Optimisation Discrète

Méthode approchée

Yannick Kergosien
Polytech Tours
LIFAT EA 6300– Equipe ROOT

Plan

- **Introduction**
  - Motivation
  - Deux exemples d'implémentation de métaheuristiques pour un problème type VRP
- **Classification**
- **Eléments importants de conception**
  - Représentation d'une solution (codage d'une solution)
  - Contraintes et évaluation d'une solution
  - Stratégies d'exploration de nouvelles solutions
  - Autres éléments
- **Quelques exemples d'heuristiques avec analyses**
- **Quelques Métaheuristiques de type parcours**
- **Quelques Métaheuristiques de type population**
- **Quelques mots sur les hybridations de métaheuristiques**

Introduction

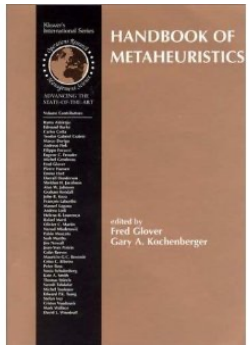
- **Pourquoi utiliser des méthodes approchées ?**
  - Complexité des problèmes rarement de classe P
  - Instances réelles souvent de grandes tailles
  - Exigence de méthodes rapides
  - Des « bonnes » solutions suffisent très souvent
  - Difficulté de garantir l'optimalité
- **Grands types de méthode approchées :**
  - Heuristiques
    - Méthodes de résolution pour un problème spécifique
  - Métaheuristiques
    - Méthode générique indépendante du problème

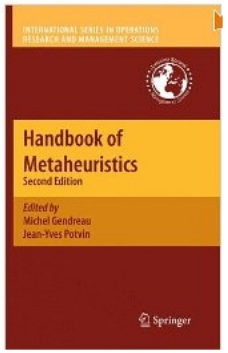
  (+Matheuristique et hybridation diverse)

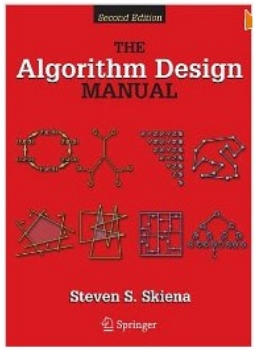
Evolutionary programming
Genetic algorithms
Local search
Scatter search
Simulated annealing
Artificial immune systems
Smoothing method
Tabu search
Greedy adaptive search procedure
Threshold accepting
Coevolutionary algorithms
Iterated local search
Ant colonies optimization
Genetic programming
Noisy method
Great deluge
Estimation of distribution algorithms Differential evolutio
Cultural algorithms
Particle swarm optimization
Variable neighborhood search
Bee colony
Guided local search
Covariance matrix adaptation evolution strategy
…

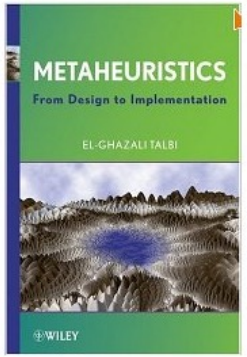
Introduction

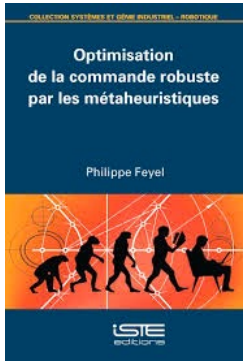
- **De nombreux ouvrages et publications scientifiques**

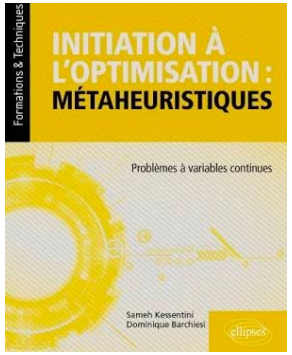












- **Deux exemples d'implémentation de métaheuristiques pour un problème type VRP**
  - Simple : recherche locale
  - Sophistiqué : Algorithme génétique avec codage indirect

=> Au tableau

Comment évalueriez-vous cette recherche locale?

A. **Très bonne**
B. **Bonne**
C. **Mauvaise**
D. **Très mauvaise**
E. **Sans opinion**

Quel serait le principal défaut ?

A. **Trop longue à itérer**
B. **Exploration que partielle des solutions**
C. **Cycle potentiel**
D. **Demande trop en mémoire**
E. **Sans opinion 2 (le retour)**

Comment évalueriez-vous cet algorithme génétique?

A. **Très bon**
B. **Bon**
C. **Mauvais**
D. **Très mauvais**
E. **Rien compris**

Quel serait le principal avantage ?

A. **Rapidité**
B. **Exploration efficace grâce à son codage**
C. **Stabilité de l'algorithme**
D. **Paramétrage simple**
E. **C'est rigolo**

Classification

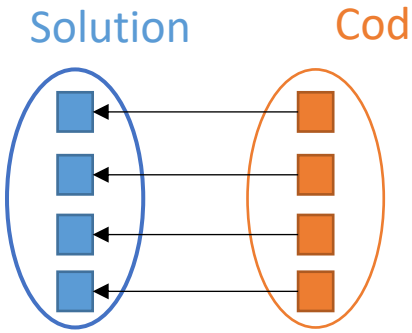
- Inspiré de la nature vs non-inspiré de la nature
- Avec ou sans mémoire
- Déterministe ou stochastique
- Constructive vs améliorative
- Parcours vs population

Eléments importants : codage d'une solution

- **Plusieurs possibles selon le problème**
- **Le choix doit tenir compte :**
  - Des propriétés du problème
  - Des opérations réalisées par la méthode
- **Un bon codage se doit être :**
  - Complet : toutes les solutions possibles sont représentées.
  - Efficace : les opérateurs manipulant les solutions doit rester simple (mémoire et temps de calcul) (ex : N-queens problem : (x,y) vs (x) => $(N.N)^N$ vs N^N)
  - Connexe : n'importe quelle solution peut être atteignable d'une solution donnée à partir d'un ensemble fini d'opérations

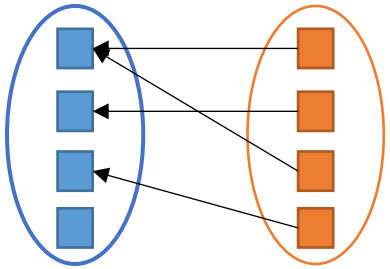
Eléments importants : codage d'une solution

- **Codage direct vs codage indirect**
- **3 type de codage :**
  - Un vers Un



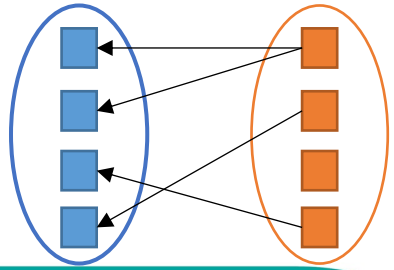
Ex : PVC et permutation

  - Un vers Plusieurs



Ex : Sac à dos et permutation + décodage par insertion des 'x' premiers objets

  - Plusieurs vers Un



Ex : VRP et le « giant-tour »

