Gaps - Projet P3

Owen Gombas

27.01.2023

Table des matieres

Chapitre 1
Introduction

1.1 Description du projet

"Gaps Solitaire" est un jeu de cartes, également connue sous le nom de "Montana Solitaire" et des "Quatre jeudis" à cause de sa difficulté. L'objectif du jeu est d'organiser les cartes séquentiellement et par couleur, de l'as à la dame, en déplaçant les cartes dans quatre espaces initialement choisis au hasard. Une carte peut être déplacée vers un espace vide si et seulement si elle est de la même couleur et directement supérieure à la carte située à gauche du dit espace.

Il existe plusieurs sites pour jouer en ligne, comme par exemple `https://www.jeuxsolitaire.fr/jeu/Gaps+Solitaire`.

"Gaps Solitaire" est un jeu très difficile à terminer, et contrairement à la majorité des jeux de patience requiert beaucoup de stratégie.

1.2 Objectifs

L'objectif du projet P3 est de développer un moteur permettant de placer le plus de cartes possible à partir de toute configuration initiale des cartes. Le moteur doit être capable de jouer seul, sans intervention de l'utilisateur, et de trouver la meilleure solution possible. Voici les sujets abordés dans le cadre du projet :

- Structures de données pour modéliser le jeu et les coups possibles
- Algorithmes élémentaires de parcours de graphes (largeur, profondeur)
- Algorithmes d'intelligence artificielle (A*, ...).

Chapitre 2
Environnement

2.1 Swift

Le choix s'est porté sur le langage d'Apple, Swift, pour la réalisation de ce projet. C'est un langage de programmation moderne, puissant et facile à apprendre. Il est conçu pour être sûr, performant et expressif. Swift est un langage de programmation multi-paradigme, qui supporte le paradigme impératif, le paradigme fonctionnel et le paradigme orienté objet.

Ce langage s'est donc imposé comme le choix idéal pour ce projet, car il permet de développer des applications performantes et facilement portables sur les différents systèmes d'exploitation d'Apple.

2.2 SwiftUI

SwiftUI est un framework de développement d'interface graphique pour les applications iOS, macOS, watchOS et tvOS. Il permet de créer des interfaces graphiques de manière déclarative, et de les mettre à jour automatiquement lorsqu'un changement est détecté. Il est possible de créer des interfaces graphiques en utilisant des composants prédéfinis, ou de créer ses propres composants.

2.3 Playground

Apple fournit un outil de développement nommé Playground, qui permet de développer et d'exécuter du code Swift. Il est possible de créer des pages de code, qui peuvent être exécutées séparément. De la manière qu'un notebook Jupyter permet de créer des cellules de code, un Playground permet de créer des pages de code. Ces pages peuvent être exécutées séparément, et les résultats sont affichés dans la console. Cela permet de tester des idées, de débugger du code, et de créer des exemples.

Cette fonctionnalité a été très utile afin d'effectuer les calculs nécessaires afin de trouver les meilleures heuristiques pour le projet.

Chapitre 3
Code

3.1 UI

Le code principal de l'interface graphique se trouve dans le fichier `ContentView.swift`. Elle permet d'afficher une configuration de jeu et d'intéragir avec. On peut déplacer les cartes en cliquant dessus, et les déplacer vers un espace vide. Lorsqu'on execute un des algorithmes, on peut consulter les logs de l'exécution dans un espace dédié, et voir le déroulement de l'algorithme dans l'interface graphique une fois que celui-ci est terminé. Des graphiques en temps réel permettent de suivre l'évolution des statistiques de l'algorithme executé et en cours d'exécution.

L'interface est composée des composants "custom" suivants:

- `StateUI`
  Affiche le plateau de jeu et permet d'interagir avec celui-ci.
- `ChartUI` et `Measure`
  Affiche un graphique représentant diver statisques.

3.2 Matrix

La structures de données utilisées pour représenter le jeu (`GameState`) est une matrice de `Card` (voir `Card.swift`). Les cartes sont stockées dans un tableau à deux dimensions, et sont indexées par leur position dans le jeu. On peut accéder à un élement en utilisant les indices de la matrice, ou en utilisant les coordonnées de l'élement. Les coordonnées sont des tuples de deux entiers, qui représentent la ligne et la colonne de l'élement dans la matrice.

