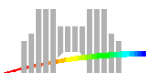
# Recherche Opérationnelle

## 4ème partie

Paul Feautrier

Ecole Normale Supérieure de Lyon
Paul.Feautrier@ens-lyon.fr

9 mai 2005

# Première partie I

# Algorithmes Génétiques

# Recuit Simulé / Algorithme Génétique

- On fait évoluer un "individu" avec intervention du hasard.
- Mécanisme d'évolution : exploration d'un voisinage.
- Pour sortir d'un piège : redémarrage aléatoire.

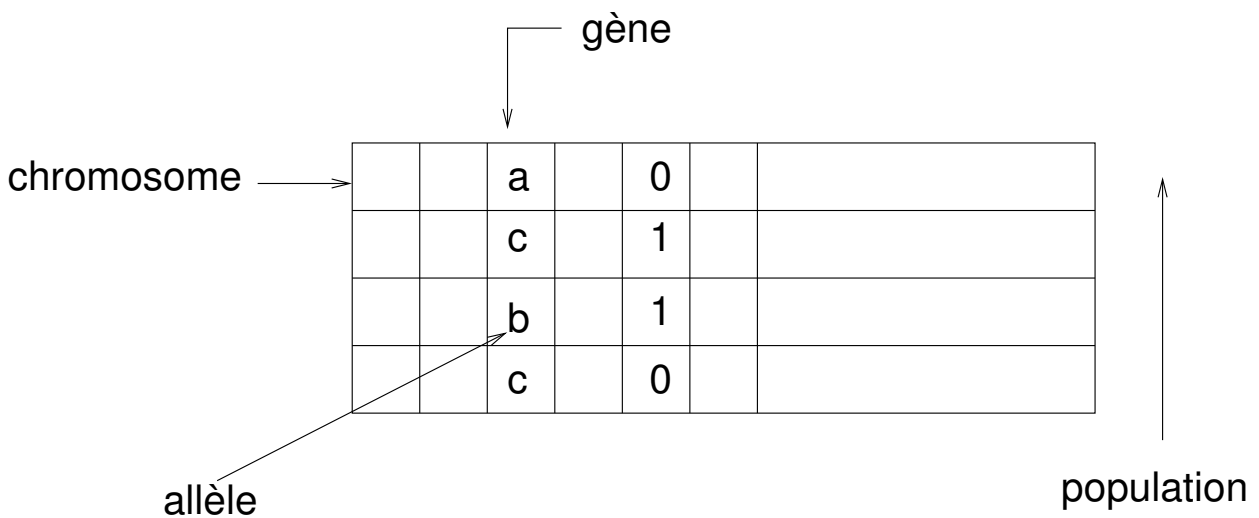
- On fait évoluer une "population" en paralléle.
- Mécanismes d'évolution plus puissants : exploration de voisinage (mutation) mais aussi combinaison entre individus (croisement).
- Une population de grande taille évite les pièges, mais on peut aussi redémarrer aléatoirement.

# Esquisse de l'Algorithme

- Une solution possible du problème d'optimisation est représentée par un *chromosome* = liste des valeurs des variables.
- On constitue une population de départ par tirage aléatoire. Taille de la population : $N \approx 200, 300$, à augmenter ou diminuer en fonction de la taille de l'espace de recherche.
- Pour chaque individu, on calcule la fonction objectif. Pénalités pour les individus "hors contraintes".
- On sélectione $n$ individus ayant une valeur de la fonction objectif élevée.
- On complète la population par croisement.
- On applique une mutation sur une petite fraction de cette population.
- On itére jusqu'à convergence.

# Chromosomes

- Traditionellement, on représente un individu par une chaîne de cararctères : chromosome.
- Chaque position dans la chaîne (*gène*) contrôle un aspect de l'individu (*allèle*).
- Les valeurs possibles de chaque gène forment un alphabet en général fini (les allèles).
- Tous les gènes peuvent avoir ou non le même alphabet.



