Algorithmes probabilistes

Montaigne 2023-2024

– mpi23@arrtes.net –

Objectifs

- Résoudre ou approcher la solution d'un problème difficile à l'aide d'un algorithme faisant intervenir l'**aléatoire**.
- Distinguer les **algorithmes de Las Vegas** et les **algorithmes de Monte Carlo**.
- Connaître quelques exemples.

Généralités

Un algorithme **probabiliste** effectue des choix aléatoires pendant son exécution.

Sa mise en œuvre implique l'utilisation d'un générateur de **nombres pseudo-aléatoires** fourni par le système ou par la bibliothèque standard.

Deux exécutions différentes d'un même programme ne donnent pas forcément les mêmes résultats, les mêmes temps d'exécution ou le même usage de la mémoire.

Par opposition, un algorithme ou un programme qui ne fait pas usage de choix aléatoires est qualifié de **déterministe**.

On peut utiliser un algorithme probabiliste pour **mélanger** des données ou réaliser un **échantillonnage**, c'est-à-dire sélectionner un sous-ensemble des données.

Dans le premier cas, on souhaite que toutes les permutations soient équiprobables.

Dans le second cas, on souhaite un tirage équitable entre tous les sous-ensembles possibles.

Par exemple : tirer aléatoirement un arbre binaire de taille n, avec équiprobabilité parmi tous les arbres binaires de taille n.

On peut utiliser un algorithme probabiliste pour **améliorer les performances d'un programme**.

Par exemple, en mélangeant un tableau avant de lui appliquer un tri rapide, le choix du pivot à chaque étape n'est plus imposé par la forme initiale du tableau. On évite ainsi des situations pathologiques, comme un tableau trié en ordre croissant ou décroissant. On montre alors que l'**espérance** du nombre de comparaisons est $O(n \log n)$ pour trier un tableau de taille n, ce qui est optimal.

Un algorithme probabiliste peut être utilisé pour **obtenir une solution approchée** à un problème difficile, dont une résolution exacte demanderait trop de temps.

Déterminer si un entier est premier est un problème difficile mais des algorithmes probabilistes permettent de vérifier efficacement qu'un entier est **probablement premier** avec une très faible probabilité d'erreur.

Deux familles d'algorithmes probabilistes sont étudiés.

- Les algorithmes de type **Las Vegas** donnent toujours un résultat correct. Le hasard influence ici le *temps de calcul*. Il est petit avec une forte probabilité. Un exemple de cette catégorie est le tri rapide.
- Les algorithmes de type **Monte Carlo** ne donnent pas forcément un résultat correct. Ici, c'est la *probabilité de la correction* qui nous intéresse. Un exemple de cette catégorie est le test de primalité.

Tri rapide

Algorithme standard

On considère un tableau dont les éléments peuvent être ordonnés de manière croissante. Dans la suite, un tableau d'entiers illustre notre propos. Par exemple :

$$[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9]$$

Son tri mène au tableau suivant.

$$[1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 8, 9, 9]$$

Le plus souvent, les algorithmes de tris réorganisent le tableau en procédant par *comparaisons*. On montre alors que la complexité moyenne optimale de tels algorithmes est en $O(n \log n)$ si n est le nombre d'éléments du tableau.

Algorithme standard

Le **tri rapide** est l'un de ces algorithmes dont le principe est rappelé ci-dessous.

- Soit $T = [x_0, x_1, \ldots, x_{n-1}]$ un tableau d'entiers.
- Choisir un entier p de T comme *pivot*.
- *Partitionner* T en deux sous-tableaux $T_<$ et $T_>$ tels que :

$$\forall x \in T_< \quad x < p \qquad\qquad \forall x \in T_> \quad x \geqslant p$$

- *Trier* récursivement $T_<$ et $T_>$.
- Renvoyer le *tableau trié* symboliquement noté $T_< \cdot [p] \cdot T_>$.

Algorithme standard

Une étape importante de cet algorithme est le **choix du pivot**.

A priori, si les données sont initialement « bien » mélangées, tout élément du tableau, et en particulier son premier élément, peut jouer le rôle du pivot.

En pratique, l'algorithme standard du tri rapide procède ainsi. Les diapositives suivantes proposent un code C pour ce tri.

Algorithme standard

```
#include <stdio.h>
#include <stdlib.h>

// échange de deux éléments
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
// partition d'un tableau
// entre les indices low et high
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[high]);
    return i;
}
```

```
// partition autour du pivot
// premier élément du sous-tableau
// commençant à l'indice low
int pPartition(int arr[], int low, int high) {
    int p = low;
    swap(&arr[p], &arr[high]);
    return partition(arr, low, high);
}
```

```
// tri rapide récursif
void qSort(int arr[], int low, int high) {
    if (low < high) {
        int p = pPartition(arr, low, high);
        qSort(arr, low, p - 1);
        qSort(arr, p + 1, high);
    }
}
```

L'appel à cette fonction sur le tableau exemple du début de diaporama renvoie le résultat suivant.

```
Tableau initial : 3 1 4 1 5 9 2 6 5 3 5 8 9
Tableau trié :    1 1 2 3 3 4 5 5 5 6 8 9 9
```

Complexité

Le choix du pivot a des conséquences sur l'efficacité du programme.

Supposons le tableau intialement trié. Il suffirait de « se rendre compte » de l'existence de cet ordre préalable pour éviter de faire trop d'opérations. Au pire, on ne devrait que parcourir le tableau et *constater* le tri, par exemple, en faisant une première boucle qui compare deux éléments successifs et s'arrête dès qu'ils ne sont pas dans l'ordre croissant.

Complexité

L'algorithme précédent ne procède pas ainsi. Une fois le pivot choisi, il partionne le tableau autour de son premier élément choisi comme pivot. Si le tableau est déjà trié, l'entier `p` renvoyé par la fonction `pPartition` est ce même entier, opération qui se fait avec un coût linéaire en la taille du sous-tableau argument de `partition`.

Le premier appel `qSort` qui suit ne s'effectue alors pas. Le second appel s'effectue sur un tableau de taille inférieure de 1 par rapport à celle du tableau initial.

Complexité

Si on note $C(n)$ le nombre de comparaisons effectuées lors de ces appels, pour tout entier n strictement positif, on a :

$$C(n) = C(n-1) + \Theta(n)$$

Cette relation mène à :

$$C(n) = \Theta(n^2)$$

Ainsi, l'**algorithme standard de tri rapide** trie un **tableau** déjà **trié** avec une **complexité quadratique**.

Pour rappel, le *tri par insertion*, qui appartient à la famille des tris quadratiques, le fait en temps linéaire en la taille du tableau !

Mélange initial

L'observation précédente montre qu'il convient de mieux choisir le pivot.

Deux alternatives sont envisageables.

- **Mélanger** préalablement les données pour que le choix du premier élément comme pivot ne soit plus celui du plus petit élément du tableau. C'est
- Choisir le **pivot** de manière **aléatoire** à chaque appel récursif.

Dans les deux cas, il convient de mesurer l'effet de ces opérations sur la complexité moyenne du programme.

