

Complexité CM6

Antonio E. Porreca

aeporreca.org/complexite

Machines de Turing non déterministes (MTND)

Non-déterminisme

- Comme dans les automates finis déterministes vs non déterministes

- Au lieu d'avoir une fonction de transition déterministe...

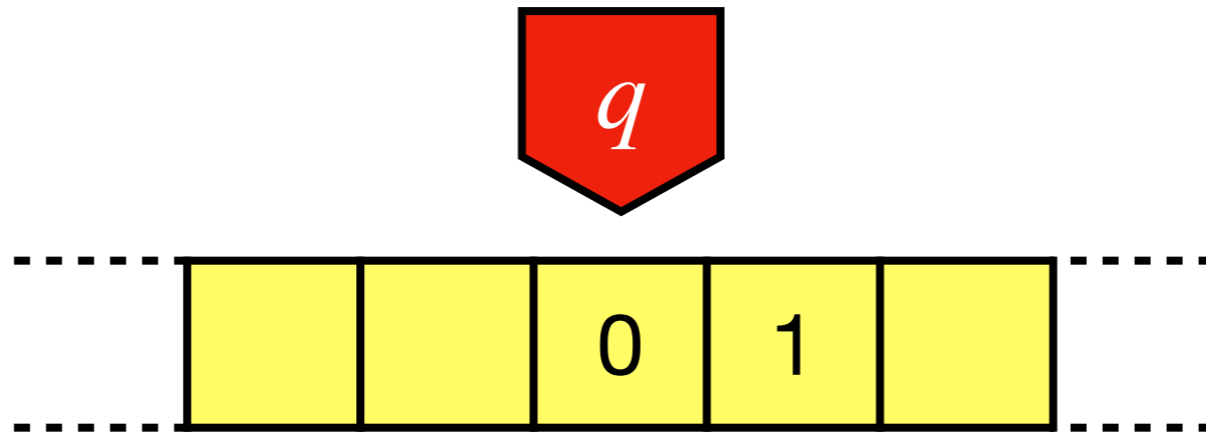
$$\delta: (Q \setminus \{q_{\text{oui}}, q_{\text{no}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$$

- ...on admet **plusieurs configurations suivantes**

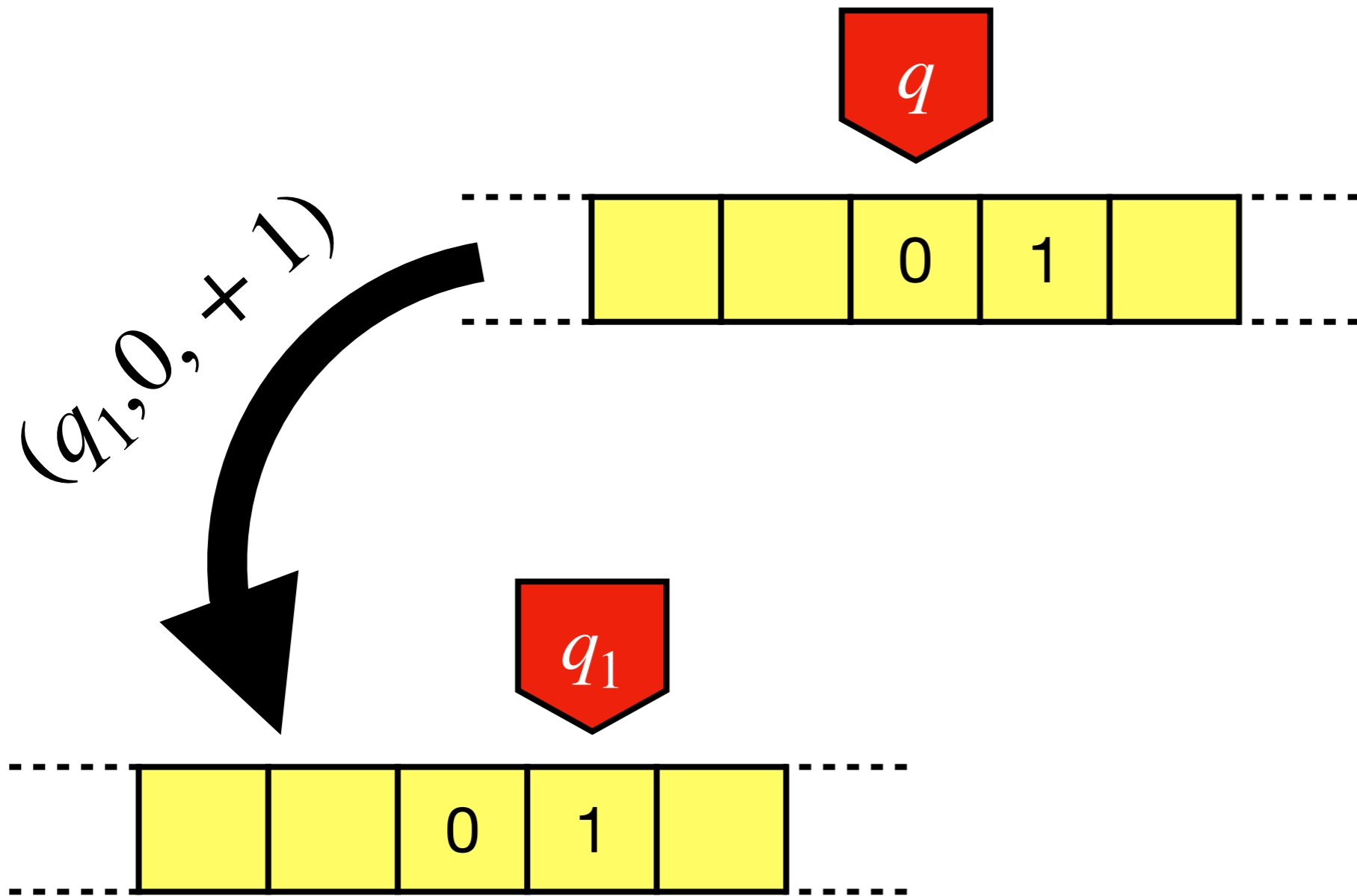
$$\delta: (Q \setminus \{q_{\text{oui}}, q_{\text{non}}\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{-1, 0, +1\})$$

- Il y a un maximum de $|Q| \times |\Gamma| \times 3$ transitions possibles, qui ne dépend pas de la taille n de l'entrée

