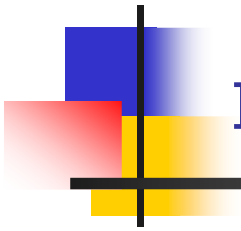
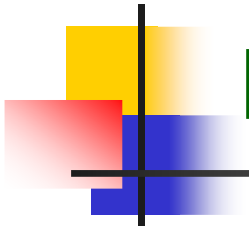


IFT1575 Modèles de recherche opérationnelle (RO)

8. Optimisation combinatoire et métaheuristiques



Heuristique et métaheuristique

- Un algorithme heuristique permet d'identifier au moins une solution réalisable à un problème d'optimisation, mais sans garantir que cette solution soit optimale
- Exemple : appliquer une fois la méthode du gradient à un modèle de programmation non convexe
- Une **métaheuristique** est une stratégie générale, applicable à un grand nombre de problèmes, à partir de laquelle on peut dériver un algorithme heuristique pour un problème particulier

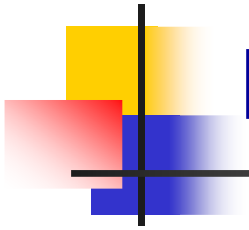
Optimisation combinatoire

- Domaine qui étudie les problèmes de la forme :

$$\max_{x \in X} f(x)$$

où X est un ensemble fini, mais de très grande taille

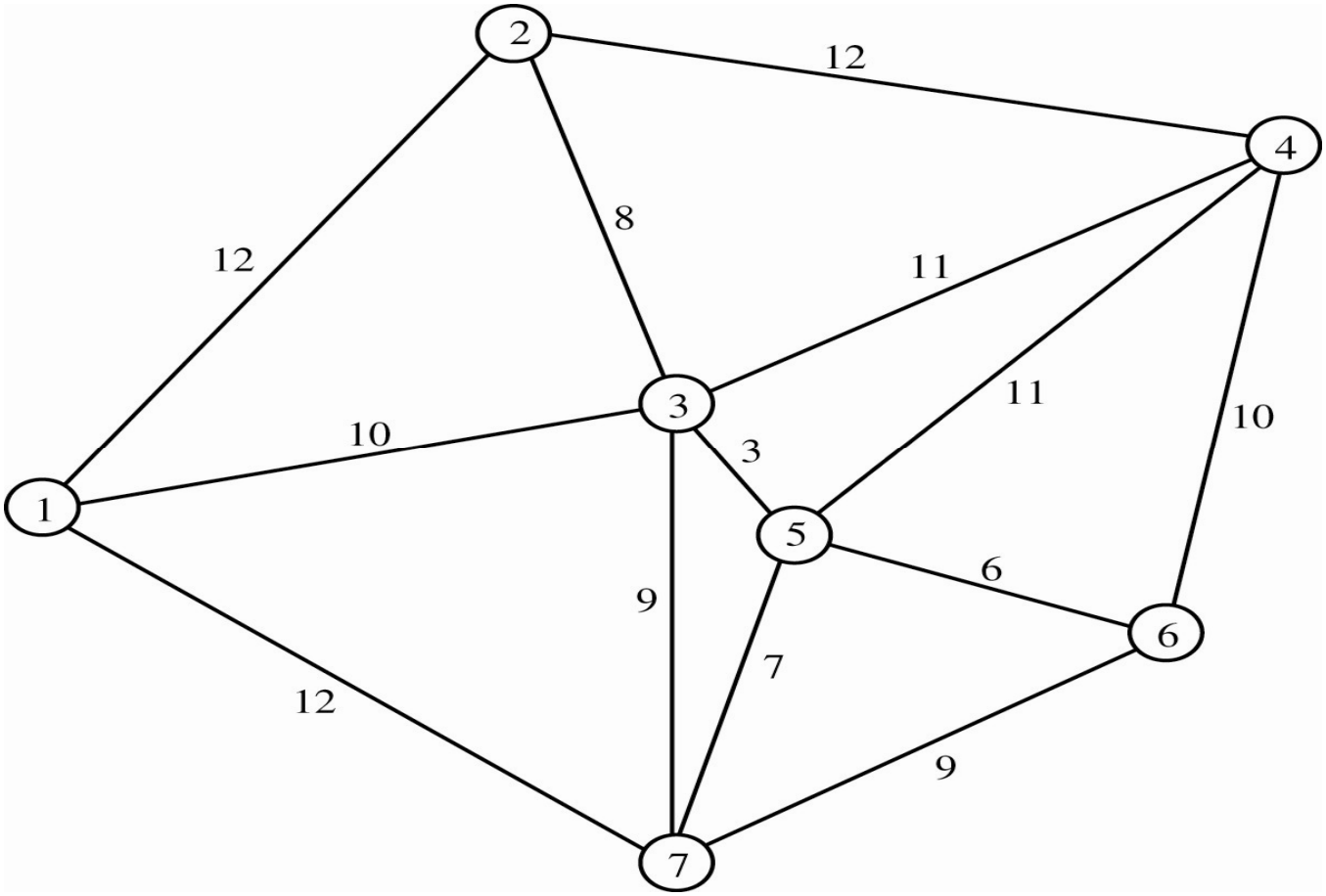
- Exemples :
  - Problème de l'arbre partiel minimum : X est l'ensemble de tous les arbres partiels possibles
  - Programmation linéaire : X est l'ensemble de tous les points extrêmes du domaine réalisable
- Dans ces deux exemples, on peut faire beaucoup mieux que d'énumérer toutes les solutions
- Pour d'autres problèmes, on ne sait pas vraiment faire beaucoup mieux!!!

