Chapitre six

Décidabilité

MPI/MPI*, lycée Faidherbe

Résumé

Dans ce chapitre on étudie la notion de problème : différents type de problèmes, taille des données. On définit le cadre théorique dans lequel on espère pouvoir résoudre les problèmes. On finit la chapitre par la notion de décidabilité qui exclut des problèmes qu'on ne peut pas résoudre.

I Machine universelle

Les problèmes informatiques sont résolus en écrivant un code sensé résoudre le problème ; la résolution en elle-même consiste alors à compiler le programme et à l'exécuter sur les données souhaitées. On considérera que les données sont codées sous forme d'une chaîne de caractère unique ; c'est le programme qui en fait l'analyse.
Une remarque importante est que le programme lui-même peut être codé sous forme d'une chaîne de caractères, c'est d'ailleurs sous cette forme qu'il est enregistré. Un programme pourra donc être considéré comme une donnée d'entrée.
Le cadre théorique sera alors une *machine universelle* qui reçoit deux chaînes de caractères en paramètres, le programme et les données, et qui renvoie le résultat sous forme d'une chaîne de caractères.
Par exemple en OCaml la machine universelle peut être `eval : string -> string -> string` telle que `eval f e` qui renvoie le résultat de la fonction écrite dans la chaîne `f`, qui doit être de type `string -> string`, évaluée pour la donnée écrite dans la chaîne `e`.
On notera que la fonction suivante **n'est pas** une machine universelle

```
let eval f e = f e
```

En effet, dans l'écriture ci-dessus, `f` est le nom d'une fonction enregistrée dans l'espace de travail de OCaml alors que la machine universelle doit lire une chaîne de caractère et la traduire. De même pour les données.
Cependant, pour simplifier les écritures, on considérera que les fonctions sont des fonctions du langage et on utilisera fe plutôt que `eval f e`.

On supposera que la machine universelle dispose d'autant de mémoire et qu'elle est capable de fonctionner autant de temps qu'il est nécessaire.
La thèse de Church conjecture que tout les modèles de calcul raisonnables sont équivalents : ils sont capables de résoudre les mêmes problèmes.

II Types de problèmes

II.1 Problèmes de décision

Définition 1

Un *problème de décision* est la donnée d'une entrée (ou *instance*) et d'une question d réponse booléenne

Instance : un objet possédant certaines caractéristiques,

Question : une question à réponse OUI/NON,

Une instance est *positive* (resp. *négative*) si la réponse à la question est OUI (resp. NON).

On peut penser les problèmes de décision sous la forme d'un problème de langage : si on considère le langage L des instances positives, le problème revient à demander si le mot qui représente une instance appartient au langage .
Inversement tout langage $L \subset \Sigma^*$ peut définir un problème de décision :

Instance : un mot $u \in \Sigma^*$,

Question : u appartient-il à L ?

Exemples

1. PREMIER
   Instance : un entier n.
   Question : n est-il premier ?
2. DIAGONALISABLE
   Instance : une matrice $M \in \mathcal{M}_n(\mathbb{R})$.
   Question : M est-elle diagonalisable ?
3. FORTEMENTCONNEXE
   Instance : un graphe orienté G.
   Question : G est-il fortement connexe ?
4. ÉQUILIBRÉ
   Instance : un arbre binaire a.
   Question : a est-il équilibré ?

II.2 Problèmes d'optimisation

Définition 2

Un *problème de décision* est la donnée d'une instance, d'une description de la solution possible et d'une mesure à optimiser

Instance : un objet possédant certaines caractéristiques.

Solution : la structure de la solution cherchée.

Optimisation : trouver une solution y pour une instance x telle que $f(x, y)$ est est minimale (ou maximale).

Exemples

1. LOGENTIER
   Instance : un entier n.
   Solution : un entier p tel que $2^p \leqslant n$.

Optimisation : trouver une solution p maximale.

2. ARBRECOUVRANTMINIMUM

   Instance : un graphe non orienté valué G.

   Solution : un arbre couvrant de G, a.

   Optimisation : trouver une solution a de poids minimal.

3. COMPOSANTECONNEXE

   Instance : un graphe non orienté G et un sommet.

   Solution : un sous-graphe connexe de G contenant s, $G[S]$.

   Optimisation : trouver S de cardinal maximal.

On peut associer un problème de décision à un problème d'optimisation en introduisant, dans l'instance, un paramètre supplémentaire : un seuil.

Instance : un objet possédant certaines caractéristiques et un seuil k,

Question : existe-t-il un objet y ayant la structure de solution cherchée telle que $f(x,y) \leqslant k$ ou existe-t-il un objet y ayant la structure de solution cherchée telle que $f(x,y) \geqslant k$?

On cherche à savoir si on peut atteindre $f(x,y) \leqslant k$ (resp. $f(x,y) \geqslant k$) lorsque l'optimum recherché était un minimum (resp. un maximum).

III Décidabilité

Jusqu'à présent les problèmes rencontrés avaient une solution algorithmique ; lorsqu'on demande d'écrire une fonction qui a telle action, on présuppose qu'un algorithme existe. Cependant, ce n'est pas toujours le cas ; les calculs de nombres réels ne peuvent pas être exacts, par exemple.
Un autre cas est le calcul de la longueur de vol de la suite définie par
$u_0 \in \mathbb{N}^*$ et $u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } n \text{ est pair} \\ 3 \cdot u_n + 1 & \text{si } n \text{ est impair} \end{cases}$.
La longueur de vol étant le premier entier n tel que $u_n = 1$.
On ne sait pas (pour l'instant) prouver qu'un tel entier existe.
On va montrer qu'il existe des problèmes dont on peut prouver qu'il n'existe pas d'algorithme permettant de les résoudre.

