Rapport Projet 0 : Tours d'Hanoï et Pavage de Penrose

Bailluet Nicolas, Piau Rémi
ENS Rennes

1er octobre 2018

1 Les tours de Hanoï

Nous traitons dans cette partie le problème des tours de Hanoï à n disques et 3 piquets. Nous exposons dans un premier temps une solution récursive classique, son étude nous mène par la suite à construire une solution itérative.

1.1 Solution initiale - Un algortithme récursif

La résolution du problème est basée sur le fait que pour déplacer un disque d'un piquet à un autre, il est nécessaire de déplacer tous les disques présents au dessus vers un piquet intermédiaire. Ainsi, pour résoudre le problème à n disques on effectue les actions suivantes (voir l'algorithme 1 en annexe) :

— résoudre le problème pour $n-1$ disques qui sont déplacés vers un piquet intermédiaire ;
— déplacer le n-ieme disque vers le piquet d'arrivée ;
— résoudre le problème pour $n-1$ disques qui sont déplacés du piquet intermédiaire vers le piquet d'arrivée.

Une solution simple et optimale en nombre de coups joués. En effet, notons $C(n)$ le nombre de coups nécessaire, alors :

$$C(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2C(n-1)+1 & \text{sinon} \end{cases} \Leftrightarrow \boxed{C(n) = 2^n - 1}$$

1.2 Étude de la solution récursive

Afin de construire l'algorithme itératif, il est tout d'abord nécessaire de montrer certaines propriétés relative à la résolution récursive décrite précedemment.

1.2.1 Représentation arborescente

Nous utiliserons par la suite une représentation arborescente des appels récursifs (et des tours de jeu). Pour cela nous indiçons les disques de 1 (le plus petit) à n (le plus grand). L'arbre suit le schéma suivant :

— on place à la racine le noeud étiqueté n correspondant au premier appel récursif ;
— tout noeud a pour fils gauche l'arbre pour $n1$ disques, ce qui représente les pré-requis pour pouvoir déplacer le disque d'indice n ;
— tout noeud a pour fild droit l'arbre pour $n-1$ disques, ce qui représente les actions à effectuer après le mouvement du disque d'indice n afin de pouvoir reformer la tour.

Un exemple pour $n = 4$ se trouve en annexe (voir figure 1).

Par construction, l'arbre binaire est complet, son parcours infixe donne l'ordre de jeu des disques et tous les noeuds d'une même génération possèdent la même étiquette. Ainsi en considérant le premier tour de jeu comme le tour d'ordre 0 on en déduit :

Le plus petit disque est joué à tous les tours d'ordre pair.

1.2.2 Direction des disques

Nous pouvons de plus montrer *qu'un même disque est toujours déplacé de la même manière, soit d'un piquet vers la droite, soit d'un piquet vers la gauche*.

Soit $k \in \{1, ..., n\}$, notons $c_k(i)$ le triplet (piquet de départ, piquet intermédiaire, piquet d'arrivée) lors du i-ieme mouvement du disque d'indice k. Montrons par récurrence descendante sur k que $c_k(i+1)$ est une rotation de $c_k(i)$ vers la droite :

— pour $k = n$, il n'y a qu'un seul mouvement du disque d'indice k ;
— supposons le résultat vrai pour un k' et notons $c_k(i) = (a, b, c)$ (pour un i fixé) :
  — si i est pair, alors $c_{k'-1}(\frac{i}{2}) = (a, c, b)$ d'où $c_{k'}(i+1) = (c, a, b)$ (voir l'algorithme 1) ;
  — sinon, alors $c_{k'-1}(\frac{i-1}{2}) = (b, a, c)$ puis par hypothèse de récurrence $c_{k'-1}(\frac{i+1}{2}) = (c, b, a)$ et donc $c_{k'}(i+1) = (c, a, b)$.

D'où le résultat.

