Sujet : Algorithmique et prédiction de branchement

Il est rappelé que le jury attend un exposé de 35 minutes, pédagogique et structuré, fondé sur ce sujet. Ce texte se conclut par une liste de pistes de réflexion pour guider la conception de l'exposé, mais toute initiative personnelle pertinente est appréciée. L'exposé doit contenir une ou plusieurs illustrations informatiques ainsi qu'une discussion autour d'une dimension éthique, sociétale, environnementale, économique ou juridique. Une piste particulière est proposée à cet effet.

1 Introduction

Le cadre habituel pour étudier la complexité d'un algorithme est de considérer que les opérations élémentaires se font en temps constant, et d'estimer comment le temps (ou l'espace nécessaire) croît en fonction de la taille de l'entrée dans le pire cas.

Cette façon d'étudier l'efficacité d'algorithmes permet de s'éloigner au maximum des machines elles-mêmes et de se concentrer sur l'algorithmique afin d'obtenir ainsi des résultats généraux, indépendants de l'architecture de l'ordinateur, de la vitesse du processeur, du langage de programmation, etc. Cela permet, à haut niveau, de préférer une solution algorithmique à une autre.

Quand des algorithmes dont l'efficacité est critique sont implantés, de nombreuses optimisations y sont apportées pour les rendre plus efficaces. Il s'agit souvent de traiter différemment les entrées de petites tailles ou de tenir compte de l'architecture des ordinateurs. On peut, par exemple, essayer de minimiser les erreurs d'accès au cache en faisant attention à l'ordre des accès mémoire.

L'importance de ces optimisations en pratique a conduit l'analyse d'algorithmes à évoluer, et à enrichir le modèle d'ordinateur utilisé en y intégrant des éléments d'architecture des ordinateurs. Pour reprendre l'exemple ci-dessus, on peut ajouter au modèle une notion de cache (dont on connaît ou non la taille) et revisiter les algorithmes classiques dans ce nouveau modèle ; cela a été fait dans de nombreux travaux théoriques et

cela a permis de trouver des solutions plus adaptées quand les problèmes d'accès au cache deviennent critiques.

Dans ce sujet, nous nous intéressons à une autre optimisation d'architecture présente dans presque tous les ordinateurs : la prédiction de branchement. Nous allons voir comment on peut essayer d'intégrer ce mécanisme au niveau théorique et étudier son impact sur des algorithmes pour des problèmes élémentaires.

2 Pipeline

Un *pipeline* (parfois appelé *chaîne de traitement*) est un élément du processeur qui optimise le traitement des instructions en les découpant en *micro-instructions*.

Supposons, par exemple, que chaque instruction à exécuter est découpée en 5 micro-instructions. On note A_i, B_i, C_i, D_i et E_i les micro-instructions, à effectuer dans l'ordre, pour la i-ème instruction. Le pipeline peut être vu comme composé de 5 parties séparées, chacune pouvant traiter un type de micro-instruction. Ainsi, A_1 va être effectuée d'abord, puis, lorsque l'on passe à B_1, la micro-instruction A_2 peut être effectuée en parallèle puisque la partie qui traite les micro-instructions A est libre. À l'étape d'après, C_1, B_2 et A_3 seront effectuées simultanément (voir la figure 1). Si tout se passe au mieux, on aura ainsi quasiment gagné un facteur 5 en traitant simultanément 5 micro-instructions de 5 instructions différentes.

Important : les instructions dont on parle ici sont celles effectuées par le processeur. Une instruction en langage `C`, telle que "`x = y + 3;`" sera décomposée en plusieurs instructions du processeur par le compilateur. Pour ne pas compliquer la présentation du mécanisme, on utilisera abusivement les instructions en langage `C` comme si c'était des instructions du processeur. De façon générale, la présentation des éléments d'architecture des ordinateurs de ce document est simplifiée, afin d'en abstraire le mécanisme à incorporer à une étude algorithmique.

