

T.I.P.E.

Résolution numérique d'un problème de logique : le Sokoban

MULOT Lendy
4909

Sommaire

Introduction : Présentation du jeu

- 1) Modélisation informatique
- 2) Idée générale de résolution
- 3) Résolution naïve
- 4) Tables de hachage et dictionnaires
- 5) Optimisations

Conclusion

Introduction

Le Sokoban : présentation

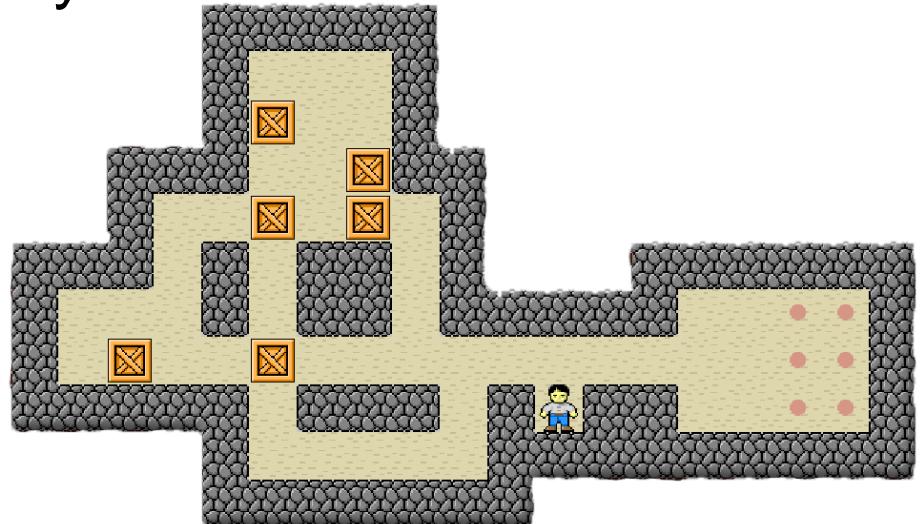
Genre : jeu de puzzle

Signification : « Garde d'entrepôt »

Concepteur : Hiroyuki Imabayashi

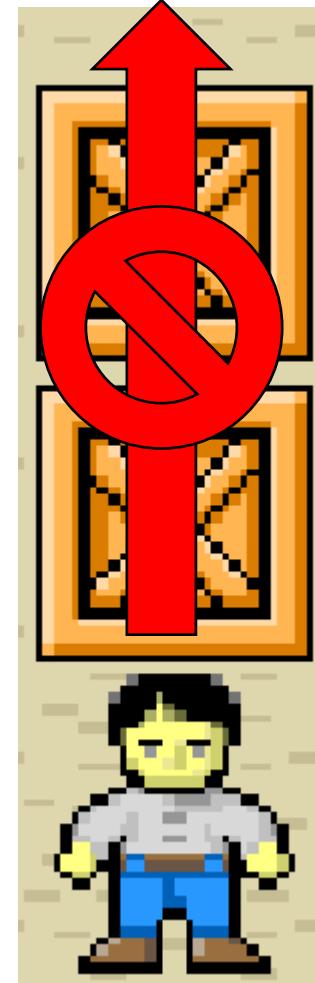
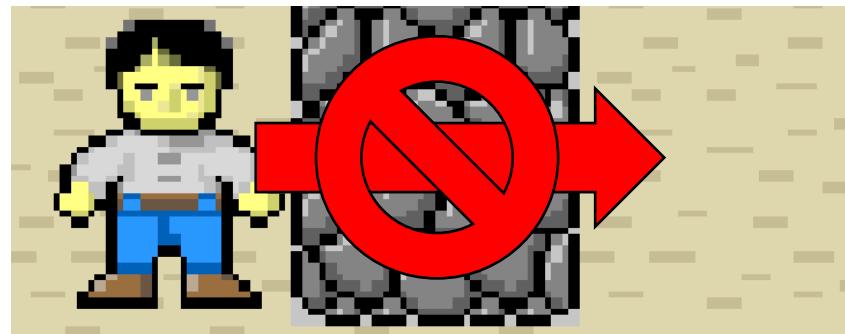
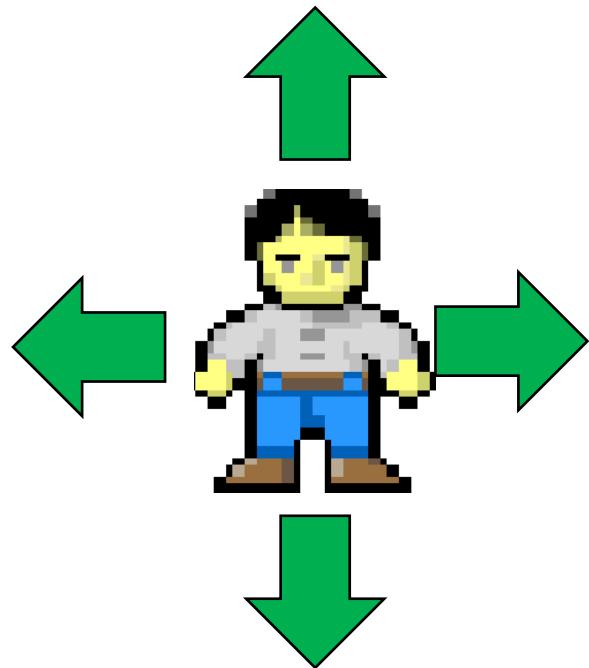
Sortie : 1982

But : amener des caisses
dans des positions
prédéterminées



Introduction

Règles du jeu



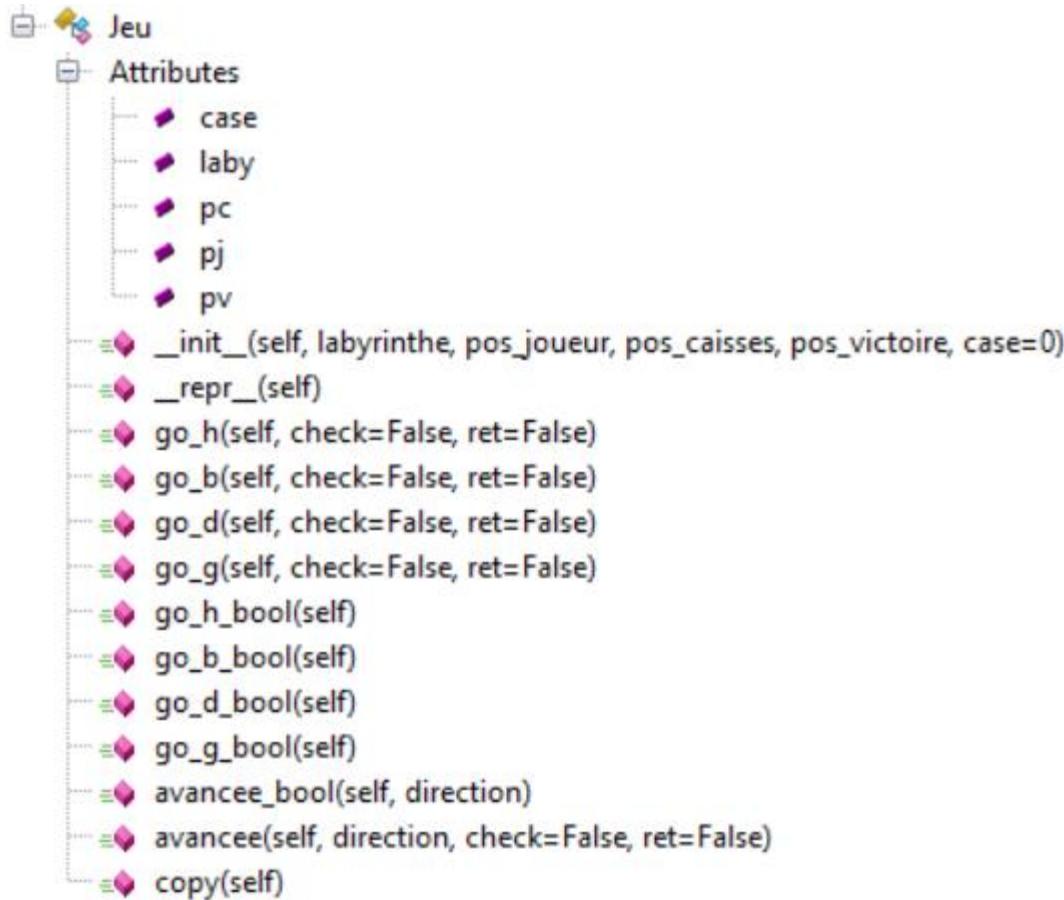
Introduction

Problématique

**Comment modéliser et résoudre
informatiquement les niveaux du Sokoban ?**

Modélisation informatique

Classe Jeu



Modélisation informatique

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

```
jeu_mini = Jeu(lvl_mini,  
(2,1),  
[[2, 2], [2, 3]],  
[[1, 1], [5, 4]])
```

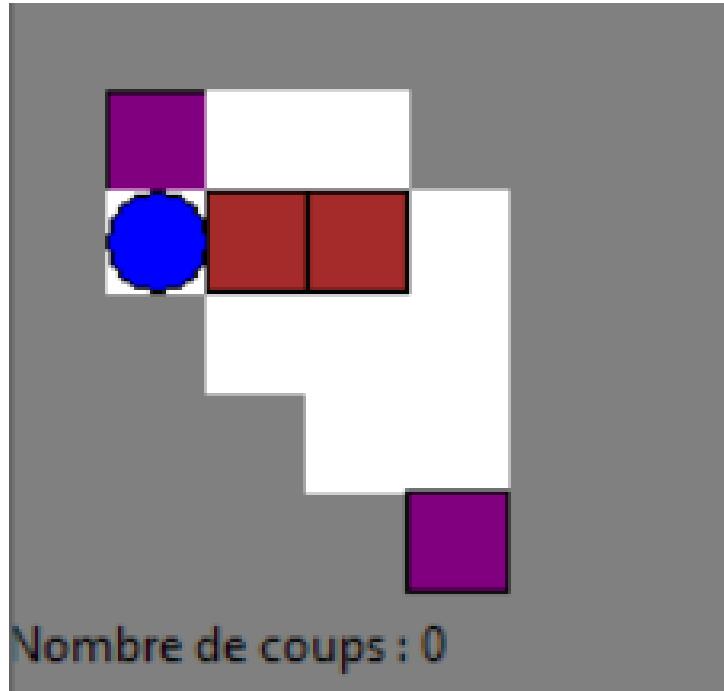
Labyrinthe

Position du joueur (5)

