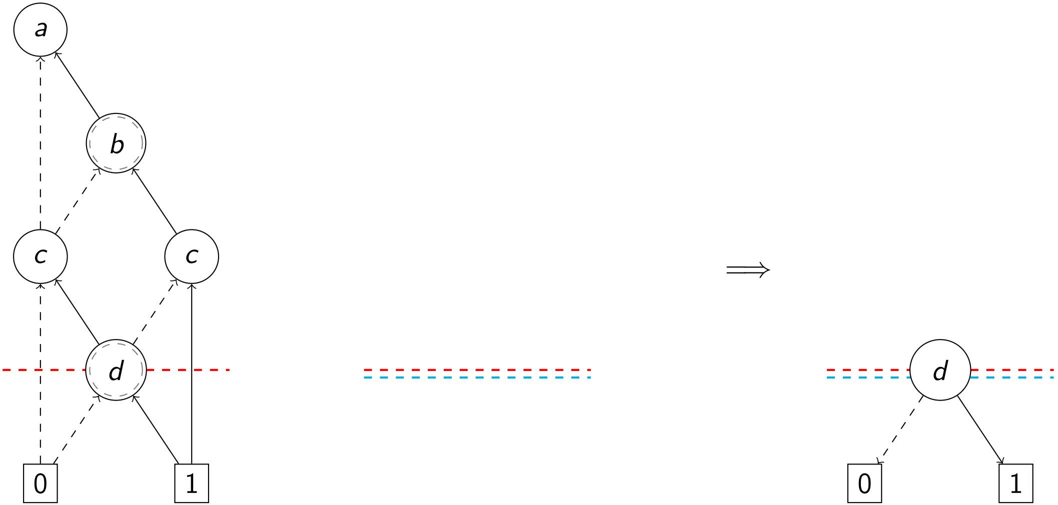
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



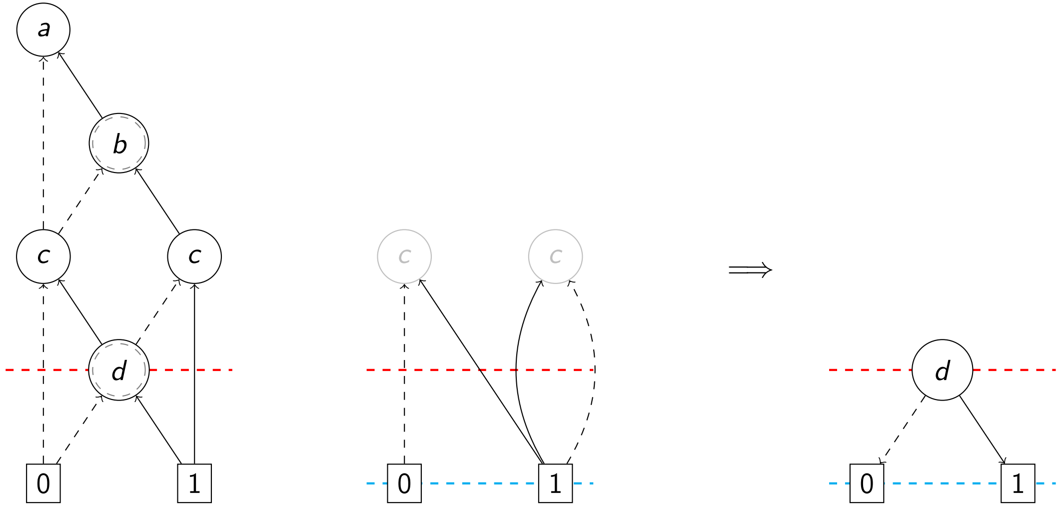
$\Longrightarrow$



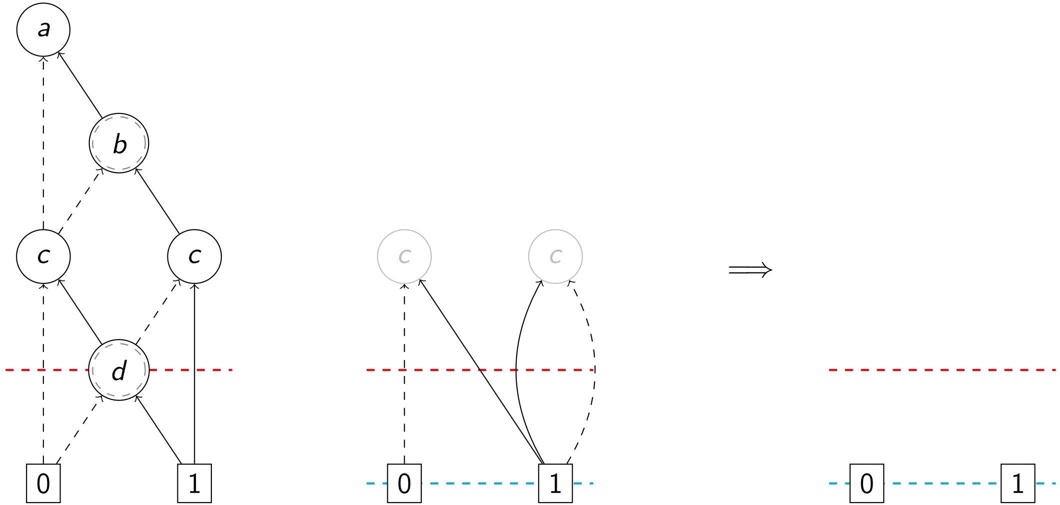
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