Mélange initial

Pour mélanger préalablement les données, on peut adopter l'*algorithme de Knuth*. En reprenant la fonction `swap` définie plus haut, la fonction `shuffle` ci-dessous mélange les entiers d'un tableau argument `arr` de taille `size`.

```
// mélange de Knuth
void shuffle(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        swap(&arr[i], &arr[rand() % (i+1)]);
    }
}
```

Mélange initial

La fonction `rand` renvoie un entier compris entre `0` et `RAND_MAX`, constante définie dans `stdlib.h`. Par exemple, le code suivant tire au hasard 100 entiers entre 0 et 100 et les affiche dans la console.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 100;
    for (int i = 0; i < n; i++)
        printf("%d\t", rand() % (n+1));
    return 0;
}
```

Mélange initial

Si on relance plusieurs fois ce programme, les mêmes nombres pseudo-aléatoires sont renvoyés.

Pour éviter cette redondance, la fonction `srand` modifie la suite d'entiers générée en choisissant une *graine*, valeur initiale permettant le calcul de la suite d'entiers. Toutefois, là encore, une fois cette valeur choisie, la suite des entiers reste la même si on relance le programme.

Pour éviter ce dernier problème, on modifie la graine à chaque exécution du programme.

Mélange initial

On peut utiliser un entier lié à l'instant d'exécution du programme. La fonction `time` du fichier d'en-tête `time.h` renvoie le nombre de secondes écoulées depuis le 1er janvier 1970. Elle reçoit un entier ; on lui passe la valeur `NULL` (constante qui vaut `0` ici).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL));
    int n = 100;
    for (int i = 0; i < n; i++)
        printf("%d\t", rand() % (n+1));
    return 0;
}
```

Random Quick sort

La seconde alternative envisagée plus haut consiste à tirer au hasard un entier du tableau pour définir le pivot. Il suffit de remplacer de remplacer la fonction `pPartition` par la fonction suivante.

```
int rndPartition(int arr[], int low, int high)
    {
    srand(time(NULL));
    int r = low + rand() % (high - low);
    swap(&arr[r], &arr[high]);
    return partition(arr, low, high);
}
```

Random Quick sort

Cet algorithme est appelé **Random Quick sort** (RQS).

En procédant ainsi, on peut espérer partitionner le tableau t initial en deux sous-tableaux de tailles proches, moitiés de celle de t.

En pratique, même si cette procédure de garantit pas toujours le respect strict de cette propriété, il est *raisonnable* de croire que l'algorithme récursif sera assez souvent *chanceux*.

On prouve ce résultat en calculant l'**espérance du nombre de comparaisons**.

Complexité

Reprenons le tableau d'entiers T, de taille n, non trié initialement.

$$T = [x_0, x_1, \ldots, x_{n-1}]$$

Trier t équivaut à trouver une permutation σ telle que le tableau suivant soit ordonné.

$$Z = [x_{\sigma(0)}, x_{\sigma(1)}, \ldots, x_{\sigma(n-1)}]$$

Dans la suite, on note :

$$\forall i \in [\![0, n-1]\!] \quad z_i = x_{\sigma(i)}$$

avec, pour tout entier i compris entre 0 et $n-2$, $z_i \leqslant z_{i+1}$.

Complexité

Considérons le sous-tableau ordonné $Z_{ij} = [z_i, \ldots, z_j]$. On définit la **variable aléatoire** :

$$\forall (i,j) \in [\![0, n-1]\!]^2 \quad X_{ij} = \begin{cases} 1 & \text{si } z_i \text{ et } z_j \text{ sont comparés} \\ & \text{lors de l'exécution de RQS} \\ 0 & \text{sinon.} \end{cases}$$

Chaque paire d'éléments de Z est comparée *au plus* une fois lors de l'appel à RQS. Le **nombre de comparaisons** est donné par :

$$X = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} X_{ij}$$

Complexité

L'**espérance du nombre de comparaisons** est alors :

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} X_{ij}\right]$$

Par linéarité de l'espérance :

$$\mathbb{E}[X] = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} \mathbb{E}[X_{ij}]$$

C'est aussi la **complexité moyenne de RQS** recherchée.

Complexité

Désignons par p_{ij} la **probabilité** que z_i et z_j soient comparés lors d'une exécution de RQS. Alors :

$$\mathbb{E}[X_{ij}] = 1 \times p_{ij} + 0 \times (1 - p_{ij})$$

soit :

$$\mathbb{E}[X_{ij}] = p_{ij}$$

Pour déterminer p_{ij}, on doit répondre à la question : quand z_i et z_j sont-ils comparés ?

Complexité

Supposons qu'au cours de l'exécution de RQS, le pivot p soit dans Z_{ij}. Si $z_i < p < z_j$ alors z_i et z_j appartiennent à des partitions différentes et ne sont donc jamais comparés.

Par conséquent, z_i et z_j sont comparés si et seulement si le premier élément de Z_{ij} choisi comment pivot est soit z_i, soit z_j. Ainsi :

$p_{ij} = \mathbb{P}[z_i \text{ est le premier pivot choisi dans } Z_{ij}] + \mathbb{P}[z_j \text{ est le premier pivot c}$

soit :

$$p_{ij} = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

Complexité

On doit finalement calculer :

$$\mathbb{E}[X] = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1}$$

soit :

$$\mathbb{E}[X] = \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{2}{k+1}$$

Puis :

$$\mathbb{E}[X] < \sum_{i=0}^{n-2} \sum_{k=1}^{n-1} \frac{2}{k}$$

Complexité

Par le théorème de comparaison série-intégrale, il vient :

$$\mathbb{E}[X] < \sum_{i=0}^{n-2} \int_{1}^{n-1} \frac{2}{u} \, \mathrm{d}u$$

Puis :

$$\mathbb{E}[X] < 2(n-1)\log(n-1)$$

soit finalement :

$$\mathbb{E}[X] = O(n \log n)$$

La complexité moyenne de l'algorithme RQS est au pire $O(n \log n)$.

Las Vegas

L'algorithme RQS est de type **Las Vegas**.

- L'*algorithme* renvoie *toujours* un tableau trié ; il est *correct*.
- Le *temps d'exécution* de l'algorithme est directement lié aux choix de tirages aléatoires.

Coupe minimum

Coupe d'un graphe

Considérons un graphe $G = (V, E)$.

- Une **coupe** de G est une partition de V en deux ensembles S et $V \setminus S$. On peut la noter :

$$C = (S, V \setminus S)$$

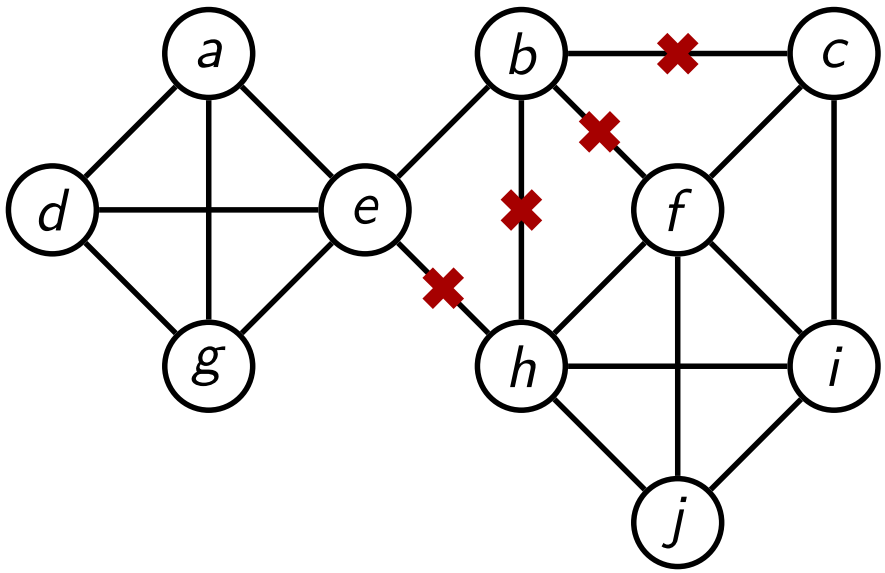
- Elle définit un ensemble d'arêtes ayant une extrémité dans S et dans $V \setminus S$.

Si G est non orienté et non pondéré, le **poids de la coupe** est le nombre d'arêtes dans la coupe. Si les arêtes sont pondérées, le poids de la coupe est égal à la somme des poids des arêtes dans la coupe.

