Introduction à la Calculabilité et la Compléxité

Prof. Salah BAÏNA

INSEA - ENSIAS

Année 2013/2014

Références

- **Ouvrages:**

Introduction à la calculabilité, Pierre Wolper,
InterEditions, 1991

- **Supports de Cours** (disponnibles sur la toile)**:**

Algorithmique et Calculabilité, Michel Rigo,
Université de Liège, 2009/2010

Cours de Calculabilité et Compléxité, Jean-François Raskin,
Université Libre de Bruxelles, 2008/2009

Langages Formels, Calculabilité Complexité, Olivier Carton,
Ecole Normale Supérieure, 2007/2008

Éléments de Théorie de la Calculabilité, Sebastiaan A. Terwijn,
Institute for Discrete Mathematics and Geometry
Technical University of Vienna,
traduit de l'anglais par Michaël Cadilhac.

Préambule 1

"The question, 'Can machines think?' I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general, educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."

Alan Turing, 1950

Plan du Cours

Plan

L'ORIGINE DU CALCUL

Depuis des siècles les mathématiciens décrivent et utilisent des méthodes de calcul pour résoudre leurs porblèmes. Mais ce n'est que très récemment que ces méthodes ont été identifiées et définies (1934)

- On a ainsi classé et identifié différents types de concept
  - Algorithmes
  - Fonctions Calculables
  - Problèmes décidables et indécidables
  - Énoncés décidables et indécidables

L'ORIGINE DU CALCUL

Depuis des siècles les mathématiciens décrivent et utilisent des méthodes de calcul pour résoudre leurs porblèmes. Mais ce n'est que très récemment que ces méthodes ont été identifiées et définies (1934)

- On a ainsi classé et identifié différents types de concept
  - Algorithmes
  - Fonctions Calculables
  - Problèmes décidables et indécidables
  - Énoncés décidables et indécidables

CALCUL ET NON CALCUL

- Aujourd'hui les mathématiciens possèdent une définition précise de ce qu'on doit appeler méthode de calcul, ils savent exactement quels sont les problèmes qu'elles peuvent traiter, et mieux encore,
  - ils savent que certains problèmes ne sont pas "traitables"
- ils permettent d'établir des résultats négatifs,
  - pour tel problème non seulement aucune méthode n'est actuellement connue, mais aucune ne le sera jamais.
- Ces problèmes pour lesquels on démontre qu'il n'existe aucune méthode de calcul adaptée s'appellent des problèmes indécidables.

CALCUL ET NON CALCUL

- Aujourd'hui les mathématiciens possèdent une définition précise de ce qu'on doit appeler méthode de calcul, ils savent exactement quels sont les problèmes qu'elles peuvent traiter, et mieux encore,
  - ils savent que certains problèmes ne sont pas "traitables"
- ils permettent d'établir des résultats négatifs,
  - pour tel problème non seulement aucune méthode n'est actuellement connue, mais aucune ne le sera jamais.
- Ces problèmes pour lesquels on démontre qu'il n'existe aucune méthode de calcul adaptée s'appellent des problèmes indécidables.

CALCUL ET NON CALCUL

- Aujourd'hui les mathématiciens possèdent une définition précise de ce qu'on doit appeler méthode de calcul, ils savent exactement quels sont les problèmes qu'elles peuvent traiter, et mieux encore,
  - ils savent que certains problèmes ne sont pas "traitables"
- ils permettent d'établir des résultats négatifs,
  - pour tel problème non seulement aucune méthode n'est actuellement connue, mais aucune ne le sera jamais.
- Ces problèmes pour lesquels on démontre qu'il n'existe aucune méthode de calcul adaptée s'appellent des problèmes indécidables.

CALCUL ET NON CALCUL

- Aujourd'hui les mathématiciens possèdent une définition précise de ce qu'on doit appeler méthode de calcul, ils savent exactement quels sont les problèmes qu'elles peuvent traiter, et mieux encore,
  - ils savent que certains problèmes ne sont pas "traitables"
- ils permettent d'établir des résultats négatifs,
  - pour tel problème non seulement aucune méthode n'est actuellement connue, mais aucune ne le sera jamais.
- Ces problèmes pour lesquels on démontre qu'il n'existe aucune méthode de calcul adaptée s'appellent des problèmes indécidables.

CALCUL ET NON CALCUL

- Aujourd'hui les mathématiciens possèdent une définition précise de ce qu'on doit appeler méthode de calcul, ils savent exactement quels sont les problèmes qu'elles peuvent traiter, et mieux encore,
  - ils savent que certains problèmes ne sont pas "traitables"
- ils permettent d'établir des résultats négatifs,
  - pour tel problème non seulement aucune méthode n'est actuellement connue, mais aucune ne le sera jamais.
- Ces problèmes pour lesquels on démontre qu'il n'existe aucune méthode de calcul adaptée s'appellent des problèmes indécidables.

Calcul et Application

- Les premiers résultats d'indécidabilité des années 1930 semblaient artificiels, très vite on s'est aperçu que des problèmes assez simples entraient dans la classe des problèmes indécidables.
- En particulier en informatique, de nombreuses questions naturelles qui se posent aux programmeurs se révèlent après étude correspondre à des problèmes indécidables.
- Lorsqu'on démontre qu'un problème P est indécidable, on en déduit :
  - qu'on doit renoncer à le résoudre tel quel, et donc que,
  - qu'il faut trouver une version simplifiée de P, puis, soit réussir à la résoudre, soit à nouveau établir qu'elle est encore indécidable et donc la simplifier encore, etc.

Calcul et Application

- Les premiers résultats d'indécidabilité des années 1930 semblaient artificiels, très vite on s'est aperçu que des problèmes assez simples entraient dans la classe des problèmes indécidables.
- En particulier en informatique, de nombreuses questions naturelles qui se posent aux programmeurs se révèlent après étude correspondre à des problèmes indécidables.
- Lorsqu'on démontre qu'un problème P est indécidable, on en déduit :
  - qu'on doit renoncer à le résoudre tel quel, et donc que,
  - qu'il faut trouver une version simplifiée de P, puis, soit réussir à la résoudre, soit à nouveau établir qu'elle est encore indécidable et donc la simplifier encore, etc.

Calcul et Application

- Les premiers résultats d'indécidabilité des années 1930 semblaient artificiels, très vite on s'est aperçu que des problèmes assez simples entraient dans la classe des problèmes indécidables.
- En particulier en informatique, de nombreuses questions naturelles qui se posent aux programmeurs se révèlent après étude correspondre à des problèmes indécidables.
- Lorsqu'on démontre qu'un problème P est indécidable, on en déduit :
  - qu'on doit renoncer à le résoudre tel quel, et donc que,
  - qu'il faut trouver une version simplifiée de P, puis, soit réussir à la résoudre, soit à nouveau établir qu'elle est encore indécidable et donc la simplifier encore, etc.

Calcul et Application

- Les premiers résultats d'indécidabilité des années 1930 semblaient artificiels, très vite on s'est aperçu que des problèmes assez simples entraient dans la classe des problèmes indécidables.
- En particulier en informatique, de nombreuses questions naturelles qui se posent aux programmeurs se révèlent après étude correspondre à des problèmes indécidables.
- Lorsqu'on démontre qu'un problème P est indécidable, on en déduit :
  - qu'on doit renoncer à le résoudre tel quel, et donc que,
  - qu'il faut trouver une version simplifiée de P, puis, soit réussir à la résoudre, soit à nouveau établir qu'elle est encore indécidable et donc la simplifier encore, etc.

Calcul et Application

- Les premiers résultats d'indécidabilité des années 1930 semblaient artificiels, très vite on s'est aperçu que des problèmes assez simples entraient dans la classe des problèmes indécidables.
- En particulier en informatique, de nombreuses questions naturelles qui se posent aux programmeurs se révèlent après étude correspondre à des problèmes indécidables.
- Lorsqu'on démontre qu'un problème P est indécidable, on en déduit :
  - qu'on doit renoncer à le résoudre tel quel, et donc que,
  - qu'il faut trouver une version simplifiée de P, puis, soit réussir à la résoudre, soit à nouveau établir qu'elle est encore indécidable et donc la simplifier encore, etc.

À L'ORIGINE IL Y AVAIT DES HOMMES

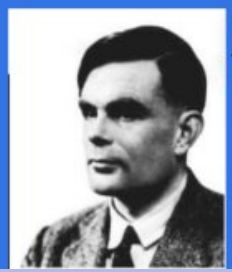
people

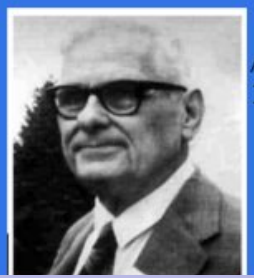
recent


Al Khwarazmi
783 - 850


Blaise Pascal
1623 - 1662


Emile Post
1897 - 1954


Alan Turing
1912 - 1954


Alonso Church
1903 - 1995


Kurt Gödel
1906 - 1978

À L'ORIGINE IL Y AVAIT DES HOMMES

- La notion informelle d'algorithme est extrêmement ancienne et les procédés que nous avons appris à l'école primaire pour additionner deux nombres écrits en base dix ou pour les multiplier sont des algorithmes.
- Le mot même d'algorithme provient du nom d'un mathématicien arabe du IXe siècle, Al Khawarizmi .
- Les mathématiciens recherchaient et élaboraient des algorithmes bien avant que le concept ne soit clairement identifié et défini.
- En fait, au début du siècle les mathématiciens ne soupçonnaient pas qu'on pourrait préciser vraiment cette notion, ni encore moins, qu'on pourrait démontrer pour certains problèmes qu'aucun algorithme n'existe.

À L'ORIGINE IL Y AVAIT DES HOMMES

- La notion informelle d'algorithme est extrêmement ancienne et les procédés que nous avons appris à l'école primaire pour additionner deux nombres écrits en base dix ou pour les multiplier sont des algorithmes.
- Le mot même d'algorithme provient du nom d'un mathématicien arabe du IXe siècle, Al Khawarizmi .
- Les mathématiciens recherchaient et élaboraient des algorithmes bien avant que le concept ne soit clairement identifié et défini.
- En fait, au début du siècle les mathématiciens ne soupçonnaient pas qu'on pourrait préciser vraiment cette notion, ni encore moins, qu'on pourrait démontrer pour certains problèmes qu'aucun algorithme n'existe.

À L'ORIGINE IL Y AVAIT DES HOMMES

- La notion informelle d'algorithme est extrêmement ancienne et les procédés que nous avons appris à l'école primaire pour additionner deux nombres écrits en base dix ou pour les multiplier sont des algorithmes.
- Le mot même d'algorithme provient du nom d'un mathématicien arabe du IXe siècle, Al Khawarizmi .
- Les mathématiciens recherchaient et élaboraient des algorithmes bien avant que le concept ne soit clairement identifié et défini.
- En fait, au début du siècle les mathématiciens ne soupçonnaient pas qu'on pourrait préciser vraiment cette notion, ni encore moins, qu'on pourrait démontrer pour certains problèmes qu'aucun algorithme n'existe.

À L'ORIGINE IL Y AVAIT DES HOMMES

- La notion informelle d'algorithme est extrêmement ancienne et les procédés que nous avons appris à l'école primaire pour additionner deux nombres écrits en base dix ou pour les multiplier sont des algorithmes.
- Le mot même d'algorithme provient du nom d'un mathématicien arabe du IXe siècle, Al Khawarizmi .
- Les mathématiciens recherchaient et élaboraient des algorithmes bien avant que le concept ne soit clairement identifié et défini.
- En fait, au début du siècle les mathématiciens ne soupçonnaient pas qu'on pourrait préciser vraiment cette notion, ni encore moins, qu'on pourrait démontrer pour certains problèmes qu'aucun algorithme n'existe.

À L'ORIGINE IL Y AVAIT DES HOMMES

- Le dixième problème de Hilbert (formulé avec 23 autres au Congrès International des Mathématiciens de Paris en 1900) se réfère implicitement à la notion d'algorithme.
- Hilbert demandait qu'on recherche une méthode générale indiquant quelles équations diophantiennes (polinômes à coéfficients entiers) ont des solutions et lesquelles n'en n'ont pas.
- Il n'envisageait pas qu'on puisse établir que ce problème est indécidable, comme cela fut fait en 1970 par Matijasevic.
- Le travail d'identification et de formulation de la notion d'algorithme fut effectué en plusieurs étapes entre 1931 et 1936 par les mathématiciens Church, Kleene, Turing et Gödel.

À L'ORIGINE IL Y AVAIT DES HOMMES

- Le dixième problème de Hilbert (formulé avec 23 autres au Congrès International des Mathématiciens de Paris en 1900) se réfère implicitement à la notion d'algorithme.
- Hilbert demandait qu'on recherche une méthode générale indiquant quelles équations diophantiennes (polinômes à coéfficients entiers) ont des solutions et lesquelles n'en n'ont pas.
- Il n'envisageait pas qu'on puisse établir que ce problème est indécidable, comme cela fut fait en 1970 par Matijasevic.
- Le travail d'identification et de formulation de la notion d'algorithme fut effectué en plusieurs étapes entre 1931 et 1936 par les mathématiciens Church, Kleene, Turing et Gödel.

À L'ORIGINE IL Y AVAIT DES HOMMES

- Le dixième problème de Hilbert (formulé avec 23 autres au Congrès International des Mathématiciens de Paris en 1900) se réfère implicitement à la notion d'algorithme.
- Hilbert demandait qu'on recherche une méthode générale indiquant quelles équations diophantiennes (polinômes à coéfficients entiers) ont des solutions et lesquelles n'en n'ont pas.
- Il n'envisageait pas qu'on puisse établir que ce problème est indécidable, comme cela fut fait en 1970 par Matijasevic.
- Le travail d'identification et de formulation de la notion d'algorithme fut effectué en plusieurs étapes entre 1931 et 1936 par les mathématiciens Church, Kleene, Turing et Gödel.

À L'ORIGINE IL Y AVAIT DES HOMMES

- Le dixième problème de Hilbert (formulé avec 23 autres au Congrès International des Mathématiciens de Paris en 1900) se réfère implicitement à la notion d'algorithme.
- Hilbert demandait qu'on recherche une méthode générale indiquant quelles équations diophantiennes (polinômes à coéfficients entiers) ont des solutions et lesquelles n'en n'ont pas.
- Il n'envisageait pas qu'on puisse établir que ce problème est indécidable, comme cela fut fait en 1970 par Matijasevic.
- Le travail d'identification et de formulation de la notion d'algorithme fut effectué en plusieurs étapes entre 1931 et 1936 par les mathématiciens Church, Kleene, Turing et Gödel.

