Exploitation de la prédiction de branchement

Sommaire

Dans ce chapitre, nous allons nous intéresser à l'influence de la prédiction de branchement sur les performances de divers algorithmes. Rappelons que le sujet de la prédiction de branchements relève du domaine de l'architecture qui a été abordé au cours du chapitre 3. Il s'agit également d'une première occasion pour nous d'affiner le modèle RAM qui fait des simplifications concernant le coût des instructions de branchement. Nous allons, en plus des coûts d'opérations basiques comme les comparaisons, nous intéresser à un nouveau type de coût, à savoir le nombre d'erreurs de prédiction. Nous distinguons donc de cette façon les coûts des opérations classiques du modèle RAM et les coûts issus de la prédiction de branchement. Le sujet de la prédiction de branchement a déjà été brièvement étudié par le passé mais principalement d'un point de vue pratique. Un premier travail expérimental est celui de Biggar et ses co-auteurs [12] qui ont mis en avant l'influence de la prédiction de branchements sur divers algorithmes de tri. Pour obtenir leurs résultats, les auteurs ont utilisé la librairie PAPI [2] que nous utilisons également pour nos expérimentations. Des analyses théoriques d'algorithmes de tri sur le nombre d'erreurs de prédiction ont également été effectuées par Brodal, Fagerberg et Moruz. Ces derniers ont cherché à étudier les compromis entre le nombre de comparaisons et le nombre d'erreurs de prédiction pour le problème du tri [16]. À l'aide d'utilisations d'arbres de décision comme ceux que nous avons vu au cours du chapitre précédent, ils montrent par exemple que si un algorithme trie une séquence de taille n en $\mathcal{O}(dn \log n)$ comparaisons, alors le nombre d'erreurs de prédiction est $\Omega(n \log_d n)$. Ce théorème est vrai quel que soit le modèle de prédicteur utilisé. Ces mêmes auteurs [14] ont également étudié l'influence du nombre d'inversions sur le nombre d'erreurs de prédiction générées à l'exécution de QuickSort. Ces travaux sont à notre connaissance les premiers pour lesquels une analyse théorique a été effectuée dans le cadre de l'utilisation de prédicteurs statiques. Sanders et Winkel [40] ont quant à eux proposé une variante de l'algorithme de tri SampleSort qui dissocie les comparaisons de la prédiction par l'utilisation d'instructions spécifiques au microprocesseur. Ces instructions peuvent s'apparenter à des instructions de mouvement conditionnelle *CMOV*. Similairement, Elmasry, Katajainen et Stenmark ont proposé une variante de MergeSort[22] qui essaye de contourner au maximum l'utilisation des instructions de branchements.

Kaligosi et Sanders [28] ont publié des travaux sur une analyse poussée du nombre d'erreurs de prédiction produites en moyenne par des prédicteurs dynamiques locaux. Cette étude a été faite dans le cadre de l'étude de QuickSort. Dans ce papier, les auteurs exploitent le fait que ces prédicteurs vont se comporter comme une chaîne de Markov avec de bonnes propriétés qui convergent rapidement vers une unique distribution stationnaire. Par la suite, dans ce chapitre, nous utilisons des raisonnements similaires. Au moment où sont écrites ces lignes, la dernière version de Java utilise en plus de l'algorithme TimSort, une version de QuickSort a deux pivots qui, contre toutes attentes, a de bonnes performances en pratique. Les travaux de Martinez, Nebel et Wild [34] ont cependant montré que l'influence de la prédiction de branchements n'est pas suffisante pour expliquer ces bonnes performances.

Brodal et Moruz ont conduit une étude expérimentale [17] sur des arbres binaires de recherches déséquilibrés, mettant en évidence qu'il était possible pour des structures non équilibrées d'être plus efficaces en pratique par rapport au cas où ces structures sont bien équilibrées. Il s'agit ici d'un des premiers travaux à notre connaissance qui essaye de guider les prédicteurs de branchement. Notre

contribution s'approche de ces travaux mais se concentre sur les algorithmes plutôt que sur la structure des données. Nos travaux ont été publiés dans la conférence STACS 2016 [10]. Ce chapitre présente une version détaillée des travaux qui ont été publiés. Les algorithmes étudiés sont principalement composés de quelques instructions de branchement qui sont exécutées à plusieurs reprises par l'action d'une boucle. C'est par exemple le cas de l'exponentiation rapide qui permet de calculer pour un réel x et un entier naturel n la valeur x^n. En plus de fournir une analyse de ces algorithmes sur le nombre d'erreurs de prédiction, nous avons proposé des modifications peu coûteuses de ces derniers afin de guider la prédiction de branchement. Une contribution que nous avons apportée avec ces travaux concerne la prédiction globale. Nous avons proposé une analyse du nombre d'erreurs de prédiction pour la recherche dichotomique dans le cas où la prédiction est effectuée par un prédicteur global. Nous y sommes arrivés en exploitant des propriétés de l'algorithme, mais il s'agit à notre connaissance du premier cas où une analyse a été faite sur ce type de prédicteur.

Dans nos travaux, nous confirmons nos résultats théoriques par des résultats expérimentaux. Bien qu'il est possible de démontrer que pour un certain type de prédicteur, nous gagnons en nombre d'erreurs de prédiction, il est difficile de conclure que l'implantation sur une machine réelle sera efficace. Les spécificités des architectures dépendent des choix des constructeurs de processeurs qui ne communiquent, pour des raisons de secret industriel, que très peu sur l'utilisation d'un prédicteur particulier. Nous ne pouvons que faire des suppositions en effectuant des benchmarks (comme ce qui a été fait par exemple ici [3]) . Comme nous ne savons pas exactement combien une erreur de prédiction coûte réellement par rapport à une comparaison, il n'est pas non plus évident qu'un algorithme A qui fait plus de comparaisons et moins d'erreurs de prédiction qu'un algorithme B sera plus ou moins efficace en pratique. Il y a bien des compromis pour ces cas, mais il n'est possible d'être sûr que l'algorithme A est plus efficace en temps d'exécution que l'algorithme B sur une machine arbitraire qu'après avoir vérifié expérimentalement que c'était bien le cas. Cependant, nous pensons que l'analyse théorique pour les prédicteurs étudiés nous permet de mettre en évidence qu'il peut y avoir un compromis à trouver en pratique. Les modèles de prédicteurs étudiés sont parmi les plus basiques et les derniers processeurs utilisent des prédicteurs qui parviennent à être encore plus performants que ces derniers. On a donc espoir qu'un algorithme qui se comporte bien en théorie pour ces prédicteurs a des chances d'être encore plus performant en pratique sur un processeur suffisamment récent.

Ce chapitre détaille l'ensemble des travaux cités plus haut. Nous proposons également des pistes pour généraliser les méthodes que nous avons employées pour obtenir les variantes des différents algorithmes que nous étudions dans ce chapitre. Le but est de proposer une abstraction de leurs propriétés intrinsèques. L'exploitation de ces propriétés nous permet aisément de modifier nos algorithmes. Ces modifications peuvent être intéressantes pour guider le prédicteur de branchements.

5.1 Recherche simultanée du minimum et du maximum

Nous avons vu dans le chapitre 3 le problème de la recherche du minimum et du maximum dans une séquence. Pour rappel, nous avons en entrée une séquence d'éléments comparables stockée dans un tableau T. Le problème consiste à obtenir les éléments minimum et maximum de ce tableau. Par exemple, si $T = \langle 7, 5, 2, 4, 14, 9 \rangle$, la sortie sera donnée par le couple $(2, 14)$ car 2 et 14 sont respectivement le minimum et le maximum de T.

5.1.1 Un premier algorithme naïf

Données : Une séquence T de taille n.
Résultat : $\min(T)$, $\max(T)$

```
1 min ← max ← T[1]
2 pour i ← 2 à n faire
3 |   a ← T[i]
4 |   si a < min alors
5 |   |   min ← a
6 |   si a > max alors
7 |   |   max ← a
```

Algorithme 8 : Algorithme de recherche du minimum et du maximum par la méthode naïve. Par la suite, nous nommons cet algorithme NaiveMinmax.

Une première méthode que l'on nomme NaiveMinmax est décrite par l'algorithme 8. L'algorithme procède en faisant un parcours de tous les éléments de la séquence. Le minimum et le maximum des éléments parcourus sont gardés en mémoire. Lorsque tous les éléments sont parcourus, nous obtenons ainsi le minimum et le maximum qui correspondent aux valeurs gardées en mémoire. Nous initialisons à la ligne 1 le minimum et le maximum par la valeur $T[1]$ qui est forcément le minimum et le maximum des éléments parcourus. Le parcours des éléments se fait par la boucle ligne 2 en partant du deuxième et en allant jusqu'au dernier. L'élément que l'on est en train de parcourir est alors comparé aux deux éléments gardés en mémoire. Si l'élément est le plus petit rencontré jusqu'à présent, il faut alors mettre à jour le minimum que l'on a gardé en mémoire. De la même façon, il faut mettre à jour le maximum si l'élément est le plus grand. Comme l'algorithme consiste en un balayage linéaire des $n-1$ éléments restants après l'initialisation, et qu'il fait deux comparaisons par itération, le coût total de l'algorithme est $2(n-1)$. La Figure 5.1 montre un exemple d'exécution de l'algorithme pour une entrée choisie arbitrairement.

5.1.2 Un algorithme optimisé pour le nombre de comparaisons

Nous avons vu lors du chapitre 3, un argument d'adversaire qui nous donne une borne inférieure au problème de la recherche du minimum et du maximum qui est de $\frac{3n}{2}$. Nous allons à présent voir un algorithme qui atteint cette borne

FIGURE 5.1 – Cette figure représente une exécution de l'algorithme de recherche naïve du minimum et du maximum pour la séquence d'entrée $\langle 5, 6, 1, 7, 3, 2, 9, 8, 4\rangle$. À l'initialisation à la première ligne, le premier élément, dont la case est remplie en jaune, est à la fois le minimum et le maximum temporaires. Les lignes suivantes représentent les itérations correspondant à la boucle de parcours des éléments. Pour chacune de ces lignes, l'élément dont la case est remplie en rouge est le maximum temporaire, l'élément dont la case est remplie en vert est le minimum temporaire, les cases en bleu ne sont, ni le maximum ni le minimum et ont déjà été parcourues aux itérations précédentes.

et qui, par conséquence, est optimal. Cet algorithme est une variante de l'algorithme naïf vu précédemment. Nous l'appelons $\frac{3}{2}$-MINMAX et il est décrit par l'Algorithme 9. L'idée consiste toujours à garder en mémoire le minimum et le maximum des éléments déjà explorés, mais la façon dont nous parcourons ces éléments est différente. Dans l'algorithme naïf précédent, nous parcourions chaque élément un par un, cette fois nous allons parcourir deux éléments x et y simultanément par itération de la boucle principale. Soit m et M respectivement les éléments minimum et maximum que l'on connaît de la séquence à une itération. Nous faisons une première comparaison entre x et y. Si $x < y$, nous savons alors que x ne peut pas être le maximum de la séquence et il n'est donc pas nécessaire de comparer x à M. Respectivement, y ne peut pas être le minimum de la séquence et il n'est donc pas nécessaire de comparer y à m. Il nous reste alors deux comparaisons à faire, la première entre x et m, la deuxième entre y et M. Si $x < m$ alors nous affectons x à m et respectivement si $y > M$ nous affectons y à M. Dans le cas où $x > y$, nous faisons les mêmes remarques et les mêmes comparaisons supplémentaires en remplaçant x par y et y par x. Pour résumer, pour chaque itération, nous faisons une première comparaison entre x et y, une comparaisons entre $\min(x, y)$ et m, puis une comparaisons entre $\max(x, y)$ et M. Pour chaque itération nous faisons donc au total 3 comparaisons. À l'initialisa-

Données : Une séquence T de taille n.
Résultat : $\min(T)$, $\max(T)$

```
1  min ← max ← T[n]
2  pour i ← 1 à ⌊n/2⌋ faire
3      a ← T[2i − 1]
4      b ← T[2i]
5      si a < b alors
6          si a < min alors
7              min ← a
8          si b > max alors
9              max ← b
10     sinon
11         si b < min alors
12             min ← b
13         si a > max alors
14             max ← a
```

Algorithme 9 : Algorithme de recherche du minimum et du maximum par une méthode optimale. Par la suite nous nommons cet algorithme $\frac{3}{2}$-MINMAX.

tion, nous choisissons d'affecter $m = M = T[n]$, et nous parcourons les éléments de la séquence dans l'ordre avec un compteur i initialisé à 1 que nous incrémentons de 2 après chaque itération. Cette affectation nous permet de faire le même parcours sans avoir à considérer la parité de n. En contrepartie, si n est pair, nous parcourons deux fois l'élément $T[n]$. Dans tous les cas, le nombre d'itérations effectuées par le parcours est de $\lfloor \frac{n}{2} \rfloor$. Le nombre total de comparaisons effectuées par l'algorithme $\frac{3}{2}$-MINMAX est ainsi $3\lfloor \frac{n}{2} \rfloor = \frac{3n}{2} + \mathcal{O}(1)$. En comparaison avec NAIVEMINMAX, nous faisons de l'ordre de 25% de comparaisons en moins avec $\frac{3}{2}$-MINMAX. Nous donnons un exemple de trace d'exécution pour la même séquence d'entrée que pour l'exemple précédent dans la Figure 5.2.

5.1.3 Un résultat inattendu

Nous avons implanté ces deux algorithmes afin de pouvoir mesurer leurs performances en temps sur une machine. Les listings 8 et 9, disponibles en Annexe B, donnent des extraits du code source de nos implantations en langage C.

Nous avons ensuite compilé ce fichier avec le logiciel *gcc* sur un environnement *gnu/linux*. Nous avons utilisé l'option *-O0* qui permet de compiler le code source sans optimisation. Ce choix nous permet d'obtenir une version de notre code en langage machine qui devrait être assez proche de notre code source initial. La machine utilisée pour faire ces tests fonctionne avec un microprocesseur *Intel(R) Celeron(R) CPU 1037U @ 1.80GHz*. Le listing 10 de l'annexe B donne le code assembleur que l'on obtient sur cette machine après compilation du fichier avec les options *-O0* et *-S*. Ce dernier nous permet de vérifier la proximité qu'a le fichier exécutable avec le code source en C. Les résultats obtenus sur cette machine ne sont pas isolés. Des résultats similaires ont été obtenus sur

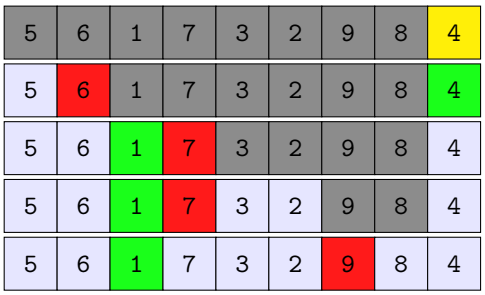

FIGURE 5.2 – Cette figure représente une exécution de l'algorithme de recherche optimisée du minimum et du maximum pour la séquence d'entrée $\langle 5, 6, 1, 7, 3, 2, 9, 8, 4\rangle$. À l'initialisation à la première ligne, le dernier élément, dont la case est remplie en jaune, est à la fois le minimum et le maximum temporaire. Les lignes suivantes représentent les itérations de la boucle principale. Pour chacune de ces lignes, l'élément dont la case est remplie en rouge est le maximum temporaire, l'élément dont la case est remplie en vert est le minimum temporaire, les cases en bleu ne sont, ni le maximum ni le minimum et ont déjà été parcourues aux itérations précédentes.

d'autres machines suffisamment récentes pour avoir des caractéristiques basées sur les dernières avancées en architecture.

Pour générer le tableau en entrée, nous le remplissons initialement de valeurs entières avec la fonction standard *random()* qui a un comportement proche d'une loi uniforme sur l'intervalle $[0, RAND\_MAX]$, avec $RAND\_MAX$ qui est une valeur particulière dépendant de l'environnement. [1] À chaque nouveau test, le tableau est ensuite mélangé avec un algorithme de type Knuth Shuffle afin d'obtenir un mélange uniforme. Le tableau obtenu est alors utilisé en entrée pour les deux algorithmes. Lorsque nous exécutons ce programme, nous obtenons sur la sortie standard les temps pris par les implantations de ces deux algorithmes pour des tableaux de tailles allant de 1000 à 1000×2^{18}, la taille étant doublée à chaque itération. Pour chaque taille, nous avons exécuté ces deux implantations 10 fois afin d'obtenir une mesure moyenne des temps. La Figure 5.3 montre les résultats ainsi obtenus sous forme de courbes de performances.

Lorsque nous observons ces deux courbes, nous pouvons être surpris du résultat. En effet, nous observons que la courbe de l'algorithme optimal en nombre de comparaisons est située en tous points au-dessus de la courbe de l'algorithme naïf. En d'autres termes, l'algorithme naïf s'avère être en pratique plus efficace que l'algorithme optimisé. Si nous considérons que notre machine est basée sur le modèle RAM, ce résultat est surprenant car les deux algorithmes font les mêmes opérations, exceptées pour les comparaisons. Or, nous avons vu que $\frac{3}{2}$-MINMAX fait moins de comparaisons que NAIVEMINMAX. Pour ces deux opérations l'algorithme optimisé fait moins de travail que l'algorithme naïf et devrait par conséquent être le plus rapide des deux. Il nous est donc nécessaire de remettre en question l'hypothèse du modèle RAM. Parmi les principales caractéristiques

1. Sur la machine où ont été réalisé les tests, la valeur de $RAND\_MAX$ est de $2^{31} - 1 \approx 2 \times 10^9$.

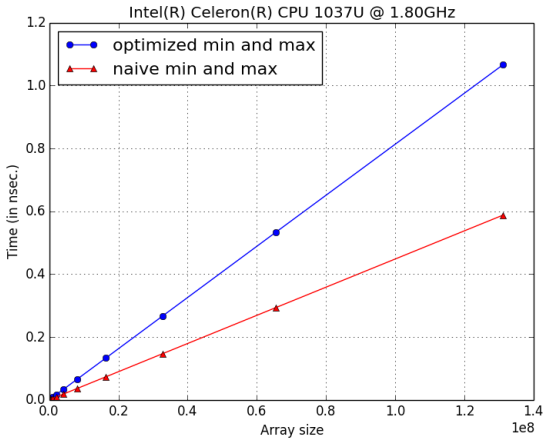

FIGURE 5.3 – Courbe de performances en temps des algorithmes de recherche du minimum et du maximum. L'abscisse correspond à la taille du tableau en entrée, l'ordonnée correspond à la durée moyenne d'exécution de l'algorithme sur 10 essais.

des machines récentes qui diffèrent de ce modèle, nous avons abordé lors du chapitre 3 la *hiérarchie mémoire* et la *prédiction de branchement*. Dans notre implantation, nous avons fait en sorte d'éliminer l'influence de la hiérarchie mémoire sur les performances. Nous faisons exactement un seul accès par élément du tableau qui est ensuite stocké dans des variables temporaires pour faire les comparaisons et les affectations qui suivent. De plus, ces accès se font exactement dans le même ordre. Il est cependant assez évident que la prédiction de branchement diffère grandement entre ces deux algorithmes. Notre intuition à ce sujet provient du fait que le premier test effectué par l'algorithme optimisé est imprédictible dans le cas où l'entrée est générée par une distribution uniforme. En effet, le test qui consiste à comparer deux éléments consécutifs dans le tableau a une probabilité de $\frac{1}{2}$ d'être vrai. Nous avons vérifié en utilisant la librairie PAPI le nombre d'erreurs de cache et d'erreurs de prédiction générées en moyenne par ces deux implantations. La méthode est relativement similaire à celle employée précédemment pour obtenir les performances de temps mais en utilisant des compteurs apportés par la librairie. Les compteurs que nous utilisons sont les mêmes que ceux utilisés par la commande *perf* qui fonctionne de manière similaire à la commande time. Nous n'utilisons pas ici cette commande car, comme pour la commande time, nous ne pouvons faire des mesures sur un bloc de code isolé dans un programme. Les implantations et la génération des entrées restent donc inchangées. Le code source modifié est disponible en Annexe B. Les optimisations au niveau hardware sur le cache font que l'influence du cache est négligeable quelle que soit la taille de l'entrée. Nous observons que le nombre d'erreurs de cache (au niveau L1) est de l'ordre de $10^{-3}\%$ de la taille du tableau en entrée. Nous donnons dans la Table 5.1, le nombre d'erreurs de prédiction générées à l'exécution de ces deux implantations en fonction de la taille de l'entrée. Nous observons une différence très importantes du nombre

Taille de l'entrée	#err. prédictions trois demi	#err. prédictions naïf
1000	263	11
2000	515	12
4000	1003	13
8000	2029	15
16000	3967	14
32000	8023	15
64000	16055	17
128000	32014	15
256000	63950	18
512000	128102	18
1024000	255842	20
2048000	512006	23
4096000	1024469	22
8192000	2047809	25
16384000	4095771	23
32768000	8191399	25
65536000	16382484	26
131072000	32766697	27

TABLE 5.1 – Nombre d'erreurs de prédiction mesuré pour les implantations des deux algorithmes de recherche du minimum et du maximum dans un tableau. Les résultats sont donnés en fonction de la taille de l'entrée.

d'erreurs de prédiction entre ces deux implantations. Sur les tailles que nous avons choisies, nous observons que l'influence de la prédiction de branchement est négligeable pour NAIVEMINMAX avec un nombre d'erreurs de prédiction qui est de l'ordre de $10^{-5}\%$ de la taille de l'entrée. On ne peut pas affirmer la même chose du côté de $\frac{3}{2}$-MINMAX qui fait en moyenne 25% de la taille de l'entrée en nombre d'erreurs de prédiction. Nous pouvons donc pour le moment conjecturer que le nombre d'erreurs de prédiction générées lors de l'exécution de $\frac{3}{2}$-MINMAX est $\frac{n}{4}$ avec n la taille de l'entrée pour le prédicteur utilisé par la machine. Nous allons voir par la suite que c'est bien le cas et que c'est vrai quel que soit le prédicteur utilisé. Nous verrons également des expressions du nombre d'erreurs de prédiction pour le cas de l'algorithme naïf. Nous avons plusieurs expressions car ces dernières dépendent du prédicteur utilisé.

5.1.4 Analyse du nombre d'erreurs de prédiction pour ces deux algorithmes

Nous allons désormais proposer une analyse de ces deux algorithmes en nombre d'erreurs de prédiction. Pour réaliser ce genre d'analyse, il est nécessaire de fixer le prédicteur que nous utilisons. Chaque prédicteur a un comportement différent et donc les erreurs qu'ils font le sont également. Il est donc fort probable que les prédicteurs utilisés dans la pratique se comportent différemment. Les prédicteurs que nous avons choisis pour cette section sont des prédicteurs locaux 1-bit, 2-bits et 3 bits saturés. Pour chaque instruction de branchement dans nos algorithmes, nous associons un prédicteur distinct qui s'occupe de pré-

dire uniquement la prochaine exécution de cette instruction. Afin de rester fidèle à l'expérimentation que nous avons décrite lors de la section précédente, nous considérons que l'entrée est une permutation sous la forme d'un mot de taille n. Cela correspond bien aux entrées de notre expérimentation en remplaçant la valeur de chaque élément par leurs rangs ce qui ne changent pas les résultats et l'ordre des comparaisons effectuées par nos deux algorithmes. La Proposition 15 donne un premier résultat sur le nombre d'erreurs de prédiction effectuées par NaiveMinmax. La preuve repose sur le fait que les comparaisons effectuées aux Lignes 4 et 6 sont en relation avec des statistiques connues et évoquées lors du Chapitre 2 les *min-records* et les *max-records*.

