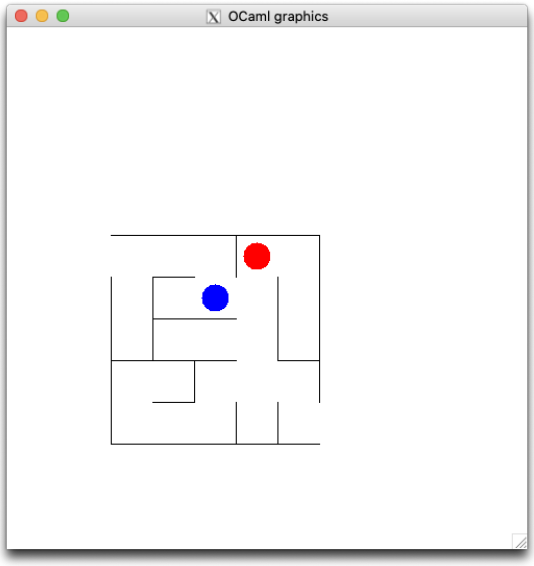
Projet de programmation fonctionnelle

Etienne Lozes

Le projet est à rendre pour le **13 décembre minuit**. Il est à faire en binôme et à rendre sur Moodle.

1 Description générale

Le but du projet est de programmer un petit jeu qui se présente sous la forme suivante



Le joueur humain déplace le disque bleu (que l'on appelera par la suite pacman) à l'aide du clavier. Initialement pacman est dans l'angle en haut à gauche et il doit se rendre dans l'angle en bas à droite du labyrinthe sans traverser de mur. Pour mettre un peu de piment, le disque rouge (que l'on appelera par la suite le fantome) pourchasse pacman et tente de lui barrer la route. Initialement, le fantome est dans l'angle en haut à droite, et à intervalle de temps régulier (par exemple, toutes les deux secondes) il se déplace d'une case de façon à se rapprocher de pacman.

Le projet est découpé en trois parties.

1. Génération aléatoire du labyrinthe et affichage. Il s'agit d'une partie de difficulté moyenne mais très guidée, qui compte pour environ 50% de la note finale.

2. Gestion du clavier et du pacman. Il s'agit d'une partie relativement facile qui compte pour environ 25% de la note finale.

3. Gestion du fantome. Il s'agit d'une partie un peu plus difficile mais très guidée qui compte pour environ 25% de la note finale.

2 La génération du labyrinthe

On souhaite pouvoir générer un labyrinthe de dimension arbitraire - ultimement cela sera le joueur qui fixera la dimension du labyrinthe au début de la partie.

On souhaite que le labyrinthe généré soit "difficile" au sens où il existe un seul chemin entre le départ et l'arrivée, et plus généralement entre deux cases quelconques du labyrinthe. Suivant la terminologie de wikipedia, un tel labyrinthe sera dit *parfait*. Pour garantir que le labyrinthe généré est parfait, on applique l'algorithme suivant: on part de la grille pleine avec des murs partout, puis on retire des murs un par un en les tirant au hasard. Cependant, un mur n'est retiré que si cela permet de créer un chemin entre les deux cases voisines qu'il sépare; si les deux cases sont déjà reliées par un chemin détourné, on ne retire pas le mur et on tire au hasard un nouveau mur, et ce jusqu'à en trouver un convenable. Au total, on retire un nombre de murs égal au nombre de cases moins 1. Une animation qui explique l'algorithme est visible sur la page wikipedia consacrée à la génération de labyrinthes.[1]

La partie subtile de l'algorithme est le test pour savoir si les deux cases sont déjà reliées par un chemin. Pour cela, on commence par numéroter les cases: les cases de la première ligne ont les numéros de 0 à $l-1$, où l désigne la largeur du labyrinthe ($l = 5$ sur l'exemple); les cases de la ligne suivante ont les numéros l à $2l-1$, etc. Ainsi la case du pacman dans la figure ci-dessus sera la numéro 7.