La situation idéale où les 5 parties du pipeline traitent 5 micro-instructions simultanément n'est pas tout le temps réalisable. Par exemple, si on exécute les instructions `y = y + 3; x = y;`, on risque de devoir attendre le résultat de `y = y + 3;` avant de pouvoir avancer dans le traitement de `x = y;`. Un des cas les plus flagrants est quand on utilise une instruction conditionnelle :

`if <condition> {<instruction1>} else {<instruction2>}`

car l'instruction qui suit `<condition>` dépend de si elle est vraie ou fausse : on ne peut pas commencer le traitement de l'instruction suivante tant qu'on n'a pas le résultat de `<condition>` puisqu'on ne sait pas encore quelle est l'instruction suivante. Le même phénomène apparaît si on fait une boucle `while` ou `for`, l'instruction suivante étant différente si on itère la boucle ou si on en sort. Tous ces cas avec deux continuations différentes sont ce qu'on appelle des *branchements*.

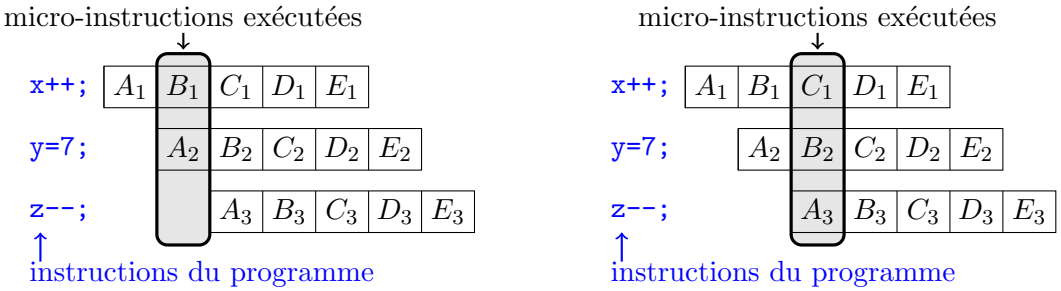


FIGURE 1 – Deux étapes successives du traitement de 3 instructions (`x++; y=7;` et `z--;`) dans un pipeline de taille 5, quand une parallèlisation totale est possible. Lors de l'étape de gauche, les micro-instructions B_1 et A_2 sont effectuées simultanément. Lors de l'étape de droite, les micro-instructions C_1, B_2 et A_3 sont effectuées simultanément. On peut ainsi effectuer jusqu'à 5 micro-instructions en parallèle.

3 Prédiction de branchement

Pour éviter que le pipeline ne soit bloqué lors d'un branchement, la plupart des processeurs ont un *prédicteur* : il s'agit d'un mécanisme de prédiction permettant d'anticiper laquelle des deux branches va être suivie avant d'avoir le résultat de la condition. On continue l'enchaînement des instructions comme prédit par le prédicteur. Si la prédiction s'avère exacte, on a gagné du temps, mais il y a une pénalité conséquente si le prédicteur s'est trompé. La qualité du prédicteur est donc particulièrement importante. Il faut qu'il fasse mieux que s'il n'y avait pas de système de prédiction du tout.

Il existe de nombreux mécanismes de prédiction. Certains, les *prédicteurs locaux*, sont propres à chaque instruction de branchement et font leur prédiction uniquement sur leur historique propre. D'autres, les *prédicteurs globaux* ont un historique commun à toutes les instructions de branchement, qu'ils utilisent pour leur prédiction. Dans la plupart des cas, un mélange de ces deux procédés est utilisé, où le prédicteur a accès à la fois à un historique global et local.

Dans toute la suite **on considèrera uniquement des prédicteurs locaux :** chaque instruction de branchement (condition d'un `if`, `while` ou `for` pour nous) possède un prédicteur propre qui s'actualise en fonction de son historique propre, c'est-à-dire la séquence des résultats de sa condition, TRUE (noté T) ou FALSE (noté F).