Proposition 15. *Le nombre moyen d'erreurs de prédiction effectuées par* NaiveMinmax*, pour une distribution uniforme sur les tableaux de taille n, est asymptotiquement équivalent à $4 \log n$ pour le prédicteur 1-bit et à $2 \log n$ pour les prédicteurs 2-bits et 3-bits saturés.*

Démonstration. Soit $\mathcal{F}$ la bijection fondamentale qui permet de faire une correspondance entre la représentation en cycle des permutations $\sigma \in \mathfrak{S}_n$ et les max-records de $\mathcal{F}(\sigma)$ vu au cours du Chapitre 2. Comme $\mathcal{F}$ est une bijection, alors pour une variable aléatoire ξ à valeurs réelles, l'identité

$$\mathbb{E}_n[\xi] = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \xi(\sigma) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \xi(\mathcal{F}(\sigma))$$

est vérifiée.

Pour commencer, nous allons nous intéresser au comportement du prédicteur 1-bit. Soit σ une permutation de $\mathfrak{S}_n$ dont les cycles, triés dans l'ordre croissant de leurs plus grands éléments, sont $C_1, \ldots, C_m$. La bijection fondamentale appliquée à σ est de la forme $\mathcal{F}(\sigma) = f(C_1) \cdot f(C_2) \ldots f(C_m)$, avec f la fonction qui concatène les éléments d'un cycle en commençant par le plus grand élément. Nous allons maintenant chercher la relation entre les cycles de σ et le nombre d'erreurs de prédiction de $\mathcal{F}(\sigma)$ au cours de l'exécution de la Ligne 6. Nous allons maintenant regarder en détail un cas particulier en fixant l'état initial du prédicteur à NT au début du parcours de $f(C_i)$. Comme le premier élément de $f(C_i)$ est un max-record par construction de $\mathcal{F}(\sigma)$, il provoque une erreur de prédiction et le prédicteur passe à l'état T. Si C_i contient au moins deux éléments, alors son deuxième élément dans $f(C_i)$ est plus petit que le premier, et va à son tour provoquer une erreur de prédiction et le prédicteur retourne alors à l'état NT. Si le cycle contient plus de deux éléments, les éléments qui restent ne provoquent plus d'erreur et laisse l'état inchangé en NT, puisqu'ils sont tous plus petits que le premier élément de $f(C_i)$. Pour résumer, nous avons vu qu'il existait deux cas importants : soit C_i est de taille 1 et le parcours de $f(C_i)$ provoque une erreur de prédiction et fait passer l'état du prédicteur à T, soit C_i est de taille supérieure ou égale à 2 et le parcours de $f(C_i)$ provoque deux erreurs de prédiction et l'état du prédicteur passe à NT. Le même raisonnement peut être fait en supposant que l'état initial du prédicteur est T. Nous résumons l'ensemble de ces cas dans le tableau qui suit en dessous.

état initial	cycle de taille 1		cycle de taille ≥ 2	
	err. pred.	état final	err. pred.	état final
NT	1	T	2	NT
T	0	T	1	NT

Pour déterminer le nombre d'erreurs de prédiction causées par l'entrée $\mathcal{F}(\sigma)$, remarquons qu'en lisant la première ligne, nous observons qu'il y a toujours au moins une erreur lorsque l'état initial est NT. En faisant une lecture des colonnes, nous remarquons que le prédicteur se trouve à l'état NT uniquement lorsque le cycle précédent est de taille au moins 2. Nous avons donc $\mathbf{Cyc}_{\geq 2}(\sigma) + \mathcal{O}(1) = \mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma) + \mathcal{O}(1)$ erreurs de prédiction qui proviennent du cas où le prédicteur commence par cet état. Le terme $\mathcal{O}(1)$ permet de capter l'état initial lors du parcours de $f(C_1)$ qui peut dépendre d'exécutions d'autres programmes dans le passé et qui a donc un caractère aléatoire. Nous remarquons en lisant la dernière colonne qu'il y a toujours au moins une erreur de prédiction lorsque le cycle est de taille supérieure ou égale à 2 pour tout état initial. Nous comptons donc encore une fois $\mathbf{Cyc}_{\geq 2}(\sigma) = \mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma)$ erreurs de prédiction. Si nous faisons la somme de la contribution des cas où le prédicteur est à l'état NT et des cas où les cycles sont de taille supérieure à 2, nous obtenons ainsi la totalité des erreurs de prédiction. Il n'y a pas de terme d'exclusions à soustraire pour le cas où l'état initial est NT et est de taille supérieure à 2 puisque dans ce cas il y a deux erreurs. La quantité d'erreurs de prédiction pour le prédicteur 1-bit associé à la Ligne 6 est donc donnée par la relation

$$2\mathbf{Cyc}(\sigma) - 2\mathbf{Cyc}_1(\sigma) + \mathcal{O}(1).$$

Le même raisonnement peut s'appliquer avec la Ligne 4, il faut dans ce cas utiliser la bijection fondamentale pour les min-records. Nous pouvons résumer les erreurs de prédiction avec le même tableau que pour le cas précédent. Ainsi, le nombre d'erreurs est alors donné par la même relation et la somme de ces deux quantités permet de donner le nombre d'erreurs de prédiction effectuées au total lors de l'exécution de NaiveMinmax avec pour entrée σ qui est donné par la formule

$$4\mathbf{Cyc}(\sigma) - 4\mathbf{Cyc}_1(\sigma) + \mathcal{O}(1).$$

Comme $\mathbb{E}_n[\mathbf{Cyc}] \sim \log n$ et $\mathbb{E}_n[\mathbf{Cyc}_1] \to 1$ quand n tend vers l'infini, cette quantité est donc asymptotiquement équivalent à $4\log(n)$ ce qui nous permet de conclure.

Pour le prédicteur 2-bits saturé, la même méthode peut être utilisée. Comme le prédicteur 2-bits a plus d'état que le 1-bit, il faut faire attention à prendre en compte des tailles de cycles plus grands qu'au raisonnement précédent. Regardons une fois de plus en détail un cas particulier. Supposons que l'état initial est NT au début du parcours de $f(C_i)$. Une erreur de prédiction se produit alors au parcours du premier élément et le prédicteur passe à l'état T. Si C_i a une longueur d'au moins 2, alors le prochain élément provoque également une erreur de prédiction et le prédicteur retourne à l'état NT. Si la longueur est d'au moins 3, alors le prédicteur passe à l'état SNT et ne provoque pas plus d'erreurs de prédiction. Nous avons résumé une fois de plus l'ensemble de ces cas dans le tableau ci-dessous.

état initial	cycle de taille 1		cycle de taille 2		cycle de taille 3		cycle de taille ≥ 4	
	err. pred.	état final	err. pred.	état final	err. pred.	état final	err. pred.	état final
SNT	1	NT	1	SNT	1	SNT	1	SNT
NT	1	T	2	NT	2	SNT	2	SNT
T	0	ST	1	T	2	NT	2	SNT
ST	0	ST	1	T	2	NT	2	SNT

Soit $\chi(\mathcal{F}(\sigma))$ le nombre d'erreurs de prédiction provoquées lors du test de la Ligne 6. Nous allons chercher un encadrement de cette quantité. La minoration se fait en lisant la dernière colonne du tableau tandis que la majoration se fait en lisant les lignes. Nous remarquons qu'il y a au moins une erreur de prédiction lorsque le cycle est de taille au moins 4, et donc nous pouvons directement en déduire que $\mathbf{Cyc}_{\geq 4}(\sigma) \leq \chi(\mathcal{F}(\sigma))$. Pour la majoration, nous avons deux cas : l'état initial est SNT et le prédicteur fait au plus 1 erreur de prédiction, l'état initial n'est pas SNT et le prédicteur fait au plus 2 erreurs de prédiction. L'état initial est SNT si le cycle parcouru précédemment est de taille au moins 2. Le nombre d'erreurs de prédiction pour ce cas ne dépasse donc pas $\mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma) + \mathcal{O}(1)$ avec le terme $\mathcal{O}(1)$ qui permet de capter les effets de bords associés à l'état initial au début de l'exécution de l'algorithme et de la taille du dernier cycle. L'état initial n'est pas SNT si le cycle qui le précède est de taille inférieure à 3. Le nombre d'erreurs pour le deuxième cas ne dépasse donc pas $2\mathbf{Cyc}_{\leq 3}(\sigma)$. En faisant la somme des contributions de ces deux cas nous obtenons un majorant du nombre d'erreurs de prédiction qui est donné par la relation

$$\chi(\mathcal{F}(\sigma)) \leq \mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma) + 2\mathbf{Cyc}_{\leq 3}(\sigma) + \mathcal{O}(1) \leq \mathbf{Cyc}_{\geq 4}(\sigma) + 3\mathbf{Cyc}_{\leq 3}(\sigma) + \mathcal{O}(1).$$

Rappelons que nous avons $\mathbb{E}_n[\mathbf{Cyc}_{\geq 4}] \sim \log n$ et $\mathbb{E}_n[\mathbf{Cyc}_{\leq 3}] = \mathcal{O}(1)$ quand n tend vers l'infini. En appliquant le même raisonnement pour la Ligne 4, nous pouvons conclure que pour le prédicteur 2-bits saturé nous avons asymptotiquement $2 \log n$ erreurs de prédiction.

Nous procédons de même avec le prédicteur 3-bits. L'ensemble des cas possibles est détaillé dans le tableau ci-dessous.

état initial	cycle de taille 1		cycle de taille 2		cycle de taille 3		cycle de taille 4	
	err. pred.	état final	err. pred.	état final	err. pred.	état final	err. pred.	état final
$SSSNT$	1	$SSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
$SSNT$	1	SNT	1	$SSNT$	1	$SSSNT$	1	$SSSNT$
SNT	1	NT	1	SNT	1	$SSNT$	1	$SSSNT$
NT	1	T	2	NT	2	SNT	2	$SSNT$
T	0	ST	1	T	2	NT	2	SNT
ST	0	SST	1	ST	2	T	3	NT
SST	0	$SSST$	1	SST	2	ST	3	T
$SSST$	0	$SSST$	1	SST	2	ST	3	T

état initial	cycle de taille 5		cycle de taille 6		cycle de taille 7		cycle de taille ≥ 8	
	err. pred.	état final	err. pred.	état final	err. pred.	état final	err. pred.	état final
$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
$SSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
SNT	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
NT	2	$SSSNT$	2	$SSSNT$	2	$SSSNT$	2	$SSSNT$
T	2	$SSNT$	2	$SSSNT$	2	$SSSNT$	2	$SSSNT$
ST	3	SNT	3	$SSNT$	3	$SSSNT$	3	$SSSNT$
SST	4	NT	4	SNT	4	$SSNT$	4	$SSSNT$
$SSST$	4	NT	4	SNT	4	$SSNT$	4	$SSSNT$

Nous allons chercher une fois de plus un nouvel encadrement de la quantité du nombre d'erreurs de prédiction effectuées par NaiveMinmax avec pour entrée σ. La lecture de la dernière colonne nous donne, une fois de plus, un minorant simple qui est le nombre de cycles de taille supérieur à 8. Pour le majorant nous allons utiliser la même astuce qui a été utilisée pour le prédicteur 2-bits. Nous pouvons remarquer en lisant la première ligne que nous faisons au plus 1

erreur de prédiction quand le prédicteur est à l'état initial SSSNT. Le prédicteur est dans cet état au plus $\mathbf{Cyc}(\sigma) + \mathcal{O}(1)$ fois avec le terme $\mathcal{O}(1)$ qui capte les effets de bords. Si l'état initial du prédicteur n'est pas SSSNT, alors le nombre d'erreurs est au plus 4. Le prédicteur n'est pas dans cet état, au début de chaque cycle, au plus $\mathbf{Cyc}_{\leq 7}$ fois. Quand on combine ces deux cas, nous obtenons une borne supérieure du nombre de prédictions qui est donné par la relation

$$\chi(\mathcal{F}(\sigma)) \leq \mathbf{Cyc}(\sigma) + 4\mathbf{Cyc}_{\leq 7}(\sigma) + \mathcal{O}(1) \leq \mathbf{Cyc}_{\geq 8}(\sigma) + 5\mathbf{Cyc}_{\leq 7}(\sigma) + \mathcal{O}(1).$$

Une fois de plus en ajoutant la contribution de la Ligne 4, nous avons asymptotiquement un total de $2 \log n$ erreurs de prédiction. □

Intéressons nous maintenant au cas du nombre d'erreurs de prédiction produites par l'algorithme $\frac{3}{2}$-MINMAX.

Le test à la Ligne 3 du $\frac{3}{2}$-MINMAX est vraie si il y a une montée à la position $i + 1$. Il faut alors analyser les alternances de montées et de descentes pour obtenir la proposition suivante.

Proposition 16. *Le nombre d'erreurs de prédiction effectuées par le* $\frac{3}{2}$-MINMAX*, pour une distribution uniforme sur un tableau de taille* n*, est asymptotiquement équivalent à* $\frac{n}{4}$ *pour tous les prédicteurs* 1*-bits,* 2*-bits,* 3*-bits saturés.*

Démonstration. Il est facile de montrer que le nombre d'erreurs de prédiction engendrées par les instructions aux lignes 6,8,11 et 13 sont en $\mathcal{O}(\log n)$ pour ces prédicteurs. En effet, ces lignes ne s'activent uniquement que quand les éléments comparés sont des records. Elles ne font donc pas plus d'erreurs de prédiction que les instructions de branchements de NAIVEMINMAX dont on a vu le comportement asymptotique dans la Proposition 15. Concernant l'instruction de la Ligne 5, sous le modèle uniforme comme il y a autant de chance de trouver les deux éléments comparés dans n'importe quel ordre, une erreur de prédiction aura lieu avec probabilité $\frac{1}{2}$ quel que soit le prédicteur utilisé. En effet, comme ces tests sont indépendants à chaque itération et qu'il y a une chance sur deux d'avoir le résultat vrai ou faux pour ce test, celui-ci est imprédictible. Ainsi en moyenne, ce test provoque $\frac{n}{4}$ erreurs de prédiction. En faisant la somme de toutes ces contributions, nous parvenons à la conclusion annoncée. □

La Table 5.2 donne similairement à la Table 5.1 obtenu avec PAPI le nombre d'erreurs de prédiction pour ces trois prédicteurs de ces deux algorithmes. Ces résultats ont été obtenus en simulant ces prédicteurs à l'aide de la librairie predictors. Une description de la librairie ainsi que le code source de cette simulation sont disponibles en Annexe A dans le Listing 6.

Une conséquence de la Proposition 16 est que le nombre d'erreurs de prédiction est négligeable devant le nombre de comparaisons pour le cas de NAIVEMINMAX. Les observations de la Table 5.1 et de la Table 5.2 semblent confirmer ce résultat.

Nous pensons que cette différence du nombre d'erreurs de prédiction entre les deux algorithmes est la raison qui explique cette différence de performance, étant donné que les erreurs de prédiction coûtent beaucoup de cycles de CPU et que les comparaisons de deux réels ne sont pas des opérations trop coûteuses. Nous avons également testé la robustesse de nos programmes aux optimisations proposées par le compilateur gcc. Dans le cas de l'optimisation -O3, tous les

branchements conditionnels excepté celui du test de la Ligne 5 du $\frac{3}{2}$-MINMAX sont remplacés par des mouvements conditionnels qui sont insensibles aux effets de la prédiction. Ainsi donc le $\frac{3}{2}$-MINMAX continue de causer environ $\frac{n}{4}$ erreurs de prédiction en moyenne.

Prédicteurs	1-bit		2-bit		3-bit	
Taille de l'entrée	#err. trois demi	#err. naïf	#err. trois demi	#err. naïf	#err. trois demi	#err. naïf
1000	274	25	274	16	264	16
2000	533	31	520	18	525	17
4000	1028	29	1019	21	1015	18
8000	2022	33	2014	22	2012	20
16000	4008	33	4008	22	3993	21
32000	8039	36	8060	22	8019	22
64000	16030	41	15975	26	16042	24
128000	32042	44	32020	26	32060	24
256000	64078	49	64149	26	64008	29
512000	128113	47	127957	30	128071	29
1024000	256092	53	256151	30	256068	29
2048000	511992	57	512108	30	511862	30
4096000	1023857	53	1023987	35	1024652	32
8192000	2047648	62	2047923	32	2047111	33
16384000	4095846	62	4095597	37	4095816	36
32768000	8192121	63	8192497	37	8192314	40
65536000	16384437	66	16384640	36	16383884	35
131072000	32767836	75	32769580	41	32768201	40

TABLE 5.2 – Tableau du nombre d'erreurs de prédiction engendrées par les algorithmes NAIVEMINMAX et $\frac{3}{2}$-MINMAX sur des prédicteurs 1-bit, 2-bits saturé et 3-bits saturé.

5.2 Exponentiation rapide

Dans la section précédente, nous avons vu qu'une instruction de branchement dont les deux issues sont équiprobables rend la prédiction de branchement impossible. Ces instructions sont alors très coûteuses lorsqu'elles sont exécutées par un processeur RISC et il serait préférable de les éviter en pratique. Nous avons deux possibilités qui nous viennent à l'esprit pour y parvenir. La première est de contourner la prédiction de branchements en utilisant des instructions spéciales du microprocesseur comme par exemple les mouvements conditionnels *CMOV*. La deuxième méthode, que nous avons choisie, consiste à modifier les algorithmes pour faire en sorte que les instructions de branchements ne soient pas imprédictibles. Cette démarche va à l'encontre des paradigmes qui ont permis de créer la grande majorité des algorithmes classiques populaires. Dans ces derniers, les branchements ont bien souvent intérêt à avoir une équiprobabilité des issues. Cette équiprobabilité permet d'obtenir dans nombreux cas des complexités optimales. C'est par exemple le cas de $\frac{3}{2}$-MINMAX que nous avons vu dans la section précédente. Cet algorithme est optimal en nombre de comparaisons mais a besoin de faire une comparaison qui a autant de chance d'être vrai ou fausse dans le cas où l'entrée suit une loi uniforme. Nous avons vu que NAIVEMINMAX est plus rapide en pratique par rapport à $\frac{3}{2}$-MINMAX bien que ce premier fait plus de comparaisons que ce dernier. Nous allons donc essayer de modifier des algorithmes classiques dont les instructions de branchements posent problème à la prédiction. Cette modification ne se fera pas toujours sans un coût supplémentaire, et il sera donc nécessaire de trouver un compromis pour que ce coût ne soit pas trop important. Dans cette section, nous allons nous intéresser en particulier à l'algorithme d'exponentiation rapide qui prend en entrée un élément x appartenant à un monoïde $(M, .)$ et un entier naturel n et donne en sortie la valeur x^n.

5.2.1 Présentation de l'exponentiation rapide et identification du problème

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1 r ← 1
2 y ← x
3 k ← n
4 tant que k ≠ 0 faire
5 |   si k est impair alors
6 |   |   r ← r · y
7 |   k ← ⌊k/2⌋
8 |   y ← y²
9 retourner r
```

Algorithme 10 : Algorithme de l'exponentiation rapide classique

Nous avons donné dans l'Algorithme 10, une version en pseudo-code de l'exponentiation rapide dont nous allons décrire dès à présent le fonctionnement. À l'initialisation, nous créons trois variables r, y et k qui sont respectivement affectées par les valeurs $1, x$ et n avec 1 qui est l'élément neutre du monoïde M. L'algorithme consiste principalement en une boucle qui s'arrête lorsque la valeur de k est 0. À chaque itération, nous élevons y au carré et nous divisons k par deux. Il est donc ici évident que la complexité en nombre d'itérations est asymptotiquement équivalente à $\log_2 n$. Pour chaque itération nous avons deux cas, soit k est impair et nous multiplions r par y, soit k est pair et nous ne faisons rien de plus. Lorsque la boucle s'arrête nous savons que $r = x^n$ et nous retournons donc le résultat. Pour démontrer que cet algorithme fonctionne correctement, nous pouvons faire une preuve en utilisant l'invariant du Lemme 8.

Lemme 8. *Au début de la ligne 4, l'équation suivante est toujours vérifiée :*

$$x^n = r \cdot y^k$$

Démonstration. À l'initialisation, comme $r = 1$, $y = x$ et $k = n$, il est évident que l'équation est vérifiée. Supposons maintenant que l'invariant est vrai au début d'une certaine itération. Nous avons alors $x^n = r \cdot y^k$ qui est vraie. Nous avons deux cas possibles. Soit k est pair, dans ce cas nous pouvons ré-écrire l'expression de droite sous la forme :

$$r \cdot y^k = r \cdot y^{2(\lfloor \frac{k}{2} \rfloor)} = r \cdot (y^2)^{\lfloor \frac{k}{2} \rfloor}.$$

Dans ce cas l'invariant est bien rétabli en affectant $y \cdot y$ à y et $\lfloor \frac{k}{2} \rfloor$ à k.

Soit k est impair, dans ce cas nous pouvons ré-écrire l'expression de droite sous la forme :

$$r \cdot y^k = r \cdot y^{2(\lfloor \frac{k}{2} \rfloor)+1} = (r \cdot y) \cdot (y^2)^{\lfloor \frac{k}{2} \rfloor}.$$

Dans ce cas l'invariant est également rétabli en affectant en plus des affectations du premier cas la valeur $r \cdot y$ à r. □

Lorsque la boucle s'arrête, l'invariant est alors vrai et $k = 0$. Nous savons

donc que $r \cdot y^k = x^n$. Or comme $k = 0$, $y^k = 1$ et donc $r = x^n$.

Nous avons donc avec l'Algorithme 10, une méthode rapide pour calculer x^n. Cependant en ce qui concerne la prédiction de branchement nous avons un problème. Supposons que le paramètre n est distribué aléatoirement et uniformément dans l'intervalle $[0, 2^K[$ pour un entier $K \in \mathbb{N}$. Nous avons alors une chance sur deux que chaque bit de n une fois écrit en binaire soit 0 ou 1. Or le test de parité revient à tester pour la i-ème itération si le i-ème bit de n est 0 ou 1. Autrement dit pour cette distribution, le test de parité de la ligne 5 est imprédictible. La Proposition 17 donne la complexité du nombre d'erreurs de prédiction pour plusieurs prédicteurs locaux décrit au Chapitre 3.

Proposition 17. *Pour l'algorithme 10, si l'entrée n est choisie uniformément dans l'intervalle $[0, 2^K[$ pour un entier $K \in \mathbb{N}$ arbitraire, alors quel que soit le prédicteur, le nombre d'erreurs de prédiction est asymptotiquement équivalente à $0.5 \log_2 n$.*

Démonstration. Comme l'instruction de branchement de la ligne 5 est imprédictible et qu'il y a en tout $\log_2 n$ itérations, ce résultat est trivial. □

5.2.2 Les modifications apportées à l'algorithme classique

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1  r ← 1
2  y ← x
3  k ← n
4  tant que k ≠ 0 faire
5  |  si k est impair alors
6  |  |  r ← r · y
7  |  y ← y²
8  |  si ⌊k/2⌋ est impair alors
9  |  |  r ← r · y
10 |  y ← y²
11 |  k ← ⌊k/4⌋
12 retourner r
```