Définition 3

Un problème de décision est *décidable* s'il existe un algorithme dans le modèle de calcul choisi permettant de le résoudre. Cet algorithme doit toujours terminer (en temps fini) et doit toujours renvoyer une réponse correcte.
Le problème est dit *indécidable* sinon.

Dans les exemples, les problèmes porteront principalement sur des fonctions/programmes/algorithmes dont la réponse est booléenne. Un tel programme exécuté sur une entrée peut avoir 3 types de résultats :

- le programme termine et renvoie `true`,
- le programme termine et renvoie `false`,
- le programme ne termine pas, parce qu'il y a une erreur ou en calculant sans fin.

III.1 Le problème de l'arrêt

Le premier exemple de problème indécidable est le problème de l'arrêt :
ARRÊT ou HALT

Instance : un programme f et un argument e (tous deux sous forme de chaîne de caractères).

Question : l'exécution de f sur e termine-t-il ?

Il est important de noter que ce n'est pas la preuve de terminaison d'un programme donné qui est en jeu ici ; c'est l'existence d'une preuve universelle de terminaison. Autrement dit, l'indécidabilité de ce problème indique que prouver la terminaison d'un programme doit se faire avec des moyens adaptés à chaque algorithme.

Théorème 1

Le problème de l'arrêt est indécidable.

Démonstration On suppose qu'il existe une fonction `arret` solution du problème de l'arrêt.
On va écrire une fonction qui admet un paramètre chaîne de caractère qui est lisible comme une fonction : ce paramètre peut servir comme fonction ou comme donnée.

```
let paradoxe x =
  if arret x x
  then while true do () done
```

Si le calcul de la fonction `x` appliquée aux données `x` termine, la fonction `paradoxe` ne termine pas.
On sauve le texte de cette fonction dans une chaîne de caractère `code_para` et on exécute la machine universelle `paradoxe code_para`.

- Si cet appel termine alors l'appel de `arret code_para code_para` renvoie `true` et la fonction `paradoxe` appliquée à`code_para` ne termine pas. Ainsi l'appel `paradoxe code_para` ne termine pas.
- Si cet appel ne termine pas alors l'appel de `arret code_para code_para` renvoie `false` et la fonction `paradoxe` appliquée à`code_para` termine. Ainsi l'appel `paradoxe code_para` termine.

Cette contradiction montre que la fonction `arret` ne peut pas exister. . . . . ■

III.2 Réductions

Pour prouver d'autres indécidabilités on peut utiliser des *réductions*.
Une réduction d'un problème A à un problème B consiste à résoudre le problème A à l'aide d'une résolution du problème B : si f_B est un algorithme résolvant B, on crée un algorithme f_A résolvant A qui fait un ou plusieurs appels à f_B. On écrira $A \leqslant B$ si A peut se réduire à B.

Théorème 2

Si A et B sont deux problèmes de décision tels que $A \leqslant B$ alors :

- si B est décidable alors A est décidable.
- si A est indécidable alors B est indécidable

C'est surtout la seconde assertion qui sera utilisée.
Démonstration Si B est décidable, il existe une fonction f_B qui résout B.
Si, de plus, on a $A \leqslant B$, alors on peut écrire, à l'aide de f_B, une fonction $f - A$ qui résout A donc A est décidable.
La deuxième assertion est la contraposée. . . . . ■

Il existe un cas particulier de réduction, par transformation des instances.
On suppose qu'il existe une fonction calculable Φ telle que pour toute instance x pour A, $\Phi(x)$ est une instance pour B et x est positive pour A si et seulement si $\Phi(x)$ est positive pour B.
Dans ce cas, pour toute fonction f_B qui résout B, $f_A = f_B \circ \Phi$ résout A donc $A \leqslant B$.
Cette réduction, aussi appelée réduction *many-to-one*, est notée $A \leqslant_m B$.

Exemples

1. ÉQUIVALENCE est indécidable.

   Instance : deux programme f et g,

   Question : f et g sont-ils équivalents, c'est-à-dire ont-ils le même comportement (terminer tous les deux et renvoyer le même résultat `true` ou `false` ou ne pas terminer tous les deux) pour toute entrée ?

   On prouve que ARRÊT $\leqslant$ ÉQUIVALENCE.

   On considère une fonction `equivalence` qui résout ÉQUIVALENCE.

   On peut alors écrire une fonction `arret` :

   ```
   let arret f x =
     let f1 () = let _ = f x in () in
     let f2 () = while true do () done in
     not (equivalence f1 f2)
   ```

   Comme ARRÊT est indécidable, ÉQUIVALENCE l'est aussi.

2. ARRÊT TOTAL est indécidable.

   Instance : un programme f,

   Question : f termine-t-il pour toute entrée ?

   On peut transformer toute instance de ARRÊT en un instance de ARRÊT TOTAL

   ```
   let transformation f x =
     let g y =  f x in
     g
   ```

   g termine pour toute entrée si et seulement si fx termine.

   Cela permet donc de réduire ARRÊT à ARRÊT TOTAL qui est ainsi indécidable.

Par contre le problème ARRÊT (BORNE) est décidable.

Instance : un programme f, un argument e et un entier n

Question : l'exécution de f sur e termine-t-elle avant n opérations ?

On exécute $f(x)$ sur une machine universelle en comptant le nombre de cycles d'horloge ; on arrête après $k \cdot n$ cycle si le programme n'est pas terminé. k est le nombre de cycle utilisés par opération élémentaire.