# Sélection

- On calcule pour chaque chromosome $c$ la valeur de la fonction objectif $\phi(c) \geq 0$ (*fitness*).
- On peut trier les chromosomes par valeur décroissantes de $\phi(c)$ et garder les $n$ premiers.
- On peut également transformer la *fitness* en probabilité par la règle:

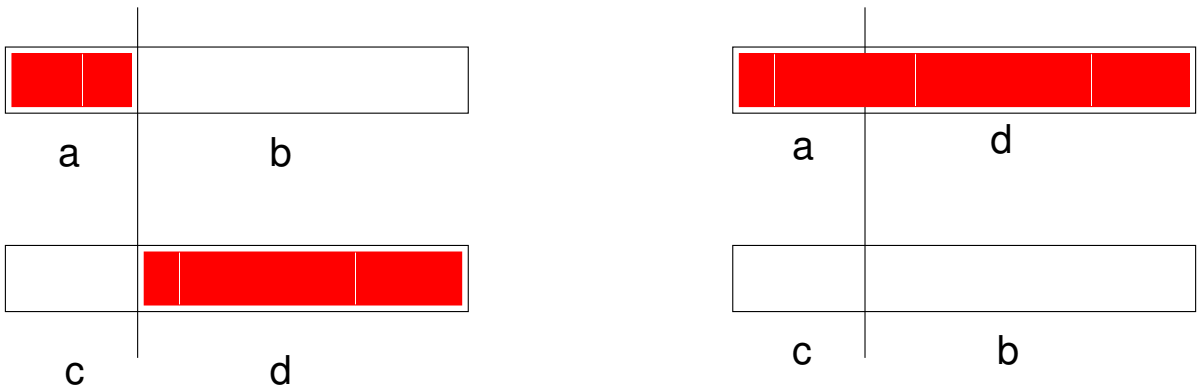
$$p(c) = \frac{\phi(c)}{\sum_c \phi(c)}.$$

  - On tire un nombre aléatoire entre 0 et 1, $X$, et on garde $c$ ssi $X \leq \alpha.p(c)$.
  - l'espérance mathématique du nombre d'individus gardés est $N\alpha = n$, d'où $\alpha = n/N$ (par exemple, 20%).
- Il faut ensuite ramener la population à $N$ individus.

# Croisement

**Intuition** : si un chromosome est bon "au début" et l'autre "à la fin", si on coupe et que l'on réarrange les fragments, on obtient un individu très bon et un autre très mauvais, que la sélection éliminera très vite.

- Sélectionner deux chromosomes au hasard, $u$ et $v$.
- Choisir un point de coupure (la longueur d'un préfixe) au hasard. $u = a.b, v = c.d$, $a$ et $c$ ont la même longueur.
- Former les chromosomes $a.d$ et $c.b$.



# Mutations

- Sélectionner au hasard une très faible proportion d'individus ($< 1\%$ typique).
- Modifier au hasard l'un de leurs gènes.
- La phase de sélection suivante élimine les "mauvaises" mutations.
- Analogue à une exploration de voisinage type *hill climbing* ou recuit simulé.
- Peu important en général, le croisement est bien plus efficace.
- Cependant, si un gène est absent de la population initiale, il ne peut être introduit ni par sélection, ni par croisement.
- Les mutations garantissent la "biodiversité" en cas de population initiale trop faible.

# Le sac-à-dos

- Le codage est très facile : le chromosome est formé de la suite des valeurs des inconnues ; chaque gène est zéro ou un.
- En calculant la valeur de chaque chromosome, on doit vérifier qu'il est faisable, et l'éliminer sinon.
- L'opération de croisement est évidente. Noter cependant que les gènes ne sont pas indépendants : le croisement de deux bons chromosomes peut donner un chromosome infaisable.
- L'opération de mutation revient à basculer la valeur de l'un des gènes choisi au hasard.
- Les résultats sont excellents, mais il faut de très gros problèmes pour surpasser les méthodes déterministes.

# Algorithmes

- Cliquer ici pour le code du méta-algorithme.
- Cliquer ici pour le code dépendant de l'application.
- Exécution.