1.2.3 Étude du tour de jeu d'ordre k

Nous pouvons ensuite déterminer l'indice du disque joué au tour d'ordre k. On remarque pour cela que le parcours infixe de l'arbre pour n disques auquel on retire tous les 1 donne le parcours de l'arbre pour $n-1$ disques (le plus petit disque est alors indicé 2), ainsi un noeud interne situé en k-ieme position dans le parcours pour n disque est situé en position $\frac{k-1}{2}$ dans le parcours pour $n-1$ disques. De plus, on sait que les tours d'ordre pair correspondent aux mouvements du plus petit disque. Ainsi, pour déterminer l'indice du disque joué on procède comme ceci :

— si k est pair, le plus petit disque est joué ;
— sinon on réitère le procédé avec $\frac{k-1}{2}$, et l'indice du disque joué est le nombre de fois qu'il a été nécessaire de diviser k.

Autrement dit :

L'indice du disque joué au tour d'ordre k est l'indice du premier bit nul dans l'écriture binaire de k.

1.3 Solution itérative

Il est alors possible, à partir des propriétés démontrées, de construire un algorithme itératif. Pour cela on effectue les mouvements suivants tant que le piquet d'arrivée ne contient pas $n-1$ disques :

— déplacer le plus petit disque vers la droite ou la gauche en fonction de la parité de n ;
— faire le seul mouvement qui n'implique pas de déplacer le plus petit disque.

Il reste alors à déplacer le plus petit disque vers le piquet d'arrivée (voir l'algorithme 2).

La terminaison, l'optimalité et la correction sont immédiates par construction, en effet les mouvements sont les mêmes que celui de l'algorithme récursif qui est correct et optimal.

1.4 Implémentation

Afin de gérer le contenu de chaque piquet, le type abstrait `Rod` a été implémenté (voir l'interface en annexe figure 2) : `type rod = { mutable size: int; name: string; mutable discs: int list}`, l'implémentation ressemble à celle d'une pile et ne tient pas compte des règles du jeu, il est donc possible de l'utiliser dans le cas d'un jeu similaire avec des règles différentes.

Afin de mieux visualiser la résolution, nous avons implémenté une animation qui affiche le schéma de résolution. Les disques sont représentés par des rectangles et celui en mouvement est coloré en rouge (voir la figure 3). Pour réaliser cette animation nous avons eu recours au module `Graphics` d'Ocaml, à chaque mouvement l'état du jeu est mis à jour (stocké dans trois objets de type `Rod`) et le nouvel état est affiché en dessinant chaque disque de chaque piquet parcouru récursivement.

2 Pavage de Penrose

Nous nous intéressons au pavage de Penrose sur un exemple avec des triangles isocèles de rapport ϕ entre leurs côtés (avec $\phi = \frac{1+\sqrt{5}}{2}$, le nombre d'or).

2.1 Solution initiale - Algorithme : un pavage par découpage récursif

L'algorithme proposé pour le pavage est un découpage récursif d'un triangle de départ en sous triangles (voir l'algorithme 3). Le découpage diffère en fonction de la nature du triangle obtu ou aïgu (voir figure 4). En reprenant les notations de la figure 4, on peut écrire les formules de découpage qui reviennent à calculer la position du point D dans le cas obtu et des points D et E dans l'autre cas. On a (avec O l'origine du repère i.e $(0,0)$) :

$$\vec{OD} = \vec{OA} + \frac{\vec{AB}}{\phi}$$

pour le triangle obtu, et

$$\vec{OD} = \vec{OC} - \frac{\vec{AC}}{\phi}$$

$$\vec{OE} = \vec{OB} + \frac{\vec{BC}}{\phi}$$

pour le triangle aïgu.

2.2 Implémentation : un code modulaire

L'implémentation du pavage à été divisée en plusieurs fichiers (voir figure 5 pour les relations entre ces fichiers, une flèche vers un fichier indique que celui-ci est utilisé par celui d'où part la flèche).

2.2.1 Une interface de programmation pour les triangles : `api\_triangle.cma`

Nous avons réalisé une interface de programmation `api_triange.cma`.

