Université d'Evry (2022/2023), **IA**, 1–8

MINI PROJET IA

L3 Informatique (CILS)

Camille Berthaud - 20202238

Projet encadré par Christophe Janodet

Abstract

n missionnaires et n cannibales sont sur la rive d'un fleuve (avec n>=3). Ils doivent tous passer de l'autre côté. Ils disposent d'une barque qui peut porter p>=2 personnes (en comptant le rameur). Mais le nombre de missionnaires d'un côté ou de l'autre du fleuve ne peut jamais être strictement inférieur au nombre de cannibales.

Avant de poser un algorithme de recherche analysons le problème de recherche.

1. Analyse théorique

- But : transporter tous les missionnaires et les cannibales de la rive gauche vers la rive droite
- Etat : On décrit 4 variables (valeurs possibles pour l'état initial)
  1. Nombre de Missionnaires sur la rive gauche : $Mg \in \{3, ..., n\}$
  2. Nombre de Cannibales sur la rive gauche : $Cg \in \{3, .., n\}$
  3. Nombre de Missionnaires sur la rive droite : $Md \in \{3, .., n\}$
  4. Nombre de Cannibales sur la rive droite : $Cd \in \{3, .., n\}$

  Nous pouvons représenter un état sous la forme d'un quadruplet tel que : (Mg, Cg, Md, Cd)

  Remarque :
  Cette configuration facilite la compréhension de l'affichage du résultat lors de la phase de programmation

  On pose :
  - L'état initial : $(n, n, 0, 0)$
  - L'état final : $(0, 0, n, n)$
- Actions: correspond à un changement de rive
  - Quand la barque est à gauche, elle se déplace vers la droite.
  - Quand la barque est à droite, elle se déplace vers la gauche.

- Contraintes:
  - La barque peut transporter au maximum p personnes (avec p>=2)
  - Il ne faut jamais que sur l'une des deux rives, il y ait un nombre de cannibales strictement supérieur au nombre de missionnaires.

  Il est important de noter que chaque action est réalisée si elle vérifie chacune des contraintes.

2. Algorithme

Pour résoudre ce problème non-informé, nous allons modéliser l'espace des états sous la forme d'un graphe. L'algorithme Graph-Search permet de choisir à partir des états visités, le sommet suivant à visiter. De plus nous utiliserons comme algorithme de recherche Graph-Search ; la recherche en largeur (Breadth-first search).

L'algorithme de recherche en largeur (BFS) peut être utilisé pour trouver la solution de ce problème, car :

- Facile à implémenter et à comprendre
- Garantit de trouver la solution optimale, si elle existe
- La complexité de l'algo est proportionnelle à la taille du graphe, permet donc de résoudre des problèmes de taille modérée

Remarque : Dans le cas d'une recherche en largeur (BFS), le coût de chaque action peut être défini comme étant égale à 1, toutes les actions ont le même coût.

Considérons les variables suivantes :

- b : le nombre max de successeurs à un noeud (facteur de branchement)
- d : profondeur du noeud but le moins profond
- m : longueur max d'un chemin dans l'espace d'états

Voici un tableau résumant les caractéristiques de l'algorithme en largeur d'abord (BFS) :

Complétude	oui si b est fini
Optimalité	oui car le coût de chaque action est égale à 1
Complexité en temps	le nombre de noeuds générés pendant la recherche : $O(b^d)$
Complexité en espace	le nombre max de noeuds en mémoire : $O(b^d)$

Si nous testons l'algorithme pour n=3 (nombre de missionnaires et cannibales) et p=2 (taille maximale du nombre de personnes sur la barque). Voici le résultat que nous obtenons :

```
---
M_g=3, C_g=3|M_d=0, C_d=0
--> M:0 C:2
<-- M:0 C:1
---
M_g=3, C_g=2|M_d=0, C_d=1
--> M:0 C:2
<-- M:0 C:1
---
M_g=3, C_g=1|M_d=0, C_d=2
--> M:2 C:0
<-- M:1 C:1
---
M_g=2, C_g=2|M_d=1, C_d=1
--> M:2 C:0
<-- M:0 C:1
---
M_g=0, C_g=3|M_d=3, C_d=0
--> M:0 C:2
<-- M:0 C:1
---
M_g=0, C_g=2|M_d=3, C_d=1
--> M:0 C:2
<-- M:0 C:0
---
M_g=0, C_g=0|M_d=3, C_d=3
Nombre détats visités: 9
Temps d'exécution : 0.007ms
```

Remarques :

1. La solution ci-dessus est la plus optimale mais n'est pas forcément unique
2. Il n'existe pas une solution pour toute entrée *(ex : n=5 et p=2)*
3. Plus on augmente n plus le nombre d'états parcourus et le temps d'exécution sont importants

3. Extensions possibles

Afin d'étendre l'action de la méthode bfs(), il serait intéressant de rendre la classe Graph générique, afin qu'elle puisse être utilisée pour d'autres types de problème et d'inclure d'autres algorithmes de recherches comme le parcours en profondeur (Depth-First in Search).

