Définition 6.1 - *algorithme probabiliste*

Un algorithme est dit *probabiliste* s'il effectue au moins un *choix aléatoire entraînant une variation comportementale*.

Définition 6.2 - *algorithme déterministe*

Un algorithme est dit *déterministe* s'il est non probabiliste : son comportement est toujours le même pour une même entrée.

Définition 6.3 - *algorithme de Las Vegas*

Un algorithme probabiliste est dit *de Las Vegas* si :

- il renvoie toujours une solution correcte ;
- son temps d'exécution est régi par une variable aléatoire.

Définition 6.4 - *loi géométrique*

Une *loi géométrique de paramètre p* est une loi modélisant le *nombre d'essais nécessaires jusqu'au premier succès* dans une suite d'expériences indépendantes identiques. Si X suit une loi géométrique, alors :

$$\forall k \in \mathbb{N}^*, \mathbb{P}(X = k) = (1-p)^{k-1}p$$

On a alors $\mathbb{E}(X) = \dfrac{1}{p}$ et $\mathbb{V}(X) = \dfrac{1-p}{p^2}$.

Implémentation - *exemple d'algorithme de Las Vegas*

```
1 int las_vegas(int* t, int taille){
2     while(true) {
3         int k = rand() % taille;
4         if (t[k] == 1){
5             return k;
6         }
7     }
8 }
```

Définition 6.5 - *algorithme de Monte Carlo*

Un algorithme probabiliste est dit *de Monte Carlo* si :

- il renvoie sous une certaine probabilité une solution correcte ;
- son temps d'exécution est constant, indépendant des choix aléatoires.

Implémentation - *exemple d'algorithme de Monte Carlo*

```
1 int monte_carlo(int* t, int taille) {
2     for (int i = 0; i < 300; i++){
3         int k = rand() % taille;
4         if (t[k] == 1){
5             return k;
6         }
7     }
8     return -1;
9 }
```

Définition 6.6 - *deux types d'algorithmes Monte Carlo pour un problème de décision*

Pour un problème de décision (dont la réponse est Vrai ou Faux), on distingue deux catégories d'algorithmes Monte Carlo :

1. *À erreur unilatérale* : pour toute entrée de sortie attendue Vrai, l'algorithme renvoie Vrai presque sûrement (avec une probabilité de 1) et pour une entrée de sortie attendue Faux, l'algorithme renvoie Vrai avec une probabilité non nulle.
2. *À erreur bilatérale* : il existe une entrée de sortie attendue Vrai, pour laquelle l'algorithme renvoie Faux avec une probabilité non nulle.

Définition 6.7 - *problème d'optimisation*

Un problème d'optimisation est défini pour un *ensemble I d'instances* : les entrées possibles.
À toute instance $i \in I$ on associe un *ensemble S_i de solutions possibles*.
À un problème d'optimisation on peut définir une *fonction $f : \bigcup_{i \in I} \to \mathbb{R}_+$ de coût* que l'on cherche soit à minimiser soit à maximiser.

Pour une instance $i \in I$, on cherche $s_{\text{opt}} \in S_i$ une solution au coût optimal :

$$\begin{cases} \forall s \in S_i,\ f(s) \leq f(s_{\text{opt}}) & \text{pour de la maximisation} \\ \forall s \in S_i,\ f(s) \geq f(s_{\text{opt}}) & \text{pour de la minimisation} \end{cases}$$

Définition 6.8 - *algorithmes de résolution exacte, d'approximation*

Pour un problème d'optimisation, pour une instance $i \in I$,

1. un algorithme de résolution exacte est un algorithme qui renvoie $s_{\text{opt}} \in S_i$
2. un algorithme d'approximation est un algorithme qui renvoie $s_{\text{approx}} \in S_i$ *une solution approximativement optimale*. On définit alors $\rho_i(s_{\text{approx}})$ *le rapport de performance pour l'instance i de* s_{approx} comme valant :

$$\rho_i(s_{\text{approx}}) = \max\left(\frac{f(s_{\text{approx}})}{f(s_{\text{opt}})}, \frac{f(s_{\text{opt}})}{f(s_{\text{approx}})}\right) \quad (\geq 1)$$

Définition 6.9 - *rapport de performance pour un algorithme*

Pour un problème d'optimisation impliquant un algorithme $\mathcal{A}$ d'approximation, on dit que ρ_n *est un rapport de performance pour une instance de taille n* si :

$$\rho_n \geq \sup_{\substack{i \in I \\ |i|=n}} \rho_i\Big(\mathcal{A}(i)\Big)$$

On dit alors que $\mathcal{A}$ *est une ρ_n-approximation*. Généralement, on cherche un majorant de cette borne supérieure.

