XpatSolver : résolution automatique de jeux de solitaire

Projet PF5 -- L3 Informatique 2022-2023 -- Université Paris Cité

Version 1.0 (15/11/2022)

Il n'est guère nécessaire de présenter la famille des jeux de cartes dits "solitaires" (ou "patiences, ou "réussites"). Certains de ces jeux sont célèbres, par exemple `Solitaire` (aussi nommé Klondike) et `FreeCell` qui ont longtemps été installés en standard sur tout OS Windows, à côté du non moins célèbre démineur. Certains de ces jeux de solitaires sont juste des exercices de maniement de la souris et nécessitent peu de réflexion. Mais d'autres cas s'apparentent à de véritables casse-tête logiques : trouver manuellement une solution est parfois très ardu, même quand toutes les cartes sont visibles. La différence entre parties difficiles mais solubles et parties sans solution peut alors être très ténue.

Lors de ce projet, vous programmerez une recherche automatique de solutions pour toute une classe de jeux de solitaire. Cette recherche pourra s'arrêter dès la première solution trouvée. Par contre, en l'absence de solution, on ira parfois jusqu'à une exploration exhaustive des coups possibles, ce qui permettra dans ce cas de conclure que certaines parties sont insolubles.

Mise en garde : comme tout jeu, et en particulier tout jeu informatique, les solitaires peuvent s'avérer addictifs, même en l'absence d'enjeu d'argent. L'objectif de ce projet est d'y faire jouer vos ordinateurs, pas d'y jouer vous-même de façon excessive ! En cas de difficulté, des structures comme https://www.joueurs-info-service.fr/ peuvent vous venir en aide.

Les solitaires considérés dans ce projet

On se limite ici à des jeux de solitaires ayant les caractéristiques suivantes:

- Un seul paquet usuel de 52 cartes est utilisé. Dans chaque couleur, l'As est la plus petite valeur, suivi du 2, et ainsi de suite jusqu'au Roi.
- Toutes les cartes sont distribuées et visibles dès le départ.
- Ces cartes sont disposées dans un certain nombre de *colonnes* qui se manipulent comme des piles : seule la carte en sommet de colonne est déplaçable (si on lui trouve une destination valide).
- On dispose parfois en plus de quelques registres temporaires pouvant contenir une unique carte chacun. Un registre vide est toujours une destination valide. Une carte d'un registre peut être déplacée si on lui trouve une destination valide.
- Enfin un *dépôt* sert à ranger les cartes en position finale par couleur et par ordre croissant. Il s'agit de quatre piles initialement vides, une pour chaque couleur de carte. Dans la pile des piques, par exemple, il faudra d'abord mettre l'As de pique, puis le 2, et ainsi de suite jusqu'au Roi de pique.

Attention, tous les programmes habituels de solitaire affichent les colonnes avec leurs sommets de pile vers le bas !

Le programme Xpat2

Nous utiliserons comme référence un programme linux open-source nommé xpat2, qui permet de jouer une quinzaine de variantes différentes de solitaire, dont seulement quatre nous intéresserons ici vus nos critères précédents : FreeCell, Seahaven, Midnight Oil et Baker's Dozen. Voir la section Références plus bas pour plus de détails.

Même si ce projet peut se faire entièrement sans disposer de xpat2, son usage pourra être pratique pour se familiariser avec les différentes règles de ces solitaires. Ce programme xpat2 est installable sur Ubuntu et Debian via le paquet du même nom : `sudo apt install xpat2`. Ailleurs, l'usage d'une machine virtuelle Debian ou Ubuntu pourra aider. Il est également possible d'utiliser une machine linux de l'UFR, ou une connexion `ssh -Y` au serveur `lulu` (à condition d'avoir au moins un "serveur X" chez soi, détails dans le fichier `INSTALL.md` sur le site du cours).

Par défaut, xpat2 démarre une partie d'un solitaire que nous n'étudierons pas (Gypsy, qui utilise 104 cartes!). Choisir alors dans le menu "Règles" (ou "Rules") une des quatre variantes considérées. On peut également faire ce choix via une option de la ligne de commande, par exemple `xpat2 -rule "Free Cell"` ou en abrégé `xpat2 -rule fc`. Voir `man xpat2` ou `xpat2 -rule help` pour plus de détails.

