FSBM

FACULTE DES SCIENCES BEN M'SICK
UNIVERSITE HASSAN II DE CASABLANCA

Algorithmes d’optimisation

Pr. Faouzia Benabbou (faouzia.benabbou@univh2c.ma)
Département de mathématiques et Informatique
Master Data Science & Big Data
2024-2025

Plan du Module: Algorithmes d'optimisation

Introduction

Rappels mathématiques

Algorithmes d'Optimisation Sans Contraintes

Algorithmes d'Optimisation avec Contraintes

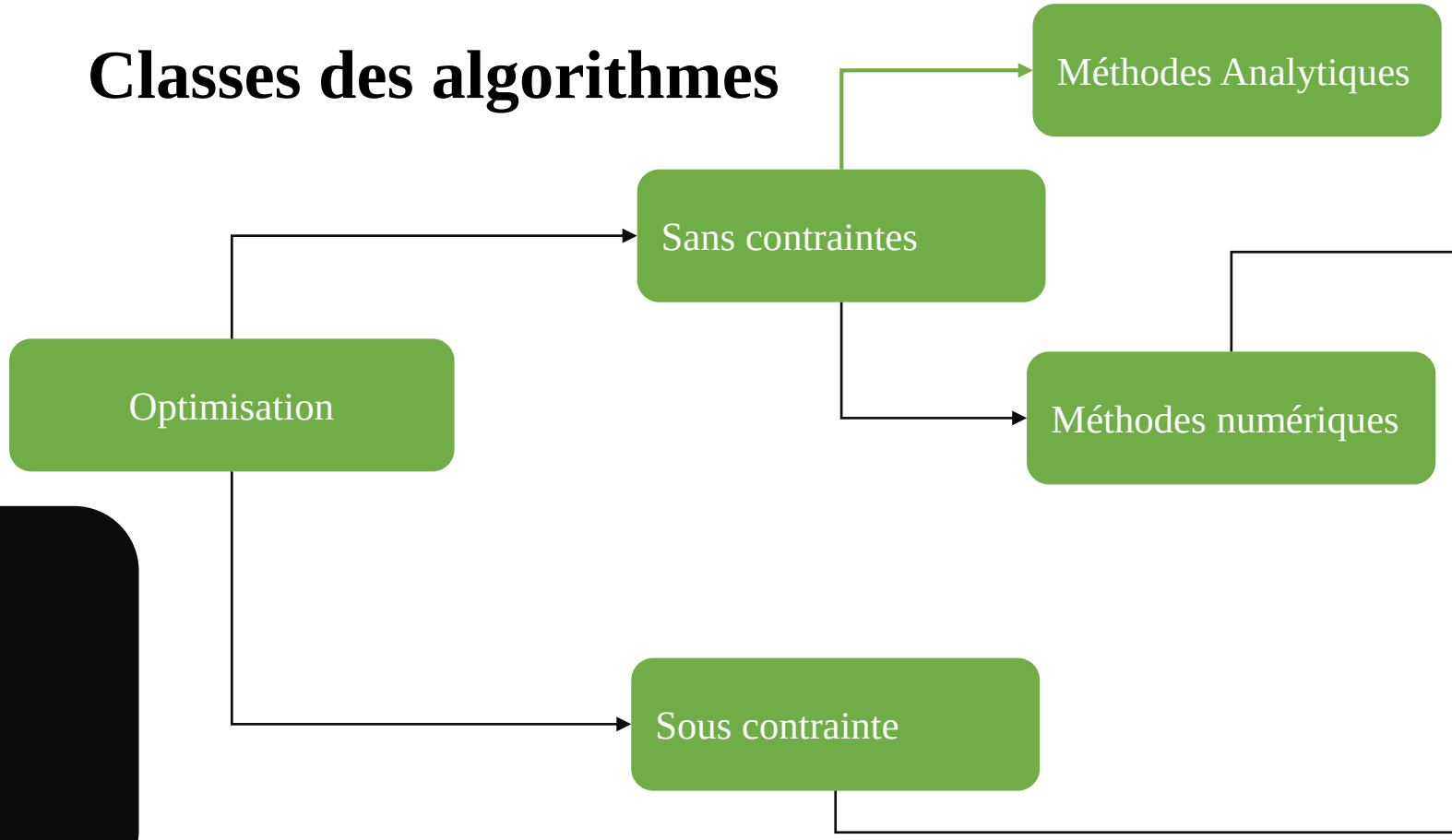
Programmation Linéaire

Méthodes Optimisation combinatoire exacte

Méthodes Optimisation combinatoire approchées

Les algorithmes d’optimisation

Classes des algorithmes



- Méthode sans gradient
- Méthode de descente de gradient:
  - ✓ Méthode de gradient à pas fixe et optimal
  - ✓ Méthode de gradient conjugué
  - ✓ Méthode de Newton
  - ✓ Méthodes du gradient quasi-newton

- Méthodes générales de résolutions de PCE, PCI
- Programmation linéaire
- Optimisation combinatoire
  - ✓ Méthodes Exactes
  - ✓ Méthodes approchées
- Algorithmes multi-objectifs

Optimisation combinatoire

- **Méthodes approchées.**
  - A la différence des méthodes exactes, les méthodes approchées permettent de trouver **une bonne solution** proche de l'optimal en un temps de calcul raisonnable.
  - Elles sont utilisées quand les méthodes **exactes sont trop coûteuses** (problèmes NP-difficiles).
  - Elles ne garantissent **pas l'optimalité**, mais elles permettent d'obtenir des solutions réalisables et souvent proches de l'optimum.
  - Les méthodes approchées se divisent en deux grandes familles :
    - ✓ Les heuristiques
    - ✓ Les métaheuristiques

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques**

Définition.

- Un algorithme heuristique est une méthode qui explore l'ensemble des solutions réalisables d'un problème d'optimisation, en exploitant **la structure du problème** afin d'identifier rapidement de bonnes solutions réalisables.
- L'algorithme utilise des connaissances spécifiques sur le problème (ses propriétés, ses contraintes) pour guider sa recherche de manière intelligente.
- Les **méthodes heuristiques** résolvent des problèmes spécifiques et fournissent une **solution acceptable dans un temps réduit**, sans garantie d'optimalité.

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques**
  - Elles peuvent être classées en deux grandes catégories :
    - ✓ **Heuristiques constructives (ou gloutonnes)** : elles construisent progressivement une solution en prenant à chaque étape la **meilleure décision locale**, dans l'espoir d'obtenir une bonne solution globale, **sans retour en arrière**.
      - Exemples: l'algorithme **glouton** et l'algorithme Dijkstra.
    - ✓ **Heuristiques d'amélioration locale** : elles partent d'une **solution initial,** générée aléatoirement ou par une méthode gloutonne, puis l'améliorent à l'aide de **modifications locales** successives.
      - **Exemples:** Hill Climbing, la recherche tabou (Tabu Search), et la recherche à voisinage variable (Variable Neighborhood Search – VNS).

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Algorithme glouton**
  - **Définition.** Une heuristique gloutonne désigne généralement une méthode qui construit une solution réalisable étape par étape, de manière à ce que chaque étape soit **localement optimale.**
  - Les heuristiques gloutonnes sont spécifique à un **problème particulier**.
  - **Exemple.** L'algorithme glouton pour le problème du voyageur de commerce consiste à partir de la ville d'origine et, à chaque étape, à sélectionner la ville la plus proche comme prochaine destination, jusqu'à visiter toutes les villes prévues.