ET BEAUCOUP DE TRAVAIL

- Ils introduisirent plusieurs classes différentes de fonctions dont ils montrèrent ensuite qu'elles coïncidaient, et qu'ils reconnurent alors comme la classe des fonctions calculables.
- Une fonction est calculable s'il existe une façon finie de la décrire qui permette effectivement d'en calculer toutes les valeurs.
- La définition précise de la notion de fonction calculable fixe en même temps celle d'algorithme, et on peut donc dire qu'en 1936 la formulation exacte de la notion d'algorithme était acquise.

ET BEAUCOUP DE TRAVAIL

- Ils introduisirent plusieurs classes différentes de fonctions dont ils montrèrent ensuite qu'elles coïncidaient, et qu'ils reconnurent alors comme la classe des fonctions calculables.
- Une fonction est calculable s'il existe une façon finie de la décrire qui permette effectivement d'en calculer toutes les valeurs.
- La définition précise de la notion de fonction calculable fixe en même temps celle d'algorithme, et on peut donc dire qu'en 1936 la formulation exacte de la notion d'algorithme était acquise.

ET BEAUCOUP DE TRAVAIL

- Ils introduisirent plusieurs classes différentes de fonctions dont ils montrèrent ensuite qu'elles coïncidaient, et qu'ils reconnurent alors comme la classe des fonctions calculables.
- Une fonction est calculable s'il existe une façon finie de la décrire qui permette effectivement d'en calculer toutes les valeurs.
- La définition précise de la notion de fonction calculable fixe en même temps celle d'algorithme, et on peut donc dire qu'en 1936 la formulation exacte de la notion d'algorithme était acquise.

Épilogue

- De toutes les définitions finalement équivalentes de la notion d'algorithme formulées dans les années 1930 et depuis, celle donnée par Alan Turing en 1936 est la plus pratique pour les théoriciens, et on l'utilise encore aujourd'hui. C'est sur elle que nous allons nous appuyer plus loin pour définir la notion d'algorithme et montrer que certains problèmes précis n'ont pas d'algorithmes.

Plan

Problèmatique

- Calculabilité : Quels problèmes peut-on résoudre (ou pas) avec une machine (indépendamment de la téchnologie actuelle)?
- *Inconnues: Qu'est-ce qu'un problème? et Qu'est-ce qu'une Machine?*

Example

- Est-ce que n est pair?
- Est-ce que n est premier?

Problèmatique

- Calculabilité : Quels problèmes peut-on résoudre (ou pas) avec une machine (indépendamment de la téchnologie actuelle)?
- *Inconnues: Qu'est-ce qu'un problème? et Qu'est-ce qu'une Machine?*

Example

- Est-ce que n est pair?
- Est-ce que n est premier?

Problèmatique

- Calculabilité : Quels problèmes peut-on résoudre (ou pas) avec une machine (indépendamment de la téchnologie actuelle)?
- *Inconnues: Qu'est-ce qu'un problème? et Qu'est-ce qu'une Machine?*

Example

- Est-ce que n est pair?
- Est-ce que n est premier?

Un Problème

- Un *Problème* est une question générique qui porte sur un ensemble de données dans une représentation fixée. La généricité provient des paramètres ou variables libres que contient l'énoncé du problème;
- Une instance de problème est la question posée pour une entrée particulière à chacune de ses variables libres;
- L'existence du problème est indépendante de l'existence d'un algorithme qui le résout;
- Un problème peut avoir plusieurs solutions, plusieurs algorithmes différents peuvent résoudre le même problème.

Un Problème

- Un *Problème* est une question générique qui porte sur un ensemble de données dans une représentation fixée. La généricité provient des paramètres ou variables libres que contient l'énoncé du problème;
- Une instance de problème est la question posée pour une entrée particulière à chacune de ses variables libres;
- L'existence du problème est indépendante de l'existence d'un algorithme qui le résout;
- Un problème peut avoir plusieurs solutions, plusieurs algorithmes différents peuvent résoudre le même problème.

Un Problème

- Un *Problème* est une question générique qui porte sur un ensemble de données dans une représentation fixée. La généricité provient des paramètres ou variables libres que contient l'énoncé du problème;
- Une instance de problème est la question posée pour une entrée particulière à chacune de ses variables libres;
- L'existence du problème est indépendante de l'existence d'un algorithme qui le résout;
- Un problème peut avoir plusieurs solutions, plusieurs algorithmes différents peuvent résoudre le même problème.

Un Problème

- Un *Problème* est une question générique qui porte sur un ensemble de données dans une représentation fixée. La généricité provient des paramètres ou variables libres que contient l'énoncé du problème;
- Une instance de problème est la question posée pour une entrée particulière à chacune de ses variables libres;
- L'existence du problème est indépendante de l'existence d'un algorithme qui le résout;
- Un problème peut avoir plusieurs solutions, plusieurs algorithmes différents peuvent résoudre le même problème.

Problèmes binaires

- Un Problème Binaire est une fonction à valeur dans {oui, non}

Example

- Est-ce que *302931* est pair?
- Est-ce que *567433* est premier?
- "LOL" est un palyndrôme?

Problèmes binaires

- Un Problème Binaire est une fonction à valeur dans {oui, non}

Example

- Est-ce que *302931* est pair?
- Est-ce que *567433* est premier?
- "LOL" est un palyndrôme?

Problèmes binaires

- Un Problème Binaire est une fonction à valeur dans {oui, non}

Example

- Est-ce que *302931* est pair?
- Est-ce que *567433* est premier?
- "LOL" est un palyndrôme?

Problèmes binaires

- Un Problème Binaire est une fonction à valeur dans {oui, non}

Example

- Est-ce que *302931* est pair?
- Est-ce que *567433* est premier?
- "LOL" est un palyndrôme?

Problèmes binaires

- Un Problème Binaire est une fonction à valeur dans {oui, non}

Example

- Est-ce que *302931* est pair?
- Est-ce que *567433* est premier?
- "LOL" est un palyndrôme?

Problèmes de décisions

- On appelle "problème de décision": la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquel les la réponse est oui.

Example

On peut considérer les problèmes suivants :

- Nombres premiers : $E = \mathbb{N}$ et $P = \{n \mid n \text{ premier}\}$.
- Automates (acceptance) : $E = \{(A, w) \mid A \text{ automate et } w \text{ mot}\}$, $P = \{(A, w) \mid A \text{ accepte } w\}$.
- Graphes connexes : $E = \{G \mid G \text{ graphe fini}\}$, $P = \{G \mid G \text{ est connexe}\}$.
- Grammaires ambiguës : $E = \{G \mid G \text{ grammaire}\}$, $P = \{G \mid G \text{ ambiguë}\}$.

Problèmes de décisions

- On appelle "problème de décision": la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquel les la réponse est oui.

Example

On peut considérer les problèmes suivants :

- Nombres premiers : $E = \mathbb{N}$ et $P = \{n \mid n \text{ premier}\}$.
- Automates (acceptance) : $E = \{(A, w) \mid A \text{ automate et } w \text{ mot}\}$, $P = \{(A, w) \mid A \text{ accepte } w\}$.
- Graphes connexes : $E = \{G \mid G \text{ graphe fini}\}$, $P = \{G \mid G \text{ est connexe}\}$.
- Grammaires ambiguës : $E = \{G \mid G \text{ grammaire}\}$, $P = \{G \mid G \text{ ambiguë}\}$.

Problèmes de décisions

- On appelle "problème de décision": la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquel les la réponse est oui.

Example

On peut considérer les problèmes suivants :

- Nombres premiers : $E = \mathbb{N}$ et $P = \{n \mid n \text{ premier}\}$.
- Automates (acceptance) : $E = \{(A, w) \mid A \text{ automate et } w \text{ mot}\}$, $P = \{(A, w) \mid A \text{ accepte } w\}$.
- Graphes connexes : $E = \{G \mid G \text{ graphe fini}\}$, $P = \{G \mid G \text{ est connexe}\}$.
- Grammaires ambiguës : $E = \{G \mid G \text{ grammaire}\}$, $P = \{G \mid G \text{ ambiguë}\}$.

Problèmes de décisions

- On appelle "problème de décision": la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquel les la réponse est oui.

Example

On peut considérer les problèmes suivants :

- Nombres premiers : $E = \mathbb{N}$ et $P = \{n \mid n \text{ premier}\}$.
- Automates (acceptance) : $E = \{(A, w) \mid A \text{ automate et } w \text{ mot}\}$, $P = \{(A, w) \mid A \text{ accepte } w\}$.
- Graphes connexes : $E = \{G \mid G \text{ graphe fini}\}$, $P = \{G \mid G \text{ est connexe}\}$.
- Grammaires ambiguës : $E = \{G \mid G \text{ grammaire}\}$, $P = \{G \mid G \text{ ambiguë}\}$.

Problèmes de décisions

- On appelle "problème de décision": la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquel les la réponse est oui.

Example

On peut considérer les problèmes suivants :

- Nombres premiers : E = $\mathbb{N}$ et P = {n | n premier}.
- Automates (acceptance) : E = {(A, w) | A automate et w mot}, P ={(A, w) | A accepte w}.
- Graphes connexes : E = {G | G graphe fini}, P = {G | G est connexe}.
- Grammaires ambiguës : E = {G | G grammaire}, P = {G | G ambiguë}.

Autres Exemples

- Analyse Syntaxique:

  Donnée: Un programme C.
  Énoncé: Est-ce que le programme est syntaxiquement correcte?

- Arrêt Universel:

  Donnée: Un programme C
  Énoncé: Est ce que le programme s'arrête sur toute les données en entrée?

- Arrêt Existentiel:

  Donnée: Un programme C
  Énoncé: Est ce que le programme s'arrête au moins sur une donnée en entrée?

Autres Exemples

- Analyse Syntaxique:

  Donnée: Un programme C.
  Énoncé: Est-ce que le programme est syntaxiquement correcte?

- Arrêt Universel:

  Donnée: Un programme C
  Énoncé: Est ce que le programme s'arrête sur toute les données en entrée?

- Arrêt Existentiel:

  Donnée: Un programme C
  Énoncé: Est ce que le programme s'arrête au moins sur une donnée en entrée?

Autres Exemples

- Analyse Syntaxique:

  Donnée: Un programme C.
  Énoncé: Est-ce que le programme est syntaxiquement correcte?

- Arrêt Universel:

  Donnée: Un programme C
  Énoncé: Est ce que le programme s'arrête sur toute les données en entrée?

- Arrêt Existentiel:

  Donnée: Un programme C
  Énoncé: Est ce que le programme s'arrête au moins sur une donnée en entrée?

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

Problème : ses entrées et ses sorties

- Une donnée ou un résultat est un mot écrit sur un alphabet A.
- L'ensemble des données d'un probléme est un langage sur A^*.
- Un problème partitionne donc A^*en trois ensembles :
  - Instances positives: L_P
  - Instances négatives: L_N
  - non instances: $A^* / (L_P \cup L_N)$

Souvent, on identifie un problème à l'ensemble de ses instances positives.

La méthode de résolution

- Derrière la notion d'algorithme se trouve la notion de procédure effective pour résoudre un problème (toutes les instances d'un problème).
- Un programme (C, Pascal, Java, Ada, Assembleur) est une procédure effective. Pourquoi ?
  - Le code peut-être compilé et ensuite exécuté.
- Une autre caractéristique essentielle d'une procédure effective est qu'elle contient exactement la marche à suivre pour a résoudre le problème et qu'aucune décision supplémentaire ne doit être prise lors de l'exécution de la procédure.

La méthode de résolution

- Derrière la notion d'algorithme se trouve la notion de procédure effective pour résoudre un problème (toutes les instances d'un problème).
- Un programme (C, Pascal, Java, Ada, Assembleur) est une procédure effective. Pourquoi ?
  - Le code peut-être compilé et ensuite exécuté.
- Une autre caractéristique essentielle d'une procédure effective est qu'elle contient exactement la marche à suivre pour a résoudre le problème et qu'aucune décision supplémentaire ne doit être prise lors de l'exécution de la procédure.

La méthode de résolution

- Derrière la notion d'algorithme se trouve la notion de procédure effective pour résoudre un problème (toutes les instances d'un problème).
- Un programme (C, Pascal, Java, Ada, Assembleur) est une procédure effective. Pourquoi ?
  - Le code peut-être compilé et ensuite exécuté.
- Une autre caractéristique essentielle d'une procédure effective est qu'elle contient exactement la marche à suivre pour a résoudre le problème et qu'aucune décision supplémentaire ne doit être prise lors de l'exécution de la procédure.

Une Procédure effective

- Il existe des problèmes qu'on ne peut pas résoudre avec un algorithme, un programme, une méthode, etc.
- Voici un exemple d'une procédure qui n'est pas effective pour résoudre
  - le problème de l'arrêt : Déterminez si le programme n'a pas de boucle infinies ou d'appels récursifs infinis.
- Il est clair que cett solution n'est pas effective : Comment détecter les boucles infinies ou les appels récursifs infinis. Notons que nous établirons dans la suite de ce cours qu'il n'existe aucune procédure effective pour résoudre le problème de l'arrêt.

Une Procédure effective

- Il existe des problèmes qu'on ne peut pas résoudre avec un algorithme, un programme, une méthode, etc.
- Voici un exemple d'une procédure qui n'est pas effective pour résoudre
  - le problème de l'arrêt : Déterminez si le programme n'a pas de boucle infinies ou d'appels récursifs infinis.
- Il est clair que cett solution n'est pas effective : Comment détecter les boucles infinies ou les appels récursifs infinis. Notons que nous établirons dans la suite de ce cours qu'il n'existe aucune procédure effective pour résoudre le problème de l'arrêt.

Une Procédure effective

- Il existe des problèmes qu'on ne peut pas résoudre avec un algorithme, un programme, une méthode, etc.
- Voici un exemple d'une procédure qui n'est pas effective pour résoudre
  - le problème de l'arrêt : Déterminez si le programme n'a pas de boucle infinies ou d'appels récursifs infinis.
- Il est clair que cett solution n'est pas effective : Comment détecter les boucles infinies ou les appels récursifs infinis. Notons que nous établirons dans la suite de ce cours qu'il n'existe aucune procédure effective pour résoudre le problème de l'arrêt.

