Chapitre 5: Méta-heuristiques

INF889B — Algorithmes d'optimisation combinatoire

Alexandre Blondin Massé

Université du Québec à Montréal

Hiver 2020

Plan

Stratégie gloutonne

Idée générale

- Soit $\mathcal{S}$ l'ensemble des **instances** possibles pour un problème
- Soit f la fonction objectif à **minimiser**
- On souhaite trouver un **ensemble** $S \in \mathcal{S}$ qui minimise f
- On **démarre** avec $S \leftarrow \emptyset$
- On choisit s_1 qui semble le plus **intéressant**
- On **ajoute** s_1 à S
- Puis on **répète**, jusqu'à ce qu'on ne puisse **plus rien** ajouter

Remarque

- Similaire à la **recherche locale**
- Mais on **construit** une solution
- Plutôt que de l'**améliorer**

Pseudocode

1: **fonction** ConstructionGloutonne(E : ensemble)
2: $S \leftarrow \emptyset$
3: **tant que** il existe $e \in E$ tel que $S \cup \{e\}$ est admissible **faire**
4: Soit e le candidat le meilleur
5: $S \leftarrow S \cup \{e\}$
6: **retourner** S

À détailler

- Quels sont les **candidats**?
- Qu'est-ce qu'une solution **admissible**?
- Comment choisir le **meilleur** candidat?

Ratio d'approximation

- Soit P un **problème** d'optimisation
- Soit f la fonction à **minimiser**
- Soit S^* une solution de P telle que $f(S^*)$ est **minimale**
- Soit α un **nombre** qui dépend de la taille de P
- Et soit A un **algorithme** qui résoud le problème P
- On dit que A a un **ratio d'approximation** α si pour toute instance, la solution S retournée par A vérifie

$$f(S) \leq \alpha f(S^*)$$

Remarque

- Plusieurs problèmes **NP-difficiles** peuvent être **approximés** à l'aide d'algorithmes gloutons
- Souvent, les algorithmes sont **faciles** à implémenter

Recouvrement d'un ensemble

- En anglais, appelé *set cover*
- Soient n et m deux **entiers positifs**
- On prend un ensemble $X = \{x_1, x_2, \dots, x_n\}$ d'**éléments**
- Et une **collection** d'ensembles $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ telle que

$$\bigcup_{i=1}^{m} S_i = X$$

- Finalement, soit $w : \mathcal{S} \to \mathbb{R}_+$ une fonction qui associe à $S \in \mathcal{S}$ un **poids** $w(S)$.
- Un **recouvrement** de X est un sous-ensemble $\mathcal{S}' \subseteq \mathcal{S}$ tel que

$$\bigcup_{S \in \mathcal{S}'} S = X$$

- On cherche un recouvrement $\mathcal{S}'$ qui **minimise** $w(\mathcal{S}')$

Exemple (1/2)

- Soient $n = 10$ et $m = 8$
- Prenons

$$X = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- Et $\mathcal{S} = \{S_0, S_1, \ldots, S_7\}$ **défini** par

$$\begin{array}{lcll} S_0 & = & \{0, 1, 3, 5\} & w(S_0) = 2.0 \\ S_1 & = & \{0, 2, 4, 8, 9\} & w(S_1) = 1.0 \\ S_2 & = & \{1, 3, 5, 7, 9\} & w(S_2) = 3.0 \\ S_3 & = & \{6, 7, 8, 9\} & w(S_3) = 2.0 \\ S_4 & = & \{1, 4\} & w(S_4) = 1.0 \\ S_5 & = & \{2, 4, 6\} & w(S_5) = 3.0 \\ S_6 & = & \{0, 1, 2, 3, 5, 8, 9\} & w(S_6) = 4.0 \\ S_7 & = & \{1, 2, 4, 8\} & w(S_7) = 5.0 \end{array}$$

- Comment peut-on construire une solution **optimale**?

Exemple (2/2)

- Meilleur **rendement**: S_1 (5 éléments pour 1.0)
- **Prochain**: S_0 (3 nouveaux éléments pour 2.0)
- **Puis**: S_3 (2 nouveaux éléments pour 2.0)
- Les 10 éléments sont couverts pour un **poids total** de 5.0.
- Difficile de vérifier si la solution est **optimale**

Exemple (2/2)

- Meilleur **rendement**: S_1 (5 éléments pour 1.0)
- **Prochain**: S_0 (3 nouveaux éléments pour 2.0)
- **Puis**: S_3 (2 nouveaux éléments pour 2.0)
- Les 10 éléments sont couverts pour un **poids total** de 5.0.
- Difficile de vérifier si la solution est **optimale**
- Par contre, on peut montrer que le **ratio d'approximation** dans l'exemple est $H(7) \approx 2.593$, où

$$H(d) = \sum_{i=1}^{d} \frac{1}{i} \approx \ln d$$

est le d-ième nombre **harmonique**

Pseudocode

1: **fonction** SetCover(X : ensemble, $\mathcal{S}$: collection)
2: $Y \leftarrow \emptyset$
3: $S \leftarrow \emptyset$
4: **tant que** $Y \neq X$ **faire**
5: Soit $T \in \mathcal{S}$ qui maximise #nouveaux éléments/coût
6: $S \leftarrow S \cup \{T\}$
7: $Y \leftarrow Y \cup T$
8: **retourner** S

Théorème

Chvátal, 1979

- Soit X un ensemble de n **éléments**
- Soit $\mathcal{S}$ une collection de **sous-ensembles** de X
- Soit $w : \mathcal{S} \rightarrow \mathbb{R}_+$ une fonction de **poids**
- Alors l'algorithme **glouton** décrit précédemment a un **ratio d'approximation** de $H(s_{\max})$, où

$$s_{\max} = \max\{|S| \ : \ S \in \mathcal{S}\}$$

Raz et Safra, 1997

Il n'existe pas d'algorithme **polynomial** pour ce problème ayant un ratio d'approximation **significativement meilleur**

Non optimalité

- En général, l'algorithme glouton ne trouve pas une **solution optimale**
- **Exemple**: $n = 12$, $m = 3$,

$$X = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

et $\mathcal{S} = \{S_0, S_1, S_2\}$, où

$$\begin{array}{lcll} S_0 & = & \{0, 1, 2, 3, 4\} & w(S_0) = 6.0 \\ S_1 & = & \{0, 1, 2, 5, 6\} & w(S_1) = 15.0 \\ S_2 & = & \{3, 4, 7, 8, 8, 10, 11\} & w(S_2) = 7.0 \end{array}$$

- L'algorithme **glouton** choisit S_2, S_0 et S_1
- Alors que S_1 et S_2 **suffisent**

Démonstration (1/6)

- Pour simplifier l'argument, on montre plutôt que le ratio est $H(|X|)$

1: **fonction** SetCover(X : ensemble, $\mathcal{S}$: collection)
2: $Y \leftarrow \emptyset$
3: $S \leftarrow \emptyset$
4: **tant que** $Y \neq X$ **faire**
5: Soit $T \in \mathcal{S}$ qui minimise $poids(T)/|T - Y|$
6: **pour** $x \in T - Y$ **faire**
7: $poids(x) \leftarrow poids(T)/|T - Y|$
8: **fin pour**
9: $S \leftarrow S \cup \{T\}$
10: $Y \leftarrow Y \cup T$
11: **retourner** S

Démonstration (2/6)

- Soit $\mathcal{S}^* = \{S_1^*, S_2^*, \ldots, S_\ell^*\}$ une solution **optimale**
- Soit $\mathcal{S}_{greedy}$ la solution **gloutonne**
- Alors

$$poids(\mathcal{S}_{greedy}) = \sum_{x \in X} poids(x) \tag{1}$$

- Soient x_1, x_2, ..., x_n les éléments de X couverts par l'algorithme glouton dans l'**ordre**
- On montre maintenant que pour $i = 1, 2, \ldots, n$, on a

$$poids(x_i) \leq \frac{poids(\mathcal{S}^*)}{|X - X_i|} \tag{2}$$

Démonstration (3/6)