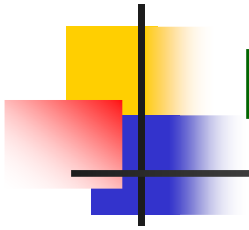


Problème du voyageur de commerce

- Un voyageur de commerce doit visiter un certain nombre de villes
- Il doit visiter chaque ville une et une seule fois
- Étant donné des distances entre chaque paire de villes, il doit minimiser la distance totale parcourue
- On peut représenter ce problème par un graphe : chaque ville correspond à un sommet et chaque arête à une paire de villes pouvant être visitées l'une à la suite de l'autre
- Le problème correspond à trouver un tour complet (circuit Hamiltonien) dans ce graphe qui minimise la somme des distances

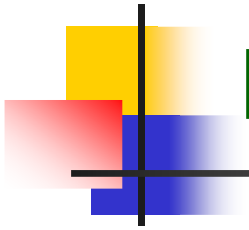
Exemple





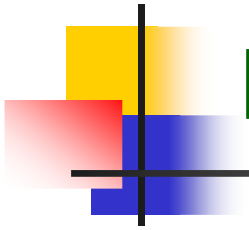
Méthodes exactes

- Problème d'optimisation combinatoire : X est l'ensemble des tours possibles
- Dans un graphe complet, il y a (n-1)!/2 tours possibles, donc X est de très grande taille
- Il n'existe pas d'algorithme efficace comme pour le problème de l'arbre partiel minimum
- Le mieux (empiriquement) est d'utiliser la programmation en nombres entiers :
  - Dantzig-Fulkerson-Johnson 1954!
  - Aujourd'hui, on peut résoudre des problèmes ayant plus de 10000 villes : http://www.tsp.gatech.edu/



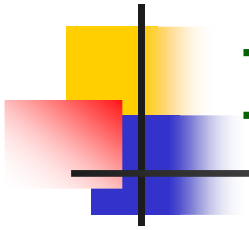
Méthodes heuristiques

- Il faut être conscient que ces méthodes exactes peuvent prendre beaucoup de temps, surtout lorsque les problèmes sont de grande taille
- Une autre approche consiste à utiliser des méthodes heuristiques visant à identifier rapidement de bonnes solutions
- On les classe souvent en deux catégories :
  - Méthodes constructives : permettent de construire une solution réalisable
  - **Méthodes d'amélioration** : permettent de visiter plusieurs solutions réalisables en tentant d'améliorer la valeur de l'objectif (l'objet de notre étude)



Méthode de montée (descente)