# Expérimentation

- On notera que les gènes ne sont pas indépendants les uns des autres. Le croisement de deux bonnes solutions peut donner un individu infaisable.
- Le problème du sac-à-dos à 4 variables est trop petit. Il n'y a que $2^4 = 16$ individus possibles.
- Avec une population de 10 individus, on a plus d'une chance sur deux d'avoir l'individu optimal.
- Avec une petite population (5 ou 6 individus) l'algorithme se coince sur des solutions sous-optimales.
- L'algorithme n'est intéressant que pour une population moyenne et un taux de mutation non négligeable.

# Le voyageur de Commerce, Codage, I

**Rappel du problème**

- On se done $m$ villes et leur matrice des distances : $d_{ij}$ est la distance de la ville $i$ à la ville $j$.
- Pour simplifier, on suppose la matrice $d$ symétrique, et qu'il est toujours possible de se rendre d'une ville à l'autre.
- Il s'agit de trouver une tournée (circuit Hamiltonien) qui passe une fois et une seule par chaque ville et dont la longueur est minimale.
- Comme on sait, le problème est très difficile.
- Comme on peut fixer arbitrairement la première ville, il y a $!(m-1)$ circuits possibles.

# Codage, II

## Première solution

- Un chromosome est la liste des villes traversées dans l'ordre.
- Un chromosome n'est valide que si aucune ville n'est visitée deux fois.
- Cette propriété n'a aucune chance de se conserver par croisement ou mutation :

$$\begin{array}{cc} AB.CDE & AB.BED \\ \Rightarrow & \\ AC.BED & AC.CED \end{array}$$

  Les deux descendants sont infaisable.
- Cette solution est infaisable.

# Codage, III

## Deuxième solution

- On se donne une fois pour toute la liste des villes : $ABCDE$.
- Un chromosome est une liste d'entiers. Chaque entier donne le rang de la ville à extraire de cette liste, *sans remplacement*.

$$\begin{aligned} 1.1.1.1.1 &\rightarrow A.B.C.D.E \\ 5.4.3.2.1 &\rightarrow E.D.C.B.A \\ 1.2.3.2.1 &\rightarrow A.C.E.D.B \end{aligned}$$

- On trouve toutes les permutations.
- La suite de nombres doit respecter la contrainte $x_i \leq m - i + 1$.
- Cette contrainte est respectée par croisement.
- Il faut l'imposer à l'algorithme de mutation.

# Algorithmes

- Cliquer ici pour le code dépendant de l'application.
- Exécution.

# Expérimentation

- Le fonctionnement est beaucoup plus satisfaisant, car les populations sont plus importantes ; on est en régime “grand nombres”.
- Mais le temps de calcul est plus important.
- L’astuce “conserver le meilleur” améliore bien les choses.
- Il faut bien ajuster la fonction de “fitness” pour avoir des différences importantes entre chromosomes.
- Noter la sensibilité à la taille de la populations et à la taille de la sélection.
- La meilleure solution tend à envahir la population : critère d’arrêt.

# Optimisation non contrainte

- Problème : maximiser la fonction $f(x) \geq 0$ dans l'intervalle $[0, 1]$.
- Un chromosome est une chaîne de bits représentant un nombre fractionnaire en base 2 (le nombre de bits est proportionnel à la précision cherchée).
- "fitness" la valeur de $f$ pour le nombre représenté par le chromosome.
- Mutation : changer un bit.
- Sélection et croisement, comme d'habitude.
- On peut également croiser en prenant le point milieu des deux chromosomes.
- Généralisation facile à plusieurs dimensions, avec contraintes.
- Essayez!

# Etude du croisement, I

- Soit $a.b$ et $c.d$ deux individus avant croisement.
- Supposons que l'on puisse écrire $f(a.b) = f_1(a) + f_2(b)$.
- Alors après croisement, il y a toujours un individu amélioré, sauf si $f_1(a) = f_1(c)$ et $f_2(b) = f_2(d)$. Preuve par examen de tous les cas possibles.
- Mais si le croisement se fait en un point où la propriété n'est pas assurée, on ne peut rien dire.

# Etude du croisement, II

- En général, les gènes ne sont pas indépendants et la fonction objectif n'est pas totalement séparable. Un cas fréquent:

$$f(x) = g(x_1, x_2) + \ldots + h(x_{m-1}, x_m).$$

- Il est facile de voir qu'on ne peut garantir une amélioration que dans un cas sur deux (coupure entre un gène pair et un gène impair).

