Etude des unités de prédictions de branchements

Tom Redon

Rapport M1 - 2021

Encadré par Mme. Carine Pivoteau (LIGM) et Mr. Olivier BOUILLOT (UGE)
Université Gustave Eiffel - Laboratoire d'informatique Gaspard Monge

Contents

1 remerciements

Je tenais particulièrement à remercier Mme. Carine Pivoteau et Mr. Cyril Nicaud pour leur gentillesse, leur qualité d'encadrement, leur investissement et le temps qu'ils m'ont accordés.
Merci aux équipes administratives du Laboratoire d'Informatique Gaspard Monge et de l'Université Gustave Eiffel de m'avoir facilité l'accès à ce stage.
Merci à l'équipe enseignante du Master 1 pour ses apports théoriques et techniques qui m'ont permis de mener à bien le projet qui m'a été confié.

2 Introduction

Ce stage de Master 1 a été effectué dans l'équipe Modèles et Algorithme du Laboratoire d'Informatique Gaspard Monge et a été encadré par Mme. Pivoteau et Mr. Nicaud, sur une durée de trois mois. Le sujet sur lequel s'est concentré ce stage est l'étude des unités de prédiction de branchements et l'observation de leur utilisation. Après avoir revu des notions d'architectures nécessaires à la compréhension du sujet, le projet s'est porté tout d'abord sur la génération des graphes, puis sur la transformation de ces dits-graphes en chaînes de Markov. Après ces travaux, il a été utile d'effectuer des calculs de probabilités stationnaires afin de permettre d'analyser les rangs de classements des différentes unités de prédiction de branchements. Dans un premier temps, nous reverrons les différentes parties théoriques nécessaires à la compréhension du projet, puis dans un second temps la description détaillée de ce dernier. Enfin, nous "tudierons les résultats obtenus ainsi que la mise en place d'un simulateur permettant d'observer une unité de prédiction de branchements en action.

3 Notions d'Architecture nécessaires

Aujourd'hui, la plupart des processeurs sont pipelinés. Cela signifie que chaque instruction est séparée en plusieurs étapes et le processeur peut ainsi exécuter plusieurs instructions en parallèle, sans avoir nécessairement besoin d'attendre la fin de l'exécution de la précédente.
Le processeur peut donc recevoir différents types d'instructions et fera en sorte d'optimiser l'exécution. On s'intéressera principalement à l'exécution des instructions conditionnelles ("if"). Ces instructions conditionnelles permettent de pouvoir "sauter" à une certaine partie de code en fonction de la valeur de la condition lue dans l'instruction. Le saut effectué est une redirection de la lecture du code vers une ligne du code assembleur, précisée par l'instruction conditionnelle. Elles sont aussi particulières car elle demande au pipeline d'attendre la fin de leur exécution avant de pouvoir continuer. On peut donc facilement se retrouver avec un pipeline en attente à cause d'instructions conditionnelles trop longues à résoudre.
Pour essayer d'optimiser l'exécution de ces instructions, le processeur essaye de réaliser des prédictions pour savoir s'il doit prendre le saut ou non. Pour pouvoir faire ces prédictions, des unités de prédictions de branchements (prédicteurs de branchements) ont été développés. Il existe de nombreux types de prédicteurs de branchements. Il existe par exemple les prédicteurs statiques ou bien encore les prédicteurs dynamiques.
Les prédicteurs statiques ne se servent pas des informations récoltées pendant l'exécution du code précédent. Il aura donc des prédictions pré-établies. Par exemple, répondre à tous les branchements qu'ils sont pris. De ce fait, leur exécution est rapide, mais leur taux d'erreur de prédiction est relativement élevé.
Le modèle de prédicteur dynamique a été établi pour essayer d'obtenir un taux d'erreur de prédiction plus restreint. Il se sert des évaluations des instructions précédentes du code pour prendre ses décisions et sera souvent plus précis. Il existe différents niveaux de prédicteurs dynamiques.
On va régulièrement parler de prédicteurs à k-bits, $k \in \mathbb{N}$, où les k bits représentent le nombre d'information retenue. Le nombre d'états de l'automate sera $2 * k$. Ainsi, on pourra voir des modèles assez simples n'utilisant que la prédiction précédente pour établir la nouvelle et qui seront considérés comme des prédicteurs à 1-bit. Les modèles les plus répandus sont les prédicteurs à 2-bits, et plus précisément le compteur saturé qui permet de répondre très efficacement quand le cas étudié prend des conditions dont les réponses sont identiques pendant un certain nombre de prédictions. On peut aussi parler du prédicteur nommé le flip-on-consecutive qui permet d'effectuer une transition de prédiction plus simplement lorsque les résultats des prédictions précédentes ne sont pas obligatoirement identiques entre elles lorsqu'elles se suivent. Vous pouvez observer un exemple de ces deux prédicteurs à 2-bits dans le schéma suivant.

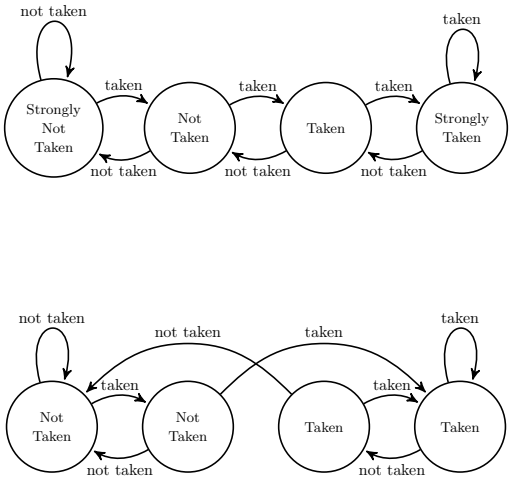


Figure 1: Deux prédicteurs à 2-bits différents (haut : compteur saturé, bas: flip-on-consecutive).

