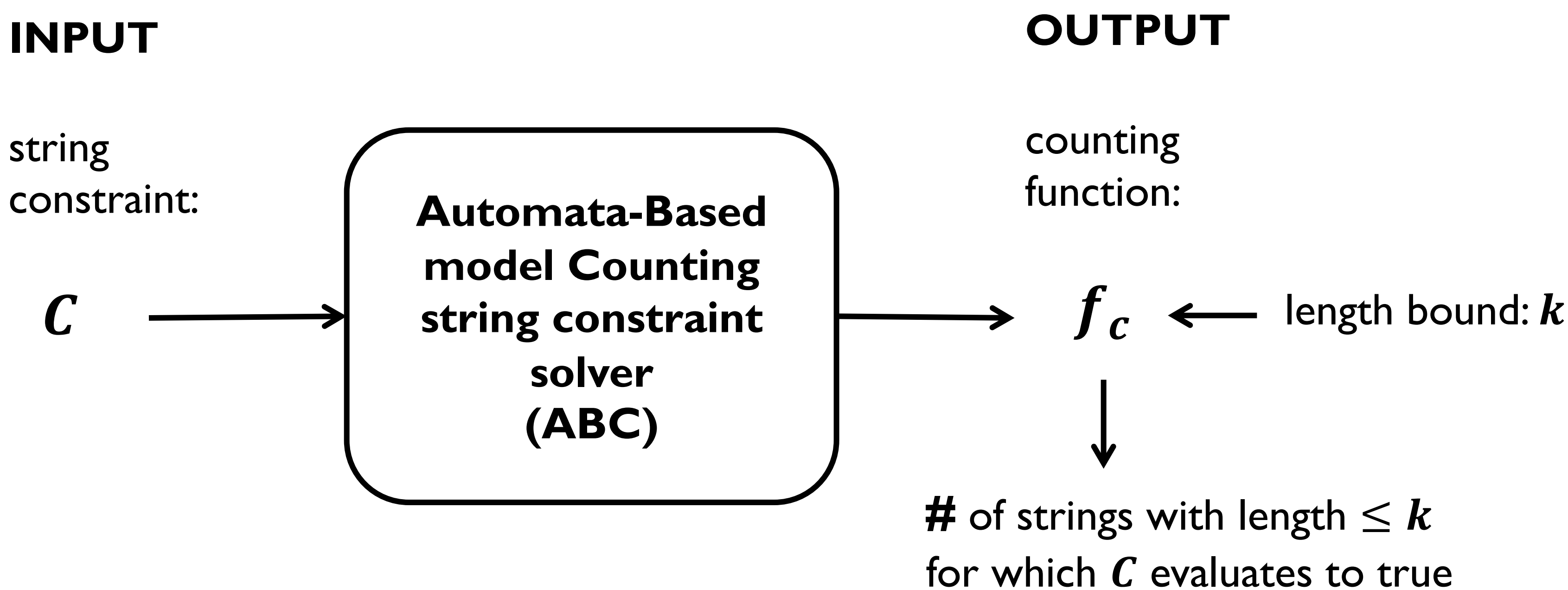
Automata-based Model Counting

Model Counting String Constraint Solver



Automata Based Counter (ABC)
A Model Counting String Constraint Solver

INPUT

string constraint:

C

Automata-Based model Counting string constraint solver (ABC)

OUTPUT

counting function:

f_C ← length bound: k

# of strings with length $\leq k$ for which C evaluates to true

String Constraint Language

$$
\begin{array}{lll}
C & \longrightarrow & \mathit{bterm} \\
 & & \\
\mathit{bterm} & \longrightarrow & \mathtt{v} \mid \mathtt{true} \mid \mathtt{false} \\
 & \mid & \neg \mathit{bterm} \mid \mathit{bterm} \wedge \mathit{bterm} \mid \mathit{bterm} \vee \mathit{bterm} \mid (\mathit{bterm}) \\
 & \mid & \mathit{sterm} = \mathit{sterm} \\
 & \mid & \mathtt{match}(\mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathtt{contains}(\mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathtt{begins}(\mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathtt{ends}(\mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathit{iterm} = \mathit{iterm} \mid \mathit{iterm} < \mathit{iterm} \mid \mathit{iterm} > \mathit{iterm} \\
 & & \\
\mathit{iterm} & \longrightarrow & \mathtt{v} \mid \mathtt{n} \\
 & \mid & \mathit{iterm} + \mathit{iterm} \mid \mathit{iterm} - \mathit{iterm} \mid \mathit{iterm} \times \mathtt{n} \mid (\mathit{iterm}) \\
 & \mid & \mathtt{length}(\mathit{sterm}) \mid \mathtt{toint}(\mathit{sterm}) \\
 & \mid & \mathtt{indexof}(\mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathtt{lastindexof}(\mathit{sterm}, \mathit{sterm}) \\
 & & \\
\mathit{sterm} & \longrightarrow & \mathtt{v} \mid \varepsilon \mid \mathtt{s} \\
 & \mid & \mathit{sterm}.\mathit{sterm} \mid \mathit{sterm}|\mathit{sterm} \mid \mathit{sterm}^* \mid (\mathit{sterm}) \\
 & \mid & \mathtt{charat}(\mathit{sterm}, \mathit{iterm}) \mid \mathtt{tostring}(\mathit{iterm}) \\
 & \mid & \mathtt{toupper}(\mathit{sterm}) \mid \mathtt{tolower}(\mathit{sterm}) \\
 & \mid & \mathtt{substring}(\mathit{sterm}, \mathit{iterm}, \mathit{iterm}) \\
 & \mid & \mathtt{replacefirst}(\mathit{sterm}, \mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathtt{replacelast}(\mathit{sterm}, \mathit{sterm}, \mathit{sterm}) \\
 & \mid & \mathtt{replaceall}(\mathit{sterm}, \mathit{sterm}, \mathit{sterm})
\end{array}
$$

ABC: Constraint language

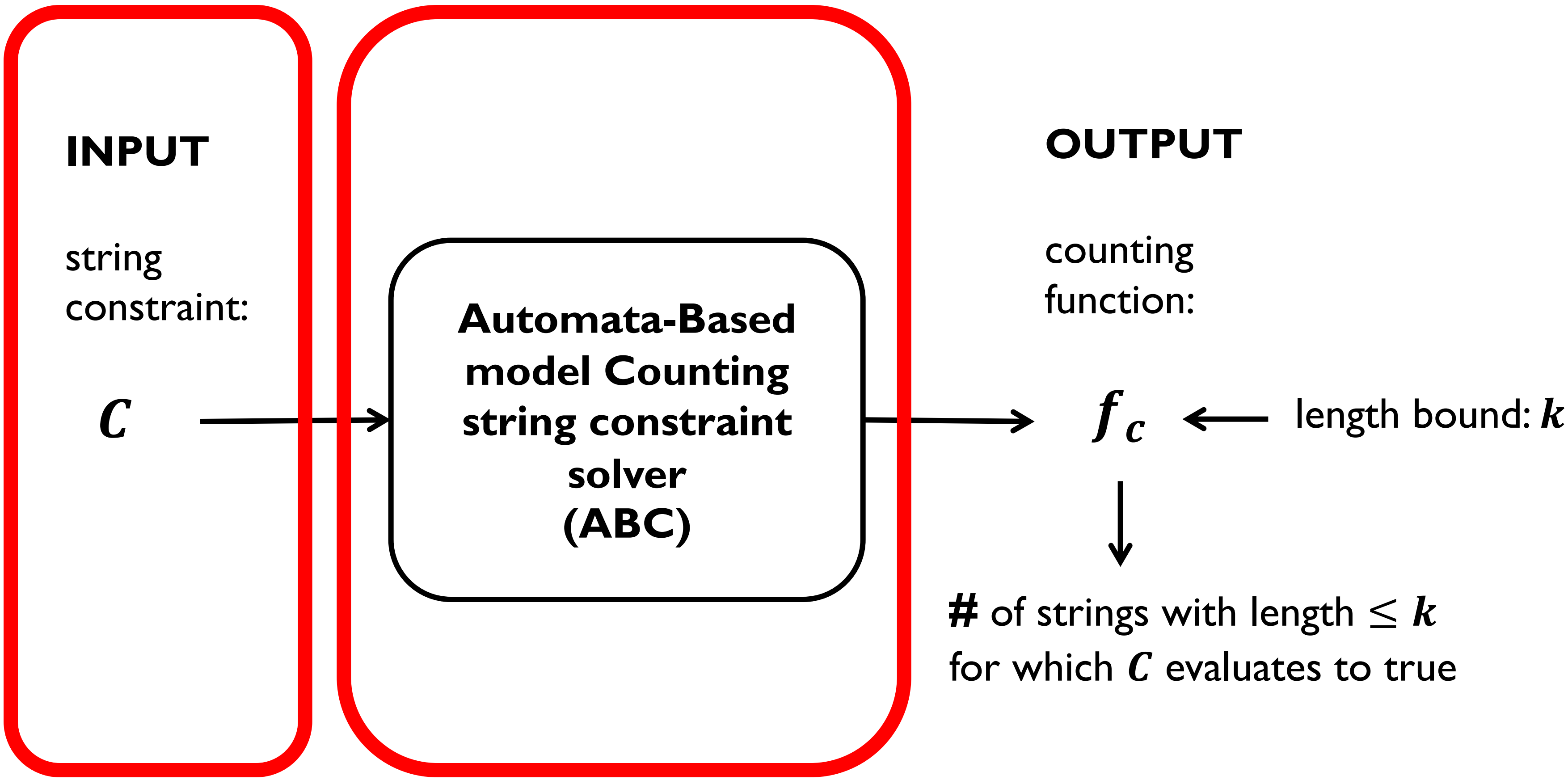
- A more compact notation

$$
\begin{array}{lcl}
\varphi & \longrightarrow & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi_{\mathbb{Z}} \mid \varphi_{\mathbb{S}} \mid \top \mid \bot \\
\varphi_{\mathbb{Z}} & \longrightarrow & \beta = \beta \mid \beta < \beta \mid \beta > \beta \\
\varphi_{\mathbb{S}} & \longrightarrow & \gamma = \gamma \mid \gamma < \gamma \mid \gamma > \gamma \mid \mathrm{match}(\gamma, \rho) \mid \mathrm{contains}(\gamma, \gamma) \mid \mathrm{begins}(\gamma, \gamma) \mid \mathrm{ends}(\gamma, \gamma) \\
\beta & \longrightarrow & v_i \mid n \mid \beta + \beta \mid \beta - \beta \mid \beta \times n \\
 & & \mid \mathrm{length}(\gamma) \mid \mathrm{toint}(\gamma) \mid \mathrm{indexof}(\gamma, \gamma) \mid \mathrm{lastindexof}(\gamma, \gamma) \\
\gamma & \longrightarrow & v_s \mid \rho \mid \gamma \cdot \gamma \mid \mathrm{reverse}(\gamma) \mid \mathrm{tostring}(\beta) \mid \mathrm{charat}(\gamma, \beta) \mid \mathrm{toupper}(\gamma) \mid \mathrm{tolower}(\gamma) \\
 & \mid & \mathrm{substring}(\gamma, \beta, \beta) \mid \mathrm{replacefirst}(\gamma, \gamma, \gamma) \mid \mathrm{replacelast}(\gamma, \gamma, \gamma) \mid \mathrm{replaceall}(\gamma, \gamma, \gamma) \\
\rho & \longrightarrow & \varepsilon \mid s \mid \rho \cdot \rho \mid \rho \shortmid \rho \mid \rho^*
\end{array}
$$

Example String Expressions

<table>
<tr><th></th><th>String Expression</th><th>Constraint Language</th></tr>
<tr><td rowspan="5">Java</td><td>s.length()</td><td>length(s)</td></tr>
<tr><td>s.isEmpty()</td><td>length(s) == 0</td></tr>
<tr><td>s.startsWith(t,n)</td><td>0 ≤ n ∧ n ≤ |s| ∧
begins(substring(s,n,|s|),t)</td></tr>
<tr><td>s.indexOf(t,n)</td><td>indexof(substring(s,n,|s|),t)</td></tr>
<tr><td>s.replaceAll(p,r)</td><td>replaceall(s,p,r)</td></tr>
<tr><td rowspan="4">PHP</td><td>strrpos(s, t)</td><td>lastindexof(s,t)</td></tr>
<tr><td>substr_replace(s,
t,i,j)</td><td>substring(s,0,i).t.substring(s,j,|s|)</td></tr>
<tr><td>strip_tags(s)</td><td>replaceall(s,("<a>"|"<p>"|...),"")</td></tr>
<tr><td>mysql_real_escape
_string(s)</td><td>...replaceall(s
,replaceall(s,“\\",“\\\\")
,"’", “\’")...</td></tr>
</table>

Model Counting String Constraint Solver



ABC in a nutshell

Automata-based constraint solving

Why?

ABC in a nutshell

Automata-based constraint solving

Basic idea:

Constructing an automaton for the set of solutions of a constraint reduces model counting problem to path counting!

Automata-based constraint solving

Generate automaton that accepts satisfying solutions for the constraint

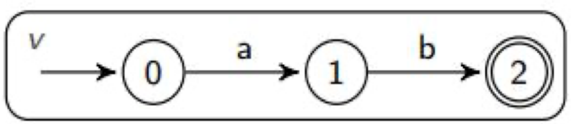
ABC can handle both
string and **integer** constraints

- Constraints over only string variables (e.g., v = “abcd”)
- Constraints over only integer variables (e.g., $i = 2 \times j$)
- Constraints over both string and integer variables (e.g., length(v) = i)

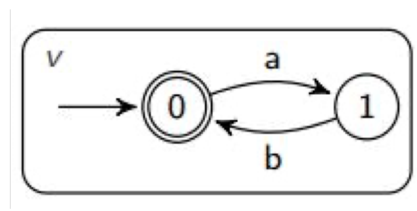
Automata-based constraint solving: expr, ¬

Basic string constraints are directly mapped to automata

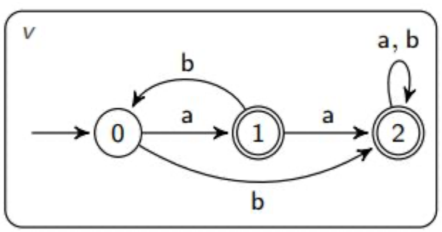
`v = "ab"`



`match(v, (ab)*)`



`¬match(v, (ab)*)`

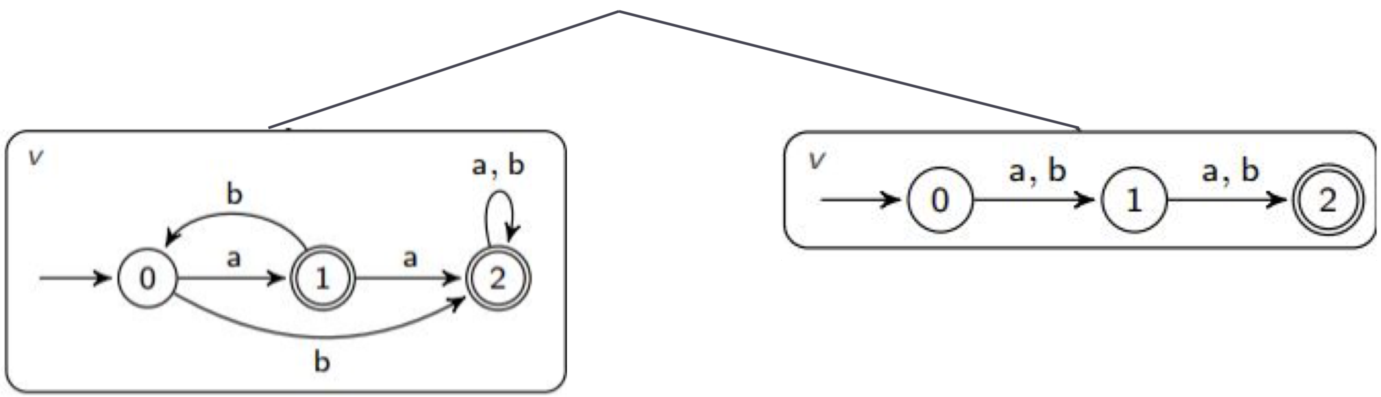


automata
complement

Automata-based constraint solving: expr, ¬, ∧, ∨

More complex constraints are solved by creating automata for subformulae then combining their results

`¬match(v, (ab)*) ∧ length(v) = 2`

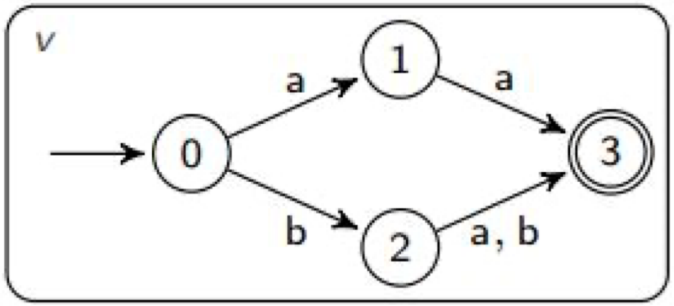


automata product

Automata-based constraint solving: expr, ¬, ∧, ∨

More complex constraints are solved by creating automata for subformulae then combining their results

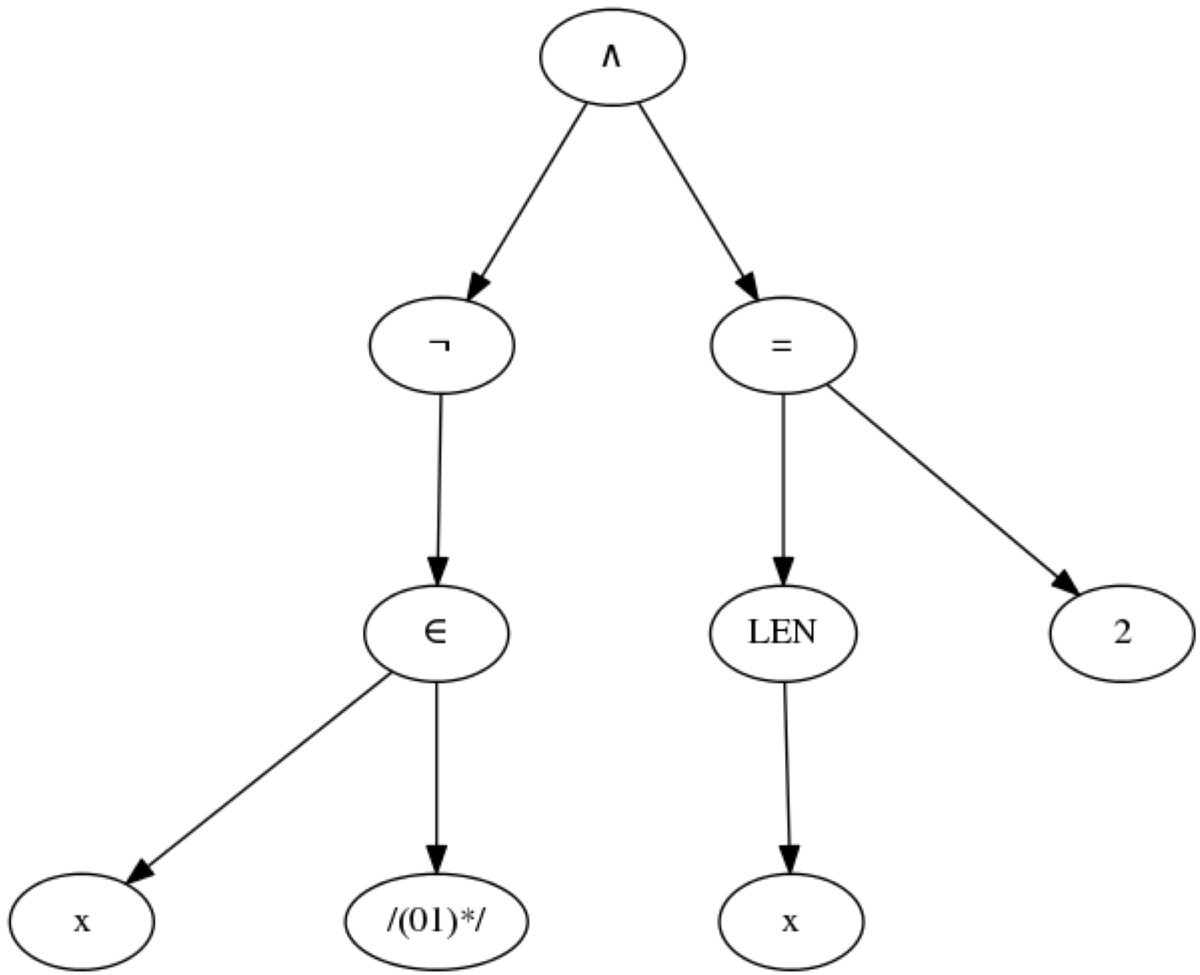
```
¬match(v, (ab)*) ∧ length(v) = 2
```



