Structures de données et complexité

Informatique scientifique pour le Calcul
Ecoles Doctorales 2014/2015

Vincent Miele

CNRS - Biométrie & Biologie Evolutive

Décembre 2014

Document disponible sur `http://lyoncalcul.univ-lyon1.fr`

Passer du papier à l'ordinateur

- quel(s) langage(s) ? [cours suivant]
- quelles algorithmes ? adaptés au calcul parallèle ? [cours suivant] comment quantifier les ressources qu'ils demandent ? [ce cours]
- quelles structures de données pour mon problème (indépendant du langage) ? la transcription de celles que j'utilise sur la papier ? [ce cours]

Passer du papier à l'ordinateur

Objectifs

- efficacité → bien comprendre les coûts de calcul, théoriques [ce cours] et coût machine [cours précédent Violaine]
- robustesse → utiliser au maximum les standards, les librairies, s'appuyer sur la communauté d'utilisateurs [cours suivant]
- maximum de cohérence avec les algorithmes du papier → lisibilité, encapsulation [ce/autre cours]

Definition

La *complexité* d'un problème caractérise les ressources nécessaires pour les résoudre. Les ressources essentielles sont le temps (d'éxecution) et l'espace (mémoire).

Jusqu'aux années 70, seule la mesure expérimentale de la complexité d'un algorithme était (parfois) effectuée et dépendait des machines. . . une mesure théorique devint souhaitable !

Definition

La *complexité algorithmique* est un ordre de grandeur théorique du temps de calcul (*complexité en temps*) et/ou de l'espace mémoire (*complexité en mémoire*) utilisé en fonction d'une mesure des données.

Souvent, si un algorithme permet de gagner du temps de calcul, il occupe davantage de place en mémoire. . . vigilance. . .

La complexité n'est pas la même selon les cas

- complexité au pire : complexité maximum, dans le cas le plus défavorable
- complexité en moyenne : il s'agit de la moyenne des complexités obtenues selon les cas
- complexité au mieux : complexité minimum, dans le cas le plus favorable. En pratique, cette complexité n'est pas très utile

Exemple (complexité en temps) : chercher un élement dans une liste de taille n
pire : n opérations
moyenne : (n+1)/2 opérations
au mieux : 1 opération

Si on veut comparer des algorithmes sans considérer l'implémentation, on peut comparer les complexités au regard de la taille attendue des données

- complexité au pire
- forme générale de la complexité, qui indique la façon dont elle évolue en fonction d'un paramètre (ou plusieurs).
  On utilisera la notation $O(\ldots)$ qui veut dire *de l'ordre de...*

Definition

Notation de Landau : g est dite en $O(f)$ s'il existe des constantes $c > 0$ et $x_0 >> 0$ tq $g(x) < c * f(x)$ pour tout $x > x_0$. On note $g = O(f)$ et on dit que g est dominée asymptotiquement par f.

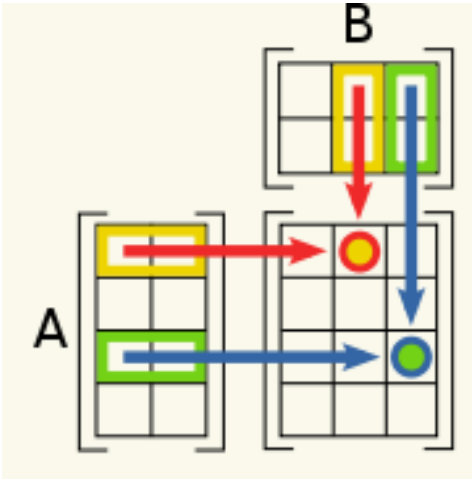
Si $g = O(f)$ et $f = O(h)$ alors $g = O(h)$
si $g = O(f)$ et k un nombre, alors $k * g = O(f)$
si $f1 = O(g1)$ et $f2 = O(g2)$ alors $f1 + f2 = O(g2 + g2)$
si $f1 = O(g1)$ et $f2 = O(g2)$ alors $f1 * f2 = O(g2 * g2)$

