

Compte rendu : Missionnaire et Cannibale

Flandin Léo

7 février 2023

Table des matières

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 2 | Étude théorique du cas général | 2 |
| 2.1 | Étude de l'environnement | 2 |
| 2.2 | Description d'un état | 2 |
| 2.3 | Nombre d'état maximum | 3 |
| 2.4 | Réponses aux questions 5,6 et 7 du I | 3 |
| 2.5 | Choix de l'algorithme de recherche et de la stratégie abordée | 4 |
| 3 | Étude expérimentale des performances de l'algorithme | 5 |
| 3.1 | Étude en fonction du nombre de personne maximale sur la barque . | 5 |
| 4 | Proposition d'extensions pour poursuivre ce travail | 7 |
| A | Annexes | 8 |
| A.1 | Etat.py | 8 |
| A.2 | algoDeResolution.py | 9 |

Chapitre 1

Introduction

Ce projet est disponible sur github au lien suivant (latex et python) :
<https://github.com/worlcrafte/probleme-cannibale-et-missionnaire>.

Dans cette revue, nous allons nous pencher sur le problème jouet des missionnaires et des cannibales généralisé à n (avec n supérieur ou égal à 3) ; où n est le nombre de cannibales et de missionnaires. Notre objectif sera de sélectionner un algorithme de recherche permettant trouver le chemin de coût minimal pour amener tous les missionnaires et les cannibales de l'autre côté de la rive sans qu'aucun missionnaire ne soit tué lors de la traversée.

En effet, pour résoudre ce problème, nous avons comme conditions :

- il ne faut jamais qu'il y ait plus de cannibales que de missionnaires
- la barque ne peut pas traverser qu'avec p personnes au maximum (avec p supérieur ou égale à 2).

Chapitre 2

Étude théorique du cas général

2.1 Étude de l'environnement

Le problème des missionnaires et des cannibales est un environnement pleinement observable. On a les informations sur le nombre de missionnaires, de cannibales et la position du bateau. Notre agent n'a donc pas besoin d'anticiper ce qui va se passer ensuite. Nous serons sur l'utilisation d'un agent simple. L'environnement est déterministe : il n'est pas probable qu'il y ait une faible probabilité qu'une personne tombe à l'eau. À chaque fois que l'on réalisera une traversée, nous serons limiter sur nos choix en fonction de l'état courant. De plus, cela aura un impacte sur le choix de l'état suivant en fonction du nombre de personnes sur les rives. Nous sommes donc dans un environnement Stochastique et statique, car l'environnement ne change pas entre le moment où nous recevons le percept et celui où nous agissons. Enfin, nous connaissons l'effet de chacune de nos actions, nous sommes donc sur un environnement connu. Pour finir, l'agent utilisé sera un agent à réflexe simple.

2.2 Description d'un état

Pour cela, nous allons commencer par décrire un état. Un état sera défini par les 3 composantes suivantes :

- le nombre de mercenaires à gauche : nbMg
- le nombre de cannibales à gauche : nbCg
- la position du bateau.

Nous n'avons pas besoin de stocker l'information sur le nombre de missionnaires ou de cannibales à droite. Celle-ci pouvant être facilement obtenu : $n - \text{nbMg}$ pour le nombre de missionnaires à droite et $n - \text{nbCg}$ pour les cannibales. Puis nous devons définir l'état initial et l'état final.

L'état initial est défini comme suit : $\text{nbCg}=n$, $\text{nbMg}=n$ et la barque est sur la rive gauche. Enfin l'état final : $\text{nbCg}=0$, $\text{nbMg}=0$ et la barque est sur la rive droite. Dans ce problème nous définiront une action comme étant la suivante : Quand la barque est à gauche : on la fait passer à droite avec au moins 1 ou 2 ou ... p personnes à l'intérieur. Quand la barque est à droite : on la fait passer à gauche avec au moins 1 ou 2 ou ... p personnes à l'intérieur. Une action est réalisée uniquement si les contraintes sur les états sont respectées. Notre fonction de coût sera basée sur l'action. C'est-à-dire, à chaque fois que la barque change de rive, on ajoute 1 au coût.

