Constraints and Automata

Nicolas Beldiceanu

nicolas.beldiceanu@mines-nantes.fr

ACP summer school, Cork, June 2016

Reminder: NFA
(non-deterministic finite automaton)

- A NFA is defined by:
  - A finite set of **states** Q
  - An **alphabet** Σ (*set of input symbols*)
  - A **transition function** Δ: $Q \times \Sigma \rightarrow$ power set of Q
  - An **initial state** $q_{\text{init}} \in Q$
  - A set of **accepting states** $F \subseteq Q$

The transition function Δ is allowed to be partial

Reminder: NFA
(non-deterministic finite automaton)

- Given a word $w = w_1\ w_2 \ldots w_n$ over the alphabet Σ an and NFA ω
  ***w* is accepted by ω**

  iff ∃ $q_0\ q_1 \ldots q_n$ in Q such: ***there is a sequence of states***
  - $q_0 = q_{\text{init}}$ ***starting at q_{init}***
  - $q_{i+1} = \Delta(q_i, w_{i+1})$ with $i \in [0, n\text{-}1]$ ***compatible with the transition function***
  - $q_n \in F$ ***ending in an accepting state***

Reminder: DFA
(deterministic finite automaton)

- A DFA is defined by:
  - A finite set of **states** Q
  - An **alphabet** Σ (*set of input symbols*)
  - A **transition function** δ: $Q \times \Sigma \rightarrow Q$
  - An **initial state** $q_{\text{init}} \in Q$
  - A set of **accepting states** $F \subseteq Q$

The transition function δ is allowed to be partial

Reminder: DFA
(deterministic finite automaton)

- Given a word $w = w_1\ w_2 \ldots w_n$ over the alphabet Σ an and NFA ω
  ***w* is accepted by ω**

  iff $\exists\ q_0\ q_1 \ldots q_n$ in Q such: ***there is a sequence of states***
  - $q_0 = q_{\text{init}}$ ***starting at q_{init}***
  - $q_{i+1} = \delta(q_i, w_{i+1})$ with $i \in [0,n\text{-}1]$ ***compatible with the transition function***
  - $q_n \in F$ ***ending in an accepting state***

The initial general idea

- Use finite automata as a general way for describing constraints

based on the **one to one correspondence** between

solutions of a constraint

and

words accepted by a finite automaton

Implicit assumption: both the automaton and the constraint
are use a sequence of same length

The main difference

- Use finite automata as a general way for describing constraints
  based on a one to one correspondence between
  solutions of a constraint
  and
  words accepted by a finite automaton

- **Go**

  from checking whether a **sequence of fixed letters** is accepted by an automaton or not

  to checking whether a **sequence of variables** has at least one assignment accepted by an automaton or not

Remark 1

- Deal with **finite sequences**
  since
  post constraints on finite sequences of variables

  You may say that everything is easy with finite sequences

Remark 2

- In the context of Constraint Programming

  no need for determinizing a non-deterministic automaton

may lead to use smaller automata

Remark 3

- Modelling constraints with automata
  is **independent from the solving technology**

⇒

- Can use **different solving techniques** like CP, LP, LS

  but even more important

- Can **develop concept/theory** which will be
  useful for more than one technology
  (e.g. see later one glue matrix)

Remark 4

- Using automata has a **compositional** flavor since:
  - **conjunction** of constraints : **product** of automata
  - **disjunction** of constraints : **union** of automata
  - **negation** of constraint : **complement** of automata
  - **reification** of constraint : **combination** of automata

Warning: limitations

- **Expressivity** limitation
  - Restrict ourselves to constraints that can be checked by **scanning once** through their variables (*e.g. no DFS*),
  - The size of the automaton has to be **bounded** by a polynomial of the number of variables (*e.g. not applicable for alldifferent*)

- **Operational** limitation
  - **For some constraints** for which there exists a specialized algorithm achieving GAC **we don't achieve GAC**

Map (precursors)

- **Constraint networks**
  - **N. R. Vempaty** [**AAAI-92**]
    Solving constraint satisfaction problems using finite automata
  - **J. Amilhastre** [PhD-99, Montpellier]
    Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes
    (in the context of configuration)

- **Arithmetic constraints**
  - **B. Boigelot**, **P. Wolper** [**ICLP-02**]
    Representing arithmetic constraints with finite automata: an overview

Map (early work)

- **Global constraints**
  - **G. Pesant** [workshop **CP-03**] [**CP-04**]
    A regular language membership constraint for finite sequences of variables (**regular** constraint)
  - **M. Carlsson**, **N. Beldiceanu**
    Revisiting the Lexicographic Ordering Constraint [**TechReport-02**]
    From constraints to finite automata to filtering algorithms [**ESOP-04**]
    Deriving Filtering Algorithms from Constraint Checkers [**CP-04**]
    (***automaton with accumulators*** *constraint*)

Map (follow up)

- **Global constraints with cost**
  - **S. Demassey**, **G. Pesant**, **L.-M. Rousseau** [CPAIOR-05]
    Constraint Programming Based Column Generation for Employee Timetable (**cost-regular** constraint, **one single** criteria)
  - **J. Menana**, **S. Demassey** [CPAIOR-09]
    *Sequencing and Counting with the **multicost-regular** Constraint* (**more than one** criteria)

Map (follow up)

- **Reformulation to Linear Programming**
  - **M.-C. Côté**, **B. Gendron**, **L.-M. Rousseau** [**CPAIOR-07**]
    Modeling the Regular Constraint with Integer Programming
    (**regular** constraint)
  - **E. Arafailova**, **N. Beldiceanu**, **R. Douence**, **P. Flener**,
    M. A. F. Rodriguez, **J. Pearson**, **H. Simonis** [**CPAIOR-16**]
    Time-Series Constraints: Improvements and Application in CP and MIP Contexts (**automaton with accumulators** constraint)

Advice for creating an automaton

- Automata without accumulator
  - Steps for creating an automaton
  - Examples
- Automata with accumulators (*see later on with the help of transducers*)

Steps for creating an automaton

- Identify the **input alphabet**
  (*most the time easy, but sometimes tricky*)

- Identify **all states**
  – Find the meaningful points wrt what we want to modelize
    (*the most difficult part*)
  – Have a systematic method for generating all states

- Add **transitions**
  (*easy if all states were identified correctly*)

But don't try to define an automaton
before having a clear view of all its states

Exercise 1 (odd numbers)

Construct and automaton that only accepts **binary odd numbers** (e.g. 1, 001, 101)

Exercise 1 (odd numbers)

Construct and automaton that only accepts **binary odd numbers** (e.g. 1, 001, 101)

Input alphabet {**0**,**1**}

Observation a binary odd number **finishes with a 1**, consequently remember last letter.

Set of states $\mathbf{s_0}$: if last letter was a 0 (*initial, non accepting*)
$\mathbf{s_1}$: if last letter was a 1 (*accepting*)

Exercise 2 (getting the states)

Construct an automaton that only accepts binary numbers that have an **even number of 0** and an **even number of 1** (e.g. 11, 0110, 00).

Construct an automaton that only accepts binary numbers that have an **even number of 0** and an **even number of 1** (e.g. 11, 0110, 00).

Input alphabet {**0**,**1**}

Observation need to remember if we encountered
an **even/odd number of 0/1**,
so need two counters:
- one to 0 if even number of 0, to 1 otherwise
- one to 0 if even number of 1, to 1 otherwise
make the cartesian product of the values of these two counters

Set of states

s_{00}: even number of 0, even number of 1
s_{01}: even number of 0, odd number of 1
s_{10}: odd number of 0, even number of 1
s_{11}: odd number of 0, odd number of 1

Exercise 3 (lets count)

Construct an automaton that only accepts binary numbers with **at most two consecutive 1** (e.g. 00, 0110001011).

Exercise 3 (lets count)

Construct an automaton that only accepts binary numbers with **at most two consecutive 1** (e.g. 00, 0110001011).

Input alphabet {**0**,**1**}

Observation since at most two consecutive 1 we have to **count number of consecutive 1**. since cannot exceed two consecutive 1, **count only up to 2**

Set of states

$\mathbf{s_0}$: the last suffix is 0
$\mathbf{s_1}$: the last suffix is 01
$\mathbf{s_2}$: the last suffix is 011

Exercise 4 (knowing where to go back)

Construct an automaton that only accepts words of the form **a (bb)* bc** .

Exercise 4 (knowing where to go back)

Construct an automaton that only accepts words of the form **a (bb)* bc** .

Input alphabet {**a**,**b**,**c**}

Observation enumerate the different prefixes of the word to recognize (except the word itself)

Set of states

s_{ε} : Recognize ε

$s_{a(bb)^*}$: Recognize **a followed by an even number of b**

$s_{a(bb)^*b}$: Recognize **a followed by an odd number of b**

Exercise 5 (limiting back-arcs)

Construct an automaton that only accepts words **finishing with 1101**.

Exercise 5 (no back-arcs)

Construct an automaton that only accepts words **finishing with 1101**.

Input alphabet {**0**,**1**}

Observation enumerate the different prefixes of the suffix to recognize (except the suffix itself)
use non-determinism to limit back-arcs

Set of states
s_ε : Recognize ε
s_1 : Recognize 1
s_{11} : Recognize 11
s_{110}: Recognize 110

Exercise 6 (representing a function)

Construct an automaton that only accepts words $x_1\ x_2\ \ldots\ x_n$ (n>1) such that $\boldsymbol{x_n=\min(x_1,x_2,\ldots,x_{n-1})}$, assuming $\boldsymbol{x_i \in [1,4]}$.

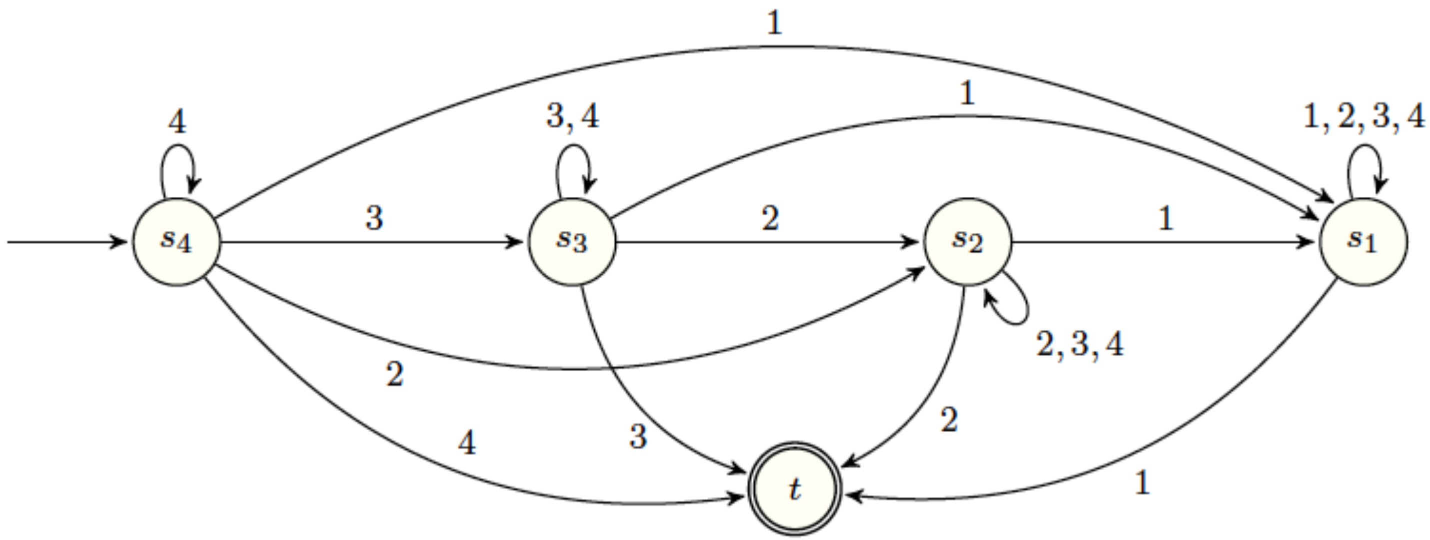
Exercise 6 (representing a function)

Construct an automaton that only accepts words $x_1\ x_2\ \dots\ x_n$ (n>1) such that $\boldsymbol{x_n}$**=min(**$\boldsymbol{x_1, x_2, \dots, x_{n-1}}$**)**, assuming $\boldsymbol{x_i \in [1,4]}$.

Input alphabet	{**1**,**2**,**3**,**4**}
Observation	for each potential value of min(x_1,x_2,…,x_i) ($i \in [1,n\text{-}1]$) we have a state + one accepting state; use **non-determinism** to get the result
Set of states	$\mathbf{s_1}$: the value of min(x_1,x_2,…,x_i) is 1 $\mathbf{s_2}$: the value of min(x_1,x_2,…,x_i) is 2 $\mathbf{s_3}$: the value of min(x_1,x_2,…,x_i) is 3 $\mathbf{s_4}$: the value of min(x_1,x_2,…,x_i) is 4 **t** : the only accepting state

Exercise 6 (representing a function)

Construct an automaton that only accepts words $x_1\ x_2\ \ldots\ x_n$ (n>1) such that $\mathbf{x_n=\min(x_1,x_2,\ldots,x_{n-1})}$, assuming $\mathbf{x_i \in [1,4]}$.



Part related to $x_1\ x_2\ \ldots\ x_{n-1}$

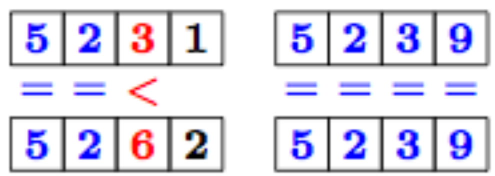
Part related to x_n

Can use the same construction for any function

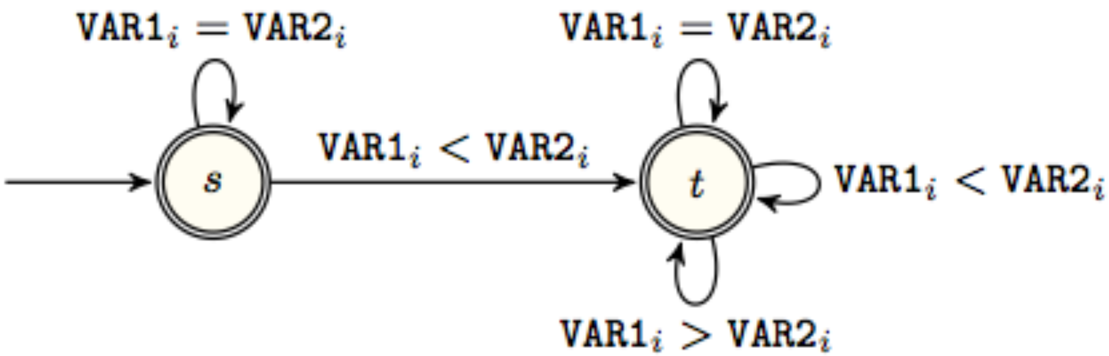
Exercise 7 (lexicographic ordering)

Construct an automaton for the constraint lex_lesseq($VECTOR_1$,$VECTOR_2$)

($VECTOR_1$, $VECTOR_2$ are two lists of variables of same length such that $VECTOR_1$ is lexicographically less than or equal to $VECTOR_2$.



Exercise 7 (lexicographic ordering)



$$(\mathtt{VAR1}_i < \mathtt{VAR2}_i \Leftrightarrow S_i = 1) \wedge (\mathtt{VAR1}_i = \mathtt{VAR2}_i \Leftrightarrow S_i = 2) \wedge (\mathtt{VAR1}_i > \mathtt{VAR2}_i \Leftrightarrow S_i = 3)$$

Lexicographic ordering example: breaking symmetry for a reversible sequence

$\langle x_1, x_2, x_3, x_4 \rangle \in [0,1]$ (no symmetry breaking) 16 solutions	$\langle x_1, x_2, x_3, x_4 \rangle \in [0,1]$ $x_1 \leq x_4$ $16 - \frac{16}{4} = 12$ solutions	$\langle x_1, x_2, x_3, x_4 \rangle \in [0,1]$ LEX_LESSEQ$(\langle x_1, x_2 \rangle, \langle x_4, x_3 \rangle)$ $16 - \frac{16-4}{2} = 10$ solutions
$\langle 0,0,0,0 \rangle$	$\langle 0,0,0,0 \rangle$	$\langle 0{,}0\ ,\ 0{,}0 \rangle$
$\langle 0,0,0,1 \rangle$	$\langle 0,0,0,1 \rangle$	$\langle 0{,}0\ ,\ 0{,}1 \rangle$
$\langle 0,0,1,0 \rangle$	$\langle 0,0,1,0 \rangle$	$\langle 0{,}0\ ,\ 1{,}0 \rangle$
$\langle 0,0,1,1 \rangle$	$\langle 0,0,1,1 \rangle$	$\langle 0{,}0\ ,\ 1{,}1 \rangle$
$\langle 0,1,0,0 \rangle$	$\langle 0,1,0,0 \rangle$	
$\langle 0,1,0,1 \rangle$	$\langle 0,1,0,1 \rangle$	$\langle 0{,}1\ ,\ 0{,}1 \rangle$
$\langle 0,1,1,0 \rangle$	$\langle 0,1,1,0 \rangle$	$\langle 0{,}1\ ,\ 1{,}0 \rangle$
$\langle 0,1,1,1 \rangle$	$\langle 0,1,1,1 \rangle$	$\langle 0{,}1\ ,\ 1{,}1 \rangle$
$\langle 1,0,0,0 \rangle$		
$\langle 1,0,0,1 \rangle$	$\langle 1,0,0,1 \rangle$	$\langle 1{,}0\ ,\ 0{,}1 \rangle$
$\langle 1,0,1,0 \rangle$		
$\langle 1,0,1,1 \rangle$	$\langle 1,0,1,1 \rangle$	$\langle 1{,}0\ ,\ 1{,}1 \rangle$
$\langle 1,1,0,0 \rangle$		
$\langle 1,1,0,1 \rangle$	$\langle 1,1,0,1 \rangle$	
$\langle 1,1,1,0 \rangle$		
$\langle 1,1,1,1 \rangle$	$\langle 1,1,1,1 \rangle$	$\langle 1{,}1\ ,\ 1{,}1 \rangle$

Exercise 8 (value ordering)

INT_VALUE_PRECEDE_CHAIN(**VALUES**, **VARIABLES**)

VALUES	:	collection(var−int)
VARIABLES	:	collection(var−dvar)

Assuming n denotes the number of items of the **VALUES** collection, the following condition holds for every $i \in [1, n-1]$: When it is defined, the first occurrence of the $(i+1)^{th}$ value of the **VALUES** collection should be preceded by the first occurrence of the i^{th} value of the **VALUES** collection.

