Découverte de la nécessité de la pile pour la résolution d'un jeu de solitaire

Christophe Declercq, INSPE de la Réunion, LIM, IREMI

20 décembre 2021

Introduction

On expérimente une situation ludique pour découvrir la nécessité de la pile en classe de terminale pour la spécialité NSI. La structure de données pile a de nombreux usages en informatique dont le plus caractéristique est probablement la fonction `UNDO` telle qu'elle est mise en oeuvre par de nombreux éditeurs de textes ou d'images. Sa programmation par les élèves dans ce contexte présente cependant la difficulté de prise en main de l'environnement logiciel complexe que représente un éditeur interactif.

On propose alors comme situation fondamentale d'apprentissage de la pile, un jeu de recherche à résoudre en solitaire : le jeu de saute-mouton. Ce jeu a des règles de déplacement très simples mais son déroulement amène couramment à des situations de blocage où l'on souhaiterait revenir en arrière pour tester un autre déplacement.

Une mise en oeuvre instrumentée permet d'effectuer l'activité en ligne à l'aide d'un simple navigateur, les programmes python de l'élève étant interprétés par l'interprète Brython [4].

L'activité est disponible à l'adresse :
`https://iremi974.gitlab.io/python-de-la-fournaise/sautemouton.html`

Présentation de l'activité

Règles et déroulement du jeu de saute-mouton

Le jeu comporte sept cases alignées numérotées de 0 à 6, trois jetons 'moutons blancs' et trois jetons 'moutons noirs'. Les moutons blancs (resp. noirs) peuvent se déplacer vers la droite (resp. gauche) vers une case vide ou en sautant par dessus un mouton noir (resp. blanc). L'objectif est de d'inverser les moutons blancs et les moutons noirs.

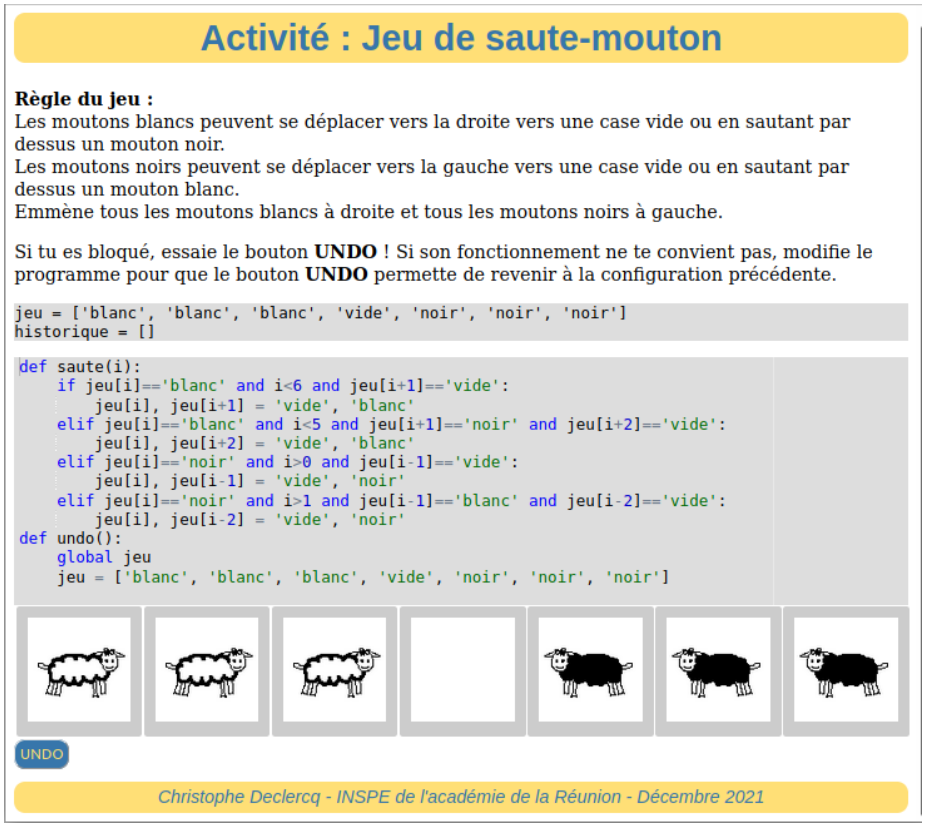

Figure 1: Interface du jeu de saute-mouton

L'interface proposée montre le code de deux fonctions, la fonction `saute(i)` qui est appelée à chaque clic sur une case du jeu et permet de déplacer le mouton qui se trouve sur cette case si le déplacement est autorisé, et la fonction `undo()` qui est appelée à chaque clic sur le bouton `undo` et est programmée de manière naïve en ramenant directement à la position initiale.

