Universal Algebra and Computational Complexity
Lecture 1

Ross Willard

University of Waterloo, Canada

Třešt', September 2008

Outline

Outline

LECTURE 1: Decision problems and Complexity Classes

Outline

LECTURE 1: Decision problems and Complexity Classes

LECTURE 2: Nondeterminism, Reductions and Complete problems

Outline

LECTURE 1: Decision problems and Complexity Classes

LECTURE 2: Nondeterminism, Reductions and Complete problems

LECTURE 3: Results and problems from Universal Algebra

Three themes: problems, algorithms, efficiency

A *Decision Problem* is . . .

- A *YES/NO question*
- parametrized by one or more *inputs*.
  - Inputs must:
    - range over an *infinite* class.
    - be "finitistically described"

Three themes: problems, algorithms, efficiency

A *Decision Problem* is . . .

- A *YES/NO question*
- parametrized by one or more *inputs*.
  - Inputs must:
    - range over an *infinite* class.
    - be "finitistically described"

What we seek:

- An *algorithm* which correctly answers the question for all possible inputs.

Three themes: problems, algorithms, efficiency

A *Decision Problem* is . . .

- A *YES/NO question*
- parametrized by one or more *inputs*.
  - Inputs must:
    - range over an *infinite* class.
    - be "finitistically described"

What we seek:

- An *algorithm* which correctly answers the question for all possible inputs.

What we ask:

- How *efficient* is this algorithm?
- Is there a better (more efficient) algorithm?

Directed Graph Reachability problem (PATH)

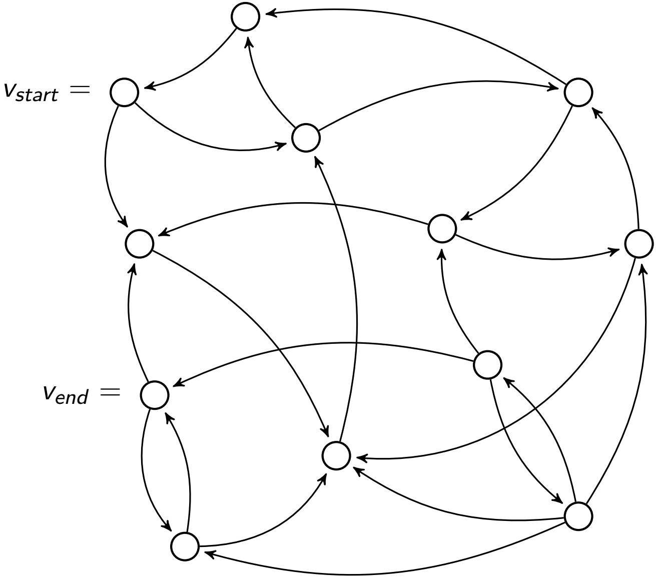
INPUT:

- A finite directed graph $G = (V, E)$
- Two distinguished vertices $v_{start}, v_{end} \in V$.

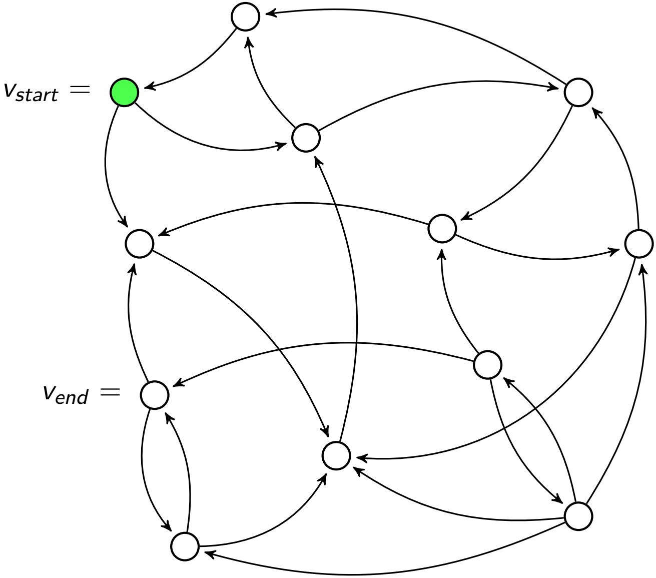
QUESTION:

- Does there exist in G a *directed path* from v_{start} to v_{end}?

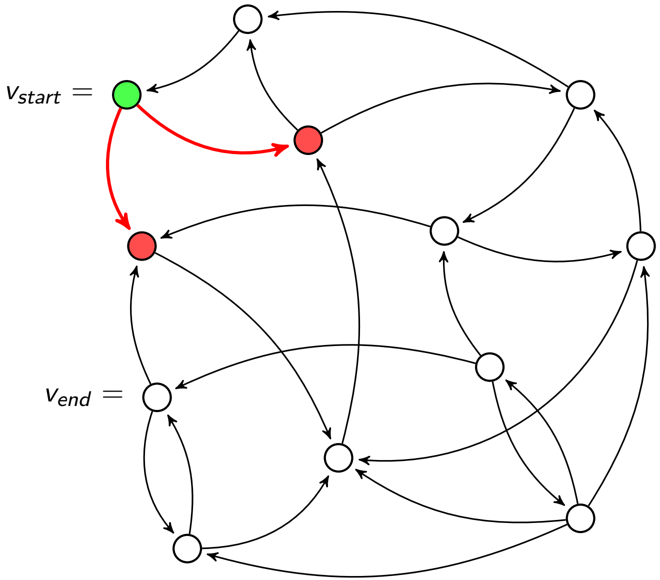
An Algorithm for PATH



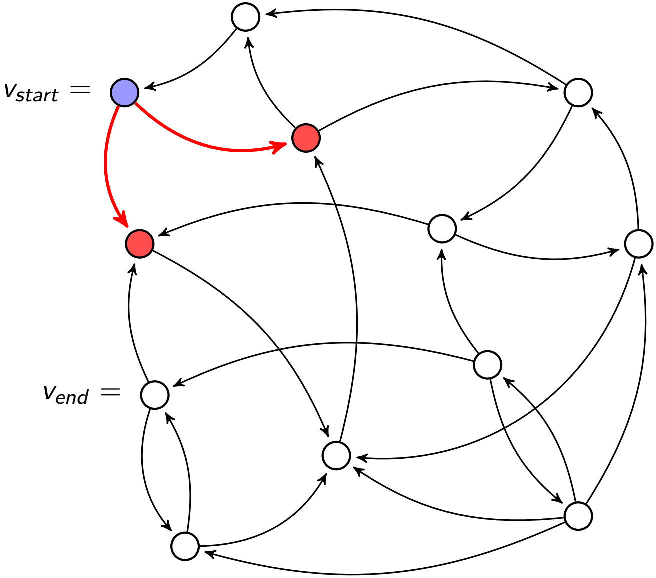
An Algorithm for PATH



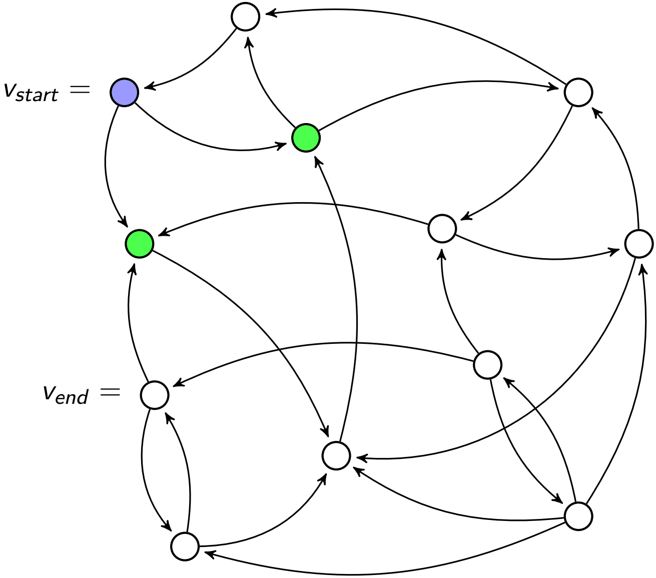
An Algorithm for PATH



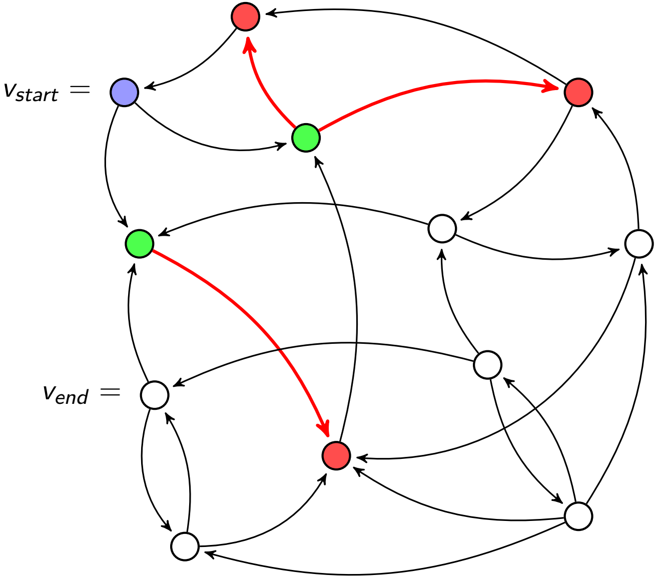
An Algorithm for PATH



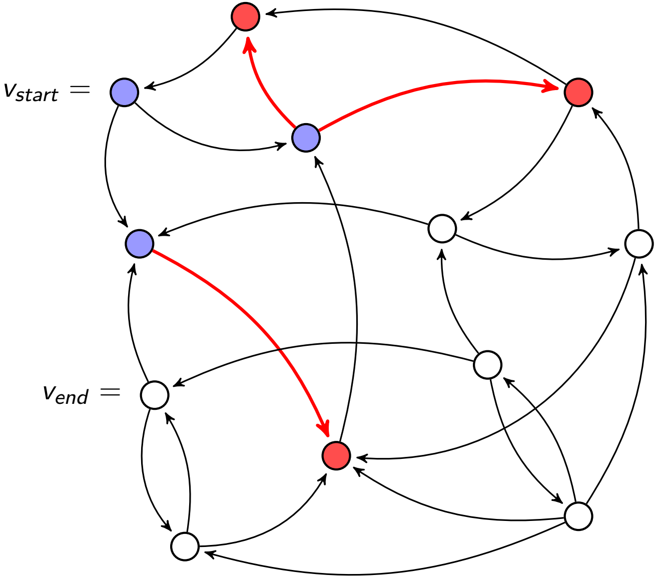
An Algorithm for PATH



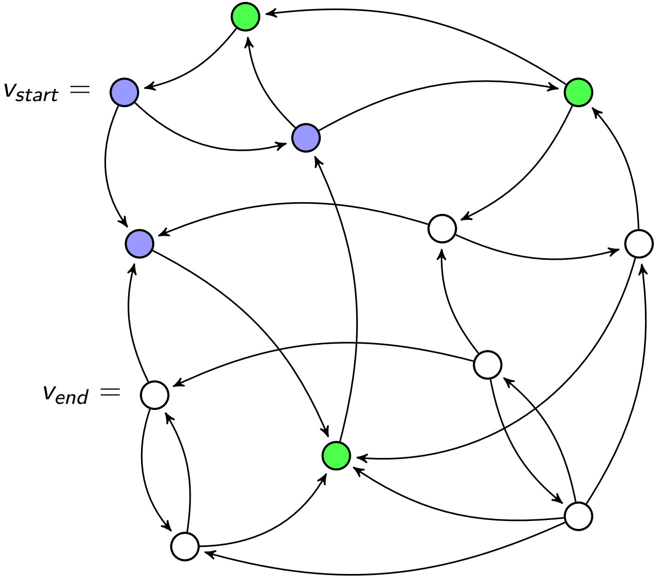
An Algorithm for PATH



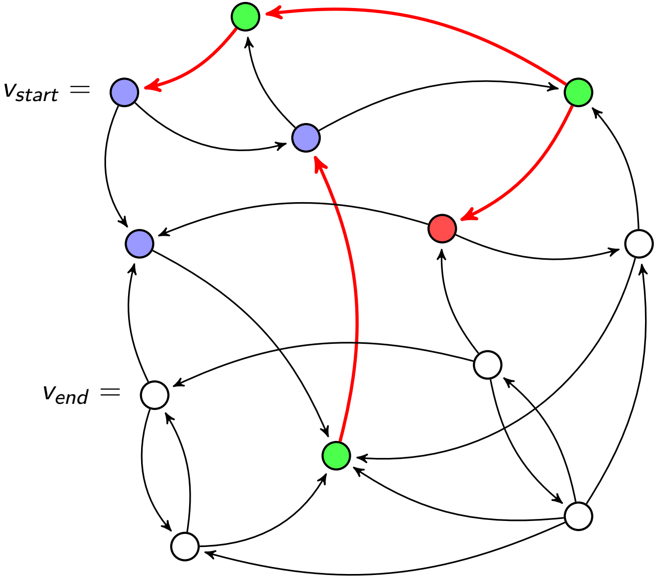
An Algorithm for PATH



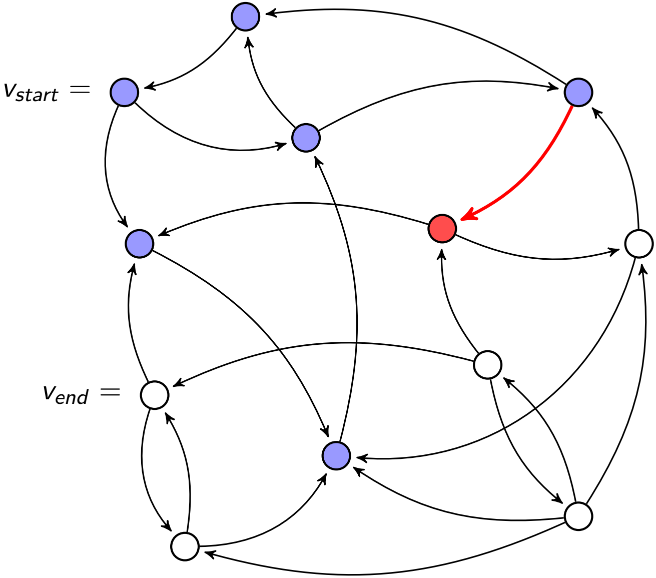
An Algorithm for PATH



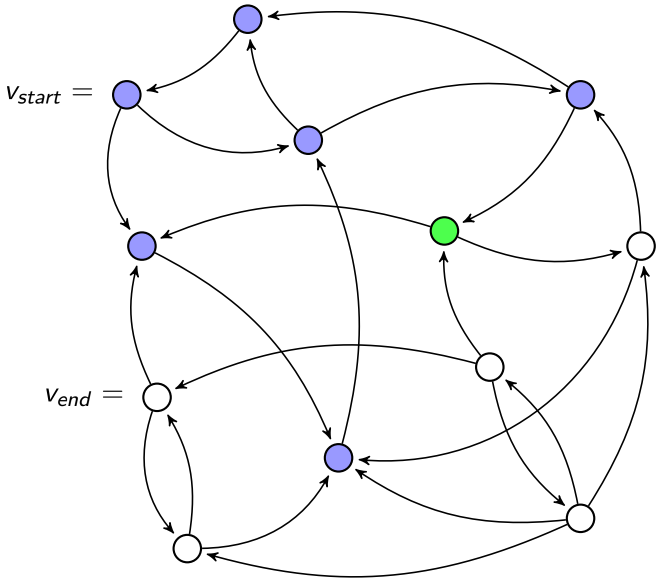
An Algorithm for PATH



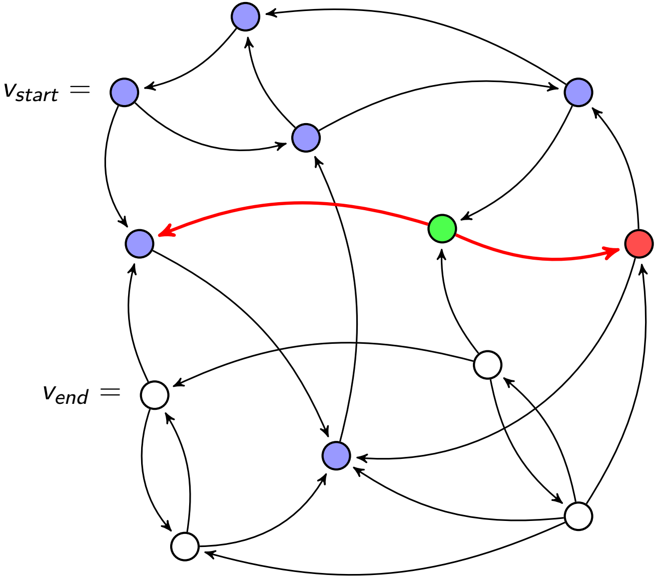
An Algorithm for PATH



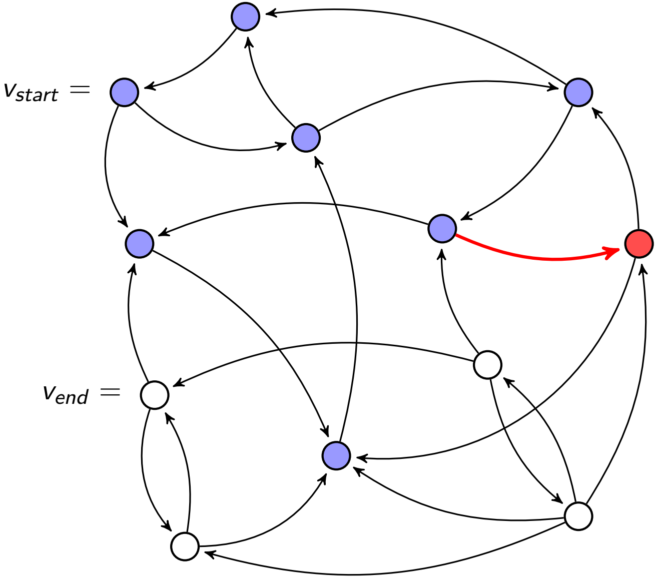
An Algorithm for PATH



An Algorithm for PATH



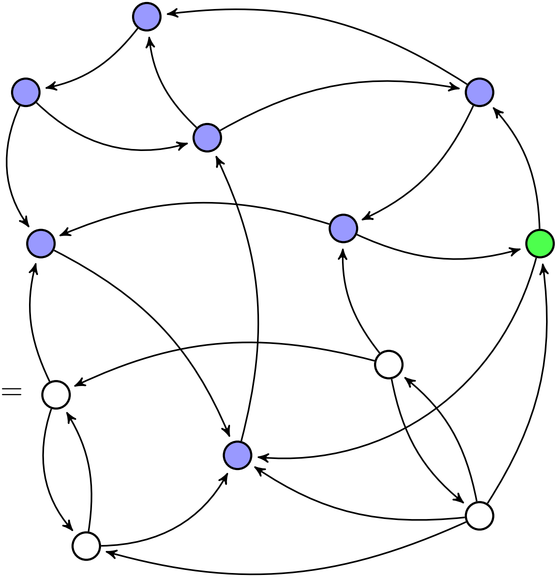
An Algorithm for PATH



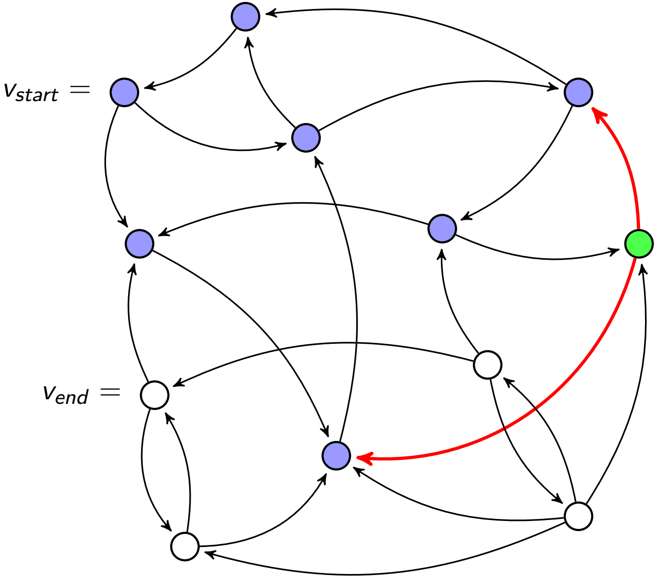
An Algorithm for PATH

$v_{start} =$

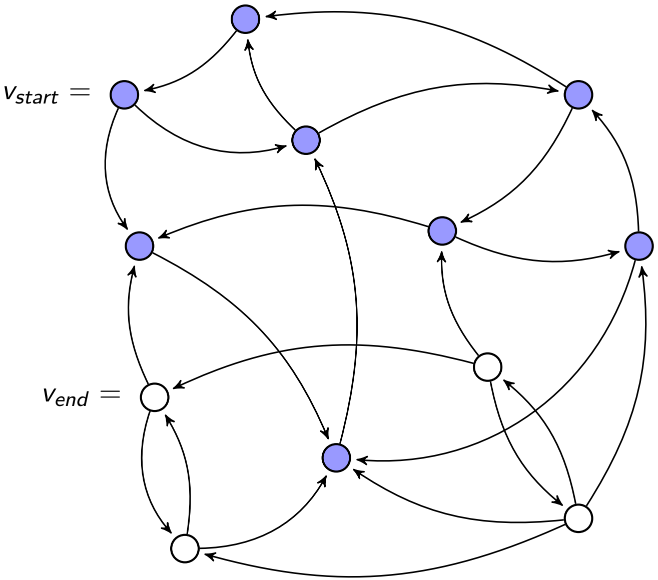
$v_{end} =$



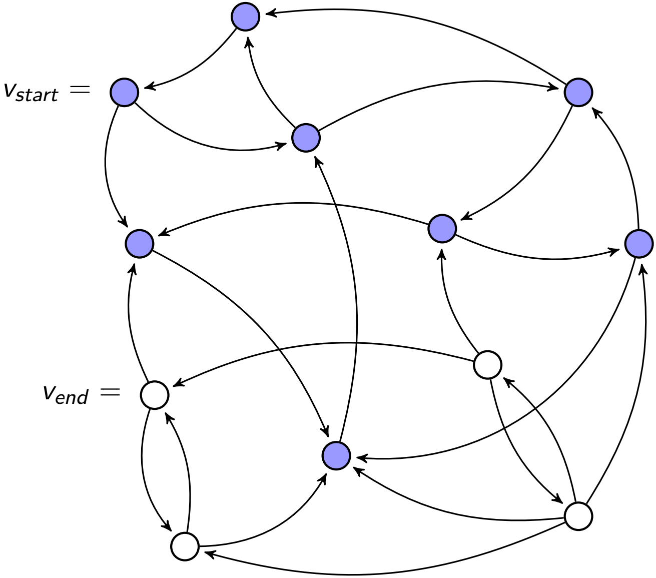
An Algorithm for PATH



An Algorithm for PATH



An Algorithm for PATH



Answer: "NO"

Efficiency of this algorithm

How long does this algorithm take?