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**

Algorithm 17. Nearest neighbor greedy

1.Intialisation

n: Nombre de villes

d(i, j), i = 1, . . . , n, j = 1, . . . , n, la distance entre deux villes

chemin_sol ← {1 } la séquence solution qu'on va construire

S← {2, . . . , n} : la liste des villes non encore visitées

ville_courante ← 1

2. Répéter

le cas où **plusieurs villes** ont la **même distance minimale** par rapport à la ville courante

choisir i $\underset{j \in S}{\text{argmin}}$ d(ville_courante , j).

ville_courante ← i.

chemin_sol ← chemin_sol ∪ {i}.

S← S\ {i}.

jusqu'à S= ∅.

3. Return chemin_sol.

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**
- **Exemple 1.**
  - On considère une instance du problème du voyageur de commerce (TSP: Traveling Salesperson Problem) avec 16 villes.
  - Un représentant doit rendre visite à 15 clients au cours de la journée.
  - En partant de son domicile, il doit planifier une tournée — c'est-à-dire une séquence de clients à visiter — de manière à minimiser la distance totale parcourue.
  - Ce problème est modélisé par un graphe, où chaque sommet représente soit le domicile, soit un client.
  - Une arête relie chaque paire de sommets, et le coût associé à chaque arête est la distance euclidienne directe entre les deux sommets.
  - La position du domicile (sommet 1) ainsi que celles des 15 clients sont données dans le tableau suivant et la distance est calculée par la distance euclidienne.

Optimisation combinatoire

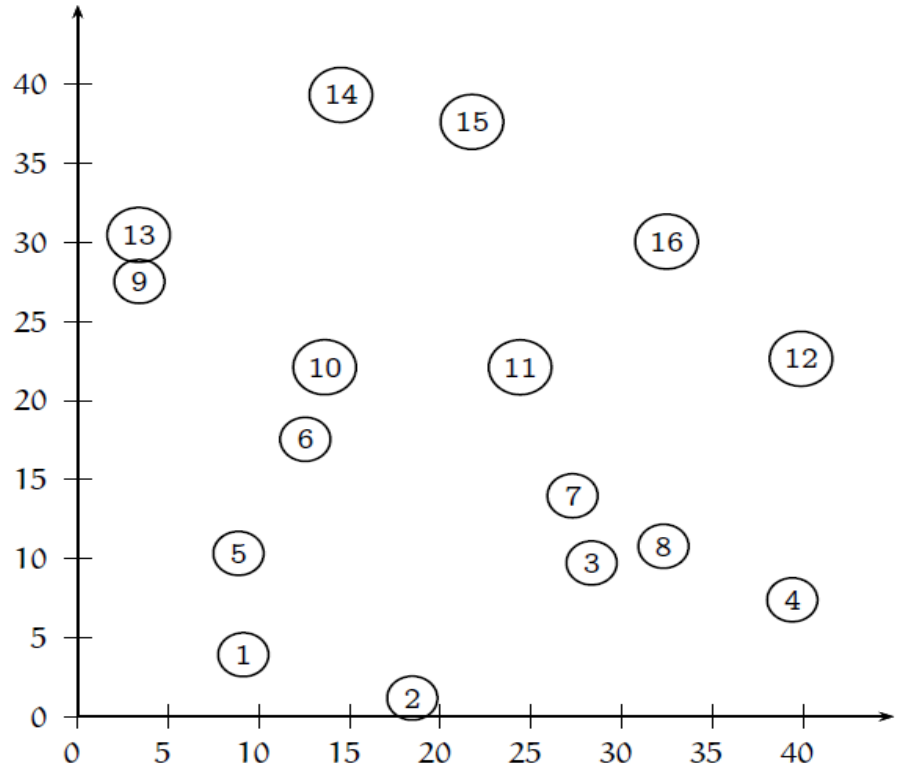
- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**
- **Exemple 1.**

```
coords = [
    (9.14, 3.92),
    (18.46, 1.17),
    (28.35, 9.72),
    (39.41, 7.39),
    (8.87, 10.33),
    (12.56, 17.55),
    (27.29, 13.97),
    (32.31, 10.78),
    (3.41, 27.54),
    (13.63, 22.11),
    (24.41, 22.10),
    (39.90, 22.64),
    (3.37, 30.48),
    (14.52, 39.34),
    (21.75, 37.64),
    (32.48, 30.06)
]
```

client	x-coord	y-coord	client	x-coord	y-coord
1	9.14	3.92	9	3.41	27.54
2	18.46	1.17	10	13.63	22.11
3	28.35	9.72	11	24.41	22.10
4	39.41	7.39	12	39.90	22.64
5	8.87	10.33	13	3.37	30.48
6	12.56	17.55	14	14.52	39.34
7	27.29	13.97	15	21.75	37.64
8	32.31	10.78	16	32.48	30.06

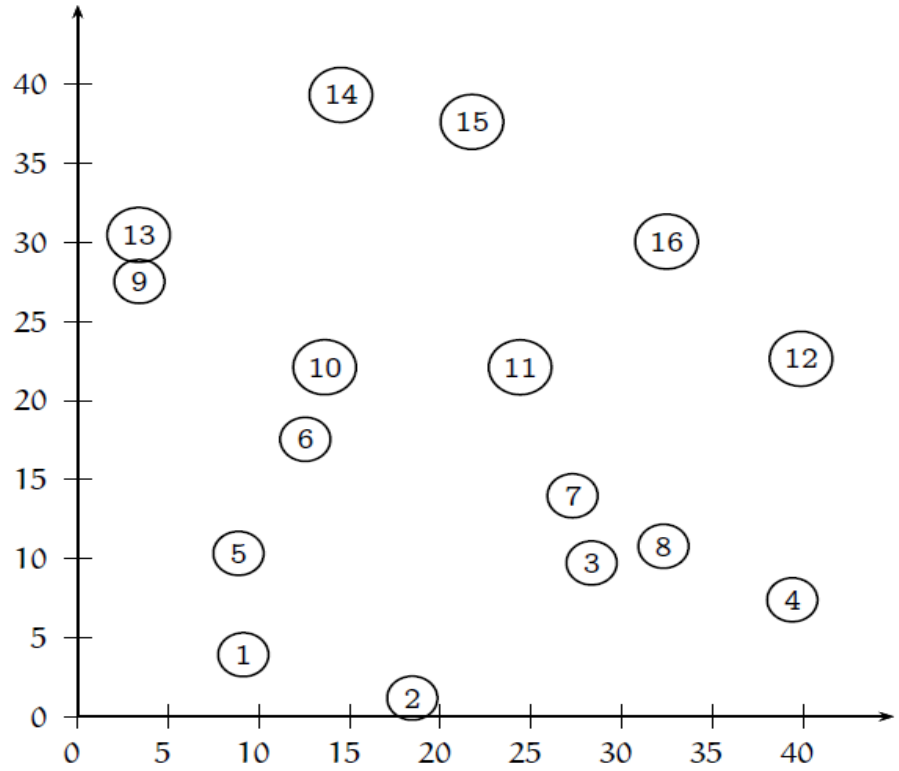
Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**
- **Exemple 1.**



Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**
- **Exemple 1.**



Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**
- **Exemple 1.**

```python
#algorithm glouton pour le voyageur de commerce
def nearest_neighbor_greedy(coords):
    distance_matrix = creer_distance_matrix(coords)
    n = len(coords)
    chemin_sol = [0]
    villes_non_visitées = set(range(1, n))
    ville_courante = 0

    while villes_non_visitées:
        distances = [(j, distance_matrix[ville_courante][j]) for j in villes_non_visitées]
        min_distance = min(distances, key=lambda x: x[1])[1]
        # on traite le cas où il y a égalité de distance
        candidats = [j for j, d in distances if d == min_distance]
        #on prend la première
        i = candidats[0]
        ville_courante = i
        chemin_sol.append(i)
        villes_non_visitées.remove(i)

    return chemin_sol, distance_matrix
```