- Pour $i = 1, 2, \ldots, n$, soit $X_i = \{x_1, x_2, \ldots, x_i\}$
- Alors

$$\begin{aligned} X - X_i &= (X - X_i) \cap X \\ &= (X - X_i) \cap \left(\bigcup_{j=1}^{\ell} S_j^* \right) \\ &= \bigcup_{j=1}^{\ell} \left((X - X_i) \cap S_j^* \right) \end{aligned}$$

- Ceci implique

$$|X - X_i| \leq \sum_{j=1}^{\ell} \left| (X - X_i) \cap S_j^* \right| \tag{3}$$

Démonstration (4/6)

- Aussi, pour $i = 1, 2, \ldots, n$, remarquons que

$$poids(x_i) \leq \frac{poids(S_j^*)}{|(X - X_i) \cap S_j^*|}, \quad \text{pour tout } j = 1, 2, \ldots, \ell$$

- En effet, par **contradiction**, supposons qu'il existe $j \in \{1, 2, \ldots, \ell\}$ tel que

$$poids(x_i) > \frac{poids(S_j^*)}{|(X - X_i) \cap S_j^*|}$$

- Alors cela voudrait dire que l'algorithme glouton n'a pas choisi S_j^* plus tôt, alors qu'il avait un poids **moindre**
- L'inéquation se réécrit

$$poids(x_i)\ |(X - X_i) \cap S_j^*| \leq poids(S_j^*) \tag{4}$$

pour tout $j = 1, 2, \ldots, \ell$.

Démonstration (5/6)

Pour $i = 1, 2, \ldots, n$, on obtient donc

$$\begin{aligned} poids(\mathcal{S}^*) &= \sum_{j=1}^{\ell} poids(S_j^*) \\ &\geq \sum_{j=1}^{\ell} poids(x_i)\ |(X - X_i) \cap S_j^*| \\ &= poids(x_i) \sum_{j=1}^{\ell} |(X - X_i) \cap S_j^*| \\ &\geq poids(x_i)\ |X - X_i| \end{aligned}$$

Ainsi,

$$poids(x_i) \leq \frac{poids(\mathcal{S}^*)}{|X - X_i|}$$

Démonstration (6/6)

Enfin,

$$
\begin{aligned}
poids(\mathcal{S}_{greedy}) &= \sum_{i=1}^{n} poids(x_i) \\
&\leq \sum_{i=1}^{n} \frac{poids(\mathcal{S}^*)}{n-i+1} \\
&= poids(\mathcal{S}^*) \sum_{i=1}^{n} \frac{1}{n-i+1} \\
&= poids(\mathcal{S}^*) \sum_{i=1}^{n} \frac{1}{i} \\
&= poids(\mathcal{S}^*)\ H(n)
\end{aligned}
$$

Implémentation (1/4)

```
$ bin/set-cover < examples/example-1.setcover
A set cover instance of 10 elements and 8 subsets
  Subset #0 (w = 2.00): 0 1 3 5
  Subset #1 (w = 1.00): 0 2 4 8 9
  Subset #2 (w = 3.00): 1 3 5 7 9
  Subset #3 (w = 2.00): 6 7 8 9
  Subset #4 (w = 1.00): 1 4
  Subset #5 (w = 3.00): 2 4 6
  Subset #6 (w = 4.00): 0 1 2 3 5 8 9
  Subset #7 (w = 5.00): 1 2 4 8
A collection of 0 (out of 8) subsets covering 0 (out of 10) elements
Total weight: 0.000000
Adding subset #1
A collection of 1 (out of 8) subsets covering 5 (out of 10) elements
  Subset #1 (w = 1.00): 0 2 4 8 9
Total weight: 1.000000
Adding subset #0
A collection of 2 (out of 8) subsets covering 8 (out of 10) elements
  Subset #0 (w = 2.00): 0 1 3 5
  Subset #1 (w = 1.00): 0 2 4 8 9
Total weight: 3.000000
Adding subset #3
A collection of 3 (out of 8) subsets covering 10 (out of 10) elements
  Subset #0 (w = 2.00): 0 1 3 5
  Subset #1 (w = 1.00): 0 2 4 8 9
  Subset #3 (w = 2.00): 6 7 8 9
Total weight: 5.000000
```

Implémentation (2/4)

```
struct Instance {                // An instance of the set cover problem
    unsigned int num_elements; // The number of elements
    unsigned int num_subsets;  // The number of subsets
    bool **subsets;            // An array of subsets
    float *weights;            // The weight for each subset
};

struct Collection {                      // A collection of subsets
    const struct Instance *instance; // A set cover instance
    unsigned int num_elements;       // The number of elements
    unsigned int num_subsets;        // The number of subsets
    bool *has_subset;                // The subset belongs to the collection?
    bool *is_covered;                // Is the element covered?
    float weight;                    // The weight of the collection
};
```

Implémentation (3/4)

```
// Instance
void load_instance(struct Instance *instance);
void print_subset(const struct Instance *instance,
                  unsigned int s);
void print_instance(const struct Instance *instance);
void free_instance(struct Instance *instance);

// Collection
void alloc_collection(struct Collection *collection,
                      const struct Instance *instance);
void free_collection(struct Collection *collection);
void print_collection(const struct Collection *collection);
void add_subset_to_collection(struct Collection *collection,
                              unsigned int s);
unsigned int num_uncovered(const struct Collection *collection,
                           unsigned int s);
unsigned int subset_with_lowest_cost(const struct Collection *collection);
```

Implémentation (4/4)

```
int main(int argc, char *argv[]) {
    struct Instance instance;
    load_instance(&instance);
    print_instance(&instance);
    struct Collection collection;
    alloc_collection(&collection, &instance);
    print_collection(&collection);
    do {
        unsigned int s = subset_with_lowest_cost(&collection);
        add_subset_to_collection(&collection, s);
        printf("Adding subset #%d\n", s);
        print_collection(&collection);
    } while (collection.num_elements < instance.num_elements);
    free_collection(&collection);
    free_instance(&instance);
    return 0;
}
```

Couverture des arêtes par les sommets

Définition

- Soit $G = (V, E)$ un **graphe** non orienté
- Soit $U \subseteq V$
- On dit que U est une **couverture des arêtes** par les sommets si pour toute arête $\{u, v\}$, on a $u \in U$ ou $v \in U$

Difficulté

- Trouver une **couverture** minimale est NP-difficile
- Problème appelé *vertex cover* en anglais
- Nous avons vu un algorithme **résoluble à paramètre fixé** si la taille d'une solution k est **bornée**

Question

- Le problème *vertex cover* est un **cas particulier** du problème *set cover*
- **Montrer** pourquoi c'est vrai
- En déduire un **algorithme d'approximation** glouton
- Calculer le **ratio d'approximation**

Programmation linéaire

- Le problème *vertex cover* peut être approximé plus **efficacement**
- Plus précisément, on connaît un **algorithme 2-approximatif**
- L'idée consiste à traduire le problème en un **programme linéaire en nombres entiers**
- En anglais, *ILP* $=$ *integer linear program*
- On utilise ensuite une méthode appelée **primale-duale**
- Elle s'applique à plusieurs autres **situations**

Modélisation du problème

- Soit $G = (V, E)$ un **graphe** non orienté
- Soit $w : V \to \mathbb{R}_+$ une fonction de **poids**
- Pour chaque $v \in V$, soit $x_v \in \{0, 1\}$ une **variable binaire**
- On souhaite

$$\text{minimiser} \quad \sum_{v \in V} w(v) x_v$$

$$\text{sous les contraintes} \quad x_u + x_v \geq 1, \text{ pour toute } \{u, v\} \in E$$