On utilise ensuite une structure de données `uf` dite *union-find* car elle permet deux opérations:

- `find uf n` : cette opération renvoie le numéro d'une case qui est actuellement reliée par un chemin à la case n. La particularité de find est que toutes les cases d'une même composante connexe prennent la même valeur pour find. Autrement dit, pour tester si deux cases n et m sont reliées par un chemin, il suffit de tester si `find uf n = find uf m`.

- `union uf n m`: cette opération modifie la structure de données uf (et donc le calcul de find) de sorte que les cases n et m soient maintenant considérées comme faisant partie de la même composante connexe. Cette opération est utilisée quand on retire le mur qui sépare les cases n et m pour fusionner les deux composantes connexes en une seule.

[1] https://fr.wikipedia.org/wiki/Modélisation_mathématique_de_labyrinthe

La structure de données `uf` se présente comme une forêt avec un arbre par composante connexe (en interne, on code cette forêt à l'aide d'un tableau). Le résultat de `find uf n` est la racine de l'arbre dans lequel se trouve n. Pour gagner en efficacité, on cherche à avoir des arbres aussi peu profonds que possible, ce qui fait appel à deux heuristiques: la compression de chemin et l'équilibrage au moment de la fusion. Tous les détails sur la structure de données union-find se trouvent sur la page wikipedia.[2]

Pour représenter le labyrinthe, on va remplir un tableau de booléens à trois entrées: `mur_present.(d).(x).(y)` sera vrai si le mur de direction `d` (0 pour vertical, 1 pour horizontal), ligne `x` et colonne `y` est présent. Par exemple, sur le labyrinthe représenté ci-dessus, on a `mur_present.(0).(2).(0) = true` car le mur vertical de la colonne 2 et la ligne 0 est présent - il s'agit du mur à gauche du fantome, on compte les colonnes à partir de la première rangée de murs interne, en sautant la rangée de murs externes.

Pour tirer un mur au hasard, on pourra tirer un entier au hasard entre 0 et #murs-1, où #murs est le nombre total de mur, soit $(l-1)h + l(h-1)$.

```
let mur_au_hasard l h = (* renvoie un triplet (d, x, y) *)
   let n = Random.int ((l-1) * h + l * (h-1)) in
   if n < (l-1) * h
   then (0, n mod (l-1), n / (l-1))
   else let n2 = n - (l-1) * h in
      (1, n2 mod l, n2 / l)
```

Parlons maintenant de l'affichage. Vous allez utiliser la librairie `Graphics` que vous avez déjà vue à certains TP, et dont la documentation se trouve en ligne.[3] Il faudra compiler votre programme avec quelque chose comme `ocamlc graphics.cma projet.ml`. Pensez à faire un appel à `read_key()` en fin de programme pour que l'affichage soit visible tant que vous n'avez pas pressé un bouton. Vous aurez essentiellement à utiliser les fonctions `open_graphics`, `moveto`, et `lineto` pour le tracé du labyrinthe. Il faudra d'abord tracer le pourtour du labyrinthe, puis parcourir le tableau `mur_present` qui code le labyrinthe, et pour chaque mur tel que `mur_present.(d).(x).(y)` est vrai, il faudra afficher le mur correspondant. Vous aurez à manipuler les paramètres suivants, qui devront être instanciés par des constantes au dernier moment (idéalement, il faudrait les demander à l'utilisateur avant de commencer):

- `upleftx` et `uplefty` : les coordonnées de l'angle en haut à gauche du labyrinthe
- `taille_case` : la largeur (et la hauteur) d'une case en pixel
- l et h, la largeur et la hauteur du labyrinthe en nombre de cases

Les questions suivantes visent à vous guider. Vous n'êtes pas obligé d'y répondre si vous voulez faire autrement, l'essentiel est d'arriver à générer le labyrinthe et à l'afficher.

[2] `https://fr.wikipedia.org/wiki/Union-find`
[3] `https://ocaml.github.io/graphics/graphics/Graphics/index.html`

1. Définissez un module `UF` qui implémente une structure de données union-find de type abstrait `t` et offre trois fonctions:

   - `create n` renvoie une nouvelle structure de données `uf` de type `t`. Le paramètre n désigne les entiers de 0 à $n-1$ qui vont être fusionnés. Initialement, `find uf i` renvoie i pour chaque i dans cette structure de données (il n'y a encore eu aucune fusion).
   - `find uf n` qui renvoie le représentant de la classe de n dans la structure d'union-find `uf`
   - `union uf n m` qui modifie `uf` pour faire fusionner les classes de n et m.