Positions des caisses (2)

Positions de victoire

Modélisation informatique



Modélisation informatique

```
def go_h_bool (self) :  
    "Renvoie True si on peut monter, False sinon"  
    x, y = self.pj  
    if self.laby[x - 1, y] in [0, 4]:  
        return True  
    elif self.laby[x - 1, y] == 2:  
        if self.laby[x - 2, y] in [2, 1, 4]:  
            return False  
        else :  
            return True  
    else :  
        return False
```

#Case supérieure = vide
#Case supérieur = caisse
#Caisse non bougeable
#Caisse bougeable

Modélisation informatique

ALGORITHME : Déplacement vers le haut

Entrée : Le jeu *self* ; un booléen *check* (*Faux* par défaut) indiquant que le déplacement est possible et a été vérifié ; un booléen *ret* (*Faux* par défaut) indiquant que l'on souhaite récupérer les données en sortie

Sortie : Rien si *ret* est *Faux*, sinon : (*bool*, *index*, *old*, *new*) où *bool* indique si une caisse a été déplacée, *index* est son indice dans la liste des caisses, *old* et *new* sont respectivement l'ancienne et nouvelle position de la caisse

```
DEBUT
SI check OU déplacement possible ALORS
    x, y ← position du joueur
    SI une caisse est au-dessus du joueur ALORS
        Déplacer la caisse dans le labyrinthe
        Déplacer le joueur dans le labyrinthe
        Mettre à jour l'attribut pj
        i ← l'indice de la caisse dans la liste des caisses
        Mettre à jour la liste des caisses
        SI ret ALORS
            |   RENVOYER (Vrai, i, [x - 1, y], [x - 2, y])
        FIN_SI
    SINON
        Déplacer le joueur dans le labyrinthe
        Mettre à jour l'attribut pj
        SI ret ALORS
            |   RENVOYER (False, None, None, None)
        FIN_SI
    FIN_SI
SINON SI ret ALORS
    |   RENVOYER (False, None, None, None)
FIN_SI
FIN
```

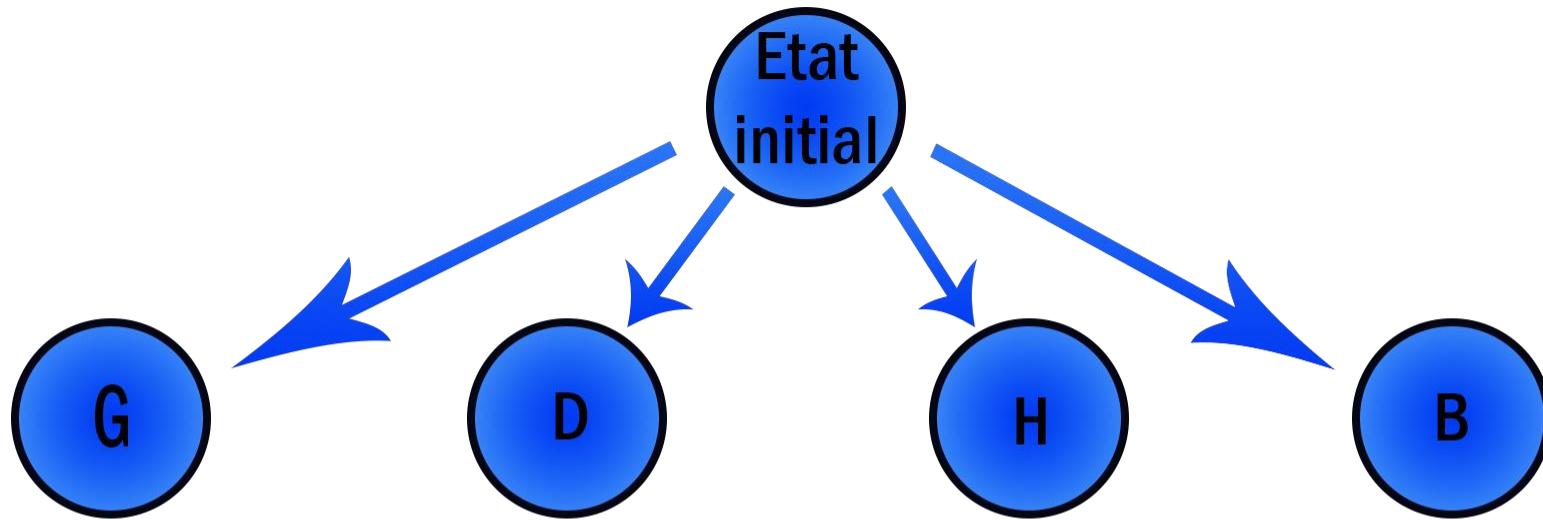
Modélisation informatique

```
def victoire(jeu):
    "Renvoie True si le jeu est gagne, False sinon"
    for i in jeu.pc:
        if not i in jeu.pv:
            return False
    return True
```

Idée générale de résolution

- Algorithme récursif à profondeur limitée
- Cas de base : victoire ou profondeur maximum atteinte
- Appels récursifs sur chaque direction possible

Idée générale de résolution



Idée générale de résolution

```
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
```

Cas de base

```
else:
    if pre_aux(jeu, args, nb_coups_restants) == False:
        return False
    nb_coups_restants -= 1
```

Actions préliminaires

```
    for d in ["Haut", "Bas", "Droite", "Gauche"]:
        if jeu.avancee_bool(d):
            args["coup"] = d
            if fonction_aux(jeu, nb_coups_restants, args):
                args["sol"].append(d)
                return True
```

Mouvement

```
return False
```

Actions auxiliaires,
appels récursifs et
résultat



Résolution naïve

```
for d in ["Haut", "Bas", "Droite", "Gauche"]:  
    if jeu.avancee_bool(d):  
        args["coup"] = d  
        if fonction_aux(jeu, nb_coups_restants, args):  
            args["sol"].append(d)  
            return True  
  
def aux_solveur_naif(jeu, nb_coups_restant, args):  
    "Fonction auxiliaire pour le solveur naïf"  
    coup, pos_caisse, affichage = args["coup"], args["pos_caisse"], args["affichage"]  
    pos_caisse = [[args["pos_caisse"]][i][0], args["pos_caisse"]][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_caisse(jeu, pos_caisse)  
    args["pos_caisse"] = [[jeu.pc[i][0], jeu.pc[i][1]] for i in range(len(jeu.pc))]  
    if affichage:  
        print(coup)  
        print(jeu.pc)  
    return res
```

Déplacement
Appel récursif
Annulation

Résolution naïve

- Limité à une vingtaine de coups → Problème
- Comment éviter les cas inutiles ?

Tables de hachage et dictionnaires

Idée : marquer les positions déjà vues pour ne pas repasser par les mêmes

→ Tables de hachage

Tables de hachage et dictionnaires

- Tentative naïve de fonction de hachage :

$$h_1(x_1, \dots, x_n) = \sum_{i=1}^n x_i \cdot i \mod p$$

Tables de hachage et dictionnaires

```
def aux_solveur_hachage(jeu, nb_coups_restants, args):
    "Fonction auxiliaire pour le solveur utilisant les tables de hachage"
    coup, affichage, tab_hash, hachage, n = args["coup"], args["affichage"], args["tab_hash"], args["hachage"], args["n"]
    boole, i, old, new = jeu.avancee(coup, True, True)
    cle = hachage(jeu, n)
    res = False
    if not tab_hash[cle]:
        tab_hash[cle] = True
        res = solveur_general(jeu, aux_solveur_hachage, pre_verif_hachage, nb_coups_restants, args)
        jeu.avancee(complementaire(coup), check=True)
        reset_caisses2(jeu, boole, i, old, new)
    if affichage:
        print(coup)
        print(jeu.pc)
        print("Hash : ", cle)
    return res
```

Vérification

Appel récursif

Annulation

Tables de hachage et dictionnaires

- Des positions totalement différentes renvoient la même clé
- Les tableaux sont trop grands
→ empêche une bonne résolution

Tables de hachage et dictionnaires

- On utilise la structure de dictionnaire

Tables de hachage et dictionnaires

Hachage en chaîne de caractère :

$$h_2(x_1, \dots, x_n) = "x_1 \dots x_n"$$

```
def hachage_str(jeu):
    "Fonction de hachage avec les chaines de caracteres pour le solveur version dictionnaire"
    L = sorted(jeu.pc)
    x, y = jeu.pj
    LL = [str(x), str(y)]
    LL += [str(L[i][j]) for i in range(len(L)) for j in range(2)]
    key = " ".join(LL)
    return key
```

Tables de hachage et dictionnaires

Hachage avec des nombres premiers :

$$h_3(x_1, \dots, x_n) = \prod_{i=1}^n p_i^{x_i}$$

```
def hachage_nbprem(jeu):
    x, y = jeu.pj
    L = [[x, y]] + sorted(jeu.pc)
    key = 1
    for k in range(len(L)):
        key *= nbprem[2 * k] ** L[k][0]
        key *= nbprem[2 * k + 1] ** L[k][1]
    return key
```

Tables de hachage et dictionnaires

ALGORITHME : Fonction auxiliaire pour le solveur utilisant les dictionnaires

Entrée : Le jeu *jeu* ; le nombre de coups restants *nb_coups_restant* ; un dictionnaire *args* contenant les autres arguments utiles