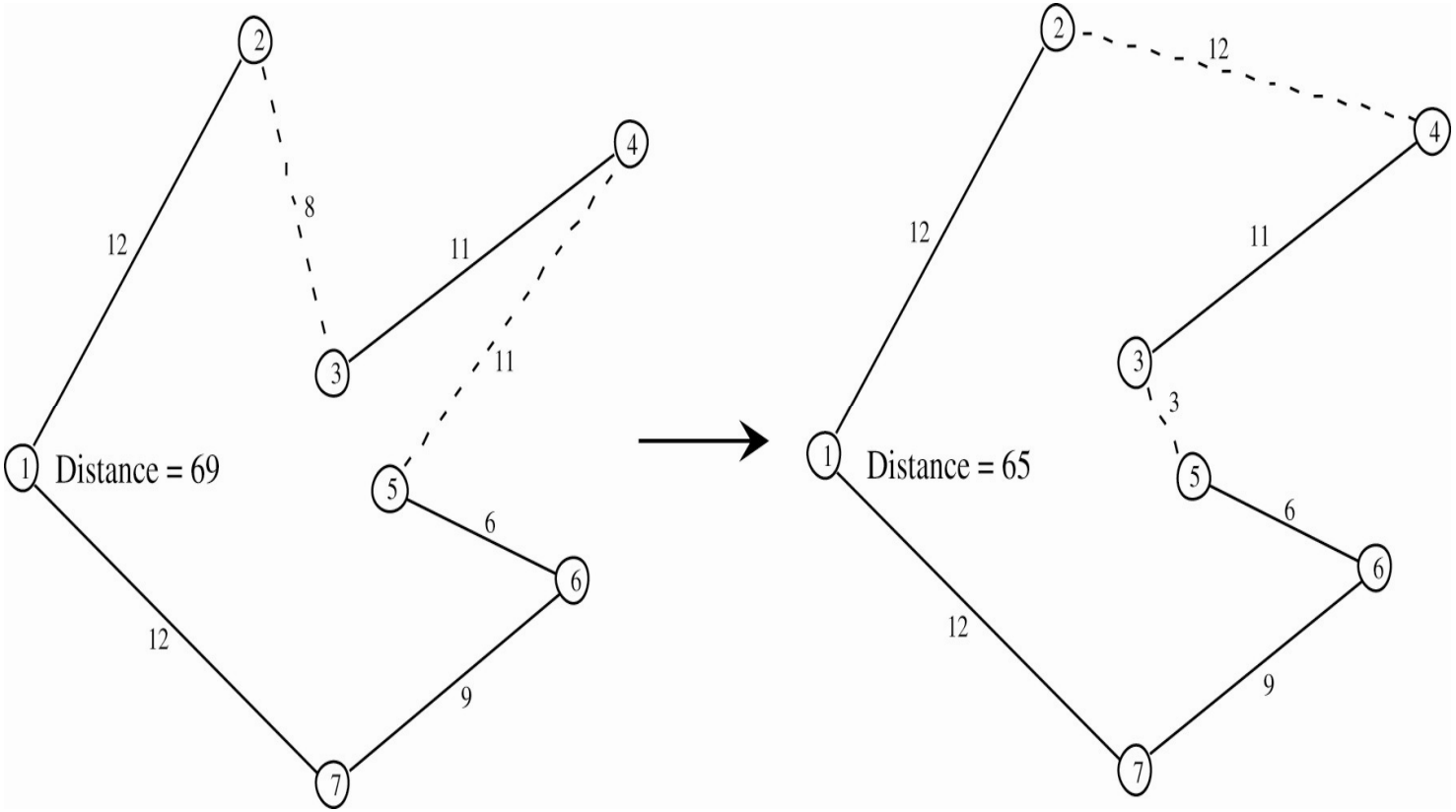
- Étant donné une solution réalisable initiale, on tente de l'améliorer par une modification d'un certain type
- Dans l'ensemble de toutes les modifications d'un certain type, on choisit celle qui améliore le plus la valeur de l'objectif (s'il y en a une!)
- Cette approche est une métaheuristique, appelée méthode de montée (max) ou de descente (min)
- Exemple : méthode du gradient en programmation non linéaire
  - Modification : faire un pas dans la direction du gradient
  - On calcule la modification (t*) qui améliore le plus la valeur de l'objectif

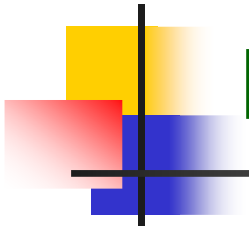


Inversion de sous-tours

- Pour le problème du voyageur de commerce, on peut définir plusieurs types de modification
- Un type de modification possible consiste à inverser l'ordre de visite d'une sous-séquence de villes
- Par exemple, si la séquence de visite des villes est 1-2-3-4-5-6-7-1 et que nous choisissions d'inverser la sous-séquence 3-4, la solution 1-2-4-3-5-6-7-1 serait obtenue suite à cette modification
- Sur notre exemple, on peut vérifier que la distance totale passe de 69 à 65 suite à cette modification

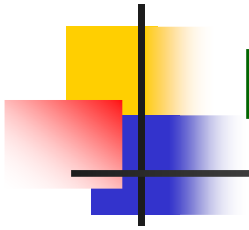
Inversion de sous-tours : exemple





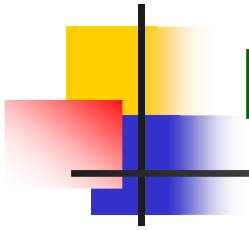
Descente par inversion de sous-tours

- Identifier une solution réalisable initiale
- Considérer toutes les inversions de sous-tours et choisir celle qui améliore le plus la distance totale parcourue par le nouveau tour ainsi obtenu
- Arrêter s'il n'y a aucune inversion de sous-tours qui permette d'améliorer la distance totale parcourue
- Cette méthode est intéressante, mais elle ne garantit pas de trouver une solution optimale
- Elle identifie plutôt un optimum local (par rapport au type de modification utilisé)