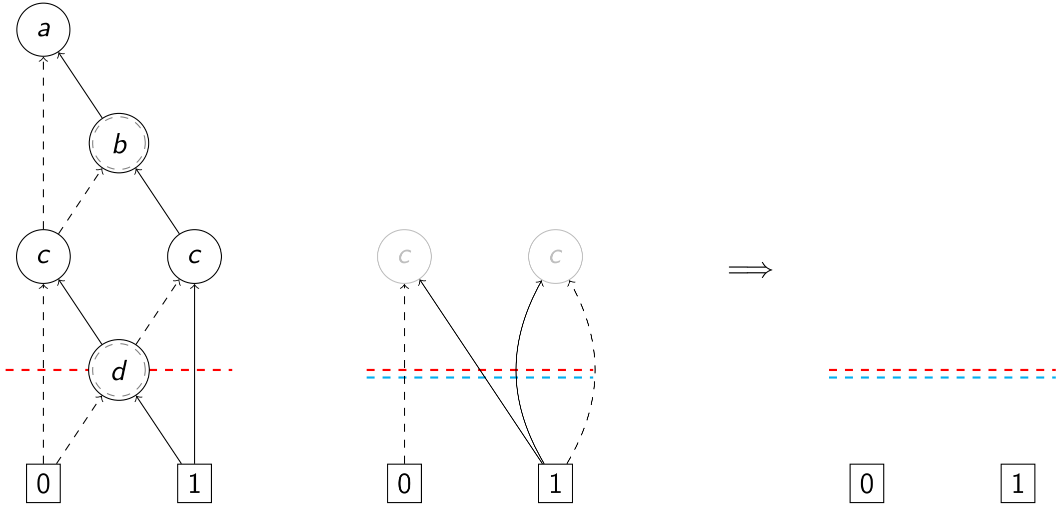
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



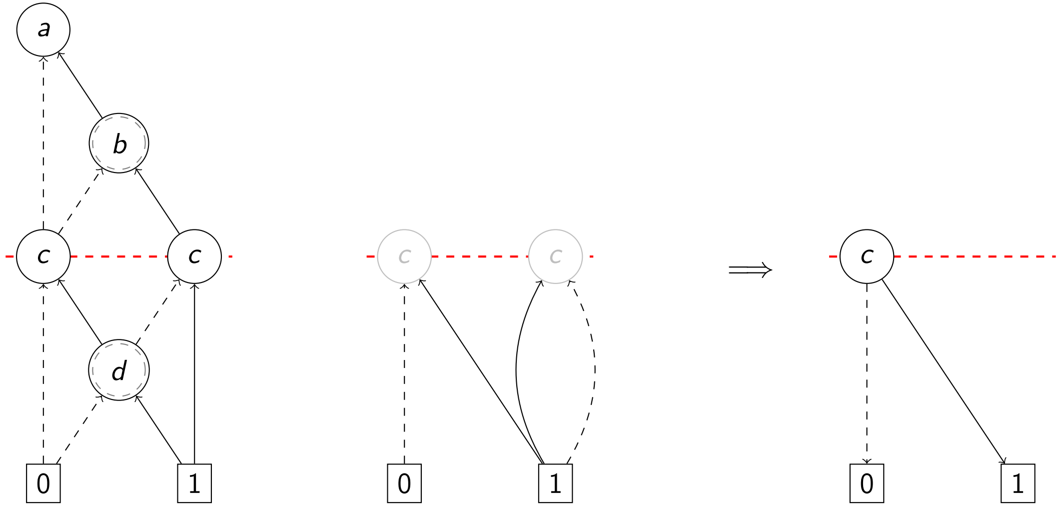
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



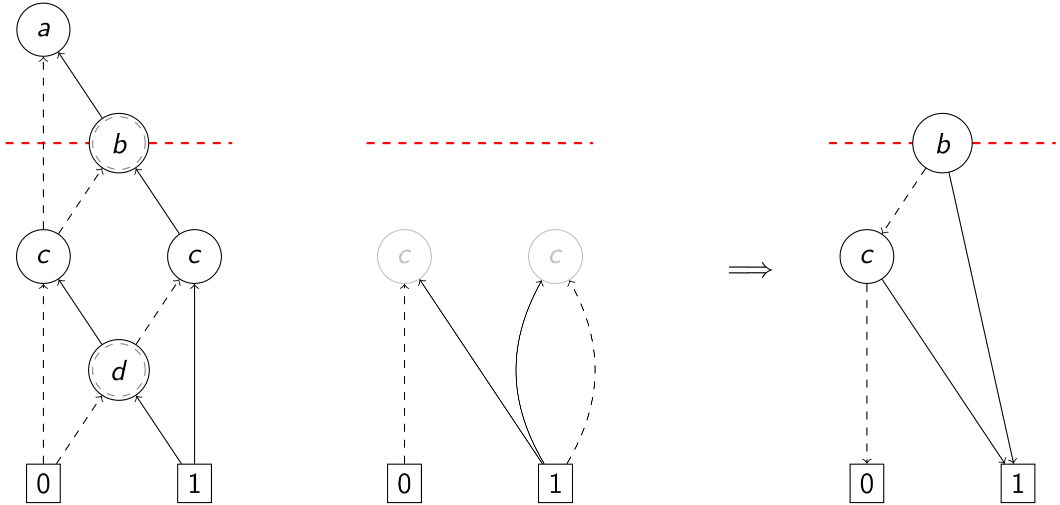
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



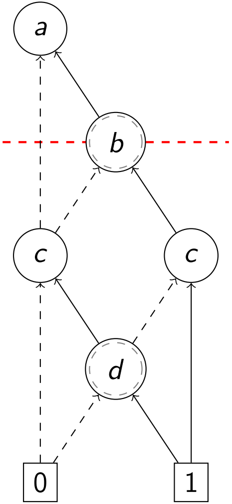
$\exists\vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