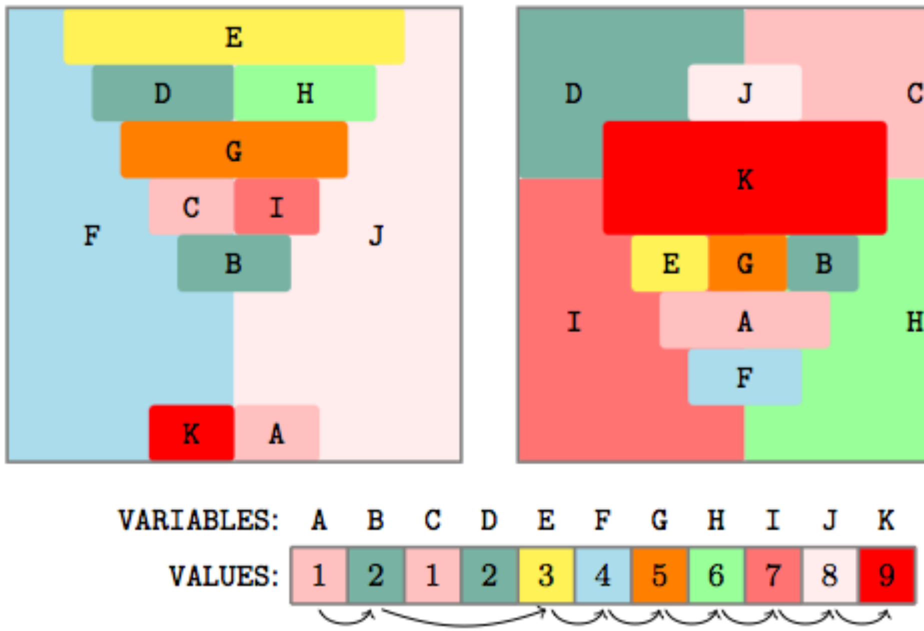
Exercise 8 (value ordering)

INT_VALUE_PRECEDE_CHAIN(**VALUES**, **VARIABLES**)

VALUES	:	collection(**var**−int)
VARIABLES	:	collection(**var**−dvar)

Assuming n denotes the number of items of the **VALUES** collection, the following condition holds for every $i \in [1, n-1]$: When it is defined, the first occurrence of the $(i+1)^{th}$ value of the **VALUES** collection should be preceded by the first occurrence of the i^{th} value of the **VALUES** collection.

INT_VALUE_PRECEDE_CHAIN(
[1,2,3,4,5,6,7,8,9],
[1,2,1,2,3,4,5,6,7,8,9])



Exercise 8 (value ordering)

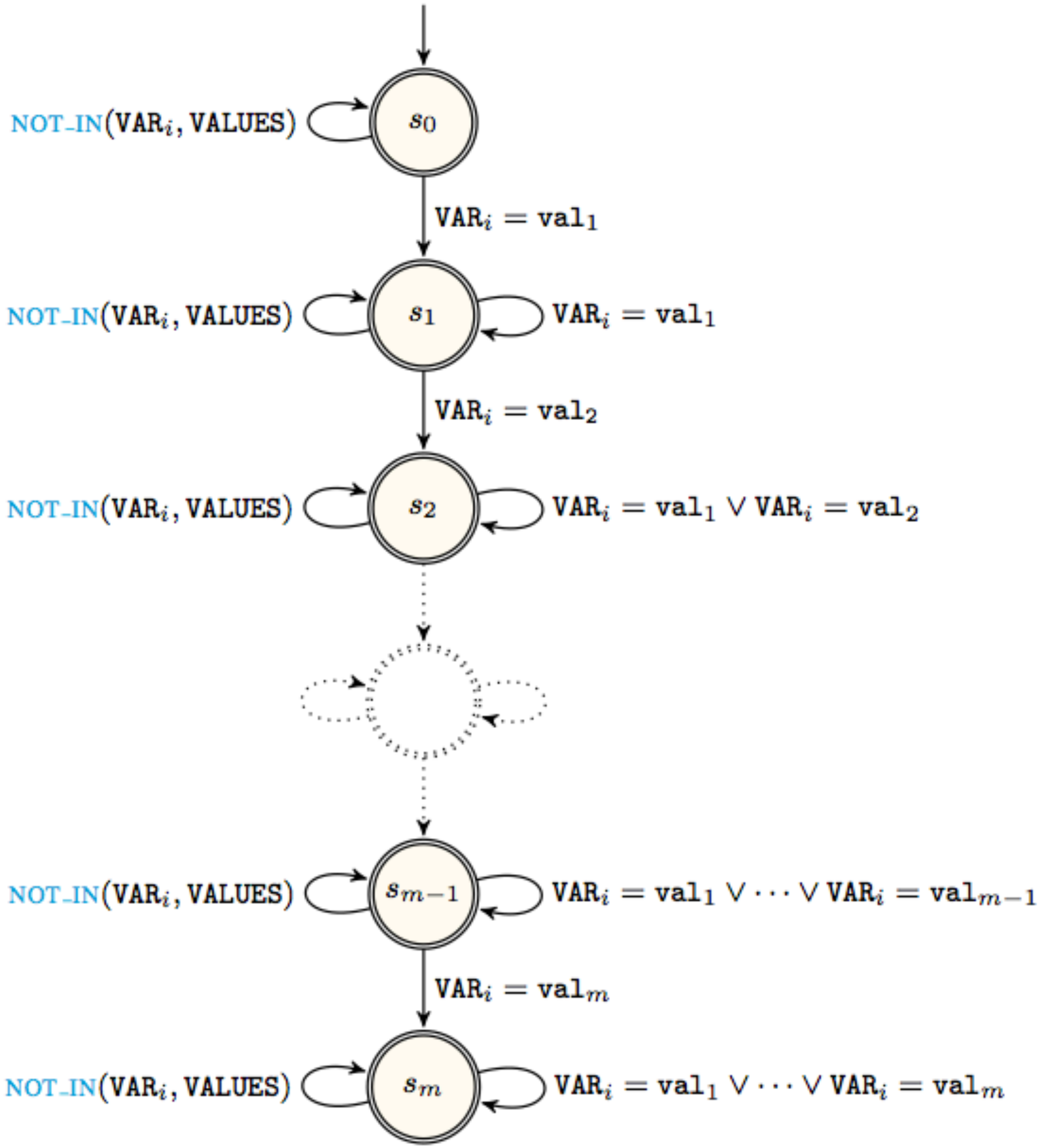
STATE s_i :

- each value val_1,val_2,…,val_i was already encountered at least once
- value val_{i+1} was not yet encountered

Exercise 8 (value ordering)

STATE s_i :

- each value $val_1, val_2, \ldots, val_i$ was already encountered at least once
- value val_{i+1} was not yet encountered



Constructing an automaton
(*enumerating the states*)

Quite often one can obtain the states by making the **cartesian product of several accumulator values**

You get a polynomial or pseudo-polynomial number of states, not very useful in practice, but:

- Can be used to show that GAC can be achieved in polynomial time …
- Can be used to test the effect of having a GAC filtering algorithm without implementing a dedicated filtering algorithm

States as the cartesian product of accumulators : example 1

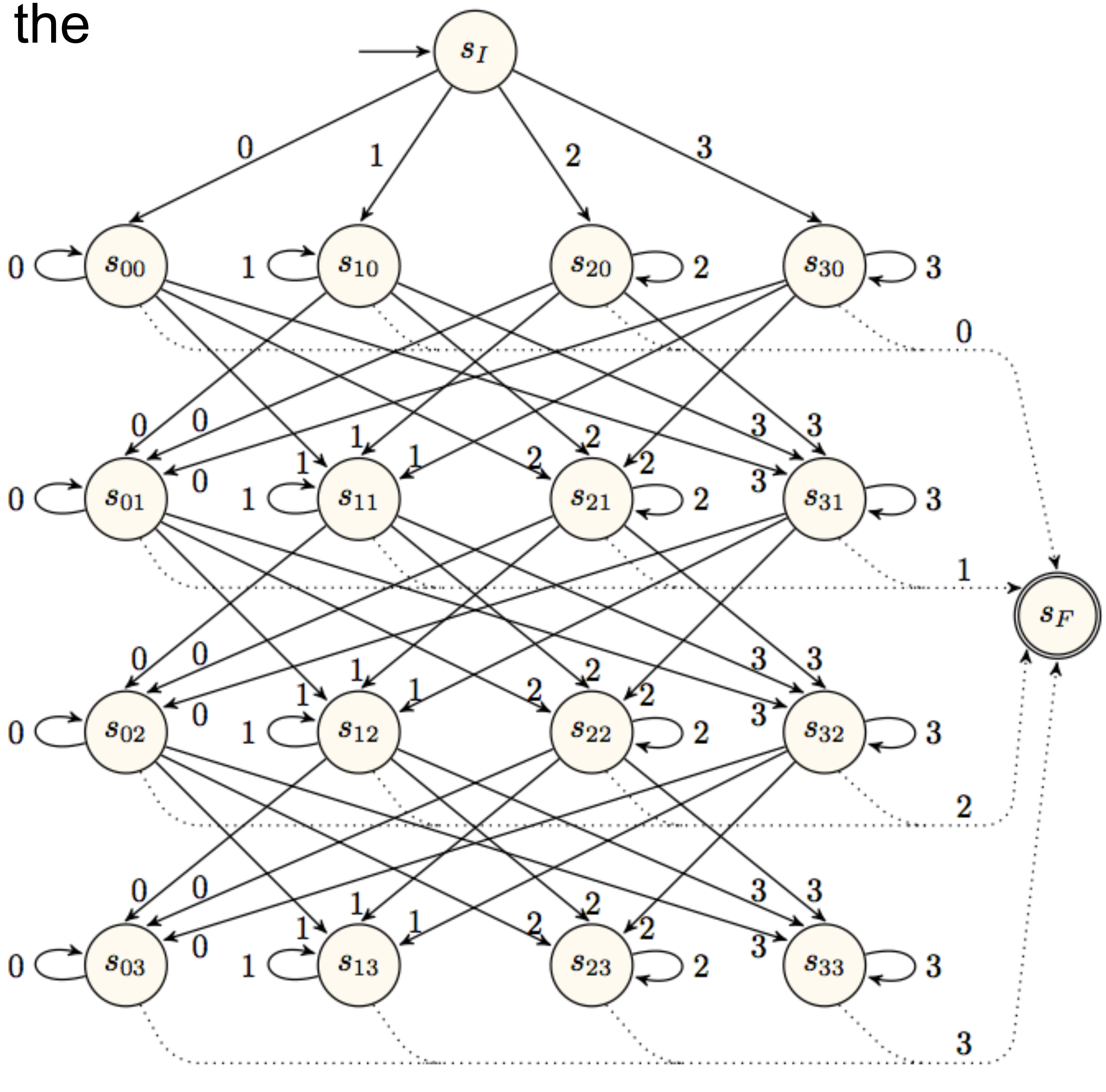
change(NCHANGE, [V_1,V_2,…,V_n])
NCHANGE is the number of times that $V_i \neq V_{i+1}$ holds

Enumerating the states of the automaton of the change constraint:

Use two accumulators:

- i: last value that was encountered
- j: number of already encountered constraints that hold of the form $V_i \neq V_{i+1}$

- i: last value that was encountered
- j: number of already encountered constraints that hold of the form $V_i \neq V_{i+1}$



Assume variables are in [0,3]

Other examples of constraints for which the states is the cartesian product of several accumulators

increasing_nvalue (increasing + nvalue)
(*number of distinct values already encountered, last encountered value*)

stretch_path (restrict the min/max length of maximum sequences of identical values)
(*value of a strech, occurrence in the stretch*)

Context underlying the creation of automata constraints

- **Application**
  (*expressing* ***regulation rules*** *for time table*)

- Automata with accumulator as a **programming language**
  (*writing* ***compact checkers***)

- Automata with **cost matrix**
  (***optimisation***)

- **Theory**
  (*go along the* ***Chomsky hierarchy****: from regular to grammar*)

Availability of automata constraints

- regular: **most CP systems** (*e.g. CHOCO, gecode, SICStus, MiniZinc*)
- cost regular, multi cost regular: CHOCO
- automata with accumulators: SICStus, SWI (*with the same syntax*)

and **all of them** can be reformulated in **linear programming**

but have to write a program in a specific language for generating the automaton

Context underlying the creation of automata constraints (our focus)

- **Application**
  (*expressing* ***regulation rules*** *for time table*)
- Automata with accumulator as a **programming language**
  (*writing* ***compact checkers***)
- Automata with **cost matrix**
  (***optimisation***)
- **Theory**
  (*go along the* ***Chomsky hierarchy****: from regular to grammar*)

Context

- Providing **efficient filtering algorithms** is challenging since:
  - There are a lot of global constraints
  - Filtering algorithms are far from obvious
  - Easy to introduce errors or to forget cases

- Want to **systematically** derive **correct** filtering algorithms from **first principle** avoiding creativity

As a first principle select a
constraint checker for the ground case

Automata with accumulators

- A **model** of automaton (with accumulators) for writing compact constraint checkers
- A **reformulation** of an automaton as a conjunction of signature and transition constraints
- A partial **characterization** of conditions for obtaining **generalized arc-consistency** for such constraint

Automata with accumulators

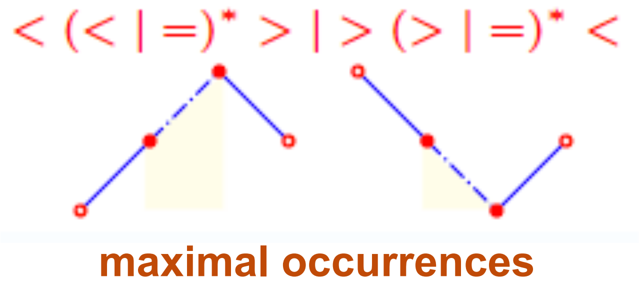
- A **model** of automaton (with accumulators) for writing compact constraint checkers
- A **reformulation** of an automaton as a conjunction of signature and transition constraints
- A partial **characterization** of conditions for obtaining **generalized arc-consistency** for such constraint

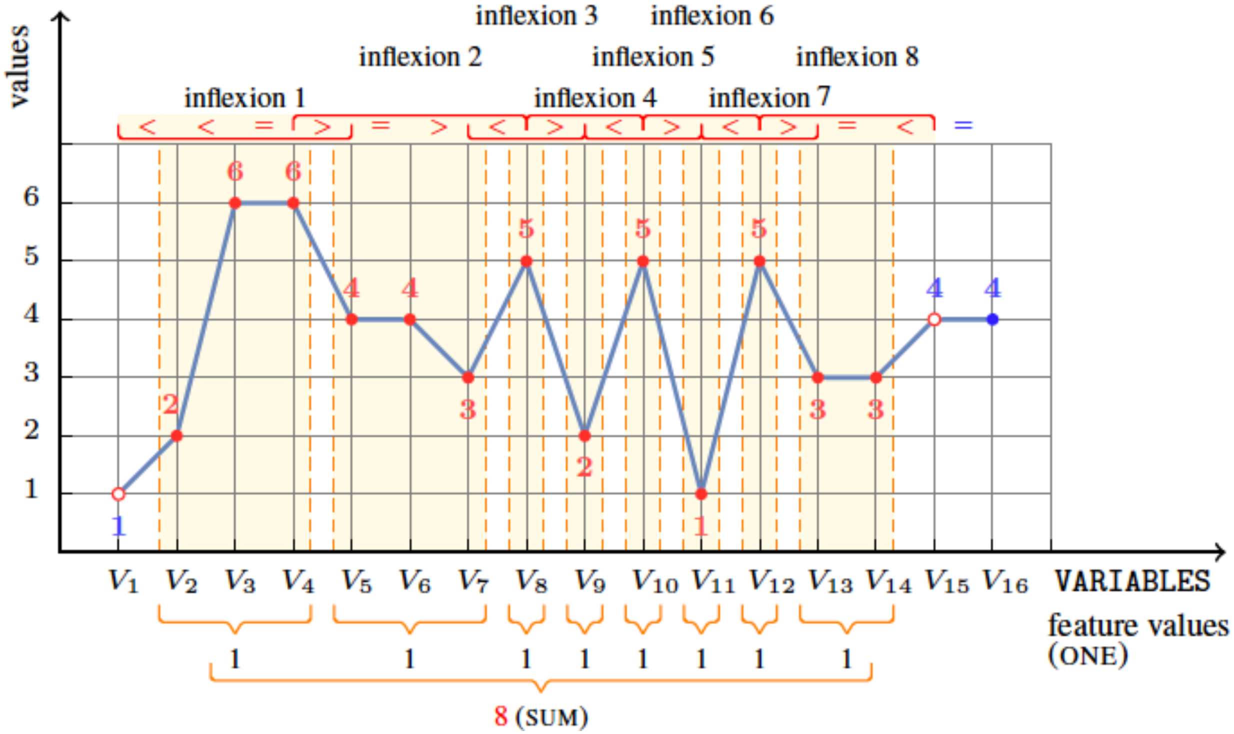
Example of constraint checker

- Check is achieved by scanning **once** through the variables **without** using any data structure

EXAMPLE

counting the **number of inflexions**



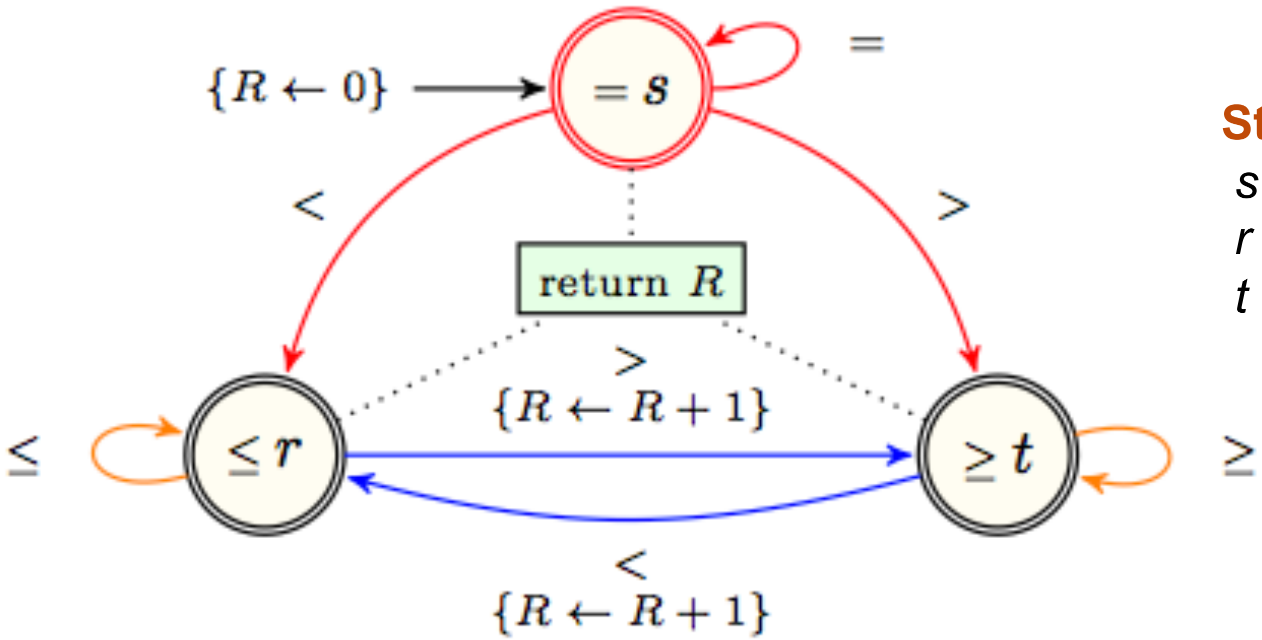


Example of constraint checker (continued)

```
inflexion(ninf,vars[0..n-1]):BOOLEAN;
BEGIN
  i=0; c=0;
  WHILE i<n-1 AND vars[i]=vars[i+1] DO i++;
  IF i<n-1 THEN less=(vars[i]<vars[i+1]);
  WHILE i<n-1 DO
    IF less THEN
      IF vars[i]>vars[i+1] THEN c++; less=FALSE;
    ELSE
      IF vars[i]<vars[i+1] THEN c++; less=TRUE;
    i++;
  RETURN (ninf=c);
END.
```

Constraint checker

- Use a deterministic automaton where all states are accepting
  - use an **accumulator** for counting number of inflexions (*updated while triggering certain transitions*)
  - final value of accumulator is returned (*green box*)



State semantics

s : **stationary** mode =*

r : **increasing** mode <{<|=}*

t : **decreasing** mode >{>|=}*

Transitions

- **Transitions** are labelled by a value in [val_1,val_2,…,val_p],

  where each value corresponds to a condition between a subset of variables of the original constraint:
  - P_0,P_1,…,P_m are these subsets (***signature arguments***)
  - to the i-th subset corresponds the **signature variable** S_i
  - the **link** between s_i and the variables of P_i is done according to p **mutually incompatible** conditions:

$$C_1(P_i) \Leftrightarrow s_i = val_1$$

$$C_2(P_i) \Leftrightarrow s_i = val_2$$

$$\dots \dots \dots \dots \dots \dots$$

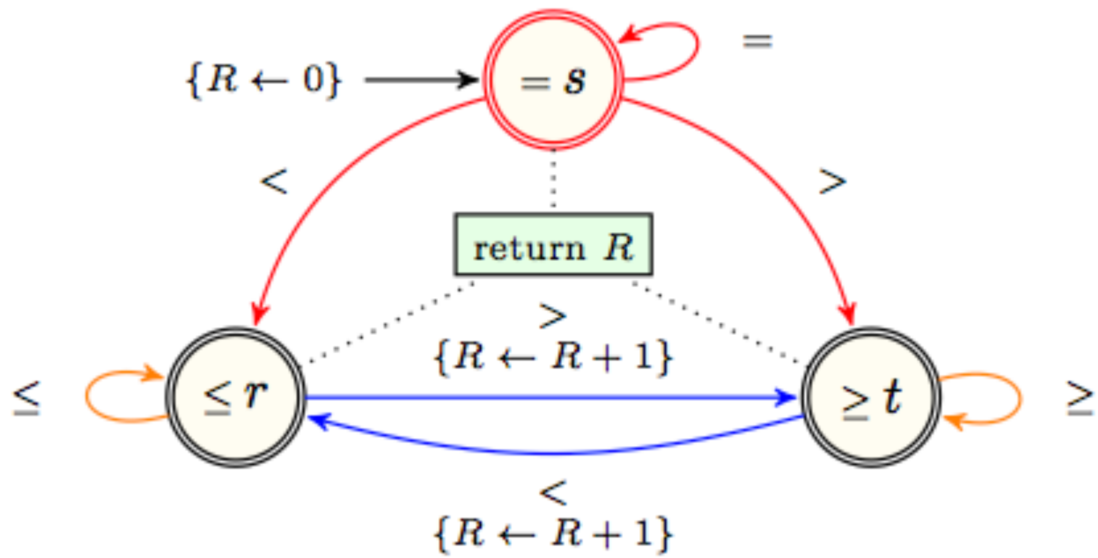
$$C_p(P_i) \Leftrightarrow s_i = val_p$$

This **conjunction** is called the **signature constraint** and is denoted **$\Psi(P_i, s_i)$**.