Pour jouer, un clic gauche sur une carte essaie de la déplacer à un endroit que xpat2 juge raisonnable. Un clic "milieu" sur une carte permet de choisir sa destination (selon les souris, ce clic milieu peut correspondre à un appui simultané sur les boutons gauche et droite, ou encore à un appui *sur* la molette de la souris). Un clic droit sur une carte permet de la voir en entier (utile pour connaître la couleur d'une "figure" en fond de colonne).

Le programme xpat2 permet de déplacer en un seul clic des blocs de plusieurs cartes consécutives (cliquer sur la carte en fond de bloc). Dans ce cas, xpat2 simule une succession de mouvements individuels des cartes de ce bloc. En particulier ce mouvement de bloc n'est possible que si l'on dispose d'assez de registres vides et/ou de colonnes vides pour faire les mouvements individuels qui correspondent. Dans le cadre de ce projet, nous ignorerons ces mouvements de bloc et ne considérerons que les mouvements individuels de cartes.

Si une carte peut être mise au dépôt, xpat2 permet de le faire, ou non. Nous utiliserons ici une heuristique simplificatrice consistant à forcer la mise au dépôt de toute carte pouvant y aller, qu'elle soit en sommet de colonne ou dans un registre. Ces mises au dépôt seront appliquées dès que possible, autant que possible, et silencieusement (ce qui correspond à l'usage systématique du bouton "Mettre au dépôt" de xpat2). En particulier ces mises au dépôt ne seront pas comptabilisées parmi les mouvements, par exemple lors de l'affichage d'une solution. On ignorera également la possibilité offerte parfois par xpat2 de faire d'éventuels mouvements de retour d'un dépôt vers une colonne ou un registre. Pour les variantes Seahaven et Midnight Oil, ces choix sont justifiables, ils n'empêchent pas de trouver une solution quand il en existe une (s'en convaincre avec le détail des règles ci-dessous). Par contre pour FreeCell et Baker's Dozen, ces choix ne sont que des heuristiques, toutes les parties de ces solitaires semblent être résolubles ainsi expérimentalement, mais ce n'est peut-être pas systématique (saurez-vous trouver un contre-exemple ?).

Détail important : xpat2 permet de générer des parties à partir d'un simple numéro (la *graine* ou *seed* en anglais). Cette graine permet de calculer une permutation des cartes, et donc une configuration initiale du jeu. On peut donc lancer une partie donnée de manière déterministe, p.ex. FreeCell avec graine 123456 via `xpat2 -rule fc 123456` (ici `fc` pour FreeCell). Si on n'indique pas de graine à xpat2, il en génère une (à partir de l'horloge!). Et on peut retrouver la graine d'une partie en cours en utilisant le bouton

"Sauvegarder". Le fichier de sauvegarde produit aura alors un nom de la forme `regle.graine`, comme par exemple `FreeCell.123456`.

Nous allons également utiliser un nom dans ce format pour identifier une partie. En particulier, votre programme devra être en mesure de prendre un tel nom en option de ligne de commande et de reproduire la même configuration initiale que xpat2. Pas besoin qu'il existe réellement un fichier portant ce nom, seul le nom importe, et suffit pour indiquer la règle du jeu et la graine.

Avant de préciser plus le comportement attendu de votre programme, regardons déjà les règles précises des solitaires considérés.

FreeCell

FreeCell suit les même règles que le programme du même nom sous Windows.

- Il y a 8 colonnes et 4 registres temporaires.
- Initialement, ces 8 colonnes contiennent respectivement 7, 6, 7, 6, 7, 6, 7 et 6 cartes. Et les registres sont vides.
- Une colonne non vide ne peut recevoir qu'une carte immédiatement inférieure, et de couleur alternée. Par exemple une colonne ayant un 3 de coeur à son sommet ne pourra recevoir qu'un 2 de pique ou un 2 de trèfle.
- Une colonne vide peut recevoir n'importe quelle carte, de même pour un registre.