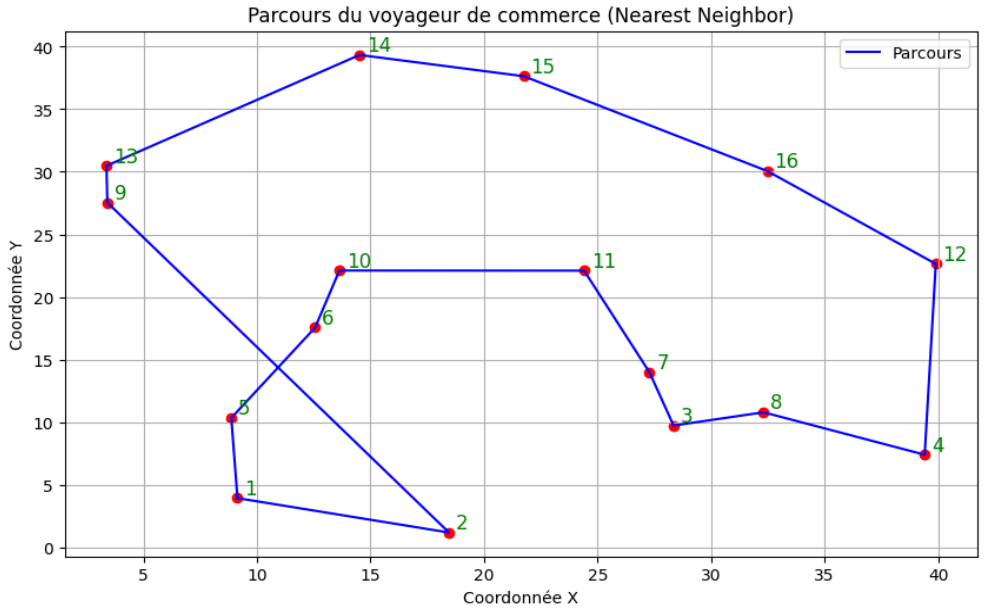
```python
#calcul du côut total
def total_cost(chemin, distance_matrix):
    cost = 0
    for k in range(len(chemin) - 1):
        cost += distance_matrix[chemin[k]][chemin[k+1]]
    # Pour revenir au point de départ et fermer le circuit (optionnel)
    cost += distance_matrix[chemin[-1]][chemin[0]]
    return cost
```

```python
# Exécution
chemin, distance_matrix = nearest_neighbor_greedy(coords)
cout = total_cost(chemin, distance_matrix)
```

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Recherche gloutonne**
- **Exemple 1.**

```
Ordre de visite des villes : [1, 5, 6, 10, 11, 7, 3, 8, 4, 12, 16, 15, 14, 13, 9, 2]
Coût total du parcours (distance totale) : 158.54
```



Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - Le principe d'une **recherche locale** est de partir d'une solution approchée, moins potentiellement bonne et d'essayer de l'améliorer itérativement.
  - L’algorithme de recherche locale compare la solution actuelle à un ensemble de solutions voisines et sélectionne celle qui est la plus prometteuse.
  - Il s'arrête généralement lorsqu'ils atteignent une solution jugée suffisamment bonne, ou lorsque toutes les solutions voisines ont été explorées sans en trouver une meilleure.

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - Les composants principaux de la recherche locale peuvent être définis par :
    - ✓ Un **espace de recherche** contenant toutes les solutions valides du problème proposé.
    - ✓ Une fonction de voisinage décrivant les **mouvements possibles** pour générer les voisins ou successeurs.
    - ✓ Une fonction heuristique qui évalue l**es solutions et** qui rend compte de la qualité des solutions.

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**

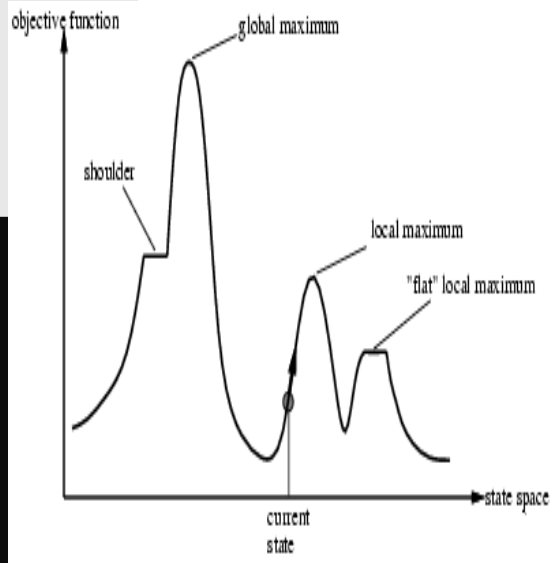
Algorithme 18. HillClimbing
Maximisation
1. initialisation :
f une fonction objective, **solution_initiale**,
générer_voisins une fonction qui retourne un ensemble de solutions voisines d'une solution donnée.
S_courante = Solution_Initiale
2. tant que (vrai)
a) List_voisins = générer_voisins(S_courante)
b) Si List_voisins =ϕ alors
return S_courante
sinon
meilleur_voisin←max_f(List_voisins)
Si f(meilleur_voisin) ≤ f(**S_courante)**
return **S_courante**
S_courante ←meilleur_voisin
Return **S_courante**

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Remarques.**
    - ✓ L’état initial peut être aléatoire ou choisi selon le problème.
    - ✓ La fonctions **générer_voisins** doit permettre d'explorer efficacement l'espace des solutions possibles et de trouver de bonnes solutions de manière efficace.
  - **Avantages**
    - ✓ L’intérêt de Hill climbing est sa capacité de trouver une solution acceptable rapidement.
    - ✓ Méthode simple

Optimisation combinatoire

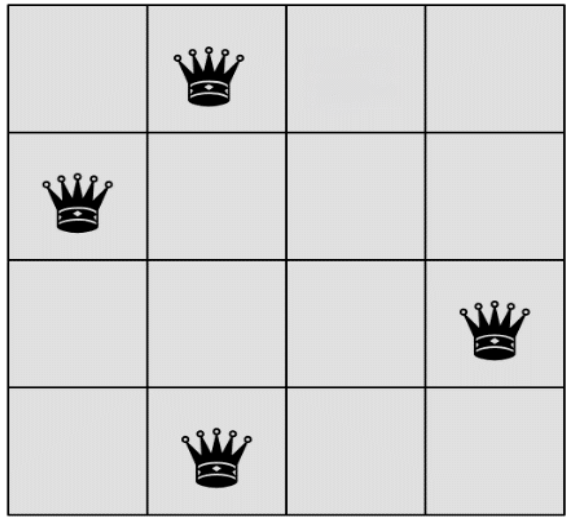
- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **limites.**

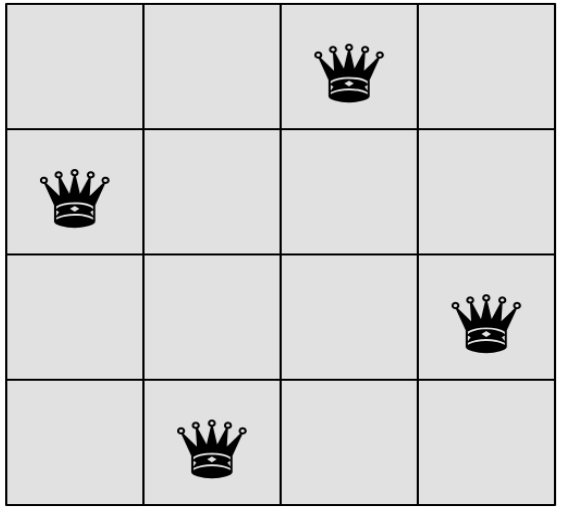


- ✓ La méthode choisi un bon état voisin sans réfléchir à l'état suivant
- ✓ Il y a grand risque d'être piégé dans des **optimums locaux**, c'est le cas où il atteint un nœud dont ses voisins immédiats sont moins bons, et il s'arrête.
- ✓ Il est cependant possible de remédier au problème du minimum local en choisissant un voisin de façon aléatoire tel que F(voisin) < F(courant), ce que fait la **version stochastique hill Climbing**.
- ✓ Cette méthode présente un autre désavantage qui est de devoir calculer f pour tous les voisins pour trouver le minimum.
- ✓ Sur des domaines très grands cela se traduira par des performances dégradées.