- Le problème est évidemment **difficile** à résoudre
- En revanche, il devient **facile** si on permet que $x_v \in [0, 1] \cap \mathbb{R}$

Problème dual

- Tout programme linéaire a un problème **dual** (nous y reviendrons)
- Dans notre cas, soit $N(v)$ l'ensemble des **voisins** de v
- Et soit $y_{u,v} \geq 0$ une variable associée à **chaque arête**
- Le problème dual consiste à

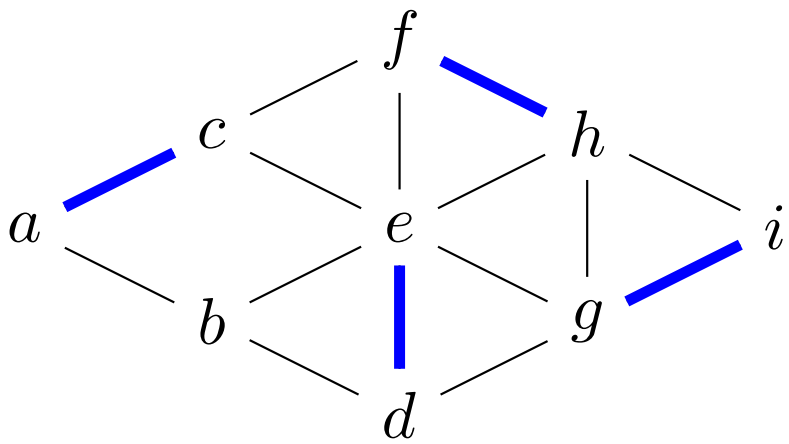
$$\text{maximiser} \quad \sum_{\{u,v\} \in E} y_{u,v}$$

$$\text{sous les contraintes} \quad \sum_{u \in N(v)} y_{u,v} \leq w(v), \text{ pour tout } v \in V$$

- Dans le cas où $w(v) = 1$ pour tout $v \in V$, le problème dual revient à calculer un **couplage maximal** dans G

Problème du couplage

- Soit $G = (V, E)$ un **graphe** non orienté
- Un sous-graphe $M = (V', E')$ de G est appelé un **couplage** de G si chacun de ses sommets est de degré **au plus** 1



Primal-dual (1/2)

- Problème **primal**:

$$\text{minimiser} \quad \sum_{v \in V} w(v) x_v$$

$$\text{sous les contraintes} \quad x_u + x_v \geq 1, \text{ pour toute } \{u, v\} \in E$$

- Problème **dual**:

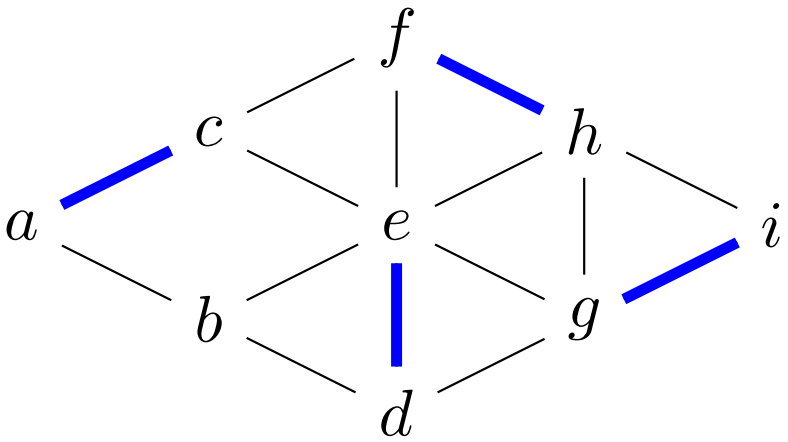
$$\text{maximiser} \quad \sum_{\{u,v\} \in E} y_{u,v}$$

$$\text{sous les contraintes} \quad \sum_{u \in N(v)} y_{u,v} \leq w(v), \text{ pour tout } v \in V$$

- Propriété de **dualité faible**: si x et y sont des solutions admissibles, alors

$$\sum_{\{u,v\} \in E} y_{u,v} \leq \sum_{v \in V} w(v) x_v$$

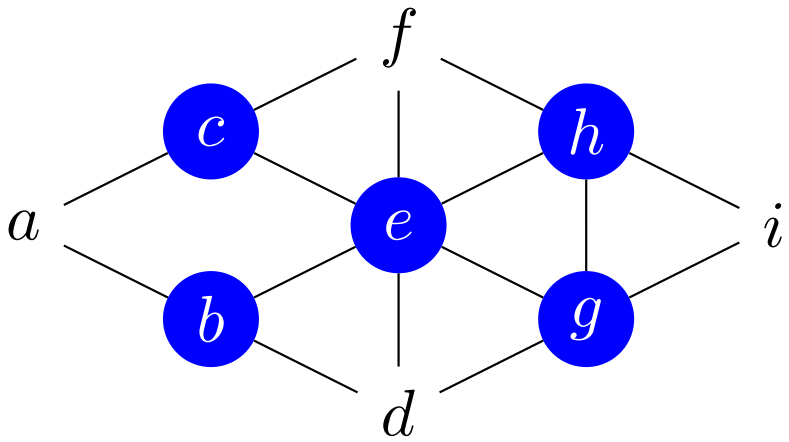
Primal-dual (2/2)



Couplage maximal

4 arêtes

$\leq$



Couverture minimale

5 sommets

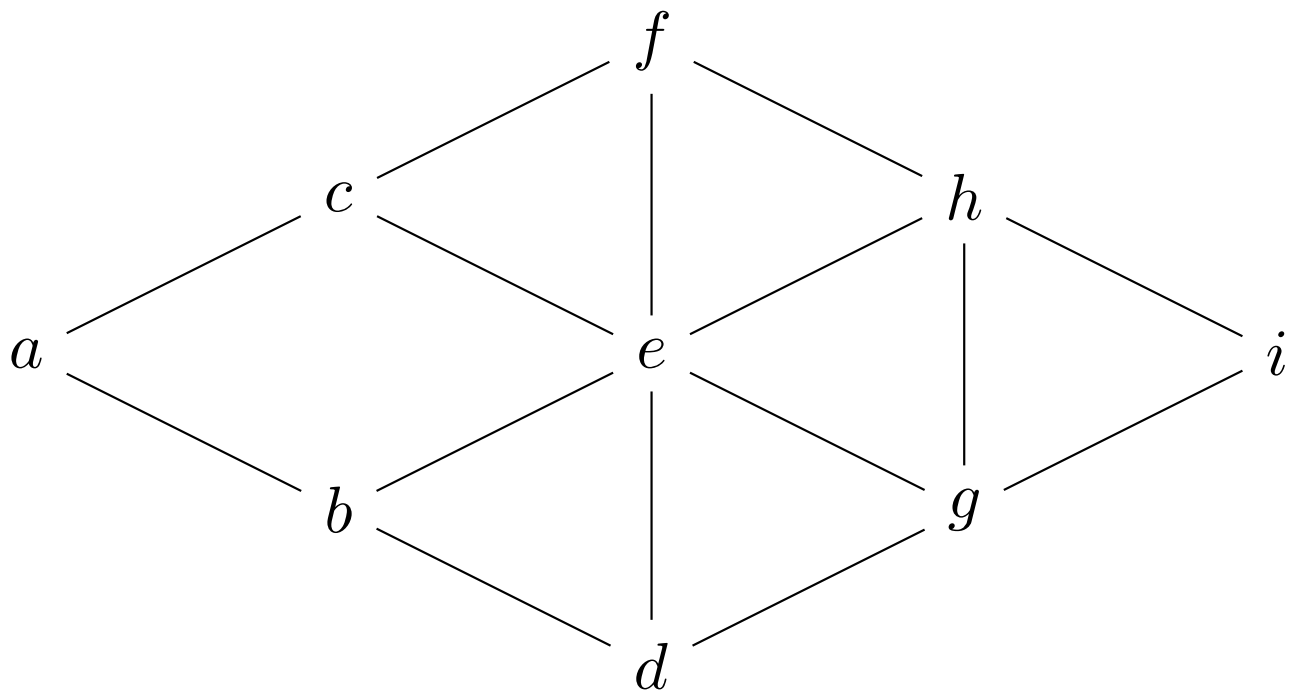
Algorithme d'approximation: idée générale