Exemple de Problème non Décidable

- Problème de géométrie élémentaire P:
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
- Il a été démontré que ce problème est indécidable.
- il n'existe aucun algorithme permettant par un calcul fini à partir des données (la liste des formes géométriques) d'établir si OUI ou NON il est possible de paver le plan avec des exemplaires des formes géométriques considérées.

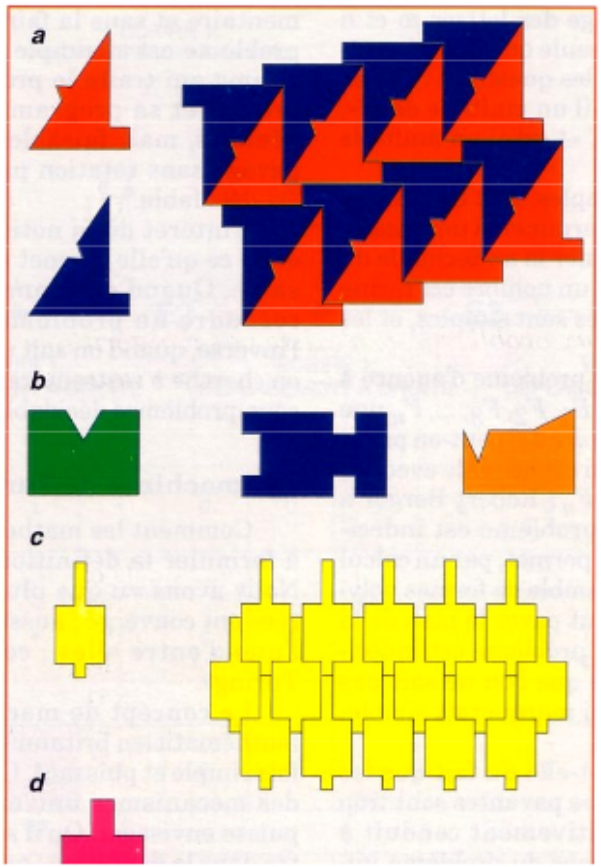
Exemple de Problème non Décidable

- Problème de géométrie élémentaire P:
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
- Il a été démontré que ce problème est indécidable.
- il n'existe aucun algorithme permettant par un calcul fini à partir des données (la liste des formes géométriques) d'établir si OUI ou NON il est possible de paver le plan avec des exemplaires des formes géométriques considérées.

Exemple de Problème non Décidable

- Problème de géométrie élémentaire P:
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
- Il a été démontré que ce problème est indécidable.
- il n'existe aucun algorithme permettant par un calcul fini à partir des données (la liste des formes géométriques) d'établir si OUI ou NON il est possible de paver le plan avec des exemplaires des formes géométriques considérées.

Exemple de Problème non Décidable



CERTAINS PROBLÈMES DE PAVAGE DU PLAN par des polygones sont indécidables. Quand on se donne les deux polygones du haut de la figure, on peut paver le plan. Dans ce cas précis, on trouve facilement un pavage. La méthode qui permet de résoudre le problème dans ce cas est-elle généralisable? R. Berger a démontré en 1966 que non : le problème est indécidable. Déjà, pour les trois formes du milieu de la figure, il faut un peu d'inventivité pour démontrer qu'un pavage du plan sans recouvrement ni espace vide est impossible. L'indécidabilité du problème du pavage signifie que, pour traiter de nouvelles situations, le mathématicien sera inévitablement amené à inventer de nouvelles méthodes de raisonnement : jamais aucun procédé général mécanique ne réussira à englober tous les cas possibles. En revanche, pour un polyomino, composé de carrés adjacents, on sait que la question du pavage est décidable : il existe un algorithme qui, lorsqu'on lui donne un polyomino, indique correctement s'il est possible de paver le plan en l'utilisant sans le faire tourner. La frontière entre le décidable et l'indécidable passe entre cette forme simplifiée du problème du pavage et la version générale. L'indécidabilité du problème des pavages du plan est liée à l'existence de pavés qui ne peuvent recouvrir le plan que non périodiquement. Un pavage non périodique d'un nouveau type a été découvert en 1994 par Charles Radin, de l'Université du Texas : contrairement à tous les pavages connus jusqu'à présent, le pavage de Radin oblige les pavés à effectuer des rotations selon une infinité d'angles différents.

Figure: problème du pavage

Exemple de Problème non Décidable

- Lorsqu'on connait un algorithme pour résoudre un problème il est alors naturel de chercher à le généraliser (i.e. être plus tolérant dans les jeux de données acceptés).
- A l'inverse quand on sait qu'un problème est indécidable, on cherche alors à en trouver des sous-problèmes décidables.

Exemple de Problème non Décidable

- Lorsqu'on connait un algorithme pour résoudre un problème il est alors naturel de chercher à le généraliser (i.e. être plus tolérant dans les jeux de données acceptés).
- A l'inverse quand on sait qu'un problème est indécidable, on cherche alors à en trouver des sous-problèmes décidables.

Exemple de Problème non Décidable

- Problème P'
  - Soient $F_1, F_2, ..., F_n$ une liste de formes géométriques composées chacune de carrés juxtaposés. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Il y a beaucoup moins de "listes de formes géométriques composées chacune de carrés juxtaposés" que de "listes de formes géométriques", le problème P' est donc plus simple que le problème P.
  - Il se pourrait donc que P', puisse être résolu par un algorithme.
  - Pourtant ce n'est pas le cas (la démonstration que le problème B était indécidable établit en fait que B' est indécidable)
- Problème P"
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver périodiquement le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Là, le sous-problème est plus simple et est décidable.

Exemple de Problème non Décidable

- Problème P'
  - Soient $F_1, F_2, ..., F_n$ une liste de formes géométriques composées chacune de carrés juxtaposés. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Il y a beaucoup moins de "listes de formes géométriques composées chacune de carrés juxtaposés" que de "listes de formes géométriques", le problème P' est donc plus simple que le problème P.
  - Il se pourrait donc que P', puisse être résolu par un algorithme.
  - Pourtant ce n'est pas le cas (la démonstration que le problème B était indécidable établit en fait que B' est indécidable)
- Problème P"
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver périodiquement le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Là, le sous-problème est plus simple et est décidable.

Exemple de Problème non Décidable

- Problème P'
  - Soient $F_1, F_2, ..., F_n$ une liste de formes géométriques composées chacune de carrés juxtaposés. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Il y a beaucoup moins de "listes de formes géométriques composées chacune de carrés juxtaposés" que de "listes de formes géométriques", le problème P' est donc plus simple que le problème P.
  - Il se pourrait donc que P', puisse être résolu par un algorithme.
  - Pourtant ce n'est pas le cas (la démonstration que le problème B était indécidable établit en fait que B' est indécidable)
- Problème P"
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver périodiquement le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Là, le sous-problème est plus simple et est décidable.

Exemple de Problème non Décidable

- Problème P'
  - Soient $F_1, F_2, ..., F_n$ une liste de formes géométriques composées chacune de carrés juxtaposés. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Il y a beaucoup moins de "listes de formes géométriques composées chacune de carrés juxtaposés" que de "listes de formes géométriques", le problème P' est donc plus simple que le problème P.
  - Il se pourrait donc que P', puisse être résolu par un algorithme.
  - Pourtant ce n'est pas le cas (la démonstration que le problème B était indécidable établit en fait que B' est indécidable)
- Problème P"
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver périodiquement le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Là, le sous-problème est plus simple et est décidable.

Exemple de Problème non Décidable

- Problème P'
  - Soient $F_1, F_2, ..., F_n$ une liste de formes géométriques composées chacune de carrés juxtaposés. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Il y a beaucoup moins de "listes de formes géométriques composées chacune de carrés juxtaposés" que de "listes de formes géométriques", le problème P' est donc plus simple que le problème P.
  - Il se pourrait donc que P', puisse être résolu par un algorithme.
  - Pourtant ce n'est pas le cas (la démonstration que le problème B était indécidable établit en fait que B' est indécidable)
- Problème P"
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver périodiquement le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Là, le sous-problème est plus simple et est décidable.

Exemple de Problème non Décidable

- Problème P'
  - Soient $F_1, F_2, ..., F_n$ une liste de formes géométriques composées chacune de carrés juxtaposés. Peut-on paver le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Il y a beaucoup moins de "listes de formes géométriques composées chacune de carrés juxtaposés" que de "listes de formes géométriques", le problème P' est donc plus simple que le problème P.
  - Il se pourrait donc que P', puisse être résolu par un algorithme.
  - Pourtant ce n'est pas le cas (la démonstration que le problème B était indécidable établit en fait que B' est indécidable)
- Problème P"
  - Soient $F_1, F_2, ..., F_n$ une liste de formes polygonales. Peut-on paver périodiquement le plan, sans recouvrement ni espace vide avec des exemplaires de $F_1, F_2, ..., F_n$?
  - Là, le sous-problème est plus simple et est décidable.

La suite

- Nous aurions pu dans ce qui suit utiliser un langage de programmation usuel pour formaliser la notion de procédure effective.
- La lourdeur et les contraintes imposées par ce choix, nous poussnent à aller voir d'autres formalismes pour nos reflexions.
- Il existe plusieurs façon de représenter et par la suite de montrer qu'un problème est « soluble »:
  - L'arithmétique
  - Les fonctions calculables
  - Les représentations sous forme de machines (automates, automates à piles, Machine de turing,.....)
  - ...
- Dans ce qui suit nous nous interesserons à deux formes en particuliers
  - Les Fonctions et les Machines de Turing

Plan

Fonctions

- On va s'interesser, maintenant, à une formalisation de la notion de procédure effective :
  - Les fonctions calculables sur les entiers naturels.
- Considérons les opérations de bases sur les nombres entiers, comme $\times, +, -, /, \ldots$ Il est clair qu'elles sont calculables.
- Les fonctions ci-dessus sont totales. On considérera aussi des fonctions partielles comme DIV ou MOINS.

Fonctions

- On va s'interesser, maintenant, à une formalisation de la notion de procédure effective :
  - Les fonctions calculables sur les entiers naturels.
- Considérons les opérations de bases sur les nombres entiers, comme $\times, +, -, /, \ldots$ Il est clair qu'elles sont calculables.
- Les fonctions ci-dessus sont totales. On considérera aussi des fonctions partielles comme DIV ou MOINS.

Fonctions

- On va s'interesser, maintenant, à une formalisation de la notion de procédure effective :
  - Les fonctions calculables sur les entiers naturels.
- Considérons les opérations de bases sur les nombres entiers, comme $\times, +, -, /, \ldots$ Il est clair qu'elles sont calculables.
- Les fonctions ci-dessus sont totales. On considérera aussi des fonctions partielles comme DIV ou MOINS.

Fonctions Calculables

- La décomposition de la fonction factorielle en terme de produits est donnée par :
  $n! = n \times (n-1) \times (n-2) \times ... \times 1$
- Malheureusement cette déomposition dépend de la valeur de n. Pourtant il est clair que cette fonction est calculable.

Fonctions Calculables

- La décomposition de la fonction factorielle en terme de produits est donnée par :
  $n! = n \times (n-1) \times (n-2) \times ... \times 1$
- Malheureusement cette déomposition dépend de la valeur de n. Pourtant il est clair que cette fonction est calculable.

Fonctions Calculables

- on peut par exemple connaître un algorithme pour +, - qui étant donné un encodage binaire de a et b calcule l'encodage binaire de $(a+b)$, $(a-b)$.
- on peut calculer les fonctions qui sont définissables à partir des fonctions de base. Par exemple, si on a un algorithme pour + et -, il est clair qu'on pourra calculer la fonction $a-(b+c)$par exemple.
- Mais il y a des fonctions qui ne peuvent pas être obtenues par composition simple de fonctions de base. Pensez à la fonction factorielle: $n!$.

Fonctions Calculables

- on peut par exemple connaître un algorithme pour +, - qui étant donné un encodage binaire de a et b calcule l'encodage binaire de $(a+b)$, $(a-b)$.
- on peut calculer les fonctions qui sont définissables à partir des fonctions de base. Par exemple, si on a un algorithme pour + et -, il est clair qu'on pourra calculer la fonction $a-(b+c)$par exemple.
- Mais il y a des fonctions qui ne peuvent pas être obtenues par composition simple de fonctions de base. Pensez à la fonction factorielle: $n!$.

Fonctions Calculables

- on peut par exemple connaître un algorithme pour +, - qui étant donné un encodage binaire de a et b calcule l'encodage binaire de $(a+b)$, $(a-b)$.
- on peut calculer les fonctions qui sont définissables à partir des fonctions de base. Par exemple, si on a un algorithme pour + et -, il est clair qu'on pourra calculer la fonction $a-(b+c)$par exemple.
- Mais il y a des fonctions qui ne peuvent pas être obtenues par composition simple de fonctions de base. Pensez à la fonction factorielle: $n!$.

Fonctions Calculables

- Pour pouvoir définir des compositions variables, nous utiliserons la notion de récursion.
- Voici une définition récursive de la fonction factorielle :
  $0! = 1$
  $(n+1)! = (n+1)! \times n!$
- Cette définition est effective car$f\,(n+1)$ est défini en terme de $f\,(n)$ et d'opérations que l'on sait calculables par ailleurs, et de plus $f\,(0)$ est trivialement calculable.

Fonctions Calculables

- Pour pouvoir définir des compositions variables, nous utiliserons la notion de récursion.
- Voici une définition récursive de la fonction factorielle :
  $0! = 1$
  $(n+1)! = (n+1)! \times n!$
- Cette définition est effective car$f\,(n+1)$ est défini en terme de $f\,(n)$ et d'opérations que l'on sait calculables par ailleurs, et de plus $f\,(0)$ est trivialement calculable.

Fonctions Calculables

- Pour pouvoir définir des compositions variables, nous utiliserons la notion de récursion.
- Voici une définition récursive de la fonction factorielle :
  $0! = 1$
  $(n+1)! = (n+1)! \times n!$
- Cette définition est effective car$f\,(n+1)$ est défini en terme de $f\,(n)$ et d'opérations que l'on sait calculables par ailleurs, et de plus $f\,(0)$ est trivialement calculable.

