Rapport XPatSolver

- Rapport XPatSolver
  - Identifiants
  - Fonctionnalités
    - Simuler une partie
    - Vérification d'une solution existante
    - Recherche de solutions
    - Extensions
  - Compilation et exécution
    - Compilation
    - Execution
  - Découpage modulaire
    - Game
    - XPatSolver
      - treat_game
    - XPatRandom
    - Search
  - Organisation du travail
    - Répartition des tâches
    - Chronologie
  - Misc

Identifiants

Nom	Prénom	Identifiant	Numéro d'étudiant
Abignoli	Léopold	@abignoli	22004535
Gilardoni	Daniel	@gilardon	22008366

Fonctionnalités

Simuler une partie

Ce projet prend en compte plusieurs variantes de **Solitaire** telles que **Freecell**, **Seahaven**, **Baker's Dozen**, **Midnight oil**. Une partie de **Solitaire** est représentée par le type **GameStruct** qui contient le nom du jeu, les registres, colonnes, dépôts et l'historique des coups. Toutes les fonctions pour simuler une partie de **Solitaire** sont définies dans le module Game.ml

Vérification d'une solution existante

Avec ce projet, vous pouvez vérifier qu'un fichier solution résout bien une partie de **Solitaire**. Pour cela, il faudra utiliser l'option `-check` suivi du nom de votre fichier. Vous allez également avoir besoin du type de **Solitaire** et d'une **graine**. Je vous invite à voir la section Execution pour avoir plus d'informations. Vous

pouvez également consulter la section XpatSolver pour comprendre comment nous avons implémenté cette partie du code.

On notera que notre projet valide tous les tests de la partie I.

Recherche de solutions

Enfin, nous avons implémenté aussi la recherche d'une solution. Pour cela, il suffit à l'utilisateur de préciser l'option `-search` suivi du nom d'un fichier (pour écrire la solution). Bien sûr, il faut également préciser le type de **Solitaire** que vous voulez résoudre accompagné d'une **graine**. Pour plus d'informations, vous pouvez consulter la section Execution.

A travers ce projet, nous avons implémenté une recherche **exhaustive** et **non_exhaustive**. Cependant, la recherche **non_exhaustive** semble ne pas fonctionner à tous les coups. C'est pourquoi nous utilisons la méthode **exhaustive** par défaut pour résoudre une partie. On notera que la plupart des tests de la partie II semblent fonctionner. Seuls deux tests ne passent pas pour cause de **timeout** (*sans compter le fichier hard.t*)

Pour plus d'informations sur notre implémentation, nous vous invitons à consulter la section sur notre module Search

Extensions

Nous avons fait le maximum pour realiser le sujet en entier. Cependant, nous n'avons pas eu le temps de rajouter des extensions.

Compilation et exécution

Compilation

Pour compiler le projet, vous pouvez executer la commande suivante:

```
dune build
```

Vous pouvez également utiliser le `Makefile`:

```
make
```

Execution

Pour lancer le projet, vous pouvez executer le fichier `run`. Il faudra alors lui passer plusieurs paramètres. Voici un exemple de la page d'aide:

```
XpatSolver <game>.<number> : search solution for Xpat2 game <number>
  -check <filename>:    Validate a solution file
  -search <filename>:   Search a solution and write it to a solution file
  -help  Display this list of options
  --help  Display this list of options
```

Voici un exemple pour chercher une solution et l'écrire dans le fichier `test.sol` si elle existe:

```
./run FreeCell.123456 -search test.sol
```

Enfin, un deuxième exemple pour vérifier si un fichier solution est correcte:

```
./run FreeCell.123 -check tests/I/fc123.sol
```

Découpage modulaire

Pour notre projet, nous avons eu besoin de créer plusieurs modules. Notamment un module **Game.ml** et **Search.ml**.

