Filière MP - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2019

Page de garde du rapport de TIPE

NOM :	MULOT	Prénoms :	Lendy, Henry, Renaud
Classe :	MP*		
Lycée :	Carnot	Numéro de candidat :	4909
Ville :	Dijon		

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

ENS Cachan	MP - Option MP		MP - Option MPI	
	Informatique			
ENS Lyon	MP - Option MP		MP - Option MPI	
	Informatique - Option M	X	Informatique - Option P	
ENS Rennes	MP - Option MP		MP - Option MPI	
	Informatique	X		
ENS Paris	MP - Option MP		MP - Option MPI	
	Informatique			

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique	X	Mathématiques		Physique	

Titre du TIPE : Résolution numérique d'un problème de logique : le Sokoban

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	4	Illustration	33	Bibliographie	1

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

Le Sokoban est un jeu de logique où le joueur incarne un personnage dans un labyrinthe, devant transporter des caisses pour les amener dans des emplacements spécifiques. Ce TIPE a pour but de résoudre informatiquement les niveaux de petite et moyenne taille.

À Dijon

Le 11 / 06 / 2019

Signature du (de la) candidat(e)

Signature du professeur responsable de la classe préparatoire dans la discipline

Cachet de l'établissement

LYCÉE CARNOT DIJON

La signature du professeur responsable et le tampon de l'établissement ne sont pas indispensables pour les candidats libres (hors CPGE).

Résolution numérique d'un problème de logique : le Sokoban

MULOT Lendy

Table des matières

I) Position du problème

1) Le Sokoban

Le Sokoban est un jeu de logique créé par Hiroyuki IMABAYASHI dans les années 1980. Le joueur incarne un garde d'entrepôt se trouvant dans un labyrinthe à deux dimensions dans lequel sont disposées un certain nombre de caisses (voir Figure 1 p. 7). Afin de résoudre le niveau, il suffit d'amener les caisses dans des positions prédéfinies. Pour cela, le joueur peut déplacer son personnage vers le haut, le bas, la droite ou la gauche. Lorsqu'il se trouve à côté d'une caisse, il a la possibilité de pousser celle-ci mais il ne peut pas la tirer. Il ne peut pas non plus pousser deux caisses en même temps ni passer au travers d'un mur du labyrinthe. Le but est alors de finir le niveau avec le moins de déplacements possibles du personnage.

2) Problématique retenue

La résolution des niveaux du Sokoban est un problème difficile d'un point de vue informatique bien que le jeu soit très simple à comprendre. Il appartient d'ailleurs à la classe des problèmes NP-Difficiles (voir **[2]** et **[4]**) et PSPACE-Complets (voir **[2]** et **[3]**), d'où le fait que les solveurs actuels ne sont capables de résoudre qu'une partie des niveaux du Sokoban en un temps raisonnable (voir **[1]** et **[2]**). Mon binôme et moi avons donc retenu la problématique suivante : comment modéliser et résoudre informatiquement les niveaux du Sokoban ?

3) Objectifs du TIPE

Dans un premier temps, nous avons donc cherché à implémenter une structure efficace pour représenter un niveau de Sokoban et permettre son évolution. Il faut ensuite trouver une structure de données adéquate pour gérer la résolution d'un niveau. Par la suite nous devons palier aux principaux obstacles en évitant de traiter plusieurs fois un même état du jeu et en détectant d'éventuelles situations bloquantes afin de réduire le temps de recherche. Enfin, ce TIPE ne se veut pas trop ambitieux et nous avons pour objectif final de pouvoir trouver une solution optimale (c'est-à-dire avec le moins de déplacements possibles) rapidement pour des petits niveaux et en un temps raisonnable pour des niveaux de difficulté moyenne.

II) Modélisation du Sokoban

1) Attributs

Pour plus de clarté dans le code et pour avoir accès plus facilement aux données utiles en rapport avec un niveau, et pour répondre à notre premier objectif, nous avons décidé de créer une classe *Jeu* (voir listing 1 p. 13 à p. 16) contenant tout le nécessaire pour représenter un niveau et le faire évoluer. Une instance de cette classe possède les attributs suivants : $case, laby, pc, pj, pv$ (voir Figure 1 et Figure 2 p. 7 et listing – 2 p. 17 à p. 21).

L'attribut $laby$ est un tableau à deux dimensions représentant le labyrinthe. Nous avons choisi de représenter les différents types de cases par différents entiers. Ainsi 0 représente une case vide ; 1 représente un mur et donc une case inaccessible ; 2 représente une caisse ; 5 représente le joueur.

Les attributs pc, pj, pv représentent respectivement les positions des caisses, du joueur et les positions de victoire. La position du joueur est représentée par un couple correspondant aux coordonnées du joueur dans la matrice. Les positions des caisses et positions de victoires sont représentées par des listes de listes à deux éléments, correspondant aux coordonnées de la caisse ou de la position de victoire. Bien qu'il soit possible de retrouver les positions des caisses directement à partir du labyrinthe, les stocker permet d'y accéder en temps constant plutôt qu'en temps $\theta(N)$ si l'on devait parcourir un labyrinthe à N cases.

L'attribut $case$ contient l'entier codant la case présente sous le joueur. Elle n'était pas présente au début du jeu et son utilité sera expliquée dans la partie V)1).

2) Méthodes

Une fois le jeu modélisé, il convient de le faire évoluer en le modifiant plutôt qu'en recréant une nouvelle instance. Pour chaque direction, nous avons développé deux méthodes. La première permet de vérifier si le mouvement dans ladite direction est possible, la seconde effectue ce mouvement s'il est possible. Pour vérifier cela, il suffit de considérer la case adjacente à celle du joueur dans la direction voulue, si elle est vide alors le mouvement est possible, si elle contient un mur alors il est impossible et si elle contient une caisse alors le mouvement n'est possible que si la case suivante est vide. Lors du déplacement d'une caisse, les anciennes et nouvelles coordonnées de celle-ci sont renvoyées, permettant d'annuler un coup lors de la résolution en se déplaçant dans le sens opposé et en redonnant à la caisse sa position initiale.

Puis pour simplifier et ne pas devoir utiliser quatre méthodes différentes, nous avons écrit une méthode générique $avancee$ prenant en argument la direction souhaitée et appelant la méthode correspondante.

III) Résolution

1) Principe de résolution

Nous nous sommes naturellement tournés vers une approche récursive pour la résolution d'un niveau, en limitant le nombre de déplacements possibles. Nous distinguons alors deux cas de base, soit le niveau est résolu (c'est-à-dire que chaque caisse se trouve dans une position de victoire), auquel cas la suite de déplacements ayant conduit à cet état est une solution, soit le nombre maximal de coups est atteint et il faut alors repartir en arrière. Si les résultats de ces deux tests sont tous les deux négatifs, il suffit de décrémenter la limite de coups et d'effectuer un appel récursif sur chaque direction où le déplacement est possible, tout en annulant le coup avant de passer à la direction suivante si aucune solution n'a été trouvée.

Dès lors, nous pourrions modéliser ceci par un arbre dans lequel chaque nœud possèderait au plus quatre fils, correspondant aux directions où le mouvement est possible. Ainsi la résolution s'apparenterait à un parcours en profondeur de cet arbre pour trouver un nœud correspondant à un état de victoire (voir Figure 3 p. 10).

Nous avons alors développé la fonction $solveur\_general$ (voir listing 3 p. 23 et p. 24) prenant en argument un niveau de Sokoban, deux fonctions auxiliaires pre_aux et $fonction\_aux$, un nombre de coups maximum et un dictionnaire contenant tous les autres arguments utiles aux fonctions auxiliaires dont notamment une liste qui contiendra la suite des coups ayant amené à l'état du jeu. La fonction pre_aux est appelée après le test des deux cas de base. Dans tous les cas, si elle renvoie *False* alors le jeu est considéré comme perdu et on n'effectue plus aucun appel récursif. Le fonctionnement de cette fonction dépend de l'approche utilisée et sera expliqué dans la partie IV). La seconde fonction auxiliaire est appelée après chaque déplacement et c'est elle qui effectue les appels récursifs et annule le coup. Son fonctionnement dépend aussi de l'approche utilisée.

Par la suite, on peut aisément trouver une solution optimale en faisant varier le nombre maximal de coups afin de trouver la limite à partir de laquelle il n'y a plus de solution.

2) Solveur naïf

Il convient de commencer nos tentatives par une implémentation naïve (voir listing 3 p. 24 à p. 26). Dans un premier temps, la fonction pre_aux va effectuer une sauvegarde de la liste des positions des caisses alors que $fonction\_aux$ va effectuer un mouvement puis lancer un appel récursif au solveur et enfin annuler le coup en effectuant le déplacement en sens inverse et en remplaçant la liste des caisses par celle sauvegardée par pre_aux. Cependant, les listes étant des objets mutables en Python, cette approche nécessite de nombreuses copies de liste inutiles et chronophages. C'est pourquoi nous avons décidé que les méthodes de déplacement devaient, si une caisse venait à être déplacée, renvoyer les anciennes et nouvelles positions de la caisse déplacée ainsi que l'indice de celle-ci dans la liste des caisses afin de ne pas perdre de temps.

IV) Optimisations : limiter le traitement de cas identiques

1) Utilisation de tables de hachage

Le premier problème dans la résolution du Sokoban est d'éviter de traiter plusieurs fois un même état. Il faut par conséquent associer à un état du jeu une clé le représentant. Nous avons dans un premier temps choisi de représenter une table de hachage par la donnée de deux listes. La case d'indice i de la première contenant *True* si l'état de clé i a déjà été vu et *False* dans le cas contraire, et la deuxième liste contient le nombre minimal de coups restant lors d'un passage par cet état.

Dans ce cas, la fonction pre_aux n'a pas d'effet et $fonction\_aux$ n'effectuera l'appel récursif que si l'état n'a pas encore été visité ou bien si le nombre de coup restant est inférieur à celui contenu dans la table, et n'oubliera pas de mettre à jour la table de hachage.

Cependant, le nombre d'état possible étant immense, cela nécessite une liste d'autant plus grande et donc un temps irraisonnable de création, sachant que la plupart des cases ne seront pas utilisées.

