Compte rendu : Mercenaire et Cannibale

Flandin Léo

7 février 2023

Table des matières

Chapitre 1

Étude théorique du cas générale

1.1 Étude de l'environnement

1.2 Descrition d'un état

Ce projet est disponible sur github au lien suivant : `https://github.com/worlcrafte/probleme-cannibale-et-missionnaire` . Dans cette revue, nous allons nous pencher sur le problème des mercenaires et des cannibales généralisé à n ; où n est le nombre de cannibales et de mercenaires. Notre objectif sera de trouver le chemin de coût minimal pour ammener tous les mercenaires et cannibales de l'autre côté de la rive tout en respectant les contions du problème. Pour cela, nous allons commencer par décrire un état. Un état sera défini par les 3 composantes suivantes :

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

Nous n'avons pas besoins de stocké l'information sur le nombre de mercenaires ou de cannibales à droite celle-ci pouvant être facilement obtenu : n-nbMg pour le nombre de mercenaire à droite et n-nbCg pour les cannibales. Puis nous devons définir l'état initiale et l'état final. L'état initiale est définie comme suit : nbCg=n, nbMg=n et la barque est sur la rive gauche. Enfin l'état final : nbCg=0, nbMg=0 et la barque et sur la rive droite. Dans ce problème nous définiront une action comme étant la suivante : au moins une personne à traversé la rive. Notre fonction de coût sera basé sur l'action. C'est à dire, à chaque fois que la barque change de rive on ajoute 1 au coût.

1.3 Nombre d'état maximum

Nous nous intéressons maintenant au nombre d'état maximale que nous pouvons avoir pour savoir si c'est raisonnable de le représenter dans la mémoire. En respectant les règles défini on peut obtenir les états suivant si on ne prend pas compte de la position du bateau :

(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,0)	~~(n,1)~~	~~(n,2)~~	~~(n,3)~~	...	(n,n)

Nous aurons donc au maximum $(3n + 1) \times 2$ états (la ligne de (0,1) à (0,n) avec n états + la colone de (1,0) à (n,0) avec n états + la diagonale de (1,1) à (n,n) avec n états + l'état (0,0)).On réalise $\times 2$ 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 mercenaires est égale au nombre de cannibales. Sinon nous pourrions avoir plus d'état.

1.4 Réponse au question 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 se 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 choisi l'état (3M,3C,G) on revient à l'état initiale ce qui n'est pas intéressant. Si on choisie l'autre état, il faudra l'expancer pour pouvoir continuer la construction de notre arbre. Nous pouvons donc en conclure qu'il faut obligatoirement garder en mémoire les états déjà expenser pour ne pas se retrouver à expencer 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).

1.5 Choix de l'algorithme de recherche et de la stratégie abordé

A traver les différentes informations ques 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 un stratégie de recherche non iformé. Nous n'exploitons pas d'informations supplémentaire pour expoiter une fonction d'évaluation. De plus, nous avons dit plus tôt qu'il nous fallait garder les ancient états déjà expancé 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 optimal. Nous avons défini notre fonction de coût de +1 pour chaque passage la barque entre chaque rive. Nous utilisons donc des coûts utiformes. Cela rend donc inutile la stratégie de coût uniforme (cela revient à faire une stratégie en largeur). Enfin on on 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)→ (0,2,D) → (3,2,G) → (0,1,D) → (3,1,G) → (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)→(3,0,D)→(3,1,G)→(0,0,D)). Nous devons donc choisir l'algorithme Graph-Search avec une stratégie de parcourt en largeur. Celui-ci est optimale dans notre situation et comme complexité O(b^d) où b est le facteur de branchement et d la profondeur de la première solution.

