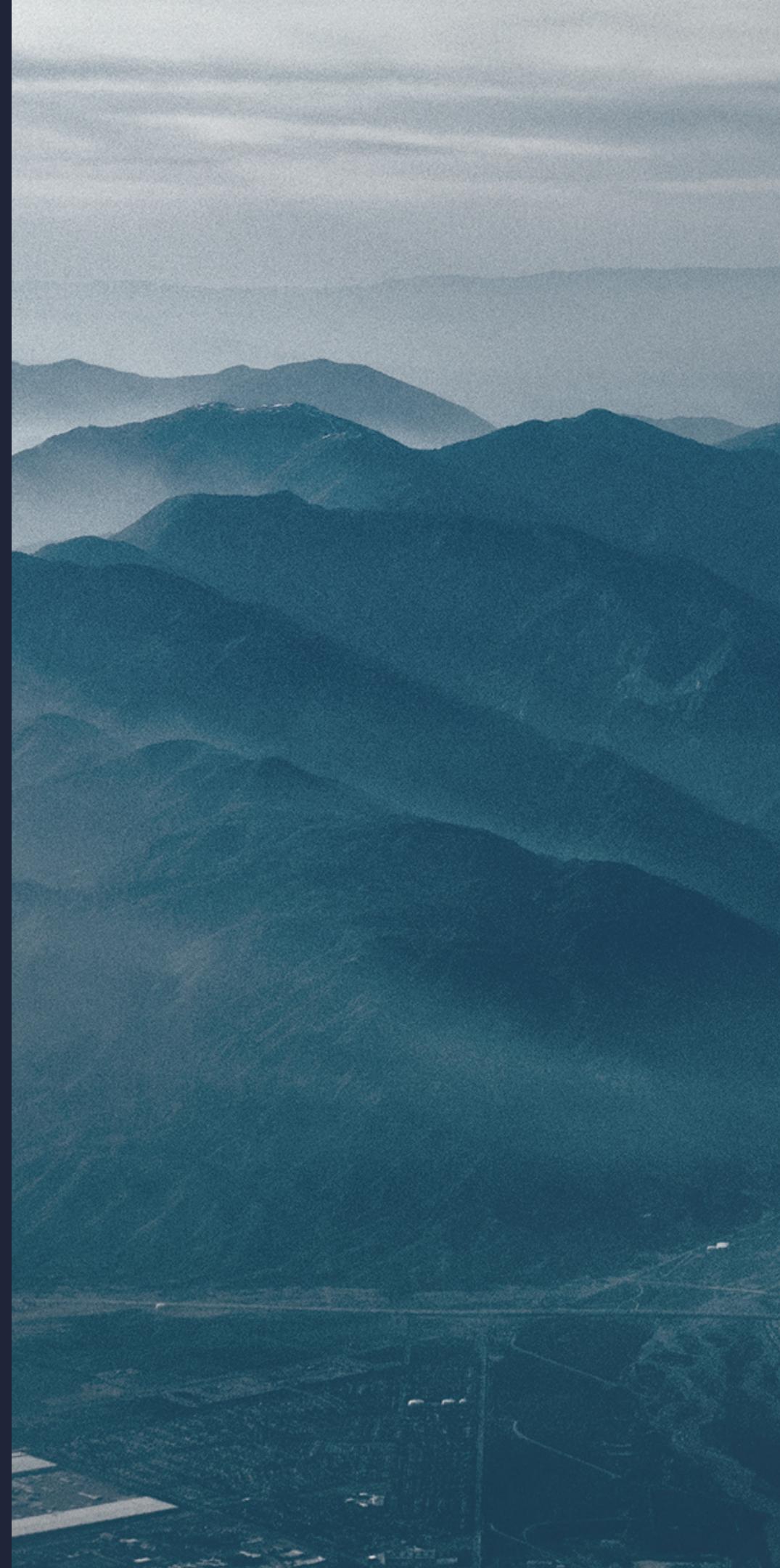


Exploration d'un algorithme

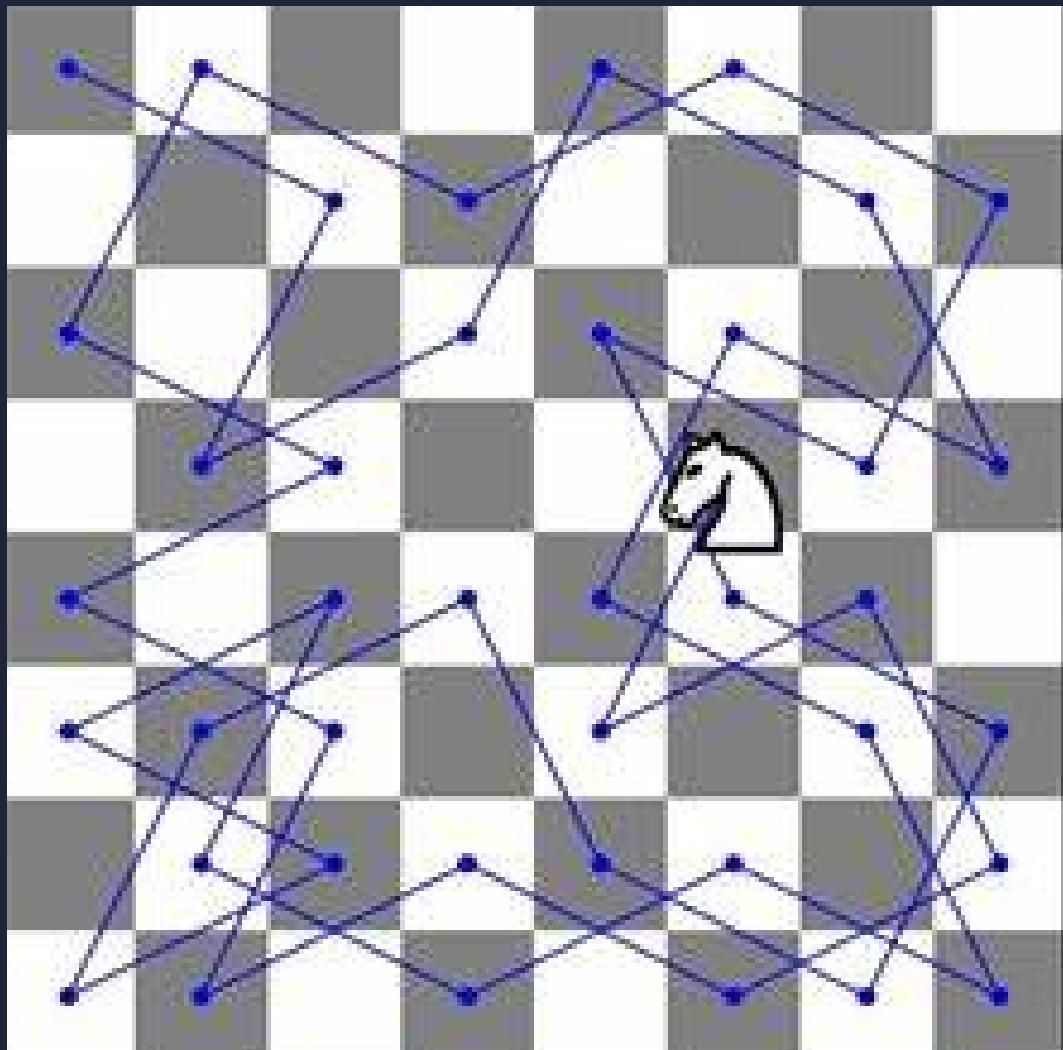
SAE 2.05

Sommaire :

- Présentation du problème
- Présentation de l'algorithme
- Description de l'algorithme
- Comparaison de l'algorithme



Présentation du problème :



- 1 Placer le cavalier
- 2 Voisins non visités
- 3 Retour en arrière
- 4 Retour arrière impossible
- 5 Fin du programme
Affichage

Description de l'algorithme 1:

- Demander la taille de l'échiquier et la case de départ
- Déplacements du cavalier
- Recherche parcours Hamiltonien
- Enregistrement des cases parcourues
- Afficher échiquier et ordre de passage
- Si le cavalier est bloqué, retour en arrière

```
import copy

def est_valide(x, y, dimensions, echiquier):
    """
    Vérifie si la position (x, y) est valide sur l'échiquier de dimensions spécifiées et si la case est vide.
    """
    return 0 <= x < dimensions and 0 <= y < dimensions and echiquier[x][y] == 'vide'

def trouver_parcours_hamiltonien(dimensions):
    """
    Trouve un parcours hamiltonien pour le cavalier sur un échiquier de dimensions spécifiées.
    """
    # Initialiser l'échiquier avec toutes les cases vides.
    echiquier = [['vide' for _ in range(dimensions)] for _ in range(dimensions)]

    # Demander la position initiale du cavalier à l'utilisateur.
    x = int(input("Entrez la position en x du cavalier : "))
    y = int(input("Entrez la position en y du cavalier : "))

    # Marquer la position initiale comme étant visitée.
    echiquier[x][y] = 'plein'

    # Initialiser le parcours avec la position initiale.
    parcours = [(x, y)]

    # Définir les déplacements possibles du cavalier en L.
    deplacements = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]

    # Fonction récursive pour trouver un parcours hamiltonien.