Example (transitions of inflexion)

inflexion(y, [x_0,x_1,x_2,x_3]) $P_0=\langle x_0,x_1\rangle$ $P_1=\langle x_1,x_2\rangle$ $P_2=\langle x_2,x_3\rangle$

$\Psi(s_i,x_i,x_{i+1})$: $(x_i > x_{i+1} \Leftrightarrow s_i = 0) \wedge (x_i = x_{i+1} \Leftrightarrow s_i = 1) \wedge (x_i < x_{i+1} \Leftrightarrow s_i = 2)$



Example of description of an automaton

inflexion(y, [$x_0,x_1,\ldots,x_{n-1}$]) $P_0=\langle x_0,x_1\rangle$ $P_1=\langle x_1,x_2\rangle$ $P_2=\langle x_2,x_3\rangle$

$\boldsymbol{\Psi(s_i,x_i,x_{i+1})}$: $(x_i > x_{i+1} \Leftrightarrow s_i = \mathbf{0}) \wedge (x_i = x_{i+1} \Leftrightarrow s_i = \mathbf{1}) \wedge (x_i < x_{i+1} \Leftrightarrow s_i = \mathbf{2})$

(1) **Signature variables** $s_0,s_1,\ldots,s_{n-2}$

(2) **Signature domain** $[\mathbf{0},\mathbf{2}]$

(3) **Signature argument** $\langle x_0,x_1\rangle$, $\langle x_1,x_2\rangle$, … , $\langle x_{n-2},x_{n-1}\rangle$

(4) **Counter(s)** R (initial value = 0)

(5) **States** s, r, t (*all accepting*), initial state: s

(6) **Transitions**

$(s, \mathbf{0}, t)$,	$(s, \mathbf{1}, s)$,	$(s, \mathbf{2}, r)$,
$(r, \mathbf{0}, t, [R+1])$,	$(r, \mathbf{1}, r)$,	$(r, \mathbf{2}, r)$,
$(t, \mathbf{0}, t)$,	$(t, \mathbf{1}, t)$,	$(t, \mathbf{2}, r, [R+1])$,

(7) **Return constraint** $y = R$

Running an automaton on a ground instance

For inflexion(4, [3,3,1,4,5,5,6,5,5,6,3]) we get:

s, R=0 $\{3 = 3 \Leftrightarrow s_0 = \mathbf{1}\} \longrightarrow s$

$\longrightarrow \{3 > 1 \Leftrightarrow s_1 = \mathbf{0}\} \longrightarrow t$

$\longrightarrow \{1 < 4 \Leftrightarrow s_2 = \mathbf{2}\} \longrightarrow r,\ R{=}1$

$\longrightarrow \{4 < 5 \Leftrightarrow s_3 = \mathbf{2}\} \longrightarrow r$

$\longrightarrow \{5 = 5 \Leftrightarrow s_4 = \mathbf{1}\} \longrightarrow r$

$\longrightarrow \{5 < 6 \Leftrightarrow s_5 = \mathbf{2}\} \longrightarrow r$

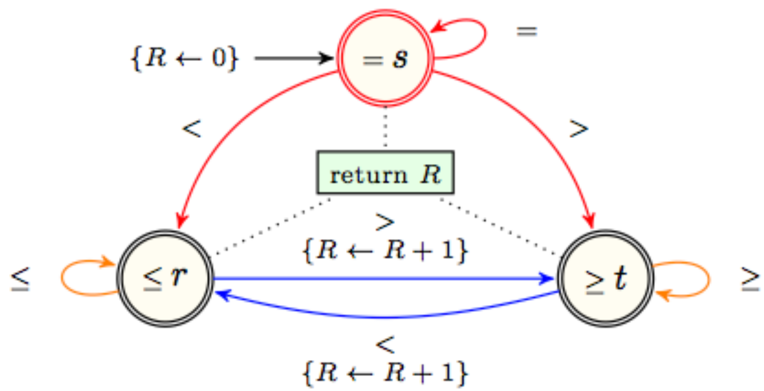
$\longrightarrow \{6 > 5 \Leftrightarrow s_6 = \mathbf{0}\} \longrightarrow t,\ R{=}2$

$\longrightarrow \{5 = 5 \Leftrightarrow s_7 = \mathbf{1}\} \longrightarrow t$

$\longrightarrow \{5 < 6 \Leftrightarrow s_8 = \mathbf{2}\} \longrightarrow r,\ R{=}3$

$\longrightarrow \{6 > 3 \Leftrightarrow s_9 = \mathbf{0}\} \longrightarrow t,\ R{=}4$

return 4



From automata to filtering algorithms

Simulate **all potentials executions** of an automaton according to the **current domain** of the variables in order to deduce **infeasible assignments**

How do we achieve this ? **First solution :**

By **reformulating** this as a conjunction of **signature** and **transition** constraints

Reformulation

Conjunction of **signature** and **transition** constraints :

signature variable — current state variable — next state variable

$$
\begin{array}{lll}
\Psi(s_0\,,\,P_0) & \wedge\; \Phi(q_0,\,K_0,\,s_0,\,q_1,\,K_1) & \wedge \\
\Psi(s_1\,,\,P_1) & \wedge\; \Phi(q_1,\,K_1,\,s_1,\,q_2,\,K_2) & \wedge \\
\ldots\;\ldots\;\ldots & \ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots\;\ldots & \\
\Psi(s_{m\text{-}1}\,,\,P_{m\text{-}1}) & \wedge\; \Phi(q_{m\text{-}1},\,K_{m\text{-}1},\,s_{m\text{-}1},\,q_m,\,K_m) &
\end{array}
$$

where

- q_0 is the **initial** state,
- $q_{m\text{-}1}$ is an **accepting** state,
- K_i is a vector containing the **counters**
  (K_0 *is the vector of initial counters value*)

Encoding transitions constraints for inflexion

$\Phi(q_0, s_0, q_1, R_0, R_1)$

$(\ q_0 = 0 \ \wedge\ s_0 = 0 \ \wedge\ q_1 = 2 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 0 \ \wedge\ s_0 = 1 \ \wedge\ q_1 = 0 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 0 \ \wedge\ s_0 = 2 \ \wedge\ q_1 = 1 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 1 \ \wedge\ s_0 = 0 \ \wedge\ q_1 = 2 \ \wedge\ R_1 = R_0 + 1) \vee$
$(\ q_0 = 1 \ \wedge\ s_0 = 1 \ \wedge\ q_1 = 1 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 1 \ \wedge\ s_0 = 2 \ \wedge\ q_1 = 1 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 2 \ \wedge\ s_0 = 0 \ \wedge\ q_1 = 2 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 2 \ \wedge\ s_0 = 1 \ \wedge\ q_1 = 2 \ \wedge\ R_1 = R_0\ ) \vee$
$(\ q_0 = 2 \ \wedge\ s_0 = 2 \ \wedge\ q_1 = 1 \ \wedge\ R_1 = R_0 + 1)$

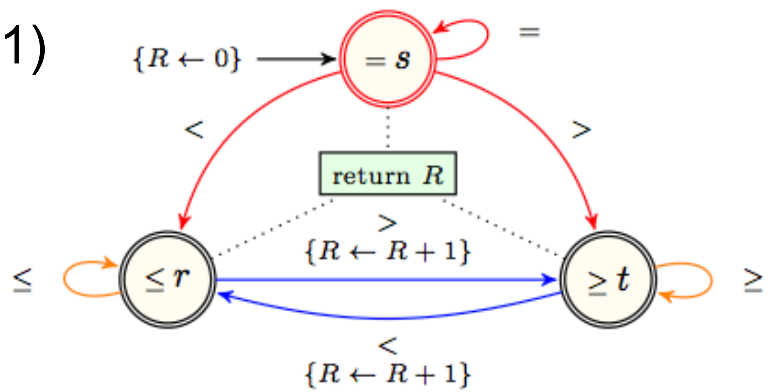
STATES

s **: 0**
r **: 1**
t **: 2**

INPUT LETTERS

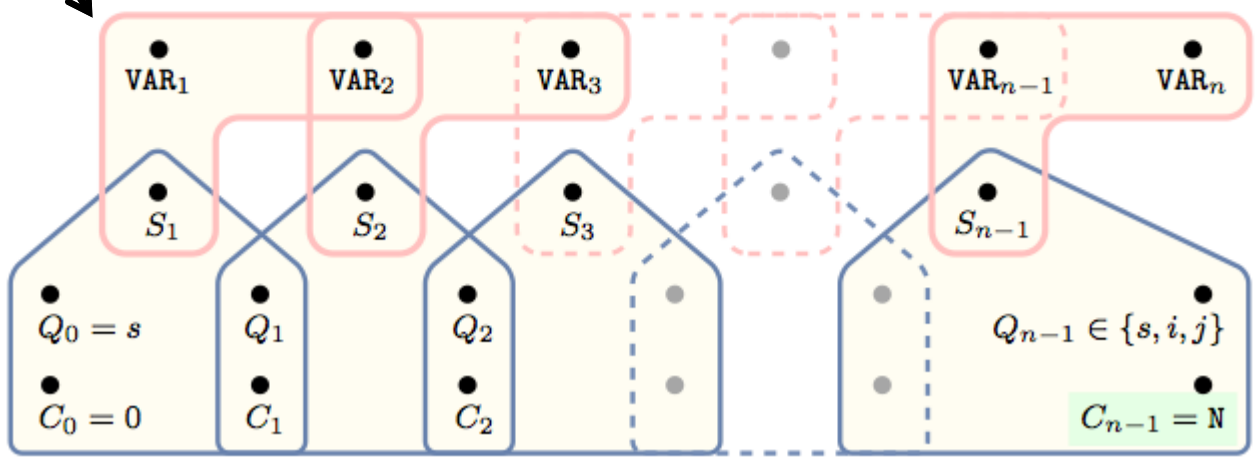
>: 0
=: 1
<: 2

can use a **table** constraint
(or logical constraints)



Hypergraph of the reformulation of the inflexion constraint

$\Psi(S_1, VAR_1, VAR_2)$



$\Phi(Q_0, C_0, S_1, Q_1, C_1)$

Berge acyclic hypergraph (constraint network)

An hypergraph is **Berge acyclic** if and only if:

1. **No more than one shared variable** between any pair of constraints.
2. The hypergraph **does not contain any cycle**.

A **cycle of length *k*** (k>2) is a sequence

(x_1, E_1, x_2, E_2, x_3, …, E_k, x_1) such:

1. E_1, E_2, … , E_k are distinct edges of the hypergraph,
2. x_1, x_2, … , x_k are distinct vertices of the hypergraph,
3. x_i, x_{i+1} belongs to E_i (i = 1,2,…,k-1),
4. x_k, x_1 belongs to E_k .

Intersection graph

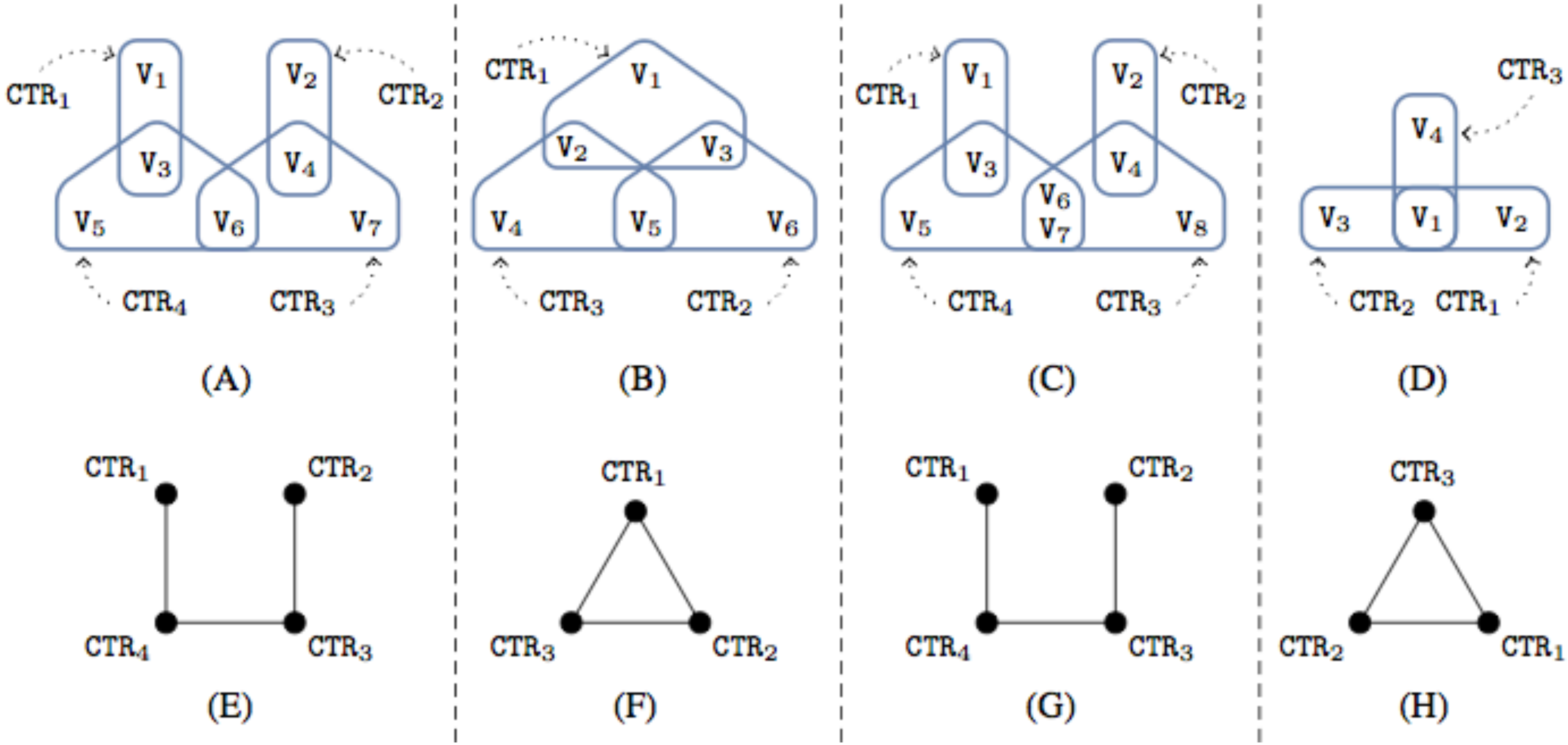
Can also check that no cycle in an hypergraph by checking that no cycle in the corresponding intersection graph.