- I.e., how many steps . . .

Efficiency of this algorithm

How long does this algorithm take?

- I.e., how many steps ...
- ... as a function of the size of the input graph.

Efficiency of this algorithm

How long does this algorithm take?

- I.e., how many steps . . .
- . . . as a function of the size of the input graph.

I'll give three answers to this.

First answer – Heuristics

Only action is changing a vertex's color.

Only changes possible are

- white $\Rightarrow$ red
- red $\Rightarrow$ green
- green $\Rightarrow$ blue.

So if $n = |V|$, then the algorithm requires at most $3n$ vertex-color changes.

Second answer – pseudo-code

Simplifying assumptions:

- $V = \{0, 1, \ldots, n-1\}$
- E is encoded by the adjacency matrix $M_E = [e_{i,j}]$ where

$$e_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{else.} \end{cases}$$

Second answer – pseudo-code

Simplifying assumptions:

- $V = \{0, 1, \ldots, n-1\}$
- E is encoded by the adjacency matrix $M_E = [e_{i,j}]$ where

$$e_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{else.} \end{cases}$$

Auxiliary variables:

- i, j will range over $\{0, 1, \ldots, n{-}1\}$.
- For $i < n$ let c_i be a variable recording the color of vertex i.
- Let *GreenVar* be a variable storing whether there are green-colored vertices.

Second answer – pseudo-code

Algorithm:

- Input n, M_E, *start* and *end*.
- For $i = 0$ to $n - 1$ set $c_i := \mathit{white}$.
- Set $c_{start} = \mathit{green}$.
- Set $\mathit{GreenVar} := \mathit{yes}$.

Second answer – pseudo-code

Algorithm:

- Input n, M_E, *start* and *end*.
- For $i = 0$ to $n - 1$ set $c_i :=$ *white*.
- Set $c_{start} =$ *green*.
- Set *GreenVar* := *yes*.
- MAIN LOOP: While *GreenVar* = *yes* do:
    - For $i = 0$ to $n - 1$; for $j = 0$ to $n - 1$
        - if $e_{i,j} = 1$ and $c_i =$ *green* and $c_j =$ *white* then set $c_j :=$ *red*.
    - For $i = 0$ to $n - 1$
        - If $c_i =$ *green* then set $c_i :=$ *blue*
    - Set *GreenVar* := *no*
    - For $i = 0$ to $n - 1$
        - If $c_i =$ *red* then (set $c_i :=$ *green* and set *GreenVar* := *yes*)

Algorithm:

- Input n, M_E, *start* and *end*.
- For $i = 0$ to $n - 1$ set $c_i :=$ *white*.
- Set $c_{start} =$ *green*.
- Set *GreenVar* := *yes*.
- MAIN LOOP: While *GreenVar* = *yes* do:
  - For $i = 0$ to $n - 1$; for $j = 0$ to $n - 1$
    - if $e_{i,j} = 1$ and $c_i =$ *green* and $c_j =$ *white* then set $c_j :=$ *red*.
  - For $i = 0$ to $n - 1$
    - If $c_i =$ *green* then set $c_i :=$ *blue*
  - Set *GreenVar* := *no*
  - For $i = 0$ to $n - 1$
    - If $c_i =$ *red* then (set $c_i :=$ *green* and set *GreenVar* := *yes*)
- If $c_{end} =$ *blue* then output YES; else output NO.

Second answer – pseudo-code

Algorithm:

- Input n, M_E, *start* and *end*.
- For $i = 0$ to $n-1$ set $c_i := white$.
- Set $c_{start} = green$.
- Set $GreenVar := yes$.
- MAIN LOOP: While $GreenVar = yes$ do:
  - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$
    - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$.
  - For $i = 0$ to $n-1$
    - If $c_i = green$ then set $c_i := blue$
  - Set $GreenVar := no$
  - For $i = 0$ to $n-1$
    - If $c_i = red$ then (set $c_i := green$ and set $GreenVar := yes$)
- If $c_{end} = blue$ then output YES; else output NO.

n loops

Second answer – pseudo-code

Algorithm:

- Input n, M_E, *start* and *end*.
- For $i = 0$ to $n-1$ set $c_i := white$.
- Set $c_{start} = green$.
- Set $GreenVar := yes$.
- MAIN LOOP: While $GreenVar = yes$ do:
  - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$
    - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$.
  - For $i = 0$ to $n-1$
    - If $c_i = green$ then set $c_i := blue$
  - Set $GreenVar := no$
  - For $i = 0$ to $n-1$
    - If $c_i = red$ then (set $c_i := green$ and set $GreenVar := yes$)
- If $c_{end} = blue$ then output YES; else output NO.

n loops

n^2 cases

Algorithm:

- Input n, M_E, *start* and *end*.
- For $i = 0$ to $n-1$ set $c_i := white$.
- Set $c_{start} = green$.
- Set $GreenVar := yes$.
- MAIN LOOP: While $GreenVar = yes$ do:
    - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$
        - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$.
    - For $i = 0$ to $n-1$
        - If $c_i = green$ then set $c_i := blue$
    - Set $GreenVar := no$
    - For $i = 0$ to $n-1$
        - If $c_i = red$ then (set $c_i := green$ and set $GreenVar := yes$)
- If $c_{end} = blue$ then output YES; else output NO.

n loops

n^2 cases

$\boxed{O(n^3) \text{ steps if } n = |V|}$

Third answer – machine implementation

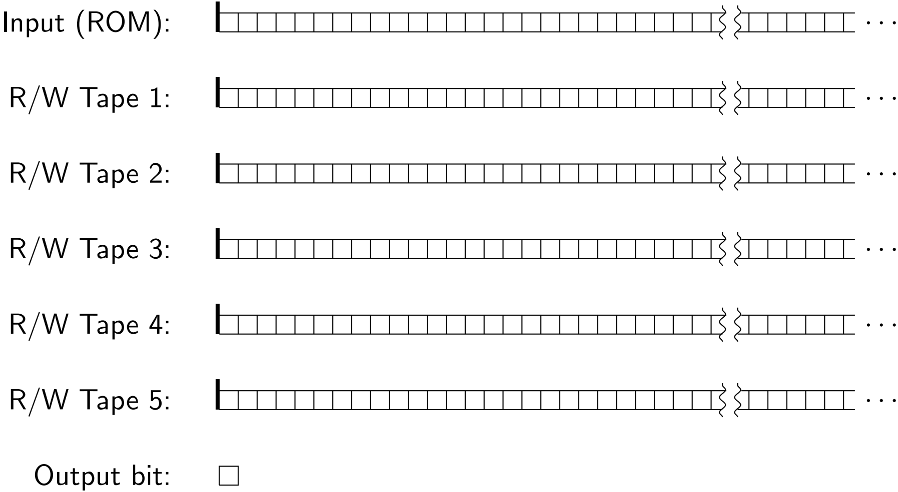
Again assume $V = \{0, 1, \ldots, n-1\}$.

Assume also that $v_{start} = 0$ and $v_{end} = 1$.

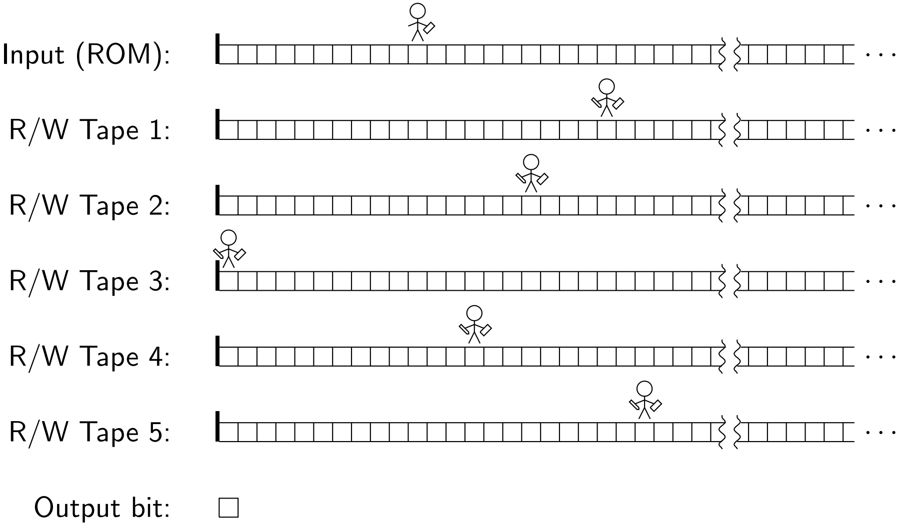
Assume the adjacency matrix is presented as a binary string of length n^2.

Implement the algorithm on a *Turing machine*.

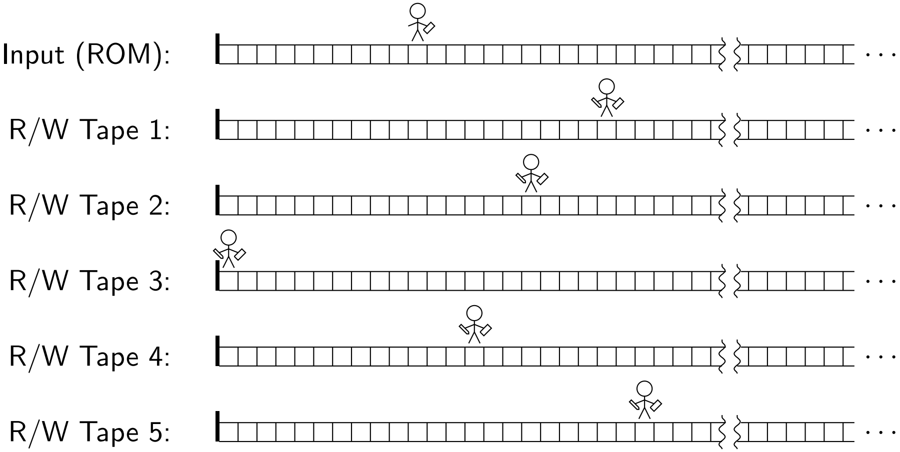
Turing machine



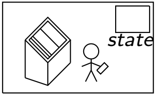
Turing machine



Turing machine



Output bit: ☐



Tape	In	1	2	3	4	5
char						
1^{st}?						

Turing machine

Input (ROM):

R/W Tape 1:

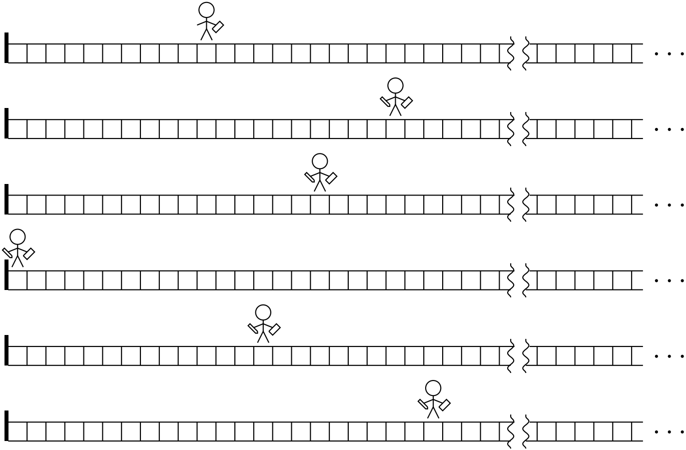
R/W Tape 2:

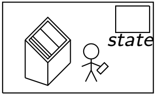
R/W Tape 3:

R/W Tape 4:

R/W Tape 5:

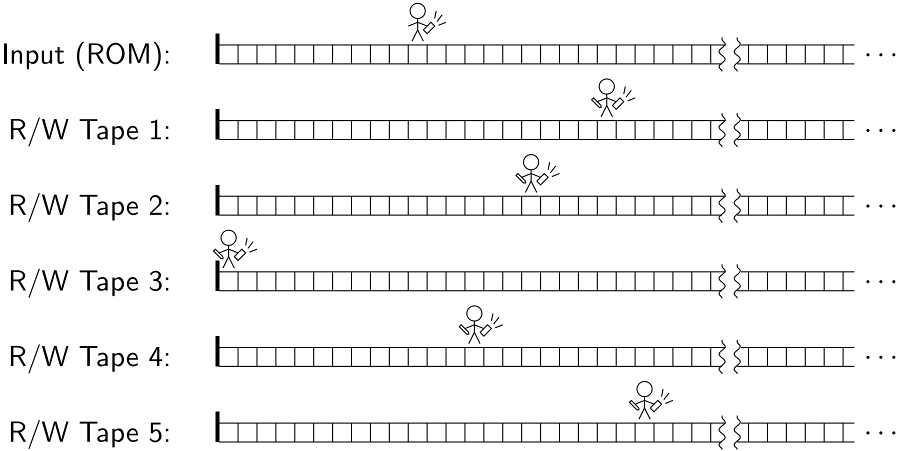
Output bit:

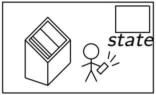




Tape	In	1	2	3	4	5
char						
1^{st}?						

Turing machine





Tape	In	1	2	3	4	5
char						
1^{st}?						

Turing machine

Input (ROM):

R/W Tape 1:

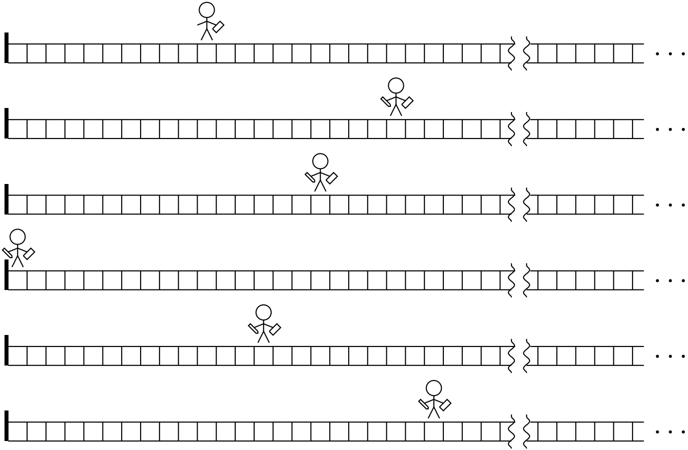
R/W Tape 2:

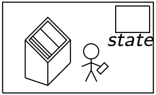
R/W Tape 3:

R/W Tape 4:

R/W Tape 5:

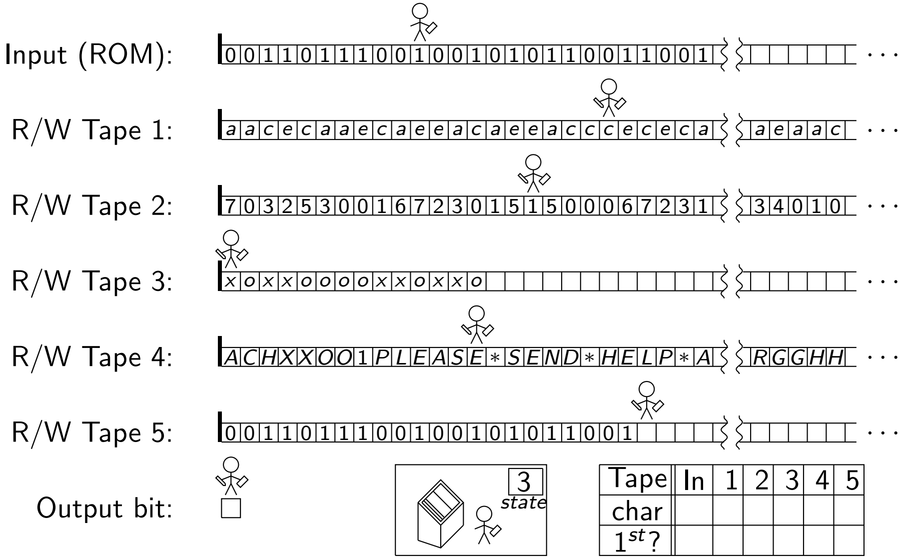
Output bit:



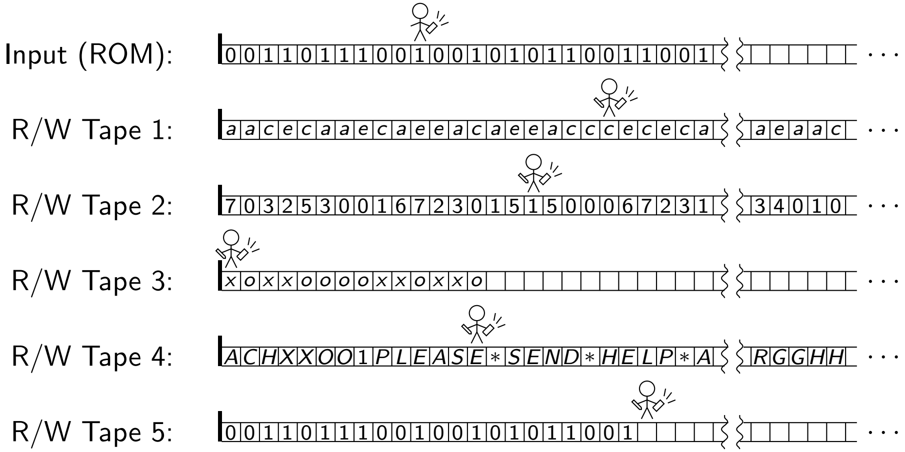


Tape	In	1	2	3	4	5
char						
1^{st}?						

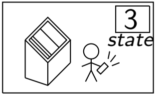
Turing machine



Turing machine

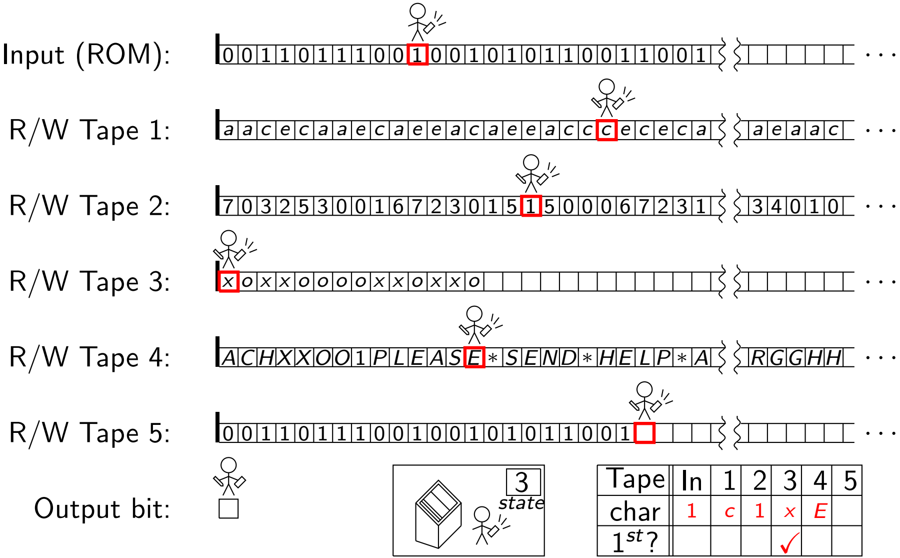


Output bit:



Tape	In	1	2	3	4	5
char						
1^{st}?						

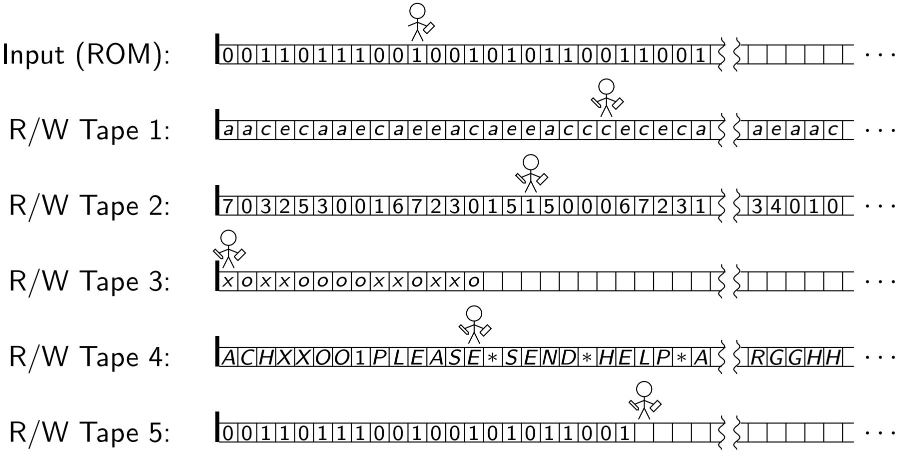
Prof asks for status

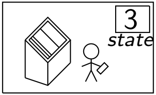


Tape	In	1	2	3	4	5
char	1	c	1	x	E	
1^{st}?				✓		

Students send reports

Turing machine

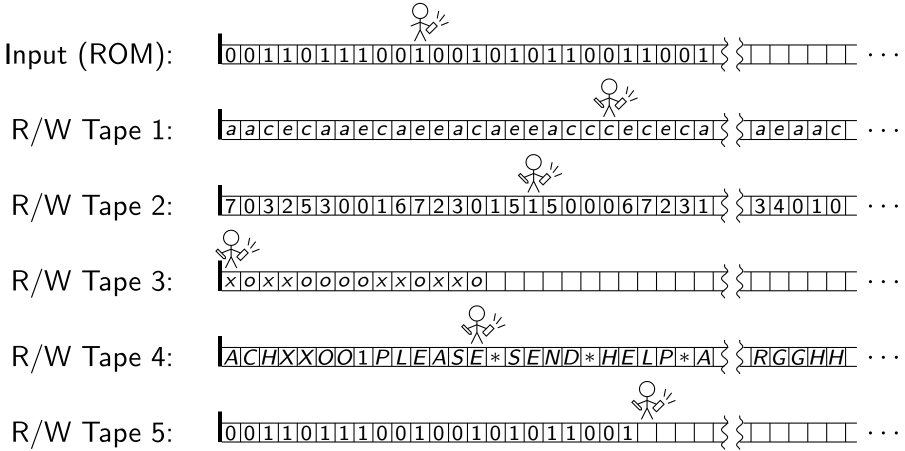




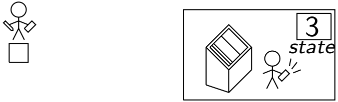
Tape	In	1	2	3	4	5
char	1	*c*	1	*x*	*E*	
1^{st}?				✓		

Prof consults manual

Turing machine



Output bit:



Tape	In	1	2	3	4	5
char	1	c	1	x	E	
1^{st}?				✓		

Prof sends instructions

Turing machine

Input (ROM): 0 0 1 1 0 1 1 1 0 0 1 0 0 1 0 1 0 1 1 0 0 1 1 0 0 1 ⋯

R/W Tape 1: a a c e c a a e c a e e a c a e e a c c c e c e c a ⋯ a e a a c ⋯

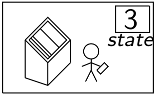
R/W Tape 2: 7 0 3 2 5 3 0 0 1 6 7 2 3 0 1 5 1 5 0 0 0 6 7 2 3 1 ⋯ 3 4 0 1 0 ⋯

R/W Tape 3: x o x x o o o o x x o x x o ⋯

R/W Tape 4: A C H X X O O 1 P L E A S $\boxed{O}$ * S E N D * H E L P * A ⋯ R G G H H ⋯

R/W Tape 5: 0 0 1 1 0 1 1 1 0 0 1 0 0 1 0 1 0 1 1 0 0 1 ⋯

Output bit: □



Tape	In	1	2	3	4	5
char	1	c	1	x	E	
1^{st}?				✓		

Action 1: 4 writes “O”.