Eléments importants : Contraintes et évaluation d'une solution

- **La fonction objectif**
  - Une ou plusieurs (front de Pareto)
  - Fonction objectif exacte vs fonction objectif guidée
    - Orienté la méthode vers les bonnes solutions
    - Sortir des solutions optimales locales
    - Tenir compte de la difficulté de trouver des solutions réalisables (cf diapo suivante)
  - Simulation possible (cas des problèmes stochastiques/avec aléas)
  - Doit être une opération non (ou peu) coûteuse en temps CPU

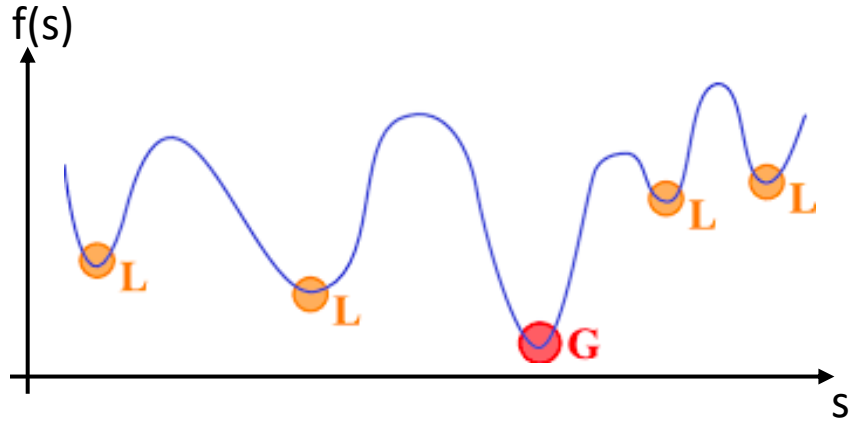
Eléments importants : Contraintes et évaluation d'une solution

- **Parfois la prise en compte des contraintes du problème peut s'avérer complexe, il existe alors des stratégies :**
  - De rejet : seul les solutions réalisables sont explorées, toute solution non-réalisable est rejetée.
  - De pénalisation : l'exploration de solution non-réalisable est autorisée mais ces solutions sont généralement pénalisées lors de l'évaluation (fonction objectif)
    - Nombre de contraintes non respectées
    - « Quantité » d'infaisabilité
    - Facteur de pénalité dans la fonction objectif constant ou non
  - De réparation : si une solution non-réalisable est générée, une procédure de réparation est appliquée
  - De décodage : utilisation de codage indirect pour représenter la solution
  - De préservation : les opérateurs d'exploration ne peuvent conduire qu'à des solutions réalisables

Eléments importants : Contraintes et évaluation d'une solution

- **Exemple : PVC avec fenêtre de temps :**
  - De rejet : Recherche locale avec 2-Opt
  - De pénalisation : Recherche locale avec 2-Opt
    - Nombre de passage en retard
    - Somme des retards
  - De réparation : Recherche locale avec 2-Opt puis SWAP pou réparer
  - De préservation : Recherche locale avec insertion et conditions à respecter

Stratégies d'exploration et autres éléments



- **Intensification vs diversification**
  - Diversifier pour éviter les optimums locaux
  - Intensifier quand nécessaire
- **Mémorisation**
  - Attention aux coûts CPU et mémoire
- **Critère d'évaluation d'une méthode (cf. chapitre suivant) :**
  - Résultats (qualité des solutions trouvées, temps de calculs, etc.)
  - Convergence
  - Stabilité
  - Paramétrage
  - Complexité à implémenter
  - Flexibilité/générique
  - Parallélisable ou non

Heuristiques et analyses

- **Heuristique**
  - Objectif : obtenir une « bonne » solution rapidement
  - Comment concevoir une heuristique ?
    - Trouver une suite logique d'opération menant à une solution réalisable
    - Chercher un exemple (une instance du problème) pour lequel l'algorithme ne trouve pas la solution optimale
    - Essayer d'améliorer l'algorithme en fonction de l'exemple trouvé
    - Itérer les deux dernières étapes jusqu'à ….

- **Exemple pour le sac à dos :**
  - Algorithme 1 : Trier les éléments par poids décroissant puis essayer de les insérer un à un dans le sac.
  - Exemple avec un objet ayant la plus petite valeur mais avec un poid égale à la capacité du sac.
  - Algorithme 2 : Trier les éléments par p_i/w_i décroissant …

Heuristiques et analyses

- **L'analyse d'une heuristique porte souvent sur les critères suivants :**
  - Complexité en temps et en espace de l'algorithme.
    - Souvent en temps linéaire ou polynomial au pire.
  - Preuve d'optimalité (ou de non optimalité).
    - Preuve qu'un algorithme est non optimale : trouver un exemple avec sa solution optimale et montrer que l'algorithme trouve une solution de moins bonne qualité.
    - Preuve d'optimalité : plusieurs manières de faire, la plus courante étant la suivante : on définit une solution optimale S^{opt} et une solution S^{h} trouvée par l'algorithme et on suppose que S^{h} est moins bonne que S^{opt}, on compare la structure des solutions et on arrive à une contradiction lié à la construction de S^{h} par l'algorithme.
  - Garantie de performance par rapport à la solution optimale.
    - Exemple : au pire des cas, le glouton trouve une solution à 10% de l'optimale
    - Méthode : trouver « ce » pire cas et comparer l'évaluation de la solution optimale de ce pire cas et l'évaluation de la solution trouvée par l'algorithme.

Heuristiques et analyses

- **Exemple avec le problème de rendu de monnaie**
  - Soit n pièces en quantité illimitée, où à chaque pièce i correspond une valeur c_i, et une somme v à rendre.
  - Quelles pièces prendre afin de trouver le nombre minimum de pièces dont la somme est égale à v ?

Complexité du problème de rendu de monnaie ?

A. **Polynomial**
B. **NP Complet au sens faible**
C. **NP Complet au sens fort**
D. **Aucune idée**

Heuristiques et analyses

- **Exemple avec le problème de rendu de monnaie**
  - Soit n pièces en quantité illimitée, où à chaque pièce i correspond une valeur c_i, et une somme v à rendre.
  - Quelles pièces prendre afin de trouver le nombre minimum de pièces dont la somme est égale à v ?

  - Exercice : Trouver une heuristique pour le rendu de monnaie.

Heuristiques et analyses

- **Exemple avec le problème de rendu de monnaie**
  - L ← Liste des pièces triées par c_i décroissant
  - Rendu ← ∅
  - i ←Premier élément de L
  - L← L \ {i}
  - Tant que (v>0) et (L≠∅) faire
    - Si c_i<v alors
      - v ← v – c_i
      - Rendu ← Rendu U {i}
    - Sinon
      - i ←Premier élément de L
      - L← L \ {i}
  - Si v==0 alors
    - Retourner Rendu
  - Sinon
    - Retourner ∅

Que pensez-vous de cet algorithme?