automata product

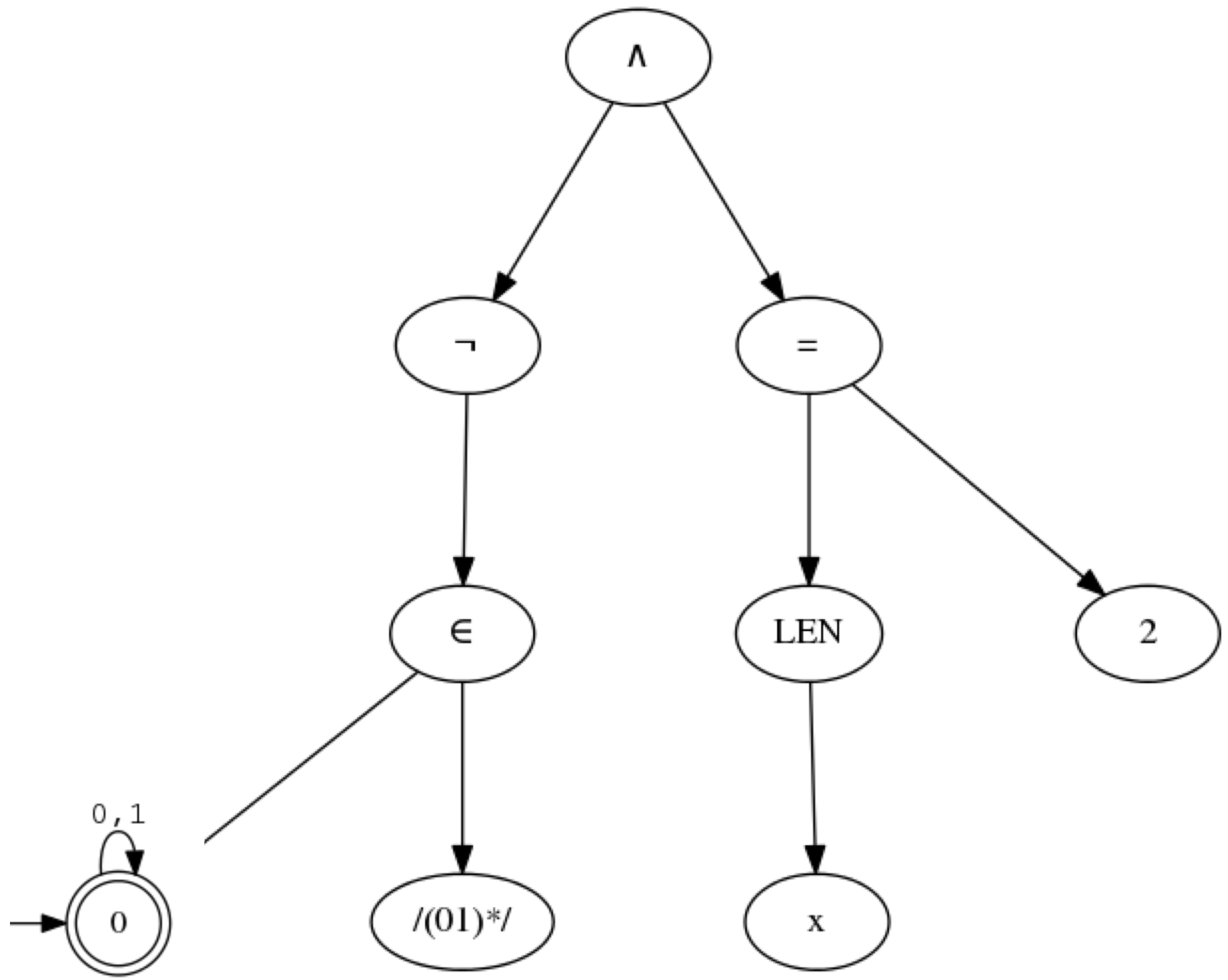
String Automata Construction: More Details

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



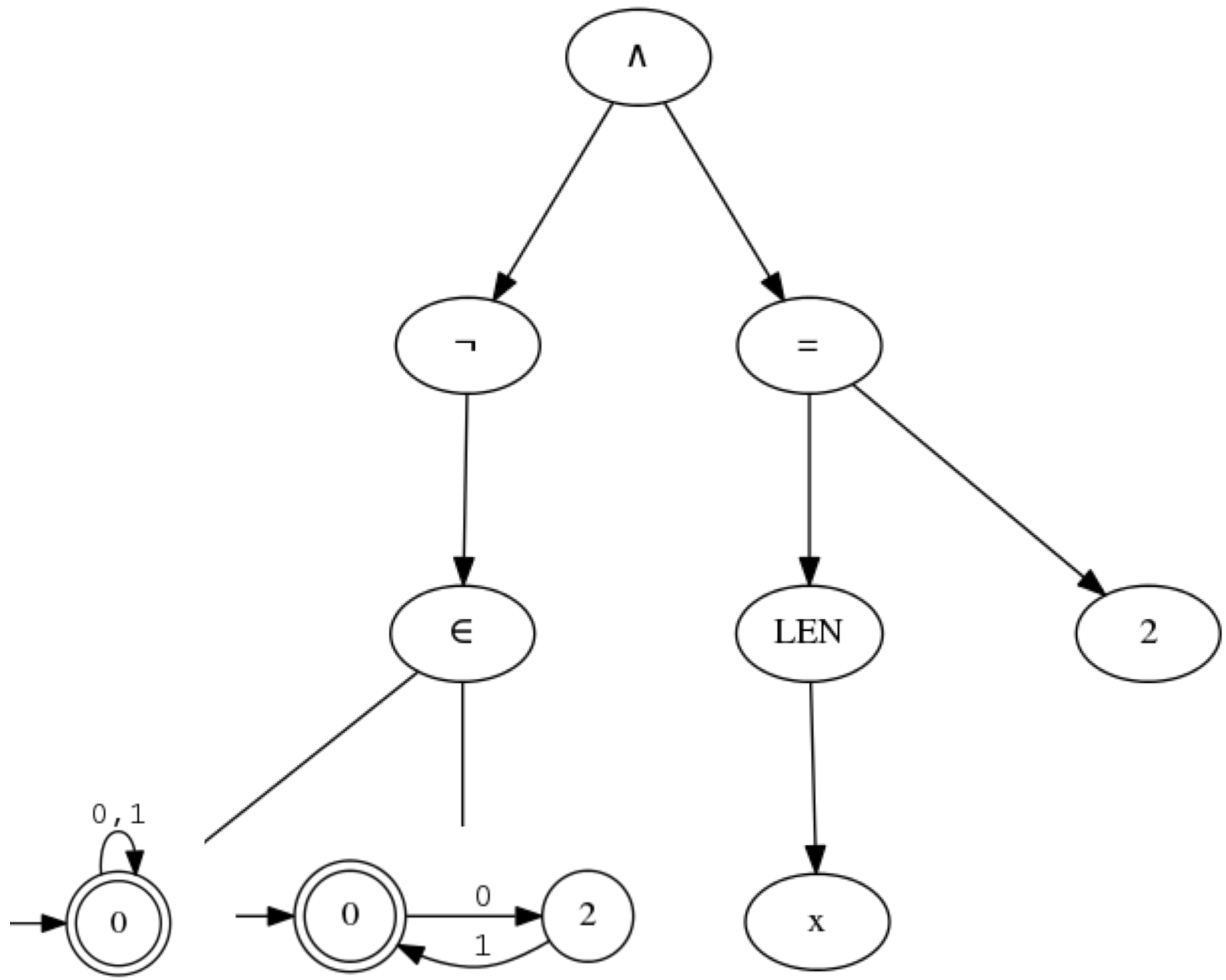
String Automata Construction

$$C \equiv \neg(x \in (01)^{*}) \wedge LEN(x) = 2$$



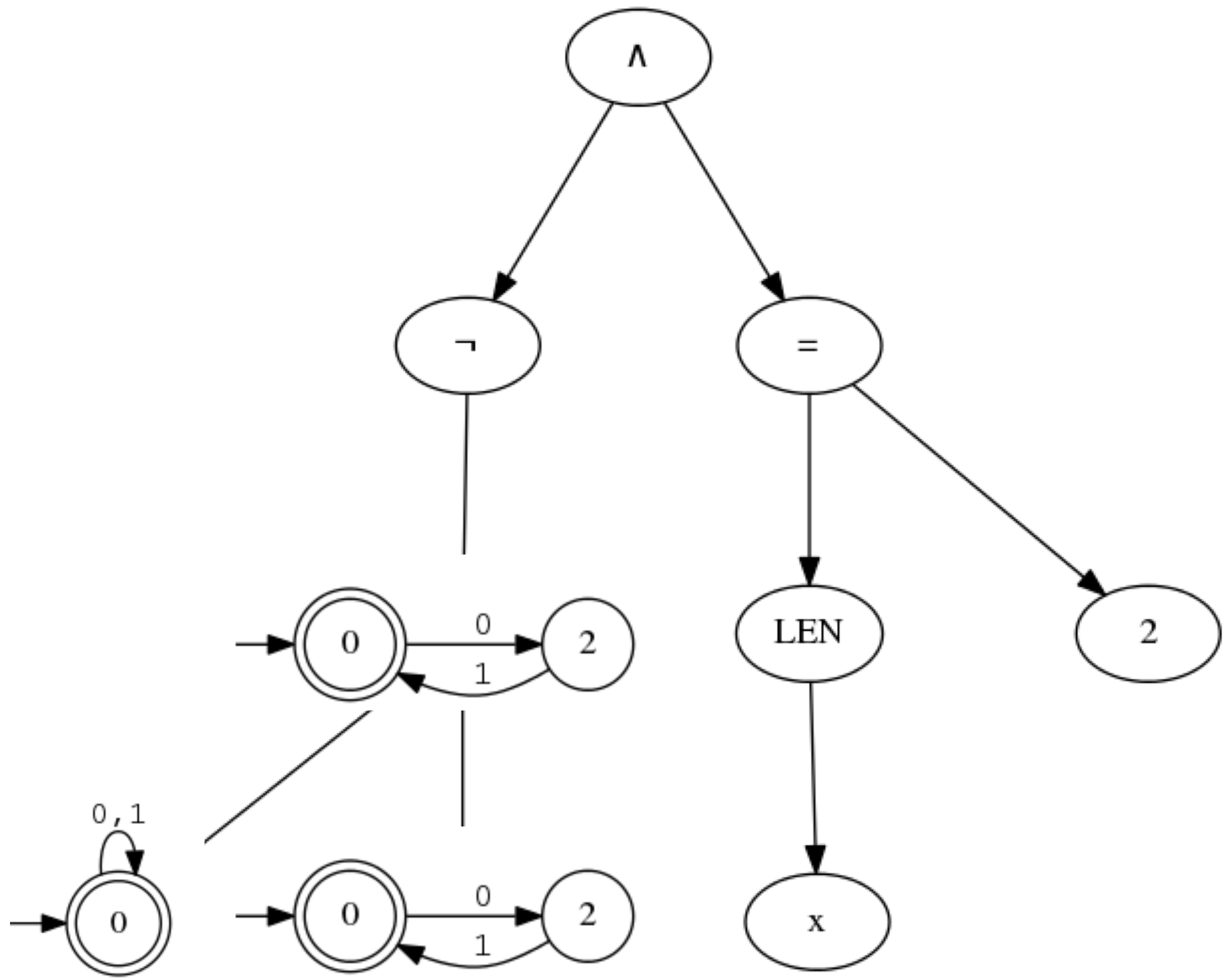
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



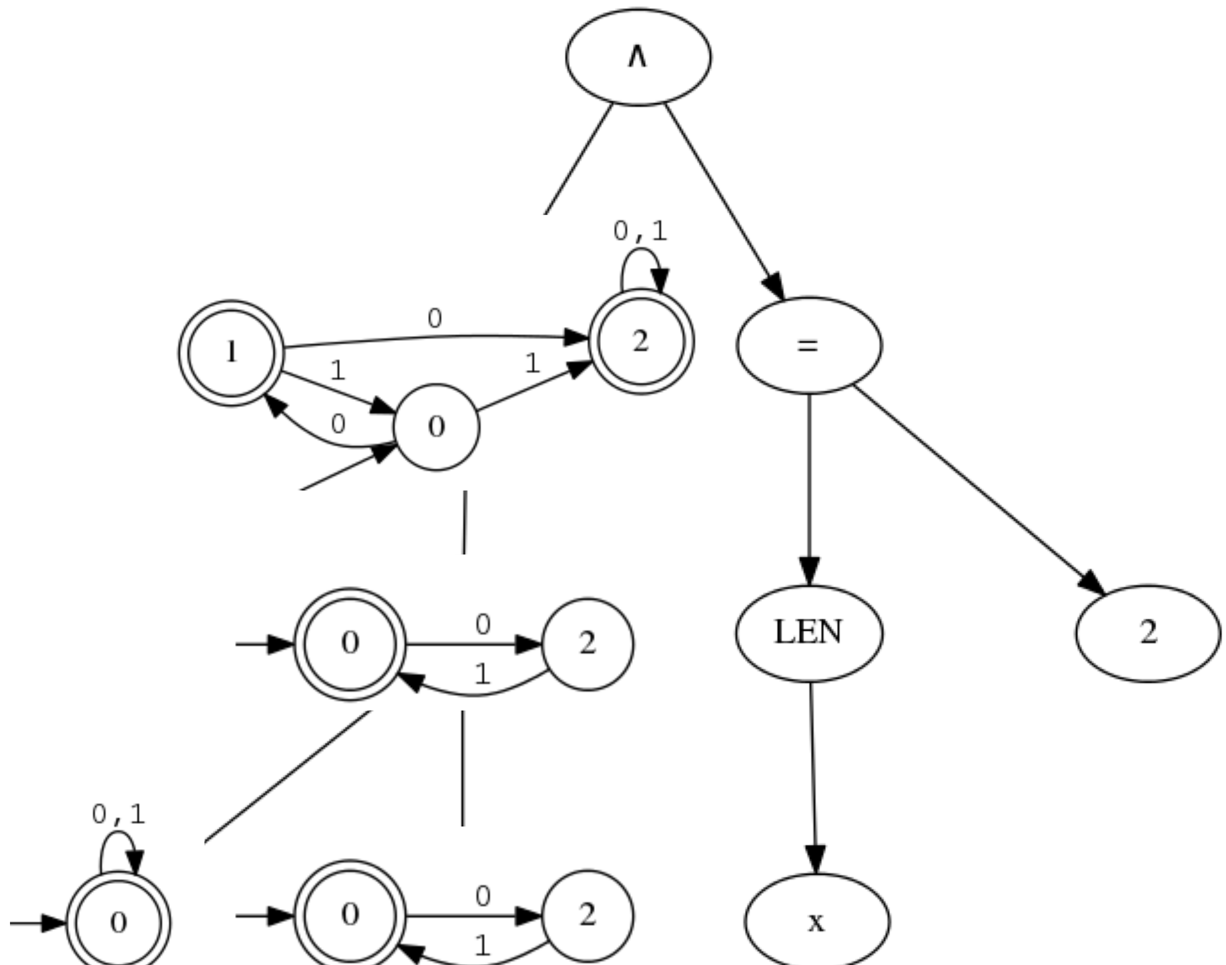
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



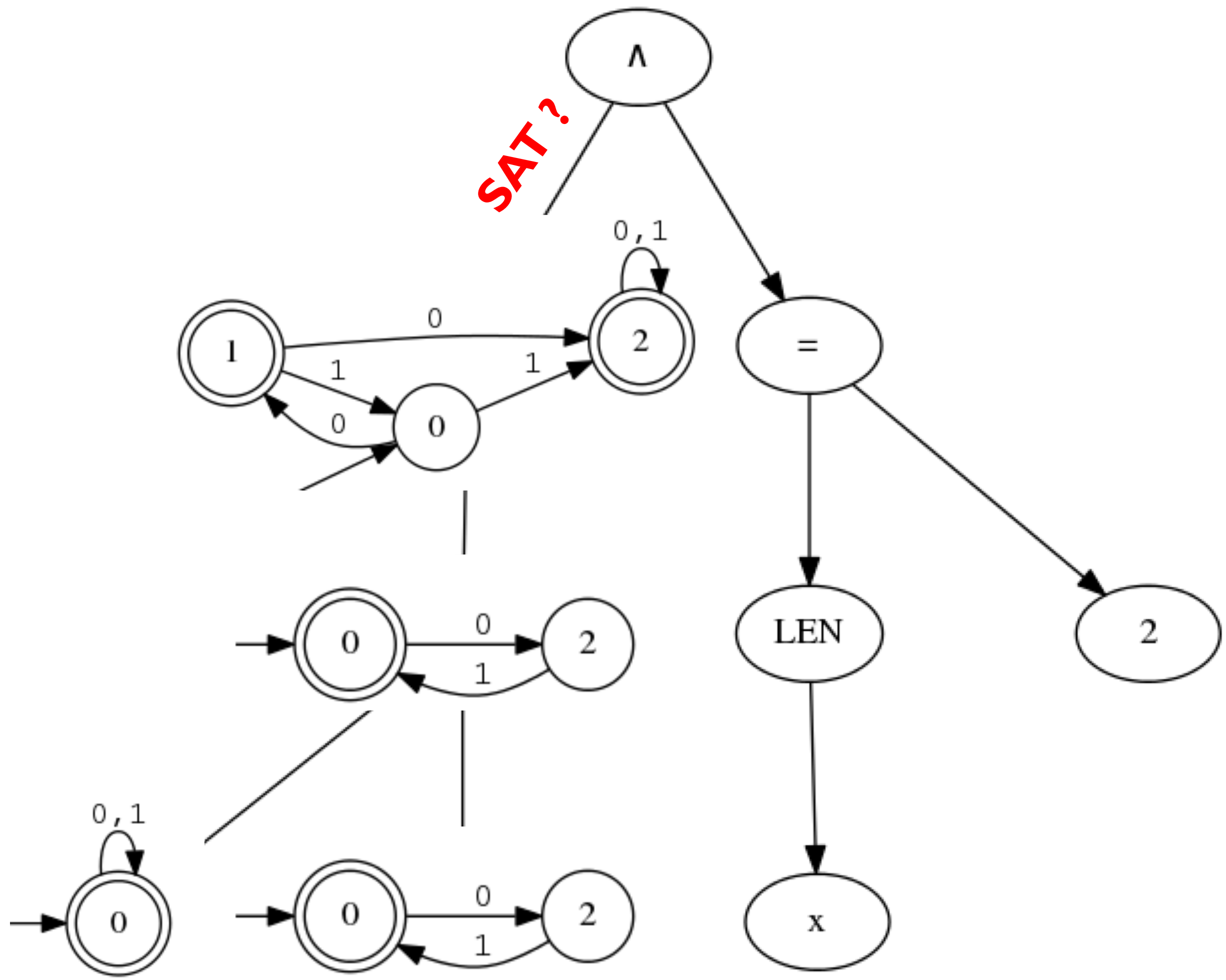
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