Algorithme 11 : Algorithme de l'exponentiation rapide classique avec déroulement de la boucle principale

Nous allons maintenant voir comment nous avons modifié l'Algorithme 10, que nous nommons dorénavant ClassicalPow, pour qu'il soit capable de guider le prédicteur de branchements. Il semble difficile de pouvoir rajouter directement un biais au branchement de la ligne 5. Nous avons donc procédé en plusieurs étapes en obtenant diverses variantes de l'algorithme. Une méthode simple dans notre cas consiste à faire un déroulement de la boucle de la ligne 4. En procédant de cette façon, nous obtenons l'Algorithme 11, que nous nommons UnrolledPow, qui se comporte exactement de la même façon que l'Algorithme 10 mais qui fait deux fois moins d'itérations. En pratique, une implantation de l'Algorithme 11 sera plus rapide que l'algorithme 10 car elle

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1  r ← 1
2  y ← x
3  k ← n
4  tant que k ≠ 0 faire
5  |  z ← y²
6  |  si k est impair ou ⌊k/2⌋ est impair alors
7  |  |  si k est impair alors
8  |  |  |  r ← r · y
9  |  |  si ⌊k/2⌋ est impair alors
10 |  |  |  r ← r · z
11 |  y ← z²
12 |  k ← ⌊k/4⌋
13 retourner r
```

Algorithme 12 : Algorithme de l'exponentiation rapide permettant de guider le prédicteur de branchement

fait moins d'opérations de contrôle qui sont liées à la boucle. Cependant, ce nouvel algorithme n'a rien changé à la complexité du nombre d'erreurs de prédiction excepté que maintenant nous avons deux instructions de branchement indépendantes aux lignes 5 et 8.

L'idée que nous avons eue pour modifier ce nouvel algorithme afin qu'il soit capable de guider le prédicteur a été de créer artificiellement de la dépendance entre ces deux instructions de branchement. Nous y parvenons en ajoutant un troisième test qui consiste simplement à vérifier qu'au moins un des deux autres tests est vrai. Si le résultat de ce test est vrai, alors nous faisons les deux autres tests comme précédemment. L'algorithme 12, que nous nommons GuidedPow, est ainsi obtenu en procédant de cette manière. Cet algorithme fait plus de comparaisons que les deux précédents, mais il est capable en contrepartie de guider le prédicteur. La Table 5.3 résume la différence entre ces trois algorithmes pour divers types d'opérations qu'elles effectuent. Le coût en multiplication est identique pour ces trois algorithmes, nous y parvenons grâce à l'ajout d'une variable supplémentaire.

Regardons maintenant ce qu'il se passe du point de vue probabiliste pour chacune de ces instructions de branchements. Gardons à l'esprit que nous sommes dans le cas uniforme où les instructions de branchements, si elles sont faites de manière indépendante comme dans l'Algorithme 11, ont une chance sur deux de produire chaque issue. Les tests des lignes 6,7 et 9 utilisent deux clauses : la première est de savoir si k est impair, la deuxième est de savoir si $\lfloor \frac{k}{2} \rfloor$ est impair. Nous avons donc quatre possibilités au total sur les valeurs de ces deux clauses qui ont chacune une chance sur quatre de se produire. Le résultat de la première instruction de branchement de la ligne 6 est faux uniquement si ces deux clauses sont fausses. Ainsi la probabilité que cette instruction soit prise est égale à $\frac{3}{4}$. Le résultat de l'instruction de branchement de la ligne 7 est vrai si la clause "k est impaire" est vraie, peu importe la valeur de l'autre clause, ce qui fait donc deux possibilités. Comme nous faisons ce test uniquement si

le résultat du test précédent est vrai, cela signifie qu'au moins une des deux clauses est vraies et il ne reste donc au total plus que trois possibilités. Nous avons donc une probabilité de $\frac{2}{3}$ que le résultat de l'instruction de la ligne 7 soit vrai. Le même raisonnement permet de montrer que le résultat de la ligne 9 est vrai avec probabilité $\frac{2}{3}$.

Remarque. Nous avons réussi, en rajoutant une connaissance sur les deux clauses apportées par l'instruction de la ligne 6, à créer plus facilement un déséquilibre sur toutes les instructions de branchements qui deviennent alors prédictibles. Remarquons cependant que cette instruction supplémentaire augmente la complexité en nombre de comparaisons comme nous pouvons le voir sur la table 5.3. Remarquons également que l'instruction que nous avons ajoutée est également parfaitement inutile au fonctionnement de l'Algorithme 12 qui se réduit au cas de l'Algorithme 11 une fois cette instruction retirée.

Version	Itérations	Mult./itér.	Branch./itér
Classique	$\log_2 n$	1.5	1
Avec déroulement	$\frac{\log_2 n}{2}$	3	2
Avec guidage de la prédiction	$\frac{\log_2 n}{2}$	3	2.5

TABLE 5.3 – Comparatifs des Algorithmes 10,11 et 12, sur le nombres d'itérations, de multiplication par itération, d'instructions de branchement en moyenne.

5.2.3 Résultats expérimentaux

Dans le but de comparer nos trois algorithmes d'exponentiation rapides, nous effectuons des expérimentations sur des implantations de ces algorithmes en C. Les résultats qui sont présentés dans cette section ont été obtenus sur la même machine que nous avons utilisée pour les mesures sur la recherche du minimum et du maximum. Pour rappel, le microprocesseur sur cette machine est un *Intel(R) Celeron(R) CPU 1037U @ 1.80GHz*. Nous avons effectué ces expérimentations sur d'autres machines avec d'autres processeurs et avons obtenu des différences de performances similaires. Les implantations de nos trois algorithmes sont données dans le Listing 11, disponible en Annexe B. Nous avons mesuré ces trois algorithmes sur plusieurs critères à savoir : le temps d'exécution, le nombre d'itérations, le nombre d'instructions de branchements, et le nombre d'erreurs de prédiction. Les résultats que nous présentons donnent une somme des mesures pour tous ces critères sur un ensemble de $5 \cdot 10^7$ exécutions de nos implantations. Une exécution correspond à choisir au hasard et uniformément un entier n dans l'intervalle $[1, 4^{13}[$, ensuite nous calculons pour toutes les implantations la valeur 2^n. Le choix de l'intervalle $[1, 4^{13}[$ permet d'avoir une chance sur deux qu'un bit de n soit 0 ou 1 sur les 26 premiers bits. Nous rendons donc ainsi les tests faits sur un seul bit sans connaissances supplémentaires imprédictibles. Les résultats obtenus sont donnés dans la table 5.4. Nous pouvons vérifier que le nombre d'itérations des implantations de l'exponentiation rapide est deux fois plus grand que celui des deux autres. Le nombre de multiplications est légèrement plus petit pour l'exponentiation classique, cette différence est due à un effet de bord qui provient du fait que n n'est pas for-

cément une puissance de 4. Venons en maintenant à ce qui nous intéresse le plus, à savoir les instructions de branchements et la prédiction de ces dernières. Nous remarquons que l'exponentiation classique fait le moins de comparaisons suivie de près par l'exponentiation avec juste le déroulement de la boucle principale. Cette différence est négligeable et s'explique encore à cause d'un effet de bord. Cependant, la différence entre l'exponentiation classique et l'exponentiation qui guide le prédicteur est bien plus importante. L'exponentiation qui guide le prédicteur compte environ 27.5% d'instructions de branchements en plus que l'exponentiation rapide. Cette différence était prévisible, nous pouvions estimer cette dernière à 25% en utilisant les données de la Table 5.3. Le résultat obtenu est donc très proche de ce que nous avons estimé. Le nombre d'erreurs de prédiction de branchement est sensiblement identique pour l'exponentiation classique et la version avec déroulement de la boucle principale. Cependant, comme nous l'avons espéré, l'implantation de l'algorithme qui guide la prédiction fait environ 16.7% d'erreurs en moins que les deux autres implantations. Ce gain en prédictions se montre efficace puisque nous observons un gain en temps d'exécution d'environ 30% contre l'exponentiation rapide classique et d'environ 14.7% contre l'exponentiation rapide avec déroulement de la boucle principale. Ce résultat laisse penser qu'il existe des compromis à chercher entre le nombre de comparaisons et le nombre d'erreurs de prédiction de branchement dans le cas où nous calculons des puissances de nombres réelles.

exponentiation	temps (en sec.)	itér. $\times 10^9$	mult. $\times 10^9$	ins. branch. $\times 10^9$	err. pred. $\times 10^9$
classique	13.859	1.250	1.900	1.300	0.674
avec déroulement	12.229	0.633	1.917	1.317	0.683
avec guide	10.659	0.633	1.917	1.658	0.577

TABLE 5.4 – Mesure de performances, réalisée à l'aide de PAPI, des implantations en C des trois versions de l'exponentiation rapide sur un total de 5.10^7 exécutions. Les mesures prises sont le temps d'exécution total sur ces exécutions, le nombre d'itérations, le nombre de multiplications, le nombre d'instructions de branchements, et le nombre d'erreurs de prédiction.

5.2.4 Analyse dans le cas moyen du nombre d'erreurs de prédiction de GUIDEDPOW

Nous allons maintenant nous intéresser à l'analyse du nombre d'erreurs de prédiction générées pour une même entrée par chacun de ces algorithmes. Les nombres d'erreurs produites par l'algorithme classique et l'algorithme avec déroulement de boucle sont des cas simples de par l'impossibilité de prédire l'issue des instructions de prédictions quel que soit le prédicteur choisit. Nous allons donc principalement nous intéresser au cas de GUIDEDPOW. Pour l'analyse de ces algorithmes nous allons considérer que l'exposant n est choisi au hasard dans un intervalle $\{0, \dots N\}$ pour un N entier strictement positif. Pour la suite, nous présenterons l'analyse du nombre d'erreurs de prédiction si le prédicteur utilisé fait partie des prédicteurs locaux qui ont été présentés dans la section 3.5 du chapitre 3.

Remarque. Les instructions de branchements permettant de tester la divisibilité par 2 et par 4 des algorithmes 10,11, et 12 reviennent à tester la valeur des deux derniers bits de la variable k, ce qui correspond à ce que nous avons fait dans

nos implantations en C. Si l'on décompose sous sa forme binaire $n = \sum_{i \geq 0} n_i 2^i$ alors les algorithmes vont tester dans l'ordre chacun de ces bits en commençant par celui de poids faible. Les algorithmes 11 et 12 testent ces bits deux par deux ce qui peut les amener à tester un bit supplémentaire en plus du bit de poids fort. En particulier pour ces algorithmes, nous regardons à l'itération j les bits $n_{2(j-1)}$ et n_{2j-1}.

Le cas idéal

Pour commencer l'analyse, nous considérons que n est choisi uniformément sur $\{0 \dots N\}$, avec $N = 4^k - 1$ pour un $k \geq 1$. Ce modèle est similaire à celui qui consiste à choisir chacun des $2k$ bits de la représentation binaire de n uniformément et indépendamment par une loi de Bernoulli de paramètre $\frac{1}{2}$. Cette indépendance des bits de n rend le cas simple à analyser et nous le considérons donc comme un cas idéal. Soit $L_k(n)$ le nombre d'itérations de la boucle principale de GuidedPow. Il s'agit d'une variable aléatoire qui est simple à analyser puisqu'elle est égale au plus petit entier l tel que 4^l est plus grand que n. En particulier, son espérance est donnée par :

$$\mathbb{E}[L_k] = k - \frac{1}{3} + o(1) \sim k.$$

Cette espérance peut facilement être obtenue en sommant la probabilité $\mathbb{P}(L_k > l) = 1 - 4^{l-k}$ car $L_k(n) > l$ implique $n \in \{4^l + 1 \dots 4^k\}$.

En appliquant les outils mathématiques vu dans la Section 3.5 du Chapitre 3, nous obtenons le résultat donné dans le Théorème 10.

Théorème 10. *Supposons que n est choisi aléatoirement selon une distribution uniforme dans* $\{0, N-1\}$. *L'espérance du nombre d'instructions de branchements effectuées par* ClassicalPow *et* UnrolledPow *est asymptotiquement équivalent à* $\log_2 N$, *tandis qu'il est asymptotiquement équivalent à* $\frac{5}{4}\log_2 N$ *pour* GuidedPow. *L'espérance du nombre d'erreurs de prédiction est asymptotiquement équivalent à* $\frac{1}{2}\log_2 N$ *pour* ClassicalPow *et* UnrolledPow, *pour tous les types de prédicteurs. Pour* GuidedPow, *il est asymptotiquement équivalent à* $\alpha \log_2 N$, *avec* $\alpha = \frac{1}{2}\mu(\frac{3}{4}) + \frac{3}{4}\mu(\frac{2}{3})$, *où* μ *est la probabilité de faire une erreur de prédiction sous la distribution stationnaire de la chaîne de Markov associée au prédicteur local (pour plus de détails , voir la section 3.5.7 du chapitre 3).*

Démonstration. La preuve est faite pour $N = 4^k$. Nous verrons plus loin comment gérer le cas général. Il est simple de montrer, comme nous l'avons fait précédemment avec UnrolledPow, que le nombre d'itérations en moyenne de ClassicalPow est asymptotiquement équivalent à $\log_2 N + \mathcal{O}(1)$. Comme pour chaque itération, un test est effectué avec une probabilité $\frac{1}{2}$ de causer une erreur de prédiction, le résultat annoncé est ainsi obtenu pour cet algorithme. Pour UnrolledPow et GuidedPow, nous savons déjà que l'espérance du nombre d'itérations est asymptotiquement équivalente à $\log_4 N = \frac{1}{2}\log_2 N$. UnrolledPow fait pour chaque itération exactement deux tests qui ont chacun une probabilité $\frac{1}{2}$ de causer une erreur de prédiction, nous amenant au résultat. GuidedPow va quant à lui pour chaque itération exécuter le premier test, si il est vrai avec probabilité $\frac{3}{4}$, alors deux tests supplémentaires sont effectués. Ceci nous amène donc au premier résultat annoncé qui est que le nombre de tests effectués en moyenne est asymptotiquement équivalent à $\frac{5}{4}\log_2 N$. Pour

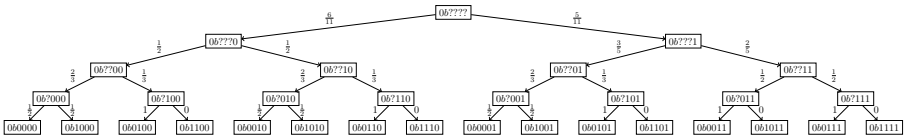

FIGURE 5.4 – Arbre de décomposition du choix d'un entier n dans l'intervalle $\{0 \dots 10\}$.

le deuxième résultat, nous utilisons le Théorème 4 ergodique et le fait que $\mathbb{E}[L_k] \sim k \sim \log_4 N$. Le premier test cause un nombre d'erreurs de prédiction en moyenne qui est asymptotiquement équivalent à $\mu(\frac{3}{4}) \log_4 N$ et chacun des deux tests internes causent un nombre d'erreurs de prédiction en moyenne qui est asymptotiquement équivalent à $\frac{3}{4}\mu(\frac{2}{3}) \log_4 N$. En combinant ces résultats, le nombre d'erreurs de prédiction en moyenne est asymptotiquement équivalent à $(\mu(\frac{3}{4}) + \frac{3}{2}\mu(\frac{2}{3})) \log_4 N$, concluant ainsi la preuve. □

En utilisant le Théorème 10 ainsi que les Equations (3.3),(3.4),(3.5) et (3.6), nous obtenons $\alpha = \frac{25}{48} \approx 0.52$, $\frac{9}{20} \approx 0.45$, $\frac{2045}{4368} \approx 0.47$ et $\frac{1095}{2788} \approx 0.39$ pour respectivement le 1-bit, le 2-bits saturé, le 2-bits avec saut et le 3-bits saturé. Ces valeurs doivent être comparées avec la constante multiplicative $\frac{1}{2}$ des deux autres algorithmes. En particulier pour le prédicteur 1-bit, l'espérance du nombre d'erreurs de prédiction est plus importante pour GUIDEDPOW que pour les deux autres. Ce prédicteur n'est donc pas suffisamment efficace pour compenser les erreurs de prédiction causées par la condition supplémentaire. Pour le compteur 3-bits saturé, GuidedPow fait approximativement $0.25 \log_2 n$ comparaisons de plus que UNROLLEDPOW, mais en contrepartie fait approximativement $0.11 \log_2 n$ erreurs de prédiction de moins. Nous avons pu vérifier ces résultats à l'aide de la librairie de simulation en python que nous présentons à l'Annexe A en utilisant le script du Listing 7 qui met en évidence la constante α pour chacun de ces prédicteurs.

Cas général

Le Théorème 10 fonctionne pour toutes valeurs de N y compris celles qui ne sont pas des puissances de 4. Pour ces cas, l'analyse est plus compliquée puisque les bits ne sont pas fixés à 0 ou à 1 avec une probabilité de $\frac{1}{2}$ et ne sont pas indépendants les uns des autres. Nous allons maintenant nous intéresser au cas général où N n'est pas nécessairement de la forme $4^k - 1$. Nous allons démontrer que le cas idéal est une approximation du terme de premier ordre du cas général. Notre idée est de montrer que le choix de n dans l'intervalle $\{0 \dots N\}$ est quasiment similaire sur les bits de poids faibles à choisir un nombre dans l'intervalle $\{0 \dots 4^{\widehat{k}} - 1\}$ avec $\widehat{k}$ la plus petite valeur telle que $4^{\widehat{k}} - 1 \geq N$. Nous pouvons représenter la génération d'un nombre choisi pour ces deux intervalles uniformément sous la forme d'un arbre de probabilité. Pour chaque nœud de hauteur i nous associons l'expérience aléatoire qui consiste à choisir la valeur du i-ème bit de ce nombre sachant que les bits de poids faibles de ce nombre sont déjà fixés par le chemin parcouru pour atteindre ce nœud. Nous notons pour un tel nœud x, par $p_x(N)$ la probabilité que le prochain bit soit 1 après la réalisation de l'expérience aléatoire associée à ce nœud pour le choix d'un entier

dans l'intervalle $\{0 \dots N\}$. Similairement, nous définissons $\widehat{p}_x(N)$ la probabilité pour l'arbre du cas idéal. La figure 5.4 donne une représentation des arbres de probabilités associés au choix aléatoire uniforme d'un entier dans les intervalles $\{0 \dots 10\}$. Nous n'avons pas donné d'exemple d'arbre pour le cas idéal car il s'agit d'un cas trivial où l'on remplace toutes les probabilités de chaque arrête par $\frac{1}{2}$. Les feuilles de chaque arbre correspond au choix d'un entier n dans l'intervalle considéré. Pour chaque nœud, nous avons au plus deux successeurs, le successeur de gauche correspond au cas où le prochain bit vaut 0 et le successeur de droite correspond au cas où le prochain bit vaut 1. Nous avons de plus étiqueté chaque arrête par la probabilité de réalisation. Par la suite, nous noterons par $d(x)$ le successeur droit de x et par $g(x)$ son successeur gauche.

Remarque. Comme la loi $\widehat{p}_i(N)$ est un cas idéal provenant d'un intervalle entre 0 et une puissance de 2 exclue. Chaque nœud correspond à une expérience de Bernoulli de paramètre $\frac{1}{2}$. La loi de $p_x(N)$ n'est évidemment pas un cas aussi simple et dépends à la fois de x et de N. Nous allons voir avec le Lemme 9 une expression bien déterminée de sa mesure de probabilité.

Par la suite, nous notons n_x le nombre obtenu en lisant les bits du chemin partant de la racine jusqu'à un nœud x d'un de ces arbres de probabilités.

Lemme 9. *La probabilité $p_x(N)$ que le résultat de l'épreuve au nœud x, de hauteur i dans l'arbre, soit 1 sachant que le nombre obtenu doit être choisi uniformément dans l'intervalle $\{0 \dots N\}$ est donné par la relation*

$$\frac{\lfloor \frac{N-n_x-2^i}{2^{i+1}} \rfloor + 1}{\lfloor \frac{N-n_x}{2^i} \rfloor + 1}.$$

Démonstration. Par définition, nous savons que

$$n_x = \sum_{j=0}^{i-1} b_j 2^j,$$

pour des $b_j \in \mathbb{B}$. À la fin, lorsque le nombre n est choisi, comme nous connaissons déjà les i premiers bits de n, nous pouvons écrire

$$\begin{aligned} n &= \sum_{j \geq 0} b_j 2^j && (5.1)\\ &= \sum_{j=0}^{i-1} b_j 2^j + \sum_{j \geq i} b_j 2^j && (5.2)\\ &= n_x + 2^i \sum_{j \geq 0} b'_j 2^j && (5.3)\\ &= n_x + 2^i n_q, && (5.4) \end{aligned}$$

pour un certain n_q qui est le quotient de la division euclidienne entre n et 2^i. Pour un nœud y, nous notons par N_y le nombre de feuilles associées au sous-arbre dont la racine est y. Comme le choix de n est uniforme, la probabilité que

l'issue de l'expérience du nœud x soit 1 est donnée par l'identité

$$p_x(N) = \frac{N_{d(x)}}{N_x} \tag{5.5}$$

D'après la relation de l'Equation (5.4), nous pouvons facilement exprimer N_x comme le cardinal de l'ensemble des quotients n_q tels que $2^i n_q + n_x \leq N$. Cet ensemble est équivalent à chercher tous les n_q tels que $n_q \leq \frac{N-n_x}{2^i}$. Or comme n_q est entier, cet ensemble est simplement l'intervalle $\{0 \dots \lfloor \frac{N-n_x}{2^i} \rfloor\}$ dont le cardinal est

$$N_x = \left\lfloor \frac{N - n_x}{2^i} \right\rfloor + 1 \tag{5.6}$$

De façon similaire, nous obtenons $N_{d(x)}$ en remarquant que $n_{d(x)} = n_x + 2^i$. Ainsi, nous avons

$$N_{d(x)} = \left\lfloor \frac{N - n_x - 2^i}{2^{i+1}} \right\rfloor + 1 \tag{5.7}$$

En combinant les Equations (5.5),(5.6) et (5.7) nous obtenons le résultat. □

Nous allons procéder à un couplage entre ces deux lois. Pour ce faire, nous allons générer des valeurs de n et $\widehat{n}$ pour respectivement le cas général et le cas idéal à partir d'une suite à valeurs réelles $u = u_0 u_1 \dots u_{\widehat{k}} \in [0,1]^{\widehat{k}}$ dont chaque élément est choisi uniformément et indépendamment des autres. Nous associons à ce mot les chemins $\mathrm{C}(u)$ et $\widehat{\mathrm{C}}(u)$ respectivement aux arbres du cas général et du cas idéal. Ces chemins sont construits en lisant les éléments de u dans l'ordre. Nous allons maintenant décrire la construction pour l'arbre du cas général. Au début nous commençons par la racine r de l'arbre et nous lisons l'élément u_0, si $u_0 < p_x(N)$ alors le prochain nœud à parcourir est $d(x)$ sinon c'est $g(x)$. Notons x_ℓ le nœud atteint après la lecture de l'élément u_ℓ. Nous continuons à parcourir l'arbre en comparant $u_{\ell+1}$ à $p_{x_\ell}(N)$ pour déterminer $x_{\ell+1}$. Pour le cas idéal, il suffit de remplacer les probabilités de la forme $p_y(N)$ par $\widehat{p}_y(N)$. La feuille obtenue, comme elle est issue de nos arbres, est équivalente à un choix uniforme dans nos intervalles. Maintenant que nous savons construire nos chemins, ce que nous aimerions connaître est à quel point pour une même suite u les chemins $\mathrm{C}(u)$ et $\widehat{\mathrm{C}}(u)$ diffèrent. Avant de répondre à cette question, il est important de connaître la probabilité qu'il y ait une bifurcation à la lecture de la lettre u_ℓ sachant que les chemins étaient identiques jusqu'alors. Cette probabilité est donnée dans le Lemme 10

Lemme 10. *La probabilité qu'il y ait une bifurcation à l'étape ℓ entre les chemins $\mathrm{C}(u)$ et $\widehat{\mathrm{C}}(u)$ à la lecture de $u = u_0 u_1 \dots u_{\widehat{k}} \in [0,1]^{\widehat{k}}$ sachant que ces chemins étaient identiques pour les ℓ premières étapes admet une borne supérieure de la forme*

$$p_{bif}(\ell) \leq \frac{\alpha 2^\ell}{N - 2^\ell},$$

avec $\alpha > 0$ constant.

Démonstration. Pour quantifier cette probabilité, commençons premièrement par voir intuitivement ce que ça signifie de bifurquer à la lecture de la lettre u_ℓ. Pour cela nous avons tracé un segment dans la figure 5.5. Cette figure montre que, pour qu'il y ait une bifurcation il faut que u_ℓ soit compris dans l'intervalle

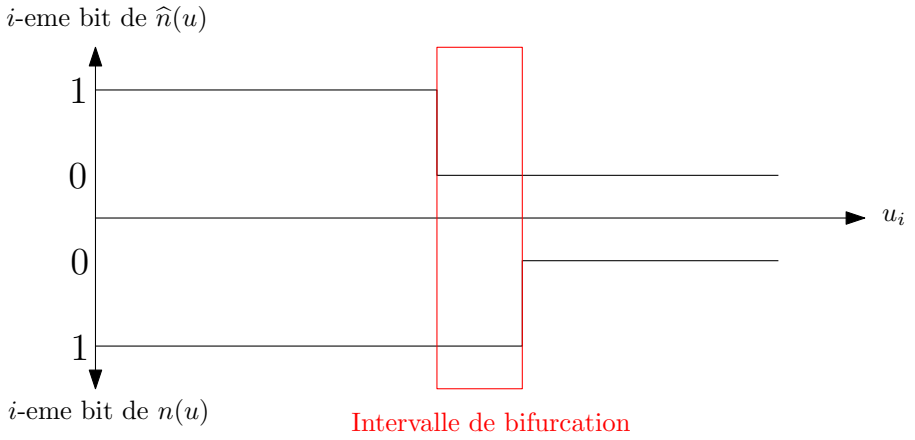

FIGURE 5.5 – Cette figure est une représentation du choix de la valeur du i-ème bit en fonction de la valeur de l'élément u_i de la suite à valeurs réelles u. la courbe du dessus représente le choix du bit du nombre $\widehat{n}(u)$ du cas idéal tandis que la courbe du dessous représente le choix du bit du nombre $n(u)$ du cas général. Les deux bits sont au début à 1 puis à partir de la valeur $\widehat{p}_x(N)$ la valeur du bit de $\widehat{n}(u)$ passe à 0 avec x qui est le noeud de l'arbre atteint à la lecture des i premiers éléments de u. Similairement, la valeur du bit de $n(u)$ passe à 0 à partir de $u_i > p_x(N)$. Entre ces deux valeurs se trouve un intervalle de bifurcation où le bit de $n(u)$ et de $\widehat{n}(u)$ sont différents.

$[\min(\widehat{p}_{x_\ell}(N), p_{x_\ell}(N)), \max(\widehat{p}_{x_\ell}(N), p_{x_\ell}(N))[$. Nous le nommerons par la suite *l'intervalle de bifurcation.*

Comme l'élément u_ℓ est choisie uniformément dans $[0,1]$, nous avons alors une identité simple pour quantifier cette probabilité qui est donnée par

$$p_{bif}(\ell) = |p_{x_\ell}(N) - \widehat{p}_{x_\ell}(N)| = |p_{x_\ell}(N) - \tfrac{1}{2}|. \tag{5.8}$$

Il ne nous reste plus qu'à déterminer un encadrement de la quantité $p_{x_\ell}(N)$ pour conclure. Remarquons premièrement par construction que $0 \leq n_{x_\ell} 2^\ell$. Rappelons que pour tout x nous avons également $x - 1 < \lfloor x \rfloor \leq x$. Avec ces deux encadrements, nous pouvons déterminer des bornes à $p_{x_\ell}(N)$. Commençons par trouver une borne inférieure,

$$\begin{aligned} p_{x_\ell}(N) &= \frac{\left\lfloor \frac{N - n_{x_\ell} - 2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N - n_{x_\ell}}{2^\ell} \right\rfloor + 1} \\ &\geq \frac{\left\lfloor \frac{N - 2^\ell - 2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N}{2^\ell} \right\rfloor + 1} \\ &\geq \frac{\frac{N}{2^{\ell+1}} - 1}{\frac{N}{2^\ell} + 1} \end{aligned}$$

Une dernière simplification de cette expression nous donne

$$p_{x_\ell}(N) \geq \frac{1}{2} - \frac{\frac{3}{2}}{\frac{N}{2^\ell}+1}. \tag{5.9}$$

Procédons de même pour trouver une borne supérieure,

$$\begin{aligned} p_{x_\ell}(N) &= \frac{\left\lfloor \frac{N-n_{x_\ell}-2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N-n_{x_\ell}}{2^\ell} \right\rfloor + 1} \\ &\leq \frac{\left\lfloor \frac{N-2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N-2^\ell}{2^\ell} \right\rfloor + 1} \\ &\leq \frac{\frac{N-2^\ell}{2^{\ell+1}} + 1}{\frac{N-2^\ell}{2^\ell} - 1 + 1} \end{aligned}$$

Après simplification, nous obtenons le majorant

$$p_{x_\ell}(N) \leq \frac{1}{2} + \frac{1}{\frac{N}{2^\ell}-1}. \tag{5.10}$$

En combinant les Equations (5.9) et (5.10), nous obtenons l'encadrement

$$-\frac{\frac{3}{2}}{\frac{N}{2^\ell}+1} \leq p_{x_\ell}(N) - \frac{1}{2} \leq \frac{1}{\frac{N}{2^\ell}-1}. \tag{5.11}$$

De ce dernier nous obtenons

$$|p_{x_\ell}(N) - \tfrac{1}{2}| \leq \frac{\frac{3}{2}}{\frac{N}{2^\ell}+1} + \frac{1}{\frac{N}{2^\ell}-1}, \tag{5.12}$$

nous permettant de conclure après simplification du terme de droite. □

Intéressons nous maintenant à la probabilité que les chemins $C(u)$ et $\widehat{C}(u)$ soient identiques après la lecture des m premières valeurs de la suite u, avec $m \in \mathbb{N}$. Nous notons cette probabilité par $P_{same}(m)$, par la définition des probabilités conditionnelles, nous avons la relation de récurrence suivante qui est vérifiée :

$$P_{same}(m) = (1 - p_{bif}(m))P_{same}(m-1),$$

pour tout $m > 0$. Nous pouvons ainsi directement montrer par récurrence que

$$P_{same}(m) = \prod_{i=0}^{m} 1 - p_{bif}(i). \tag{5.13}$$

Nous sommes maintenant prêt à démontrer que le terme dominant du cas idéal est une approximation du cas général. Nous commençons ainsi avec la Proposition 18.

Proposition 18. *Soit $u = (u_1, u_2, \ldots, u_{\widehat{k}})$ une suite de réels choisis uniformément et indépendamment dans $[0, 1]$. Soit $C(u)$ et $\widehat{C}(u)$ les chemins respectifs sur l'arbre du cas général et du cas idéal. La probabilité que les deux chemins soient identiques avant les $\log_2(N) - \sqrt{\log N}$ premières itérations est d'au moins $1 - o\left(\frac{1}{\log N}\right)$.*

Démonstration. D'après l'Equation (5.13), nous savons que la probabilité que les deux chemins soient identiques avant les $\delta_N = \log_2(N) - \lambda_N$ étapes est donnée par

$$P_{same}(\lfloor \delta_N \rfloor) = \prod_{i=0}^{\lfloor \delta_N \rfloor} 1 - p_{bif}(i),$$

pour un certain λ_N.

Nous savons de plus par le Lemme 10 que

$$p_{bif}(i) \leq \frac{\alpha 2^i}{N - 2^i},$$

avec $\alpha > 0$ constant.

En combinant ces deux lemmes, nous obtenons

$$P_{same}(\lfloor \delta_N \rfloor) \geq \prod_{i=0}^{\lfloor \delta_N \rfloor} 1 - \frac{\alpha 2^i}{N - 2^i}.$$

De plus pour tous les termes de ce produit nous avons l'encadrement trivial

$$1 \leq 2^i < 2^{\delta_N}.$$

En combinant tout ça, nous obtenons

$$P_{same}(\lfloor \delta_N \rfloor) > \left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)^{\delta_N + 1} \tag{5.14}$$

Nous allons utiliser plusieurs développements limités pour parvenir à nos fins. Commençons par ré-écrire le terme de droite de l'Equation (5.14),

$$\left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)^{\delta_N + 1} = \exp\left((\delta_N + 1)\left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)\right).$$

Quand N est grand, si

$$\lim_{N \to \infty} \frac{2^{\delta_N}}{N - 2^{\delta_N}} = 0,$$

alors nous avons

$$\exp\left((\delta_N + 1)\left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)\right) = \exp\left(-(\delta_N + 1)\frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}(1 + o(1))\right).$$

Si

$$\lim_{N \to \infty} \frac{\delta_N 2^{\delta_N}}{N - 2^{\delta_N}} = 0,$$

alors nous avons

$$\exp\left(-(\delta_N+1)\frac{\alpha 2^{\delta_N}}{N-2^{\delta_N}}(1+o(1))\right) = 1-(\delta_N+1)\frac{\alpha 2^{\delta_N}}{N-2^{\delta_N}}+o\left((\delta_N+1)\frac{\alpha 2^{\delta_N}}{N-2^{\delta_N}}\right).$$

Si nous choisissons λ_n de façon à avoir $\lim_{N\to\infty}\delta_N = \infty$ et $\frac{\delta_N 2^{\delta_N}}{N-2^{\delta_N}} = o\left(\frac{1}{\log N}\right)$, alors cette dernière égalité est équivalente à $1 - o\left(\frac{1}{\log N}\right)$.

Le choix de $\lambda_N = \sqrt{\log N}$ vérifie ces hypothèses et nous permet donc de conclure. □

Remarque. $\lambda_N = \sqrt{\log N}$ est aussi en $o(\log N)$ ce qui va nous être utile pour le théorème qui va suivre.

Théorème 11. *Le nombre d'erreurs de prédiction engendrées par l'Algorithme 12 lorsque l'exposant est choisi uniformément dans l'intervalle $\{0\dots N\}$ est asymptotiquement équivalent à $\alpha\log_2 N$, avec $\alpha = \frac{1}{2}\mu(\frac{3}{4}) + \frac{3}{4}\mu(\frac{2}{3})$, où μ est la probabilité de faire une erreur de prédiction sous la distribution stationnaire de la chaîne de Markov associée au prédicteur local (pour plus de détails , voir la section 3.5.7 du chapitre 3).*

Démonstration. Soit l'ensemble $\mathcal{G}_N$ des suites réelles $u \in [0,1]^{\widehat{k}}$ dont les chemins $C(u)$ et $\widehat{C}(u)$ sont identiques pour les $\log_2 N - \sqrt{\log N}$ premières étapes. Si une suite appartient à $\mathcal{G}_N$, alors le cas général diffère du cas idéal en nombre de prédictions sur les instructions de branchement qui lisent les $\sqrt{\log N}$ derniers bits de $n(u)$ et de $\widehat{n}(u)$. Notons $\mathcal{M}_N(u)$ et $\mathcal{I}_N(u)$ respectivement le nombre d'erreurs de prédiction de l'Algorithmes 12 lorsque l'exposant est $n(u)$ et lorsque l'exposant est $\widehat{n}(u)$. Il existe donc pour $u \in \mathcal{G}_N$ une constante $\beta > 0$ telle que

$$|\mathcal{M}_N(u) - \mathcal{I}_N(u)| < \beta\sqrt{\log N}.$$

Si $u \notin \mathcal{G}_N$, alors il existe une constante γ telle que

$$|\mathcal{M}_N(u) - \mathcal{I}_N(u)| < \gamma \log N.$$

Nous pouvons maintenant procéder de la façon suivante :

$$\begin{aligned}
|\mathbb{E}[\mathcal{M}_N(u)] - \mathbb{E}[\mathcal{I}_N(u)]| &\leq \mathbb{E}[|\mathcal{M}_N(u) - \mathcal{I}_N(u)|] \\
&< \beta\sqrt{\log N}\mathbb{P}(u \in \mathcal{G}_N) + \gamma \log N \mathbb{P}(u \notin \mathcal{G}_N) \\
&< \beta\sqrt{\log N}\cdot\left(1 - o\left(\frac{1}{\log N}\right)\right) + \gamma\log N \cdot o\left(\frac{1}{\log N}\right) \\
&< o(\log N)
\end{aligned}$$

Le cas idéal approxime le terme dominant du cas général et nous pouvons donc conclure qu'asymptotiquement le nombre d'erreurs de prédiction est le même que celui qui est donné dans le Théorème 10.

□

Nous venons de discuter du cas de l'exponentiation rapide et nous avons vu comment modifier l'algorithme classique pour l'aider à guider la prédiction de branchement.

Au cours de l'analyse du nombre d'erreurs de prédiction, à l'exécution de ces algorithmes, nous avons distinguer deux cas. Le premier, le cas idéal considère que chaque instruction de branchement d'intérêt a une probabilité fixée d'obtenir chacune de ses issues possibles. Nous avons vu que ce cas se produit lorsque l'exposant était une puissance de 4. Le deuxième, le cas général ne fait aucune considération sur la valeur de l'exposant. Ainsi, pour le cas général la probabilité n'est pas fixée comme précédemment, mais, elle n'est jamais vraiment très loin du cas idéal. Nous avons démontré cette proximité entre les deux cas avec ce dernier théorème.

De plus, nous avons obtenu divers résultats expérimentaux à l'aide de la librairie PAPI et de nos simulations. Ces résultats n'entrent pas en conflit avec notre analyse théorique.

Nous allons maintenant nous intéresser à un nouvel algorithme et nous verrons que les méthodes utilisées dans cette section pourront être en partie réutilisées.

5.3 Recherche dichotomique et variantes

Données : Une séquence T de taille n. Une valeur x dans T.
Résultat : la position de x dans T.