Game

Le module `Game` contient toutes les fonctions pour modéliser et jouer une partie de **Solitaire** parmis les jeux demandés. Il contient aussi le type **gameStruct** qui représente un état d'une partie. Le type **gameStruct** est constitué :

- d'un FArray de Card option pour les registres. A la creation de l'état le nombre de registres est fixée et le tableau contient uniquement des None si il n'y pas de cartes dans les registres.
- de deux FArray de list de Card pour les colonnes et les dépots. Les listes de cartes représentent chacune une colonne.
- D'une list de tuple de la forme (int * string) pour représenter l'historique des deplacements, avec une liste de couple de mots, le premier représentant la carte deplacée et le second la location où la carte à été déplacée.

Voici une description des différentes fonctions de notre module:

kings_on_back: Trie une colonne pour récupérer dans une liste les rois avec **List.filter**, et dans une autre, les autres cartes. Concatène ensuite ces 2 listes en mettant les rois au fond. La fonction fait cela pour

chaque colonne grâce à **List.map**.
Elle renvoit un nouveau tableau de colonnes.

add: Ajoute un nombre choisi de card d'une liste de card à une autre liste. Elle renvoie un tuple avec la liste obtenue et les card non ajoutées.

add_column: Ajoute dans les colonnes du tableau **columns** passé en paramètre le nombre de card voulu pour chaque colonne. Le nombre de card souhaité est préciser dans la liste **cardsPerCol**. On utilise alors la fonction **add** pour chaque colonne. On "met à jour" (crée un nouveau) le tableau de colonnes avec `FArray.set` pour chaque colonne après avoir ajouté les cartes.
Elle renvoie un tuple avec le nouveau tableau de colonnes et la liste de cards restantes.

initGame: Initialise une partie (renvoie un **gameStruct**) en fonction du type de jeu souhaité en appelant **initGameAux**. Pour Baker on utilise aussi **kings_on_back**.

initGameAux: Initialise et retourne un **gameStruct** en fonction des paramètres gameType, nbReg, cards et cardsPerCol donné.

disp: Affiche le contenu d'un état en utilisant **disp_history**, **disp_regs**, **disp_list** et **disp_list_list** et **FArray.to_list**.

Les fonctions **get_col** et **get_reg** permettent de récupérer l'index de la colonne ou du registre qui contient la carte dont le numéro est donné en paramètre. Les fonctions **empty_col** et **empty_reg** utilisent les 2 fonctions précédentes pour renvoyer l'index d'une colonne ou d'un registre vide.

remove: Prends le numéro d'une carte et un état **gameStruct** et supprime la carte de l'état. On essaye alors de supprimer la carte des registres avec **remove_in_reg**. Si cela lève une exception alors à l'aide d'un **try with** on utilise **remove_in_col**. Si la carte n'a pas été supprimé, la fonction lève une exception.

Les fonctions **add_to_reg** et **add_to_col** sont utilisées pour essayer d'ajouter une carte dans un registre, une colonne vide ou une carte se situant en haut d'une colonne. Sinon, ces fonctions lèvent une exception. La fonction **add_to_depots** permet d'ajouter une carte dans un dépot.

move: Prends un état, un numéro de carte et une chaine de caractères. Effectue si possible le déplacement de la carte correspondant au numéro vers un registre, une colonne vide ou une autre carte en fonction de la chaine de caractères. Elle utilise **add_to_reg** ou **add_to_col** et renvoie un nouvel état ou lève une exception. Il faut l'utiliser après avoir supprimé la carte que l'on déplace sinon la carte pourrait être en double dans la partie.

rules: Vérifie si le déplacement d'une carte vers une colonne, un registre vide ou une autre carte est autorisé en fonction des régles propre à chaque type de jeu. Par exemple, dans MidnightOil, on ne peut pas ajouter une carte dans une colonne vide. Si la carte est deplacée sur une autre carte on vérifie que son rang est bien directement inférieur et que sa couleur est en accord avec les régles du mode de jeu.

wanted_depot_cards: Permet d'obtenir la liste des cartes qui peuvent être ajoutées dans un dépot. Par exemple si le depot des cartes **Coeur** est vide alors l'As de **Coeur** peut être ajouté au depot. Renvoie une liste de Card option dans le même ordre que les depots, avec **None** si le depot est complet.

normalisation: Utilise **wanted_depot_cards** pour obtenir la liste des cartes pouvant être ajoutée aux dépôts. Elle contient une fonction récursive auxiliaire qui va pour chaque carte de la liste, si elle est **None** continuer, sinon essayer de la supprimer avec **remove** et de l'ajouter aux dépôts avec **add_to_depots**.