    def trouver_parcours_recuratif(x, y, etape_actuelle):
        # Si toutes les étapes ont été parcourues, on a trouvé un parcours hamiltonien.
        if etape_actuelle == dimensions**2:
            # Le plateau est en n * n cases donc n au carré
            return True

        # Essayer chaque déplacement possible.
        for dx, dy in deplacements:  # dx et dy pour 'déplacement x ou y'
            nx, ny = x + dx, y + dy  # nx et ny pour 'nouveau x ou y'
            if est_valide(nx, ny, dimensions, echiquier):
                # Marquer la case comme étant visitée et ajouter la position au parcours.
                echiquier[nx][ny] = 'plein'
                parcours.append((nx, ny))
                if trouver_parcours_recuratif(nx, ny, etape_actuelle+1):
                    return True
                # Si on n'a pas trouvé de parcours hamiltonien, enlever la dernière position du parcours et marquer la case comme vide.
                echiquier[nx][ny] = 'vide'
                parcours.pop()

        return False

    # Trouver un parcours hamiltonien à partir de la position initiale.
    if trouver_parcours_recuratif(x, y, 1):
        print("Parcours hamiltonien trouvé :")
        for i in range(dimensions):
            for j in range(dimensions):
                print(parcours.index((i,j))+1, end="\t")
            print("\n")
    else:
        print("Aucun parcours hamiltonien trouvé.")

    # Afficher l'ordre de parcours de chaque case sous forme de plateau.
    plateau = copy.deepcopy(echiquier)
    for i, (x, y) in enumerate(parcours):
        plateau[x][y] = str(i+1)

    print("Ordre de parcours de chaque case :")
    for ligne in plateau: print(ligne)

dimensions = int(input("Entrez la dimension de l'échiquier : "))
trouver_parcours_hamiltonien(dimensions)
```