2.3 Nombre d'état maximum

Nous nous intéressons maintenant au nombre d'état maximal que nous pouvons avoir pour savoir si c'est raisonnable de le représenter dans la mémoire. En respectant les règles définies, on peut obtenir les états suivants si on ne prend pas compte de la position du bateau et que nous avons n missionnaire et n cannibales (nbMg, nbCg) :

| | | | | | |
|--------------------|--------------------|--------------------|--------------------|-----|--------------------|
| (0,0) | (0,1) | (0,2) | (0,3) | ... | (0,n) |
| (1,0) | (1,1) | (1,2) | (1,3) | ... | (1,n) |
| (2,0) | (2,1) | (2,2) | (2,3) | ... | (2,n) |
| ... | ... | ... | ... | ... | ... |
| (n-1,0) | (n-1,1) | (n-1,2) | (n-1,3) | ... | (n-1,n) |
| (n,0) | (n,1) | (n,2) | (n,3) | ... | (n,n) |

Nous aurons donc au maximum $(3n + 2) \times 2$ états possibles. Nous réalisons $\times 2$ car l'état peut être soit avec la barque à gauche, soit avec celle-ci à droite. Ce résultat n'est valable que pour cette configuration, car le nombre de missionnaires est égal au nombre de cannibales. Sinon, nous pourrions avoir plus d'états.

2.4 Réponses aux questions 5,6 et 7 du I

La partie qui suit sera la réponse au question 5,6 et 7 du 1 du document.

5)

Si nous nous basons sur le fait que $n=3$ et $p=2$ (où p représente le nombre maximum de presonne pouvant monter sur la barque) nous avons, par exemple l'état, suivant : $(3M,3C,G) \leftarrow (3M,1C,D) \rightarrow (3M,2C,G)$. Si on choisit l'état $(3M,3C,G)$ on revient à l'état initial ce qui n'est pas intéressant. Si on choisit l'autre état, il faudra l'expanser pour pouvoir continuer la construction de notre arbre. Nous pouvons donc en conclure qu'il faut obligatoirement garder en mémoire les états déjà expansés pour ne pas se retrouver à expanser un état à l'infini.

6) Si on reprend notre exemple précédent avec $n=3$ et $p=2$ on aurait, par exemple l'état suivant : $(3M,2C,D) \rightarrow (3M,3C,G)$. Ici le seul état pouvant être choisi et le retour à l'état initiale.

7) Il n'existe pas d'action qui ne mène vers aucun n'état. Dans le pire des cas on devra retourner dans un état précédant comme dans le 6).

2.5 Choix de l'algorithme de recherche et de la stratégie abordée

À travers les différentes informations que nous avons rassemblé ci-dessus, nous pouvons choisir quel type d'algorithme serait intéressant pour notre problème. Dans un premier temps, nous pouvons déjà affirmer que nous sommes sur une stratégie de recherche non informée. Nous n'exploitons pas d'informations supplémentaires pour utiliser une fonction d'évaluation. De plus, nous avons dit plutôt qu'il nous fallait garder les anciens états déjà expansés en mémoire. Nous pouvons donc oublier Three-Search. De plus, notre fonction de coût est de 1 par passage. On doit donc sélectionner l'algorithme de graph-search. Nous voulons maintenant choisir quelle stratégie choisir pour trouver la solution optimale. Nous avons défini notre fonction de coût de +1 pour chaque passage la barque entre chaque rive. Nous utilisons donc des coûts uniformes. Cela rend donc inutile la stratégie de coût uniforme (cela revient à faire une stratégie en largeur). Enfin, nous voulons une solution de coût minimal. Nous ne pouvons donc pas choisir la stratégie de profondeur d'abord qui ne donne pas une solution de coût minimal. (par exemple si $n=3$ et $p=4$ nous pourrions avoir : $(3,3,G) \rightarrow (0,2,D) \rightarrow (3,2,G) \rightarrow (0,1,D) \rightarrow (3,1,G) \rightarrow (0,0,D)$. Ce qui retourne une solution avec un coût de 5. Alors que la solution optimal à un coût de 3 : $(3,3,G) \rightarrow (3,0,D) \rightarrow (3,1,G) \rightarrow (0,0,D)$). Nous devons donc choisir l'algorithme Graph-Search avec une stratégie de parcours en largeur. Celui-ci est optimal dans notre situation et comme complexité $O(b^d)$ où b est le facteur de branchement et d la profondeur de la première solution. Nous pouvons aussi nous demander si la méthode de recher en "escalade" est possible : L'escalade est une méthode de recherche en profondeur dans laquelle on introduit une fonction heuristique pour choisir à chaque étape le noeud à générer, au lieu de générer un noeud en prenant une règle de production au hasard. Or celui-ci, maldrès qu'il soit très puissant, n'assure pas une solution optimale.

