Rapport de projet

Marco Freire, Clément Legrand-Duchesne

26 septembre 2017

Résumé

Dans ce rapport seront expliqués les codes du pavage de Penrose suivant différents types de parcours, et des tours de Hanoi, problème à contrainte résolu récursivement, ainsi que leurs extensions ; et les choix d'implémentation que nous avons été menés à faire.

Mots-clés : Pavage ; parcours ; récursivité ; contrainte.

1 Penrose

1.1 Principe : algorithme de base

Ce pavage de Penrose s'obtient par divisions successives de triangles aux dimensions particulières. Le pavage de Penrose implémenté utilise deux tuiles de base, des triangles d'or. Ce sont des triangles isocèles dont la longueur des cotés sont 1 et le nombre d'or. Les dimensions de leurs côtés sont donc des des nombres flottants. Afin d'éviter des erreurs d'arrondi susceptibles de se cumuler, et d'alléger le code, nous avons choisi de ne travailler qu'avec des flottants, avant des les arrondir en entiers juste avant le tracé.

Nous avons choisi de representer les triangles par un tableau contenant les coordonnées des trois sommets. Par commodité, nous avons choisi de suivre la convention suivante : le premier sommet est celui duquel sont issus les deux cotés de longeurs égales, le second est le suivant en tournant dans le sens horaire (Figures **??** et 3).

Afin de différentier les triangles obtus des triangles aigus, nous avons créé un type `triangle` :

```
type triangle = Obtuse | Acute;;
```

Le travail se résume alors à écrire une fonction `divide`, prenant en paramètre un triangle (le tableau des coordonnées de ses sommets et son type) ainsi que la le nombre de subdivisions

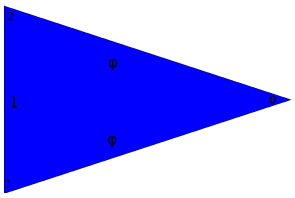


FIGURE 1 – Triangle d'or aigu

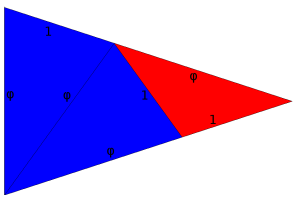


FIGURE 2 – Découpe d'un triangle d'or aigu

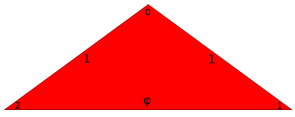


FIGURE 3 – Triangle d'or obtu

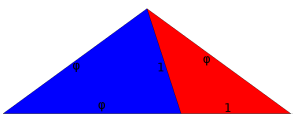


FIGURE 4 – Découpe d'un triangle d'or obtu

restant à effectuer, qui effectue la découpe de celui-ci, et s'appellant récursivement avec en paramètre les nouveaux triangles obtenus (Figures 2 et 4).

L'idée est alors de découper récursivement un triangle initial de grande taille.

Inserer image !

Une des principales difficultés rencontrées lors de l'implémentation de ce pavage est le fait la fonction `fill_poly` du module `Graphics.cma` remplit le polygone avec une couleur donnée, mais recouvre et donc efface les bords de celui ci si ceux ci ont été tracés avant. Cela n'est pas encore trop contraignant ici, puisqu'il suffit d'écrire la fonction `draw` de la sorte :

```
let draw points triangle =
    (if triangle = Obtuse then set_color red
    else set_color blue);
    fill_poly (iof_array points);
    move points.(0);
    set_color black;
    line points.(1);
    line points.(2);
    line points.(0)
;;
```

1.2 Améliorations

Le premier problème du code précédent est que les côtés des triangles sont tous tracés deux fois. Pour y remédier, il suffit, lors de chaque subdivision d'un triangle, de ne tracer que les séparations entre les nouveaux triangles obtenus (et de ne pas oublier les cotés du triangle initial). Il apparaît alors qu'il est necéssaire de tracer ces lignes après que les triangles soient remplis, afin que celles ci ne soient pas effacées.