Turing machine

Input (ROM):

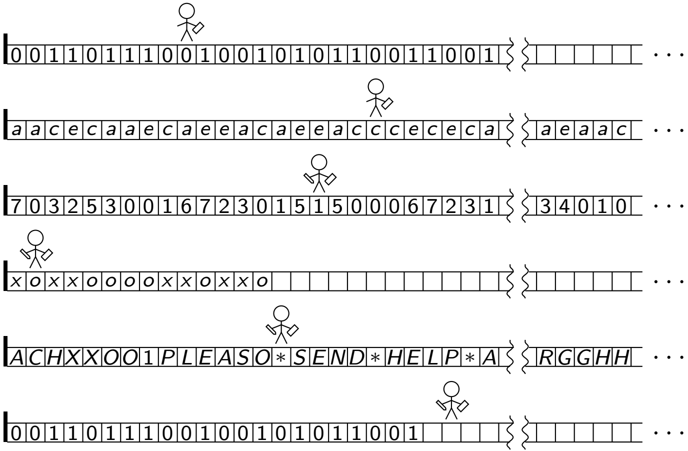
R/W Tape 1:

R/W Tape 2:

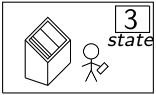
R/W Tape 3:

R/W Tape 4:

R/W Tape 5:



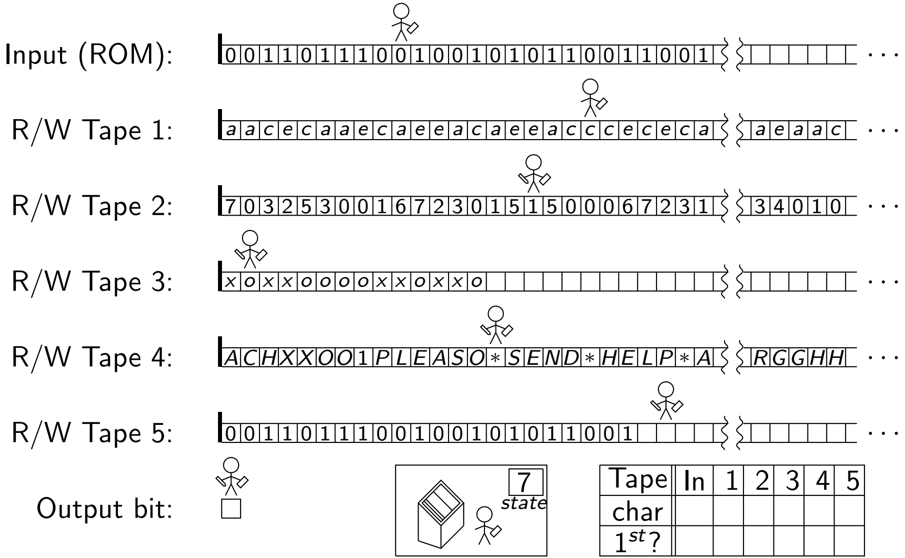
Output bit:



Tape	In	1	2	3	4	5
char	1	c	1	x	E	
1^{st}?				✓		

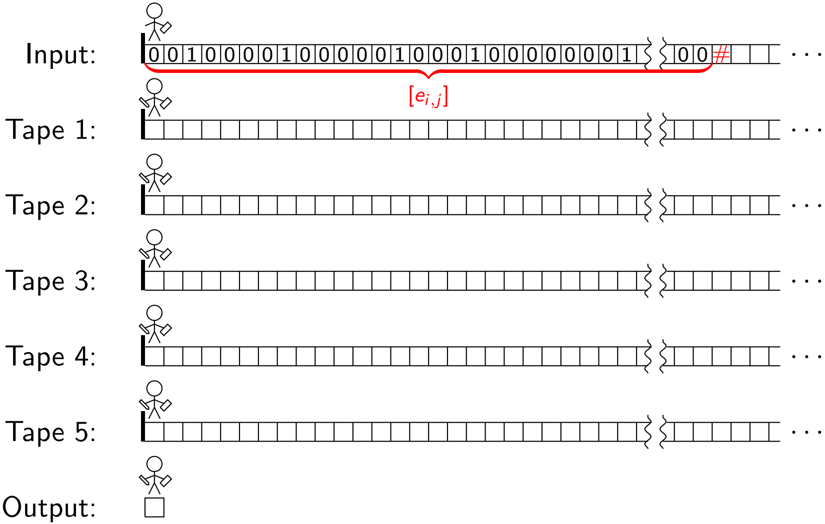
Action 2: Students move as directed

Turing machine

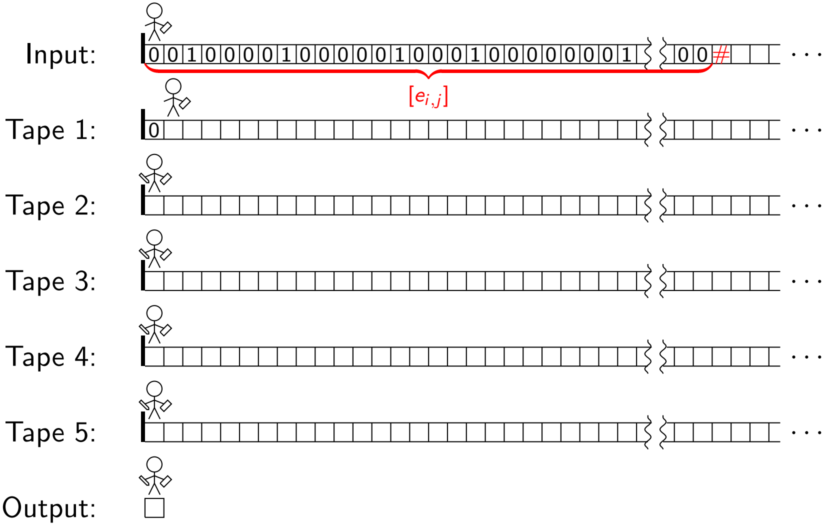


Action 3: Update state, ready for next "step"