Remarque: comme indiqué précédemment, expérimentalement *toutes* les parties de FreeCell générées par xpat2 semblent résolubles, même avec l'heuristique de mise au dépôt systématique. Saurez-vous trouver un contre-exemple à cette constatation ? Attention, l'exploration exhaustive des configurations d'une partie peut être d'une complexité nettement trop élevée, voir plus bas la section "Conseils" pour une heuristique ne gardant que les configurations les plus prometteuses.

Seahaven

Seahaven ressemble beaucoup à FreeCell, la principale différence étant que l'on empile des cartes de même couleur.

- Il y a 10 colonnes et 4 registres temporaires.
- Initialement, ces 10 colonnes contiennent chacune 5 cartes, et les deux dernières cartes sont dans des registres.
- Une colonne non vide ne peut recevoir que la carte immédiatement inférieure et de même couleur. Par exemple une colonne ayant un 3 de coeur à son sommet ne pourra recevoir qu'un 2 de coeur.
- Une colonne vide ne peut recevoir qu'un Roi.

Remarque: ici des parties sans solution se rencontrent assez souvent. Et comme l'exploration exhaustive des configurations est faisable pour cette variante, on pourra détecter de façon fiable ces parties sans solution. Cela peut nécessiter d'éviter certaines configurations dont on peut montrer qu'elles seront forcément infructueuses (voir la section "Conseils" ci-dessous pour plus de détails).

Midnight Oil

- Il y a 18 colonnes et aucun registre temporaire.
- Initialement, toutes les colonnes contiennent 3 cartes sauf la dernière qui n'en contient qu'une seule.

- Comme pour Seahaven, une colonne ne peut recevoir que la carte immédiatement inférieure et de même couleur.
- Par contre ici une colonne vide n'est *pas* remplissable.

Remarque: les parties de Midnight Oil sont très souvent insolubles. Par contre ces règles sont plus simples et conduisent à nettement moins de configurations que pour les autres variantes, ce qui peut aider lors de la mise au point de la recherche automatique de solutions.

En fait, xpat2 autorise deux choses de plus que nous ne prendrons pas en compte dans ce projet, et qui améliorent un peu le taux de réussite de ce jeux:

- Le bouton "Cartes" peut être utilisé deux fois par partie pour re-mélanger les cartes pas encore au dépôt.
- Une fois par partie, une carte arbitraire peut être remontée au sommet de sa colonne (touche "b" puis cliquer sur la carte).

Baker's Dozen

- Il y a 13 colonnes contenant chacune 4 cartes initialement.
- Aucun registre.
- Une colonne peut recevoir une carte immédiatement inférieure, peu importe sa couleur.
- Une colonne vide n'est *pas* remplissable.

Lors de la distribution initiale, les rois sont descendus au fond de leurs colonnes (attention il peut y avoir plusieurs rois dans une même colonne).

Mêmes remarques que pour FreeCell : expérimentalement *toutes* les parties de Baker's Dozen générées par xpat2 semblent résolubles, même avec l'heuristique de mise au dépôt systématique. Saurez-vous trouver un contre-exemple à cette constatation ? Attention, l'exploration exhaustive des configurations d'une partie peut être d'une complexité nettement trop élevée, voir plus bas la section "Conseils" pour une heuristique ne gardant que les configurations les plus prometteuses.

Partie I : simuler une partie, valider une solution existante

I.1 État initial d'une partie, représentation des états

La première tâche de votre programme consiste à lire sur sa ligne de commande le nom d'une partie, par exemple `FreeCell.123456` et en déduire la configuration initiale de la partie en question, puis afficher cette configuration initiale (au format de votre choix). Ces noms de partie seront des chaînes de la forme `FreeCell.n` ou `Seahaven.n` ou `MidnightOil.n` ou `BakersDozen.n`, avec `n` un entier entre 1 et un milliard. Une fois décodé cet entier `n` que nous appellerons la "graine", il faut en déduire une permutation des 52 cartes. C'est le rôle de la la fonction `XpatRandom.shuffle` (voir le fichier `XpatRandom.ml` et son interface `XpatRandom.mli`). La fonction `XpatRandom.shuffle` fournie est incomplète, elle ne sait gérer que quelques cas particulier, par exemple `123456`. Elle est donc utilisable pour commencer quelques tests, mais devra être complétée plus tard (voir la tâche I.3 ci-dessous).