# Etude du croisement, III

- Si maintenant:

$$f(x) = g(x_1, x_m) + h(x_2, x_{m-1}) + \ldots,$$

  presque tous les croisements vont "casser" le couplage entre deux gènes.
- Pourtant, il s'agit de deux problèmes de même nature.
- **Conclusion** L'ordre des gènes dans le chromosome n'est pas indifférent.
- Les algorithmes génétiques favorisent les petits groupes de gènes contigus.

# Améliorations, I

**Première suggestion** On réordonne les chromosomes (aléatoirement) de temps à autre.

- La fonction de "fitness" et la fonction de mutation doivent être ajustées en conséquence.
- Pour le sac-à-dos, cela revient à permuter les tableaux `weights` et `benef`.
- Pour le voyageur de commerce, celà revient à appliquer une permutation à la liste des villes.
- On peut faire *co-évoluer* plusieurs populations ayant des ordres différents et sélectionner la meilleure: il y a de multiples variantes ("hill-climbing" sur l'ordre des gènes).

# Amélioration, II

**"Randomiser" les croisements**

- A partir de deux chromosomes, en construire un troisième en prenant chaque gène, aléatoirement, dans l'un des parents ou dans l'autre.
- La probabilité que deux gènes soient séparés est indépendante de leur distance et égale à 1/2.
- Expérimentation pour le voyageur de commerce: amélioration, mais pas spectaculaire.

# Parallélisation

- Les algorithmes génétiques sont faciles à paralléliser.
- Le calcul de la fonction objectif est entièrement parallèle.
- Il est un peu plus délicat de paralléliser l'algorithme de sélection (accès en section critique à la liste des sélectionnés). Même difficulté pour le croisement. La mutation ne pose pas de problème.
- Il faut veiller à ce que les générateurs aléatoires de chaque processeur soient indépendants.

# Principes de Codage

**Principes généraux**

- Utiliser un système d'encodage naturel (ne pas chercher à toute force à coder à l'aide de chaînes de bits).
- Repenser les opérateurs de croisement et mutation en fonction de la représentation choisie.
- Si possible, s'assurer que tous les chromosomes sont faisables.
- Si possile, garantir que les opérations de mutation et de croisement conservent la faisabilité.
- Essayer d'obtenir l'indépendance des gènes.

# Codage par poids

- Il s'agit d'affecter à $m$ objets un poids $w_i$ tel que :

$$\sum_{i=1}^{m} w_i = 1.$$

- Exemple : la sélection d'un portefeuille d'actions.
- On code à l'aide d'une liste d'entiers ou de flottants $x_i, i = 1, \ldots, m$ ; les poids sont donnés par la formule:

$$w_i = \frac{x_i}{\sum_{i=1}^{m} x_i}.$$

- Tous les chromosomes sont faisables ; l'opérateur de croisement ne pose pas de problème.
- Pour l'opérateur de mutation, on peut être amené à borner les $x_i$ ou à utiliser des distributions non uniformes.

# Optimisation numérique sous contraintes

- Exemple: un problème de programmation linéaire.
- Codage par la liste des valeurs des inconnues.
- Problème I : borner les solutions, en général à partir de connaissances à priori sur le problème.
- Problème II : faisabilité.
  - Méthodes de pénalité : un chromosome infaisable n'a aucune chance de se reproduire.
  - Méthodes de réparation : on ajuste une variable artificielle pour rendre le chromosome faisable. L'objectif est de minimiser la variable artificielle.

# Problèmes de permutation

- On se donne $m$ objets et on doit trouver la meilleure permutation de ces objets.
- Exemple, le voyageur de commerce.
- Codage direct de la permutation.
- Codage par adjacence : pour chaque objet, on spécifie l'objet immédiatement à sa droite.
- Codage de l'ordre d'extraction.
- Codage par tri : on affecte une valeur à chaque objet. La permutation est obtenue par tri des objets selon leur valeur.
- On notera que les opérations génétiques conservent la faisabilité pour les deux dernières représentations, mais pas pour les deux premières.