- On considère le problème **dual**
- On initialise toutes les valeurs à 0
- On augmente le plus possible et **uniformément** toutes les variables, sauf celles qui sont déjà à leur limite
- Puis on répète jusqu'à ce qu'on ne **puisse plus augmenter**
- On obtient la **couverture** en retenant les variables pour lesquelles la contrainte a été atteinte
- Noter que la contrainte est **atteinte** si

$$\sum_{u \in N(v)} y_{u,v} = w(v)$$

Exemple

- Soit le graphe ci-bas avec $w(v) = 1$ pour tout $v \in V$
- Alors on obtient la couverture $\{b, c, e, g, h\}$



Algorithme glouton (Clarkson, 1983)

1: **fonction** CouvertureApproximative(G : graphe) : couverture
2: $W(v) \leftarrow w(v)$ pour tout $v \in V$
3: $D(v) \leftarrow \deg(v)$ pour tout $v \in V$
4: $S \leftarrow \emptyset$
5: **tant que** $E \neq \emptyset$ **faire**
6: Soit $v \in V$ tel que $W(v)/D(v)$ est minimal
7: **pour** $u \in N(v)$ **faire**
8: $E \leftarrow E - \{\{u, v\}\}$
9: $W(u) \leftarrow W(u) - W(v)/D(v)$
10: $D(u) \leftarrow D(u) - 1$
11: $S \leftarrow S \cup \{v\}$
12: $V \leftarrow V - \{v\}$

Ratio d'approximation

Théorème

L'algorithme précédent retourne une couverture des arêtes par les sommets avec un **ratio d'approximation** de 2

Idée de la preuve

- Soit S l'**ensemble** obtenu par l'algorithme
- S est bien une **couverture** puisque pour chaque arête, $\{u, v\}$, on a

$$\sum_{w \in N(u)} y_{u,w} = w(u) \quad \text{ou} \quad \sum_{w \in N(v)} y_{v,w} = w(v)$$

Recherche locale

Idée de base

- Soit f la fonction **objectif** (à minimiser)
- On démarre avec une solution **admissible** S
- On explore le **voisinage** $\mathcal{N}(S)$ de S
- On sélectionne $S' \in \mathcal{N}(S)$ tel que $f(S')$ est minimal
- Si $f(S') < f(S)$, alors on remplace S par S'
- Puis on **répète**
- Sinon, on **arrête** et on retourne le résultat

Remarque

- Il s'agit d'un algorithme **glouton** puisqu'on sélectionne la meilleure solution localement
- Comme il n'y a pas de **retour en arrière** (*backtracking*), la solution retournée n'est généralement **pas optimale**

Pseudocode

1: **fonction** RechercheLocale(S : solution admissible)
2: $S_{\text{best}} \leftarrow S$
3: **pour** $S' \in \mathcal{N}(S)$ **faire**
4: **si** $f(S') < f(S)$ **alors**
5: $S_{\text{best}} \leftarrow S'$
6: **si** $f(S_{\text{best}}) < f(S)$ **alors**
7: **retourner** RechercheLocale(S_{best})
8: **sinon**
9: **retourner** S

Fonction de voisinage et distance

Voisinage

- Soit $\mathcal{S}$ l'espace des solutions **admissibles**
- Une **fonction de voisinage** pour $\mathcal{S}$ est une fonction de la forme

$$\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$$

telle que $\mathrm{dist}(S, \mathcal{N}(S))$ est *petite* pour toute $S \in \mathcal{S}$

Distance

On dit que dist est une **distance** si elle respecte les propriétés suivantes:

1. $\mathrm{dist}(S, S') \geq 0$
2. $\mathrm{dist}(S, S') = 0 \Leftrightarrow S = S'$
3. $\mathrm{dist}(S, S') = \mathrm{dist}(S', S)$
4. $\mathrm{dist}(S, S') \leq \mathrm{dist}(S, S'') + \mathrm{dist}(S'', S')$

Écart local (*locality gap*)

- Soit f une fonction objectif qu'on souhaite **minimiser**
- Soit $S \in \mathcal{S}$ une solution **admissible**
- Alors S est un **minimum global** si $f(S) \leq f(S')$ pour tout $S' \in \mathcal{S}$
- Et S est un **minimum local** si $f(S) \leq f(S')$ pour tout $S' \in \mathcal{N}(S)$
- Soit $\mathcal{L}$ l'ensemble de tous les **minimums locaux**
- Et soit S^* un **minimum global**
- Alors on définit l'**écart local** (*locality gap*) par

$$\alpha = \max \left\{ \frac{f(S)}{f(S^*)} \;\middle|\; S \in \mathcal{L} \right\}$$

Question

- Peut-on **borner** le ratio α ?
- Si oui, alors on obtient un **algorithme d'approximation** de ratio α

Traitement de tâches

- Un ensemble de tâches J
- Un ensemble de machines M
- Une fonction $t : J \to \mathbb{R}^*$ qui indique le temps pris par chaque tâche
- Objectif: compléter toutes les tâches en un temps minimum

Exemple d'instance

- $J = \{j_1, j_2, j_3, j_4, j_5, j_6\}$
- $M = \{m_1, m_2, m_3\}$
- La fonction t est donnée par le tableau suivant

i	1	2	3	4	5	6
$t(j_i)$	3.0	2.0	3.0	4.0	2.0	1.0

Représentation possible

- Une fonction $S : M \rightarrow 2^J$ qui associe à chaque machine la liste des tâches qu'elle traitera
- Exemple: $S(m_1) = \{j_1, j_2, j_6\}$, $S(m_2) = \{j_4\}$, $S(m_3) = \{j_5, j_3\}$
- Calcul de $t(S)$ pour tout S:

$$t(S) = \max \left\{ \sum_{j \in S(m)} t(j) \;\middle|\; m \in M \right\}$$

Représentation alternative

- Une fonction $S : J \rightarrow M$ qui associe à chaque tâche une machine
- Plus facile à **implémenter**!

Fonction de voisinage

Question

- Soit $S : J \rightarrow M$ une solution
- Comment peut-on modifier S pour obtenir une solution S' **proche** de S?

Fonction de voisinage

Question

- Soit $S : J \to M$ une solution
- Comment peut-on modifier S pour obtenir une solution S' **proche** de S?