Une fois la permutation récupérée, les cartes de cette permutation sont à placer successivement dans les colonnes, de la première à la dernière, du fond de la colonne à son sommet. Le nombre de cartes par colonne est décrit dans les règles du jeu indiquées précédemment. Attention à Barker's Dozen et son

placement particulier des rois expliqué précédemment. Et dans le cas de Seahaven, il restera deux cartes après remplissage des colonnes, ces deux cartes sont à placer dans les deux registres temporaires.

Au final, une configuration du jeu (ou "état" en plus court) sera formée de quatre parties:

- le dépôt,
- les colonnes,
- les registres temporaires,
- un historique des coups qui ont amené à cet état (utile pour afficher une solution à la fin, cette zone pourra être ignorée dans la première partie du projet).

Dans ces états, les cartes devront être représenté via le type `Card.card` (voir `Card.ml`) fourni. Deux fonctions fournies `Card.of_num` et `Card.to_num` permettent de convertir les nombres 0..51 en cartes et vice-versa. Pour le reste, la représentation d'un état est libre, mais vos choix devront être pertinents. Pour la zone de dépôt, pas besoin d'y stocker de véritables cartes, il suffit de connaître le nombre de cartes déjà déposé pour chacune des quatre couleurs, zéro initialement et jusqu'à 13 dans chaque couleur quand la partie est gagnée. Pour les colonnes, il y aura beaucoup de variabilité dans la taille de chaque colonne, utilisée de plus à la façon d'une pile, donc un codage via une liste OCaml est recommandé. Pour regrouper ensuite ces colonnes et y accéder par leur numéro, l'usage d'un tableau peut être tentant, et n'est pas interdit, bien lire cependant le paragraphe suivant à ce propos. Enfin du côté des registres temporaires, peu importe si une carte est dans la première case ou la seconde, tant qu'elle est quelque part. Pour réduire le nombre de configurations à considérer en partie II, on pourra donc stocker les cartes dans ces registres en les triant.

Concernant le possible usage de tableaux impératifs dans la représentation des états : attention cependant à ce que votre projet reste aussi fonctionnel que possible, selon le critère 6 du fichier `CONSIGNES.md` ci-joint. Attention aussi à se servir convenablement d'éventuels `array`, et à ne pas perturber via une affectation en place un autre état encore utile, ou même une structure d'ensemble d'états (voir la description de `Set.Make` en partie II). En conséquence, si le type `array` est utilisé ici, alors `Array.copy` sera fréquemment nécessaire, et vos tableaux devront rester immutables de facto dès qu'ils servent dans des ensembles. D'autres approches plus sûres existent également, par exemple des tableaux purement fonctionnels (voir le fichier `FArray.mli` fourni, accès logarithmiques, aucune recopie nécessaire) ou des tableaux persistants (voir `PArray.mli`, utilisant des tableaux impératifs en interne et forçant des recopies avant toute écriture). Les coûts de ces approches alternatives sont à comparer avec ceux des `Array.copy` indispensables autrement, nous vous incitons à mener vos propres essais.

I.2 Valider un fichier solution

Lors d'une partie, un *coup* est ici un déplacement de carte du sommet d'une colonne vers une autre colonne ou vers un registre, ou réciproquement depuis un registre vers une colonne. On rappelle que les mises au dépôt des cartes devront être systématiques et ne seront pas comptabilisées comme des coups. Au départ puis entre chaque coup, il faudra donc "saturer" toutes les mises au dépôt successives possibles. Nous appellerons *normalisation* ce processus de mise au dépôt, et un état sera dit *normalisé* s'il vient de le subir.