2. Définissez la fonction `cases_adjacentes l h (d,x,y)` qui au mur `(d,x,y)` associe le couple (i, j) des numéros des deux cases séparées par ce mur. l et h sont la largeur et la hauteur du labyrinthe en nombre de cases. Par exemple, sur un labyrinthe 5×5 comme dans l'exemple,
   `cases_adjacentes 5 5 (0,0,2)` renvoie `(2,3)`, tandis que
   `cases_adjacentes 5 5 (1,0,1)` renvoie `(1,6)`.

3. Définissez la fonction `generate_lab l h` qui renvoie un tableau `mur_present` à trois entrées codant les murs du labyrinthe. Pour vous aider un peu, on vous donne cette fonction en pseudo-code

```
allouer le tableau mur_present en mettant toutes les cases à true
allouer uf avec UF.create (l*h)
pour i allant de 1 à l*h-1 faire
    soit m un mur au hasard
    soit i et j les cases adjacentes séparées par m
    si UF.find uf i = UF.find uf j
    alors reprendre la boucle au début sans incrémenter i
    sinon faire
       UF.union uf i j
       avec m=(d,x,y) faire
         mur_present.(d).(x).(y) <- false
    fin sinon
fin pour
renvoyer mur_present
```

4. Définissez la fonction `trace_pourtour upleftx uplefty taille_case l h` qui dessine le pourtour du labyrinthe.

5. Définissez la fonction `trace_mur upleftx uplefty taille_case (d,x,y)` qui dessine le mur `(d,x,y)`.

6. Définissez la fonction
   `trace_lab upleftx uplefty taille_case l h mur_present` qui dessine le labyrinthe codé par le tableau de booléens `mur_present`.

Si votre code est correct, vous devriez voir s'afficher un labyrinthe parfait, sans partie fermée et sans "salles" puisqu'il y a toujours exactement un seul chemin entre deux cases.

3 Le pacman

Pour gérer le pacman, il sera commode d'avoir une variable globale `case_pacman` qui contient le numéro de la case du pacman, initialement en haut à gauche sur la case 0.

```
let case_pacman = ref 0
```

Cette variable globale sera modifiée à chaque pression d'un bouton. La bibliothèque Graphics offre la fonction `read_key`. Vous ne pourrez malheureusement pas récupérer les touches de flèches, la fonction ne renvoyant que les caractères imprimables. Le plus simple sera donc de déplacer le pacman avec des touches correspondant à des lettres. Sur mon clavier américain, j'ai pris 'a' pour gauche, 'w' pour haut, 's' pour droit, et 'z' pour bas.

La fonction `read_key` sera appellée dans une boucle while infinie. En dehors de l'appel à la fonction `read_key`, les actions effectuées dans cette boucle sont:

- analyse de la touche pressée et vérification que le mouvement est autorisé (pas de mur)
- mise à jour de `case_pacman`
- mise à jour de l'affichage
- test si la case finale est atteinte, et sortie du jeu en affichant "GAGNE" si c'est le cas

Pour mettre à jour l'affichage, il y a deux stratégies possibles: soit vous effacez toute la fenetre avec `clear_graph` et vous redessinez tout, soit vous effacez seulement la dernière position du pacman (en dessinant par exemple un disque blanc dessus) et vous dessinez pacman à sa nouvelle position.

Bonus: si vous voulez, vous pouvez émettre un son quand pacman se cogne à un mur. Vous pouvez aussi rajouter une touche pour terminer la partie sans fermer la fenetre (par exemple la touche 'q' pour 'quitter').