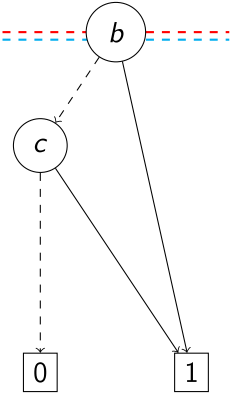
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



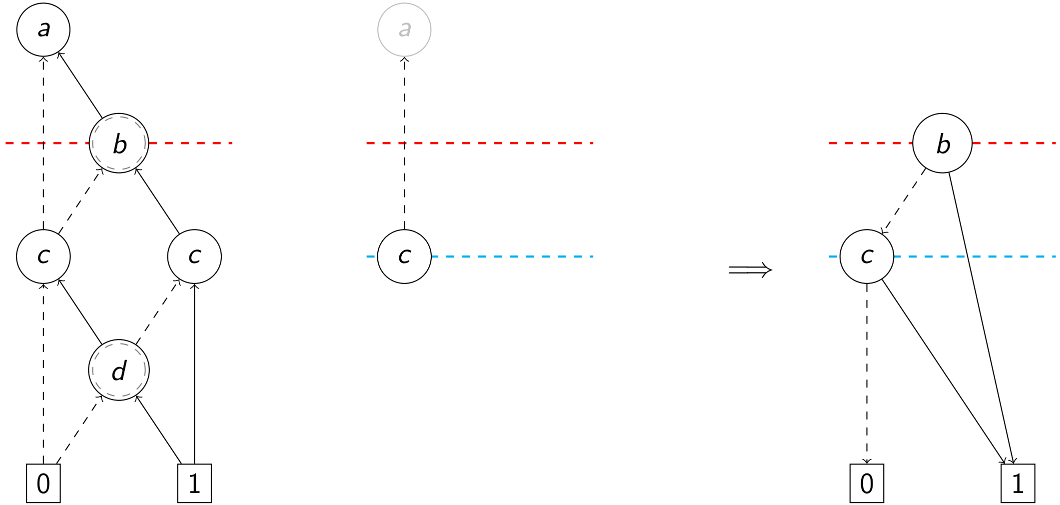
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



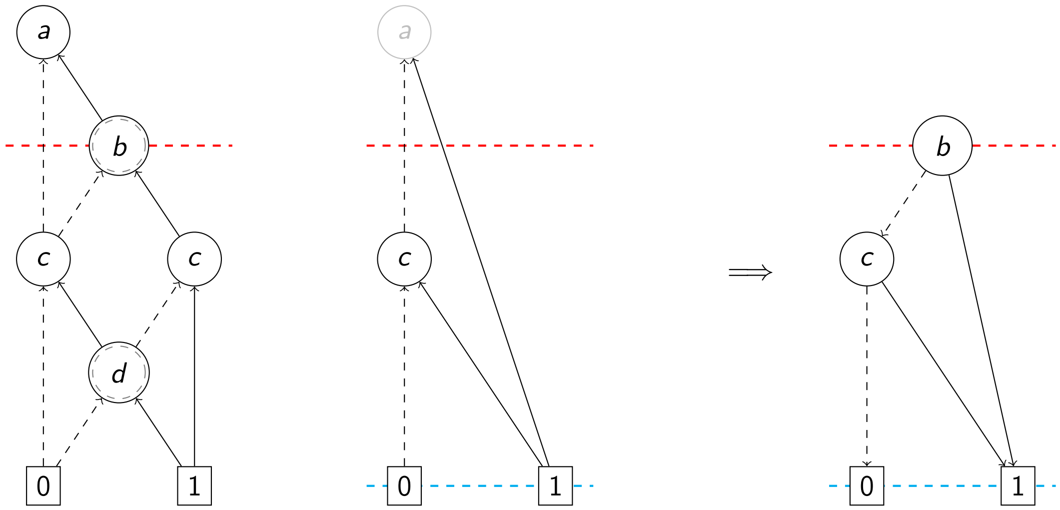
$\Longrightarrow$



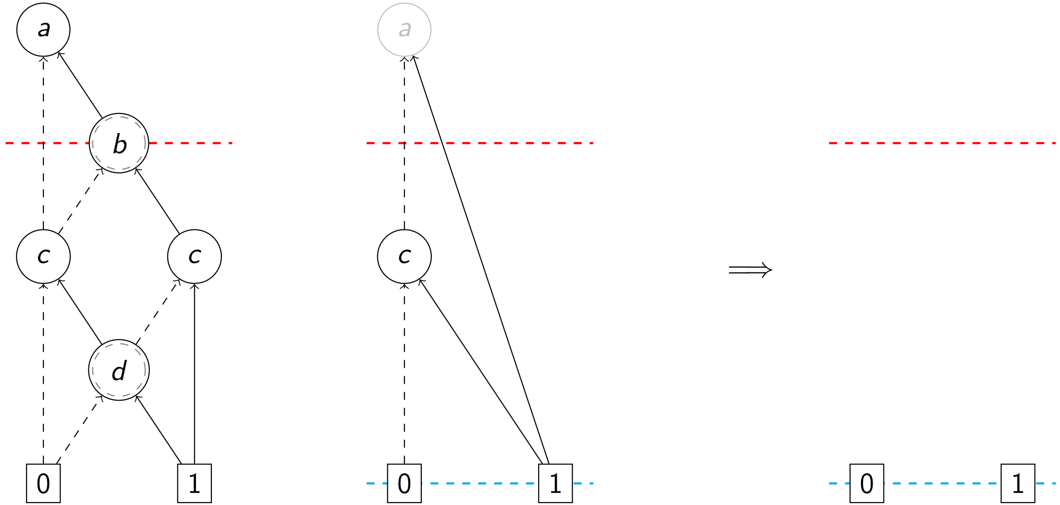
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



