Programmation Avancée 2020

Devoir à la Maison

Le Pingouin

3 avril 2021

Ce devoir à la maison tient lieu de partiel. Il est à rendre pour le 19 Avril.

Vous incarnez un pingouin qui ne sait pas nager, mais qui aime bien manger. Vous êtes seul sur une banquise représentée par une grille hexagonale. Une case peut y être recouverte de glace, sinon il n'y a que de l'eau. Sur chaque case de glace, vous pouvez pêcher un poisson. Vous pouvez ensuite vous déplacer sur une autre case en glissant en ligne droite selon l'un des six axes de la grille, aussi loin que vous voulez mais en restant sur la glace — vous ne savez pas nager ! La case que vous avez quittée fond à ce moment, et la glace y devient de l'eau.

L'histoire ne dit pas si à la fin le pingouin mort affamé ou noyé. . . En tout cas, votre objectif est de déterminer le parcours qui vous permettra de manger un maximum de poissons [1].

Considérons par exemple la banquise ci-dessous à gauche, où la position de départ du pingouin est représentée par `#`, et les cases de glace par `*` :

```
  *   *       |       *   *
   * *        |        e f
* * # * *     |     * d # g h
   * *        |        c b
  *   *       |       *   a
```

Sur cet exemple, le pingouin peut manger tous les poissons sauf quatre en se déplaçant suivant le chemin `a,b,...`, décrit ci-dessus à droite.

1 Instructions générales

Une base de code OCaml est disponible en ligne. Vous y trouverez tout d'abord des interfaces (fichiers `.mli`) qui vous guideront. Vous ne devez en aucun

1. Ce projet est inspiré du jeu de plateau "Pingouins" (en anglais, "Hey, that's my fish !") où plusieurs joueurs contrôlent chacun plusieurs pingouins, avec le même objectif.

cas modifier ces interfaces, sauf pour ajouter quelques fonctions (déboguage, utilitaires, etc.) Les interfaces sont (partiellement) documentées, donnant des détails qui ne sont pas répétés ci-dessous : pensez à systématiquement aller les lire.

Je vous fournis aussi un `Makefile`, qui devrait compiler automatiquement les fichiers `*.ml` que vous créerez, et qui liera tous vos modules pour obtenir l'exécutable `pingouins` dès que le fichier `main.ml` sera créé. En début de `Makefile` vous trouverez commentés des paramètres de compilation : par défaut la compilation produit du code adapté au déboguage ; une fois sûrs de votre code, vous pourrez le compiler avec des options qui donneront de meilleures performances.

Enfin, plusieurs problèmes sont définis dans les fichiers `problems/*`, représentant des grilles hexagonales selon une syntaxe évidente, et où le symbole `#` représente la position de départ du pingouin.

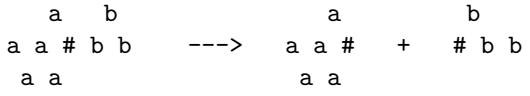
Le projet devra être rendu par mail sous la forme d'un tarball contenant les fichiers source finaux (`*.ml` et `*.mli`) ainsi que le `Makefile`, un éventuel `README`, mais aucun fichier compilé.

La note sera basée sur la correction, la clarté et enfin l'efficacité du code écrit en réponse aux questions de l'énoncé. Concernant la clarté, les commentaires sont bien sûr attendus, dans le code ou dans un fichier `README`.

2 L'algorithme

Pour expliquer l'algorithme que nous allons mettre en oeuvre, partons de l'algorithme naïf consistant à explorer tous les cheminements possibles de notre pingouin. Cette exploration déroule un arbre qui devient rapidement très gros. À chaque noeud de l'arbre on trouve une configuration composée d'une banquise partiellement fondue et d'une nouvelle position du pingouin.

On remarque aisément que plusieurs noeuds vont correspondre à la même configuration ; il sera bénéfique de ne pas explorer deux fois la même configuration. Pour éviter de différencier artificiellement des noeuds, il va être important de ne pas garder dans une configuration des cases qui sont évidemment inaccessibles. Plus précisemment, les configurations seront toujours des composantes connexes de banquise. Quand le pingouin va se placer en une case qui découpe la banquise en deux composantes connexes, cela va donner lieu à deux configurations possibles :



Ensuite, il est clair que si l'exploration d'une partie de l'arbre nous a donné une solution possible de taille n, on peut couper court à des explorations d'autres parties de l'arbre qui n'ont plus aucune chance de donner d'aussi bonnes solutions, simplement parce qu'il ne reste plus assez de cases de banquise dans la configuration associée.

Enfin, l'ordre d'exploration des différentes parties de l'arbre va avoir un impact important. Par exemple, sur une configuration pour laquelle il existe une solution qui consomme toutes les cases, si l'on fait une recherche qui favorise le chemin le plus long pour lequel on n'a encore "perdu" aucune case de banquise, on trouvera rapidement une solution optimale, et on pourra se passer d'explorer le gros de l'arbre.

Pour implémenter cet algorithme, nous utiliserons une représentation efficace des configurations (ensembles de cases) et une file de priorité pour représenter les configuration à explorer.

3 Structure de grille hexagonale

La première étape du projet est d'implémenter le module `Hex`, qui définit un système de coordonnées et de déplacements pour les grilles hexagonales. Les grilles hexagonales seront représentées par des matrices, comme illustré ci-dessous :

```
 a b c d e      a b c d e     [| [| a ; b ; ... |] ;
e f g h i  -->  e f g h i -->    [| e ; f ; ... |] ;
 j k l m n      j k l m n        [| j ; k ; ... |] |]
```

La case `a` de cette grille a pour position $(0, 0)$, la case `g` a pour position $(1, 2)$. Dans le code, on représentera la seconde case par `grid.(1).(2)`. Sur cet exemple, le déplacement `NE` (Nord-Est) à partir de la case `g` nous mènera à la case `c`.

