Tours de Hanoï et Pavage de Penrose

Guillaume Barbier, Romain Ferrand

2 octobre 2017

Résumé

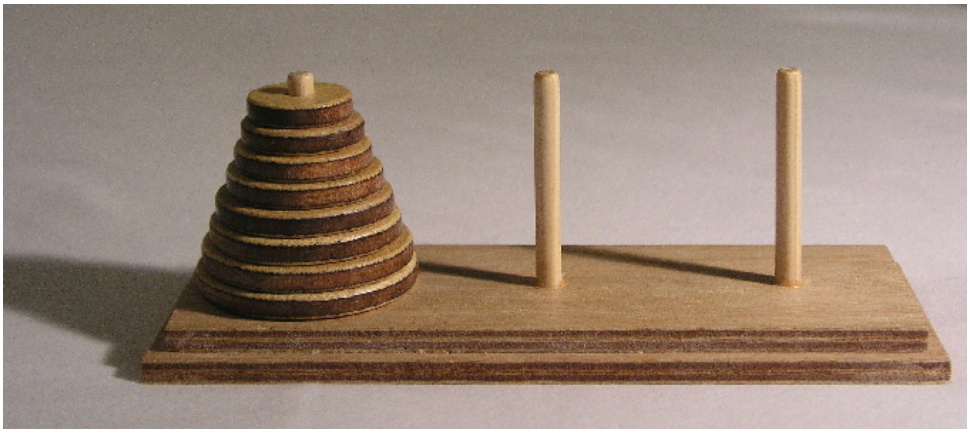

FIGURE 1 – Evanherk wikipédia

Introduction

Dans ce document nous allons vous présenter notre étude du problème de Hanoï, ainsi que les différentes implémentations visant à sa résolution. Puis étudierons différents pavages de Penrose.

1 Tours de Hanoï

1.1 Présentation du problème

Les Tours de Hanoï est un jeu de réflexion inventé par le mathématicien **Édouard Lucas** (1842-1891). Il consiste à déplacer un nombre donné de disques de différents diamètre de la tour de départ vers la tour d'arriver, tout en passant par une tour intermédiaire.

Le joueur doit également respecter deux règles :

— déplacer un disque à la fois ;

— ne pas déplacer un disque donné sur un disque plus petit.

1.1.1 Implémentation sans affichage graphique

L'algorithme le plus intuitif pour résoudre le problème des Tours de Hanoï à trois tours, est un algorithme récursif très simple.

Algorithm 1 Tours de Hanoï

```
1: procedure HANOI(disque, src, aux, dest)   ▷ Trois tours : source, auxiliaire et destination
2:     if disque = 0 then ne rien faire
3:     else
4:         Hanoi (disque − 1, src, dest, aux)
5:         déplacer disque de src à dest
6:         Hanoi (disque − 1, aux, src, dest)
7:     end if
8: end procedure
```

Il a donc été sans trop difficulté implémenté, avec, dans un premier temps, un affichage console pour signifier les mouvements effectués.

```
let rec hanoi (nb_disc:int) (a:rod) (b:rod) (c:rod) =
  if nb_disc = 0 then ()
  else
    begin
      hanoi a c b (n_disc−1);
      movement a c;
      hanoi b a c (n_disc−1)
    end
;;
```

Bilan 1ère implémentation et retour sur le code

Comme expliqué précèdement l'implémentation basique de Hanoï étant assez simple aucunes réelle difficulté n'a été relevée. Nous avons simplement défini un tour par un nouveau type **rod**, qui dans cette première implémentation est une lettre. Lors de la première évaluation du code, il nous a été demandé d'utiliser le *patter matching* de **Ocaml** que lorsque cela le justifiait. La procédure Hanoï ne justifiant pas son utilisation, puisqu'elle peut être remplacée par un simple **if-else**, elle a été changé en conséquence.