Notons que l'ensemble du code peut aussi être recopié dans un éditeur de programme et testé simplement avec la console python, par exemple avec la suite d'appels suivante :

```
saute(2)
saute(4)
saute(3)
saute(1)
jeu
undo()
```

A l'issue des quatre premiers mouvements, on arrive à une situation de blocage caractéristique avec la variable `jeu` à `['blanc', 'vide', 'noir', 'blanc', 'blanc', 'noir', 'noir']` où deux moutons blancs font face à deux moutons noirs sans qu'aucun ne puisse plus avancer. C'est la situation déclenchante qui va permettre de réfléchir à la nécessité d'un retour arrière avec la fonction `undo`.

L'interface et les parties de code qui ne sont pas montrées à l'élève ne sont utiles que pour fournir un retour instrumental plus élaboré. Après chaque mouvement, la visualisation graphique est recalculée à partir des nouvelles valeurs contenues dans le tableau `jeu`.

Analyse a priori de l'activité

Le bouton `undo`, tel qu'il est programmé initialement, a un comportement trop radical puisqu'il ramène à la configuration initiale. La réflexion sur la nécessité de revenir seulement à la configuration précédente devrait amener les élèves à construire la nécessité d'une *variable* mémorisant cette configuration précédente.

Dans un premier temps une variable peut effectivement suffire, reste à savoir quand l'initialiser - avant de jouer un coup - et quand utiliser la valeur mémorisée - au clic sur la fonction `undo` - pour remettre dans le tableau `jeu`, la configuration mémorisée.

Cette première étape peut être utile dans la résolution du problème initial et amènera alors à un obstacle quand, lors d'une partie de jeu, le joueur s'apercevra qu'il a besoin de revenir en arrière de plus d'un mouvement.

Se pose alors inductivement la question de la mémorisation de deux, trois puis de tous les mouvements depuis le début du jeu. C'est à ce moment précis qu'il est utile de faire réfléchir les élèves sur le moment où ils risquent d'avoir besoin des valeurs mémorisées. Le premier clic sur la fonction `undo` a besoin de la dernière valeur mémorisée, le second clic de la précédente...

Une digression, via l'informatique débranchée, peut permettre de matérialiser, avec des petits papiers sur lesquels sont notées les configurations, la manière de les ranger en les empilant les uns sur les autres à chaque mouvement, puis en les dépilant lors d'un ou plusieurs `undo`.

La présence dans le programme fourni, de la variable `historique`, initialisée en tant que liste vide, est prévue pour inciter à l'usage de cette liste pour y mémoriser les configurations successives du jeu. Il ne reste plus qu'à fournir aux élèves les méthodes d'accès pour ajouter en fin de liste `append`, et retirer un élément à cette même fin de liste `pop`, pour leur permettre de réussir l'activité.

L'institutionnalisation prendra soin de préciser qu'on appelle **pile** cette manière particulière de mémoriser une série de données, lorsque la dernière à ajouter - à empiler - doit être la première à retirer - à dépiler.

Plusieurs solutions peuvent être imaginées pour enregistrer l'historique, l'historique des coups joués - `[2, 4, 3, 1]` dans notre exemple -, ou l'historique des configurations du jeu.

Chaque solution a ses avantages et ses inconvénients : l'historique des coups est une structure de données plus élémentaire mais nécessite le recalcul des configurations ; l'historique des configurations est une liste (pile) de tableaux, ce qui nécessite en Python des précautions pour éviter les alias.

Une aide peut être fournie en précisant qu'il convient de faire une *copie* dans l'historique, de la configuration de jeu actuelle, pour éviter que cette mémorisation ne soit modifiée lors de la suite du déroulement du jeu.