Chapitre 2

Étude expérimentale des performance de l'algorithme

2.1 étude sur le nombre de personne maximale sur la barque

On pourras remarquer que pour un capacité maximum de 4 personnes sur la barque, nous trouverons toujours une solution. En effet, nous pouvons amener 2 mercenaires et 2 cannibales de l'autre côté de la rive puis n'en ramener que 1 de chaque. Ce qui assure de tours trouver une solution lorsque le nombre de mercenaire et de cannibale à faire passer sont égaux. De plus, selon Martin Gardner, lorsque que nous avons le même nombre de mercenaire et de cannibale 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 observé à travers ce tableau le temps réaliser pour trouvé la solution où n est le nombre de mercecnaire ou cannibales et p la capacité maximale de la barque.

TABLE 2.1 – Tableau de l'étude des performance de l'algorithme sur le temps pour un p fixé à 4.

n	**p**	**coût**	**temps (en seconde)**
5	4	7	0.0011
15	4	27	0.0100
25	4	47	0.0141
35	4	67	0.0193
45	4	87	0.0318
55	4	107	0.0422
65	4	127	0.0625
75	4	147	0.0775
85	4	167	0.0843
95	4	187	0.1112
105	4	207	0.1383
115	4	227	0.1315
125	4	247	0.1521
135	4	267	0.1776
145	4	287	0.1979
155	4	307	0.2315
165	4	327	0.2483
175	4	347	0.2779
185	4	367	0.3102
195	4	387	0.3399
205	4	407	0.3779
215	4	427	0.4072
225	4	447	0.4617
235	4	467	0.4860

n	**p**	**coût**	**temps (en seconde)**
10	4	17	0.0055
20	4	37	0.0162
30	4	57	0.0152
40	4	77	0.0274
50	4	97	0.0364
60	4	117	0.0833
70	4	137	0.0619
80	4	157	0.0825
90	4	177	0.0866
100	4	197	0.1216
110	4	217	0.1233
120	4	237	0.1400
130	4	257	0.1780
140	4	277	0.1836
150	4	297	0.2137
160	4	317	0.2401
170	4	337	0.2847
180	4	357	0.2966
190	4	377	0.3247
200	4	397	0.3649
210	4	417	0.3928
220	4	437	0.4255
230	4	457	0.4638
240	4	477	0.5105

Chapitre 3

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 superieur ou égale à nbCg. De plus nommes aussi en droit de se demander s'il existe un algorithme plus performant ou non pour réaliser ce problème jouet lorsque nbCg != nbMg.

Annexe A

Annexe

A.1 Etat.py

```
 class Etat:

 def __init__(self, nbMg, nbCg, boatPosition, parent, cout):
     self.nbMg = nbMg
     self.nbCg = nbCg
     # La position de la barque sera defini par un booleen. Si sa valeur est egale a vraie alors la barque est a guache sinon elle est a droite
     self.boatPosition = boatPosition
     self.parent = parent
     self.cout = cout

 def get_nbMg(self):
     return self.nbMg

 def get_nbCg(self):
     return self.nbCg

 def get_boatPosition(self):
     return self.boatPosition

 def get_parent(self):
     return self.parent

 def get_cout(self):
     return self.cout

 # __eq__ (methode "dunder/magique") permet de redefinir la fonction ==. Cela nous sera utile pour verifie si 2 etats sont
```

```
egaux. (fonctionne avec .remove() pour la comparaison d'etat)
    # il est possible de redefinir pour >, <, ... mais inutile pour notre cas.

    def __eq__(self, other):
        # on verifie que les 2 objets sont de la meme classes.
        if isinstance(other, Etat):
            return self.nbCg == other.nbCg and self.nbMg == other.nbMg and self.boatPosition == other.boatPosition
```