2) Utilisation de dictionnaires

Cette dernière remarque est la raison pour laquelle nous nous sommes tournés vers l'utilisation de dictionnaires (voir listing3 p. 30 à p. 32). Une instance de la classe *Jeu* étant mutable, nous ne pouvons donc pas utiliser le jeu lui-même comme clé et nous devons donc effectuer un premier hachage. Cependant nous n'avons plus la contrainte de devoir renvoyer un entier. Afin d'obtenir une fonction injective simple. Nous avons utilisé la fonction suivante :

$$h(x_1, x_2, \dots, x_n) = "x_1 \ x_2 \ \dots \ x_n" \quad (1)$$

où les x_i représentent les différentes coordonnées des caisses et du joueur.

Pour cette modélisation, pre_aux n'aura pas d'effet et $fonction\_aux$ agira de façon similaire à celle décrite en IV)1) mais adaptée à la structure de dictionnaire.

Une petite optimisation a été effectuée : trier la liste des positions des caisses. En effet, l'ordre de celles-ci n'importe pas dans la résolution mais la fonction de hachage donnerait cependant des résultats différents si l'on permutait deux caisses.

Cette structure est la plus efficace que nous ayons trouvée, elle conclut cette partie du TIPE.

V) Optimisations : *deadlocks*

1) Détection avant la résolution

Une autre façon de diminuer le nombre d'états visités consiste à détecter des situations bloquantes, c'est-à-dire à partir desquelles le joueur ne peut plus gagner et où il est donc inutile de continuer la recherche de solution. On peut séparer ces *deadlocks* (terme repris de **[2]**) en deux catégories : ceux impliquant une seule caisse et ceux en impliquant plusieurs. Cette partie s'intéresse à la première.

A titre d'exemple, si une caisse se retrouve dans un coin alors le joueur ne peut plus la sortir et si ce n'est pas une position de victoire alors la partie est perdue (voir Figure 4 p. 9). De même, si cette caisse se trouve le long d'un mur sans la possibilité d'en sortir et si aucune position de victoire le long de celui-ci alors le jeu est là encore perdu (voir Figure 5 et Figure 6 p. 9). Or n'impliquant qu'une seule caisse, ces positions peuvent être repérées avant la résolution du jeu, et marquée pour indiquer que le joueur ne doit pas pousser une caisse sur ces positions mais qu'il peut lui-même se trouver dessus. Nous commençons donc par marquer ces cases avec l'entier 4 (voir Figure 7 p. 10). C'est alors qu'est venu l'intérêt de l'attribut *case* pour ne pas supprimer une case marquée lors de la résolution.

2) Détection pendant la résolution

Par exemple, deux caisses l'une à côté de l'autre contre un mur ne pourront pas être déplacées, (voir Figure 8 p. 10) ou encore un carré de quatre caisses ne pourra pas non plus être déplacé (voir Figure 9 p. 10).

Les *deadlocks* à plusieurs caisses sont plus difficiles à détecter et le seront donc pendant la résolution. Or cette vérification demande du temps et par conséquent ne peut pas être effectuée à chaque appel récursif. Augmenter les vérifications augmentera la durée tout en diminuant le nombre d'états visités. Après un certain nombre de mesures effectuées, nous avons conclu qu'effectuer cette vérification tous les sept coups était un bon compromis. Cette vérification est alors effectuée dans la fonction pre_aux qui mettra alors fin à la recherche si le jeu est bloqué.

VI) Résultats obtenus

Alors que le solveur naïf est déjà trop lent si la résolution nécessitait plus d'une vingtaine de coups et que le solveur de hachage est efficace mais trop chronophage pour créer la table, le solveur utilisant les dictionnaires est déjà capable de résoudre en un temps bien moindre des niveaux plus conséquents.

Nous avons ensuite effectué différents tests sur certains niveaux afin de mesurer le temps de résolution d'un niveau et le nombre d'états visités pour les différents solveurs en fonction du nombre de coups maximal pour lequel est appelé le solveur (voir Graphique 1 p. 11). Nous avons par la suite effectué les mêmes mesures mais avec le solveur dictionnaire qui était le plus efficace, pour différents cas : solveur seul, solveur avec le tri de la liste des caisses, solveur avec détection des *deadlocks*, solveur avec les deux optimisations précédentes (voir Graphique 2 p. 11, Graphique 3 et Graphique 4 p. 12). Nous constatons que l'utilisation de ces deux optimisations permet de réduire le nombre d'états visités d'un facteur dix, allant jusqu'à un facteur cent pour certains niveaux.

Cette version permet donc de résoudre sans problème les petits niveaux et de résoudre plusieurs niveaux nécessitant plus d'une centaine de coups en des temps corrects (allant jusqu'à 1h30 pour un niveau nécessitant 133 coups). Notre objectif est alors rempli.

VII) Améliorations possibles

A partir de ce point, il me parait difficile d'aller plus loin avec ce type de résolution. Il serait toujours possible d'implémenter de nouveaux *deadlocks* mais l'optimisation ne serait pas significative. Pour accélérer la résolution il me paraitrait intéressant d'utiliser un parcours en largeur plutôt qu'en profondeur, mais cela nécessiterait de pouvoir reconstruire rapidement un état du jeu. Sinon, il faudrait changer complétement d'approche.

Le problème étant PSPACE-Complet (voir **[2]** et **[3]**), il est donc nécessairement possible de trouver une structure de donnée bien moins gourmande en espace mémoire.

Le document **[2]** propose différentes améliorations telles que l'utilisation d'un algorithme de recherche de plus court chemin afin d'atteindre une caisse, ou encore l'assignation de pénalités lors de déplacements pouvant probablement bloquer le jeu.

Références

[1] Nicolas Baskiotis : *Sokoban - Solveur* :
http://www-connex.lip6.fr/~baskiotisn/index.php/2016/10/19/sokoban-solveur/

[2] Michaël Hoste : *Jeu de Sokoban - recherche de solutions optimales* :
http://informatique.umons.ac.be/ftp_infofs/2008/Hoste2008-memoire.pdf

[3] Joseph C. Culberson *: Sokoban is PSPACE-complete* :
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.41&rep=rep1&type=pdf

[4] Dorit Dor, Uri Zwick : *SOKOBAN and other motion planning problems (extended abstract)* :
http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=C8AD9809463B8326F2896D3984C87CBD?doi=10.1.1.50.585&rep=rep1&type=pdf

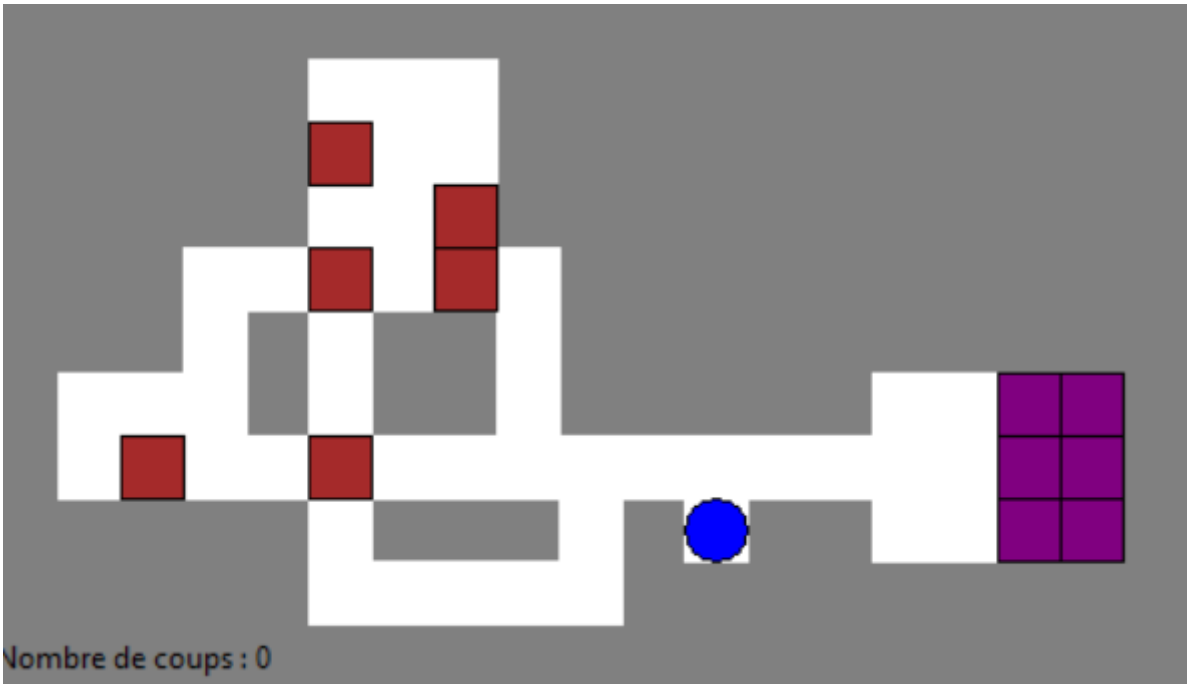


Figure 1 : Exemple de niveau de Sokoban. En gris les murs ; en bleu le joueur ; en violet les positions de victoire ; en marron les caisses.

```
lvl1 = np.array(
[[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,2,0,0,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,0,0,2,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,0,0,2,0,2,0,1,1,1,1,1,1,1,1,1,1],
[1,1,1,0,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1],
[1,0,0,0,1,0,1,1,0,1,1,1,1,1,0,0,0,0,1],
[1,0,2,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1],
[1,1,1,1,1,0,1,1,1,0,1,5,1,1,0,0,0,0,1],
[1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]])


jeu1 = Jeu(lvl1,                                                    #laby
(8, 11),                                                            #pj
[[2, 5], [3, 7], [4, 5], [4, 7],[7, 2], [7, 5]],                    #pc
[[6, 17], [7, 17], [8, 17], [6, 16], [7, 16], [8, 16]]) #pv
```

Figure 2 : Niveau implémenté. Ceci représente le niveau en Figure 1.