Réponses

- (*jump*) On choisit une **tâche au hasard** et on la change de machine
- (k-*jump*) On choisit k **tâches au hasard** et on les change de machine
- (*optimal jump*) On choisit une tâche traitée par une machine de **charge maximale** et on la place sur une machine de **charge minimale**
- (*swap*) On **échange** deux tâches qui sont sur des machines différentes

Théorème (Graham, 1966)

- Soit $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ la fonction qui associe à une solution S toutes les solutions obtenues de S en déplaçant une tâche d'une machine **maximalement chargée** vers une machine **minimalement chargée**
- Soit $f : \mathcal{S} \to \mathbb{R} \times \mathbb{N}$ la fonction **objectif** qui associe à une solution S le couple (t, m), où t est temps maximal de traitement d'une machine et m le nombre de machines dont la durée est maximale
- Soit $\prec$ la relation définie sur $\mathbb{R} \times \mathbb{N}$ par

$$(t_1, m_1) \prec (t_2, m_2) \quad \text{ssi} \quad t_1 < t_2 \text{ ou } (t_1 = t_2 \text{ et } m_1 < m_2)$$

- Alors l'algorithme d'approximation utilisant $\mathcal{N}$ et f trouve toujours une solution avec **écart local** d'au plus $2 - 1/m$, où m est le nombre de **machines**

Implémentation (1/2)

Code source complet disponible sur GitLab

```
struct Batch {                    // A batch to process
    unsigned int num_machines; // The number of available machines
    unsigned int num_jobs;     // The number of jobs to process
    float *durations;          // The duration of each job
};

struct Schedule {                       // A schedule for processing the batch
    const struct Batch *batch;          // The scheduled batch
    unsigned int *assigned_machine; // The machine assigned to each job
    float *duration_by_machine;     // The total duration for each machine
};

struct Search {                  // A naive search
    struct Schedule schedule; // The current schedule
    float optimal_duration;   // The optimal duration found so far
    bool verbose;             // If true, prints trace to stdout
};
```

Implémentation (2/2)

```
// Operation on batches
void load_batch(struct Batch *batch);
void print_batch(const struct Batch *batch);
void free_batch(struct Batch *batch);

// Operation on schedules
float min_duration(const struct Schedule *schedule);
float max_duration(const struct Schedule *schedule);
unsigned int num_machines_max_load(const struct Schedule *schedule);
void update_schedule_durations(struct Schedule *schedule);
void alloc_schedule(struct Schedule *schedule,
                    const struct Batch *batch);
void initialize_random_schedule(struct Schedule *schedule,
                                const struct Batch *batch);
void print_schedule(const struct Schedule *schedule);
void free_schedule(struct Schedule *schedule);

// Approximation algorithm
void move_job(struct Schedule *schedule,
              unsigned int j, unsigned int m);
bool improve_schedule(struct Schedule *schedule);

// Naive algorithm
void compute_optimal_schedule_recursive(struct Search *search,
                                        unsigned int j);
float optimal_duration(const struct Batch *batch,
                       bool verbose);
```

Représentation d'un ensemble de tâches

- Format **simple** pour minimiser le travail de chargement
- Soit `m` et `j` le nombre de machines et le nombre de tâches
- Alors on utilise le **format** suivant:

```
m j
t1
t2
[...]
tj
```

Exemple:

```
$ cat examples/example-1.jobs
3 6
3
2
[...]
1
```

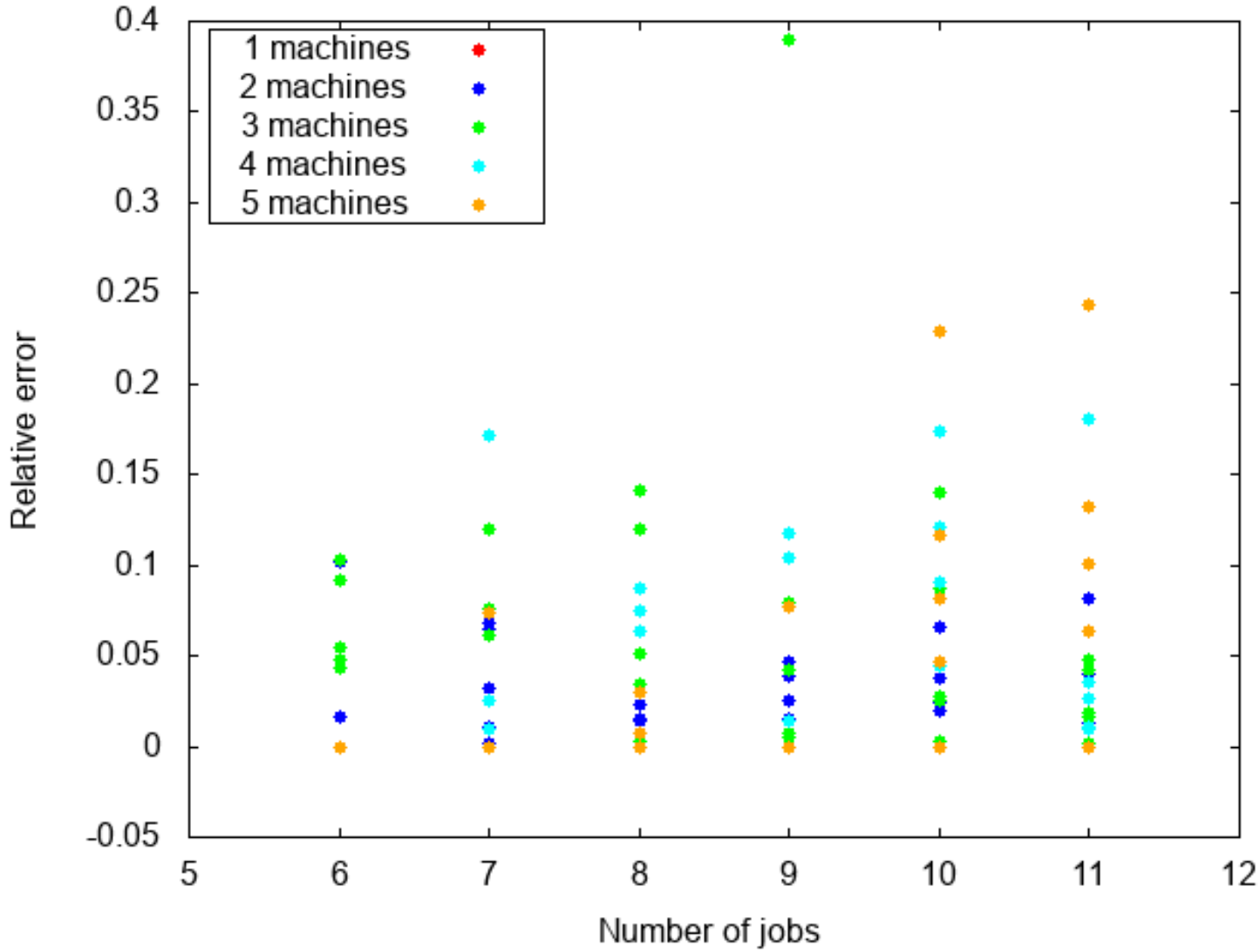
Exemple d'exécution

```
$ bin/jobs-processing rn < examples/example-1.jobs
5.000000

$ bin/jobs-processing ra < examples/example-1.jobs
6.000000

$ bin/jobs-processing va < examples/example-1.jobs
A batch of 6 jobs to be processed by 3 machines whose durations are
  t(j0) = 3.00
  t(j1) = 2.00
  t(j2) = 3.00
  t(j3) = 4.00
  t(j4) = 2.00
  t(j5) = 1.00
[...]
A schedule of 6 jobs dispatched onto 3 machines as follows:
  Job #0 is assigned to machine #0
  Job #1 is assigned to machine #2
  Job #2 is assigned to machine #0
  Job #3 is assigned to machine #1
  Job #4 is assigned to machine #2
  Job #5 is assigned to machine #2
  Machine #0's duration is 6.000000
  Machine #1's duration is 4.000000
  Machine #2's duration is 5.000000
6.000000
```