Sortie : Un booléen indiquant si on peut résoudre le niveau en partant de l'état donné en moins de *nb_coups_restant* coups

DEBUT

```
dic, coup ← args["dic"], args["coup"]
boole, i, old, new ← jeu.avancee(Coup, Vrai, Vrai)
cle ← hachage(jeu)
```

SI la position a déjà été visitée **ALORS**

```
    nb_coups ← dic[cle]
    res ← solveur_general(jeu, aux_solveur_dico, pre_verif_dico, nb_coups_restant, args)
```

SINON

```
    dic[cle] ← nb_coups_restant
    res ← solveur_general(jeu, aux_solveur_dico, pre_verif_dico, nb_coups_restant, args)
```

FIN_SI

Annuler le coup

RENOVYER *res*

FIN

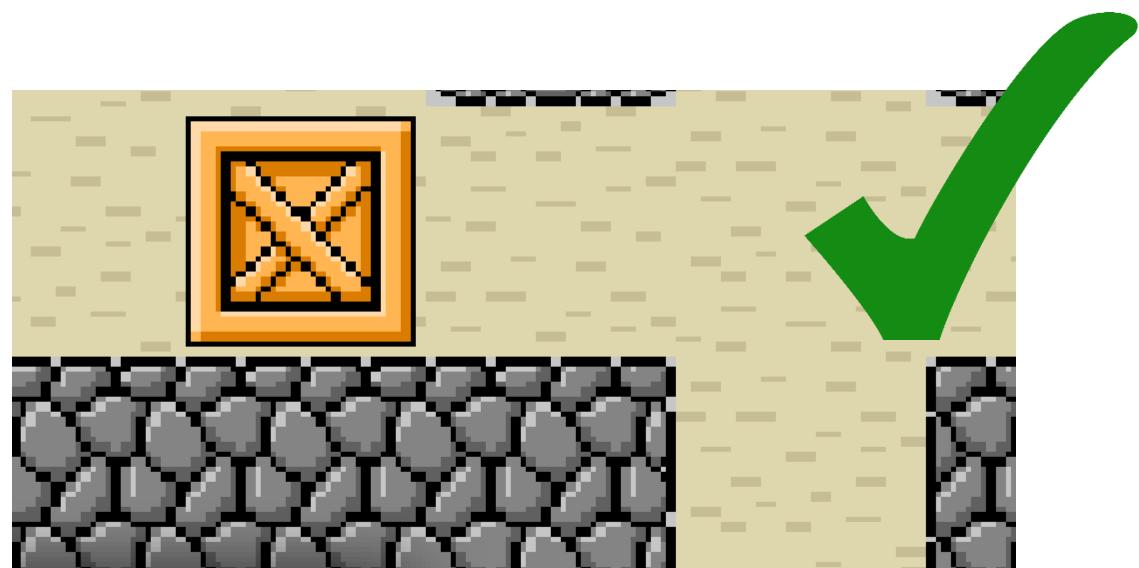
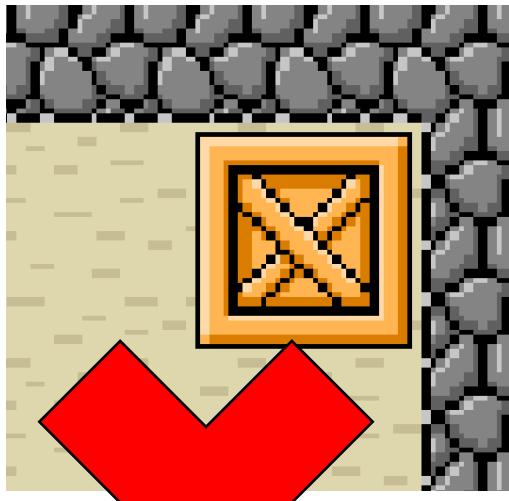
Tables de hachage et dictionnaires

- C'est la structure de données la plus efficace que nous ayons trouvé
- Il faut maintenant optimiser !

Optimisations

- Permuter des caisses dans la liste ne change pas le jeu représenté → On trie
- Il existe des positions à partir desquelles il est impossible de gagner

Optimisations

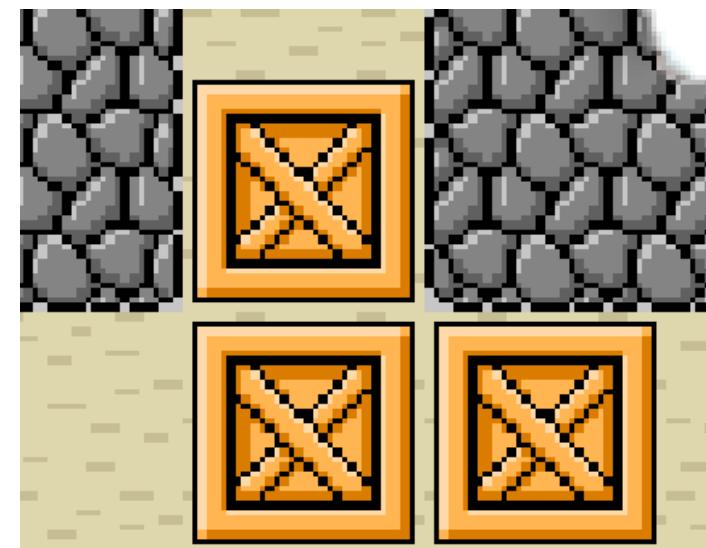
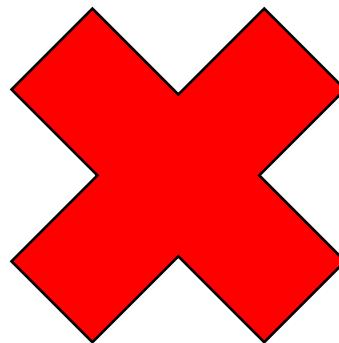
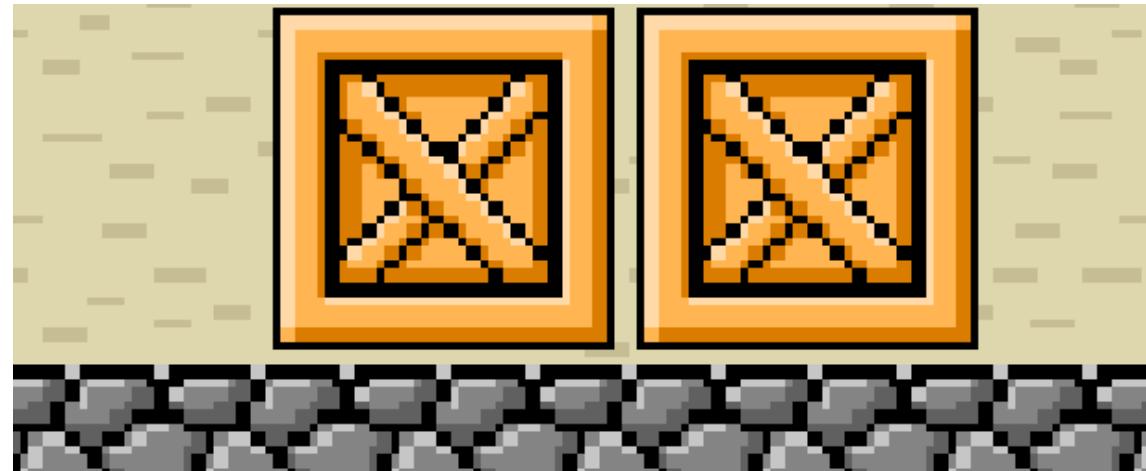
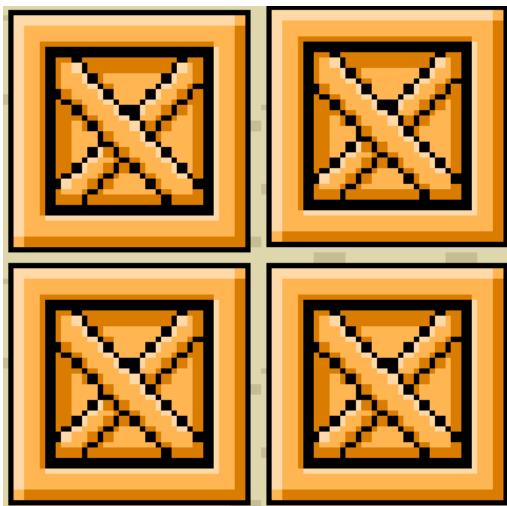


Optimisations

- Sont repérable dès le début
→ On peut les marquer

```
def est_deadlock1(jeu, x, y):  
    "Verifie si la case x, y est un coin dans le labyrinthe (case supposee non position de victoire)"  
    L = [((1, 0), (0, 1)), ((1, 0), (0, -1)), ((-1, 0), (0, 1)), ((-1, 0), (0, -1))]  
    for ((a, b), (c, d)) in L:  
        if jeu.laby[x + a, y + b] == 1 == jeu.laby[x + c, y + d]:  
            return True  
    return False
```