```
1 debut ← 1
2 fin ← n
3 tant que debut < fin faire
4 |   m ← (debut+fin)/2
5 |   si x > T[m] alors
6 |   |   debut ← m + 1
7 |   sinon
8 |   |   fin ← m
9 retourner fin
```

Algorithme 13 : Algorithme de recherche dichotomique classique

Le dernier algorithme que nous étudions dans ce chapitre est l'algorithme de recherche dichotomique. Cet algorithme résout le problème suivant : en entrée nous avons une séquence triée X et une valeur x, le but est d'obtenir en sortie la position de x dans X. Nous donnons une version en pseudo-code de la recherche dichotomique dans l'Algorithme 13. Il est possible que x ne soit pas dans X, dans ce cas nous retournons quand même un indice et il est donc nécessaire de faire un test supplémentaire pour vérifier que l'élément à cet indice est identique à x. L'algorithme est optimal en nombre de comparaisons et fonctionne selon un principe simple. Au début x peut se trouver à n'importe quelle position dans la séquence X. Il est cependant possible de réduire ce nombre de possibilités de moitié. Pour cela, il suffit de comparer x avec la valeur médiane de la séquence qui se situe au milieu de X car cette dernière est triée, notons cette valeur médiane x_m. Nous avons deux possibilités ; soit $x \leq x_m$ et par transitivité, x est située avant x_m dans la séquence X, soit $x > x_m$ et par transitivité, x est située après x_m dans la séquence X. Dans tous les cas, comme m est au milieu de X, nous éliminons la moitié des possibilités. Nous réitérons ainsi de suite sur la

sous-séquence de possibilités restantes jusqu'à ce qu'il ne reste plus qu'une seule possibilité. Comme à chaque itération nous divisons le nombre de possibilités par 2, si n est la taille de la séquence X, alors nous réduisons le nombre de possibilités à 1 en $\lceil \log_2 n \rceil$ étapes. Comme nous faisons une seule comparaison par itération, ce nombre d'étapes est également le nombre de comparaisons effectuées au total.

5.3.1 Identification des problèmes pour la prédiction de branchement et solutions possibles

Données : Une séquence T de taille n. Une valeur x dans T.
Résultat : la position de x dans T.

```
1 debut ← 1
2 fin ← n
3 tant que debut < fin faire
4 |  q1 ← (3debut+fin)/4
5 |  si x > T[q1] alors
6 |  |  debut ← q1 + 1
7 |  sinon
8 |  |  fin ← q1
9 retourner fin
```

Algorithme 14 : Algorithme du BiasedBinarySearch

Données : Une séquence T de taille n. Une valeur x dans T.
Résultat : la position de x dans T.