Un fichier solution est un fichier textuel contenant autant de lignes que de coups à jouer pour atteindre la solution. Chaque ligne sera formé de deux mots séparés par un unique espace. Le premier mot sera un entier dans 0..51, et indiquera le numéro de la carte à déplacer (cf. `Card.of_num`), cette carte pourra être au sommet d'une colonne ou bien dans un registre. Le second mot pourra être :

- Soit un autre entier dans 0..51, ce numéro indiquera une carte en sommet de colonne, le mouvement se fera donc vers cette colonne destination.
- Soit la lettre majuscule V, ce qui indiquera un mouvement vers la première colonne vide disponible (il devra alors y en avoir au moins une).
- Soit la lettre majuscule T, ce qui indiquera un mouvement vers un registre temporaire inoccupé (il devra alors y en avoir au moins un).

Votre programme devra accepter l'option `-check` suivi d'un nom de fichier (et cela en plus de l'argument du style `FreeCell.123456` donnant le nom de la partie, cf auparavant). Dans ce cas, votre programme devra alors lire les lignes de ce fichier solution, et reproduire tous les coups de cette solution depuis la configuration initiale jusqu'à la fin de partie. Vérifiez que chaque coup est bien légal en fonction de la configuration de la partie à ce moment-là et des règles du jeu en question, puis que la dernière configuration atteinte est bien gagnante. Dans ce cas, votre programme doit afficher en dernier le mot `SUCCES` seul sur une ligne puis exécuter un `exit 0`. Sinon, votre programme doit s'arrêter au premier souci, afficher une dernière ligne `ECHEC n` puis faire un `exit 1`. Le `n` dans l'affichage doit être le numéro du premier coup illégal (compté à partir de 1), ou bien le nombre de coups plus un si tous les coups fonctionnent mais ne mènent pas à l'état gagnant. Avant ces derniers affichages codifiés (destinés à des tests automatiques), vous pouvez afficher ce que vous voulez. Par exemple il peut être intéressant (mais pas obligatoire) d'afficher le dernier état atteint lors de l'exécution, sous la forme de votre choix, ou au moins le nombre de cartes mises au dépôt.

I.3 Création de la permutation

Comme indiqué précédemment, la fonction `XpatRandom.shuffle` fournie est incomplète, elle n'est qu'une succession de cas particuliers pour permettre quelques essais initiaux. Vous devez la remplacer par une fonction complète compatible avec l'algorithme utilisé par xpat2. Voir la description de cet algorithme en commentaire dans le fichier `XpatRandom.ml`. Le code actuel pourra vous servir de test pour votre nouvelle implémentation de `shuffle`. Votre code de `shuffle` nécessitera l'usage d'une structure `Fifo` (first-in first-out). Une première version de `Fifo.ml` et son interface `Fifo.mli` est fournie, mais l'implémentation proposée est inefficace (ajout linéaire dans la fifo). En conservant exactement l'interface `Fifo.mli`, vous pourrez changer l'implémentation `Fifo.ml` afin de disposer d'une implémentation fonctionnelle avec ajout et retrait en temps constant en moyenne. Pour cela une possibilité est d'utiliser une paire de listes (au lieu d'une seule liste).

Tests

Vérifiez que vous obtenez des résultats corrects avec la commande `dune runtest tests/I` qui doit ne rien afficher, et sans erreur. Les tests ne garantissent pas que tout est parfait, mais peuvent aider à détecter des erreurs communes.

Partie II : recherche automatique de solutions

II.1 Comportement attendu

Votre programme devra accepter l'option `-search` suivi d'un nom de fichier, (toujours en plus d'un argument du style `FreeCell.123456`). Dans ce cas, votre programme recherchera une solution pour cette partie.

- Si une solution est trouvée, elle devra être écrite dans le fichier au nom indiqué après `-search`, dans le format des fichiers solution vu en Partie I. De plus, votre programme devra afficher ensuite `SUCCES` seul sur une dernière ligne de sa sortie standard, puis exécuter un `exit 0`.

- Si votre programme ne trouve aucune solution après une recherche exhaustive, votre programme affichera `INSOLUBLE` seul sur une dernière ligne, puis exécutera un `exit 2`. Si par contre la recherche n'a pas été exhaustive, le dernier message sera `ECHEC`, suivi d'un `exit 1`. Cette situation d'une recherche non-exhaustive pourra venir d'heuristiques limitant arbitrairement l'espace de recherche afin de terminer en temps raisonnable (voir la section "Conseils" ci-dessous).