Pour représenter ces différents prédicteurs dynamiques, on souhaite étudier et donc générer les graphes à $2 * k$-états, $k \in \mathbb{N}$, fortement connexes et non-isomorphes entre eux.

4 Génération des graphes

Pour produire ces graphes, plusieurs méthodes ont été appliquées pendant l'avancée du projet. La première n'a pas été sélectionnée car elle était largement moins performante que la seconde. Vous pouvez donc continuer votre lecture après l'exemple.

4.1 Génération générale des graphes et algorithme de Tarjan

En premier lieu, tous les graphes à k-états, $k \in \mathbb{N}$, étaient générés à partir de combinaisons établies pour représenter les différentes destinations de chaque arête.
Ainsi, pour un graphe à k-états, on a $2 * k$ arêtes (Une arête "Taken" et une arête "Not Taken" par état).
On doit déterminer, pour chacune, l'état vers lequel elle se dirige.
Pour représenter cela, on générait des combinaisons de taille $2*k$ et dont chaque valeur pouvait appartenir entre 0 et $k-1$.
Ces combinaisons étaient ensuite permutées de toutes les façons possibles pour pouvoir générer tous les graphes à k-états.

Exemple 1 *Pour un graphe à 4-états, on pouvait par exemple avoir la combinaison suivante :*

$$Combinaison = \begin{pmatrix} 0 & 3 & 3 & 2 & 1 & 2 & 0 & 1 \end{pmatrix}$$

On peut donc le lire de la façon suivante :
$\forall n \in [0, k-1], k \in \mathbb{N}, Combinaison[2*n]$ *désigne l'état-destination de la branche "Not Taken".*
$\forall n \in [0, k-1], k \in \mathbb{N}, Combinaison[2*n+1]$ *désigne l'état-destination de la branche "Taken".*
Cette combinaison produira le graphe suivant :

On pourra ensuite permuter la combinaison pour obtenir d'autres graphes.

Cette solution n'étant pas optimale en terme de performance, il a été décidé de générer toutes les combinaisons dont les graphes découlants seraient fortement connexes et n'ayant pas déjà été générés précédemment.
On transforme ensuite ces combinaisons en graphes, puis on vérifie grâce à l'algorithme de Tarjan que les graphes ne forment bien qu'une seule composante fortement connexe chacun.
L'algorithme de Tarjan étant un algorithme connu, il ne sera pas détaillé de nouveau ici. Vous pouvez cependant retrouver une description exhaustive de son but et de son implémentation ici : [1].

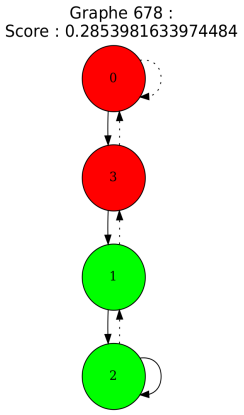


Figure 2: Premier graphe

4.2 Parcours en profondeur et signature des graphes

On effectue par la suite un parcours en profondeur avec étiquetage des chemins, pour vérifier efficacement l'isomorphisme entre les graphes générés.
La signature d'un graphe est une étiquette représentant la disposition de ses états et de ses arêtes, qui aura une formulation bien précise afin de pouvoir identifier deux graphes qui auraient le même type de disposition des états, sans pour autant avoir exactement la même numérotation d'état.
Nous verrons dans l'exemple suivant comment déterminer les signatures d'un graphe.
Le principe du parcours en profondeur est donc le suivant.
Pour chaque graphe ayant été vérifié par l'algorithme de Tarjan :
- on établit une étiquette pour le graphe courant en partant du premier état (généralement l'état 0).
- on compare cette étiquette à celles précédemment enregistrées dans un ensemble, si elle n'est pas dans l'ensemble, on enregistre les k signatures existantes pour le graphe (les autres signatures sont déterminées en commençant à partir des autres états). Si elle est dans l'ensemble, alors on passe à un autre graphe.

Exemple 2 *En prenant donc en considération le graphe représenté précédemment (2), on pourra facilement observer la génération de la signature du graphe. On va tout d'abord observer l'état 0.*

On créé plusieurs listes qui nous seront utiles pour la suite :
En premier lieu, on réalise une première exploration du graphe pour marquer l'ordre de rencontre du graphe.
Pour cela, on va tout d'abord marquer l'état comme visité, puis on va ajouter ce dernier dans une liste qui référencera les états dans leur ordre de découverte, que l'on nommera la liste "order". Par la suite, on continuera l'exploration dans les états voisins qui n'ont pas encore été visités. Quand tous les états sont identifiés comme visités, on arrête l'exploration.
En partant de l'état 0, on obtiendra donc la liste "order" suivante :

$$[0, 3, 1, 2]$$

On effectue un second parcours, en utilisant cette liste "order" pour établir l'étiquette.
Pour chaque état dans la liste "order", on va regarder ses voisins dans l'ordre suivant :
En premier lieu, on observera le voisin auquel on peut accéder par l'arête "Not Taken", puis celui lié par l'arête "Taken". On construit ensuite la signature, en ajoutant à chaque fois la position dans la liste "order" de l'état voisin que l'on observe actuellement.
Ici avec la liste précédemment établie, on obtiendrait la signature suivante :

"01021323"

Cette chaîne de caractères est donc l'une des quatres signatures possibles pour le graphe étudié.