Coupe d'un graphe

$$V = \{a, b, c, d, e, f, g, h, i, j\}$$

$$S = \{a, b, d, e, g\} \qquad V \setminus S = \{c, f, h, i, j\}$$

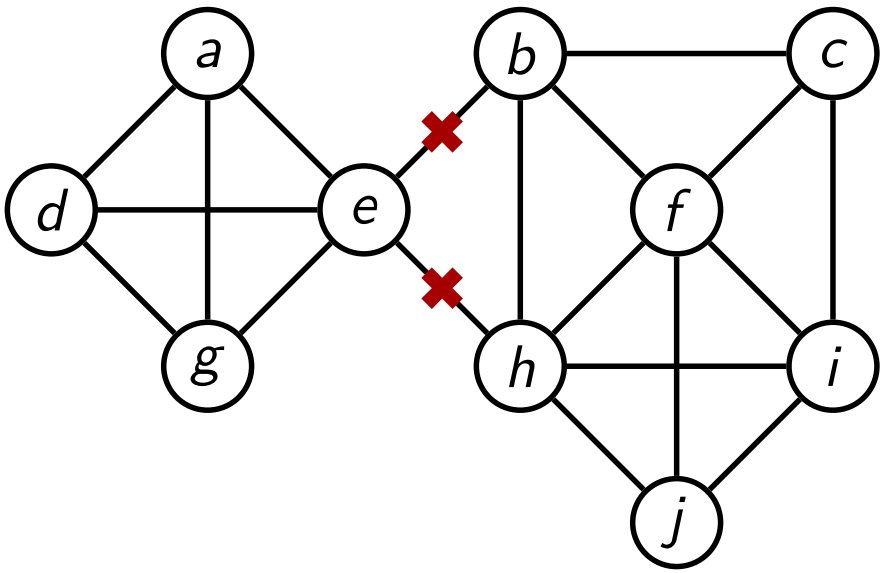


$C = (S, V \setminus S)$ est de poids 4.

Coupe d'un graphe

$$V = \{a, b, c, d, e, f, g, h, i, j\}$$

$$S = \{a, d, e, g\} \qquad V \setminus S = \{b, c, f, h, i, j\}$$



$C = (S, V \setminus S)$ est de poids 2.

Coupe minimum

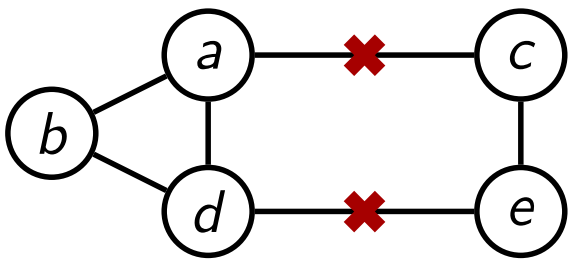
Une **coupe minimum** (*min-cut* en anglais) d'un graphe est une coupe contenant un nombre minimal d'arêtes.

Un graphe peut admettre plusieurs coupes minimums.

La figure suivange présente une coupe minimum dans un *graphe simple non orienté et non pondéré*.

$$V = \{a, b, c, d, e\}$$

$$S = \{a, b, d, e, g\} \qquad V \setminus S = \{c, f, h, i, j\}$$



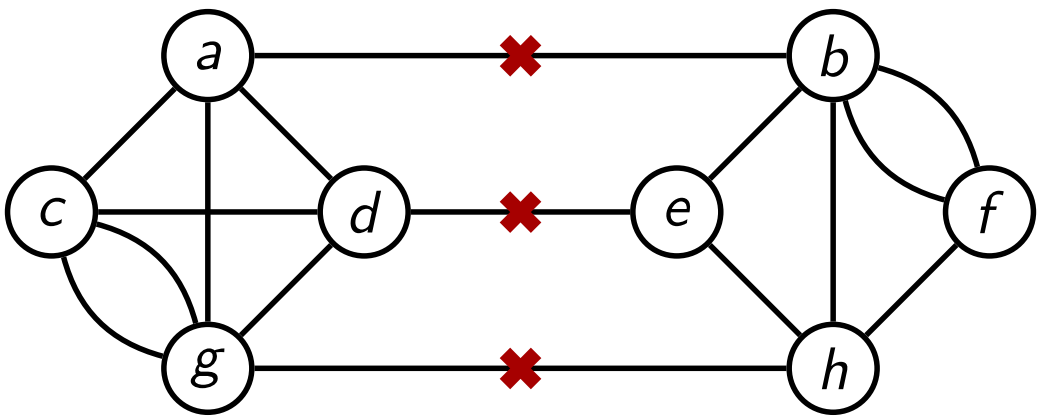
$C = (S, V \setminus S)$ est une coupe minimum de poids 2.

Coupe minimum

La figure suivante présente une coupe minimum dans un *multigraphe non orienté et non pondéré*.

$$V = \{a, b, c, d, e, f, g\}$$

$$S = \{a, c, d, g\} \qquad V \setminus S = \{b, e, f, h\}$$



$C = (S, V \setminus S)$ est une coupe minimum de poids 3.

Coupe minimum

Le problème algorithmique de la *coupe minimum* peut être résolu en temps polynomial.

Les algorithmes déterministes qui le résolvent reposent l'étude des *réseaux de flot*.

Pour des graphes d'ordre n, leurs complexités algorithmiques sont $O(n^3)$.

L'*algorithme de Karger*, **algorithme probabiliste**, produit une solution *correcte avec une certaine probabilité*.

Contraction d'une arête simple

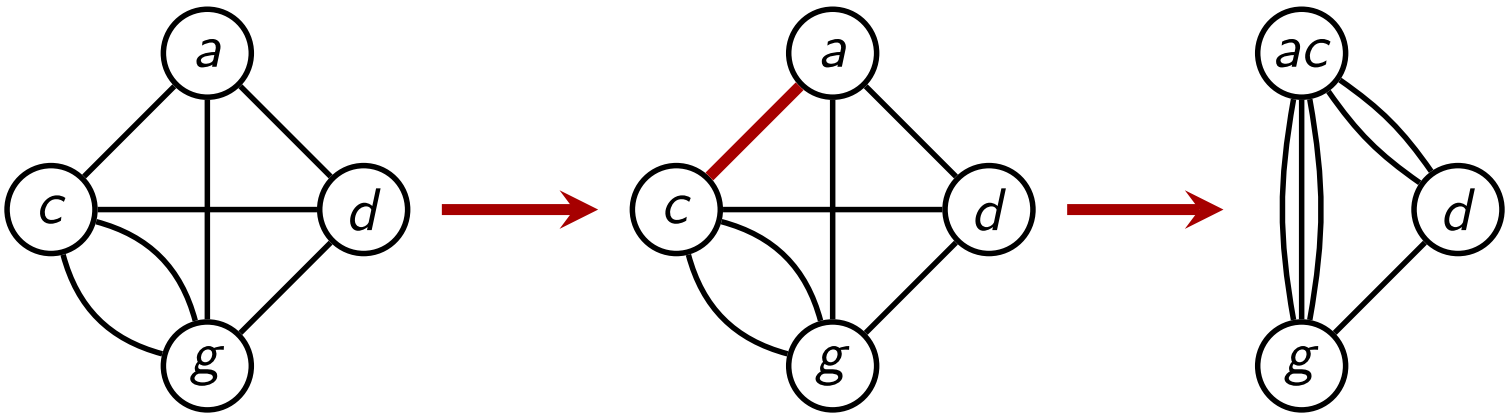
L'**algorithme de Karger** repose sur une opération essentielle appelée **contraction d'arête**.

Si u et v sont deux sommets adjacents d'un multigraphe simple non orienté et non pondéré G, contracter la ou les arêtes $\{u, v\}$ consiste à :

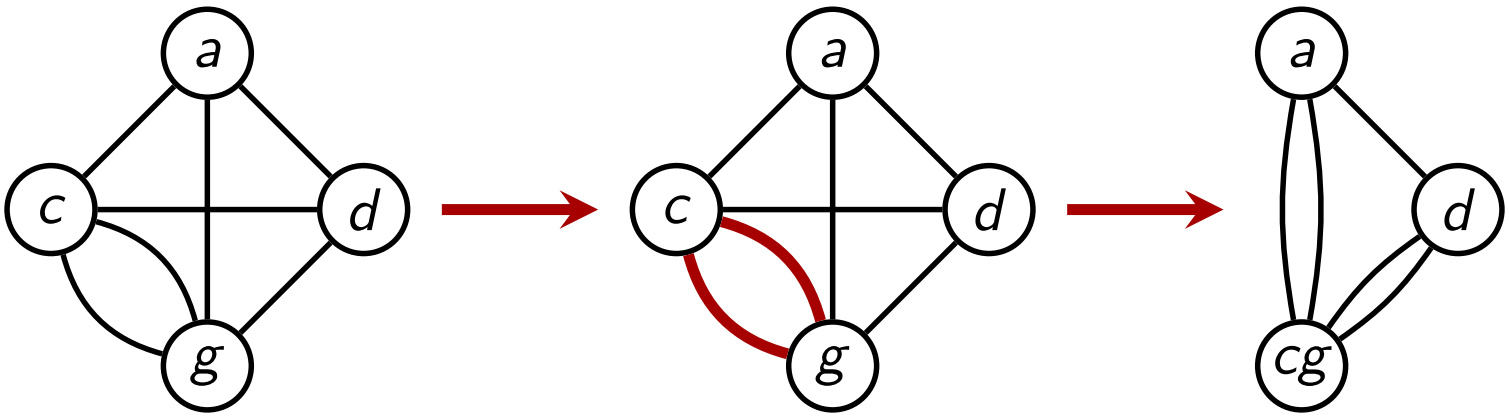
- fusionner les sommets u et v en un sommet unique, éventuellement noté uv ;
- remplacer chaque arête de la forme $\{u, w\}$ et $\{v, w\}$ par une arête $\{uv, w\}$;
- supprimer les sommets u et v.