# Méthodes Hybrides

- On introduit de l'optimisation déterministe à l'intérieur d'un algorithme génétique.
- Par exemple, au lieu de muter au hasard (ce qui échoue presque toujours, surtout à la fin de la recherche), on peut chercher la meilleure valeur possible du gène à modifier (recherche exhaustive ou optimisation unidimensionnelle).
- Pour un croisement, on peut rechercher la meilleure position.
- Dans la population initiale, on peut introduire de bons individus obtenus par une heuristique.
- Souvent très efficace.

# Autres Structures de Chromosomes

- Les chromosomes ne sont pas nécessairement de taille fixe. Exemple : optimiser les termes d'un développement en série.
- On doit introduire un nouvel opérateur, le rallongement.
- Les gènes ne sont pas nécessairement ordonnés. Ils peuvent par exemple former un ensemble.
- Pour le sac-à-dos, l'ordre des gènes (des objets) n'a pas de sens, et une représentation ensembliste est mieux adaptée.
- Les chromosomes peuvent avoir une structure, voit plus loin.

# Autres Opérateurs Génétiques

- Les opérateurs génétiques classiques peuvent ne pas convenir ou ne pas être satisfaisants suivant la représentation choisie.
- Par exemple, si les chromosomes sont des ensembles, le croisement classique n'a pas de sens. Il vaut mieux utiliser le croisement aléatoire.
- Si les gènes codent pour une position sur un axe, on peut envisager des opérateurs de croisement géométriques (par exemple, le choix du point milieu).
- Pour un problème de permutation, la mutation classique n'a pas grand sens. Par contre, l'interchange de deux gènes (l'interversion de deux villes dans une tournée) a du sens. On peut être amené à gérer une double représentation.

# Programmation Génétique

**Idée de base** Un chromosome n'est plus une liste de valeurs, mais un programme.

- Résoudre le problème posé, c'est exécuter le programme sur les données du problème.
- La qualité du programmme est donnée par le nombre d'instances correctement résolues.
- Dans certains cas, on peut énumérer toutes les instances (ex : l'inférence de fonctions logiques).
- Dans d'autre, on est obligé d'échantillonner les instances.

# Représentation du Programme

- En général, chaque chromosome est l'arbre de syntaxe abstraite du programme.
- On a donc intérêt à choisir un langage dont la syntaxe est la plus simple possible. Ce langage existe, c'est LISP.
- Les objets et les programmes de LISP sont des arbres écrits en notation préfixée et parenthésée.

```
NIL
(+ X (* Y Z))
(CONS X Y)
(CAR (1 2 3))
```

# Le vocabulaire de LISP

- Dans un programme, l'élément de tête de chaque liste est un opérateur.
- On distingue les opérateurs classiques (`+`, `*`, `CAR`, `CDR`, etc.)
- Les opérateurs de contrôle (`PROGN`, `IF`, `WHILE`),
- Les opérateurs spécifiques du problème traité.
- Par exemple, au jeu de tic-tac-toe (*nought and crosses*), on peut avoir un opérateur `CROSS5`, vrai ssi il y a une croix sur la case centrale de l'échiquier.
- On peut avoir également des constantes et des variables (opérateur d'affectation `SETQ`).

# Opérateurs Génétiques

- L'opérateur de croisement consiste à interchanger deux sous-arbres des deux programmes à combiner.
- En général, on peut s'arranger pour que le croisement de deux programmes corrects soit correct, mais il peut y avoir des exception.
- Par exemple, dans l'affectation `(SETQ X (1 * 3))` le sous-arbre `X` ne peut être croisé qu'avec une autre variable.
- On peut avoir également des contraintes de type.
- La mutation n'a pas beaucoup de sens, sauf pour inférer la valeur de constantes.

# Exemples, I

**Inférence de circuits logiques**

- Il s'agit de construire un circuit logique qui implémente une fonction logique donnée par sa table de vérité.
- Le vocabulaire se réduit aux noms des signaux d'entrée et a une sélection d'opérateurs primitifs (`AND`, `OR`, `NOT` ou `NOR`).
- On peut évaluer une implémentation suivant deux critères, le nombre de cas correctement traités, et la simplicité.
- Il est très facile d'introduire des cas *dont'care*.