Premières définitions

Fonctions primitives récursives

Definition

Les fonctions primitives récursives sont le sous-ensemble des fonctions:

$$\left\{\mathbb{N}^k \to \mathbb{N} \mid k \geq 0\right\}$$

qui peuvent être définies à l'aide des fonctions primitives récursives de base, d'une règle de composition et d'une règle de récursion.

Premières définitions

fonctions de base

Definition

Les fonctions primitives récursives de base sont:

1. la fonction $0()$, elle n'a pas d'argument et renvoit la valeur 0;
2. les fonctions de projection $\pi(n_1, ..., n_k)$:
   La fonction π_i renvoit comme valeur le $i^{ème}$ argument parmi k;
3. la fonction successeur $\sigma(n)$ est définie par $\sigma(n) = n + 1$.

Décomposition
Définition

- Nous définissons maintenant de manière formelle la notion de composition:

Definition

Soit g une fonction à l arguments et$h_1, ..., h_l$ des fonctions à k arguments. Si nous dénotons $n_1, ..., n_k$ par $\bar{n}$, alors la composition de g et de $h_1, ..., h_l$ est la fonction $\mathbb{N}^\star \to \mathbb{N}$ définie par:
$f(\bar{n}) = g(h_1(\bar{n}), ..., h_l(\bar{n}))$

Récursion Primitive
Définition

- Nous définissons maintenant de manière formelle la notion de récursion primitive:

Definition

Soit g une fonction à k arguments et h une fonction à $k+2$ arguments. la fonction f à $k+1$ arguments telle que :

$$f(\bar{n}, 0) = g(\bar{n})$$

$$f(\bar{n}, m+1) = h(\bar{n}, m, f(\bar{n}, m))$$

est la fonction définie à partir de g et h par *récursion primitive*.

Fonctions primitives récursives
Définitions

Si les fonctions g et h qui sont utilisées pour définir une fonction f par récursion primitive sont calculables par une procédure effective, alors f est aussi calculable.

Definition

Les fonctions primitives récursives sont finalement définies comme suit :

- les fonctions primitives récursives de base;
- toutes les fonctions obtenues à partir des fonctions primitives de base par un nombre quelconque d'applications de la composition et de la récursion primitive.

Fonctions primitives récursives

Définitions

Si les fonctions g et h qui sont utilisées pour définir une fonction f par récursion primitive sont calculables par une procédure effective, alors f est aussi calculable.

Definition

Les fonctions primitives récursives sont finalement définies comme suit :

- les fonctions primitives récursives de base;
- toutes les fonctions obtenues à partir des fonctions primitives de base par un nombre quelconque d'applications de la composition et de la récursion primitive.

Fonctions primitives récursives

Exemples

Examples

Montrez que les fonctions suivantes sont primitives récursives

1. Plus
2. Produit
3. Puissance
4. Moins
5. Signe
6. ....

Prédicats
Définition

- Un prédicat d'arité k est une fonction de $\mathbb{N}^k$ dans $\{vrai, faux\}$ou $\{0, 1\}$.
- Pour ne pas introduire l'ensemble de valeurs $\{vrai, faux\}$, on peut également considérer un prédicat d'arité k comme un sous-ensemble de $\mathbb{N}^k$ (le sous-ensemble des éléments de $\mathbb{N}^k$ où le prédicat est vrai).

Example

nous pouvons définir le prédicat pair:

- $pair\ (n)$est vrai si et seulement si n est un nombre pair.
- Est-ce que ce prédicat est primitif récursif ?

Prédicats
Définition

- Un prédicat d'arité k est une fonction de $\mathbb{N}^k$ dans $\{vrai, faux\}$ou $\{0, 1\}$.
- Pour ne pas introduire l'ensemble de valeurs $\{vrai, faux\}$, on peut également considérer un prédicat d'arité k comme un sous-ensemble de $\mathbb{N}^k$ (le sous-ensemble des éléments de $\mathbb{N}^k$ où le prédicat est vrai).

Example

nous pouvons définir le prédicat pair:

- $pair\,(n)$est vrai si et seulement si n est un nombre pair.
- Est-ce que ce prédicat est primitif récursif ?

Prédicats
Définition

- Pour donner une définition de prédicat primitif récursif, nous allons profiter du fait qu'un prédicat peut-être vu comme une fonction (sa fonction caractéristique) dans$\{0,1\}$: nous déduirons de notre notion de fonction primitive récursive celle de prédicat primitif récursif.
- Définissons formellement la notion de fonction caractéristique.

Definition

La fonction caractéristique d'un prédicat $P \subseteq \mathbb{N}^k$ est la fonction $\mathrm{f} : \mathbb{N}^k \to \{0,1\}$ telle que :

$$\mathrm{f}(\bar{\mathrm{n}}) = \begin{cases} 0 & si\ \bar{\mathrm{n}} \notin P \\ 1 & si\ \bar{\mathrm{n}} \in P \end{cases}$$

Corollary

Un prédicat est primitif récursif si et seulement si sa fonction caractéristique est primitive récursive

Prédicats
Définition

- Pour donner une définition de prédicat primitif récursif, nous allons profiter du fait qu'un prédicat peut-être vu comme une fonction (sa fonction caractéristique) dans$\{0,1\}$: nous déduirons de notre notion de fonction primitive récursive celle de prédicat primitif récursif.
- Définissons formellement la notion de fonction caractéristique.

Definition

La fonction caractéristique d'un prédicat $P \subseteq \mathbb{N}^k$ est la fonction $\mathrm{f} : \mathbb{N}^k \to \{0,1\}$ telle que :

$$\mathrm{f}(\bar{\mathrm{n}}) = \begin{cases} 0 & si\ \bar{\mathrm{n}} \notin P \\ 1 & si\ \bar{\mathrm{n}} \in P \end{cases}$$

Corollary

Un prédicat est primitif récursif si et seulement si sa fonction caractéristique est primitive récursive

Prédicats
Définition

- Pour donner une définition de prédicat primitif récursif, nous allons profiter du fait qu'un prédicat peut-être vu comme une fonction (sa fonction caractéristique) dans$\{0,1\}$: nous déduirons de notre notion de fonction primitive récursive celle de prédicat primitif récursif.
- Définissons formellement la notion de fonction caractéristique.

Definition

La fonction caractéristique d'un prédicat $P \subseteq \mathbb{N}^k$ est la fonction $\mathrm{f} : \mathbb{N}^k \to \{0,1\}$ telle que :

$$\mathrm{f}(\bar{\mathrm{n}}) = \begin{cases} 0 & si\ \bar{\mathrm{n}} \notin P \\ 1 & si\ \bar{\mathrm{n}} \in P \end{cases}$$

Corollary

Un prédicat est primitif récursif si et seulement si sa fonction caractéristique est primitive récursive

Prédicats
Exemples

Examples

1. Considérons le prédicat *zerop*, qui n'est vrai que pour la valeur 0. Sa définition est la suivante :

$$\begin{cases} zerop\,(0) = 1 \\ zerop\,(n+1) = 0 \end{cases}$$

2. On définit le prédicat plus petit que (n<m) de la façon suivante :

$$petit\,(n, m) = sg\,(m - n)$$

Ce prédicat est donc bien primitif récursif.

Prédicats
Exemples

Corollary

- *Les prédicats obtenus par combinaisons booléennes à partir de prédicats primitifs récursifs (g_1, g_2) sont également primitifs récursifs.*
- *Le prédicat égalité est également primitif récursif.*

Proof.

à faire en exercice □

Prédicats
Exemples

Corollary

- *Les prédicats obtenus par combinaisons booléennes à partir de prédicats primitifs récursifs (g_1, g_2) sont également primitifs récursifs.*
- *Le prédicat égalité est également primitif récursif.*

Proof.

à faire en exercice □

Fonctions primitives récursives

Prédicats - Exemples

Observons maintenant d'autres opérations logiques.

- La quantification universelle bornée permet d'exprimer qu'un prédicat est vrai pour toutes les valeurs inférieures à une certaine borne.

$$\forall i \leq m \; p\left(\bar{n}, i\right)$$

est vrai si $p\left(\bar{n}, i\right)$ est vrai pour tout $i \leq m$.

- La quantification existentielle bornée permet d'exprimer qu'un prédicat est vrai pour au moins une valeur inférieure à une certaine borne.

$$\exists i \leq m \; p\left(\bar{n}, i\right)$$

est vrai si $p\left(\bar{n}, i\right)$ est vrai pour au moins un $i \leq m$

Autres schémas de construction

Les Cas

Soient des prédicats primitifs récursifs $P_1, ..., P_n$ d'arité p mutuellement exclusifs, (i.e., pour tous $i \neq j$, $\delta P_i \cap \delta P_j = \emptyset$) et $f_1, ..., f_n$; $f_{n+1} \in \mathcal{F}_p$des fonctions PR . L a fonction $g \in \mathcal{F}_p$ péfinie par:

$$g(x_1, ..., x_p) = \begin{cases} f_1(x_1, ..., x_p) & si\ P_1(x_1, ..., x_p) \\ f_2(x_1, ..., x_p) & si\ P_2(x_1, ..., x_p) \\ ... & \\ f_n(x_1, ..., x_p) & si\ P_n(x_1, ..., x_p) \\ f_{n+1}(x_1, ..., x_p) & sinon \end{cases}$$

g est primitive récursive.

Autres schémas de construction

La Minimisation bornée

Soit A une partie de $\mathbb{N}^{p+1}$. La fonction $f \in \digamma_{p+1}$ est définie par:

$f(x_1, ..., x_p, n) = 0$ s'il n'existe aucun $t \leq n$ tel que $(x_1, ..., x_p, t) \in A$.

Dans le cas contraire, elle est définie par:

$$f(x_1, ..., x_p, n) = inf(t \leq n \mid (x_1, ..., x_p, t) \in A)$$

On note cette fonction ion comme suit :

$$f(x_1, ..., x_p, n) = \mu_{t \leq n}((x_1, ..., x_p, t) \in A)$$

Fonctions primitives récursives

Limites

- On vient de voir que les fonctions primitives récursives permettent de définir les fonctions et les prédicats les plus usuels. On pourrait maintenant se demander si les fonctions primitives récursives couvrent bien la notion intuitive que nous avons de fonctions calculables?
- On va malheureusement montrer que ce n'est pas le cas!
- On va montrer qu'il existe des fonctions qui sont intuitivement calculables et qui ne sont pas primitives récursives.

Fonctions primitives récursives

Limites

- On vient de voir que les fonctions primitives récursives permettent de définir les fonctions et les prédicats les plus usuels. On pourrait maintenant se demander si les fonctions primitives récursives couvrent bien la notion intuitive que nous avons de fonctions calculables?
- On va malheureusement montrer que ce n'est pas le cas!
- On va montrer qu'il existe des fonctions qui sont intuitivement calculables et qui ne sont pas primitives récursives.

Fonctions primitives récursives

Limites

- On vient de voir que les fonctions primitives récursives permettent de définir les fonctions et les prédicats les plus usuels. On pourrait maintenant se demander si les fonctions primitives récursives couvrent bien la notion intuitive que nous avons de fonctions calculables?
- On va malheureusement montrer que ce n'est pas le cas!
- On va montrer qu'il existe des fonctions qui sont intuitivement calculables et qui ne sont pas primitives récursives.

Fonctions primitives récursives

Limites

- Il est évident qu'il existe des fonctions qui ne sont pas primitives récursives.
- En effet, les fonctions primitives récursives sont énumérables (on peut énumérer les chaînes de caractères qui les représentent) il y a en a donc un nombre dénombrable, or il y a e un nombre non dénombrable de fonctions.
- Heureusement, il existe également des fonctions qui sont calculables mais pas primitives récursives.

Fonctions primitives récursives

Limites

Theorem

Il existe des fonctions calculables qui ne sont pas primitives récursives.

Proof.

Notons tout d'abord que l'on peut effectivement énumérer les fonctions primitives récursives. Soit $f_1, f_2, ..., f_n$; une telle énumération.
Construisons à partir de cette énumération une fonction calculable F qui n'est pas primitive récursive. Nous définissons F de la façon suivante :

$$F(n) = f_n(n) + 1$$

Par construction la fonction F est différente de toutes les fonctions primitives récursives. Montrons maintenant qu'elle est calculable. □

Fonctions primitives récursives

Limites

Proof.

(Suite)
Pour cela, on doit établir une méthode de calcul qui soit valable pour calculer F (n) pour n'importe quel n.
La méthode pour calculer f_n est la suivante :

1. énumérer les fonctions primitives récursives jusque la fonction f_n;
2. évaluer $f_n(n)$;
3. ajouter 1 à la valeur obtenue à l'étape 2.

Il est important de noter que la démonstration est valide car les fonctions primitives récursives sont effectivement énumérables et de plus l'évaluation d'une fonction primitive récursive termine toujours.

□

Fonctions Récursives

- Pour définir les fonctions récursives, on introduit le schéma de minimisation (non bornée).
- Soit k et r deux entiers et soit f une fonction de N^{k+1} dans N^r éventuellement partielle. On définit alors la fonction $g = Min(f)$ de N^k dans N de la manière suivante.
- Pour m dans N^r, $g(m)$ est le plus petit entier n, s'il existe, tel que $f(n, m) = d_r(0)$ (on rappelle que $d_r(0)$ est $(0, 0, ...0)$ où 0 est répété r fois). Si un tel entier n n'existe pas, $g(m)$ n'est pas définie.
- Même si la fonction f est totale, la fonction $g = Min(f)$ peut être partielle.
- La famille des fonctions récursives est la plus petite famille de fonctions qui vérifie les conditions suivantes:
  1. Les fonctions primitives récursives sont récursives.
  2. La famille des fonctions récursives est close pour les schémas de composition, de récurrence et de minimisation.

Plan

Machine

- Notion de Machine : *entrées* $\rightarrow$ **Machine** $\rightarrow$ *sorties*
- Formalismes:
  - Automates:
    - – Mémoire bornée (états de l'automate).
    - Remarque : Si on autorise un nombre infini (dénombrable) d'états, on peut reconnaître n'importe quel langage avec un automate. Le problème est que l'automate n'admet plus une description finie.
  - Automates à piles:
    - ++ Mémoire : états + pile : non bornée mais toujours finie durant un calcul.
    - – On ne sait pas reconnaître $\{a^n b^n | n \in \mathbb{N}\}$

Ce sont des classes de machines obéissant aux mêmes règles. Des machines d'une même classe différent par leur programme.