II.2 Méthode de recherche

Lors d'une partie, les états atteignables peuvent être vus comme les noeuds d'un graphe, et les coups légaux lors de la partie sont alors les arêtes de ce graphe orienté. On cherche donc à relier l'état initial de la partie avec l'état final gagnant (celui où toutes les cartes sont au dépôt), et trouver un chemin reliant ces deux états (s'il en existe un). Il s'agit a priori d'un problème classique de l'algorithmique des graphes. Mais même si ce graphe des états d'une partie est forcément fini, il sera presque toujours gigantesque, ce qui oblige à le visiter prudemment, "au vol", en optimisant les données manipulées et les méthodes utilisées. En particulier les états atteignables ne seront créés que progressivement.

L'idée générale est celle d'un parcours de graphe. A tout moment, on a un ensemble d'états restant à visiter (au début juste l'état initial), et un ensemble d'états déjà traités (au début vide). La visite d'un état consiste à calculer les états atteignables par des coups légaux à partir de cet état, puis mettre ces nouveaux états parmi ceux à visiter (sauf ceux qui ne sont pas si nouveaux que cela mais au contraire sont déjà parmi les traités), puis mettre l'état qui vient d'être visité parmi les traités.

On rappelle que les états qu'on manipule devront toujours être normalisés (i.e. avoir le maximum possible de cartes mis au dépôt). En particulier, l'état initial de la partie devra être normalisé avant le début de la recherche. Ensuite, chaque action d'un coup légal sur un état en cours de visite devra être suivi d'une possible renormalisation. Cela permet de limiter le nombre d'états à considérer et de coups à explorer. Pour les règles Seahaven et Midnight Oil, c'est compatible avec une recherche exhaustive, mais pas pour FreeCell et Baker's Dozen (mais en pratique cela ne semble pas empêcher la découverte de solution).

On appellera *score* d'un état le nombre de cartes qu'il a dans son dépôt. Il y a un seul état de score 52, c'est la configuration gagnante. Si la recherche rencontre cet état, une solution existe, et on arrête la recherche. Comme indiqué en partie I, il est recommandé d'avoir dans la représentation d'un état une zone donnant un historique des coups ayant mené à cet état. Dans ce cas, l'historique des coups de l'état gagnant donne alors une solution à la partie explorée. Attention, cette approche nécessite par contre d'utiliser une comparaison particulière entre états, et non `Stdlib.compare`, voir détails plus bas. Maintenant, si on ne rencontre jamais l'état gagnant, et que l'ensemble des états restant à visiter devient vide, c'est que la recherche est terminée sans solution. Et cette recherche a été a priori exhaustive, sauf si l'on a utilisé des heuristiques supprimant brutalement des états peu prometteurs (voir la section "Conseils" ci-dessous).

Dans quel ordre visiter les états ? Pour une visite exhaustive et infructueuse, peu importe, au final on sera passé partout. Par contre si une solution existe, autant essayer d'y aboutir au plus vite, et éviter la visite d'états qui ne donneront rien, ou alors bien plus tard. A vous d'implémenter une approche aussi efficace que possible sur de véritables parties (voir par exemple les tests fournis). Une méthode simple à mettre en oeuvre (mais pas forcément rapide) consiste à procéder par profondeur progressive, ce qui revient à faire un parcours en largeur d'abord. La profondeur est ici le nombre de coup minimal pour atteindre un état. La

profondeur 0, c'est l'état initial, et la profondeur (n+1), ce sont les états atteignables depuis l'un des états de profondeur n, et pas déjà rencontrés auparavant. On peut calculer progressivement ces ensembles d'états à des profondeurs croissantes, ainsi que l'ensemble des anciens états déjà rencontrés à profondeur plus faible. En procédant ainsi, on est sûr en outre d'obtenir une solution qui aura un nombre de coup minimal (même si ce n'est pas imposé). Une autre méthode possible est de s'intéresser aux scores des états, et de visiter d'abord les états ayant les scores les plus élevés. Les scores étant entre 0 et 52, on pourra ranger les états encore à visiter selon leurs scores. Attention, un tel parcours peut souvent passer par des "impasses" (états aux scores élevés mais sans coups légaux restants, ou seulement vers des états déjà vus). Il faut alors retourner à des états restants à visiter mais ayant des scores un peu moins bon. Pour indiquer à l'utilisateur où en est la recherche, quelques affichages intermédiaires pourront être utiles, ni trop ni trop peu.