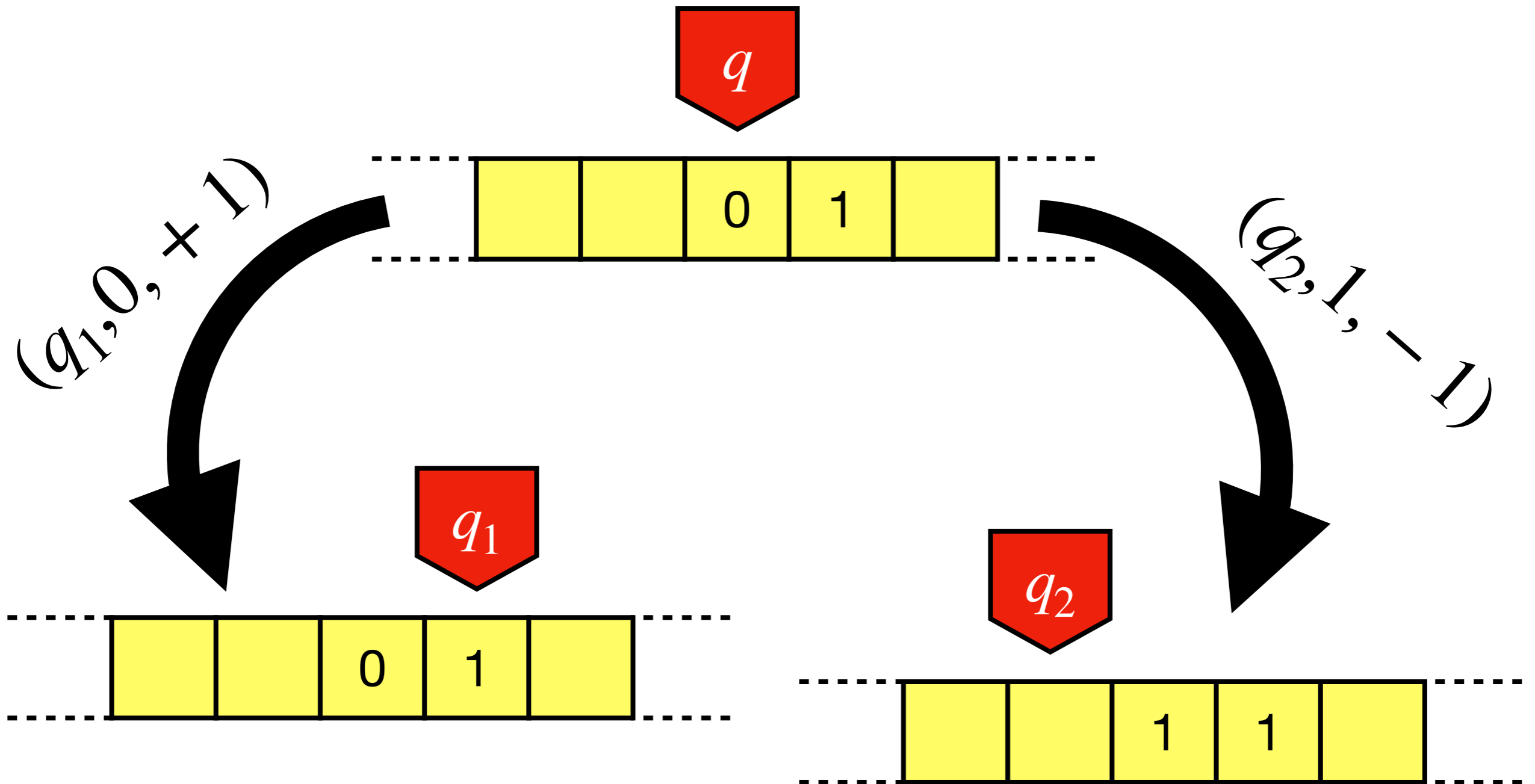
$$\text{Transition } \delta(q,0) = \left\{ \begin{array}{l} (q_1, 0, +1) \\ (q_2, 1, -1) \end{array} \right\}$$



$$\text{Transition } \delta(q,0) = \left\{ \begin{array}{l} (q_1, 0, +1) \\ (q_2, 1, -1) \end{array} \right\}$$

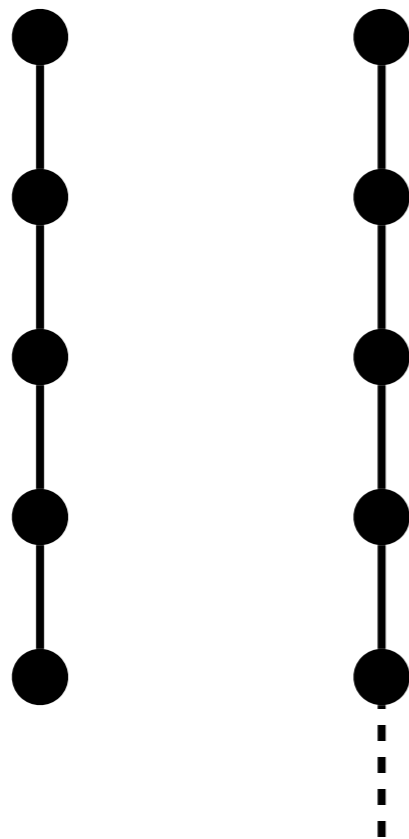


$$\text{Transition } \delta(q,0) = \left\{ \begin{array}{l} (q_1, 0, +1) \\ (q_2, 1, -1) \end{array} \right\}$$