Description de l'algorithme 2:

```
def graphe(n) :
    E = dict()
    for k in range(n*n) :
        i = k//n # ligne de la case k
        j = k%n # colonne de la case k

        E[k] = [] # E[k] : liste des voisins de la case k
        if 0 <= i-2 < n and 0 <= j-1 < n : E[k].append((i-2)*n+(j-1))
        if 0 <= i-2 < n and 0 <= j+1 < n : E[k].append((i-2)*n+(j+1))
        if 0 <= i+2 < n and 0 <= j-1 < n : E[k].append((i+2)*n+(j-1))
        if 0 <= i+2 < n and 0 <= j+1 < n : E[k].append((i+2)*n+(j+1))

        if 0 <= i-1 < n and 0 <= j-2 < n : E[k].append((i-1)*n+(j-2))
        if 0 <= i-1 < n and 0 <= j+2 < n : E[k].append((i-1)*n+(j+2))
        if 0 <= i+1 < n and 0 <= j-2 < n : E[k].append((i+1)*n+(j-2))
        if 0 <= i+1 < n and 0 <= j+2 < n : E[k].append((i+1)*n+(j+2))
    return E

def cavalierHamilton(n) :
    """ recherche d'un chemin hamiltonien dans le graphe du cavalier """
    E = graphe(n)
    chemin = [] # contiendra les cases dans leur ordre de visite

    def parcours( case ) :
        """
        case : case actuelle du cavalier.
        """
        chemin.append(case) # case est ajoutée au chemin, ce qui la marque comme visitée également

        if len(chemin) == n*n :
            gagne = True
        else :
            gagne = False
            voisins = [ u for u in E[case] if u not in chemin ] # voisins non visités de case
            for v in voisins :
                if gagne : break
                else : gagne = parcours( v )
            if not gagne :
                chemin.pop() # case est supprimée de chemin si elle a mené à une impasse

        return gagne

    # départ d'une case prise au hasard :
    parcours(int(input("Entrez le numéro de la case initiale : ")))
    return chemin

def affichage(n) :
    """ affichage simple de l'échiquier avec indication de l'ordre de parcours des cellules réalisées """
    t = [[0 for j in range(n)] for k in range(n)]
    chemin = cavalierHamilton(n)

    rg = 1
    for x in chemin :
        if rg > 9 : t[x//n][x%n] = str(rg)
        else : t[x//n][x%n] = '0' + str(rg)
        rg += 1

    for ligne in t :
        for c in ligne :
            print(c, end=" ")
        print()

n = int(input("Entrez les dimensions du plateau : "))
while n<5 and n>10 :
    n = int(input("La valeur doit être comprise entre 5 et 10 : "))
affichage(n)
```

- Création des coups possibles
- Demande de la taille de l'échiquier
- Recherche à l'aide du parcours Hamiltonien
- Chaque cases sont enregistrées les unes à la suite des autres
- Parcours de l'échiquier avec des possibles retour en arrière
- Affichage de l'échiquier et du parcours

Comparaison des algorithmes :

1

PRINCIPE D'EXPLORATION
DE L'ÉCHIQUIER

2

TEMPS D'EXECUTION DU
PROGRAMME

3

TAILLE DU PROGRAMME

4

AFFICHAGE

The background is a photograph of a dense forest of tall evergreen trees, likely Douglas firs, standing in a misty or foggy environment. The trees are dark green and have long, thin needles. The ground in the foreground is covered in a mix of green grass and brown, fallen leaves. The overall atmosphere is hazy and ethereal.

Bilan