Le graphe obtenu est parfois noté $G/\{u, v\}$.

Contraction d'une arête simple



Contraction d'une arête double



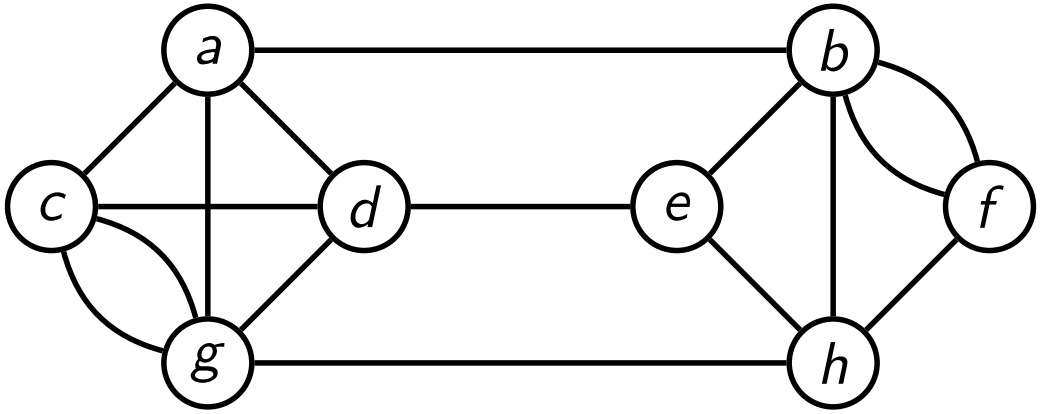
Algorithme de Karger

Cet algorithme repose sur l'observation qu'une coupe du graphe *après* contraction était déjà une coupe du graphe *avant* contraction. Soit G un multigraphe simple non orienté et non pondéré.

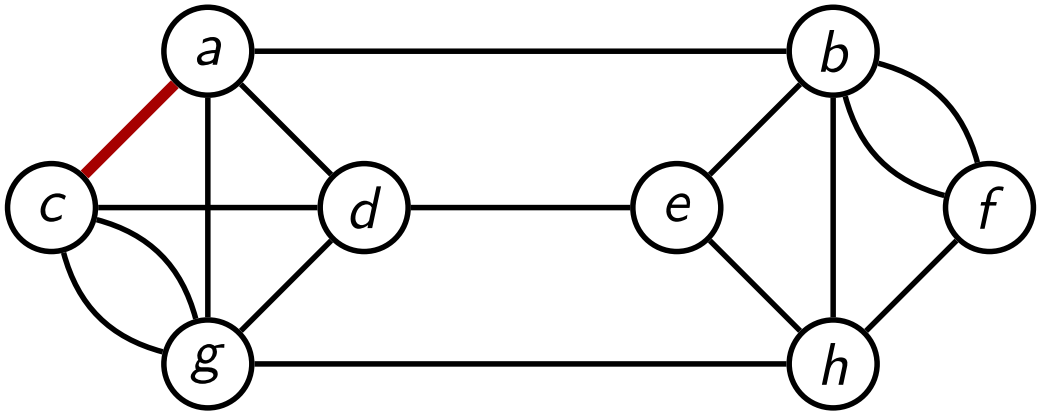
- Tant que $|G| > 2$, choisir *au hasard* deux sommets adjacents et contracter la ou leurs arêtes communes.
- Si $|G| = 2$, l'algorithme se termine. Les arêtes présentes entre les deux sommets forment une coupe de G.

Si initialement $|G| = n$, chaque contraction étant de complexité $O(n)$, l'algorithme est de complexité au pire $O(n^2)$.

