Langages formels (11X003) - Automne 2024

11. Machines de Turing non déterministes (et autres)

Enseignant: Arnaud Casteigts

Assistants: A.-Q. Berger & M. De Francesco
Monitrices: L. Heiniger & A. Tekkoyun

Dans ce cours, nous allons voir différentes notions liées aux machines de Turing, sans lien particulier. Nous présentons d'abord un exemple de machine de Turing non-déterministe. Nous donnons ensuite plusieurs exemples de machines de Turing qui ne font pas que reconnaître des mots, elles calculent aussi et produisent une sortie ! Enfin, nous discutons brièvement de la thèse de Church-Turing, qui lie ces machines à nos ordinateurs.

11.1 Hiérarchie de Chomsky

La hiérarchie de Chomsky regroupe les équivalences que nous avons vu entre différentes familles de langages (et grammaires correspondantes) et différents modèles de calculs. Les correspondances sont exactes. Les voici :

Grammaire	**Langage**	**Machine**
Régulière	Régulier	Automate fini
Hors-contexte	Hors-contexte	Automate à pile non-déterministe
Contextuelle	Contextuel	MT non-déterministe linéairement bornée
Générale	Turing-reconnaissable	Machine de Turing

Note : Pour des raisons historiques, les langages Turing-reconnaissables sont aussi appelés langages **récursivement énumérable**.

11.2 Machines de Turing non-déterministes

Contrairement aux automates à pile, l'expressivité des machines de Turing (MT) est la même pour les versions déterministes et non-déterministes : les deux reconnaissent les mêmes langages. Cela se démontre par le fait que toute MT non-déterministe peut être **simulée** par une MT déterministe, qui explore séquentiellement toutes les exécutions possible de la machine non-déterministe (nous verrons cela au second semestre). Bien sûr, cette simulation a un surcoût important en temps de calcul, les deux modèles ne sont donc pas équivalents si l'on s'intéresse au temps d'exécution (complexité algorithmique).

D'une certaine manière, les machines non-déterministes n'existent pas (nos ordinateurs sont déterministes). On pourrait donc être tentés d'ignorer leur étude. Mais ces machines

s'avèrent centrales pour plusieurs questions en informatique fondamentale. Par ailleurs, elles permettent souvent d'exprimer un traitement de manière plus élégante.

Comme pour les automates finis ou les automates à pile, le non-déterminisme peut être vu comme le fait d'effectuer plusieurs actions simultanément (dans des univers parallèles), d'où la fonction de transition qui aura pour signature :

$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, S\})$

Une telle machine accepte un mot si et seulement si au moins une des branches d'exécution l'accepte. Les autres branches peuvent rejeter.

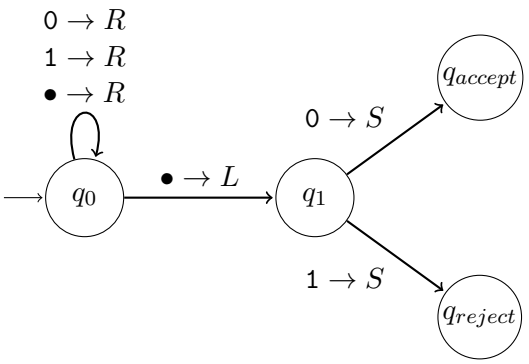
Exemple

Voyons un exemple d'une telle machine : on reçoit en entrée une liste de mots binaires séparés par des • et on veut déterminer si au moins l'un d'entre eux est pair (termine par `0`). Par simplicité, on supposera qu'un point supplémentaire est présent après le dernier mot, par exemple, voici une entrée possible :

`01101` • `0110` • `1010101` • `100100` •

Une machine déterministe pour ce problème examinerait chaque mot séquentiellement. Avec le non-déterminisme, on peut créer différentes branches de calcul qui examinent chacune un mot en parallèle. Plus précisément, lorsque le premier point est atteint, une branche non-déterministe va examiner ce mot, tandis qu'une autre va se diriger directement vers le point suivant (et ainsi de suite). La machine acceptera à condition qu'*au moins une* de ces branches atteigne l'état q_{accept}. À l'intérieur de chaque branche, toutes les transitions qui ne sont pas spécifiées explicitement sont supposées mener à l'état q_{reject}.

Voici la machine correspondante :

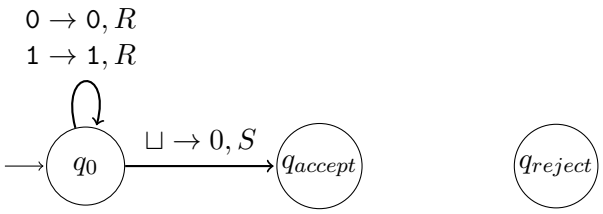


11.3 Calculer avec une machine de Turing

Contrairement aux automates finis ou aux automates à pile, les machines de Turing sont capables de faire d'autres choses que de répondre OUI ou NON. Elles peuvent calculer et produire des choses en sortie, comme nos ordinateurs.

Exemple 1 : multiplier un nombre (binaire) par deux

Description textuelle : aller au bout du mot, rajouter un `0`, puis terminer.



Remarque : L'état q_{reject} n'est pas utilisé ici, mais on pourrait lui donner un sens, par exemple pour signaler une erreur dans le format d'entrée.