Elle renvoie un tuple contenant l'état (nouveau si au moins une carte à été ajoutée aux dépôts) et un MidnightOil valant true si il n'y a eu aucune mise au dépôt et false sinon. Ce bool va être utile dans la fonction **normalisation_full**.

normalisation_full: Elle va appeler **normalisation** puis continuer récursivement jusqu'à ce qu'aucune carte ne soit mise au dépôt, donc lorsque le bool retourné par **normalisation** soit true. Cela permet d'éviter d'oublier les cas où une mise au dépot permet de rendre accessible une carte qui peut à son tour être mise au dépôt.
Cette fonction renvoie l'état de la partie normalisée.

is_won: Elle vérifie en détail si la partie est gagnée ou non. Elle utilise **is_empty** et **is_empty_reg** pour vérifier si les colonnes et registres sont bien vides. Puis **are_depot_complete** va vérifier que toutes les cartes sont dans les dépots et dans le bon ordre en utilisant **is_depot_complete** sur chaque dépôt. Elle renvoit un bool.
On aurait pu se contenter de vérifier qu'il y avait 52 cartes dans les dépôts, mais ainsi on s'assure qu'il n'y a pas eu d'erreurs.

score: Elle renvoie le nombre de cartes dans les dépôts, ce qui est donc le score de la partie.

XPatSolver

Le module `XPatSolver` est le module avec la fonction **main**. C'est celui qui parse les arguments puis qui lance une **recherche** de solutions ou une **vérification** d'un fichier solution de **Solitaire**. Dans ce module, nous avons uniquement modifié la fonction **treat_game**.

treat_game

Cette fonction nous permet d'effectuer une **recherche** de solutions ou de **vérifier** un fichier solution.

Le début de la fonction n'a pas changé. Elle calcule la **permutation** des cartes selon la **graine** puis l'affiche à l'écran. A l'aide de cette permutation, on va alors créer un objet **GameStruct** avec la fonction **Game.initGame** (voir Game)

Voici ensuite les différentes étapes de l'algorithme:

I) On effectue une recherche exhaustive ou non_exhaustive

- On ouvre le fichier dans lequel on va écrire la solution (**file**)
- On appelle la fonction **exhaustive** ou **non_exhaustive** (Voir Search)
- Si il n'y a pas de solution:
  - **Exhaustive**: on affiche **INSOLUBLE** et code erreur 2
  - **Non exhaustive**: il faut remplacer **INSOLUBLE** par **ECHEC** et exit 1
- Si il y a une solution, on écrit les mouvements dans le fichier **file** (*write_moves*) et on affiche **SUCCES**

II) On vérifie si un fichier solution est correcte

- On ouvre le fichier (**file**)
- On lit une ligne du fichier file avec **read_aux** (renvoie **None** si on arrive au bout du fichier)
- On appelle la fonction **treat_game_aux** qui permet d'executer tous les coups du fichier file.
  - On lit le prochain coup (**read_aux**)

    - Si on est au bout du fichier
        - On normalise la partie
        - Si la partie est gagné (**is_won**), on affiche **SUCCES**
        - Sinon, la partie est perdu et le fichier n'est pas une solution. On affiche **ECHEC**.
    - Sinon, on prend la prochaine ligne
        - On récupère la carte et l'endroit où on veut la placer.
        - On normalise la partie
        - On vérifie si le mouvement donné par le fichier est autorisé
        - Si Le mouvement est *interdit*, on affiche **ECHEC**
        - Sinon, si le mouvement est *autorisé*
            - On retire la carte et on la place sur sa nouvelle location
            - On appelle recursivement treat_game pour effectuer le prochain coup