Si n est la taille des données, complexité en temps :
$O(1)$ *constante* : extraction d'un élément dans un tableau
$O(log(n))$ *logarithmique* : recherche dichotomique dans une liste triée (voir après)
$O(n)$ *linéaire* : parcours d'une liste
$O(nlog(n))$ *quasi-linéaire* : tri d'un tableau
$O(n^2)$ *quadratique* : calcul du maximum d'une matrice carré
$O(n^3)$ *cubique* : multiplication matricielle (voir après)
$O(2^n)$ *exponantielle* : problème du sac à dos par force brute (énumération)...

NB : la notation de Landau peut utiliser plusieurs paramètres décrivant la taille du problème (exple : $O(m * n + q)$)

Produit matrice x matrice

Pour la produit de deux matrice de taille mxn et nxp, on réalise n produits $+n$ sommes pour chaque élement de la matrice finale de taille mxp.
On a donc une complexité $O(m * p)$ en espace et $O(n * m * p)$en temps



Pour le cas où $m = p = n$, $O(n^2)$ en espace et $O(n^3)$en temps.

"L'algorithme de Coppersmith-Winograd [...] est en $O(n^{2,376})$ [...] Mais aucune implémentation de l'algorithme n'est utilisée car la constante dans le grand O est prohibitive." (Wikipedia)

Recherche dichotomique dans un tableau trié

On coupe le tableau en deux et on cherche l'élément dans une des deux parties en répétant le même traitement récursivement.

```
fonction rechercheElement(tab, x)
   i <- 0; j <- tab.longueur-1;
   tantque (i <= j) faire
      si (tab[(j+i)/2] = x) alors retourne VRAI;
      sinon
         si (tab[(j+i)/2] > x) alors j <- (j+i)/2 - 1;
         sinon i <- (j+i)/2 + 1;
```

Au pire, la longueur de la partie du segment de tableau $[i, j]$ à traiter est d'abord n, puis $n/2$, puis $n/4$, ..., jusqu'à ce que $n/2^t = 1$.
Le nombre de tours de boucles est $t = log2(n)$. La complexité en temps est donc $O(log(n))$. $O(n)$ en espace.

Le nombre de processeurs disponibles est une données à prendre en compte dans la complexité. Deux alternatives pour exprimer la complexité :

- $O(f(n))$ avec $g(n)$ processeurs : exprime la complexité obtenue avec un nombre idéal de processeurs
- $O(f(n,p))$ avec p le nombre de processeurs : plus pragmatique
  Permet d'intégrer la loi d'Amdhal : $O_{seq}(f(n))) + O_{par}(g(n)/p)$
  Si $O_{seq}(f(n))$ est dominé, alors on garde $O_{par}(g(n)/p)$.

NB : attention à $O_{comm}(h(n))$!

La complexité nous aide à choisir le bon algorithme et les bonnes structures de données. C'est un indicateur.

Mais attention aux constantes devant le $O(\ldots)$!

Et attention aux coûts réels engendrés par les accès mémoire, la bande passante, les architectures des processeurs.

Un bon algorithme en machine est d'abord celui qui privilégie la localité spatiale et temporelle !

Deux algorithmes de même complexité, mais l'un respecte la localité spatiale et temporelle, l'autre non.

```
1  #include <vector>
   using namespace std;
3  int main(int argc, char ** argv )
   {
5    int s = 18000;
     vector<vector<double> > v(s,vector<double>(s, 0.));
7
     // real 0m8.970s
9    for (int i=0; i<s; i++)
       for (int j=0; j<s; j++)
11       v[i][j] = v[i][j] + v[i][j];

13   // real 0m45.077s
     for (int i=0; i<s; i++)
15     for (int j=0; j<s; j++)
         v[j][i] = v[j][i] + v[j][i];
17 }
```

Prélude : dans ce qui suit, on s'appuit sur les *containeurs* de la STL C++, mais des équivalents existent dans les autres langages.