Optimisations

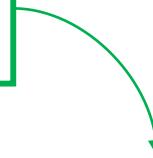


Optimisations

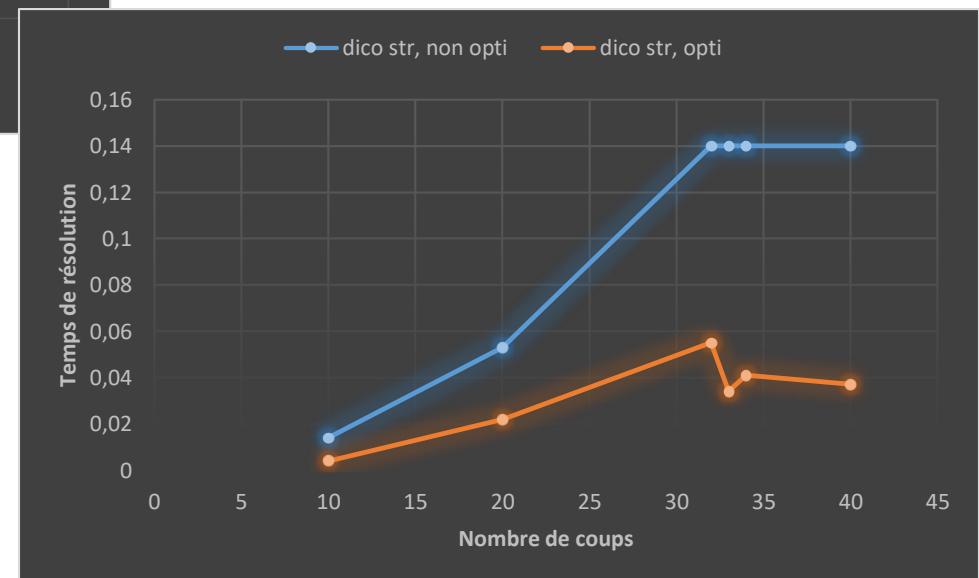
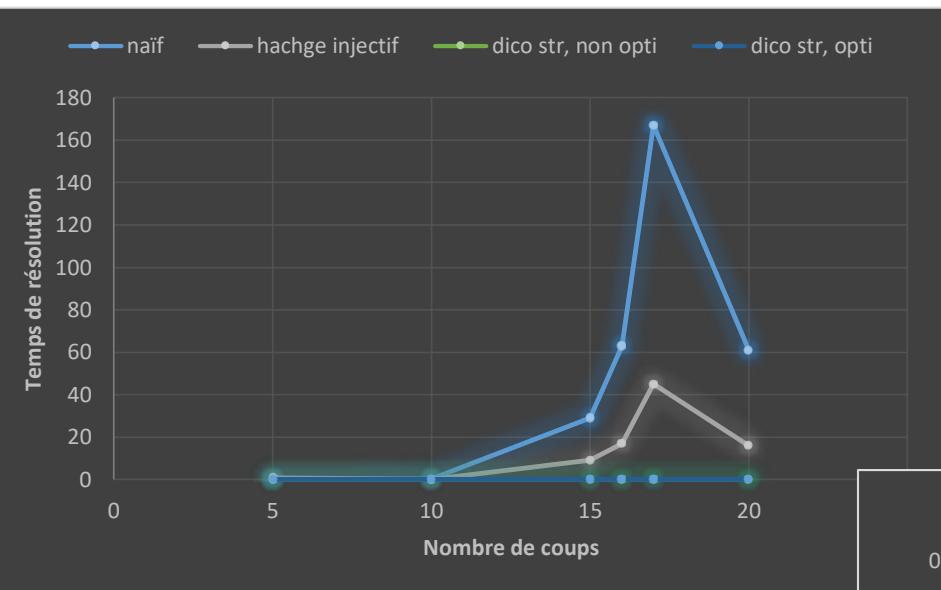
- Doivent être vérifiés en cours de résolution

```
else:  
    if pre_aux(jeu, args, nb_coups_restants) == False:  
        return False  
    nb_coups_restants -= 1
```

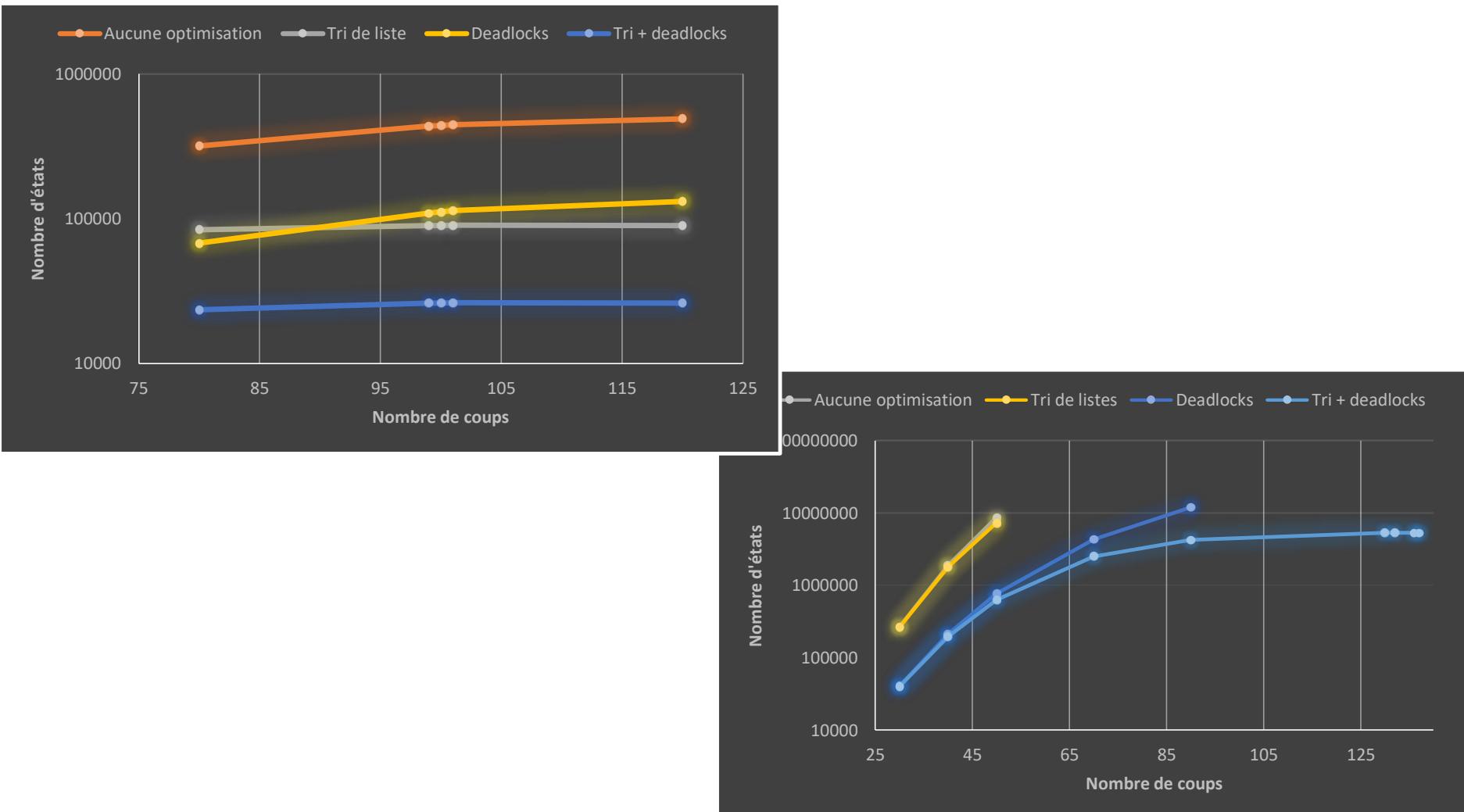
```
def pre_verif_dico(jeu, args, nb_coups_restant):  
    "Actions préliminaires pour le solveur utilisant les dictionnaires"  
    if args["dead"] and nb_coups_restant % 7 == 1:  
        stop = check_deadlock(jeu)  
        if not stop:  
            return False
```



Optimisations



Optimisations



Conclusion

- Objectifs ✓
- Problème classé NP-Difficile et PSPACE-complet
 - Optimisation spatiale possible
- Améliorations possibles :
 - Parcours en largeur
 - Changement total d'approche

Des questions ?

```

8 """
9 ##### Classe de Jeu #####
10 #           Classe de Jeu          #
11 ##### ##### ##### ##### #####
12 """
13 import numpy as np
14
15 class Jeu:
16     "Definit l'objet jeu"
17
18     def __init__(self, labyrinthe, pos_joueur, pos_caisse, pos_victoire, case=0):
19         self.laby = labyrinthe
20         self.pj = pos_joueur
21         self.pc = pos_caisse
22         self.pv = pos_victoire
23         self.case = case
24
25     def __repr__(self):
26         res = self.laby.__repr__() + '\n'
27         res += 'Position du joueur : ' + str(self.pj) + '\n'
28         res += 'Positions des caisses : ' + self.pc.__repr__() + '\n'
29         res += 'Positions de victoire : ' + self.pv.__repr__()
30         return res
31
32     def go_h (self, check=False, ret=False) :
33         "Modifie (si possible) le labyrinthe en faisant monter le joueur"
34         if check or self.go_h_bool():
35             x, y = self.pj
36             if self.laby[x - 1, y] == 2 :
37                 self.laby[x - 2, y] = 2
38                 self.laby[x, y], self.case = self.case, 0
39                 self.laby[x - 1, y] = 5
40                 self.pj = (x - 1, y)
41                 i = self.pc.index([x - 1, y])
42                 self.pc[i] = [x - 2, y]
43                 if ret :
44                     return True, i, [x - 1, y], [x - 2, y]
45             else :
46                 self.laby[x, y], self.case = self.case, self.laby[x - 1, y]
47                 self.laby[x - 1, y] = 5