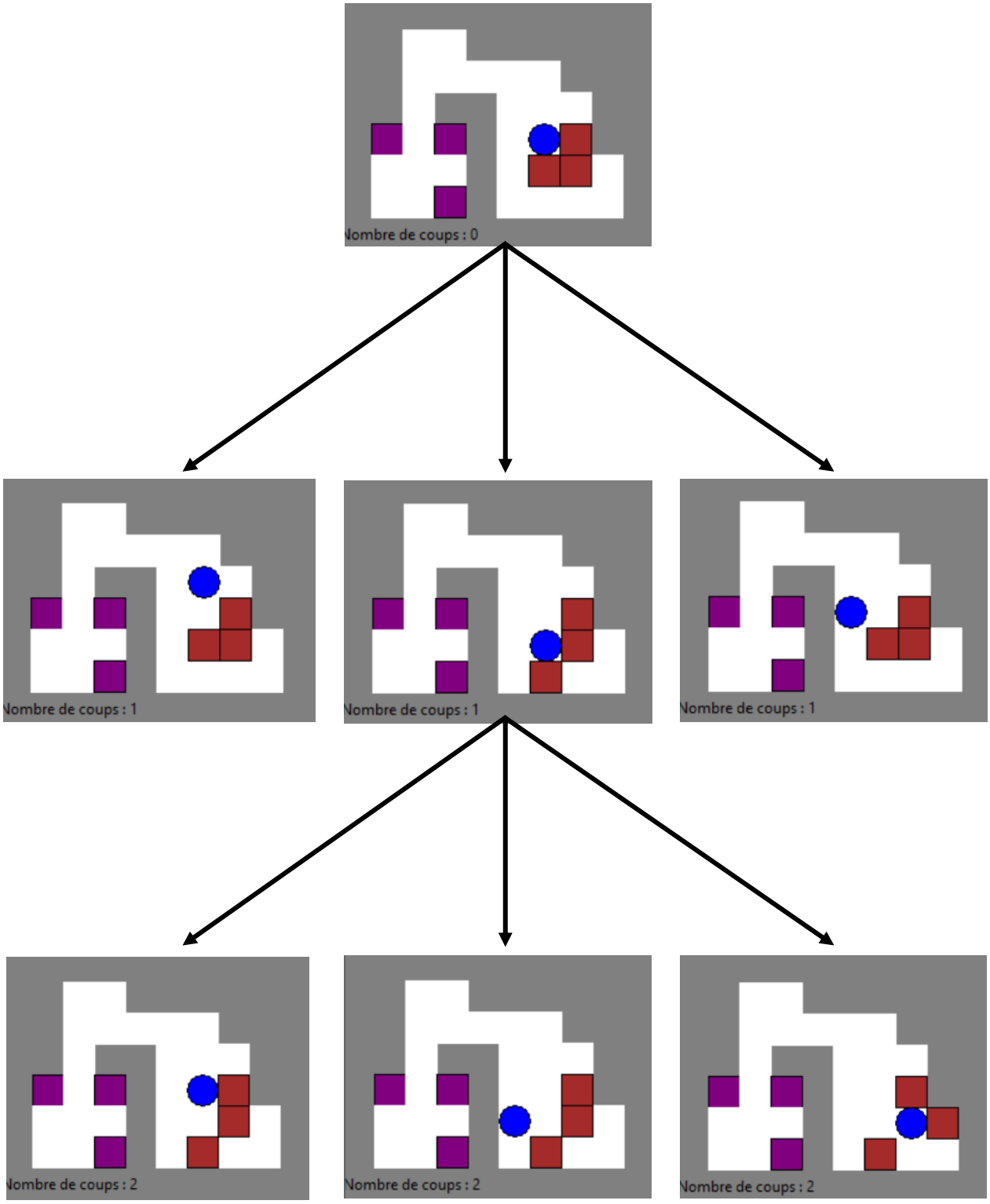


Figure 3 : Partie de l'arbre représentant les différents mouvements effectués par le solveur.

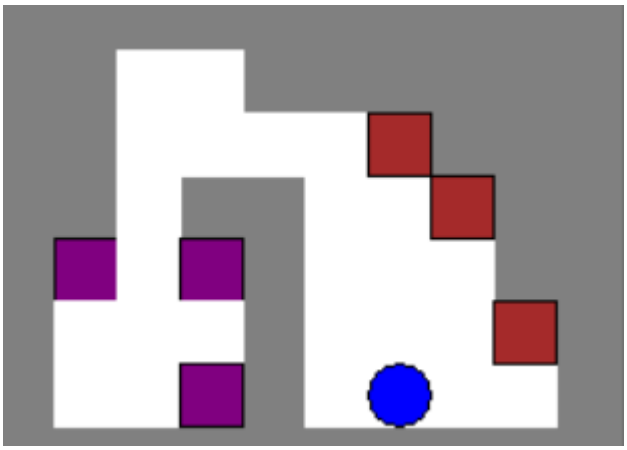


Figure 4 : Exemple de deadlock dans un coin. Ici les trois caisses ne peuvent plus être déplacées.

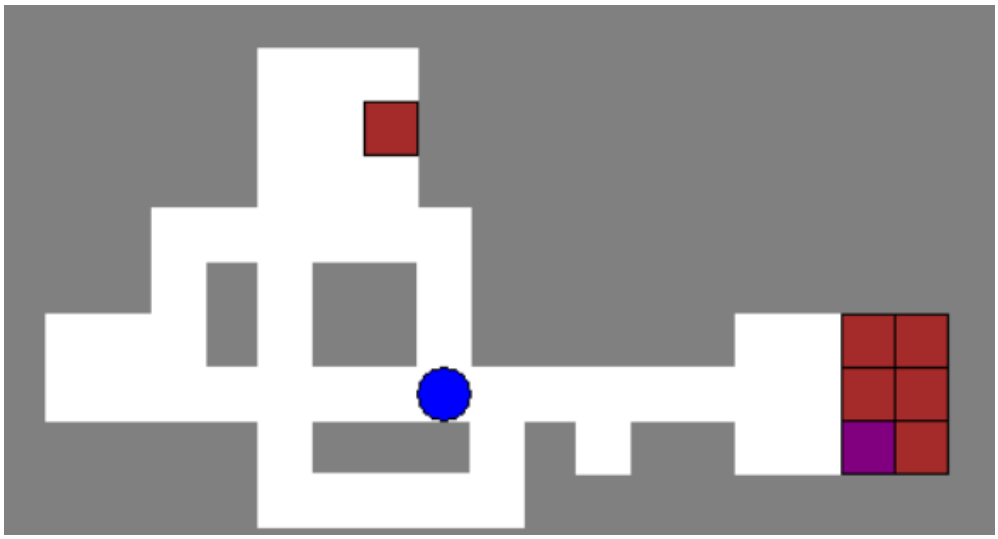


Figure 5 : Caisse contre un mur, non bloquée.

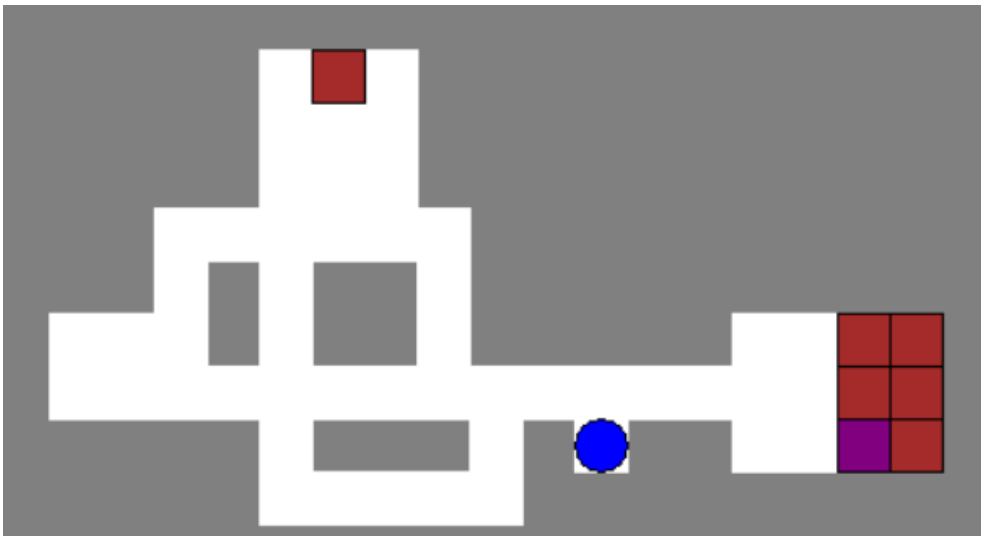


Figure 6 : Caisse contre un mur, bloquée.