Une fois que les graphes sont validés comme étant fortement connexes et non-isomorphes entre eux, on peut les considérer comme des graphes de chaînes de Markov.

Exemple 3 *On peut exposer un second exemple de graphe :*

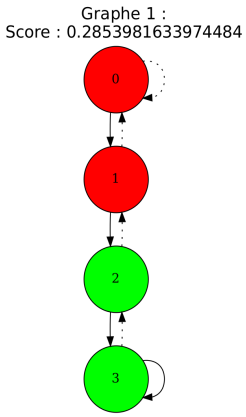


Figure 3: Second graphe

On peut donc établir toutes les signatures des graphes tels que :

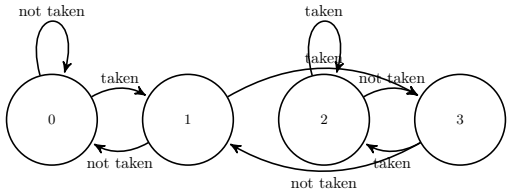
$$Graphe1 : "01021323", "12100323", "13202103", "10203132"$$

$$Graphe2 : "01021323", "12100323", "13202103", "10203132"$$

On peut donc voir que la signatures des graphes, il existe au moins une signature identique. Le second graphe est donc rejeté par l'algorithme et ne sera pas utilisé.

Les deux graphes étudiés dans l'exemple précédent sont quasiment identiques, il est très simple d'observer leur isomorphisme visuellement. On peut montrer un autre exemple d'invalidation par isomorphisme avec un graphe plus différent.

Exemple 4 *On peut exposer un troisième exemple de graphe :*



On peut observer que le graphe est tout d'abord différent visuellement du graphe 2. On rappelle que les signatures du graphe cité, sont :

"01021323","12100323","13202103","10203132"

Établissons les signatures du graphe 4. On aura donc les signatures suivantes :

"01021323","12100323","13202103","10203132"

On peut observer qu'il existe au moins une signature en commun entre les deux graphes, on peut donc dire que les graphes sont isomorphes et que le graphe 4 est rejeté.

Après avoir sélectionné les graphes, on va pouvoir les transformer en chaînes de Markov et les étudier plus en profondeur.

5 Chaînes de Markov

Après avoir généré tous les graphes, nous associons des probabilités aux arêtes pour les transformer en chaînes de Markov. Tout d'abord nous devons établir quelques définitions avant de continuer. On peut tout d'abord définir une chaîne de Markov comme étant un processus qui peut effectuer chaque prédiction en ne se basant que sur l'état courant. Ces prédictions sont aussi bonnes que des prédictions réalisées en connaissant tout l'historique. [6]

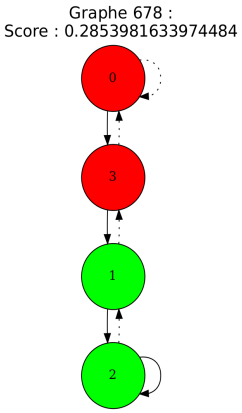
Définition 1 *Etant donné une chaîne de Markov défini sur l'ensemble d'états E, alors le nombre* $\mathbb{P}(X_1 = j \mid X_0 = i)$ *est appelé probabilité de transition de l'état i à l'état j en un pas, ou bien probabilité de transition de l'état i à l'état j, s'il n'y a pas d'ambiguïté.*
On note souvent ce nombre $\pi_{i,j}$ *:* $\pi_{i,j} = \mathbb{P}(X_1 = j \mid X_0 = i)$.
La famille de nombres $P = (\pi_{i,j})_{(i,j)\in E^2}$ *est appelée matrice de transition.*
Le graphe G d'une chaîne de Markov est un graphe orienté défini à partir de l'espace d'états E et de la matrice de transition $P = (\Pi_{i,j})$ *avec* $(i, j) \in E^2$ *de cette chaîne de Markov :*
- les sommets de G sont les éléments de E,
- les arêtes de G sont les couples $(i, j) \in E^2$ *vérifiant* $\Pi_{i,j} > 0$.

Dans notre cas, on observe des instructions conditionnelles, dont la probabilité que la condition soit réalisée est p, tel que $p \in [0, 1]$. Cela représente la probabilité réelle d'aller d'un état au suivant en empruntant la branche "Taken". On peut donc établir que $\pi_{i,j}$ pourra avoir comme valeur :
- $\pi_{i,j} = p$, pour les arêtes "Taken".
- $\pi_{i,j} = 1 - p$, pour les arêtes "Not Taken".
On peut en profiter pour établir la proposition suivante qui nous sera utile lors de l'utilisation de la matrice de transition pour les calculs futurs :

Proposition 1 *La matrice de transition* $P = (p_{i,j})_{(i,j)\in E^2}$ *est stochastique : la somme des termes de n'importe quelle ligne de P donne toujours 1.*

On considérera donc que ces graphes de chaînes de Markov sont représentés par des matrices de transition à l'état initial 0.

En ayant établi ces matrices et les probabilités de transitions possibles, on pourra calculer les probabilités stationnaires θ_i pour chaque état $i \in E$, lié au graphe G.