```

```

44             return True, i, [x - 1, y], [x - 2, y]
45     else :
46         self.laby[x, y], self.case = self.case, self.laby[x - 1, y]
47         self.laby[x - 1, y] = 5
48         self.pj = (x - 1, y)
49         if ret:
50             return False, None, None, None
51     if ret:
52         return False, None, None, None
53
54 def go_b (self, check=False, ret=False) :
55     "Modifie (si possible) le labyrinthe en faisant descendre le joueur"
56     if check or self.go_b_bool():
57         x, y = self.pj
58         if self.laby[x + 1, y] == 2 :
59             self.laby[x + 2, y] = 2
60             self.laby[x, y], self.case = self.case, 0
61             self.laby[x + 1, y] = 5
62             self.pj = (x + 1, y)
63             i = self.pc.index([x + 1, y])
64             self.pc[i] = [x + 2, y]
65             if ret :
66                 return True, i, [x + 1, y], [x + 2, y]
67     else :
68         self.laby[x, y], self.case = self.case, self.laby[x + 1, y]
69         self.laby[x + 1, y] = 5
70         self.pj = (x + 1, y)
71         if ret:
72             return False, None, None, None
73     if ret:
74         return False, None, None, None
75
76 def go_d (self, check=False, ret=False) :
77     "Modifie (si possible) le labyrinthe en faisant aller a droite le joueur"
78     if check or self.go_d_bool():
79         x, y = self.pj
80         if self.laby[x, y + 1] == 2 :
81             self.laby[x, y + 2] = 2
82             self.laby[x, y], self.case = self.case, 0
83             self.laby[x, y + 1] = 5

```

```

80         if self.laby[x, y + 1] == 2 :
81             self.laby[x, y + 2] = 2
82             self.laby[x, y], self.case = self.case, 0
83             self.laby[x, y + 1] = 5
84             self.pj = (x, y + 1)
85             i = self.pc.index([x, y + 1])
86             self.pc[i] = [x, y + 2]
87             if ret :
88                 return True, i, [x, y + 1], [x, y + 2]
89         else :
90             self.laby[x, y], self.case = self.case, self.laby[x, y + 1]
91             self.laby[x, y + 1] = 5
92             self.pj = (x, y + 1)
93             if ret:
94                 return False, None, None, None
95     if ret:
96         return False, None, None, None
97
98 def go_g (self, check=False, ret=False) :
99     "Modifie (si possible) le labyrinthe en faisant aller a gauche le joueur"
100    if check or self.go_g_bool():
101        x, y = self.pj
102        if self.laby[x, y - 1] == 2 :
103            self.laby[x, y - 2] = 2
104            self.laby[x, y], self.case = self.case, 0
105            self.laby[x, y - 1] = 5
106            self.pj = (x, y - 1)
107            i = self.pc.index([x, y - 1])
108            self.pc[i] = [x, y - 2]
109            if ret :
110                return True, i, [x, y - 1], [x, y - 2]
111        else :
112            self.laby[x, y], self.case = self.case, self.laby[x, y - 1]
113            self.laby[x, y - 1] = 5
114            self.pj = (x, y - 1)
115            if ret:
116                return False, None, None, None
117     if ret:
118         return False, None, None, None
119

```

```

120 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]:
124         return True
125     elif self.laby[x - 1, y] == 2:
126         if self.laby[x - 2, y] in [2, 1, 4]:
127             return False
128         else :
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]:
137         return True
138     elif self.laby[x + 1, y] == 2:
139         if self.laby[x + 2, y] in [2, 1, 4]:
140             return False
141         else :
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]:
150         return True
151     elif self.laby[x, y + 1] == 2:
152         if self.laby[x, y + 2] in [2, 1, 4]:
153             return False
154         else :
155             return True
156     else :
157         return False
#Case superieure = vide
#Case superieur = caisse
#Caisse non bougeable
#Caisse bougeable
#Case inferieure = vide
#Case inferieure = caisse
#Caisse non bougeable
#Caisse bougeable
#Case de droite = vide
#Case de droite = caisse
#Caisse non bougeable
#Caisse bougeable

```



```

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]:
163         return True
164     elif self.laby[x, y - 1] == 2:
165         if self.laby[x, y - 2] in [2, 1, 4]:
166             return False
167         else :
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" : Jeu.go_d_bool, "Gauche" : Jeu.go_g_bool}
175     return D[direction](self)
176
177 def avancee(self, direction, check=False, ret=False) :
178     "Fait avancer le personnage dans la direction donnee (si possible)"
179     D = {"Haut" : Jeu.go_h, "Bas" : Jeu.go_b, "Droite" : Jeu.go_d, "Gauche" : Jeu.go_g}
180     res = D[direction](self, check, ret)
181     if ret:
182         return res
183
184 def copy(self):
185     "Renvoie une copie du jeu"
186     lab = np.array([[self.laby[i, j] for j in range(len(self.laby[0]))] for i in range(len(self.laby))])
187     pc = [[self.pc[i][j] for j in range(len(self.pc[0]))] for i in range(len(self.pc))]
188     pj = self.pj
189     pv = [[self.pv[i][j] for j in range(len(self.pv[0]))] for i in range(len(self.pv))]
190     case = self.case
191     return Jeu(lab, pj, pc, pv, case)

```



```

16 from os import *
17 chdir(path.dirname(__file__))
18 from _Classe_jeu import *
19 import numpy as np
20
21
22 """
23 ##### Bibliotheque de niveaux #####
24 #          Bibliothèque de niveaux      #
25 #####
26 """
27
28
29 lvl1 = np.array(
30 [[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],           #Faisable en 238 coups
31 [1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
32 [1,1,1,1,1,2,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
33 [1,1,1,1,1,0,0,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
34 [1,1,1,0,0,2,0,2,0,1,1,1,1,1,1,1,1,1,1,1,1,1],
35 [1,1,1,0,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1],
36 [1,0,0,0,1,0,1,1,0,1,1,1,1,1,1,0,0,0,0,1],
37 [1,0,2,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1],
38 [1,1,1,1,1,0,1,1,1,0,1,5,1,1,0,0,0,0,1],
39 [1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1],
40 [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],
81 [1, 0, 0, 0, 0, 1],
82 [1, 1, 1, 1, 1, 1]])

```

```

84 jeu_test2 = Jeu(lvl_test2,
85 (2, 3),
86 [[3, 2], [3, 3], [4, 3]],
87 [[3, 3], [4, 3], [4, 2]])
88
89
90 lvl_micro1 = np.array(
91 [[1, 1, 1, 1, 1, 1],
92 [1, 0, 0, 1, 1, 1],
93 [1, 0, 0, 1, 1, 1],
94 [1, 2, 5, 0, 0, 1],
95 [1, 0, 0, 2, 0, 1],
96 [1, 0, 0, 1, 1, 1],
97 [1, 1, 1, 1, 1, 1]])
98
99 jeu_micro1 = Jeu(lvl_micro1,
100 (3, 2),
101 [[3, 1], [4, 3]],
102 [[1, 2], [3, 1]])
103
104 lvl_micro3 = np.array(
105 [[1, 1, 1, 1, 1, 1, 1, 1, 1],
106 [1, 1, 1, 0, 1, 1, 1, 1, 1],
107 [1, 0, 0, 0, 0, 0, 2, 0, 1],
108 [1, 0, 1, 0, 0, 1, 2, 0, 1],
109 [1, 0, 0, 0, 0, 1, 5, 0, 1],
110 [1, 1, 1, 1, 1, 1, 1, 1, 1]])
111
112 jeu_micro3 = Jeu(lvl_micro3,
113 (4, 6),
114 [[2, 6], [3, 6]],
115 [[4, 2], [4, 4]])
116

```

```

117 lvl_micro11 = np.array(                                     #Faisable en 78 coups
118 [[1,1,1,1,1,1,1,1,1],
119 [1,1,1,0,0,0,0,1,1],
120 [1,1,1,0,1,1,5,1,1],
121 [1,1,1,0,1,0,2,0,1],
122 [1,0,0,0,1,0,2,0,1],
123 [1,0,0,0,0,0,0,0,1],
124 [1,0,0,1,1,1,1,1,1],
125 [1,1,1,1,1,1,1,1,1]])
126
127 jeu_micro11 = Jeu(lvl_micro11,
128 (2,6),
129 [[3,6],[4,6]],
130 [[4,2],[4,3]]))
131
132 lvl_micro16 = np.array(                                     #Faisable en 100 coups
133 [[1,1,1,1,1,1,1,1,1],
134 [1,1,0,0,1,1,1,1,1],
135 [1,1,0,0,0,0,0,1,1],
136 [1,1,0,1,1,0,0,0,1],
137 [1,0,0,0,1,0,5,2,1],
138 [1,0,0,0,1,0,2,2,0],
139 [1,0,0,0,1,0,0,0,1],
140 [1,1,1,1,1,1,1,1,1]])
141
142 jeu_micro16 = Jeu(lvl_micro16,
143 (4,6),
144 [[4,7],[5,6],[5,7]],
145 [[4,1],[4,3],[6,3]]))
146
147 lvl_cosmos1 = np.array(                                     #Faisable en 49 coups
148 [[1,1,1,1,1,1,1,1,1],
149 [1,0,0,1,1,1,0,0,1],
150 [1,0,2,0,2,0,2,0,1],
151 [1,0,0,0,5,0,0,0,1],
152 [1,1,1,0,0,2,1,1,1],
153 [1,1,1,0,0,0,1,1,1],
154 [1,1,1,1,1,1,1,1,1]])

```

```
156 jeu_cosmos1 = Jeu(lvl_cosmos1,  
157 (3,4),  
158 [[2,2], [2,4], [2, 6], [4, 5]],  
159 [[2,4], [3,4], [4,4], [5,4]])  
160  
161 lvl_freebox = np.array( #Faisable en 133 coups  
162 [[1,1,1,1,1,1,1,1,1],  
163 [1,1,1,0,0,0,1,1,1],  
164 [1,0,0,2,0,0,2,0,1],  
165 [1,0,1,0,1,1,0,0,1],  
166 [1,0,1,0,1,1,5,1,1],  
167 [1,0,0,2,0,0,2,1,1],  
168 [1,1,1,0,1,2,0,1,1],  
169 [1,1,1,0,0,0,0,1,1],  
170 [1,1,1,2,1,0,0,1,1],  
171 [1,1,1,0,0,2,0,1,1],  
172 [1,1,1,1,1,0,0,1,1],  
173 [1,1,1,1,1,1,1,1,1]])  
174  
175 jeu_freebox = Jeu(lvl_freebox,  
176 (4,6),  
177 [[2,3], [2,6], [5,3], [5,6], [6,5], [8,3], [9,5]],  
178 [[2,4], [2,5], [5,4], [5,5], [7,4], [7,6], [8,6]])
```

```

18 from os import *
19 chdir(path.dirname(__file__))
20 import numpy as np
21 import time
22 from _Interface_graphique import *
23 from _Classe_jeu import *
24 from _Bibliotheque_niveaux import *
25
26
27
28 """
29 ##### Implementation #####
30 #           Implementation      #
31 #####
32 """
33
34
35
36 |
37 def victoire(jeu):
38     "Renvoie True si le jeu est gagne, False sinon"
39     for i in jeu.pc:
40         if not i in jeu.pv:                                #Teste si chaque position de victoire est occupee
41             return False
42     return True
43
44 def jouer(jeu) :
45     "Permet de jouer a un niveau"
46     cop = jeu.copy()
47     #cop = jeu
48     L = []
49     D = {"z" : "Haut", "s" : "Bas", "q" : "Gauche", "d" : "Droite"}
50     while 1:
51         if victoire(cop) :
52             print("Probleme resolu !")
53             print("Resolu en {} coups".format(len(L)))
54             return L