4. Annexes

```python
from collections import deque
import time

# la classe Node represente un etat du problme


class Node:
    toLeft = ''
    toRight = ''

    # constructeur
```

```
    def __init__(self, C_g, M_g, C_d, M_d):
        # C_g -> cannibales sur la rive gauche
        # M_g -> missionnaires sur la rive gauche
        # C_d -> cannibales sur la rive droite
        # M_d -> missionnaires sur la rive droite
        self.C_g = C_g
        self.M_g = M_g
        self.C_d = C_d
        self.M_d = M_d

    # methodes permettant d'utiliser Node dans des structures de
donnees (ici: dictionnaire et ensemble)

    def __eq__(self, other):
        return self.C_g == other.C_g and self.M_g == other.M_g
and self.C_d == other.C_d and self.M_d == other.M_d

    # permet de determiner si 2 noeuds sont equivalents avec leur
 table de hachage
    def __hash__(self):
        return hash((self.C_g, self.M_g, self.C_d, self.M_d))

    def __str__(self):
        return f'M_g={self.M_g},␣C_g={self.C_g}|M_d={self.M_d},␣
C_d={self.C_d}'

    # methode permettant de definir si un etat est valide
    # contrainte principale : le nombre de Missionnaires ne doit
jamais tre strictement inferieure au nombre de Cannibales
    def is_valid(self):
        return (self.M_g == 0 or self.M_g >= self.C_g) and (self.
M_d == 0 or self.M_d >= self.C_d)


# la classe Graph represente le graphe de recherche
class Graph:
    visited = {}  # noeuds dej traites
    queued = {}  # noeuds  traiter
    states = []  # sous-noeuds  traiter avant d'tre empiles dans
queued
    parent = {}  # dictionnaire qui indexe les parents de chaque
noeud

    def __init__(self, start, end, maxp_boats):
        self.start = start
```

```
        self.end = end
        self.maxp_boats = maxp_boats

    # la methode get_next_states retourne les prochains etats
partir d'un etat donne
    def get_next_states(self, root):
        prequeued = {}
        queued = {}

    # calculer les etats   passer  avec p max passagers pour
aller sur le cte droit
    # prendre 0  p missionnaires dans la barque
        for M in range(self.maxp_boats+1):

            # si le nombre de missionnaires n'est pas suffisant
sur le cte gauche => inutile d'aller plus loin
            if (M > root.M_g):
                break

            # remplir la barque de 0  p cannibales du cte droit
            for C in range(self.maxp_boats+1):
                # si personne ne revient de l'autre cte, il n'est
 pas necessaire d'aller plus loin, sauf si l'etat final est
atteint.
                if (M == 0 and C == 0 and (root.M_g != 0 or root.
C_g != 0)):
                    continue

                # si le nombre de cannibales n'est pas suffisant,
 pas besoin d'aller plus loin
                if (C > root.C_g):
                    break

                # verifier que l'on a suffisamment de place sur
la barque
                if ((M+C) > self.maxp_boats):
                    break

                # creer un nouvel etat quand la barque est sur le
 cte droite
                node = Node(root.C_g-C, root.M_g-M, root.C_d+C,
root.M_d+M)
                node.toRight = f'-->␣M:{M}␣C:{C}'
```

```
                # si l'etat n'est pas valide, pas besoin d'aller
plus loin
                if node.is_valid() != True:
                    continue

                # si l'etat n'est pas dej enregistre, on le met
dans la liste d'attente
                if node not in prequeued:
                    prequeued[node] = node

        # calcule les etats  passer en retournant de nouveau sur
le cte gauche.
        for node in prequeued:
            for M in range(self.maxp_boats+1):
                if (M > node.M_d):
                    break

                for C in range(self.maxp_boats+1):
                    if (M == 0 and C == 0 and (node.M_g != 0 or
node.C_g != 0)):
                        continue
                    if (C > node.C_d):
                        break
                    if ((node.C_d-C) == 0 and (node.M_d-M) == 0):
                        continue
                    if ((M+C) > self.maxp_boats):
                        break
                    newNode = Node(node.C_g+C, node.M_g+M,
                                   node.C_d-C, node.M_d-M)
                    newNode.toRight = node.toRight
                    newNode.toLeft = f'<--␣M:{M}␣C:{C}'

                    if newNode.is_valid() != True:
                        continue

                    if newNode not in queued:
                        queued[newNode] = newNode

        # les etats enregistres sont ajoutes dans le tableau
states

        return queued.values()

    def bfs(self):
        self.queued.clear()
```

```
        self.visited.clear()
        self.parent.clear()

        self.queued = deque([self.start])  # on part du noeud
initial
        self.parent = {self.start: None}  # on le reference sans
parent

        while self.queued:
            state = self.queued.popleft()
            self.visited[state] = state
            if state == self.end:
                # pour mettre  jour les deplacements
                self.end = state
                return self.end

            for next_state in graph.get_next_states(state):
                if next_state not in self.visited:
                    self.visited[next_state] = next_state
                    self.parent[next_state] = state
                    self.queued.append(next_state)
        return None

    def get_solution(self):
        solutions = []
        parent = self.end

        while parent:  # tant que parent!=None
            solutions.append(parent)
            parent = self.parent[parent]

        solutions.reverse()
        return solutions


n = 3
root = Node(n, n, 0, 0)
end_root = Node(0, 0, n, n)
graph = Graph(root, end_root, 2)
start = time.time()

if graph.bfs():
    solutions = graph.get_solution()
    print("Solution:")
    for solution in solutions:
```

```
            print(solution.toRight)
            print(solution.toLeft)
            print('---')
            print(solution)

        print(f'Nombre␣detats␣visites:␣{len(graph.visited)}')
        end = time.time()
        elapsed = end - start
        print(f'Temps␣d\'execution␣:␣{elapsed:.2}ms')
    else:
        print("Pas␣de␣solution␣trouvee")
```