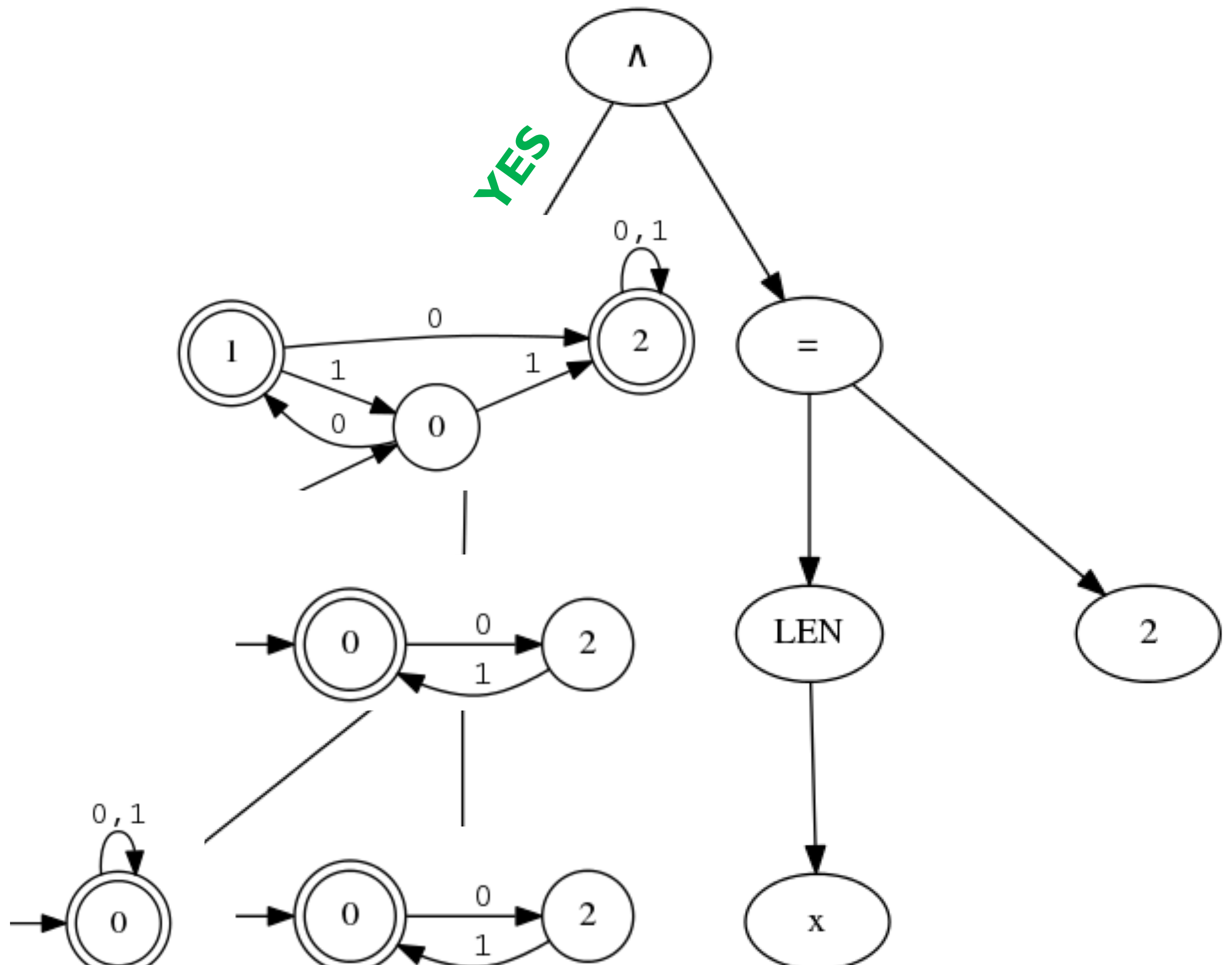
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



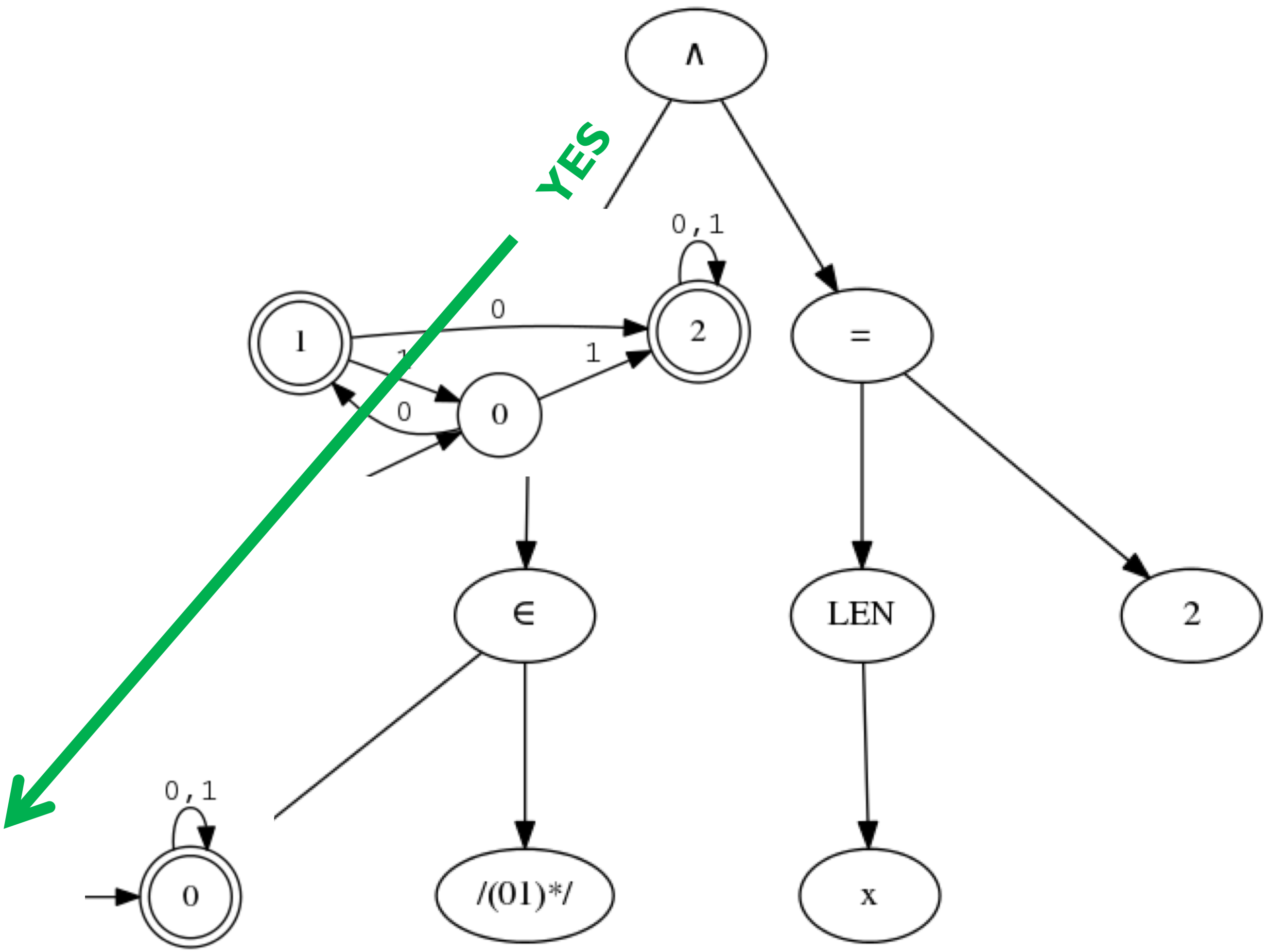
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



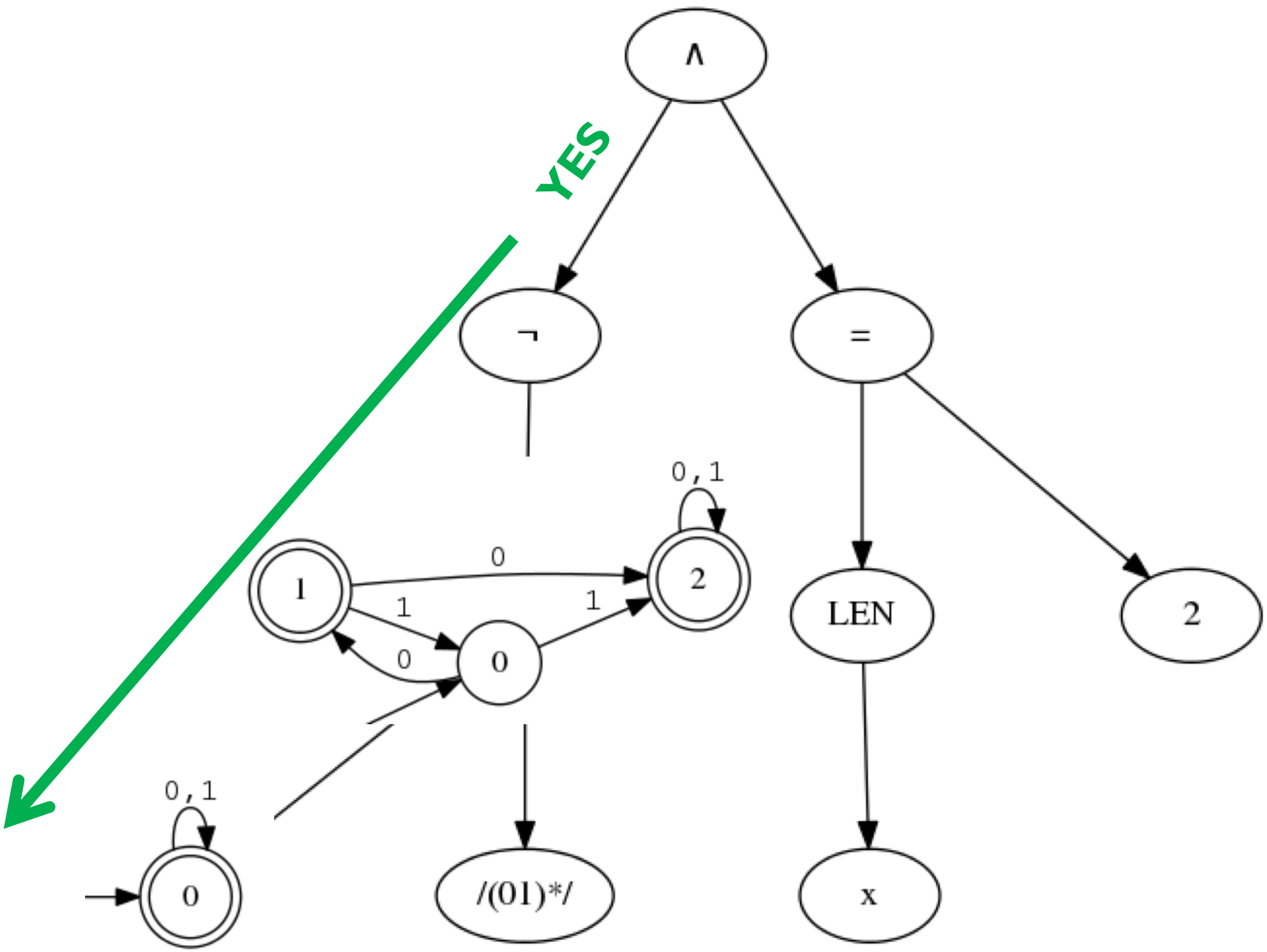
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



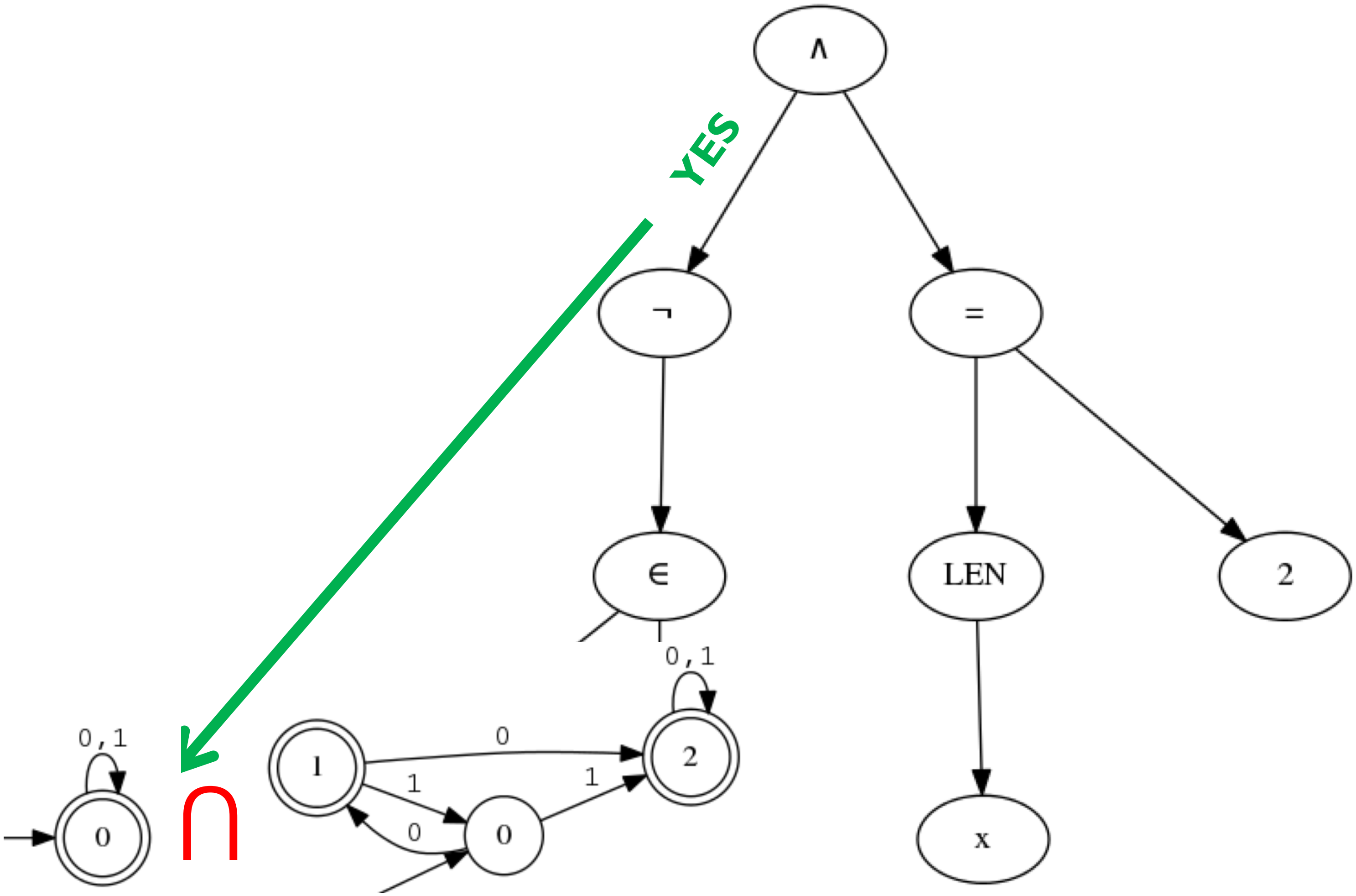
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



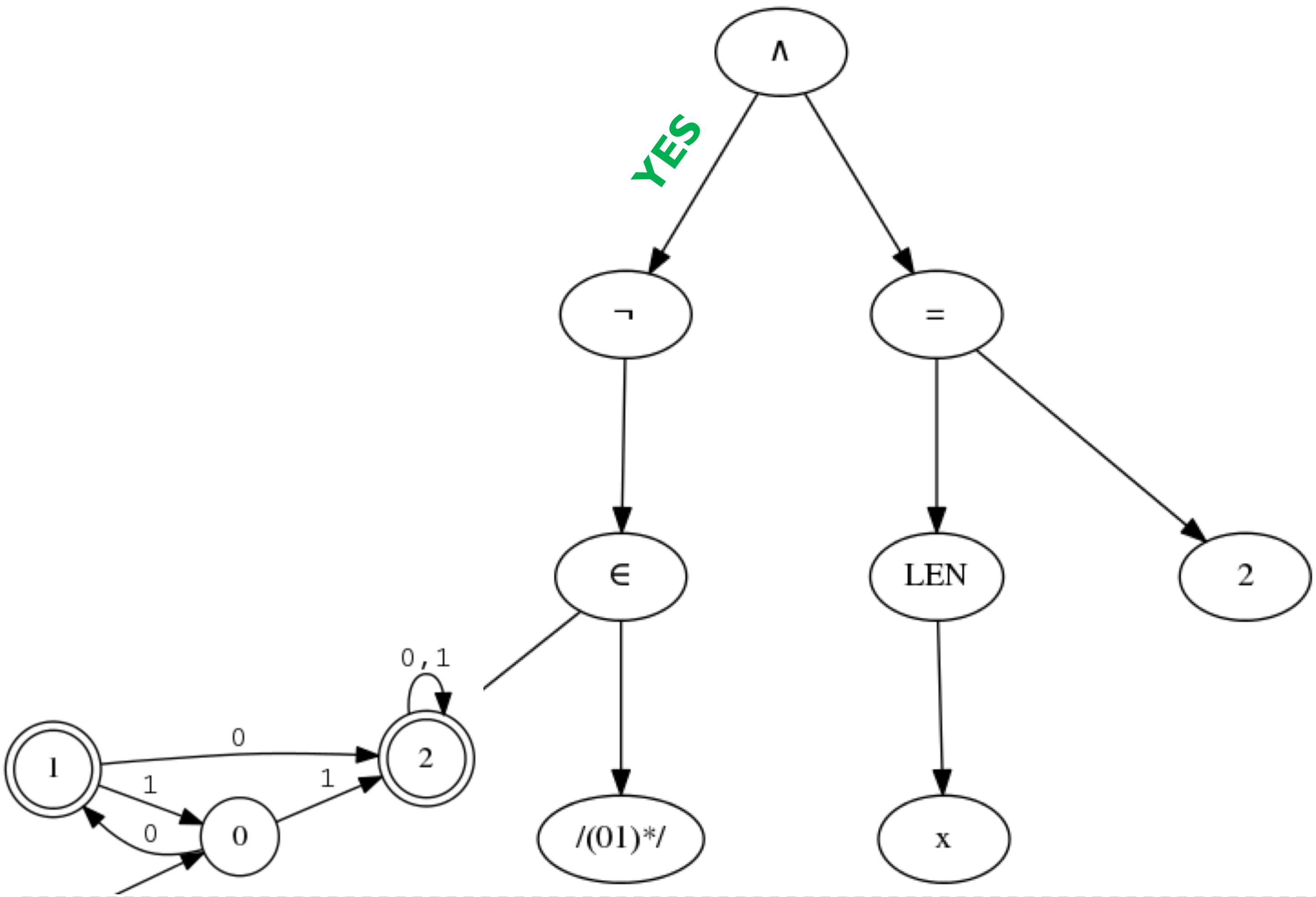
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \land LEN(x) = 2$$



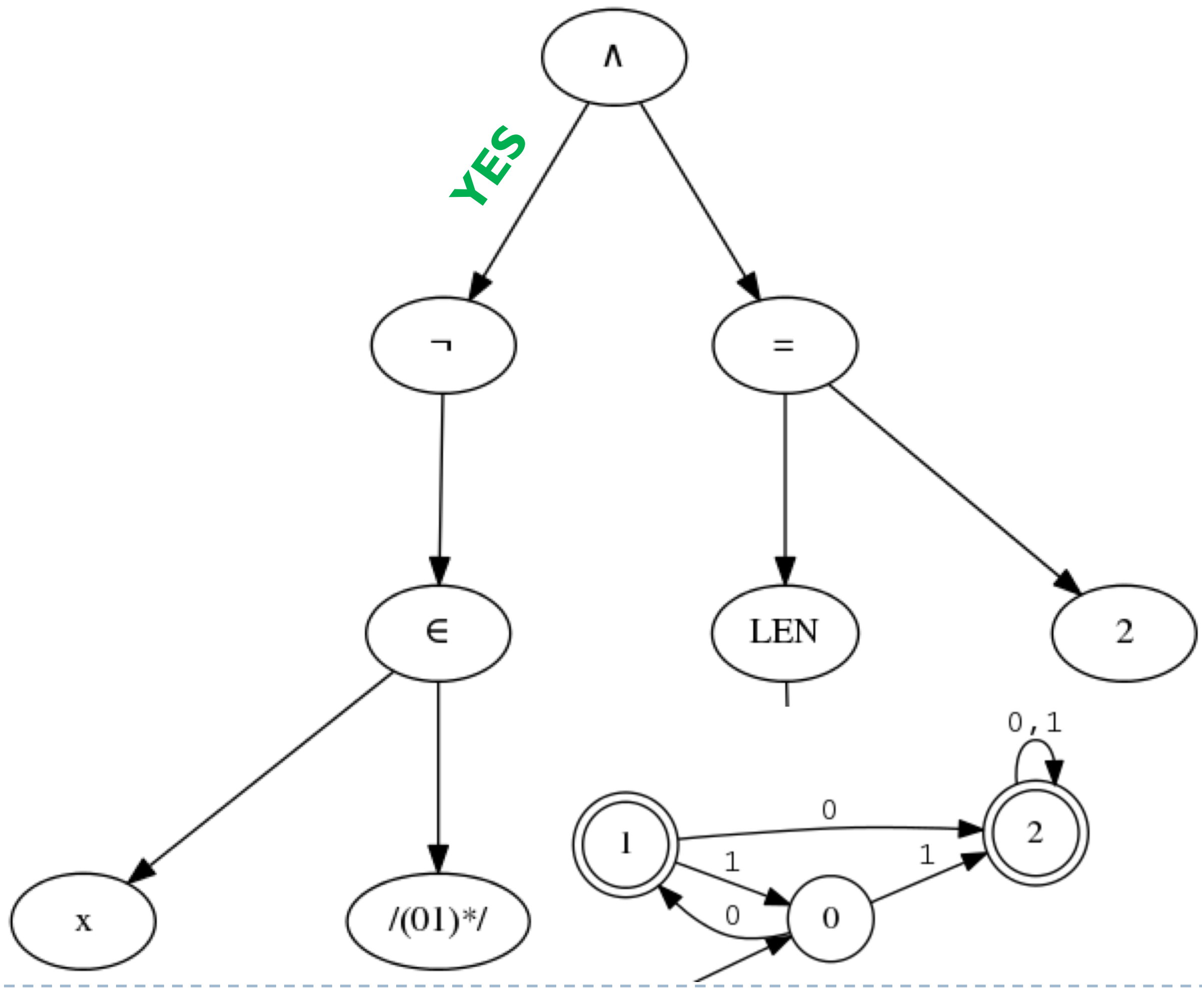
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \land LEN(x) = 2$$



