Ingénierie d'algorithmes et «optimisation» de code
— ou —
Les limites de l'analyse asymptotique

INF7440
Automne 2007

Aperçu

1. Limites de l'analyse asymptotique
2. Ingénierie d'algorithmes
3. Optimisation de code
4. Profilage de code

Ingénierie d'algorithmes et optimisation

Limites
1.1 Avantages
1.2 Désavantages
1.3 Morale
2. Ingénierie d'algorithmes
2.1 Définitions
2.2 Temps exécution
2.3 Évaluations
2.4 Expérimentations
2.5 Stratégies
3. Optimisation de code
3.1 Optimiser
3.2 Pas à la légère
3.3 Processus
3.4 Techniques
Changements macros
Changements micros
Les règles de Bentley
Profilage
4.1 Rôle
4.2 gprof

Avantages analyse asymptotique pire cas [Mor02]

- asymptotique
  ⇒ permet d'ignorer le «mauvais» comportement de l'algorithme sur des petites instances de problèmes
  ⇒ montre le *taux de croissance*
- pire cas
  ⇒ permet d'identifier une borne *supérieure* claire
  ⇒ analyse simple

Ingénierie d'algorithmes et optimisation

Limites
1.1 Avantages
1.2 Désavantages
1.3 Morale
2. Ingénierie d'algorithmes
2.1 Définitions
2.2 Temps exécution
2.3 Évaluations
2.4 Expérimentations
2.5 Stratégies
3. Optimisation de code
3.1 Optimiser
3.2 Pas à la légère
3.3 Processus
3.4 Techniques
Changements macros
Changements micros
Les règles de Bentley
Profilage
4.1 Rôle
4.2 gprof

Désavantages asympotique pire cas

- Intervalle de valeurs parfois valable au-delà de ce qui est possible dans des applications réelles
  Exemple = Algos arbre recouvrement minimum
- Constantes cachées peuvent rendrent impossible l'exécution effective d'un programme
  Exemple = Algorithme pour *mineur* d'un graphe
- Pire cas parfois associé à un nombre limité d'instances du problème, peu usuelles
  Exemple = Méthode du simplexe (prog. linéaire)
- Autre danger possible = algorithmes *«papier et crayon»*

Ingénierie d'algorithmes et optimisation

Limites
1.1 Avantages
1.2 Désavantages
1.3 Morale
2. Ingénierie d'algorithmes
2.1 Définitions
2.2 Temps exécution
2.3 Évaluations
2.4 Expérimentations
2.5 Stratégies
3. Optimisation de code
3.1 Optimiser
3.2 Pas à la légère
3.3 Processus
3.4 Techniques
Changements macros
Changements micros
Les règles de Bentley
Profilage
4.1 Rôle
4.2 gprof

Morale

Selon Moret [Mor02] :

- l'analyse asymptotique pire cas est importante

= outil de base de l'analyse

... mais

- doit être complétée par des expérimentations

⇒ ingénierie d'algorithmes (*algorithms engineering*)

Définitions et caractérisations du domaine

- Nouveau champ de recherche ≈ milieu 90
- Cause = Nombreux domaines où des résultats *théoriques* importants n'ont pas conduit à des technologies utilisables
  - Trop difficiles à mettre en oeuvre
  - Trop d'idéalisations ou de simplifications
- Définition de Cattaneao et Italiano [CI99] : *Algorithm engineering consists of the design, analysis, experimental testing and characterization of robust algorithms: it is mainly concerned with issues of realistic algorithm performance, and studies algorithms and data structures by carefully combining traditional theoretical methods together with thorough experimental investigations.*

Une vision plus concrète et plus large du «temps d'exécution»

Un bon algorithme doit avoir ... :

- Bonnes performances asymptotiques
- Faibles constantes de proportionnalité
- Bonnes performances sur des données réelles
- Performances robustes sur données variées
- Performances robustes sur plusieurs plate-formes
- Possibilité de «passage à l'échelle» (*scalable*) sur des plate-formes plus rapides et plus puissantes

Divers modes d'évaluation empirique

Le mode d'évaluation empirique dépend de l'objectif :

- Comparer différents algorithmes pour un problème donné
- Identifier les facteurs permettant d'optimiser les performances de programmes
- Évaluer des heuristiques
- Vérifier justesse des résultats
- Évaluer des accélérations

Expérimentations

Procédure pour expérimentations :

- Formuler clairement un ensemble d'objectifs : quelles sont les questions posées, que cherche-t-on à tester et vérifier par rapport au comportement de cet algorithme?
- Collecter les informations
- Analyser les données obtenues

Autres points importants :