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 2, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 0, 0, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 4, 0, 2, 0, 2, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 4, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 4, 0, 0, 1],
       [1, 4, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 5, 1, 1, 4, 0, 0, 1],
       [1, 1, 1, 1, 1, 4, 4, 4, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
Position du joueur : (8, 11)
Positions des caisses : [[2, 5], [3, 7], [4, 5], [4, 7], [7, 2], [7, 5]]
Positions de victoire : [[6, 17], [7, 17], [8, 17], [6, 16], [7, 16], [8, 16]]
```

Figure 7 : Niveau de la Figure 1 après marquage des deadlocks à une caisse.

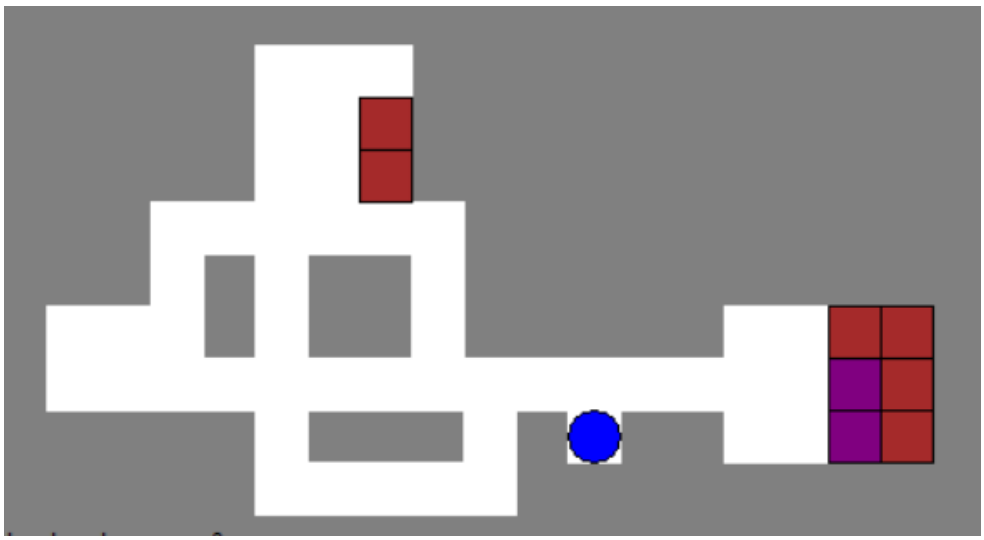


Figure 8 : Deux caisses l'une contre l'autre, bloquées le long d'un mur.

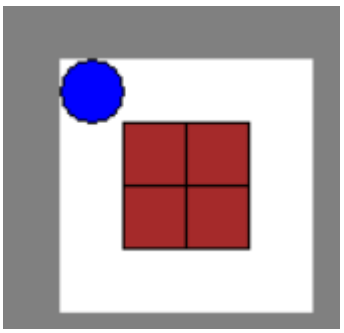
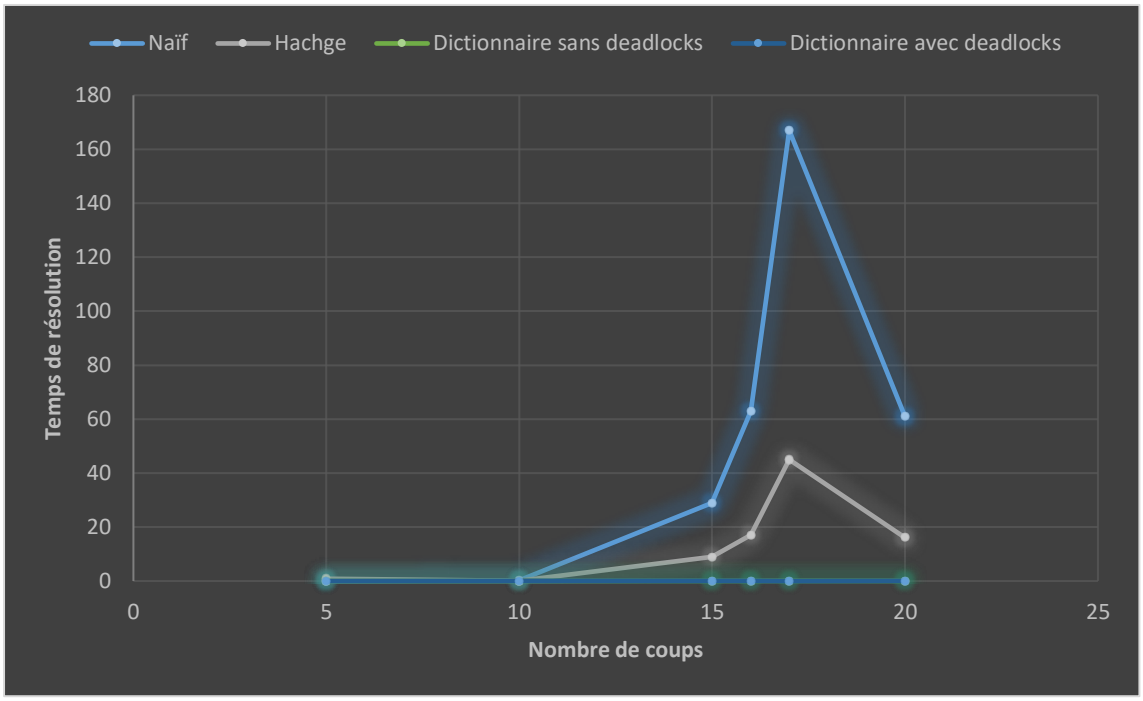
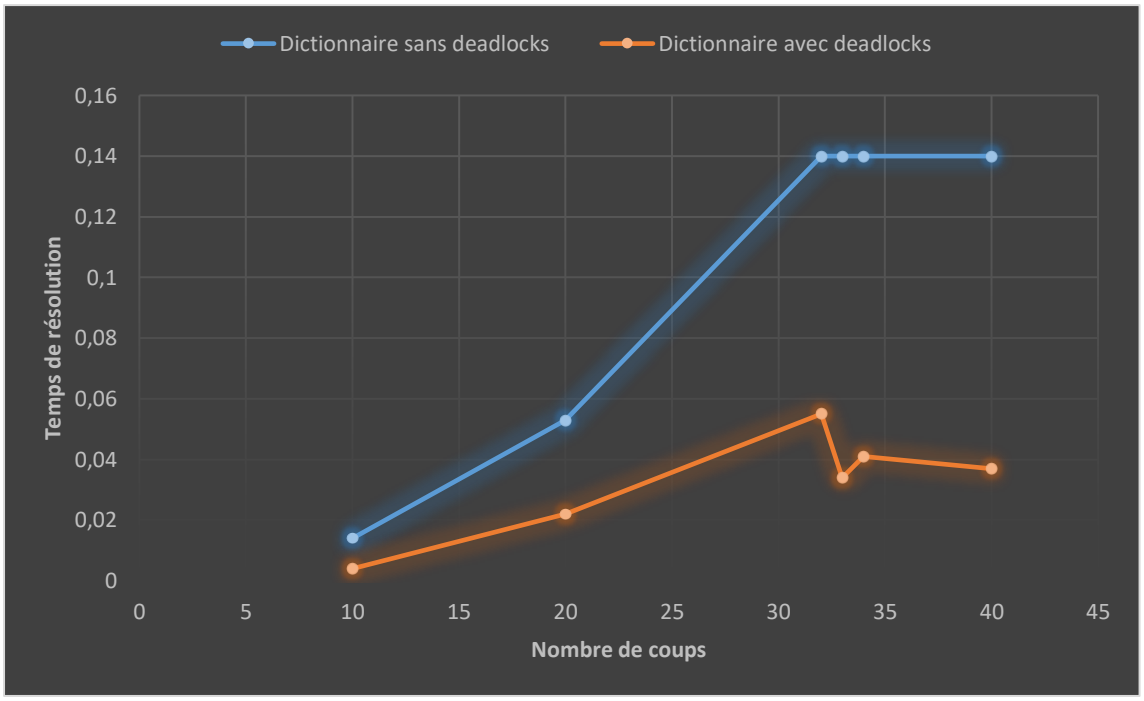


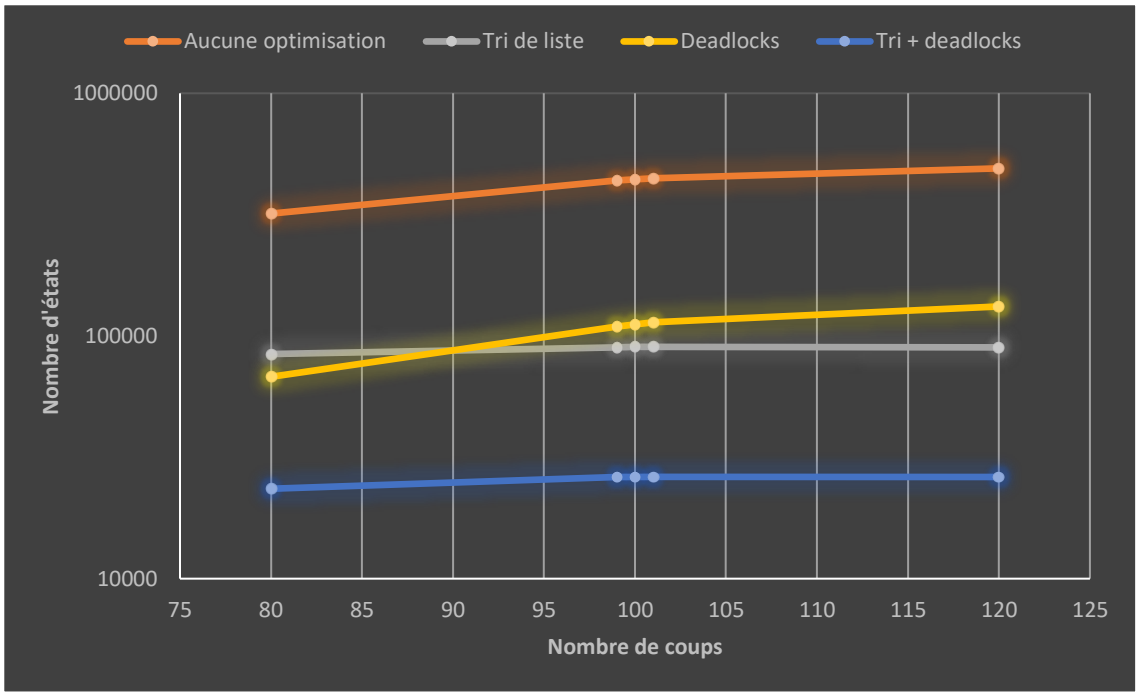
Figure 9 : Quatre caisses bloquées en formant un carré.



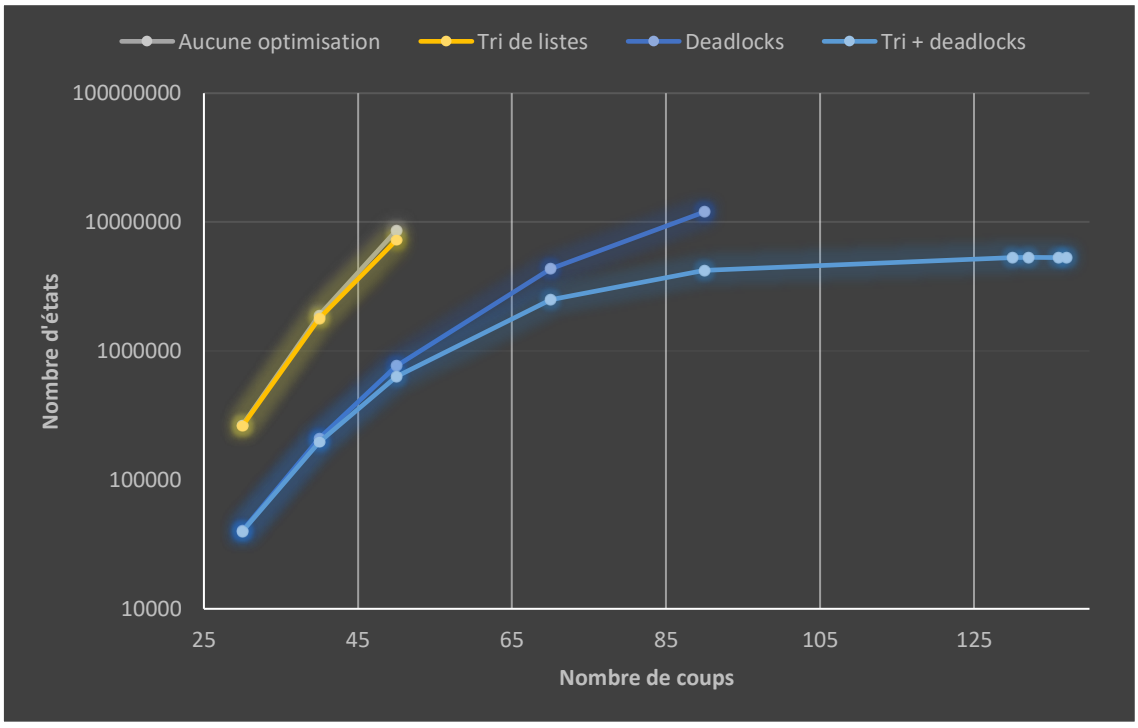
Graphique 1: Temps de résolution en fonction du nombre limite de coups pour différents solveur. Le niveau nécessitait au minimum 16 coups.



Graphique 2 : Temps d'exécution avec et sans utilisation des deadlocks pour un niveau nécessitant 33 coups.



Graphique 3 : Nombre d'états visités en fonction du nombre limite de coups pour différentes optimisations. Le niveau nécessitait au minimum 100 coups.



Graphique 4 : Nombre d'états visités en fonction du nombre limite de coups pour différentes optimisations. Le niveau nécessitait au minimum 133 coups.

Listing 1 – La classe de jeu