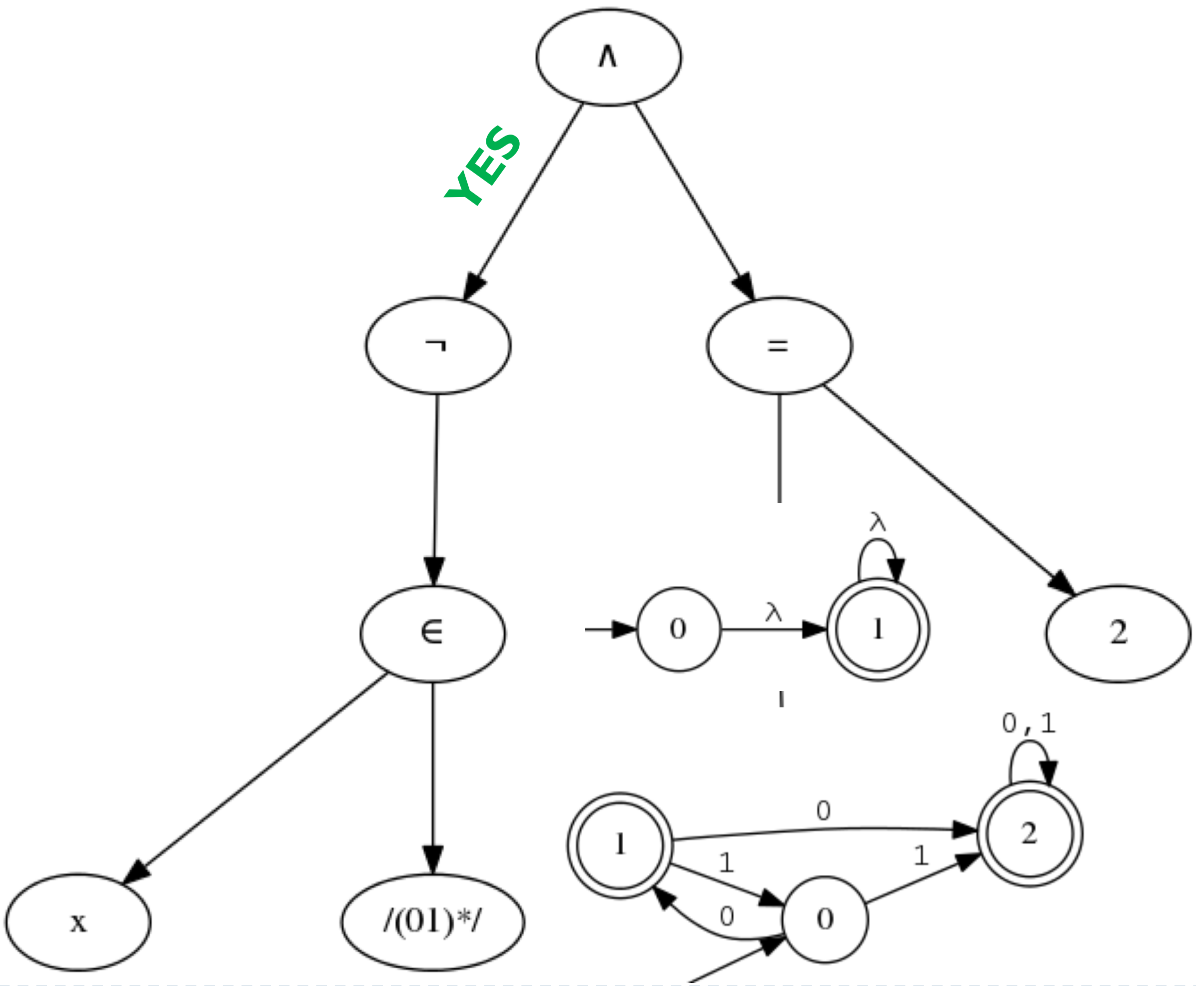
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



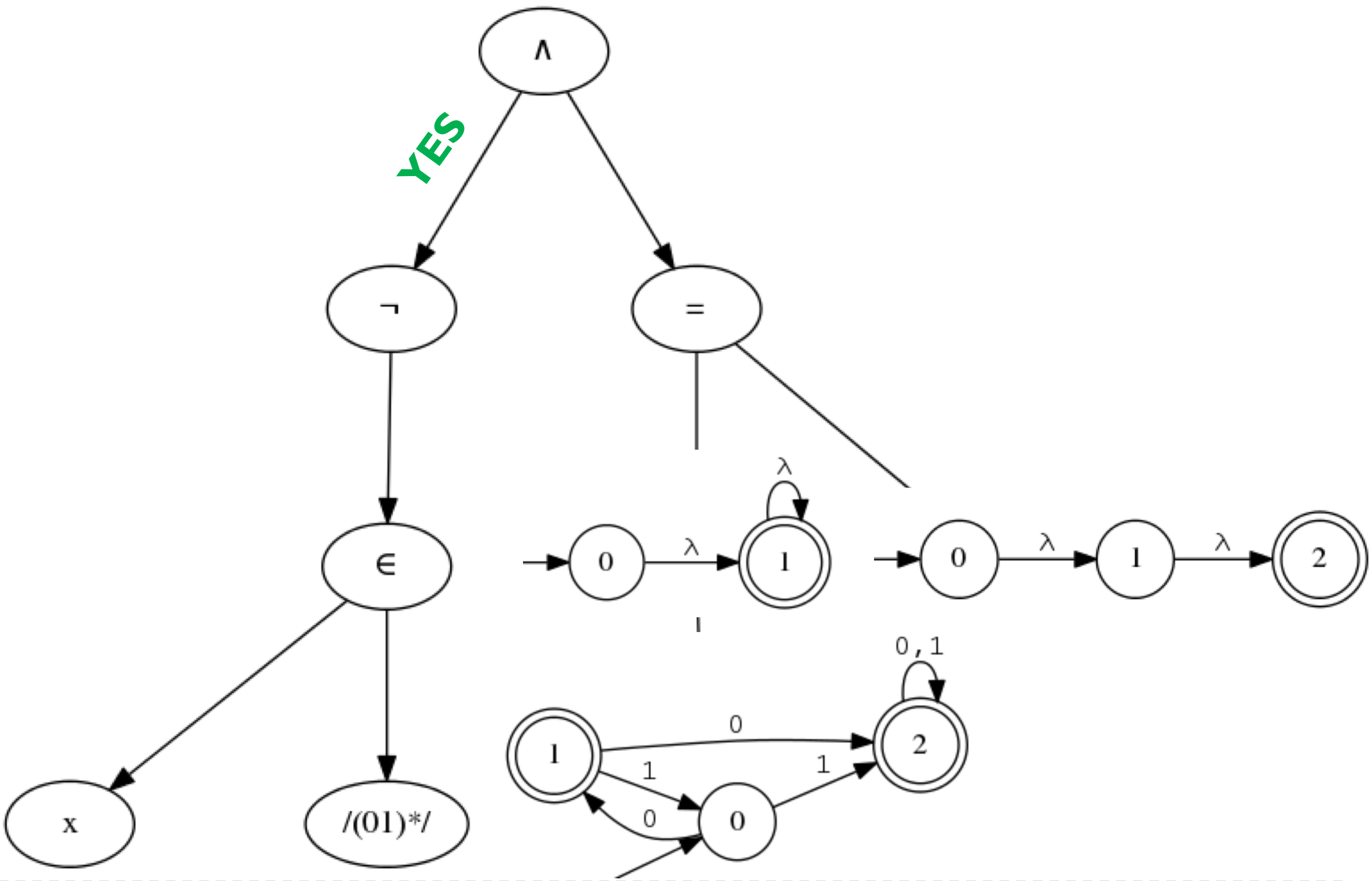
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



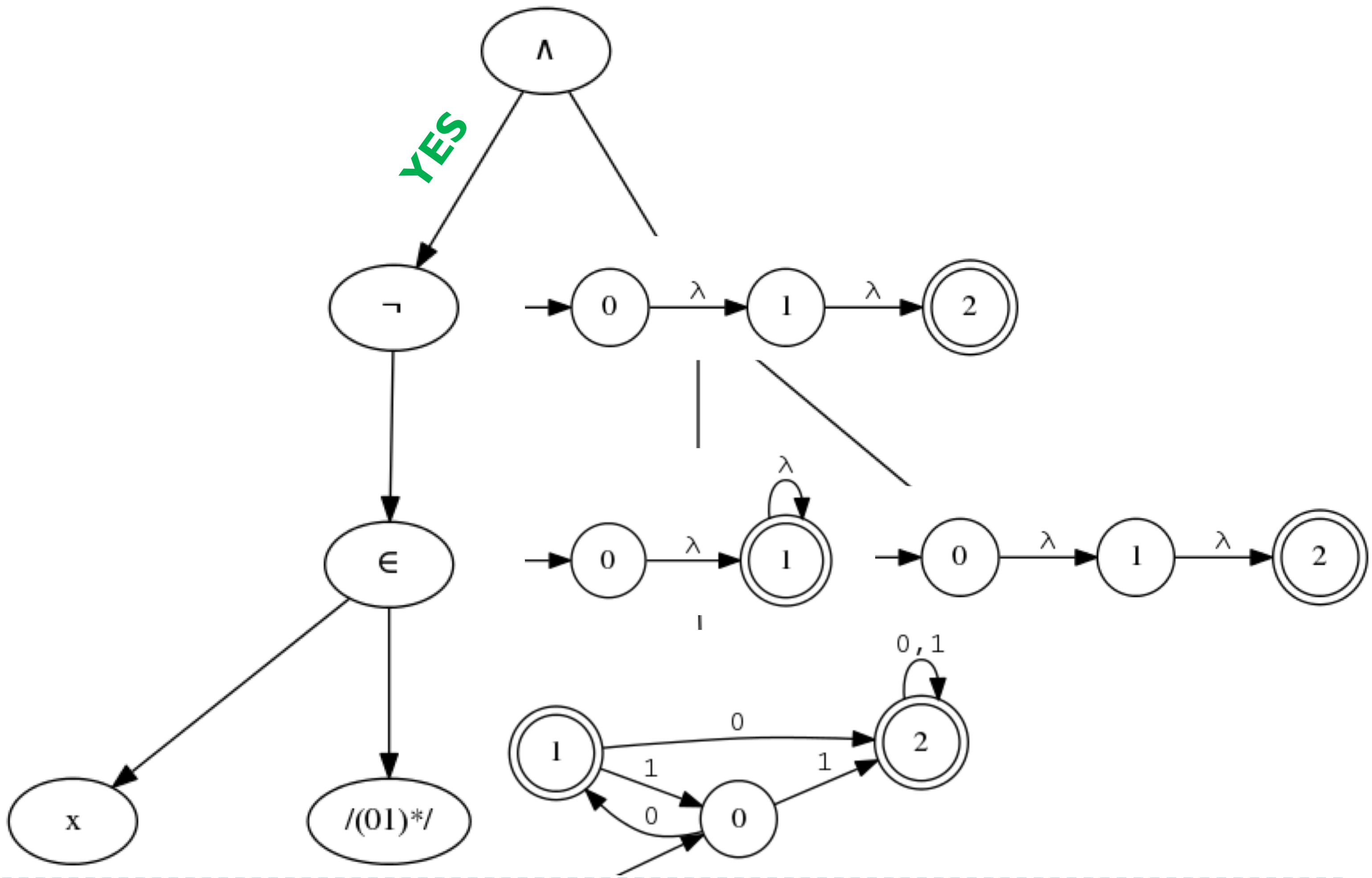
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



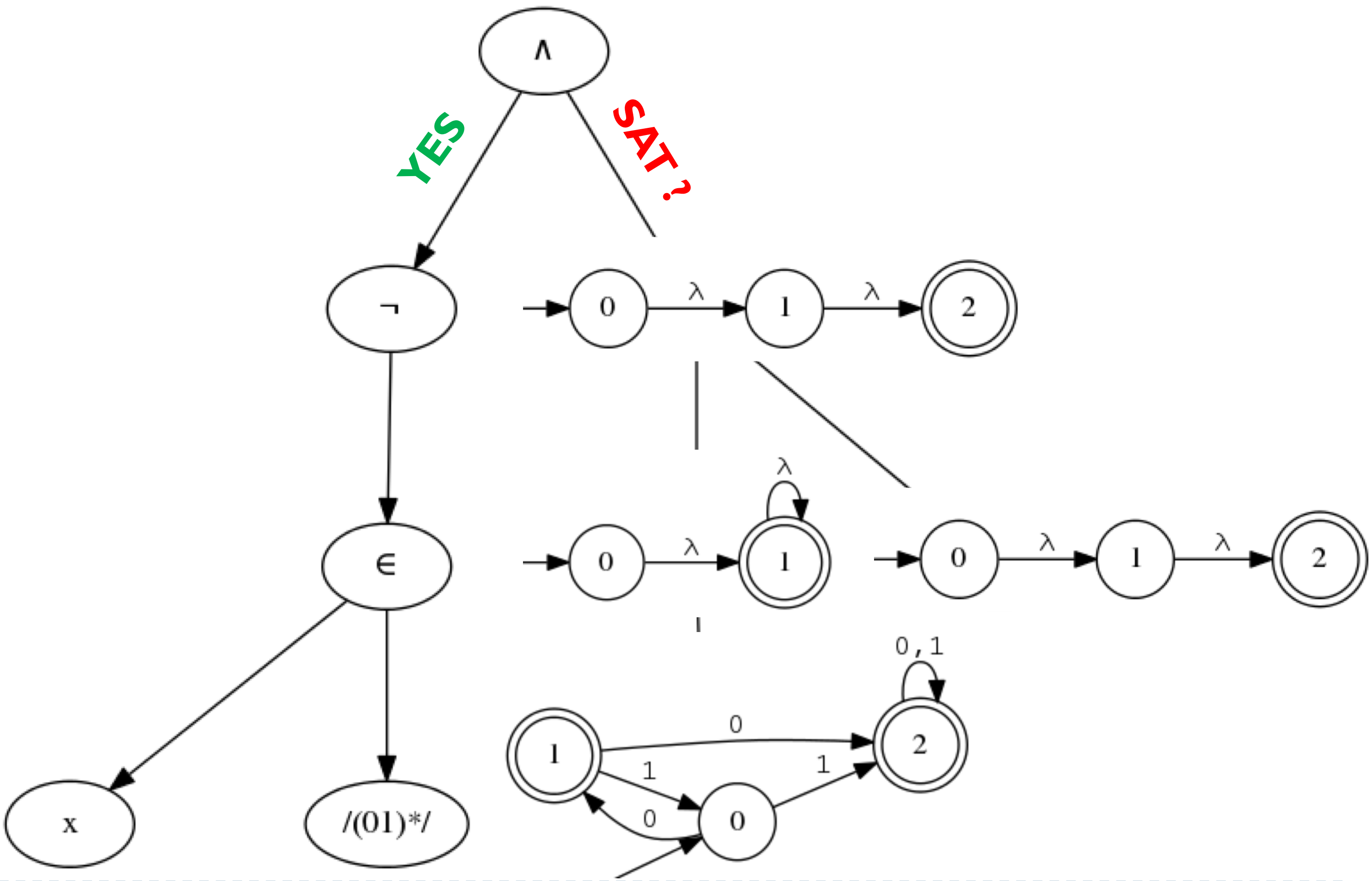
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



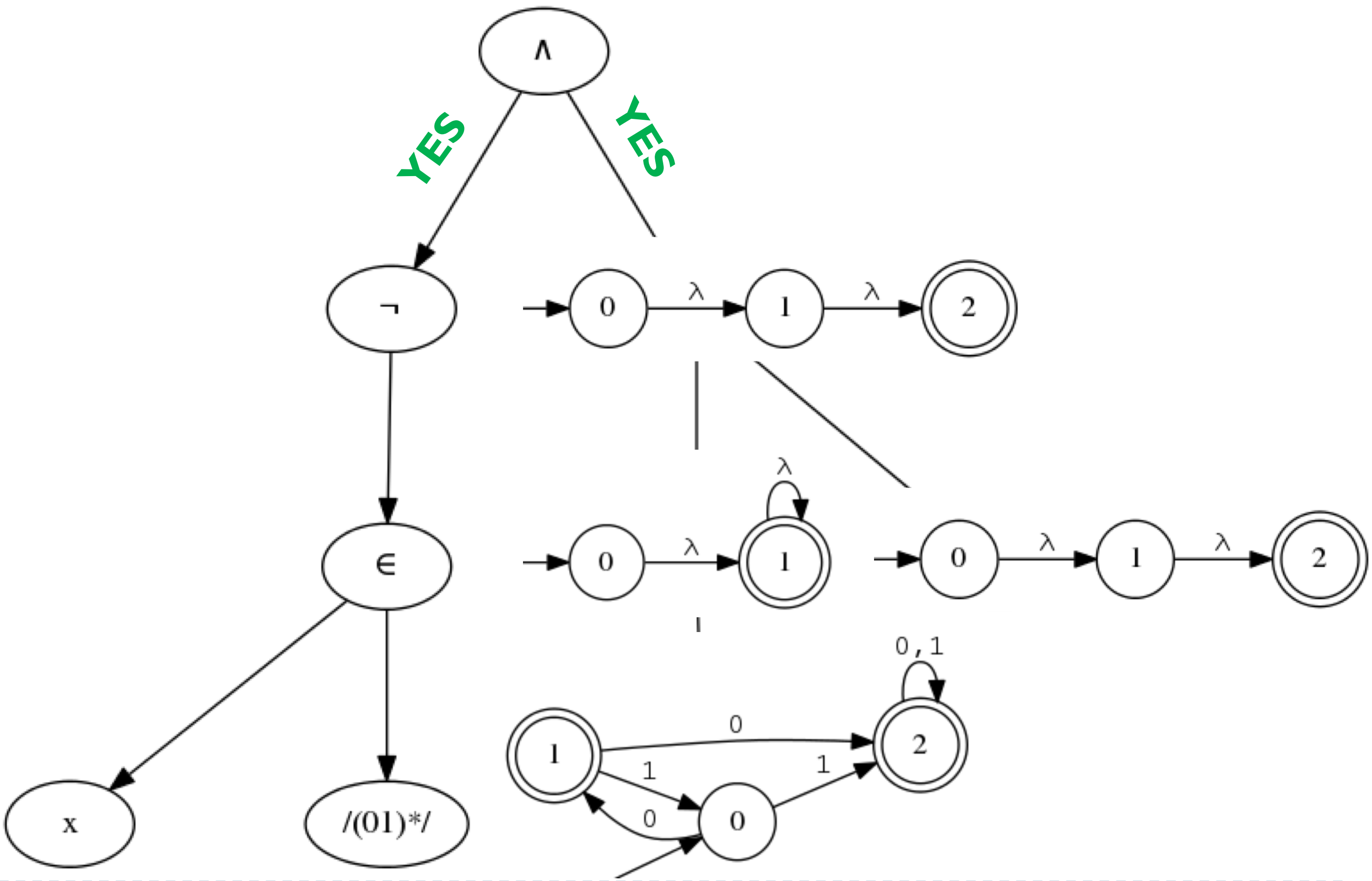
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