Cette interface [6] expose plusieurs types abstraits :

— les types d'angles, aîgus ou obtus `type angletype = Obtuse | Acute`,
— les points représentés par un couple de flottants i.e. `type point = (float * float)`,
— les triangles `type triangle = {points:(point array); typ:angletype}`, repésentés par une liste de points et un type d'angle.

Ces types sont accessibles et manipulables *uniquement* au travers des fonctions exposées par l'interface comme la fonction `is_acute : triangle -> bool`. On notera l'implémentation d'opérateurs infixes comme `-- : point -> point -> point`(soustraction) ou `// : point -> float -> point`(division par un scalaire) qui permettent de manipuler le type point de manière élégante et plus compréhensible.

2.2.2 Fichier pour les fonction graphiques : `trgraphics.ml`

Ce fichier contient les fonctions permettant de facilement afficher les différents éléments du pavage (lignes, triangles) ainsi que les outils nécessaires au contrôle clavier de l'interface.

2.3 Améliorations

2.3.1 Dessin unique des lignes de séparations : `penrose\_noDoubleLine.ml`

Notre première amélioration concerne les lignes de découpes entre les triangles, ces dernières sont dessinées deux fois dans la première implémentation, une fois avec chaque triangle lors de l'affichage en utilisant la fonction `draw_triangle_with_line` du fichier `trgraphics.ml`. Dans cette implémentation les lignes sont dessinées au fur et à mesure à chaque génération. En reprenant les notations de la figure 4, on trace les lignes AB, AC, BC pour le triangle initial puis à chaque division il suffit de tracer CD si le triangle est obtu ou DE, DB sinon avant de passer à la génération $(n-1)$ et de dessiner enfin les triangles sans bordure à la génération 0 avec `draw_triangle`.

2.3.2 Déplacement et gestion des générations au clavier : `penrose\_anime.ml`

Nous avons aussi réalisé une version du programme qui est capable de réagir aux entrées clavier et qui permet ainsi de se déplacer sur le pavage, de changer d'échelle ainsi que de génération. Pour ce faire nous utilisons une boucle infinie bloquée par une fonction qui attend les évènements clavier et les passe à la fonction chargée du traitement de ces derniers. Pour faciliter la communication entre les différentes fonctions et la gestion des données, nous avons créé un nouveau type :

```
type environnement =
{mutable table: triangle list; mutable scale: float; mutable offset: point}
```

Ce type une fois instancié permet de garder dans une seule variable toute l'information utile à l'affichage, c'est-à-dire le décalage par rapport au centre (offset), le grossissement (scale) ainsi que la liste (table) de tous les triangles de la génération courante. Cette liste permet grâce à une fonction adaptée de calculer la géneration suivante à partir de celle enregistrée, sans avoir besoin de recalculer toutes les divisions précédentes ce qui permet un gain important en performances quand le nombre de divisions est important. De plus pour éviter les scintillements dûs au temps nécessaire pour le calcul puis le dessin, ce programme utilise la technique du "double buffering" qui consiste à dessiner dans une mémoire tampon puis à afficher. Cette amélioration permet de mieux comprendre le processus de pavage car l'utilisateur peut se déplacer à son gré dans celui-ci.

3 Conclusion

Les tours de Hanoï et le pavage de Penrose sont deux problèmes qui possèdent un intérêt pédagogique de par leur simplicité d'implémentation ce qui en fait de bons exemples pour l'apprentissage de la récursivité et des domaines où cette technique montre toute sa puissance. Mais ces sujets bien que simples en apparence deviennent vite très complexes si on relache un tant soit peu leurs hypothèses de construction ce qui les rend intéressant pour la recherche.

Au delà des aspects pédagogiques, l'étude de ces sujets nous a mené à traiter des problèmes de dérécursivation et d'optimisation. De plus, il nous a été nécessaire de réfléchir aux choix d'implémentation les plus adaptés aux problèmes.