Optimisation combinatoire

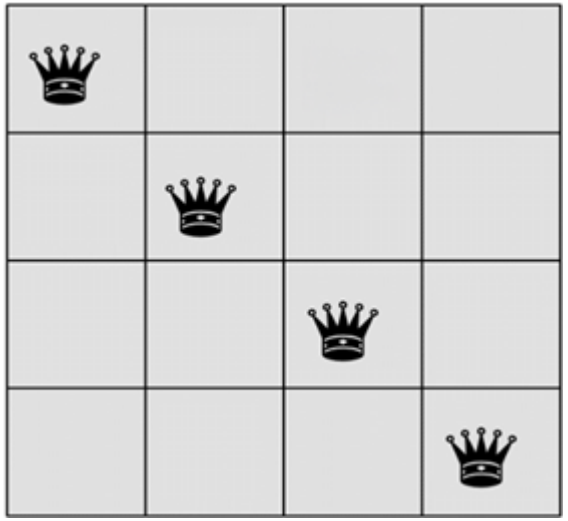
- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des n-reines. Les reines doivent être placées chacune dans une ligne sans que les reines ne s'attaquent entre elles.





Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des n-reines.
    - ✓ La **fonction objective**: le nombre de conflits entres reines qu'on veut minimiser.
    - ✓ La génération des voisins d'une solution (grille ou les reines sont placées) consiste à déplacer chaque reine dans sa ligne.
    - ✓ [0, 1, 3, 2] veut que la reine de la première ligne est placée dans la case 0, la reine de la deuxième ligne est placée dans la case 1, la reine de la troisième ligne est placée dans la case 3, et la dernière est placée dans la case 2.



Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

Itération 0.

État initial : [0,1,2,3],

Le nombre de conflits est 6.

Les Voisins possibles sont:

Reine à déplacer	Nouvelles colonnes possibles	Voisins générés
Reine 0 (col 0)	1, 2, 3	[1,1,2,3], [2,1,2,3], [3,1,2,3]
Reine 1 (col 1)	0, 2, 3	[0,0,2,3], [0,2,2,3], [0,3,2,3]
Reine 2 (col 2)	0, 1, 3	[0,1,0,3], [0,1,1,3], [0,1,3,3]
Reine 3 (col 3)	0, 1, 2	[0,1,2,0], [0,1,2,1], [0,1,2,2]

Le nombre de conflit pour chaque solution

Voisin	Description	Conflit
[1, 1, 2, 3]	Reine 0 → ligne 1	5
[2, 1, 2, 3]	Reine 0 → ligne 2	4
[3, 1, 2, 3]	Reine 0 → ligne 3	3
[0, 0, 2, 3]	Reine 1 → ligne 0	5
[0, 2, 2, 3]	Reine 1 → ligne 2	5
[0, 3, 2, 3]	Reine 1 → ligne 3	4
[0, 1, 0, 3]	Reine 2 → ligne 0	5
[0, 1, 1, 3]	Reine 2 → ligne 1	4
[0, 1, 3, 3]	Reine 2 → ligne 3	5
[0, 1, 2, 0]	Reine 3 → ligne 0	3
[0, 1, 2, 1]	Reine 3 → ligne 1	5
[0, 1, 2, 2]	Reine 3 → ligne 2	4

Solution courante = [3, 1, 2, 3] meilleur coût =3

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

Itération 1.

État courant : [3, 1, 2, 3]

Le nombre de conflits est 3.

Les Voisins possibles sont:

Reine à déplacer	Nouvelles colonnes possibles	Voisins générés
Reine 0 (col 0)	1, 2, 3	[1, 1, 2, 3], [2, 1, 2, 3], [3, 1, 2, 3]
Reine 1 (col 1)	0, 2, 3	[0, 0, 2, 3], [0, 2, 2, 3], [0, 3, 2, 3]
Reine 2 (col 2)	0, 1, 3	[0, 1, 0, 3], [0, 1, 1, 3] [0, 1, 3, 3]
Reine 3 (col 3)	0, 1, 2	[0, 1, 2, 0], [0, 1, 2, 1], [0, 1, 2, 2]

Voisin	Description	Conflits
[0, 1, 2, 3]	Reine 0 → ligne 0	6
[1, 1, 2, 3]	Reine 0 → ligne 1	5
[2, 1, 2, 3]	Reine 0 → ligne 2	4
[3, 0, 2, 3]	Reine 1 → ligne 0	4
[3, 2, 2, 3]	Reine 1 → ligne 2	4
[3, 3, 2, 3]	Reine 1 → ligne 3	4
[3, 1, 0, 3]	Reine 2 → ligne 0	4
[3, 1, 1, 3]	Reine 2 → ligne 1	4
[3, 1, 3, 3]	Reine 2 → ligne 3	4
[3, 1, 2, 0]	Reine 3 → ligne 0	4
[3, 1, 2, 1]	Reine 3 → ligne 1	4
[3, 1, 2, 2]	Reine 3 → ligne 2	4

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

Itération 1.

État courant : [3,1,2,3],

Le nombre de conflits est 3.

Aucun voisin n'a moins de **3 conflits** → **plateau local** à 4 atteint.

État final local : [3, 1, 2, 3]

Conflits = 3

L'algorithme est **converge vers un minimum local**.

Le nombre de conflit pour chaque solution

Voisin	Description	Conflits
[0, 1, 2, 3]	Reine 0 → ligne 0	6
[1, 1, 2, 3]	Reine 0 → ligne 1	5
[2, 1, 2, 3]	Reine 0 → ligne 2	4
[3, 1, 2, 3]	Reine 1 → ligne 0	4
[0, 2, 2, 3]	Reine 1 → ligne 2	4
[0, 3, 2, 3]	Reine 1 → ligne 3	4
[0, 1, 0, 3]	Reine 2 → ligne 0	4
[0, 1, 1, 3]	Reine 2 → ligne 1	4
[0, 1, 3, 3]	Reine 2 → ligne 3	4
[0, 1, 2, 0]	Reine 3 → ligne 0	4
[0, 1, 2, 1]	Reine 3 → ligne 1	4
[0, 1, 2, 2]	Reine 3 → ligne 2	4

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

Itération 0.

État initial : [2, 3, 3, 2]

Le nombre de conflits est 4.

Génération de voisins et calcul de nombre de conflits.

Solution courante : [0, 3, 3, 2],

Nombre de conflit=2.

Voisin	Description	Conflits
[0, 3, 3, 2]	Reine 0 → ligne 0	2
[1, 3, 3, 2]	Reine 0 → ligne 1	3
[3, 3, 3, 2]	Reine 0 → ligne 3	4
[2, 0, 3, 2]	Reine 1 → ligne 0	3
[2, 1, 3, 2]	Reine 1 → ligne 1	3
[2, 2, 3, 2]	Reine 1 → ligne 2	5
[2, 3, 0, 2]	Reine 2 → ligne 0	3
[2, 3, 1, 2]	Reine 2 → ligne 1	3
[2, 3, 2, 2]	Reine 2 → ligne 2	5
[2, 3, 3, 0]	Reine 3 → ligne 0	2
[2, 3, 3, 1]	Reine 3 → ligne 1	3
[2, 3, 3, 3]	Reine 3 → ligne 3	4

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

Itération 1.