Il y a deux branchements dans l'exemple du code en langage `C` de la figure 2 : le `while` de la ligne 4 et le `if` de la ligne 5. On considère que ce sont exactement les branchements qui apparaissent quand le programme est compilé.

```
1  int somme_positifs(int *t, int n) {
2    int i, s;
3    i = s = 0;
4    while (i < n) {
5      if (t[i] > 0)
6        s += t[i];
7      i ++;
8    }
9    return s;
10 }
```

FIGURE 2 – Une fonction en C qui calcule la somme des éléments positifs dans un tableau d'entiers.

4 Un modèle local : le prédicteur 2-bits saturé

Ce modèle consiste à utiliser deux bits par instruction de branchement pour encoder un des quatre états d'un automate de prédiction. L'automate est décrit dans la figure 3 ci-dessous.

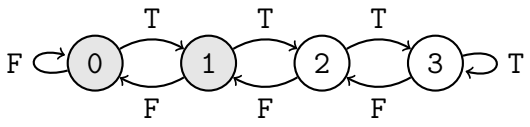
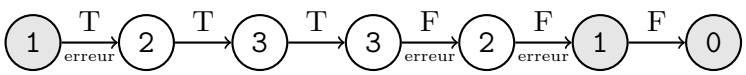


FIGURE 3 – Le prédicteur 2-bits saturé.

Il s'agit d'un système à 4 états, dont les états 0 et 1 prédisent FALSE, et les états 2 et 3 prédisent TRUE. À chaque fois que l'instruction de branchement rattachée au prédicteur est effectuée, l'état est mis à jour en fonction de ce qui s'est réellement passé, en diminuant le numéro de l'état si le résultat était FALSE et en l'augmentant s'il était TRUE.

Par exemple, si on commence dans l'état 1 et que la suite des évaluations de la condition est TRUE,TRUE,TRUE,FALSE,FALSE,FALSE (qu'on notera plus simplement TTTFFF), on fait 3 erreurs de prédiction. La première erreur a lieu au tout début : comme on est dans l'état 1, on prédit F alors que le résultat de la condition est T. L'évolution du processus sur l'exemple est décrite ci-dessous.



Le prédicteur 2-bits saturé est fréquemment utilisé, mais souvent en conjonction avec d'autres mécanismes que nous ne détaillerons pas ici. Il sera le principal objet d'étude de ce document.

5 Étude pire cas et en moyenne du programme `somme_positifs`

Le programme `somme_positifs` de la figure 2 fait la somme des nombres strictement positifs dans un tableau de taille n. On voit facilement que, quel que soit l'état de départ du prédicteur 2-bits, on peut choisir le signe des éléments du tableau pour forcer une erreur de prédiction à chaque étape sur le `if` de la ligne 5. Il y aura donc n erreurs de prédiction dans le pire cas pour ce programme.

Cette analyse dans le pire cas n'est pas forcément très pertinente pour mesurer l'impact du mécanisme de prédiction. Nous allons donc procéder à une analyse en moyenne. Pour cela, on va admettre et utiliser le résultat suivant.

Théorème 1 *Soit $p \in]0,1[$ un réel. Si on part de n'importe quel état d'un prédicteur 2-bits saturé et qu'on effectue n transitions où à chaque fois on tire au sort* T *avec probabilité p et* F *avec probabilité $1-p$ (indépendamment des autres tirages), alors le nombre moyen d'erreurs de prédiction est asymptotiquement équivalent à $\mu_2(p) \cdot n$ avec $\mu_2(p) = \frac{p(1-p)}{1-2p(1-p)}$.*

Ainsi, si on considère que chaque nombre dans le tableau `t` est strictement positif avec probabilité $p = 3/4$ et négatif avec probabilité $1 - p = 1/4$, on peut appliquer directement le théorème 1 et obtenir qu'en moyenne on va faire de l'ordre de $\frac{3n}{10}$ erreurs de prédiction, quel que soit l'état initial du prédicteur. On remarque que si on fixe $p = 1/2$, le prédicteur se trompe en moyenne une fois sur deux ; il n'arrive pas à prédire correctement.