Chapitre 3

Étude expérimentale des performances de l'algorithme

3.1 Étude en fonction du nombre de personne maximale sur la barque

On pourra remarquer que pour une capacité maximum de 4 personnes sur la barque, nous trouverons toujours une solution. En effet, nous pouvons amener 2 missionnaires et 2 cannibales de l'autre côté de la rive puis n'en ramener que 1 de chaque. Ce qui assure de tout trouver une solution lorsque le nombre de missionnaires et de cannibales à faire passer sont égaux. De plus, selon Martin Gardner, lorsque que nous avons le même nombre de missionnaires et de cannibales à faire passer, et une capacité maximale de la barque égale à 4, le coût minimal pour trouver la solution est de $2*n-3$. Nous pouvons donc observer à travers ce tableau le temps réaliser pour trouver la solution où n est le nombre de mercecnaires ou cannibales et p la capacité maximale de la barque.

TABLE 3.1 – Tableau de l’étude des performances de l’algorithme sur le temps pour un p fixé à 4.

| n | p | coût | temps (en seconde) | | n | p | coût | temps (en seconde) |
|----------|----------|-------------|---------------------------|--|----------|----------|-------------|---------------------------|
| 5 | 4 | 7 | 0.0011 | | 10 | 4 | 17 | 0.0055 |
| 15 | 4 | 27 | 0.0100 | | 20 | 4 | 37 | 0.0162 |
| 25 | 4 | 47 | 0.0141 | | 30 | 4 | 57 | 0.0152 |
| 35 | 4 | 67 | 0.0193 | | 40 | 4 | 77 | 0.0274 |
| 45 | 4 | 87 | 0.0318 | | 50 | 4 | 97 | 0.0364 |
| 55 | 4 | 107 | 0.0422 | | 60 | 4 | 117 | 0.0833 |
| 65 | 4 | 127 | 0.0625 | | 70 | 4 | 137 | 0.0619 |
| 75 | 4 | 147 | 0.0775 | | 80 | 4 | 157 | 0.0825 |
| 85 | 4 | 167 | 0.0843 | | 90 | 4 | 177 | 0.0866 |
| 95 | 4 | 187 | 0.1112 | | 100 | 4 | 197 | 0.1216 |
| 105 | 4 | 207 | 0.1383 | | 110 | 4 | 217 | 0.1233 |
| 115 | 4 | 227 | 0.1315 | | 120 | 4 | 237 | 0.1400 |
| 125 | 4 | 247 | 0.1521 | | 130 | 4 | 257 | 0.1780 |
| 135 | 4 | 267 | 0.1776 | | 140 | 4 | 277 | 0.1836 |
| 145 | 4 | 287 | 0.1979 | | 150 | 4 | 297 | 0.2137 |
| 155 | 4 | 307 | 0.2315 | | 160 | 4 | 317 | 0.2401 |
| 165 | 4 | 327 | 0.2483 | | 170 | 4 | 337 | 0.2847 |
| 175 | 4 | 347 | 0.2779 | | 180 | 4 | 357 | 0.2966 |
| 185 | 4 | 367 | 0.3102 | | 190 | 4 | 377 | 0.3247 |
| 195 | 4 | 387 | 0.3399 | | 200 | 4 | 397 | 0.3649 |
| 205 | 4 | 407 | 0.3779 | | 210 | 4 | 417 | 0.3928 |
| 215 | 4 | 427 | 0.4072 | | 220 | 4 | 437 | 0.4255 |
| 225 | 4 | 447 | 0.4617 | | 230 | 4 | 457 | 0.4638 |
| 235 | 4 | 467 | 0.4860 | | 240 | 4 | 477 | 0.5105 |