3.3 GameState

Cette classe représente l'état du jeu. Elle nous permet de stocker les cartes. Elle permet aussi de déplacer les cartes, de vérifier si le jeu est terminé, de générer les coups possibles, et d'effectuer toutes les opérations permettant l'intéraction avec le jeu. Elle hérite de la classe `Matrix`. Les enfants (`state.getMoves()`) du jeu sont les configurations de jeu obtenues en déplaçant une carte. Pour obtenir les enfants, on parcourt la matrice de cartes, et on déplace les cartes en respectant les règles du jeu. On obtient ainsi une liste de configurations de jeu, qui sont les enfants de la configuration actuelle. On peut alors utiliser les algorithme de parcours de graphe ou de recherche par heuristique pour trouver la solution.

3.4 Heuristic

Cette classe représente une heuristique. Elle permet de calculer la valeur d'une configuration de jeu. Elle est utilisée par les algorithmes de recherche par heuristique pour trouver la solution.

Une heuristique est une fonction qui prend en paramètre une configuration de jeu (`GameState`), et qui retourne un nombre (`Int`). Plus ce nombre est petit, plus la configuration est proche de la solution. Plus ce nombre est grand, plus la configuration est éloignée de la solution.

La classe permet de composer des heuristiques en les combinant de la manière suivante avec des poids:

```
Heuristic.combine(
    heuristics: [
        Heuristic.countMisplacedCards,
        Heuristic.wrongColumnPlacement
    ],
    weights: [
        1,
        2
    ]
)
// or
Heuristic.combine([
    (1, Heuristic.countMisplacedCards),
    (2, Heuristic.wrongColumnPlacement)
])
```

`Heuristic.combine` prend en paramètre une liste d'heuristiques (fonctions prenant une configuration de jeu en paramètre et retournant un nombre) et une liste de poids. Elle retourne une heuristique (une fonction prenant une configuration de jeu en paramètre et retournant un nombre) qui est la somme des heuristiques pondérées.

3.5 Statistics

Cette classe permet de comparer les algorithmes de recherche par heuristique et les algorithmes de parcours de graphe. Elle permet de:

- `generateGames`
  Générer X configurations de jeu aléatoires de taille MxN.
- `getArrangements`
  Générer tous les arrangements N de nombres dans un intervalle donné, ce qui permet plus tard de tester les poids des heuristiques afin de trouver les plus performants.
- `findBestWeights`
  Met en competition les differentes heuristiques générées avec les poids passés en paramètre en utilisant A*, ce qui nous permet de determiner les poids générant l'heuristique la plus performante.
- `executeAlgorithmsOnMultipleGames`
  On donne N jeux et différents algorithmes et nous renvoit un rapport sur l'execution des algorithmes sur ces jeux afin de connaître le plus performant.
- `executeAlgorithmsOnOneGame`
  Execute plusieurs algorithmes sur un jeu donné et revoit un rapport de l'exécution.

3.6 Heuristics.playground

Le fichier `Heuristics.playground` va permettre d'utiliser la classe `Statistics` afin d'effectuer des analyses d'heuristiques et des parcours de graphe.
Il peut être amélioré afin d'afficher le composant graphique `StatUI` et d'effectuer l'animation de parcours des algorithmes comme sur l'application SwiftUI. On peut également l'utiliser pour faire un rapport en bonne et due forme avec nos résultats.

3.7 Documentation

Vous pouvez générez la documentation du projet en utilisant XCode. Pour cela allez dans `Product` -> `Build Documentation`. La documentation est générée via les commentaires de code.

Chapitre 4
Heuristiques

4.1 Méthodologie

J'ai choisis de faire en sorte que l'heuristique avec la valeur la plus basse représente un état le plus près de la solution. Dans cette logique j'ai alors implémentées trois heuristiques (détaillées dans les sections suivantes) et de les combiner pour obtenir une heuristique finale. Pour cela j'ai utilisé la méthode de la moyenne pondérée. Pour chaque heuristique j'ai déterminé un poids qui représente l'importance de l'heuristique dans la recherche de la solution. Pour déterminer ces poids j'ai créer une fonction dans la classe `Statistics` qui met en competition les heuristiques entre elles. Cette fonction prend en paramètre un nombre de parties à jouer et retourne un tableau de poids. Pour chaque partie, elle joue avec les trois heuristiques et enregistre le nombre de cartes mal placées de chaque heuristique. Elle trie ensuite ce tableau et retourne un tableau de poids où le poids de l'heuristique qui a le moins de cartes mal placées est en premier. Cette méthode permet de déterminer quelles heuristiques sont les plus efficaces et de les mettre en avant dans la recherche de la solution.