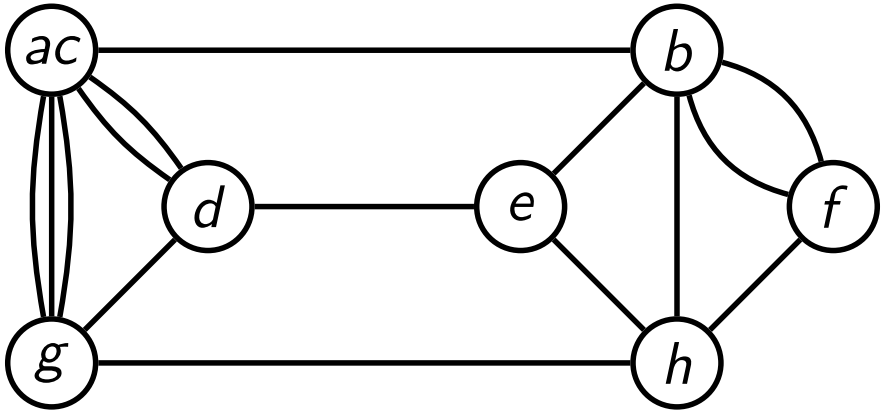
Algorithme de Karger



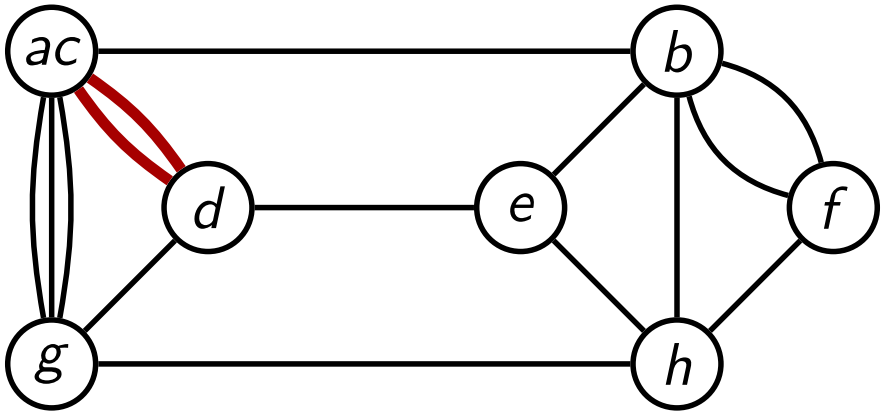
Algorithme de Karger



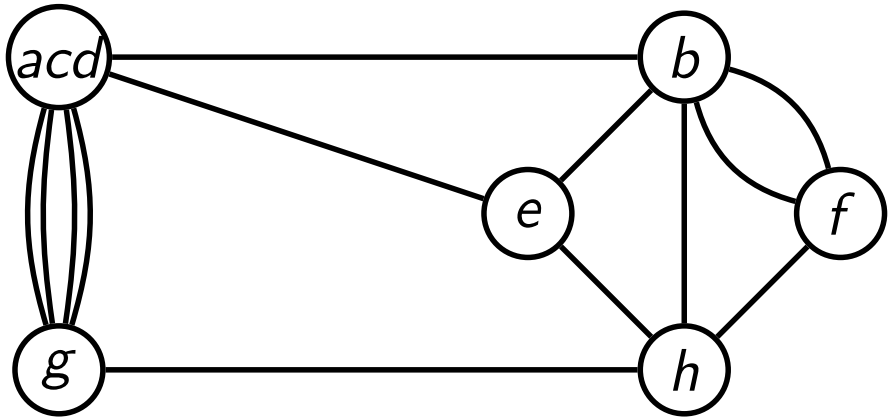
Algorithme de Karger



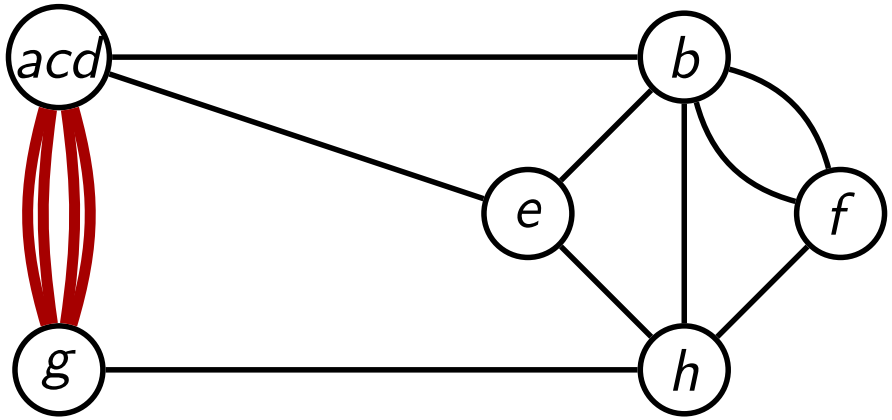
Algorithme de Karger



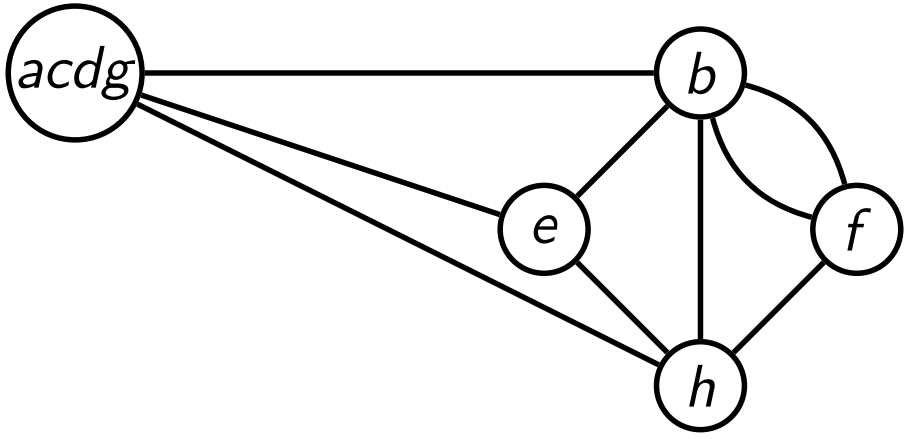
Algorithme de Karger



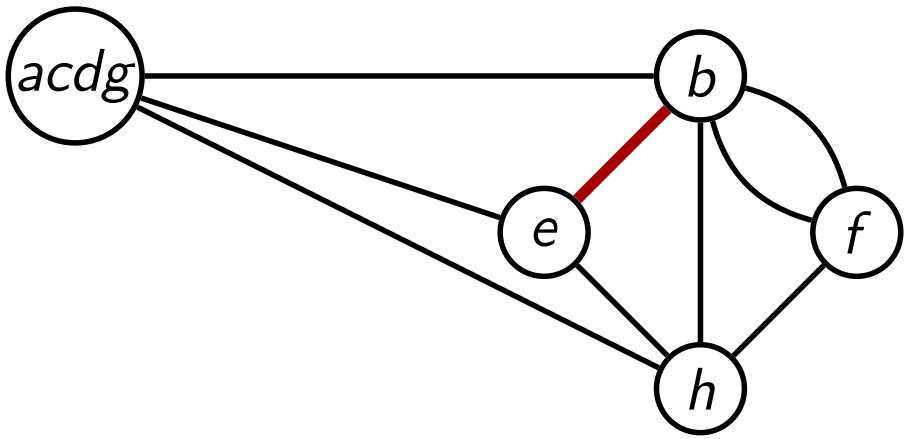
Algorithme de Karger



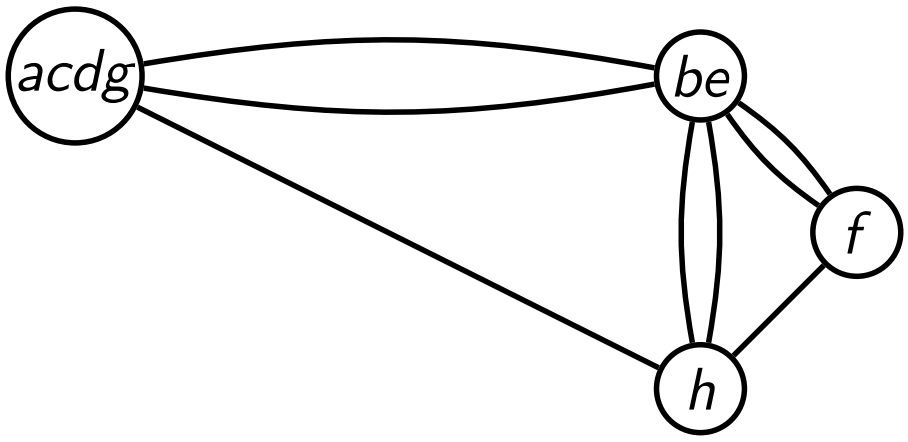
Algorithme de Karger



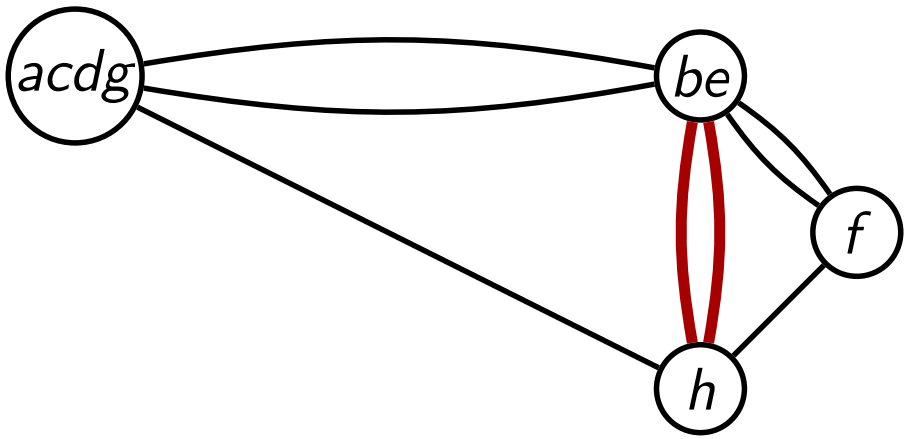
Algorithme de Karger



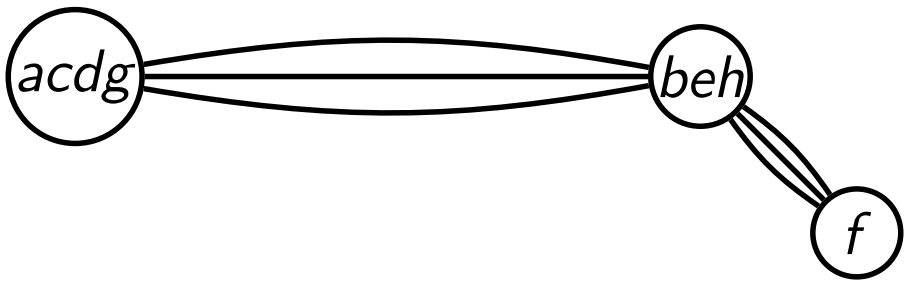
Algorithme de Karger



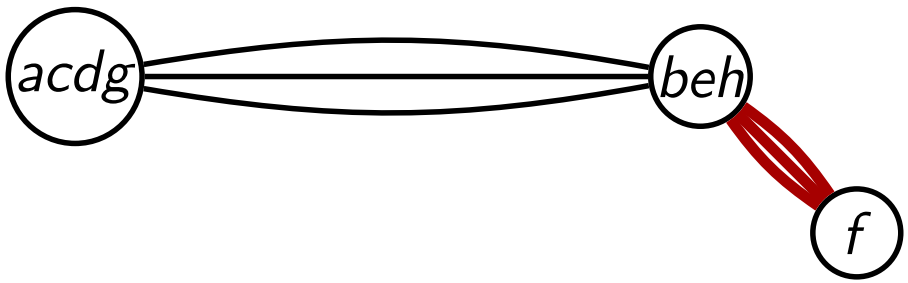
Algorithme de Karger



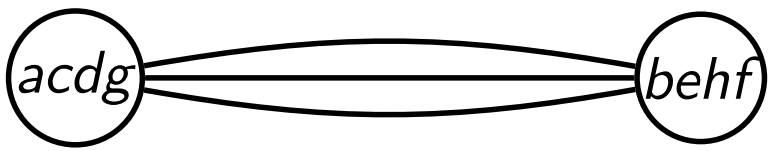
Algorithme de Karger



Algorithme de Karger



Algorithme de Karger



Probabilité d'obtenir coupe minimum

Quel est le lien entre cet algorithme et le problème *min-cut* ?