Expérimentation

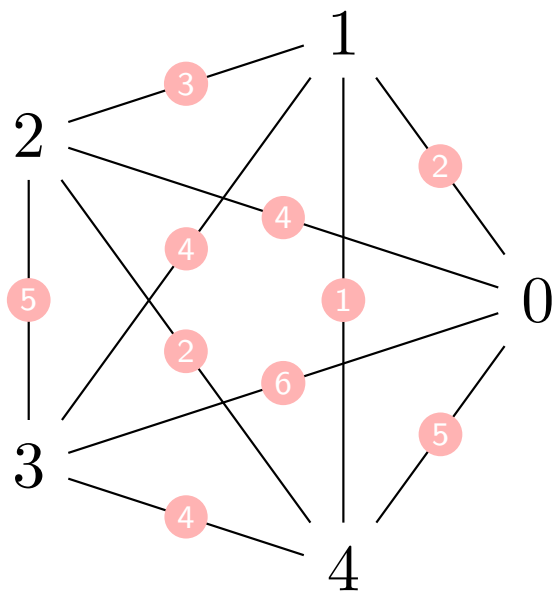


Voyageur de commerce

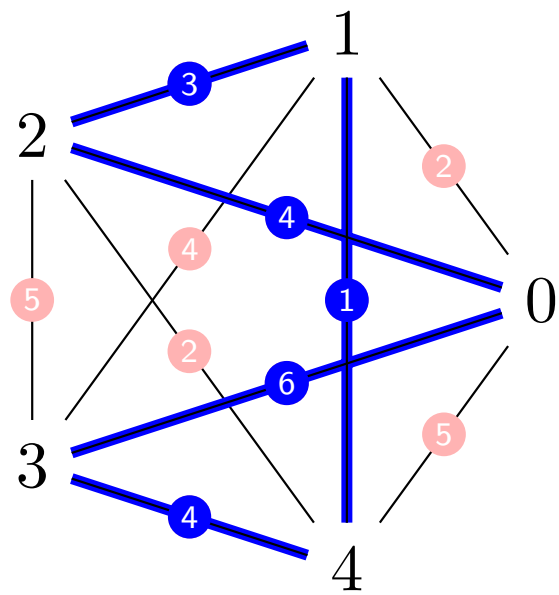
- Soit $G = (V, E)$ le graphe simple **complet** de n sommets
- On considère une fonction de **poids** sur les arêtes

$$w : E \to \mathbb{R}_+ \cup \{+\infty\}$$

- On cherche un **cycle hamiltonien** dont le poids total est **minimal**



4! tournées possibles



poids $= 18$

Difficulté

- C'est un problème **NP-difficile**
- Réduction à partir du problème du **cycle hamiltonien**
- Espace de solutions: l'ensemble des **permutations** de n
- Le nombre de **solutions** possibles est donc $n!$
- Devient $(n-1)!$ si on fixe la **première ville**
- Plusieurs approches **non exactes** proposées, généralement assez **efficaces**

Fonction de voisinage

- Soit n un **entier** positif
- Soit C l'ensemble des cycles **hamiltoniens** du graphe complet de n sommets
- Soit $c \in C$
- Soient $e_1 = \{u_1, v_1\}$ et $e_2 = \{u_2, v_2\}$ deux arêtes de c, où u_1, v_1, u_2, v_2 apparaissent **dans l'ordre** dans c
- Alors on peut construire un nouveau cycle hamiltonien c' en **remplaçant** les arêtes e_1 et e_2 par $e'_1 = \{u_1, u_2\}$ et $e'_2 = \{v_1, v_2\}$
- Soit $\mathcal{N} : C \rightarrow 2^C$ l'ensemble des voisins obtenus par un **échange** d'arêtes
- Que vaut $|\mathcal{N}(c)|$?

Fonction de voisinage

- Soit n un **entier** positif
- Soit C l'ensemble des cycles **hamiltoniens** du graphe complet de n sommets
- Soit $c \in C$
- Soient $e_1 = \{u_1, v_1\}$ et $e_2 = \{u_2, v_2\}$ deux arêtes de c, où u_1, v_1, u_2, v_2 apparaissent **dans l'ordre** dans c
- Alors on peut construire un nouveau cycle hamiltonien c' en **remplaçant** les arêtes e_1 et e_2 par $e'_1 = \{u_1, u_2\}$ et $e'_2 = \{v_1, v_2\}$
- Soit $\mathcal{N} : C \to 2^C$ l'ensemble des voisins obtenus par un **échange** d'arêtes
- Que vaut $|\mathcal{N}(c)|$?

$\to$ J'ai n choix pour la **1re arête**

$\to$ J'ai $n-1$ choix pour la **2e arête**

$\to$ On divise par 2 pour tenir compte de la **symétrie**

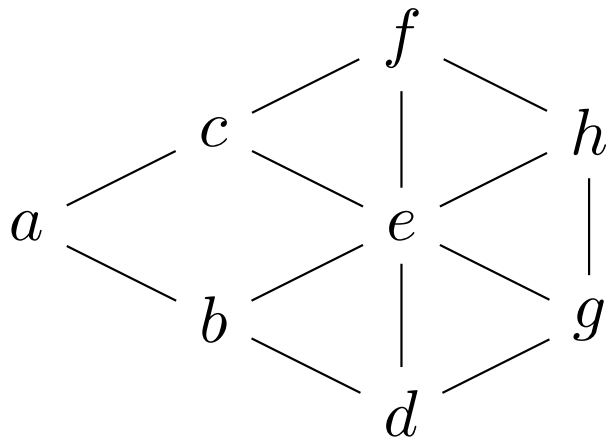
$\to$ Donc $n(n-1)/2 \in \Theta(n^2)$ **voisins**

Recherche locale

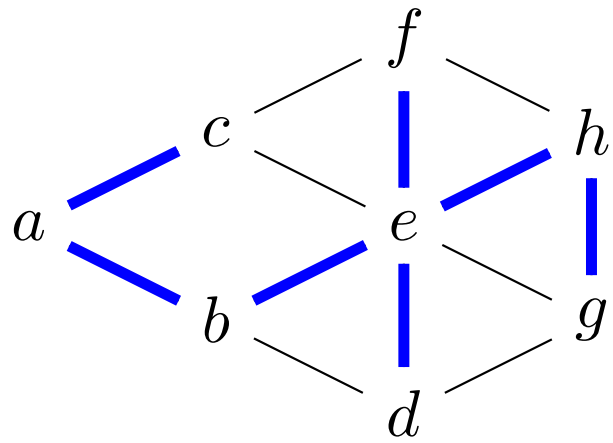
- En **1995**, Johnson et McGeoch ont étudié l'heuristique basée sur la **recherche locale** donnait de bons résultats en pratique
- En particulier, ils ont montré que si les n villes sont distribuées **aléatoirement** dans le plan...
- ...alors en **moyenne**, on obtient des solutions dont l'écart local est de **25%**
- Nous verrons que cette heuristique est très **performante** lorsque combinée à d'autres (colonies de fourmis, algorithmes mémétiques, etc.)

Arbres de recouvrement

- Soit $G = (V, E)$ un graphe simple
- Soit $T = (V', E')$ un sous-arbre de G
- T est un **arbre de recouvrement** de G si $V' = V$ et $E' \subseteq E$
- Algorithmes connus: **Prim** et **Kruskal** (voir INF7440, par exemple), qui s'exécutent en temps **polynomial**
- **Variante**: on cherche un arbre de recouvrement qui **maximise** le nombre de **feuilles** dans l'arbre
- Cette variante est **NP-difficile**