A. **Trouve toujours la solution optimale et s'exécute en temps polynomial**
B. **Trouve toujours la solution optimale mais ne s'exécute pas en temps polynomial**
C. **Ne trouve pas toujours la solution optimale et s'exécute en temps polynomial**
D. **Ne trouve pas toujours la solution optimale et ne s'exécute pas en temps polynomial**
E. **Je m'en fou, je ne suis pas Kaftier**

Heuristiques et analyses

- **Exemple avec le problème de rendu de monnaie**
  - A noter que cet algorithme ne trouve pas toujours de solution réalisable alors qu'il pourrait en exister une.
  - Analyse sur la complexité :
    - Le trie peut se réaliser en O(n.log(n)). Puis à chaque étape de l'algorithme, on soustrait la valeur d'une pièce de la somme v. Il peut donc y avoir jusqu'à v étapes au pire des cas. Le temps d'exécution est donc O(v + n.log(n)) au pire des cas. Ce temps d'exécution n'est pas polynomial car on n'est pas polynomial en la taille de l'entrée (v).
    - Cependant il est possible d'améliorer l'algorithme.

Heuristiques et analyses

- **Exemple avec le problème de rendu de monnaie**
  - L ← Liste des pièces triées par c_i décroissant
  - Rendu ← ∅
  - i ←Premier élément de L
  - L← L \ {i}
  - Tant que (v>0) et (L≠∅) faire
    - Si c_i<v alors
      - v ← v – c_i
      - Rendu ← Rendu ∪ {i}
    - Sinon
      - i ←Premier élément de L
      - L← L \ {i}
  - Si v==0 alors
    - Retourner Rendu
  - Sinon
    - Retourner ∅

- m ← ⌊v/c_i⌋
- v ← v – m.c_i
- Rendu ← Rendu ∪ {m x i}

Heuristiques et analyses

- **Exemple avec le problème de rendu de monnaie**
  - Analyse sur la complexité :
    - Le trie peut se réaliser en O(n.log(n)). Puis à chaque étape de l'algorithme, on retire une pièce de L. Il y a donc n étapes au pire des cas. Le temps d'exécution est donc O(n + n.log(n)) au pire des cas.
  - Analyse sur l'optimalité ? Non...
    - Soit l'instance suivante :
      - 3 pièces de valeur 6, 4 et 1, et une somme à rendre de 8
    - Solution trouvée par le glouton : {6,1,1}.
    - Solution optimale : {4,4}
  - Remarque :
    - Cet algorithme est optimale pour des jeux de pièces qualifiés de « Canoniques »
    - Il existe des algorithmes en $O(n^2)$ pour décider si un jeu de pièce est canonique.

Heuristiques et analyses

- **Exemple avec une variante du problème de sac à dos**
  - Soit un ensemble de n objets, où chaque objet i est en quantité limitée n_i et est représenté par un couple (p_i,w_i): p_i est sa valeur pour un poids w_i, et une capacité W. Chaque objet peut être fractionné.
  - Quelles quantités de chaque objet prendre pour maximiser la valeur totale sans dépasser la capacité maximale W ?
  - Exercice : Trouver (adapter) une heuristique pour ce problème de sac à dos.

Heuristiques et analyses

- **Exemple avec une variante du problème de sac à dos**
  - L ← Liste des objets triés par p_i/w_i décroissant
  - Sac ← ∅
  - PS ← 0
  - Tant que (L≠∅) faire
    - i ←Premier élément de L
    - L← L \ {i}
    - a ← min(n_i ; (W-PS))
    - PS ← PS + a
    - Sac ← Sac ∪ {a quantité de i}
  - Retourner Sac

Cette heuristique trouve-t-il toujours la solution optimale?

A. **Oui !**
B. **Non !**
C. **Je pense que oui**
D. **Je pense que non**
E. **J'essaye encore de comprendre l'algorithme**

Heuristiques et analyses

- **Exemple avec une variante du problème de sac à dos**
  - Sans perdre de généralité, on suppose que les objets sont indicés par p_i/w_i décroissant.
  - On note:
    - x_i^h la quantité d'objet i dans le sac de la solution S^h donnée par l'heuristique
    - x_i^{opt} la quantité d'objet i dans le sac d'une solution optimale S^{opt}. Comme il peut y avoir plusieurs solutions optimales, S^{opt} représente la solution optimale pour laquelle $x_i^{opt} = x_i^h$ pour la plus grande valeur possible i (c-à-d $x_i^{opt} = x_i^h$ pour les i premiers objets).
  - On suppose que S^h n'est pas optimale et soit k le plus petit indice tel que $x_k^{opt} \neq x_k^h$
    - Cas 1 : $x_k^{opt} > x_k^h$: Impossible car l'heuristique aurait choisi la plus grande quantité possible de l'objet k donc $x_k^{opt} \leq x_k^h$
    - Cas 2 : $x_k^{opt} < x_k^h$: Parmi les objets k+1 à n, S^{opt} contient au moins un poids $b = x_k^h - x_k^{opt}$ (puisque cette quantité b a été mise dans S^h). Cependant puisque les objets sont triés par ratio p_i/w_i croissant, on peut remplacer x_k^{opt} par x_k^h et diminuer de poids b le poids pris par les objets k+1 à n. Donc la nouvelle solution optimale a $x_k^{opt} = x_k^h$ => contradiction sur le choix de la solution optimale (c-à-d on n'a pas la plus grande valeur possible i tel que $x_i^{opt} = x_i^h$).

Heuristiques et analyses

- **Exemples avec le problème de bin packing**
  - Soit un ensemble de n objets, où chaque objet i a une taille réelle $c_i \in [0,1]$, et une capacité de boîte égale à 1. Comment regrouper les objets dans les boîtes de manière à utiliser le minimum de boîte ?
  - Une première heuristique : NextFit
    - L ← Liste des objets non triées
    - k ← 1
    - $P_k \leftarrow 0$
    - Tant que (L≠∅) faire
      - i ←Premier élément de L
      - L← L \ {i}
      - Si ($P_k + c_i \leq 1$) alors
        - $P_k \leftarrow P_k + c_i$
      - Sinon
        - k ← k + 1
        - $P_k \leftarrow c_i$
    - Retourner k

Heuristiques et analyses

- **Exemples avec le problème de bin packing :**
  - Garantie de performance de NextFit :
    - Bien que clairement sous optimale, on peut montrer que : $f^{NextFit} < 2\, f^{OPT}$ avec
      - $f^{NextFit}$ le nombre de bin utilisé par le glouton NextFit
      - f^{OPT} le nombre de bin utilisé dans la solution optimale
    - Démonstration :
      - Quelque soit l'instance d'entrée, 2 bins consécutifs sont effectivement utilisés par le glouton si la somme des tailles des objets contenus dans ces 2 bins est supérieur à 1.
      - Donc pour placer une suite d'objets de taille équivalent à 1, il faut au plus deux bins

Somme = 1 → 2 bins => $\sum_{i=1}^{n} c_i \rightarrow 2 \lceil \sum_{i=1}^{n} c_i \rceil$ => $f^{NextFit} < 2 \lceil \sum_{i=1}^{n} c_i \rceil$

Or $f^{OPT} > \lceil \sum_{i=1}^{n} c_i \rceil$ donc $f^{NextFit} / f^{OPT} < 2 \lceil \sum_{i=1}^{n} c_i \rceil / \lceil \sum_{i=1}^{n} c_i \rceil$ => $f^{NextFit} / f^{OPT} < 2$