```
"""
########################################
#                TIPE                  #
########################################
"""


"""
########################################
#          Classe de Jeu               #
########################################
"""
import numpy as np

class Jeu:
    "Definit l'objet jeu"

    def __init__(self, labyrinthe, pos_joueur, pos_caisses, pos_victoire, case=0):
        self.laby = labyrinthe
        self.pj = pos_joueur
        self.pc = pos_caisses
        self.pv = pos_victoire
        self.case = case

    def __repr__(self):
        res = self.laby.__repr__() + '\n'
        res += 'Position du joueur : ' + str(self.pj) + '\n'
        res += 'Positions des caisses : ' + self.pc.__repr__() + '\n'
        res += 'Positions de victoire : ' + self.pv.__repr__()
        return res

    def go_h (self, check=False, ret=False) :
        "Modifie (si possible) le labyrinthe en faisant monter le joueur"
        if check or self.go_h_bool():
            x, y = self.pj
            if self.laby[x - 1, y] == 2 :
                self.laby[x - 2, y] = 2
                self.laby[x, y], self.case = self.case, 0
                self.laby[x - 1, y] = 5
                self.pj = (x - 1, y)
```

```
                i = self.pc.index([x - 1, y])
                self.pc[i] = [x - 2, y]
                if ret :
                    return True, i, [x - 1, y], [x - 2, y]
            else :
                self.laby[x, y], self.case = self.case, self.laby[x - 1, y]
                self.laby[x - 1, y] = 5
                self.pj = (x - 1, y)
                if ret:
                    return False, None, None, None
        if ret:
            return False, None, None, None

    def go_b (self, check=False, ret=False) :
        "Modifie (si possible) le labyrinthe en faisant descendre le joueur"
        if check or self.go_b_bool():
            x, y = self.pj
            if self.laby[x + 1, y] == 2 :
                self.laby[x + 2, y] = 2
                self.laby[x, y], self.case = self.case, 0
                self.laby[x + 1, y] = 5
                self.pj = (x + 1, y)
                i = self.pc.index([x + 1, y])
                self.pc[i] = [x + 2, y]
                if ret :
                    return True, i, [x + 1, y], [x + 2, y]
            else :
                self.laby[x, y], self.case = self.case, self.laby[x + 1, y]
                self.laby[x + 1, y] = 5
                self.pj = (x + 1, y)
                if ret:
                    return False, None, None, None
        if ret:
            return False, None, None, None

    def go_d (self, check=False, ret=False) :
        "Modifie (si possible) le labyrinthe en faisant aller a droite le joueur"
        if check or self.go_d_bool():
            x, y = self.pj
            if self.laby[x, y + 1] == 2 :
```

```
                self.laby[x, y + 2] = 2
                self.laby[x, y], self.case = self.case, 0
                self.laby[x, y + 1] = 5
                self.pj = (x, y + 1)
                i = self.pc.index([x, y + 1])
                self.pc[i] = [x, y + 2]
                if ret :
                    return True, i, [x, y + 1], [x, y + 2]
            else :
                self.laby[x, y], self.case = self.case, self.laby[x, y + 1]
                self.laby[x, y + 1] = 5
                self.pj = (x, y + 1)
                if ret:
                    return False, None, None, None
        if ret:
            return False, None, None, None

    def go_g (self, check=False, ret=False) :
        "Modifie (si possible) le labyrinthe en faisant aller a gauche le joueur
        if check or self.go_g_bool():
            x, y = self.pj
            if self.laby[x, y - 1] == 2 :
                self.laby[x, y - 2] = 2
                self.laby[x, y], self.case = self.case, 0
                self.laby[x, y - 1] = 5
                self.pj = (x, y - 1)
                i = self.pc.index([x, y - 1])
                self.pc[i] = [x, y - 2]
                if ret :
                    return True, i, [x, y - 1], [x, y - 2]
            else :
                self.laby[x, y], self.case = self.case, self.laby[x, y - 1]
                self.laby[x, y - 1] = 5
                self.pj = (x, y - 1)
                if ret:
                    return False, None, None, None
        if ret:
            return False, None, None, None

    def go_h_bool (self) :
```

```
121         "Renvoie True si on peut monter, False sinon"
122         x, y = self.pj
123         if self.laby[x - 1, y] in [0, 4]:           #Case superieure = vide
124             return True
125         elif self.laby[x - 1, y] == 2:              #Case superieur = caisse
126             if self.laby[x - 2, y] in [2, 1, 4]:    #Caisse non bougeable
127                 return False
128             else :                                  #Caisse bougeable
129                 return True
130         else :
131             return False
132
133     def go_b_bool (self) :
134         "Renvoie True si on peut descendre, False sinon"
135         x, y = self.pj
136         if self.laby[x + 1, y] in [0, 4]:           #Case inferieure = vide
137             return True
138         elif self.laby[x + 1, y] == 2:              #Case inferieure = caisse
139             if self.laby[x + 2, y] in [2, 1, 4]:    #Caisse non bougeable
140                 return False
141             else :                                  #Caisse bougeable
142                 return True
143         else :
144             return False
145
146     def go_d_bool (self) :
147         "Renvoie True si on peut aller a droite, False sinon"
148         x, y = self.pj
149         if self.laby[x, y + 1] in [0, 4]:           #Case de droite = vide
150             return True
151         elif self.laby[x, y + 1] == 2:              #Case de droite = caisse
152             if self.laby[x, y + 2] in [2, 1, 4]:    #Caisse non bougeable
153                 return False
154             else :                                  #Caisse bougeable
155                 return True
156         else :
157             return False
158
159     def go_g_bool (self) :
160         "Renvoie True si on peut aller a gauche, False sinon"
```

```
161         x, y = self.pj
162         if self.laby[x, y - 1] in [0, 4]:           #Case de gauche = vide
163             return True
164         elif self.laby[x, y - 1] == 2:              #Case de gauche = caisse
165             if self.laby[x, y - 2] in [2, 1, 4]:    #Caisse non bougeable
166                 return False
167             else :                                  #Caisse bougeable
168                 return True
169         else :
170             return False
171
172     def avancee_bool(self, direction):
173         "Renvoie True si on peut aller dans la direction, False sinon"
174         D = {"Haut" : Jeu.go_h_bool, "Bas" : Jeu.go_b_bool, "Droite" : \
175             Jeu.go_d_bool, "Gauche" : Jeu.go_g_bool}
176         return D[direction](self)
177
178     def avancee(self, direction, check=False, ret=False) :
179         "Fait avancer le personnage dans la direction donnee (si possible)"
180         D = {"Haut" : Jeu.go_h, "Bas" : Jeu.go_b, "Droite" : Jeu.go_d, \
181             "Gauche" : Jeu.go_g}
182         res = D[direction](self, check, ret)
183         if ret:
184             return res
185
186     def copy(self):
187         "Renvoie une copie du jeu"
188         lab = np.array([[self.laby[i, j] for j in range(len(self.laby[0]))] \
189             for i in range(len(self.laby))])
190         pc = [[self.pc[i][j] for j in range(len(self.pc[0]))] \
191             for i in range(len(self.pc))]
192         pj = self.pj
193         pv = [[self.pv[i][j] for j in range(len(self.pv[0]))] \
194             for i in range(len(self.pv))]
195         case = self.case
196         return Jeu(lab, pj, pc, pv, case)
```

Listing 2 – Bibliothèque de niveaux

```
"""
########################################
#                 TIPE                 #
########################################
"""



"""
########################################
#            Importations              #
########################################
"""


from os import *
chdir(path.dirname(__file__))
from _Classe_jeu import *
import numpy as np


"""
########################################
#       Bibliotheque de niveaux        #
########################################
"""


lvl1 = np.array(                                        #Faisable en 238 coups
[[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,2,0,0,1,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,0,0,2,1,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,0,0,2,0,2,0,1,1,1,1,1,1,1,1,1,1,1],
[1,1,1,0,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1],
[1,0,0,0,1,0,1,1,0,1,1,1,1,1,0,0,0,0,1,1],
[1,0,2,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
[1,1,1,1,1,0,1,1,1,0,1,5,1,1,0,0,0,0,1,1],
[1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1],
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]])
```