L'idée clé est que la probabilité de contracter des arêtes qui n'appartiennent pas à la coupe minimum est plus élevée que celle d'en choisir une car les arêtes présentes dans une coupe minimum sont bien moins nombreuses que celles présentes dans une coupe non minimum.

L'algorithme « parie » donc sur la préservation des arêtes d'une coupe minimum chaque fois qu'une contraction aléatoire est effectuée.

Probabilité d'obtenir coupe minimum

Pour simplifier, supposons que G ait exactement une coupe minimum de taille k.

Alors chaque sommet appartient à au moins k arêtes. Si ce n'était pas le cas, un sommet pourrait être séparé de G et, de fait, ainsi définir une coupe petite que k !

En conséquence, le nombre minimal de paires sommet-arête est kn. Et comme chaque arête est incidente à exactement deux sommets, G a au moins $kn/2$ arêtes.

Si on tire uniformément au hasard une arête de G, la probabilité qu'elle appartiennent à une coupe minimum est au plus $2/n$. Dès lors, la **probabilité de ne pas tirer une arête de la coupe minimum** lors pour la première contraction est $1 - 2/n$.

Probabilité d'obtenir coupe minimum

En poursuivant cette analye avec le graphe de taille $n-1$ obtenu après la première contraction, sachant que chaque choix de contraction est indépendant, la probabilité $\mathbb{P}[n]$ que l'algorithme renvoie une coupe minimum vérifie :

$$\mathbb{P}[n] \geqslant \left(1 - \frac{2}{n}\right) \mathbb{P}[n-1]$$

Sachant $\mathbb{P}[2] = 1$, il vient :

$$\mathbb{P}[n] \geqslant \frac{2}{n(n-1)}$$

Si $n = 10^{\alpha}$, cette probabilité est donc minorée par $2 \times 10^{-2\alpha}$.

Amélioration

Ce résultat peut être amélioré en répétant l'algorithme N fois. La probabilité de succès est alors au moins :

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^N \leqslant 1 - e^{-2N/n(n-1)}$$

On peut ainsi choisir N pour rendre cette probabilité aussi proche de 1 que possible.

Remarque

Se rappeler que $1 + x \leqslant e^x$.

Amélioration

Un choix possible est $N = cn(n-1)\ln n/2$, $c > 0$, de sorte que les N appels indépendants de l'algorithme renvoient un résultat correct avec une probabilité au moins égale à :

$$1 - \frac{1}{n^c}$$

ce qui constitue une *forte probabilité* de succès.

Le contrepartie est toutefois la complexité de la procédure qui est désormais $O(Nn^2) = O(n^3 \log n)$.

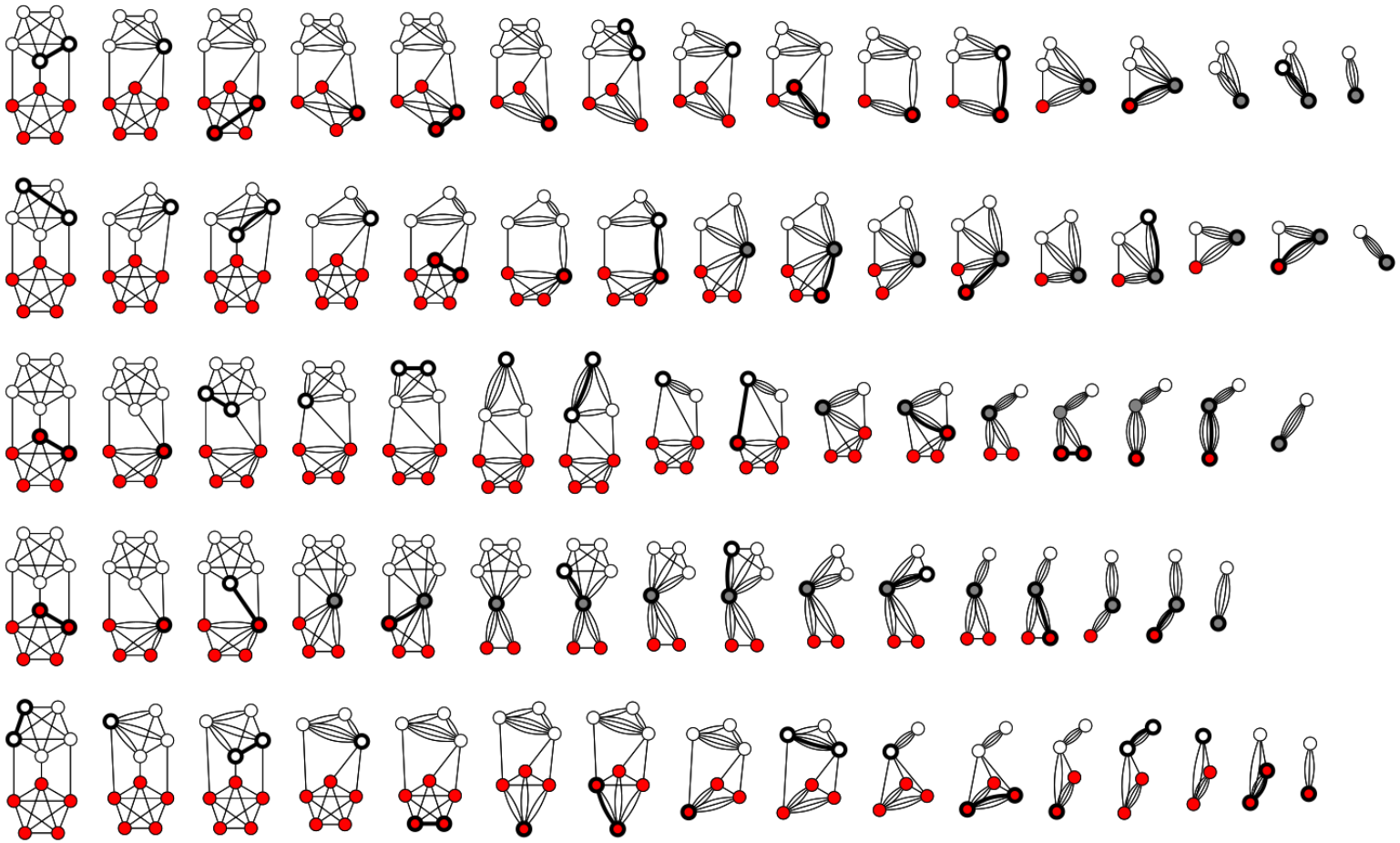
Des solutions existent pour améliorer ces résultats mais nous n'en dirons pas davantage sur ce sujet.

Monte Carlo

L'*algorithme de Karger* est de type **Monte Carlo**.

- L'algorithme renvoie une coupe qui *peut* être minimum, le résultat étant directement lié aux choix des tirages aléatoires.
- Le *temps d'exécution* de l'algorithme est toujours *borné*.