Exemple 5 *Servons nous donc du graphe ci-dessus, pour donner un exemple de la matrice d'adjacence que l'on obtiendrait :*

$$P = \begin{pmatrix} 1-p & 0 & 0 & p \\ 0 & 0 & p & 1-p \\ 0 & 1-p & p & 0 \\ 1-p & p & 0 & 0 \end{pmatrix}$$

6 Calcul de probabilités stationnaires

Les probabilités stationnaires θ_i pour chaque état $i \in E$, lié au graphe G représentent la fraction du temps passé en chaque état i, asymptotiquement. Il existe deux façons de calculer des probabilités stationnaires. On peut soit étudier la probabilité de se retrouver dans chaque état après chaque changement :
En considérant $X^{(n)}$, le vecteur de probabilité. La i-ème composante désigne la probabilité d'être dans l'état i à l'instant n, pour $n \in \mathbb{N}$. On définit k un état précédent, soit $k \in [0, n]$:

$$X^{(n)} = X^{(n-k)} P^k$$

Exemple 6 *En reprenant l'exemple ci-dessus (5), on peut choisir X, telle que l'on commence à l'état 0. On aura donc :*

$$X^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$$

On pourra donc calculer les probabilités d'être dans les prochains états à la prochaine prise de branche :

$$\begin{aligned} X^{(1)} &= X^{(0)} P = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1-p & 0 & 0 & p \\ 0 & 0 & p & 1-p \\ 0 & 1-p & p & 0 \\ 1-p & p & 0 & 0 \end{pmatrix} \\ \Leftrightarrow \quad X^{(1)} &= \begin{pmatrix} 1-p & 0 & 0 & p \end{pmatrix} \end{aligned}$$

On a donc calculé pour chaque état i, la probabilité de se retrouver à l'intérieur, en suivant une arête du graphe.

On peut considérer avec la loi des grands nombres que l'on peut établir, tel qu'en considérant le vecteur X et la variable n établis précédemment :

$$q = \lim_{n \to \infty} X^{(n)}$$

Considérons donc la matrice P (établie dans la section précédente), tel qu'il existe un vecteur q qui peut vérifier l'égalité suivante :

$$qP = q$$

Introduisons la matrice identité I, qui nous permettra d'établir l'égalité suivante :

$$\begin{aligned} &\Leftrightarrow \quad qP = qI \\ &\Leftrightarrow \quad q(I - P) = 0 \end{aligned}$$

Exemple 7 *En reprenant l'exemple ci-dessus, on peut établir la matrice q, telle que :*

$$q = \begin{pmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{pmatrix}$$

On aura donc :

$$\begin{aligned} q(I-P) &= \begin{pmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{pmatrix} \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 1-p & 0 & 0 & p \\ 0 & 0 & p & 1-p \\ 0 & 1-p & p & 0 \\ 1-p & p & 0 & 0 \end{bmatrix} \right) \\ &\Leftrightarrow \begin{pmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{pmatrix} \begin{pmatrix} p & 0 & 0 & -p \\ 0 & 1 & -p & p-1 \\ 0 & p-1 & 1-p & 0 \\ p-1 & -p & 0 & 1 \end{pmatrix} \\ &\Leftrightarrow \begin{pmatrix} (-1+p)\theta_3 + p\theta_0 & -p\theta_3 + \theta_1 + (-1+p)\theta_2 & -p\theta_1 + (1-p)\theta_2 & \theta_3 - p\theta_0 + (-1+p)\theta_1 \end{pmatrix} \end{aligned}$$

On obtiendra donc le système d'équation suivant :

$$\begin{cases} (-1+p)\theta_3 + p\theta_0 = 0 \\ -p\theta_3 + \theta_1 + (-1+p)\theta_2 = 0 \\ -p\theta_1 + (1-p)\theta_2 = 0 \\ \theta_3 - p\theta_0 + (-1+p)\theta_1 = 0 \\ \theta_0 + \theta_1 + \theta_2 + \theta_3 = 1 \end{cases}$$

Ce qui établira les prochaines égalités :

$$\begin{cases} \theta_0 = \frac{1-3p+3p^2-p^3}{1-2p+2p^2} \\ \theta_1 = \frac{p^2-p^3}{1-2p+2p^2} \\ \theta_2 = \frac{p^3}{1-2p+2p^2} \\ \theta_3 = \frac{(-1+p)^2 p}{1-2p+2p^2} \end{cases}$$

On a donc obtenu des valeurs pour chaque état de leur probabilité stationnaire. Cela nous permet donc de savoir la probabilité que l'on a de se retrouver dans l'état i après les n premiers pas dans la chaîne de Markov. On pourra donc, en fonction de ces valeurs, facilement déduire si les états doivent être "Taken" ou "Not Taken".

Plus précisément, pour déterminer le statut des états, on appliquera les calculs d'intégrales suivants, pour tous les états π_i du graphe donné :

$$\begin{aligned} x &= \int_0^1 (\theta_i * (1-p))\, dp \\ y &= \int_0^1 (\theta_i * p)\, dp \end{aligned}$$

Le x équivaut à prendre l'arête "Not Taken" pour toutes les valeurs de p possibles depuis l'état i. Le y équivaut à prendre l'arête "Taken" pour toutes les valeurs de p possibles depuis l'état i. On choisira la valeur inférieure entre les deux intégrales calculées. Cela représentera donc la prise de branche qui diminuera au maximum l'erreur de prédictions possible pour l'état. On pourra ensuite étiqueter chaque état comme étant "Taken" ou "Not Taken".