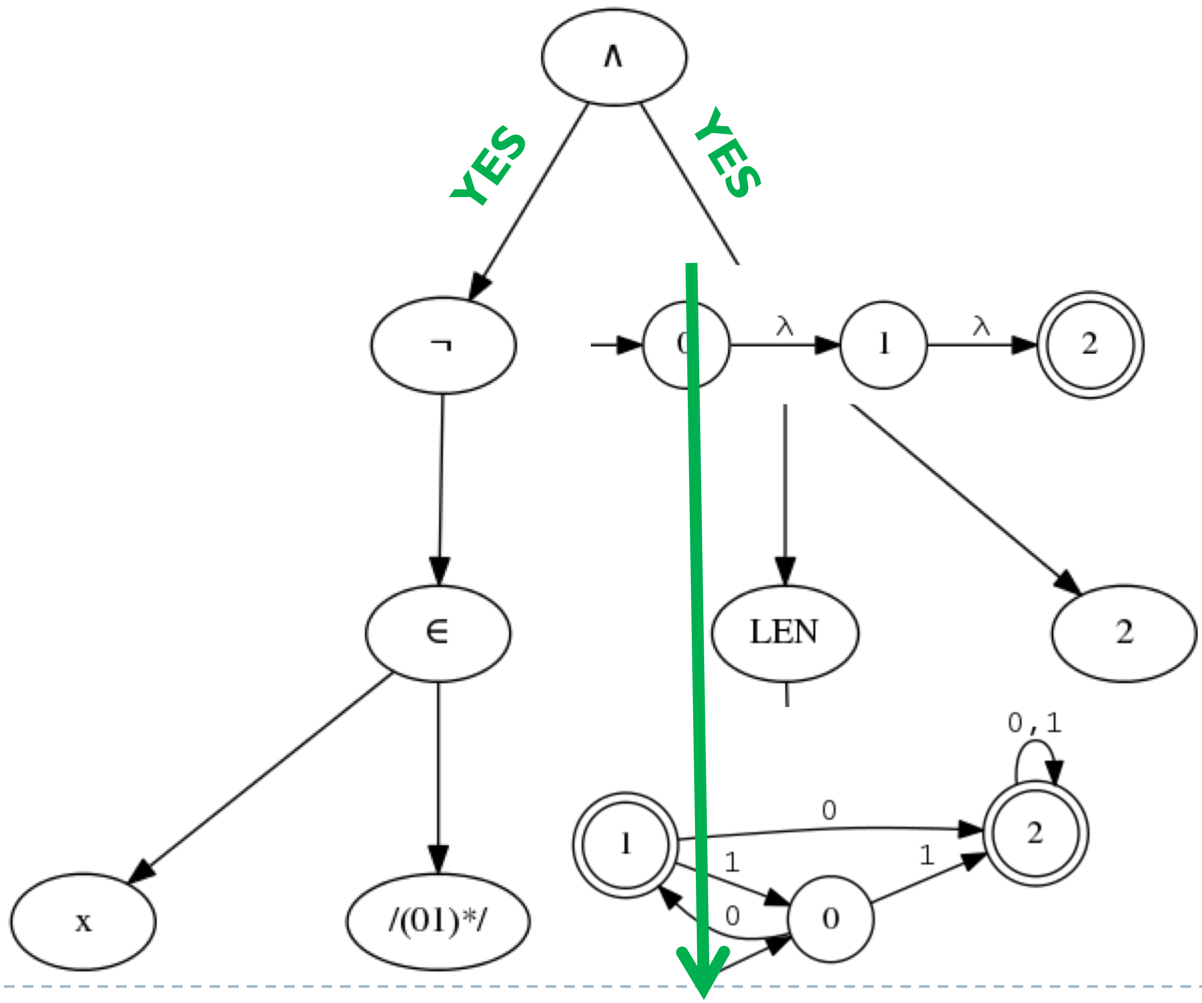
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \land LEN(x) = 2$$



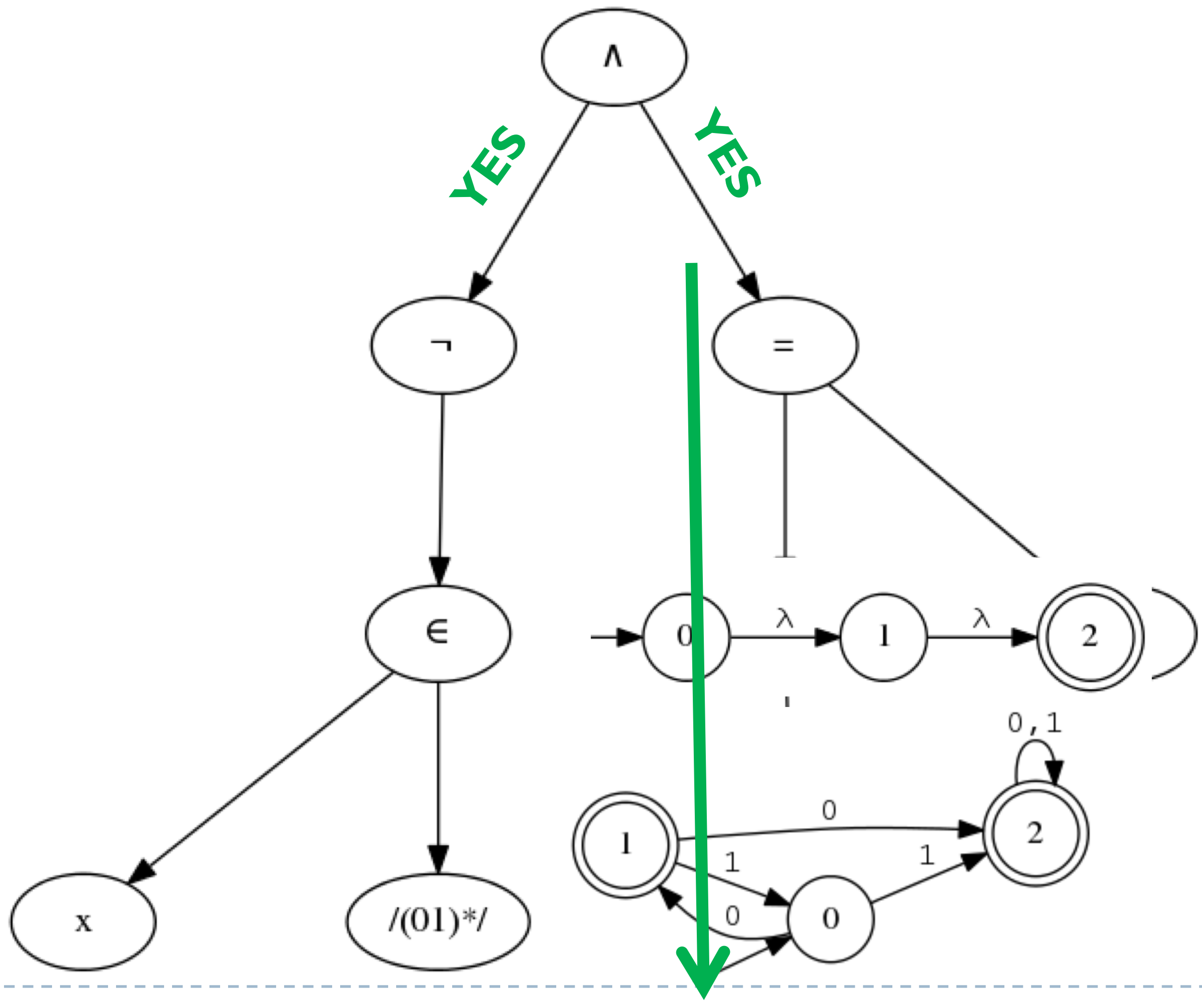
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



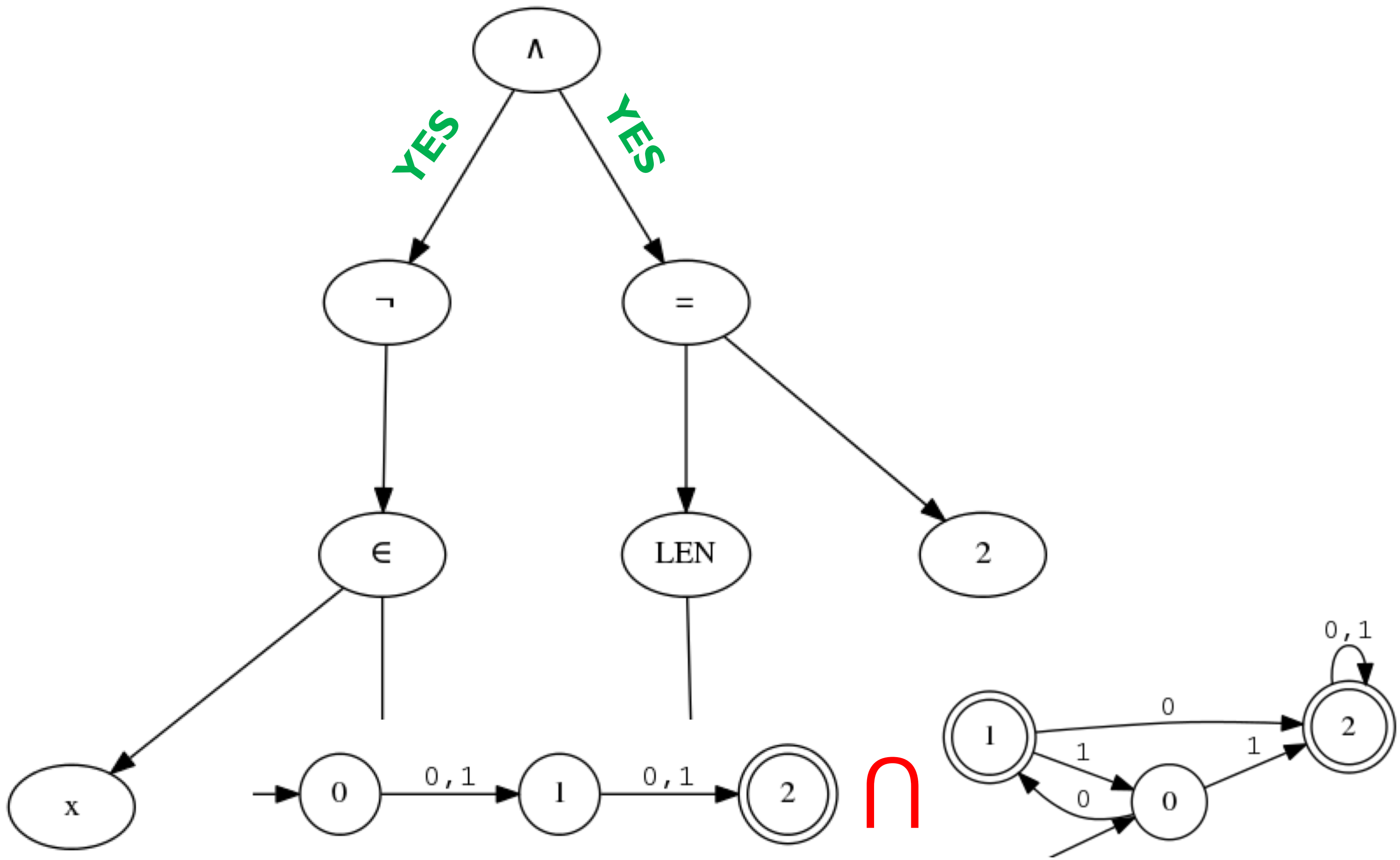
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



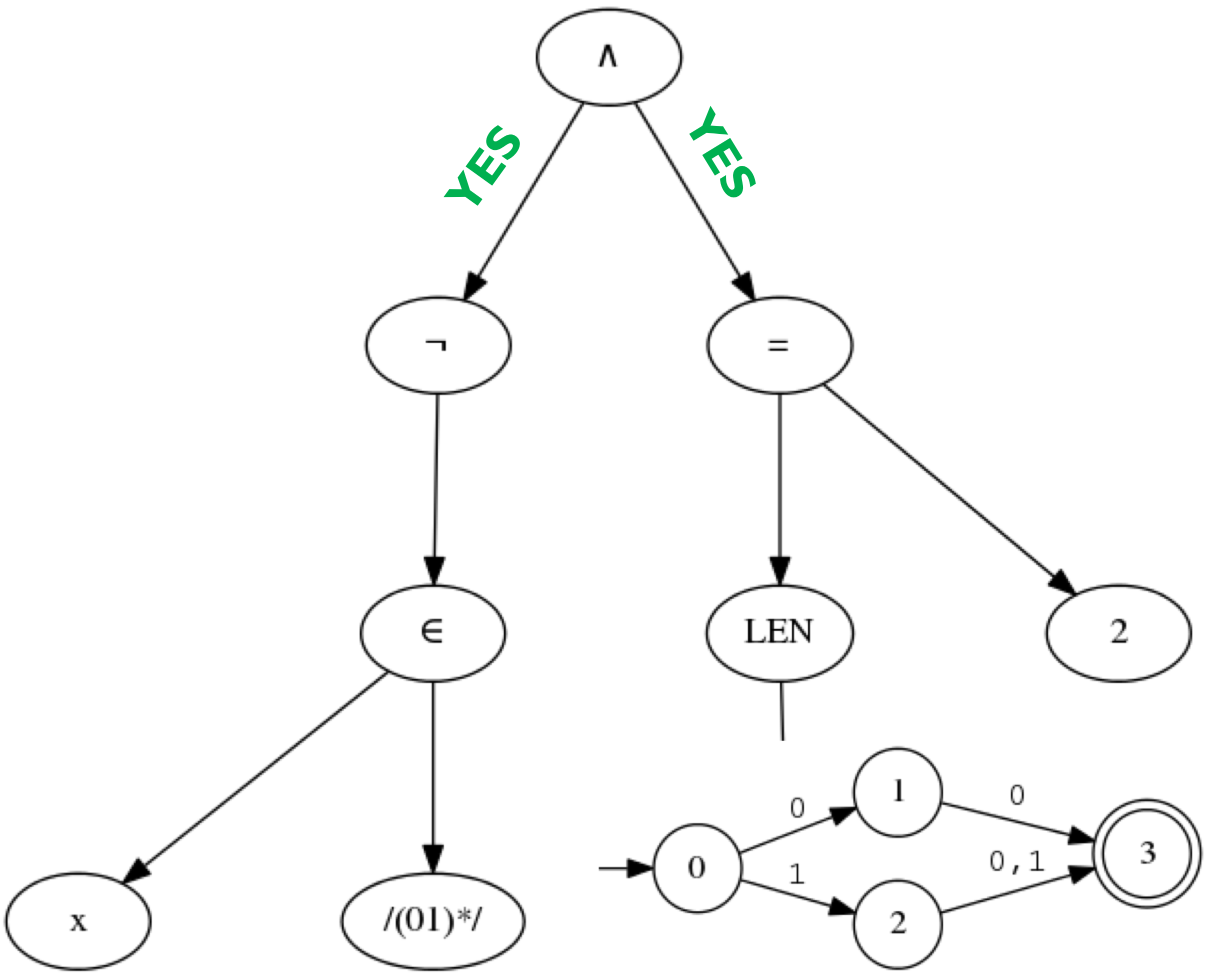
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



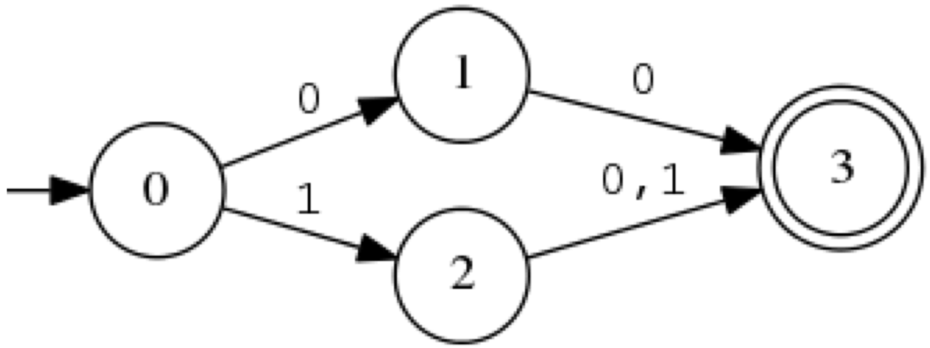
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \land LEN(x) = 2$$



String Automata Construction

$$C \equiv \neg(x \in (01)^*) \wedge LEN(x) = 2$$



00, 10, 11

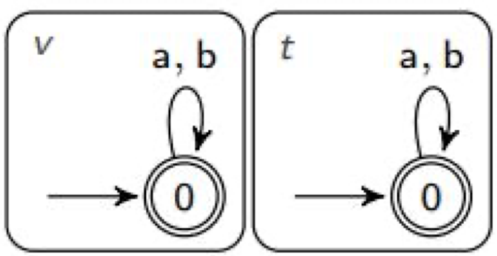
Relational constraints

- Relational constraints:
  - Constraints that involve multiple variables
- How do we handle relational constraints with automata?

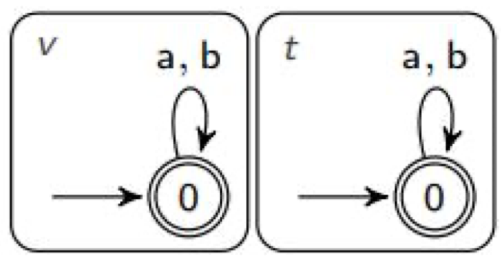
Automata-based constraint solving: relational

For multi-variable constraints, generate an automaton for each variable

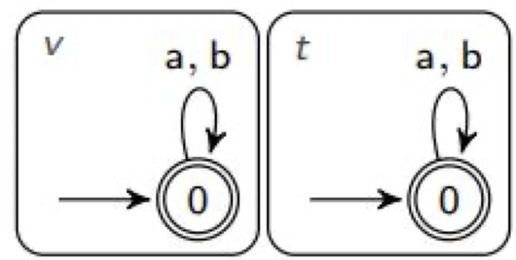
`v = t`



`v ≠ t`



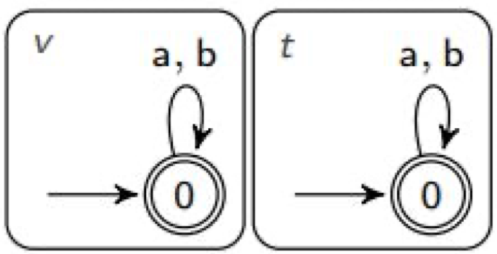
`v = t` $\wedge$ `v ≠ t`



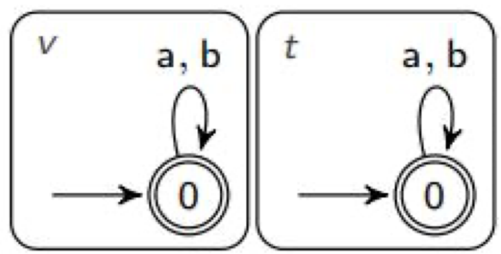
Automata-based constraint solving: relational

For multi-variable constraints, generate an automaton for each variable

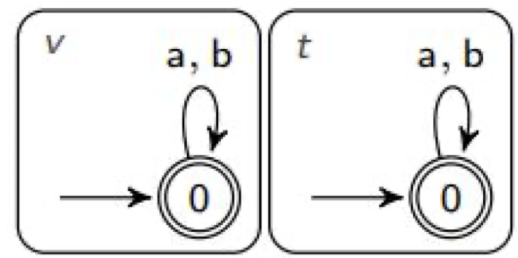
`v = t`



`v ≠ t`



`v = t ∧ v ≠ t`



Satisfiable!