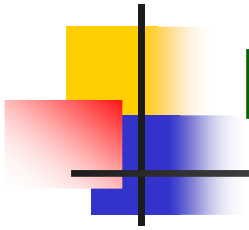
Exemple

- Solution initiale : 1-2-3-4-5-6-7-1 (69)
- Inversions de sous-tours (les autres possibilités ne mènent pas à une solution réalisable) :
  - 1-3-2-4-5-6-7-1 : 68
  - 1-2-4-3-5-6-7-1 : 65
  - 1-2-3-5-4-6-7-1 : 65
  - 1-2-3-4-6-5-7-1 : 66
- Il y a deux modifications qui diminuent le plus la distance : on choisit la première



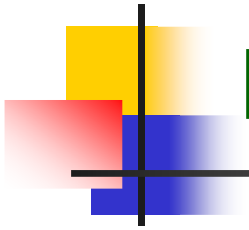
Exemple (suite)

- Solution courante : 1-2-4-3-5-6-7-1 (65)
- Inversions de sous-tours :
  - 1-2-3-4-5-6-7-1 : 69 (solution précédente!)
  - 1-2-4-6-5-3-7-1 : 64
- Il n'y a qu'une seule modification qui diminue la distance totale : on la choisit
- Inversions à partir de 1-2-4-6-5-3-7-1 :
  - 1-2-4-3-5-6-7-1 : 65 (solution précédente!)
  - 1-2-4-6-5-7-3-1 : 66
- Aucune modification n'améliore la valeur de l'objectif: on arrête



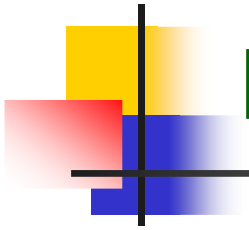
Exemple (suite)

- Pourtant, la solution obtenue de valeur 64 n'est pas optimale : la solution 1-2-4-6-7-5-3-1 est de valeur 63 et on peut vérifier qu'elle est optimale
- Plusieurs métaheuristiques utilisent la même approche (par type de modifications, ou **voisinage**) que la descente, mais tentent d'éviter de demeurer piégé dans un minimum local (recherche avec tabous, recuit simulé)
- D'autres tentent de combiner plusieurs solutions (**populations**) pour en générer de nouvelles (algorithmes génétiques)



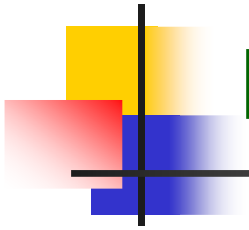
Recherche avec tabous

- L'idée de cette méthode est de permettre des modifications qui n'améliorent pas la valeur de l'objectif
- Toutefois, on choisira toujours la meilleure modification possible
- Mais nous venons de voir qu'une des modifications possibles nous ramène à la solution précédente
- Il faut donc changer la définition de l'ensemble des modifications possibles pour interdire celles qui nous ramènent à la solution précédente



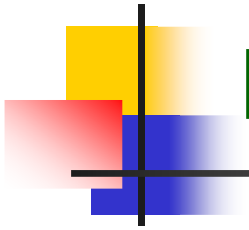
Recherche avec tabous (suite)

- À cette fin, on conservera une liste des dernières modifications effectuées en rendant **taboue** (en interdisant) la modification inverse
- Cette liste taboue peut être vue comme une mémoire à court terme permettant de guider la recherche
- A chaque itération, on choisit la meilleure modification possible (excluant celles qui sont taboues), puis on met à jour cette liste en ajoutant la modification inverse de celle effectuée



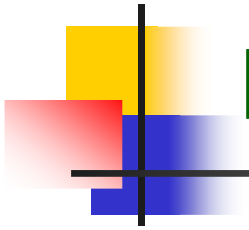
Recherche avec tabous (suite)

- Contrairement à la méthode de descente, il n'y a pas de critère d'arrêt simple
- Typiquement, on utilise une combinaison des critères suivants :
  - Nombre maximum d'itérations
  - Temps limite
  - Nombre d'itérations successives sans amélioration
  - Il n'y a plus de modification possible
- Adaptation de cette métaheuristique pour résoudre un problème particulier : structure de **voisinage** + implantation de la **liste taboue**



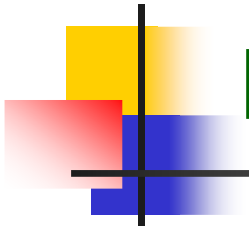
Retour au voyageur de commerce

- **Voisinage** : inversion de sous-tours, ce qui implique l'ajout de deux liens et l'élimination de deux liens
- **Liste taboue** : les deux liens ajoutés sont insérés dans la liste taboue; une modification est taboue si les deux liens à éliminer sont dans la liste taboue
- On ne conservera dans la liste taboue que les liens ajoutés lors des deux dernières itérations : on dit que la longueur de la liste taboue est 4
- On arrête l'algorithme lorsque trois itérations consécutives sans amélioration ont été exécutées (ou lorsqu'il n'y a plus de modification possible)