4.2 Heuristique 1 - Nombre de cartes mal placées

Cette heuristique renvoit le nombre de cartes mal placées. Relativement à l'as en début de ligne. C'est à dire que la couleur de l'as en début de ligne détermine la couleur de la ligne. Si une carte n'est pas de la même couleur que l'as en début de ligne alors elle est mal placée. Cette heuristique est relativement simple à implémenter et donne de bons résultats. Elle est donc utilisée comme heuristique de base.

4.3 Heuristique 2 - Nombre de cartes mal placées relatif à la colonne

Cette heuristique renvoit le nombre de cartes mal placées relativement à la colonne. Contrairement à la première heuristique, ici on ne prend pas en compte la couleur déterminée de la ligne. Elle regarde donc si la carte est bien placée dans sa colonne. Par exemple un 10, peu importe sa couleur, est mal placé si il est placé s'il est à la colonne 9 ou 11 alors, il devrait être à la colonne 10.

4.4 Heuristique 3 - Nombre de gap bloqués

Cette heuristique renvoit le nombre de gap bloqués. A noté qu'il y'a deux types de gaps bloqués:

- Un gap est bloqué par un autre gap qui le précède.
- Un gap est bloqué s'il suit la carte de rang maximal du jeu (par exemple une reine dans la configuration 13x4).

Elle prend alors deux paramètres (`stuckByMaxRankWeight` et `stuckByGapWeight`), qui sont les poids des deux types de gaps bloqués. Ces deux paramètres multiplient le nombre de gap bloqués par leur poids respectif.

La function de cette heuristique s'utilise diffemment des autres dû aux deux paramètres. Pour être utilisée il faut l'appeler avec les poids, pour obtenir la fonction finale.

```
let heuristic = Heuristic.stuckGaps(
    stuckByMaxRankWeight = 10,
    stuckByGapWeight = 1
)
```

4.5 Résultats

En utilisant la configuration suivantes:

```
let heuristics = [
    Heuristic.countMisplacedCards,
    Heuristic.stuckGaps(),
    Heuristic.wrongColumnPlacement
]

let bestWeights = await Statistics.findBestWeights(
    games: Statistics.generateGames(n: 5, rows: 4, columns: 13),
    range: 0...5,
    heuristics: heuristics,
    maxClosed: 3000
)
```

J'ai obtenue les poids suivants:

- Heuristique 1: 5
- Heuristique 2: 1
- Heuristique 3: 4

Ces résultats ne représentent pas forcément les meilleurs poids. Ils sont simplement le résultat de l'observation que j'ai fait sur les résultats de la fonction `findBestWeights` que je n'ai pas pu complétement éxecuter à cause de sa complexité.

Cela signifie que l'heuristique 1 est la plus importante, l'heuristique 2 est la moins importante et l'heuristique 3 est la deuxième plus importante. Ces poids sont donc utilisés pour l'application finale. Et peuvent être modifiés via le code dans le fichier `ContentView.swift` par la variable `_h`.

Une piste d'amélioration serait de faire varier les sous-poids de la deuxième heuristique avec `stuckByMaxRankWeight` et `stuckByGapWeight`. On peut le faire de la façon suivante:

```
let heuristics = [
    Heuristic.countMisplacedCards,
    Heuristic.stuckGaps(stuckByGapWeight: 0),
    Heuristic.stuckGaps(stuckByMaxRankWeight: 0),
    Heuristic.wrongColumnPlacement
]

let bestWeights = await Statistics.findBestWeights(
    games: Statistics.generateGames(n: 5, rows: 4, columns: 13),
    range: 0...5,
    heuristics: heuristics,
    maxClosed: 3000
)
```

Chapitre 5
Conclusion

Pour conclure, j'ai pu constater que l'implémentation d'un algorithme de résolution "Gaps" est assez complexe. Il est nécessaire de bien penser à la structure de données utilisée pour représenter le jeu et les coups possibles, ainsi que l'implémentation de l'algorithme de résolution. Il est également nécessaire de bien choisir les heuristiques à utiliser, ainsi que leurs poids pour obtenir le meilleur résultat possible. La fonction `findBestWeights` permet de trouver les meilleurs poids pour un ensemble d'heuristiques. J'ai pu constater que l'algorithme de résolution A* est plus efficace que l'algorithme de parcours en profondeur lorsque les heuristiques sont bien déterminées et efficaces. La fonction `findBestWeights` est une fonction très complexe à éxecuter, elle peut prendre beaucoup de temps pour trouver les meilleurs poids. Il est donc possible d'optimiser cette fonction pour qu'elle soit plus efficace.