Il reste toutefois de nouveaux aspects que nous n'avons pas exploré. Il serait maintenant intéressant de traiter les cas des tours de Hanoï avec plus de 3 piquets, il serait alors avantageux de définir des fonctions d'animation plus génériques, qui peuvent s'adapter à de nouvelles règles. Concernant le pavage de Penrose, nous pourrions essayer d'implémenter d'autres types de pavage plus réguliers comme un pavage en losange.

A Annexe - Hanoï

Algorithme 1 : Hanoi(n, s, i, d)

Entrées : n : nombre de disques,
s : piquet de départ,
i : piquet intermédiaire,
d : piquet d'arrivée

si $n > 1$ **alors**
  Hanoi($n-1$, s, d, i)
  Déplacer un disque de s vers d
  Hanoi($n-1$, i, s, d)
sinon
  Déplacer un disque de s vers d
fin

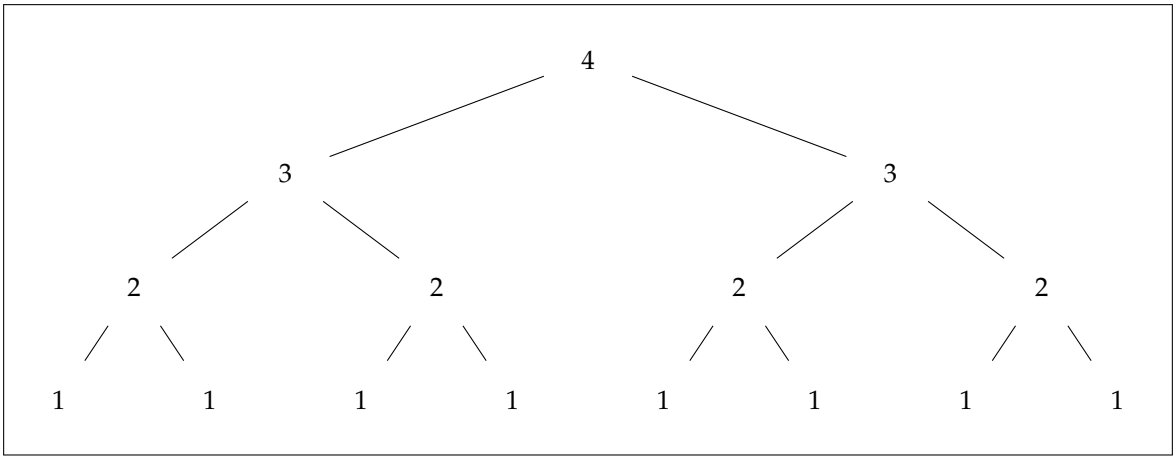

FIGURE 1 – Arbre pour $n = 4$

Algorithme 2 : Hanoi(n, s, i, d)

tant que *hauteur(d)* $< n - 1$ **faire**
  si *n est pair* **alors**
    Déplacer le plus petit disque d'un piquet vers la droite
  sinon
    Déplacer le plus petit disque d'un piquet vers la gauche
  fin
  Effectuer le seul mouvement qui ne déplace pas le plus petit disque
fin
Déplacer le plus petit disque vers d

```
type rod
val push : rod -> int -> unit
val pop : rod -> int
val is_empty : rod -> bool
val top : rod -> int
val init_rod : int -> string -> rod
val move : rod -> rod -> unit
val get_size : rod -> int
val get_name : rod -> string
```

FIGURE 2 – Interface du type abstrait Rod

FIGURE 3 – Animation de la résolution

B Annexe - Penrose

Algorithme 3 : Penrose(n, t)

Entrées : n : nombre de générations,
t : triangle initial
si $n = 0$ **alors**
  dessiner t
sinon
  Découper t et appliquer Penrose($n-1$,t_i) pour chaque sous triangle t_i.
fin