```

```

50     while 1:
51         if victoire(cop) :
52             print("Probleme resolu !")
53             print("Resolu en {} coups".format(len(L)))
54             return L
55         else :
56             print(cop.laby)
57             direction = input("Entrez la direction : ")
58             if direction in D.keys():
59                 cop.avancee(D[direction])
60                 L.append(D[direction])
61             elif direction == "Quit" :
62                 print("Abandon")
63                 break
64             print('Deplacements : {}'.format(len(L)))
65
66 def complementaire(direc):
67     "Renvoie la direction opposee a direc"
68     D = {"Haut" : "Bas", "Bas" : "Haut", "Droite" : "Gauche", "Gauche" : "Droite"}
69     return D[direc]
70
71 def solveur_general(jeu, fonction_aux, pre_aux, nb_coups_restants, args={}):
72     "Solveur general sur lequel sont utilises les differents modes de resolution"
73     if args["affichage"]:
74         print(args["coup"], " ", nb_coups_restants)
75         print(jeu)
76
77     if victoire(jeu):
78         return True
79
80     elif nb_coups_restants == 0:
81         return False
82
83     else:
84         if pre_aux(jeu, args, nb_coups_restants) == False:
85             return False
86         nb_coups_restants -= 1
87
88         for d in ["Haut", "Bas", "Droite", "Gauche"]:

```

```

86 nb_coups_restants -= 1
87
88 for d in ["Haut", "Bas", "Droite", "Gauche"]:
89     if jeu.avancee_bool(d):
90         args["coup"] = d
91         if fonction_aux(jeu, nb_coups_restants, args):
92             args["sol"].append(d)
93             return True
94
95 return False
96
97 def aux_solveur_naif(jeu, nb_coups_restant, args):
98     "Fonction auxiliaire pour le solveur naif"
99     coup, pos_caisse, affichage = args["coup"], args["pos_caisse"], args["affichage"]
100    pos_caisse = [[args["pos_caisse"][i][0], args["pos_caisse"][i][1]] for i in range(len(jeu.pc))]
101    jeu.avancee(coup)
102    res = solveur_general(jeu, aux_solveur_naif, pre_verif_naif, nb_coups_restant, args)
103    jeu.avancee(complementaire(coup))
104    reset_caisse(jeu, pos_caisse)
105    args["pos_caisse"] = [[jeu.pc[i][0], jeu.pc[i][1]] for i in range(len(jeu.pc))]
106    if affichage:
107        print(coup)
108        print(jeu.pc)
109    return res
110
111 def pre_verif_naif(jeu, args, nb_coups_restant):
112     "Actions préliminaires pour le solveur naïf"
113     args["pos_caisse"] = [[jeu.pc[i][0], jeu.pc[i][1]] for i in range(len(jeu.pc))] #Copie de la position des caisses pour le reset
114
115 def solveur_naif(jeu, nb_coups_restants, affichage=False):
116     "Solveur naïf v1"
117     args = {"affichage": affichage, "sol": [], "coup": "Haut"}
118     solveur_general(jeu, aux_solveur_naif, pre_verif_naif, nb_coups_restants, args)
119     args["sol"].reverse()
120     return args["sol"]

```

```

122 def reset_caisses(jeu, pos_caisses):
123     "Retablie les positions des caisses du jeu aux positions pos_caisses"
124     set_caisses = set((pos_caisses[i][0], pos_caisses[i][1]) for i in range(len(pos_caisses)))
125     newcaisses = set((jeu.pc[i][0], jeu.pc[i][1]) for i in range(len(jeu.pc)))
126     caisses_a_enlever = newcaisses - set_caisses
127     caisses_a_rajouter = set_caisses - newcaisses
128     for pos in caisses_a_enlever:
129         jeu.laby[pos] = 0
130     for pos in caisses_a_rajouter:
131         jeu.laby[pos] = 2
132     jeu.pc = [[pos_caisses[i][0], pos_caisses[i][1]] for i in range(len(pos_caisses))]
133
134 """
135
136 def aux_solveur_un_tout_petit_peu_moins_naif(jeu, nb_coups_restants, args):
137     "Fonction auxiliaire pour le solveur un tout petit peu moins naif"
138     coup, affichage = args["coup"], args["affichage"]
139     boole, i, old, new = jeu.avancee(coup, True, True)
140     res = solveur_general(jeu, aux_solveur_un_tout_petit_peu_moins_naif, pre_verif_un_tout_petit_peu_moins_naif, nb_coups_restants, args)
141     jeu.avancee(complementaire(coup), check = True)
142     reset_caisses2(jeu, boole, i, old, new)
143     if affichage:
144         print(coup)
145         print(jeu.pc)
146     return res
147
148 def pre_verif_un_tout_petit_peu_moins_naif(jeu, args, nb_coups_restant):
149     "Actions preliminaires pour le solveur un tout petit peu moins naif"
150     pass
151
152 def solveur_un_tout_petit_peu_moins_naif(jeu, nb_coups_restants, affichage=False):
153     "Solveur naif v2"
154     args = {"affichage" : affichage, "sol" : [], "coup" : "Haut"}
155     solveur_general(jeu, aux_solveur_un_tout_petit_peu_moins_naif, pre_verif_un_tout_petit_peu_moins_naif, nb_coups_restants, args)
156     args["sol"].reverse()
157     return args["sol"]
158
#Transformation en ensemble pour utiliser l'intersection
#Pas d'actions preliminaires pour ce solveur

```



```
160 def reset_caisse(jeu, boole, i, old, new):
161     "Retablie les positions des caisses du jeu"
162     if boole:
163         jeu.pc[i] = old
164         [a, b] = old
165         jeu.laby[a, b] = 2
166         [a, b] = new
167         jeu.laby[a, b] = 0
168
169
170
171 """
172 #####
173 #          Tables de hachages          #
174 #####
175 #####
176 """
177
178
179
180
181 def table_hachage(n=257) :
182     "Cree une table de False"
183     return np.array([False for _ in range(n)])
184
185 def hachage1(jeu, n=257) :
186     "Fonction de hachage"
187     liste = []
188     for x in jeu.pc :
189         liste += x
190     a, b = jeu.pj
191     liste += [a, b]
192     cle = 0
193     i = 1
194     for x in liste :
195         cle += (x * i) ** 5
196         i += 1
197     return cle % n
198
```

```

199 def hachage2(jeu, n=257):
200     "Autre fonction de hachage"
201     liste = []
202     for x in jeu.pc :
203         liste += x
204     a, b = jeu.pj
205     liste += [a, b]
206     cle = 0
207     i = 1
208     for x in liste :
209         cle += i ** x
210         i += 1
211     return cle % n
212
213 def aux_solveur_hachage(jeu, nb_coups_restants, args):
214     "Fonction auxiliaire pour le solveur utilisant les tables de hachage"
215     coup, affichage, tab_hash, hachage, n = args["coup"], args["affichage"], args["tab_hash"], args["hachage"], args["n"]
216     boole, i, old, new = jeu.avancee(coup, True, True)
217     cle = hachage(jeu, n)
218     res = False
219     if not tab_hash[cle]:                                     #Position pas encore visitée
220         tab_hash[cle] = True
221         res = solveur_general(jeu, aux_solveur_hachage, pre_verif_hachage, nb_coups_restants, args)
222     jeu.avancee(complementaire(coup), check=True)
223     reset_caisse2(jeu, boole, i, old, new)
224     if affichage:
225         print(coup)
226         print(jeu.pc)
227         print("Hash : ", cle)
228     return res
229
230 def pre_verif_hachage(jeu, args, nb_coups_restant):
231     "Actions préliminaires pour le solveur utilisant les tables de hachage"
232     pass                                                 #Pas d'actions préliminaires pour ce solveur
233

```

```

234 def solveur_hachage(jeu, nb_coups_restants, tab_hash, hachage, affichage=False):
235     "Solveur hachage v1"
236     n = len(tab_hash)
237     args = {"tab_hash" : tab_hash, "hachage" : hachage, "n" : n, "coup" : "Haut", "affichage" : affichage, "sol" : []}
238     solveur_general(jeu, aux_solveur_hachage, pre_verif_hachage, nb_coups_restants, args)
239     args["sol"].reverse()
240     return args["sol"]
241
242 """-----
243
244 def table_hachage_injectif(jeu) :
245     "Cree une table de False pour le hachage injectif"
246     n = max(len(jeu.laby), len(jeu.laby[0]))
247     p = len(str(n))
248     taille = n ** (p * 2 * (len(jeu.pc) + 1))                                #Nombre d'etats possibles : n ** ((nb_cais
249     return np.array([False for _ in range(int(taille))]), np.array([0 for _ in range(int(taille))]))
250
251 def to_str_len_p(x,p):
252     "Renvoie x convertie en str sur p caracteres"
253     x = str(x)
254     xlen = len(x)
255     if xlen < p:
256         x = "0" * (p - xlen) + x                                         #Ajout d'eventuels 0 pour obtenir une chaîne de longueur p
257     return x
258
259 def hachage_injectif(jeu) :
260     "Fonction de hachage injective"
261     n, p = max(len(jeu.laby), len(jeu.laby[0])), len(jeu.pc)
262     cle = 0
263     for i in range(p):
264         cle += jeu.pc[i][0] * (n ** (2 * i)) + jeu.pc[i][1] * (n ** (2 * i + 1))      #Ecriture en "base n"
265     x, y = jeu.pj
266     cle += x * (n ** (2 * p)) + y * (n ** (2 * p + 1))
267     return cle