Définition 6.10 - *arbre de recherche de solutions*

Dans le cadre d'un problème d'optimisation combinatoire, étant donnée $i \in I$ un instance, on appelle *arbre de recherche de solutions de i* un arbre étiqueté par des parties de S_i :

- la racine est étiquetée par S_i tout entier
- chaque noeud étiqueté par $S \subset S_i$ a pour fils des noeuds d'étiquette $S_{\mathrm{f}_1}, \ldots, S_{\mathrm{f}_k}$ tels que $S = \bigsqcup_{i=1}^{k} S_{\mathrm{f}_i}$
- les feuilles sont étiquetées par des singletons, on les associe en fait à une solution chacune.

On définit alors une *fonction φ d'évaluation* qui à un noeud n d'étiquette S associe ce qu'on peut espérer au mieux dans l'exploration du sous-arbre :

$$\begin{cases} \varphi(n) \geq \max\limits_{s \in S} f(s) & \text{pour de la maximisation} \\ \varphi(n) \leq \min\limits_{s \in S} f(s) & \text{pour de la minimisation} \end{cases}$$

Définition 6.11 - *algorithme par séparation-évaluation*

Un *algorithme par séparation évaluation (Branch & Bound)* repose *une fonction d'évaluation φ* et sur trois règles :

1. *règle de sélection* : impose un ordre d'exploration des fils de chaque sous-arbre de recherche de solutions
2. *règle de séparation* : impose la méthode de partitionnement de chaque étiquette des noeuds de l'arbre de recherche de solutions
3. *règle d'évaluation* : impose de ne pas explorer certains fils.

Implémentation - *algorithme générique par séparation-évaluation (cadre de la maximisation)*

Pour tout arbre $\mathcal{A}$ de recherche de solution, on note $S(\mathcal{A})$ son étiquette.

- **Entrée** :
  - S un ensemble de solutions possibles
  - $\varphi : \mathcal{P}(S) \to \mathbb{R}_+$ une fonction d'évaluation
  - f la fonction de coût du problème, ici supposée à maximiser
- **Sortie** : $s_{\text{opt}} \in S$ telle que $\forall s \in S$, $f(s_{\text{opt}}) \geq f(s)$

```
1  separation_evaluation(S, φ, f):
2      A = arbre feuille de racine d'étiquette S
3      initialiser un sac contenant A
4      s_opt = None // noeud courant
5      max = −∞ // coût courant
6      tant que le sac n'est pas vide:
7          n = pop le premier élément du sac // règle de sélection
8          si |S(n)| == 1:
9              identifier {s} = S(n)
10             si f(s) > max:
11                 s_opt = s
12                 max = f(s)
13         sinon si φ(S(n)) > max: // règle d'évaluation
14             partitionner S(n) = ⊔_{i=1}^{k} S_i // règle de séparation
15             pour tout i ∈ ⟦1, k⟧:
16                 ajouter à n le fils f_i d'étiquette S_i
17                 ajouter f_i au sac
18     renvoyer s_opt
```

Définition 6.12 - *relaxation d'un problème d'optimisation*

On appelle *relaxation d'un problème d'optimisation* $\mathcal{P}$ un problème $\mathcal{P}'$ construit à partir de $\mathcal{P}$ en omettant une ou plusieurs contraintes. $\mathcal{P}'$ comporte alors de la même fonction de coût, les mêmes instances, mais l'ensemble des solutions possibles est plus grand pour l'inclusion.

Définition 6.13 - *généralisation d'un problème d'optimisation*

On appelle *généralisation d'un problème d'optimisation* $\mathcal{P}$ un problème $\mathcal{P}'$ tel que :

- les deux ont la même fonction de coût, avec le même objectif (maximisation ou minimisation)
- $I_{\mathcal{P}} \subset I_{\mathcal{P}'}$
- pour toute instance de $I_{\mathcal{P}}$, une solution optimale pour $\mathcal{P}'$ est une solution optimale pour $\mathcal{P}$.