Chapitre 4

Proposition d'extensions pour poursuivre ce travail

Pour poursuivre ce travail, nous pouvons maintenant se poser la question cette fois-ci avec nbMg différent de nbCg et nbMg supérieur ou égale à nbCg . De plus, nous pouvons aussi nous demander s'il existe un algorithme plus performant ou non pour réaliser ce problème jouet lorsque $\text{nbCg} \neq \text{nbMg}$. Enfin, il pourrait être aussi intéressant de réaliser un algorithme de recherche en modifiant légèrement l'environnement de la manière suivante : une personne sur la barque a une faible chance de tomber dans l'eau. Et donc, de faire en sorte de prendre en considération qu'au moins une personne puisse tomber tout en respectant les contraintes imposées.

Annexe A

Annexes

A.1 Etat.py

```
1
2     class Etat:
3
4     def __init__(self, nbMg, nbCg, boatPosition, parent, cout):
5         self.nbMg = nbMg
6         self.nbCg = nbCg
7         # La position de la barque sera defini par un booleen. Si
8         sa valeur est egale a vraie alors la barque est a guache sinon
9         elle est a droite
10        self.boatPosition = boatPosition
11        self.parent = parent
12        self.cout = cout
13
14    def get_nbMg(self):
15        return self.nbMg
16
17    def get_nbCg(self):
18        return self.nbCg
19
20    def get_boatPosition(self):
21        return self.boatPosition
22
23    def get_parent(self):
24        return self.parent
25
26    def get_cout(self):
27        return self.cout
28
29    # __eq__ (methode "dunder/magique") permet de redefinir la
30    fonction ==. Cela nous sera utile pour verifie si 2 etats sont
```

```

28     egaux. (fonctionne avec .remove() pour la comparaison d'etat)
        # il est possible de redefinir pour >, <, ... mais inutile
        pour notre cas.
29
30     def __eq__(self, other):
31         # on verifie que les 2 objets sont de la meme classes.
32         if isinstance(other, Etat):
33             return self.nbCg == other.nbCg and self.nbMg == other.
nbMg and self.boatPosition == other.boatPosition

```

Listing A.1 – Python example

A.2 algoDeResolution.py

```

1     from Etat import Etat
2
3     # regle permet de verifier que l'etat calculer respecte bien
    les regles etablis : qu'il n'y ai pas plus de Canibale que de
    missionnaire sur un cote de la rive (sauf s'il n'y a aucun
    missionnaire alors il n'y a aucun risque pour eux)
4
5
6     def rule(etat, n, ajout, ajout2):
7         if 0 <= etat.get_nbMg()+ajout <= n and 0 <= etat.get_nbCg
    (+ajout2) <= n:
8             return (((etat.get_nbMg() + ajout) >= (etat.get_nbCg
    (+ ajout2)) or
9                 (etat.get_nbMg()+ajout) == 0) and
10                 (((n-(etat.get_nbMg()+ajout)) == 0 or
11                 ((n-(etat.get_nbMg()+ajout)) >= (n-(etat.
    get_nbCg()+ajout2))))))
12         else:
13             return False
14
15     # expanse permet d'explorer l'etat courant. Il prend en entree
    l'etat courant, la capacite maximale du tableau, et le nombre
    total de mercenaires et cannibales (ici uniquement n car c 2
    valeurs sont egales ). Il retournera une liste contenant tous
    les etats trouver par le programme respectant les regles
    etablis.
16
17
18     def expanse(etat, p, n):
19         result = []
20         for mEmbarque in range(0, p+1):
21             for cEmbarque in range((0, 1)[mEmbarque == 0], (1, p-
    mEmbarque+1)[p-mEmbarque+1 > 0]):