The **intersection graph** is defined as:

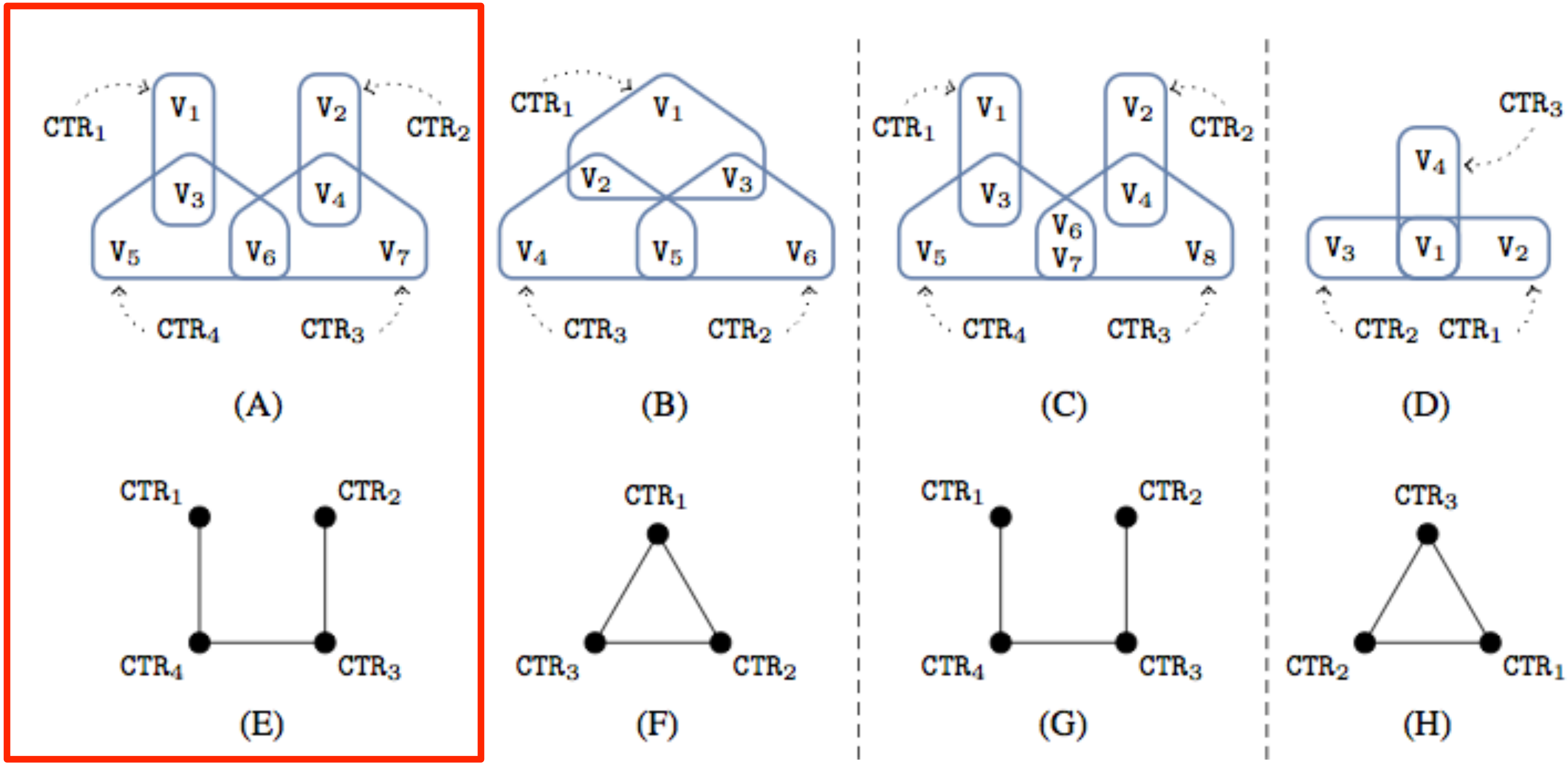
. to each constraint corresponds a vertex

. to each pair of constraints sharing at least one variable corresponds an edge

Berge acyclic constraint network: examples and counter examples



Berge acyclic constraint network: examples and counter examples

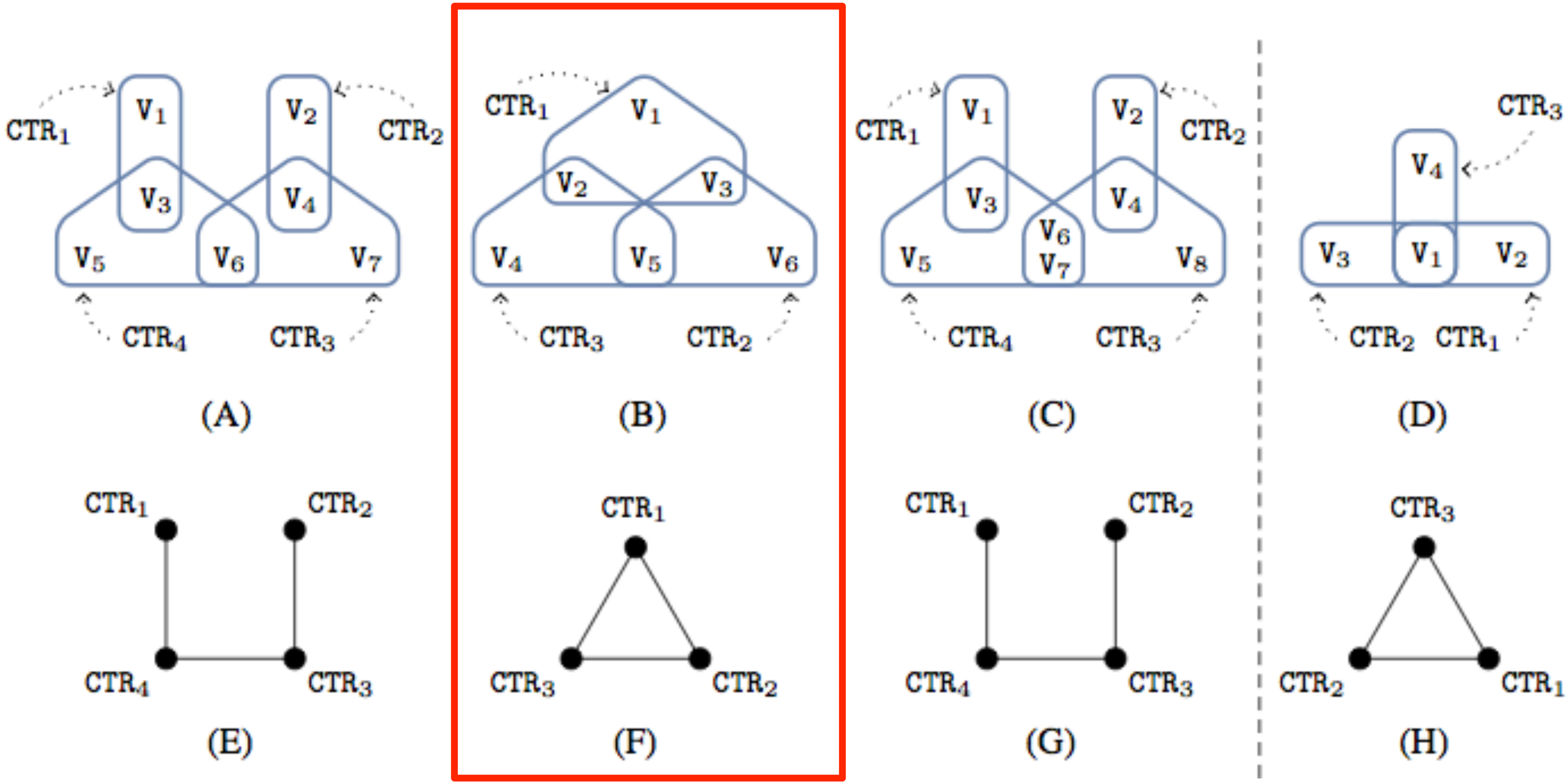


Yes since:

No more than one variable in common in (A)

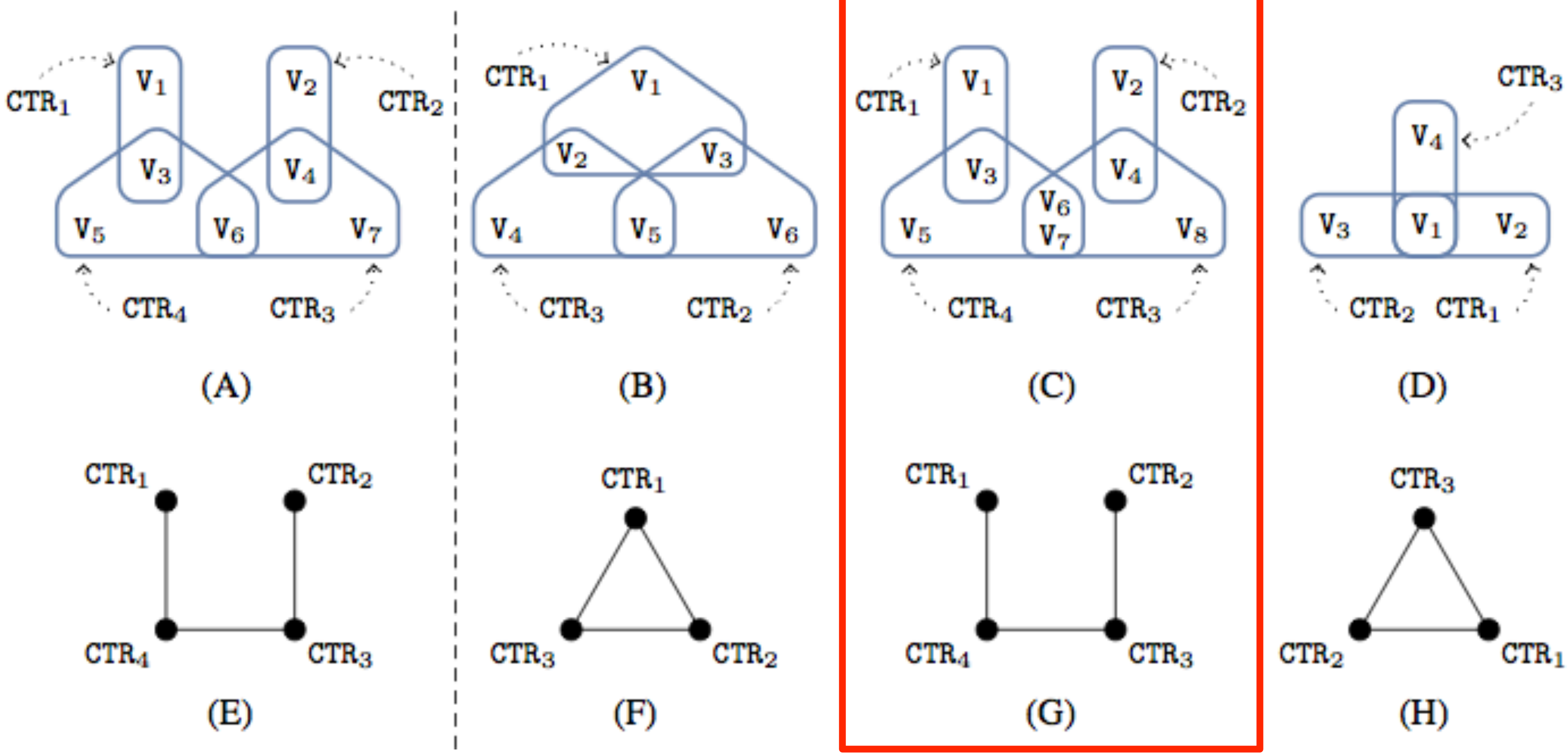
No cycle in the intersection graph (E)

Berge acyclic constraint network: examples and counter examples



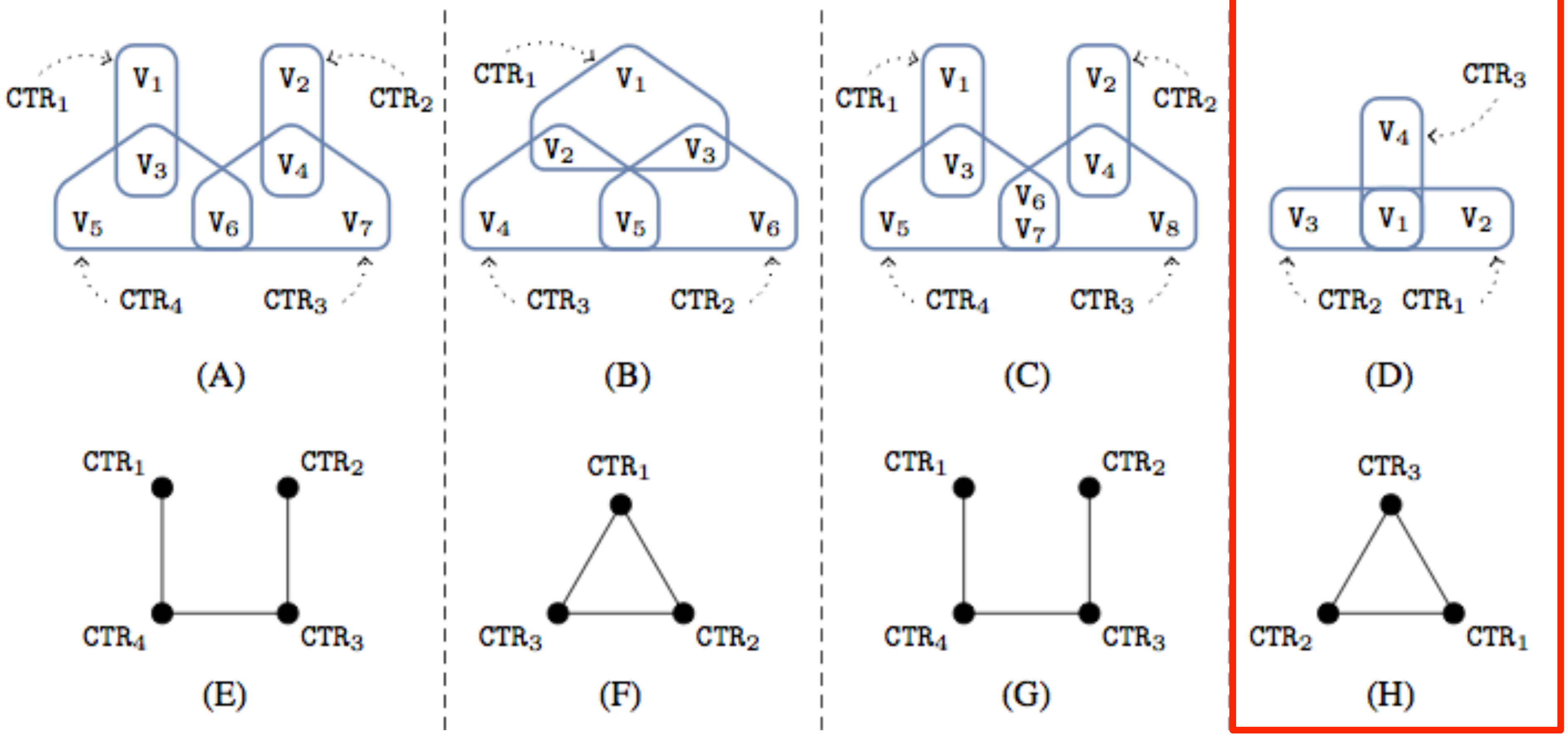
No since:
The hypergraph (B)
contains a cycle

Berge acyclic constraint network: examples and counter examples



No since:
The CTR_3 and CTR_4 have two variables in common in (C)

Berge acyclic constraint network: examples and counter examples



Yes since:
No more than one variable in common in (D)
Even if the intersection graph contains a cycle (see H), the hypergraph (see D) does not contain any cycle.

Consistency

- **Property** : if the constraint hypergraph associated with the reformulation is **Berge-acyclic** and if we have **GAC on each constraint** then the **full network is GAC** [Jansen, Vilarem 88].

- **Observation 1** : the **table** constraint **achieves GAC**.
- **Observation 2** : when no counter is used the transition constraint is encoded with **one single** compact **table** constraint.

RESULT If the automaton does not use any accumulator and no intersection between the signature arguments then the constraint hypergraph of the reformulation is Berge-acyclic.

If the constraint hypergraph is Berge-acyclic and the signature constraint achieves GAC then the **reformulation achieves GAC**.

Algorithmic approach when no accumulator (Pesant)

- **Initialisation**: **unfold** the automaton wrt a concrete sequence of variables and there respective domains into a **DAG**:
  - Each layer is the set of states of the automaton
  - Each arc represents:
    - a transition from one state to the next state
    - a value in the domain of a variables
- **Forward phase**: mark all vertices that can be reached from the initial state
- **Backward phase**: mark all vertices that can be reached from at least one of the final states (*by reversing the arcs*).
- **Filtering phase**: remove all arcs from *u* to *v* such that
  - *u* not reachable from an initial state **or**
  - *v* cannot reach at least one of the accepting states.

Reformulation versus dedicated algorithm (automaton + accumulators)

- Dedicated algorithm
  - \- Unfold the automaton wrt a sequence: **huge graph** when domains are not sparse (***memory is a problem***)
  - \+ For extensions of regular can record specific information to do better deductions (*without getting the problem of loosing information because projecting information onto the domain of variables*)

Reformulation versus dedicated algorithm (automaton + accumulators)

- Reformulation
  - \- For automata with accumulators may perform poorly when necessary conditions are not added (*the disjunction coming from the choice of transitions may hinder propagation*)
  - \+ Use less memory since do not unfold the automata (*note that the table encoding the transition constraint is the same for all transition constraints and note that the reformulation introduces only a linear number of extra variables*)
  - \+ The reformulation introduces variables (*i.e. states variables*) that can be useful for expressing additional constraints like accessibility constraints (*e.g. is a given state accessible ?*).

Automata in the global constraint catalog (*without accumulator*)

- AND,
- ARITH,
- ARITH_OR,
- BETWEEN_MIN_MAX,
- CLAUSE_AND,
- CLAUSE_OR,
- COND_LEX_COST,
- CONSECUTIVE_GROUPS_OF_ONES,
- DECREASING,
- DOMAIN_CONSTRAINT,
- ELEM,
- ELEM_FROM_TO,
- ELEMENT,
- ELEMENT_GREATEREQ,
- ELEMENT_LESSEQ,
- ELEMENT_MATRIX,
- ELEMENT_SPARSE,
- ELEMENTN,
- EQUIVALENT,
- GLOBAL_CONTIGUITY,
- IMPLY,
- IN,
- IN_INTERVAL,
- IN_SAME_PARTITION,
- INCREASING,
- INCREASING_GLOBAL_CARDINALITY,
- INCREASING_NVALUE,
- INT_VALUE_PRECEDE,
- INT_VALUE_PRECEDE_CHAIN,
- LESSEQ_IF_OCCURS,
- LEX_BETWEEN,
- LEX_DIFFERENT,
- LEX_EQUAL,
- LEX_GREATER,
- LEX_GREATEREQ,
- LEX_LESS,
- LEX_LESSEQ,
- MAXIMUM,
- MINIMUM,
- MINIMUM_EXCEPT_0,
- MINIMUM_GREATER_THAN,
- NAND,
- NEXT_ELEMENT,
- NO_PEAK,
- NO_VALLEY,
- NOR,
- NOT_ALL_EQUAL,
- NOT_IN,
- OPEN_MAXIMUM,
- OPEN_MINIMUM,
- OR,
- PATTERN,
- SEQUENCE_FOLDING,
- STAGE_ELEMENT,
- STRETCH_PATH,
- STRETCH_PATH_PARTITION,
- STRICTLY_DECREASING,
- STRICTLY_INCREASING,
- TWO_ORTH_ARE_IN_CONTACT,
- TWO_ORTH_DO_NOT_OVERLAP,
- XOR.

Automata in the global constraint catalog (*with accumulator*)

- ALL_EQUAL_EXCEPT_0,
- ALL_EQUAL_PEAK,
- ALL_EQUAL_PEAK_MAX,
- ALL_EQUAL_VALLEY,
- ALL_EQUAL_VALLEY_MIN,
- AMONG,
- AMONG_DIFF_0,
- AMONG_INTERVAL,
- AMONG_LOW_UP,
- AMONG_MODULO,
- ARITH_SLIDING,
- ATLEAST,
- ATMOST,
- BIG_PEAK,
- BIG_VALLEY,
- CHANGE,
- CHANGE_CONTINUITY,
- CHANGE_PAIR,
- CHANGE_VECTORS,
- CIRCULAR_CHANGE,
- COUNT,
- COUNTS,
- CYCLIC_CHANGE,
- CYCLIC_CHANGE_JOKER,
- DECREASING_PEAK,
- DECREASING_VALLEY,
- DEEPEST_VALLEY,
- DIFFER_FROM_AT_LEAST_K_POS,
- DISTANCE_CHANGE,
- EQUILIBRIUM,
- EXACTLY,
- FIRST_VALUE_DIFF_0,
- FULL_GROUP,
- GROUP,
- GROUP_SKIP_ISOLATED_ITEM,
- HIGHEST_PEAK,
- INCREASING_PEAK,
- INCREASING_VALLEY,
- INFLEXION,
- ITH_POS_DIFFERENT_FROM_0,
- LENGTH_FIRST_SEQUENCE,
- LENGTH_LAST_SEQUENCE,
- LONGEST_CHANGE,
- LONGEST_DECREASING_SEQUENCE,
- LONGEST_INCREASING_SEQUENCE,
- MAX_DECREASING_SLOPE,
- MAX_INCREASING_SLOPE,
- MIN_DECREASING_SLOPE,
- MIN_DIST_BETWEEN_INFLEXION,
- MIN_INCREASING_SLOPE,
- MIN_SIZE_FULL_ZERO_STRETCH,
- MIN_SURF_PEAK,
- MIN_WIDTH_PEAK,
- MIN_WIDTH_PLATEAU,
- MIN_WIDTH_VALLEY,
- PEAK,
- SLIDING_CARD_SKIP0,
- SMOOTH,
- VALLEY.

Exercise: GAC for a conjunction of constraints

1. Provide an automaton without accumulator for the between(*A*,*X*,*B*) constraint, where *A*,*B* are lists of integer values, and *X* is a list of domain variables; the between constraint enforces that *A* is lexicographically less than or equal to *X* and that *X* is lexicographically less than or equal to *B* (*all lists have the same length*)

2. Provide an automaton without accumulator for the exactly_one(*X*,*V*) constraint, where *X* is list of domain variables and V is a list of distinct integers; the exactly_one constraint holds if exactly one variable from the list list is assigned a value in *V*.

