On travaillera depuis la machine virtuelle ClefAgreg2019. On commencera par créer un fichier OCaml `TP20.ml`, initialement vide.

Attention : la partie 3 et la question 11. et suivantes manipulent des entiers sur (au moins) 49 bits, et donc cela ne marchera pas depuis `https://BetterOCaml.ml` qui manipule des entiers machines de JavaScript, sur 32 bits et pas 64 bits.

On rappelle qu'on compile ce fichier avec `COMPILATEUR = ocamlc` ou `ocamlopt` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal).

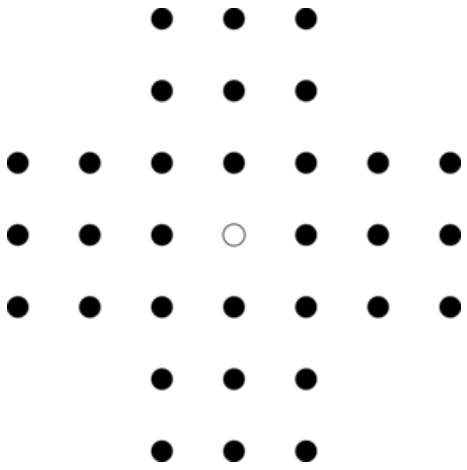
```
$ COMPILATEUR -o TP19.exe TP19.ml
$ ./TP20.exe
```

Introduction

On s'intéresse dans ce TP au jeu de solitaire, et à sa résolution par exploration exhaustive.

Ce jeu de solitaire n'est pas le jeu de carte mais un jeu de plateau comme le montre la figure ci-dessus (photo du produit sur Philibert), qui comme son nom l'indique se joue tout seul.

Il faut utiliser un plateau carré de taille $n \times n$ avec $n = 7$, comportant 33 emplacements possibles, et initialement 32 pions, représentés par des ronds noirs •, et un emplacement vide au centre représenté par ∘ :



Les différents mouvements possibles consistent à passer d'une configuration • • ○ à ○ ○ •, et ainsi à diminuer d'exactement un pion le nombre total de pions. Ces configurations peuvent être rencontrées dans les directions horizontales ou verticales.

On considère que la partie est gagnée quand il n'y a plus qu'un pion sur le plateau, peu importe son emplacement.

Première implémentation naïve

On va représenter un plateau par les types OCaml suivants :

```
type case = Vide | Pion | Invalide;;
type plateau = case array array;;

(* Un plateau est une matrice 7x7, avec des cases invalides aux coins *)
let n = 7;;
```

En C, on pourrait utiliser un type `enum`, ou simplement un type `typedef case int;` et définir `case Vide = 0`, `case Pion = 1` et `case Invalide = 2` (ou n'importe quel triplet de valeurs).

1. Définir une fonction `print_plateau` qui affiche un plateau sous un format textuel lisible. On supposera que toutes les valeurs de type `plateau` sont bien de taille `n * n`.
2. Écrire une fonction `plateau_initial : unit -> plateau` qui renvoie un plateau correspondant à la configuration de départ.

Il devrait s'afficher comme cela (si vous avez choisi de représenter un `Vide` par le symbole `'o'` et un `Pion` par le symbole `#`) :

```
  ###
  ###
#######
###o###
#######
  ###
  ###
```

Un mouvement peut être assimilé à un triplet de coordonnées décrivant, dans l'ordre, les trois cases concernées. Il se trouve que la case centrale est toujours le milieu des deux autres, on peut donc se contenter de donner un couple de coordonnées (gauche/droite, ou bas/haut) pour décrire un mouvement.

```
type mouvement = (int * int) * (int * int);;
```

3. On commencera par introduire une fonction `possible : plateau -> (int * int) -> (int * int) -> bool` qui prend un plateau `p`, deux couples `(i1,j1)` et `(i3,j3)` de coordonnées, et calcule `(i2,j2)` la position au milieu des deux autres, et vérifie si le mouvement est valide (réfléchissez à ce que cela signifie sur les valeurs de `p.(i1).(j1)`, `p.(i2).(j2)` et `p.(i3).(j3)`).

4. Écrire maintenant une fonction `mouvements : plateau -> mouvement list` qui renvoie la liste des mouvements possibles sur le plateau passé en paramètre. L'ordre de la liste n'est pas important. Il ne faudra pas inclure des mouvements en doublon (en particulier un `mouvement = ((i1,j1), (i3,j3))` sera toujours de la gauche vers la droite (`i3 = i1 + 2` et `j3 = j1`) ou du bas vers le haut (`i3 = i1` et `j3 = j1 + 2`)).
5. Écrire une fonction `compte_pions : plateau -> int` qui renvoie le nombre de pions sur un plateau.
6. En déduire une fonction `est_gagnant : plateau -> bool` qui indique si un plateau correspond à une partie gagnante.
7. Écrire deux fonctions `applique` et `annule`, de type `plateau -> mouvement -> unit`, permettant de faire et défaire un mouvement. Si le mouvement n'est pas possible, `faire` ne fera rien.
8. Définir une exception paramétrique `Solution of (mouvement list)`.
9. Écrire une fonction `enumere : plateau -> mouvement list -> unit`, telle que `enumere pos chemin` énumère les plateaux accessibles depuis `pos` jusqu'à obtenir une solution, et sachant que `chemin` est la liste de `mouvement`s, du plus récent au plus ancien, qui ont conduit jusqu'à `pos`.