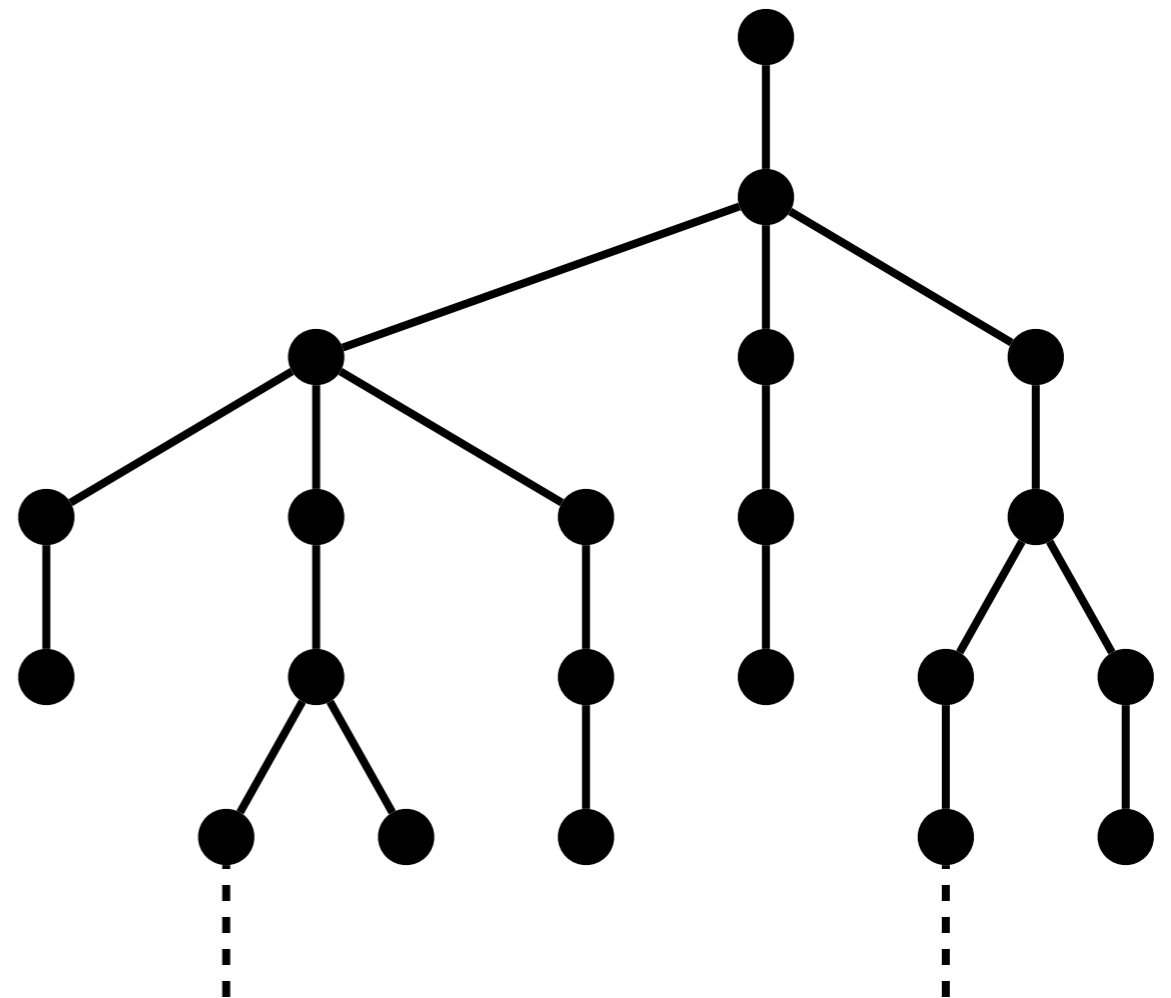


Arbres de calcul

Machines déterministes



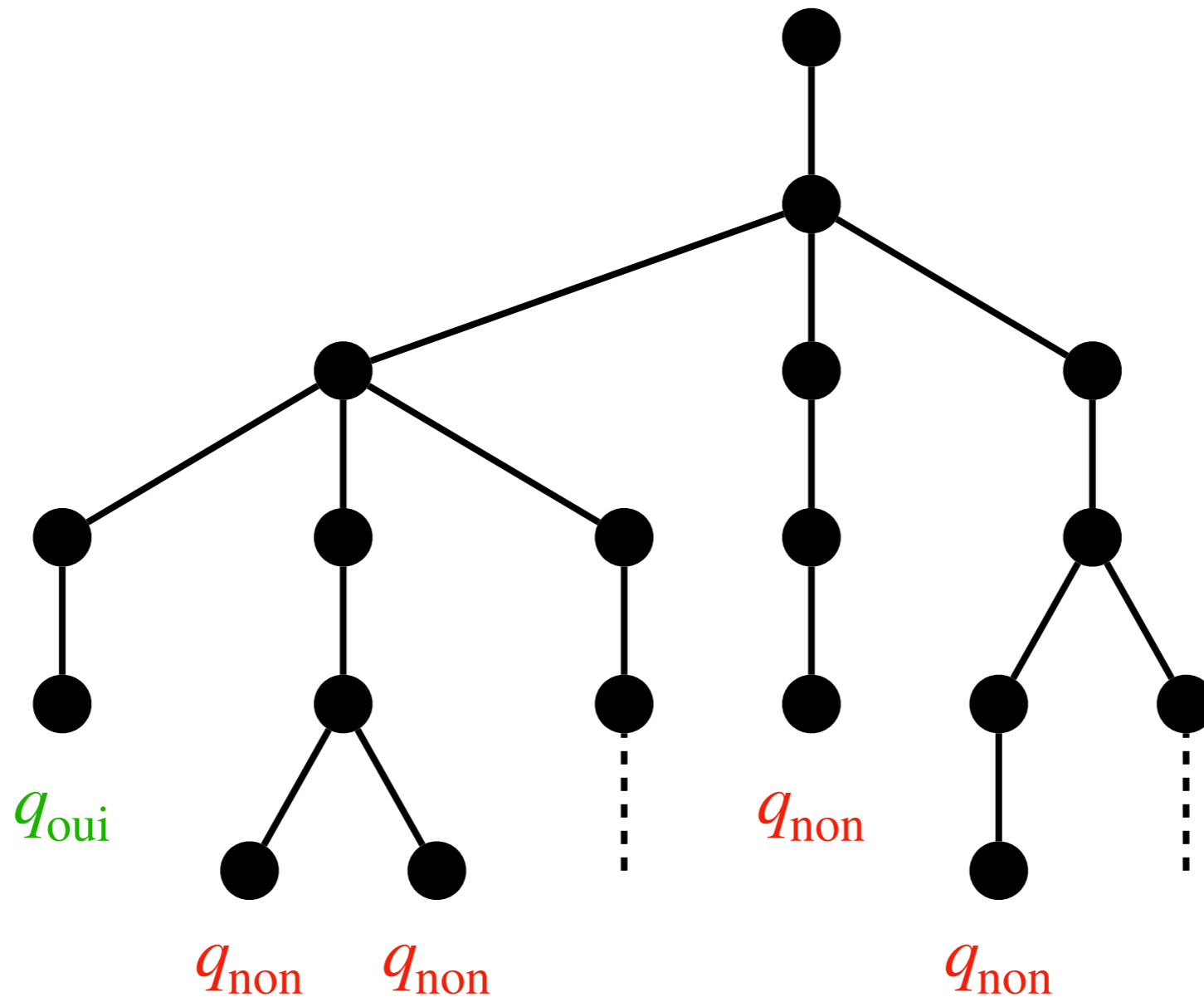
Machine non déterministe



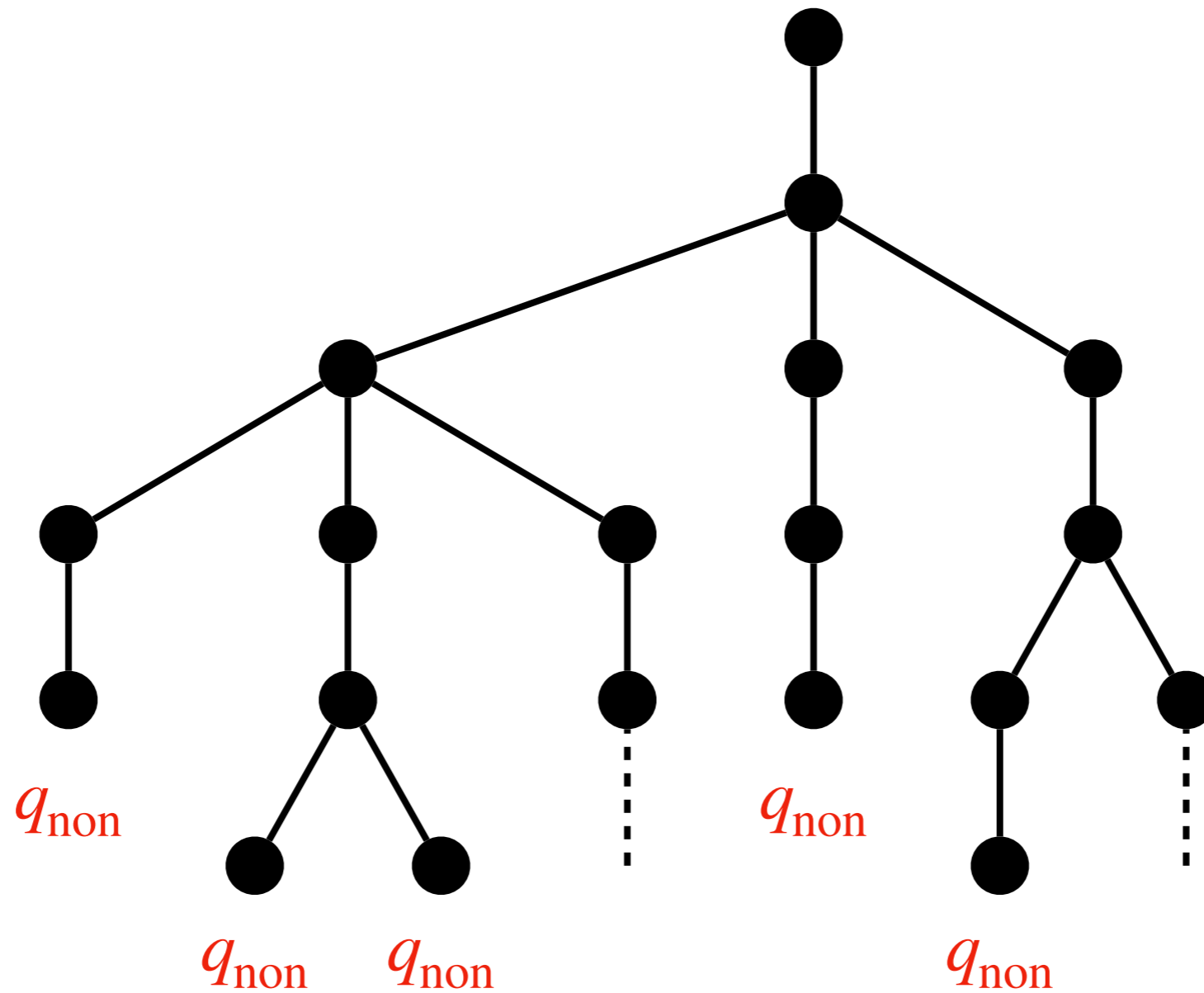
Langage reconnu par une machine de Turing non déterministe N