3. Provide an automaton without accumulator for the conjunction of between(*A*,*X*,*B*) and exactly_one(*X*,*V*). Does it provides GAC ?

Hint: think about the signature constraints

Exercise: between(*A*,*X*,*B*)

Alphabet: all 9 combinations of these two groups of conditions
(*compare* x_i *with the corresponding bounds of A and B*)

- $l : a_i < x_i$
- $e : a_i = x_i$
- $g : a_i > x_i$

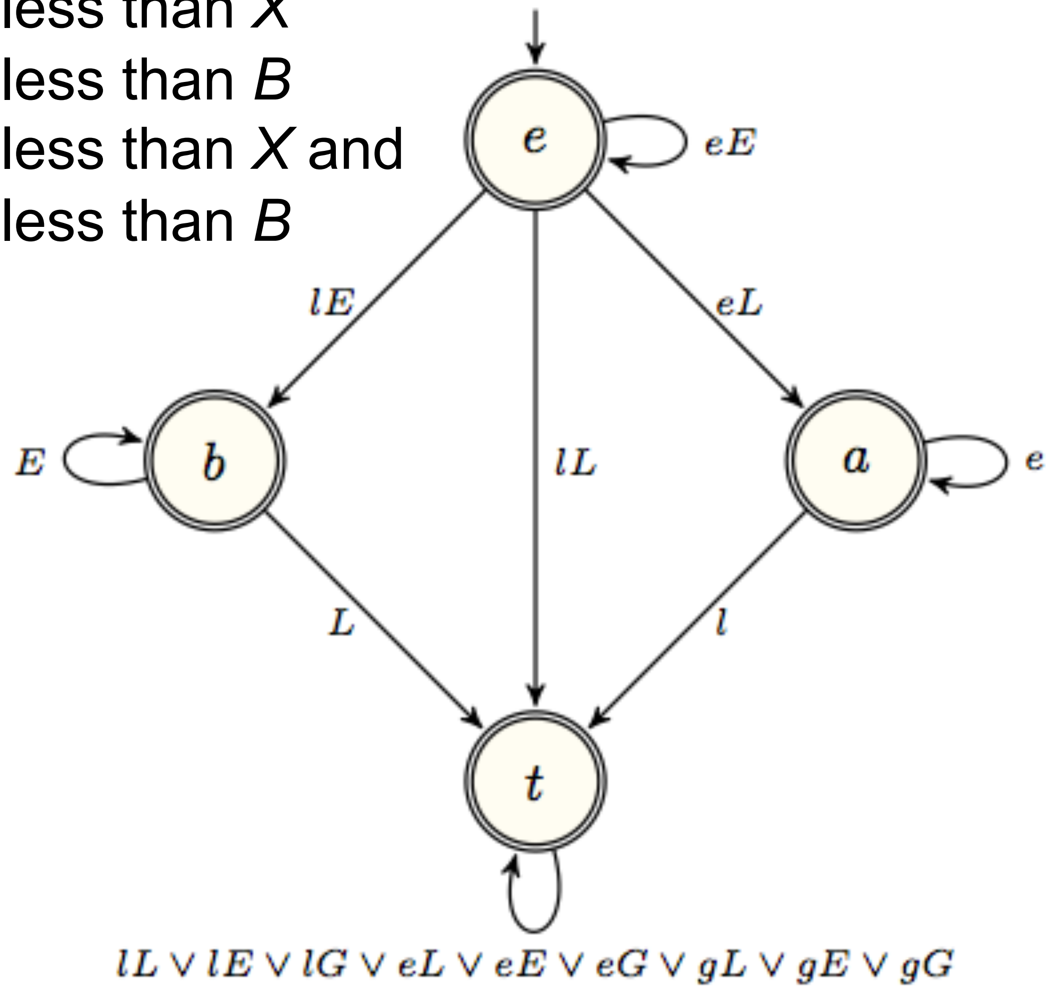
- $L : x_i < b_i$
- $E : x_i = b_i$
- $G : x_i > b_i$

Exercise: between(*A*,*X*,*B*)

- $l : a_i < x_i$
- $e : a_i = x_i$
- $g : a_i > x_i$
- $L : x_i < b_i$
- $E : x_i = b_i$
- $G : x_i > b_i$

State semantics:

e : prefixes of *A*, *X* and *B* are identical
b : *A* is lexicographically strictly less than *X*
a : *X* is lexicographically strictly less than *B*
t : *A* is lexicographically strictly less than *X* and *X* is lexicographically strictly less than *B*



Exercise: exactly_one(X,V)

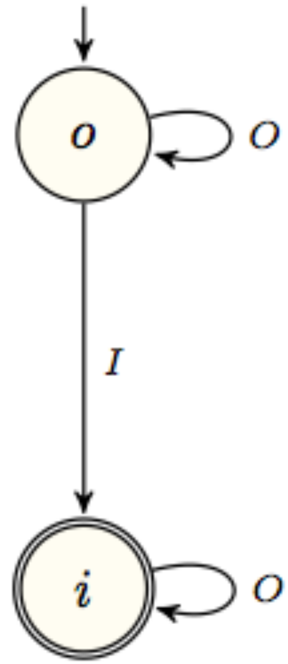
Alphabet:

- $I : x_i \in V$
- $O : x_i \notin V$

Exercise: exactly_one(X,V)

Alphabet:

- $I : x_i \in V$
- $O : x_i \notin V$



State semantics:

o : did not see any value in *V*
i : exactly one variable of X is assigned a value in *V*

Exercise: between(A,X,B) and exactly_one(X,V)

First question: what is the input alphabet ?

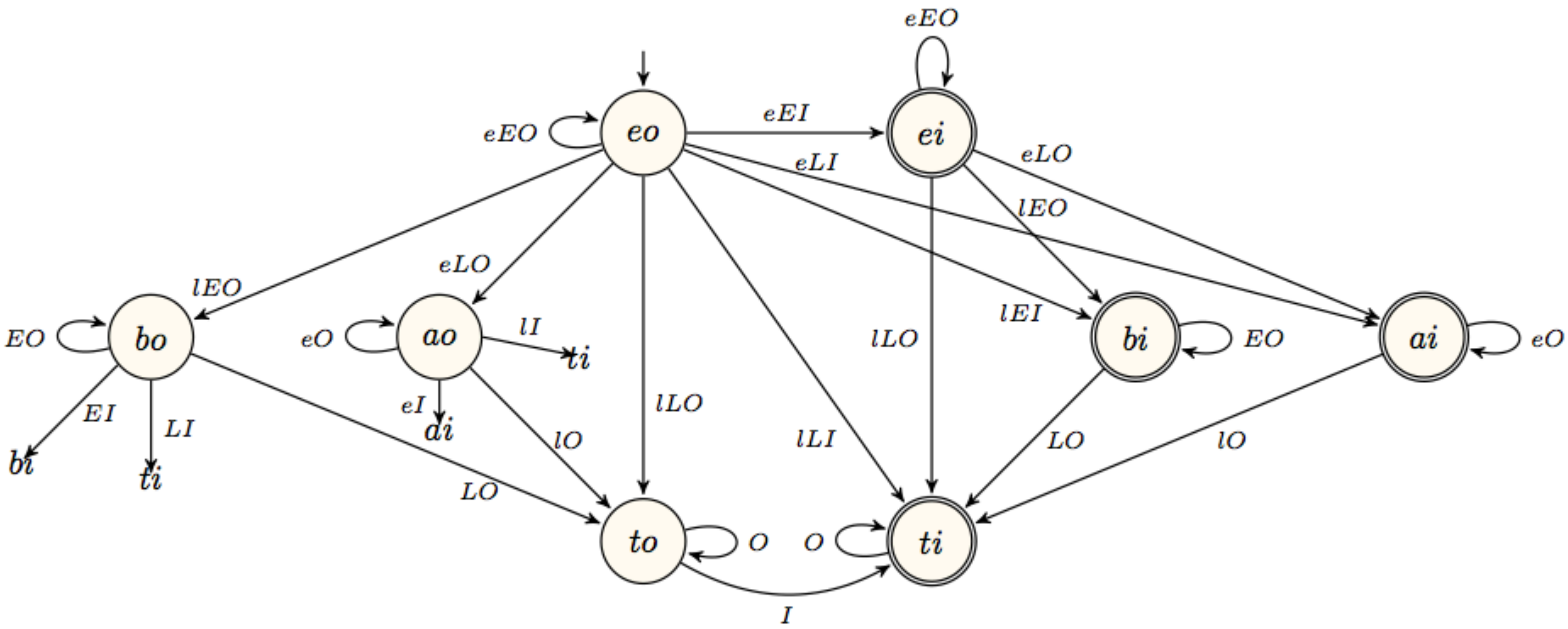
Exercise: between(A,X,B) and exactly_one(X,V)

Transition constraint for the conjunction **combines** the following set of conditions:
$\{a_i<x_i,\ a_i=x_i,\ a_i>x_i\}$, $\{b_i>x_i,\ b_i=x_i,\ b_i<x_i\}$, $\{x_i\in values,\ x_i\notin values\}$.

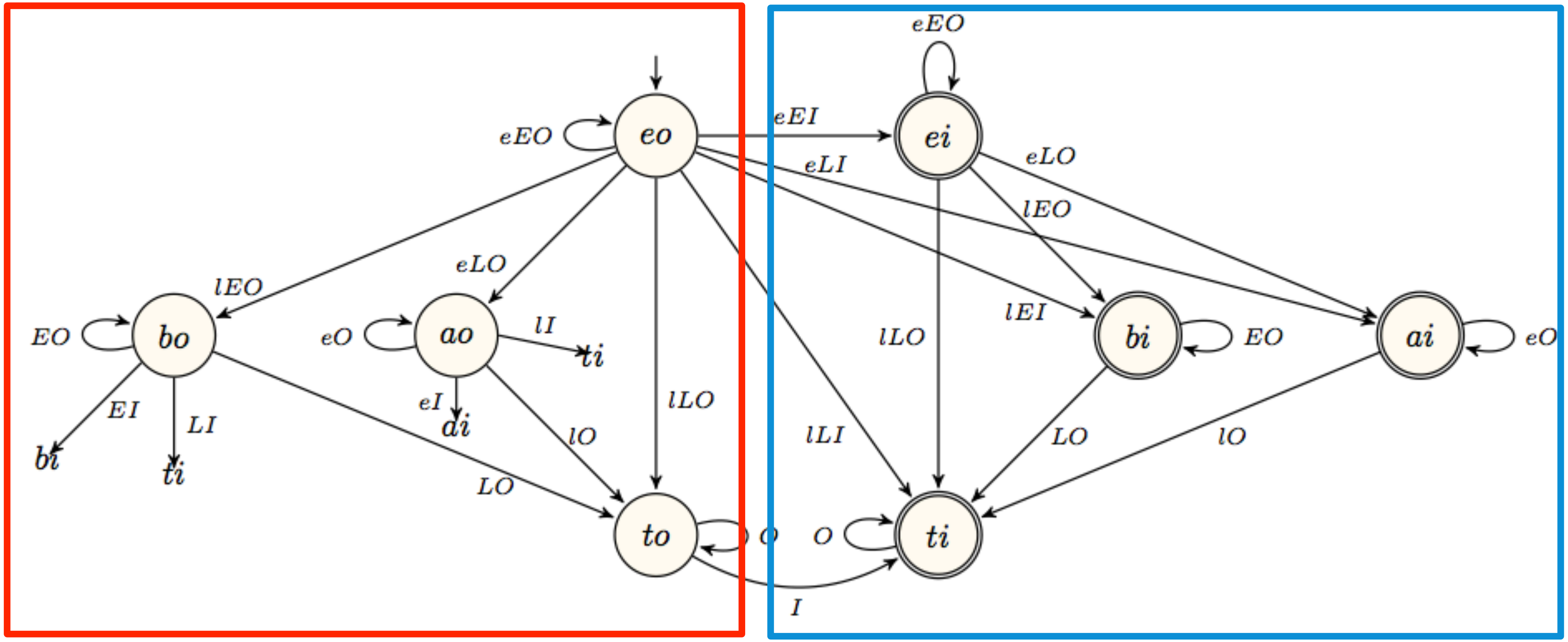
$$
S_i = \begin{array}{ll}
0 \text{ if } a_i<x_i \wedge b_i>x_i \wedge x_i\notin values, & 9 \text{ if } a_i<x_i \wedge b_i>x_i \wedge x_i\in values, \\
1 \text{ if } a_i<x_i \wedge b_i=x_i \wedge x_i\notin values, & 10 \text{ if } a_i<x_i \wedge b_i=x_i \wedge x_i\in values, \\
2 \text{ if } a_i<x_i \wedge b_i<x_i \wedge x_i\notin values, & 11 \text{ if } a_i<x_i \wedge b_i<x_i \wedge x_i\in values, \\
3 \text{ if } a_i=x_i \wedge b_i>x_i \wedge x_i\notin values, & 12 \text{ if } a_i=x_i \wedge b_i>x_i \wedge x_i\in values, \\
4 \text{ if } a_i=x_i \wedge b_i=x_i \wedge x_i\notin values, & 13 \text{ if } a_i=x_i \wedge b_i=x_i \wedge x_i\in values, \\
5 \text{ if } a_i=x_i \wedge b_i>x_i \wedge x_i\notin values, & 14 \text{ if } a_i=x_i \wedge b_i>x_i \wedge x_i\in values, \\
6 \text{ if } a_i>x_i \wedge b_i>x_i \wedge x_i\notin values, & 15 \text{ if } a_i>x_i \wedge b_i>x_i \wedge x_i\in values, \\
7 \text{ if } a_i>x_i \wedge b_i=x_i \wedge x_i\notin values, & 16 \text{ if } a_i>x_i \wedge b_i=x_i \wedge x_i\in values, \\
8 \text{ if } a_i>x_i \wedge b_i>x_i \wedge x_i\notin values, & 17 \text{ if } a_i>x_i \wedge b_i>x_i \wedge x_i\in values.
\end{array}
$$

Remark: the signature /transition constraint is **still a unary constraint**

Exercise: between(A,X,B) and exactly_one(X,V)



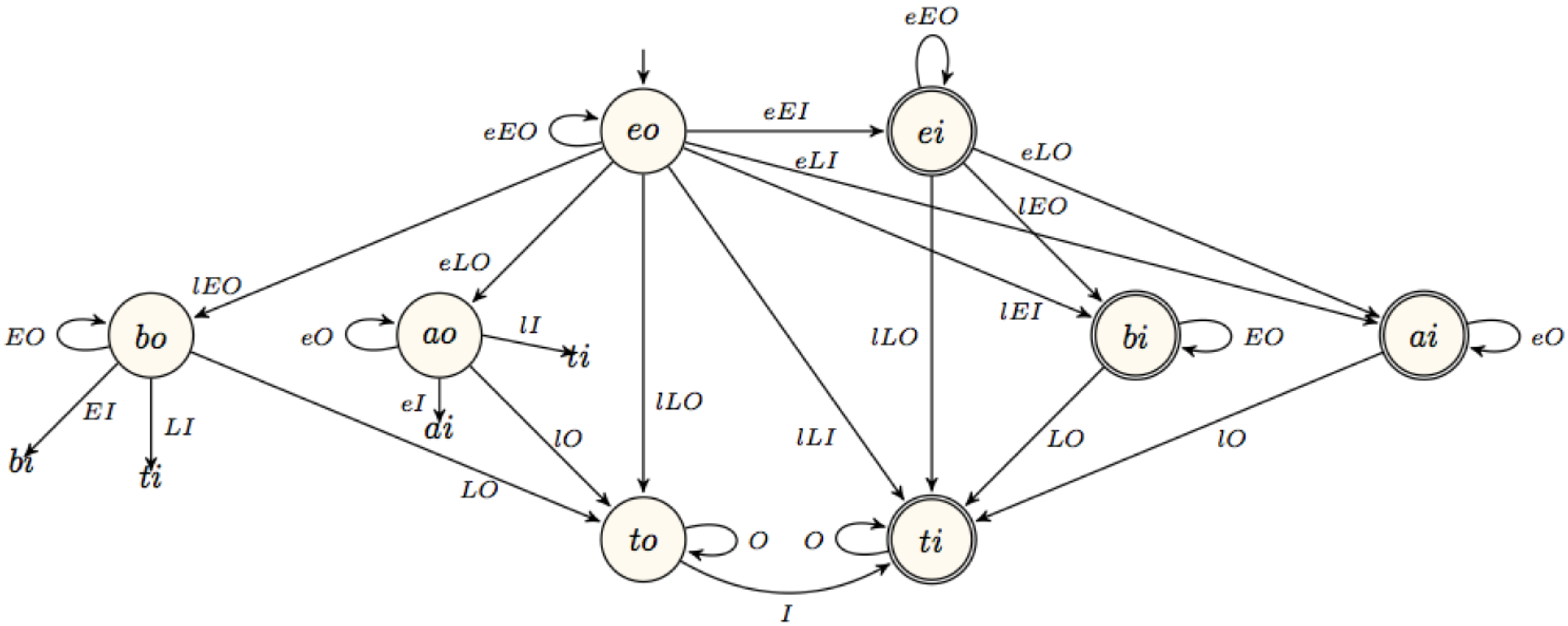
Exercise: between(A,X,B) and exactly_one(X,V)



no occurrence of V in the prefix

exactly one occurrence of a V in the prefix

Exercise: between(*A*,*X*,*B*) and exactly_one(*X*,*V*)



EXAMPLE OF PRUNING

(u=1,v=0,w=0): unique solution such that w=0

u ∈ {0,1}, v ∈ {0,3}, w ∈ {0,1,2,3}
between(<0,3,1>, <u,v,w>, <1,0,2>) and exactly_one(<u,v,w>, {0})

Finding out that **w≠0** requires to **reason globally on both constraints**:

After two transitions, the automaton will be either in state **ai** or in state **bi**.
In either state, a 0 must already have been seen,
and so there is no support for w=0.

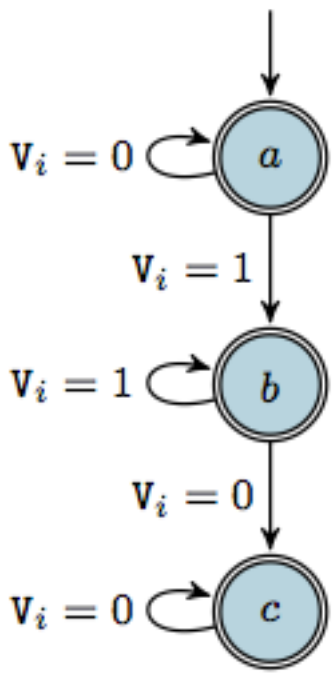
Exercise: Reification of a constraint specified by an automaton without accumulator

1. Provide an automaton for the global_contiguity constraint: all variables are assigned value 0 or 1, and no multiple occurrences of 1 separated by at least one 0

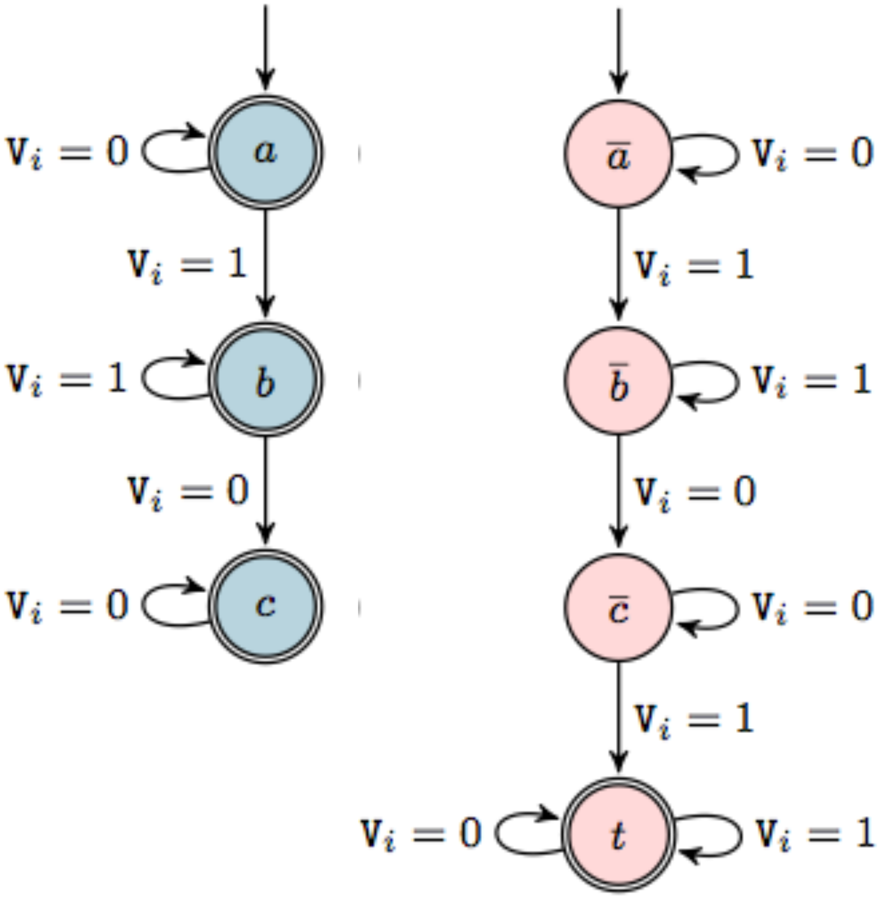
2. Provide an automaton for the negation of the global_contiguity constraint.