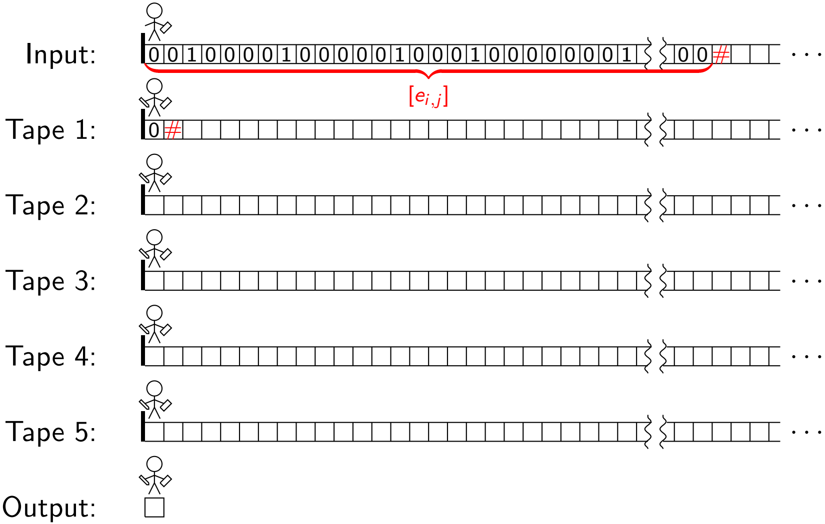
Implementing the algorithm for *PATH*



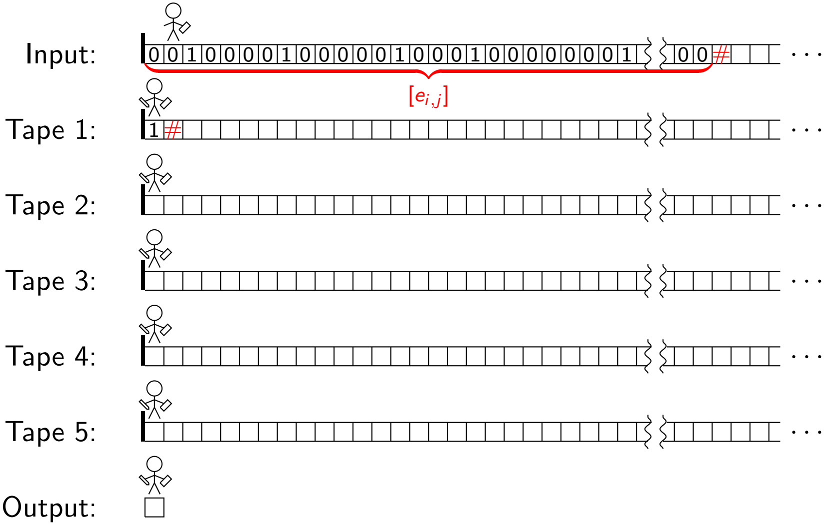
Implementing the algorithm for *PATH*



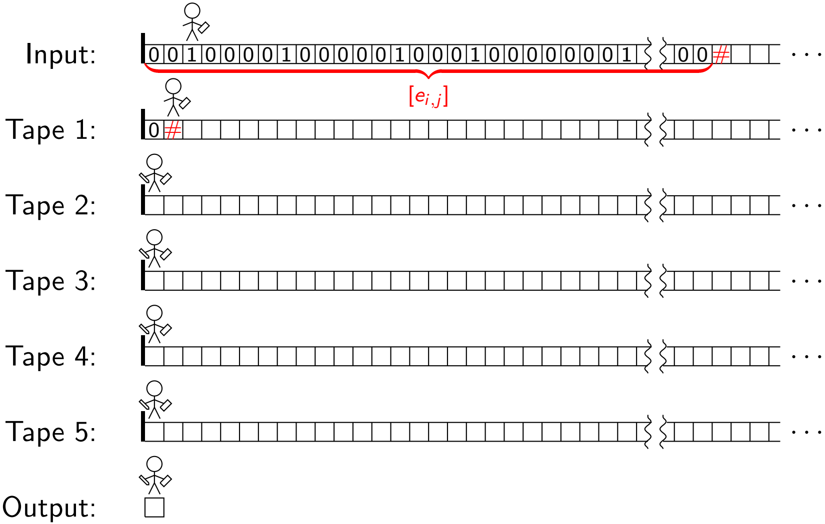
Implementing the algorithm for *PATH*



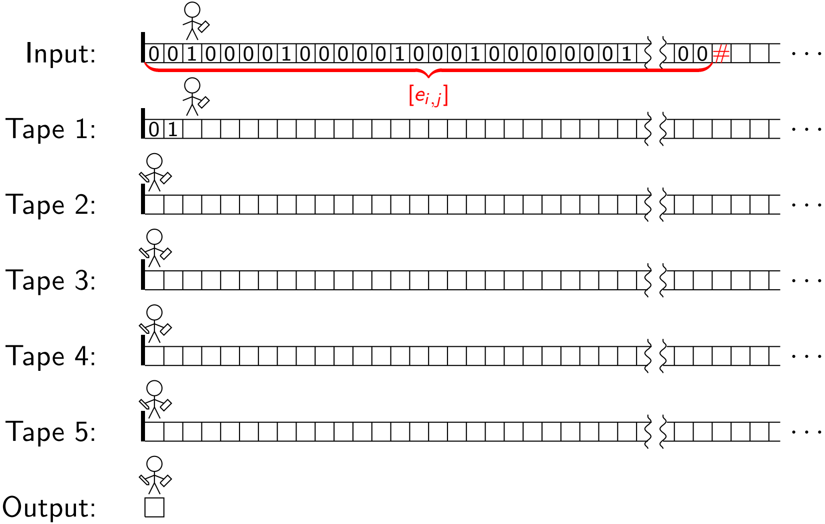
Implementing the algorithm for *PATH*



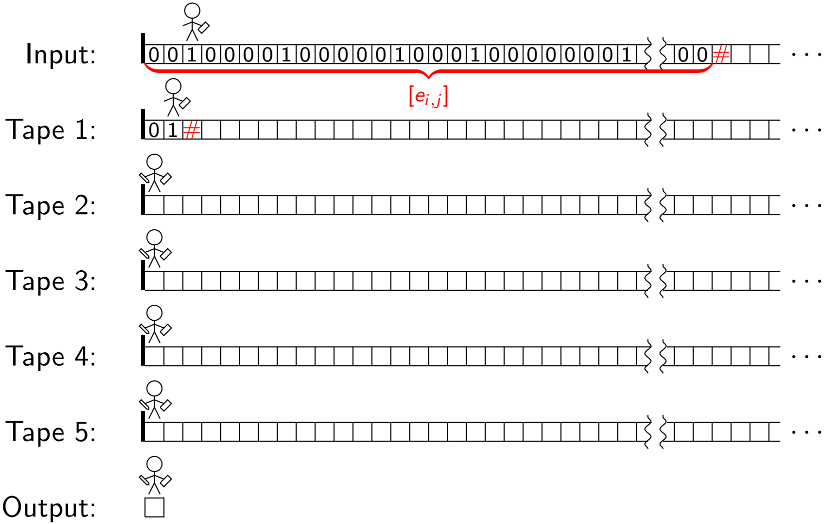
Implementing the algorithm for *PATH*



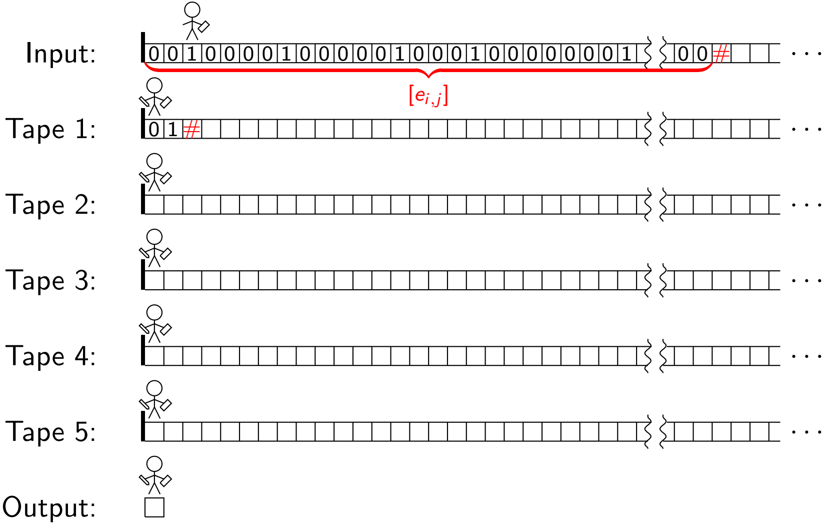
Implementing the algorithm for *PATH*



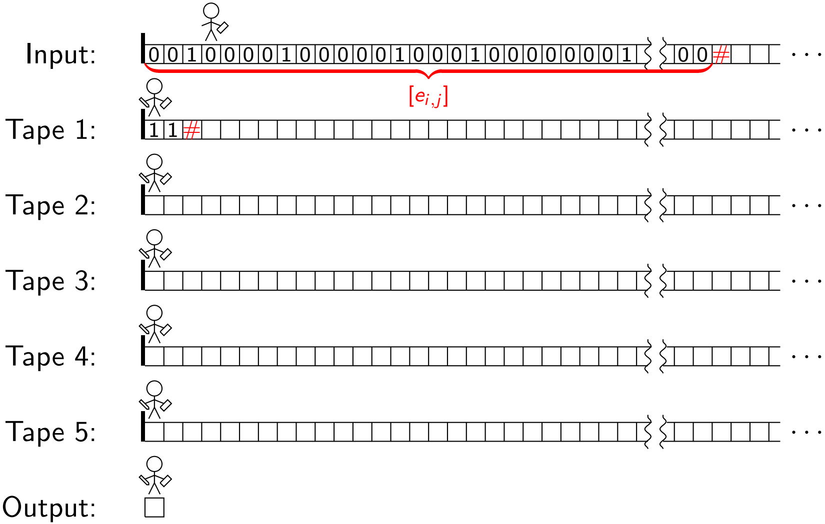
Implementing the algorithm for *PATH*



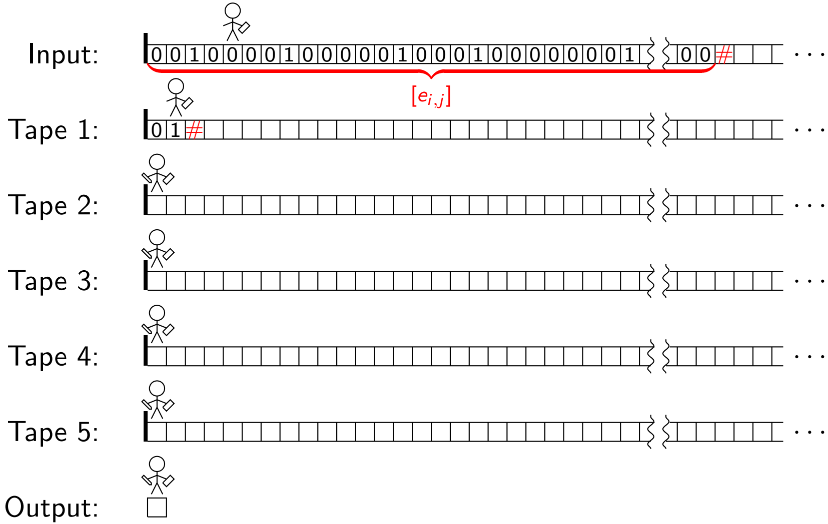
Implementing the algorithm for *PATH*



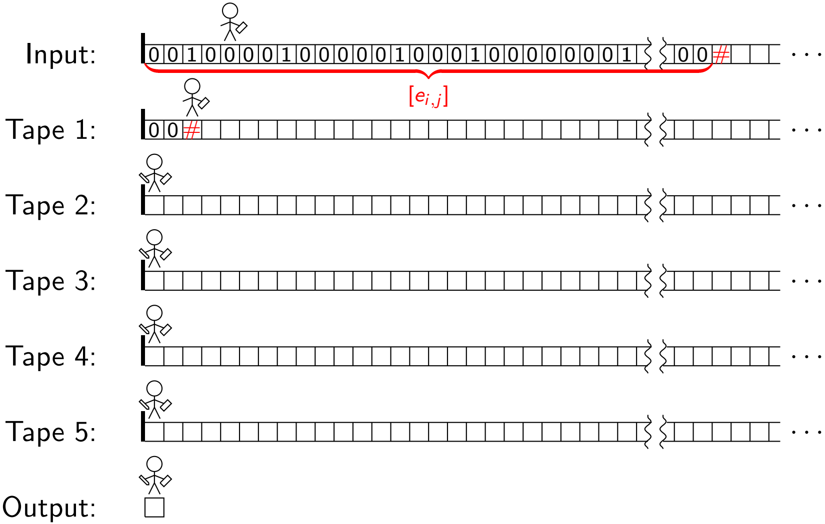
Implementing the algorithm for *PATH*



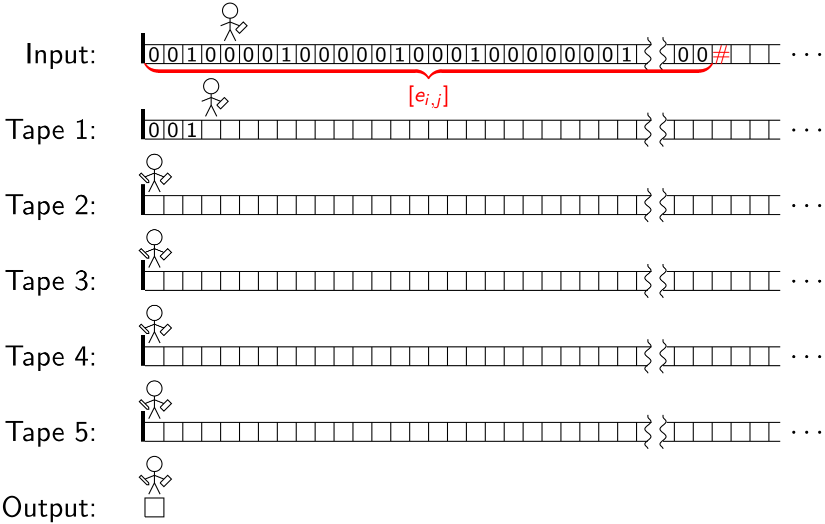
Implementing the algorithm for *PATH*



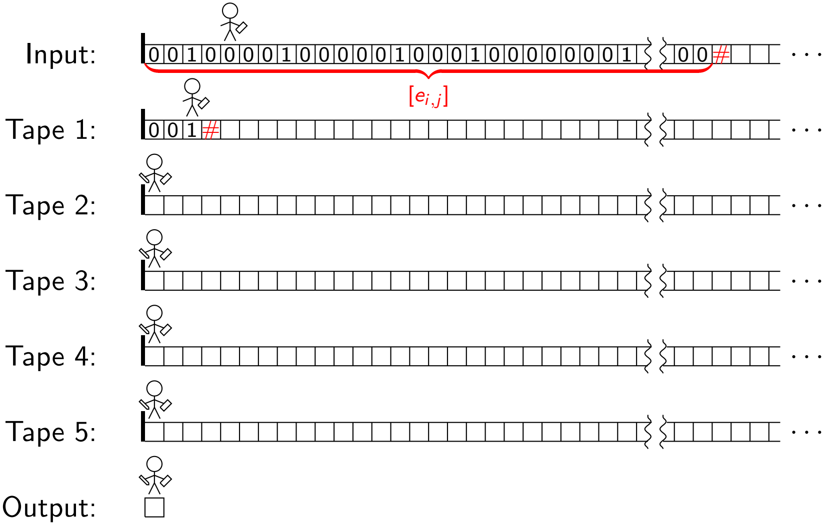
Implementing the algorithm for *PATH*



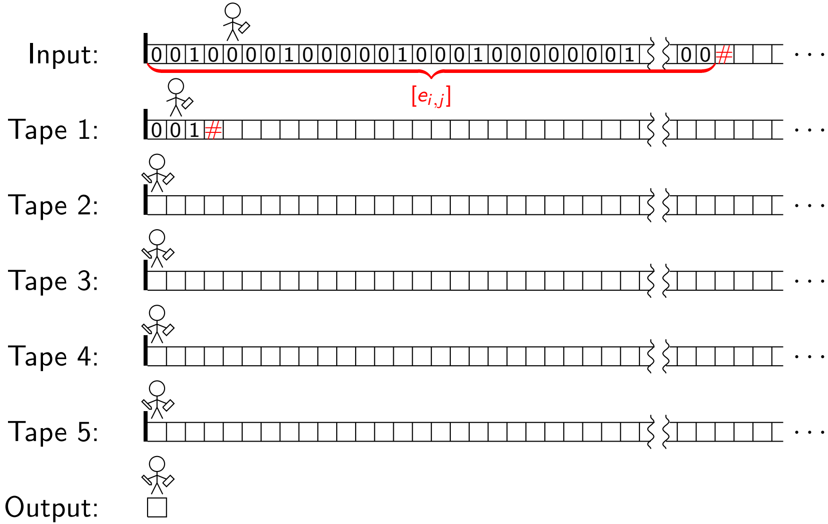
Implementing the algorithm for *PATH*



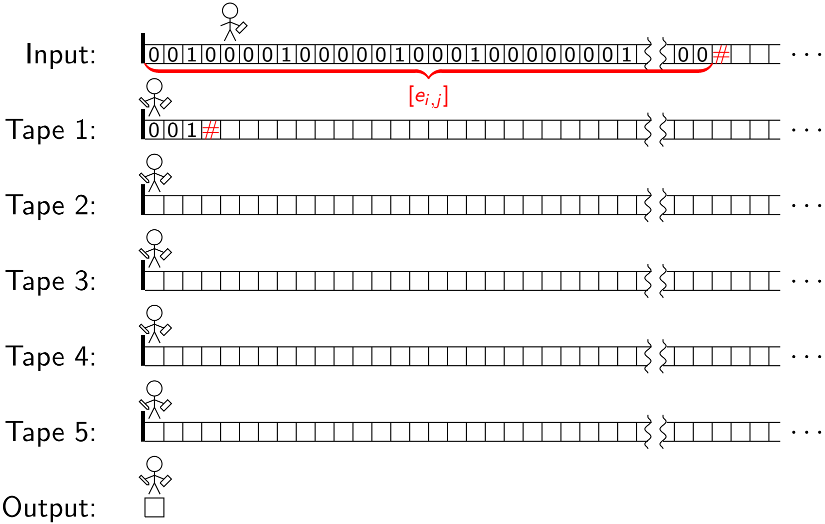
Implementing the algorithm for *PATH*



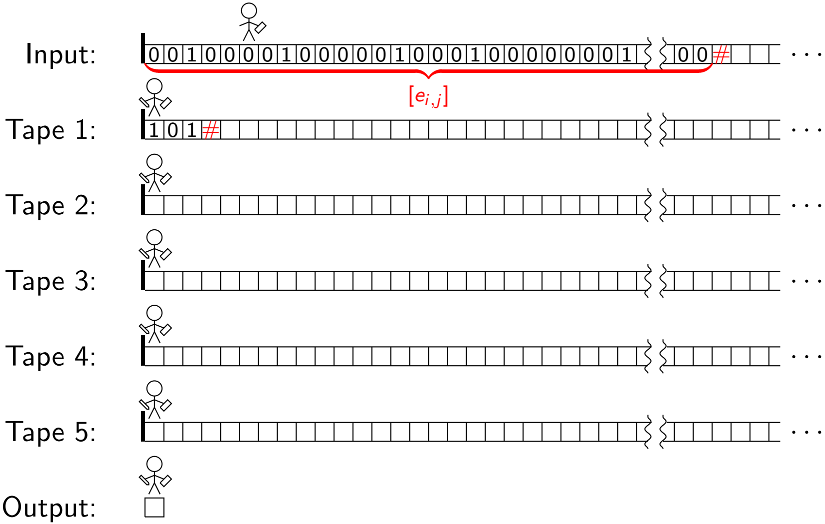
Implementing the algorithm for *PATH*



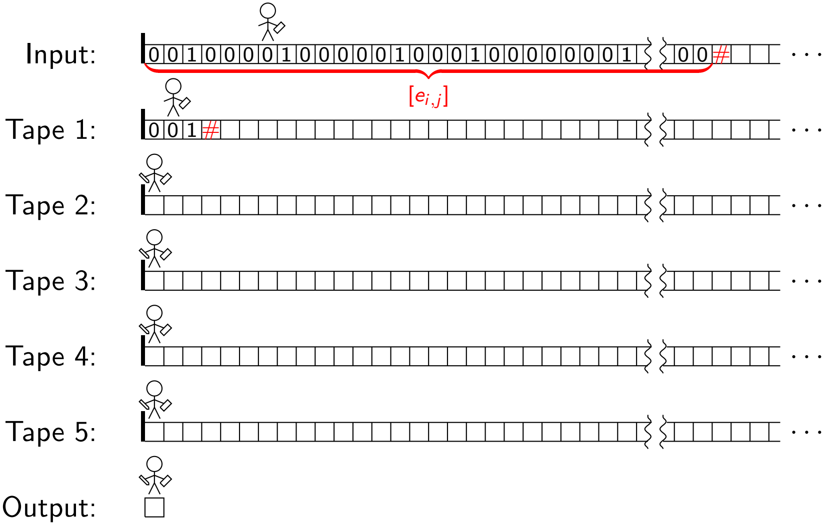
Implementing the algorithm for *PATH*



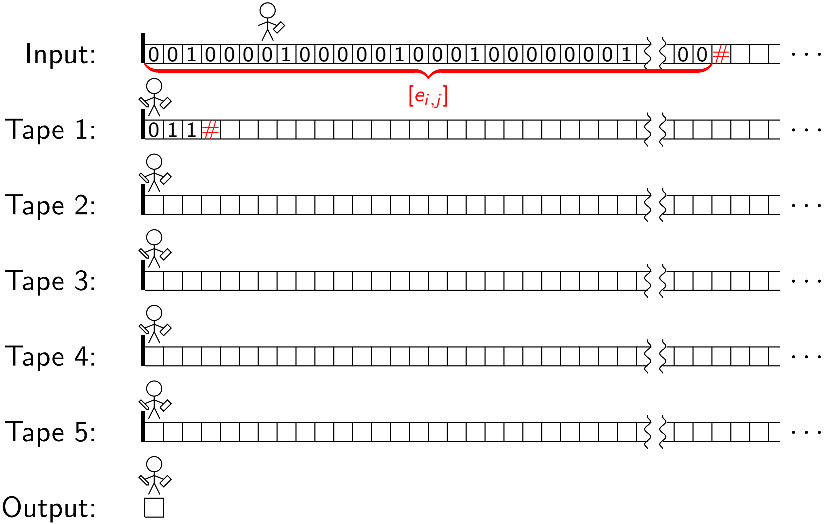
Implementing the algorithm for *PATH*

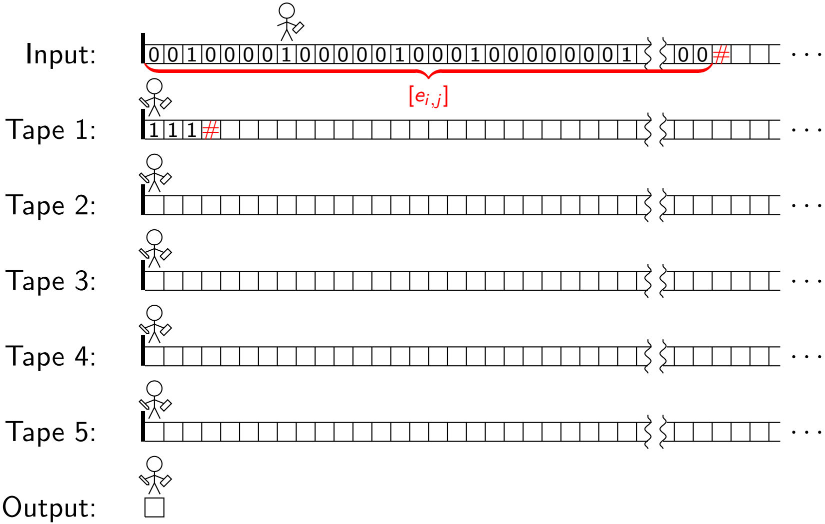


Implementing the algorithm for *PATH*



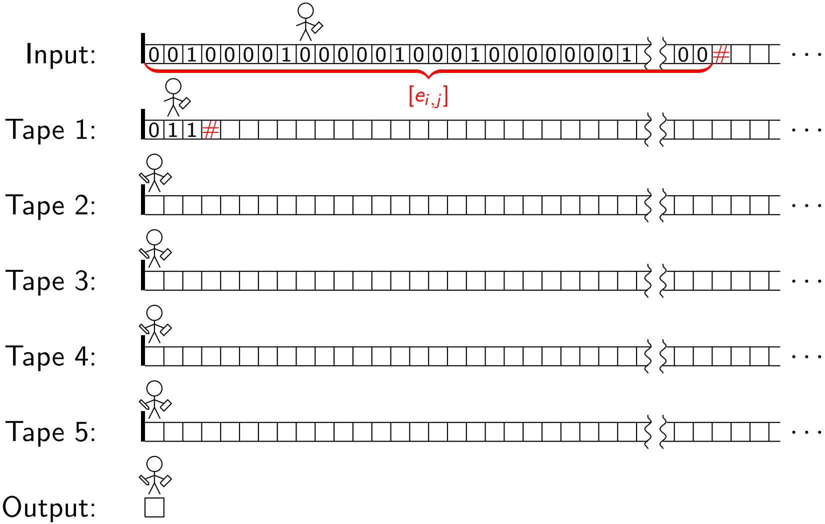
Implementing the algorithm for *PATH*



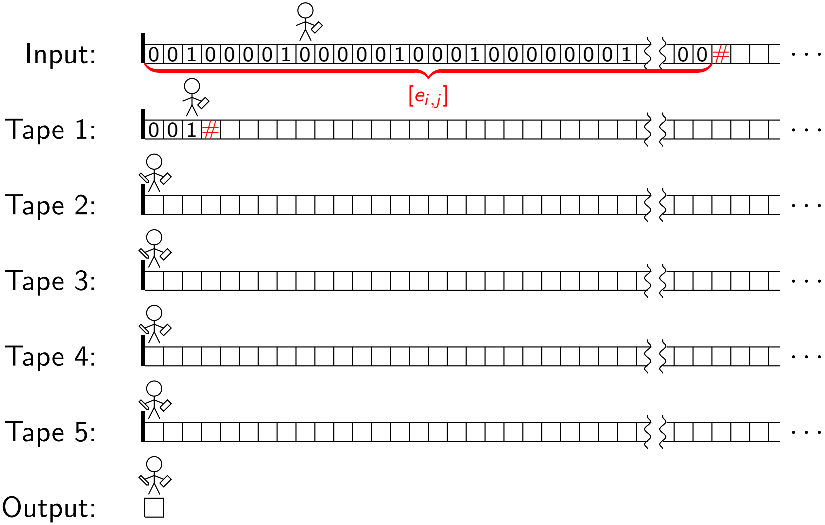


Input:
001000010000010001000000001 00#
[e_{i,j}]
Tape 1:
111#
Tape 2:
Tape 3:
Tape 4:
Tape 5:
Output:

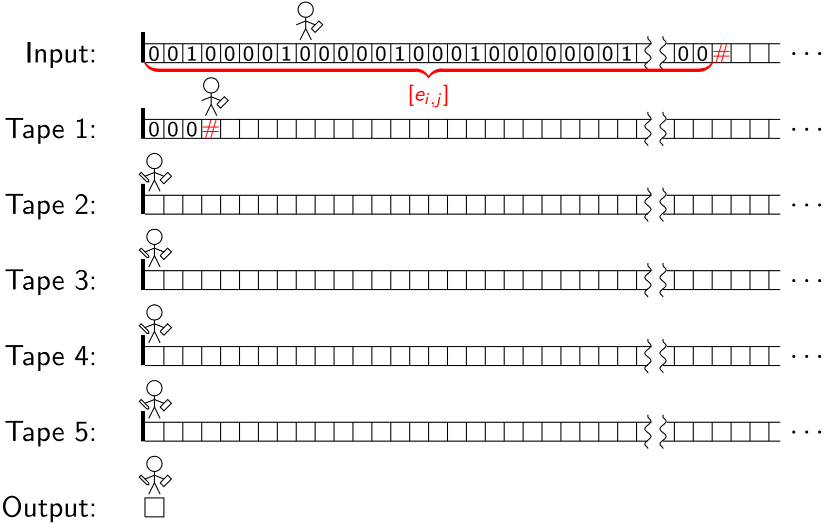
Implementing the algorithm for *PATH*



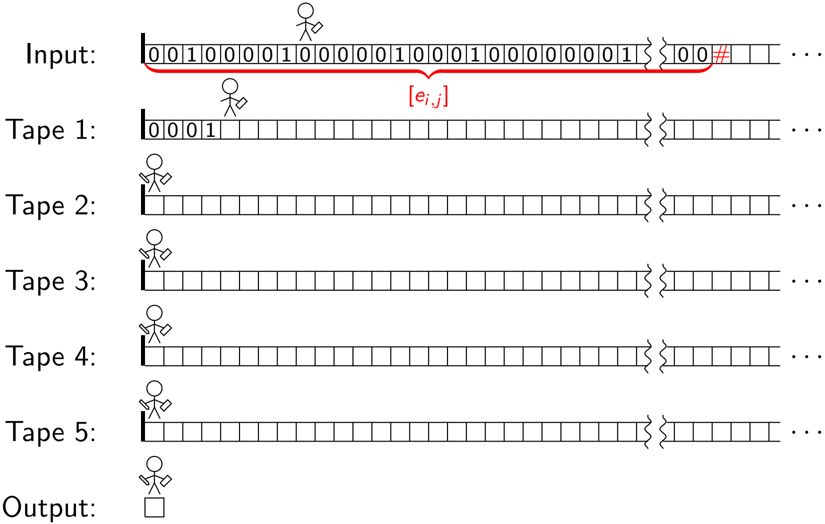
Implementing the algorithm for *PATH*



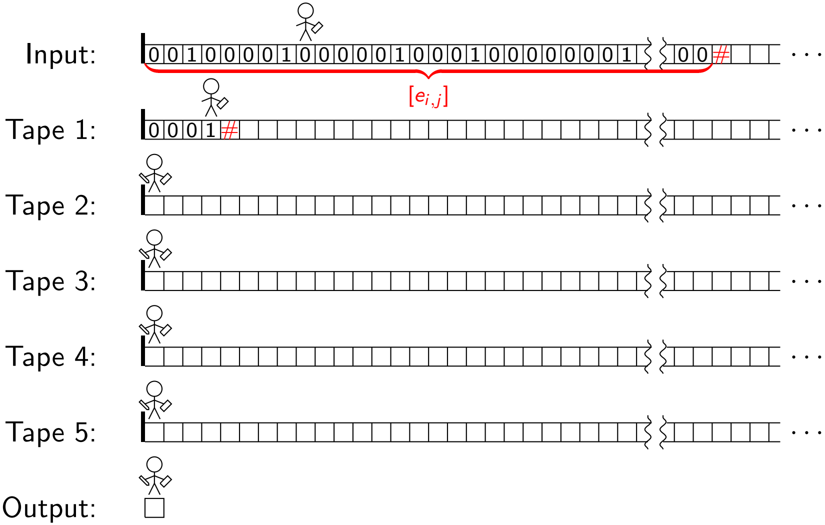
Implementing the algorithm for *PATH*



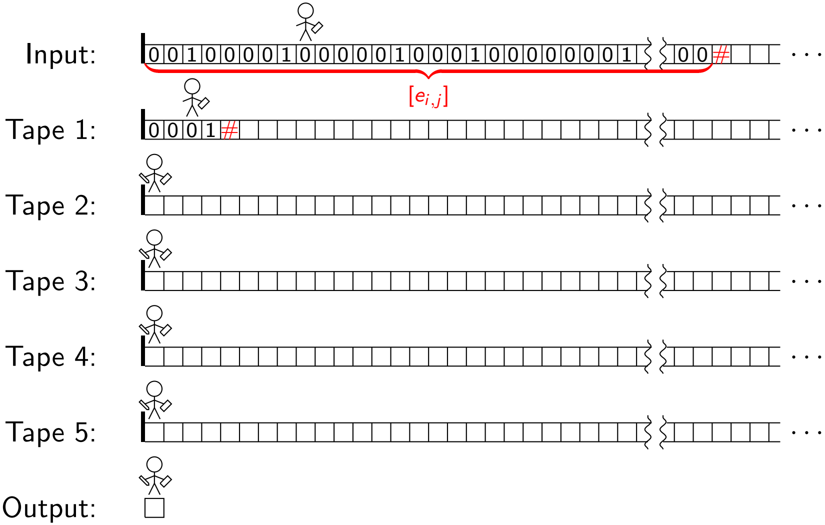
Implementing the algorithm for *PATH*



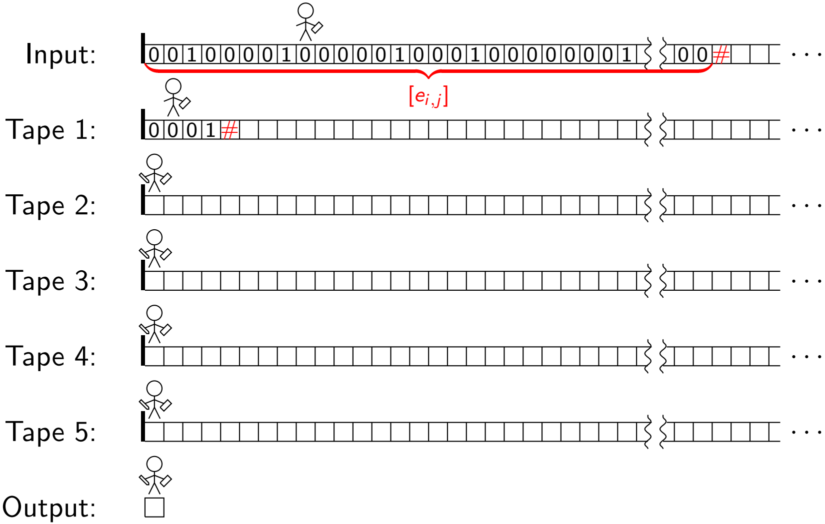
Implementing the algorithm for *PATH*



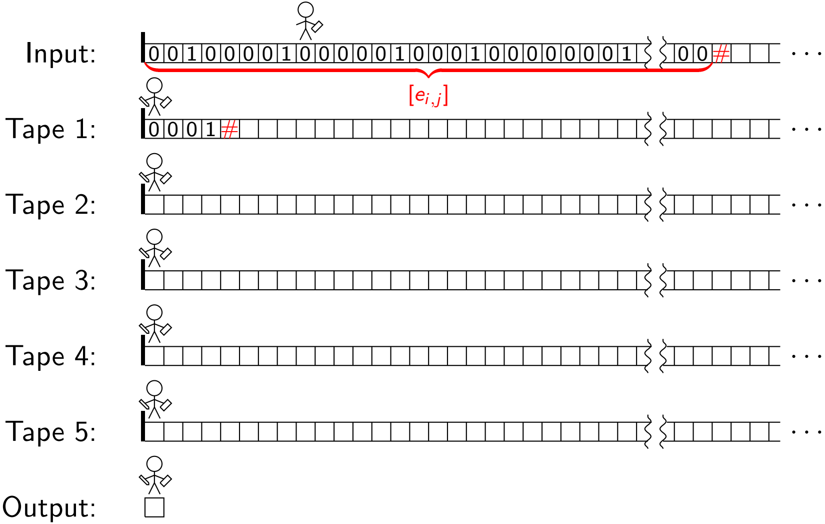
Implementing the algorithm for *PATH*



Implementing the algorithm for *PATH*



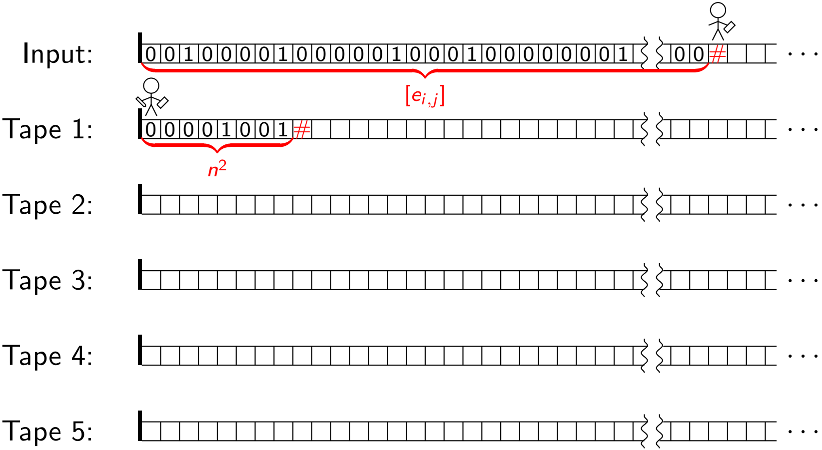
Implementing the algorithm for *PATH*



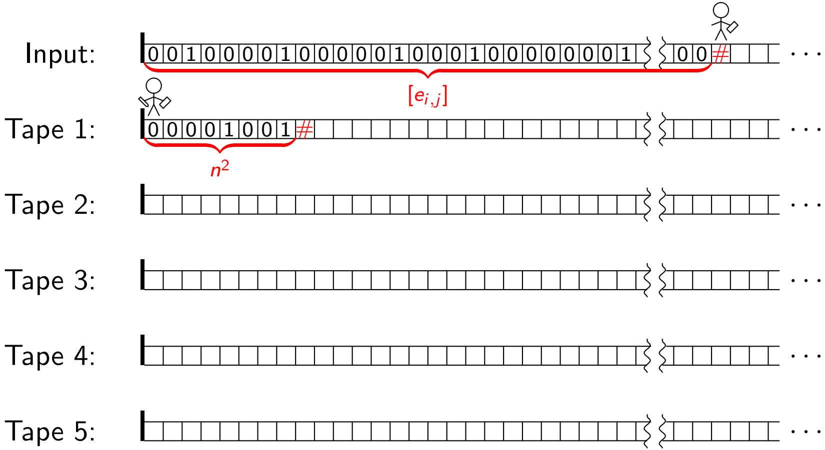
Implementing the algorithm for *PATH*

Many steps later ...