```

```

269 def aux_solveur_hachage_injectif(jeu, nb_coups_restants, args):
270     "Fonction auxiliaire pour le solveur utilisant la fonction de hachage injective"
271     tab_coups, tab_cle, coup, affichage = args["tab_coups"], args["tab_cle"], args["coup"], args["affichage"]
272     boole, i, old, new = jeu.avancee(coup, True, True)
273     cle = hachage_injectif(jeu)
274     res = False
275     if not tab_cle[cle] or (nb_coups_restants > tab_coups[cle]):                                #Position non deja visitez ou visitez avec un nombre
276         tab_cle[cle] = True
277         tab_coups[cle] = nb_coups_restants
278         res = solveur_general(jeu, aux_solveur_hachage_injectif, pre_verif_hachage_injectif, nb_coups_restants, args)
279     jeu.avancee(complementaire(coup), check = True)
280     reset_caisses2(jeu, boole, i, old, new)
281     if affichage:
282         print(coup)
283         print(jeu.pc)
284         print("Hash : ", cle)
285     return res
286
287 def pre_verif_hachage_injectif(jeu, args, nb_coups_restant):
288     "Actions preliminaires pour le solveur utilisant la fonction de hachage injective"
289     pass                                              #Pas d'actions preliminaires pour ce solveur
290
291 def solveur_hachage_injectif(jeu, nb_coups_restants, affichage=False):
292     "Solveur hachage v2"
293     tab_cle, tab_coups = table_hachage_injectif(jeu)
294     args = {"sol" : [], "coup" : "Haut", "tab_coups" : tab_coups, "tab_cle" : tab_cle, "affichage" : affichage}
295     solveur_general(jeu, aux_solveur_hachage_injectif, pre_verif_hachage_injectif, nb_coups_restants, args)
296     args["sol"].reverse()
297     return args["sol"]
298
299
300
301
302 """
303 ##### Version dictionnaire #####
304 #      Version dictionnaire      #
305 ##### Version dictionnaire #####
306 """

```

```

311 def hachage_tuple(jeu):
312     "Fonction de hachage avec les tuples pour le solveur version dictionnaire"
313     L = sorted(jeu.pc)
314     #L = jeu.pc
315     pc_tuple = tuple(tuple(L[i]) for i in range(len(jeu.pc)))
316     key = jeu.pj, pc_tuple
317     return key
318
319 def hachage_str(jeu):
320     "Fonction de hachage avec les chaines de caracteres pour le solveur version dictionnaire"
321     L = sorted(jeu.pc)
322     #L = jeu.pc
323     x, y = jeu.pj
324     LL = [str(x), str(y)]
325     LL += [str(L[i][j]) for i in range(len(L)) for j in range(2)]
326     key = " ".join(LL)
327     return key
328
329 def lprem():
330     "Liste des nombres premiers inferieurs a 1000"
331     return [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,153]
332
333 nbprem = lprem()
334
335 def hachage_nbprem(jeu):
336     x, y = jeu.pj
337     L = [[x, y]] + sorted(jeu.pc)
338     key = 1
339     for k in range(len(L)):
340         key *= nbprem[2 * k] ** L[k][0]
341         key *= nbprem[2 * k + 1] ** L[k][1]
342     return key

```

#Transformation en tuple

#Transformation en string

```

344 def aux_solveur_dico(jeu, nb_coups_restant, args):
• 345     "Fonction auxiliaire pour le solveur utilisant les dictionnaires"
• 346     dic, coup, affichage = args["dic"], args["coup"], args["affichage"]
• 347     boole, i, old, new = jeu.avancee(coup, True, True)
• 348     cle = hachage_str(jeu)
• 349     res = False
• 350     try:
• 351         nb_coup = dic[cle]
• 352         if nb_coups_restant > nb_coup:
• 353             dic[cle] = nb_coups_restant
• 354             res = solveur_general(jeu, aux_solveur_dico, pre_verif_dico, nb_coups_restant, args)
• 355     except KeyError:
• 356         dic[cle] = nb_coups_restant
• 357         res = solveur_general(jeu, aux_solveur_dico, pre_verif_dico, nb_coups_restant, args)
jeu.avancee(complementaire(coup), check=True)
• 358     reset_caisses2(jeu, boole, i, old, new)
• 359     if affichage:
• 360         if coup:
• 361             print(coup)
• 362             print(jeu.pc)
• 363             print("Hash : ", cle)
• 364     return res
365
366 def pre_verif_dico(jeu, args, nb_coups_restant):
• 367     "Actions preliminaires pour le solveur utilisant les dictionnaires"
• 368     if args["dead"] and nb_coups_restant % 7 == 1:
• 369         stop = check_deadlock(jeu)
• 370         if not stop:
• 371             return False
372
373 def solveur_dico(jeu, nb_coups_restants, dic=None, dead=True, aff=False):
• 374     "Solveur dico v1"
• 375     if dic is None:
• 376         dic = {}
• 377     args = {"sol" : [], "coup" : "Haut", "affichage" : aff, "dic" : dic, "dead" : dead}
• 378     if dead :
• 379         reperage(jeu)
• 380     solveur_general(jeu, aux_solveur_dico, pre_verif_dico, nb_coups_restants, args)
• 381     args["sol"].reverse()

```

#Mouvement
#Position déjà vue
#On ne réessaie que si il nous reste plus de coups que la
#Si la position n'a pas été vue (Exception KeyError déclarée)
#Annulation du coup
#Pour le debug
#On vérifie les deadlocks tous les 7 coups
#On repère les cases menant à un coup sur une défaite

```

380     solveur_general(jeu, aux_solveur_dico, pre_verif_dico, nb_coups_restants, args)
381     args["sol"].reverse()
382     if dead:
383         remove_deads(jeu)
384     return args["sol"]
385
386 def test_sol(jeu, sol):
387     "Verefie si sol est bien solution du jeu"
388     cop = jeu.copy()
389     for x in sol:
390         cop.avancee(x)
391     if victoire(cop):
392         print(cop.laby)
393         return "Probleme resolu !"
394     else:
395         print(cop.laby)
396         return "Ce n'est pas une solution"
397
398
399
400
401 """
402 ##### Deadlocks #####
403 #             Deadlocks          #
404 #####
405 """
406
407
408
409 def est_deadlock1(jeu, x, y):
410     "Verifie si la case x, y est un coin dans le labyrinthe (case supposee non position de victoire)"
411     L = [(1, 0), (0, 1), (1, 0), (0, -1), (-1, 0), (0, 1), (-1, 0), (0, -1)]           # Coin bas - droite / bas - gauche /
412     for ((a, b), (c, d)) in L:
413         if jeu.laby[x + a, y + b] == 1 == jeu.laby[x + c, y + d]:
414             return True
415     return False

```

```

417 def deadlock2(jeu):
418     "Deadlocks des carres"
419     if len(jeu.pc) < 4:
420         return True
421     else:
422         for [x, y] in jeu.pc:
423             if jeu.laby[x + 1, y] == jeu.laby[x, y + 1] == jeu.laby[x + 1, y + 1] == 2 or \
424                 jeu.laby[x + 1, y] == jeu.laby[x, y - 1] == jeu.laby[x + 1, y - 1] == 2 or \
425                 jeu.laby[x - 1, y] == jeu.laby[x, y + 1] == jeu.laby[x - 1, y + 1] == 2 or \
426                 jeu.laby[x - 1, y] == jeu.laby[x, y - 1] == jeu.laby[x - 1, y - 1] == 2:
427                 if not [x, y] in jeu.pv:
428                     return False
429     return True
430
431 def est_deadlock3(jeu, x, y) :
432     "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)"
433     #Chaque boucle fais le meme teste, en changeant le cote longe
434     if jeu.laby[x, y + 1] == 1:                                #Mur vertical droit
435         i = x + 1                                         #On avance vers le bas
436         test_pv_i = False                                 #True si on a rencontre une p_v, False sinon
437         rencontre_mur_i = False                           #True si on est arrive en face d'un mur, False sinon
438         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
439             if [i, y] in jeu.pv:                            #Si on a rencontre un mur mais pas de p_v, on essaie en descente
440                 test_pv_i = True
441             elif jeu.laby[i, y] in [1, 4]:                  #Si on a rencontré un mur dans les deux sens, c'est perdu
442                 rencontre_mur_i = True
443             i += 1
444         if rencontre_mur_i :                            #Si on a rencontré un mur dans les deux sens, c'est perdu
445             j = x - 1
446             test_pv_j = False
447             rencontre_mur_j = False
448             while not test_pv_j and not rencontre_mur_j and jeu.laby[j, y + 1] == 1:
449                 if [j, y] in jeu.pv:
450                     test_pv_j = True
451                 elif jeu.laby[j, y] in [1, 4]:
452                     rencontre_mur_j = True
453                 j -= 1
454         if rencontre_mur_j:
455             return False