- $L(N) = \text{Accepte}(N) = \left\{ x \in \Sigma^* : \text{il existe un calcul de } N \text{ sur } x \text{ qui se termine par } q_{\text{oui}} \right\}$
- $\text{Rejet}(N) = \left\{ x \in \Sigma^* : \text{au moins un calcul de } N \text{ sur } x \text{ rejette et aucun calcul n'accepte} \right\}$
- $\text{Boucle}(N) = \left\{ x \in \Sigma^* : \text{aucun calcul de } N \text{ sur } x \text{ ne s'arrête} \right\}$

N accepte x



N n'accepte pas x



Résolution de problèmes par une machine de Turing non déterministe

On dit qu'un programme N pour MTND **résout un problème de décision π** sous un système de codage S si, pour chaque entrée $m \in \Sigma^*$, **tous les calculs** de N sur m s'arrêtent et si N reconnaît le langage associé au problème, c-à-d si $L(N) = L(\pi, S)$

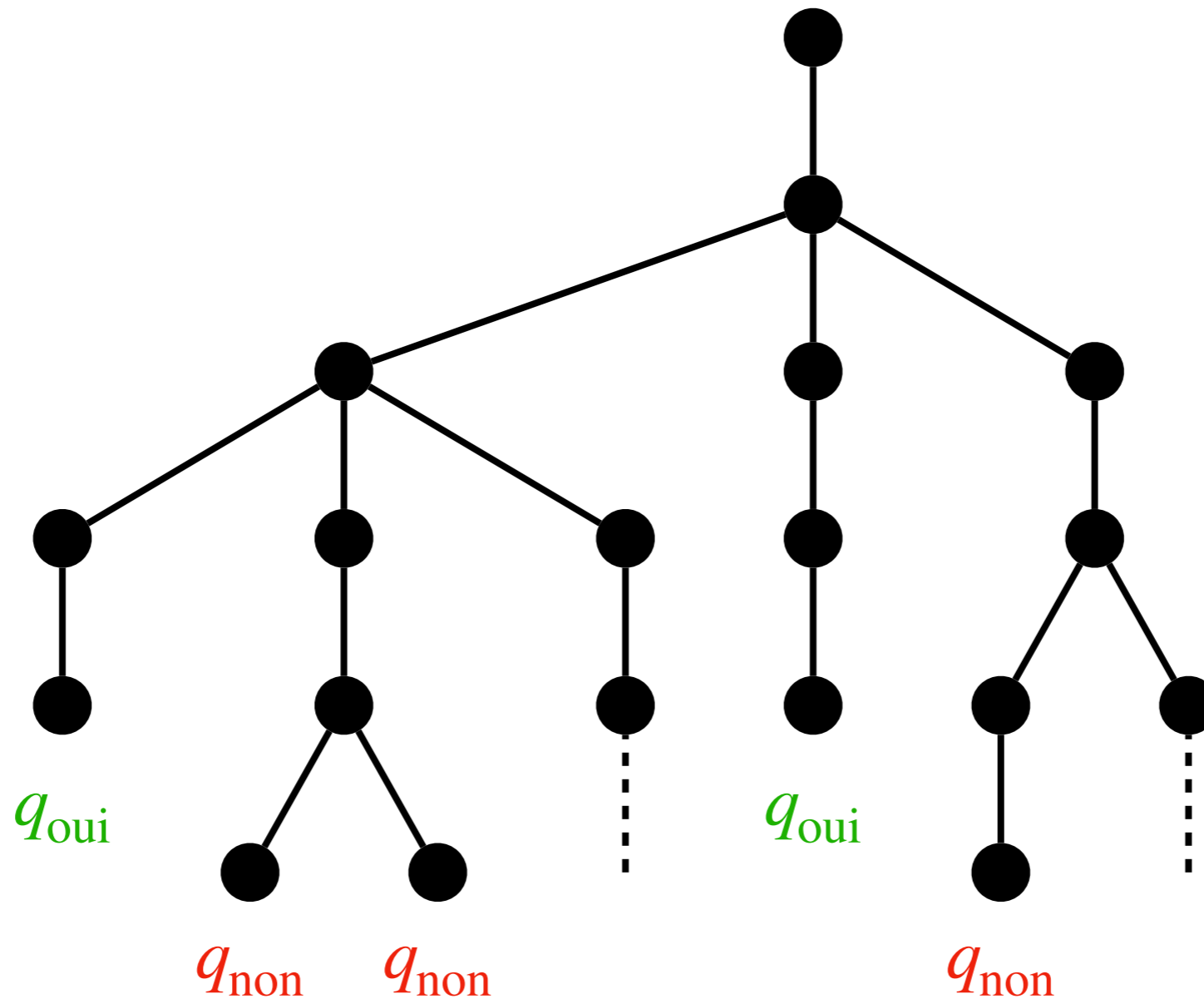
Puissance des machines de Turing non déterministes

- Les machines non déterministes sont **plus générales** que les déterministes
- Chaque machine déterministe est un cas particulier de machine non déterministe, avec $|\delta(q, a)| = 1$ pour chaque (q, a)
- Les machines non déterministes ont l'air plus puissant, mais en réalité...