Implementing the algorithm for *PATH*

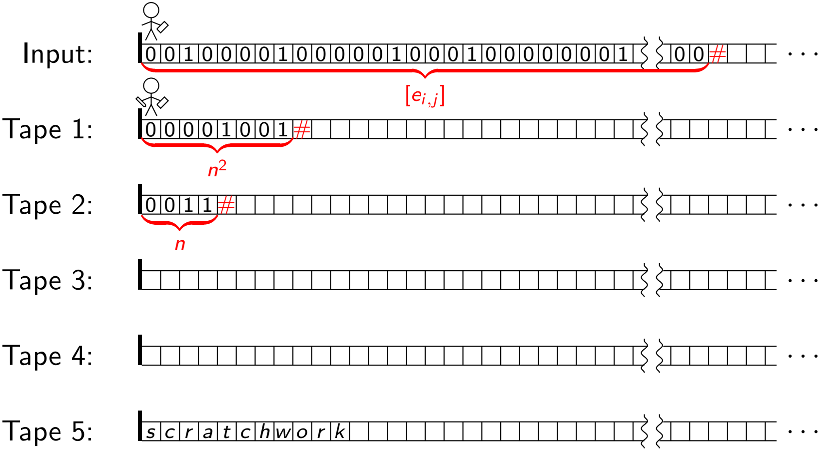


Implementing the algorithm for *PATH*

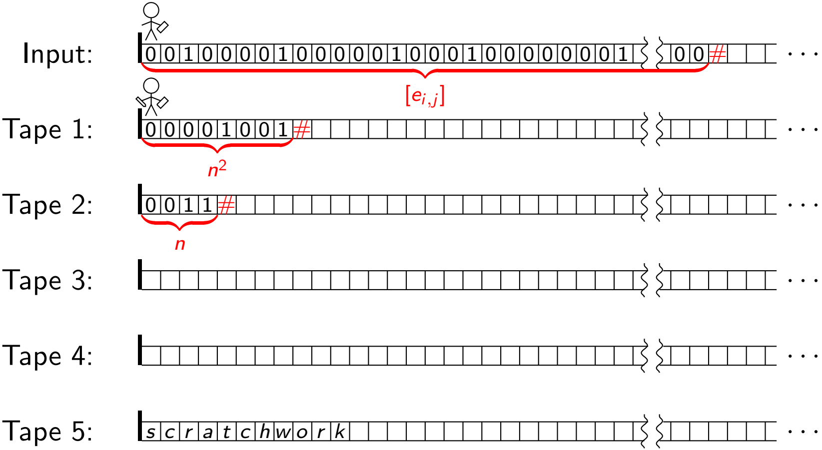


Many more steps later ...

Implementing the algorithm for *PATH*

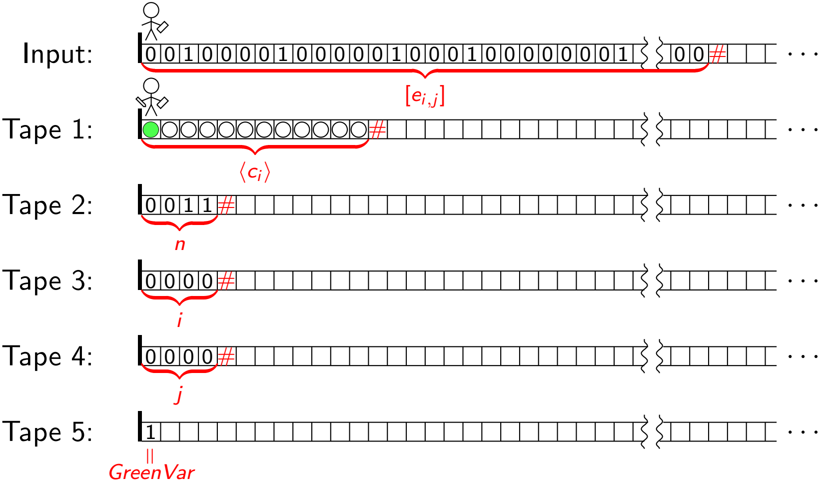


Implementing the algorithm for *PATH*

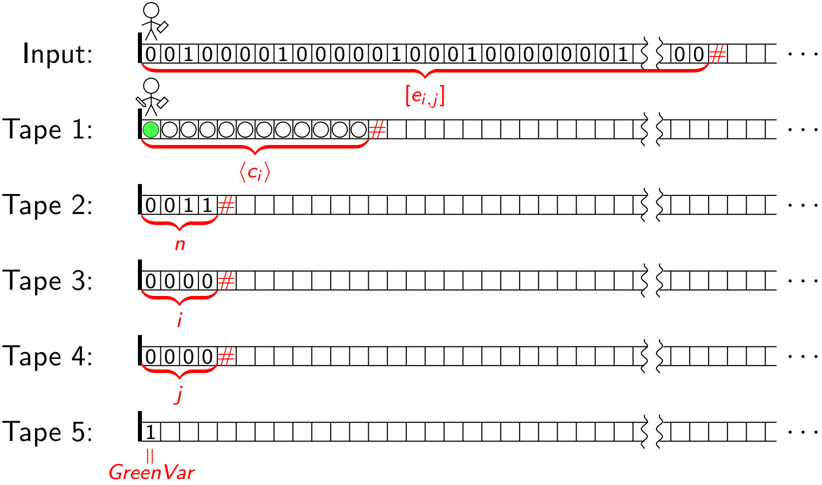


Set up variables . . .

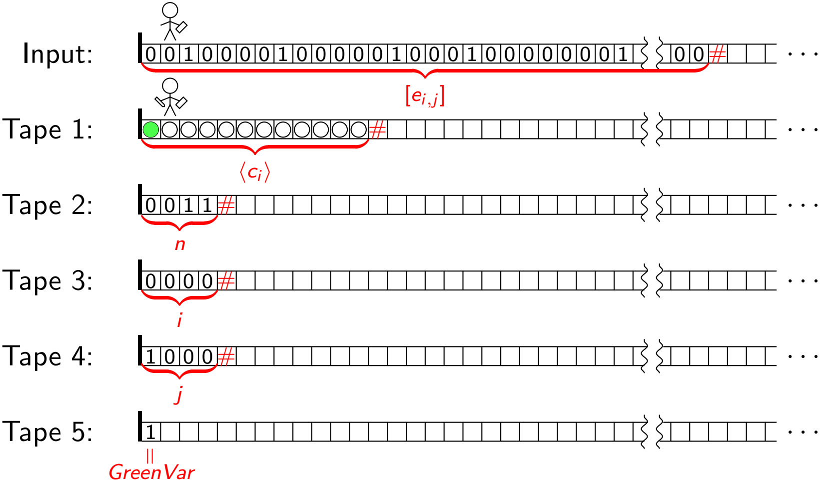
Implementing the algorithm for *PATH*



Set up variables . . .

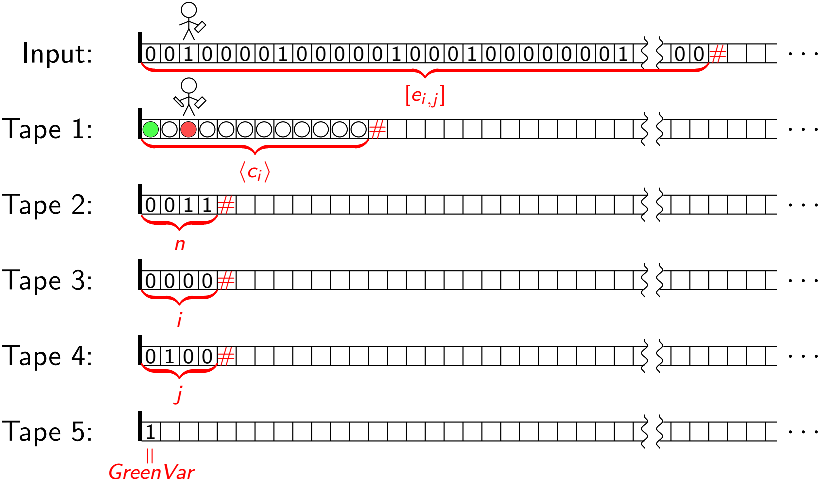


Main loop: For $i, j = 0$ to $n - 1$ . . .

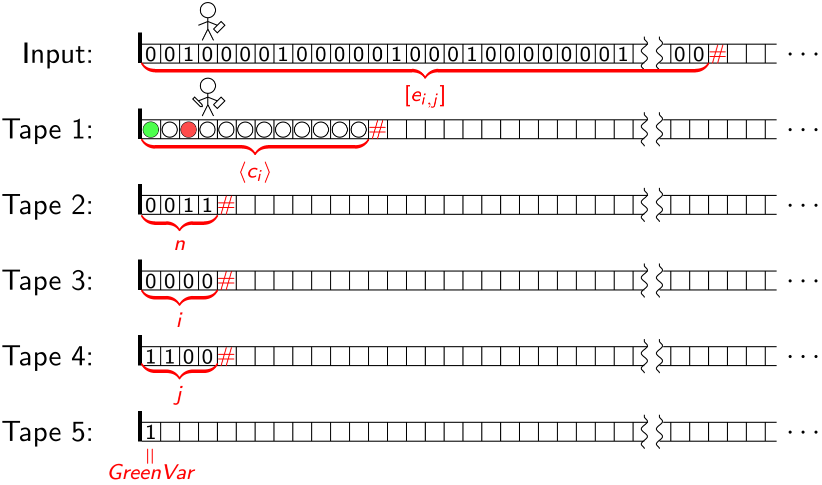


Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*

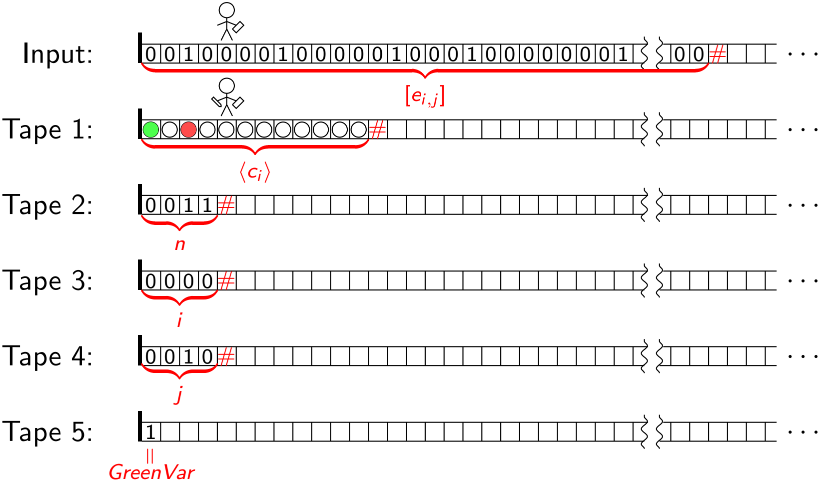


Main loop: For $i, j = 0$ to $n - 1$ . . .



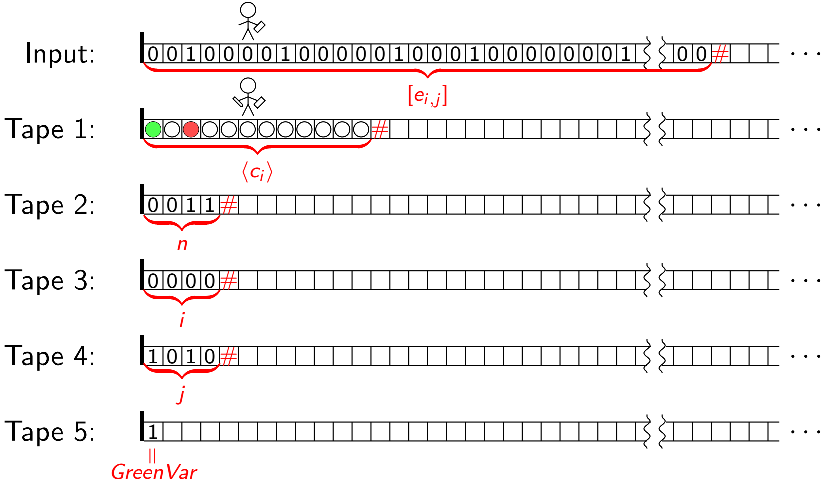
Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*

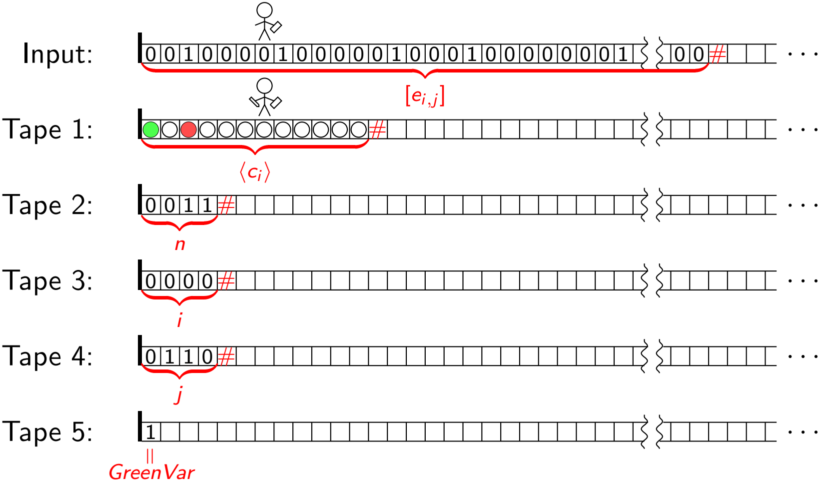


Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*

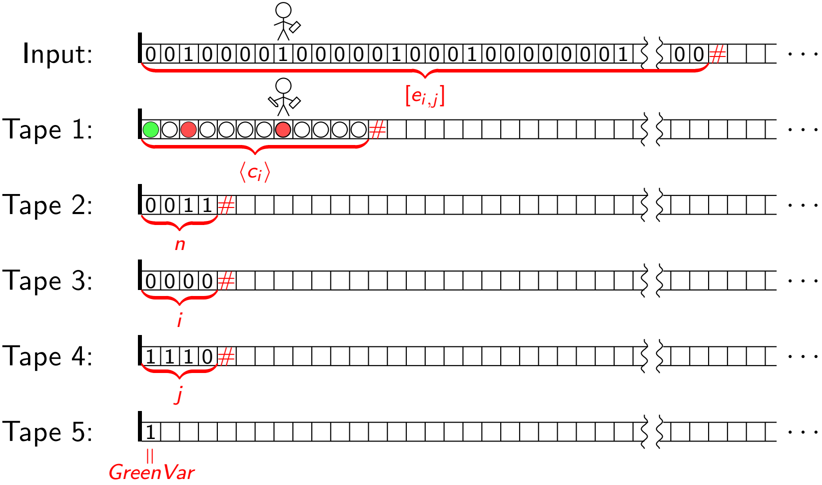


Main loop: For $i, j = 0$ to $n - 1$ . . .



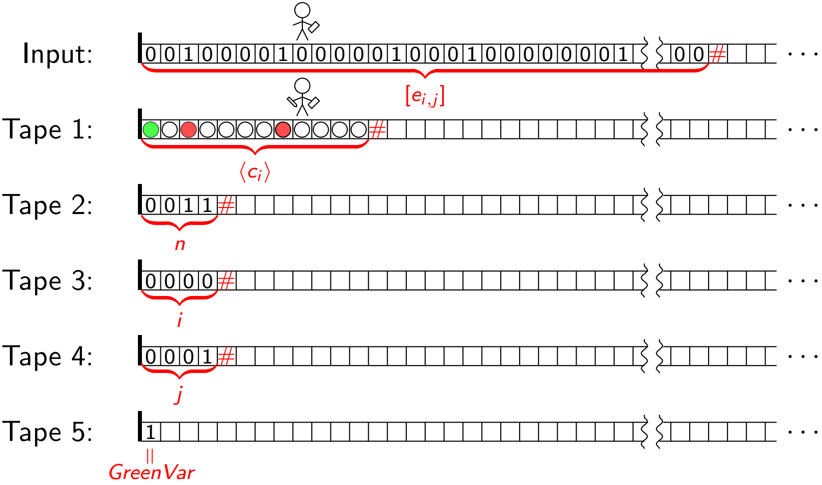
Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*



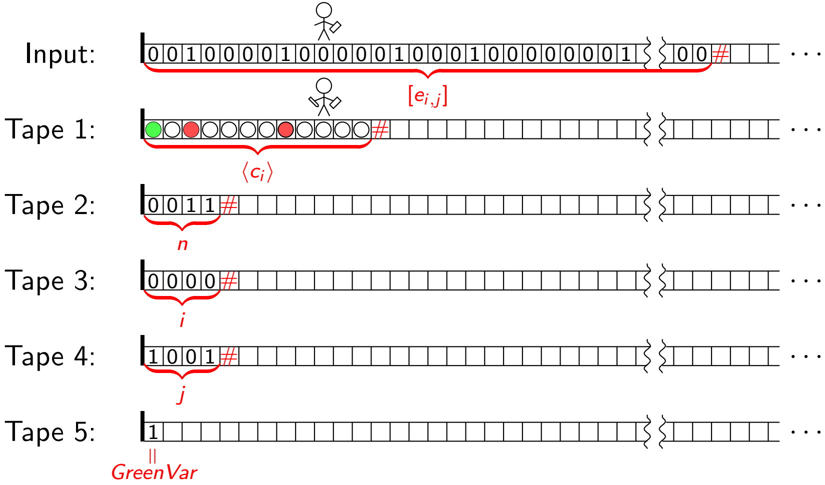
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



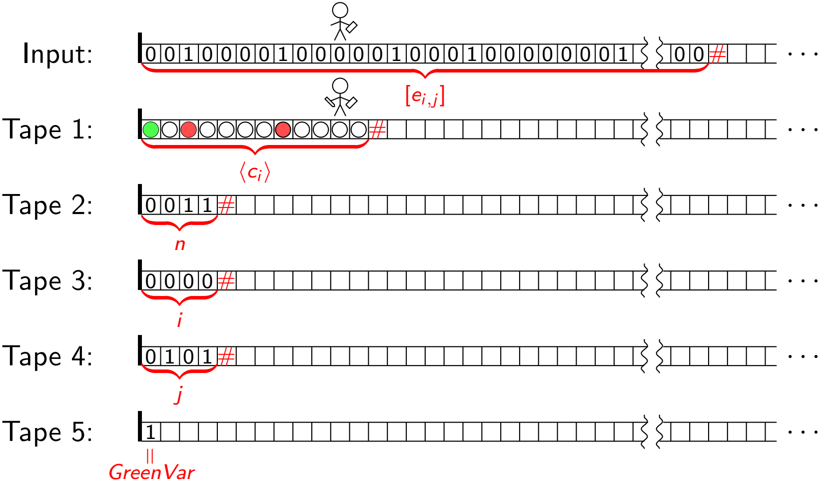
Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*

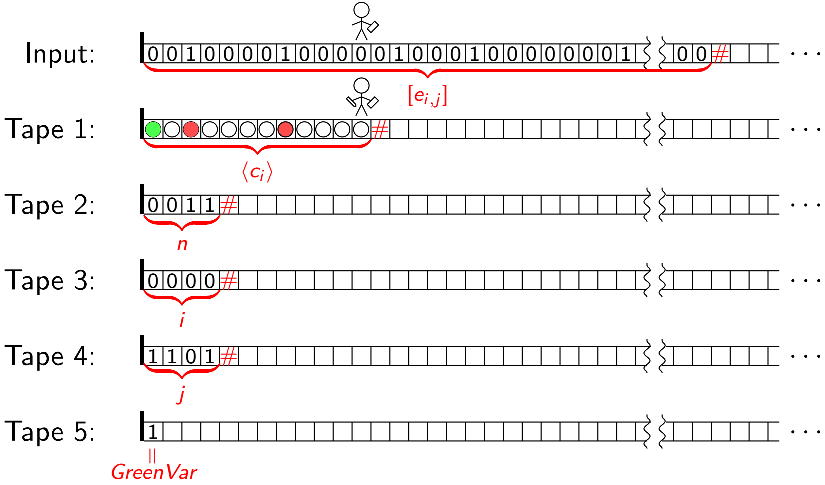


Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*

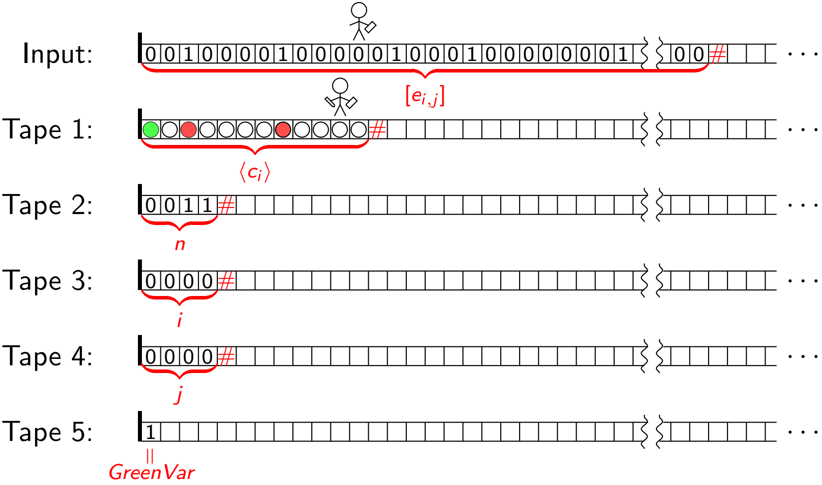


Main loop: For $i, j = 0$ to $n - 1$ ...