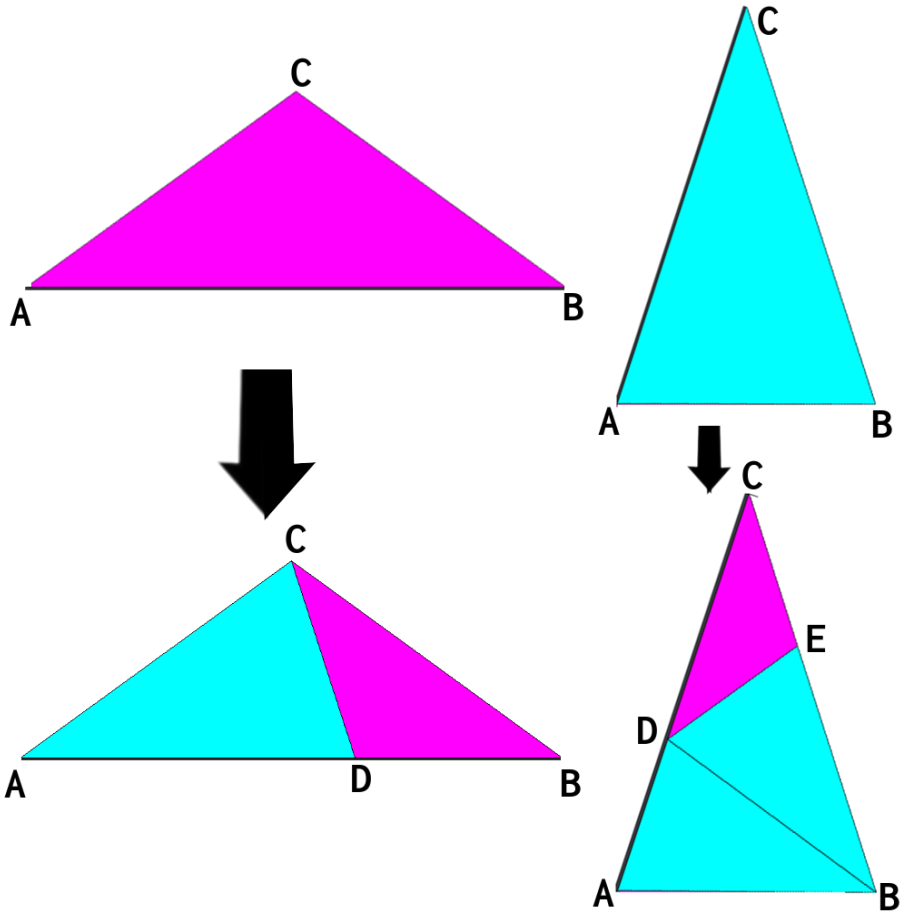

FIGURE 4 – Division des triangles (obtus à gauche, aïgus à droite.

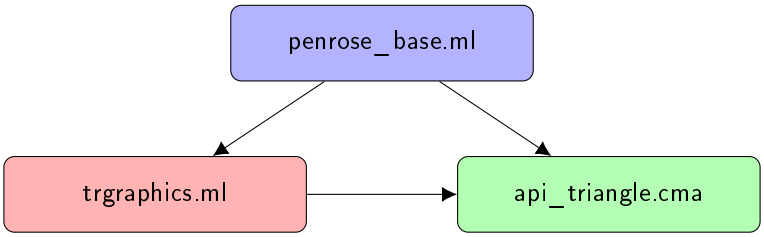

FIGURE 5 – Relations entre les fichiers

.

```
type angletype
type point
type coord = int * int
type triangle
val ( ++ ) : point -> point -> point
val ( -- ) : point -> point -> point
val ( ** ) : point -> float -> point
val ( // ) : point -> float -> point
val norm : point -> float
val scale : point -> float -> point
val translate : point -> float -> float -> point
val new_point : float -> float -> point
val new_triangle : point -> point -> point -> bool -> triangle
val is_acute : triangle -> bool
val get_point_vect : triangle -> point array
val get_points : triangle -> point * point * point
val get_point : int -> triangle -> point
val round : float -> int
val coordvect_of_pointvect : point array -> coord array
val coordvect_of_triangle : triangle -> coord array
```

FIGURE 6 – api_triangle.mli