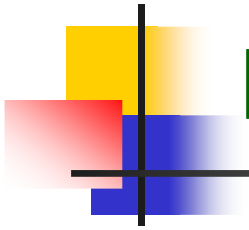
Exemple

- Reprenons le même exemple : la recherche avec tabous effectue les mêmes modifications que la descente, mais gère également la liste taboue
- Solution initiale : 1-2-3-4-5-6-7-1 (69)
- Itération 1 :
  - Inversion de la sous-séquence 3-4 → ajout des liens 2-4 et 3-5 → Liste taboue = 2-4, 3-5
  - Nouvelle solution : 1-2-4-3-5-6-7-1 (65)
- Itération 2 :
  - Inversion de la sous-séquence 3-5-6 → ajout des liens 4-6 et 3-7 → Liste taboue = 2-4, 3-5, 4-6, 3-7
  - Nouvelle solution : 1-2-4-6-5-3-7-1 (64)



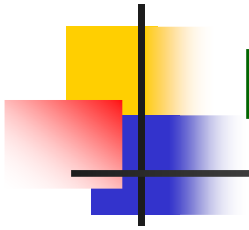
Exemple (suite)

- À partir de la solution courante 1-2-4-6-5-3-7-1, il y a deux modifications :
  - Inversion de la sous-séquence 6-5-3 → élimination des liens 4-6 et 3-7, mais les deux sont dans la liste taboue : modification taboue
  - Inversion de la sous-séquence 3-7 → ajout des liens 5-7 et 3-1, élimination des liens 5-3 et 7-1 → Liste taboue = 4-6, 3-7, 5-7, 3-1
  - Nouvelle solution : 1-2-4-6-5-7-3-1 (66 > 64)
- Voyons cet algorithme implanté dans IOR Tutorial



Recuit simulé

- Comme dans la recherche avec tabous, on permet des modifications qui n'améliorent pas la valeur de l'objectif
- Au lieu de choisir la modification la plus intéressante parmi toutes les modifications possibles, on en choisit une **au hasard**
- On va biaiser le choix vers des modifications qui améliorent ou tout au moins ne détériorent pas trop la valeur de l'objectif



Recuit simulé (suite)

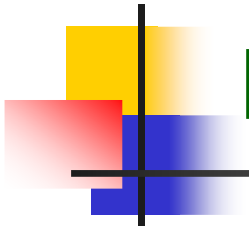
- A partir d'une solution courante, on effectue une modification au hasard qui nous amène à une solution candidate
- Z_c = valeur de l'objectif pour la solution courante
- Z_n = valeur de l'objectif pour la solution candidate
- T = paramètre appelé **température**, qui prend une valeur élevée lors des premières itérations, puis diminue au fil des itérations
- Supposons un problème de maximisation

Recuit simulé (suite)

- Si $Z_n \geq Z_c$ accepter la solution candidate
- Si $Z_n < Z_c$ accepter la solution candidate avec une probabilité e^x, où $x = (Z_n - Z_c)/T$
- Si T est élevé, x est près de 0 et alors e^x est élevé : on a de fortes chances d'accepter la modification même si elle mène à une moins bonne solution
- Si T est petit, x est près de 0 lorsque Z_n est près de Z_c et alors e^x est élevé : on a de fortes chances d'accepter une modification qui ne détériore pas trop la valeur de l'objectif
- Pour un problème de minimisation, on inverse Z_n et Z_c dans la formule

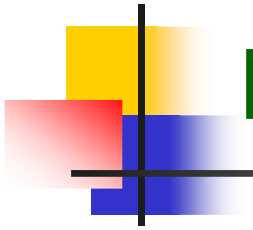
Recuit simulé (suite)

$x = (Z_n - Z_c)/T$	Prob(acceptation) = e^x
-0,01	0,990
-0,1	0,905
-0,25	0,779
-0,5	0,607
-1	0,368
-2	0,135
-3	0,050
-4	0,018
-5	0,007



Retour au voyageur de commerce

- Voisinage : inversion de sous-tours
- Choix d'une modification au hasard : on choisit au hasard le début et la fin de la sous-séquence à inverser (si l'inversion n'amène pas vers une solution réalisable, on recommence jusqu'à ce qu'on obtienne une solution candidate réalisable)
- La température initiale est fixée à $T = 0{,}2.Z_c$, ce qui est relativement élevé comparé à $Z_n - Z_c$
- Ensuite, on divise la température par 2 à toutes les cinq itérations
- On arrête après 25 itérations