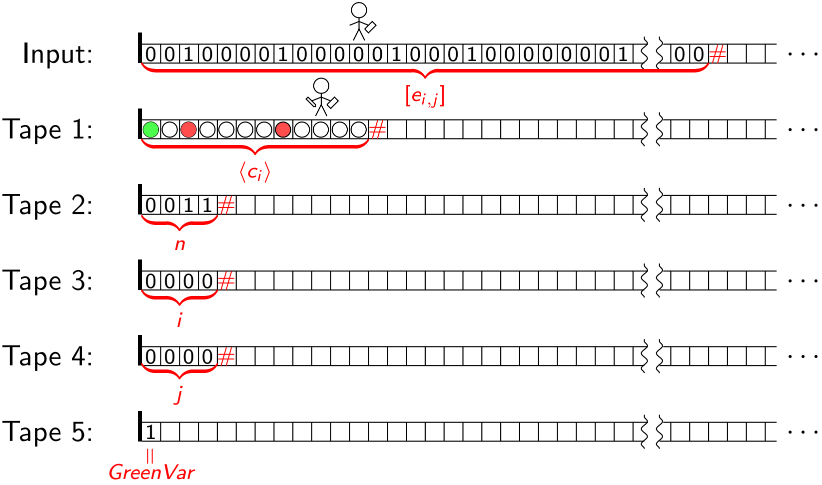
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



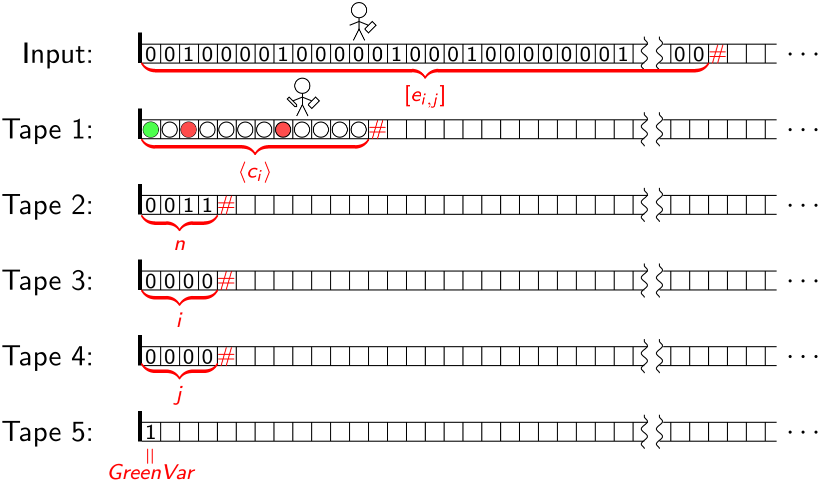
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



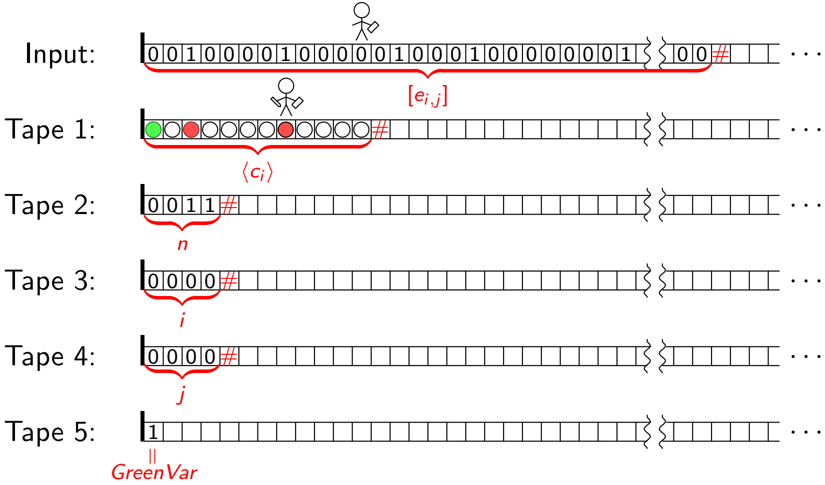
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



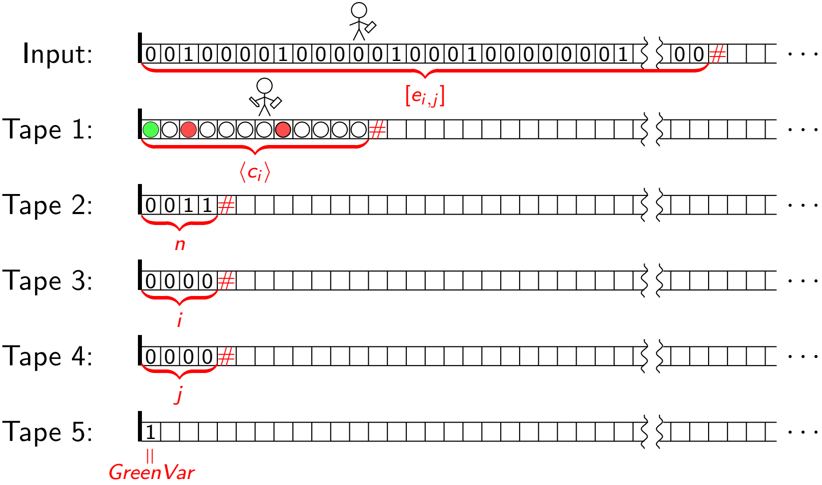
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



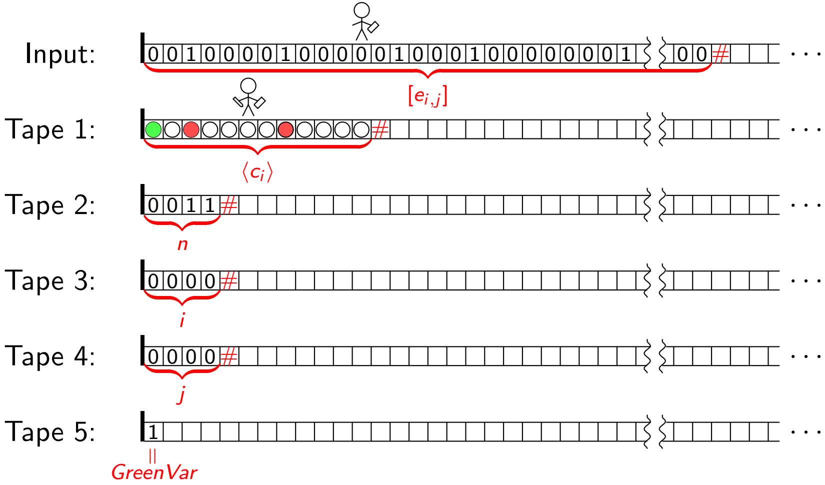
Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*

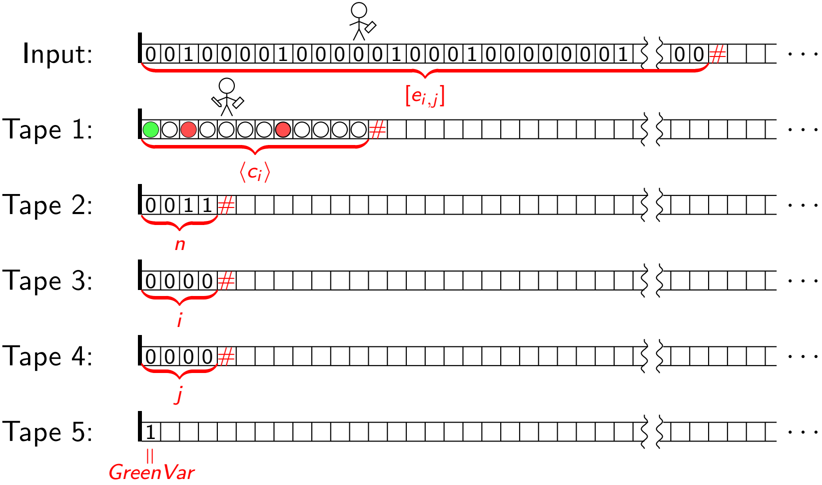


Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*

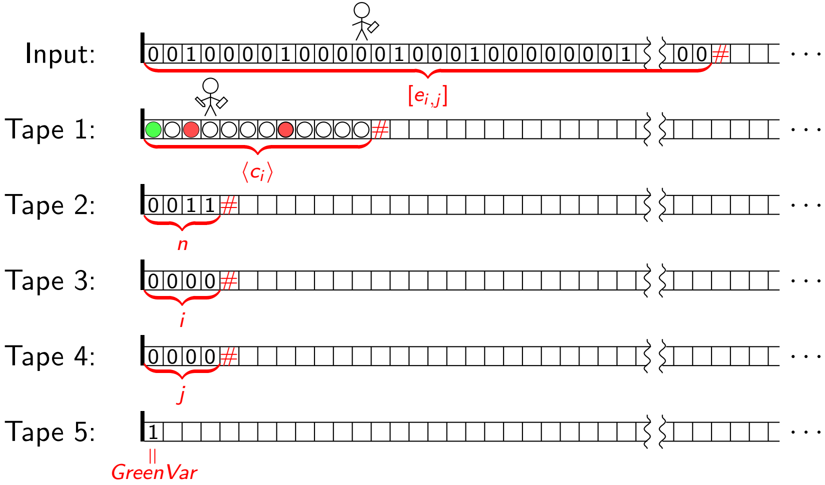


Main loop: For $i, j = 0$ to $n - 1$ . . .



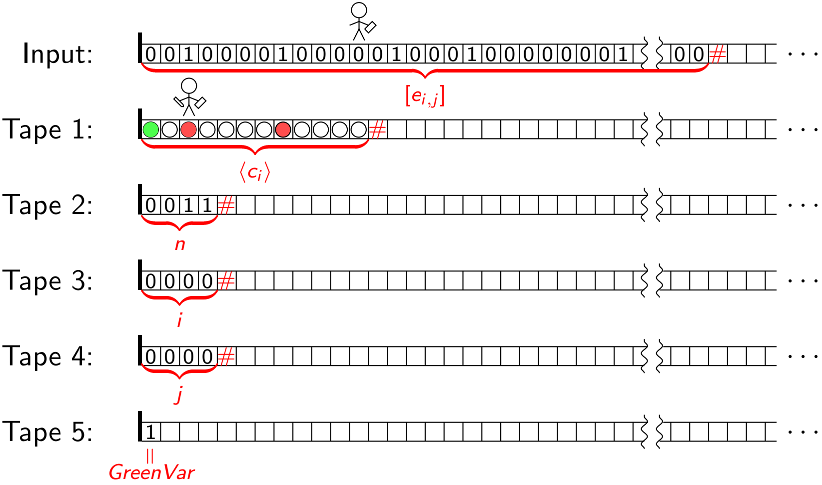
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



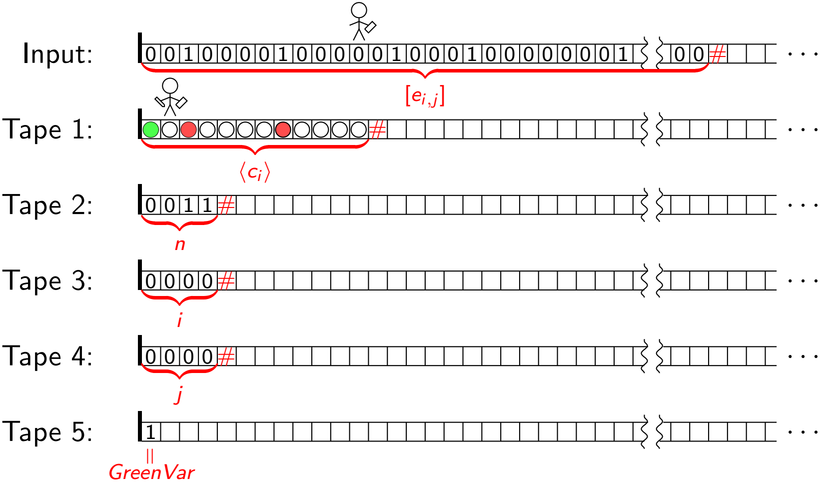
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



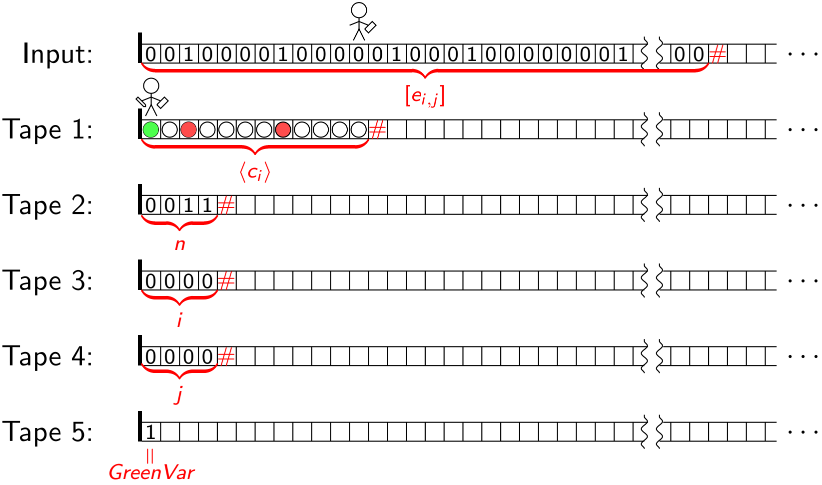
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



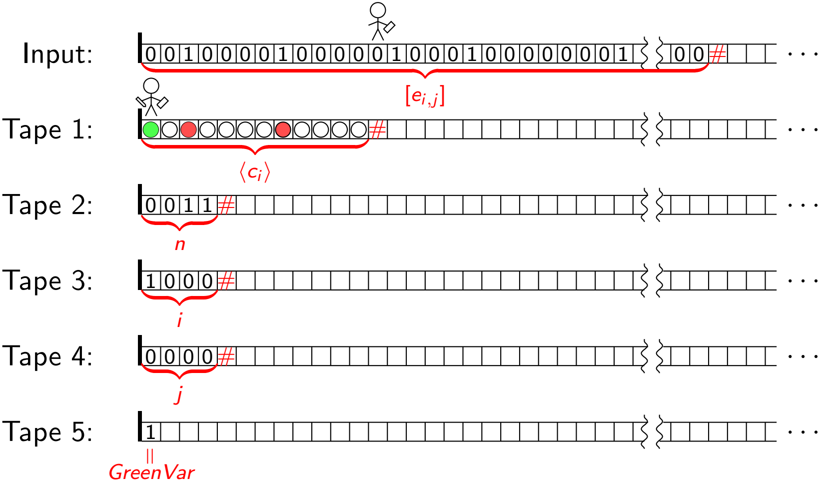
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*



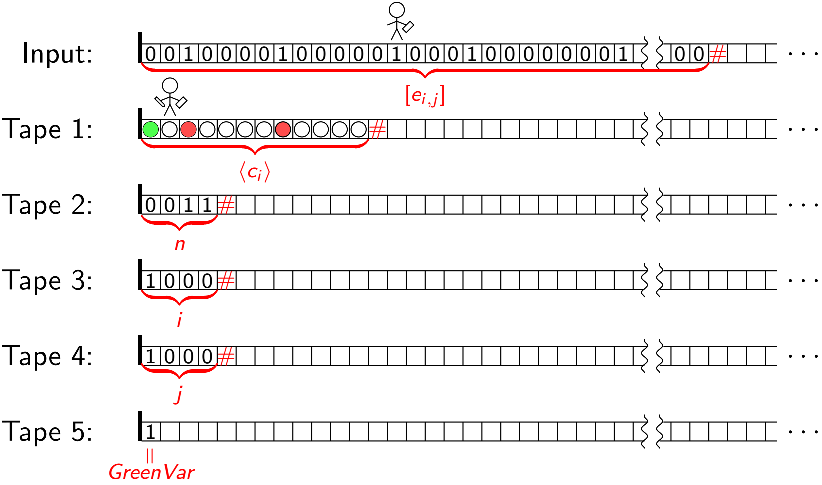
Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*



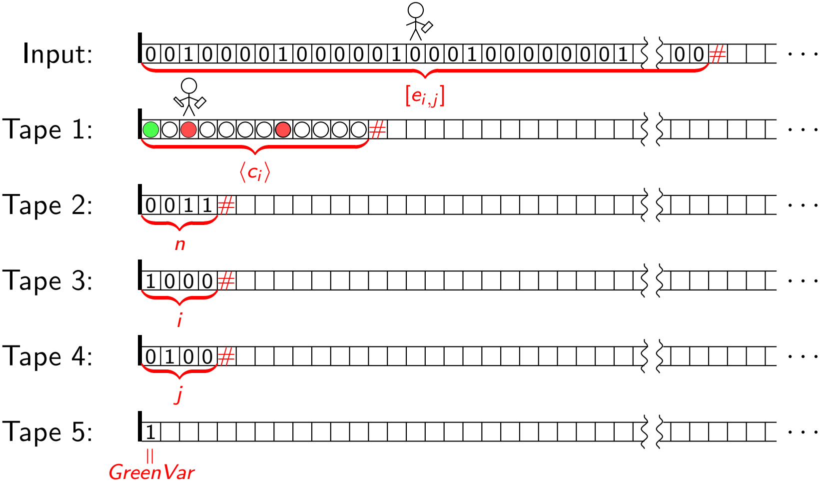
Main loop: For $i, j = 0$ to $n - 1$ . . .

Implementing the algorithm for *PATH*

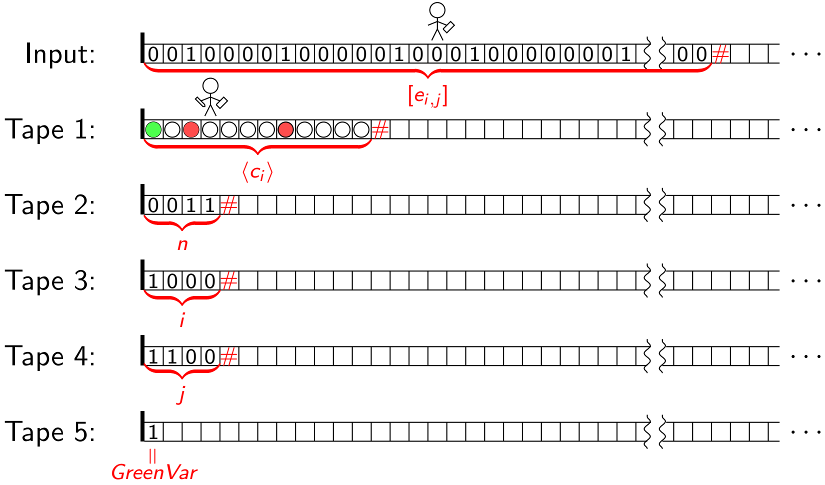


Main loop: For $i, j = 0$ to $n - 1$ ...

Implementing the algorithm for *PATH*



Main loop: For $i, j = 0$ to $n-1$ . . .



Main loop: and so on . . .

Pseudo-code revisited

Point: overhead needed to keep track of i, j, c_i, c_j.

Thus:

- While $GreenVar = yes$ do: n loops
  - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$ n^2 cases
    - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$.

Pseudo-code revisited

Point: overhead needed to keep track of i, j, c_i, c_j.

Thus:

- While $GreenVar = yes$ do: — n loops
  - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$ — n^2 cases
    - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$. — $O(n \log n)$ steps

Pseudo-code revisited

Point: overhead needed to keep track of i, j, c_i, c_j.

Thus:

- While $GreenVar = yes$ do: — n loops
  - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$ — n^2 cases
    - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$. — $O(n \log n)$ steps

SUMMARY: on an input graph $G = (V, E)$ with $|V| = n$, our algorithm decides the answer to $PATH$ using:

Heuristics	$3n$ color changes
Pseudo-code	$O(n^3)$ operations
Turing machine	$O(n^4 \log n)$ steps (Time)

Pseudo-code revisited

Point: overhead needed to keep track of i, j, c_i, c_j.

Thus:

- While $GreenVar = yes$ do: $\quad$ n loops
  - For $i = 0$ to $n-1$; for $j = 0$ to $n-1$ $\quad$ n^2 cases
    - if $e_{i,j} = 1$ and $c_i = green$ and $c_j = white$ then set $c_j := red$. $\quad$ $O(n \log n)$ steps

SUMMARY: on an input graph $G = (V, E)$ with $|V| = n$, our algorithm decides the answer to $PATH$ using:

Heuristics	$3n$ color changes
Pseudo-code	$O(n^3)$ operations
Turing machine	$O(n^4 \log n)$ steps (Time) $O(n)$ memory cells (Space)

Turing machine complexity

Let $f : \mathbb{N} \to \mathbb{N}$ be given.

Definition

A decision problem D (with a specified encoding of its inputs) is:

1. in $TIME(f(N))$ if there exists a Turing machine solving D in at most $O(f(N))$ steps on inputs of length N.

Turing machine complexity

Let $f : \mathbb{N} \to \mathbb{N}$ be given.

Definition

A decision problem D (with a specified encoding of its inputs) is:

1. in $TIME(f(N))$ if there exists a Turing machine solving D in at most $O(f(N))$ steps on inputs of length N.
2. in $SPACE(f(N))$ if there exists a Turing machine solving D requiring at most $O(f(N))$ memory cells (not including the input tape) on inputs of length N.

Complexity of *PATH*

Recall that our Turing machine solves *PATH* on graphs with n vertices in

- Time: $O(n^4 \log n)$ steps
- Space: $O(n)$ memory cells.

Since "length N of input" $= n^2$ (when $n = |V|$), this at least proves

Complexity of *PATH*

Recall that our Turing machine solves *PATH* on graphs with n vertices in

- Time: $O(n^4 \log n)$ steps
- Space: $O(n)$ memory cells.

Since "length N of input" $= n^2$ (when $n = |V|$), this at least proves

Theorem

$$\begin{aligned} PATH &\in TIME(N^{2+\epsilon}) \\ PATH &\in SPACE(\sqrt{N}) \end{aligned}$$

Complexity of *PATH*

Recall that our Turing machine solves *PATH* on graphs with n vertices in

- Time: $O(n^4 \log n)$ steps
- Space: $O(n)$ memory cells.

Since "length N of input" $= n^2$ (when $n = |V|$), this at least proves

Theorem

$$\begin{aligned} PATH &\in TIME(N^{2+\epsilon}) \\ PATH &\in SPACE(\sqrt{N}) \end{aligned}$$

(Question: can we do better?...)

Another problem: Boolean Formula Value (*FVAL*)

INPUT:

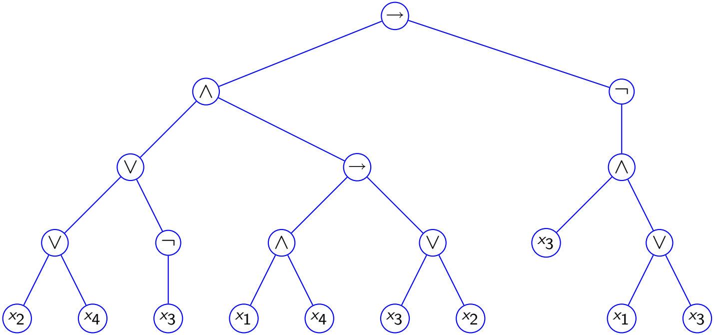
- A boolean formula φ in propositional variables $x_1, \ldots, x_n$.
- A sequence $\mathbf{c} = (c_1, \ldots, c_n) \in \{0, 1\}^n$.