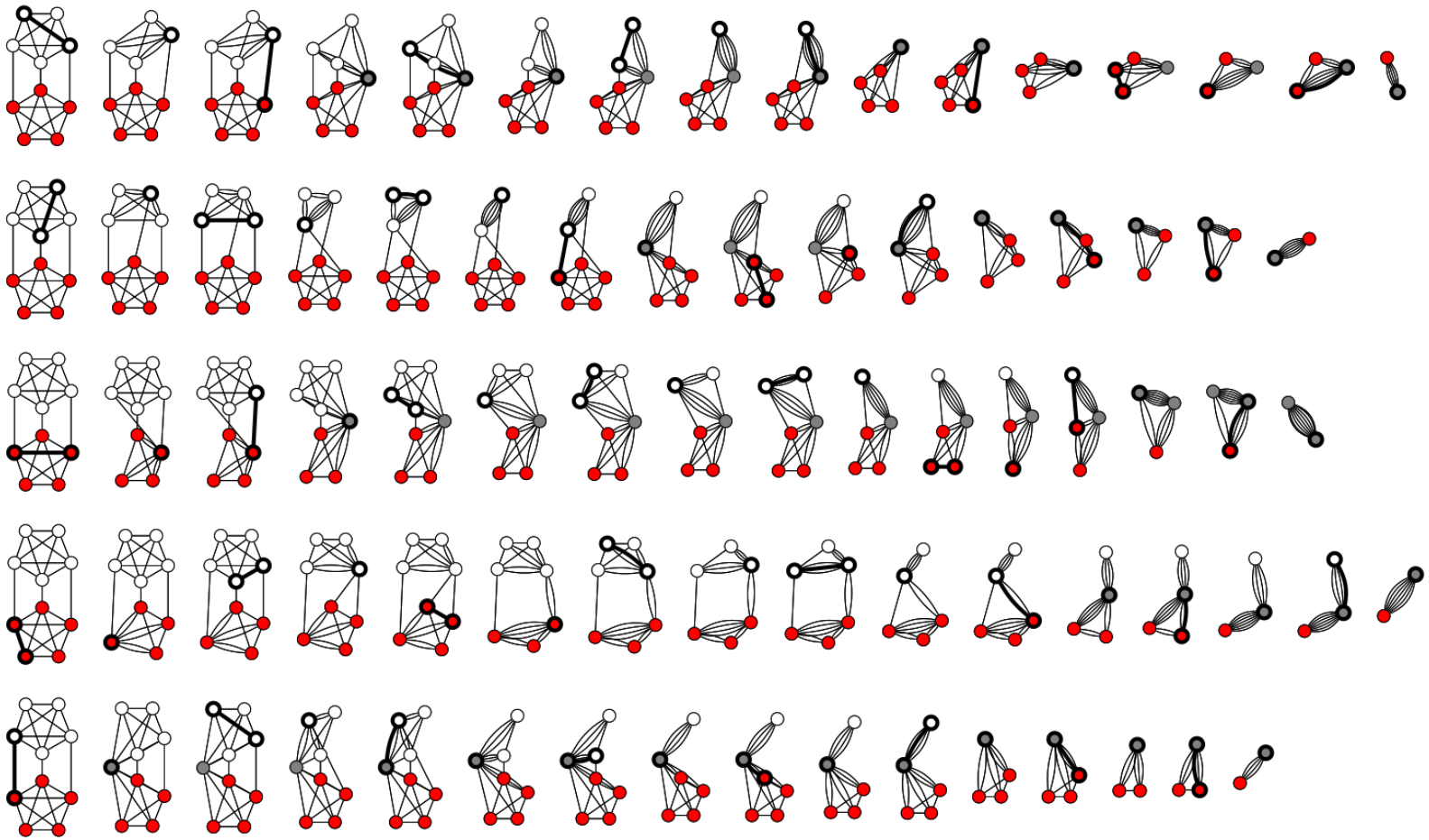
Cinq répétitions de l'algorithme de Karger



La cinquième répétition trouve la coupe minimum 3.

Source : `https://en.wikipedia.org/wiki/Karger's_algorithm`

Cinq autres répétitions de l'algorithme de Karger

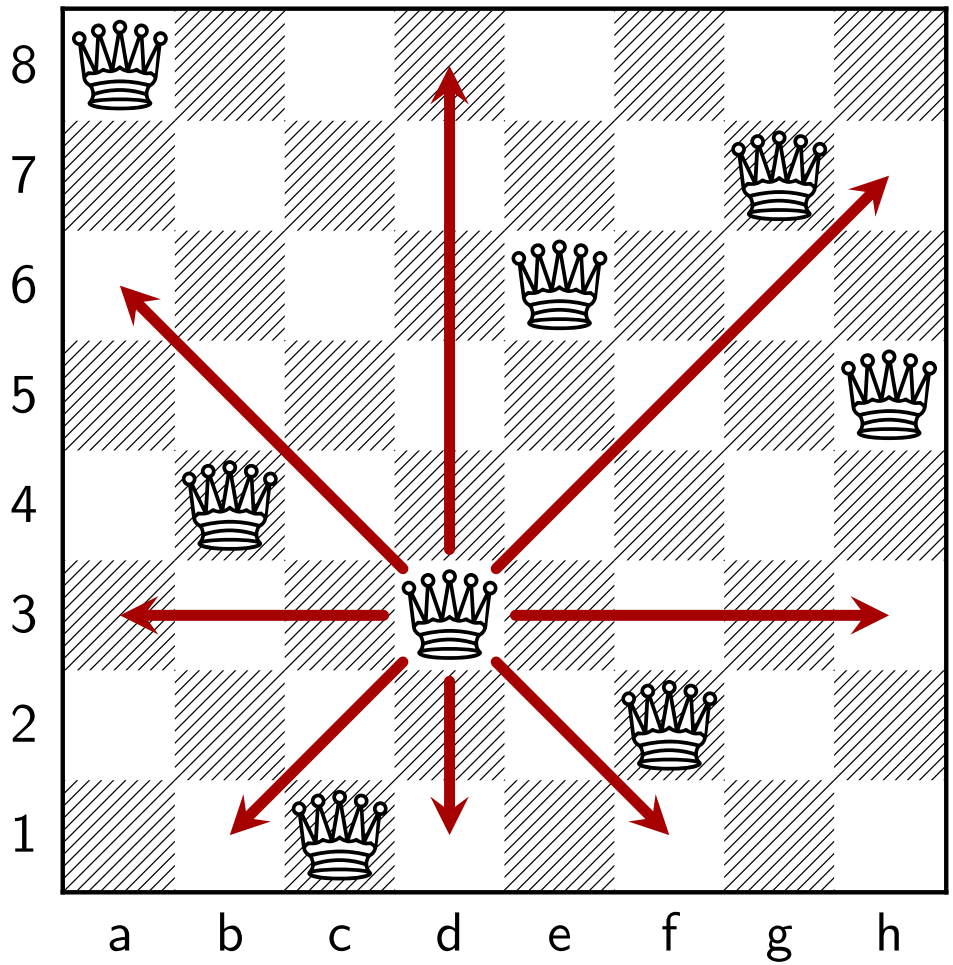


Source : `https://en.wikipedia.org/wiki/Karger's_algorithm`

Problème des reines

Position du problème

Le problème des N reines consiste à placer N reines sur un échiquier $N \times N$ sans qu'elles soient en prise deux à deux.



Retour sur trace

La technique du **retour sur trace** résout efficacement ce problème pour de petites valeurs de N, par exemple $N \leqslant 30$.

Si on cherche à résoudre le problème pour une valeur relativement grande de N, par exemple $N = 100$, l'algorithme de retour sur trace va tourner *très longtemps* avant de trouver une solution.

Approche probabiliste

On place les reines successivement sur chaque ligne, en vérifiant à chaque fois que la reine placée sur la ligne k n'est pas en conflit avec les reines placées sur les lignes précédentes.

Mais à la différence du retour sur trace, on n'explore pas systématiquement toutes les possibilités pour placer une reine sur la ligne k. On en choisit une seule, **au hasard**.

Il est tout à fait possible que cela mène à une impasse, c'est-à-dire à une ligne sur laquelle il n'est plus possible de placer une seule reine. Dans ce cas, on recommence **depuis le début**.