6 Exemple : calcul du minimum et du maximum

6.1 Deux algorithmes pour résoudre le problème

On considère le problème suivant : étant donné un tableau de nombres `t` de taille n, calculer à la fois la valeur minimale et maximale du tableau en utilisant des comparaisons entre éléments de `t`. On considère d'abord la solution naïve présentée en Python dans la figure 4. Le programme `naif_min_max` effectue précisément $2(n-1)$ comparaisons entre éléments du tableau, quelle que soit l'entrée `t` de taille n.

Une solution plus performante en nombre de comparaisons [1] consiste à comparer d'abord entre eux deux éléments de `t`, puis le plus petit des deux au minimum courant et le plus grand des deux au maximum courant. Cette solution utilise 3 comparaisons d'éléments du tableau pour traiter deux d'entre eux, et fait donc de l'ordre de $\frac{3}{2}n$ comparaisons au total. L'algorithme est donné figure 5.

1. On peut même montrer que cet algorithme est optimal en nombre de comparaisons si on ne s'autorise que les comparaisons de valeurs comme seul accès aux données de `t`.

```
1 def naif_min_max(t):
2     mini = maxi = t[0]
3     for i in range(1,len(t)):
4         valeur = t[i]
5         if valeur < mini: mini = valeur
6         if valeur > maxi: maxi = valeur
7     return mini, maxi
```

FIGURE 4 – Algorithme min-max naïf, qui effectue $2(n-1)$ comparaisons d'éléments du tableau.

```
1 def opti_min_max(t):
2     mini = maxi = t[-1]
3     for i in range(0,len(t)-1,2):
4         valeur_min, valeur_max = t[i], t[i+1]
5         if valeur_min > valeur_max:
6             valeur_min, valeur_max = valeur_max, valeur_min
7         if valeur_min < mini: mini = valeur_min
8         if valeur_max > maxi: maxi = valeur_max
9     return mini, maxi
```

FIGURE 5 – Algorithme min-max optimal, qui effectue $\sim\frac{3}{2}n$ comparaisons d'éléments du tableau.

6.2 Nombre d'erreurs de prédiction de `naif_min_max`

On s'intéresse aux erreurs de prédiction des `if` des lignes 5 et 6 de l'algorithme `naif_min_max` de la figure 4. L'analyse dans le pire cas n'est à nouveau pas très instructive : même si on ne connaît pas l'état de départ du prédicteur, on peut construire `t` de sorte qu'on ait une erreur de prédiction à chaque fois qu'on fait le `if` de la ligne 5 ou de la ligne 6, sauf peut-être pour les quelques premières itérations. Il y aura donc de l'ordre de $\sim 2n$ erreurs de prédiction dans le pire cas, la contribution du `for` étant négligeable pour les erreurs de prédiction.

On s'intéresse donc à nouveau au cas moyen en considérant que le tableau est une permutation aléatoire uniforme des nombres de $\{0,\dots,n-1\}$, ce qui signifie que chaque permutation a une probabilité $\frac{1}{n!}$ d'être l'entrée `t` de l'algorithme.

On va estimer le nombre moyen d'erreurs de prédiction du `if` de la ligne 5. Par symétrie, c'est aussi le nombre moyen d'erreurs du `if` de la ligne 6. Par les propriétés de la moyenne (linéarité de l'espérance) on peut donc juste multiplier par deux le résultat pour le `if` de la ligne 5 et avoir le résultat final.

Pour une permutation σ de $\{0,\dots,n-1\}$, on dit que i est un *record* pour σ si $\sigma(i) < \sigma(j)$

pour tout $j \in \{0, \ldots, i-1\}$. On remarque que cela correspond aux mises à jour du `if` de la ligne 5, qui ont lieu exactement quand on a un record dans `t`. On va utiliser le résultat suivant sur les permutations aléatoires.

Théorème 2 *Si on prend une permutation de* $\{0, \ldots, n-1\}$ *uniformément au hasard, le nombre moyen de records est asymptotiquement équivalent à* $\log n$.