Automata-based constraint solving: relational

Single track automata cannot precisely capture relational constraints

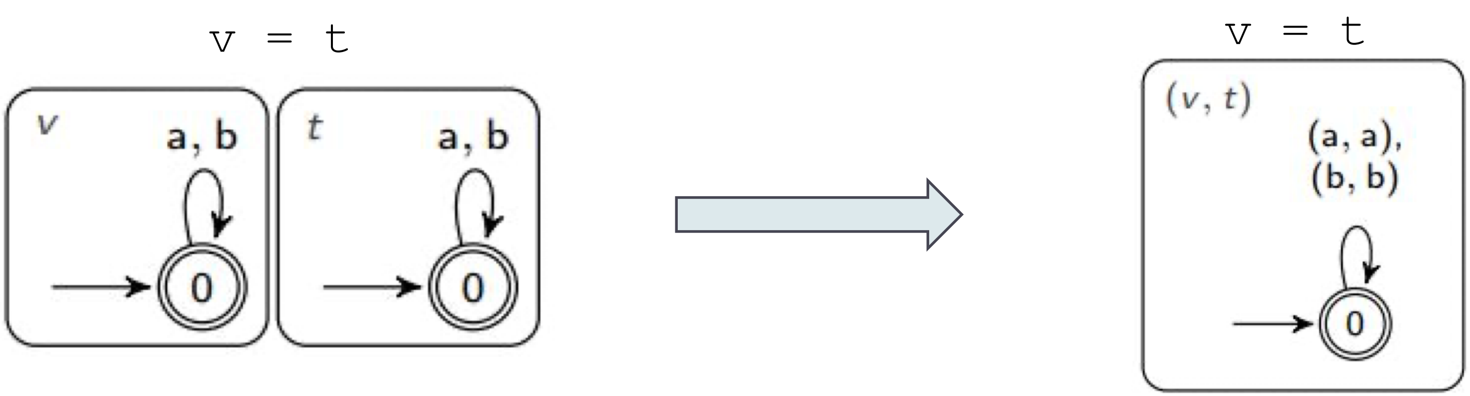
Generated automata significantly over-approximate # of satisfying solutions

Use multi-track automata

Multi-track automata

Multi-track automaton = DFA accepting tuples of strings

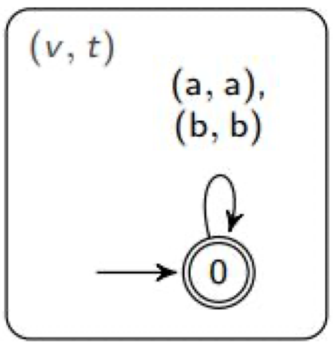
Each track represents the values of a single variable



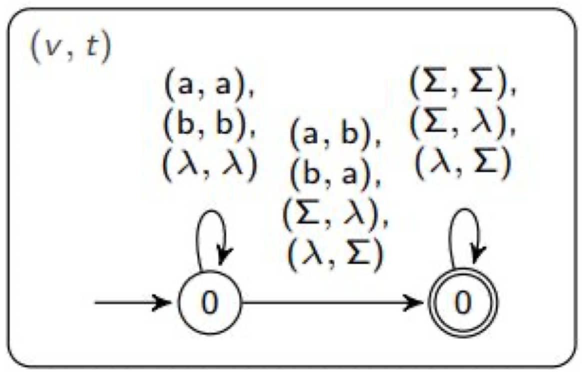
Preserves relations among variables!

Multi-track automata

`v = t`



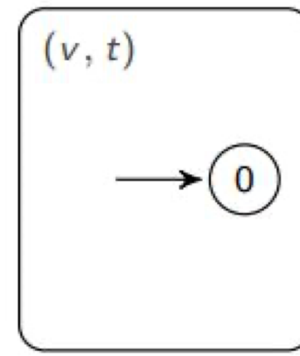
`v ≠ t`



Padding symbol λ ∉ Σ used to align tracks of different length (appears at the end)

automata product

`v = t ∧ v ≠ t`



Correctly encodes the constraint

Relational String Constraints: Summary

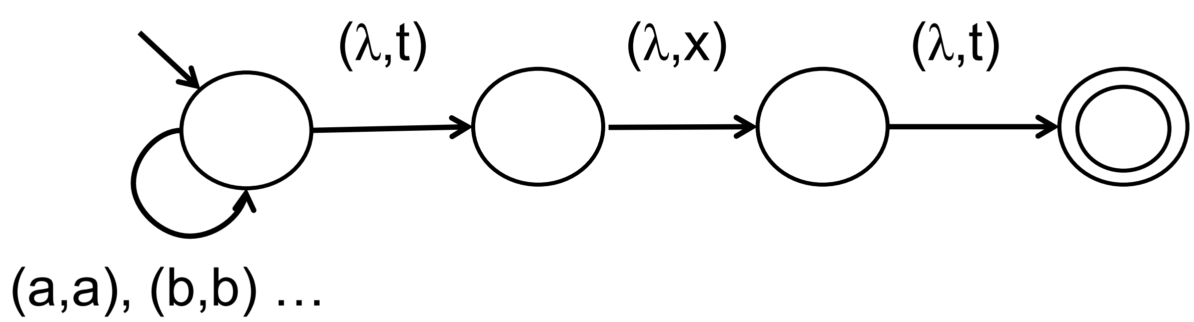
- How to handle constraints with multiple string variables?
- One approach is to use multiple single-track DFAs
  - One DFA per variable
- Alternative approach: Use one multi-track DFAs
  - Each track represents the values of one string variable

- Using multi-track DFAs:
  - Identifies the relations among string variables
  - Improves the precision
  - Can be used to represent properties that depend on relations among string variables, e.g., $file = $usr.txt

Multi-track Automata

- Let X (the first track), Y (the second track), be two string variables
- λ is the padding symbol
- A multi-track automaton that encodes the word equation:

Y = X.txt



Alignment

- To conduct relational string analysis, we need to compute union or intersection of multi-track automata
  - Intersection is closed under aligned multi-track automata
    - In an aligned multi-track automaton λs are right justified in all tracks, e.g., abλλ instead of aλbλ
- However, there exist unaligned multi-track automata that are not equivalent to any aligned multi-track automata
  - Use an alignment algorithm that constructs aligned automata which over or under approximates unaligned ones
    - Over approximation: Generates an aligned multi-track automaton that accepts a super set of the language recognized by the unaligned multi-track automaton
    - Under approximation: Generates an aligned multi-track automaton that accepts a subset of the language recognized by the unaligned multi-track automaton

Word Equations

- Word equations: Equality of two expressions that consist of concatenation of a set of variables and constants
  - Example: X = Y . txt
- Word equations and their combinations (using Boolean connectives) can be expressed using only equations of the form X = Y . c, X = c . Y, c = X . Y, X = Y. Z, Boolean connectives and existential quantification
- Construct multi-track automata from basic word equations
  - The automata should accept tuples of strings that satisfy the equation
- Boolean connectives can be handled using intersection, union and complement
- Existential quantification can be handled using projection

Word Equations to Automata

- Basic equations X = Y . c, X = c .Y, c = X .Y and their Boolean combinations can be represented precisely using multi-track automata

- The size of the aligned multi-track automaton for X = c . Y is exponential in the length of c

- The nonlinear equation X = Y . Z cannot be represented precisely using an aligned multi-track automaton

Word Equations to Automata

- When we cannot represent an equation precisely, we can generate an over or under-approximation of it
  - Over-approximation: The automaton accepts all string tuples that satisfy the equation and possibly more
  - Under-approximation: The automaton accepts only the string tuples that satify the equation but possibly not all of them
- We can implement a function CONSTRUCT(equation, sign)
  - Which takes a word equation and a sign and creates a multi-track automata that over or under-approximation of the equation based on the input sign

Integer Constraints

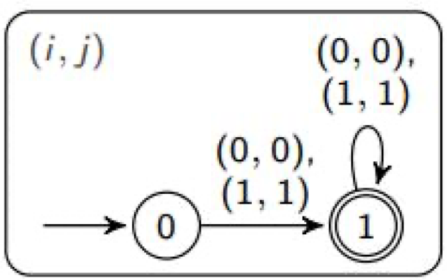
$$
\begin{array}{lcl}
C & \longrightarrow & bterm \\
\\
bterm & \longrightarrow & \mathtt{v} \mid \mathtt{true} \mid \mathtt{false} \\
& \mid & \neg bterm \mid bterm \wedge bterm \mid bterm \vee bterm \mid (bterm) \\
& \mid & sterm = sterm \\
& \mid & \mathtt{match}(sterm, sterm) \\
& \mid & \mathtt{contains}(sterm, sterm) \\
& \mid & \mathtt{begins}(sterm, sterm) \\
& \mid & \mathtt{ends}(sterm, sterm) \\
& \mid & iterm = iterm \mid iterm < iterm \mid iterm > iterm \\
\\
iterm & \longrightarrow & \mathtt{v} \mid \mathtt{n} \\
& \mid & iterm + iterm \mid iterm - iterm \mid iterm \times \mathtt{n} \mid (iterm) \\
& \mid & \mathtt{length}(sterm) \mid \mathtt{toint}(sterm) \\
& \mid & \mathtt{indexof}(sterm, sterm) \\
& \mid & \mathtt{lastindexof}(sterm, sterm) \\
\\
sterm & \longrightarrow & \mathtt{v} \mid \varepsilon \mid \mathtt{s} \\
& \mid & sterm.sterm \mid sterm|sterm \mid sterm^* \mid (sterm) \\
& \mid & \mathtt{charat}(sterm, iterm) \mid \mathtt{tostring}(iterm) \\
& \mid & \mathtt{toupper}(sterm) \mid \mathtt{tolower}(sterm) \\
& \mid & \mathtt{substring}(sterm, iterm, iterm) \\
& \mid & \mathtt{replacefirst}(sterm, sterm, sterm) \\
& \mid & \mathtt{replacelast}(sterm, sterm, sterm) \\
& \mid & \mathtt{replaceall}(sterm, sterm, sterm)
\end{array}
$$

Multi-track automata

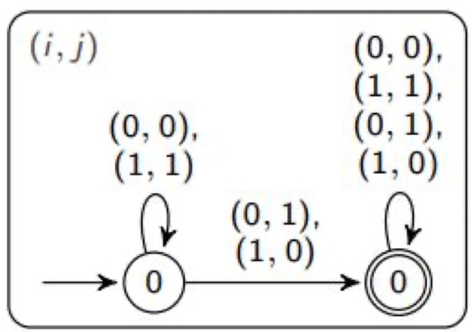
Multi-track automata can also represent Presburger (linear arithmetic) arithmetic constraints

- Each track represents a single numeric variable
- Encoded as binary integers in 2's complement form

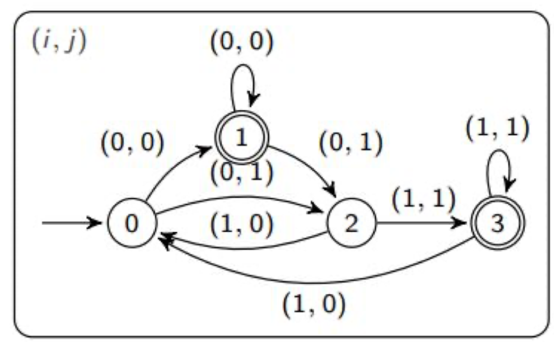
`i = j`



`i ≠ j`



`i = 2×j`



Linear Arithmetic Constraints

- Can be used to represent sets of valuations of unbounded integers
- Linear integer arithmetic formulas can be stored as a set of polyhedra

$$F = \bigvee_k \bigwedge_l c_{kl}$$

where each c_{kl} is a linear equality or inequality constraint and each

$$\bigwedge_l c_{kl}$$

is a polyhedron

Automata Representation for Arithmetic Constraints

[Bartzis, Bultan CIAA' 02, IJFCS ' 02]

- Given an atomic linear arithmetic constraint in one of the following two forms

$$\sum_{i=1}^{v} a_i \cdot x_i = c \qquad\qquad \sum_{i=1}^{v} a_i \cdot x_i < c$$

we construct a DFA which accepts all the solutions to the given constraint

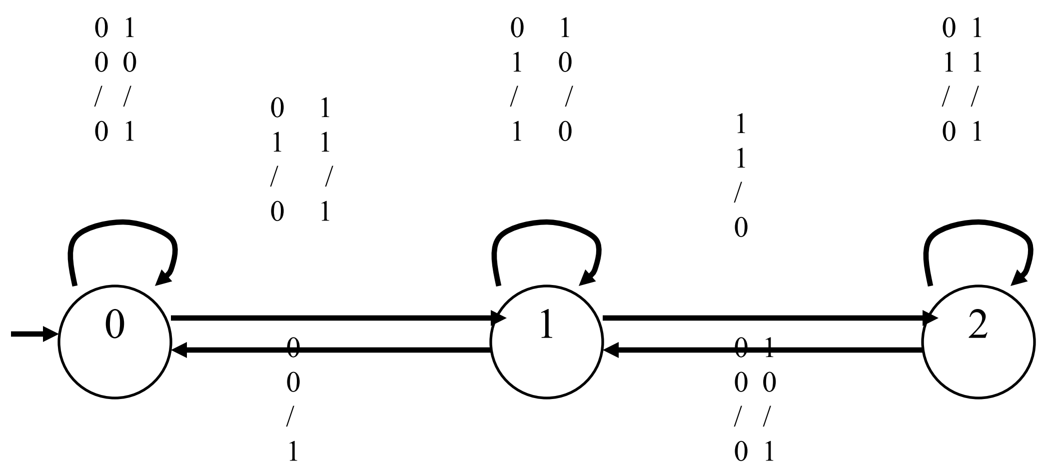
- By combining such automata one can handle full Presburger arithmetic (linear arithmetic constraints + quantification)

Basic Construction

- We first construct a basic state machine which
  - Reads one bit of each variable at each step, starting from the least significant bits
  - and executes bitwise binary addition and stores the carry in each step in its state

Example
x + 2y

```
      010
+ 2 × 001
---------
    1 0 0
```



Number of states: $O(\sum_{i=1}^{v} |a_i|)$

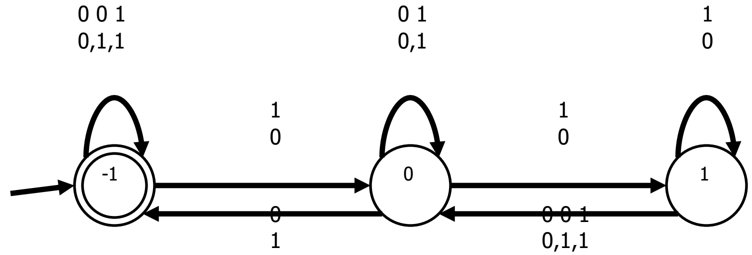
Automaton Construction

- Equality With 0
  - All transitions writing 1 go to a sink state
  - State labeled 0 is the only accepting state
  - For disequations (≠), state labeled 0 is the only rejecting state
- Inequality (<0)
  - States with negative carries are accepting
  - No sink state
- Non-zero Constant Term c
  - Same as before, but now -c is the initial state
  - If there is no such state, create one (and possibly some intermediate states which can increase the size by |c|)

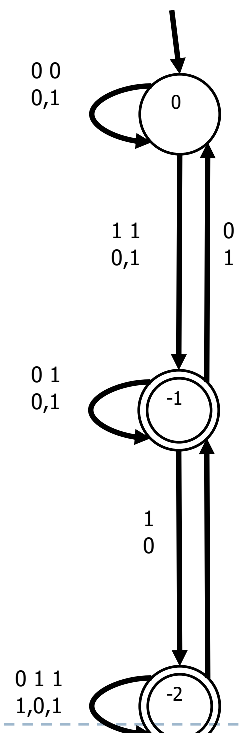
Conjunction and Disjunction

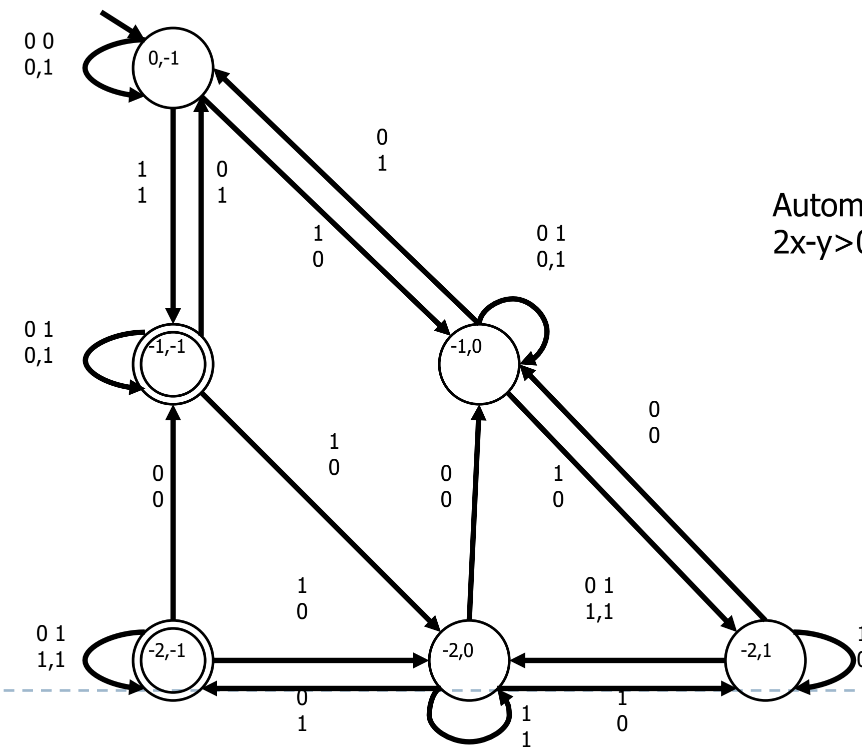
- Conjunction and disjunction is handled by generating the product automaton