```
41
42
43 jeu1 = Jeu(lvl1,                                                    #laby
44 (8, 11),                                                           #pj
45 [[2, 5], [3, 7], [4, 5], [4, 7],[7, 2], [7, 5]],                   #pc
46 [[6, 17], [7, 17], [8, 17], [6, 16], [7, 16], [8, 16]]) #pv
47
48 lvl_mini = np.array(                                               #Faisable en 30 coups
49 [[1,1,1,1,1,1],
50 [1,0,0,0,1,1],
51 [1,5,2,2,0,1],
52 [1,1,0,0,0,1],
53 [1,1,1,0,0,1],
54 [1,1,1,1,0,1],
55 [1,1,1,1,1,1]])
56
57 jeu_mini = Jeu(lvl_mini,
58 (2,1),
59 [[2, 2], [2, 3]],
60 [[1, 1], [5, 4]])
61
62
63 lvl_test1 = np.array(                                        #Faisable en 2 coups
64 [[1, 1, 1, 1],
65 [1, 0, 0, 1],
66 [1, 2, 0, 1],
67 [1, 0, 5, 1],
68 [1, 1, 1, 1]])
69
70 jeu_test1 = Jeu(lvl_test1,
71 (3, 2),
72 [[2, 1]],
73 [[1, 1]])
74
75 lvl_test2 = np.array(                                        #Faisable en 16 coups
76 [[1, 1, 1, 1, 1, 1],
77 [1, 0, 0, 0, 0, 1],
78 [1, 0, 1, 5, 0, 1],
79 [1, 0, 2, 2, 0, 1],
80 [1, 0, 0, 2, 0, 1],
```

```
[1, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1]])

jeu_test2 = Jeu(lvl_test2,
(2, 3),
[[3, 2], [3, 3], [4, 3]],
[[3, 3], [4, 3], [4, 2]])


lvl_micro1 = np.array(                                  #Faisable en 33 coups
[[1, 1, 1, 1, 1, 1],
[1 ,0 ,0 ,1 ,1 ,1],
[1, 0, 0, 1, 1, 1],
[1, 2, 5, 0, 0, 1],
[1, 0, 0, 2, 0, 1],
[1, 0, 0, 1, 1, 1],
[1, 1, 1, 1, 1, 1]])

jeu_micro1 = Jeu(lvl_micro1,
(3, 2),
[[3, 1], [4, 3]],
[[1, 2], [3, 1]])

lvl_micro3 = np.array(                                 #Faisable en 41 coups
[[1,1,1,1,1,1,1,1,1],
[1,1,1,0,0,1,1,1,1],
[1,0,0,0,0,0,2,0,1],
[1,0,1,0,0,1,2,0,1],
[1,0,0,0,0,1,5,0,1],
[1,1,1,1,1,1,1,1,1]])

jeu_micro3 = Jeu(lvl_micro3,
(4,6),
[[2,6],[3,6]],
[[4,2],[4,4]])

lvl_micro11 = np.array(                                #Faisalbe en 78 coups
[[1,1,1,1,1,1,1,1,1],
[1,1,1,0,0,0,0,1,1],
[1,1,1,0,1,1,5,1,1],
```

```
[1,1,1,0,1,0,2,0,1],
[1,0,0,0,1,0,2,0,1],
[1,0,0,0,0,0,0,0,1],
[1,0,0,1,1,1,1,1,1],
[1,1,1,1,1,1,1,1,1]])

jeu_micro11 = Jeu(lvl_micro11,
(2,6),
[[3,6],[4,6]],
[[4,2],[4,3]])

lvl_micro16 = np.array(                          #Faisable en 100 coups
[[1,1,1,1,1,1,1,1,1,1],
[1,1,0,0,1,1,1,1,1,1],
[1,1,0,0,0,0,0,1,1,1],
[1,1,0,1,1,0,0,0,1,1],
[1,0,0,0,1,0,5,2,1,1],
[1,0,0,0,1,0,2,2,0,1],
[1,0,0,0,1,0,0,0,0,1],
[1,1,1,1,1,1,1,1,1,1]])

jeu_micro16 = Jeu(lvl_micro16,
(4,6),
[[4,7],[5,6],[5,7]],
[[4,1],[4,3],[6,3]])

lvl_cosmos1 = np.array(                          #Faisable en 49 coups
[[1,1,1,1,1,1,1,1,1],
[1,0,0,1,1,1,0,0,1],
[1,0,2,0,2,0,2,0,1],
[1,0,0,0,5,0,0,0,1],
[1,1,1,0,0,2,1,1,1],
[1,1,1,0,0,0,1,1,1],
[1,1,1,1,1,1,1,1,1]])

jeu_cosmos1 = Jeu(lvl_cosmos1,
(3,4),
[[2,2], [2,4], [2, 6], [4, 5]],
[[2,4], [3,4], [4,4], [5,4]])
```

```
lvl_freebox = np.array(                          #Faisable en 133 coups
[[1,1,1,1,1,1,1,1,1],
[1,1,1,0,0,0,1,1,1],
[1,0,0,2,0,0,2,0,1],
[1,0,1,0,1,1,0,0,1],
[1,0,1,0,1,1,5,1,1],
[1,0,0,2,0,0,2,1,1],
[1,1,1,0,1,2,0,1,1],
[1,1,1,0,0,0,0,1,1],
[1,1,1,2,1,0,0,1,1],
[1,1,1,0,0,2,0,1,1],
[1,1,1,1,1,0,0,1,1],
[1,1,1,1,1,1,1,1,1]])

jeu_freebox = Jeu(lvl_freebox,
(4,6),
[[2,3], [2,6], [5,3], [5,6], [6,5], [8,3], [9,5]],
[[2,4], [2,5], [5,4], [5,5], [7,4], [7,6], [8,6]])
```

Listing 3 - Résolution

```
"""
########################################
#                 TIPE                 #
########################################
"""



"""
########################################
#            Importations              #
########################################
"""


from os import *
chdir(path.dirname(__file__))
import numpy as np
import time
from _Interface_graphique import *
from _Classe_jeu import *
from _Bibliotheque_niveaux import *


"""
########################################
#          Implementation              #
########################################
"""



def victoire(jeu):
    "Renvoie True si le jeu est gagne, False sinon"
    for i in jeu.pc:
        if not i in jeu.pv:    #Teste si chaque position de victoire est occupee
```

```
            return False
    return True

def jouer(jeu) :
    "Permet de jouer a un niveau"
    cop = jeu.copy()
    #cop = jeu
    L = []
    D = {"z" : "Haut", "s" : "Bas", "q" : "Gauche", "d" : "Droite"}
    while 1:
        if victoire(cop) :
            print("Probleme resolu !")
            print("Resolu en {} coups".format(len(L)))
            return L
        else :
            print(cop.laby)
            direction = input("Entrez la direction : ")
            if direction in D.keys():
                cop.avancee(D[direction])
                L.append(D[direction])
            elif direction == "Quit" :
                print("Abandon")
                break
        print('Deplacements : {}'.format(len(L)))

def complementaire(direc):
    "Renvoie la direction opposee a direc"
    D = {"Haut" : "Bas", "Bas" : "Haut", "Droite" : "Gauche", \
        "Gauche" : "Droite"}
    return D[direc]

def solveur_general(jeu, fonction_aux, pre_aux, nb_coups_restants, args={}):
    "Solveur general sur lequel sont utilises les differents modes de resolution
    if args["affichage"]:
        print(args["coup"], " ", nb_coups_restants)
        print(jeu)

    if victoire(jeu):
        return True

```

```
    elif nb_coups_restants == 0:
        return False

    else:
        if pre_aux(jeu, args, nb_coups_restants) == False:
  #Actions preliminaires, renvoyant eventuellement False si la partie est perdue
            return False
        nb_coups_restants -= 1

        for d in ["Haut", "Bas", "Droite", "Gauche"]:
                #On teste chaque direction dans cet ordre
            if jeu.avancee_bool(d):   #On verifie qu'on puisse avancer
                args["coup"] = d
                if fonction_aux(jeu, nb_coups_restants, args):
                        #Fonction auxiliaire d'exploration
                    args["sol"].append(d)
                    return True

        return False

def aux_solveur_naif(jeu, nb_coups_restant, args):
    "Fonction auxiliaire pour le solveur naif"
    coup, pos_caisses = args["coup"], args["pos_caisses"]
    affichage = args["affichage"]
    pos_caisses = [[args["pos_caisses"][i][0], args["pos_caisses"][i][1]]  \
        for i in range(len(jeu.pc))]
    jeu.avancee(coup)
    res = solveur_general(jeu, aux_solveur_naif, pre_verif_naif, \
        nb_coups_restant, args)
    jeu.avancee(complementaire(coup))
    reset_caisses(jeu, pos_caisses)
    args["pos_caisses"] = [[jeu.pc[i][0], jeu.pc[i][1]] for i in \
        range(len(jeu.pc))]
    if affichage:
        print(coup)
        print(jeu.pc)
    return res

def pre_verif_naif(jeu, args, nb_coups_restant):
    "Actions preliminaires pour le solveur naif"
```

```
121     args["pos_caisses"] = [[jeu.pc[i][0], jeu.pc[i][1]] for i in \
122         range(len(jeu.pc))] #Copie de la position des caisses pour le reset
123 
124 def solveur_naif(jeu, nb_coups_restants, affichage=False):
125     "Solveur naif v1"
126     args = {"affichage" : affichage, "sol" : [], "coup" : "Haut"}
127     solveur_general(jeu, aux_solveur_naif, pre_verif_naif, \
128         nb_coups_restants, args)
129     args["sol"].reverse()
130     return args["sol"]
131 
132 def reset_caisses(jeu, pos_caisses):
133     "Retablit les positions des caisses du jeu aux positions pos_caisses"
134     set_caisses = set((pos_caisses[i][0], pos_caisses[i][1]) for i in \
135         range(len(pos_caisses)))
136         #Transformation en ensemble pour utiliser l'intersection
137     newcaisses = set((jeu.pc[i][0], jeu.pc[i][1]) for i in range(len(jeu.pc)))
138     caisses_a_enlever = newcaisses - set_caisses
139     caisses_a_rajouter = set_caisses - newcaisses
140     for pos in caisses_a_enlever:
141         jeu.laby[pos] = 0
142     for pos in caisses_a_rajouter:
143         jeu.laby[pos] = 2
144     jeu.pc = [[pos_caisses[i][0], pos_caisses[i][1]] for i in \
145         range(len(pos_caisses))]
146 
147 """--------------------------------------------------------------------------"""
148 
149 def aux_solveur_un_tout_petit_peu_moins_naif(jeu, nb_coups_restants, args):
150     "Fonction auxiliaire pour le solveur un tout petit peu moins naif"
151     coup, affichage = args["coup"], args["affichage"]
152     boole, i, old, new = jeu.avancee(coup, True, True)
153     res = solveur_general(jeu, aux_solveur_un_tout_petit_peu_moins_naif, \
154         pre_verif_un_tout_petit_peu_moins_naif, nb_coups_restants, args)
155     jeu.avancee(complementaire(coup), check = True)
156     reset_caisses2(jeu, boole, i, old, new)
157     if affichage:
158         print(coup)
159         print(jeu.pc)
160     return res
```