XPatRandom

Le module `XPatRandom` contient la fonction **shuffle** qui va nous permettre de générer une permutation de carte pour une certaine graine (**seed**). Pour écrire cette fonction, nous avons suivi toutes les instructions données au début du fichier. Voici donc, une petite description de comment chaque étape à été implémenté: a) Pour créer les 55 première paires, on utilise une fonction **substraction** puis une fonction **create_pairs** (récursive terminale). b) On applique un `List.sort` pour trier les paires par ordre croissant. Ensuite, on utilise une fonction **split_list** pour couper la liste en deux morceaux. La fonction `FIFO.of_list` est finalement utilisé pour créer les deux **FIFO**. c) On crée une fonction **tirage** qui renvoie un triplet avec la valeur du tirage et les deux **FIFO** modifié. d) On crée une fonction **tirages** pour faire 165 tirages successifs (pour mélanger les deux **FIFO**). La fonction renvoie les deux **FIFO** modifiées. e) Enfin, on a une fonction **get_and_delete** qui prend une **liste** et un **index**. Elle renvoie un tuple avec l'élément placé à l'**index** et la liste vidé de cet élément. Pour finir, on a la fonction **gen_cards** qui va créer une liste avec 52 cartes dans l'ordre. Elle effectue ensuite 52 tirages successifs pour mélanger les cartes et créer la permutation finale.

Search

Le module `Search` contient toutes les fonctions pour effectuer une recherche de solutions. Il nous permet d'effectuer une recherche **exhaustive** et **non-exhaustive** parmis tous les jeux de Solitaire compatibles. Voici une description des différentes fonctions de notre module:

- **compare_games**: Cette fonction nous permet de comparer deux **gameStruct** (deux états d'une partie). Elle utilise notamment la fonction **FArray.compare** qui permet de comparer les tableaux de *registres* et *colonnes*. La fonction **FArray.compare** utilise elle même **Stdlib.compare** (elle convertie d'abord les FArray en listes).

- **States**: Le module **States** est une implementation de l'interface **Set** où les éléments sont de type **Game.gameStruct** (état d'une partie) et la fonction de comparaison est **compare_games**.

- **set_of_list**: Permet de convertir une liste d'états en **States** (Set).

- **set_reachable**: Prend un **States** reachable (ensemble d'états atteignables), un **States** reached (ensemble d'états déjà atteints), et une liste de d'états. Elle renvoie l'ensemble reachable modifié en

ajoutant les états de la liste qui n'ont pas déjà été atteints (pas dans **reached**) ou qui ne sont pas déjà dans les états atteignables (pas dans **reachable**).

- **add**: Prend en entrée un **état** de jeu de Solitaire (**Game.gameStruct**), une destination ("T" pour une **colonne vide**, "V" pour les **registres**, ou une carte correspondant à une colonne) et une liste d'états. Elle calcule tous les états de jeu atteignables en déplaçant une carte vers la destination spécifiée et les ajoute à la liste des états.

- **add_reachable**: Prend en entrée un **état** de jeu de Solitaire, un ensemble d'états atteignables (**reachable**) et un ensemble d'états déjà atteints (**reached**). Elle calcule tous les états atteignables en un coup à partir de l'état donné et les ajoute à l'ensemble de parties atteignables, à moins qu'ils ne soient déjà présents dans l'ensemble **reached**. Pour ce faire, elle utilise la fonction **add**.

- **exhaustive**: Prend en entrée un **état** de jeu de Solitaire et appelle la fonction **search_sol** (avec **best_score=-1**) pour trouver une solution à cette partie. Elle renvoie l'enchainement des coups si une solution a été trouvé.

- **non_exhaustive**: Prend en entrée un **état** de jeu de Solitaire et appelle la fonction **search_sol** (avec **best_score=0**) pour trouver une solution à cette partie. Cette fois, **search_sol** va utiliser la fonction **heuristic** et mettre à jour le **best_score** pour trouver plus rapidement une solution.

Pour finir, la **search_sol** permet de trouver une solution à une partie de Solitaire (si il en existe une). Elle est appelée par deux autres fonctions: **exhaustive** et **non_exhaustive** (voir plus haut). C'est une fonction de recherche récursive qui prend un ensemble d'états **reached** et **reachable**, un **best_score** (le score de l'état avec le meilleur score) et une fonction **heuristic** qui prend en entrée un score, le meilleur score et renvoie un booléen indiquant si le score est suffisamment proche de ce meilleur score. Dans le cas d'une recherche **exhaustive**, le **best_score** vaut -1 et on ne regarde pas la fonction heuristic.