II.3 Codage d'ensembles d'états

Un point crucial de cette recherche est donc de pouvoir déterminer si un état a déjà été rencontré, pour éviter les cycles dans la recherche. Pour cela, on utilise ici des ensembles d'états, et on tester l'appartenance à un tel ensemble. L'usage du module `Set` d'OCaml est recommandé pour cela. Dès que vous disposez d'un type OCaml de vos états (disons `state`) et d'une fonction de comparaison pour ces états (`compare_state : state -> state -> int`), alors la ligne "magique" suivante dans votre code vous fournit un type des ensembles d'états, et de nombreuses fonctions associées:

```
(* définir le type state et la fonction compare_state auparavant *)
module States = Set.Make (struct type t = state let compare = compare_state
end)
```

Le type des ensembles d'états est utilisable après cette ligne, il s'appelle `States.t`, et on dispose alors d'un ensemble vide `States.empty` et d'opérations telles que le test d'appartenance `States.mem`, mais aussi `States.is_empty` `States.add` `States.remove` `States.union` `States.diff` `States.filter` `States.cardinal` `States.fold` `States.choose` etc. Pour plus de détails, voir la documentation https://v2.ocaml.org/api/Set.html et https://v2.ocaml.org/api/Set.S.html . Cette structure est fonctionnelle, et les opérations concernant un élément (test d'appartenance, ajout, suppression) sont logarithmiques en la taille de l'ensemble. En interne, ces ensembles sont des arbres équilibrés, et votre comparaison `compare_state` sert à situer un élément dans l'arbre par dichotomie (comparaison avec la racine, puis éventuellement à un des fils, etc).

Deux points importants:

a) Pour un fonctionnement correct, les résultats de ces appels à `compare_state` doivent être stables : si on compare deux états deux fois, on doit avoir le même résultat. C'est pourquoi dès l'ajout d'un état dans un ensemble `States.t` il ne faudra plus modifier impérativement cet état. Si votre type `state` contient des structures impératives comme des tableaux, l'usage explicite de `Array.copy` pourra donc être nécessaire pour créer un nouvel état dérivé d'un ancien.

b) Maintenant, quelle comparaison `compare_state` doit-on utiliser ? OCaml propose une comparaison générique nommée `Stdlib.compare` qui est souvent convenable. Mais ici on vous propose de mettre un historique des coups dans chaque état. Or deux historiques différents peuvent pourtant mener à des états rigoureusement identiques à part ça (mêmes colonnes, mêmes registres, et donc mêmes dépôts). Pour le

bon fonctionnement de l'algorithme de recherche, il est alors crucial que dans ce cas `compare_state` réponde une égalité (code `0`). Bref, pour `compare_state`, au lieu de comparer deux états entiers directement via `Stdlib.compare`, on pourra comparer leurs zones de registres respectives (p.ex. par `Stdlib.compare`), et en cas d'égalité seulement comparer leurs zones de colonnes respectives (via un autre `Stdlib.compare`). Et rien de plus.

Enfin, notons que OCaml propose également en standard une structure (impérative) de tables de hachage (module `Hashtbl`) qui peut être utilisé pour coder des ensembles. Les performances annoncées sont attrayantes (tests d'appartenance, ajout, suppression peuvent théoriquement être en temps constant). Mais attention à ne pas minimiser les temps de calcul des fonctions de hachage (ainsi que de comparaisons qui sont aussi nécessaires ici). De plus les points délicats a) et b) précédents s'appliquent également aux `Hashtbl`. Vous pouvez expérimenter avec des `Hashtbl` à condition de pouvoir nous présenter *aussi* une version à base de `Set.Make`, et une étude des performances des deux versions.