Heuristiques et analyses

- **Exemples avec le problème de bin packing**
  - Une deuxième heuristique : FirstFit
    - L ← Liste des objets triées par c_i décroissant
    - k ← 1
    - P_k ← 0
    - Tant que (L≠∅) faire
      - i ←Premier élément de L
      - L← L \ {i}
      - k' ←1
      - Tant que ($P_{k'}$ + c_i > 1) & (k'≤ k) alors
        - k' ←k' + 1
      - Si k'≤ k alors
        - $P_{k'}$ ← $P_{k'}$ + c_i
      - Sinon
        - k ← k + 1
        - P_k ← c_i
    - Retourner k

Heuristiques et analyses

- **Exemples avec le problème de bin packing :**
  - Garantie de performance de FirstFit : $f^{FirstFit} \leq 1{,}5\, f^{OPT} + 1$
  - Démonstration :
    - Soit la répartition des objets en 4 ensembles :
      - A = { i tel que $c_i > 2/3$}
      - B = { i tel que $2/3 \geq c_i > 1/2$}
      - C = { i tel que $1/2 \geq c_i > 1/3$}
      - D = { i tel que $1/3 \geq c_i$ }
    - L'idée est de démonter que la relation est vérifiée dans les deux cas suivants :
      - Cas 1 : dans la solution de l'heuristique, il existe au moins un bin qui ne contient que des objets de D
      - Cas 2 : dans la solution de l'heuristique, il n'y a aucun bin qui contient uniquement des objets de D

Heuristiques et analyses

- **Exemples avec le problème de bin packing :**
  - Garantie de performance de FirstFit : $f^{FirstFit} \leq 1{,}5\, f^{OPT} + 1$
  - Démonstration :
    - Cas 1 : (au moins un bin qui ne contient que des objets de D)
      - => au plus un bin rempli à moins de 2/3 (le dernier)
      - Sur les 2/3 des $f^{FirstFit}$ -1 bins, tous les objets ne logent pas : $(2/3).(f^{FirstFit} - 1) < \sum_{i=1}^{n} c_i$
      - Sachant que :

$$f^{OPT} \geq \left\lceil \sum_{i=1}^{n} c_i \right\rceil \qquad \Rightarrow (2/3).(f^{FirstFit} - 1) \leq f^{OPT}$$

      - La relation est bien vérifiée.

Heuristiques et analyses

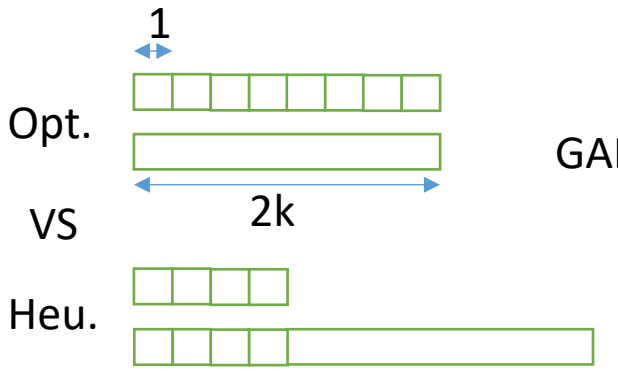
- **Exemples avec le problème de bin packing :**
  - Garantie de performance de FirstFit : $f^{FirstFit} \leq 1{,}5\ f^{OPT} + 1$
  - Démonstration :
    - Cas 2 : (aucun bin ne contient uniquement des objets de D)
      - Si les objets D sont retirés de l'instance alors la solution de l'heuristique ne change pas : $f^{FirstFit} = f^{FirstFit/D}$
      - Supposons cette instance : dans une solution optimale, les objets de A sont nécessairement tous seul dans un bin et chaque bin contient au plus 2 objets.
      - Or l'heuristique construit une telle instance
      - Donc pour ce type d'instance : $f^{FirstFit/D} = f^{OPT/D}$
      - Comme $f^{OPT/D} \leq f^{OPT}$, la relation est vérifiée. On peut même en déduire que : $f^{FirstFit} = f^{OPT}$ (optimale)

Heuristiques et analyses

- **Heuristique de liste**
  - Méthodologie commune à beaucoup d'heuristique
  - Algorithme :
    - Trier les variables du problème
    - Tant que toutes les variables ne sont pas instanciées faire
      - Sélectionner une variable parmi les variables non-instanciées
      - Affecter une valeur à cette variable
  - Exemple pour le $P_m||C_{max}$
    - L <- liste des tâches triées par durée croissante
    - Tant que L ≠ ∅ faire
      - Sélectionner la tâche i (premier élément de L) et la retirer de L
      - Placer la tâche sur la machine libre au plus tôt

Garantie de performance ?