# Exemples, II

**Interpolation symbolique**

- On se donne une table de valeur d'une fonction (par ex., des résultats de mesures), et une liste de fonctions élémentaires.
- Un individu est un programme utilisant les fonctions élémentaires, des constantes et les variables libres.
- La qualité est une norme mesurant la distance entre les valeurs de la table.
- Les algorithmes de croisement et mutation ne posent pas de problème.
- L'inférence des constantes est plus difficile :
  - Si on les impose ou si on les tire au hasard, le système a tendance à les ajuster en appliquant des fonctions élémentaires sans signification.
  - C'est sans doute ici le cas d'utiliser un algorithme hybride (ajustement de la constante par optimisation unidimensionnelle).

# Programmes de Jeux

- Le problème est de trouver une stratégie, c'est-à-dire un programme qui donne le prochain coup à jouer en fonction de la situation actuelle. La qualité dépend du nombre de parties gagnées.
- Il n'y a pas de problème pour un jeu de solitaire, sauf de bien choisir les fonctions élémentaires.
- Dans un jeu de compétition, comment représenter l'adversaire ?
- Si on connaît la stratégie optimale de l'adversaire, on peut faire jouer le programme génétique contre elle mais c'est artificiel.

# Co-évolution

- On cherche deux programmes, un pour chaque joueur (ou même un programme pour chaque coup de chaque joueur).
- Pour chaque programme on fait évoluer une population selon un algorithme génétique.
- On fait jouer chaque individu contre tous les autres pour évaluer sa qualité.
- On espère que chaque population va évoluer vers sa stratégie optimale.

# Algorithmes Génétiques, Evaluation

- Méthodes en théorie très puissantes, mais difficiles à administrer.
- L'efficacité de l'algorithme dépend énormément de la qualité du codage (faisabilité des chromosomes).
- En particulier, les critères d'arrêts sont flous. On n'est jamais sûr d'avoir atteint l'optimum global.
- A n'utiliser que pour des problèmes ayant un grand nombre de variables et un grand espace de solution.

# Deuxième partie II

# Applications en Informatique

# Applications en Informatique, Généralités

- En Informatique, de très nombreux algorithmes peuvent être vus comme des algorithmes d'optimisation.
  - Algorithmes de plus court et de plus long chemin ;
  - Algorithmes de reconnaissance de formes, et la plupart des algorithmes de l'intelligence artificielle.
  - Algorithmes de simplification algébrique ou booléenne.
- Il peut être intéressant de reformuler un algorithme comme une optimisation (méthodes variationnelles).
  - Par exemple, trier, c'est trouver la permutation ayant le moins d'inversions.
- On essaie de se limiter à des algorithmes polynomiaux.
- Un aspect nouveau : l'optimisation *en ligne*.

# Application aux réseaux

**Routage** Un réseau informatique est un graphe dont les sommets sont les stations et les arcs sont les liaisons.

- Chaque arc a un débit (ou une latence) qui représente le temps nécessaire à la transmission d'un message.
- Le problème du routage est le problème du plus court chemin entre deux stations dans ce graphe.
- C'est un problème de programmation en nombres entiers qui peut être résolu en temps polynomial.
- Le problème devient non linéaire si on note que le passage d'autres messages réduit le débit disponible.

# Dimensionnement d'un réseau

- Le problème est de déterminer le débit de chaque lien pour écouler au moindre prix le trafic prévu.
- On se donne la matrice de trafic, $t_{ij}$ : le nombre de messages (ou le nombre d'octets) émis par seconde de la station $i$ vers la station $j$.
- Chaque lien est capable de transmettre $d_{ij}$ messages de la station $i$ à la station $j$.
- Le prix du lien $ij$ est $u_{ij} \times d_{ij}$ où $u_{ij}$ est un facteur "géographique".