3. From 1. and 2. construct an automaton for the reification of the global_contiguity constraint (it contains an additional 0/1 variables that is set to 1 if and only if the global_contiguity constraint holds)

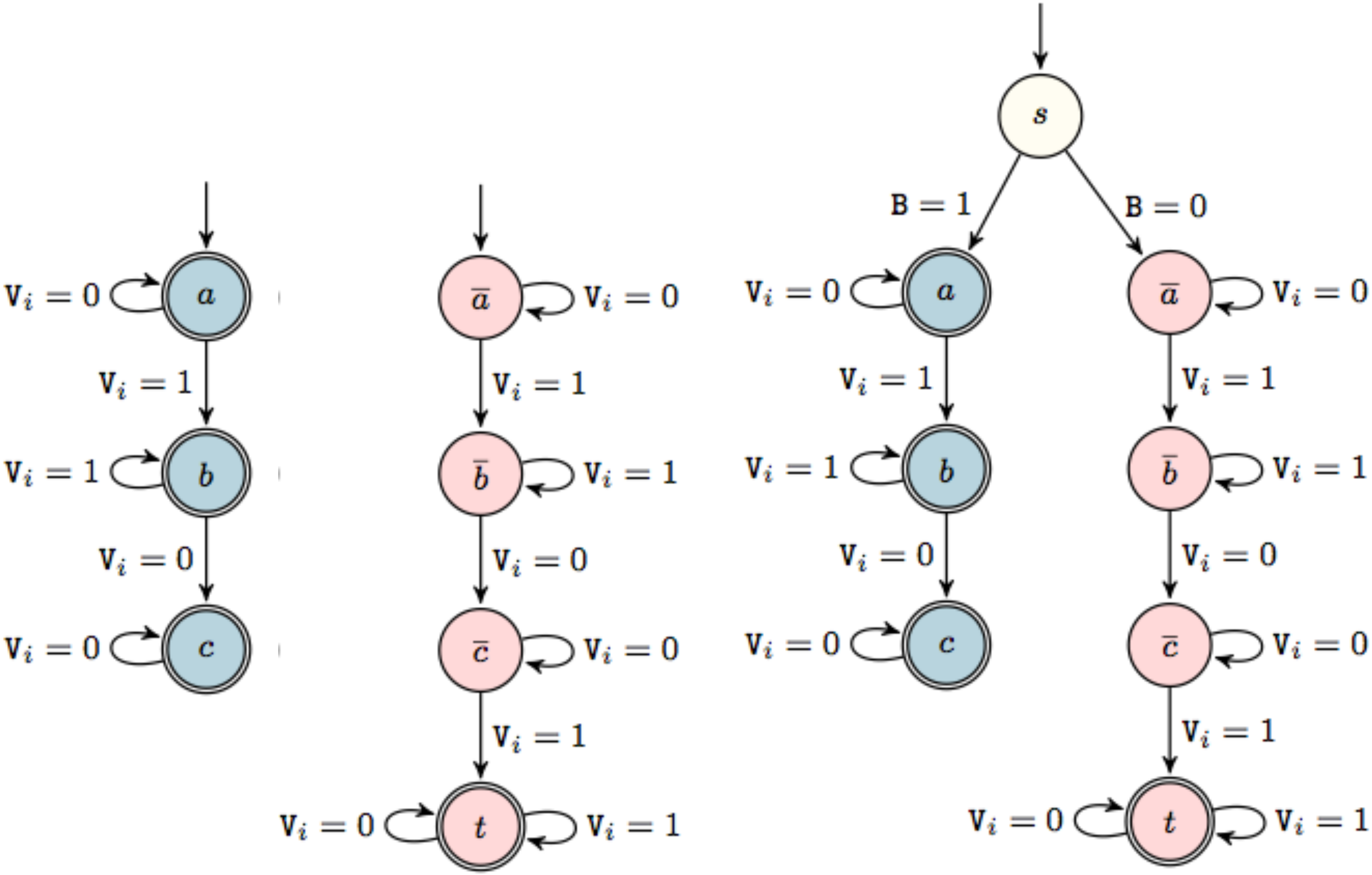
Exercise: automaton for global_contiguity



Exercise: automaton for the negation of global_contiguity



Exercise: reification of global_contiguity



Reversible automata constraints and **glue matrix** (*in the context of automata with accumulators*)

- Reversibility
- Glue matrix
- Applications of glue matrix

Reversible constraint

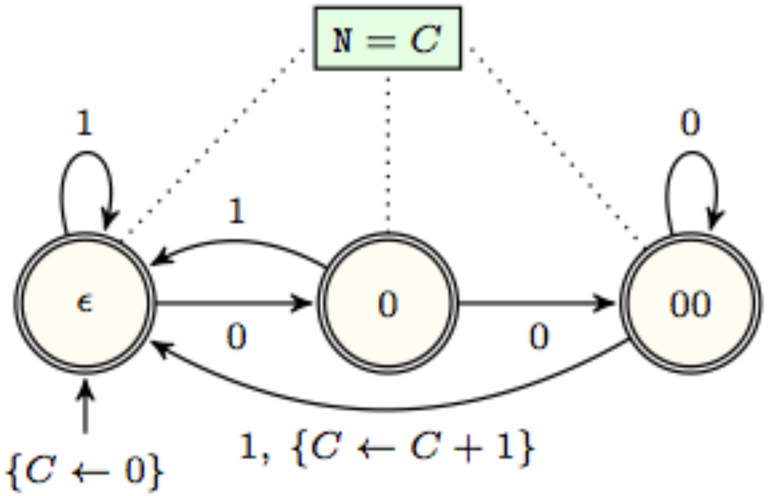
Definition

A *totally functional constraint* $C(R, S_1, S_2, \ldots, S_n)$ is a constraint where R is uniquely determined by $S_1, S_2, \ldots, S_n$.

The *reverse of a totally functional constraint* C is a constraint C' such that $C(R, S_1, S_2, \ldots, S_n) \Leftrightarrow C'(R, S_n, \ldots, S_2, S_1)$

- $\textsc{among}(2, \langle 1, 0, 0 \rangle, \{0\}) \Leftrightarrow \textsc{among}(2, \langle 0, 0, 1 \rangle, \{0\})$
- $\textsc{peak}(1, \langle 0, 6, 4, 3 \rangle) \Leftrightarrow \textsc{peak}(1, \langle 3, 4, 6, 0 \rangle)$
- $\textsc{length\_first\_stretch}(2, \langle 3, 3, 5, 6 \rangle) \Leftrightarrow \textsc{length\_last\_stretch}(2, \langle 6, 5, 3, 3 \rangle)$

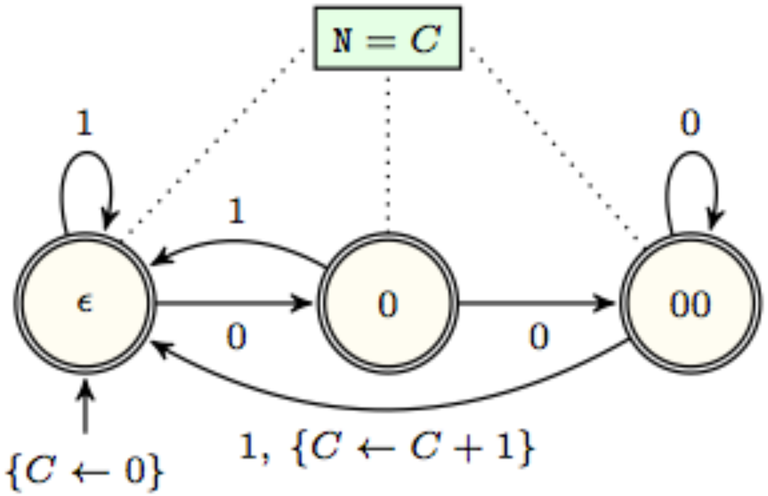
Reversible constraint (example)



nocc_001(N, [x_1, x_2, … , x_n])

reverse ?

Reversible constraint (example)

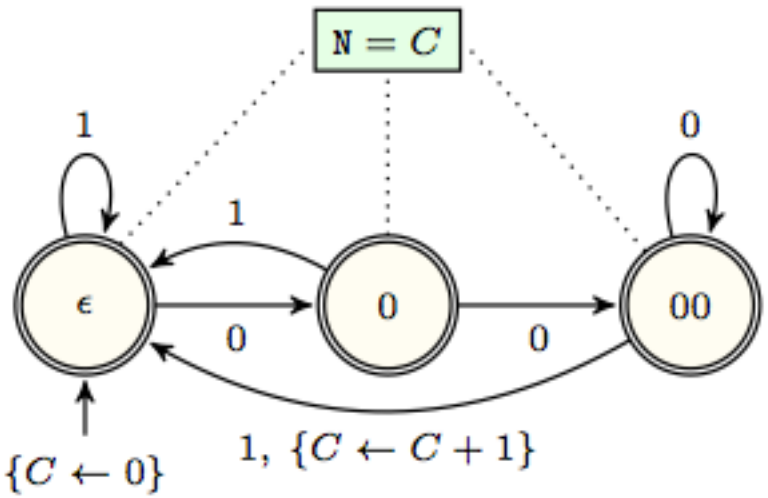


reverse ?

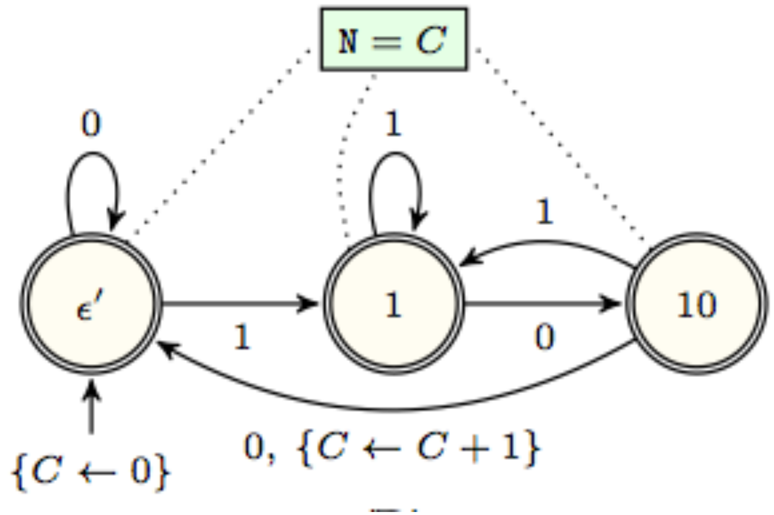
nocc_001(N, [x_1, x_2, … , x_n])

nocc_100(N, [x_1, x_2, … , x_n])

Reversible constraint (example)



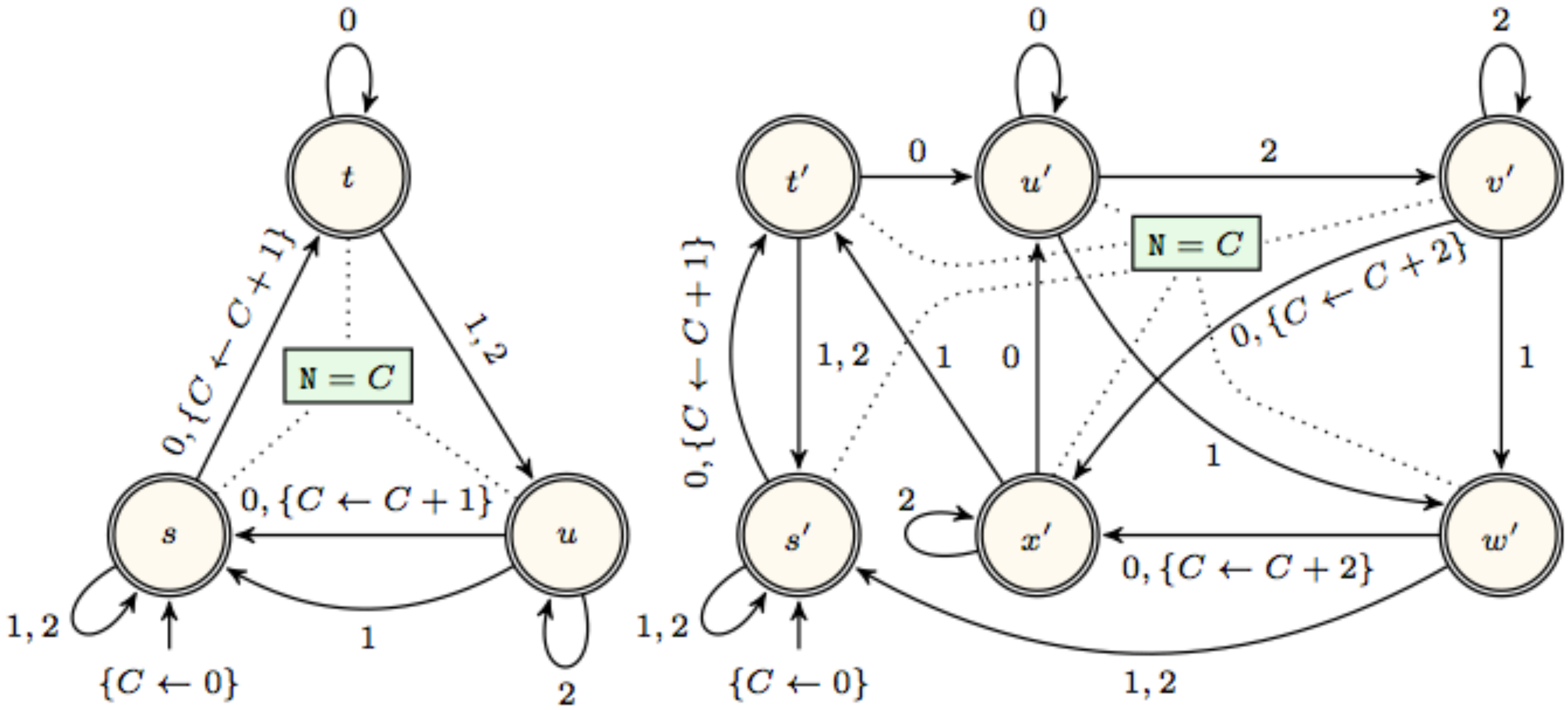
nocc_001(N, [x_1, x_2, … , x_n])



nocc_100(N, [x_1, x_2, … , x_n])

Can be computed mechanically if only one accumulator and use only incrementation (*but may be non-deterministic and contains* ε)

Reversible constraint (other example)



Not always same number of states

Reversible constraints in the context of global constraints: constraints on sequences

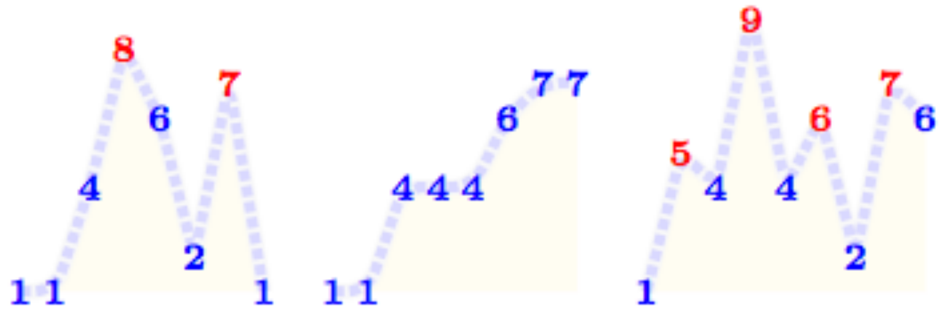
In practice :

- Many cases where a constraint is its own reverse

- When a constraint is not its own reverse most of the time the corresponding automata are symmetric (*i.e., permute some letters of the alphabet*)

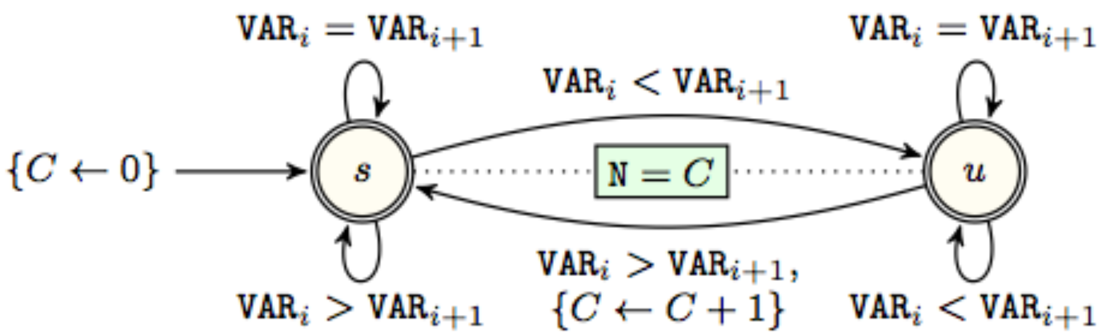
Reversible constraints: nb_peak

$(2, \langle 1, 1, 4, 8, 6, 2, 7, 1 \rangle)$
$(0, \langle 1, 1, 4, 4, 4, 6, 7, 7 \rangle)$
$(4, \langle 1, 5, 4, 9, 4, 6, 2, 7, 6 \rangle)$