Sur la papier, on a des structures de données de type :

- vecteur
- matrice pleine
- matrice creuse (ou graphe) avec m élements tq $m \sim O(n)$ ou $m << n^2$
- des listes (triés), des piles
- des couples "identifiants-observations"
- ...

Doit-on chercher à utiliser formellement les même structures sur le papier et en machine ? ou seulement moralement ?
On veut des performances en temps ET de la parcimonie en espace.

La clé de voûte de l'efficacité, car cohérence avec la structure en lignes de caches (cf. cours précédent)

- simple tableau de taile n (mémoire contigüe !) avec information de longueur
- access $O(1)$
- insert $O(n)$, 0(1) à la fin
- find $O(n)$
- delete $O(n)$, 0(1) à la fin
- sort $O(nlog(n))$

NB : on peut aussi créer des tableaux comme en C

Encapsulation d'un vecteur interne pour gérer les matrices pleines

```
class Matrix
{
private:
  int _s;
  vector<double> _vinternal;
public:
  Matrix(int s)
    : _vinternal(s*s, 0), _s(s) {}
  Matrix(const Matrix& m)
    : _vinternal(m._vinternal), _s(v._s) {}
  ~Matrix() {}
  inline double& element(int i, int j){
    return _vinternal[i*_s+j];
  }
};
```

Structure favorisant l'ajout/retrait des élements. La notion de longueur n'est pas fondamentale car variable.

- liste doublement-chainée
- access NA
- insert $0(1)$
- find $O(n)$
- delete $O(1)$
- sort $O(nlog(n))$

Exple : Un liste d'élements évolue, chacun pouvant survire ou non (delete de liste) et donner naissance à un nouvel élément (ajout en fin)

- une `list<.>` devrait convenir
- une approche basée sur les `vector<.>` mérite d'etre testée, avec ajout en fin de vecteur (naissance) ou copie du dernier élement à la place du delete (non survie) $\rightarrow$ danger de la copie

A tester après profiling !

Liste triée - set<.>

Permet de gérer un ensemble (au sens ensembliste - unicité) d'élements qui seront placés dans la structure selon un critère de tri.

Premier containeur associatif, i.e. on accède à l'élément par une clé.

- red-black tree (variante des arbres binaires [1]) Démo
- access NA
- insert $O(log(n))$
- find $O(log(n))$
- delete $O(log(N))$
- sort NA

Structure triée par construction, mais as-t-on besoin qu'elle le soit à chaque étape de sa construction ?

1. plus petites (grandes) valeurs dans le sous arbre de gauche (droite resp.)

Structures de données

Liste triée - set<.>

Exple : Insertion dans un set<.> ou tri/unicité à la fin avec un vector<.> ?

```
1  #include<set>
   #include<vector>
3  #include<algorithm>
   #include<stdlib.h>
5  using namespace std;
   int tirage(int n){
7    return (int)((double)rand()/((double)RAND_MAX + 1) * n);
   }
9  int main(int argc, char ** argv)
   {
11   vector<int> vec;
     set<int> s;
13   int n=1e7;

15   // 2 sec
     for (int i=0; i<n; i++)
17     vec.push_back(tirage(n));
     sort(vec.begin(), vec.end());
19   vec.erase( unique( vec.begin(), vec.end() ), vec.end() );

21   // 16 secondes
     for (int i=0; i<n; i++)
23     s.insert(tirage(n));
   }
```

Exple : Une matrice creuse en 0/1 (cas d'école ici) est une liste d'élements non nuls avec leur index. Comment l'interroger ? A-t-on besoin de la modifier ?

- `list<pair<unsigned int, unsigned int> >` : liste des paires d'indices → difficile à interroger
- `list<list<unsigned int> >` : une liste par ligne de la matrice, chaque liste contient les indices des colonnes non nulles → dynamique mais pas de contiguïté
- `vecteur<list<unsigned int> >` : si on connait le nombre de lignes, comme précédemment mais plus efficace ?
- `vecteur<set<unsigned int> >` : comme précédemment mais on a le tri et l'unicité des indices des colonnes → le tri est-il nécessaire à chaque étape ?