```
1  debut ← 0
2  fin ← n
3  tant que debut < fin faire
4  |  q1 ← (3debut+fin)/4
5  |  si x > T[q1] alors
6  |  |  m ← (debut+fin)/2
7  |  |  si x > T[m] alors
8  |  |  |  debut ← m + 1
9  |  |  sinon
10 |  |  |  debut ← q1 + 1
11 |  |  |  fin ← m
12 |  sinon
13 |  |  fin ← q1
14 retourner fin
```

Algorithme 15 : Algorithme du SkewSearch

Avec l'algorithme de recherche dichotomique, nous sommes une fois de plus confrontés à un problème de prédiction de branchement. Supposons que nous choisissons uniformément un élément x dans X comme paramètre de la recherche dichotomique. La comparaison effectuée à la ligne 5 de l'Algorithme 13

branche sur l'une des issues possibles de manière équiprobable. Une fois de plus cette instruction permet d'obtenir l'optimalité du nombre de comparaisons mais pose problème au prédicteur car elle est imprédictible. Nous allons donc, comme pour l'exponentiation rapide, chercher un moyen de guider le prédicteur. En conséquence, nous ne ferons plus un nombre de comparaisons optimal mais nous espérons trouver un compromis qui nous permettra de gagner en performance de temps d'exécution en pratique. Cette fois, nous ne pouvons pas faire un déroulement de boucle comme pour l'exponentiation rapide. Dans le cas de l'exponentiation rapide, nous pouvions effectuer les instructions de branchements de manière complètement indépendantes et nous avons créé une dépendance. Dans le cas de la recherche dichotomique, les instructions de branchements après déroulement de la boucle présentent déjà une forte dépendance entre elles. Il est cependant plus facile de transformer l'instruction de la ligne 5 pour rendre cette dernière prédictible. Notre première idée a donc été de remplacer la comparaison avec la médiane par le premier quartile. De cette manière, la probabilité que x soit plus petit que ce premier quartile est proche de $\frac{1}{4}$ [2], ce qui rend le branchement prédictible. Nous donnons dans l'Algorithme 14 une version pseudo-code de cet algorithme que nous nommerons par la suite BiasedBinarySearch. Cette méthode crée cependant un gros déséquilibre ce qui peut augmenter drastiquement le nombre de comparaisons. Nous avons pensé à une autre variante qui crée moins de déséquilibre tout en guidant la prédiction de branchement. Cette solution est un peu entre la solution de déroulement de boucle vue dans la section précédente pour l'exponentiation rapide, et la solution provenant de BiasedBinarySearch. Nous avons introduit un découpage dans BiasedBinarySearch qui est $(\frac{1}{4}, \frac{3}{4})$, c'est-à-dire que si le test de la ligne 5 de l'Algorithme 14 est vrai alors x est positionné dans le premier quart des éléments possibles à l'étape en cours, si le test est faux alors x est positionné dans les trois derniers quarts d'éléments possibles. Ce dernier découpage en environ $\frac{3}{4}$ peut engendrer un plus grand nombre de comparaisons. Notre dernière modification va consister à trouver un découpage plus intéressant. Le découpage que nous proposons est de la forme $\frac{1}{4}, \frac{1}{4}, \frac{1}{2}$. Pour obtenir ce découpage, nous pouvons modifier BiasedBinarySearch en rajoutant une comparaison supplémentaire avec la médiane de X dans le cas où x est plus grand que le premier quartile. En procédant de cette manière nous obtenons alors l'Algorithme 15 que nous nommons SkewSearch.

Remarque. Ce dernier algorithme est un "semi-déroulement" de boucle malin de la recherche dichotomique classique. En effet, il s'agit ici de procéder en un déroulement de la boucle uniquement pour le cas où x est plus petit que la médiane. Enfin, remarquons que le test avec la médiane n'est plus imprédictible dans ce cas, car nous savons alors que x est plus grand que le premier quartile, et intuitivement nous avons donc plus de chance d'être plus grand que la médiane. Le premier test a une probabilité d'environ $\frac{1}{4}$ d'être vrai, tandis que le deuxième test a une probabilité d'environ $\frac{1}{3}$ de l'être. Il pourrait être tentant d'essayer de faire des découpages de types $(\frac{1}{3}, \frac{2}{3})$ ou encore $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, nous avons cependant éviter de le faire pour des raisons de coût de la division qui sont plus important pour des diviseurs qui ne sont pas des puissances de 2 ce qui est le cas de 3.

2. En vérité, nous avons, en premier, choisi de regarder le premier tercile. Le problème de ce choix est qu'il nous faut faire une division par 3 ce qui se révèle être trop couteux. Pour cette raison, nous avons choisi de remplacer ce choix par le quartile qui demande des division par 4, qui est une puissance de 2 et qui se fait rapidement par décalage dans les registres.

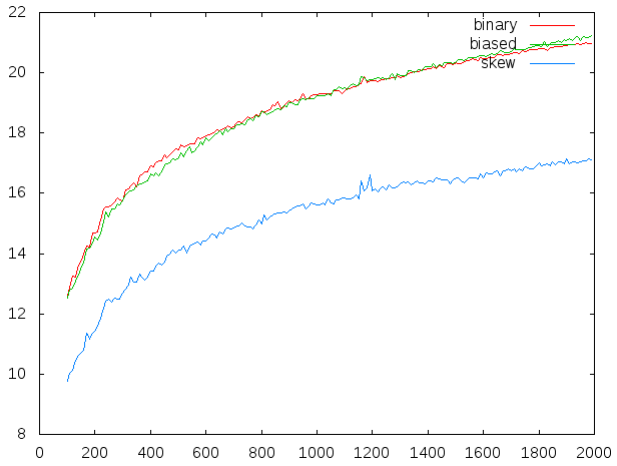

FIGURE 5.6 – Courbes de performances de nos trois versions de la recherche dichotomique. La taille des séquences d'entrées est petite et rentre entièrement dans le cache.

5.3.2 Expérimentations

Nous avons une fois de plus souhaité vérifier les performances de nos algorithmes modifiés. Nous présentons dans cette section deux expériences que nous avons réalisées. La première expérience a été faite sur des implantations en C de nos trois algorithmes de recherche dichotomique. Cette première expérience met en évidence que nos modifications peuvent présenter un intérêt pour certains microprocesseurs. Nous montrons une fois de plus les résultats pour le même microprocesseur à savoir un *Intel(R) Celeron(R) CPU 1037U @ 1.80GHz*, mais nous avons pu vérifier des résultats similaires sur d'autres machines. Nous donnons dans le Listing 12, disponible en Annexe B, les implantations de nos trois algorithmes qui ont été utilisées pour réaliser nos expériences. Dans le détail, nous avons testé chaque implantation pour les mêmes séquences sur différentes tailles. Chacune de ces séquences sont des tableaux triés de flottants simples choisis uniformément dans l'intervalle $[0, 1]$. Pour chaque séquence et pour chaque algorithme, nous avons effectué la recherche, dans le même ordre, de plusieurs éléments appartenant à ce même intervalle.

Remarque. Contrairement aux algorithmes étudiés précédemment, l'influence du cache est importante dans le cadre de la recherche dichotomique. Il nous a donc été nécessaire de prendre quelques précautions à ce sujet.

Par la suite les résultats que nous montrons proviennent du cas où ces implantations ont été compilées sans optimisation. Nous avons regroupé nos résultats sur plusieurs graphiques. Le premier graphique est donné dans la Figure 5.6 et concerne les cas où la séquence d'entrée est petite et peut tenir entièrement dans le cache. C'est un cas intéressant qui nous permet d'observer l'influence de la prédiction de branchement sur les performances de ces algorithmes sans que l'influence du cache ne vienne bruiter ces résultats. Pour ce cas nous observons que SKEWSEARCH a les meilleures performances tandis que les deux autres ont des performances relativement identiques. En particulier, nous observons un gain

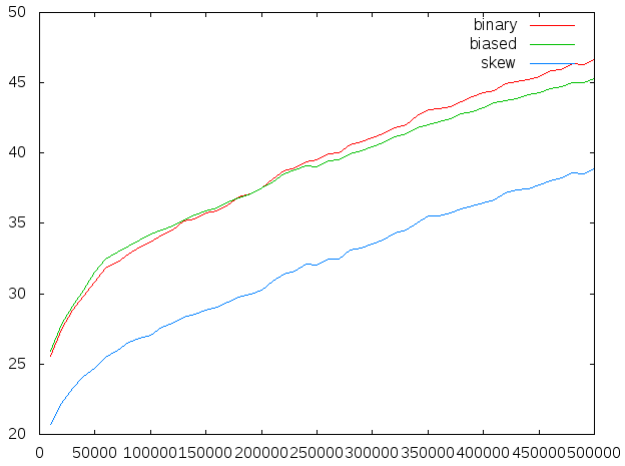

FIGURE 5.7 – Courbes de performances de nos trois versions de la recherche dichotomique. La taille des séquences d'entrées est de taille moyenne et rentre entièrement dans les caches de plus bas niveaux.

pour SKEWSEARCH en tout point supérieur de 22% à la recherche dichotomique classique sur ce graphe.

La Figure 5.7 donne le résultat obtenu pour des séquences de tailles moyennes. Les séquences de tailles moyennes sont influencées par le cache mais uniquement pour les niveaux les plus proches du microprocesseur. Nous observons des résultats similaires au cas où les tailles sont petites. Le gain est cependant moins important entre SKEWSEARCH et la recherche dichotomique classique qui est cette fois en tout point supérieur à 19%.

La Figure 5.8 donne le résultat obtenu pour des séquences de grandes tailles. Les séquences de grandes tailles sont lourdement influencées par les effets de cache puisque ces dernières n'entrent pas dans le cache L3. Nous observons alors qu'il semble exister une taille critique pour laquelle l'algorithme le plus performant dans ce cas est le BIASEDBINARYSEARCH avec des gains allant environ jusqu'à 30% par rapport à la recherche dichotomique classique. Pour ce dernier cas, BIASEDBINARYSEARCH reste un cas intéressant et une analyse de l'influence du cache pour comprendre la différence de performance pour ce cas avec SKEWSEARCH semble nécessaire.

La recherche dichotomique est un algorithme qui est implanté dans la librairie standard du langage C. Nous pourrions avoir envie de vérifier si nos modifications apportées à la fonction *bsearch* permettent d'obtenir des gains en temps d'exécutions. Le problème de cette fonction et qu'elle utilise le pattern *Template* en laissant le choix à l'utilisateur de la fonction de comparaison dont il donne l'adresse. La fonction ainsi donnée en paramètre ne peut pas être transformée en fonction *inline* et va ainsi s'exécuter lentement. De ce fait, le coût de la comparaison dans la fonction *bsearch* est très coûteuse et il est préférable de minimiser le nombre de comparaisons en préservant l'équilibre apporté par l'implantation standard. Cependant, il existe un ensemble de surcharges d'implantations de la recherche dichotomique en Java qui utilisent les fonctions de comparaisons standards pour les types primitifs qui sont peu coûteuses. Notre

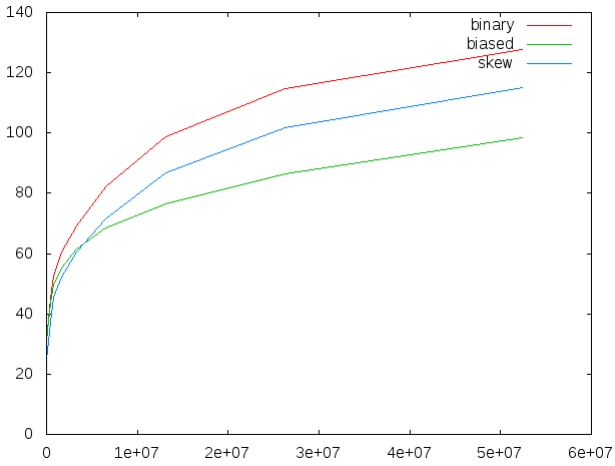

FIGURE 5.8 – Courbes de performances de nos trois versions de la recherche dichotomique. La taille des séquences d'entrées est de grande taille et ne rentre pas entièrement dans le cache de plus bas niveau.

deuxième expérimentation consiste donc à comparer une de ces fonctions de la librairie standard Java avec nos versions modifiées et qui guident la prédiction de branchement. Nous avons choisi de réaliser cette expérience lorsque la séquence est un tableau de *double* et l'élément à rechercher est également un *double*. Pour réaliser ces comparaisons de performances, nous avons utilisé la librarie *jmh* [1] qui permet de créer des benchmarks. Les codes sources utilisés pour réaliser ces benchmarks sont disponibles en téléchargement [6].

L'expérimentation a été effectuée sur un microprocesseur *Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz* dans un environnement *GNU/Linux*. Nous avons fait tourner ces benchmarks pour comparer les implantations sur des tableaux de petites et moyennes tailles. L'implantation de la recherche dichotomique classique est ici directement celle utilisée dans la librairie standard Java qui porte le nom *binarySearch()*. Nous retrouvons des résultats similaires à ceux que nous avons obtenus avec nos implantations en C. Ces résultats sont donnés sous forme de courbes de performances dans la Figure 5.9. Nous avons donc potentiellement, sur des machines avec des processeurs récents, une implantation d'une variante de la recherche dichotomique qui se montre plus efficace sur des tableaux de tailles moyennes que les fonctions de la librairie standard de Java. Nous avons testé ces fonctions uniquement sur le cas particulier que nous avons décrit ci-dessus. Il est donc trop tôt pour conclure et il faudrait très certainement faire tout une batterie de benchmarks, du même niveau de ceux qu'a probablement utilisés l'entreprise Oracle avant de choisir d'implanter TimSort dans leur librairie standard.

5.3.3 Analyse pour des prédicteurs locaux

Comme il a été vu dans la Section 5.2, nous cherchons à utiliser le théorème ergodique afin d'obtenir une bonne estimation asymptotique du nombre d'erreurs de prédiction. Nous devons donc commencer par calculer le nombre de fois

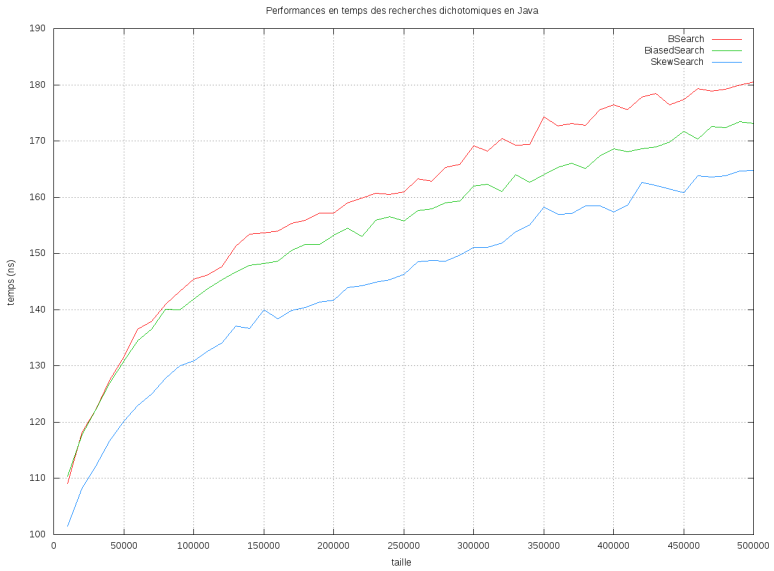

FIGURE 5.9 – Courbes de performances de temps mesurées sur les implantations Java des variantes de la recherche dichotomique.

que chaque condition est exécutée en moyenne, dans nos différents algorithmes. Nous considérons le modèle où toutes les sorties possibles sont équiprobables. Autrement dit, l'indice de l'élément à rechercher dans le tableau est choisi selon une loi uniforme dans $[n]$. Une estimation du terme dominant du nombre d'exécutions d'une instruction de branchement peut être obtenue en utilisant la version suivante du Théorème maître de Roura [39], qui a été simplifié pour nos besoins :

Théorème 12. *Soit une suite $(F_n)_{n\in\mathbb{N}}$ définie pour tout $n > 0$ par l'équation de récurrence*

$$F_n = Bn + \sum_{d=1}^{D} F_{S_{d,n}}, \tag{5.15}$$

avec $S_{d,n} = z_d \cdot n + \mathcal{O}(1)$ et D et B deux constantes positives. Si nous avons $z_d > 0$ et $\sum_{d=1}^{D} z_d = 1$, alors nous avons

$$F_n \sim -\frac{Bn \log n}{\sum_{d=1}^{D} z_d \log z_d}. \tag{5.16}$$

Démonstration. Nous sommes dans un cas particulier du théorème 2.3 de l'article de Roura [39]. Nous avons $w_d = 1$ et $r_{d,n} = 0$ pour tout $1 \geq d \geq D$ et vérifions donc la condition que $\sum_{d=1}^{D} |r_{d,n}| = \mathcal{O}(n^{-1})$. Nous avons $z_d > 0$ et $\sum_{d=1}^{D} z_d = 1$ et $s_{d,n} = \mathcal{O}(1)$ ce qui implique que $\frac{\sum_{d=1}^{D} |s_{d,n}|}{n} = \mathcal{O}(n^{-1})$. La fonction t_n est simplement égale à $n = 1 \times n^1 \times \log^0 n \times 1$. Nous sommes alors dans le cas (2.1) ce qui nous donne directement le résultat annoncé. □

Nombre d'itération en moyenne

Nous pouvons appliquer le théorème de Roura afin de déterminer le nombre de fois où chaque instruction de branchement est exécutée.

Proposition 19. *L'espérance du nombre d'exécution $I_c(n)$ de la ligne 5 de l'Algorithme 13 est asymptotiquement équivalent à $I_c(n) \sim \log_2 n$.*

Démonstration. Cette espérance $I_c(n)$ satisfait la relation

$$I_c(0) = 0;\ I_c(n) = 1 + \frac{\left\lfloor \frac{n}{2} \right\rfloor}{n} I_c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \frac{\left\lceil \frac{n}{2} \right\rceil}{n} I_c\left(\left\lceil \frac{n}{2} \right\rceil\right).$$

Le Théorème 12 est applicable en prenant $J_c(n) = nI_c(n)$ et nous donne directement le résultat. □

Proposition 20. *L'espérance du nombre d'exécution $I_b(n)$ de la ligne 5 de l'Algorithme 14 est asymptotiquement équivalent à $I_b(n) \sim \lambda \log n$, avec $\lambda = \frac{4}{4\log 4 - 3\log 3} \approx 1.78$.*

Démonstration. Cette espérance $I_b(n)$ satisfait la relation

$$I_b(0) = 0;\ I_b(n) = 1 + \frac{\left\lfloor \frac{n}{4} \right\rfloor}{n} I_b\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \frac{\left\lceil \frac{3n}{4} \right\rceil}{n} I_b\left(\left\lceil \frac{3n}{4} \right\rceil\right).$$

Le Théorème 12 est applicable pour $J_b(n) = nI_b(n)$ et nous donne directement le résultat. □

Proposition 21. *L'espérance du nombre d'exécutions $I_s(n)$ de la ligne 5 de l'Algorithme 15 est asymptotiquement équivalent à $I_s(n) \sim \frac{2}{3}\log_2 n$. L'espérance du nombre d'exécution $I'_s(n)$ de la ligne 7 est asymptotiquement équivalent à $I_s(n) \sim \frac{1}{2}\log_2 n$. La somme de ces deux termes nous donne un équivalent asymptotique de la totalité des comparaisons effectuées en moyenne par cet algorithme qui est approximativement égal à $\frac{7}{6}\log_2 n$.*

Démonstration. L'espérance du nombre d'itérations $I_s(n)$ de SkewSearch satisfait les relations

$$\begin{aligned}
&I_s(0) = 0;\ I_s(n) = 1 + \frac{\alpha_n}{n} I_s(\alpha_n) + \frac{\beta_n}{n} I_s(\beta_n) + \frac{\delta_n}{n} I_s(\delta_n)\\
&\alpha_n = \left\lfloor \frac{n}{4} \right\rfloor\\
&\beta_n = \left\lfloor \frac{n}{2} \right\rfloor - \left\lfloor \frac{n}{4} \right\rfloor\\
&\delta_n = \left\lceil \frac{n}{2} \right\rceil.
\end{aligned}$$

Pour ces mêmes α_n,β_n et δ_n, nous avons

$$I'_s(0) = 0;\ I'_s(n) = \frac{3}{4} + \frac{\alpha_n}{n} I'_s(\alpha_n) + \frac{\beta_n}{n} I'_s(\beta_n) + \frac{\delta_n}{n} I'_s(\delta_n)$$

Le Théorème 12 est applicable pour ces deux systèmes en posant $J_s(n) = nI_s(n)$ et $J'_s(n) = nI'_s(n)$ et nous donne directement le résultat. □

Remarque. Comme nous l'avons anticipé lors de la présentation des algorithmes BiasedBinarySearch et SkewSearch, ces derniers font chacun plus de comparaisons en moyenne que la recherche dichotomique classique. On remarque également que SkewSearch est bien celui qui génère un plus faible déséquilibre, ce dernier fait environ 5% de comparaisons en moins que BiasedBinarySearch.

Nous allons maintenant effectuer un raisonnement similaire à ce qui a été fait au cours de la Section 5.2. Dans un premier temps nous allons faire l'analyse de nos algorithmes en considérant un cas idéal, où la probabilité qu'une instruction de branchement soit vraie est donnée par une loi de Bernoulli de paramètre p pour chaque itération. Dans la pratique, il y a de légère fluctuation autour de cette valeur et nous montrerons qu'elles sont négligeables et que l'analyse dans le cas idéal est suffisant pour obtenir le comportement asymptotique de nos algorithmes. C'est ce que nous montrerons dans un deuxième temps.

Cas idéal

Le cas idéal correspond au cas où l'on considère qu'il n'y a pas de fluctuation de la probabilité que les différentes instructions de branchements soient vraies. Plus concrètement, nous allons considérer qu'à chaque itération la probabilité que l'instruction de branchement soit vraie à la ligne 5 de l'Algorithme 13 est $\frac{1}{2}$, celle de l'instruction de la ligne 5 de l'Algorithme 14 est $\frac{1}{4}$, celle de la ligne 5 de l'Algorithme 15 est $\frac{1}{4}$ et celle de la ligne 7 de ce même algorithme est $\frac{1}{3}$. Pour ce cas, la marche aléatoire du prédicteur est une chaîne de Markov qui converge rapidement vers sa seule distribution stationnaire et nous pouvons appliquer le Théorème 4 Ergodique du Chapitre 3.

Théorème 13. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}_{\mathcal{I}}[M_c(n)]$ du nombre d'erreurs de prédiction produites à l'exécution de l'Algorithme 13 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\mu\left(\frac{1}{2}\right)\log_2(n).$$

Démonstration. La Proposition 19 nous donne l'équivalent asymptotique de l'espérance du nombre $I_c(n)$ de fois que l'instruction de la ligne 5 de l'Algorithme 13 est exécutée. Comme nous considérons dans ce cas idéal que cette instruction est vraie avec probabilité $\frac{1}{2}$, à chaque itération à l'état stationnaire il y a une probabilité

$$\mu\left(\frac{1}{2}\right)$$

d'obtenir une erreur de prédiction. En appliquant le Théorème 4 ergodique nous obtenons

$$\mu\left(\frac{1}{2}\right)I_c(n),$$

ce qui correspond au résultat annoncé. □

Avec les mêmes raisonnements, nous pouvons trouver des théorèmes similaires pour BiasedBinarySearch et SkewSearch.

Théorème 14. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}_{\mathcal{I}}[M_b(n)]$ du nombre d'erreurs de prédiction produites à l'exécution de l'Algorithme 14 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\frac{4\mu\left(\frac{1}{4}\right)}{4\log 4 - 3\log 3}\log n$$

Démonstration. La preuve est similaire à celle du théorème précédent qui ne change uniquement que sur les constantes choisies. □

Théorème 15. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}_{\mathcal{I}}[M_s(n)]$ du nombre d'erreurs de prédiction produites à l'exécution de l'Algorithme 15 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\left(\frac{2}{3}\mu\left(\frac{1}{4}\right) + \frac{1}{2}\mu\left(\frac{1}{3}\right)\right)\log_2 n$$

Démonstration. La preuve est similaire aux preuves précédentes mais il faut faire attention car nous avons deux instructions de branchement. La Proposition 21 nous donne le nombre de fois que les instructions des lignes 5 et 7 sont chacune exécutées. Pour l'instruction de la ligne 5, nous avons quand n est grand de l'ordre de $\frac{2}{3}\log_2 n$ itérations effectuées. Pour cette ligne dans le cas idéal, nous avons dans l'état stationnaire une probabilité $\mu\left(\frac{1}{4}\right)$ d'avoir une erreur de prédiction. En appliquant le Théorème 4 nous avons que cette ligne génère en moyenne de l'ordre de

$$\frac{2}{3}\mu\left(\frac{1}{4}\right)\log_2 n$$

erreurs de prédiction. En procédant de la même façon pour la ligne 7, nous obtenons en moyenne de l'ordre de

$$\frac{1}{2}\mu\left(\frac{1}{3}\right)\log_2 n$$

erreurs de prédiction. C'est en faisant la somme de ces deux contributions que nous obtenons le résultat annoncé. □

Cas général

Dans le cas général, nous avons des fluctuations de la probabilité à chaque itération. Cependant pour les premières itérations nous sommes très proches du cas idéal. Nous pouvons une fois de plus appliquer une méthode de couplage similaire à ce que nous avons fait lors de la section précédente avec l'algorithme de l'exponentiation rapide.

Soit un *arbre de décomposition* τ associé à chacun des algorithmes de recherche, qui est définit comme suit. Si l'entrée est de taille n, sa racine est étiquetée par la paire $(0, n)$, et chaque nœud correspond aux valeurs possibles d et f (respectivement le début et la fin de l'intervalle) lors d'une itération de l'algorithme. Les feuilles sont les paires (i, i), pour $i \in [1, n]$ et correspondent

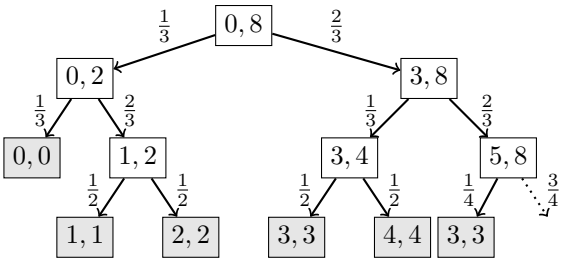

FIGURE 5.10 – Arbre de décomposition de BIASEDBINARYSEARCH pour $n = 8$.

aux différentes sorties possibles de l'algorithme. Les nœuds (d, f) et (d', f') sont adjacents si d et f peuvent être modifiés en d' et f' au cours de l'itération de la boucle. Une arrête entre ces deux nœuds est étiquetée par la probabilité $\frac{f'-d'+1}{f-d+1}$, qui est la probabilité que la modification ait lieu dans le modèle que nous considérons. Les enfants d'un nœud interne sont alors ordonnés par leur valeur d de la gauche vers la droite dans l'arbre. Un exemple d'arbre de décomposition pour BIASEDBINARYSEARCH peut être vu sur la Figure 5.10. Soit $u = (u_0, u_1, \dots)$, une suite infinie de valeurs de $[0, 1]$ choisies uniformément et indépendamment. Le chemin de la racine à une feuille dans τ associé à u est $\mathbf{Path}_n(\tau, u)$, où pour l'étape i, nous allons à gauche si u_i est plus petit que la probabilité de l'arrête allant vers le fils gauche et à droite dans le cas contraire. Soit $L_n(\tau, u)$ la longueur de $\mathbf{Path}_n(\tau, u)$. Soit également, $\mathbf{Path}_n(\mathcal{I}, u)$ le chemin suivi par la valeur u dans l'arbre idéal (et infini) $\mathcal{I}$ où l'on va à gauche avec probabilité $\frac{1}{4}$ et à droite avec probabilité $\frac{3}{4}$. Le résultat suivant est vérifié.

Lemme 11. *La probabilité que les chemins* $\mathbf{Path}_n(\tau, u)$ *et* $\mathbf{Path}_n(\mathcal{I}, u)$ *de la recherche dichotomique classique diffèrent sur l'une des* $L_n(\tau, u) - \sqrt{\log n}$ *premières étapes est* $o\left(\frac{1}{\log n}\right)$.

Démonstration. Soit (d, f) un nœud de τ qui n'est pas une feuille, et soit $t = f - d + 1$. La probabilité d'aller à gauche à la prochaine étape est donnée par $p(d, f) = \frac{m-d+1}{t}$, où $m = \left\lfloor \frac{d+f}{2} \right\rfloor$. Rappelons que pour tout $x \in \mathbb{R}$, nous avons $x-1 < \lfloor x \rfloor \leq x$. Avec cette inégalité, il est facile de montrer que $|p(d, f) - \frac{1}{2}| \leq \frac{\alpha}{t}$ pour une certaine constante $\alpha > 0$ et ainsi, la probabilité que les deux chemins diffèrent à cette étape est au plus $\frac{\alpha}{t}$. Soit $t_g = m - d + 1$ et $t_d = t - t_g = f - m$. Comme (d, f) n'est pas une feuille on a forcément $t \geq 2$. Il est alors facile de montrer que $\frac{1}{4} \leq \frac{t_g}{t}, \frac{t_d}{t} \leq \frac{3}{4}$. Ainsi pour t' la taille de l'intervalle du prochain nœud atteint à l'étape suivante, on a $t' = c \times t$ avec $c \in \left[\frac{1}{4}, \frac{3}{4}\right]$. Soit (d', f') le nœud atteint après $L_n(u) - \lambda_n$ étapes. Comme le dernier nœud atteint correspond à un intervalle de taille 1 et qu'il reste λ_n étapes. Nous avons en utilisant l'observation précédente,

$$t' = f' - d' + 1 \geq \left(\frac{4}{3}\right)^{\lambda_n}.$$

Par conséquent, tout nœud sur le chemin induit par u se trouvant à une distance $L_n(u) - \lambda_n$ sera donc plus grand que $\gamma_n = \left(\frac{4}{3}\right)^{\lambda_n}$. De plus pour n suffisam-

ment grand, on a $L_n(u) \leq \log_{4/3} n$. Ainsi la probabilité que les deux chemins $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ soient identiques pour les $L_n(u) - \lambda_n$ premières étapes est au moins $\left(1 - \frac{\alpha}{\gamma_n}\right)^{\log_{4/3} n - \lambda_n}$. Nous avons alors,

$$\begin{aligned}\left(1 - \frac{\alpha}{\gamma_n}\right)^{\log_{4/3} n - \lambda_n} &= \exp\left((\log_{4/3} n - \lambda_n) \log\left(1 - \frac{\alpha}{\gamma_n}\right)\right) \\ &= \exp\left(-\frac{\alpha}{\gamma_n}(\log_{4/3} n - \lambda_n)(1 + o(1))\right) \\ &= 1 - \frac{\alpha}{\gamma_n}(\log_{4/3} n - \lambda_n) + o\left(\frac{\alpha}{\gamma_n}(\log_{4/3} n - \lambda_n)\right).\end{aligned}$$

Si on choisit maintenant $\lambda_n = o(\log n)$ et $\lambda_n \to \infty$ quand n tend vers l'infini, on a

$$\begin{aligned}\left(1 - \frac{\alpha}{\gamma_n}\right)^{\log_{4/3} n - \lambda_n} &= 1 - \frac{\alpha}{\gamma_n} \log_{4/3} n + o\left(\frac{\log n}{\gamma_n}\right) \\ &= 1 - o\left(\frac{1}{\log n}\right),\end{aligned}$$

ce qui conclut la preuve puisque $\lambda_n = \sqrt{\log n}$ est un choix de fonction qui vérifie les conditions précédentes. □

Lemme 12. *La probabilité que les chemins* $\mathbf{Path}_n(\tau, u)$ *et* $\mathbf{Path}_n(\mathcal{I}, u)$ *de* BiasedBinarySearch *diffèrent sur l'une des* $L_n(\tau, u) - \sqrt{\log n}$ *premières étapes est* $o\left(\frac{1}{\log n}\right)$.

Démonstration. Soit (d, f) un nœud de τ qui n'est pas une feuille, et soit $t = f - d + 1$. La probabilité d'aller à gauche à la prochaine étape est donnée par $p(d, f) = \frac{m-d+1}{t}$, où $m = \left\lfloor \frac{3d+f}{4} \right\rfloor$. Rappelons que pour tout $x \in \mathbb{R}$, nous avons $x - 1 < \lfloor x \rfloor \leq x$. Avec cette inégalité, il est facile de montrer que $|p(d, f) - \frac{1}{4}| \leq \frac{\alpha}{t}$ pour une certaine constante $\alpha > 0$ et ainsi, la probabilité que les deux chemins diffèrent à cette étape est au plus $\frac{\alpha}{t}$. Soit $t_g = m - d + 1$ et $t_d = t - t_g = f - m$. Comme (d, f) n'est pas une feuille on a forcément $t \geq 2$. Il est alors facile de montrer que $\frac{1}{8} \leq \frac{t_g}{t}, \frac{t_d}{t} \leq \frac{7}{8}$. Ainsi pour t' la taille de l'intervalle du prochain nœud atteint à l'étape suivante, on a $t' = c \times t$ avec $c \in \left[\frac{1}{8}, \frac{7}{8}\right]$. Soit (d', f') le nœud atteint après $L_n(u) - \lambda_n$ étapes. Comme le dernier nœud atteint correspond à un intervalle de taille 1 et qu'il reste λ_n étapes. Nous avons en utilisant l'observation précédente,

$$t' = f' - d' + 1 \geq \left(\frac{8}{7}\right)^{\lambda_n}.$$

Par conséquent, tout nœud sur le chemin induit par u se trouvant à une distance $L_n(u) - \lambda_n$ sera donc plus grand que $\gamma_n = \left(\frac{8}{7}\right)^{\lambda_n}$. De plus pour n suffisamment grand, on a $L_n(u) \leq \log_{8/7} n$. Ainsi la probabilité que les deux chemins $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ soient identiques pour les $L_n(u) - \lambda_n$ premières