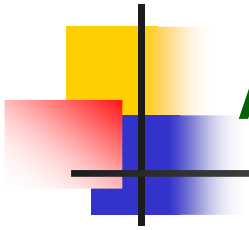


Exemple

- Solution initiale : 1-2-3-4-5-6-7-1; $Z_c = 69$
- Température initiale : $T = 0{,}2.69 = 13{,}8$
- Modification choisie au hasard : inverser la sous-séquence 3-4 $\rightarrow$ solution candidate 1-2-4-3-5-6-7-1; $Z_n = 65$
- Puisque $Z_n \leq Z_c$, on accepte cette modification
- Solution courante : 1-2-4-3-5-6-7-1; $Z_c = 65$
- Modification choisie au hasard : inverser la sous-séquence 3-5-6$\rightarrow$ solution candidate 1-2-4-6-5-3-7-1; $Z_n = 64$: on l'accepte

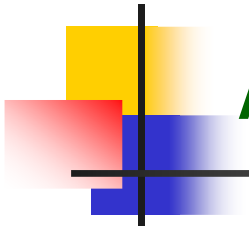
Exemple (suite)

- Solution courante : 1-2-4-6-5-3-7-1; $Z_c = 64$
- Modification choisie au hasard : inverser la sous-séquence 3-7 $\rightarrow$ solution candidate 1-2-4-6-5-7-3-1; $Z_n = 66$
- Puisque $Z_n > Z_c$, Prob(acceptation) = $e^{(-2/13,8)} = 0,865$
- On génère un nombre selon une loi U[0,1] : si ce nombre est < 0,865, on accepte la modification, sinon, on la refuse
- Voyons cet algorithme dans IOR Tutorial
- La même approche peut être adaptée pour résoudre des modèles de programmation non convexe (voir dans IOR Tutorial)



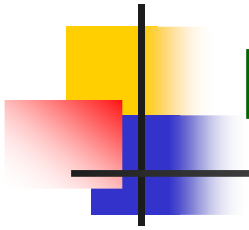
Algorithmes génétiques

- Ces métaheuristiques sont basées sur une analogie entre le processus de génération d'une population de solutions et la théorie de l'évolution
- Les solutions (individus) survivent et deviennent **parents** en croisant leurs caractéristiques (gènes), ce qui donne de nouvelles solutions, les **enfants**
- Des **mutations** peuvent également intervenir permettant aux solutions d'acquérir des caractéristiques qui ne se retrouvent pas chez leurs parents



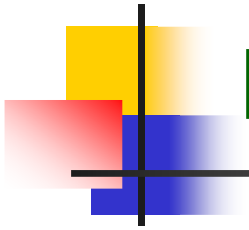
Algorithmes génétiques (suite)

- Étant donné une population de solutions, on évalue la valeur de l'objectif (le degré d'adaptation) pour chacune
- En biaisant le choix vers les solutions les mieux adaptées, on choisit les solutions qui seront croisées pour devenir des parents
- Chaque couple de parents donnera naissance à deux enfants en croisant leurs caractéristiques (et en permettant des mutations de temps en temps)
- Ajouter les enfants à la population et éliminer de la population les solutions les moins adaptées



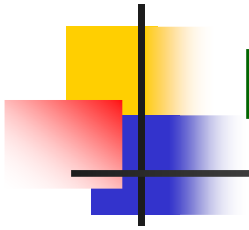
Retour au voyageur de commerce

- Les solutions qui forment la population initiale sont choisies au hasard
- Des parents sont choisis au hasard (mais on choisit davantage parmi les mieux adaptés)
- Supposons qu'on a les deux parents suivants :
  - P1 : 1-2-3-4-5-6-7-1
  - P2 : 1-2-4-6-5-7-3-1
- Afin de générer un enfant, on choisira pour chaque sommet un lien choisi au hasard parmi les liens des deux parents



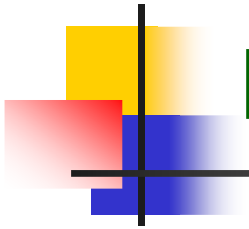
Exemple

- P1 : 1-2-3-4-5-6-7-1 et P2 : 1-2-4-6-5-7-3-1
- Pour le sommet 1, on peut choisir 1-2 ou 1-7 (P1) ou 1-2 ou 1-3 : P(1-2) = ½, P(1-7) = ¼, P(1-3) = ¼
- Supposons qu'on choisit 1-2
- Pour le sommet 2, on peut choisir 2-3 ou 2-4
- Supposons qu'on choisit 2-4 : 1-2-4
- Pour le sommet 4, on peut choisir 4-3 ou 4-5 ou 4-6
- Supposons qu'on choisit 4-3 : 1-2-4-3
- Pour le sommet 3, on peut choisir 3-7, mais rien d'autre!...