État courant: [0, 3, 3, 2]

Le nombre de conflits est 2.

Génération de voisins et calcul de nombre de conflits.

Solution courante : [0, 3, 0, 2],

Nombre de conflit=1.

Voisin	Description	Conflits
[1, 3, 3, 2]	Reine 0 → ligne1	3
[2, 3, 3, 2]	Reine 0 → ligne 2	4
[3, 3, 3, 2]	Reine 0 → ligne 3	4
[0, 0, 3, 2]	Reine 1 → ligne 0	3
[0, 1, 3, 2]	Reine 1 → ligne 1	2
[0, 2, 3, 2]	Reine 1 → ligne 2	3
[0, 3, 0, 2]	Reine 2 → ligne 0	1
[0, 3, 1, 2]	Reine 2 → ligne 1	1
[0, 3, 2, 2]	Reine 2 → ligne 2	3
[0, 3, 3, 0]	Reine 3 → ligne 0	2
[0, 3, 3, 1]	Reine 3 → ligne 1	2
[0, 3, 3, 3]	Reine 3 → ligne 3	4

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

<u>Itération 2.</u>

État courant = [0, 3, 0, 2]

Conflits : 1

Génération de voisins et calcul de nombre de conflits.

Solution courante : [1, 3, 0, 2],

Nombre de conflit=0.

Minimum globale atteint à [1, 3, 0, 2].

Voisin	Description	Conflits
[1, 3, 0, 2]	Reine 0 → ligne1	0
[2, 3, 0, 2]	Reine 0 → ligne 2	3
[3, 3, 0, 2]	Reine 0 → ligne 3	1
[0, 0, 0, 2]	Reine 1 → ligne 0	4
[0, 1, 0, 2]	Reine 1 → ligne 1	3
[0, 2, 0, 2]	Reine 1 → ligne 2	2
[0, 3, 1, 2]	Reine 2 → ligne 1	1
[0, 3, 2, 2]	Reine 2 → ligne 2	3
[0, 3, 3, 2]	Reine 2 → ligne 3	2
[0, 3, 0, 0]	Reine 3 → ligne 0	3
[0, 3, 0, 1]	Reine 3 → ligne 1	3
[0, 3, 0, 3]	Reine 3 → ligne 3	3

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

```python
#fct pour calculer le nombre de conflits dans une solution (etat)
def compter_conflits(etat):
    conflits = 0
    n = len(etat)
    for i in range(n):
        for j in range(i+1, n):
            # Même ligne ou même diagonale ?
            if etat[i] == etat[j] or abs(etat[i] - etat[j]) == abs(i - j):
                conflits += 1
    return conflits
```

```python
#fct pour générer les voisins
def voisins(etat):
    liste_voisins = []
    n = len(etat)
    for col in range(n):
        for ligne in range(n):
            if ligne != etat[col]:
                voisin = list(etat)
                voisin[col] = ligne
                nb_conflits = compter_conflits(voisin)
                liste_voisins.append((voisin, nb_conflits))
    return liste_voisins
```

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

```python
def hill_climbing(etat_initial):
    S_courante = etat_initial
    conflits_actuel = compter_conflits(S_courante)
    iteration = 0
    print(f"État initial : {S_courante} avec {conflits_actuel} conflits\n")
    while conflits_actuel > 0:
        liste_voisins = voisins(S_courante)
        print(f"Itération {iteration} : État courant = {S_courante} (Conflits : {conflits_actuel})")
        print("Voisins et nombre de conflits :")
        for v, c in liste_voisins:
            print(f"  {v} -> {c}")
        # On choisit le voisin avec le moins de conflits
        liste_voisins.sort(key=lambda x: x[1])
        meilleur_voisin, conflits_min = liste_voisins[0]
        if conflits_min >= conflits_actuel:
            print("Aucun voisin meilleur trouvé, arrêt de l'algorithme.")
            break
        S_courante = meilleur_voisin
        conflits_actuel = conflits_min
        iteration += 1
        print()
    print(f"État final : {S_courante} avec {conflits_actuel} conflits")
    return S_courante, conflits_actuel
```

Optimisation combinatoire

- **Méthodes approchées. Algorithmes heuristiques. Hill Climbing**
  - **Exemple.** Le problème des 4-reines.

```
État initial : [2, 3, 3, 2] avec 4 conflits

Itération 0 : État courant = [2, 3, 3, 2] (Conflits : 4)
Voisins et nombre de conflits :
  [0, 3, 3, 2] -> 2
  [1, 3, 3, 2] -> 3
  [3, 3, 3, 2] -> 4
  [2, 0, 3, 2] -> 3
  [2, 1, 3, 2] -> 3
  [2, 2, 3, 2] -> 5
  [2, 3, 0, 2] -> 3
  [2, 3, 1, 2] -> 3
  [2, 3, 2, 2] -> 5
  [2, 3, 3, 0] -> 2
  [2, 3, 3, 1] -> 3
  [2, 3, 3, 3] -> 4

Itération 1 : État courant = [0, 3, 3, 2] (Conflits : 2)
Voisins et nombre de conflits :
  [1, 3, 3, 2] -> 3
  [2, 3, 3, 2] -> 4
  [3, 3, 3, 2] -> 4
  [0, 0, 3, 2] -> 3
  [0, 1, 3, 2] -> 2
  [0, 2, 3, 2] -> 3
  [0, 3, 0, 2] -> 1
  [0, 3, 1, 2] -> 1
  [0, 3, 2, 2] -> 3
  [0, 3, 3, 0] -> 2
  [0, 3, 3, 1] -> 2
  [0, 3, 3, 3] -> 4

Itération 2 : État courant = [0, 3, 0, 2] (Conflits : 1)
Voisins et nombre de conflits :
  [1, 3, 0, 2] -> 0
  [2, 3, 0, 2] -> 3
  [3, 3, 0, 2] -> 1
  [0, 0, 0, 2] -> 4
  [0, 1, 0, 2] -> 3
  [0, 2, 0, 2] -> 2
  [0, 3, 1, 2] -> 1
  [0, 3, 2, 2] -> 3
  [0, 3, 3, 2] -> 2
  [0, 3, 0, 0] -> 3
  [0, 3, 0, 1] -> 3
  [0, 3, 0, 3] -> 3

État final : [1, 3, 0, 2] avec 0 conflits
```

Optimisation combinatoire

- **Méthodes Métaheuristiques à population**
  - **Définition.** Une métaheuristique à population est une stratégie d'optimisation qui, au lieu d'améliorer itérativement une seule solution comme le fait la recherche locale, maintient et fait évoluer un ensemble de solutions candidates en parallèle.
  - Elles sont adaptatives à plus d'un problème
  - Ces solutions interagissent entre elles, échangent des informations, et s'améliorent collectivement au fil des itérations grâce à des mécanismes inspirés de **phénomènes naturels ou sociaux**.
  - Les métaheuristiques permettent d'explorer l'espace de recherche de manière plus large, en cherchant à approcher l'optimal global.
  - Elles ne garantissent pas l'optimalité, mais sont généralement plus efficaces que les simples heuristiques pour trouver des solutions de haute qualité.