Exemple 8 *En reprenant les calculs effectués précédemment, on obtient :*
Les calculs d'intégrales de π_0 :

$$\begin{aligned} x &= \int_0^1 (\frac{1-3p+3p^2-p^3}{1-2p+2p^2} * (1-p))\, dp = 0.27400 \\ y &= \int_0^1 (\frac{1-3p+3p^2-p^3}{1-2p+2p^2} * p)\, dp = 0.08333 \end{aligned}$$

Comme $x \geq y$, alors l'état est "Not Taken" car cela minimise la probabilité d'erreur de prédiction.
Les calculs d'intégrales de π_1 :

$$\begin{aligned} x &= \int_0^1 (\frac{p^2-p^3}{1-2p+2p^2} * (1-p))\, dp = 0.05937 \\ y &= \int_0^1 (\frac{p^2-p^3}{1-2p+2p^2} * p)\, dp = 0.08333 \end{aligned}$$

Comme $x \leq y$, alors l'état est "Taken".
Les calculs d'intégrales de π_2 :

$$\begin{aligned} x &= \int_0^1 (\frac{p^3}{1-2p+2p^2} * (1-p))\, dp = 0.08333 \\ y &= \int_0^1 (\frac{p^3}{1-2p+2p^2} * p)\, dp = 0.27400 \end{aligned}$$

Comme $x \leq y$, alors l'état est "Taken".
Les calculs d'intégrales de π_3 :

$$\begin{aligned} x &= \int_0^1 (\frac{(-1+p)^2 p}{1-2p+2p^2} * (1-p))\, dp = 0.08333 \\ y &= \int_0^1 (\frac{(-1+p)^2 p}{1-2p+2p^2} * p)\, dp = 0.05937 \end{aligned}$$

Comme $x \geq y$, alors l'état est "Not Taken".

Le "score" des graphes représente le taux d'erreur de prédiction d'un graphe. Pour le calculer, on additionne juste les résultats obtenus :

Exemple 9 *On aura donc, pour ce graphe :*

$$score = 0.08333 + 0.05937 + 0.08333 + 0.05937 = 0.2854$$

Grâce à ces scores, nous avons pu en déduire des résultats et ainsi observer différents graphes optimaux.

7 Résultats

En effet, après avoir réalisé ces différents calculs, le programme est capable de réaliser un classement des prédicteurs, ordonné du plus optimal à celui qui réalise le plus d'erreur de prédictions. Le programme a un temps d'exécution de 85 secondes pour $k = 4$. La génération des graphes prend une partie majeure de cette durée, mais permet de rentrer des valeurs de K plus importantes. Il faut ainsi donner au programme un k, tel que $k \in \mathbb{N}$ et qui représentera le nombre d'états des graphes liés aux prédicteurs. On se concentrera ici sur les prédicteurs à 2-bits, soit à $k = 4$ états. Plus précisément, nous observerons les quatres prédicteurs les plus optimaux. Pour rappel, les numérotations d'états n'ont que peu d'importance. On peut complètement modifier cette numérotation sans provoquer de réels changements dans les prédicteurs. On s'en servira uniquement pour donner plus de clarté dans les observations.

Pour commencer, le prédicteur qui effectuera le moins d'erreurs de prédiction est le compteur saturé.

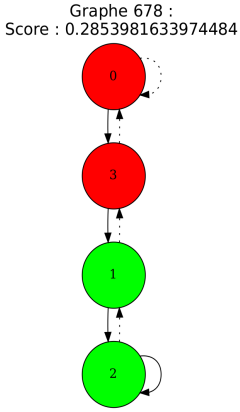


Figure 4: Compteur saturé

On peut facilement observer que le graphe représenté est identique au schéma que l'on peut observer ici, 3. Ce prédicteur fonctionne de façon optimale dans la plupart des situations.

Le second prédicteur est un mélange entre le compteur saturé et le flip-on-consecutive.

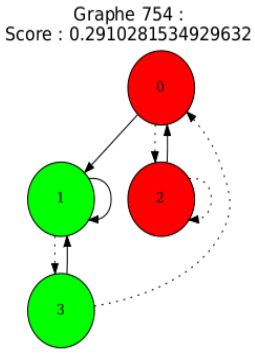


Figure 5: Hybride entre compteur saturé et flip-on-consecutive

Nous pouvons voir que la différence majeure avec le compteur saturé se fait dans les transitions entre états sans boucle. en effet, on peut même le résumer à une seule branche "Taken" de l'état "Not Taken" (l'état 0) vers l'état "Strongly Taken" (l'état 1). Ce graphe aura l'avantage de facilement transiter vers l'état "Strongly Taken" si des branches "Taken" sont souvent prises de suite.

Le troisième prédicteur est relativement identique au précédent.

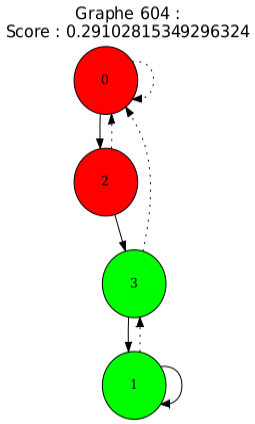


Figure 6: Second hybride entre compteur saturé et flip-on-consecutive

La différence majeure entre le second prédicteur et celui que l'on étudie actuellement est exactement l'inverse de ce que l'on a pu observer précédemment. En effet, On pourra le résumer à la branche "Not Taken" de l'état "Taken"

(l'état 3) vers l'état "Strongly Not Taken" (l'état 0). Ce graphe aura l'avantage de facilement transiter vers l'état "Strongly Not Taken" si des branches "Not Taken" sont souvent prises de suite.