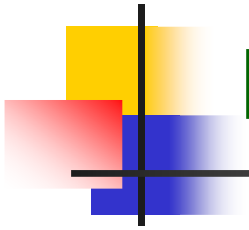
Exemple (suite)

- Pour offrir plus de choix, on va considérer que la tournée construite jusqu'à maintenant (1-2-4-3) est une inversion de celle de P1 : 1-2-3-4-5-6-7-1
- Pour compléter l'inversion de 3-4, on ajouterait le lien 3-5, qu'on considère alors comme un choix possible pour l'enfant
- Supposons qu'on fasse ce choix : 1-2-4-3-5
- Les choix sont alors : 5-6 (P1) ou 5-6 ou 5-7 (P2)
- Choisissons 5-6 : 1-2-4-3-5-6
- On doit alors compléter le tour : 1-2-4-3-5-6-7-1



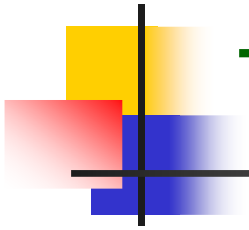
Exemple (suite)

- Cette procédure de génération d'un enfant inclut également une mutation (avec probabilité < 0,1), qui consiste à rejeter le lien choisi provenant d'un parent et à choisir un lien au hasard parmi tous ceux possibles
- Il est possible que les choix successifs nous amènent dans un cul-de-sac (aucun tour complet possible à partir du tour partiel) : on reprend alors la procédure depuis le début
- Voir cet algorithme génétique dans IOR Tutorial



Programmation par contraintes (PPC)

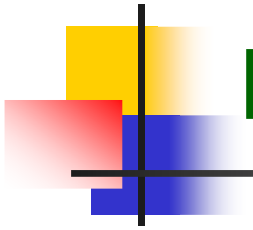
- Domaine de l'*informatique* (souvent associé à l'intelligence artificielle) s'intéressant à la résolution de problèmes de nature combinatoire impliquant:
  - Des variables à domaine fini
  - Des contraintes sur les valeurs de ces variables
- Développement de *langages informatiques* permettant de représenter et résoudre de tels problèmes
- **Contrainte**
  - Expression d'une relation entre des variables
  - Algorithme de réduction de domaine



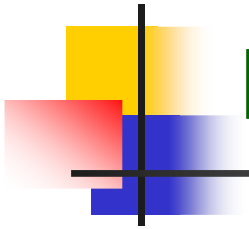
Types de contraintes

- Mathématiques: $x + y < z$
- Logiques: si $x = 5$ et $y \geq 3$, alors $6 \leq z \leq 8$
- Relationnelles: x, y et z doivent avoir des valeurs différentes
- Explicites: (x,y,z) peut prendre les valeurs (1,1,1), (2,4,5) ou (3,4,6)
- Un langage (au sens informatique) de programmation par contraintes doit permettre l'expression de ces différents types de contraintes

Réduction de domaine

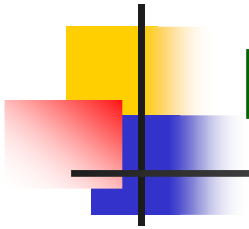


- Algorithme associé à chaque contrainte permettant d'éliminer des valeurs des domaines des variables impliquées dans l'expression de cette contrainte
- Exemple: la contrainte *alldiff*
- *alldiff*(x_1, x_2, …, x_n) veut dire que les *n* variables doivent toutes prendre des valeurs différentes
- Si $x_1 \in \{1,2\}$, $x_2 \in \{1,2\}$, $x_3 \in \{1,2,3\}$, $x_4 \in \{1,2,3,4,5\}$, alors l'algorithme de réduction de domaine pour *alldiff* déduit que :
  - x_3 ne peut prendre les valeurs 1 et 2
  - x_4 ne peut prendre les valeurs 1, 2 et 3
- Résultat: $x_1 \in \{1,2\}$, $x_2 \in \{1,2\}$, $x_3 \in \{3\}$, $x_4 \in \{4,5\}$



Propagation de contraintes

- Mécanisme par lequel les réductions de domaines se propagent d'une contrainte à l'autre
- Ajoutons la contrainte $x_1 + x_3 = 4$ dans notre exemple
- L'algorithme de réduction de domaine pour cette contrainte déduit que x_1 ne peut prendre la valeur 2, d'où $x_1 \varepsilon \{1\}$
- Si on applique à nouveau l'algorithme de réduction de domaine pour *alldiff*, on déduit que $x_2 \varepsilon \{2\}$: il y a eu *propagation de contraintes*