Conseils

Votre programme doit pouvoir gérer plusieurs règles du jeu différentes (FreeCell, ...). Attention du coup à bien factoriser autant que possible votre code pour éviter les redondances. Il est recommandé en particulier de proposer une description abstraite de chaque règle du jeu via ses caractéristiques (quelles contraintes de couleur sur un enchaînement de cartes ? Quelles cartes peuvent aller sur une colonne vide ? etc). Dans l'idéal, ajouter à votre programme une règle du jeu supplémentaire devrait pouvoir se faire en quelques lignes seulement si elle reprend des caractéristiques d'autres règles (p.ex. une variante de Seahaven où une colonne vide peut recevoir toute carte, pas seulement un roi). Et cela sans avoir à reparcourir tout votre code.

Dans la description des états et des coups en Partie I, on a déjà suggéré quelques manières de réduire un peu le nombre d'états et donc l'espace de recherche :

- ne pas distinguer l'ordre des cartes dans les registres (ce qui revient p.ex. à trier ces registres)
- si on a plusieurs colonnes vides, pas besoin de considérer le déplacement d'une carte vers toutes ces colonnes vides, le faire vers la première suffira. De même, lors de la recherche, certains coups pourtant légaux pourront être ignorés, car ils n'apporteront rien de plus:
- Par exemple si une colonne est constituée d'une seule carte, ce n'est pas utile de considérer le déplacement de cette carte vers une colonne vide, on se retrouverait alors dans une situation équivalente.
- La remarque suivante concerne spécifiquement les parties qui empilent des cartes de même couleur (Seahaven et Midnight Oil). Comme en plus ces parties ont un usage contraint des colonnes vides, alors une longue séquence de cartes décroissantes de même couleur ne pourra plus être déplacée, et ne pourra évoluer que via une mise au dépôt. Par "longue", on entend ici (n+2) cartes au moins si l'on a n registres. Dans cette même colonne, si enfin une "petite" carte de la même couleur se trouve bloquée quelque part sous la "longue" séquence, alors la mise au dépôt de ces cartes sera toujours impossible. On pourra donc chercher en sommet de colonnes de telles "longues séquences bloquées", et supprimer de la recherche les états qui en contiennent, car ils sont insolubles.

Ce projet n'est pas un exercice de calcul intensif, les exemples considérés doivent pouvoir se traiter en quelques minutes (disons 2 ou 3 minutes maximum). Si ce n'est pas le cas, on pourra essayer des heuristiques simples, comme de supprimer de la recherche les états dont le score est loin du meilleur score trouvé jusqu'à maintenant: si on a déjà trouvé un état de score 30, il est en effet peu probable que les états

de score moins de 20 soit encore utiles pour cette recherche (on pourra alors paramétrer cet écart via une option du programme). Une telle "distance d'oubli" fait évidemment perdre l'exhaustivité de la recherche, mais peut aider en pratique pour les parties à "fort branchement" comme FreeCell ou Baker's Dozen.

Extensions possibles

Voici quelques extensions possibles. Cette liste n'est pas exhaustive.

- Recherche aléatoire de parties de FreeCell ou Baker's Dozen qui ne seraient pas solubles dans le cadre de ce projet. S'il y en a, sont-elles solubles via des mises au dépôt non-systématiques et/ou des "retours en arrière" du dépôt vers colonnes et registres ?
- Pour Midnight Oil, prendre en compte les règles complètes : a. avec une "remontée" possible d'une carte b. avec deux redistributions possibles des cartes en cours de partie.
- Retrouver via les sources de xpat2 le format des fichiers de sauvegarde de xpat2 et proposer une conversion entre nos fichiers solution et ces fichiers sauvegarde.

Références

Xpat2

- Page Debian du paquet : https://packages.debian.org/bullseye/xpat2
- Code source (tgz) : http://deb.debian.org/debian/pool/main/x/xpat2/xpat2_1.07.orig.tar.gz
- Code source (navigable) : https://sources.debian.org/src/xpat2/1.07-20/