étapes est au moins $\left(1-\frac{\alpha}{\gamma_n}\right)^{\log_{8/7} n-\lambda_n}$. Nous avons alors,

$$\begin{aligned}\left(1-\frac{\alpha}{\gamma_n}\right)^{\log_{8/7} n-\lambda_n} &= \exp\left((\log_{8/7} n-\lambda_n)\log\left(1-\frac{\alpha}{\gamma_n}\right)\right)\\ &= \exp\left(-\frac{\alpha}{\gamma_n}(\log_{8/7} n-\lambda_n)(1+o(1))\right)\\ &= 1-\frac{\alpha}{\gamma_n}(\log_{8/7} n-\lambda_n)+o\left(\frac{\alpha}{\gamma_n}(\log_{8/7} n-\lambda_n)\right).\end{aligned}$$

Si on choisit maintenant $\lambda_n = o(\log n)$ et $\lambda_n \to \infty$ quand n tend vers l'infini, on a

$$\begin{aligned}\left(1-\frac{\alpha}{\gamma_n}\right)^{\log_{8/7} n-\lambda_n} &= 1-\frac{\alpha}{\gamma_n}\log_{8/7} n+o\left(\frac{\log n}{\gamma_n}\right)\\ &= 1-o\left(\frac{1}{\log n}\right),\end{aligned}$$

ce qui conclut la preuve puisque $\lambda_n = \sqrt{\log n}$ est un choix de fonction qui vérifie les conditions précédentes. □

Nous pouvons maintenant démontrer que l'algorithme de la recherche dichotomique classique et BiasedBinarySearch produisent un nombre d'erreurs de prédiction similaire au cas idéal.

Théorème 16. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}[M_b(n)]$ d'erreurs de prédiction produites à l'exécution de l'Algorithme 13 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\mu\left(\frac{1}{2}\right)\log_2(n).$$

Démonstration. Soit $M_n(u)$ le nombre d'erreurs de prédiction produites par l'algorithme en suivant le chemin induit par u, et soit $I_n(u)$ le nombre d'erreurs de prédiction produites par l'algorithme idéalisé en suivant les $L_n(u)$ premières étapes dans l'arbre idéal $\mathcal{I}$. Soit $\mathcal{G}_n$ l'ensemble de $u \in [0,1]^+$ tel que $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ sont égaux pour les premières $L_n(u) - \lambda_n$ étapes. Remarquons que si $u \in \mathcal{G}_n$, alors nous avons $|M_n(u) - I_n(u)| \leq \lambda_n$ comme ils ne peuvent différer que sur les λ_n dernières étapes. Si $u \notin \mathcal{G}_n$, alors nous avons forcément $|M_n(u) - I_n(u)| \leq L_n(u) \leq \beta \log n$ pour une constante β bien choisie. En utilisant le Lemme 11 et en choisissant $\lambda_n = \sqrt{\log n}$ nous avons

$$\begin{aligned}|\mathbb{E}[M_n] - \mathbb{E}[I_n]| &\leq \lambda_n \mathbb{P}(u \in \mathcal{G}_n) + \beta \log n \mathbb{P}(u \notin \mathcal{G}_n)\\ &= \lambda_n \cdot \left(1 - o\left(\frac{1}{\log n}\right)\right) + \beta \log n \cdot o\left(\frac{1}{\log n}\right)\\ &= o(\log n)\end{aligned}$$

Ainsi $\mathbb{E}[M_n] \sim \mathbb{E}[I_n]$, et le premier ordre de l'équivalent asymptotique du modèle idéalisé est identique pour le cas réel qui est donné dans le théorème 13. □

Le même raisonnement peut s'appliquer pour BiasedBinarySearch.

Théorème 17. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}[M_b(n)]$ d'erreurs de prédiction produites à l'exécution de l'Algorithme 14 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\frac{4\mu\left(\frac{1}{4}\right)}{4\log 4 - 3\log 3}\log n$$

Pour l'analyse de SkewSearch, nous devons faire attention à l'arité de l'arbre. Pour chaque itération nous avons 3 choix possibles et l'arbre est donc ternaire. Pour construire un chemin dans cet arbre à l'étape i, nous considérons les probabilités $p_g(i), p_m(i)$ et $p_d(i)$ d'aller respectivement à gauche, au milieu ou à droite. Pour une suite u, nous allons ensuite à gauche si $u_i < p_g(i)$, au milieu si $p_g(i) \leq u_i < p_g(i) + p_m(i)$ et à droite sinon.

Lemme 13. *La probabilité que les chemins $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ de* BiasedBinarySearch *diffèrent sur l'une des $L_n(\tau, u) - \sqrt{\log n}$ premières étapes est $o\left(\frac{1}{\log n}\right)$.*

Démonstration. La preuve est similaire à celle des Lemmes 11 et 12. Nous devons regarder la probabilité qu'à l'étape i nous ayons une bifurcation. C'est le cas si u_i se trouve dans l'union d'intervalles où les choix du cas idéal et du cas général sont différents. Il s'agit donc de faire la somme d'inégalités similaires à celles déjà vu auparavant et il existe donc une constante $\alpha > 0$ telle que la probabilité que les chemins soient différents à l'étape i est plus petite que $\frac{\alpha}{t}$ avec t le nombre d'éléments restants à traiter de l'étape. De plus comme le nombre d'éléments restants est d'au moins deux, nous pouvons encore une fois trouver une constante $c \in]0, 1[$ telle que pour un nœud de hauteur plus grande que λ_n, nous avons que le nombre d'éléments restants à traiter pour ce nœud est au moins

$$\gamma_n = c^{-n}.$$

En procédant à une suite de développements limités similaires aux preuves des Lemmes 11 et 12, nous arrivons à la même conclusion en choisissant $\lambda_n = \sqrt{\log n}$. □

Nous pouvons directement, comme les fois précédentes, déduire le prochain théorème de ce dernier lemme.

Théorème 18. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}[M_s(n)]$ d'erreurs de prédiction produites à l'exécution de l'Algorithme 15 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\left(\frac{2}{3}\mu\left(\frac{1}{4}\right) + \frac{1}{2}\mu\left(\frac{1}{3}\right)\right)\log_2 n$$

5.3.4 Analyse pour le prédicteur global de SkewSearch

Cette section a pour but de donner des pistes concernant le comportement du prédicteur global comme évoqué dans le Chapitre 3 à la Section 3.5. Il s'agit donc

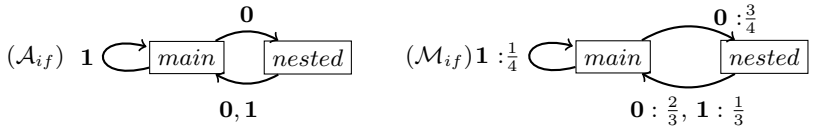

FIGURE 5.11 – À gauche, l'automate $\mathcal{A}_{if}$. À droite, son automate de Markov associée $\mathcal{M}_{if}$ avec les probabilités de transition $\mathbb{P}(1 \mid main) = \frac{1}{4}$, $\mathbb{P}(0 \mid main) = \frac{3}{4}$, $\mathbb{P}(0 \mid nested) = \frac{2}{3}$ and $\mathbb{P}(1 \mid nested) = \frac{1}{3}$.

d'un prédicteur avec une table d'histoires dont nous considérons qu'à chaque histoire est associée un prédicteur local 2-bits. Nous allons par la suite faire l'analyse du nombre d'erreurs de prédiction lors de l'exécution de SKEWSEARCH. Pour simplifier le problème, nous allons considérer que nous sommes dans le cas idéal que nous avons étudié à la section précédente. Comme en pratique les processeurs disposent également de prédicteurs de boucles, nous ne considéreront pas les branchements provenant de la boucle principale de l'Algorithme 15. Nous encodons le résultat d'une condition vraie par un 1 et dans le cas contraire par un 0. La trace de l'exécution de l'algorithme est donc donnée par un mot binaire $w = w_1 \dots w_N$. Par la suite, nous nommons par **main** l'instruction de branchement de la ligne 5 et par **nested** l'instruction de branchement de la ligne 7. Il est possible de garder une trace de l'instruction de branchement en cours d'exécution en utilisant un simple automate déterministe $\mathcal{A}_{if}$ constitué de deux états et qui est représenté dans la Figure 5.11 à gauche. Dans notre modèle, **main** est vraie avec probabilité $\frac{1}{4}$ et **nested** l'est avec probabilité $\frac{1}{3}$. Il est alors possible d'obtenir une chaîne de Markov associée à $\mathcal{A}_{if}$ en utilisant une méthode similaire à l'association d'une chaîne à un prédicteur (voir Figure 5.11 à droite). Un calcul direct permet de démontrer que le vecteur stationnaire π_{if} satisfait $\pi_{if}(\mathbf{main}) = \frac{4}{7}$ et $\pi_{if}(\mathbf{nested}) = \frac{3}{7}$. Nous considérons donc à partir de maintenant que la variable histoire h n'est modifiée que par les résultats des conditions **main** et **nested**. Nous supposerons pour la suite que la taille ℓ de cette variable est paire. Le mot 0^ℓ représente l'histoire consistant uniquement en des 0. Lorsqu'un test est effectué au temps t, le prédicteur utilise alors l'entrée dans sa table en position h_t pour faire une prédiction, avec h_t l'histoire au temps t.

Pour suivre l'évolution de l'algorithme au temps $t + 1$, nous avons donc seulement besoin de garder des traces de la table d'histoires T_t, l'histoire h_t et l'instruction de branchement exécutée IF_t. Cette dernière connaissance est nécessaire et suffisante pour calculer la probabilité d'avoir la sortie 0 ou 1 à la fin de l'instruction de branchement à l'étape t. À tout moment t, nous avons donc un ensemble d'états possibles pour chaque valeurs possibles de h_t, T_t et IF_t. Après l'exécution de la prochaine instruction de branchement, il y a eu une transition vers un nouvel état de ce même ensemble qui correspond au valeur prise par h_{t+1}, T_{t+1} et IF_{t+1}. Tout cela définit une chaîne de Markov $\mathcal{M}_{up}$ des modifications de la table d'histoires. À partir de $\mathcal{M}_{up}$, il est possible d'estimer théoriquement le nombre moyen d'erreurs de prédiction en utilisant le théorème ergodique, comme pour le cas des prédicteurs locaux. Le principal problème avec cette approche est que le calcul de π_{up} prend typiquement $\mathcal{O}(m^3)$ opérations, où m est le nombre d'états dans $\mathcal{M}_{up}$. Comme le nombre d'états est exponentiel en ℓ, le calcul est alors impraticable pour une longueur de l'histoire raisonnable

(comme par exemple $\ell > 6$) même en retirant les états non-atteignables. Par la suite, nous exploiterons donc la structure particulière de $\mathcal{M}_{up}$ pour calculer directement le nombre d'erreurs de prédiction.

Remarque. Dans la Figure 5.11, l'automate admet un mot synchronisant trivial qui est "1". Après la lecture de ce mot nous remarquons en effet que l'instruction de branchement qui suit est toujours **main**.

Définition 16. Nous définissons par $\mathcal{H}_{\mathbf{main}} \subset \mathbb{B}^\ell$ l'ensemble des histoires dont nous savons avec certitude que si $h_t = h$ alors $IF_t = \mathbf{main}$.

De façon similaire, nous définissons par $\mathcal{H}_{\mathbf{nested}} \subset \mathbb{B}^\ell$ l'ensemble des histoires dont nous savons avec certitude que si $h_t = h$ alors $IF_t = \mathbf{nested}$.

Exemple. *Pour $\ell = 8$, nous avons "00101011" qui appartient à l'ensemble $\mathcal{H}_{\mathbf{main}}$, tandis que, "10101010" appartient à l'ensemble $\mathcal{H}_{\mathbf{nested}}$.*

Remarque. D'après la remarque précédente, il est évident que le seul mot qui n'appartient à aucun de ces deux ensembles est le mot 0^ℓ. De plus, par construction, ces deux ensembles sont disjoints.

En conséquence de cette dernière remarque, chaque entrée $h \neq 0^\ell$ dans la table T se comporte comme un prédicteur local 2-bits saturé avec probabilité fixée à $\frac{1}{4}$ (respectivement $\frac{1}{3}$) pour une histoire associée à *main* (respectivement *nested*). C'est donc l'histoire $h = 0^\ell$ qui concentre les différences entre le prédicteur global et les prédicteurs locaux dans ce cas. Cet automate peut être transformé en une chaine de Markov, et le théorème Ergodique permet d'obtenir une estimation précise du nombre moyen d'erreurs de prédiction. C'est un cas particulier qui provient de l'existence de ce mot synchronisant. En suivant cette idée, nous obtenons le résultat du Théorème 19.

Le prédicteur global suit une marche aléatoire qui peut être représentée sous la forme d'une grande chaîne de Markov comme nous l'avons évoqué lors de l'introduction. Il est possible de représenter cette marche comme un automate dont chaque nœud consiste en un état qui retient l'histoire du prédicteur, les états de chaque prédicteur 2-bits local associé à chaque histoire, et l'instruction de branchement en cours d'exécution. Nous nommons cet automate $\mathcal{M}_{up}$. Nous avons donné une représentation partielle de cet automate dans la figure 5.12. Nous ne donnons pas une représentation exhaustive car cet automate contient beaucoup d'états (à savoir $2^{(\ell+1)+2^{\ell+1}}$).

Commençons par faire l'analyse du cas particulier quand $h = 0^\ell$.

Lemme 14. *Au cours de l'exécution de l'Algorithme 15, dans le cadre d'un prédicteur global avec table de prédicteurs 2-bits saturés, soit $h_t = 0^\ell$ avec ℓ un entier positif pair et un instant $t > 1$ suffisamment grand, la probabilité d'avoir une erreur de prédiction est donnée par*

$$\mu_0 = \frac{41}{119}.$$

Démonstration. Soit le processus dont chaque état consiste en une paire (s, i) avec s l'état de la table pour $h = 0^\ell$ et i l'instruction de branchement en cours d'exécution. Ce processus est une chaîne de Markov qui peut être représentée sous la forme d'un automate que nous nommons $\mathcal{M}_0$ que nous avons dessiné dans la Figure 5.13.

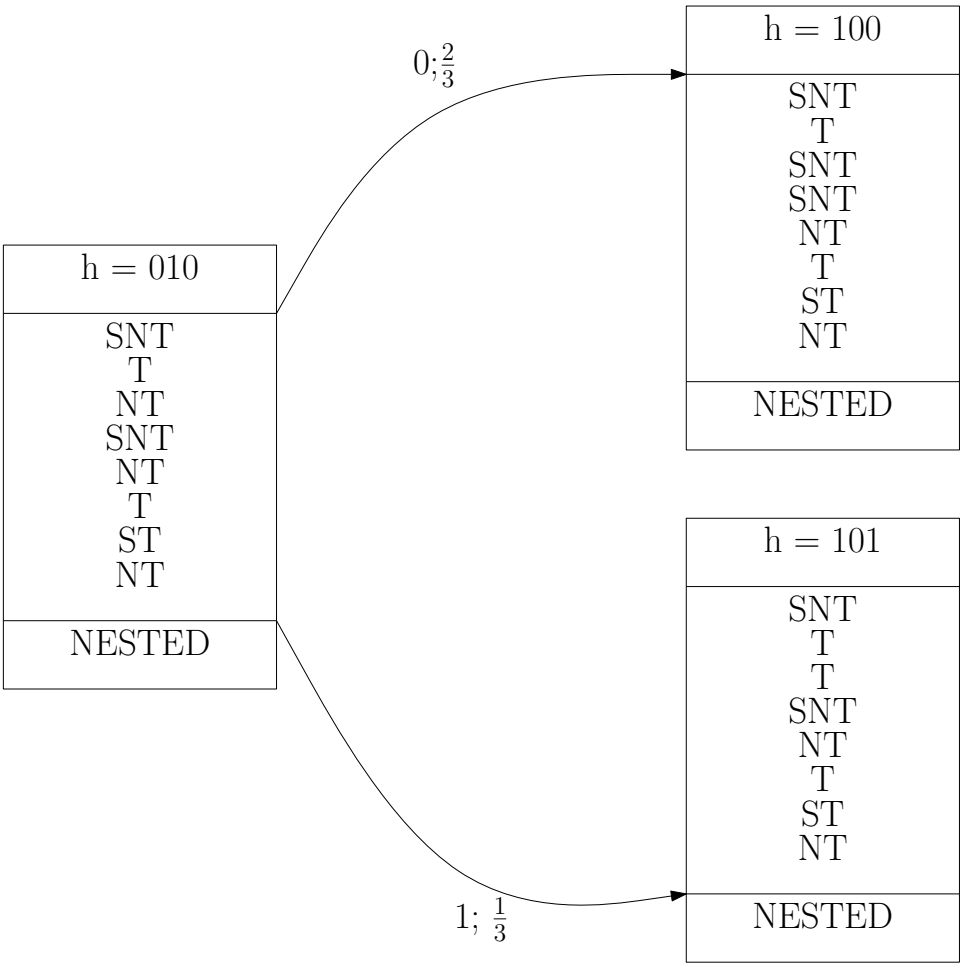

FIGURE 5.12 – Représentation partielle de la chaîne de Markov avec l'ensemble des états possibles à l'exécution de SKEWSEARCH

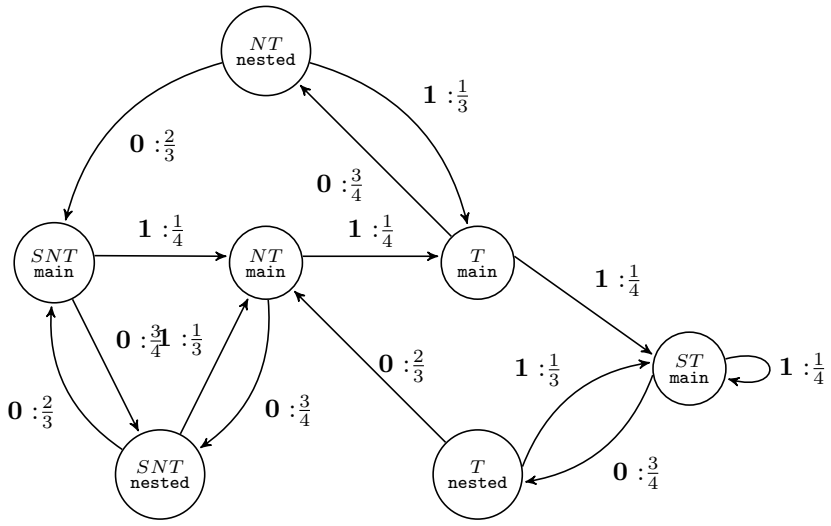

FIGURE 5.13 – La chaîne de Markov associée à la table 0^ℓ.

Supposons que nous sommes dans le cas où l'histoire au moment où nous exécutons la prochaine instruction de branchement est $h = 0^\ell$, nous avons alors deux cas possibles. Le premier cas est que le résultat de l'instruction de branchement est faux. Pour ce cas, la prochaine valeur de h est donnée une fois encore par $h' = 0^\ell$. D'après la Figure 5.11, si le prochain résultat d'une instruction est faux, alors la prochaine instruction qui sera exécutée n'est pas la même. Pour ce cas, si (s, i) représente l'état de l'automate $\mathcal{M}_0$ pour l'histoire h, alors l'état suivant est (s', i') avec $s' = \max(s - 1, 0)$ et $i' \neq i$. Le deuxième cas est que le résultat de l'instruction de branchement est vraie. Pour ce cas, la prochaine valeur de h est donnée par $h' = 0^{\ell-1}1$. Comme $h' \neq 0^\ell$, nous ne modifions pas lors de l'exécution de la prochaine instruction la table pour 0^ℓ. Cependant, nous pouvons obtenir après un certain nombre d'étapes l'histoire $h = 0^\ell$ à nouveau. Comme nous avons un 1 dans l'histoire à partir de h', il est alors nécessaire qu'à l'étape qui précède cette nouvelle occurrence, nous ayons une histoire de la forme $10^{\ell-1}$ et que le résultat de l'instruction à cette étape soit faux. Il se trouve que comme ℓ est pair, nous savons que $10^{\ell-1} \in \mathcal{H}_{nested}$, et donc comme le résultat est faux, à la prochain étape nous aurons à nouveau l'histoire de la forme 0^ℓ et l'instruction de branchement en cours est **main**. Ainsi pour ce cas, si l'état précédent est de la forme (s, i), le prochain état est de la forme $(s', \textbf{main})$ avec $s' = \min(3, s + 1)$. La probabilité des transitions correspond à la probabilité des transitions de l'automate à droite de la Figure 5.11. Il est possible de calculer la distribution stationnaire de cette chaîne de Markov. La conclusion est une conséquence directe de l'application du théorème ergodique pour cette distribution. □

Théorème 19. *Pour le prédicteur global avec table d'historiques et qui pour chaque histoire de longueur ℓ associe un prédicteur 2-bits saturé, le nombre moyen d'erreurs de prédiction obtenues lors de l'exécution de l'Algorithme 15*

pour une entrée de taille n est asymptotiquement équivalente à

$$\left(\frac{12}{35}+\frac{1}{595\cdot\sqrt{2}^{\ell}}\right)\frac{7}{6}\log_2 n.$$

Démonstration. Soit $w = w_0 w_1 \dots w_{N-1} \in \mathbb{B}^N$, le mot représentant les résultats, dans l'ordre de leurs exécutions, des instructions de branchements des lignes 5 et 7, où 1 signifie que le résultat est vrai et 0 qu'il est faux. Soit $h \neq 0^\ell$ une histoire. Soit les temps $1 \leq \tau_1 < \tau_2 < \cdots < \tau_m \leq N-1$ telles que pour tout t dans cet ensemble $h_t = h$. Il y a une occurrence de h qui se termine à la position t dans le mot w. Bien évidemment, les τ_i et m dépendent de N et de h, et sont des variables aléatoires pour des entrées aléatoires de l'algorithme. D'après le théorème ergodique, il existe une constante $\alpha_h > 0$ tel que $\mathbb{E}[m(h)] \sim \alpha_h N$, quand n tend vers l'infini. En effet, soit π_{up} le vecteur stationnaire de la partie fortement connexe de $\mathcal{M}_{up}$ définie plus haut, alors α_h est la somme de tous les $\pi_{up}(x)$ avec x qui fait partie de l'ensemble des états dont l'histoire est h. Observons également que le prédicteur $T_t(h)$ ne peut être modifié qu'aux temps τ_i. Autrement dit, $T_t(h)$ est constant pour $\tau_i + 1 \leq \tau < \tau_{i+1}$, en prenant pour convention $\tau_{m+1} = N$. Soit $\mathcal{H}_{main}$ et $\mathcal{H}_{nested}$ comme définis lors de la Définition 16. Pour tout $h \in \mathcal{H}_{main}$, les déplacements dans $T_t(h)$ pris aux temps $\tau_1 + 1, \tau_2 + 1, \dots, \tau_m + 1$ décrivent une marche aléatoire sur la chaîne de Markov associée au prédicteur 2-bits saturé de paramètre $\frac{1}{4}$. Ainsi le nombre moyen d'erreurs de prédiction provoquées par le prédicteur local $T[h]$ est asymptotiquement équivalent à $\mu\left(\frac{1}{4}\right)\alpha_h N$, d'après le Théorème 4. Respectivement pour tout $h \in \mathcal{H}_{nested}$, l'espérance du nombre d'erreurs de prédiction provoquées par $T[h]$ est asymptotiquement équivalent à $\mu\left(\frac{1}{3}\right)\alpha_h N$.

Il reste maintenant à analyser le comportement pour $h = 0^\ell$. C'est ce que nous avons déjà fait avec le résultat du Lemme 5.3.4 qui nous donne la probabilité d'erreur de prédiction espérée de $\mathcal{M}_0$ qui est $\mu_0 = \frac{41}{119}$. La probabilité d'avoir à un temps t l'histoire $h_t = 0^\ell$ est donnée par

$$\begin{aligned}
p_0 &= \mathbb{P}(h_t = 0^\ell) \\
&= \mathbb{P}(h_t = 0^\ell \textit{ et } IF_{t-\ell+1} = \textbf{main}) + \mathbb{P}(h_t = 0^\ell \textit{ et } IF_{t-\ell+1} = \textbf{nested}) \\
&= \mathbb{P}(IF_{t-\ell+1} = \textbf{main})\mathbb{P}(w_{t-\ell+1} = 0 \textit{ et } w_{t-\ell+2} = 0 \dots w_t = 0 \mid IF_{t-\ell+1} = \textbf{main}) \\
&\qquad + \mathbb{P}(IF_{t-\ell+1} = \textbf{nested})\mathbb{P}(w_{t-\ell+1} = 0 \textit{ et } w_{t-\ell+2} = 0 \dots w_t = 0 \mid IF_{t-\ell+1} = \textbf{nested}) \\
&= \mathbb{P}(IF_{t-\ell+1} = \textbf{main})\left(\frac{3}{4}\right)^{\frac{\ell}{2}}\left(\frac{2}{3}\right)^{\frac{\ell}{2}} + \mathbb{P}(IF_{t-\ell+1} = \textbf{nested})\left(\frac{3}{4}\right)^{\frac{\ell}{2}}\left(\frac{2}{3}\right)^{\frac{\ell}{2}} \\
&= \sqrt{2}^{-\ell}.
\end{aligned}$$

Nous posons par la suite C_n le nombre de comparaisons effectuées au total par l'Algorithme 15. D'après le théorème Ergodique, le nombre G_n d'erreurs de prédiction effectuées par le prédicteur global admet une espérance qui est asymptotiquement équivalent à

$$\mathbb{E}[G_n] \sim \left(\sum_{h\in\mathcal{H}_{main}} \alpha_h \mu_2\left(\frac{1}{4}\right) + \sum_{h\in\mathcal{H}_{nested}} \alpha_h \mu_2\left(\frac{1}{3}\right) + \frac{\mu_0}{2^\ell}\right)\mathbb{E}[C_n].$$

Cependant, comme la probabilité stationnaire d'être dans l'état *main* est donnée par $\pi_{up}(\mathbf{main}) = \frac{4}{7}$, nous avons

$$\sum_{h \in \mathcal{H}_{main}} \alpha_h = \pi_{up}(\mathbf{main})(1 - p_0) = \frac{4}{7}(1 - \sqrt{2}^{-\ell}).$$

De même comme $\pi_{up}(\mathbf{nested}) = \frac{3}{7}$,

$$\sum_{h \in \mathcal{H}_{nested}} \alpha_h = \pi_{up}(\mathbf{nested})(1 - p_0) = \frac{3}{7}(1 - \sqrt{2}^{-\ell}).$$

Comme $\mu_2\left(\frac{1}{4}\right) = \frac{3}{10}$ et $\mu_2\left(\frac{1}{3}\right) = \frac{2}{5}$, nous avons

$$\mathbb{E}[G_n] \sim \left(\frac{12}{35}(1 - \sqrt{2}^{-\ell}) + \frac{41}{119}\sqrt{2}^{-\ell}\right)\mathbb{E}[C_n] = \left(\frac{12}{35} + \frac{1}{595 \cdot \sqrt{2}^{\ell}}\right)\mathbb{E}[C_n].$$

Il nous suffit de remplacer la valeur de $\mathbb{E}[C_n]$ par $\frac{7}{6}\log_2 n$ que nous avons obtenu dans la Proposition 21 pour obtenir le résultat annoncé. □

D'après le Théorème 13, si on utilise un prédicteur local 2-bit pour chaque condition, le nombre d'erreurs de prédiction moyen est asymptotiquement équivalent à $\frac{12}{35}\frac{7}{6}\log_2 n$. La différence avec le prédicteur global est donc extrêmement faible, ce qui n'est pas une surprise puisque la seule différence entre les deux modèles pour le cas de l'algorithme 15 provient principalement de l'histoire $h = 0^{\ell}$ qui entremêle les deux instructions de branchement.

Dans cette section, nous avons proposé une modification de l'algorithme de la recherche dichotomique classique. Contrairement à l'exponentiation rapide, nous avons proposé un type différent de modifications à apporter pour que ce dernier soit capable de guider le prédicteur. Nous verrons, dans la section suivante, qu'il y a, malgré tout, quelques points communs entre les modifications apportées à ces deux algorithmes.

Pour finir, nous avons proposé une analyse des erreurs de prédiction produites par l'algorithme SkewSearch. Bien que, nous utilisons des propriétés de l'algorithme, à savoir l'existence d'un mot synchronisant, nous pensons qu'il s'agit du premier résultat théorique concernant ce type de prédicteur dans la littérature.

5.4 Généralisation à d'autres algorithmes

Dans la section qui suit, nous donnons des pistes dans le but de généraliser les modifications que nous avons apportées à nos algorithmes. Nous allons voir en effet qu'il existe des propriétés dans nos algorithmes que nous pouvons utiliser pour obtenir facilement des variantes de ces derniers. Ces variantes ont des chances de donner de bons résultats pour la prédiction comme nous avons pu le voir pour les cas que nous avons étudiés précédemment qui sont des cas particuliers de ces dernières

5.4.1 Algorithmes sous forme d'automates

Pour déterminer des propriétés intéressantes, nous allons définir la représentation d'un algorithme sous forme d'automate. Ce type de représentation est connue, elle apparaît dans la littérature, par exemple, sous la nomination de flot de contrôles.

Nous commençons par définir les ensembles ε et Σ qui représentent respectivement l'ensemble des entrées et des sorties de l'algorithme. Par exemple dans le cas de l'algorithme de l'exponentiation rapide, nous pouvons avoir $\varepsilon = \mathbb{R} \times \mathbb{N}$ et $\Sigma = \mathbb{R}$, puisque l'algorithme cherche à obtenir des valeurs de la forme x^n pour tout couple $(x, n) \in \mathbb{R} \times \mathbb{N}$. Soit l'ensemble $\mathcal{H}$, qui représente *l'environnement* dans lequel l'algorithme agit. Nous pouvons imaginer cet environnement comme étant l'ensemble des variables internes utilisées par l'algorithme pour fonctionner. Si nous reprenons notre exemple précédent, l'environnement sur lequel travaille l'exponentiation rapide est $\mathbb{R}^2 \times \mathbb{N}$ car à chaque instant nous retenons un triplet (r, y, k) tel que pour une entrée $(x, n) \in \varepsilon$, nous avons $r \times y^k = x^n$.

Définition 17. Une fonction *d'initialisation* $\mathcal{I} : \varepsilon \to \mathcal{H}$ transforme une entrée en environnement.

Dans le cas de l'exponentiation rapide, cette fonction est définie pour tout $(x, n) \in \varepsilon$ par $\mathcal{I}(x, n) = (1, x, n)$.

Définition 18. Une fonction de *sortie*, $\sigma : \mathcal{H} \to \Sigma$ transforme un environnement en une sortie.

Dans le cas de l'exponentiation rapide, cette fonction est définie pour tout $(r, y, k) \in \mathcal{H}$ par $\sigma(r, y, k) = r \times y^k$.

Définition 19. Une *question sur l'environnement* $q : \mathcal{H} \to \mathbb{B}$, est une fonction qui associe à tout état de l'environnement $h \in \mathcal{H}$ une réponse *oui* ou *non*. L'ensemble des questions sur l'environnement est noté par $\mathbb{B}^{\mathcal{H}}$.

Une question sur l'environnement qui est utilisée dans le cas de la boucle principale de l'exponentiation rapide est la fonction qui à tout $h = (r, y, k)$ associe $q(h) = oui$ si $k = 0$ et $q(h) = non$ sinon.

Définition 20. Une **action sur l'environnement** $a : \mathcal{H} \to \mathcal{H}$ est une fonction qui associe à un environnement $h \in \mathcal{H}$ un autre environnement $h' \in \mathcal{H}$ par l'action de a. L'ensemble des actions sur l'environnement est noté par $\mathcal{H}^{\mathcal{H}}$.

Une action qui est utilisée dans le cas de l'exponentiation rapide est l'action qui à tout $h = (r, y, k)$ associe $a(h) = (r \times y, y^2, \lfloor \frac{k}{2} \rfloor)$. Par la suite nous nommons une *fonction d'environnement*, une fonction qui est soit une question sur l'environnement soit une action sur l'environnement. L'ensemble des fonctions d'environnement est donc $\mathcal{H}^{\mathcal{H}} \cup \mathbb{B}^{\mathcal{H}}$.

Définition 21. Un automate $\mathcal{A}$ *à action sur* $\mathcal{H}$ est un septuple $\langle S, \mathbb{B} \cup \{\epsilon\}, \delta, i_0, F, h_0, \alpha \rangle$ avec :

- S l'ensemble des états de l'automate.
- $\delta : S \times (\mathbb{B} \cup \{\epsilon\}) \to S$ la fonction de transition.
- i_0 l'état initial.
- F l'ensemble des états finaux.

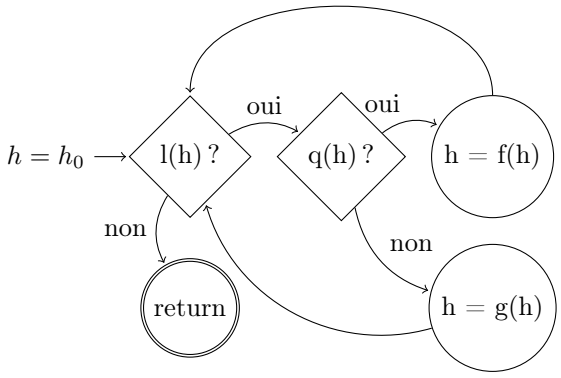

FIGURE 5.14 – Automate à action composé d'une boucle qui s'arrête quand $l(h)$ donne la réponse non, qui exécute $f(h)$ si $q(h)$ est oui et $f(h)$ sinon.

— $\alpha : S \to \mathcal{H}^{\mathcal{H}} \cup \mathbb{B}^{\mathcal{H}}$, la fonction d'environnement effectuée lorsque l'on atteint un des états de l'automate.

Remarque. La fonction α définit deux types possibles des états. Soit il s'agit d'une question qui appartient à $\mathbb{B}^{\mathcal{H}}$ et donc la seule chose que fait cet état c'est de poser la question en laissant l'environnement inchangé. Nous représentons dans nos figures ces états par une forme en losange. Soit il s'agit d'une action qui, à la fin de son application, change l'environnement. Nous représentons ces derniers états par une forme circulaire.

Définition 22. Nous appelons les états dont la fonction d'environnement est une question dans $\mathbb{B}^{\mathcal{H}}$ des *états de branchement*. Un état de branchement qui est au commencement d'un cycle est nommé un *état bouclant*.

Nous imposons des contraintes supplémentaires à ces automates pour la suite :

— Il est impossible d'avoir un cycle commençant par un état qui n'est pas de branchement.
— Si un état est de branchement, alors il n'appartient pas à F.
— Tous les états dans l'ensemble F sont des puits, c'est-à-dire qu'ils n'admettent pas de transition vers un autre état.

Une marche sur un automate d'action $\mathcal{A}$ initialisée par l'environnement $h_0 \in \mathcal{H}$ consiste en un chemin commençant par l'état initial i_0. Le chemin consiste à suivre les transitions définies par δ, ainsi lorsque nous atteignons un état $x \in S$ et que l'environnement est $h \in \mathcal{H}$, dont la fonction d'environnement $\alpha(x)$ est une question, l'état suivant est alors $\delta\left((\alpha(x))(h)\right)$. Soit $(x_0 = i_0, \ldots, x_m \in F)$ le chemin obtenu après cette marche. Nous pouvons extraire la sous-séquence de ce chemin dont les fonctions d'environnements sont des actions sur l'environnement sous la forme $(y_1, \ldots y_k)$, les valeurs prises par l'ensemble forment également une suite de la forme $(h_0, \ldots h_k)$ qui est construite par la récurrence valable à partir de $i > 0$, $h_i = (\alpha(x_i))(h_{i-1})$.

Théorème 20. *Pour une fonction d'initialisation $\mathcal{I} : \varepsilon \to \mathcal{H}$, un automate à action sur $\mathcal{H}$, et une fonction de sortie $\sigma : \mathcal{H} \to \Sigma$ il existe un algorithme A*