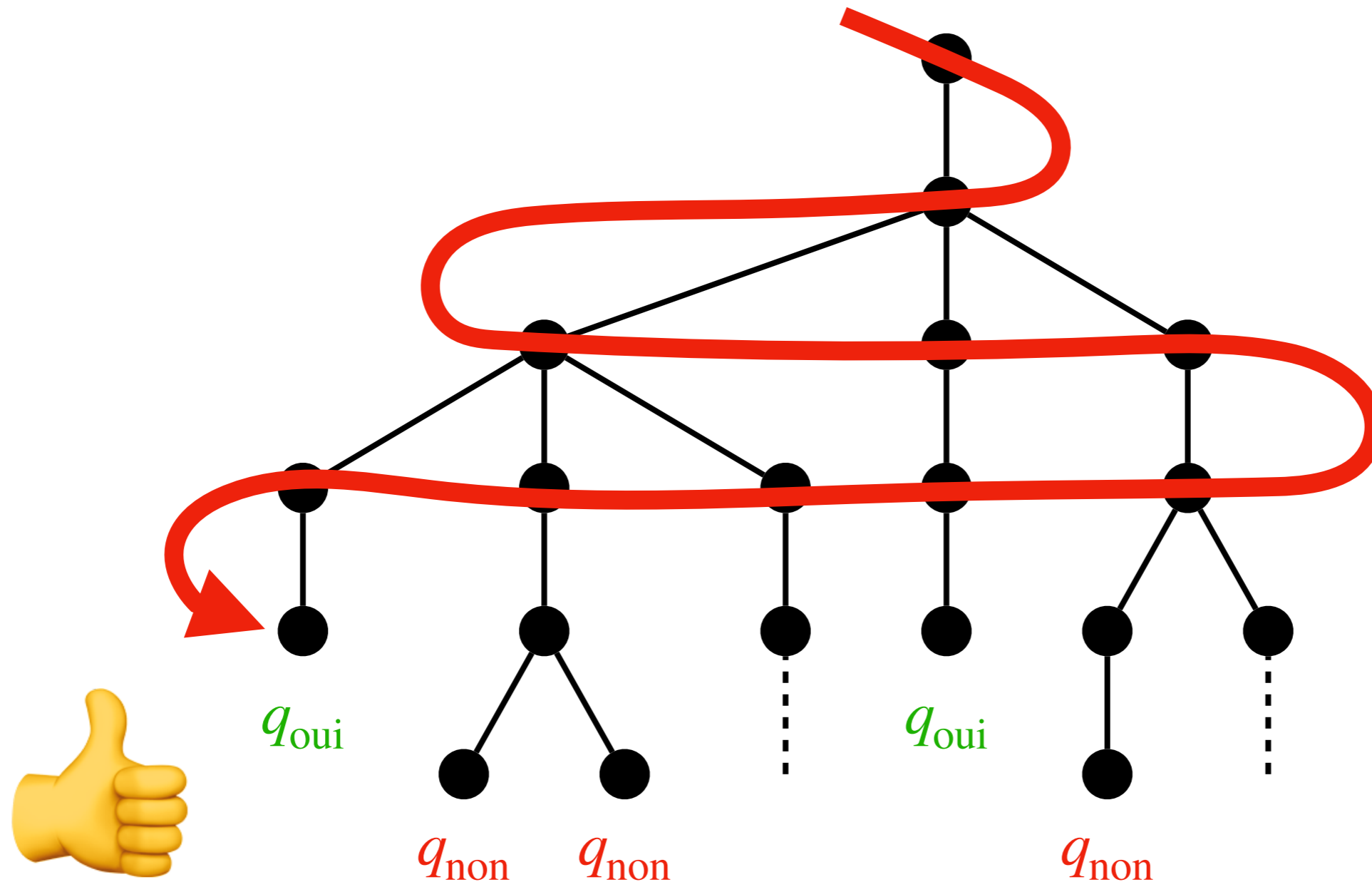
Equivalence des machines déterministes et non déterministes

- Un langage L est reconnu par une machine **non déterministe** ssi il est reconnu par une machine **déterministe**
- On a vu que chaque machine déterministe est un type de machine non déterministe
- Vice-versa, on peut simuler de façon déterministe une machine non déterministe en **parcourant en largeur** son arbre de calcul

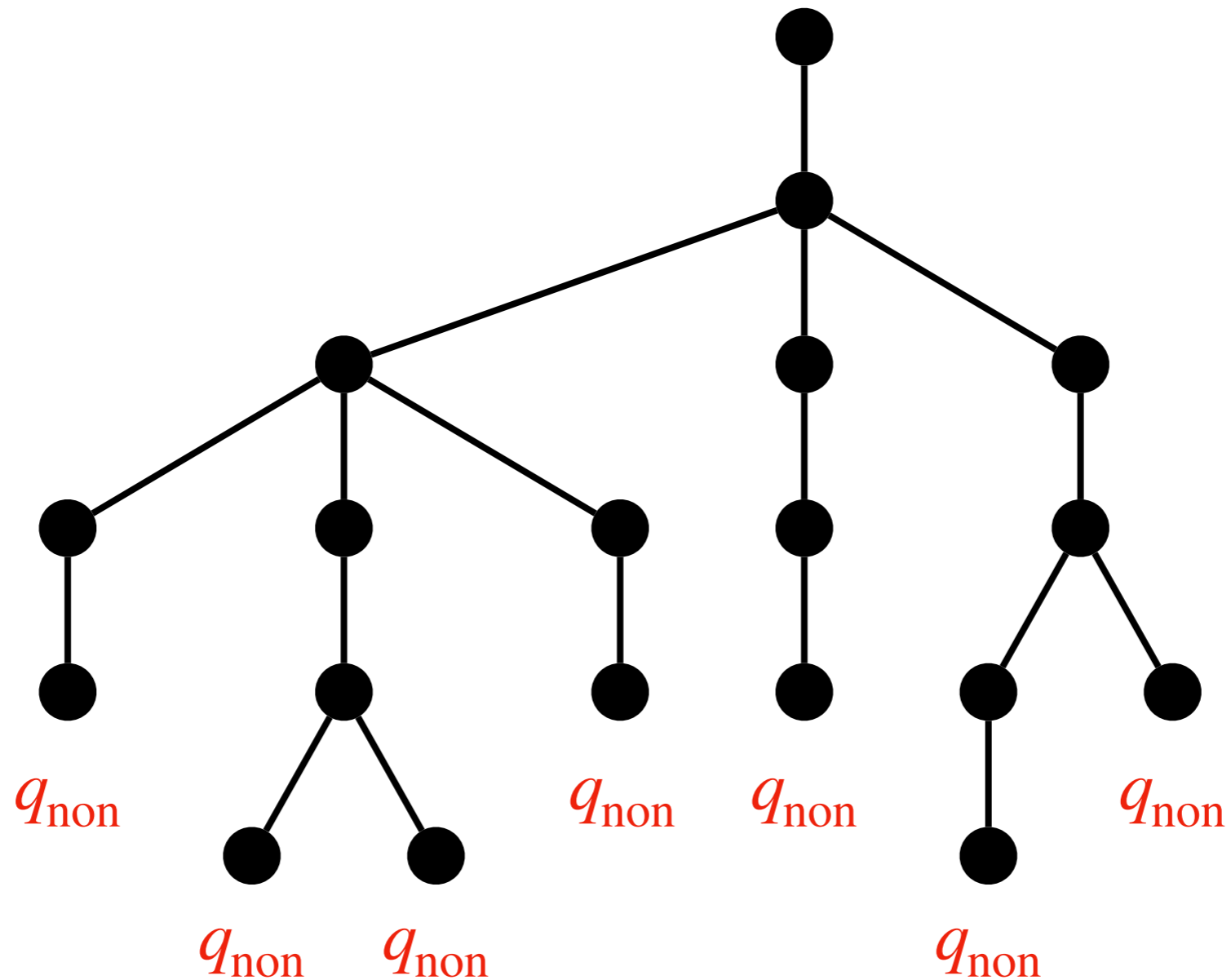
On s'arrête en acceptant si on trouve un calcul qui se termine par q_{oui}