Correction de l'activité

Parmi les solutions possibles, on propose la suivante, en tant que corrigé :

```
def saute(i):
    historique.append(jeu.copy())
    if jeu[i]=='blanc' and i<6 and jeu[i+1]=='vide':
        jeu[i], jeu[i+1] = 'vide', 'blanc'
    elif jeu[i]=='blanc' and i<5 and jeu[i+1]=='noir' and jeu[i+2]=='vide':
        jeu[i], jeu[i+2] = 'vide', 'blanc'
    elif jeu[i]=='noir' and i>0 and jeu[i-1]=='vide':
        jeu[i], jeu[i-1] = 'vide', 'noir'
    elif jeu[i]=='noir' and i>1 and jeu[i-1]=='blanc' and jeu[i-2]=='vide':
        jeu[i], jeu[i-2] = 'vide', 'noir'
    else :
        historique.pop()

def undo():
    global jeu
    jeu = historique.pop()
```

On a choisi ici de mémoriser les configurations et de réaliser la pile par les méthodes `append` et `pop` des listes. Le choix d'empiler systématiquement la configuration précédente, avant de tester si le coup est jouable, peut amener à empiler une configuration identique à la précédente. C'est la raison de la correction proposée ici, consistant à dépiler dans le `else` une configuration empilée inutilement. Une solution plus élémentaire consisterait à empiler uniquement dans les alternatives quand le coup est jouable. Cette réflexion est marginale et n'est bien sûr pas exigible de tous les élèves. Si on omet le `else`, l'historique sera seulement un peu plus long (avec des configurations répétées) de même que la suite des `undo`.

L'essentiel est dans la compréhension du mécanisme : si la mémorisation de l'historique a bien été effectuée lors du jeu, la fonction `undo` consiste à dépiler la dernière configuration mémorisée pour la remettre en `jeu`.

L'indication de variable globale pour `jeu` est fournie pour éviter un problème dû à la spécificité des variables mutables Python associée à la déclaration automatique des variables à leur première affectation. Si on l'omet, c'est une nouvelle variable locale qui est créée sans rapport avec la variable globale définie à l'initialisation et utilisée dans la fonction `saute`.

Conclusion

La présente situation a été mise en ligne pour être proposée à l'expérimentation. Le recueil de données permettra de valider ou non les hypothèses relatives à son usage.

On postule en particulier les hypothèses suivantes :

- Une situation ludique, de type jeu de solitaire, est de nature à engager les élèves dans l'activité.
- La difficulté de trouver directement la stratégie gagnante du jeu encourage à améliorer le jeu en programmant le bouton `undo`.
- la situation est signifiante par rapport à l'usage de la pile et est à ce titre candidate pour être considérée comme une `situation fondamentale`.

On se permet de plus de penser que cette situation a plus de sens que les exercices de style consistant à programmer une file avec deux piles [1] ou à dépiler/rempiler une pile pour en calculer la hauteur [2].

La proposition est transposable à n'importe quel jeu de réussite en solitaire [3] comme les jeux de plateau en bois disposés en croix où l'objectif est de retirer toutes les billes sauf une.

On montre ainsi le caractère générique de la pile pour empiler n'importe quelle données représentant la configuration courante d'un jeu quel qu'il soit, pour ensuite dépiler au besoin quand l'utilisateur veut utiliser la fonction `UNDO`.

Références

[1] Programmes et ressources en numérique et sciences informatiques—Voie G. éduscol, Ministère de l'Éducation nationale, de la Jeunesse et des Sports - Direction générale de l'enseignement scolaire. Consulté 20 décembre 2021, à l'adresse https://eduscol.education.fr/2068/programmes-et-ressources-en-numerique-et-sciences-informatiques-voie-g

[2] Sujets zéro et spécimens pour le baccalauréat GT. éduscol, Ministère de l'Éducation nationale, de la Jeunesse et des Sports - Direction générale de l'enseignement scolaire. Consulté 20 décembre 2021, à l'adresse https://eduscol.education.fr/1987/sujets-zero-et-specimens-pour-le-baccalaureat-gt

[3] Le jeu du Solitaire, Python des Neiges https://sebhoa.gitlab.io/iremi/01_Graphes/solitaire/

[4] Brython, Une implémentation de Python 3 pour la programmation web côté client.Consulté à l'adresse https://www.brython.info/