```

```

22         if (not etat.get_boatPosition()):
23             # il es possible aussi de creer 2 variables
temporaires et de modifier les informations contenus (+ ou -
mais cela revient au meme)
24             if rule(etat, n, mEmbarque, cEmbarque):
25                 result.append(Etat(
26                     etat.get_nbMg()+mEmbarque,
27                     etat.get_nbCg()+cEmbarque,
28                     not etat.get_boatPosition(),
29                     etat,
30                     etat.get_cout()+1
31                 ))
32             else:
33                 if rule(etat, n, -mEmbarque, -cEmbarque):
34                     result.append(Etat(
35                         etat.get_nbMg()-mEmbarque,
36                         etat.get_nbCg()-cEmbarque,
37                         not etat.get_boatPosition(),
38                         etat,
39                         etat.get_cout()+1
40                     ))
41
42         return result
43
44     # solution prend en parametre l'etat finale trouver par l'algo
. Il retournera un liste de tous les parents de la solution. (
La racine a la variable parent def a None)
45
46
47     def solution(etat):
48         soluce = []
49         soluce.append(etat)
50         etatC = etat
51         while not etatC.get_parent() == None:
52             etatC = etatC.get_parent()
53             soluce.append(etatC)
54         return soluce
55
56     # Application de l'algorithme graph-Search
57
58
59     def graph_Search(etat_Initiale, p, n):
60         frontiere = []
61         explore = []
62         # Nous allons simuler une file, nous allons donc utiliser
append qui rajoute l'objet en fin de file et l'etat choisie a
expanser sera celui en t^te de file (donc a la position 0)
63         frontiere.append(etat_Initiale)
64         etat_Final = Etat(0, 0, False, None, None)

```

```

65
66     while True:
67
68         if frontiere == []:
69             return None
70         elif frontiere[0] == etat_Final:
71             return solution(frontiere[0])
72         else:
73             S = expance(frontiere[0], p, n)
74             for si in S:
75                 frontiere.append(si)
76                 explore.append(frontiere.pop(0))
77                 # Supprime les etats dans frontiere qui sont
present dans explorer (donc les etas deja expance)
78                 for etat in explore:
79                     # la fonction remove retourne une erreur
lorsqu'elle ne trouve pas l'element a supprimer dans la file.
On va donc capter cette erreur pour eviter de faire "while etat
in frontiere". Ce qui nous obligera a chaque fois de
parcourir la file pour s'avoir s'il y a un etat correspondant
a "etat".
80
81                     while True:
82                         try:
83                             frontiere.remove(etat)
84                         except:
85                             break
86
87     def main():
88         n = int(input(
89             "Veillez renseigner le nombre n de missionnaires |
cannibale a faires traverser (minimum 3): "))
90         p = int(input(
91             "Veillez renseigner le nombre maximal de personne
pouvant monter sur le bateau (au minimum 2) : "))
92         if (p < 2 or n < 3):
93             print("veillez rentrer une valeur correcte.")
94         else:
95             solution = graph_Search(Etat(n, n, True, None, 0), p,
n)
96             if (solution != None):
97                 solution.reverse()
98                 for etat in solution:
99                     print("etat solution:", str(etat.get_nbMg()),
"M", str(
100                         etat.get_nbCg()), "C", ("droite ", "gauche
") [etat.get_boatPosition()], str(etat.get_cout()))
101             else:
102                 print("Aucune solution trouve")

```

103
104
105

```
main()
```

Listing A.2 – Python example