```

```

453         j -= 1
454     if rencontre_mur_j:
455         return False
456 if jeu.laby[x, y - 1] == 1:
457     i = x + 1
458     test_pv_i = False
459     rencontre_mur_i = False
460     while not test_pv_i and not rencontre_mur_i and jeu.laby[i, y - 1] == 1:
461         if [i, y] in jeu.pv:
462             test_pv_i = True
463         elif jeu.laby[i, y] in [1, 4]:
464             rencontre_mur_i = True
465         i += 1
466     if rencontre_mur_i:
467         j = x - 1
468         test_pv_j = False
469         rencontre_mur_j = False
470         while not test_pv_j and not rencontre_mur_j and jeu.laby[j, y - 1] == 1:
471             if [j, y] in jeu.pv:
472                 test_pv_j = True
473             elif jeu.laby[j, y] in [1, 4]:
474                 rencontre_mur_j = True
475             j -= 1
476     if rencontre_mur_j:
477         return False
478 if jeu.laby[x - 1, y] == 1:
479     i = y + 1
480     test_pv_i = False
481     rencontre_mur_i = False
482     while not test_pv_i and not rencontre_mur_i and jeu.laby[x - 1, i] == 1:
483         if [x, i] in jeu.pv:
484             test_pv_i = True
485         elif jeu.laby[x, i] in [1, 4]:
486             rencontre_mur_i = True
487         i += 1
488     if rencontre_mur_i:
489         j = y - 1
490         test_pv_j = False
491         rencontre_mur_j = False
492         while not test_pv_j and not rencontre_mur_j and jeu.laby[x - 1, j] == 1:
493             if [x, j] in jeu.pv:
494                 test_pv_j = True
495             elif jeu.laby[x, j] in [1, 4]:
496                 rencontre_mur_j = True
497             j -= 1
498         if rencontre_mur_j:
499             return False
500
501     #Si on a rencontré un mur dans les
502     #Mur vertical gauche
503
504     #Mur horizontal haut
505     #On avance vers la droite
506
507     #On avance vers la gauche
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779

```



```

489     j = y - 1
490     test_pv_j = False
491     rencontre_mur_j = False
492     while not test_pv_j and not rencontre_mur_j and jeu.laby[x - 1, j] == 1:
493         if [x, j] in jeu.pv:
494             test_pv_j = True
495         elif jeu.laby[x, j] in [1, 4]:
496             rencontre_mur_j = True
497             j -= 1
498         if rencontre_mur_j :
499             return False
500     if jeu.laby[x + 1, y] == 1: #Mur horizontal bas
501         i = y + 1
502         test_pv_i = False
503         rencontre_mur_i = False
504         while not test_pv_i and not rencontre_mur_i and jeu.laby[x + 1, i] == 1:
505             if [x, i] in jeu.pv:
506                 test_pv_i = True
507             elif jeu.laby[x, i] in [1, 4]:
508                 rencontre_mur_i = True
509                 i += 1
510             if rencontre_mur_i :
511                 j = y - 1
512                 test_pv_j = False
513                 rencontre_mur_j = False
514                 while not test_pv_j and not rencontre_mur_j and jeu.laby[x + 1, j] == 1:
515                     if [x, j] in jeu.pv:
516                         test_pv_j = True
517                     elif jeu.laby[x, j] in [1, 4]:
518                         rencontre_mur_j = True
519                         j -= 1
520                     if rencontre_mur_j:
521                         return False
522     return True
523

```



```

524 def deadlock4(jeu) :
525     "Test si on a 2 caisses l'une contre l'autre contre un mur"
526     for [x,y] in jeu.pc :
527         if not([x, y] in jeu.pv) :
528             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 \
529                 (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 \
530                 (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 \
531                 (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
532             return False
533     return True
534
535 def deadlock5(jeu):
536     "Deadlock des trois caisses dans un angle"
537     if len(jeu.pc) < 3:
538         return True
539     for [x, y] in jeu.pc:
540         if (jeu.laby[x + 1, y - 1] == 1 and (jeu.laby[x, y - 1] == 2 == jeu.laby[x + 1, y])) \
541             or (jeu.laby[x + 1, y + 1] == 1 and (jeu.laby[x, y + 1] == 2 == jeu.laby[x + 1, y])) \
542             or (jeu.laby[x - 1, y - 1] == 1 and (jeu.laby[x, y - 1] == 2 == jeu.laby[x - 1, y])) \
543             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 - droit
544             return False
545     return True
546
547 def remove_deads(jeu):
548     "Enleve les 4 du jeu"
549     nx, ny = len(jeu.laby), len(jeu.laby[0])
550     for x in range(1, nx - 1):
551         for y in range(1, ny - 1):
552             if jeu.laby[x, y] == 4:
553                 jeu.laby[x, y] = 0
554     jeu.case = 0

```

```

556 def check_deadlock(jeu):
557     "Verifie si le jeu est bloque pour l'un des deadlocks implemente"
558     L = [deadlock2, deadlock4, deadlock5]
559     res = True
560     for f in L:
561         res = res and f(jeu)
562     return res
563
564 def reperage(jeu) :
565     "Mise en place des 4 (deadlocks previsibles)"
566     for x in range(len(jeu.laby)):
567         for y in range(len(jeu.laby[0])):
568             if jeu.laby[x, y] == 0 and not [x, y] in jeu.pv:
569                 if est_deadlock1(jeu, x, y) or not est_deadlock3(jeu, x, y):
570                     jeu.laby[x, y] = 4
571     x, y = jeu.pj
572     if not [x, y] in jeu.pv and (est_deadlock1(jeu, x, y) or not est_deadlock3(jeu, x, y)):
573         jeu.case = 4
574
575
576
577
578 """
579 ##### Exploration guidee #####
580 #           Exploration guidee          #
581 ##### Exploration guidee #####
582 """
583
584
585
586
587 def dir_caisse(x, y, xc, yc):
588     "Renvoie l'ordre des directions dans lesquelles aller pour atteindre la caisse (xc, yc) depuis (x, y)"
589     dx, dy = x - xc, y - yc
590     L = []
591     if dx > 0:
592         if dy > 0:
593             return ["Haut", "Gauche", "Bas", "Droite"]

```

#Fonctions de deadlocks à vérifier

```
591 if dx > 0:
592     if dy > 0:
593         return ["Haut", "Gauche", "Bas", "Droite"]
594     else:
595         return ["Haut", "Droite", "Bas", "Gauche"]
596 elif dx < 0:
597     if dy > 0:
598         return ["Bas", "Gauche", "Haut", "Droite"]
599     else:
600         return ["Bas", "Droite", "Haut", "Gauche"]
601 elif dy > 0:
602     return ["Gauche", "Haut", "Bas", "Droite"]
603 return ["Droite", "Haut", "Bas", "Gauche"]
604
605 def order_dir(jeu):
606     "Renvoie l'ordre dans lequel doivent etre effectus les mouvements pour se rapprocher des caisses"
607     x, y = jeu.pj
608     Ld = [abs(x - jeu.pc[i][0] + y - jeu.pc[i][1]) for i in range(len(jeu.pc))]
609     i = 0
610     for k in range(len(Ld)):
611         if Ld[k] < Ld[i]:
612             i = k
613     return dir_caisse(x, y, jeu.pc[i][0], jeu.pc[i][1])
614
615 def solveur_general2(jeu, fonction_aux, pre_aux, nb_coups_restants, args={}):
616     "Solveur ou l'ordre d'exploration est determine par la distance a la caisse la plus proche"
617     if args["affichage"]:
618         print(args["coup"], " ", nb_coups_restants)
619         print(jeu)
620
621     if victoire(jeu):
622         return True
623
624     elif nb_coups_restants == 0:
625         return False
626
627     else:
628         if pre_aux(jeu, args, nb_coups_restants) == False:
629             return False
```



```

627     else:
628         if pre_aux(jeu, args, nb_coups_restants) == False:
629             return False
630         nb_coups_restants -= 1
631         bh, bb, bd, bg = jeu.go_h_bool(), jeu.go_b_bool(), jeu.go_d_bool(), jeu.go_g_bool()
632         Ldir = order_dir(jeu)
633
634         for d in Ldir:
635             if jeu.avancee_bool(d):
636                 args["coup"] = d
637                 if fonction_aux(jeu, nb_coups_restants, args):
638                     args["sol"].append(d)
639                     return True
640
641     return False
642
643 def aux_solveur_dico2(jeu, nb_coups_restant, args):
644     "Fonction auxiliaire pour le second solveur utilisant les dictionnaires"
645     dic, coup, affichage = args["dic"], args["coup"], args["affichage"]
646     boole, i, old, new = jeu.avancee(coup, True, True)                      #Mouvement
647     cle = hachage_str(jeu)
648     res = False
649     try:                                                               #Position deja vue
650         nb_coup = dic[cle]
651         if nb_coups_restant > nb_coup:
652             dic[cle] = nb_coups_restant
653             res = solveur_general2(jeu, aux_solveur_dico2, pre_verif_dico, nb_coups_restant, args) #On ne reessaie que si il nous reste plus
654     except KeyError:                                                 #Si la position n'a pas ete vue (Exception)
655         dic[cle] = nb_coups_restant
656         res = solveur_general2(jeu, aux_solveur_dico2, pre_verif_dico, nb_coups_restant, args)
657     jeu.avancee(complementaire(coup), check=True)                   #Annulation du coup
658     reset_caisse2(jeu, boole, i, old, new)
659     if affichage:
660         print(coup)
661         print(jeu.pc)
662         print("Hash : ", cle)                                         #Pour Le debug
663
return res

```



```
665 def solveur_dico2(jeu, nb_coups_restants, dic=None, dead=True, aff=False):
666     "Solveur dico pour le second solveur"
667     if dic is None:
668         dic = {}
669     args = {"sol" : [], "coup" : "Haut", "affichage" : aff, "dic" : dic, "dead" : dead}
670     if dead :
671         reperage(jeu)
672     solveur_general2(jeu, aux_solveur_dico2, pre_verif_dico, nb_coups_restants, args)
673     args["sol"].reverse()
674     if dead:
675         remove_deads(jeu)
676     return args["sol"]
```