Le dernier prédicteur est le flip-on-consecutive.

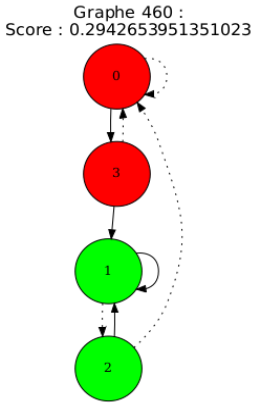


Figure 7: Flip-on-consecutive

Ce prédicteur aura l'avantage de permettre de facilement changer de type d'états entre des états "Taken" et "Not Taken" et de ne pas rester bloquer comme l'on pourrait l'être dans un compteur saturé. Il peut donc être plus optimal dans certaines situations où les résultats des instructions conditionnelles ne se suivent pas obligatoirement. Il serait intéressant de simuler l'exécution de divers algorithmes et observer le prédicteur le plus adapté à résoudre ce type d'algorithme.

8 Simulateur

Cet exercice est complémentaire à l'étude des prédicteurs de branchements et à leur efficacité. Après avoir établi les prédicteurs possédant les taux d'erreur de prédictions les plus faibles, tels que le compteur saturé ou bien encore le flip-on-consecutive, il est assez naturel de vouloir les observer en action. On pourra ainsi vérifier qu'ils sont assez performants lors de l'exécution de divers algorithmes que l'on aura pu choisir au préalable.
Pour simplifier la tâche de la personne qui voudrait réaliser ces tests, un programme a été développé. Le but de ce programme est donc de pouvoir implémenter n'importe quel algorithme et de facilement rajouter l'utilisation d'un prédicteur choisi et ainsi pouvoir simuler les résultats que l'on pourrait obtenir avec.
Nous allons donc en premier lieu détailler la façon dont les classes fonctionnent et peuvent être utilisés pour implémenter d'autres prédicteurs dans d'autres algorithmes. On pourra ensuite observer les résultats établis dans l'exemple d'implémentation donné.

8.1 Fonctionnement des classes du simulateur

Comme expliqué précédemment, nous avons souhaité faciliter la mise en place d'expériences sur les prédicteurs en pouvant facilement les simuler lors de l'exécution d'algorithmes. Pour cela, diverses méthodes ont été étudiées et développées. Nous nous concentrerons principalement sur la plus optimale, puis l'on expliquera l'évolution de pensée qui aura menée à cette solution.
Les classes ont été développées en Python pour simplifier leur utilisation dans le développement des futures expériences qui pourraient être réalisées. Deux classes ont donc été réalisées pour ce simulateur.
En premier lieu, on peut observer la classe "State". Cette classe permet de représenter un des états du prédicteur. On se sert de cette classe comme d'un objet contenant diverses informations. On pourra retrouver le champ state qui contient le numéro de l'état, ou bien encore le champ taken_state qui contient un booléen permettant de savoir si l'état est "Taken" ou "Not Taken". Le champ not_taken (resp. taken) va contenir un nombre entier permettant de désigner l'état dans lequel se dirige l'arête sortante not_taken (resp. taken).

Désormais, nous pouvons représenter les états des prédicteurs. Pour nous simplifier la tâche, il faut pouvoir représenter et se servir des prédicteurs dans l'exécution des instructions conditionnelles que l'on pourrait rencontrer lors de la mise en place des algorithmes. Pour cela, une seconde classe a été développée, la classe Graph.
Cette seconde classe est donc un objet possédant des champs représentants le nombre d'états, l'état dans lequel se trouve le prédicteur à l'étape actuelle de l'exécution, la liste des états dans le graphe et le nombre d'erreur de prédiction effectuée par le prédicteur. Cette classe possède une seule fonction qui permet d'aider à l'implémentation des prédicteurs dans les algorithmes.

La fonction evolving_graph permet de faire évoluer le graphe à chaque rencontre d'une instruction conditionnelle. Cette fonction vérifie si l'état courant est "Taken" en se servant de la fonction check_current_state puis observera si la comparaison est bien "Taken" ou non. En fonction de la réponse, si l'état est bien "Taken" (resp. "Not Taken") et que la comparaison aussi, alors le graphe ne fait qu'évoluer en se servant de la fonction running_graph. Sinon, le graphe évolue tout de même, mais l'on compte une erreur de prédiction en plus.
Pour aider à faciliter l'utilisation, trois fonctions permettant de reproduire les graphes les plus utilisés ont été implémentées : la fonction create_tbsc (pour créer le compteur saturé à 2 bits), la fonction create_obsc (pour créer le compteur saturé à 2 bits), et la fonction create_swapped (pour créer le flip-on-consecutive). Nous possédons maintenant toutes les classes nécessaires pour implémenter des prédicteurs dans des algorithmes. Nous allons donc expliquer la façon de les intégrer.

Pour cela, implémenter l'algorithme d'une façon classique. Par la suite, en fonction de votre souhait, créer les prédicteurs que vous souhaitez, au nombre que vous le souhaitez, pour reproduire le type de prédicteur local ou global qui vous convient. Il vous suffira désormais de modifier les instruction conditionnelles en appelant la fonction evolving_graph et en donnant comme paramètre à la fonction, la condition préalablement vérifiée. Vous voici avec un algorithme dont les prédicteurs sont simulés de la façon dont vous le souhaitez.