Machine

- Notion de Machine : *entrées* $\rightarrow$ **Machine** $\rightarrow$ *sorties*
- Formalismes:
  - Automates:
    - – Mémoire bornée (états de l'automate).
    - Remarque : Si on autorise un nombre infini (dénombrable) d'états, on peut reconnaître n'importe quel langage avec un automate. Le problème est que l'automate n'admet plus une description finie.
  - Automates à piles:
    - $++$ Mémoire : états $+$ pile : non bornée mais toujours finie durant un calcul.
    - – On ne sait pas reconnaître $\{a^n b^n | n \in \mathbb{N}\}$

Ce sont des classes de machines obéissant aux mêmes règles. Des machines d'une même classe différent par leur programme.

Machine de Turing

Definition

Machine de Turing

- Nombre fini d'états
- Mémoire auxiliaire : bande infinie avec une tête de lecture écriture.
- On peut
  - lire la case courante,
  - écrire dans la case courante,
  - déplacer la tête d'une case vers la gauche ou vers la droite.

Configuration

- Pour définir la notion d'exécution d'une machine de Turing ainsi que la notion de mot accepté par une machine de Turing, nous avons besoin de la notion de configuration.
- Une configuration contient toute l'information nécessaire à la poursuite de l'exécution de la machine, c'est-à-dire:
  1. l'état dans lequel la machine se trouve,
  2. le contenu du ruban,
  3. la position de la tête de lecture sur le ruban.

Configuration

- Pour définir la notion d'exécution d'une machine de Turing ainsi que la notion de mot accepté par une machine de Turing, nous avons besoin de la notion de configuration.
- Une configuration contient toute l'information nécessaire à la poursuite de l'exécution de la machine, c'est-à-dire:
  1. l'état dans lequel la machine se trouve,
  2. le contenu du ruban,
  3. la position de la tête de lecture sur le ruban.

Configuration

- Pour définir la notion d'exécution d'une machine de Turing ainsi que la notion de mot accepté par une machine de Turing, nous avons besoin de la notion de configuration.
- Une configuration contient toute l'information nécessaire à la poursuite de l'exécution de la machine, c'est-à-dire:
  1. l'état dans lequel la machine se trouve,
  2. le contenu du ruban,
  3. la position de la tête de lecture sur le ruban.

Configuration

- Pour définir la notion d'exécution d'une machine de Turing ainsi que la notion de mot accepté par une machine de Turing, nous avons besoin de la notion de configuration.
- Une configuration contient toute l'information nécessaire à la poursuite de l'exécution de la machine, c'est-à-dire:
  1. l'état dans lequel la machine se trouve,
  2. le contenu du ruban,
  3. la position de la tête de lecture sur le ruban.

Ruban

- La seule information "difficile" à représenter est *le ruban*
- c'est une suite infinie de symboles.
- Mais notons tout de même, cela nous sera utile dans la suite, qu'à tout moment seul un nombre fini de symboles sur le ruban sont différents du symbole #.
- Il importe donc seulement de garder le prefixe fini de symboles qui sont différents de #.

Ruban

- La seule information "difficile" à représenter est *le ruban*
- c'est une suite infinie de symboles.
- Mais notons tout de même, cela nous sera utile dans la suite, qu'à tout moment seul un nombre fini de symboles sur le ruban sont différents du symbole #.
- Il importe donc seulement de garder le prefixe fini de symboles qui sont différents de #.

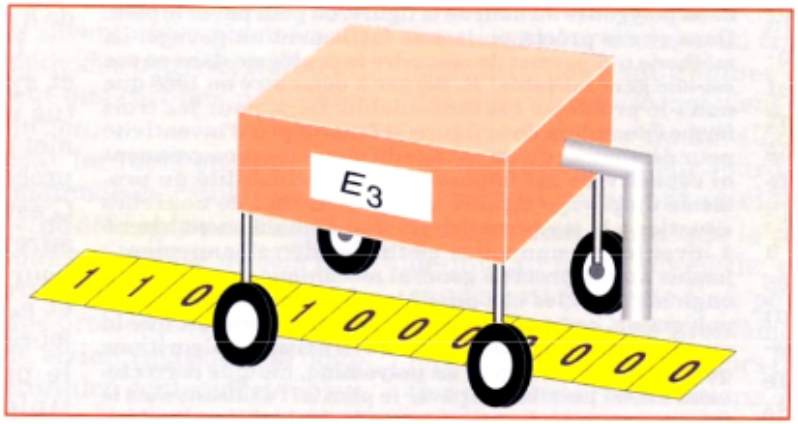
Ruban

- La seule information "difficile" à représenter est *le ruban*
- c'est une suite infinie de symboles.
- Mais notons tout de même, cela nous sera utile dans la suite, qu'à tout moment seul un nombre fini de symboles sur le ruban sont différents du symbole #.
- Il importe donc seulement de garder le prefixe fini de symboles qui sont différents de #.

Ruban

- La seule information "difficile" à représenter est *le ruban*
- c'est une suite infinie de symboles.
- Mais notons tout de même, cela nous sera utile dans la suite, qu'à tout moment seul un nombre fini de symboles sur le ruban sont différents du symbole #.
- Il importe donc seulement de garder le prefixe fini de symboles qui sont différents de #.

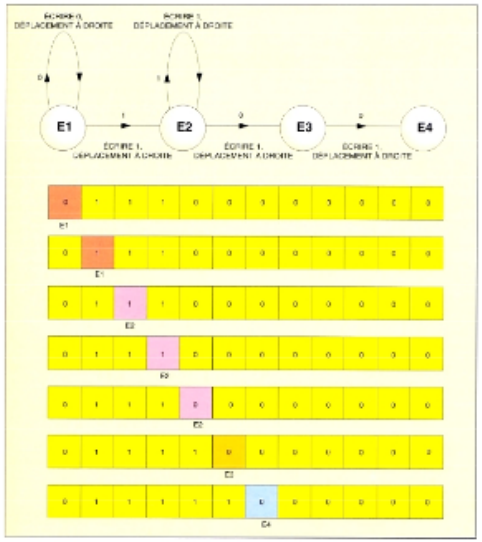
Illustration (1)



UNE MACHINE DE TURING est un mécanisme qui possède un nombre fini d'états intérieurs et qui, selon l'état où il se trouve et selon ce qu'il lit sur le ruban, efface la case du ruban qui est sous sa tête de lecture-écriture, y écrit un symbole et se déplace vers la gauche ou vers la droite. Le programme de la machine est une suite finie d'instructions du type : «Si je suis dans l'état E_3 et si je lis un 0 sur le ruban, alors je le remplace par un 1, je me déplace d'une case vers la droite et je passe dans l'état E_2 ; en abrégé, on note une telle instruction ($E_3\ 0 \rightarrow E_2\ 1\ D$). Tout calcul peut être exécuté par une machine de Turing.

Figure: Illustration Machine de Turing

Illustration (2)



LA MACHINE DE TURING qui calcule la fonction $f(n) = n + 2$ peut être représentée par un graphe *(en haut)* qui résume la liste des instructions. Chaque instruction, déterminée par un état et par une valeur lue sur la bande, est représentée par une flèche joignant l'état de départ à l'état d'arrivée, avec des indications d'écriture et de déplacement. En bas, on a indiqué le détail des états successifs de la machine et de son ruban pour la donnée initiale $n = 3$. Partie de l'état E_1, la machine passe dans l'état E_2 dès qu'elle rencontre un 1. Puis elle parcourt les n cases portant des 1, en restant dans l'état E_2 et, dès qu'elle trouve un 0, elle le remplace par un 1, passe dans l'état E_3, remplace encore le 0 suivant par un 1, passe dans l'état E_4 et s'arrête. Le bilan de ce travail est que deux 1 supplémentaires ont été ajoutés. Les n symboles 1 sont devenus $n + 2$ symboles 1.

Figure: Illustration Machine de Turing

Illustration (3)

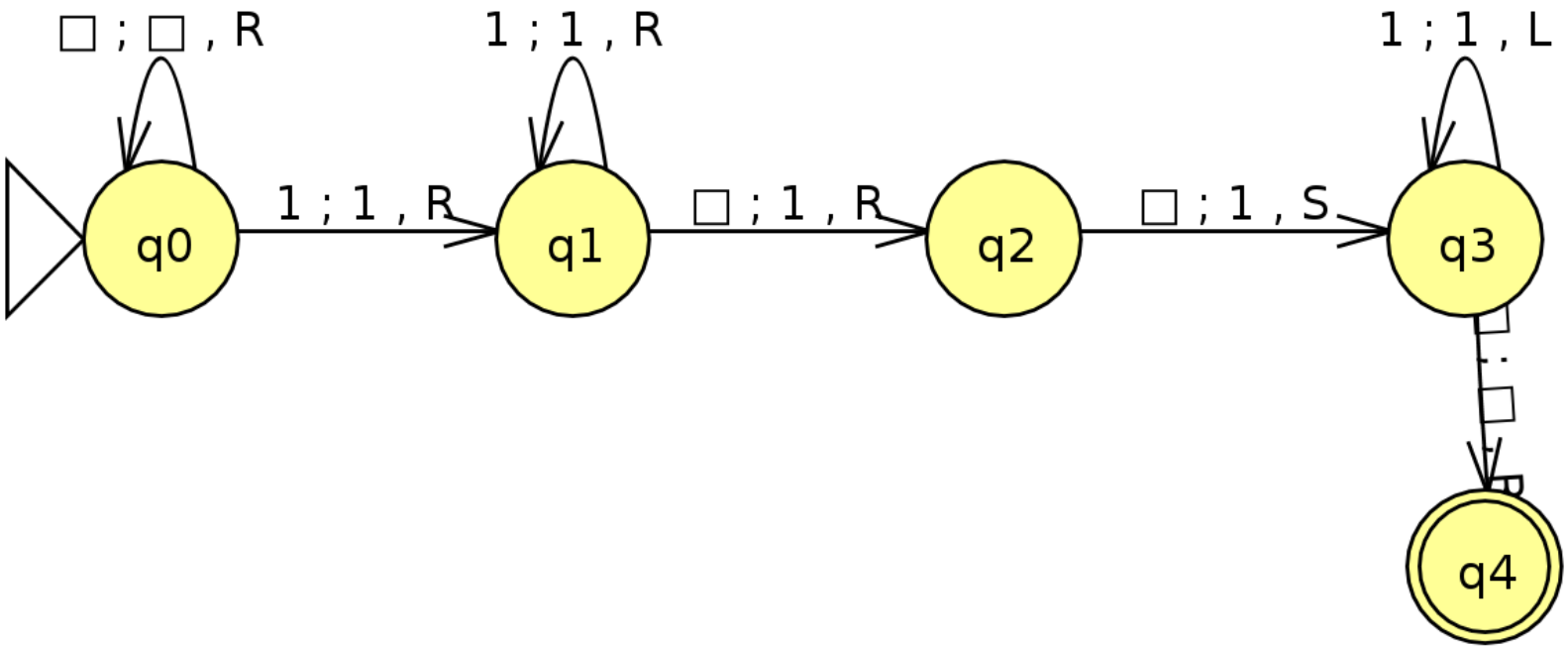


Figure: Illustration Machine de Turing

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc $\#$.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in \mathcal{Q}$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- $\#$ est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc $\#$.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in \mathcal{Q}$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- $\#$ est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc $\#$.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in \mathcal{Q}$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- $\#$ est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet $(Q, \Sigma, \Gamma, E, q_0, F, \#)$ où

- $Q = \{q_0, ..., q_n\}$ est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc #.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in \mathcal{Q}$ est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- # est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc #.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in \mathcal{Q}$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- # est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc #.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in Q$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- # est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc #.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in Q$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- # est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Machine de Turing

Une machine de Turing M est un septuplet$(Q, \Sigma, \Gamma, E, q_0, F, \#)$où

- $Q = \{q_0, ..., q_n\}$est un ensemble fini d'états de contrôle.
- Σ est l'alphabet d'entrée. C'est un ensemble fini de symboles qui ne contient pas le symbole blanc #.
- Γ est l'alphabet de bande. C'est un ensemble fini qui comprend tous les symboles qui peuvent être écrits sur la bande.
- E est un ensemble fini de transitions de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) peu être aussi notée $p, a \rightarrow q, b, x$
- $q_0 \in Q$est l'état initial dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des états finaux appelés aussi états d'acceptation.
- # est le symbole blanc qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Configuration

- Plus formellement, nous dénoterons une configuration par un triplet (q, w_1, w_2) où :
  1. $q \in Q$ est l'état de la machine;
  2. $w_1 \in \Gamma^*$est le mot apparaissant sur le ruban avant (strictement) la position de la tête de lecture;
  3. $w_2 \in \{\epsilon\} \cup \Gamma^*\{\Gamma \backslash \#\}$, le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

Configuration

- Plus formellement, nous dénoterons une configuration par un triplet (q, w_1, w_2) où :
  1. $q \in Q$ est l'état de la machine;
  2. $w_1 \in \Gamma^*$est le mot apparaissant sur le ruban avant (strictement) la position de la tête de lecture;
  3. $w_2 \in \{\epsilon\} \cup \Gamma^*\{\Gamma \backslash \#\}$, le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

Configuration

- Plus formellement, nous dénoterons une configuration par un triplet (q, w_1, w_2) où :
  1. $q \in Q$ est l'état de la machine;
  2. $w_1 \in \Gamma^*$est le mot apparaissant sur le ruban avant (strictement) la position de la tête de lecture;
  3. $w_2 \in \{\epsilon\} \cup \Gamma^*\{\Gamma \backslash \#\}$, le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

Configuration

- Plus formellement, nous dénoterons une configuration par un triplet (q, w_1, w_2) où :
  1. $q \in Q$ est l'état de la machine;
  2. $w_1 \in \Gamma^*$est le mot apparaissant sur le ruban avant (strictement) la position de la tête de lecture;
  3. $w_2 \in \{\epsilon\} \cup \Gamma^*\{\Gamma \backslash \#\}$, le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