=> Gap max de 1/2

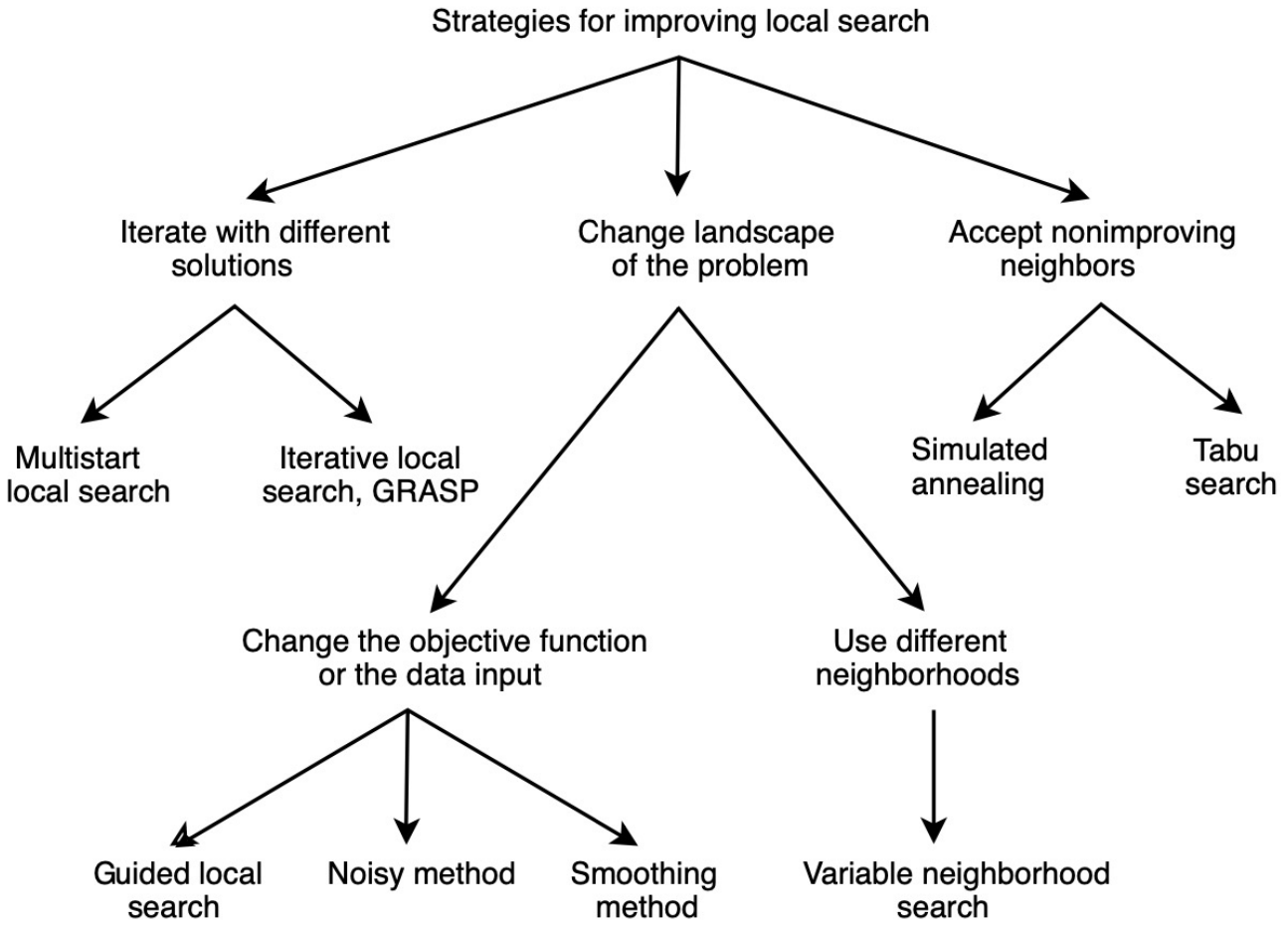


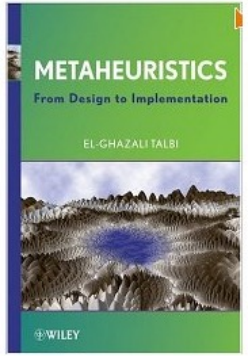
GAP $=(F^h-F^{opt})/F^{opt}$
$=(3k-2k)/2k$
$=1/2$

Métaheuristique de type parcours

- **Concepts communs :**
  - Exploration d'un voisinage à partir d'une solution et d'un opérateur de voisinage
  - Eléments importants du voisinage :
    - Opérateur de voisinage
    - Exhaustif ou non
    - Déterministe ou stochastique
    - Taille du voisinage
    - Cas des solutions infaisables
    - Complexité/temps de calcul
    - Voisinage « adaptatif » ou non
    - Quelques exemples au tableau : SWAP, 2-Opt, …
- **Pour chaque métaheuristique, il existe plusieurs versions, déclinaisons, adaptations, améliorations, etc.**

Métaheuristique de type parcours





Métaheuristique de type parcours

- **Recherche Locale :**
  - S, S’ <- Calcul d’une solution initiale
  - Faire
    - N(S) <- Générer l’ensemble du voisinage de S
    - S’ <- Meilleure solution de N(S)
    - Si f(S’) < f(S) alors
      - S <- S’
  - Tant que (S ≠ S’)
- **/!\ Ne pas implémenter un algorithme tel quel**
  - Des étapes peuvent être fusionnées, simplifiées, etc.
- **Amélioration simple : multi-start**
- **Deux stratégies d’exploration : « best » or « first » amélioration**

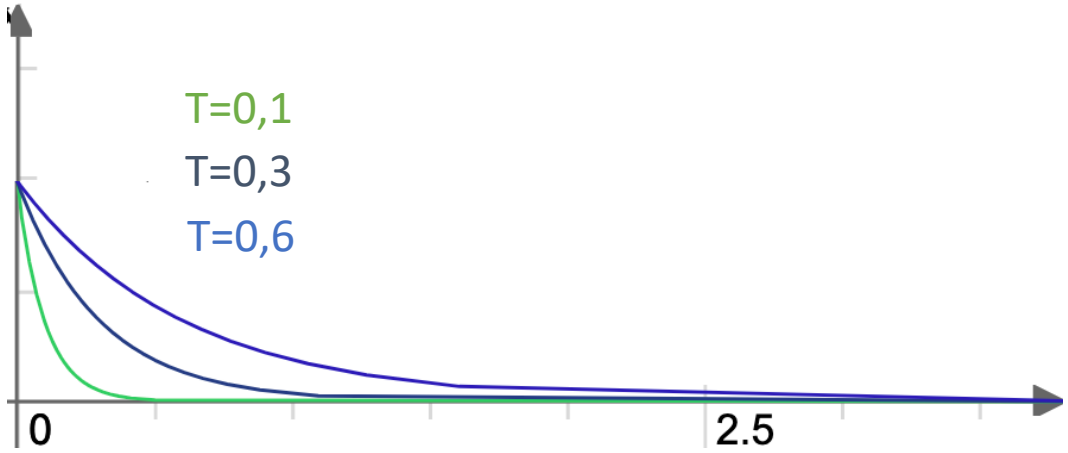
Métaheuristique de type parcours

- **Recuit simulé**
  - Inspiration de la métallurgie :
    - << La méthode vient du constat que le refroidissement naturel de certains métaux ne permet pas aux atomes de se placer dans la configuration la plus solide. La configuration la plus stable est atteinte en maîtrisant le refroidissement et en le ralentissant par un apport de chaleur externe, ou bien par une isolation. >>

Métaheuristique de type parcours

- **Recuit simulé**
  - S <- Calcul d'une solution initiale
  - T <- T_{max}
  - Faire
    - Faire
      - S' <- Générer une solution aléatoire voisine de S
      - Δ <- f(S')-f(S)
      - Si Δ $\leq$ 0 alors
        - S <- S'
      - Sinon
        - S <- S' avec une probabilité de $e^{-\Delta/T}$
    - Tant qu'une certaine condition d'équilibre //souvent un nombre d'itération
    - Mettre à jour la Température T en la réduisant
  - Tant que la condition d'arrêt finale n'est pas satisfaite

//nombre d'itération, qualité de solution, temps de calcul, etc.



Métaheuristique de type parcours

- **Recherche Tabou**
  - S, S’ <- Calcul d’une solution initiale
  - ListeTabou <- Ø
  - Faire
    - N(S) <- Générer l’ensemble du voisinage de S
    - S’ <- Meilleure solution de N(S) et ∉ ListeTabou
    - Mise à jour de ListeTabou avec S’
    - Si f(S’) < f(S) alors
      - S <- S’
    - [Phase d’intensification]
    - [Phase de diversification]
  - Tant que la condition d’arrêt finale n’est pas satisfaite

Métaheuristique de type parcours

- **Recherche Tabou**
  - Quelques points importants :
    - ListeTabou : théoriquement contient toutes les solutions déjà visitées mais en pratique ce n'est pas le cas (problème de temps de calcul), les « mouvements améliorant » sont stockés à la place et la taille est limitée (statique ou dynamique).
  - Critère d'arrêt : nombre d'itération maximal, nombre d'itération sans amélioration de la meilleure solution connues, temps de calcul maximal, etc.
  - Critère d'aspiration : autoriser l'exploration d'une solution taboue mais qui améliore la meilleure solution connue.
  - Phase d'intensification : lors de l'exploration de région « intéressante », agir pour intensifier la recherche dans la région (vide la liste tabou, agir sur la méthode de voisinage, etc.).
  - Phase de diversification : contraire à l'intensification, essayer d'explorer des régions pas ou peu explorées (random restart, random move, agir sur la fonction objectif, etc.)