- Expérimenter avec plusieurs machines, plusieurs compilateurs
- Grande variété de problèmes de tailles différentes
- Vraies données, vraies instances de problèmes

Stratégies pour améliorer la vitesse

Plusieurs approches pour améliorer la vitesse :

- Algorithme asymptotiquement meilleur — mais avec constantes multiplicatives faibles
- Algorithme parallèle
- Optimisations du compilateur
- Optimisations (améliorations) du code :
  - Optimiser les structures de données
  - Optimiser les détails algorithmiques
  - Optimiser le code
  - Réduire l'empreinte mémoire
  - Maximiser les références locales

Important d'évaluer l'effort requis [KP99] :

> *The main criterion is whether the changes will yield enough to be worthwhile. As a guideline, the personal time spent making a program faster should not be more than the time the speedup will recover during the lifetime of the program.*

Qu'est-ce qu'optimiser le code

Optimiser = Améliorer le code

Améliorations dépendent de différents objectifs possibles [Goo06] :

- Améliorer le temps d'exécution
- Diminuer l'espace mémoire utilisé
- Diminuer la taille du code exécutable
- Améliorer la qualité du code
- Améliorer la précision des résultats
- Diminuer le temps de démarrage
- Améliorer le débit de traitement des données (*data throughput*)

Pourquoi l'optimisation de code ne doit pas être faite à la légère

- P. Goodliffe [Goo06] : *Correct code is far more important than fast code. There's no point in arriving quickly at the wrong answer.*
- C.A.R. Hoare : *We should forget about small efficiencies, say about 97 percent of the time. Premature optimization is the root of all evil.*
- «Lois» de M. Jackson :
  1. *The First Rule of Program Optimization: Don't do it.*
  2. *The Second Rule of Program Optimization (for experts only): Don't do it yet.*

Pourquoi l'optimisation de code ne doit pas être faite à la légère (suite)

Pourquoi il est parfois préférable de *ne pas optimiser* [Goo06]:

- Réduit la *lisibilité*
- Augmente la «complexité»
- Code plus difficile à maintenir, à généraliser
- Possibilité de conflits avec d'autres parties du code
- Peut demander beaucoup de travail et d'effort. . . parfois en vain : (

Processus d'optimisation du code

1. Vérifier que le programme est vraiment trop lent et montrer qu'il doit absolument être optimisé.
2. Identifier la partie la plus lente du code = bien identifier la cible des optimisations.
3. Tester les performances de la cible des optimisations.
4. Optimiser le code.
5. Tester le code optimisé = montrer que cela a *vraiment* amélioré la partie ciblée.
6. Finalement, vérifier que l'optimisation a *vraiment* un effet sur le temps *global*.

Ensuite, décider de la prochaine étape — arrêter ou poursuivre les optimisations.

Techniques et catégories de Goodliffe [Goo06]

Trois principales techniques d'optimisation du code :

- Accélérer les parties lentes.
- Exécuter moins fréquemment les parties lentes.
- Retarder le plus possible l'exécution des parties lentes, jusqu'à ce qu'elles soient vraiment nécessaires.

Deux principaux «niveaux» [Goo06] :

- Changements de niveau *macro*, donc au niveau de la conception du code (haut niveau).
- Changements de niveau *micro*, donc au niveau du code (bas niveau).

Changements macros

Quelques exemples :

- Remplacer un algorithme ou structure de données par un–une autre plus efficace.
- Ajouter divers niveaux de cache pour éviter les recalculs ou pour améliorer la localité.
- Créer des bassins de ressources pour diminuer les surcoûts liés à l'allocation des ressources.
- Lorsque possible, réduire la précision des résultats de façon à améliorer le temps d'exécution.
- Etc.

Changements micros

Choses à éviter de faire, sauf si absolument nécessaire :

- Déroulage de boucles.
- Inclusion textuelles des fonctions et procédures.
- Pliage des constantes.
- Évaluation à la compilation plutôt qu'à l'exécution.
- Réduction de force des opérateurs.
- Élimination des sous-expressions communes.

À faire avec précaution :

- Mémoriser les résultats d'une fonction pour éviter de recalculer.
- Modifier l'ordre du code :
  - Retarder le travail le plus possible.
  - Déplacer les invariants à l'extérieur des boucles.
- Utiliser des tables de résultats pré-calculés.
- Exploiter l'évaluation court-circuit des expressions.