La fonction commence par vérifier si l'ensemble des **états atteignables** est vide, auquel cas elle renvoie None, indiquant qu'aucune solution n'a été trouvée. Sinon, elle récupère le premier **état atteignable** de reachable et le retire de l'ensemble. Elle normalise cet état et calcule son score. Si l'état a déjà été atteint ou si son score n'est pas suffisamment proche du meilleur score connu selon la fonction **heuristic** (uniquement si recherche **non_exhaustive**), elle appelle récursivement la fonction en supprimant l'état de **reachable**.

Sinon, si le score de l'état est égal à 52 (ce qui signifie que toutes les cartes ont été déplacées dans les dépôts), la fonction renvoie l'**historique** de cet état (l'enchainement des coups pour résoudre la partie).

Sinon, elle calcule tous les **états atteignables** en un coup à partir de l'**état actuel**, en utilisant la fonction **add_reachable** et appelle récursivement la fonction en lui passant ces nouveaux états atteignables. Le **best_score** est également mis à jour si besoin.

Organisation du travail

Répartition des tâches

Pour ce projet, nous avons testé une nouvelle méthode de travail. Nous avons utilisé l'extension **liveshare** de **VSCode**. Elle permet de partager son environnement de travail et d'écrire, compiler et executer du code côte à côte. C'est pourquoi, nous avons écrit la plupart du temps le code ensemble. On peut d'ailleurs voir le **Co-authored** sur les commits, qui montre que le code à été écrit en **concurrence**.

Cependant, on peut noter que :

- La partie I avec le module **Game.ml** a été réalisé entièrement à deux
- La fonction **shuffle** dans **XPatRandom.ml** a été écrite par Léopold Abignoli
- Le module **Search.ml** a été réalisé ensemble.
- Une grosse partie des tests et corrections de bugs pour le module Search.ml ont été fait par Daniel Gilardoni ainsi que plusieurs tentatives d'optimisations qui n'ont pas aboutis à cause de différents bugs rencontrés.

Chronologie

Nous avons commencé par créer le module **Game.ml**. Nous avons écrit dans cet ordre :

1. La structure **GameStruct** (pour représenter une partie) et la fonction **initGame** pour l'initialiser.
2. **get_reg**, **get_col**, **empty_reg**, **empty_col**. Ces fonctions permettent de récupérer l'index des cartes dans les **registres** et dans une **colonne**.
3. **remove_in_col**, **remove_in_reg**, **add_to_reg**, **add_to_reg**. Elles permettent d'enlever/ajouter des cartes dans des registres/colonnes.
4. **remove**, **move**, **rules**, **normalisation**. Voici les fonctions principales pour gérer une partie. On utilise alors les fonctions précédentes pour pouvoir enlever/déplacer des cartes entre les colonnes/registres. Vérifier si un mouvement est autorisé, normaliser le jeu...

Ensuite, nous nous sommes occupés de la fonction **shuffle** du module **XpatRandom.ml**.

Pour finir, nous avons codé le module **Search.ml**. Il contient les fonctions pour effectuer une recherche de solution. Elles ont été écrit dans cet ordre :

1. Le module **States** et sa fonction pour comparer deux états, **compare_games** (compare deux **gameStruct**).
2. **set_reachable**, **add**, **add_reachable**. Ces fonctions permettent de récupérer tous les états atteignables en un coup (en partant d'un certain état).
3. **search_sol**, **write_moves**. Ces fonctions permettent de faire une recherche **exhaustive** ou non. On obtient alors la liste des coups pour terminer une partie et on écrit le résultat dans un fichier (**write_moves**).
4. **exhaustive**, **non_exhaustive**, **heuristic**. Permettent d'appeler **search_sol** avec certains paramètres et d'effectuer la recherche voulue.

On a finalement modifié **treat_game** de **XpatSolver.ml** pour pouvoir executer les fonctions **exhaustive** et **non_exhaustive**.

Misc

A travers ce projet, nous avons essayé d'écrire le code le plus lisible possible. De plus, toutes nos fonctions ont été écrite en **récursion terminale**.