Exple : Une matrice creuse en 0/1 (cas d'école ici) est une liste d'élements non nuls avec leur index. Comment l'interroger ? A-t-on besoin de la modifier ?

- `vector<list<unsigned int> >` puis `sort/uniq` puis `vector<vector<unsigned int> >` : on passe par une structure temporaire pour aller vers la structure la plus efficace $\rightarrow$ poids de la recopie versus poids de l'utilisation de la structure finale

Quid des performances en machine : il faut tester/profiler.

Les conteneurs STL sont hyper-optimisés, les différences sont donc sensibles au delà d'une certaine taille de problèmes.
C++ coding standards, Herb Sutter, item 8 "Don't optimize prematurely"

Dictionnaire et table de hachage - `map<.,.>` et `unordered_map<.,.>`

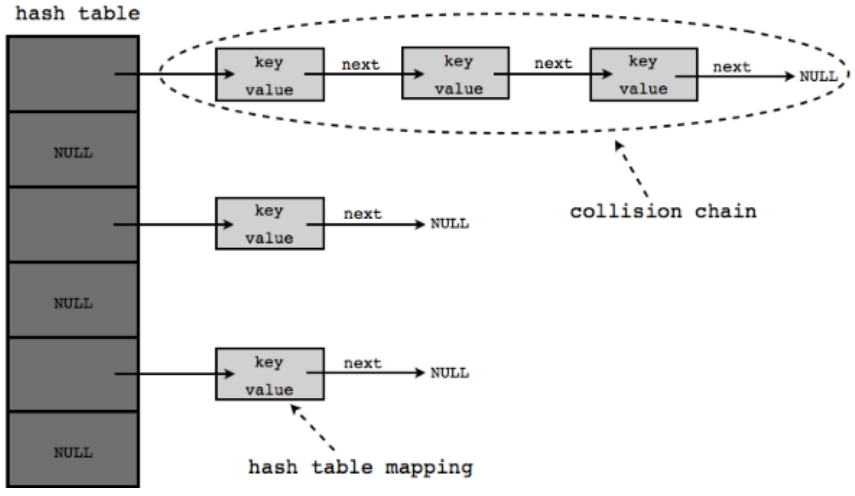
Associer des valeurs à des clés qui ne sont pas des entiers consécutifs (containeur associatif)

`map`

- `map<.,.>` == `set<pair<.,.> >`
- access avec opérateur [ ] mais attention $O(logn)$!
- même comportement que le `set<.>` pour le reste

`unordered_map`

- access par fonction de hachage (calcul d'un index depuis une clé) en $O(1)$ au mieux
- comme son nom l'indique, les élements ne sont pas ordonnés mais on peut les parcourir
- très efficace en pratique



Structures de données

Dictionnaire et table de hachage - map<.,.> et unordered_map<.,.>

```cpp
#include<map>
#include<vector>
#include<string>
#include<stdlib.h>
using namespace std;
int tirage(int n){
  return (int)((double)rand()/((double)RAND_MAX + 1) * n);
}
int main()
{
  vector<string> v;
  map<int, string> m;
  int n = 1e7;

  /******* init ***************/
  for (int i=0; i<n; i++)
    v.push_back("PuyDeDome");
  for (int i=0; i<n; i++)
    m[i] = "PuyDeDome";

  /******** acces **************/
  // 4 secondes
  for (int i=0; i<n; i++){
    int j = tirage(n);
    v[j] = "Cantal";
  }

  // 25 secondes
  for (int i=0; i<n; i++){
    int j = tirage(n);
    m[j] = "Cantal";
  }
}
```