Le théorème 2 permet assez facilement de prouver que le nombre moyen d'erreurs de prédiction de `naif_min_max` est en $O(\log n)$. Un simple prédicteur comme le 2-bits saturé permet donc de profiter pleinement du pipeline pour cet algorithme.

6.3 Nombre d'erreurs de prédiction de `opti_min_max`

Le pire cas n'est à nouveau pas très intéressant ; on peut construire des exemples qui forcent les trois `if` à faire des erreurs de prédiction presque tout le temps.

Pour le cas moyen, la situation est différente de celle de `naif_min_max`. En effet, pour notre modèle de permutation, la condition `valeur_min > valeur_max` du `if` de la ligne 5 a une chance sur deux d'être évaluée à TRUE et une chance sur deux d'être évaluée à FALSE. Donc, d'après le théorème 1, le nombre moyen d'erreurs de prédiction sur les $\sim n/2$ itérations de la boucle est asymptotiquement équivalent à $n/4$. Les autres erreurs de prédiction ont une contribution en $O(\log n)$, en utilisant les mêmes arguments que pour l'algorithme naïf.

6.4 Expériences

La conclusion de cette partie est que, pour notre analyse aléatoire, on propose deux solutions :

- `naif_min_max` qui fait de l'ordre de $2n$ comparaisons et $O(\log n)$ erreurs de prédiction.
- `opti_min_max` qui fait de l'ordre de $\frac{3}{2}n$ comparaisons et de l'ordre de $\frac{1}{4}n$ erreurs de prédiction.

Un point clé pour savoir quel est le plus performant des deux est de connaître le coût relatif d'une erreur de prédiction par rapport à une comparaison. On a donc effectué deux études expérimentales sur un ordinateur personnel :

- La première expérience utilise le code Python présenté ci-dessus, dans un notebook Jupyter. Le temps a été mesuré très simplement avec la bibliothèque `time` de Python, en regardant l'heure avant et après l'application de l'algorithme pour chaque taille de tableau considérée.
- La seconde expérience utilise du code écrit en `C`, compilé sans optimisations pour se rapprocher du code source le plus possible. Le temps a été mesuré le plus précisément possible.

Les résultats des expériences sont donnés dans la figure 6 et la figure 7.

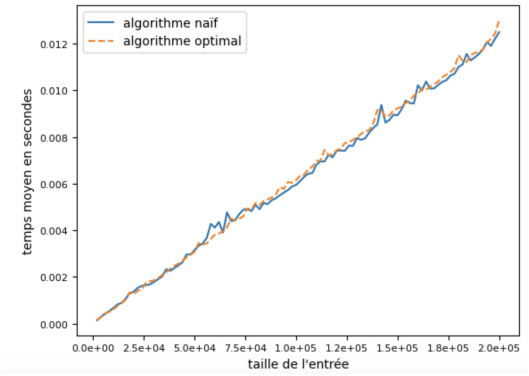


FIGURE 6 – Étude des performances sur des tableaux aléatoires avec un notebook Python. Le programme `naif_min_max` est en bleu, `opti_min_max` en pointillés orange.

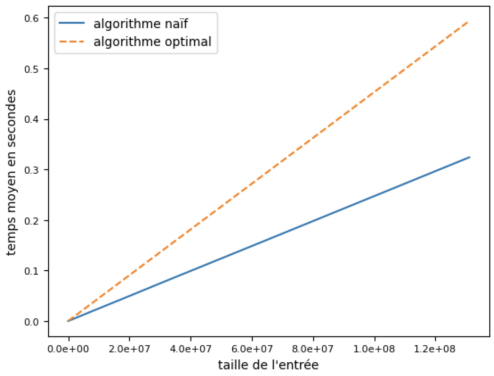


FIGURE 7 – Étude des performances sur des tableaux aléatoires en C. Le programme `naif_min_max` est en bleu, `opti_min_max` en pointillés orange.