# Dimensionnement, II

- On traite le flot des messages comme un fluide : soit $x_{ij} \geq 0$ le flot qui traverse le lien $ij$.
- On écrit la loi de conservation des messages ou loi de Kirchof : ce qui entre dans la station $i$ plus les messages envoyés par cette station est égal à ce qui en sort plus ce que cette station reçoit.
- Messages reçus par $i$ : $r_i = \sum_k t_{ki}$. Formule analogue pour ce qui est émis par $i$, $e_i$.

$$\sum_k x_{ki} + e_i = \sum_j x_{ij} + r_i.$$

- On ne doit pas excéder la capacité du lien : $x_{ij} \leq d_{ij}$.
- On doit minimiser $\sum_{ij} u_{ij} d_{ij}$.

# Dimensionnement, III

- C'est un problème de programmation linéaire continu (parce que le nombre de messages est très grand).
- On a supposé que l'on pouvait ajuster le débit de façon continue : ce n'est pas le cas en général. Il existe une gamme de prix assortie à une gamme de débits. Le problème devient combinatoire.

# Algorithmes approchés

- Dans les problèmes de ce type, les données ne sont pas connues avec une grande précision.
- On peut donc se contenter de solutions approximatives, obtenues par exemple en arrondissant les $t_{ij}$.
- Mais il est difficile de chiffrer l'erreur.

# Compilation, Généralités

- Dans l'abstrait, la compilation est un problème d'optimisation : *Etant donné un programme source, trouver le programme objet équivalent qui optimise une certaine mesure : performance, taille du programme, taille des données, consommation électrique, etc.*
- L'équivalence des programmes est la contrainte du problème. Elle est difficle à formaliser : l'équivalence de deux programmes est indécidable.
- La fonction à optimiser peut se mesurer ou s'estimer ou être définie intuitivement. On suppose par exemple qu'en supprimant des calculs redondants on améliore les performances, ce qui n'est pas évident.

# Ordonnancement et Parallèlisme

- En compilation parallèle, on peut simplifier le problème en introduisant la notion d'ordonnancement.
- Ordonnancer un calcul, c'est fixer la date d'exécution de chaque opération.
- On peut trouver un critère simple d'équivalence car on ne change pas la liste des opérations. Il suffit de repérer les opérations qui *commutent*.
- Qu'est-ce qu'une opération ? On peut faire varier la *granularité* depuis l'instruction ou fragment d'instruction jusqu'à une application complète (exemple de `pmake`).

# Contraintes de Précédence

- On considère un programme linéaire ou *bloc de base*.
- Soit $\theta(u)$ la date d'exécution de l'opération $u$.
- Si $u$ et $v$ ne commutent pas et si $u$ est exécutée avant $v$ dans le programme original, on a la contrainte de précédence :

$$\theta(u) + 1 \leq \theta(v).$$

- La durée du programme est définie par les contraintes :

$$0 < \theta(u) + 1 \leq L.$$

- Minimiser $L$ sous les contraintes de précédence est un programme linéaire.

# Contraintes de Précédence, II

- Deux opération telles que $\theta(u) = \theta(v)$ s'exécutent en parallèle. Il existe des méthodes pour reconstituer le programme parallèle connaissant $\theta$.
- On peut traiter des programmes plus complexes (boucles et tests) moyennant des hypothèses de régularité.
- Les choses se gâtent pour les programmes dont le contrôle est dépendant des données.

# Contraintes de Ressources

- Pour exécuter $n$ opérations en parallèle, il faut $n$ processeurs.
- Le problème est contraint par les ressources si l'on ne dispose que de $P \ll n$ processeurs.
- A chaque instant, le nombre d'opérations en cours d'exécution ne doit pas excéder $P$. Codage en $0/1$. $X_{ut} = 1$ si $u$ est exécutée au temps $t$.

$$\sum_t X_{ut} = 1 \quad , \quad \sum_u X_{ut} \leq P,$$

$$\theta(u) \quad = \quad \sum_t t X_{ut}.$$

- Résolution par la PLNE, mais :
  - Le nombre d'inconnues est de l'ordre du carré du nombre d'opérations.
  - Il apparaît des coefficients de l'ordre du nombre d'opérations.

# Heuristiques et Garanties

- Pour éviter ces inconvénients, on a inventé des heuristiques.
- Ordonnancement par liste : dès qu'un processeur se libère, on exécute une opération disponible s'il y en a.
- On démontre que la durée ne dépasse pas deux fois la durée optimale.
- L'algorithme peut être exécuté *en ligne*.