```
161
162 def pre_verif_un_tout_petit_peu_moins_naif(jeu, args, nb_coups_restant):
163     "Actions preliminaires pour le solveur un tout petit peu moins naif"
164     pass  #Pas d'actions preliminaires pour ce solveur
165
166
167 def solveur_un_tout_petit_peu_moins_naif(jeu, nb_coups_restants, affichage=False
168     "Solveur naif v2"
169     args = {"affichage" : affichage, "sol" : [], "coup" : "Haut"}
170     solveur_general(jeu, aux_solveur_un_tout_petit_peu_moins_naif, \
171         pre_verif_un_tout_petit_peu_moins_naif, nb_coups_restants, args)
172     args["sol"].reverse()
173     return args["sol"]
174
175 def reset_caisses2(jeu, boole, i, old, new):
176     "Retablit les positions des caisses du jeu"
177     if boole:
178         jeu.pc[i] = old
179         [a, b] = old
180         jeu.laby[a, b] = 2
181         [a, b] = new
182         jeu.laby[a, b] = 0
183
184
185
186
187 """
188 ##########################################
189 #           Tables de hachage            #
190 ##########################################
191 """
192
193
194
195
196 def table_hachage(n=257) :
197     "Cree une table de False"
198     return np.array([False for _ in range(n)])
199
200 def hachage1(jeu, n=257) :
```

```
201     "Fonction de hachage"
202     liste = []
203     for x in jeu.pc :
204         liste += x
205     a, b = jeu.pj
206     liste += [a, b]
207     cle = 0
208     i = 1
209     for x in liste :
210         cle += (x * i) ** 5
211         i += 1
212     return cle % n
213 
214 def hachage2(jeu, n=257):
215     "Autre fonction de hachage"
216     liste = []
217     for x in jeu.pc :
218         liste += x
219     a, b = jeu.pj
220     liste += [a, b]
221     cle = 0
222     i = 1
223     for x in liste :
224         cle += i ** x
225         i += 1
226     return cle % n
227 
228 def aux_solveur_hachage(jeu, nb_coups_restants, args):
229     "Fonction auxiliaire pour le solveur utilisant les tables de hachage"
230     coup, affichage, tab_hash, hachage, n = args["coup"], args["affichage"], \
231         args["tab_hash"], args["hachage"], args["n"]
232     boole, i, old, new = jeu.avancee(coup, True, True)
233     cle = hachage(jeu, n)
234     res = False
235     if not tab_hash[cle]:  #Position pas encore visitee
236         tab_hash[cle] = True
237         res = solveur_general(jeu, aux_solveur_hachage, pre_verif_hachage, \
238             nb_coups_restants, args)
239     jeu.avancee(complementaire(coup), check=True)
240     reset_caisses2(jeu, boole, i, old, new)
```

```
241     if affichage:
242         print(coup)
243         print(jeu.pc)
244         print("Hash : ", cle)
245     return res
246 
247 def pre_verif_hachage(jeu, args, nb_coups_restant):
248     "Actions preliminaires pour le solveur utilisant les tables de hachage"
249     pass #Pas d'actions preliminaires pour ce solveur
250 
251 def solveur_hachage(jeu, nb_coups_restants, tab_hash, hachage, affichage=False):
252     "Solveur hachage v1"
253     n = len(tab_hash)
254     args = {"tab_hash" : tab_hash, "hachage" : hachage, "n" : n, \
255         "coup" : "Haut", "affichage" : affichage, "sol" : []}
256     solveur_general(jeu, aux_solveur_hachage, pre_verif_hachage, \
257         nb_coups_restants, args)
258     args["sol"].reverse()
259     return args["sol"]
260 
261 """-----------------------------------------------------------------------------"""
262 
263 def table_hachage_injectif(jeu) :
264     "Cree une table de False pour le hachage injectif"
265     n = max(len(jeu.laby), len(jeu.laby[0]))
266     p = len(str(n))
267     taille = n ** (p * 2 * (len(jeu.pc) + 1))
268         #Nombre d'etats possibles : n ** ((nb_caisse + pj) * p)
269     return np.array([False for _ in range(int(taille))]), \
270         np.array([0 for _ in range(int(taille))])
271 
272 def to_str_len_p(x,p):
273     "Renvoie x converti en str sur p caracteres"
274     x = str(x)
275     xlen = len(x)
276     if xlen < p:
277         x = "0" * (p - xlen) + x
278          #Ajout d'eventuels 0 pour obtenir une chaine de la bonne taille
279     return x
280 
```

```
281 def hachage_injectif(jeu) :
282     "Fonction de hachage injective"
283     n, p = max(len(jeu.laby), len(jeu.laby[0])), len(jeu.pc)
284     cle = 0
285     for i in range(p):
286         cle += jeu.pc[i][0] * (n ** (2 * i)) + jeu.pc[i][1] * \
287             (n **  (2 * i + 1)) #Ecriture en "base n"
288     x, y = jeu.pj
289     cle += x * (n ** (2 * p)) + y * (n **  (2 * p + 1))
290     return cle
291 
292 def aux_solveur_hachage_injectif(jeu, nb_coups_restants, args):
293     "Fonction auxiliaire pour le solveur utilisant la fonction de \
294         hachage injective"
295     tab_coups, tab_cle, coup, affichage = args["tab_coups"], args["tab_cle"], \
296         args["coup"], args["affichage"]
297     boole, i, old, new = jeu.avancee(coup, True, True)
298     cle = hachage_injectif(jeu)
299     res = False
300     if not tab_cle[cle] or (nb_coups_restants > tab_coups[cle]):
301         #Position non deja visitee ou visitee avec un nombre de coups plus petit
302         tab_cle[cle] = True
303         tab_coups[cle] = nb_coups_restants
304         res = solveur_general(jeu, aux_solveur_hachage_injectif, \
305             pre_verif_hachage_injectif, nb_coups_restants, args)
306     jeu.avancee(complementaire(coup), check = True)
307     reset_caisses2(jeu, boole, i, old, new)
308     if affichage:
309         print(coup)
310         print(jeu.pc)
311         print("Hash : ", cle)
312     return res
313 
314 def pre_verif_hachage_injectif(jeu, args, nb_coups_restant):
315     "Actions preliminaires pour le solveur utilisant la fonction \
316     de hachage injective"
317     pass  #Pas d'actions preliminaires pour ce solveur
318 
319 def solveur_hachage_injectif(jeu, nb_coups_restants, affichage=False):
320     "Solveur hachage v2"
```

```
321     tab_cle, tab_coups = table_hachage_injectif(jeu)
322     args = {"sol" : [], "coup" : "Haut", "tab_coups" : tab_coups, \
323         "tab_cle" : tab_cle, "affichage" : affichage}
324     solveur_general(jeu, aux_solveur_hachage_injectif, \
325         pre_verif_hachage_injectif, nb_coups_restants, args)
326     args["sol"].reverse()
327     return args["sol"]
328
329
330
331
332 """
333 ##########################################
334 #          Version dictionnaire          #
335 ##########################################
336 """
337
338
339
340
341 def hachage_tuple(jeu):
342     "Fonction de hachage avec les tuples pour le solveur version dictionnaire"
343     L = sorted(jeu.pc)
344     #L = jeu.pc
345     pc_tuple = tuple(tuple(L[i]) for i in range(len(jeu.pc)))
346         #Transformation en tuple
347     key = jeu.pj, pc_tuple
348     return key
349
350 def hachage_str(jeu):
351     "Fonction de hachage avec les chaines de caracteres pour le solveur \
352         version dictionnaire"
353     L = sorted(jeu.pc)
354     #L = jeu.pc
355     x, y = jeu.pj
356     LL = [str(x), str(y)]
357     LL += [str(L[i][j]) for i in range(len(L)) for j in range(2)]
358     key = " ".join(LL)    #Transformation en string
359     return key
360
```

```
361 def lprem():
362     "Liste des nombres premiers inferieurs a 1000"
363     return [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,\
364             89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,\
365             179,181,191,193,197,199,211,223,227,229,233,239,241,251,257,263,\
366             269,271,277,281,283,293,307,311,313,317,331,337,347,349,353,359,\
367             367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,457,\
368             461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,569,\
369             571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,\
370             661,673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,\
371             773,787,797,809,811,821,823,827,829,839,853,857,859,863,877,881,\
372             883,887,907,911,919,929,937,941,947,953,967,971,977,983,991,997]
373
374 nbprem = lprem()
375
376 def hachage_nbprem(jeu):
377     x, y = jeu.pj
378     L = [[x, y]] + sorted(jeu.pc)
379     key = 1
380     for k in range(len(L)):
381         key *= nbprem[2 * k] ** L[k][0]
382         key *= nbprem[2 * k + 1] ** L[k][1]
383     return key
384
385 def aux_solveur_dico(jeu, nb_coups_restant, args):
386     "Fonction auxiliaire pour le solveur utilisant les dictionnaires"
387     dic, coup, affichage = args["dic"], args["coup"], args["affichage"]
388     boole, i, old, new = jeu.avancee(coup, True, True)    #Mouvement
389     cle = hachage_str(jeu)
390     res = False
391     try:              #Position deja vue
392         nb_coup = dic[cle]
393         if nb_coups_restant > nb_coup:
394             dic[cle] = nb_coups_restant
395             res = solveur_general(jeu, aux_solveur_dico, pre_verif_dico, \
396                 nb_coups_restant, args)
397     #On ne reessaie que si il nous reste plus de coups que la fois precedente
398     except KeyError:
399         #Si la position n'a pas ete vue (Exception KeyError declenchee)
400         dic[cle] = nb_coups_restant
```