Question. Implémentez le module `Hex`, sauf la fonction `from_channel`. La fonction d'affichage d'une grille (`pp_grid`) doit simplement produire un résultat proche des fichiers d'exemple fournis, sans les balises `<problem>`. Noter que cette fonction peut s'utiliser simplement ainsi :

```
Format.printf
  "Ma␣grille␣(%d␣x␣%d):\n%a\n"
  (Array.length grid)
  (Array.length grid.(0))
  pp_grid grid ;
```

Je vous recommande fortement de ne pas utiliser `Printf` mais uniquement `Format` dans votre code.

Question. Implémentez maintenant la fonction `from_channel`. Celle ci devra lire sur l'entrée standard un problème dans la syntaxe des fichiers `problems/*`. Elle renverra la position de départ du problème (représentée par `#`) et une grille de booléens, où `true` dénote une case de glace, représentée par un caractère autre que `' '` dans le fichier d'entrée. Pour vous faciliter la vie ensuite, et éviter d'avoir à écrire du code supplémentaire pour éviter les déplacements hors de la grille, je vous recommande de faire en sorte que `from_channel` produise toujours une grille qui contienne un pourtour de cases vides — ainsi le pingouin pourra

s'arrêter naturellement sur ces cases plutôt que se cogner sur des vérifications de bornes.

Question. Créez un fichier `main.ml` qui lit un problème sur l'entrée standard et l'affiche sur la sortie standard. Compilez, tester. Il ne serait pas inutile de profiter de ce premier exécutable pour tester votre code relatif aux mouvements.

4 Ensembles finis

La première structure de donnée dont nous aurons besoin est une notion d'ensemble dont les éléments forment un type fini. Au final, il s'agira de positions sur une grille fixée, dont les bornes sont donc connues. Mais nous allons implémenter la structure de données de façon générique, en utilisant un foncteur, pour la spécialiser plus tard.

Étant donné que l'ensemble de positions restera assez limité, une implémentation très efficace des ensembles est possible sous la forme de bitmaps. Plus précisemment, je vous recommande d'utiliser un tableau d'entiers 64 bits (`Int64.t array`).

Question. Implémentez le module `Bitset` ; tout est expliqué dans le `.mli`.

Notez bien que le cardinal d'un ensemble doit être calculé en temps constant. Pour la fonction `add`, vous pouvez supposer (pour vous simplifier la vie et éviter une vérification inutile) que l'élément à ajouter n'est pas déja présent — cependant, pour éviter de vous prendre les pieds dedans, je vous recommande très fortement d'agrémenter ces fonctions (et beaucoup d'autres) d'un `assert`, que vous pourrez toujours désactiver à la compilation une fois sûrs de l'absence de bugs.

5 Files de priorité

La seconde et dernière structure de données dont nous auront besoin sera une file de priorité. Votre tâche sera de l'implémenter sous la forme d'un tas binaire dans le module `Priority`.

Plus précisemment, vous implémenterez un foncteur, qui prend en entrée un module décrivant le type des clés, muni d'une opération `compare` suivant la convention OCaml.

Vous pourrez implémenter le tas binaire de façon efficace comme un tableau, vous devrez bien sûr prendre en compte les données et pas seulement les clés. De plus, comme vous codez en OCaml, vous ne pourrez laisser les cases du tableau non initialisées : notez, à ce propos, que la fonction `create` vous fournit des valeurs par défaut qui devraient vous être utiles.

Question. Implémentez le module `Priority`.

6 Recherche du meilleur chemin

Nous allons enfin implémenter l'algorithme décrit plus haut, dans le module `Paths`. Ce module est encore une fois un foncteur, dont l'argument donne simplement la grille initiale. Le module renvoyé par le foncteur contiendra notamment une instance de `Bitset.Make` sur le type des positions dans cette grille initiale.

Question. Implémentez la fonction `Paths.accessible`, dont la spécification est donnée dans le `.mli`.

Question. Implémentez les fonctions `Paths.disconnected` et `Paths.split`.

Question. Implémentez les autres fonctions utilitaires de l'interface.

Question. Afin de ne pas revisiter deux fois la même configuration dans votre algorithme de recherche, il vous faudra utiliser une table associant un état à toutes les configurations déja rencontrées ; vous êtes libres de choisir la représentation qui vous semble la plus efficace pour cette table. Implémentez cette fonctionalité dans le module `Paths.Make(...).HMap`.

Question. Implémentez l'algorithme final, `Paths.Make(...).maxpath`. À titre d'indication vague, le type des clés à utiliser comme priorités devra comporter (1) le nombre de cases déja consommées et (2) le nombre de cases restantes dans la configuration.

Question. Mettez le tout en place dans le fichier `main.ml`, où l'on devra instancier le foncteur `Paths.Make` en fonction de la grille lue sur l'entrée standard. Il est souhaitable que votre code lise le problème, l'affiche, le résolve (avec ou sans debug, je vous laisse libre de choisir comment contrôler cela) et affiche la solution. On pourra de surcroit afficher des informations comme le temps de calcul, le nombre d'entrées ajoutées ou modifiées dans la file de priorité, etc.

Un bonus sera attribué aux codes les plus élégants, et au(x) plus efficace(s) !