Un graphe

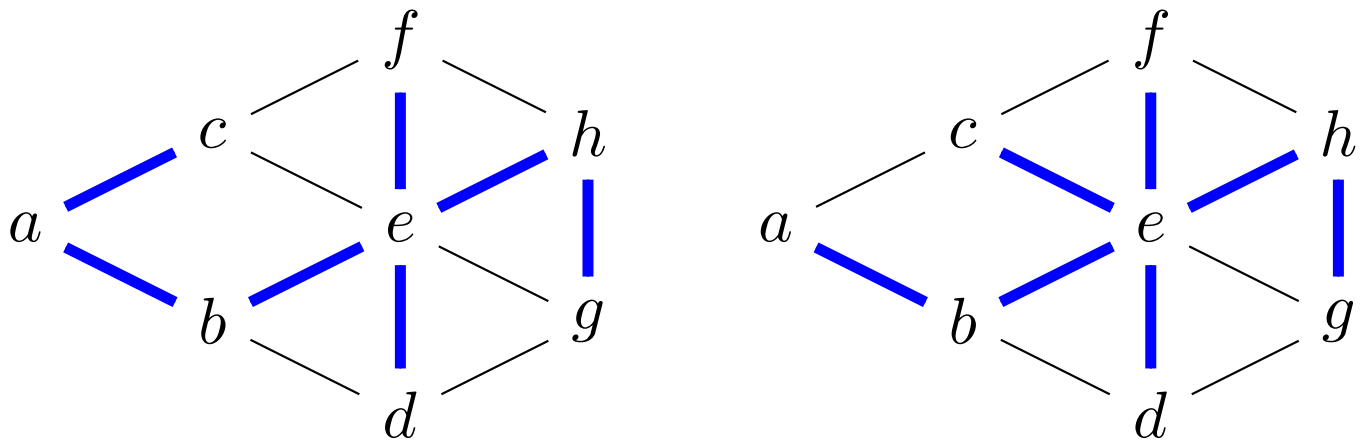


Arbre de recouvrement

Fonction de voisinage

- Soient T et T' **deux** arbres de recouvrement
- Alors on dit que T et T' sont **voisins** s'ils diffèrent d'exactement **deux arêtes**
- Ainsi, $\mathcal{S}$ est l'ensemble des arbres de recouvrement et

$$\mathcal{N}(T) = \{T' \in \mathcal{S} \mid T \text{ et } T' \text{ diffèrent de deux arêtes}\}$$



Deux arbres voisins

Pseudocode

1: **fonction** RechercheLocale(T : arbre de recouvrement)
2: $T_{\text{best}} \leftarrow T$
3: **pour chaque** $\{u, v\} \in E(G) - E(T)$ **faire**
4: $c \leftarrow$ l'unique chemin dans T entre u et v
5: **pour** $\{u', v'\} \in E(c)$ **faire**
6: $T' \leftarrow T \cup \{\{u, v\}\} - \{\{u', v'\}\}$
7: **si** T' a plus de feuilles que T **alors**
8: $T_{\text{best}} \leftarrow T'$
9: **si** T_{best} a moins de feuilles que T **alors**
10: **retourner** RechercheLocale(T_{best})
11: **sinon**
12: **retourner** T

Complexité

- Supposons que $G = (V, E)$ ait n **sommets** et m **arêtes**
- On choisit une **arête** $\{u, v\}$ de G qui n'est pas dans T
- Il y a un **unique chemin** c entre u et v dans T
- On obtient T' en **ajoutant** $\{u, v\}$ et en **enlevant** n'importe quelle arête $\{u', v'\}$ dans le chemin c
- Ainsi, on a au plus m **choix** pour $\{u, v\}$
- Et au plus $n - 1$ **choix** pour $\{u', v'\}$
- Donc au total, il y a $\mathcal{O}(mn)$ **voisins**

Nombre d'itérations

- On commence avec au moins 2 **feuilles**
- À chaque tour, on augmente d'**au plus** 1 feuille
- On finit avec au plus $n - 1$ **feuilles**
- Donc au total au plus $n - 3$ **itérations**

Complexité totale: $\mathcal{O}(mn^2)$

Écart local

Théorème (Lu et Ravi, 1992)

- Soit α le ratio de l'écart maximal entre un optimum local et un optimum global
- Alors $\alpha \leq 5$

Théorème (Lu et Ravi, 1992)

- Si on permet d'échanger **deux paires d'arêtes**, alors $\alpha \leq 3$

Démonstration

Arguments **combinatoires** basés sur les propriétés des arbres de recouvrement et sur le lien entre degré et feuilles (lemme des poignées de main)

Recuit simulé

Idée générale

Combinaison de deux stratégies

- On souhaite améliorer **localement** la solution actuelle

$\rightarrow$ Recherche locale

- Et permettre aussi des **sauts** dans l'espace de solution

$\rightarrow$ Diversification

Inspiration

- Recuit $\rightarrow$ obtenir un état de basse énergie dans un solide
- Initialement, la température est **élevée**

$\rightarrow$ l'arrangement est aléatoire et irrégulier

- Puis on la diminue **progressivement** pour stabiliser

$\rightarrow$ la structure se cristallise et s'organise

- La température initiale doit être **suffisamment élevée**
- Le refroidissement doit être **suffisamment lent**

Modèle

- En **1953**, Metropolis et al. proposent un modèle discret pour **simuler** l'évolution d'un solide qui subit un recuit
- Utilise une technique de **Monte Carlo** (algorithme randomisé)

Idée

- Le solide se trouve dans un état i, pour $i = 1, 2, \ldots, n$
- Soit E_i l'**énergie** du solide à l'état i
- On calcule un état j en déformant **légèrement** i
- Si $E_j - E_i \leq 0$, alors $E_{i+1} \leftarrow E_j$
- Sinon, on change l'état avec probabilité

$$\exp\left(\frac{E_i - E_j}{k_B T}\right)$$

où T est la **température courante** et k_B est la constante de Boltzmann

Pseudocode

1: **procedure** RecuitSimulé
2: Soit S une solution initiale
3: Soit c un réel positif
4: **tant que** condition d'arrêt **faire**
5: **pour** $i \leftarrow 1, 2, \ldots, L$ **faire**
6: Choisir aléatoirement un voisin S' de S
7: **si** S' est meilleur que S **alors**
8: $S \leftarrow S'$
9: **sinon**
10: Soit $r \in [0, 1)$ choisi aléatoirement
11: **si** $r < \exp((f(S) - f(S'))/c)$ **alors** $S \leftarrow S'$
12: Mettre à jour L et c

Paramètres

Deux paramètres

- c: paramètre de contrôle

$\rightarrow$ doit évoluer avec le temps

- L: nombre de voisins à considérer

$\rightarrow$ généralement fixé

Plan de refroidissement

- En anglais, *cooling schedule*
- On doit préciser la valeur **initiale** c_0
- Et comment calculer c_{i+1} en fonction de c_i, pour $i = 0, 1, ...$

Plan de refroidissement statique

Initialisation

- On peut choisir $c_0 \approx \Delta f_{\max}$ où $\Delta f_{\max}$ est la différence de coût maximale entre deux **voisins** quelconques

$\rightarrow$ généralement impossible à calculer de façon exacte

$\rightarrow$ souvent approximé

Mise à jour

- On prend $c_{i+1} = \alpha c_i$, où $0 < \alpha < 1$
- Typiquement, on prend $0.8 \leq \alpha \leq 0.99$

Nombre de répétitions

Souvent L est proportionnel à la taille du voisinage

Plan de refroidissement dynamique

Initialisation

- On fixe c_0 à une petite valeur
- Puis on multiplie c_0 par une constante > 1 jusqu'à ce que le ratio de solutions acceptées approche 1

Nombre de répétitions

S'il y a beaucoup d'**améliorations**, on augmente le nombre d'itérations

Terminaison

On arrête lorsqu'il y a **stagnation**

Comportement asymptotique

Hypothèses

- Le graphe de **voisinage** est fortement connexe
- Tout état est atteignable de tout autre état à partir d'une **hauteur** non nulle
- La famille $\{c_i\}_{i=1}^{+\infty}$ satisfait

$$c_i = \frac{\Gamma}{\log(i+1)}$$

pour une constante Γ qui majore l'écart entre un maximum local et le maximum global