Automaton for x-y<1



Automaton for 2x-y>0

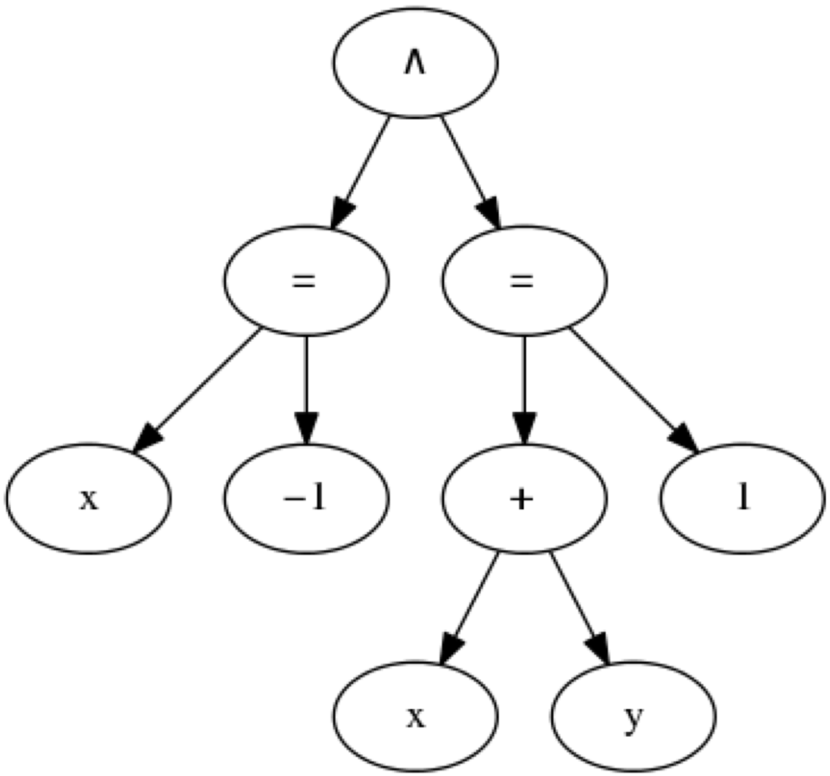




Automaton for x-y<1 ∧ 2x-y>0

Integer Automata Construction

$$C \equiv x = -1 \wedge \mathrm{x} + \mathrm{y} = 1$$

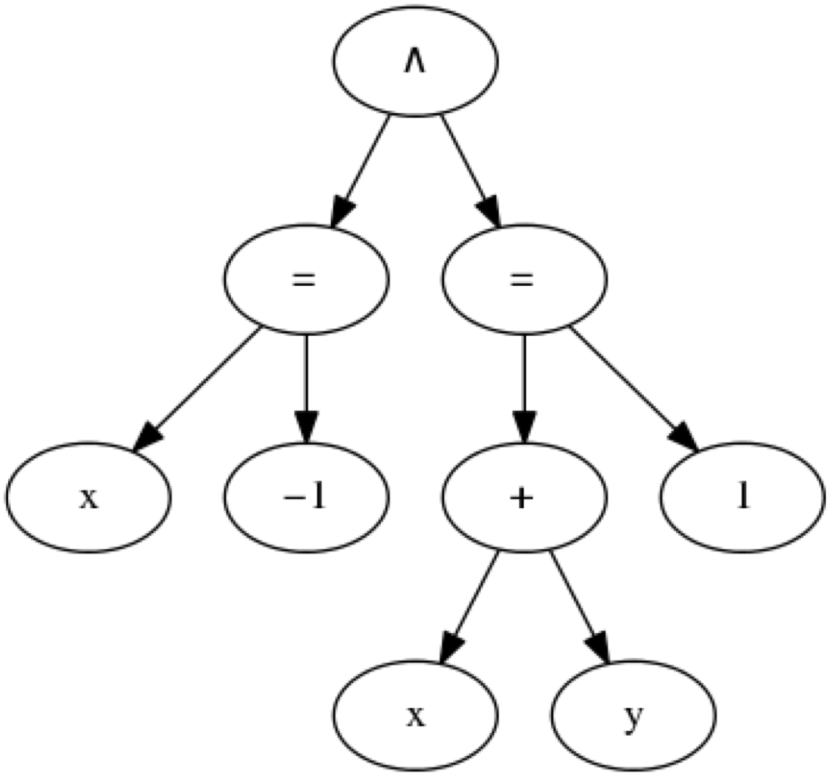


Integer Automata Construction

$$C \equiv x = -1 \wedge \mathrm{x} + \mathrm{y} = 1$$

$$C_1 \equiv x + 0 * y + 1 = 0 \Rightarrow [1\ 0\ 1]$$

$$C_2 \equiv x + y\ - 1 = 0 \Rightarrow [1\ 1 - 1]$$



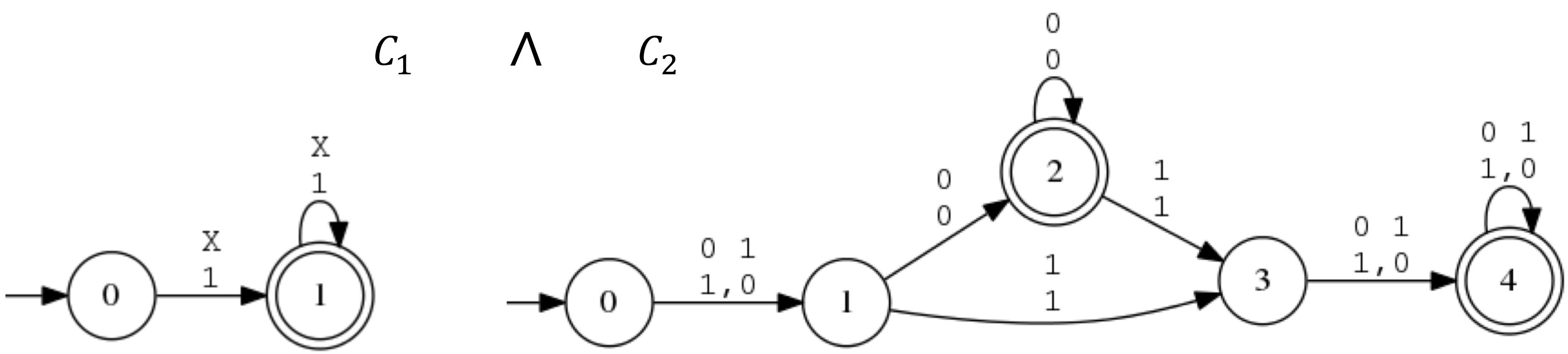
Integer Automata Construction

$$C \equiv x = -1 \wedge x + y = 1$$

$$C_1 \equiv x + 0 * y + 1 = 0 \Rightarrow [1\ 0\ 1]$$

$$C_2 \equiv x + y\ - 1 = 0 \Rightarrow [1\ 1 - 1]$$

$C_1 \quad \wedge \quad C_2$

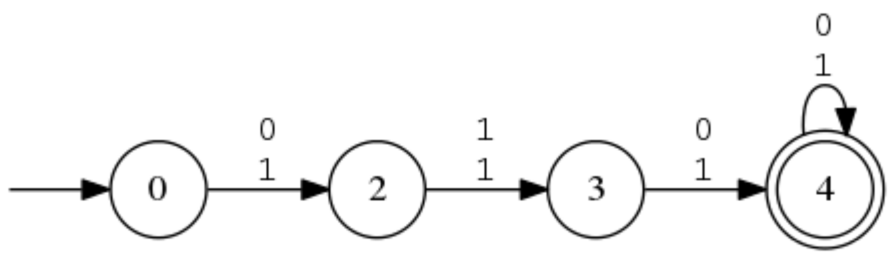


- Using automata construction techniques described in:

C. Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 2003

Integer Automata Construction

$$C \equiv x = -1 \wedge \mathrm{x} + \mathrm{y} = 1$$

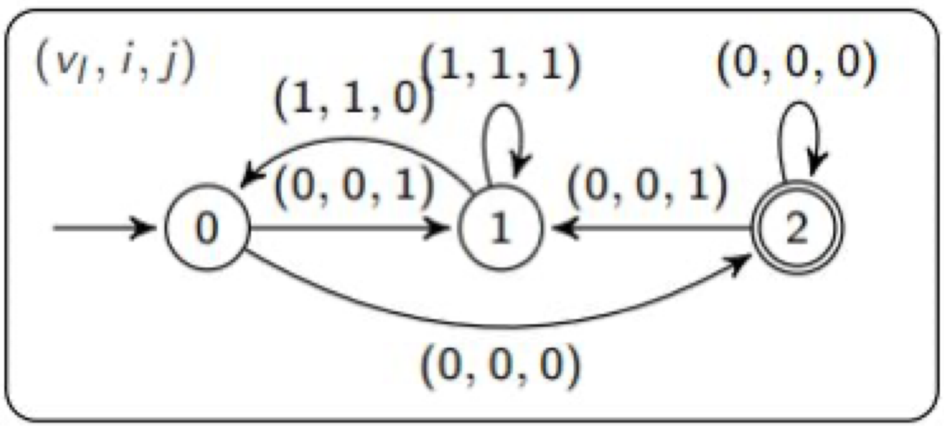


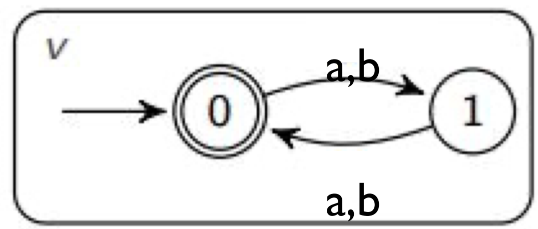
(111, 010) = (−1, 2)

- Conjunction and disjunction is handled by automata product, negation is handled by automata complement

Constraint Solving: Example Combining String and Integer Constraints

`i = 2×j ∧ length(v) = i ∧ match(v, (a | b)*)`





automaton for numeric variables
(v_l auxiliary variable encoding length of v)

automaton for string variables

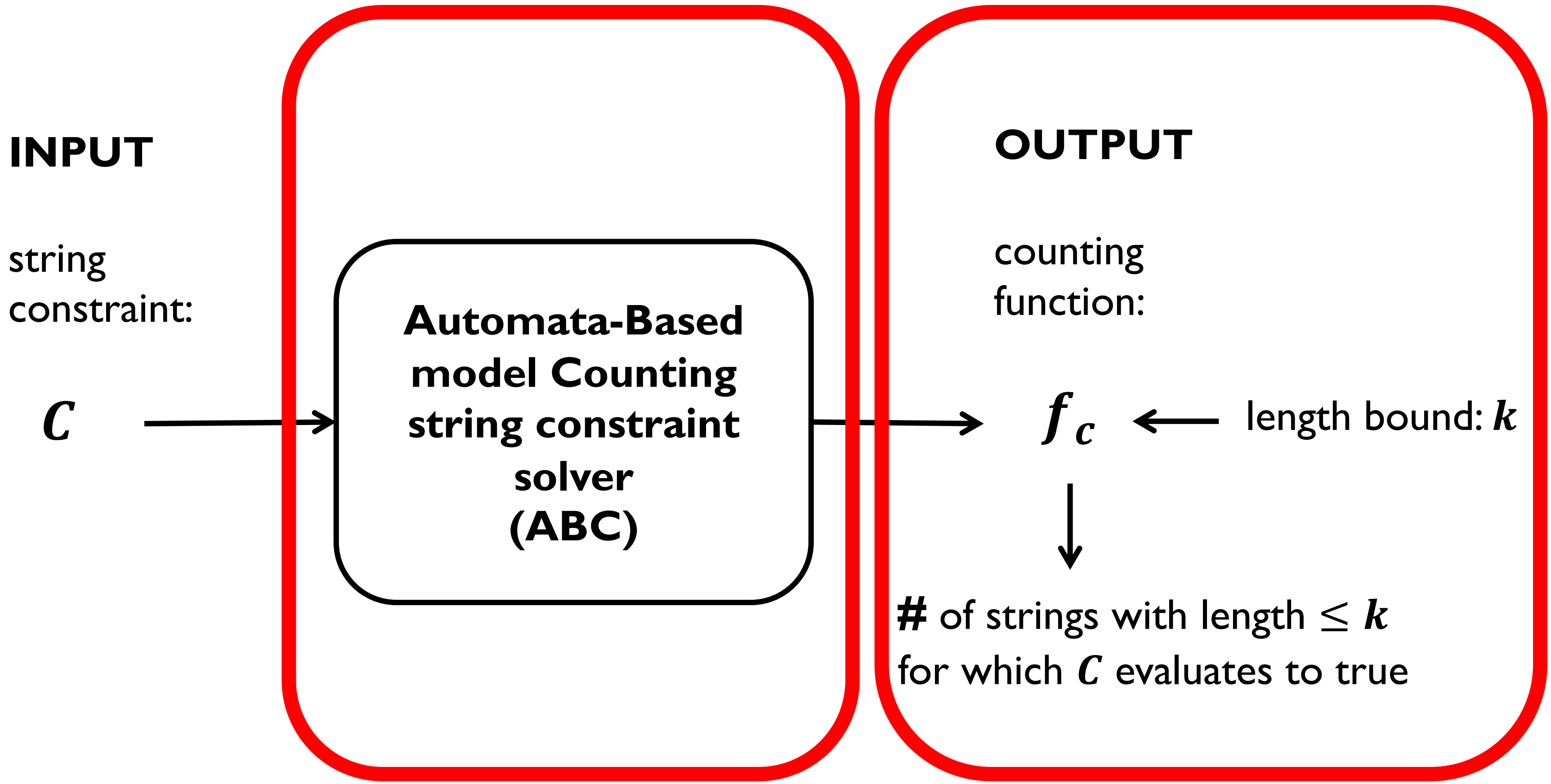
Apporoximation

- In general ABC constructs automata that over approximate the solution set
  - Some string constraints and combinations of string and integer constraints can lead to non-regular sets,
  - which means they are not representable as automata

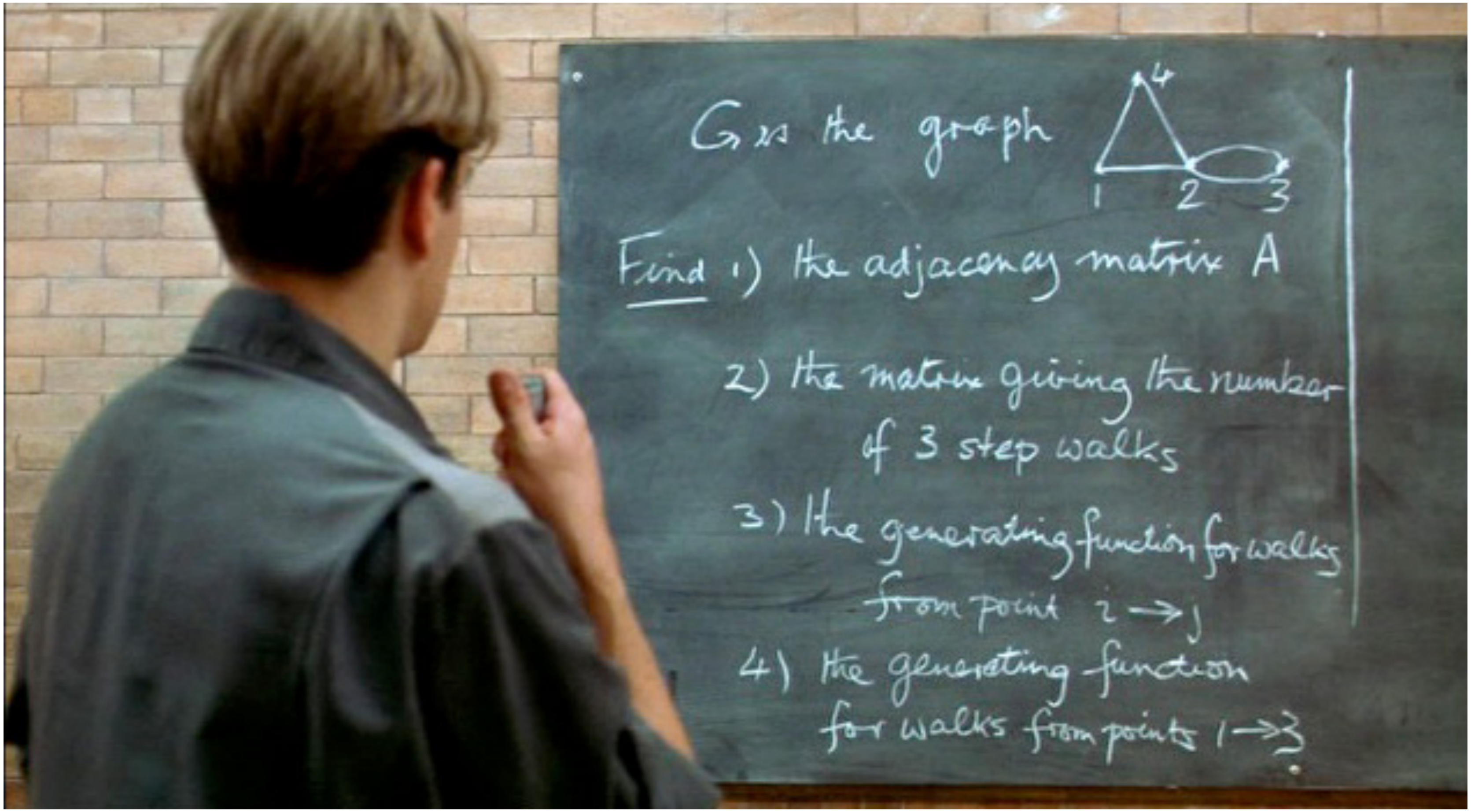
- ABC provides a sound over-approximation/abstraction:
  - If the automata does not accept any strings then the original formula is guaranteed to be NOT satisfiable

- It is possible to also provide a sound under-approximation using automata

Model Counting String Constraints Solver



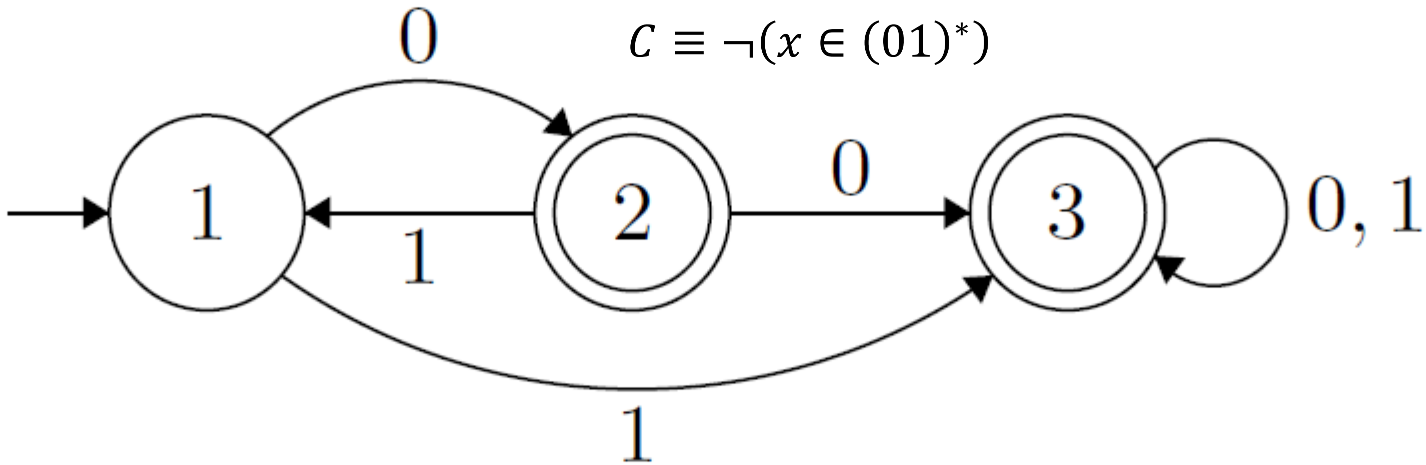
Can you solve it Will Hunting?



Automata-based Model Counting

- Converting constraints to automata reduces the model counting problem to path counting problem in graphs

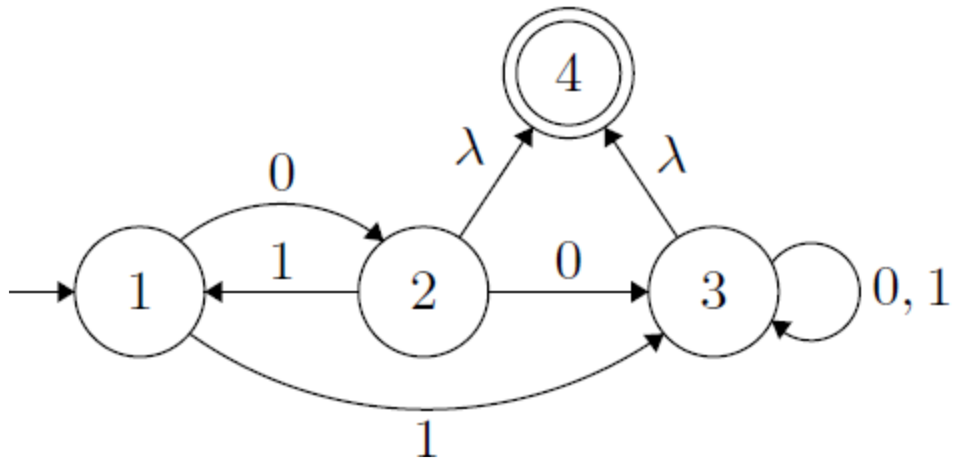
$C \equiv \neg(x \in (01)^*)$



- We will generate a function $f(k)$
  - Given length bound k, it will count the number of paths with length k.
  - $f(0) = 0,\ \{\}$
  - $f(1) = 2, \{0,1\}$
  - $f(2) = 3, \{00,10,11\}$

Path Counting via Matrix Exponentiation

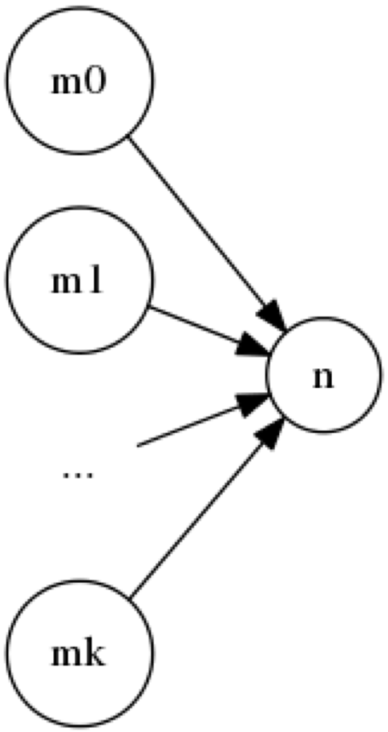
$$C = \neg(x \in (01)^*)$$



$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^2 = \begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 4 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^3 = \begin{bmatrix} 0 & 1 & 7 & 3 \\ 1 & 0 & 7 & 4 \\ 0 & 0 & 8 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^4 = \begin{bmatrix} 0 & 1 & 15 & 8 \\ 1 & 0 & 15 & 7 \\ 0 & 0 & 16 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$f(0) = 0$ $\quad$ $f(1) = 2$ $\quad$ $f(2) = 3$ $\quad$ $f(3) = 8$