QUESTION:

- Is $\varphi(\mathbf{c}) = 1$?

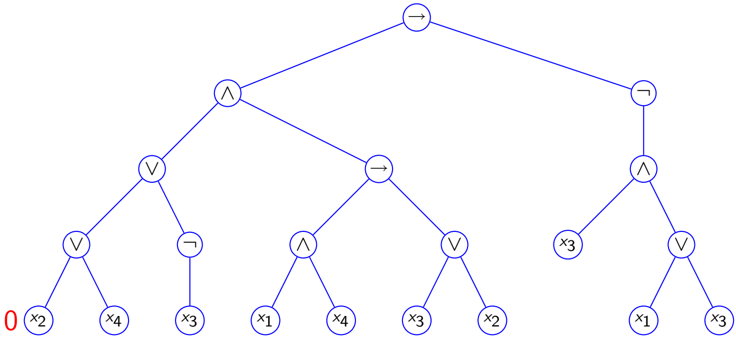
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



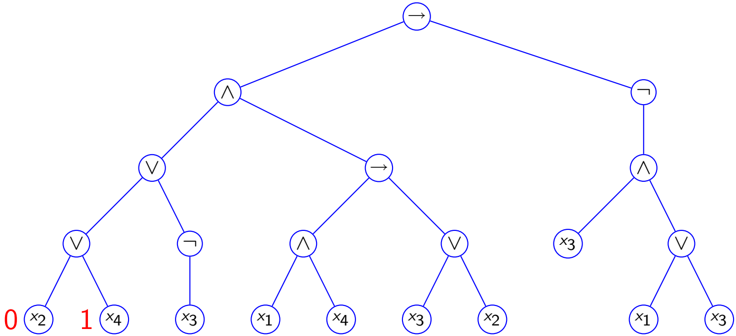
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



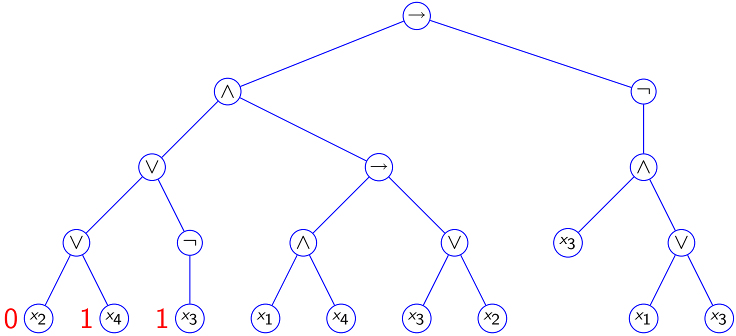
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



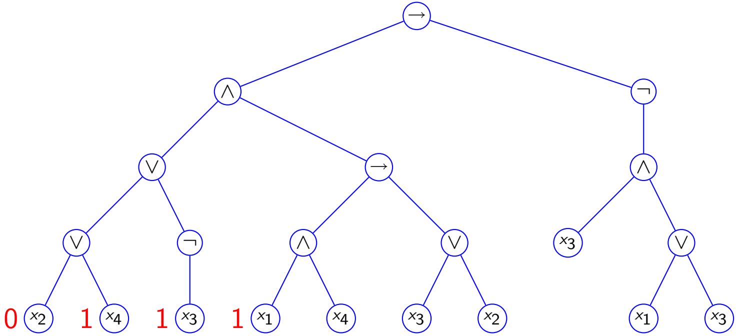
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1{,}0{,}1{,}1)$.



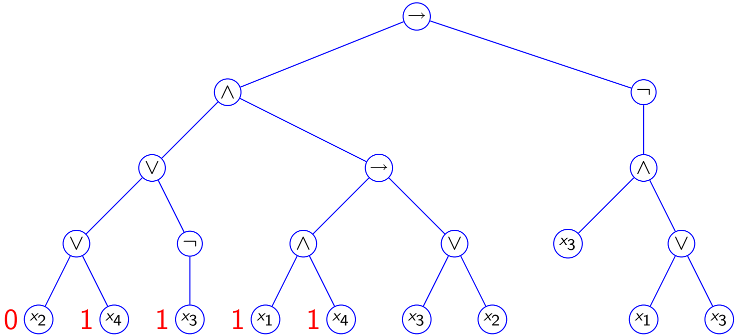
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



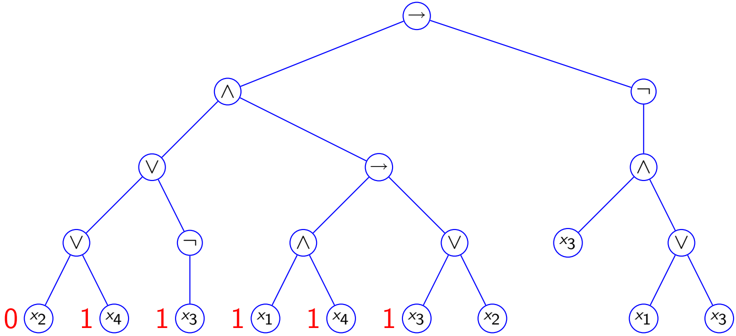
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



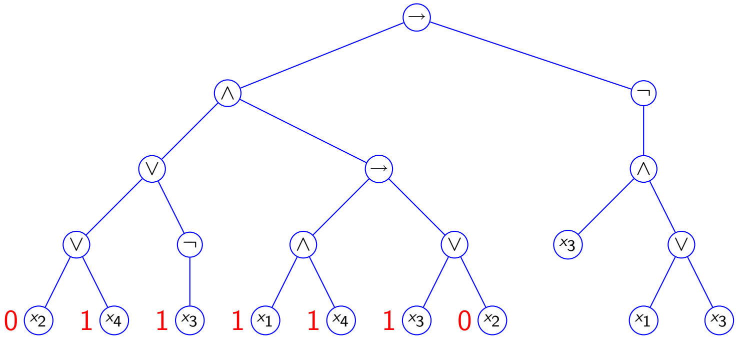
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1{,}0{,}1{,}1)$.



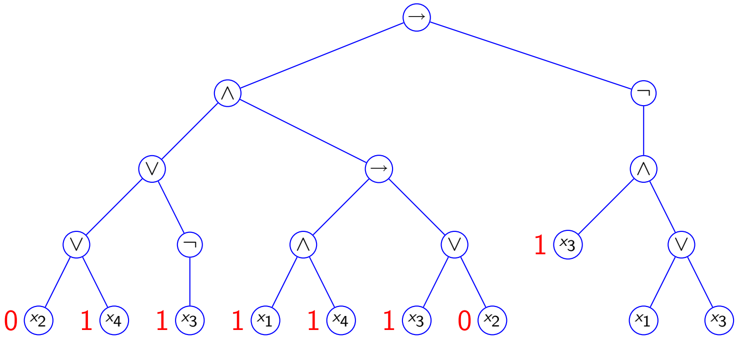
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1{,}0{,}1{,}1)$.



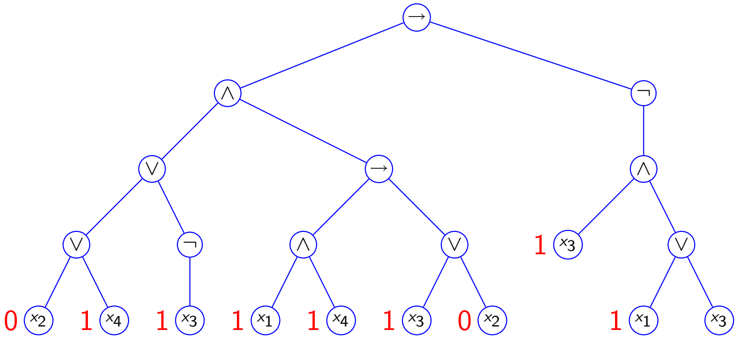
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



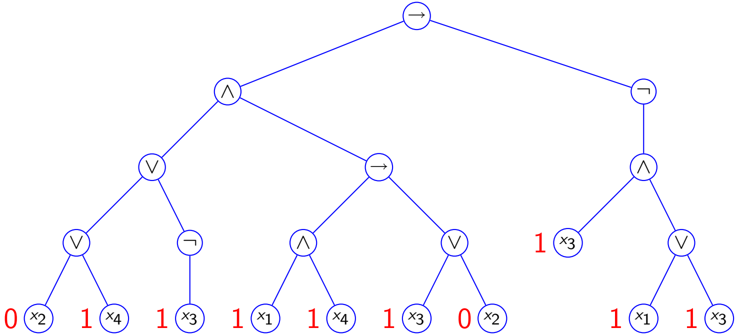
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



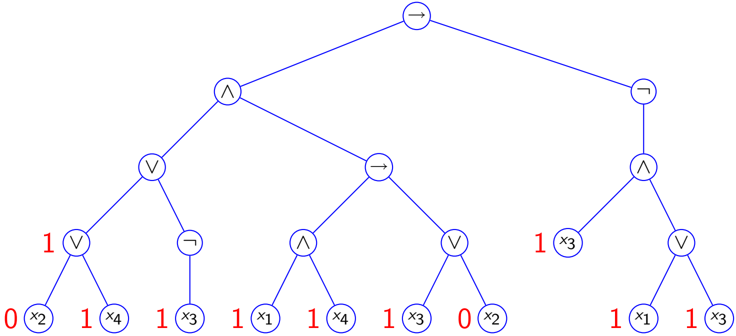
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



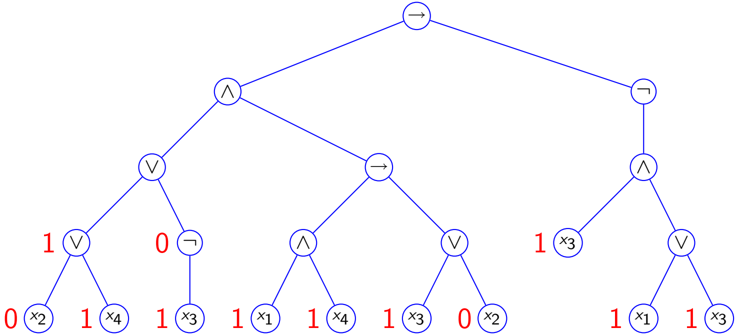
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



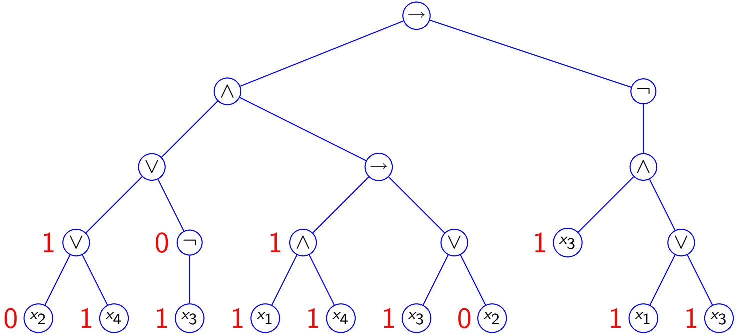
An algorithm for *FVAL*

$$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$$



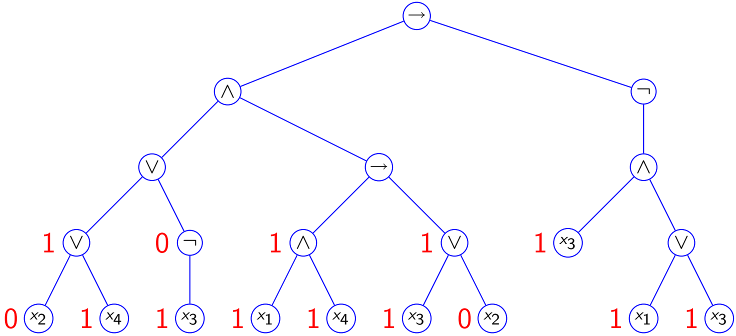
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



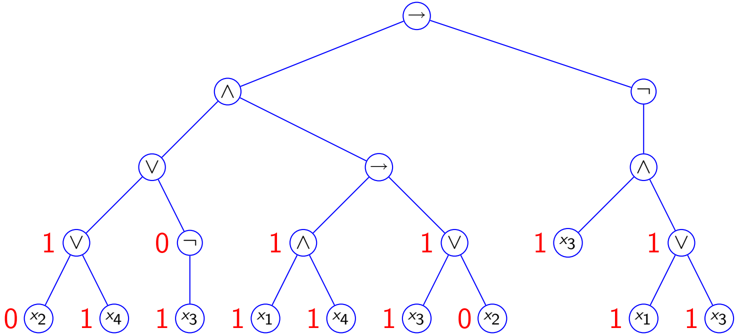
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



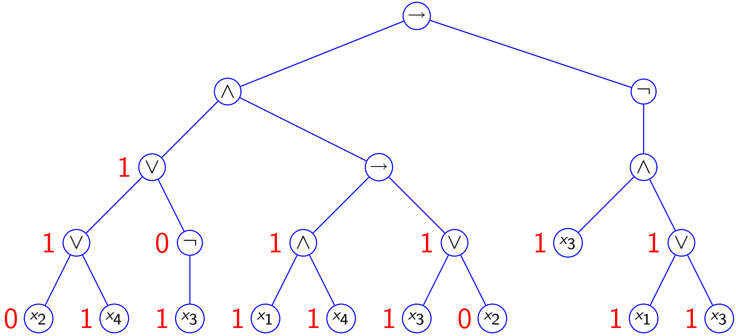
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



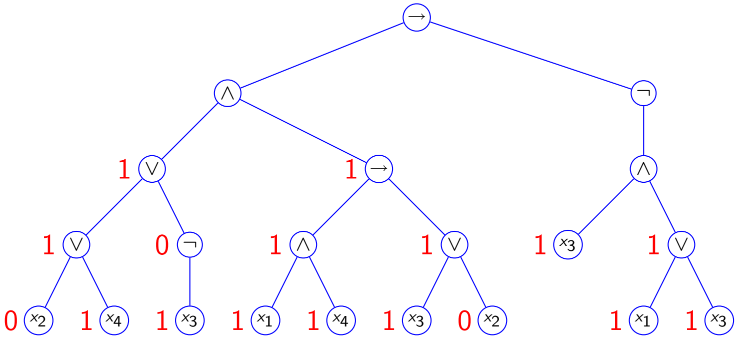
An algorithm for *FVAL*

$$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$$



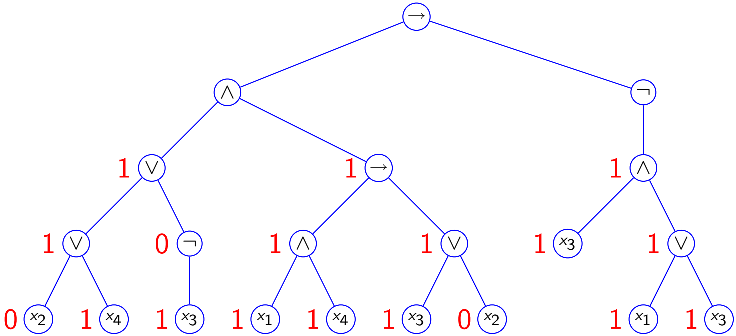
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1{,}0{,}1{,}1)$.



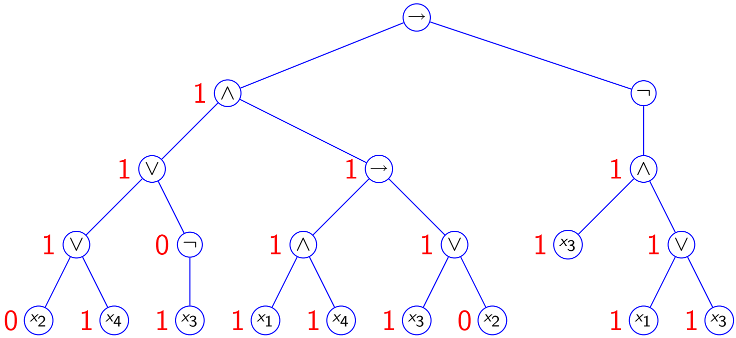
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



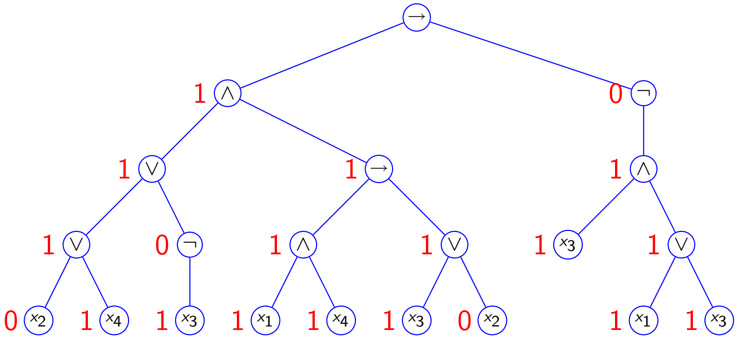
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1{,}0{,}1{,}1)$.



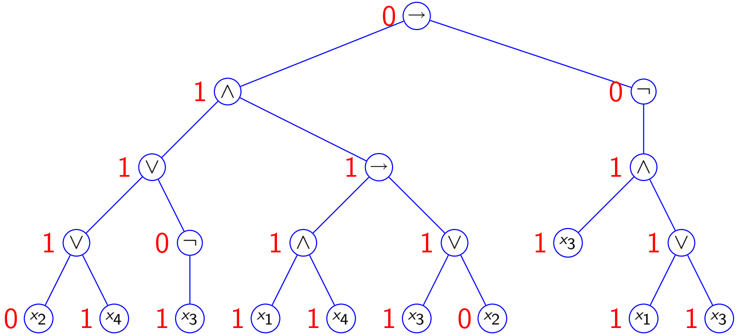
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



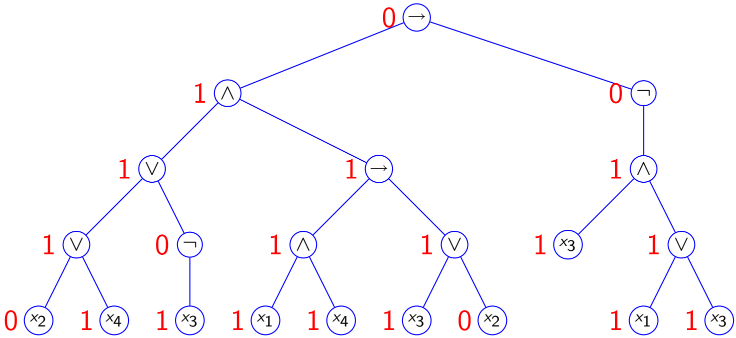
An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



An algorithm for *FVAL*

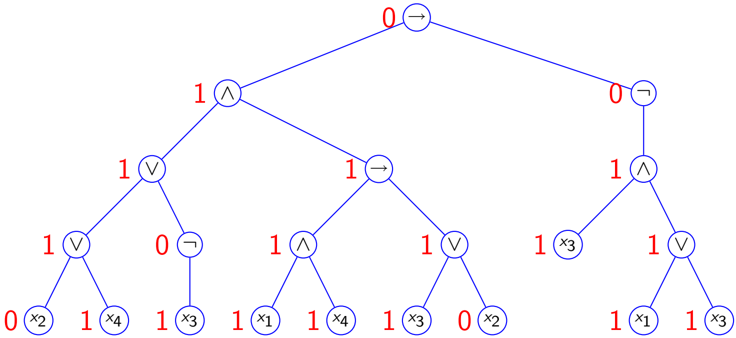
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1{,}0{,}1{,}1).$



Seems to use *TIME*(N) and *SPACE*(N).