Dérivation

- Nous allons maintenant définir formellement l'évolution de la configuration d'une machine en définissant une fonction de transition entre les configurations.
- Soit une configuration (q, α_1, α_2). Ecrivons cette configuration sous la forme $\left(q, \alpha_1, b\alpha_2^{'}\right)$ en-prenant $b = \#$ si $\alpha_2 = \epsilon$.
- Les configurations atteignables à partir de (q, α_1, α_2) dans la machine $\mathcal{M}$ sont alors définies comme suit:
  - si $\delta(q, b) = (q', b', R)$
    - nous avons alors: $\left(q, \alpha_1, \alpha_2^{'}\right) \vdash_M \left(q', \alpha_1 b', \alpha_2^{'}\right)$
  - si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \epsilon$ et est donc de la forme $\alpha' a$,
    - nous avons alors: $\left(q, \alpha_1^{'} a, b\alpha_2^{'}\right) \vdash_M \left(q', \alpha_1^{'}, ab^{'}\alpha_2^{'}\right)$

Dérivation

- Nous allons maintenant définir formellement l'évolution de la configuration d'une machine en définissant une fonction de transition entre les configurations.
- Soit une configuration (q, α_1, α_2). Ecrivons cette configuration sous la forme $\left(q, \alpha_1, b\alpha_2^{'}\right)$ en-prenant $b = \#$ si $\alpha_2 = \epsilon$.
- Les configurations atteignables à partir de (q, α_1, α_2) dans la machine $\mathcal{M}$ sont alors définies comme suit:
  - si $\delta(q, b) = (q', b', R)$
    - nous avons alors: $\left(q, \alpha_1, \alpha_2^{'}\right) \vdash_M \left(q', \alpha_1 b', \alpha_2^{'}\right)$
  - si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \epsilon$ et est donc de la forme $\alpha' a$,
    - nous avons alors: $\left(q, \alpha_1^{'} a, b\alpha_2^{'}\right) \vdash_M \left(q', \alpha_1^{'}, ab^{'}\alpha_2^{'}\right)$

Dérivation

- Nous allons maintenant définir formellement l'évolution de la configuration d'une machine en définissant une fonction de transition entre les configurations.
- Soit une configuration (q, α_1, α_2). Ecrivons cette configuration sous la forme $\left(q, \alpha_1, b\alpha_2^{'}\right)$ en-prenant $b = \#$ si $\alpha_2 = \epsilon$.
- Les configurations atteignables à partir de (q, α_1, α_2) dans la machine $\mathcal{M}$ sont alors définies comme suit:
  - si $\delta(q, b) = (q', b', R)$
    - nous avons alors: $\left(q, \alpha_1, \alpha_2^{'}\right) \vdash_M \left(q', \alpha_1 b', \alpha_2^{'}\right)$
  - si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \epsilon$ et est donc de la forme $\alpha' a$,
    - nous avons alors: $\left(q, \alpha_1^{'} a, b\alpha_2^{'}\right) \vdash_M \left(q', \alpha_1^{'}, ab^{'}\alpha_2^{'}\right)$

Dérivation

- Nous allons maintenant définir formellement l'évolution de la configuration d'une machine en définissant une fonction de transition entre les configurations.
- Soit une configuration (q, α_1, α_2). Ecrivons cette configuration sous la forme $\left(q, \alpha_1, b\alpha_2^{'}\right)$ en-prenant $b = \#$ si $\alpha_2 = \epsilon$.
- Les configurations atteignables à partir de (q, α_1, α_2) dans la machine $\mathcal{M}$ sont alors définies comme suit:
  - si $\delta(q, b) = (q', b', R)$
    - nous avons alors: $\left(q, \alpha_1, \alpha_2^{'}\right) \vdash_M \left(q', \alpha_1 b', \alpha_2^{'}\right)$
  - si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \epsilon$ et est donc de la forme $\alpha' a$,
    - nous avons alors: $\left(q, \alpha_1^{'} a, b\alpha_2^{'}\right) \vdash_M \left(q', \alpha_1^{'}, ab^{'}\alpha_2^{'}\right)$

Dérivation multiples

- Nous pouvons maintenant définir la notion de dérivation en plusieurs étapes.
- Une configuration C' est dérivable en plusieurs étapes de la configuration C avec la machine M, sera notée $C \vdash_M^* C'$, si il existe $k \geq 0$ et M des configurations intermédiaires $C_0, C_1, ..., C_k$
  - telles que :
    - $C_0 = C$;
    - $C' = C_k$;
    - $\forall i, 0 \leq i \leq k \ : \ C_i \vdash_M C_{i+1}$.

Dérivation multiples

- Nous pouvons maintenant définir la notion de dérivation en plusieurs étapes.
- Une configuration C' est dérivable en plusieurs étapes de la configuration C avec la machine M, sera notée $C \vdash_M^* C'$, si il existe $k \geq 0$ et M des configurations intermédiaires $C_0, C_1, ..., C_k$
  - telles que :
    - $C_0 = C$;
    - $C' = C_k$;
    - $\forall i, 0 \leq i \leq k \ : \ C_i \vdash_M C_{i+1}$.

Dérivation multiples

- Nous pouvons maintenant définir la notion de dérivation en plusieurs étapes.
- Une configuration C' est dérivable en plusieurs étapes de la configuration C avec la machine M, sera notée $C \vdash_M^* C'$, si il existe $k \geq 0$ et M des configurations intermédiaires $C_0, C_1, ..., C_k$
  - telles que :
    - $C_0 = C$;
    - $C' = C_k$;
    - $\forall i, 0 \leq i \leq k \ : \quad C_i \vdash_M C_{i+1}$.

Execution

À partir d'une configuration (s, ϵ, w), plusieurs cas peuvent se présenter :

- l'exécution de la machine atteint un état accepteur et la machine accepte le mot;
- l'exécution de la machine s'arrête après un temps fini soit parce qu'aucune transition n'est possible ou bien l'exécution tente de faire bouger la tête de lecture vers la gauche alors que la tête pointe sur la première case du ruban. Dans ces deux cas le mot est rejeté par la machine;
- l'exécution de la machine est infinie et donc ne termine jamais. Le problème est qu'à aucun moment on ne peut être sûr que l'on peut arrêter l'exécution et que le mot sera rejeté.

Execution

À partir d'une configuration (s, ϵ, w), plusieurs cas peuvent se présenter :

- l'exécution de la machine atteint un état accepteur et la machine accepte le mot;
- l'exécution de la machine s'arrête après un temps fini soit parce qu'aucune transition n'est possible ou bien l'exécution tente de faire bouger la tête de lecture vers la gauche alors que la tête pointe sur la première case du ruban. Dans ces deux cas le mot est rejeté par la machine;
- l'exécution de la machine est infinie et donc ne termine jamais. Le problème est qu'à aucun moment on ne peut être sûr que l'on peut arrêter l'exécution et que le mot sera rejeté.

Execution

À partir d'une configuration (s, ϵ, w), plusieurs cas peuvent se présenter :

- l'exécution de la machine atteint un état accepteur et la machine accepte le mot;
- l'exécution de la machine s'arrête après un temps fini soit parce qu'aucune transition n'est possible ou bien l'exécution tente de faire bouger la tête de lecture vers la gauche alors que la tête pointe sur la première case du ruban. Dans ces deux cas le mot est rejeté par la machine;
- l'exécution de la machine est infinie et donc ne termine jamais. Le problème est qu'à aucun moment on ne peut être sûr que l'on peut arrêter l'exécution et que le mot sera rejeté.

Execution

À partir d'une configuration (s, ϵ, w), plusieurs cas peuvent se présenter :

- l'exécution de la machine atteint un état accepteur et la machine accepte le mot;
- l'exécution de la machine s'arrête après un temps fini soit parce qu'aucune transition n'est possible ou bien l'exécution tente de faire bouger la tête de lecture vers la gauche alors que la tête pointe sur la première case du ruban. Dans ces deux cas le mot est rejeté par la machine;
- l'exécution de la machine est infinie et donc ne termine jamais. Le problème est qu'à aucun moment on ne peut être sûr que l'on peut arrêter l'exécution et que le mot sera rejeté.

Acceptation

- Le langage $L(M)$ accepté par une machine de Turing est l'ensemble des mots w tels que
  - $(s, \epsilon, w) \vdash_M^* (q, \alpha_1, \alpha_2)$avec $q \in F$.
- Ce sont donc les mots sur lesquels l'exécution de la machine atteint un état accepteur.

Décidabilité

- L est récursivement énumérable RE (ou semi-décidable) s'il existe une Machine de Turing M telle que $L = L(M)$.
- L est récursif R (ou décidable, calculable) s'il existe une MT sans exécution infinie qui l'accepte.

Corollary

$$R \subseteq RE$$

Décidabilité

- L est récursivement énumérable RE (ou semi-décidable) s'il existe une Machine de Turing M telle que $L = L(M)$.
- L est récursif R (ou décidable, calculable) s'il existe une MT sans exécution infinie qui l'accepte.

Corollary

$$R \subseteq RE$$

Décidabilité

- L est récursivement énumérable RE (ou semi-décidable) s'il existe une Machine de Turing M telle que $L = L(M)$.
- L est récursif R (ou décidable, calculable) s'il existe une MT sans exécution infinie qui l'accepte.

Corollary

$$R \subseteq RE$$

Décidabilité

- L est récursivement énumérable RE (ou semi-décidable) s'il existe une Machine de Turing M telle que $L = L(M)$.
- L est récursif R (ou décidable, calculable) s'il existe une MT sans exécution infinie qui l'accepte.

Corollary

$$R \subseteq RE$$

Ma Premiére Machine de Turing (1)

Machine de Turing M $=(Q, \Sigma, \Gamma, E, q_0, F, \#)$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- q_0 état initial,
- E est représenté par:

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	(q_1, a, L)	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Ma Premiére Machine de Turing (1)

Machine de Turing M $=(Q, \Sigma, \Gamma, E, q_0, F, \#)$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- q_0 état initial,
- E est représenté par:

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	(q_1, a, L)	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Ma Premiére Machine de Turing (1)

Machine de Turing M $=(Q, \Sigma, \Gamma, E, q_0, F, \#)$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- q_0 état initial,
- E est représenté par:

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	(q_1, a, L)	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Ma Premiére Machine de Turing (1)

Machine de Turing M $=(Q, \Sigma, \Gamma, E, q_0, F, \#)$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- q_0 état initial,
- E est représenté par:

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	(q_1, a, L)	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Ma Premiére Machine de Turing (1)

Machine de Turing M $=(Q, \Sigma, \Gamma, E, q_0, F, \#)$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- q_0 état initial,
- E est représenté par:

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	(q_1, a, L)	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Ma Premiére Machine de Turing (2)

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	(q_1, a, L)	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Example

1. Définir F.
2. Déssiner M.
3. Identifier $L(M)$.

Variantes de Machine de Turing

ruban doublement infini

- **Que se passe-t-il si on considère une machine avec un ruban infini vers la droite mais également vers la gauche ?**
- La définition d'une telle machine est identique celle d'une machine de Turing habituelle à l'exception du fait suivant: la tête de lecture peut toujours se déplacer vers la gauche.
- On peut montrer que toute machine à ruban doublement infini peut être "simulée" par une machine avec un ruban simplement infini.

Corollary

- *Soit* $M = (Q, \Sigma, \Gamma, E, q0, F, \#)$ *une MT à bande infinie à droite uniquement. On peut effectivement construire une MT M équivalente à bande bi-infinie.*
- *Soit* $M = (Q, \Sigma, \Gamma, E, q0, F, \#)$ *une MT à bande bi-infinie. On peut effectivement construire une MT M équivalente à bande à droite uniquement.*

Variantes de Machine de Turing

ruban doublement infini

- **Que se passe-t-il si on considère une machine avec un ruban infini vers la droite mais également vers la gauche ?**
- La définition d'une telle machine est identique celle d'une machine de Turing habituelle à l'exception du fait suivant: la tête de lecture peut toujours se déplacer vers la gauche.
- On peut montrer que toute machine à ruban doublement infini peut être "simulée" par une machine avec un ruban simplement infini.

Corollary

- *Soit* $M = (Q, \Sigma, \Gamma, E, q0, F, \#)$ *une MT à bande infinie à droite uniquement. On peut effectivement construire une MT M équivalente à bande bi-infinie.*
- *Soit* $M = (Q, \Sigma, \Gamma, E, q0, F, \#)$ *une MT à bande bi-infinie. On peut effectivement construire une MT M équivalente à bande à droite uniquement.*

Variantes de Machine de Turing

ruban doublement infini

- **Que se passe-t-il si on considère une machine avec un ruban infini vers la droite mais également vers la gauche ?**
- La définition d'une telle machine est identique celle d'une machine de Turing habituelle à l'exception du fait suivant: la tête de lecture peut toujours se déplacer vers la gauche.
- On peut montrer que toute machine à ruban doublement infini peut être "simulée" par une machine avec un ruban simplement infini.

Corollary

- *Soit* $M = (Q, \Sigma, \Gamma, E, q0, F, \#)$ *une MT à bande infinie à droite uniquement. On peut effectivement construire une MT M équivalente à bande bi-infinie.*
- *Soit* $M = (Q, \Sigma, \Gamma, E, q0, F, \#)$ *une MT à bande bi-infinie. On peut effectivement construire une MT M équivalente à bande à droite uniquement.*

Variantes de Machine de Turing

Machine multi-ruban

- Considérons une machine avec plusieurs rubans et plusieurs têtes de lecture.
- L'état d'une telle machine est constitué de la location courante, du contenu de chaque ruban et de la position de chaque tête de lecture.
- On peut à nouveau simuler le comportement a d'une telle machine avec un alphabet de symboles composés décrivant le contenu des différents rubans et la position des têtes de lecture.

Variantes de Machine de Turing

Machine multi-ruban

- Considérons une machine avec plusieurs rubans et plusieurs têtes de lecture.
- L'état d'une telle machine est constitué de la location courante, du contenu de chaque ruban et de la position de chaque tête de lecture.
- On peut à nouveau simuler le comportement a d'une telle machine avec un alphabet de symboles composés décrivant le contenu des différents rubans et la position des têtes de lecture.

Variantes de Machine de Turing

Machine multi-ruban

- Considérons une machine avec plusieurs rubans et plusieurs têtes de lecture.
- L'état d'une telle machine est constitué de la location courante, du contenu de chaque ruban et de la position de chaque tête de lecture.
- On peut à nouveau simuler le comportement a d'une telle machine avec un alphabet de symboles composés décrivant le contenu des différents rubans et la position des têtes de lecture.