Optimisation combinatoire

- **Méthodes Métaheuristiques à population**
  - Quelques exemples :
    - ✓ **Algorithme génétique (Genetic Algorithm)** Basé sur l'évolution biologique (sélection, croisement, mutation).
    - ✓ **Colonies de fourmis (Ant Colony Optimization)** s'inspire du comportement des fourmis pour explorer les chemins.
    - ✓ **Algorithmes à Essaims (inspirés du comportement collectif)**
      - Optimisation par Essaim de Particules (PSO - Particle Swarm Optimization)
      - Artificial Bee Colony (ABC), **2005**
      - Firefly Algorithm (FA), **2008**
      - Bat Algorithm (BA), **2010**
      - Cuckoo Search (CS), **2009**
      - Grey Wolf Optimizer (GWO), **2014**
      - Whale Optimization Algorithm (WOA), **2016**
      - Moth-Flame Optimization (MFO), **2015**

Optimisation combinatoire

- **Méthodes Métaheuristiques à population. Optimisation par Essaim de Particules (PSO - Particle Swarm Optimization)**
  - L’algorithme Particle Swarm Optimization a été proposé par James Kennedy & Russell Eberhart (1995).
  - PSO est un algorithme qui imite le comportement collectif des animaux comme les oiseaux, les poissons ou les insectes, qui se déplacent ensemble en échangeant des informations pour s’adapter à leur environnement.
  - C’est un algorithme d'optimisation **stochastique** basé sur la population.
  - le PSO est bien adapté aux problèmes d'optimisation **continue**, mais il peut aussi être adapté pour des problèmes **discrets** ou combinatoires.

Optimisation combinatoire

- **Méthodes Métaheuristiques à population. Optimisation par Essaim de Particules (PSO - Particle Swarm Optimization)**

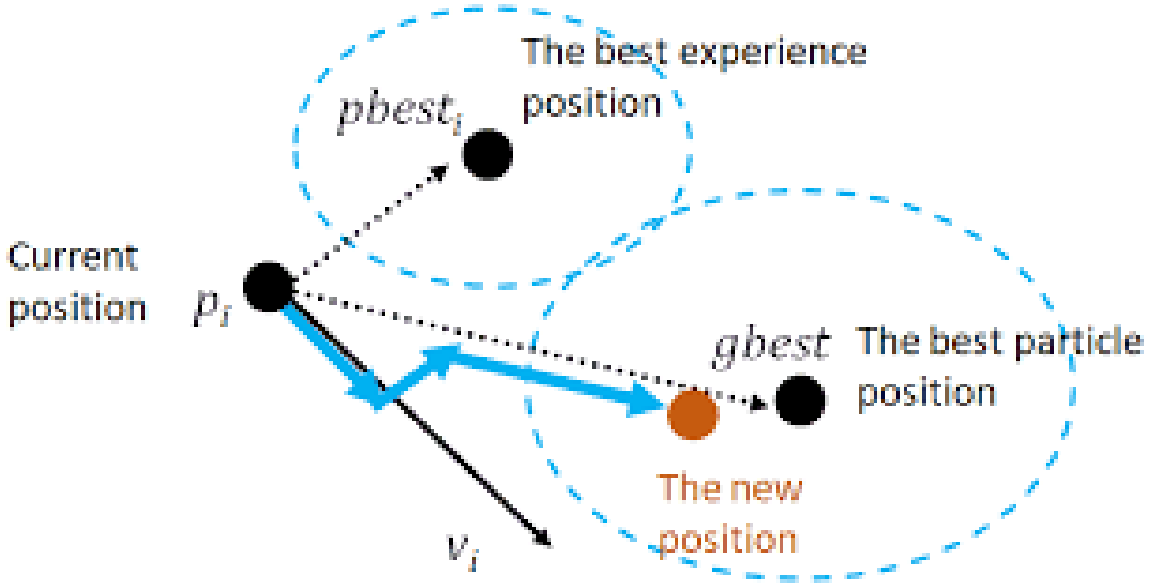
Concepts de base de PSO

  - **Particules :** Chaque particule i est une solution candidate au problème d'optimisation, elle est caractérisée par :
    - ✓ sa **position** : Représente une solution spécifique dans l'espace de recherche (un point dans l'espace de décision).
    - ✓ Sa **vitesse** : Indique la direction et la magnitude du mouvement de la particule dans l'espace de recherche.
    - ✓ Sa **meilleure position personnelle** ($pbest_i$) : La meilleure position que la particule **i** a trouvée jusqu'à présent (recherche local).
  - La **meilleure position globale** de l'essaim (gbest) : La meilleure position trouvée par n'importe quelle particule dans tout l'essaim jusqu'à présent (recherche global, social).
    - ○

Optimisation combinatoire

- **Méthodes Métaheuristiques à population. Optimisation par Essaim de Particules (PSO - Particle Swarm Optimization)**

Concepts de base de PSO



Optimisation combinatoire

- **Méthodes Métaheuristiques à population. Optimisation par Essaim de Particules (PSO - Particle Swarm Optimization)**
  - **Mouvement des Particules :** À chaque itération, la position et la vitesse de chaque particule sont mises à jour en tenant compte de trois composantes principales :
    - **Composante Inertielle ω :** la vitesse actuelle de la particule, cela représente la tendance de la particule à continuer dans sa direction actuelle.
    - **Composante Cognitive (ou Perso-best) :** l'attraction de la particule vers sa propre meilleure position personnelle pbest[i]), cela représente l'apprentissage et la mémoire de la particule.
    - **Composante Sociale (ou Global-best) :** l'attraction de la particule vers la meilleure position trouvée par l'ensemble de l'essaim (gbest), cela représente l'échange d'informations et la collaboration au sein de l'essaim.

Concepts de base de PSO

Optimisation combinatoire

- **Méthodes Métaheuristiques à population. Optimisation par Essaim de Particules (PSO - Particle Swarm Optimization)**
  - À chaque itération **t** la vitesse et la position de chaque particule sont mise à jour comme suit :

    $$v_i(\mathrm{t}+1) = \omega v_i(t) + c_1 r_1(\mathrm{pbest}_i(\mathrm{t}) - x_i(\mathrm{t})) + c_2 r_2(gbest(\mathrm{t}) - x_i(\mathrm{t}))$$

  - Où :

$\boldsymbol{v_i(t)}$: Vitesse de la particule i à l'itération t.

ω (coefficient d'inertie) : Poids de la vitesse précédente. Un ω élevé encourage l'exploration (recherche globale), un ω faible encourage l'exploitation (recherche locale).

$\boldsymbol{c_1}$ (coefficient cognitif) : Poids de l'attraction vers la meilleure position personnelle.

$\boldsymbol{c_2}$ (coefficient social) : Poids de l'attraction vers la meilleure position globale.

$\boldsymbol{r_1, r_2}$: Nombres aléatoires uniformément distribués dans [0,1] qui ajoutent un effet stochastique au mouvement.

$\boldsymbol{pbest_i}$**(t)** : Meilleure position personnelle de la particule i jusqu'à l'itération t.

gbest(t) : Meilleure position globale de l'essaim jusqu'à l'itération t.

$\boldsymbol{x_i}$**(t):** Position actuelle de la particule i à l'itération t.

Mise à jour de la Position **:** $x_i(\mathrm{t}+1) = x_i(\mathrm{t}) + v_i(\mathrm{t}+1)$

Optimisation combinatoire. PSO

Algorithme 19. Particle Swarm Optimization (PSO)

1. Initialisation: (minimisation)

f une fonction objective

d : dimension de l'espace de recherche (nombre de variables).

n : nombre de particules dans l'essaim.

t : iteration, **i** : particule

x[i] : position de i, v[i]: vitesse de i

max_iter : Nombre maximum d'itérations.