An algorithm for *FVAL*

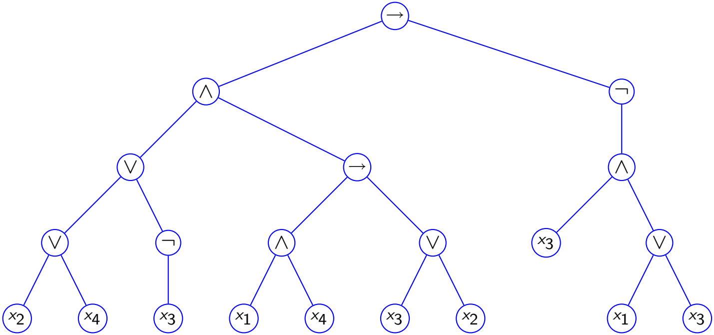
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



Seems to use $TIME(N)$ and $SPACE(N)$.
But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

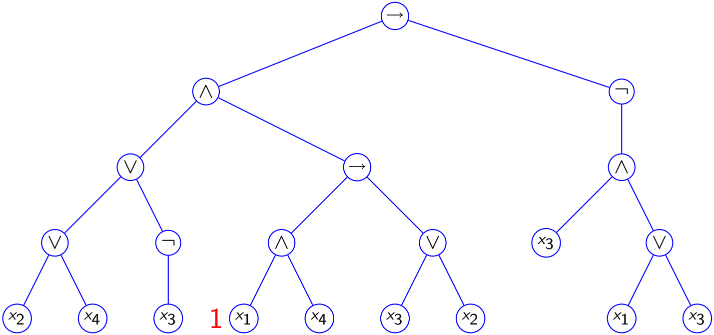
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

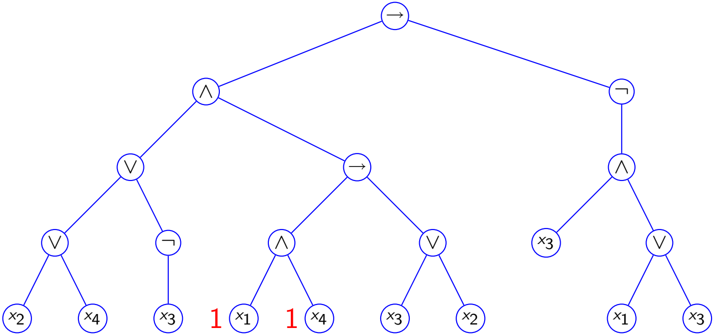
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

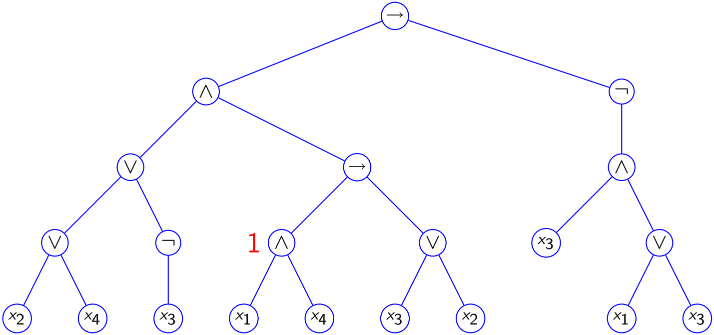
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3)))),\quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

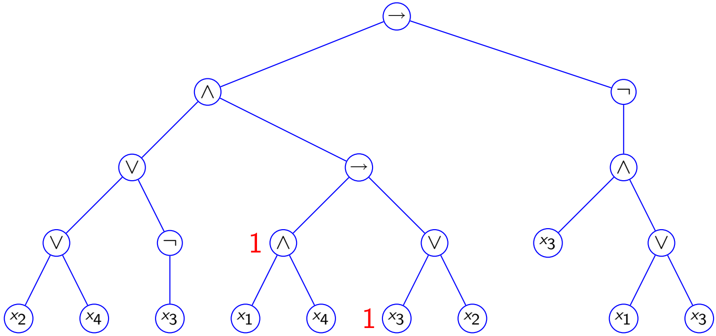
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

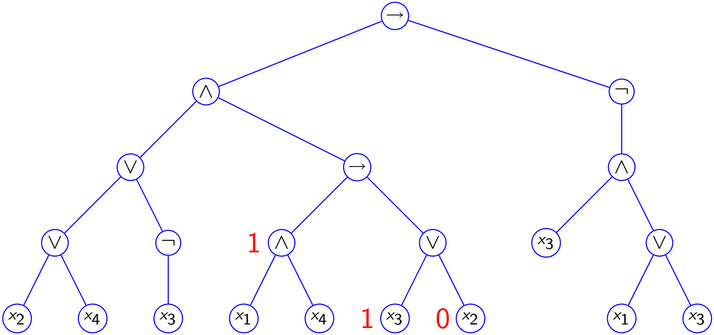
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

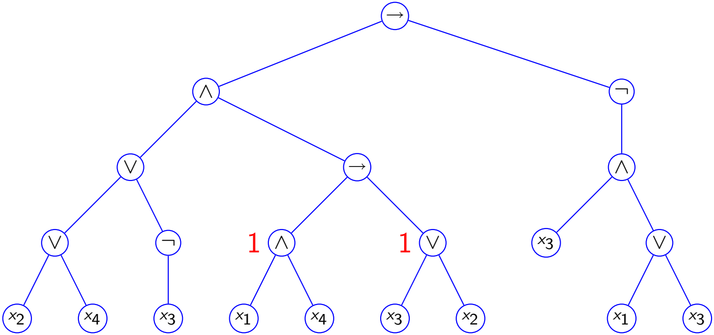
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

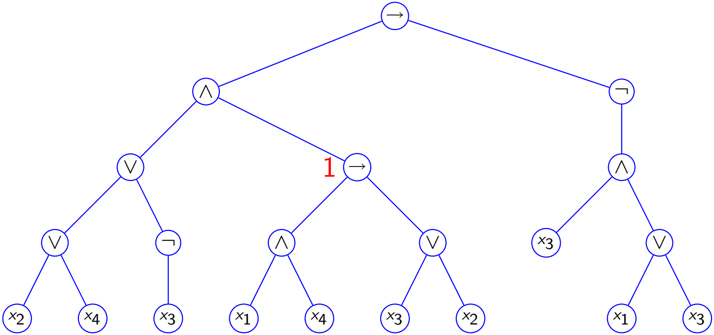
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

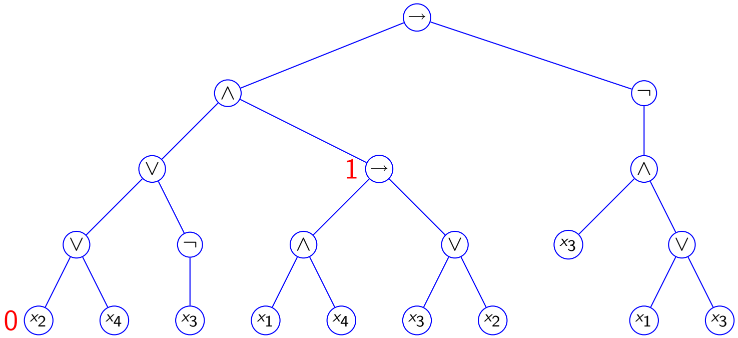
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

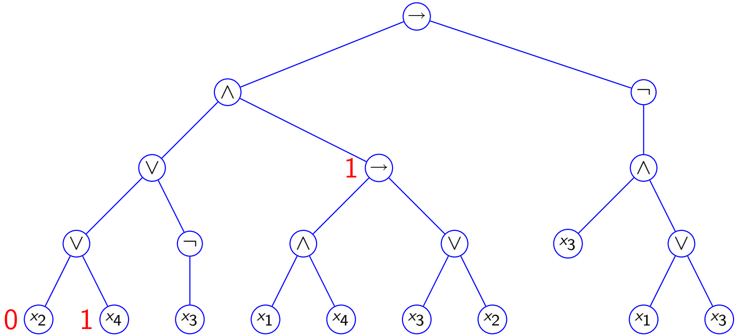
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

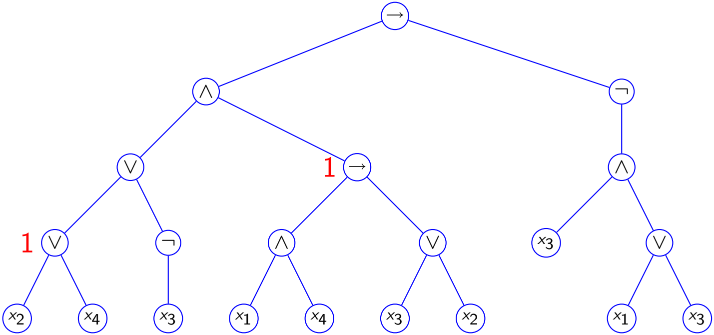
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

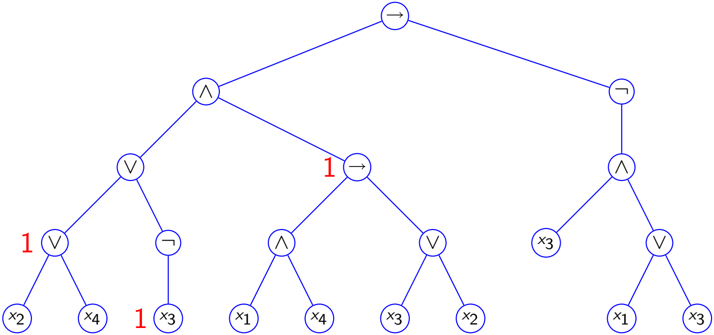
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

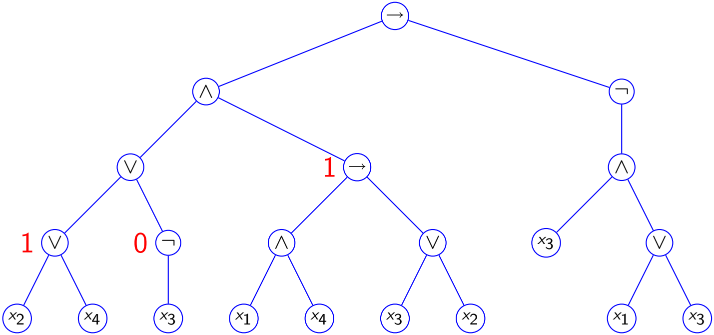
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

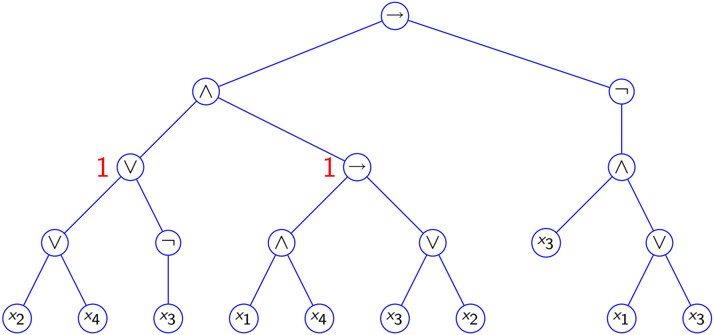
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1{,}0{,}1{,}1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

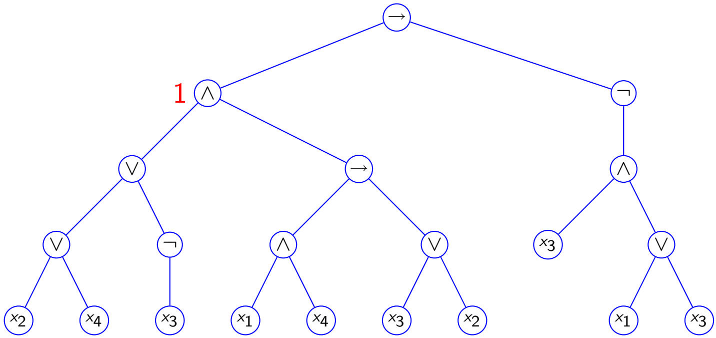
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1{,}0{,}1{,}1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

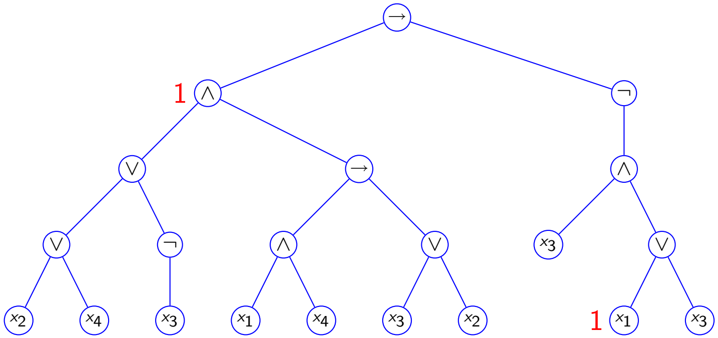
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3)))),\quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

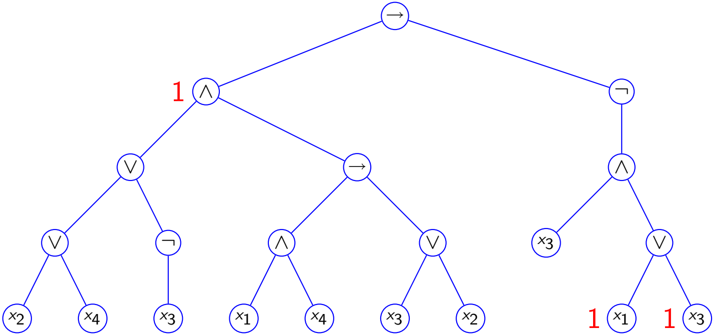
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \to (x_3 \vee x_2))) \to (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

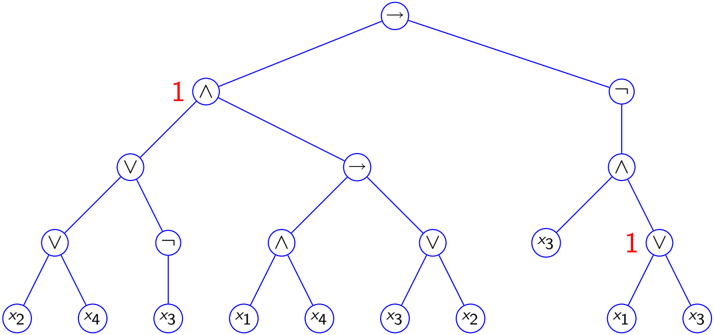
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

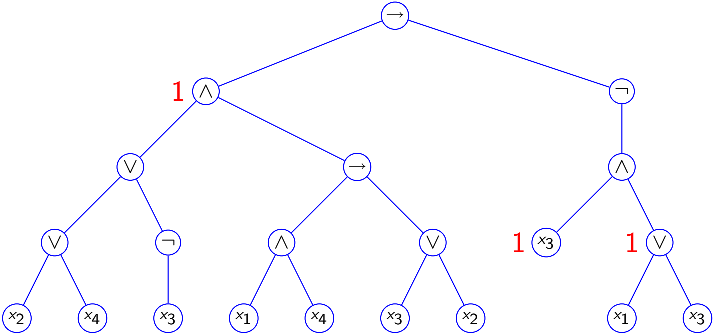
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

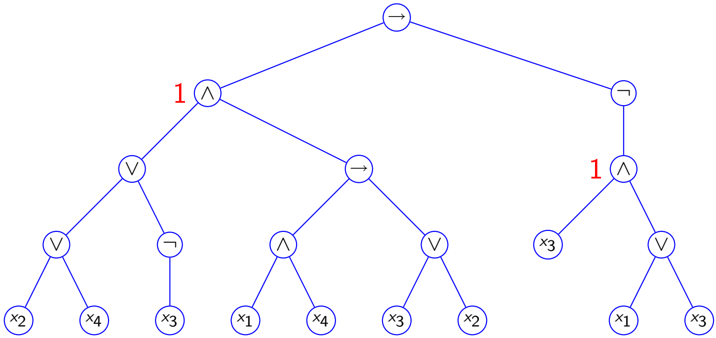
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1{,}0{,}1{,}1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

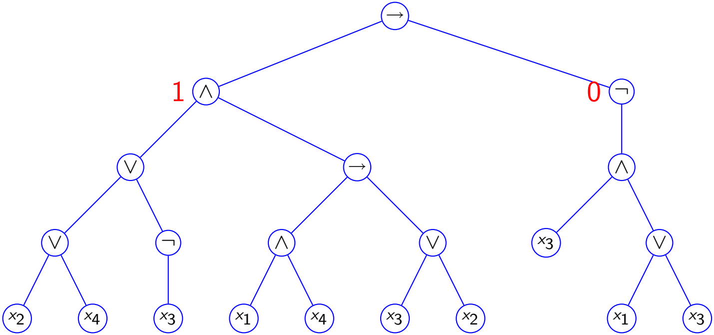
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3)))), \quad \mathbf{c} = (1,0,1,1).$



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

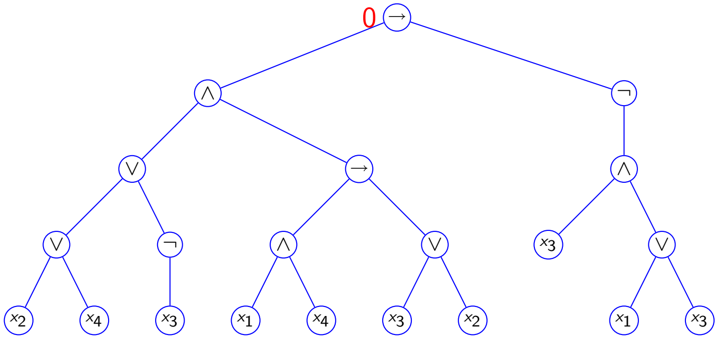
$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

An algorithm for *FVAL*

$\varphi = ((((x_2 \vee x_4) \vee (\neg(x_3))) \wedge ((x_1 \wedge x_4) \rightarrow (x_3 \vee x_2))) \rightarrow (\neg(x_3 \wedge (x_1 \vee x_3))))$, $\quad \mathbf{c} = (1,0,1,1)$.



But space can be re-used. In this example, 3 memory bits suffice.

Complexity of *FVAL*

In general, a bottom-up computation, always computing a larger subtree first, can be organized to need only $O(\log|\varphi|)$ intermediate values.

A careful implementation on a Turing machine yields:

Complexity of *FVAL*

In general, a bottom-up computation, always computing a larger subtree first, can be organized to need only $O(\log|\varphi|)$ intermediate values.

A careful implementation on a Turing machine yields:

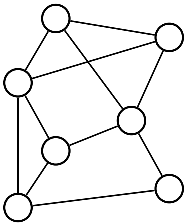
Theorem (Nancy Lynch, 1977)

$$\begin{aligned} FVAL &\in TIME(N^{2+\epsilon}) \\ FVAL &\in SPACE(\log N). \end{aligned}$$

A third problem: Graph 3-Colorability ($3COL$)

INPUT: a finite graph $G = (V, E)$.

QUESTION: Is it possible to color the vertices **red**, **green** or **blue**, so that no two adjacent vertices have the same color?

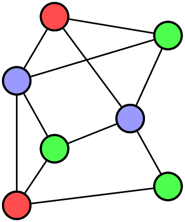


G

A third problem: Graph 3-Colorability ($3COL$)

INPUT: a finite graph $G = (V, E)$.

QUESTION: Is it possible to color the vertices **red**, **green** or **blue**, so that no two adjacent vertices have the same color?



G

Yes

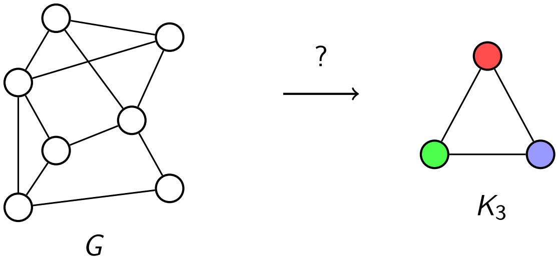
A third problem: Graph 3-Colorability ($3COL$)

INPUT: a finite graph $G = (V, E)$.

QUESTION: Is it possible to color the vertices **red**, **green** or **blue**, so that no two adjacent vertices have the same color?

Equivalently: does there exist a homomorphism

χ :



An algorithm for $3COL$

Brute force search algorithm:

- For each function $\chi : V \to K_3$:
    - Test if χ works.

An algorithm for $3COL$

Brute force search algorithm:

- For each function $\chi : V \to K_3$:
  - Test if χ works.

$3^{|V|} = 2^{O(\sqrt{N})}$ loops

An algorithm for $3COL$

Brute force search algorithm:

- For each function $\chi : V \to K_3$:
  - Test if χ works.

$3^{|V|} = 2^{O(\sqrt{N})}$ loops

$O(N^2)$ time,
$O(\sqrt{N})$ space

An algorithm for $3COL$

Brute force search algorithm:

- For each function $\chi : V \to K_3$:
  - Test if χ works.

$3^{|V|} = 2^{O(\sqrt{N})}$ loops

$O(N^2)$ time,
$O(\sqrt{N})$ space

Theorem

This at least proves:

$$\begin{aligned} 3COL &\in TIME(2^{O(\sqrt{N})}) \\ 3COL &\in SPACE(\sqrt{N}) \end{aligned}$$

An algorithm for $3COL$

Brute force search algorithm:

- For each function $\chi : V \to K_3$:
  - Test if χ works.

$3^{|V|} = 2^{O(\sqrt{N})}$ loops

$O(N^2)$ time,
$O(\sqrt{N})$ space

Theorem

This at least proves:

$$\begin{aligned} 3COL &\in TIME(2^{O(\sqrt{N})}) \\ 3COL &\in SPACE(\sqrt{N}) \end{aligned}$$

(Question: can we do better?...)

A fourth problem: Clone membership (CLO)

INPUT:

- A finite algebra $\mathbf{A}$.
- An operation $g : A^k \longrightarrow A$.

QUESTION: Is g a term operation of $\mathbf{A}$?

A fourth problem: Clone membership (*CLO*)

INPUT:

- A finite algebra $\mathbf{A}$.
- An operation $g : A^k \longrightarrow A$.

QUESTION: Is g a term operation of $\mathbf{A}$?

All known algorithms essentially generate the full k-generated free algebra in $\mathbf{V}(\mathbf{A})$,

$$\mathbf{F}_k \leq \mathbf{A}^{(A^k)}$$

and test whether $g \in F_k$.

A fourth problem: Clone membership (*CLO*)

INPUT:

- A finite algebra $\mathbf{A}$.
- An operation $g : A^k \to A$.

QUESTION: Is g a term operation of $\mathbf{A}$?

All known algorithms essentially generate the full k-generated free algebra in $\mathbf{V}(\mathbf{A})$,

$$\mathbf{F}_k \leq \mathbf{A}^{(A^k)}$$

and test whether $g \in F_k$.

In the worst case this could require as much as $O(|A^{(A^k)}|) = 2^{(|A|^k)^{1+\epsilon}}$ time and space.

I.e., exponential in the size of the input. (More on this in Lecture 3.)

Some important complexity classes

Definition

1. $P = PTIME = \bigcup_{k=1}^{\infty} TIME(N^k) = TIME(N^{O(1)})$.
2. $PSPACE = \bigcup_{k=1}^{\infty} SPACE(N^k) = SPACE(N^{O(1)})$.

Problems known to be in P are said to be *feasible* or *tractable*.

$$\begin{array}{ccccc} & P & \subseteq & PSPACE \\ & \cup\!\!\!\!\! & & \cup \\ & PATH & & 3COL \\ FVAL & & & \end{array}$$

Some important complexity classes

Definition

1. $P = PTIME = \bigcup_{k=1}^{\infty} TIME(N^k) = TIME(N^{O(1)})$.
2. $PSPACE = \bigcup_{k=1}^{\infty} SPACE(N^k) = SPACE(N^{O(1)})$.

Problems known to be in P are said to be *feasible* or *tractable*.

Definition

3. $EXPTIME = \bigcup_{k=1}^{\infty} TIME(2^{N^k}) = TIME(2^{N^{O(1)}})$.

$$\begin{array}{ccccccc} & P & \subseteq & PSPACE & \subseteq & EXPTIME \\ & \rotatebox[origin=c]{90}{$\in$} & & \rotatebox[origin=c]{90}{$\in$} & & \rotatebox[origin=c]{90}{$\in$} \\ & PATH & & 3COL & & CLO \\ FVAL & & & & & \end{array}$$

Some important complexity classes

Definition

1. $P = PTIME = \bigcup_{k=1}^{\infty} TIME(N^k) = TIME(N^{O(1)})$.
2. $PSPACE = \bigcup_{k=1}^{\infty} SPACE(N^k) = SPACE(N^{O(1)})$.

Problems known to be in P are said to be *feasible* or *tractable*.

Definition

3. $EXPTIME = \bigcup_{k=1}^{\infty} TIME(2^{N^k}) = TIME(2^{N^{O(1)}})$.
4. $L = LOGSPACE = SPACE(\log(N))$.

$$\begin{array}{ccccccc} L & \subseteq & P & \subseteq & PSPACE & \subseteq & EXPTIME \\ \Cup & & \Cup & & \Cup & & \Cup \\ FVAL & & PATH & & 3COL & & CLO \end{array}$$

Tomorrow

$$
\begin{array}{ccccccc}
L & \subseteq & P & \subseteq & PSPACE & \subseteq & EXPTIME \\
\Cup & & \Cup & & \Cup & & \Cup \\
FVAL & & PATH & & 3COL & & CLO
\end{array}
$$

In tomorrow's lecture I will:

- Introduce "nondeterministic" versions of these 4 classes.
- Introduce problems which are "hardest" for each class.