Variantes Machine de Turing

Machine à accès direct

- Une machine RAM (random access memory) est une machine "semblable" aux ordinateurs réels.
- Supposons que la machine à simuler comporte une mémoire à accès direct et un à certain nombre de registres dont un compteur de programme.
- Une machine de Turing (multi-rubans et donc mono-ruban) peut simuler une machine RAM.

Variantes Machine de Turing

Machine à accès direct

- Une machine RAM (random access memory) est une machine "semblable" aux ordinateurs réels.
- Supposons que la machine à simuler comporte une mémoire à accès direct et un à certain nombre de registres dont un compteur de programme.
- Une machine de Turing (multi-rubans et donc mono-ruban) peut simuler une machine RAM.

Variantes Machine de Turing

Machine à accès direct

- Une machine RAM (random access memory) est une machine "semblable" aux ordinateurs réels.
- Supposons que la machine à simuler comporte une mémoire à accès direct et un à certain nombre de registres dont un compteur de programme.
- Une machine de Turing (multi-rubans et donc mono-ruban) peut simuler une machine RAM.

Variantes Machine de Turing

Machine Non-deterministe

- Jusqu'à présent nous nous sommes limités à des machines qui pour chacune de leurs configurations ont au plus une configuration atteignable.
- Que se passe-t-il si on considère une relation de transition à la place d'une fonction de transition ?

Theorem

Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Idée de preuve: montrer qu'une machine déterministe peut "simuler" une machine non-déterministe.

- La notion de "décidé" n'a pas de sens pour une machine non-déterministe.
- Comment peut-on montrer que le non-déterminisme n'apporte pas de pouvoir d'expression supplémentaire ?

Variantes Machine de Turing

Machine Non-deterministe

- Jusqu'à présent nous nous sommes limités à des machines qui pour chacune de leurs configurations ont au plus une configuration atteignable.
- Que se passe-t-il si on considère une relation de transition à la place d'une fonction de transition ?

Theorem

Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Idée de preuve: montrer qu'une machine déterministe peut "simuler" une machine non-déterministe.

- La notion de "décidé" n'a pas de sens pour une machine non-déterministe.
- Comment peut-on montrer que le non-déterminisme n'apporte pas de pouvoir d'expression supplémentaire ?

Variantes Machine de Turing

Machine Non-deterministe

- Jusqu'à présent nous nous sommes limités à des machines qui pour chacune de leurs configurations ont au plus une configuration atteignable.
- Que se passe-t-il si on considère une relation de transition à la place d'une fonction de transition ?

Theorem

Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Idée de preuve: montrer qu'une machine déterministe peut "simuler" une machine non-déterministe.

- La notion de "décidé" n'a pas de sens pour une machine non-déterministe.
- Comment peut-on montrer que le non-déterminisme n'apporte pas de pouvoir d'expression supplémentaire ?

Variantes Machine de Turing

Machine Non-deterministe

- Jusqu'à présent nous nous sommes limités à des machines qui pour chacune de leurs configurations ont au plus une configuration atteignable.
- Que se passe-t-il si on considère une relation de transition à la place d'une fonction de transition ?

Theorem

Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Idée de preuve: montrer qu'une machine déterministe peut "simuler" une machine non-déterministe.

- La notion de "décidé" n'a pas de sens pour une machine non-déterministe.
- Comment peut-on montrer que le non-déterminisme n'apporte pas de pouvoir d'expression supplémentaire ?

Variantes Machine de Turing

Machine Non-deterministe

- Jusqu'à présent nous nous sommes limités à des machines qui pour chacune de leurs configurations ont au plus une configuration atteignable.
- Que se passe-t-il si on considère une relation de transition à la place d'une fonction de transition ?

Theorem

Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Idée de preuve: montrer qu'une machine déterministe peut "simuler" une machine non-déterministe.

- La notion de "décidé" n'a pas de sens pour une machine non-déterministe.
- Comment peut-on montrer que le non-déterminisme n'apporte pas de pouvoir d'expression supplémentaire ?

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

Thèse de Church

Thèse de Church

Tout ce qui est décidé par une machine de Turing l'est également par une procédure effective.

Résultat

On peut interpréter (ou simuler) une machine de Turing sur un ordinateur.

- Par contre l'autre direction est plus délicate :
  - Tout langage reconnu par une procédure effective est-il décidé par une machine de Turing?
  - Tout probléme soluble algorithmiquement est-il soluble avec une machine de Turing?
- Une MT M définit une procédure effective pour décider si un mot $u \in L(M)$ si et seulement si M s'arrête toujours.

Machine de Turing et Procédure effective

- On peut réénoncé la thèse de Church-Turing en termes de calcul de fonctions.

Theorem

Les fonctions calculables par une procédure effective sont les fonctions calculables par une machine de Turing

Machine de Turing et Procédure effective

- On peut réénoncé la thèse de Church-Turing en termes de calcul de fonctions.

Theorem

Les fonctions calculables par une procédure effective sont les fonctions calculables par une machine de Turing

Exercices

- Expliquez comment on peut faire calculer une fonction de $\mathbb{N}$ dans $\mathbb{N}$ par une machine de Turing;
- Prouvez qu'il existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ qui ne sont pas calculables par une machine de Turing.

Exercices

- Expliquez comment on peut faire calculer une fonction de $\mathbb{N}$ dans $\mathbb{N}$ par une machine de Turing;
- Prouvez qu'il existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ qui ne sont pas calculables par une machine de Turing.

Plan

Classes de Décidabilité

Correspondance entre problème et langage de l'encodage des instances positives.

Definition

La classe de décidabilité R est l'ensemble des langages décidables par une machine de Turing.

La classe R est la classe des langages (problémes)

- décidés par machine de Turing,
- récursifs , décidables, calculables,
- solubles algorithmiquement.

Est-ce qu'il existe un langage $L \notin R$?

- oui (argument de dénombrement)
- Mais comment en exhiber un?

Classes de Décidabilité

Correspondance entre problème et langage de l'encodage des instances positives.

Definition

La classe de décidabilité R est l'ensemble des langages décidables par une machine de Turing.

La classe R est la classe des langages (problémes)

- décidés par machine de Turing,
- récursifs , décidables, calculables,
- solubles algorithmiquement.

Est-ce qu'il existe un langage $L \notin R$?

- oui (argument de dénombrement)
- Mais comment en exhiber un?

Classes de Décidabilité

Correspondance entre problème et langage de l'encodage des instances positives.

Definition

La classe de décidabilité R est l'ensemble des langages décidables par une machine de Turing.

La classe R est la classe des langages (problémes)

- décidés par machine de Turing,
- récursifs , décidables, calculables,
- solubles algorithmiquement.

Est-ce qu'il existe un langage $L \notin R$?

- oui (argument de dénombrement)
- Mais comment en exhiber un?

Classes de Décidabilité

Definition

La classe de décidabilité *RE* est l'ensemble des langages acceptés par une machine de Turing.

La classe RE est la classe des langages (problèmes)

- acceptés par machine de Turing,
- partiellement récursifs , partiellement décidables, partiellement calculables,
- partiellement solubles algorithmiquement,
- récursivement énumérables.

Classes de Décidabilité

Definition

La classe de décidabilité *RE* est l'ensemble des langages acceptés par une machine de Turing.

La classe RE est la classe des langages (problèmes)

- acceptés par machine de Turing,
- partiellement récursifs , partiellement décidables, partiellement calculables,
- partiellement solubles algorithmiquement,
- récursivement énumérables.

Indécidabilité

Premier langage indécidable

Example

Notations :

- L'ensemble des MT est dénombrable : M1 , M2 , . . .
- L'ensemble des mots (alphabet commun) : w1 , w2 , . . .

Le langage $L_0 = \{w \mid w = w_i \wedge M_i n' \textit{accepte pas } w_i\}$n'est pas dans la classe RE.

Corollary

1. *Le complément d'un langage de la classe R est un langage de la classe R.*
2. *Si un langage L et son complément $\bar{L}$ sont tous deux dans RE, alors à la fois L et $\bar{L}$ sont dans R.*

Indécidabilité

Il y a donc trois cas possibles :

1. $L \notin R$ et $\bar{L} \notin R$,
2. $L \notin RE$ et $\bar{L} \notin RE$,
3. $L \notin RE$ et $\bar{L} \in RE \cap \overline{R}$

Corollary

$L_0 = \{w \mid w = w_i \wedge M_i$ accepte $w_i\}$est dans la classe RE.

Theorem

Le langage $\overline{L_0}$ est indécidable n'appartient ni à R ni à RE.

Indécidabilité

Il y a donc trois cas possibles :

1. $L \notin R$ et $\bar{L} \notin R$,
2. $L \notin RE$ et $\bar{L} \notin RE$,
3. $L \notin RE$ et $\bar{L} \in RE \cap \overline{R}$

Corollary

$L_0 = \{w \mid w = w_i \wedge M_i$ accepte $w_i\}$est dans la classe RE.

Theorem

Le langage $\overline{L_0}$ est indécidable n'appartient ni à R ni à RE.

Indécidabilité

Il y a donc trois cas possibles :

1. $L \notin R$ et $\bar{L} \notin R$,
2. $L \notin RE$ et $\bar{L} \notin RE$,
3. $L \notin RE$ et $\bar{L} \in RE \cap \overline{R}$

Corollary

$L_0 = \{w \mid w = w_i \wedge M_i$ accepte $w_i\}$est dans la classe RE.

Theorem

Le langage $\overline{L_0}$ est indécidable n'appartient ni à R ni à RE.

Technique de la Réduction

Pour montrer que $L_2 \in R$, sachant que $L_1 \in R$,

1. On démontre que s'il existe un algorithme qui décide le langage L_2 , alors il existe aussi un algorithme qui décide le langage L_1. Cela se fait en donnant un algorithme (formellement une machine de Turing s'arrêtant toujours) qui décide le langage L_1 en se servant comme d'un sous-programme d'un algorithme décidant L_2.
2. On conclut que L_2 n'est pas décidable ($L_2 \in R$) car la réduction de L_1 vers L_2 montre que si L_2 était décidable L_1 le serait aussi, ce qui contredit l'hypothèse que L_1 est un langage indécidable.

Ce type d'algorithme est appelé une réduction de L_1 a L_2 . En effet, il réduit la décidabilité de L_1 à celle de L_2.

Technique de la Réduction

Pour montrer que $L_2 \in R$, sachant que $L_1 \in R$,

1. On démontre que s'il existe un algorithme qui décide le langage L_2 , alors il existe aussi un algorithme qui décide le langage L_1. Cela se fait en donnant un algorithme (formellement une machine de Turing s'arrêtant toujours) qui décide le langage L_1 en se servant comme d'un sous-programme d'un algorithme décidant L_2.
2. On conclut que L_2 n'est pas décidable ($L_2 \in R$) car la réduction de L_1 vers L_2 montre que si L_2 était décidable L_1 le serait aussi, ce qui contredit l'hypothèse que L_1 est un langage indécidable.

Ce type d'algorithme est appelé une réduction de L_1 a L_2 . En effet, il réduit la décidabilité de L_1 à celle de L_2.

Indécidabilité

Deuxième langage indécidable

Example

Le langage universel LU est indécidable

$$LU = \{< M, w > | \ M \ accepte \ w\}$$

Proof.

Réduction à partir du langage L_0 : □

1. Déterminer l'indice i tel que w = wi.
   1. Déterminer la machine de Turing M.
   2. Appliquer la procédure de décision pour LU au mot $< M_i, w_i >$, si le résultat est positif, w est accepté, sinon il est rejeté.

Fact

$\overline{LU} \notin RE$

Indécidabilité

Deuxième langage indécidable

Example

Le langage universel LU est indécidable

$$LU = \{< M, w > | \ M \ accepte \ w\}$$

Proof.

Réduction à partir du langage L_0 : □

1. Déterminer l'indice i tel que w = wi.
   1. Déterminer la machine de Turing M.
   2. Appliquer la procédure de décision pour LU au mot $< M_i, w_i >$, si le résultat est positif, w est accepté, sinon il est rejeté.

Fact

$\overline{LU} \notin RE$

Indécidabilité

Deuxième langage indécidable

Example

Le langage universel LU est indécidable

$$LU = \{< M, w > | \ M \ accepte \ w\}$$

Proof.

Réduction à partir du langage L_0 : □

1. Déterminer l'indice i tel que w = wi.
   1. Déterminer la machine de Turing M.
   2. Appliquer la procédure de décision pour LU au mot $< M_i, w_i >$, si le résultat est positif, w est accepté, sinon il est rejeté.

Fact

$\overline{LU} \notin RE$

Problèmes Indécidables

Le problème de l'arrêt :

$$H = \{< M, w > | \ M \ s'arrête \ sur \ w\}$$

est un problème indécidable.

Proof.

Réduction à partir de LU.

1. Appliquer l'algorithme décidant H.
   1. Si l'algorithme décidant H donne la réponse "non" (i.e. la machine M ne s'arrête pas), répondre "non" (dans ce cas, on a effectivement $< M, w > \notin LU$).
   2. Si l'algorithme décidant H donne la réponse "oui", simuler l'exécution de M sur w et donner la réponse obtenue (dans ce cas, l'exécution de M sur w est finie et l'on obtient donc toujours une réponse).

Problèmes Indécidables

Le problème de l'arrêt :

$$H = \{< M, w > | \ M \ s'arrête \ sur \ w\}$$

est un problème indécidable.

Proof.

Réduction à partir de LU.

1. Appliquer l'algorithme décidant H.
   1. Si l'algorithme décidant H donne la réponse "non" (i.e. la machine M ne s'arrête pas), répondre "non" (dans ce cas, on a effectivement $< M, w > \notin LU$).
   2. Si l'algorithme décidant H donne la réponse "oui", simuler l'exécution de M sur w et donner la réponse obtenue (dans ce cas, l'exécution de M sur w est finie et l'on obtient donc toujours une réponse).

Problèmes Indécidables

Le problème de déterminer si un programme écrit dans un langage de programmation usuel (par exemple le Java) s'arrête pour des valeurs fixées de ses données est indécidable.

Proof.

Réduction à partir du problème de l'arrêt pour les machines de Turing.

1. Construire un programme Java P qui, étant donné une machine de Turing M et un mot w, simule le comportement de M sur w.
2. Décider si le programme P s'arrête pour les données$< M, w >$et transmettre la réponse obtenue.