Métaheuristique de type parcours

- **Recherche Locale itérée :**
  - S <- Calcul d'une solution initiale
  - Sc <- Recherche locale(S)
  - Faire
    - S <- Perturbation(Sc). //Consiste à garder une partie de la solution Sc et de modifier aléatoirement l'autre partie de Sc
    - S <- Recherche locale(S)
    - Si un critère d'acceptation est satisfait alors //Peut servir de phase d'intensification (on accepte que si améliore) ou de phase de diversification
      - Sc <- S
  - Tant que la condition d'arrêt finale n'est pas satisfaite

Métaheuristique de type parcours

- **Descente à voisinage variable :**
  - S <- Calcul d'une solution initiale
  - k <- 1
  - Tant que ($k \leq K_{max}$) faire
    - S' <- Meilleure solution du voisinage de S avec le $k^{ième}$ opérateur
    - Si f(S') < f(S) alors
      - S <- S'
      - k <- 1
    - Sinon
      - K <- k + 1
- **Suppose un ensemble de K_{max} opérateurs de voisinage différents triés selon un « certain » ordre (statique ou dynamique).**

Métaheuristique de type parcours

- **Recherche à voisinage variable :**
  - S <- Calcul d'une solution initiale
  - Faire
    - k <- 1
    - Faire
      - S' <- Shake en utilisant le $k^{ième}$ opérateur et en partant de la solution S
      - S'' <- Recherche Locale (S')
      - Si f(S'') < f(S) alors
        - S <- S''
        - k <- 1
      - Sinon
        - K <- k + 1
    - Tant que ($k \leq K_{max}$)
  - Tant que la condition d'arrêt finale n'est pas satisfaite

Métaheuristique de type parcours

- **GRASP :**
  - Faire
    - S <- Calcul d'une solution initiale à l'aide d'une heuristique semi-randomisée*
    - S <- Recherche Locale (S)
  - Tant que la condition d'arrêt finale n'est pas satisfaite
- * :
  - Tant que toutes les variables ne sont pas instanciées faire
    - Construire une RCL « Restricted candidate list » en fonction d'une heuristique
    - Prendre au hasard un élément de RCL
    - Instancier la ou les variable en fonction de cet élément

Métaheuristique de type population

- **Concepts communs :**
  - Explorer plusieurs solutions à la fois pour garder une certaine diversité.
  - Population ≈ mémoire
  - Eléments importants :
    - Importance de la diversité de la population
    - Assurer une convergence vers les bonnes solutions en évitant les convergences prématurées
    - Maîtriser « l'aléatoire » souvent présent (ne pas avoir une méthode instable)
    - Taille de la population
    - Cas des solutions infaisables
    - Critère d'arrêt
  - Pour chaque métaheuristique, il existe plusieurs versions, déclinaisons, adaptations, améliorations, etc.

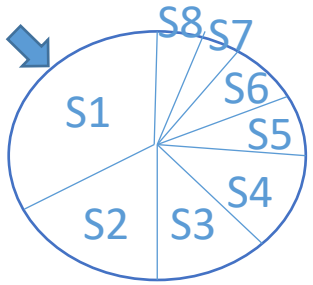
Métaheuristique de type population

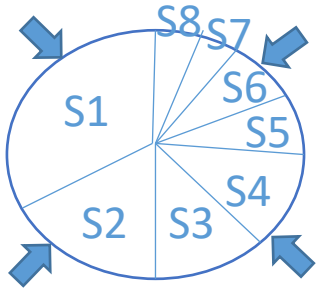
- **Algorithme évolutionnaire :**
  - P_0 <- Population Initiale
  - t <- 0
  - Tant que la condition d'arrêt finale n'est pas satisfaite faire :
    - Evaluer(P_t)
    - P'_t <- Sélection(P_t)
    - P'_t <- Reproduction(P'_t , Evaluer(P'_t))
    - P'_t <- Remplacer(P'_t , P_t)
    - t <- t+1

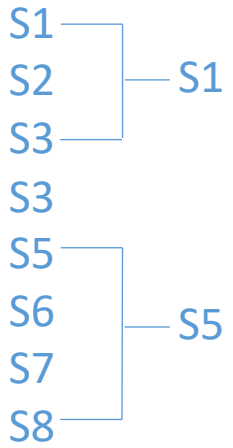
- **Plusieurs déclinaisons possibles dont l'algorithme génétique**

Métaheuristique de type population

- **Les éléments importants d'un algorithme évolutionnaire :**
  - La sélection : l'idée est de sélectionner les solutions avec une probabilité qui dépend de la qualité de la solution. Les opérateurs plus connus : par roulette, échantillonnage universel stochastique, par tournois, par rang, etc.