```
401         res = solveur_general(jeu, aux_solveur_dico, pre_verif_dico, \
402             nb_coups_restant, args)
403     jeu.avancee(complementaire(coup), check=True)   #Annulation du coup
404     reset_caisses2(jeu, boole, i, old, new)
405     if affichage:       #Pour le debug
406         print(coup)
407         print(jeu.pc)
408         print("Hash : ", cle)
409     return res
410
411 def pre_verif_dico(jeu, args, nb_coups_restant):
412     "Actions preliminaires pour le solveur utilisant les dictionnaires"
413     if args["dead"] and nb_coups_restant % 7 == 1:
414         #On verifie les deadlocks tous les 7 coups
415             stop = check_deadlock(jeu)
416             if not stop:
417                 return False
418
419 def solveur_dico(jeu, nb_coups_restants, dic=None, dead=True, aff=False):
420     "Solveur dico v1"
421     if dic is None:
422         dic = {}
423     args = {"sol" : [], "coup" : "Haut", "affichage" : aff, "dic" : dic, \
424         "dead" : dead}
425     if dead :
426         reperage(jeu)  #On repere les case menant a coup sur a une defaite
427     solveur_general(jeu, aux_solveur_dico, pre_verif_dico, \
428         nb_coups_restants, args)
429     args["sol"].reverse()
430     if dead:
431         remove_deads(jeu)
432     return args["sol"]
433
434 def test_sol(jeu, sol):
435     "Verefie si sol est bien solution du jeu"
436     cop = jeu.copy()
437     for x in sol:
438         cop.avancee(x)
439     if victoire(cop) :
440         print(cop.laby)
```

```
441         return "Probleme resolu !"
442     else :
443         print(cop.laby)
444         return "Ce n'est pas une solution"
445
446
447
448
449 """
450 ##########################################
451 #                Deadlocks                 #
452 ##########################################
453 """
454
455
456
457 def est_deadlock1(jeu, x, y):
458     "Verifie si la case x, y est un coin dans le labyrinthe (case supposee \
459         non position de victoire)"
460     L = [((1, 0), (0, 1)), ((1, 0), (0, -1)), ((-1, 0), (0, 1)), \
461         ((-1, 0), (0, -1))]
462         # Coin bas - droite / bas - gauche / haut - droite / haut - gauche
463     for ((a, b), (c, d)) in L:
464         if jeu.laby[x + a, y + b] == 1 == jeu.laby[x + c, y + d]:
465             return True
466     return False
467
468 def deadlock2(jeu):
469     "Deadlocks des carres"
470     if len(jeu.pc) < 4:
471         return True
472     else:
473         for [x, y] in jeu.pc:
474             if jeu.laby[x + 1, y] == jeu.laby[x, y + 1] == \
475                 jeu.laby[x + 1, y + 1] == 2 or jeu.laby[x + 1, y] == \
476                 jeu.laby[x, y - 1] == jeu.laby[x + 1, y - 1] == 2 or \
477                 jeu.laby[x - 1, y] == jeu.laby[x, y + 1] == \
478                 jeu.laby[x - 1, y + 1] == 2 or jeu.laby[x - 1, y] == \
479                 jeu.laby[x, y - 1] == jeu.laby[x - 1, y - 1] == 2:
480   #La case est le coin haut - gauche / haut - droite / bas -
```

```
    #gauche / bas - droite du carre
                    if not [x, y] in jeu.pv:
                        return False
        return True

def est_deadlock3(jeu, x, y) :
    "Teste si une position est contre un mur et qu'on ne peut pas en retirer \
    une caisse eventuelle ne peut pas partir (case supposee non position \
    de victoire)"
    #Chaque boucle fait le meme teste, en changeant le cote longe
    if jeu.laby[x, y + 1] == 1:     #Mur vertical droit
        i = x + 1                   #On avance vers le bas
        test_pv_i = False           #True si on a rencontre une p_v, False sinon
        rencontre_mur_i = False     #True si on est arrive en face d'un mur
        while not test_pv_i and not rencontre_mur_i and jeu.laby[i, y + 1] == 1:
            #Verifie qu'on est toujours contre un mur et qu'on a pas encore
            #rencontre de p_v ou de mur
            if [i, y] in jeu.pv:
                test_pv_i = True
            elif jeu.laby[i, y] in [1, 4]:
                rencontre_mur_i = True
            i += 1
        if rencontre_mur_i :
            #Si on a rencontre un mur mais pas de p_v, on essaie en descendant
            j = x - 1
            test_pv_j = False
            rencontre_mur_j = False
            while not test_pv_j and not rencontre_mur_j and \
                jeu.laby[j, y + 1] == 1:
                if [j, y] in jeu.pv:
                    test_pv_j = True
                elif jeu.laby[j, y] in [1, 4]:
                    rencontre_mur_j = True
                j -= 1
            if rencontre_mur_j:
                #Si on a rencontre un mur dans les deux sens, c'est perdu
                return False
    if jeu.laby[x, y - 1] == 1:     #Mur vertical gauche
        i = x + 1
        test_pv_i = False
```

```
        rencontre_mur_i = False
        while not test_pv_i and not rencontre_mur_i and jeu.laby[i, y - 1] == 1:
            if [i, y] in jeu.pv:
                test_pv_i = True
            elif jeu.laby[i, y] in [1, 4]:
                rencontre_mur_i = True
            i += 1
        if rencontre_mur_i:
            j = x - 1
            test_pv_j = False
            rencontre_mur_j = False
            while not test_pv_j and not rencontre_mur_j and \
                jeu.laby[j, y - 1] == 1:
                if [j, y] in jeu.pv:
                    test_pv_j = True
                elif jeu.laby[j, y] in [1, 4]:
                    rencontre_mur_j = True
                j -= 1
            if rencontre_mur_j:
                return False
    if jeu.laby[x - 1, y] == 1: #Mur horizontal haut
        i = y + 1               #On avance vers la droite
        test_pv_i = False
        rencontre_mur_i = False
        while not test_pv_i and not rencontre_mur_i and jeu.laby[x - 1, i] == 1:
            if [x, i] in jeu.pv:
                test_pv_i = True
            elif jeu.laby[x, i] in [1, 4]:
                rencontre_mur_i = True
            i += 1
        if rencontre_mur_i:     #On avance vers la gauche
            j = y - 1
            test_pv_j = False
            rencontre_mur_j = False
            while not test_pv_j and not rencontre_mur_j and \
                jeu.laby[x - 1, j] == 1:
                if [x, j] in jeu.pv:
                    test_pv_j = True
                elif jeu.laby[x, j] in [1, 4]:
                    rencontre_mur_j = True
```

```
                j -= 1
            if rencontre_mur_j :
                return False
    if jeu.laby[x + 1, y] == 1: #Mur horizontal bas
        i = y + 1
        test_pv_i = False
        rencontre_mur_i = False
        while not test_pv_i and not rencontre_mur_i and jeu.laby[x + 1, i] == 1:
            if [x, i] in jeu.pv:
                test_pv_i = True
            elif jeu.laby[x, i] in [1, 4]:
                rencontre_mur_i = True
            i += 1
        if rencontre_mur_i :
            j = y - 1
            test_pv_j = False
            rencontre_mur_j = False
            while not test_pv_j and not rencontre_mur_j and \
                jeu.laby[x + 1, j] == 1:
                if [x, j] in jeu.pv:
                    test_pv_j = True
                elif jeu.laby[x, j] in [1, 4]:
                    rencontre_mur_j = True
                j -= 1
            if rencontre_mur_j:
                return False
    return True

def deadlock4(jeu) :
    "Test si on a 2 caisses l'une contre l'autre contre un mur"
    for [x,y] in jeu.pc :
        if not([x, y] in jeu.pv) :
            if (jeu.laby[x, y + 1] == 2 and (jeu.laby[x - 1, y] == \
                jeu.laby[x - 1, y + 1] == 1 or jeu.laby[x + 1, y] == \
                jeu.laby[x + 1, y + 1] == 1)) or (jeu.laby[x, y - 1] == 2 and \
                (jeu.laby[x - 1, y] == jeu.laby[x - 1, y - 1] == 1  or \
                jeu.laby[x + 1, y] == jeu.laby[x + 1, y - 1] == 1)) or\
                (jeu.laby[x - 1, y] == 2 and (jeu.laby[x, y + 1] == \
                jeu.laby[x - 1, y + 1] == 1 or jeu.laby[x, y - 1] == \
                jeu.laby[x - 1, y - 1] == 1)) or (jeu.laby[x + 1, y] == 2 and \
```

```
                (jeu.laby[x, y + 1] == jeu.laby[x + 1, y + 1] == 1 or \
                jeu.laby[x, y - 1] == jeu.laby[x + 1, y - 1] == 1)):
                    #Caisse a droite / a gauche / en haut / en bas
                return False
    return True

def deadlock5(jeu):
    "Deadlock des trois caisses dans un angle"
    if len(jeu.pc) < 3:
        return True
    for [x, y] in jeu.pc:
        if (jeu.laby[x + 1, y - 1] == 1 and (jeu.laby[x, y - 1] == 2 == \
            jeu.laby[x + 1, y])) or (jeu.laby[x + 1, y + 1] == 1 and \
            (jeu.laby[x, y + 1] == 2 == jeu.laby[x + 1, y])) \
            or (jeu.laby[x - 1, y - 1] == 1 and (jeu.laby[x, y - 1] == 2 == \
            jeu.laby[x - 1, y])) or (jeu.laby[x - 1, y + 1] == 1 and \
            (jeu.laby[x, y + 1] == 2 == jeu.laby[x - 1, y])):
            #Angle bas - gauche / bas - droite / haut - gauche / haut - droite
                return False
    return True

def remove_deads(jeu):
    "Enleve les 4 du jeu"
    nx, ny = len(jeu.laby), len(jeu.laby[0])
    for x in range(1, nx - 1):
        for y in range(1, ny - 1):
            if jeu.laby[x, y] == 4:
                jeu.laby[x, y] = 0
    jeu.case = 0

def check_deadlock(jeu):
    "Verifie si le jeu est bloque pour l'un des deadlocks implemente"
    L = [deadlock2, deadlock4, deadlock5]  #Fonctions de deadlocks a verifier
    res = True
    for f in L:
        res = res and f(jeu)
    return res

def reperage(jeu) :
    "Mise en place des 4 (deadlocks previsibles)"
```

```
641     for x in range(len(jeu.laby)):
642         for y in range(len(jeu.laby[0])):
643             if jeu.laby[x, y] == 0 and not [x, y] in jeu.pv:
644                 if est_deadlock1(jeu, x, y) or not est_deadlock3(jeu, x, y):
645                     jeu.laby[x, y] = 4
646     x, y = jeu.pj
647     if not [x, y] in jeu.pv and (est_deadlock1(jeu, x, y) or not \
648         est_deadlock3(jeu, x, y)):
649         jeu.case = 4
650
```