Programme

La fonction `check` vérifie si le choix pour la ligne `k` est compatible avec les lignes précédentes. S'il n'y a aucun choix possible, il renvoie `false`. Sinon, il continue avec la ligne suivante.

```
bool check(int n, int sol[], int k) {
  for (int i = 0; i < k; i++)
    if (sol[i] == sol[k] ||
        abs(sol[i]-sol[k]) == abs(i-k))
      return false;
  return true;
}
```

Programme

La fonction `solve_prob` reçoit en paramètre une solution partielle pour les `k` premières lignes, dans `sol[0..k[`. Elle choisit au hasard une colonne `c` pour la reine de la ligne `k`.

```
bool solve_prob(int n, int sol[], int k) {
  if (k == n) return true;
  int c = 0; // colonne candidate
  int t = 0; // nombre de candidats
  for (int v = 0; v < n; v++) {
    sol[k] = v;
    if (check(n, sol, k)) {
      t++;
      if (rand () % t == 0) c = v;
    }
  }
  if (t == 0) return false;
  sol[k] = c;
  return solve_prob(n, sol, k+1);
}
```

Elle renvoie `true` si le tableau `sol` a pu être complété avec une solution, `false` si on a échoué. Mais cela ne veut pas dire que cela n'était pas possible. C'est là la différence avec le retour sur trace !

Programme

La fonction principale `queens_prob` boucle tant qu'elle n'a pas trouvé de solution. Il s'agit donc là d'un **algorithme de type Las Vegas**.

```
void queens_prob(int n, int sol[]) {
  while (true) {
    if (solve_prob(n, sol, 0))
      return;
  }
}
```

Noter que pour $N = 2$ et $N = 3$, où il n'y a pas de solution au problème, la fonction `queens_prob` ne va jamais terminer. Mais comme on l'a dit plus haut, cette version probabiliste se justifie plutôt pour de grandes valeurs de N.

Programme

Cette version probabiliste des N reines est étonnamment efficace. Pour $N = 100$, elle trouve une solution en un quart de seconde, après seulement 344 descentes.

Il est difficile d'expliquer précisément ce résultat, la structure du problème des N reines étant encore mal connue. Mais il est frappant de voir à quel point cette approche probabiliste est efficace.

Test de primalité

Généralités

Dans cette section, on s'intéresse au *test de primalité de Fermat*.

Il s'agit d'un **algorithme probabiliste de type Monte Carlo**.

S'il affirme qu'un nombre est composé, alors il l'est effectivement. Mais s'il affirme qu'un nombre est premier, alors il se peut que ce ne soit pas le cas. Cependant, la probabilité d'un faux positif est inférieure à $1/2$. Dès lors, il suffit de répéter le test k fois pour diminuer la probabilité d'erreur à moins de $1/2^k$.

Principe

Le test de primalité de Fermat repose sur le petit théorème de Fermat qui indique que si un entier n est premier alors, pour tout entier naturel $a \in [\![1, n-1]\!]$, on a :

$$a^{n-1} \equiv 1 \bmod n$$

L'algorithme consiste à choisir un entier a au hasard dans l'intervalle $[2, n-1]$ puis à calculer $a^{n-1} \bmod n$. Si le résultat n'est pas égal à 1, on est certain que le nombre n est composé. Dans le cas contraire, le nombre est possiblement premier. On peut répéter le test un certain nombre de fois.

Programme

La fonction `pow_mod` ci-dessous réalise une exponentiation rapide x^n modulo m dans les entiers 64 bits de C.

```
uint64_t pow_mod(uint64_t x, uint64_t n, uint64_t m) {
  uint64_t y = 1;
  while (n > 0) {
    if (n % 2 == 1) {
      y = (y * x) % m;
    }
    x = (x * x) % m;
    n = n / 2;
  }
  return y;
}
```

Programme

Le programme suivant met en œuvre le test de primalité de Fermat. Les premières lignes évacuent des cas triviaux. Puis une boucle `for` répète le test `k` fois, pour une valeur de `k` fixée. Si tous les tests passent avec succès, on renvoie `true` en présupposant que `n` est premier.

```
bool fermat(uint64_t n, int k) {
  if (n <= 1) {return false;}
  if (n == 2 || n == 3) {return true;}
  if (n % 2 == 0) {return false;}
  for (int i = 1; i <= k ; i++) {
    // a dans [2, n - 1]
    int a = 2 + (rand() % (n - 2));
    if (pow_mod(a, n - 1, n) != 1) {
      return false; // n est composé
    }
  }
  return true; // n est probablement premier
}
```

Il est bien sûr possible que n soit composé mais passe le test pour certaines valeurs de a, c'est-à-dire que $a^n \equiv 1 \bmod n$ alors que n est composé.

Par exemple, $341 = 11 \times 31$ n'est pas un nombre premier mais $2^{340} \equiv 1 \bmod 341$.

Mais on peut *espérer* que pour *la plupart* des valeurs de a le test pour un nombre composé va rater. Et sauf cas particulier, c'est le cas pour environ la moitié des choix de a. C'est ce qui motive à choisir a de manière aléatoire.

Il existe cependant des nombres composés particuliers, appelés *nombres de Carmichael*, qui passent le test pour tous les choix de a ! En conséquence, l'algorithme présenté plus haut se trompe systématiquement pour ces nombres (avec probabilité 1).

Mais ces nombres deviennent rapidement extrêmement rares. Et il existe des généralisations de la méthode présentée ici, comme le *test de Miller–Rabin*, qui savent gérer correctement ces cas pathologiques.

Pour simplifier notre analyse, on va désormais supposer que ces nombres n'existent pas et se placer dans un monde imaginaire sans nombres de Carmichael.

Analyse probabiliste

Montrons que la probabilité qu'un nombre composé n passe le test de Fermat avec succès est inférieure à $1/2$.

Soit un nombre composé $n > 3$ et le sous-groupe multiplicatif de $(\mathbb{Z}/n\mathbb{Z})$ formé de ses éléments inversibles, c'est-à-dire des classes d'équivalences des entiers premiers avec n.

$$(\mathbb{Z}/n\mathbb{Z})^\times = \{a \in [\![0, n-1]\!] \mid a \equiv 1 \mod n\}$$

Puisque n est composé, on a $|(\mathbb{Z}/n\mathbb{Z})^\times| < n - 1$.

Analyse probabiliste

Considérons maintenant l'ensemble $A = \{a \in (\mathbb{Z}/n\mathbb{Z})^\times \mid a^{n-1} \equiv 1 \mod n\}$ le sous-ensemble de $(\mathbb{Z}/n\mathbb{Z})^\times$ des éléments qui passent indûment le test.

A est un sous-groupe de $(\mathbb{Z}/n\mathbb{Z})^\times$. Par le théorème de Lagrange, on déduit que son ordre $|A|$ divise celui de $(\mathbb{Z}/n\mathbb{Z})^\times$. Ainsi, soit $|A| = |(\mathbb{Z}/n\mathbb{Z})^\times|$, soit $|A| \leqslant |(\mathbb{Z}/n\mathbb{Z})^\times|/2$.

Comme nous avons mis de côté les nombres de Carmichael, si n est composé, il existe au moins un nombre $a \in [\![2, n-1]\!]$ qui ne passe pas le test et ainsi $|A| < (n-1)/2$. Dit autrement, moins de la moitié des entiers a dans $[2, n-1]$ passe le test avec succès.

Comme on répète k fois le test, on en déduit que la probabilité que la fonction `fermat` déclare incorrectement un nombre composé comme étant premier est inférieure à $1/2^k$.

Remarque

Il est important de comprendre que les entiers 64 bits ne nous permettent pas d'aller très loin. En effet, il faut pouvoir calculer un produit sans débordement arithmétique, ce qui nous limite de fait à $n < 2^{32}$. Or, tester la primalité d'un entier 32 bits de façon exacte se fait facilement.

Pour que le test devienne intéressant, il faut de grands entiers, comme par exemple ceux de la bibliothèque GMP (*GNU Multiple Precision* `https://gmplib.org/`). Cette bibliothèque logicielle de calcul multiprécision sur le nombres entiers, rationnels et en virgule flottante offre un test de primalité probabiliste, qui utilise en particulier l'algorithme de Miller–Rabin.