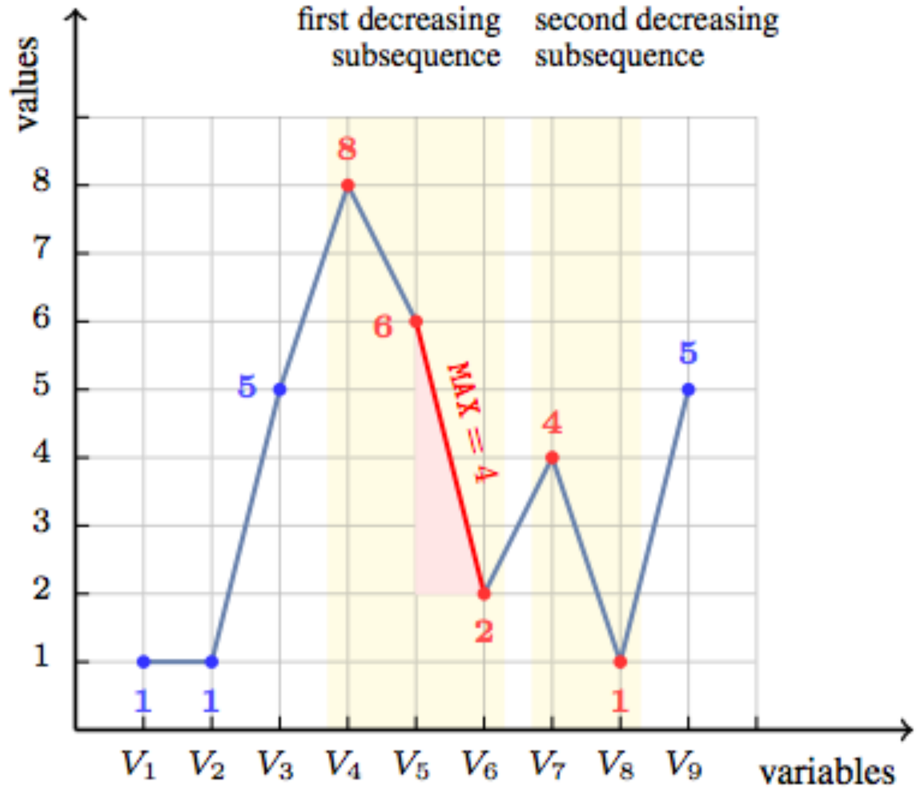
STATE SEMANTICS

s	: stationary/decreasing mode	$(\{> \mid =\}^*)$
u	: increasing mode	$(< \{< \mid =\}^*)$

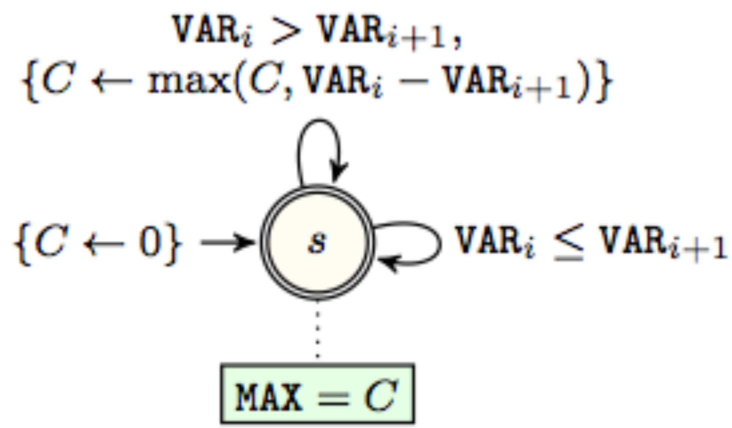


Example where a constraint is its **own reverse**

Reverse of max_decreasing_slope

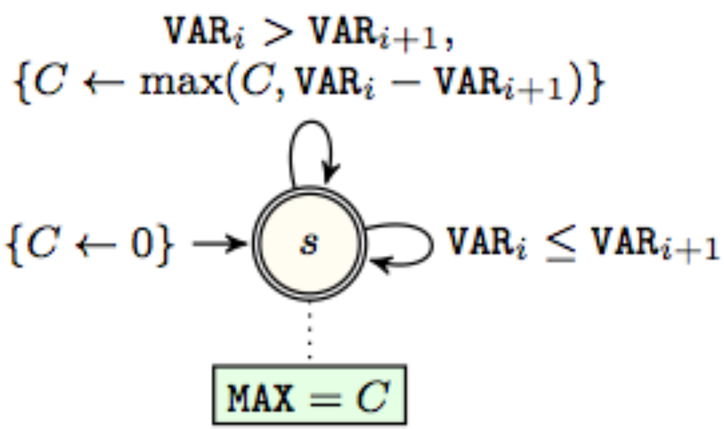


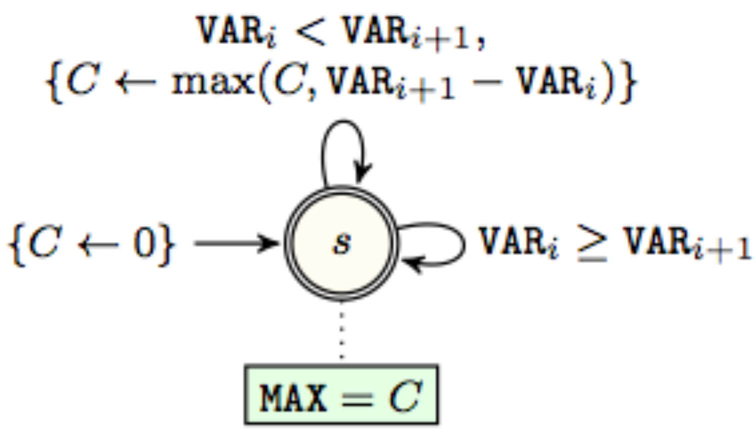
max_decreasing_slope example



max_decreasing_slope automaton

Reverse of max_decreasing_slope is max_increasing_slope





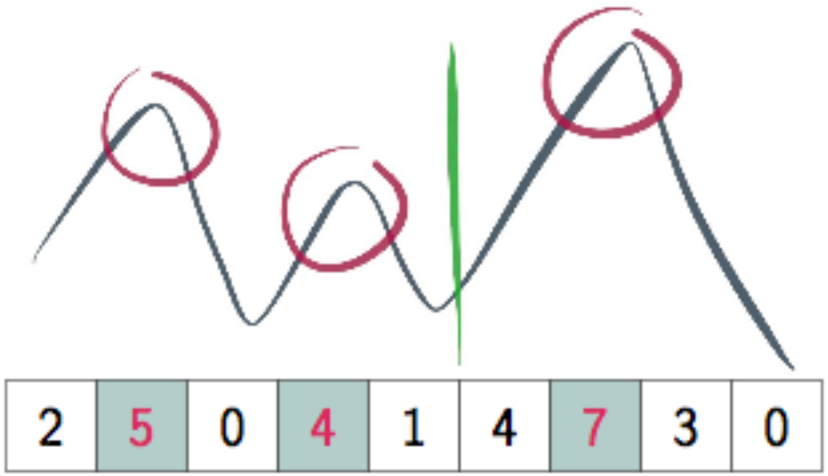
max_decreasing_slope automaton

max_increasing_slope automaton

- Reversibility
- **Glue matrix**
- Applications of glue matrix

Glue matrix for a constraint and its reverse: what it is all about?

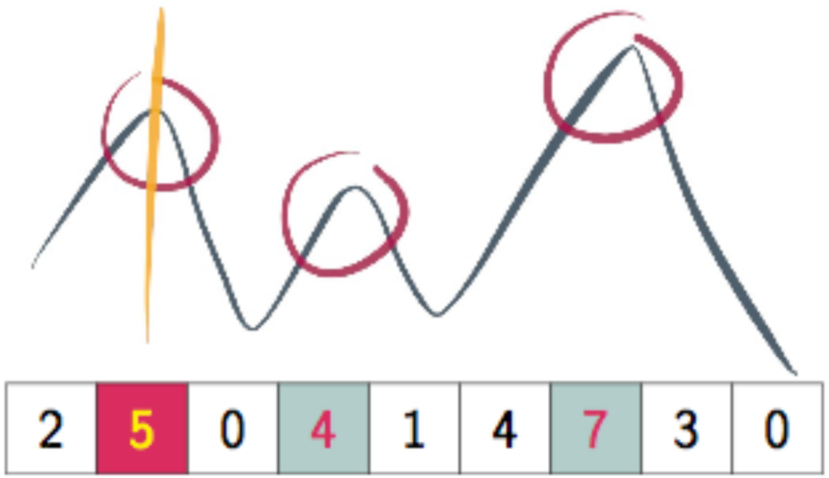
Consider counting peaks in a sequence:



$2 + 1 = 3$ peaks

Glue matrix for a constraint and its reverse: what it is all about?

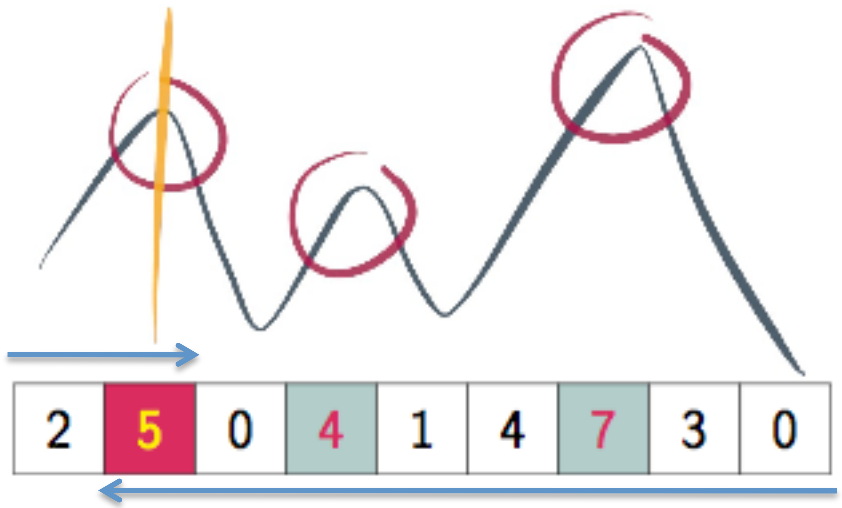
Consider counting peaks in a sequence:



$0 + 2 \neq 3$ peaks

Glue matrix for a constraint and its reverse: what it is all about?

Consider counting peaks in a sequence:



$0 + 2 \neq 3$ peaks

REMARK: for a signature constraint of arity k, the prefix and suffix **overlap by k-1** positions

Glue matrix for a constraint and its reverse: what it is all about?

Count the number of occurrence of a pattern in a word using an **automaton with accumulators**.

Characterize the prefix-suffix relationship and use it in different contexts, e.g. :

- constraint programming
- local search
- linear programming (*most likely*)

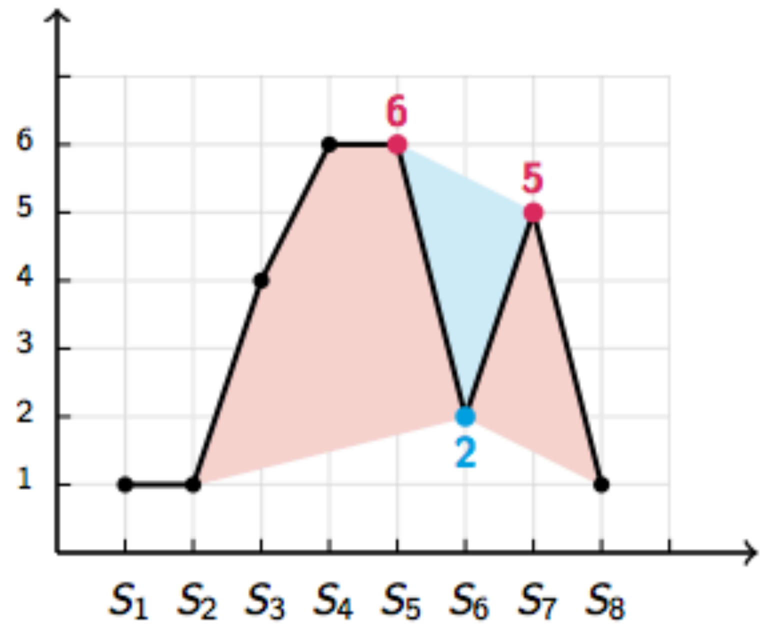
Notions of peak and valley

Definition

For an integer sequence $S_1, ..., S_m$, we say S_k $(1 < k < m)$ is a **peak** iff $\exists i \in [2, k] : \mathbf{S_{i-1} < S_i} \ \wedge \ S_i = S_{i+1} = \cdots = S_k \ \wedge \ \mathbf{S_k > S_{k+1}}$.

For an integer sequence $S_1, ..., S_m$, we say S_k $(1 < k < m)$ is a **valley** iff $\exists i \in [2, k] : \mathbf{S_{i-1} > S_i} \ \wedge \ S_i = S_{i+1} = \cdots = S_k \ \wedge \ \mathbf{S_k < S_{k+1}}$.

There are 2 peaks and 1 valley in the sequence $1, 1, 4, 6, 6, 2, 5, 1$



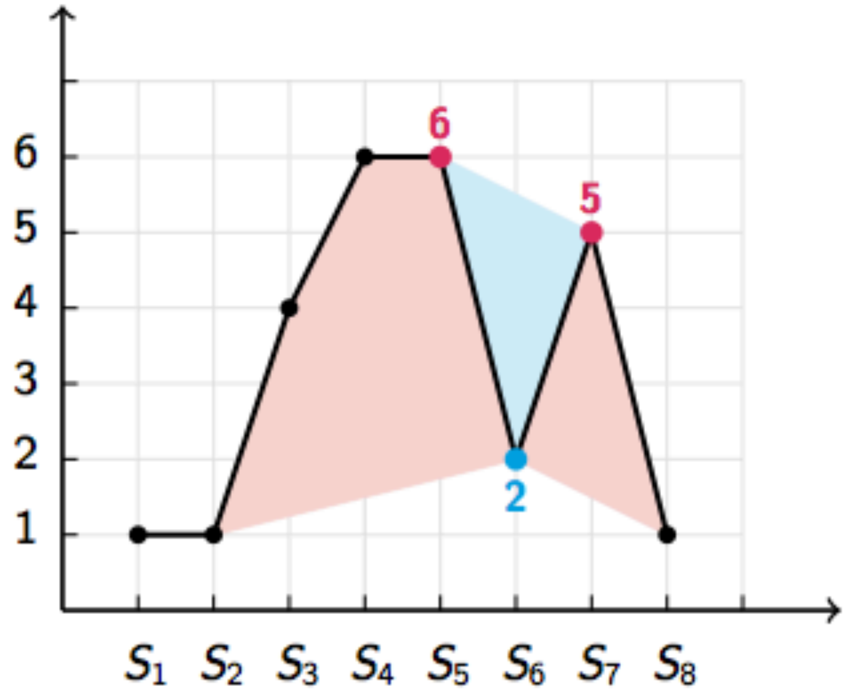
The PEAK and VALLEY constraints

Definition

$\text{PEAK}(\mathtt{P}, \mathtt{S}) \equiv$ sequence $\mathtt{S}$ contains $\mathtt{P}$ peaks.

$\text{VALLEY}(\mathtt{V}, \mathtt{S}) \equiv$ sequence $\mathtt{S}$ contains $\mathtt{V}$ valleys.

$\text{PEAK}(\mathbf{2}, \langle 1, 1, 4, 6, 6, 2, 5, 1 \rangle)$
$\text{VALLEY}(\mathbf{1}, \langle 1, 1, 4, 6, 6, 2, 5, 1 \rangle)$

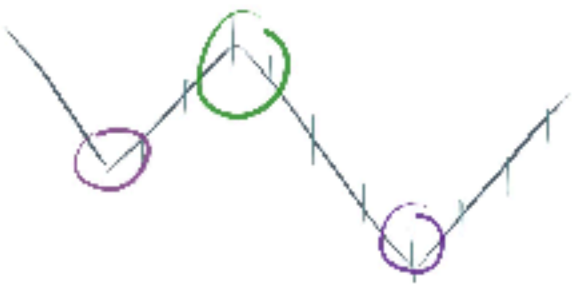


Invariant on the numbers of peaks and valleys

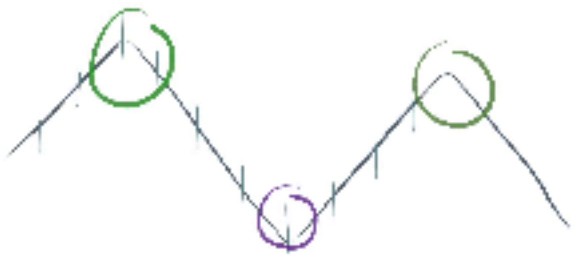
In order to improve propagation on
a conjunction of constraints on the same sequence,
we can add implied constraints linking accumulators.

For example:

$$\begin{cases} \text{PEAK}(\mathtt{P}, \mathtt{S}) \\ \text{VALLEY}(\mathtt{V}, \mathtt{S}) \end{cases} \Rightarrow \quad \boxed{|\mathtt{P} - \mathtt{V}| \leq 1}$$



alternation of valleys and peaks



alternation of peaks and valleys

Need of an invariant on all prefixes and suffixes

$$\begin{cases} \mathtt{S} = \langle S_1, \ldots, S_n \rangle, \text{ with } S_i \in [1, 2] \\ \textsc{peak}(\mathtt{P}, \mathtt{S}) \\ \textsc{valley}(\mathtt{V}, \mathtt{S}) \\ |\mathtt{P} - \mathtt{V}| \leq 1 \\ \mathtt{P} < \mathtt{V} \end{cases}$$

$S_1 = 1$: *no solution*
$S_1 = 2$: *plenty of solutions*

The invariant $|\mathtt{P} - \mathtt{V}| \leq 1$ on the full sequence is not enough.

Backtracks can drop from $2^{n-3} - 1$ *to* $\frac{(n-3)\cdot(n-2)}{2}$
if there is an invariant on all prefixes and suffixes.

Linking the prefixes and suffixes of a sequence

To enhance propagation we should post the invariant on **all prefixes and suffixes** of the sequence:

$$\overbrace{\underbrace{S_1, \ldots, S_i,}_{\overrightarrow{P_i}}\ \underbrace{S_{i+1}, \ldots, S_n}_{\overleftarrow{P_i}}}^{P} \qquad \overbrace{\underbrace{S_1, \ldots, S_i,}_{\overrightarrow{V_i}}\ \underbrace{S_{i+1}, \ldots, S_n}_{\overleftarrow{V_i}}}^{V}$$

$$\begin{cases} \textsc{peak}(P, \langle S_1, \ldots, S_n\rangle) \\ \forall i : \textsc{peak}(\overrightarrow{P_i}, \langle S_1, \ldots, S_i\rangle) \\ \forall i : \textsc{peak}(\overleftarrow{P_i}, \langle S_n, \ldots, S_{i+1}\rangle) \end{cases} \qquad \begin{cases} \textsc{valley}(V, \langle S_1, \ldots, S_n\rangle) \\ \forall i : \textsc{valley}(\overrightarrow{V_i}, \langle S_1, \ldots, S_i\rangle) \\ \forall i : \textsc{valley}(\overleftarrow{V_i}, \langle S_n, \ldots, S_{i+1}\rangle) \end{cases}$$

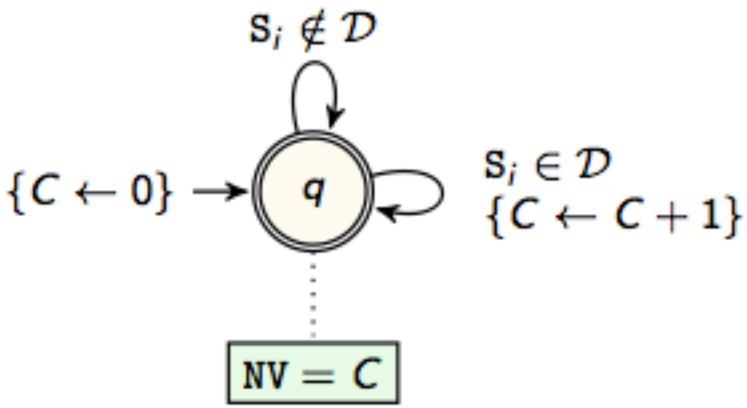
$$\begin{cases} \forall i : |\overrightarrow{P_i} - \overrightarrow{V_i}| \leq 1 \\ \forall i : |\overleftarrow{P_i} - \overleftarrow{V_i}| \leq 1 \end{cases}$$

$$\overrightarrow{P_i} + \overleftarrow{P_i} \stackrel{?}{=} P \qquad \overrightarrow{V_i} + \overleftarrow{V_i} \stackrel{?}{=} V$$

A gentle start: glue matrix for the AMONG constraint

Question: Consider an integer sequence split into a prefix and a suffix. Suppose that value v occurs $\overrightarrow{C}$ times in the prefix and $\overleftarrow{C}$ times in the suffix, how many times does v occur in the whole sequence?