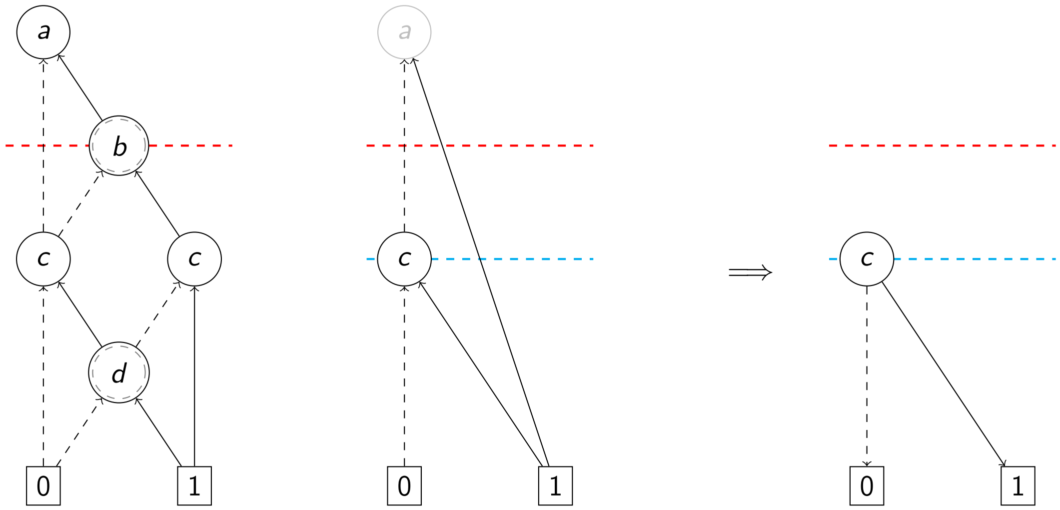
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



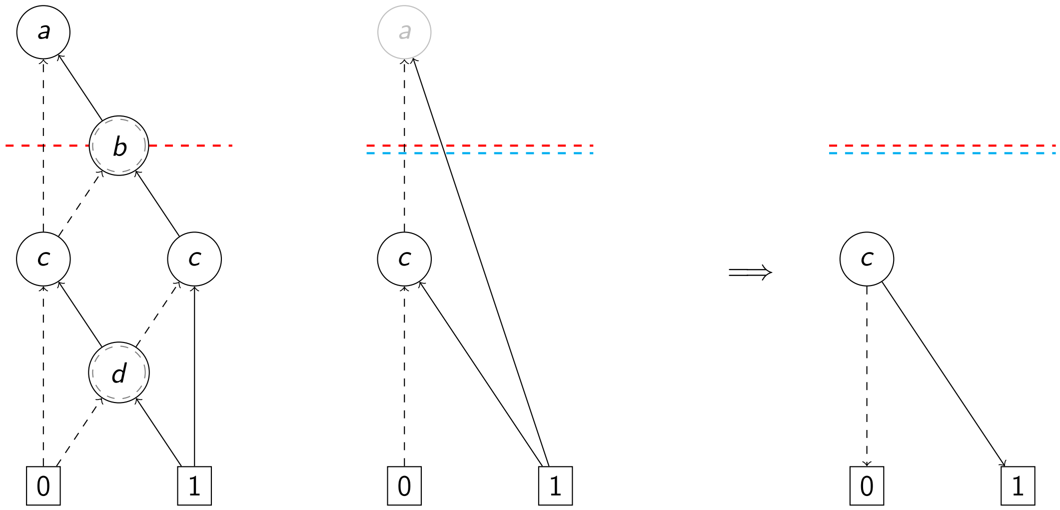
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



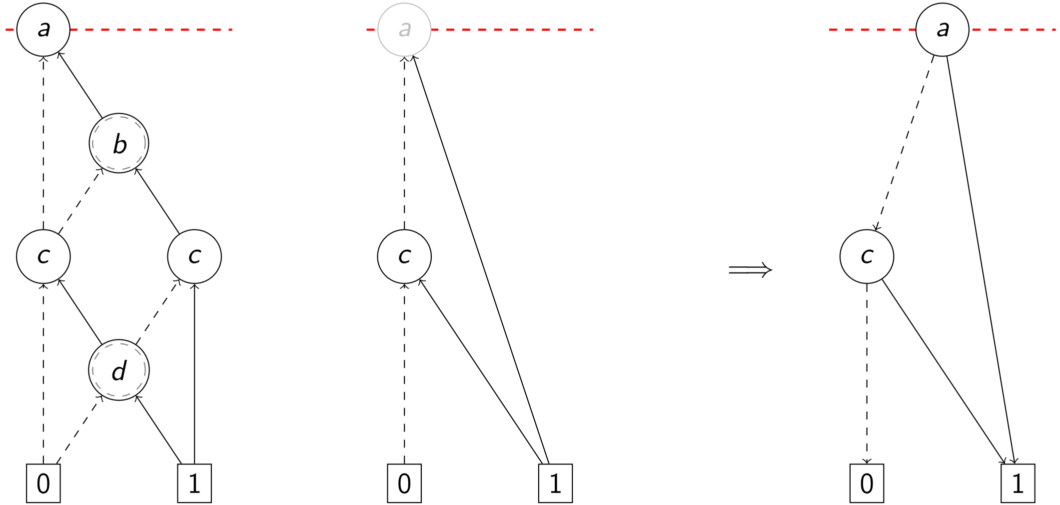
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