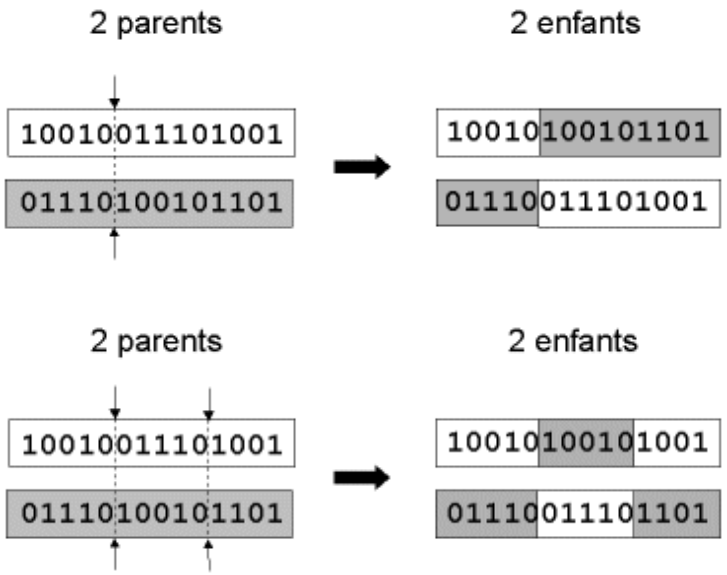


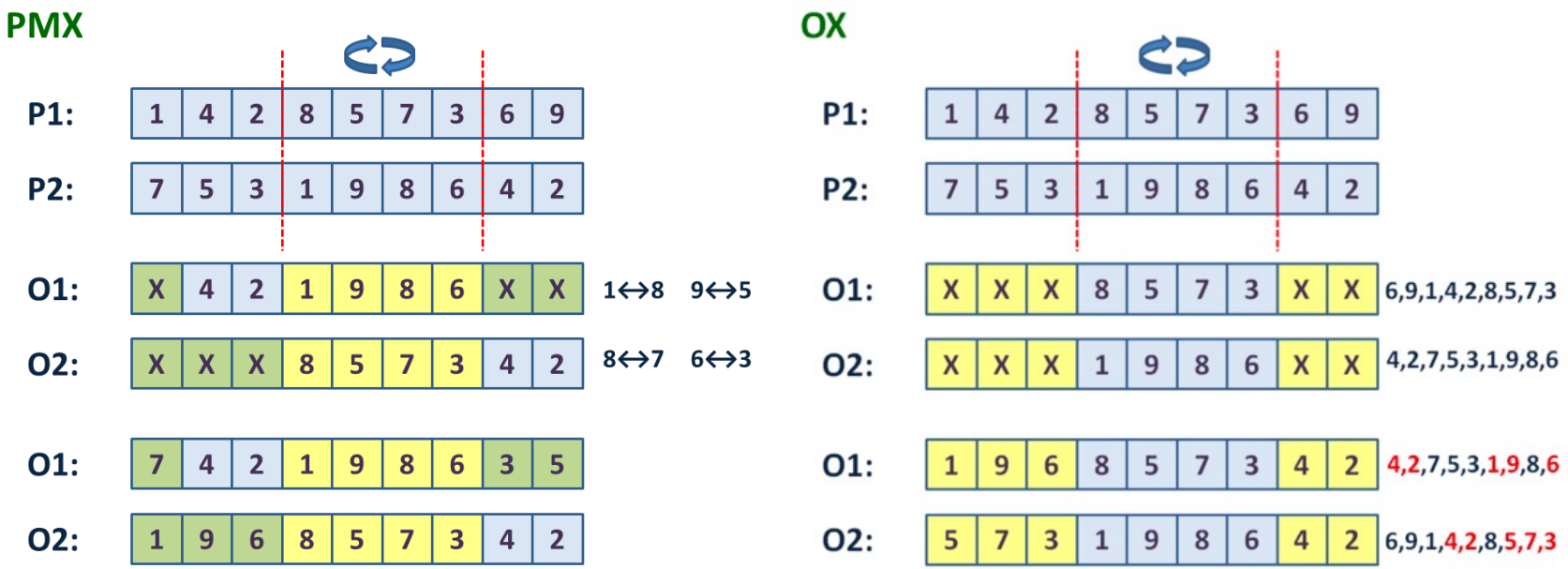




Métaheuristique de type population

- **Les éléments importants d'un algorithme évolutionnaire :**
  - L'évaluation : très souvent la fonction objectif.
  - La reproduction : l'idée est de créer de nouvelles solutions à partir de nouvelles solutions (souvent 2 pour 2), chaque nouvelle solution prend « une partie » de chaque solution parente. Plusieurs opérateurs dont les plus connues sont : 1-point, 2-point, 3-point, …, PMX et OX. Ces opérateurs dépendent fortement du codage des solutions.





Métaheuristique de type population

- **Les éléments importants d'un algorithme évolutionnaire :**
  - Remplacement de la population : l'idée est de garder un nombre de solutions de la population constant et donc de sélectionner un sous ensemble de solutions parmi les « anciennes » et « nouvelles solutions ». Plusieurs manières d'opérer :
    - On ne garde que les nouvelles solutions
    - On garde 50% des meilleurs solutions
    - On garde X% des meilleurs anciennes solutions et (100-X)% des meilleurs nouvelles solutions
  - Opérateur de mutation : consiste à « muter » (=appliquer une modification aléatoire dans le codage d'une solution) chaque nouvelle solution créée et avec une (faible) probabilité. Cet opérateur a pour objectif de diversifier la recherche.

Métaheuristique de type population

- **Les éléments importants d'un algorithme évolutionnaire :**
  - Gestion de la population : il est important de garder une population hétérogène et de bonne qualité. Pour cela, il peut être intéressant d'ajouter des procédures de gestion de population comme par exemple :
    - Ajouter une nouvelle solution à la population seulement si elle n'est pas similaire à une autre solution présente dans la population courante (définir une fonction de similarité).
    - Injecter de nouvelles solutions quand il est nécessaire.
    - Ajuster dynamiquement la taille de la population.

Métaheuristique de type population

- **Scatter search :**
  - P <- Population Initiale
  - P <- Amélioration de chaque solution de P (ex. par une recherche locale)
  - R <- Construction de l'ensemble de solution de référence à partir de P
  - Tant que la condition d'arrêt finale n'est pas satisfaite faire :
    - E <- Génération d'un sous ensemble de solution à partir de R
    - Tant qu'une condition d'arrêt n'est pas satisfaite faire :
      - E' <- combiner les solutions de E
      - E'' <- Amélioration de chaque solution de E'(ex. par une recherche locale)
    - R <- Mise à jour de l'ensemble de solution de référence à partir de E''

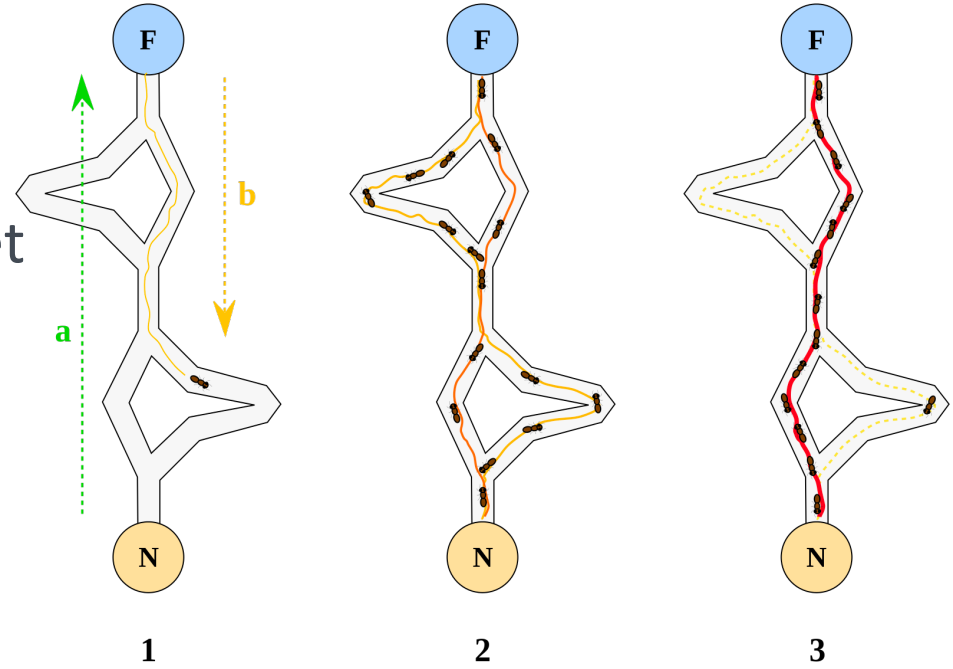
- **Méthode déterministe**

Métaheuristique de type population

- **Scatter search :**
  - Amélioration de chaque solution doit être rapide (recherche locale)
  - La construction et la mise à jour de l'ensemble de solution de référence consiste souvent à prendre garder deux types de solutions :
    - Les meilleures
    - Les plus diversifiées (définir une fonction de similarité)
  - La génération d'un sous ensemble de solution à partir de R est similaire au processus de sélection d'un algorithme évolutionnaire mais doit être déterministe.
  - La combinaison de solutions est similaire au processus de croisement d'un algorithme évolutionnaire.