La fonction `divide` est en réalité un simple parcours en profondeur de l'arbre des divisions des triangles. Plutôt que d'afficher directement le pavage, il semble plus interressant de montrer la division succéssive des triangles pendant la construction du pavage. Survient alors une nouvelle difficulté : l'algorithme de parcours en profondeur décrit précédemment ne convient plus.

Une première solution est d'implémenter un parcours largeur et de marquer une pause avant d'entamer chaque nouvelle profondeur de l'arbre. Il faut pour cela maintenir une structure de file. Une des conséquences de ce choix d'implémentation est que la complexité mémoire est un $\Theta(2^n)$ où n est le nombre de générations et non plus un $\Theta(n)$ correspondant à la profondeur de la ple de récursion. Par ailleurs, il est alors nettement plus difficle d'afficher les couleurs des triangles dessinés à chaque génération et de ne tracer chaque coté qu'une seule fois. En effet, à chaque nouvelle génération, la coloration des triangles efface les cotés de ceux ci si ils ont été précédemment tracés.

Une seconde solution est d'effectuer plusieurs parcours profondeur, en augmentant le nombre de génération à chaque fois, et en marquant une pause entre chaque nouveau parcours. Cela permet de conserver une complexité mémoire linéaire en le nombre de générations et d'afficher les couleurs des triangles à chaque génération. De plus, il est possible de partir d'un triangle initial de petite taille et d'effectuer une homothétie à la fin de chaque parcours, afin de donner l'impression que le pavage s'expand. La subdivision d'un grand triangle initial, en contrepartie d'une certaine simplicité et efficacité, nécessite de prévoir le nombre de généra-

tions à l'avance, problème que l'on évite ainsi. L'implémentation de l'homothétie se fait en modifiant la fonction `draw` :

```
let homothety tab factor =
    Array.map (fun (x,y) -> (width/.2. +. (x -. width/.2.)*.factor,
                             height/.2. +. (y -. height/.2.)*.factor)) tab
;;

let draw points triangle homo_factor=
   (if triangle = Obtuse then set_color red
   else set_color blue);
   if homo_factor = 1. then fill_poly (iof_array points)
   else fill_poly (iof_array (homothety points homo_factor))
;;
```

En revanche, du fait que l'on recommence systématiquement le parcours depuis le début, deux fois plus de calculs sont effectués.

2 Hanoi

2.1 Principe : algorithme de base

Le problème des tours de Hanoi est essentiellement récursif. Pour déplacer n disques du premier poteau jusqu'au troisième il suffit de savoir en déplacer n-1 sur le deuxième poteau (étape 1), puis de déplacer le n-ième disque sur le troisième poteau (étape 2) et enfin de déplacer encore une fois les n-1 disques sur le troisième poteau (étape 3).

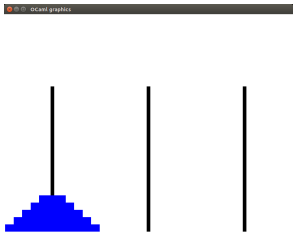


FIGURE 5 – État initial

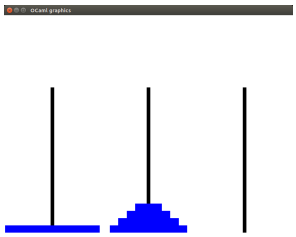


FIGURE 6 – Fin étape 1

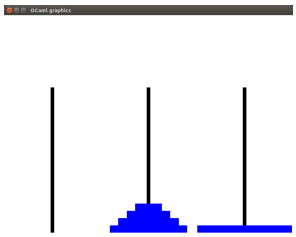


FIGURE 7 – Fin étape 2

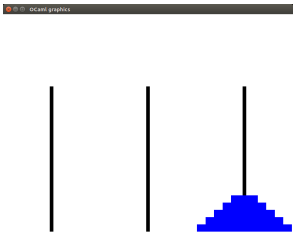


FIGURE 8 – État final

Nous avons choisi pour représenter la situation un tableau de piles `rods` : chaque pile du tableau représente chacun des poteaux et contient les disques qui y sont empilés.