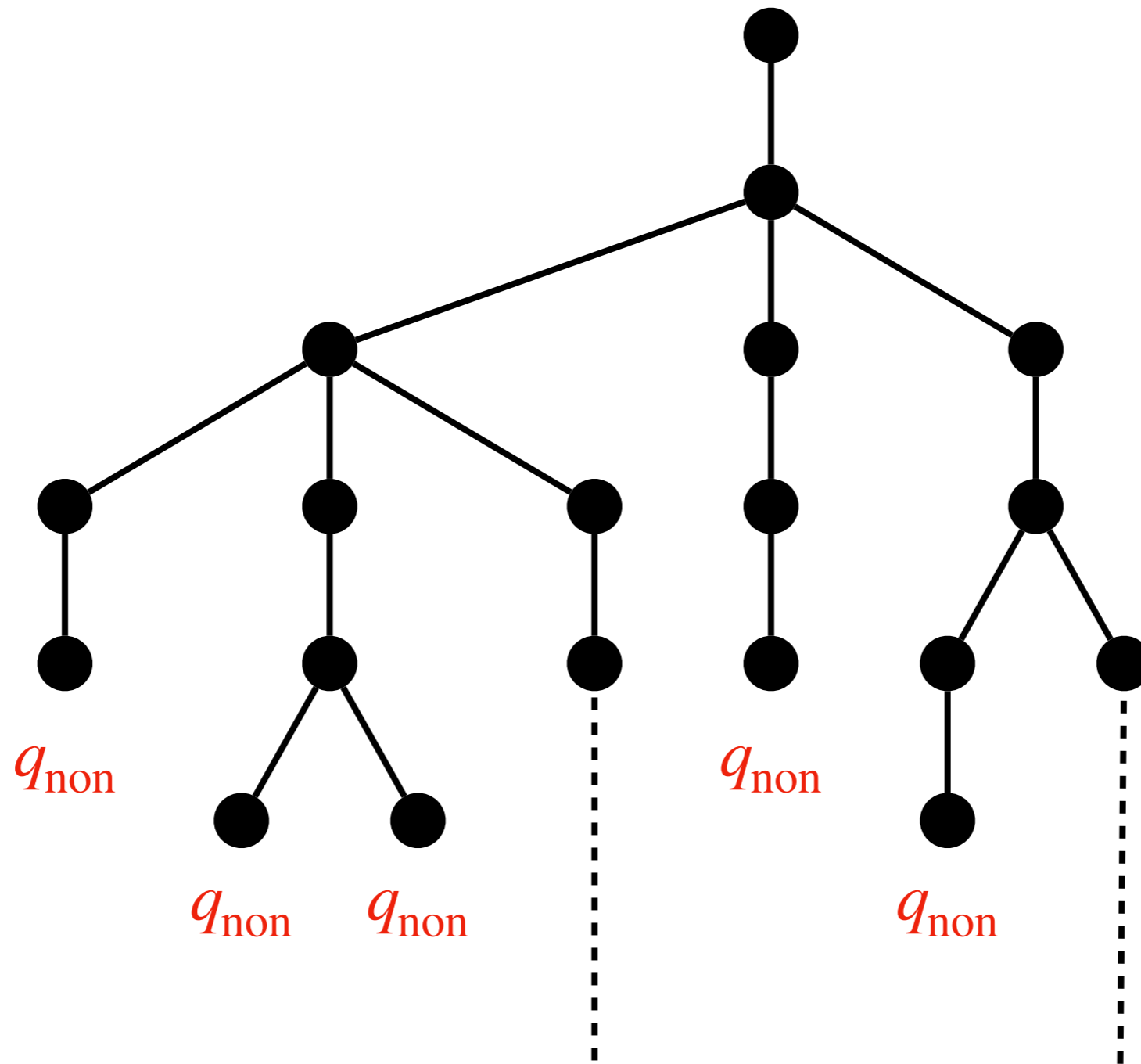
On s'arrête en acceptant si on trouve un calcul qui se termine par q_{oui}



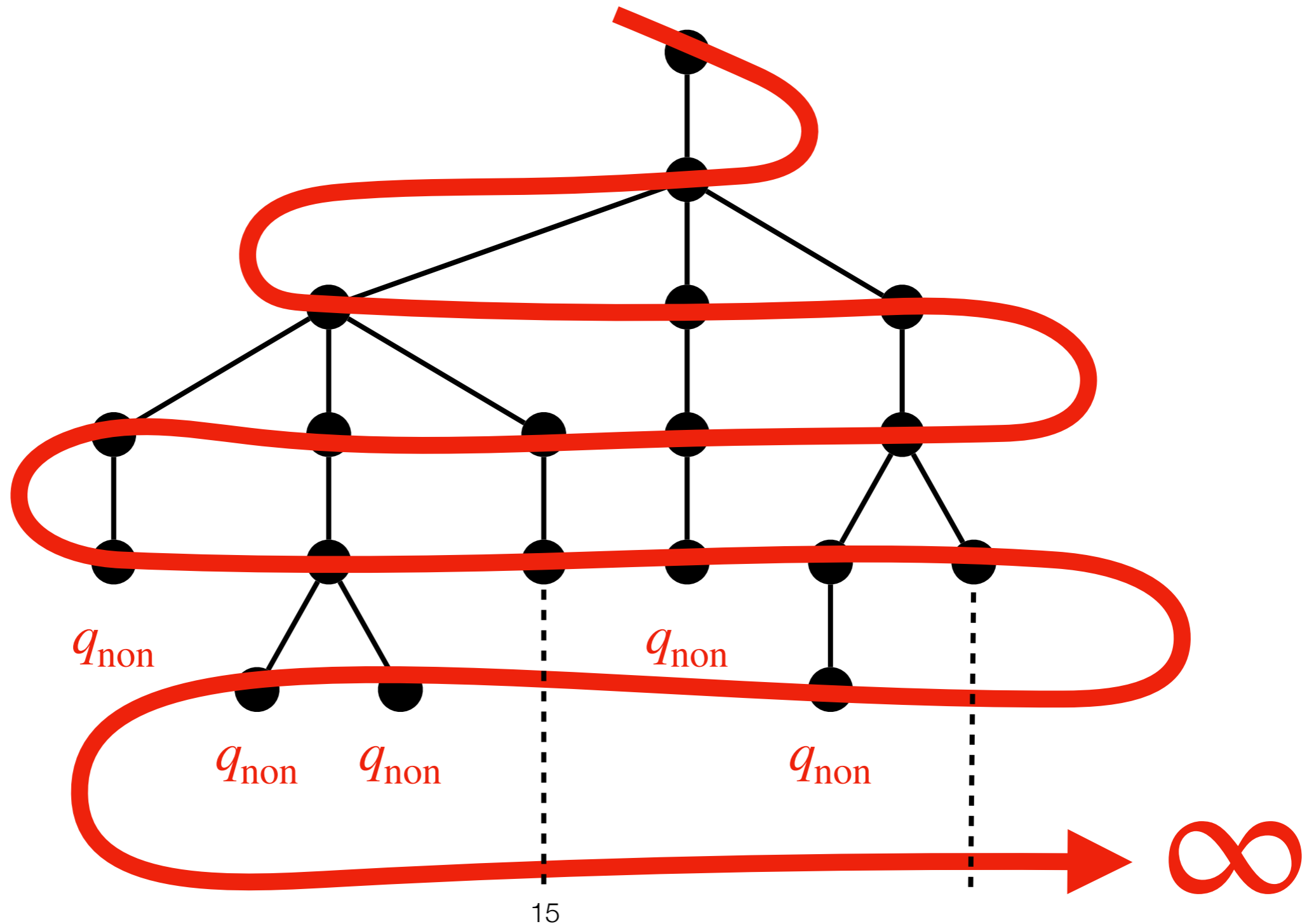
On s'arrête en rejetant si tous
les calculs se terminent par q_{no}



Sinon, on ne s'arrête pas



Sinon, on ne s'arrête pas



Algorithmes non déterministes

Pseudo-code non déterministe

- On exprime un choix non déterministe en pseudo-code avec une fonction « **devine** », qui choisit parmi un ensemble fini de valeurs de taille **constante** :

$$x := \text{devine}(\text{choix}_1, \text{choix}_2, \dots, \text{choix}_k)$$

- On peut toujours se reconduire à une divination **binaire**, si besoin est :

$$x := \text{devine}(0,1)$$

Divination d'un element x d'un ensemble X de taille non constante

$m := |X|$

$i := 0$

tant que $i \leq m - 2$ **et** $\text{devine}(0,1) = 0$ **faire**

$i := i + 1$

$x := X[i]$

Divination d'un element x d'un ensemble X de taille non constante

$m := |X|$

$i := 0$

tant que $i \leq m - 2$ **et** $\text{devine}(0,1) = 0$ **faire**

$i := i + 1$

$x := X[i]$



$x := \text{devine}(X)$

Un exemple : cycle hamiltonien

```
fonction hamiltonien( $V, E$ )  
   $n := |V|$   
   $perm := \text{tableau}(n)$   
  pour  $i := 0$  à  $n - 1$  faire  
     $perm[i] := \text{devine}(0, \dots, n - 1)$   
  pour chaque  $v \in V$  faire  
    si  $perm$  ne contient pas  $v$  exactement 1 fois alors  
      rejeter  
  pour  $i := 0$  à  $n - 1$  faire  
    si  $(perm[i], perm[(i + 1) \bmod n]) \notin E$  alors  
      rejeter  
  accepter  
fin
```