4 Le fantome

Comme pour le pacman, il sera commode pour le fantome d'avoir une variable globale qui contient le numéro de la case sur laquelle il se trouve. Initialement, le fantome se trouve sur la case en haut à droite de numéro $l-1$. En supposant que la largeur du labyrinthe est elle aussi stockée dans une variable globale l, on aura

```
let case_fantome = ref (l-1)   (* ou !l -1 *)
```

Le coeur du code du fantome est une boucle while infinie qui exécute les actions suivantes:

- s'endormir un laps de temps (par exemple 2 secondes)
- calculer la prochaine case où se rendre et mettre à jour `case_fantome`
- mettre à jour l'affichage
- tester si la case du pacman est atteinte et si oui quitter la partie en affichant "PERDU"

Le test pour savoir si la partie est perdue doit d'ailleurs être fait aussi dans la boucle infinie du pacman car cela peut être un mouvement de pacman vers le fantome qui met fin à la partie. C'est l'occasion de refactoriser le code et d'écrire des fonctions pour les parties communes aux deux boucles: affichage, test si perdu, et sortie du programme.

Vous vous demandez peut-être comment il peut y avoir deux boucles infinies dans le programme... bonne question! La boucle infinie du fantome doit en effet être exécutée en tâche de fond. Pour ce faire, il faut la faire exécuter par un thread différent du thread principal, le thread principal étant celui qui exécute la boucle infinie du pacman.[4]

Pour illustrer l'utilisation des threads et d'un minuteur, voici un programme assez analogue à ce qui nous intéresse. Ce progamme attend la saisie d'un texte avec `read_line` et l'affiche (l'analogue du pacman) tandis qu'en tâche de fond un autre thread affiche "hello" toutes les deux secondes (l'analogue du fantome).

```
let rec affiche_hello () =
  Unix.sleep 2;
  print_string "hello" ; print_newline();
  affiche_hello()

let _ = Thread.create affiche_hello ()

let () = (* thread principal *)
  let s = read_line () in
  print_string s
```

Pour compiler ce programme, taper

`ocamlc -thread unix.cma threads.cma hello.ml`

où `hello.ml` est le nom du fichier qui contient ce code. De même, pour compiler votre projet, il vous faudra reprendre ces options ainsi que la librairie graphics:

`ocamlc -thread graphics.cma unix.cma threads.cma projet.ml`.

[4] on pourrait aussi mettre la boucle du pacman dans un thread secondaire, mais cela n'a aucun intérêt, et il faut veiller à ce que le thread principal ne termine pas pour ne pas risquer de terminer tous les autres threads.

Dans un premier temps, simplifiez le calcul de la prochaine case pour pouvoir tester votre programme avec le thread; par exemple vous pouvez faire un fantome qui traverse les murs et qui se dirige en "ligne droite" vers le pacman.

Pour être tout à fait juste, le programme comportera un petit bug que l'on va s'autoriser: si la mise à jour du pacman et du fantome se produisent en même temps, on peut observer des comportements bizarres, par exemple le fantome peut être affiché à deux positions différentes simultanément. Pour résoudre ce problème il faudrait utiliser un verrou et vous expliquer comment cela fonctionne. Ce n'est pas très compliqué mais c'est hors programme; on va simplifier et ne pas se soucier de ce bug qui a de toute façon très peu de chance de se produire.

Passons maintenant au calcul de la prochaine case où doit se rendre le fantome. A tout moment le pacman et le fantome sont reliés par un chemin qui est unique. La case sur laquelle le fantome doit se rendre est la première case de ce chemin. Pour la calculer, on va supposer que l'on connait pour chaque case c la liste voisines(c) des cases voisines accessibles en un pas; on pourra remplir un tableau `voisines` à l'intérieur de la fonction `generate_lab` et le renvoyer en même temps que le tableau `mur_present`.

La fonction principale pour la recherche de chemin est la fonction `est_relie src dst evite voisines` qui renvoie vrai s'il existe un chemin allant de la case `src` à la case `dst` sans passer par la case `evite` en premier. On a besoin de rajouter cette case à éviter pour ne pas créer une boucle infinie dans la recherche. La fonction `est_relie src dst evite voisines` procède comme suit:

```
si src=dst alors renvoyer true fin si.
pour toute case c dans voisines.(src), c<>evite, faire
   si est_relie c dst src voisines alors renvoyer true fin si
fin pour
renvoyer false
```

A vous de jouer.