La seule complication du code est le choix du poteau temporaire utilisé pour chaque déplacement de disques. La première implémentation réalisée repose sur la fonction suivante `choose` :

```
let choose a b =
  if a = b then failwith "a␣and␣b␣are␣equal"
  else if a = 0 then
         if b = 1 then 2
         else 1
  else if a = 1 then
         if b = 0 then 2
         else 0
  else if a = 2 then
         if b = 0 then 1
         else 0
```

```
    else failwith "a␣or␣b␣are␣not␣between␣0␣and␣2"
;;
```

Cette fonction prend en argument deux éléments distincts de 0, 1, 2 et renvoie le troisième et est utilisée pour choisir automatiquement dans la fonction `move` le poteau temporaire à utiliser :

```
let rec move rods num_discs orig_rod dest_rod =
  if num_discs = 1 then
        move_disc rods orig_rod dest_rod
  else
      (
        let temp_rod = choose orig_rod dest_rod in
        move rods (num_discs - 1) orig_rod temp_rod;

        move_disc rods orig_rod dest_rod;

        move rods (num_discs - 1) temp_rod dest_rod;
      )
;;
```

Lors de l'implémentation de la fonction résolvant le problème des tours de Hanoi à n poteaux, nous nous sommes rendu compte que la fonction `choose` était difficilement généralisable. Nous avons donc choisi de passer le poteau intermédiaire en argument à la fonction `move`.

Cette modification permet de choisir comme l'on veut le poteau intermédiaire, ce qui est nécessaire pour la version généralisée de l'algorithme.

2.2 Analyse du problème

Il est possible de calculer le nombre minimal de déplacements à effectuer pour résoudre le problème.

Si l'on veut déplacer n disques du premier piquet au dernier, il faut nécessairement déplacer le plus grand. Or il n'est possible de le déplacer que si les (n-1) disques plus petits ne sont pas sur celui-ci. Puisque l'on veut déplacer le plus grand disque sur le dernier piquet, les (n-1) disques doivent se trouver empilés en ordre décroissant de taille sur le piquet central ; dans quel cas on peut déplacer le grand disque sur le dernier poteau, et déplacer encore les autres disques sur le dernier piquet.

On a ainsi la relation de récurrence suivante, qui se résout facilement :