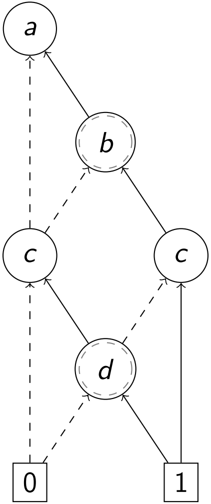
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



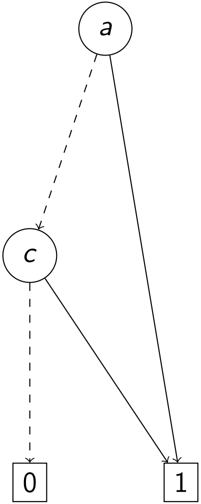
$\exists \vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



$\exists\vec{x} : \phi(\vec{x}) \qquad \vec{x} = \{b, d\}$



$\Longrightarrow$



Single Variable Quantification

$$\mathcal{O}(N^{2^k} \log(N^{2^k}))$$

Nested Sweeping

$$\mathcal{O}(N^{2^k} \log(N^{2^k}))$$

but $1.8\times$ *faster!*

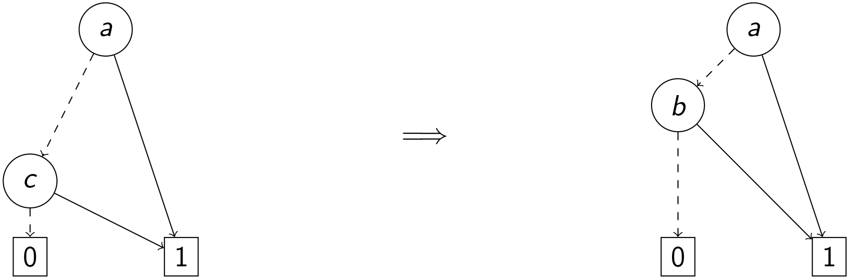
N: Input Size

$$[x \mapsto y]$$

$[x \mapsto y]$

Definition
A relabelling π is monotone if $x_i < x_j \implies \pi(x_i) < \pi(x_j)$.

Lemma
If π is monotone, then the BDD $f(\vec{x})$ is isomorphic to $f(\pi(\vec{x}))$.



$[x \mapsto y]$

Definition
A relabelling π is monotone if $x_i < x_j \implies \pi(x_i) < \pi(x_j)$.

Lemma
If π is monotone, then the BDD $f(\vec{x})$ is isomorphic to $f(\pi(\vec{x}))$.

- **One can apply π in a single linear scan.**
  $\mathcal{O}(N)$ time, $2 \cdot \text{scan}(N)$ I/Os, and N external space.
- **One can incorporate π into a (succeeding) top-down sweep.**
  $\mathcal{O}(N)$ time, 0 I/Os, and 0 external space.
- **One can incorporate π into a (preceeding) bottom-up sweep.**
  $\mathcal{O}(n)$ time, 0 I/Os, and 0 external space.

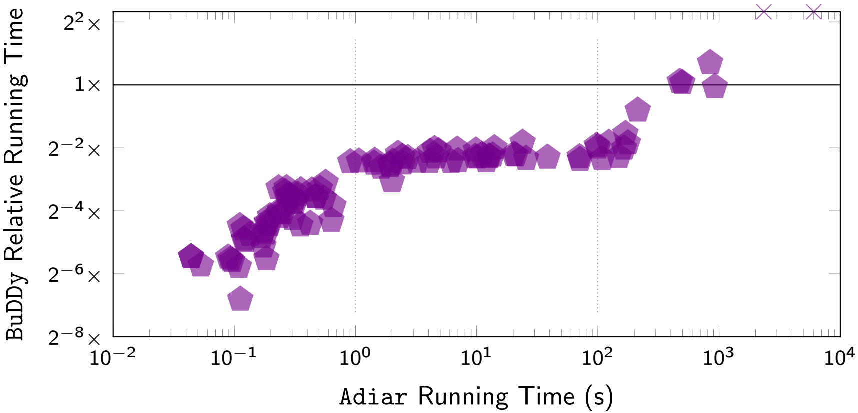
Adiar[1]

github.com/ssoelvsten/adiar

[1] adiar ⟨ portugese ⟩ (verb) : to defer, to postpone.

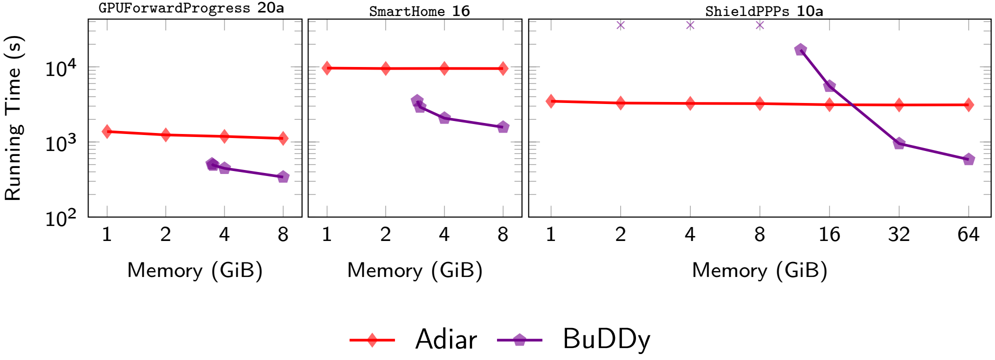
Quantified Boolean Formulæ (QBF) of 2-player games:

$$\exists \vec{x} \forall \vec{y} \dots \exists \vec{z} : \phi(\vec{x}, \vec{y}, \dots, \vec{z}) \stackrel{?}{=} 1$$