D'autres méthodes avaient été imaginées au préalable, mais n'ont pas été retenues car elles modifiaient fondamentalement les algorithmes dans lesquels on voulait implémenter les prédicteurs en y rajoutant des instructions conditionnelles. Vous pouvez observer les anciennes versions dans le GitHub donné en lien sur la page de garde.

8.2 Observation des exemples mis en place

Pour montrer la façon dont la mise en place de la simulation doit être effectuée et pour vérifier quelques résultats établis dans le document [2], des exemples ont été produits. Dans ce document, nous pouvons retrouver deux parties qui nous seront utiles pour observer si les exemples sont corrects. On peut tout d'abord observer la citation suivante (dans une version traduite) :

Citation 1 *Dans les paramètres classiques de l'analyse, l'algorithme* $\frac{3}{2}$*-MinMax est optimal, le nombre de comparaison effectué est asymptotiquement équivalent à* $\frac{3}{2}n$*. De toute évidence, NaiveMinMax a besoin de* $2n - 2$ *comparaisons.*

Puis on peut aussi considérer la proposition suivante (traduite elle aussi) :

Proposition 2 *Le nombre d'erreur de prédiction attendu réalisé par NaiveMinMax, pour une distribution uniforme de tableaux de taille n, est asymptotiquement équivalent à 4logn pour le prédicteur à 1 bit et à 2log(n) pour les deux prédicteurs à 2 bits et le compteur saturé à 3-bits. Le nombre d'erreur de prédiction attendu réalisé par $\frac{3}{2}$-MinMax est asymptotiquement équivalent à n/4logn pour tous les prédicteurs étudiés.*

Nous allons donc nous concentrer sur les deux algorithmes énoncés ici : NaiveMinMax et $\frac{3}{2}$-MinMax. Vous pouvez retrouver les pseudos-code dans le document cité ci-dessus. De plus, on utilisera le compteur saturé à 2-bits car il est celui dont on connaît le plus d'informations et dont on sait désormais qu'il est le plus optimal. Détenant les outils nécessaires à la construction des exemples, nous pouvons nous atteler à la tâche.
Pour le premier algorithme, on a donc créés deux compteurs saturés locaux, qui permettront d'avoir une prédiction des deux instructions conditionnelles complètement indépendantes entre elles. Pour le second algorithme, on a créé cinq compteurs saturés locaux afin que les prédictions soient toutes indépendantes. Pour générer les tableaux de taille n, on les remplit de façon aléatoire afin de ne pas impacter les résultats. Pour observer plus finement les résultats, on a testé les tableaux de taille 2 à 10002, en effectuant la moyenne des résultats sur des échantillons de 100 tableaux.
Pour chaque résultat, on compte le nombre de comparaisons effectuées et le nombre d'erreur de prédictions réalisées pendant l'exécution de l'algorithme. Afin de simplifier la lisibilité des résultats, on a utilisé le module Graphviz en Python qui nous permet de facilement représenter des graphiques avec les résultats obtenus.

Après exécution du programme, on obtient donc quatre graphiques que nous allons observer plus en détail. Tout d'abord, étudions ceux que l'on a obtenu et qui concerne l'algorithme NaiveMinMax :

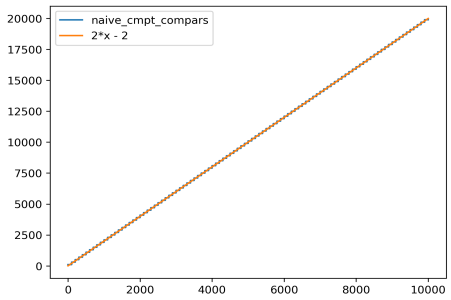


Figure 8: Comparaisons pour NaiveMinMax

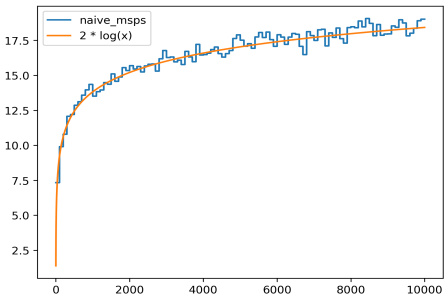


Figure 9: Erreurs de prédictions pour NaiveMinMax

Ces graphiques nous permettent d'observer assez facilement que les comparaisons de NaiveMinMax sont bien asymptotiquement équivalentes à $2n - 2$ et que les erreurs de prédictions sont aussi asymptotiquement équivalentes à $2log(n)$. L'implémentation des simulateurs semblent pour l'instant fonctionner correctement en vérifiant ces résultats.

On peut donc maintenant observer les résultats obtenus pour le second algorithme : $\frac{3}{2}$-MinMax.