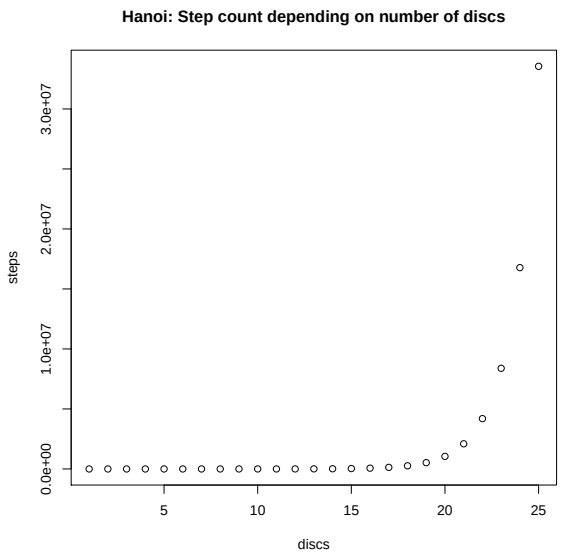
Initialisation $M(1) = 1$

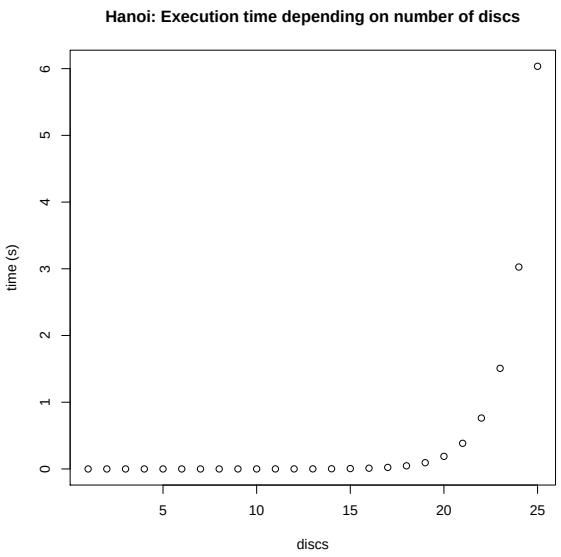
Relation de récurrence $M(n) = 2M(n-1) + 1$

Relation générale $M(n) = 2^n - 1$

Le nombre de déplacements de disques effectués pas l'algorithme dans le cas où il n'y a que trois piquets est optimal.

Les résultats expérimentaux sont en accord avec le calcul théorique : le nombre de mouvements est fonction exponentielle du nombre initial de disques.





2.3 Principe : algorithme généralisé

Pour ce problème, l'implémentation choisie est la deuxième présentée.

Cette fois il s'agit d'utiliser au mieux les poteaux supplémentaires. Nous avons calculé le nombre de disques pouvant être sur les poteaux intermédiaires, pour répartir ces disques sur chacun des poteaux intermédiaires en utilisant comme poteau temporaire le dernier (étape 1), ensuite il faut déplacer le disque le plus grand sur le dernier poteau (étape 2), et finalement il suffit de réaliser le parcours inverse de celui effectué dans l'étape 1, et de regrouper les disques intermédiaires sur le dernier poteau (étape 3).

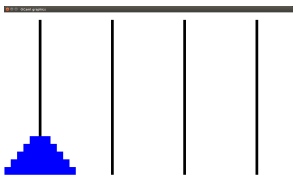


FIGURE 9 – État initial

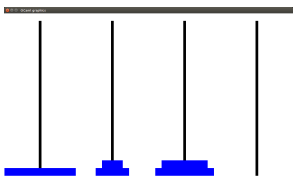


FIGURE 10 – Fin étape 1

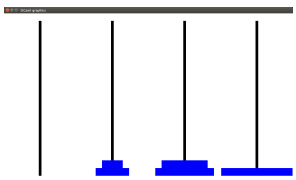


FIGURE 11 – Fin étape 2

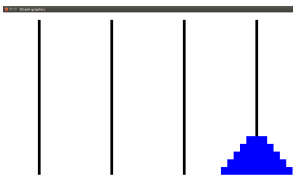


FIGURE 12 – État final

Ainsi beaucoup moins de déplacements sont nécessaires pour la résolution du problème.

2.4 Affichage

Nous avons créé un affichage grâce au module `Graphics` de OCaml qui permet de visualiser la résolution du problème à n disques et p poteaux pour n et p raisonnables.

De plus l'affichage s'adapte à la taille de la fenêtre :

La taille des disques est donc variable et dépend de plusieurs facteurs : au moins il y a de poteaux et au plus la fenêtre est large, au plus les disques sont larges ; au plus il y a de disques et au plus la fenêtre est grande, au plus les disques sont hauts.