Problèmes Indécidables

Le problème de déterminer si un programme écrit dans un langage de programmation usuel (par exemple le Java) s'arrête pour des valeurs fixées de ses données est indécidable.

Proof.

Réduction à partir du problème de l'arrêt pour les machines de Turing.

1. Construire un programme Java P qui, étant donné une machine de Turing M et un mot w, simule le comportement de M sur w.
2. Décider si le programme P s'arrête pour les données$< M, w >$et transmettre la réponse obtenue.

□

Problèmes Indécidables

Examples

1. Le problème de déterminer si une machine de Turing s'arrête lorsque son mot d'entrée est le mot vide (problème de l'arrêt sur mot vide) est indécidable.
   - Réduction à partir du problème de l'arrêt.
2. Le problème de déterminer si une machine de Turing s'arrête pour au moins un mot d'entrée (problème de l'arrêt existentiel) est indécidable.
   - Réduction à partir du problème de l'arrêt sur mot vide.

Proof.

voir TD. □

Problèmes Indécidables

Examples

1. Le problème de déterminer si une machine de Turing s'arrête lorsque son mot d'entrée est le mot vide (problème de l'arrêt sur mot vide) est indécidable.
   - Réduction à partir du problème de l'arrêt.
2. Le problème de déterminer si une machine de Turing s'arrête pour au moins un mot d'entrée (problème de l'arrêt existentiel) est indécidable.
   - Réduction à partir du problème de l'arrêt sur mot vide.

Proof.

voir TD. □

Problèmes Indécidables

Examples

1. Déterminer si le langage accepté par une machine de Turing est récursif (langage accepté récursif) est indécidable.
   - Réduction à partir de LU.
2. Déterminer si le langage accepté par une machine de Turing est indécidable (langage accepté décidable) est indécidable.
   - Réduction à partir de $\overline{LU}$.
3. Déterminer si le langage accepté par une machine de Turing est vide (langage accepté vide) est indécidable.
   - Réduction à partir de $\overline{LU}$.

Proof.

à faire. □

Problèmes Indécidables

Examples

1. Déterminer si le langage accepté par une machine de Turing est récursif (langage accepté récursif) est indécidable.
   - Réduction à partir de LU.
2. Déterminer si le langage accepté par une machine de Turing est indécidable (langage accepté décidable) est indécidable.
   - Réduction à partir de $\overline{LU}$.
3. Déterminer si le langage accepté par une machine de Turing est vide (langage accepté vide) est indécidable.
   - Réduction à partir de $\overline{LU}$.

Proof.

à faire. □

Plan

Compléxité
Motivation

- Problème soluble – problème soluble efficacement.
- Mesure de la complexité : fonction de complexité.

Compléxité
Introduction

- Abstraction par rapport à la machine utilisée.
- Abstraction par rapport aux données (taille uniquement).
- Notation O.

Compléxité
Introduction

Definition

la compléxité d'une Machine de Turing est une fonction qui, à une taille d'entrée fait associer un nombre maximal de pas de calcul que la machine va faire avant de s'arreter.

- Chaque transition de la MT prend 1 unité de temps.
- Temps de calcul: nombre de transitions jusqu'à un état d'arrêt.
- f : $\mathbb{N} \to \mathbb{N}$, une MT M travaille en temps f si $\forall u \in \Sigma*$, M sur u s'arrête en au plus $f(|u|)$ transitions.
- En général, on suppose que $f(n) \geq n$: il faut au moins lire la donnée.
- Ce n'est pas toujours le cas : être pair en binaire.

Compléxité
Introduction

Definition

la compléxité d'une Machine de Turing est une fonction qui, à une taille d'entrée fait associer un nombre maximal de pas de calcul que la machine va faire avant de s'arreter.

- Chaque transition de la MT prend 1 unité de temps.
- Temps de calcul: nombre de transitions jusqu'à un état d'arrêt.
- f : $\mathbb{N} \to \mathbb{N}$, une MT M travaille en temps f si $\forall u \in \Sigma*$, M sur u s'arrête en au plus $f(|u|)$ transitions.
- En général, on suppose que $f(n) \geq n$: il faut au moins lire la donnée.
- Ce n'est pas toujours le cas : être pair en binaire.

Compléxité

Introduction

Definition

la compléxité d'une Machine de Turing est une fonction qui, à une taille d'entrée fait associer un nombre maximal de pas de calcul que la machine va faire avant de s'arreter.

- Chaque transition de la MT prend 1 unité de temps.
- Temps de calcul: nombre de transitions jusqu'à un état d'arrêt.
- f : $\mathbb{N} \to \mathbb{N}$, une MT M travaille en temps f si $\forall u \in \Sigma*$, M sur u s'arrête en au plus $f(|u|)$ transitions.
- En général, on suppose que $f(n) \geq n$: il faut au moins lire la donnée.
- Ce n'est pas toujours le cas : être pair en binaire.

Compléxité

Introduction

Definition

la compléxité d'une Machine de Turing est une fonction qui, à une taille d'entrée fait associer un nombre maximal de pas de calcul que la machine va faire avant de s'arreter.

- Chaque transition de la MT prend 1 unité de temps.
- Temps de calcul: nombre de transitions jusqu'à un état d'arrêt.
- f : $\mathbb{N} \to \mathbb{N}$, une MT M travaille en temps f si $\forall u \in \Sigma*$, M sur u s'arrête en au plus $f(|u|)$ transitions.
- En général, on suppose que $f(n) \geq n$: il faut au moins lire la donnée.
- Ce n'est pas toujours le cas : être pair en binaire.

Compléxité

Introduction

Definition

la compléxité d'une Machine de Turing est une fonction qui, à une taille d'entrée fait associer un nombre maximal de pas de calcul que la machine va faire avant de s'arreter.

- Chaque transition de la MT prend 1 unité de temps.
- Temps de calcul: nombre de transitions jusqu'à un état d'arrêt.
- f : $\mathbb{N} \to \mathbb{N}$, une MT M travaille en temps f si $\forall u \in \Sigma*$, M sur u s'arrête en au plus $f(|u|)$ transitions.
- En général, on suppose que $f(n) \geq n$: il faut au moins lire la donnée.
- Ce n'est pas toujours le cas : être pair en binaire.

Compléxité
Introduction

Grand O : $g(n) \in O(f(n))$ s'il existe k_1 et n_1 tel que

$$\forall n > n_1;\ |g(n)| \leq k_1 f(n)$$

donc si g est majorée par f à partir d'un certain rang.

Omega : $g(n) \in \Omega(f(n))$ s'il existe k_2 et n_2 tel que

$$\forall n > n_2;\ |g(n)| \geq k_2 f(n)$$

en d'autres termes si g majore f à partir d'un certain rang.

Theta : Finalement, $g(n) \in \Theta(f(n))$ si $g(n) \in \Omega(f(n))$ et $g(n) \in O(f(n))$

Le but étant de trouver des relations de type Θ ou O entre les complexités de divers algorithmes.
Il existe divers fonctions de référence:

$$\log(n)\ ;\ n\ ;\ n\log(n)\ ;\ n^k\ ;\ k^n\ ;\ n!\ ;\ n^n$$

Compléxité
Introduction

Grand O : $g(n) \in O(f(n))$ s'il existe k_1 et n_1 tel que

$$\forall n > n_1;\ |g(n)| \leq k_1 f(n)$$

donc si g est majorée par f à partir d'un certain rang.

Omega : $g(n) \in \Omega(f(n))$ s'il existe k_2 et n_2 tel que

$$\forall n > n_2;\ |g(n)| \geq k_2 f(n)$$

en d'autres termes si g majore f à partir d'un certain rang.

Theta : Finalement, $g(n) \in \Theta(f(n))$ si $g(n) \in \Omega(f(n))$ et $g(n) \in O(f(n))$

Le but étant de trouver des relations de type Θ ou O entre les complexités de divers algorithmes.
Il existe divers fonctions de référence:

$$\log(n)\ ;\ n\ ;\ n\log(n)\ ;\ n^k\ ;\ k^n\ ;\ n!\ ;\ n^n$$

Compléxité
Introduction

Grand O : $g(n) \in O(f(n))$ s'il existe k_1 et n_1 tel que

$$\forall n > n_1;\ |g(n)| \leq k_1 f(n)$$

donc si g est majorée par f à partir d'un certain rang.

Omega : $g(n) \in \Omega(f(n))$ s'il existe k_2 et n_2 tel que

$$\forall n > n_2;\ |g(n)| \geq k_2 f(n)$$

en d'autres termes si g majore f à partir d'un certain rang.

Theta : Finalement, $g(n) \in \Theta(f(n))$ si $g(n) \in \Omega(f(n))$ et $g(n) \in O(f(n))$

Le but étant de trouver des relations de type Θ ou O entre les complexités de divers algorithmes.
Il existe divers fonctions de référence:

$$\log(n)\ ;\ n\ ;\ n\log(n)\ ;\ n^k\ ;\ k^n\ ;\ n!\ ;\ n^n$$

Compléxité
Introduction

Grand O : $g(n) \in O(f(n))$ s'il existe k_1 et n_1 tel que

$$\forall n > n_1;\ |g(n)| \leq k_1 f(n)$$

donc si g est majorée par f à partir d'un certain rang.

Omega : $g(n) \in \Omega(f(n))$ s'il existe k_2 et n_2 tel que

$$\forall n > n_2;\ |g(n)| \geq k_2 f(n)$$

en d'autres termes si g majore f à partir d'un certain rang.

Theta : Finalement, $g(n) \in \Theta(f(n))$ si $g(n) \in \Omega(f(n))$ et $g(n) \in O(f(n))$

Le but étant de trouver des relations de type Θ ou O entre les complexités de divers algorithmes.
Il existe divers fonctions de référence:

$$\log(n)\ ;\ n\ ;\ n\log(n)\ ;\ n^k\ ;\ k^n\ ;\ n!\ ;\ n^n$$

Compléxité
Introduction

Grand O : $g(n) \in O(f(n))$ s'il existe k_1 et n_1 tel que

$$\forall n > n_1;\ |g(n)| \leq k_1 f(n)$$

donc si g est majorée par f à partir d'un certain rang.

Omega : $g(n) \in \Omega(f(n))$ s'il existe k_2 et n_2 tel que

$$\forall n > n_2;\ |g(n)| \geq k_2 f(n)$$

en d'autres termes si g majore f à partir d'un certain rang.

Theta : Finalement, $g(n) \in \Theta(f(n))$ si $g(n) \in \Omega(f(n))$ et $g(n) \in O(f(n))$

Le but étant de trouver des relations de type Θ ou O entre les complexités de divers algorithmes.
Il existe divers fonctions de référence:

$$\log(n)\ ;\ n\ ;\ n\log(n)\ ;\ n^k\ ;\ k^n\ ;\ n!\ ;\ n^n$$

Compléxité
Introduction

Definition

Soient M une machine de Turing (déterministe ou non) et w un mot. La durée d'exécution de M sur w, notée $d_M(w)$ est:

- 0 si w n'est pas accepté par M,
  - Le nombre minimum de transitions permettant d'atteindre un état accepteur depuis la configuration $q_0,\#w\#$

On peut maintenant définir la fonction de compléxité d'une machine

Example

Calculer la compléxité de la MT reconnaissant $a^n b^n$

Definition

La fonction de complexité d'une machine M (déterministe ou non) est la fonction $\mathcal{T}_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}$définie par $T_M(n) = \sup(d_M(w) \mid |w| = n)$

Compléxité
Introduction

Definition

Une machine de Turing M (déterministe ou non) est qualifiée de polynomiale s'il existe $k \in \mathbb{N}$ tel que :

$$T_M(n) \in O\left(n^k\right)$$

ou de manière équivalente, s'il existe un polynôme Q tel que :

$$\forall n \in \mathbb{N} : T_M(n) \leq Q(n)$$

Cette seconde forme étant parfois plus aisée à manipuler. Autrement dit, une machine de Turing est polynomiale si sa fonction de complexité est majorée par un polynôme.
En particulier, une fonction est calculable en un temps polynomial ou P-calculable s'il existe une machine de Turing polynomiale qui la calcule.

Transformation Polymônomiale

Definition

Soient $L_i \subset \Sigma*$, $i = 1, 2$ deux langages. Une fonction $f : \Sigma* \to \Sigma*$est une transformation polynomiale de L_1 vers L_2si:

- f est P-calculable
  - $f(x) \in L_2 \Longleftrightarrow x \in L_1$.

S'il existe une transformation polynomiale de L_1vers L_2, alors on notera $L_1 \leq_P L_2$ ou plus simplement $L_1 \leq L_2$.

Classes d'équivalence

Definition

1. La classe de complexité P est formée des langages décidés par une machine de Turing (déterministe) polynomiale.
2. La classe de complexité NP est formée des langages acceptés par une machine de Turing non déterministe polynomiale.
3. La classe de complexité NPC est un sous-ensemble de NP défini comme suit,

$$NPC = \left\{ L \in NP \mid \forall L' \in NP, L' \leq L \right\}.$$

Au vu de cette définition, on peut dire en quelque sorte que les problèmes *NP-complets* sont "les plus difficiles" du point de vue algorithmique.

$$P \subset NP \text{ et } NPC \subset NP.$$

Conclusion

Apparues à l'occasion des problèmes de la logique mathématique, les notions de calculabilité et de décidabilité ont progressivement touché un grand nombre de disciplines, y compris en dehors des mathématiques.
Il n'est pas excessif de dire que par leurs implications concrètes (en informatique en particulier) mathématiques et philosophiques, ces notions sont parmi les plus importantes qui aient été mises à jour au vingtième siècle, à l'égal de celles élaborées en relativité, en mécanique quantique, ou en biologie moléculaire.

L'indécidabilité et la prédictibilité en physique.

Certains des résultats présentés peuvent être interprétés d'un point de vue physique.
En effet réaliser matériellement une machine de Turing où mécanisme ayant la puissance des machines de Turing, c'est-à-dire la puissance de calcul maximale concevable, est facile.
L'indécidabilité du problème de l'arrêt prend alors un sens nouveau concernant la prédictibilité en physique.même si on connait parfaitement un système physique et toutes les lois qui le régissent, même si de plus ce système ne répond qu'à des lois déterministes, il se peut quand même que son comportement à long terme ne soit pas prédictible. Même dans un univers simplifié, non quantique, qu'on connaîtrait parfaitement, l'avenir continuerait de nous échapper