Relational Product in a Transition System:

$$RelProd(S_{\vec{x}}, T_{\vec{x},\vec{x}'}) \triangleq (\exists \vec{x} \,.\, S_{\vec{x}} \wedge T_{\vec{x},\vec{x}'})[\vec{x}' \mapsto \vec{x}]$$



Adiar

- github.com/ssoelvsten/adiar
- ssoelvsten.github.io/adiar
- MIT
- 3.462 unit tests

Adiar

github.com/ssoelvsten/adiar

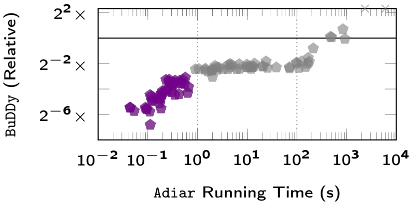
ssoelvsten.github.io/adiar

MIT

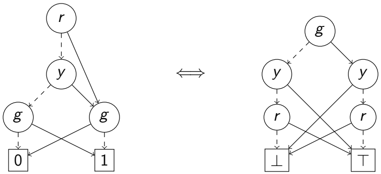
3.462 unit tests

Future Work

Small Instances



Variable Reordering



Manual Variable Reordering

Consider the `bdd_replace`(f, π) from `BuDDy` to compute $[x \mapsto \pi(x)]$ on an input with N BDD nodes and n variables and an output of T BDD nodes.

		Depth-first	Time-forwarding
Any π	Time & I/Os	$\mathcal{O}(N + n \cdot T)$	$\mathcal{O}(\text{sort}(T \cdot \sum_{i=1}^{n} C^{\emptyset}_{1:f[i]}))$
	Space		$\mathcal{O}(\text{sort}(T \cdot \max_i(C^{\emptyset}_{1:f[i]})))$
Exchange	Time & I/Os	$\mathcal{O}(N + T)$	$\mathcal{O}(\text{sort}(N + T))$
	Space		$\mathcal{O}(N + T)$
Adjacent Swap	Time & I/Os	$\mathcal{O}(N + T)$	$\mathcal{O}(\text{sort}(N + T))$
	Space		$\mathcal{O}(N + T)$

Dynamic Variable Reordering

We have surveyed current dynamic variable ordering methods to uncover how our I/O-efficient manual reordering algorithms can be applied.

- **Metaheuristics:** simulated annealing, genetic and memetic algorithms, and swarm intelligence algorithms, via *exchanges* and *adjacent swaps*[1].
- **Sifting:** Rudell's sifting algorithm, via repeated *exchanges*.
- **Parallel Sifting:** Rudell's procedure via repeated *adjacent swaps*; this is akin to the 2-window algorithm.

In terms of space, these I/O-efficient variants are on par with the depth-first approach.

---

[1] Or any *non-monotone* π if that does not break the memory limits.

- **Unique Identifier:**
  Sorting predicates can be turned into mere 64-bit integer comparison.
- **Levelised Priority Queue:**
  Defer sorting of level ℓ in the priority queue until level ℓ has to be processed.
- **Equality Checking:**
  If BDDs ϕ and ψ are created by `Reduce` then they are bit-wise equivalent iff $\phi \equiv \psi$.
- **Levelised Cuts:**
  The priority queue's size is at most the maximum cut in the BDD.
- **Levelised Random Access:**
  If a BDD's level fits into memory then random access can be used (in moderation).
- **Node Table:**
  If a BDD is small enough then compute on it with the conventional approach.

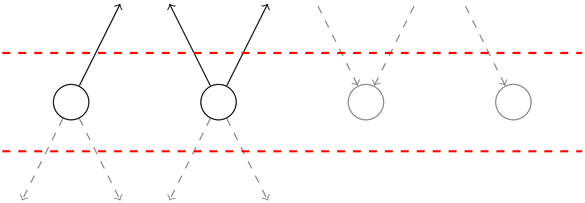
Unique Identifier

(a) `null`.

(b) `leaf` with value 0 or 1.

(c) `node` with label, `i`, identifier, `id`, and an out-degree index, `o`.

Levelised Priority Queue



Observation: When processing level i, no new requests for the same level are made.

Optimisation: Sort bucket of *all* requests for level i at once with Quicksort ($\sim 2\times$ faster than a priority queue).

	Improvement (%)
Queens (14)	25.3
Tic-Tac-Toe (22)	37.0

Equality Checking

$\mathcal{O}(\textbf{sort}(N^2))$: compute $f \leftrightarrow g$ and check whether it is the 1 BDD.

$\mathcal{O}(\textbf{sort}(N))$: Fail-fast during a product construction if more than $N_{f,i}$ ($N_{g,i}$) pairs of nodes are checked on level i.

$2 \cdot \textbf{scan}(N)$: Fail-fast during a linear scan of both BDDs bit-by-bit.

	Time (s)
$\mathcal{O}(\text{sort}(N^2))$	0.38
$\mathcal{O}(\text{sort}(N))$	0.058
$2 \cdot \text{scan}(N)$	0.006

Checking the (EPFL Benchmark) *voter* circuit's single output gate ($|N_f| = |N_g| = 5.76$ MiB).

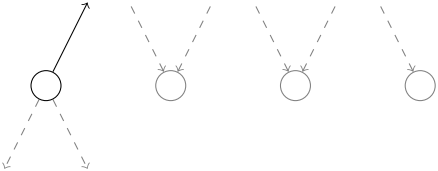
Levelised Cuts

Lemma
The maximum 2-level cut for BDD ϕ is at most $\frac{3}{2}$ larger than its maximum 1-level cut.

Theorem
Given maximum 2-level cuts size C_ϕ for BDD ϕ and C_ψ for BDD ψ, the maximum 2-level cut for the BDD $\phi \land \psi$ is less than or equal to $C_\phi \cdot C_\psi$.