Les règles de Bentley

«*Writing efficient programs*» = diverses règles classées en différentes catégories :

- Règles visant à échanger du temps pour de l'espace
- Règles visant à échanger de l'espace pour du temps
- Règles pour les boucles
- Règles logiques
- Règles pour les procédures
- Règles pour les expressions

Rôle et importance du profilage

Outil important :

> *Profiling is invaluable in algorithm engineering—multiple cycles of profiling and revising the most time-consuming routines can easily yield gains of one to two orders of magnitude in running time.*

Permet d'identifier les points chauds : règle du 20/80 :

- «20 %» du code consomme «80 %» du temps

Mais, rôle du profilage :

- $\neq$ identifier la partie du code la plus lente
- $=$ identifier la partie du code où l'UCT passe la plus grande partie de son temps
- $\Rightarrow$ Le programmeur doit bien analyser les résultats

L'outil de profilage `gprof`

- Rôle de l'outil
  Fonction de `gprof` = compter le nombre d'appels effectués à chacune des fonctions
- Compilation du programme source

```
gcc -pg -o prog1.out prog1.c
```

- Exécution du programme exécutable
  Exécution $\Rightarrow$ génère fichier `gmon.out`
- Analyse de `gmon.out` avec `gprof`

```
gprof options [executable-file [profile-data-files...]]
              [> outfile]
```

- Rapport `grof` avec trois parties :
  - Profil plat : temps total consacré à chaque fonction
  - Graphe de appels : temps passé dans chaque fonction et ses enfants
  - Index des fonctions

L'outil de profilage `gprof` (suite)

- Limites du profilage

  Informations *statistiques*

  $\Rightarrow$ 100 échantillons par seconde!

  Donc :
  - Temps d'exécution suffisamment long
  - Plusieurs exécutions distinctes

Annexes

Comment comprendre le rapport de `gprof`

Traduction d'un extrait du manuel pour `gprof` : `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`

This manual was edited January 1993 by Jeffrey Osier. Copyright (C) 1988, 1992 Free Software Foundation, Inc. [...] Permission is granted to copy and distribute translations of this manual into another language, under the same conditions as for modified versions.

Ingénierie d'algorithmes et optimisation

Rapport `gprof`
A.1 Profil plat
A.2 Graphe appels
Exemples
B.1 McConnell
B.2 Kernighan & Pike
B.3 Autre
Utilisation de `mpdprof`

Comment comprendre le profil *plat*

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				mcount
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	profil
0.00	0.06	0.00	1	0.00	50.00	report

...

Comment comprendre le graphe des appels

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Exemples de S. McConnell [McC04]

– Mettre les invariants à l'extérieur des boucles :

- Avant :

```
for ( i = 0; i < image.width() * image.height(); i++ ) {
  ...
}
```

- Après :

```
int area = image.width() * image.height();
for ( i = 0; i < area; i++ ) {
  ...
}
```

Exemples de S. McConnell (suite)

– Mettre la boucle la plus active à l'intérieur :

- Avant :

```
for ( i = 0; i < 1000; i++ ) {
  for ( j = 0; j < 10; j++ ) {
    ...
  }
}
```

- Après :

```
for ( j = 0; j < 10; j++ ) {
  for ( i = 0; i < 1000; i++ ) {
    ...
  }
}
```

Exemples de S. McConnell (suite)

– Mettre les cas les plus fréquent au début :

- Avant :

```
if (c == '&' || c == '|') {
  ...
} else if (c >= '0' && c <= '9') {
  ...
} else if (c >= 'a' && c <= 'z') {
  ...
}
```

- Après :

```
if (c >= 'a' && c <= 'z') {
  ...
} else if (c >= '0' && c <= '9') {
  ...
} else if (c == '&' || c == '|') {
  ...
}
```

Exemples de S. McConnell (suite)

– Utiliser des opérateurs moins coûteux :

- Avant :

```
x = power(y, 2) / 2 - 2 * w;
```

- Après :

```
x = 0.5 * y * y - (w + w);
```

Exemples de S. McConnell (suite)

– Éviter les calculs inutiles :

- Avant :

```
dist(x1, y1, x2, y2) <= dist(u1, v1, u2, v2)

float dist( float a1, float b1, float a2, float b2 )
{
  return sqrt( (a1-a2)**2 + (b1-b2)**2 );
}
```

- Après :

```
dist_(x1, y1, x2, y2) <= dist_(u1, v1, u2, v2)

float dist_( float a1, float b1, float a2, float b2 )
{
  float a1a2 = a1 - a2;
  float b1b2 = b1 - b2;
  return( a1a2 * a1a2 + b1b2 * b1b2 );
}
```

Exemples de S. McConnell (suite)

– Fusionner les boucles :

- Avant :

```
for( i = 0; i < k; i++ ) {
  ins1;
}
...
for( i = 0; i < k; i++ ) {
  ins2;
}
```