ω : Coefficient d'inertie.

c1 : Coefficient cognitif (attraction vers pbest, encourage l'**exploitation** locale.).

c2 : Coefficient social (attraction vers gbest, encourage la **coopération**, suivre la **meilleure particule du groupe**).

x_min, x_max : Bornes de l'espace de recherche pour chaque dimension.

vit_max : vitesse maximale permise pour les particules .

pbest[i] : meilleure position personnelle d'une particule i

gbest : **meilleure position globale** trouvée par tout l'essaim à l'iération t

Optimisation combinatoire

Algorithme 18. Particle Swarm Optimization (PSO)

 # initialiser la position de chaque particule et sa vitesse

2. Pour chaque particule i de 1 à n:

x[i] ← position aléatoire ∈ [x_min, x_max]

v[i] ← vitesse aléatoire ∈ [-v_max, v_max]

pbest[i] ← x[i]

f_pbest[i] ← f(x[i])

gbest ← pbest[i] ayant le plus petit f_pbest[i]

Algorithme 18. Particle Swarm Optimization (PSO)

Mise à jour de la position de chaque particule et sa vitesse

3. **Répéter pour t = 1 à max_iter** :

  Pour chaque particule i = 1 à n :

    r1, r2 ← nombres aléatoires ∈ [0, 1]

    v[i] ← ω * v[i] # terme_inertie

        + c1 * r1 * (pbest[i] - x[i]) # terme_cognitif

            + c2 * r2 * (gbest - x[i]) #terme social

    v[i] ← clip(v[i], -v_max, v_max)

    x[i] ← x[i] + v[i]

    x[i] ← clip(x[i], x_min, x_max)

    f_curr ← f(x[i])

    Si f_curr < f_pbest[i] : #minimisation

    pbest[i] ← x[i]

    f_pbest[i] ← f_curr

    Si f_pbest[i] < f(gbest) :

    gbest ← pbest[i]

4.Retourner gbest, f(gbest)

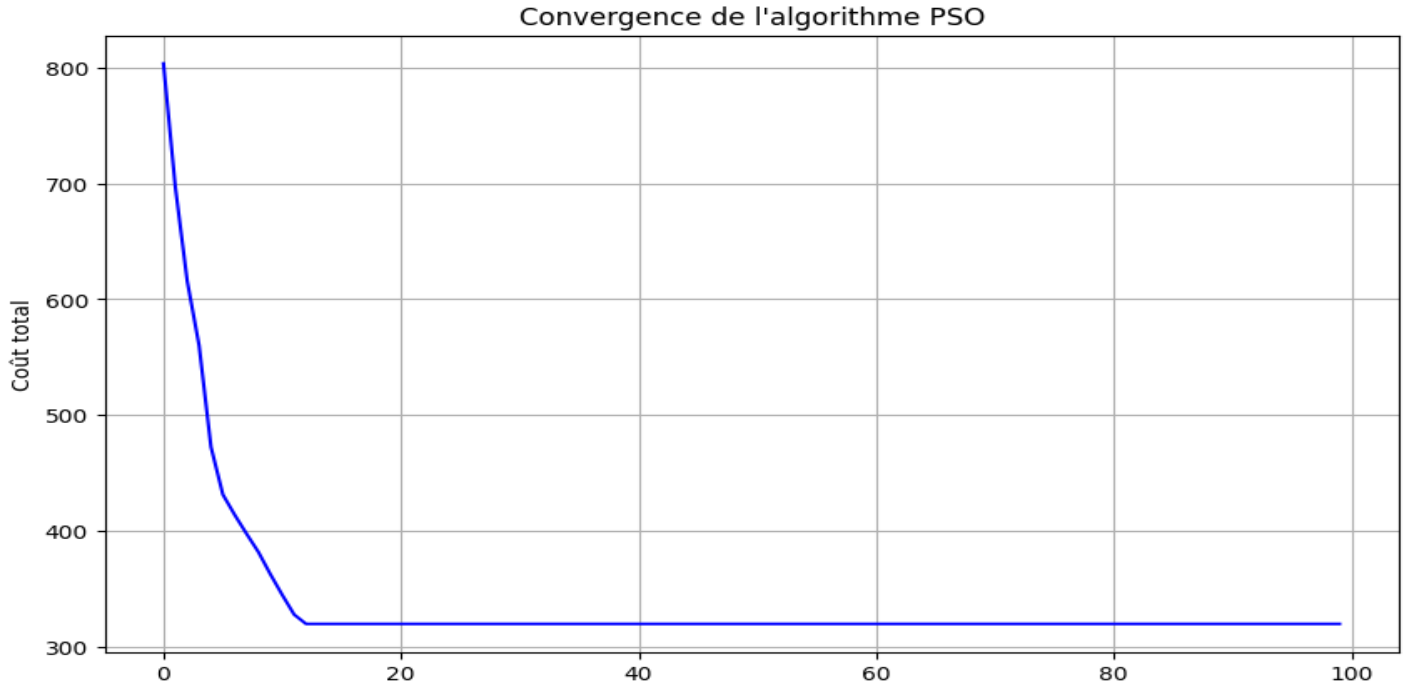
Optimisation combinatoire. PSO

- **Méthodes Métaheuristiques à population. PSO - Particle Swarm Optimization**
  - **Exemple.** Une usine veut minimiser ses coûts totaux qui dépendent de 3 paramètres:
    - ✓ **x1**: Nombre d'heures de travail par jour (4-12h)
    - ✓ **x2**: Température du processus (100-300°C)
    - ✓ **x3**: Taux d'utilisation des machines (50-100%)
  - La fonction coût à minimiser est: Coût total = (coût main d'œuvre) + (coût énergétique) + (coût maintenance) = $30*x_1 + 0.5*(x_2-80)^2 + 20*(100-x_3)$

Optimisation combinatoire. PSO

- **Méthodes Métaheuristiques à population. PSO - Particle Swarm Optimization**
  - **Exemple.** Une usine veut minimiser ses coûts totaux qui dépendent de 3
  -

```
Solution optimale trouvée:
- Heures de travail: 4.00 h/jour
- Température de production: 100.00 °C
- Taux d'utilisation machines: 100.00 %

Coût total minimum: 320.00 €
```