		Overhead	Precision
1-level cut	:	1.0%	69.2%
2-level cut	:	3.3%	86.3%

Levelised Cuts

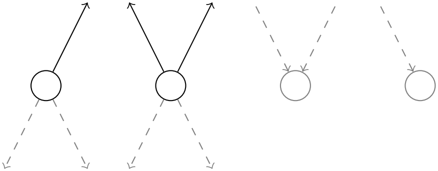


Lemma
The maximum 2-level cut for BDD ϕ is at most $\frac{3}{2}$ larger than its maximum 1-level cut.

Theorem
Given maximum 2-level cuts size C_ϕ for BDD ϕ and C_ψ for BDD ψ, the maximum 2-level cut for the BDD $\phi \land \psi$ is less than or equal to $C_\phi \cdot C_\psi$.

		+ Overhead	Precision
1-level cut	:	1.0%	69.2%
2-level cut	:	3.3%	86.3%

Levelised Cuts

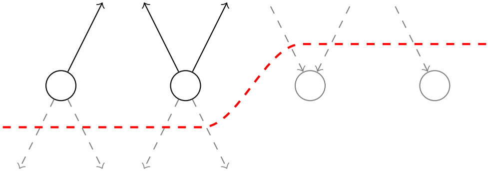


Lemma
The maximum 2-level cut for BDD ϕ is at most $\frac{3}{2}$ larger than its maximum 1-level cut.

Theorem
Given maximum 2-level cuts size C_ϕ for BDD ϕ and C_ψ for BDD ψ, the maximum 2-level cut for the BDD $\phi \land \psi$ is less than or equal to $C_\phi \cdot C_\psi$.

		+ Overhead	Precision
1-level cut	:	1.0%	69.2%
2-level cut	:	3.3%	86.3%

Levelised Cuts

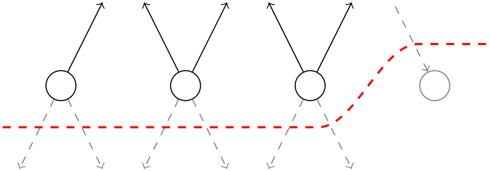


Lemma
The maximum 2-level cut for BDD ϕ is at most $\frac{3}{2}$ larger than its maximum 1-level cut.

Theorem
Given maximum 2-level cuts size C_ϕ for BDD ϕ and C_ψ for BDD ψ, the maximum 2-level cut for the BDD $\phi \land \psi$ is less than or equal to $C_\phi \cdot C_\psi$.

		+ Overhead	Precision
1-level cut	:	1.0%	69.2%
2-level cut	:	3.3%	86.3%

Levelised Cuts

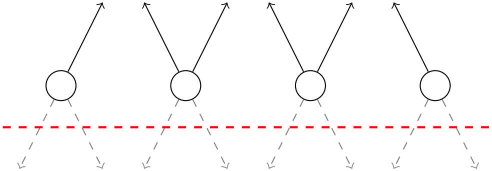


Lemma
The maximum 2-level cut for BDD ϕ is at most $\frac{3}{2}$ larger than its maximum 1-level cut.

Theorem
Given maximum 2-level cuts size C_ϕ for BDD ϕ and C_ψ for BDD ψ, the maximum 2-level cut for the BDD $\phi \wedge \psi$ is less than or equal to $C_\phi \cdot C_\psi$.

		+ Overhead	Precision
1-level cut	:	1.0%	69.2%
2-level cut	:	3.3%	86.3%

Levelised Cuts

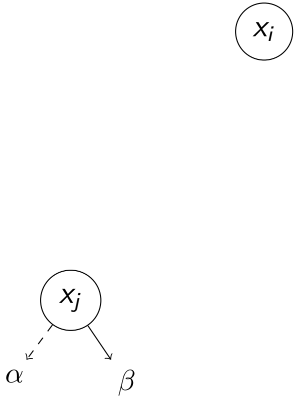


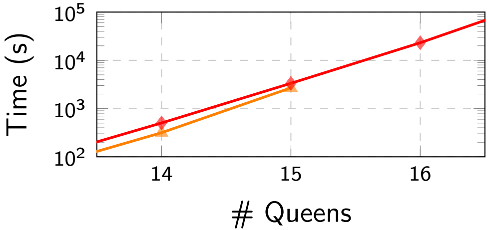
Theorem
Given maximum 2-level cuts size C_ϕ for BDD ϕ and C_ψ for BDD ψ, the maximum 2-level cut for the BDD $\phi \wedge \psi$ is less than or equal to $C_\phi \cdot C_\psi$.

Lemma
The maximum 2-level cut for BDD ϕ is at most $\frac{3}{2}$ larger than its maximum 1-level cut.

		+ Overhead	Precision
1-level cut	:	1.0%	69.2%
2-level cut	:	3.3%	86.3%

Levelised Random Access

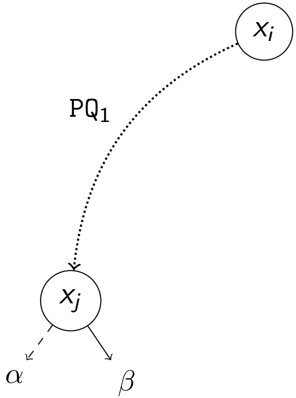


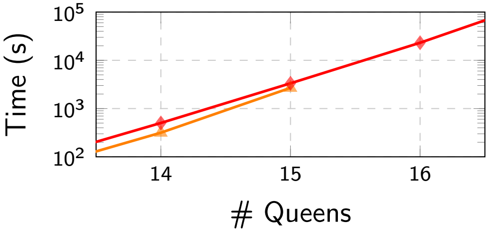


△	CUDD	v3.0	:	44.8 min
◇	Adiar	v1.0	:	66.7 min
	+ cuts		:	56.8 min

Counting solutions for the N-Queens Problem.