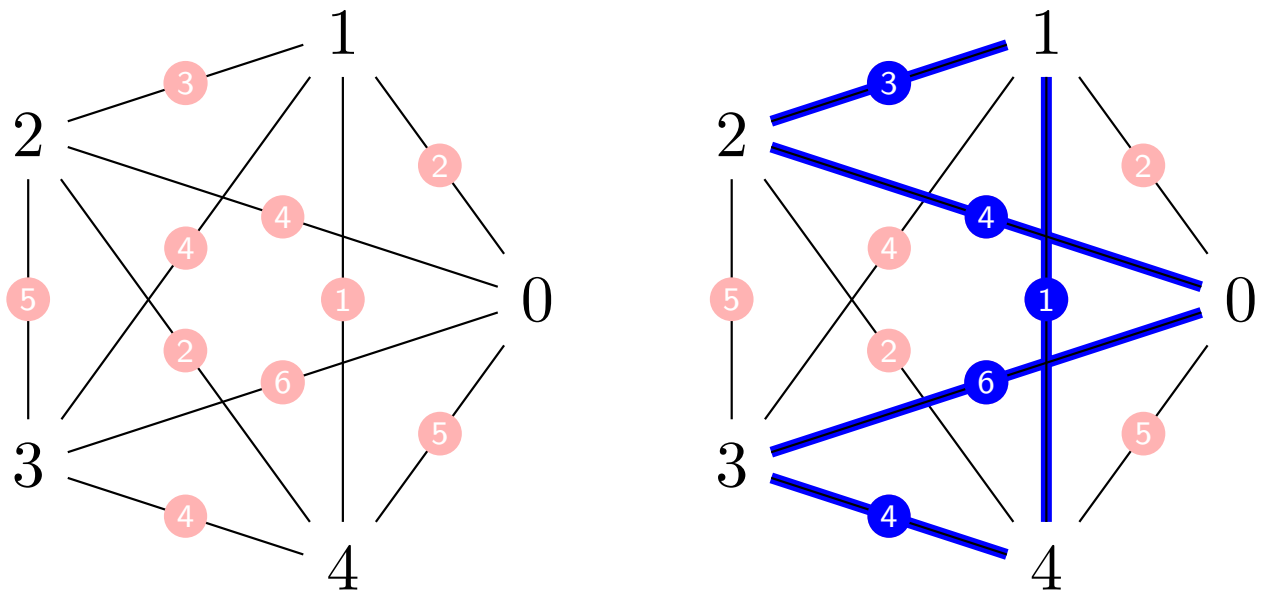
Convergence

- Alors l'algorithme **converge** vers une solution optimale
- Si on attend **assez longtemps**

Voyageur de commerce

- Soit $G = (V, E)$ le graphe simple **complet** de n sommets
- On considère une fonction de **poids** sur les arêtes

$$w : E \to \mathbb{R}_+ \cup \{+\infty\}$$



- On cherche une tournée de **poids minimal**
- Voir démonstration

Recherche taboue

Caractéristiques principales

Mémoire flexible

- Tient compte de l'**historique**
- **Sans mémoire**: algorithmes randomisés, recuit simulé
- **Mémoire rigide**: séparation et évaluation progressive, recherche A*

Mécanismes de contrôle

- **Contraintes**: restrictions taboues
- **Liberté**: critère d'aspiration

Type de mémoire

- **Court terme**: intensification de la recherche
- **Long terme**: diversification de la recherche

Pseudocode

1: **procedure** RechercheTaboue
2: Soit S une solution initiale
3: $B \leftarrow \emptyset$
4: **tant que** il reste des itérations **ou** pas de stagnation **faire**
5: $S' \leftarrow \emptyset$
6: **pour chaque** solution admissible S'' voisine de S **faire**
7: Si S'' est strictement meilleur que S' alors $S' \leftarrow S''$
8: Si S'' est strictement meilleur que B alors $B \leftarrow S''$
9: $S \leftarrow S'$
10: Mettre à jour les restrictions taboues

Choix du meilleur candidat

- On vérifie chaque **candidat** potentiel
- **Meilleur** trouvé jusqu'à maintenant?

→ **oui**: mettre à jour

- Est-ce que le candidat est **tabou**?

→ **non**: mettre à jour comme candidat admissible

→ **oui**: satisfait le critère d'**aspiration**?

→ **oui**: mettre à jour comme candidat admissible

→ **non**: on passe au candidat suivant

Critère d'aspiration

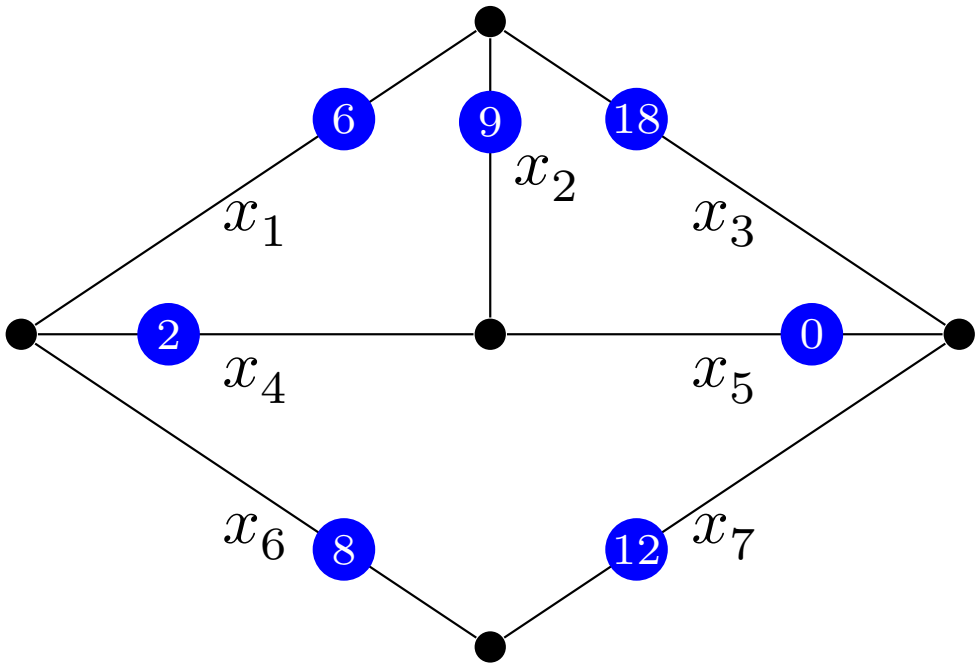
- Permet de **court-circuiter** le tabou
- **Exemple**: solution strictement meilleure depuis le début, découverte d'un nouveau sous-espace de recherche

Exemple: arbre de recouvrement contraint

- On considère un graphe simple $G = (V, E)$
- On cherche un arbre de **recouvrement** T de G de poids minimal
- Respectant certaines **contraintes**

$\rightarrow$ on ne peut choisir **simultanément** certaines arêtes

$\rightarrow$ on doit choisir certaines arêtes **conditionnellement**



$$x_1 + x_2 + x_6 \leq 1$$

$$x_1 \leq x_3$$

Pénalité: 50

Stratégie taboue

Voisinage

T et T' sont voisins s'ils diffèrent d'exactement **deux arêtes**

Restriction taboue

Chaque arête **ajoutée** devient taboue pour les deux prochaines itérations

Critère d'aspiration

Si on trouve un nouveau maximum global, on le sauvegarde

Exemple

Effectuer la trace sur l'exemple précédent en commençant avec

$$\{x_1, x_4, x_5, x_6\}$$

Observations

Taille de la liste taboue?

- **Expérimentation**: entre 5 et 12 semble optimal
- Nombre **récurrent**: 7

Diversification

- Recherche taboue souvent **combinée** à d'autres stratégies
- Différents types de **mémoire**

En pratique

- Excellents **résultats** dans plusieurs problèmes
- Très **générique**: peut être appliqué à n'importe quel problème d'optimisation en principe