ıergétique) +

Optimisation combinatoire. PSO

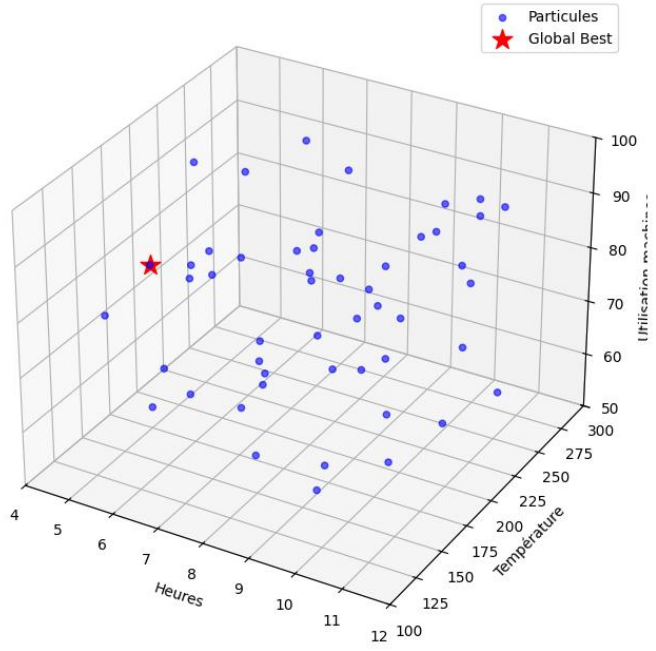
- **Méthodes Métaheuristiques à population. PSO - Particle Swarm Optimization**

```python
import numpy as np
import random
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Paramètres PSO
n = 50
d = 3  # Nombre de dimensions (x1, x2, x3)
max_iter = 100
w = 0.7    # inertie
c1 = 1.5   # cognitif
c2 = 1.5   # social

# Bornes pour chaque variable
x_min = np.array([4, 100, 50])    # [heures min, température min, utilisation min]
x_max = np.array([12, 300, 100])   # [heures max, température max, utilisation max]
v_max = (x_max - x_min) * 0.1      # Vitesse maximale (10% de l'intervalle)

# Fonction coût à minimiser
def cost_function(x):
    x1, x2, x3 = x
    return 30*x1 + 0.5*(x2-80)**2 + 20*(100-x3)
```

```python
def pso_minimization(f, d, n, max_iter, w, c1, c2, x_min, x_max, v_max):
    # Initialisation
    x = np.random.uniform(low=x_min, high=x_max, size=(n, d))  # Positions
    v = np.random.uniform(low=-v_max, high=v_max, size=(n, d))  # Vitesses
    pbest = x.copy()  # Meilleures positions personnelles
    f_pbest = np.array([f(xi) for xi in x])  # Valeurs associées

    # Meilleure solution globale initiale
    gbest = pbest[np.argmin(f_pbest)].copy()
    gbest_val = f(gbest)
    # Historique de convergence
    convergence = []
    # Boucle d'optimisation
    for t in range(max_iter):
        for i in range(n):
            # Mise à jour de la vitesse
            r1, r2 = np.random.rand(d), np.random.rand(d)
            v[i] = (w * v[i]
                    + c1 * r1 * (pbest[i] - x[i])
                    + c2 * r2 * (gbest - x[i]))
     # Limitation des vitesses
            v[i] = np.clip(v[i], -v_max, v_max)
            # Mise à jour position avec respect des bornes
            x[i] += v[i]
            x[i] = np.clip(x[i], x_min, x_max)
    # Évaluation
            current_score = f(x[i])
    # Mise à jour meilleure solution personnelle
            if current_score < f_pbest[i]:
                pbest[i] = x[i].copy()
                f_pbest[i] = current_score
     # Mise à jour meilleure solution globale
        current_pbest = np.argmin(f_pbest)
        if f_pbest[current_pbest] < gbest_val:
            gbest = pbest[current_pbest].copy()
            gbest_val = f_pbest[current_pbest]
        convergence.append(gbest_val)
    return gbest, gbest_val, convergence
```

Optimisation combinatoire. PSO

- **Méthodes Métaheuristiques à population. PSO - Particle Swarm Optimization**



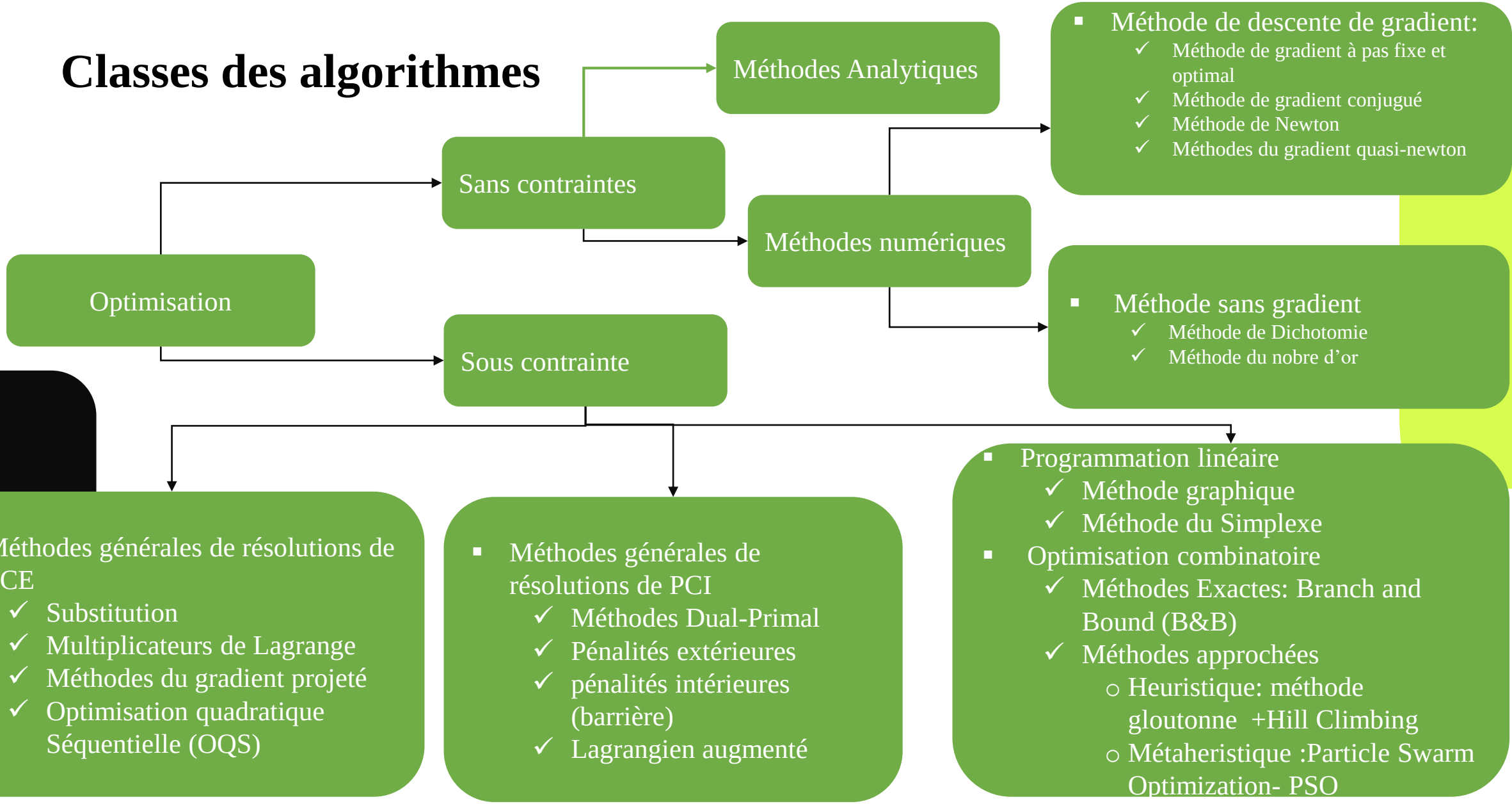

```python
# Exécution de l'optimisation
best_solution, best_cost, convergence_history = pso_minimization(
    f=cost_function,
    d=d,
    n=n,
    max_iter=max_iter,
    w=w,
    c1=c1,
    c2=c2,
    x_min=x_min,
    x_max=x_max,
    v_max=v_max
)
```

```
Solution optimale trouvée:
- Heures de travail: 4.00 h/jour
- Température de production: 100.00 °C
- Taux d'utilisation machines: 100.00 %

Coût total minimum: 320.00 €
```

Optimisation combinatoire. PSO

- **Méthodes Métaheuristiques à population**. **PSO - Particle Swarm Optimization**
  - **Avantages**
    - ✓ **Simple et facile à implémenter** : peu de paramètres que d'autres algorithmes évolutionnaires.
    - ✓ **Convergence rapide** grâce au partage d'informations entre particules.
    - ✓ **Fonctionne sans gradient** , pas besoin des fonctions différentiables ou continues.
    - ✓ **Équilibre exploration/exploitation** → Contrôlé par l'inertie (w) et les coefficients $c1$, $c2$.
    - ✓ **Parallélisable car le c**alculs indépendants pour chaque particule.
  - **Limites**
    - ✓ **Convergence prématurée** , il y a risque de rester bloqué dans un optimum local.
    - ✓ **Sensible aux paramètres** si on a une mauvaise configuration de w, $c1$, $c2$
    - ✓ **Pas de garantie d'optimum global** comme la plupart des métaheuristiques.
    - ✓ **Gestion difficile des contraintes,** il faut utiliser aussi des adaptations comme les pénalisations.

Classes des algorithmes



Fin du Cours