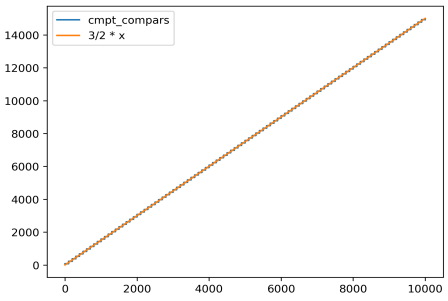


Figure 10: Comparaisons pour $\frac{3}{2}$-MinMax

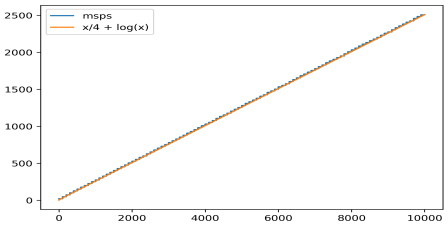


Figure 11: Erreurs de prédictions pour $\frac{3}{2}$-MinMax

Comme précédemment, on peut observer que le simulateur obtient des résultats quasiment identiques aux valeurs asymptotiques données précédemment, soit $\frac{3}{2}n$ comparaisons et $\frac{n}{4}log(n)$ erreurs de prédictions.
On peut donc en conclure que le simulateur et les classes liées, si elles sont bien utilisées, permettent de représenter efficacement l'utilisation de prédicteurs de branchement choisis.

9 conclusion

Pour conclure, on aura pu remarquer lors de l'étude des prédicteurs de branchements, que certains prédicteurs sont donc plus performants que d'autres, dans certains cas ou certaines situations. Le compteur saturé semble tout de même être l'un des plus performants dans un certain nombre de cas. La mise en place d'un simulateur aura permis d'observer des résultats concrets quand à l'application de prédicteur de branchements dans des algorithmes. En rendant plus performant le programme de génération des graphes, on pourrait obtenir des prédicteurs établis avec un plus grand nombre de bits et qui pourraient, peut-être, faire évoluer l'informatique moderne s'ils obtiennent des résultats plus intéressants en terme de taux d'erreur de prédictions. On pourrait tout aussi bien développer de nombreux autres exemples avec les classes produites pour le simulateur.

10 Résumé (Version française)

Dans le projet mené tout au long du stage, nous nous sommes intéressés aux unités de prédiction de branchement. Les branchements à prédire sont des déplacements dans le code effectués par le processeur, qui sont souvent représentés par des instructions conditionnelles. Ces unités permettent de rendre l'utilisation du pipeline du processeur plus efficace, en prédisant le résultat de la question suivante, pour chaque instruction conditionnelle : est-ce que le branchement observé sera pris ou non ? Il est intéressant de se demander à quoi ressemblent ces unités et comment fonctionnent elles. Tout d'abord, on souhaitera observer toutes les unités différentes et existantes, puis on cherchera assez rapidement à identifier la ou les plus performantes. Il faudra donc effectuer des calculs de probabilités stationnaires pour obtenir un classement entre les différentes unités et pouvoir ainsi affirmer que le compteur saturé est le plus performant dans la plupart des situations. Une fois que l'on a obtenu ces résultats, on pourra les observer un peu plus en détail pour remarquer les légères différences entre les premiers résultats du classement. Nous essayerons, par la suite, de les mettre en place dans un simulateur afin de les étudier dans des situations que l'on veut proches de la réalité. Le simulateur permettra ainsi d'insérer les unités de prédiction voulues dans des algorithmes et plus précisément dans chaque instruction conditionnelle, sans modifier la structure de ces dits-algorithmes. Finalement, les résultats observés permettront de mettre en lumière la proximité entre les valeurs théoriques et simulées, et ainsi aider à la compréhension des résultats du classement.

11 Résumé (English version)

In the project conducted throughout the internship, we were interested in branch prediction units. The branches to be predicted are moves in the code made by the processor, which are often represented by conditional instructions. These units make the use of the processor pipeline more efficient, by predicting the outcome of the following question, for each conditional instruction: will the observed branch be taken or not? It is interesting to ask what these units look like and how they work. First of all, we will want to observe all the different existing units, and then we will try to identify the most efficient one(s) quite quickly. It will then be necessary to perform stationary probability calculations to obtain a ranking between the different units and to be able to affirm that the saturated counter is the most efficient in most situations. Once we have obtained these results, we can observe them in a little more detail to notice the slight differences between the first results of the ranking. We will then try to implement them in a simulator in order to study them in situations that we want to be close to reality. The simulator will thus allow to insert the desired prediction units in algorithms and more precisely in each conditional instruction, without modifying the structure of these said algorithms. Finally, the observed results will allow to highlight the proximity between the theoretical and simulated values, and thus help to understand the results of the classification.

References

[1] *Algorithme de Tarjan*. URL: https://fr.wikipedia.org/wiki/Algorithme_de_Tarjan. (Visité: 03.07.2021).

[2] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. "Good predictions are worth a few comparisons". In: *STACS'16*. 2016. http://www-igm.univ-mlv.fr/~nicaud/articles/stacs16.pdf.

[3] *Chaîne de Markov*. URL: https://fr.wikipedia.org/wiki/Cha%C3%AEne_de_Markov. (Visité: 30.06.2021).

[4] *Graphe d'une chaîne de Markov*. URL: https://fr.wikipedia.org/wiki/Graphe_d%27une_cha%C3%AEne_de_Markov_et_classification_des_%C3%A9tats. (Visité: 30.06.2021).

[5] John L. Hennessy and David A. Patterson. *Computer Architecture : A quantitave approach (Fifth Edition)*. Morgan Kauffman, 2011.

[6] *Markov Chain*. URL: https://en.wikipedia.org/wiki/Markov_chain. (Visité: 03.07.2021).

12 Annexes

Voici le lien vers le répertoire GitHub contenant toutes les ressources liées au développement du générateur d'unités de prédiction de branchement et du simulateur : https://github.com/TomRedLev/stage$_u$ge$_2$021