Recherche de solutions

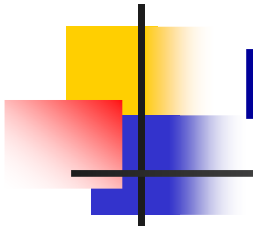
- Lorsque le domaine de chaque variable est réduit à une seule valeur, on obtient une solution (ou instanciation)
- La combinaison réduction de domaine/propagation de contraintes ne fournit pas nécessairement des solutions
- Il faut alors recourir à un algorithme de recherche de solutions: le plus courant procède à la *construction de l'arbre des solutions* comme dans l'algorithme de branch-and-bound en programmation en nombres entiers (PNE)
- Dans notre exemple, on choisirait la variable non instanciée x_4 et on générerait les deux alternatives (sommets) $x_4 = 4$ et $x_4 = 5$

Optimisation combinatoire et PPC

- Soit un problème d'optimisation combinatoire

$$\max_{x \in X} f(x)$$

- Si on a une représentation (un **modèle**) de l'ensemble X à l'aide d'un langage de PPC, on peut alors résoudre un tel problème:
  - En appliquant un algorithme de construction de l'arbre des solutions dans lequel on applique à *chaque sommet* la combinaison réduction de domaine/propagation de contraintes
  - On ajoute une variable Z représentant la valeur de l'objectif et une contrainte $Z \geq L$: la borne inférieure peut être améliorée chaque fois qu'une nouvelle solution est identifiée
  - Idéalement, on aimerait aussi ajouter une contrainte $Z \leq U$



Problème d'affectation

- *n* tâches à affecter à *n* machines de telle sorte que chaque machine soit affectée à une seule tâche et chaque tâche soit affectée à une seule machine
- Si on affecte la tâche *j* à la machine *i*, il en coûte c_{ij}
- L'objectif est de minimiser la somme des coûts
- Il s'agit d'un problème d'optimisation combinatoire: *X* est l'ensemble de toutes les affectations possibles des *n* tâches aux *n* machines (il y en a *n*!)

Modèle de PNE

- Variables:

$$x_{ij} = \begin{cases} 1 & \text{si la machine } i \text{ est affectée à la tâche } j \\ 0 & \text{sinon} \end{cases}$$

- Objectif: $\min Z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$

- Contraintes:

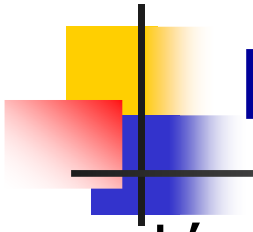
$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1,2,\ldots,n$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1,2,\ldots,n$$

- n^2 variables et $2n$ contraintes

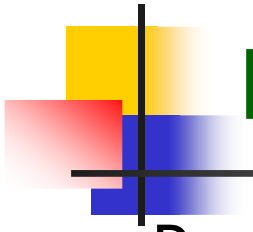
Modèle de PPC

- *element*(y, [c_1,c_2,…,c_n], z): contrainte qui affecte à z la $y^{\text{ème}}$ valeur du vecteur [c_1,c_2,…,c_n] ($z = c_y$)
- Variables:
  - y_i = tâche à laquelle est affectée la machine i : $y_i \in \{1,2,\ldots,n\}$
  - z_i = contribution à la valeur de l'objectif de l'affectation de la machine i : $z_i \in \{c_{i1}, c_{i2}, \ldots, c_{in}\}$
- Objectif: $$\min Z = \sum_{i=1}^{n} z_i$$
- Contraintes:
  - *element*(y_i, [c_{i1},c_{i2},…,c_{in}], z_i), $i = 1,2,\ldots,n$
  - *alldiff*(y_1, y_2,…,y_n)
- $2n$ variables et $n+1$ contraintes



Potentiel de la PPC

- L'exemple précédent montre une réduction importante du nombre de variables et de contraintes lorsqu'on utilise un modèle de PPC plutôt qu'un modèle de PNE
- Toutefois, le modèle de PNE pour le problème d'affectation peut être résolu par la programmation linéaire, car la relaxation PL possède une solution optimale entière!
- Mais si on ajoutait des contraintes supplémentaires, cette belle propriété ne tiendrait plus nécessairement: la PPC serait alors une alternative intéressante!
- En particulier, si ces contraintes supplémentaires peuvent difficilement s'exprimer sous forme de contraintes linéaires



PPC et méthodes de RO

- Pour résoudre un problème d'optimisation combinatoire, il peut être intéressant de combiner la PPC et la RO
- Par exemple, un problème d'affectation avec contraintes supplémentaires peut être résolu avec un algorithme de construction de l'arbre des solutions dans lequel :
  - On génère des solutions réalisables par la PPC
  - On calcule des bornes inférieures sur la valeur de l'objectif par la relaxation PL
- Une autre possibilité est de combiner la PPC avec une approche métaheuristique pour explorer des voisinages
- Mentionnons aussi que plusieurs algorithmes de réduction de domaine sont basées sur des méthodes de RO