```
   Données : e ∈ ε
   Résultat : s ∈ Σ
1  h ← I(e)
2  tant que l(h) est oui faire
3  |  si q(h) est oui alors
4  |  |  h ← f(h)
5  |  sinon
6  |  |  h ← g(h)
7  retourner σ(h)
8  A(i_0, h)
```

Algorithme 16 : Algorithme équivalent à l'automate de la Figure 5.14.

qui pour toute entrée $e \in \varepsilon$ produit la même sortie $s = \sigma(h^*(e))$, avec $h^*(e)$ l'environnement final après une marche sur l'automate $\mathcal{A}$ initialisée par $h_0 = \mathcal{I}(e)$.

Démonstration. Notons $A(x, h)$, avec $x \in S$ et $h \in \mathcal{H}$, l'algorithme équivalent au sous-automate de $\mathcal{A}$ et dont l'état initial est x. L'algorithme A que nous cherchons à obtenir consiste alors à créer une variable h et à lui affecter la valeur $\mathcal{I}(e)$ et à appeler l'algorithme $A(i_0, h)$ qui va simuler la marche sur $\mathcal{A}$ avec l'environnement initialisé correctement. L'algorithme A est donc de la forme suivante :

```
   Données : e ∈ ε
   Résultat : s ∈ Σ
1  h ← I(e)
2  A(i_0, h)
```

Nous allons maintenant voir que $A(i_0, h)$ existe et que nous pouvons le construire. La construction se fait de manière récursive à l'aide de différentes règles. Essayons de construire $A(x, h)$ pour un état $x \in S$ et un environnement $h \in \mathcal{H}$ quelconque. Nous avons quatre cas possibles que nous allons maintenant détailler.

— Soit x n'est pas un état de branchement et $x \notin F$. Dans ce cas, la marche sur l'automate consiste à exécuter l'action $h = (\alpha(x))(h)$ et de continuer la marche à partir de $\delta(x, \epsilon)$. Cela revient donc à faire une affectation de h par $(\alpha(x))(h)$ et d'appeler l'algorithme $A(\delta(x, \epsilon), (\alpha(x))(h))$ ce qui revient à écrire l'algorithme suivant :

```
1  h ← (α(x))(h)
2  A(δ(x, ε), h)
```

— Soit x n'est pas un état de branchement et $x \in F$. Dans ce cas nous arrivons à la fin de la marche où nous exécutons l'action de x et nous obtenons alors que la sortie est donnée par $\sigma((\alpha(x))(h))$. Il est ainsi facile d'obtenir la même sortie en écrivant $A(x, h)$ sous la forme de l'algorithme suivant :

```
1  h ← (α(x))(h)
2  retourner σ(h)
```

— Soit x est un état de branchement non-bouclant. Si les deux branchements ne se rejoignent en aucun état, nous avons alors deux marches possibles, soit $(\alpha(x))(h) = oui$ et dans ce cas nous continuons la marche à partir de $\delta(x, oui)$, sinon nous continuons à partir de $\delta(x, non)$. Cela revient donc à tester $(\alpha(x))(h)$ et d'exécuter $A(\delta(x, oui), h)$ ou $A(\delta(x, non), h)$ selon le résultat. Nous obtenons alors l'algorithme $A(x, h)$ sous la forme suivante :