Un exemple : cycle hamiltonien

fonction hamiltonien(V, E)

$n := |V|$

$perm := \text{tableau}(n)$

pour $i := 0$ à $n - 1$ **faire**

$perm[i] := \text{devine}(0, \dots, n - 1)$

pour chaque $v \in V$ **faire**

si $perm$ ne contient pas v exactement 1 fois **alors**

rejeter

pour $i := 0$ à $n - 1$ **faire**

si $(perm[i], perm[(i + 1) \bmod n]) \notin E$ **alors**

rejeter

accepter

fin

perm est-elle une
permutation ?

Un exemple : cycle hamiltonien

fonction hamiltonien(V, E)

$n := |V|$

$perm := \text{tableau}(n)$

pour $i := 0$ à $n - 1$ **faire**

$perm[i] := \text{devine}(0, \dots, n - 1)$

pour chaque $v \in V$ **faire**

si $perm$ ne contient pas v exactement 1 fois **alors**
rejeter

pour $i := 0$ à $n - 1$ **faire**

si $(perm[i], perm[(i + 1) \bmod n]) \notin E$ **alors**
rejeter

accepter

fin

perm est-elle une
permutation ?

perm est-il un cycle
dans le graphe ?

Un exemple : cycle hamiltonien

fonction hamiltonien(V, E)

$n := |V|$

$perm := \text{tableau}(n)$

pour $i := 0$ à $n - 1$ **faire**

$perm[i] := \text{devine}(0, \dots, n - 1)$

pour chaque $v \in V$ **faire**

si $perm$ ne contient pas v exactement 1 fois **alors**
rejeter

pour $i := 0$ à $n - 1$ **faire**

si $(perm[i], perm[(i + 1) \bmod n]) \notin E$ **alors**
rejeter

accepter

fin

$O(n \log n)$ bits devinés

perm est-elle une permutation ?

perm est-il un cycle dans le graphe ?

Simulation du non déterminisme dans le monde réel*

```
from nondeterminism import *

@nondeterministic
def hamiltonian(vertices, edges):
    n = len(vertices)
    perm = []
    for i in range(n):
        v = guess(vertices)
        perm.append(v)
    for v in vertices:
        if perm.count(v) != 1:
            reject()
    for i in range(n):
        if (perm[i], perm[(i+1)%n]) not in edges:
            reject()
    accept()
```

Simulation du non déterminisme dans le monde réel*

```
from nondeterminism import *
```

```
@nondeterministic
```

```
def hamiltonian(vertices, edges):
```

```
    n = len(vertices)
```

```
    perm = []
```

```
    for i in range(n):
```

```
        v = guess()
```

```
        perm.append(v)
```

```
    for v in vertices:
```

```
        if perm.count(v) > 1:
```

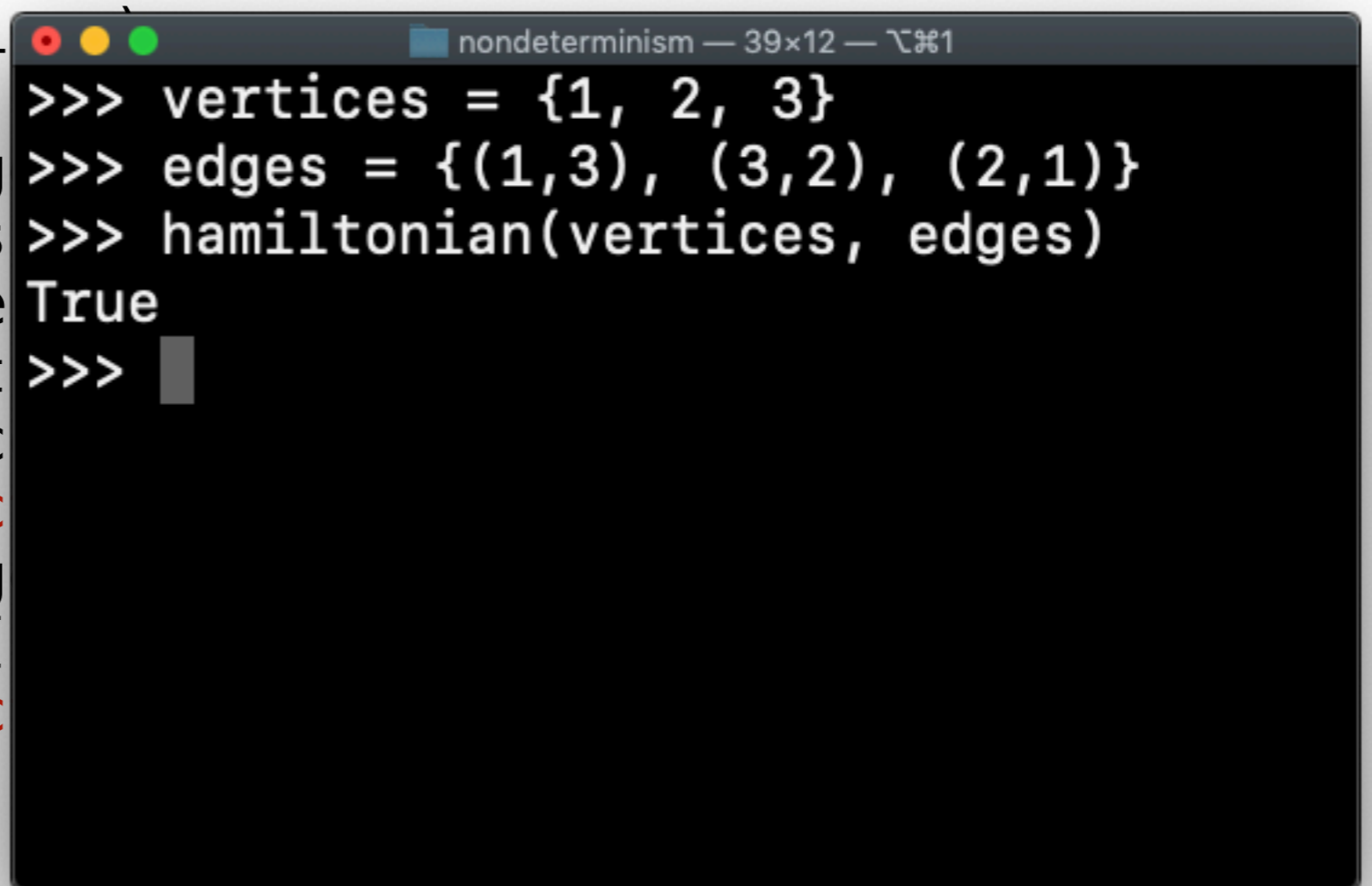
```
            reject()
```

```
    for i in range(n):
```

```
        if perm[i] == vertices[i]:
```

```
            reject()
```