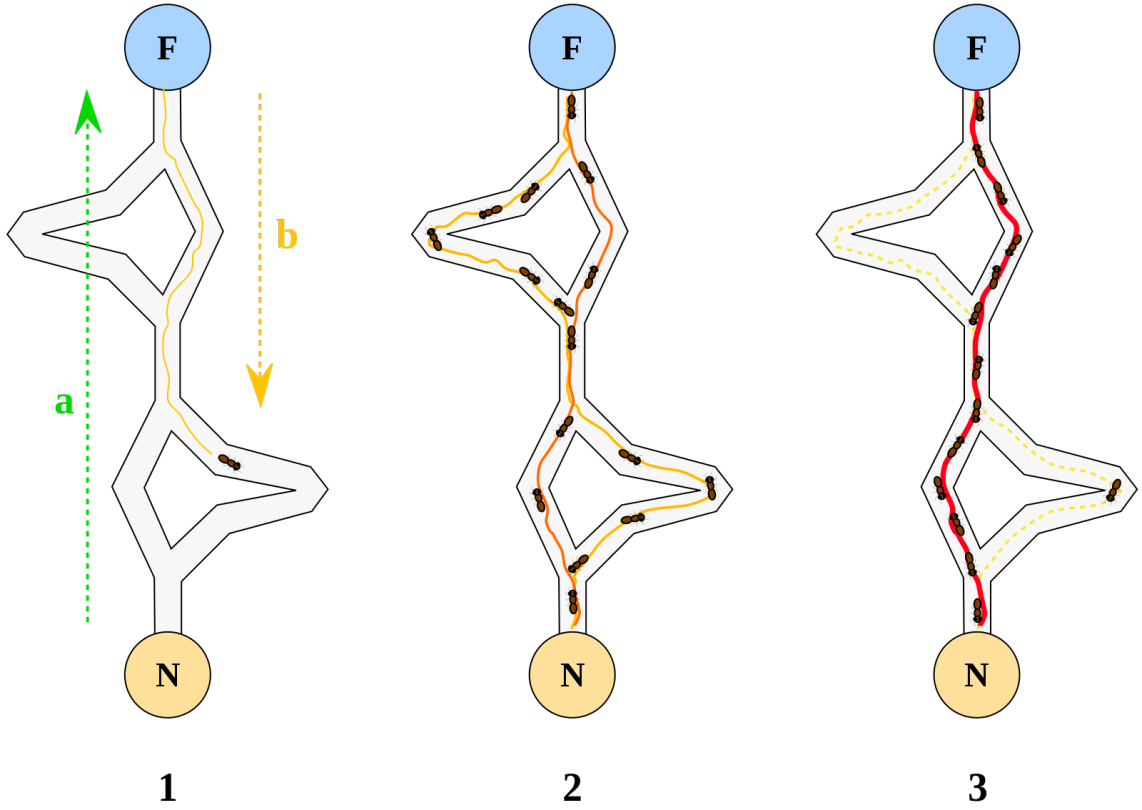
Métaheuristique de type population

- **Algorithme de fourmis :**
  - Observation de départ « collectivement, elles sont capable de trouve le plus court chemin entre leur nid et la nourriture »
  - Principe de phéromones :
    - dépôt de phéromone attire d'autres fourmis qui vont la **renforcer** à leur tour
    - Les phéromones s'**évaporent** avec le temps



Métaheuristique de type population

- **Algorithme de fourmis :**
  - Observation de départ « collectivement, elles sont capable de trouve le plus court chemin entre leur nid et la nourriture »
  - Principe de phéromones :
    - dépôt de phéromone attire d'autres fourmis qui vont la **renforcer** à leur tour
    - Les phéromones s'**évaporent** avec le temps



Métaheuristique de type population

- **Algorithmes de fourmis :**
  - Initialisation des « pistes de phéromones »
  - Tant que la condition d'arrêt finale n'est pas satisfaite faire :
    - Pour chaque fourmi
      - Construire une solution en utilisant les pistes de phéromones
      - Mise à jour des phéromones :
        - Evaporation (diminution des phéromones)
        - Renforcement (augmentation des phéromones qui ont permis la construction de bonnes solutions)

- **De nombreuses interprétations/implémentations possibles selon le problème**
- **Les pistes phéromones servent de mémoire (caractériser les bonnes solutions) et vont être utilisés pour construire des solutions par les fourmis.**

Métaheuristique de type population

- **Algorithmes de fourmis pour le PVC :**

Initialisation : $\tau_{ij} \leftarrow \tau_0 \forall (i,j) \in \{1,\ldots,n\}^2$, placer aléatoirement chaque fourmi sur une ville
pour $t = 1$ à $t = t_{\max}$ **faire**
  pour chaque fourmi k **faire**
    Construire un chemin $T^k(t)$
    Calculer la longueur $L^k(t)$ de ce chemin
  finpour
  Soient T^+ le meilleur chemin trouvé et L^+ la longueur correspondante
  Mettre à jour les traces de phéromones
finpour
retourner T^+ et L^+

Métaheuristique de type population

- **Algorithmes de fourmis pour le PVC :**
  - Le chemin $T^k(t)$ est construit avec une probabilité :

$$p_{ij} = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\nu_{ij}]^{\beta}}{\sum_{l \notin L_k(i)} [\tau_{il}(t)]^{\alpha} \cdot [\nu_{il}]^{\beta}} & \text{si } j \notin L_k(i) \\ 0 & \text{sinon} \end{cases}$$

Où $L_k(i)$ sont les arcs visités de la fourmi k et v_{ij} est l'inverse de la distance

  - Mise à jour des phéromones :

$$\tau_{ij} \leftarrow \tau_{ij}(1-\rho) + \sum_{k=1}^{m} \Delta\tau_{ij}^k$$

Où : $\Delta\tau_{ij}^k = \begin{cases} \frac{Q}{L^k} & \text{si la fourmi } k \text{ est passée par l'arc } (i,j) \\ 0 & \text{sinon} \end{cases}$

Métaheuristique de type population

- **Et bien d'autres ….**
  - Les essaims particulaires
  - L'évolution différentielle
  - Réseau immunitaire artificiel
  - L'algorithmes des abeilles
  - ….

Hybridation de Métaheuristiques

- **Classification**
  - Niveau d'imbrication :
    - Bas niveau : une des méthodes est utilisée comme composant d'une autre méthode
    - Haut niveau : les méthodes sont indépendantes (elles n'interviennent pas dans le fonctionnement de l'autre
  - Niveau de coopération :
    - Séquentiel : les méthodes sont exécutées les uns à la suite des autres (les sorties d'une méthode servant aux entrées aux autres méthodes
    - Coopératif : les méthodes agissent comme des agents qui parcourent l'espace de recherche de manière indépendante puis coopèrent en mettant en commun les résultats de leurs recherches.

- **L'hybridation d'une métaheuristique et d'une méthode exacte est appelée matheuristique.**

Hybridation de Métaheuristiques

- **Quelques exemples/idées :**
  - Coupler la grand diversification de recherche des métaheuristiques type population avec l'intensification de recherches des métaheuristiques par voisianges
  - Explorer les solutions d'un problème à l'aide d'une métaheuristique et sous-traiter la résolution d'autres sous-problèmes à d'autres méthodes.
  - Algorithme génétique + recherche locale = algorithme mémétique
  - Méthode arborescente tronquée avec recherche locale pour améliorer les bornes aux nœuds intermédiaires
  - Métaheuristque + machine learning
  - …