1.2 Implémentation avec affichage graphique

Dans le cas des tours de Hanoï, nous avons implémenté la première extension proposée à savoir un affichage graphique. Dans notre implémentation nous avons souhaité afficher à la fois les piquets et les disques.

Notre première approche a été de considérer les types suivants :

type : Disque
- hauteur
- largeur

type : Contenu piquet
- liste de disque

type : Forme piquet
- couleur
- rectangle (4 points)

type : Position piquet
- point

type : Piquet
- position
- forme
- contenu

Les disques comme ayant une relation de combinaison avec les piquets, ainsi chaque piquet "possède" sa pile de disque et l'affiche en fonction de sa position. l'idée étant de gérer le déplacement donné par l'algorithme de Hanoï par un pop du piquet source en push du piquet destination, et de gérer par la suite l'affichage piquet par piquet.

Dans un premier temps, nous avons implémenté ces types sous forme de tuples, cette manière de faire à rapidement crée un code lourd et complexe à gérer. En effet, dans le cas du type **Piquet**, nous devions reflèchir de manière positionelle ou utiliser l'unpacking de Ocaml, tout en utilisant à la fois que la partie du tuple qui était nécéssaire, nous avions donc a traité plus de données que nous en avions besoin. Dans un même temps le "champ" *contenu* du type

piquet était immutable, cela se caractèrisait par la nécéssité de reconstruire une instance de **piquet** à chaque modification de sa liste de disque.

Nous avons donc décidé de remplacer le tuple de **piquet** par un record, tout en rendant mutable le champ contenu. Cette manière de faire à permis de clarifier énormément le code, puisque grâce à l'opérateur de résolution de portée ".", nous avions un moyen simple de spécifier le lien entre les données. Cela a permit également de simplifier le code, puisque nous avions plus besoin de réfléchir en terme de position ou d'unpacking, et la liste mutable de disque nous permetait de ne pas avoir à recrée des instances de type **piquet**.

1.2.1 retour sur le code

Après l'évaluation du code il nous a été demandé de privilègier un code modulable. Un module Rod (piquet) à donc été crée. Dans sa signature se module permet de crée des piquet, toutes les stuctures auxiliaires qu'il et gère également l'affichage des structures. Ce module permet de gérer les données de manière abstraite dans le code principale, ainsi même si notre implémentation, gère la forme du piquet et sa position comme des tuples ainsi que son contenu comme une liste de disque, cette concidèration est interne, non visible à l'utilisateur et générique.

Optimisation possible de l'affichage : Notre implémentation va a chaque mouvement, nettoyer la fenêtre et tout réafficher. Si cette implémentation du code semble peu optimale par rapport à un remplacement par un carré rectangle sur la précédente position du disque. Elle se justifie par la présence des piquets, qui nécéssite donc l'affichage d'un deuxième rectangle de la bonne couleur en fonction du piquet, mais également par le fait que cette optimisation ne justifie sans doute pas la complexification du code au regard de la simplicité des opérations effectuées.

Une seconde optimisation cette fois effectué a été d'utiliser le double buffering du module Graphics : Graphics par défaut "dessine" à la fois sur l'écran et dans une zone mémoire appelé backing store. Grâce à l'option **autosynchronize false**. Nous avons pu faire en sorte que graphics écrive que dans le "backing store" et qu'il se synchronise avec l'affichage que lorsqu'on le juge utile, à savoir lorsque toutes les opérations d'affichage ont été effectuées. Cette manière de faire permet d'éviter les problèmes de *flickering* même lorsque la vitesse du jeu est très rapide.

Modification de type forme : Lors de la précédente implémentation le type **forme piquet** à été conçu comme un tuple d'une couleur et de 4 points, cette représentation est différente du type de **disque** sans que cela se justifie. En effet comme nous stockons la position des piquets il aurait été redondant de stocker les points de la forme du piquet calculer en fonction de cette position. Nous avons donc juger utile de ne garder que sa position, sa hauteur et sa largeur.