On observe que même s'il fait beaucoup moins de comparaisons, les performances de l'algorithme "optimal" ne sont pas meilleures que celle de l'algorithme naïf en Python. Elles sont même moins bonnes en `C`, où les comparaisons sont moins coûteuses que dans un langage interprété non typé comme Python. En s'assurant qu'il n'y a pas d'autres mécanismes d'architecture qui perturbent l'analyse, on peut ainsi mettre en évidence un réel impact de la prédiction de branchement sur le choix de la meilleure solution.

7 Conclusion

On peut trouver d'autres exemples où des solutions sous-optimales a priori sont meilleures grâce aux prédicteurs. Si les comparaisons sont rapides (sur des types primitifs), on peut montrer qu'il peut être intéressant de ne pas couper au milieu quand on fait des recherches dichotomiques, mais par exemple au quart : on effectue plus d'itérations en moyenne et donc plus de comparaisons, mais on aide le prédicteur à mieux anticiper. Selon les caractéristiques du processeur, on peut rendre cette dichotomie biaisée plus efficace que la dichotomie classique.

Une étude systématique est pourtant difficile à mener. Tout d'abord, il existe de nombreux prédicteurs différents, dont des prédicteurs globaux, mentionnés en introduction, qui utilisent un historique global des résultats des conditions de branchement ; ceux-ci sont plus compliqués à analyser. Ensuite, comme on l'a vu, l'analyse pire cas n'est souvent pas intéressante, et faire une analyse en moyenne nécessite d'avoir une bonne modélisation aléatoire des entrées pour être pertinente. Une autre difficulté importante est que la plupart des constructeurs de processeurs n'indiquent pas, volontairement, les mécanismes de prédiction de branchement qu'ils utilisent dans leur matériel. Enfin, les gains sont souvent marginaux, et ne sont visibles que s'il y a peu d'instructions élémentaires par rapport aux erreurs de prédiction.

Pistes de réflexion pour l'exposé

1. Vérifier expérimentalement le théorème 1 en écrivant un programme qui simule un prédicteur 2-bits saturé. Est-ce que, avec les mêmes hypothèses que dans l'énoncé du théorème, un prédicteur 3-bits saturé (avec 8 états) a un meilleur taux de prédiction moyen ?
2. Discuter de l'efficacité d'un prédicteur local 2-bits saturé pour des boucles `for` et des boucles `while`.
3. Proposer une permutation de taille n pour laquelle `naif_min_max` fait asymptotiquement de l'ordre de $2n$ erreurs de prédiction, quels que soient les états initiaux des prédicteurs.
4. Ajouter des simulations de prédicteurs locaux 2-bits saturés dans les codes de `naif_min_max` et `opti_min_max` pour vérifier expérimentalement les résultats annoncés dans la partie 6.
5. Discuter des irrégularités observées sur les courbes de la figure 6.
6. Si le programme est écrit en `C` et compilé, les choix effectués par le compilateur (éventuellement guidés par des options de compilations) peuvent changer le nombre d'erreurs de prédiction. Discuter de cet état de fait.
7. En utilisant des simulations, estimer le nombre moyen d'appels récursifs effectués lorsqu'on utilise une dichotomie où on coupe au quart au lieu de la moitié. Utiliser cette quantité expérimentale pour estimer le nombre d'erreurs de prédiction de la dichotomie ainsi biaisée.
8. Discuter des erreurs de prédictions lors du partitionnement de l'algorithme de quicksort, si on utilise des prédicteurs locaux 2-bits saturés. Est-ce qu'il y a un moyen de sélectionner le pivot pour aider le prédicteur de branchement ? Si oui, comment s'assurer que la solution ne dégrade pas les performances de l'algorithme ?
9. Reprendre la question précédente pour la construction des arbres binaires de recherche.
10. Le détail des mécanismes de prédiction de branchement utilisés dans les processeurs vendus sur le marché est souvent gardé secret. Discuter des conséquences sociétales et économiques de cet état de fait.