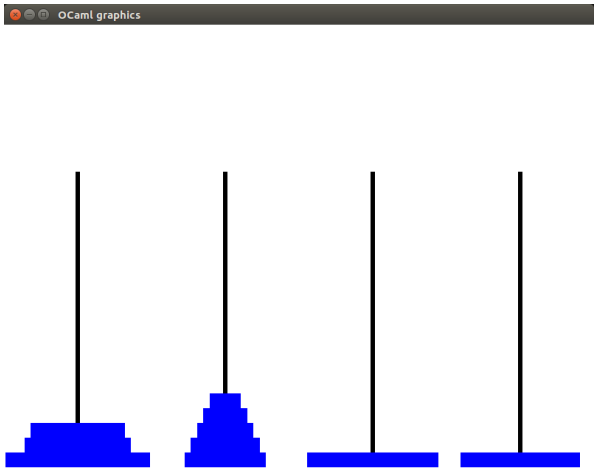


FIGURE 13 – Affichage de `hanoi 10 4`

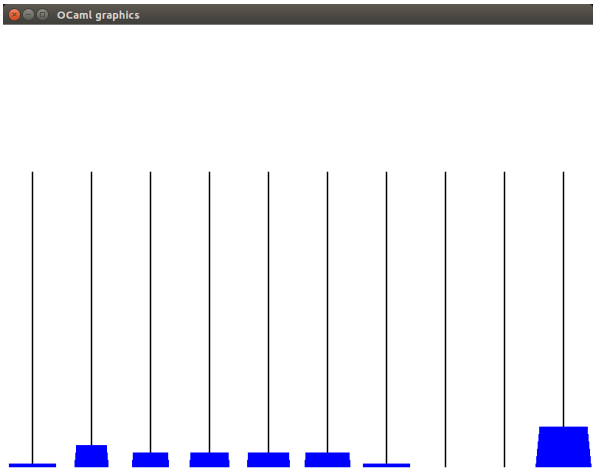


FIGURE 14 – Affichage de `hanoi 35 10`

FIGURE 15 – 1200 x 200

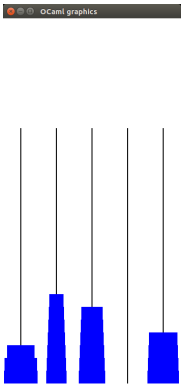


FIGURE 16 – 350 x 750

2.5 Améliorations

Nous nous sommes rendus compte qu'il était possible d'améliorer l'algorithme généralisé, puisqu'on observe que à chaque fois qu'un pile de disques est déplacée, seul le premier ou le dernier piquet est utilisé comme piquet temporaire, alors que l'on dispose de plusieurs piquets libres.

Conclusion

— Nous avons implémenté le programme dessinant le pavage de Penrose, à l'aide d'un parcours en profondeur ou d'un parcours en largeur, avec la possibilité d'utiliser une homothétie pour agrandir la figure à chaque génération ; puis le programme résolvant le problème des tours de Hanoi simple et généralisé, avec affichage graphique.

— Il serait intéressant d'implémenter pour les pavages de Penrose une version où l'un des triangles serait fixe au cours des générations afin de créer l'illusion de la construction du motif autour d'un seul triangle. Pour les tours de Hanoi, il serait possible de corriger l'algorithme généralisé et utiliser tous les piquets libres à notre disposition afin de résoudre le problème en moins de déplacements.

Auto-Évaluation

Forces. L'avantage principal de notre implémentation du pavage de Penrose est que le type de parcours effectué présente des avantages différents, et ce choix est par conséquent laissé à l'utilisateur. Par ailleurs la version finale avec homothétie est assez aboutie et pourrait par exemple être utilisée comme économiseur d'écran. En ce qui concerne la partie consacrée au Tours de Hanoi je pense que le point fort est l'affichage très flexible que nous avons implémenté.

Faiblesses. Pour Penrose, les triangles en dehors de l'écran sont tout de même traités, ce qui ralentis considérablement le code quand le nombre de générations est élevé. D'autre part, l'homothétie pourrait se faire de manière plus progressive, pour donner l'impression d'un zoom continu et fluide. Le manque de modularité et de séparation du code s'est vu très pénible au moment d'harmoniser les différentes versions du code à envoyer (algorithme basique, généralisé et avec affichage).

Opportunités. Ce projet m'a appris l'importance de la modularité du code, pour les raisons avant mentionnées, et aussi à gérer un projet mené par plusieurs personnes, ce qui peut s'avérer plus difficile que prévu.

Menaces. Il est trop souvent facile de remplir le code de littéraux et de constantes, mais quand il faut généraliser le code déjà écrit, cet effort non effectué fait perdre énormément de temps.