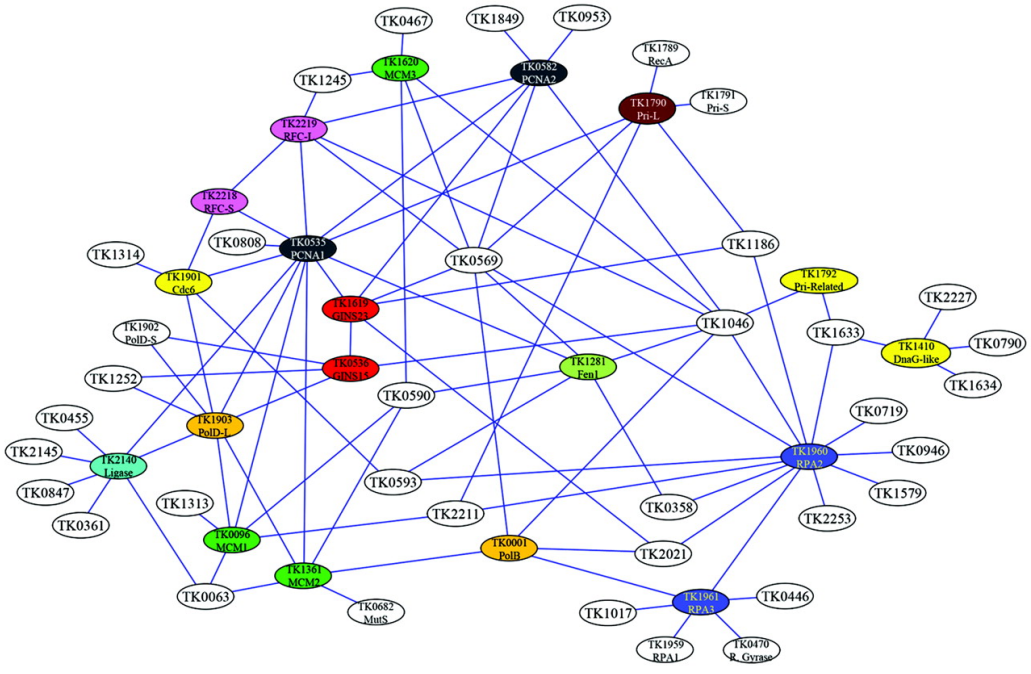
Dictionnaire et table de hachage - `map<.,.>` et `unordered_map<.,.>`

Exple : Une matrice creuse à valeurs réelles. . .

- `map<pair<unsigned int, unsigned int>, double>` : pratique mais pas de contiguité en mémoire, empreinte mémoire conséquente
- triplet de `vector` : les valeurs, le positionnement par ligne et les indices de colonnes $\rightarrow$ encapsulation
- s'appuyer sur une librairie

Dictionnaire et table de hachage - `map<.,.>` et `unordered_map<.,.>`

Exple :



Nécessité de retenir les noms des n sommets d'un graphe pour le pre/post-traitement, mais inutilité dans les étapes de calculs :

- conversion des noms en entiers consecutifs + stockage de arètes en matrice creuse
- utilisation de `map<string,unsigned int>` pour la conversion ou `vector<string>` + `find` en $O(log(n))$
- attention à la consommation de mémoire additionnelle

Use vector by default !

Lecture : Effective STL, Scott Meyers, item 23 “Consider replacing associative containers with sorted vectors”

Lecture : C++ coding standards, Herb Sutter, item 76 “Use vector by default. Otherwise choose an appropriate container”

Le reflexe du brainstorming

No container independent code !

Lecture : Effective STL, Scott Meyers, item 44 “Prefer member functions to algorithms with the same names”

“Be curious” = Lire les specs des structures proposées (cppreference, Wikipedia,. . .)

Lecture : C++ coding standards, Herb Sutter, item 8 “Don’t optimize prematurely”

Lecture :The C++ standard library, Nicolai Josuttis, page 10, “Remeber to think big when you consider complexity”

Faire des benchmarks pour se convaincre.

“Be lazy” = Toujours utiliser les structures de données des bibliothéques au plus haut niveau possible.

Ouvrir des livres et des revues

+ faire des benchmarks et du profiling (voir cours suivants)

+ participer à des réseaux métiers/collègues

+ éviter tout dogmatisme

+ Lecture : C++ coding standards, Herb Sutter, item 6 “Correctness, simplicity and clarity come first”