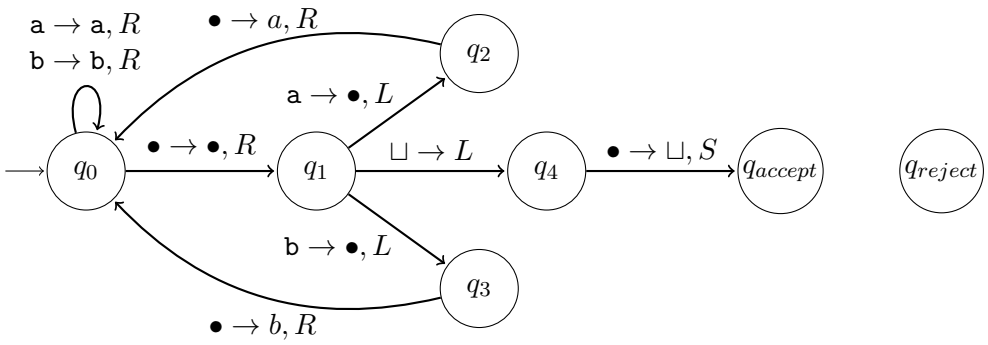
Exemple 2 : concaténer deux mots

On souhaite concaténer deux mots en entrée, disons sur l'alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \bullet\}$, où $\bullet$ est le symbol séparateur entre les deux mots.

Par exemple : `abba` $\bullet$ `bab` doit produire `abbabab`.

Idée générale : décaler le deuxième mot d'un cran vers la gauche.

Description textuelle : on commence par aller vers la droite jusqu'au séparateur. Ensuite on répète : passer le séparateur, remplacer le symbole courant par un deuxième séparateur en reculant d'un cran à gauche, réécrire ce symbole à la place du premier séparateur en avançant d'un cran à droite. Si un espace est rencontré, effacer le dernier point et terminer.

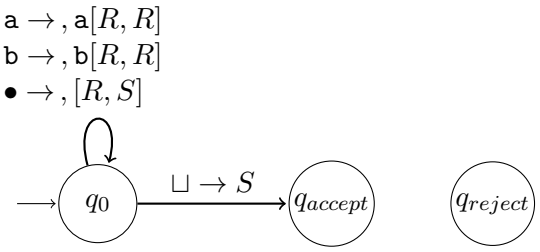


Versions à deux bandes

Comme déjà évoqué, il est fréquent d'utiliser un modèle de MT à plusieurs bandes, surtout lorsque la machine produit une sortie qui ne correspond pas simplement à accepter ou rejeter le mot d'entrée. Cela peut simplifier le traitement à effectuer.

Rappelons que dans le cas général, pour une machine à k bandes, la fonction de transition devient $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$. On peut alors adopter la convention graphique suivante pour dessiner les transitions sur cette machine (par exemple ici pour une machine à deux bandes) : $\mathtt{a}, \mathtt{b} \to \mathtt{c}, \mathtt{d}[R, L]$, avec la signification suivante : si on lit `a` sur la première bande et `b` sur la deuxième bande, alors on écrit `c` sur la première bande, `d` sur la deuxième bande, on déplace la première tête de lecture vers la droite et la deuxième vers la gauche (et bien sûr on se déplace dans l'automate selon cette transition). Les crochets ne servent ici qu'à séparer les déplacement du reste, pour plus de lisibilité. Cette notation peut être généralisée pour n'importe quel nombre de bandes.

Revisitons l'exemple précédent (concaténation) en supposant que la machine a une bande d'entrée (contenant les deux mots à concaténer, séparés d'un point, comme précédemment) et une bande de sortie (qui doit contenir le résultat de la concaténation). Le traitement à effectuer devient plus simple, car il suffit de recopier les deux mots en s'abstenant simplement de copier le séparateur. Cela donne la machine suivante :



Exemple 3 : additionner deux nombres binaires

→ En séance d'exercices. La principale difficulté sera de gérer la retenue.

11.4 Thèse de Church-Turing

La notion d'algorithme existe depuis l'antiquité (au minimum), on peut citer par exemple l'algorithme d'Euclide pour trouver le PGCD de deux nombres. Cependant, c'est seulement au vingtième siècle que la notion d'algorithme a été définie de manière précise, par Alonzo Church et Alan Turing. Le premier a défini le λ-calcul et le second les machines de Turing. Bien que très différents, il s'avère que ces deux modèles sont équivalents en termes d'expressivité (chacun des deux peut simuler l'autre). Church et Turing ont alors conjecturé que ces modèles peuvent exprimer n'importe quel traitement physiquement réalisable. C'est ce qu'on appelle la **thèse de Church-Turing**, que personne n'a jamais réussi à contredire jusqu'à aujourd'hui (malgré de nombreuses tentatives). Autrement dit, ces modèles semblent capturer exactement ce qui est physiquement calculable, y compris par nos ordinateurs actuels, avec le grand avantage qu'ils peuvent être définis de manière mathématique et donc étudiés rigoureusement.

En fait, l'écrasante majorité des modèles de calcul imaginés depuis se sont avérés équivalents aux machines de Turing. Autrement dit, à partir d'un certain niveau d'expressivité, tous les modèles sont universels et peuvent se simuler les uns les autres. Cela vaut pour les langages de programmation également : tous les langages de programmations classiques tels que le C, Python, Java, Typescript, *etc.* sont équivalents aux machines de Turing et sont capables de se simuler les uns les autres. On dit qu'ils sont **Turing-complets**.