Listing A.1 – Python example

A.2 algoDeResolution.py

```
from Etat import Etat

# regle permet de verifier que l'etat calculer respecte bien les regles etablis : qu'il n'y ai pas plus de Canibale que de mercenaire sur un cote de la rive (sauf s'il n'y a aucun mercenaire alors il n'y a aucun risque pour eux)


def rule(etat, n, ajout, ajout2):
    if 0 <= etat.get_nbMg()+ajout <= n and 0 <= etat.get_nbCg()+ajout2 <= n:
        return ((((etat.get_nbMg() + ajout) >= (etat.get_nbCg() + ajout2)) or
                  (etat.get_nbMg()+ajout) == 0) and
                 (((n-(etat.get_nbMg()+ajout)) == 0 or
                   ((n-(etat.get_nbMg()+ajout)) >= (n-(etat.get_nbCg()+ajout2))))))
    else:
        return False

# expance permet d'expancer l'etat courant. Il prend en entree l'etat courant, la capacite maximale du tableau, et le nombre total de mercerniares et cannibales (ici uniquement n car c 2 valeurs sont egaux ). Il retournera une liste contenant tous les etats trouver par le programme respectant les regles etablis.


def expance(etat, p, n):
    result = []
    for mEmbarque in range(0, p+1):
        for cEmbarque in range((0, 1)[mEmbarque == 0], (1, p-mEmbarque+1)[p-mEmbarque+1 > 0]):
```

```
             if (not etat.get_boatPosition()):
                 # il es possible aussi de creer 2 variables temporaires et de modifier les informations contenus (+ ou - mais cela revient au meme)
                 if rule(etat, n, mEmbarque, cEmbarque):
                     result.append(Etat(
                         etat.get_nbMg()+mEmbarque,
                         etat.get_nbCg()+cEmbarque,
                         not etat.get_boatPosition(),
                         etat,
                         etat.get_cout()+1
                     ))
             else:
                 if rule(etat, n, -mEmbarque, -cEmbarque):
                     result.append(Etat(
                         etat.get_nbMg()-mEmbarque,
                         etat.get_nbCg()-cEmbarque,
                         not etat.get_boatPosition(),
                         etat,
                         etat.get_cout()+1
                     ))

     return result

 # 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)


 def solution(etat):
     soluce = []
     soluce.append(etat)
     etatC = etat
     while not etatC.get_parent() == None:
         etatC = etatC.get_parent()
         soluce.append(etatC)
     return soluce

 # Application de l'algorithme graph-Search


 def graph_Search(etat_Initiale, p, n):
     frontiere = []
     explore = []
     # Nous allons simuler une file, nous allons donc utiliser append qui rajoute l'objet en fin de file et l'etat choisie a expancer sera celui en t^te de file (donc a la position 0)
     frontiere.append(etat_Initiale)
     etat_Final = Etat(0, 0, False, None, None)
```

```

    while True:

        if frontiere == []:
            return None
        elif frontiere[0] == etat_Final:
            return solution(frontiere[0])
        else:
            S = expance(frontiere[0], p, n)
            for si in S:
                frontiere.append(si)
            explore.append(frontiere.pop(0))
            # Supprime les etats dans frontiere qui sont present dans eplorer (donc les etas deja expance)
            for etat in explore:
                # 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 obligerais a chaque fois de parcourire la file pour s'avoir s'il y a un etat correspondant a "etat".
                while True:
                    try:
                        frontiere.remove(etat)
                    except:
                        break


def main():
    n = int(input(
        "Veillez renseigner le nombre n de missionnaires|cannibale a faires traverser (minimum 3): "))
    p = int(input(
        "Veillez renseigner le nombre maximal de personne pouvant monter sur le bateau (au mininimum 2) : "))
    if (p < 2 or n < 3):
        print("veillez rentrer une valeur correcte.")
    else:
        solution = graph_Search(Etat(n, n, True, None, 0), p, n)
        if (solution != None):
            solution.reverse()
            for etat in solution:
                print("etat solution:", str(etat.get_nbMg()), "M", str(
                    etat.get_nbCg()), "C", ("droite ", "gauche ")[etat.get_boatPosition()], str(etat.get_cout()))
        else:
            print("Aucune solution trouve")
```

```


    main()
```

Listing A.2 – Python example