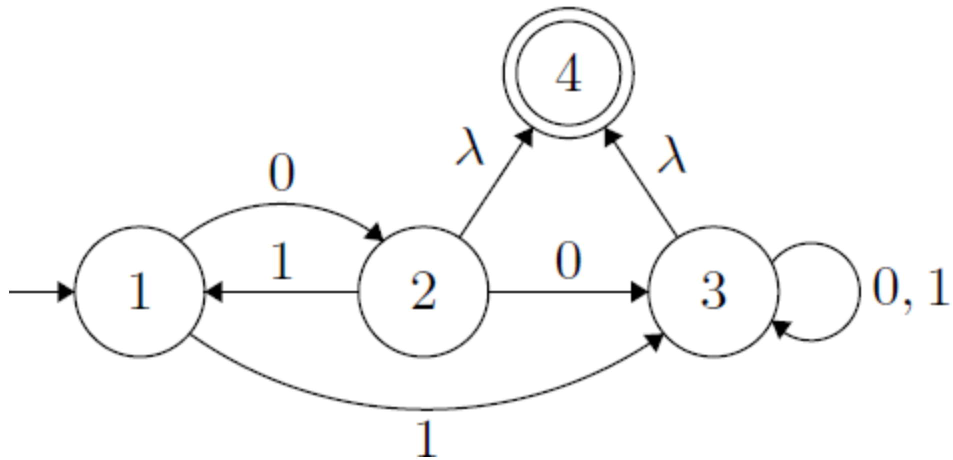
Path Counting via Recurrence Relation



$$f(n,k) = \sum_{(m,n)\in E} f(m,k-1)$$

$$f(0,0) = 1$$
$$f(1,0) = 0$$
$$f(2,0) = 0$$
$$\ldots$$
$$f(i,0) = 0$$

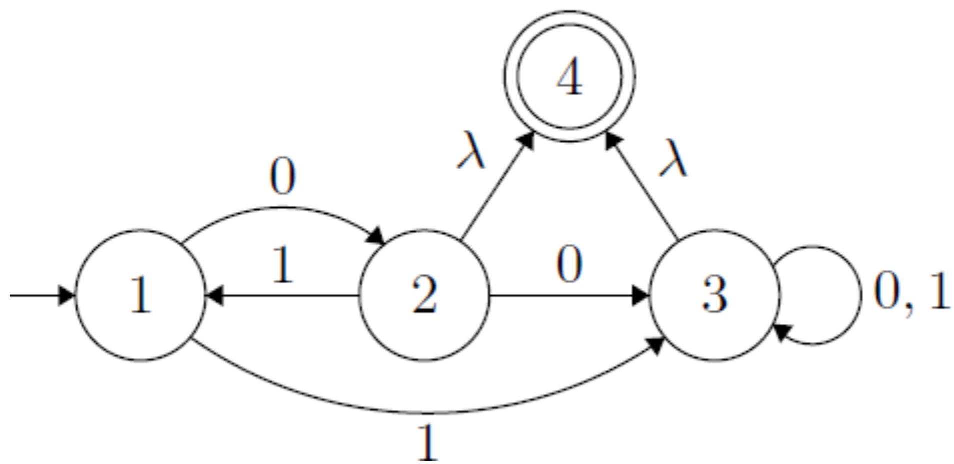
Path Counting via Recurrence Relation



$$f(4,k) = f(2,k-1) + f(3,k-1)$$
$$f(3,k) = f(1,k-1) + f(2,k-1) + f(3,k-1)$$
$$f(2,k) = f(1,k-1)$$
$$f(1,k) = f(2,k-1)$$
$$f(1,0) = 1, f(2,0) = 0, f(3,0) = 0, f(4,0) = 0$$

Path Counting via Recurrence Relation

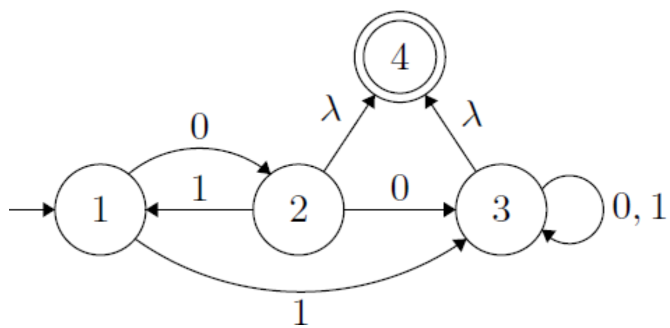
- We can solve system of recurrence relations for final node



$$f(0) = 0, f(1) = 2, f(2) = 3$$
$$f(k) = 2f(k-1) + f(k-2) - 2f(k-3)$$

Counting Paths via Generating Functions

- We can compute a generating function, $g(z)$, for a DFA from the associated matrix



$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$g(z) = (-1)^n \frac{\det(I - zT\colon n+1, 1)}{z \times \det(I - zT)} = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}$$

Counting Paths via Generating Functions

$$g(z) = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}$$

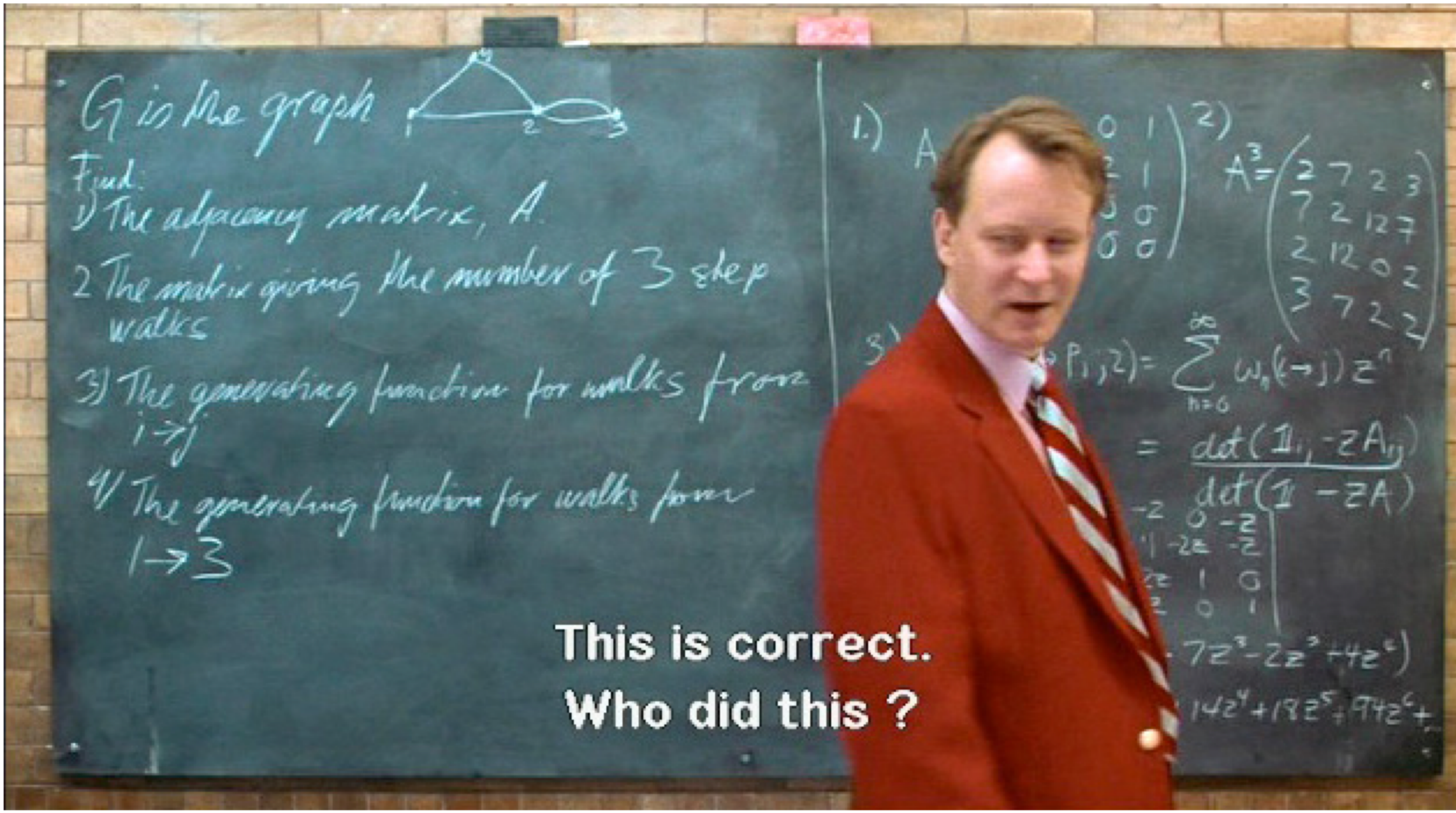
- Each $f(i)$ can be computed by Taylor expansion of $g(z)$

$$g(z) = \frac{g(0)}{0!}z^0 + \frac{g^{(1)}(0)}{1!}z^1 + \frac{g^{(2)}(0)}{2!}z^2 + \cdots + \frac{g^{(n)}(0)}{n!}z^n + \cdots$$

$$g(z) = 0z^0 + 2z^1 + 3z^2 + 8z^3 + 15z^4 + \cdots$$

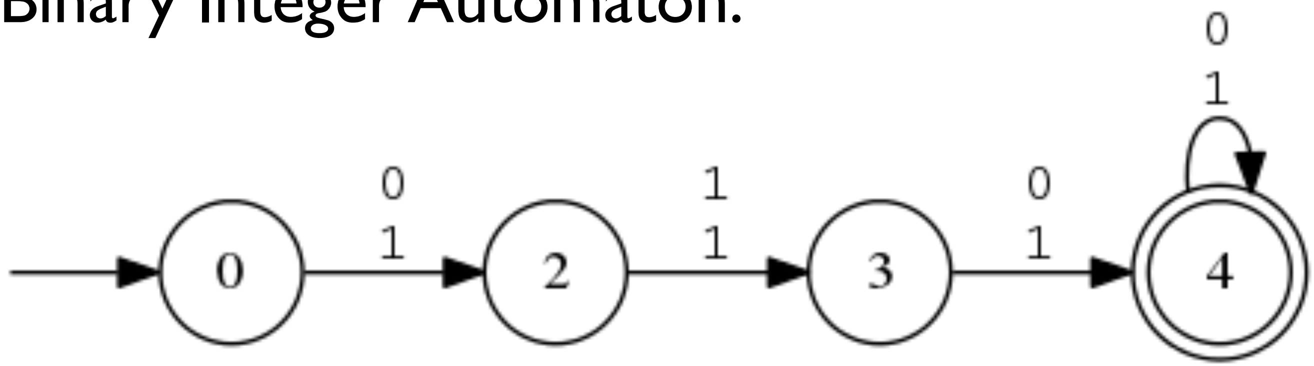
$$g(z) = f(0)z^0 + f(1)z^1 + f(2)z^2 + f(3)z^3 + f(4)z^4 + \cdots$$

Good job Will Hunting!

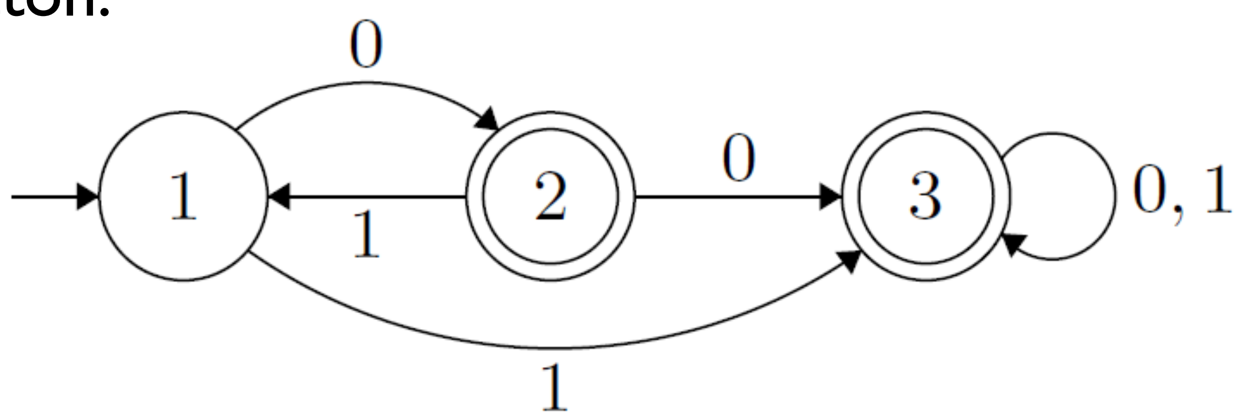


Applicable to Both Automata

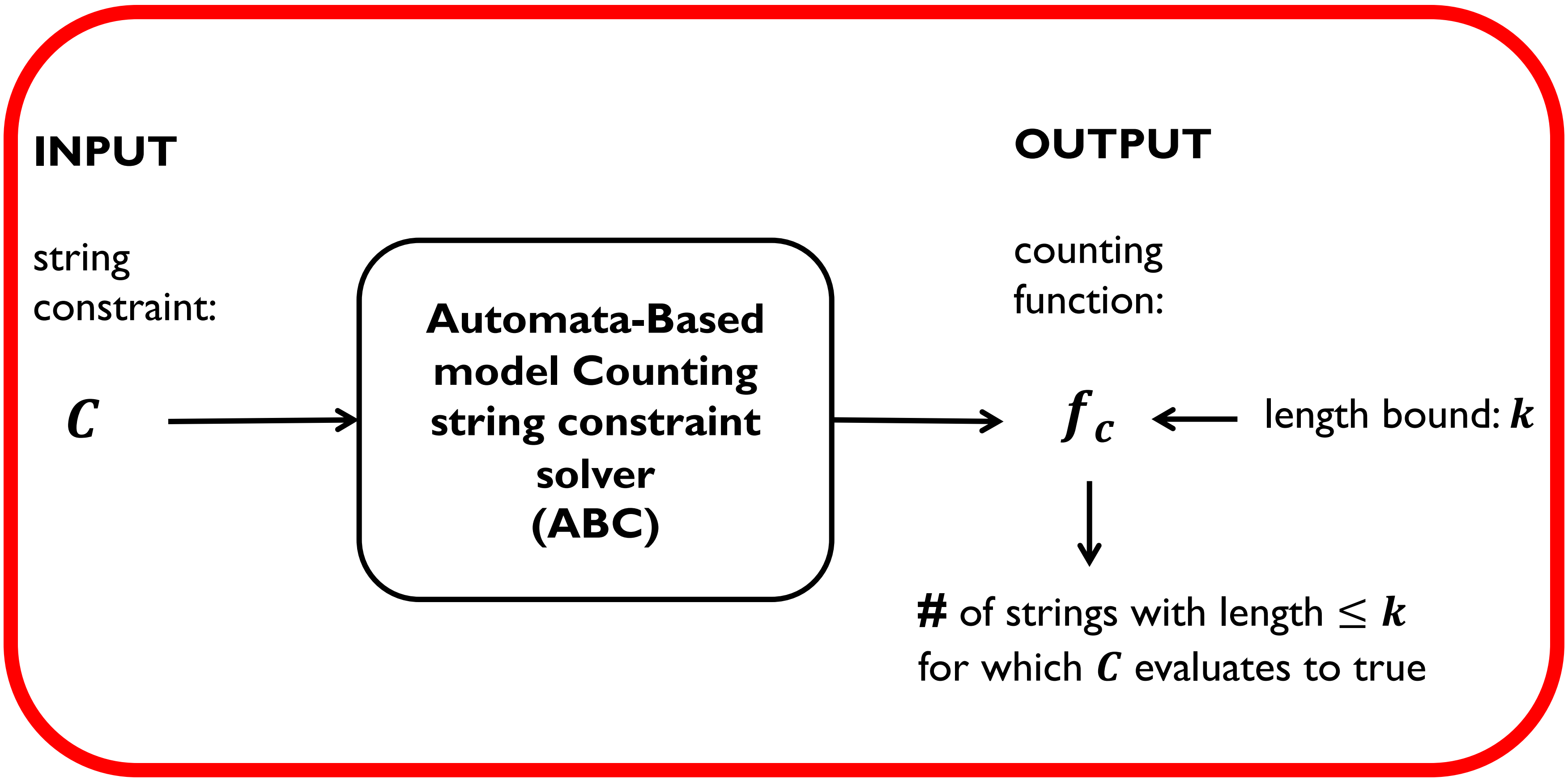
- Multi-track Binary Integer Automaton:



- String Automaton:



Model Counting String Constraints Solver

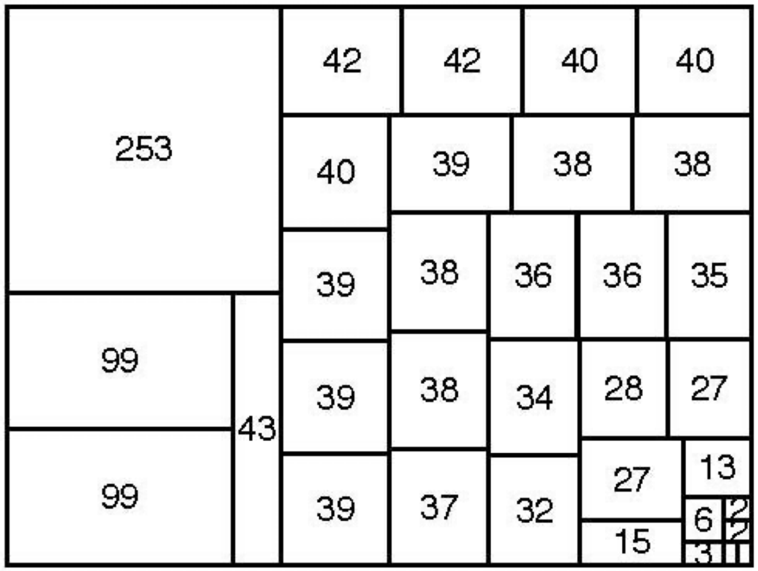


Automata-based model counting extensions

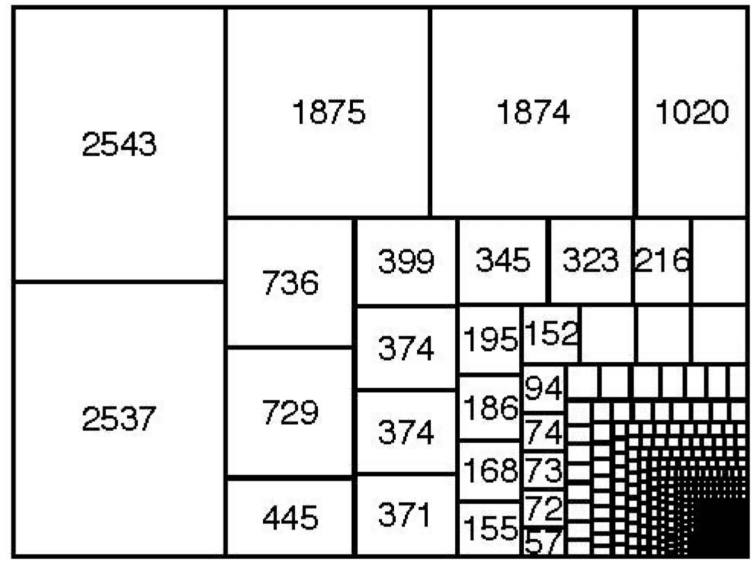
- In order to scale the automata-based model counting, it is necessary to cache the prior results
- Many constraints generated from programs are equivalent
  - By normalizing constraints we can identify many equivalent constraints
- 87X improvement for the Kaluza big data set

UCSB

Kaluza Dataset:
1,342 big constraints and 17,554 small



1,342 big constraints are reduced to 34 equivalent constraints after normalization



17,554 small constraints are reduced to 360 equivalent constraints after normalization

Automata-based model counting extensions

- More caching
  - Cache subformulas
  - Automata provide a canonical form for constraints after minimization and determinization
  - Generate keys for automata and use a compute cache like BDDs
- Subformula caching leads to order of magnitude improvement for attack synthesis

ABC DEMO

https://github.com/vlab-cs-ucsb/ABC

http://ec2-52-35-130-176.us-west-2.compute.amazonaws.com/