- Après :

```
for( i = 0; i < k; i++ ) {
  ins1;
  ins2;
}
```

Exemples de S. McConnell (suite)

– Utiliser des sentinelles pour simplifier la fin de boucle :

- Après :

```
boolean fastLinearSearch( int[] arr, int target )
{
  int n = arr.length;
  if ( arr[n - 1] == target ) {  // Cible = dernier element?
    return true;
  }

  // Cible cherchee pas en derniere position.
  int old = arr[n - 1];      // On note la vieille valeur.
  arr[n - 1] = target;
  i = 0;
  while ( arr[i] != target ) {
    i++;
  }
  boolean result = i < arr.length;
  arr[n - 1] = old;          // On restaure la vieille valeur.
  return result;
}
```

Exemples de S. McConnell (suite)

– Bien utiliser les chaînes de caractères :

- Avant :

```
static String concatener( String[] chaines )
{
  String resultat = "";
  for ( int i = 0; i < chaines.length; i++ ) {
    resultat += chaines[i];
  }
  return( resultat );
}
```

- Après :

```
static String concatener( String[] chaines )
{
  StringBuffer resultat = new StringBuffer();
  for ( int i = 0; i < chaines.length; i++ ) {
    resultat.append( chaines[i] );
  }
  return( resultat.toString() );
}
```

Exemple de Kernighan et Pike [KP99]

– Mettre «en cache» (mémoriser) les valeurs fréquemment utilisées :

```
/* On verifie si c'est le meme qu'a l'appel precedent. */
if (c != lastc) {
  /* Non : alors on met a jour la valeur cachee. */
  lastc = c;
  lastcode = lookup( c );
}
show( lastcode );
```

Autre exemple

– Déroulage de boucles.

- Avant :

```
int s = 0;
for [i = 1 to N] {
  s += a[i];
}
printf( "s = %d\n", s );
```

- Après :

```
int s = 0;
for [i = 1 to N by 4] {
  s += a[i] + a[i+1] + a[i+2] + a[i+3];
}
printf( "s = %d\n", s );
```

Machine	1	2	4	8	16
Linux i686	0.323	0.308	0.303	0.291	0.288
arabica SunOs	2.60	2.46	2.40	2.43	2.42

Profil d'exécution d'un programme MPD

On utilise la commande `mpdprof` comme suit :

1. On associe un nom de fichier à la variable d'environnement `MPD_TRACE`.
2. On exécute le programme.
3. On exécute la commande `mpdprof -a` sur le fichier indiqué par `MPD_TRACE`.

Exemple dans l'environnement `bash` :

```
bash$ MPD_TRACE="fichier-trace.txt"
bash$ export MPD_TRACE
bash$ a.out
...
Termine en 0 ms
Sommaire: 1 suite(s), 3 test(s), 3 assertion(s), 0 cas echoue(s), 0
bash$ mpdprof -a fichier-trace.txt > listing-annote.txt
```

Exemple de listing annoté produit par `mpdprof`

```
bash$ cat listing-annote.txt
#====================
#  tests.mpd
#====================
resource TestsVecteurs()
#  BODY:1  ENDBODY:1
  import MPDUnit;
  #  CREATEG:1
  [...]

  procedure testScalaire1()
  #  PROC:1  ENDPROC:1
  {
    nommerCasDeTest( "Test 1" );
    #  CALL:1
    assertIntEquals( 200, produitScalaire(([1] 10), ([1] 20)) );
    #  CALL:2
  }

  procedure testScalaire2()
  #  PROC:1  ENDPROC:1
  {
    nommerCasDeTest( "Test 2" );
    #  CALL:1
    assertIntEquals( 800, produitScalaire((10, 20), (20, 30) ) );
    #  CALL:2
  }
  [...]
```


Références

G. Cattaneo and G. Italiano.
Algorithm engineering.
ACM Computing Surveys, 31(3es), 1999.
Article No. 3.

P. Goodliffe.
Code Craft—The Practice of Writing Excellent Code.
No Starch Press, 2006.

B.W. Kernighan and R. Pike.
The Practice of Programming.
Addison-Wesley, 1999.

S. McConnell.
Code Complete—A Practical Handbook of Software Construction (Second Edition).
Microsoft Press, Redmond, WA, 2004.

B.M.E. Moret.
Towards a discipline of experimental algorithmics.
In *Proc. 5th DIMACS Challenge*, volume DIMACS Monographs 59, pages 197–213. American Mathematical Society, 2002.

Ingénierie d'algorithmes et optimisation

Rapport `gprof`
A.1 Profil plat
A.2 Graphe appels

Exemples
B.1 McConnell
B.2 Kernighan & Pike
B.3 Autre

Utilisation de `mpdprof`