Answer: Just sum up the two quantities.



$$\begin{array}{cc} & q \\ q & \boxed{\overrightarrow{C} + \overleftarrow{C}} \end{array}$$

Glue matrix where $\overrightarrow{C}$ and $\overleftarrow{C}$ represent the accumulator value C at the end of a prefix and at the end of the corresponding reverse suffix of the sequence.

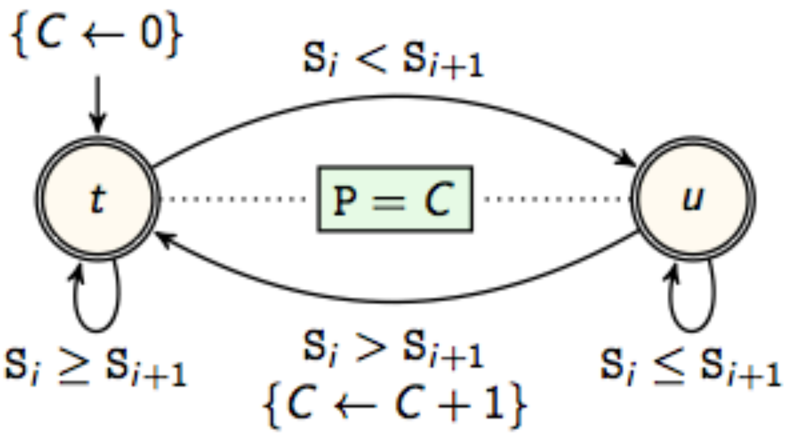
Glue matrix for the PEAK constraint

state semantics

t	stationary/decreasing
u	stationary/increasing

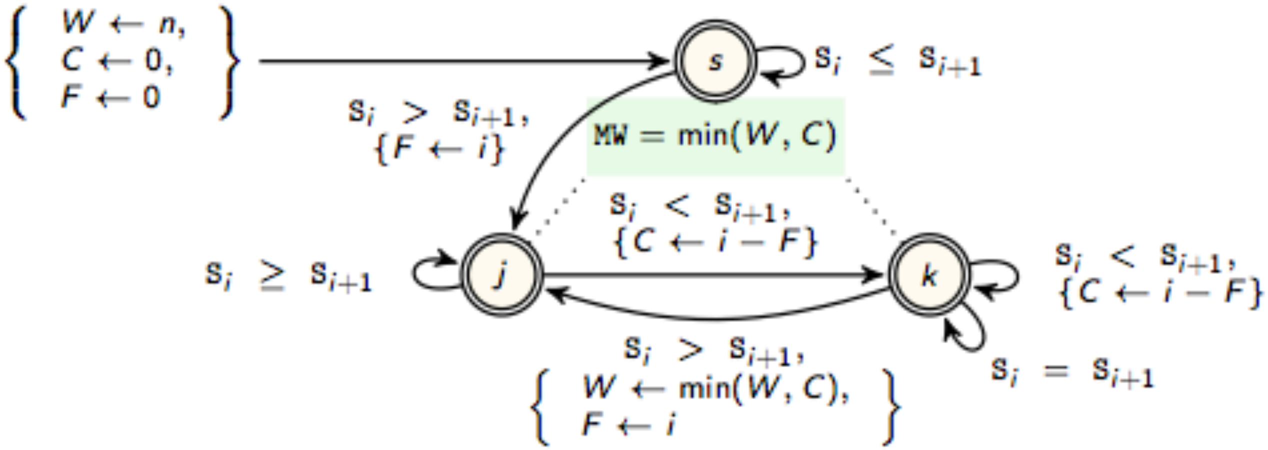
$$\begin{cases} \textsc{peak}(\mathbf{1}, \langle 2, 2, \mathbf{6}, 4, 1\rangle) \\ \textsc{peak}(\mathbf{0}, \langle 2, 2, 6\rangle) \\ \textsc{peak}(\mathbf{0}, \langle 1, 4, 6\rangle) \end{cases}$$

not just a sum!



	t	u
t	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + \overleftarrow{C}$
u	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + 1 + \overleftarrow{C}$

Glue matrix for the MIN_WIDTH_VALLEY constraint



	s	j	k
s	0	$\overleftarrow{\mathtt{MW}}$	$\overleftarrow{\mathtt{MW}}$
j	$\overrightarrow{\mathtt{MW}}$	$\min\left(\begin{array}{c}\overrightarrow{W},\\ n - \overrightarrow{F} - \overleftarrow{F},\\ \overleftarrow{W}\end{array}\right)$	$\min\left(\begin{array}{c}\overrightarrow{\mathtt{MW}},\\ n - \overrightarrow{F} - \overleftarrow{F},\\ \overleftarrow{\mathtt{MW}}\end{array}\right)$
k	$\overrightarrow{\mathtt{MW}}$	$\min\left(\begin{array}{c}\overrightarrow{\mathtt{MW}},\\ n - \overrightarrow{F} - \overleftarrow{F},\\ \overleftarrow{\mathtt{MW}}\end{array}\right)$	$\min\left(\begin{array}{c}\overrightarrow{\mathtt{MW}},\\ \overleftarrow{\mathtt{MW}}\end{array}\right)$

Main results on glue matrices

(*with a signature constraint of arity one*)

Existence of the glue matrix
Given an automaton with accumulators and its reverse, the glue matrix is **well defined**.

Computation of the glue matrix
With **a single accumulator** and **only incrementation**, we **can compute the glue matrix mechanically**.

The global constraint catalog contains a number of glue matrices

- Reversibility
- Glue matrix
- **Applications of glue matrix**

Stronger invariants (Constraint Programming)

Consider various patterns in a sequence.
Key point: Their numbers **do not vary independently**.

Already seen in the motivation section (*recall example*):

$$\begin{cases} \textsc{peak}(P, \langle S_1, \ldots, S_n \rangle) \\ \forall i : \textsc{peak}(\overrightarrow{P_i}, \langle S_1, \ldots, S_i \rangle) \\ \forall i : \textsc{peak}(\overleftarrow{P_i}, \langle S_n, \ldots, S_{i+1} \rangle) \end{cases} \qquad \begin{cases} \textsc{valley}(V, \langle S_1, \ldots, S_n \rangle) \\ \forall i : \textsc{valley}(\overrightarrow{V_i}, \langle S_1, \ldots, S_i \rangle) \\ \forall i : \textsc{valley}(\overleftarrow{V_i}, \langle S_n, \ldots, S_{i+1} \rangle) \end{cases}$$

$$\begin{cases} \forall i : |\overrightarrow{P_i} - \overrightarrow{V_i}| \leq 1 \\ \forall i : |\overleftarrow{P_i} - \overleftarrow{V_i}| \leq 1 \end{cases}$$

$$\forall i : \mathit{glue}_{\textsc{peak}}(P, \overrightarrow{P_i}, \overleftarrow{P_i}) \qquad \forall i : \mathit{glue}_{\textsc{valley}}(V, \overrightarrow{V_i}, \overleftarrow{V_i})$$

Stronger invariants (Constraint Programming)

Need to encode the glue matrix as a constraint:

- We need access to the sequence of **state variables** Q_i and the sequence of **accumulator variables** C_i.
  For the PEAK constraint, we have $P_i = C_i$.
- The encoding can be done with a **logical expression** involving state and accumulator variables.

For $glue_{\text{PEAK}}(P, \overrightarrow{P_i}, \overleftarrow{P_i})$, we get:

$$
\begin{array}{c}
B \in \{0,1\} \\
\wedge \\
(\overrightarrow{Q_i} = u \wedge \overleftarrow{Q_i} = u) \Leftrightarrow B = 1 \\
\wedge \\
(P = \overrightarrow{P_i} + B + \overleftarrow{P_i})
\end{array}
$$

	t	u
t	$\overrightarrow{c} + \overleftarrow{c}$	$\overrightarrow{c} + \overleftarrow{c}$
u	$\overrightarrow{c} + \overleftarrow{c}$	$\overrightarrow{c} + 1 + \overleftarrow{c}$

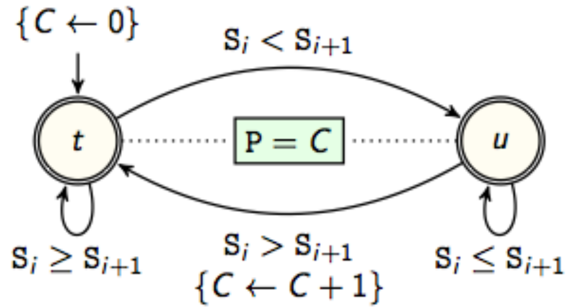
Enhancing bound on the full sequence by propagating the information from the prefix (constraint programming)

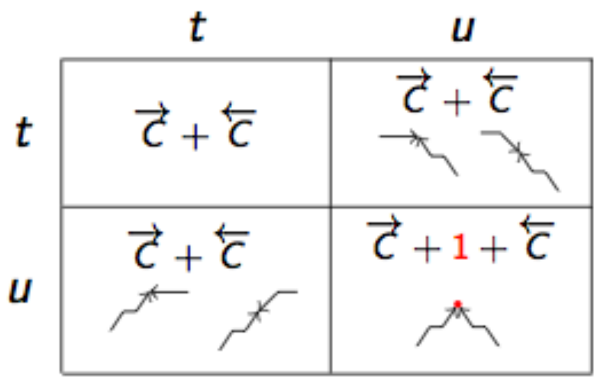
- Evaluating a bound on the full sequence is not enough as soon as we fix variables during the labelling (want to take into account the effect of the fixed variables on the bound on the full sequence)

- Solution
  - Set bound on the full sequence
  - Set bound of each prefix and each suffix (*linear*)
  - Use the glue matrix to link each prefix with its complement (*remark: the automaton with accumulator constraint will also update the bound*)

Enhancing bound on the full sequence: example

- Consider the peak constraint
- Maximum number of peaks on
- a sequence of length n: (n-1) div 2 (e.g., 010, 01010)
- Set bound on all prefix and suffix + glue constraint





Glue constraint (Qi,Qj,Ci,Cj,N):

$(Qi = t \quad \wedge \quad Qj = t \quad \Leftrightarrow \quad B = 0) \quad \wedge$

$(Qi = t \quad \wedge \quad Qj = u \quad \Leftrightarrow \quad B = 0) \quad \wedge$

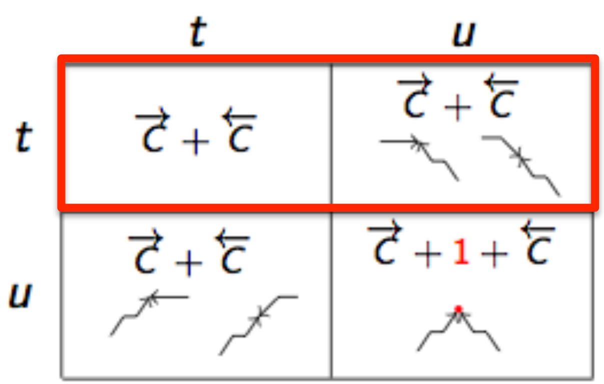
$(Qi = u \quad \wedge \quad Qj = t \quad \Leftrightarrow \quad B = 0) \quad \wedge$

$(Qi = u \quad \wedge \quad Qj = u \quad \Leftrightarrow \quad B = 1) \quad \wedge$

$N = Ci + Cj + B$

Enhancing bound on the full sequence: example

- Maximum number of peaks on sequence 00001000 x_1,x_2,x_3,x_4 :
  - Maximum number of peaks on **0000100**: 1 (*remark: go down*)
  - Maximum number of peaks on 0 x_1,x_2,x_3,x_4: (5-1)div 2=2
  - Maximum number of peak on the full sequence (using the glue constraint to channel the information from the prefix to the full sequence is 1+2).



Glue constraint (Qi,Qj,Ci,Cj,N):

(Qi = *t* $\wedge$ Qj = *t* $\Leftrightarrow$ B = 0) $\wedge$
(Qi = *t* $\wedge$ Qj = *u* $\Leftrightarrow$ B = 0) $\wedge$
(Qi = *u* $\wedge$ Qj = *t* $\Leftrightarrow$ B = 0) $\wedge$
(Qi = *u* $\wedge$ Qj = *u* $\Leftrightarrow$ B = 1) $\wedge$
N = Ci + Cj + B

Constant-time move probing (Local Search)

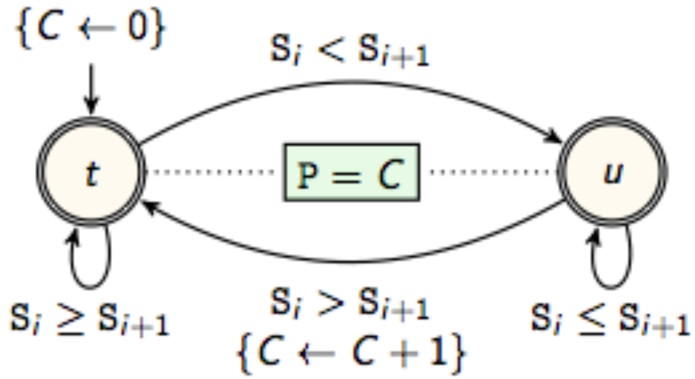
$\textsc{peak}(\mathrm{P} = 2, \langle 1, 1, 4, 8, 6, 2, 7, 1 \rangle)$

	1	1	4	8	8	6	2	7	1	
		=	<	<		<	<	>	<	
i	0	1	2	3	4	3	2	1	0	i
$\overrightarrow{Q_i}$	t	t	u	u	u	u	t	u	t	$\overleftarrow{Q_i}$
$\overrightarrow{C_i}$	0	0	0	0	1	1	1	0	0	$\overleftarrow{C_i}$
$\overrightarrow{\mathrm{P}_i}$	0	0	0	0	1	1	1	0	0	$\overleftarrow{\mathrm{P}_i}$

$\textsc{peak}\left(\begin{matrix} \overrightarrow{\mathrm{P}_3} = 0, \\ \langle 1, 1, 4, 8 \rangle \end{matrix} \right)$ $\textsc{peak}\left(\begin{matrix} \overleftarrow{\mathrm{P}_4} = 1, \\ \langle 8, 6, 2, 7, 1 \rangle \end{matrix} \right)$

glue matrix entry associated with the state pair (u, u):

$\mathrm{P} = \overrightarrow{\mathrm{C}_3} + 1 + \overleftarrow{\mathrm{C}_4} = 0 + 1 + 1 = 2$



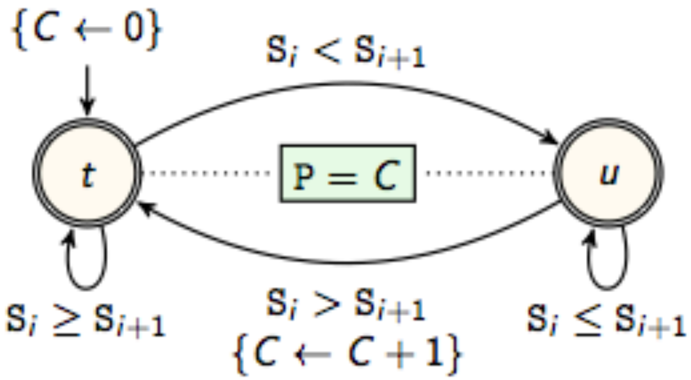
	t	u
t	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + \overleftarrow{C}$
u	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + 1 + \overleftarrow{C}$

Constant-time move probing (Local Search)

$\text{PEAK}(\mathrm{P} = ?, \langle 1, 1, 4, 1, 6, 2, 7, 1 \rangle)$

	1	1	4	1		1	6	2	7	1	
		$=$	$<$	$<$			$<$	$<$	$>$	$<$	
i	0	1	2	3	⋮	4	3	2	1	0	i
$\overrightarrow{Q_i}$	t	t	u	?	⋮	?	u	t	u	t	$\overleftarrow{Q_i}$
$\overrightarrow{C_i}$	0	0	0	?	⋮	?	1	1	0	0	$\overleftarrow{C_i}$
$\overrightarrow{\mathrm{P}_i}$	0	0	0	?	⋮	?	1	1	0	0	$\overleftarrow{\mathrm{P}_i}$

$\text{PEAK}\left(\begin{array}{c} \overrightarrow{\mathrm{P}_3} = ?, \\ \langle 1, 1, 4, 1 \rangle \end{array} \right) \text{PEAK}\left(\begin{array}{c} \overleftarrow{\mathrm{P}_4} = ?, \\ \langle 1, 6, 2, 7, 1 \rangle \end{array} \right)$



	t	u
t	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + \overleftarrow{C}$
u	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + 1 + \overleftarrow{C}$

Constant-time move probing (Local Search)

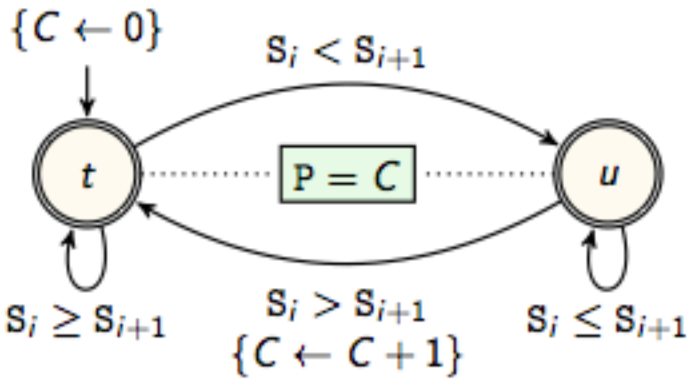
$$\textsc{peak}(\mathrm{P} = 3, \langle 1, 1, 4, 1, 6, 2, 7, 1 \rangle)$$

	1	1	4	1		1	6	2	7	1	
	=	<	<			<	<	>	<		
i	0	1	2	3		4	3	2	1	0	i
$\overrightarrow{Q_i}$	t	t	u	t		t	u	t	u	t	$\overleftarrow{Q_i}$
$\overrightarrow{C_i}$	0	0	0	1		2	1	1	0	0	$\overleftarrow{C_i}$
$\overrightarrow{\mathrm{P}_i}$	0	0	0	1		2	1	1	0	0	$\overleftarrow{\mathrm{P}_i}$

$$\textsc{peak}\left(\begin{array}{c}\overrightarrow{\mathrm{P}_3} = 1,\\ \langle 1, 1, 4, 1 \rangle\end{array}\right) \quad \textsc{peak}\left(\begin{array}{c}\overleftarrow{\mathrm{P}_4} = 2,\\ \langle 1, 6, 2, 7, 1 \rangle\end{array}\right)$$

glue matrix entry associated with the state pair (t, t):

$$\mathrm{P} = \overrightarrow{\mathrm{C}_3} + \overleftarrow{\mathrm{C}_4} = 1 + 2 = 3$$



	t	u
t	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + \overleftarrow{C}$
u	$\overrightarrow{C} + \overleftarrow{C}$	$\overrightarrow{C} + 1 + \overleftarrow{C}$

Conclusion

Given an automaton with accumulators:

- We came up with an **abstraction**, the **glue matrix**, for defining the **relation** between the accumulators on the full sequence, and on its prefixes and suffixes.
- This characterisation is **independent** of the solving technology (it is just about automata with accumulators).
- We show how to exploit the **glue matrix** in the context of **constraint programming** and **local search**.

41 *glue matrices available in the Global Constraint Catalogue*
(see the **glue matrix** keyword)