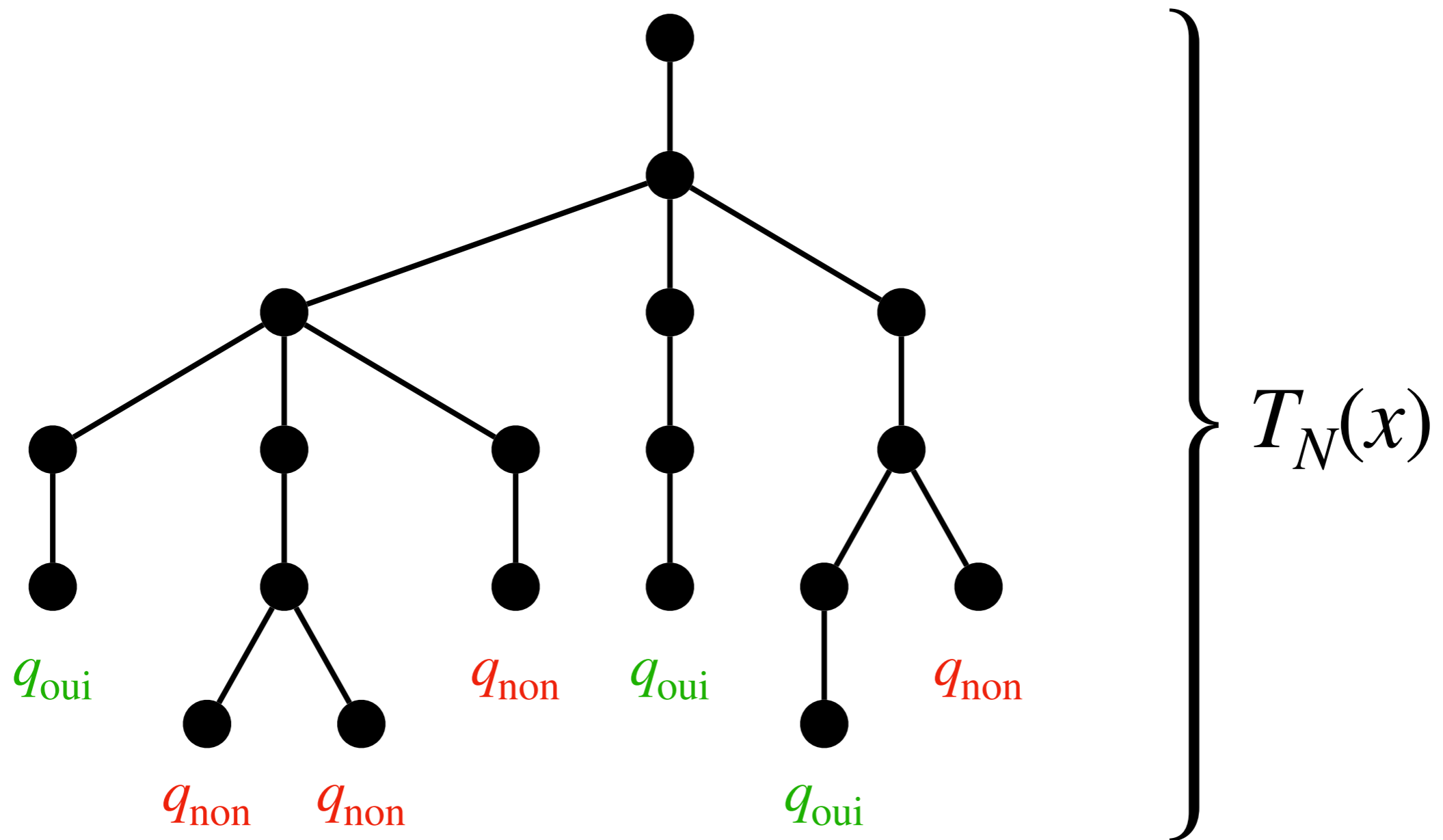
```
    accept()
```

A terminal window titled "nondeterminism — 39x12 — Ƶ#1" showing the execution of the provided Python code. The user enters the following commands: vertices = {1, 2, 3}, edges = {(1,3), (3,2), (2,1)}, and hamiltonian(vertices, edges). The terminal outputs True, and the prompt >>> is shown again on the next line.

```
>>> vertices = {1, 2, 3}
>>> edges = {(1,3), (3,2), (2,1)}
>>> hamiltonian(vertices, edges)
True
>>>
```

Mesure du temps de calcul non déterministe

Temps de calcul sur l'entrée $x =$
hauteur de l'arbre de calcul



Temps de calcul d'une machine non déterministe N

- Comme dans le cas déterministe, on prend le **max** des temps de calcul des entrées de taille n :

$$T_N(n) = \max \{ T_N(x) : x \in \Sigma^* \text{ et } |x| = n \}$$

- Donc la longueur du chemin de calcul le plus long des arbres de calcul des entrées de taille n
- Le temps de calcul est **polynomial** si $T_N(n) \in O(p(n))$ pour un polynôme p

La classe de complexité NP

- C'est la classe de langages reconnus par des machines de Turing **non déterministes** en temps polynomial

$$\mathbf{NP} = \left\{ \begin{array}{l} L : \text{il existe une machine de Turing} \\ \text{non déterministe } N \text{ qui fonctionne en temps} \\ \text{polynomial telle que } L = L(N) \end{array} \right\}$$

- De façon équivalente, c'est aussi la classe de problèmes π sous le codage S tels que $L(\pi, S) \in \mathbf{NP}$


P vs NP

- Comme on peut voir chaque machine déterministe comme machine non déterministe, on a automatiquement $P \subseteq NP$
- On sait qu'on peut **simuler de façon déterministe** chaque machine non déterministe, mais on ne sais pas si on peut le faire **efficacement** (en temps polynomial)
- Donc on ne sais pas si $NP \subseteq P$ et donc si $P = NP$
- C'est l'un des **Millennium Prize Problems** du Clay Mathematics Institute, et il y a un prix de 1000000 \$ pour celle ou celui qui trouve la réponse !

La classe de complexité **coNP**

- C'est la classe de langages dont le complément appartient à **NP** :

$$\mathbf{coNP} = \{L : \text{co-}L \in \mathbf{NP}\}$$

-  **coNP** n'est pas le complément de **NP**, c'est-à-dire, ce n'est pas la classe des problèmes qu'on ne peut pas résoudre en temps polynomial de façon non déterministe
- Notamment, on a $\mathbf{P} = \mathbf{coP}$, donc $\mathbf{P} \subseteq \mathbf{coNP}$, ce qui implique $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$: **NP** et **coNP** ne sont pas disjointes
- On ne sait pas si $\mathbf{NP} = \mathbf{coNP}$ non plus !

$\mathbf{NP} \neq \mathbf{coNP}$ vaut 1000000 \$

- Si $\mathbf{NP} \neq \mathbf{coNP}$ alors $\mathbf{P} \neq \mathbf{NP}$!
- On a vu que $\mathbf{P} = \mathbf{coP}$, donc $\mathbf{P} = \mathbf{NP}$ impliquerait $\mathbf{coNP} = \mathbf{coP} = \mathbf{P}$ et donc $\mathbf{NP} = \mathbf{coNP}$
- La proposition contraposée de $\mathbf{P} = \mathbf{NP} \Rightarrow \mathbf{NP} = \mathbf{coNP}$ est $\mathbf{NP} \neq \mathbf{coNP} \Rightarrow \mathbf{P} \neq \mathbf{NP}$
- \mathbf{NP} vs \mathbf{coNP} ne semble pas du tout plus simple à résoudre que \mathbf{P} vs \mathbf{NP} ...