```
1 si (α(x))(h) est oui alors
2 |  A(δ(x, oui), h)
3 sinon
4 |  A(δ(x, non), h)
```

Notons $A_y(z, h)$ l'algorithme qui consiste à exécuter la marche sur l'automate $\mathcal{A}$ à partir de l'état z en s'arrêtant avant l'état y. Plus concrètement, $A_y(z, h)$ respecte les mêmes règles que $A(z, h)$ en ajoutant une règle supplémentaire qui consiste à omettre les lignes qui appellent $A(y, h')$ pour tout $h' \in \mathcal{H}$. Si les deux branchements se rejoignent en un état y, alors la marche consiste à exécuter les actions de la branche qui est prise. C'est ce qui est fait en exécutant $A_y(\delta(x, oui, h)$ ou $A_y(\delta(x, non, h)$ en fonction de la valeur de $(\alpha(x))(h)$. Nous obtenons alors dans ce cas l'algorithme $A(x, h)$ sous la forme suivante :

```
1 si (α(x))(h) est oui alors
2 |  A_y(δ(x, oui), h)
3 sinon
4 |  A_y(δ(x, non), h)
5 A(y, h)
```

— Soit x est un état bouclant. Nous allons avoir une boucle qui va répéter les actions à partir de l'état $\delta(x, oui)$ tant que $(\alpha(x))(h)$ a la valeur oui. L'algorithme $A(x, h)$ est de la forme suivante :

```
1 tant que (α(x))(h) est oui faire
2 |  A(δ(x, oui), h)
3 A(δ(x, non), h)
```

En appliquant toutes ces règles nous obtenons alors une écriture d'un algorithme équivalent à l'automate $\mathcal{A}$. □

Un algorithme qui agit sur $\mathcal{H}$ initialisé par l'état $h_0 \in \mathcal{H}$ est alors équivalent à un automate $\mathcal{A}$ à action sur $\mathcal{H}$ pour ce même h_0. Un exemple d'un tel automate est donné dans la Figure 5.14 qui est équivalent à l'Algorithme 16. En choisissant les fonctions $f, g \in \mathcal{H}^{\mathcal{H}}$ et des fonctions $l, q \in \mathbb{B}^{\mathcal{H}}$, il est possible d'obtenir l'algorithme d'exponentiation rapide ou de la recherche dichotomique. Ces deux algorithmes partagent donc un schéma commun. Pour le cas de l'exponentiation rapide, les fonctions à choisir sont définies de la façon suivante : Soit $h = (r, y, k)$,

— $l(h) :=$ "$k = 0$?"
— $q(h) :=$ "k *pair* ?"
— $f(h) = (r, y^2, \lfloor \frac{k}{2} \rfloor)$
— $g(h) = (r \times y, y^2, \lfloor \frac{k}{2} \rfloor)$

5.4.2 Automates à actions équivalentes

Nous allons désormais travailler principalement sur les automates. Notre but est d'apporter des transformations à nos automates qui permettent d'obtenir des résultats équivalents.

Définition 23. Deux automates initialisés respectivement par $\mathcal{I}_1 : \varepsilon \to \mathcal{H}_1$ et $\mathcal{I}_2 : \varepsilon \to \mathcal{H}_2$ sont *équivalents en actions* sur l'environnement si pour tout entrée $e \in \varepsilon$ on a $\sigma_1(h_1^*(e)) = \sigma_2(h_2^*(e))$, avec $\sigma_1 : \mathcal{H}_1 \to \Sigma$, $\sigma_2 : \mathcal{H}_2 \to \Sigma$ deux fonctions de sorties et $h_1^* : \varepsilon \to \mathcal{H}_1$, $h_2^* : \varepsilon \to \mathcal{H}_2$ les environnements atteints à l'état final pour toutes les entrées possibles. Nous avons donc deux automates qui travaillent sur des environnements pas nécessairement identiques mais qui sont initialisés par une même entrée et qui donne une même sortie à la fin de leurs marches.

Remarque. Comme nous allons étudier des transformations sur les automates, nous ne changeons pas l'environnement sur lequel travaille sa transformée, nous aurons donc $\mathcal{H}_1 = \mathcal{H}_2 = \mathcal{H}$. Nous utiliserons également les mêmes fonctions d'initialisation et de sortie. Nous verrons cependant des cas où $h_1^* \neq h_2^*$.

Changement des fonctions qui agissent sur l'environnement

Une première transformation qui peut être effectuée pour obtenir un automate équivalent en actions consiste à simplement changer les fonctions de α. La forme de l'automate reste donc inchangée, mais le comportement est différent. L'exemple où nous procédons ainsi est la recherche dichotomique où nous remplaçons par exemple la question q "est-ce que la valeur recherchée est avant la médiane ?" par la question q' "est-ce que la valeur recherchée est avant le premier quartile ?". Bien sûr avec cette nouvelle question il faut également changer en conséquence les actions des autres états de l'automate pour faire en sorte que les automates soient équivalents en actions.

Remarque. Cette méthode fonctionne bien si nous considérons des fonctions d'environnements dépendant d'un paramètre particulier. C'est un peu le cas avec la recherche dichotomique où le paramètre est la sélection du pivot. Si l'on note φ le nombre d'éléments plus petits que le pivot et ρ le nombre d'éléments plus grands que le pivot. Nous avons alors que la recherche dichotomique classique est paramétrée par $\frac{\varphi}{\varphi+\rho}$ que l'on souhaite le plus proche possible de $\frac{1}{2}$. Dans le cas de BiasedBinarySearch nous voulons que ce paramètre soit le plus proche possible de $\frac{1}{4}$.

Déroulement de boucle

La deuxième opération à appliquer concerne le déroulement de boucle. Il s'agit d'une opération qui peut être effectuée sur des automates comme celui de la Figure 5.14 et qui donne une fois transformé l'automate de la Figure 5.15.

Sous certaines conditions sur les fonctions associées aux actions de l'automate, il est possible d'obtenir un automate équivalent qui est celui de la Figure 5.16.

Lemme 15. *Soit pour toute entrée $e \in \varepsilon$ l'environnement $h^*(e)$ atteint à l'état final après une marche sur l'automate de la Figure 5.14 pour des fonctions l, q, f, g arbitraires. Si $q(h^*(e)) = oui$ et $l(f(h^*(e))) = non$ et $\sigma(f(h^*(e))) =$*

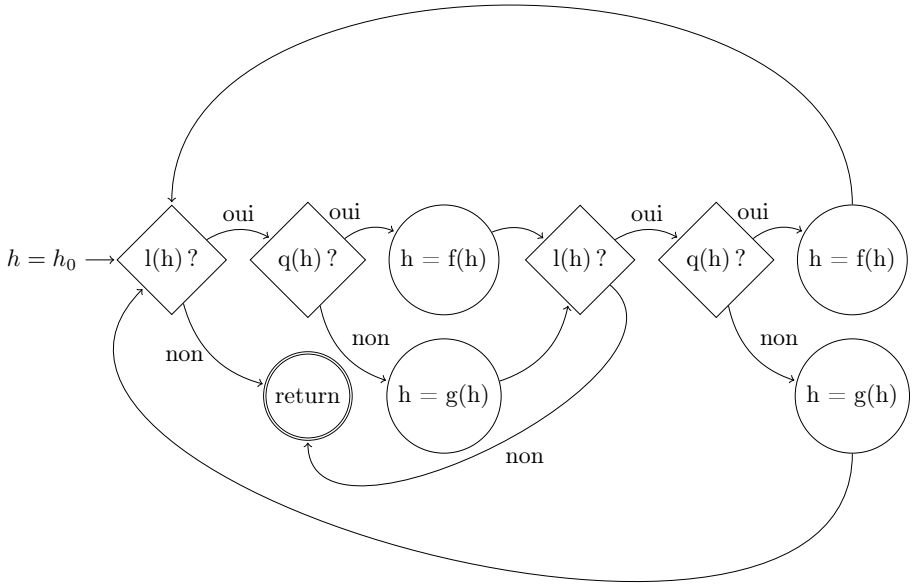

FIGURE 5.15 – déroulement de boucle simple

$\sigma(h^*(e))$ ou si $q(h^*(e)) = non$ et $l(g(h^*(e))) = non$ et $\sigma(g(h^*(e))) = \sigma(h^*(e))$, alors cet automate et l'automate de la Figure 5.16 sont équivalents en actions en gardant ces mêmes choix de fonctions.

Démonstration. Les modifications apportées à l'environnement par les marches sur ces deux automates sont identiques excepté à l'approche de la dernière étape pour une question de parité. Les différents environnements obtenus par l'action de l'automate de la Figure 5.14 forment une suite $(h_0 = \mathcal{I}(e), h_1, \ldots, h_m = h^*(e))$. Soit $m = 2k$ pair, et dans ce cas les environnements obtenus par l'action de l'automate de la Figure 5.16 forment la suite $(h_0, h_2, h_4, h_6, \ldots h_{2k} = h_m)$. Soit $m = 2k + 1$ impair, et dans ce cas les environnements obtenus par l'action de l'automate de la Figure 5.16 forment la suite $(h_0, h_2, \ldots, h_{2k}, h_{2k+2})$ avec $h_{2k+2} = f(h^*(e))$ si $q(h^*(e)) = oui$ et $l(f(h^*(e))) = non$ ou $h_{2k+2} = g(h^*(e))$ si $q(h^*(e)) = non$ et $l(g(h^*(e))) = non$. Si $q(h^*(e)) = oui$ il est donc suffisant que $\sigma(f(h^*(e))) = \sigma(h^*(e))$ pour que les deux automates donnent la même sortie. De même, si $q(h^*(e)) = non$ il est donc suffisant que $\sigma(g(h^*(e))) = \sigma(h^*(e))$. □

La condition nécessaire pour appliquer ce lemme est par exemple vérifiée dans le cas de l'exponentiation rapide. Rappelons que nos choix de fonctions sont les suivants :

— $l(h) :=$ "$k = 0$?"
— $q(h) :=$ "k *pair* ?"

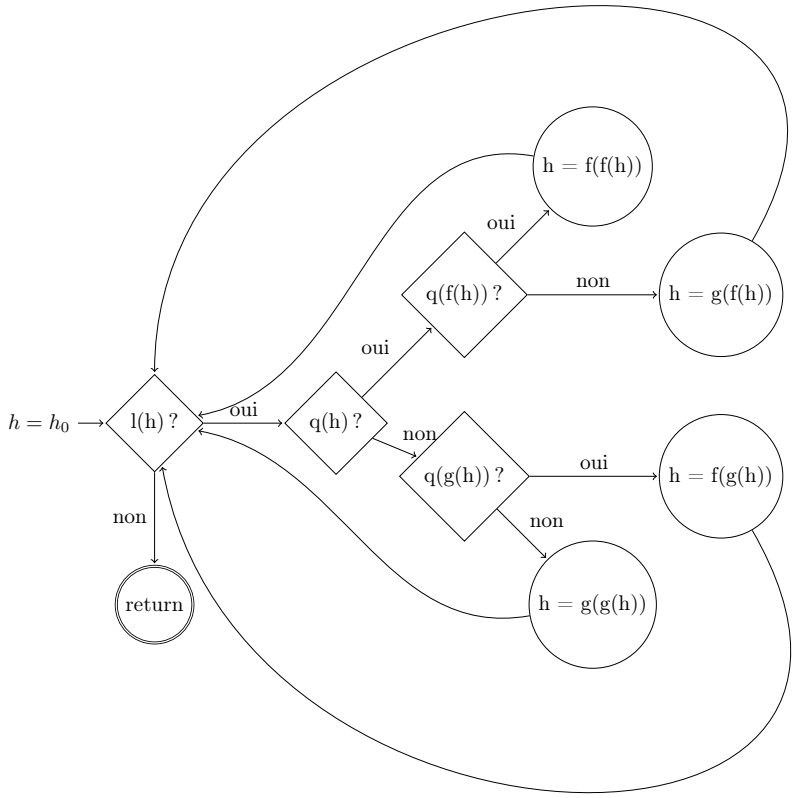

FIGURE 5.16 – déroulement de boucle complexe

— $f(h) = (r, y^2, \lfloor \frac{k}{2} \rfloor)$
— $g(h) = (r \times y, y^2, \lfloor \frac{k}{2} \rfloor)$

Pour l'automate de la Figure 5.14, pour une entrée $e = (x, n)$, l'environnement $h^*(e)$ est de la forme $(x^n, x^{2^i}, 0)$ avec $i \in \mathbb{N}$ le nombre d'itérations effectuées en tout. Si on applique $\sigma(h^*(e))$ nous obtenons bien x^n. Remarquons que nous avons $q(h^*(e)) = oui$ et que de plus $f(h) = (x^n, x^{2^{i+1}}, 0)$. Nous avons ainsi que $\sigma(f(h)) = x^n$ et $l(f(h)) = non$, la condition est donc satisfaite ce qui nous permet d'appliquer le lemme.

5.4.3 Ajout d'une redondance

Nous allons maintenant généraliser la transformation que nous avons appliquée pour obtenir l'exponentiation rapide qui guide la prédiction de branchement. L'idée est de rajouter une redondance qui est inutile au fonctionnement

de l'algorithme mais qui peut avoir pour effet d'améliorer la prédiction. Nous donnons le lemme suivant :

Lemme 16. *Soit pour toute entrée $e \in \varepsilon$ l'environnement $h^*(e)$ atteint à l'état final après une marche sur l'automate de la Figure 5.16 pour des fonctions l, q, f, g arbitraires. Si pour tout $h \in \mathcal{H}$, $q(f(h)) = q(g(h))$ alors il y a équivalence en action entre cet automate et l'automate de la Figure 5.17 en gardant ces mêmes choix de fonctions.*

Démonstration. La seule différence entre ces deux automates est l'ajout d'un état qui fait le test de savoir si $q(h)$ est oui ou si $q(f(h))$ est oui. Comme par hypothèse, $q(f(h)) = q(g(h))$ faire le test de $q(f(h))$ revient à faire le test de $q(g(h))$. Nous pouvons vérifier pour chaque cas que les deux automates vont changer l'environnement de la même façon à chaque itération. Si $q(h)$ est oui, alors $q(h)|q(f(h))$ est oui également. Dans ce cas les deux automates testent alors la valeur de $q(f(h))$ et font les mêmes modifications de la valeur de h. Si $q(h)$ est non, alors l'automate de la Figure 5.16 fait le test de la valeur de $q(g(h))$, si $q(g(h)) = oui$ alors h est modifiée par $f(g(h))$. Dans ce cas, $q(f(h)) = q(g(h)) = oui$ et donc l'automate de la Figure 5.17 fait la même action. Si $q(h) = non$ et $q(g(h)) = non$ alors h est modifiée par $h = g(g(h))$. Dans ce cas le test $q(h)|q(f(h))$ est non également et l'on voit que la même modification est apportée. □

En appliquant ce dernier lemme et le Théorème 20 au cas de l'exponentiation rapide, nous obtenons l'Algorithme 12. Il est facile de vérifier ici que $q(f(h)) = f(g(h))$ pour tout $h \in \mathcal{H}$. En effet si $h = (r, y, k)$, le test $q(h)$ se fait uniquement sur la variable k, et dans les deux cas après avoir appliqué $f(h)$ ou $g(h)$ nous avons que k est divisé par 2.

Dans la Figure 5.17, nous remarquons qu'il y a un état qui est inaccessible. En retirant cet état, nous obtenons l'automate de la Figure 5.18 qui est de ce fait équivalent en action au premier. Nous ne nous étions pas aperçus de ce cas lorsque nous avons fait l'étude de cet algorithme dans nos travaux. En appliquant le Théorème 20, nous obtenons ainsi l'Algorithme 17 qui réduit encore légèrement le nombre d'erreurs de prédiction par rapport à l'Algorithme 12 ainsi que le nombre de comparaisons et est donc plus efficace encore en pratique.

Dans cette section, nous donnons des pistes permettant de généraliser les modifications que nous avons faites pour modifier l'algorithme de l'exponentiation rapide et de la recherche dichotomique. Nous avons vu que ces algorithmes avaient des propriétés particulières que nous avons exploitées pour parvenir à nos fins.

Les travaux présentés ici ne sont encore qu'à un stade préliminaire, mais ils peuvent donner des méthodes pour permettre de modifier d'autres algorithmes, afin que ces derniers soient capables de guider la prédiction.

Dans l'avenir, il pourrait être intéressant de continuer à développer ces travaux afin d'exploiter d'autres propriétés des algorithmes. Nous pourrions, aussi, essayer de mettre en évidence les probabilités des branchements, afin de démontrer que ces modifications nous conduisent à des gains en ce qui concerne la prédiction de branchement.

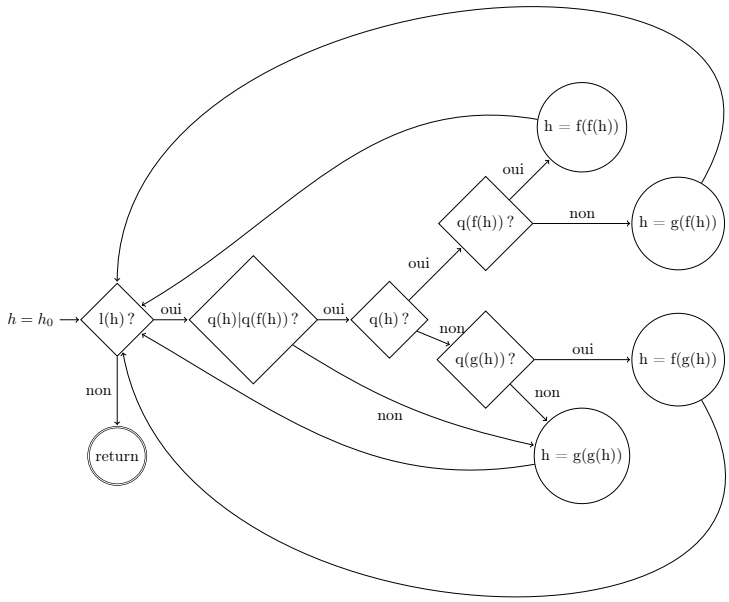

FIGURE 5.17 – Ajout d'une redondance dans l'automate Figure 5.16.

5.5 Conclusion

Dans ce chapitre, notre but était de modifier des algorithmes connus dans le but de guider la prédiction. Nous avions pour espoir que ces variantes, une fois implantées sur nos machines actuelles, aient de bonnes performances en temps d'exécution. Il nous faut souvent pour faire moins d'erreurs de prédiction faire plus d'opérations standards. Il y a donc des compromis que nous avons dû faire. Notre espoir est cependant justifié, nous avons, par exemple, montré que l'algorithme naïf de recherche simultanée du minimum et du maximum dans une séquence est plus efficace, en pratique, que l'algorithme optimisé qui fait pourtant 25% de comparaisons en moins. Ce premier résultat mettait en évidence que ces compromis étaient possibles.

Nous avons ensuite proposé nos propres modifications à deux algorithmes classiques, l'exponentiation rapide et la recherche dichotomique. Pour ces deux cas, il s'agit de deux algorithmes avec une structure similaire. Ces deux algorithmes consistent en une boucle au sein de laquelle nous avons une instruction de branchement. Comme cette instruction de branchement est exécutée plusieurs fois, il peut être intéressant de guider la prédiction. Pour y parvenir, dans les deux cas, nous avons fait apparaître de nouvelles instructions de branchement dans la boucle tout en faisant en sorte que le comportement de ces variantes reste similaire aux algorithmes d'origines. Nous avons, par exemple, effectué un déroulement de boucle pour l'exponentiation rapide. Nous avons ensuite fait en

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1  r ← 1
2  y ← x
3  k ← n
4  tant que k ≠ 0 faire
5      z ← y²
6      si k est impaire ou ⌊k/2⌋ est impaire alors
7          si k est impaire alors
8              r ← r · y
9              si ⌊k/2⌋ est impaire alors
10                 r ← r · z
11         sinon
12             r ← r · z
13     y ← z²
14     k ← ⌊k/4⌋
15 retourner r
```

Algorithme 17 : Algorithme de l'exponentiation rapide permettant de guider le prédicteur de branchement

sorte que ces instructions de branchement soient dépendantes entre-elles pour déséquilibrer les probabilités de l'issue choisie par ces dernières. C'est ce déséquilibre qui nous permet de guider la prédiction. En contrepartie, nous avions plus d'opérations à effectuer, dans le cas de l'exponentiation rapide nous avions environ 25% d'instructions de branchement supplémentaires à faire. Cependant, nous avons vu que ce surcout s'est avéré être compensé par la prédiction de branchement au point d'observer de meilleures performances en temps d'exécution de nos variantes par rapport aux algorithmes originaux.

Comme les structures de nos algorithmes sont similaires, nous avons donné des pistes pour essayer de déterminer des règles pour modifier des algorithmes avec la même structure. Nous sommes ainsi parvenus à extraire des propriétés abstraites que nous avons exploitées pour obtenir nos variantes. Nous pensons qu'il reste à ce jour de nombreux algorithmes qui peuvent être modifiés pour guider suffisamment bien la prédiction de branchement et obtenir de meilleures performances en pratique. Les pistes que nous avons données à la fin de ce chapitre pourraient par exemple permettre de détecter ces algorithmes et d'appliquer le même genre de modifications. Par la suite, il pourrait donc être intéressant de développer un peu plus cette généralisation et de les appliquer à des algorithmes encore plus complexes.

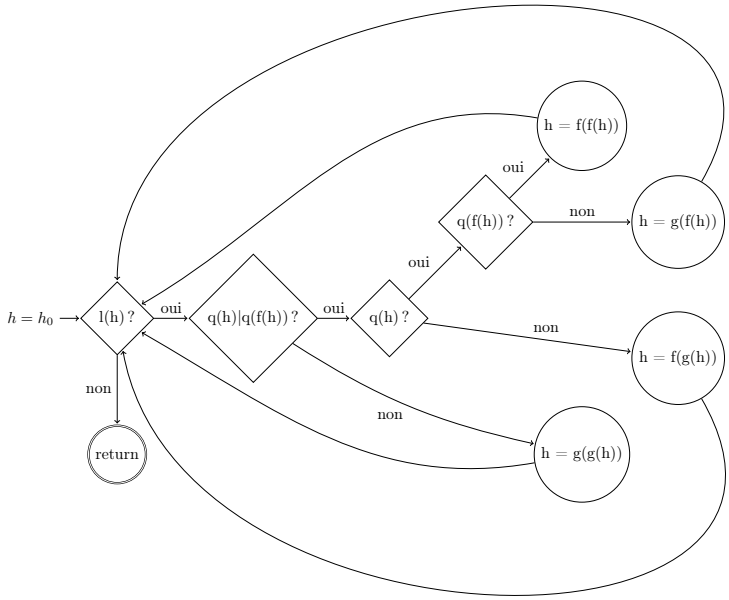

FIGURE 5.18 – Automate obtenu à partir de la Figure 5.17 en retirant l'état inaccessible.