   En cas de succès, on produira une exception `Solution of mouvement list`, renvoyant la liste des mouvements ayant conduits à une solution.
10. En déduire une fonction `resout : unit -> mouvement list` qui renvoie une liste de mouvement permettant de résoudre le solitaire. Elle commencera par créer la `position` du plateau initial, puis appellera `enumere pos []` depuis un `chemin = []`. Si aucune exception `Solution liste` n'est déclenchée, il faudra déclencher `Not_found` (avec `raise Not_found`). On fera attention à renverser (`List.rev`) le chemin obtenu pour que le premier mouvement de la liste soit le premier mouvement effectué.

Attention, si vous essayez la fonction `resout` vous risquez d'attendre (très...) longtemps !

Même si c'est assez décevant, ce n'est pas très surprenant : ce code ne permet pas de calculer la solution ! En effet, il prend beaucoup trop de temps en raison du nombre de positions étudiées.

Amélioration par un cache des mauvaises positions

On se rend compte que de nombreuses positions sont réétudiées alors qu'on sait déjà qu'elles ne peuvent permettre d'aboutir à une solution. En effet, il y a souvent des coups indépendants pouvant être joués au même moment, ce qui fait qu'on peut aboutir à une même position de beaucoup de manière différente, ce qui augmente exceptionnellement le nombre d'appels récursifs. Pour ces raisons là, `resout` n'avait aucune chance de terminer en un temps raisonnable.

Une stratégie consiste a maintenir un ensemble de configurations mauvaises. Pour réaliser un tel ensemble, on va utiliser une table de hachage dont les clés sont les `position`s, et les valeurs `unit`. Si une `position` a une valeur associée dans la table (i.e. elle est présente l'ensemble des mauvaises `position`s), c'est qu'elle sera mauvaise.

Cela pose la question de la représentation persistante et immuable des positions. Une première stratégie peut consister à transformer le plateau en `case list list`. Cette stratégie est beaucoup trop coûteuse et elle ne permet pas de répondre instantanément.

On va profiter du fait qu'il n'y est que $n^2 = 49$ cases dans le plateau pour le représenter par un entiers sur 49 bits : $a_{00} + a_{10}2 + a_{20}2^2 + \ldots. + a_{60}2^6 + a_{01}2^7 + \ldots. + a_{66}2^{48}$, où $a_{ij} \in \{0, 1\}$ vaut 1 lorsqu'il y a un pion sur la j-ième ligne et la i-ème colonne, c'est-à-dire quand `p.(i).(j) = Pion` (et 0 sinon).

Indication : l'entier `1 lsl n` est 2^n. L'opérateur infixe `lsl` (left shift) signifie qu'on décale le chiffre `1` de `n` bits vers la gauche dans son écriture binaire. C'est plus rapide à calculer que calculer 2^n par multiplications successives.

11. Écrire une fonction `code : plateau -> int` qui renvoie le numéro associé à un plateau.

Vous pouvez tester votre solution sur le `plateau` produit par `plateau_initial ()`, qui devrait renvoyer un numéro égal à `124141717933596`.

Pour manipuler un ensemble de positions, on va utiliser une table de hachage avec le module `Hashtbl` de OCaml. Sa documentation est sur `https://ocaml.org/api/Hashtbl.html`, si besoin. On va définir la table `mauvaises` et deux fonctions suivantes (à recopier) :

```
let mauvaises  = Hashtbl.create 42;; (* taille initiale arbitraire *)
let ajoute   x = Hashtbl.add mauvaises x ();;
let contient x = Hashtbl.mem mauvaises x;;
```

L'appel à `ajoute x` rajoute `x` dans l'ensemble `mauvaises` des mauvaises positions, et `contient x` vérifie si `x` est dans cet ensemble.

12. Reprendre la fonction `enumere` avec un ensemble de mauvais codes, et écrire une fonction `enumere2` de même signature.
13. Conclure en écrivant une fonction `resout2` de même signature que `resout`.

Elle devrait être assez rapide, sur ma machine elle s'exécute en environ 8 secondes.

Vous pouvez vérifier que la liste de mouvements proposés contient 31 mouvements, ce qui est très logique car un mouvement retire exactement un pion du plateau, et qu'il y a au début 32 pions.

Extensions possibles

On peut mentionner quelques pistes d'extensions possibles :

— rajouter un affichage de la résolution au fur et à mesure ;

— déterminer la proportion de recalculs évités ;

— gérer les symétries des plateaux ;

— proposer un jeu interactif où une personne peut proposer des coups successivement, et l'ordinateur vérifie les coups et les applique s'ils sont valides.