Levelised Random Access

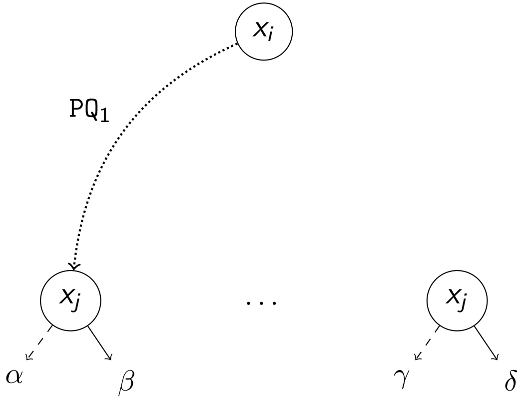


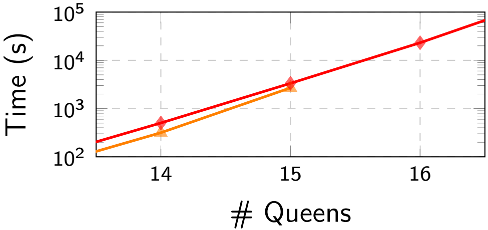


△	CUDD	v3.0	:	44.8 min
◇	Adiar	v1.0	:	66.7 min
	+ cuts		:	56.8 min

Counting solutions for the N-Queens Problem.

Levelised Random Access

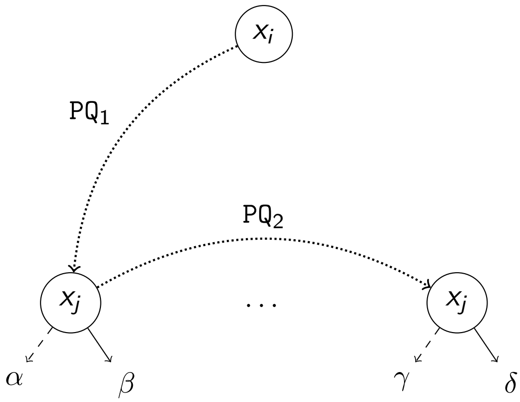


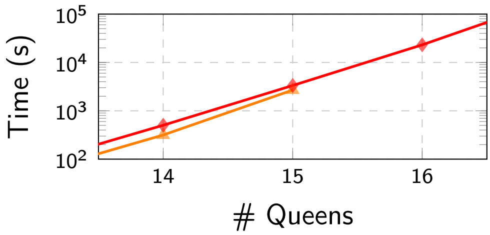


△	CUDD	v3.0	:	44.8 min
◇	Adiar	v1.0	:	66.7 min
	+ cuts		:	56.8 min

Counting solutions for the N-Queens Problem.

Levelised Random Access

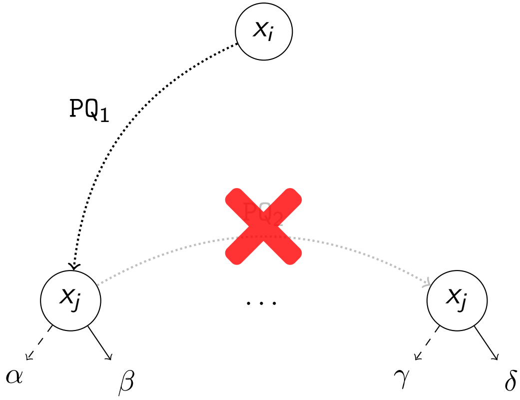


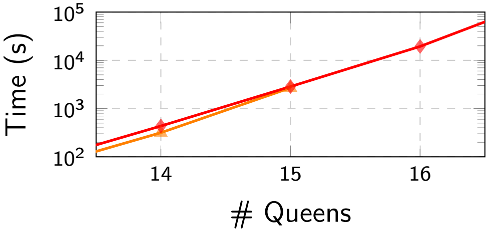


△	CUDD	v3.0	:	44.8 min
◇	Adiar	v1.0	:	66.7 min
	+ cuts		:	56.8 min

Counting solutions for the N-Queens Problem.

Levelised Random Access

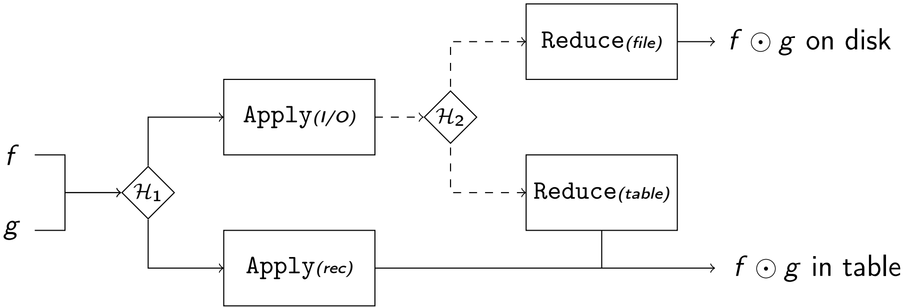




				⏱
△	CUDD	v3.0	:	44.8 min
◇	Adiar	v1.0	:	66.7 min
	+ cuts		:	56.8 min
	+ random access		:	47.2 min

Counting solutions for the N-Queens Problem.

Node Table



f
g
$\mathcal{H}_1$
Apply(I/O)
Apply(rec)
$\mathcal{H}_2$
Reduce(file)
Reduce(table)
$f \odot g$ on disk
$f \odot g$ in table