Notions d'informatique

Sommaire

3.1 Algorithmes et théorie de la complexité

Dans cette section, nous allons présenter la discipline qu'est l'algorithmique. Pour commencer, nous donnerons une définition de ce qu'est un algorithme. Nous aborderons également la théorie de la complexité, qui est le domaine dans lequel s'inscrit ce manuscrit. Puis, nous discuterons des paradigmes les plus utilisés dans l'élaboration d'algorithmes.

3.1.1 Qu'est-ce qu'un algorithme ?

Il n'est pas aisé de donner une définition de ce qu'est un algorithme. L'origine du mot viendrait d'une déformation du nom du mathématicien perse Al-Khwārizmāri. Une analogie qui est souvent faite est celle de la recette de cuisine qui consiste en une suite d'instruction pour transformer les ingrédients en un plat. Concrètement un algorithme fait la même chose, il donne une suite d'instruction à effectuer pour transformer une entrée en une sortie, chaque instruction forme une étape de l'algorithme. Un algorithme connu de beaucoup, est le célèbre algorithme d'Euclide que nous apprenons tôt à l'école pour déterminer le plus grand commun diviseur PGCD entre deux nombres entiers. Donald Knuth, qui est l'un des pionniers dans le domaine de l'algorithmique, définit 5 caractéristiques qu'un algorithme doit avoir [29][pp. 5-6] Pour illustrer ses propos, Knuth utilise l'exemple de l'algorithme d'Euclide, nous allons procéder de la même façon en donnant l'algorithme d'Euclide sous la forme suivante :

Données : Deux entiers m et n
Résultat : Le PGCD de m et n, qui est le plus grand entier positif qui divise à la fois n et m.

1 **tant que** $n \neq 0$ **faire**
2 | Faire la division euclidienne de m par n, soit r le reste obtenu de cette dernière.
3 | $m \leftarrow n$
4 | $n \leftarrow r$
5 **retourner** m

Algorithme 1 : Algorithme d'Euclide

Dans son livre, Knuth avait décomposé l'algorithme en 3 étapes. La première, *E1* est l'étape de calcul du reste r de la division euclidienne de m par n qui correspond à la ligne 2. La deuxième, *E2* est l'étape du test de r qui se fait par la combinaison de la ligne 4 suivie par la ligne 1, cette étape vérifie si le reste est 0 et dans ce cas l'algorithme s'arrête. La troisième *E3* est l'étape de réduction, dans cette étape nous réduisons le problème de calculer le PGCD entre m et n par le problème de calculer le PGCD entre n et r. Cela revient à modifier la variable m par la valeur de n, puis de modifier la variable n par le reste r, ce que nous faisons aux lignes 3 et 4.

Nous pouvons dès à présent énumérer les 5 caractéristiques qu'un algorithme possède selon Knuth.

— *Finitude.* Un algorithme doit toujours s'arrêter après un nombre fini d'étapes. L'intérêt d'utiliser un algorithme est évidemment de trouver une solution à un problème, et il est donc nécessaire que ce dernier se termine en un temps fini pour espérer pouvoir obtenir la solution. Dans notre exemple, l'algorithme d'Euclide se termine toujours, si n et m sont

eux même finis. En effet, après l'étape de réduction $n = r$ et $0 \geq r < m$, si on note par $n_0 = n, n_1 = r, \ldots$ les valeurs prises par n après chaque modification, cette suite est alors strictement décroissante et positive et sera donc bien amenée à se terminer par 0 au bout d'un certain nombre fini d'étapes.

— *Définitude.* Chaque étape doit être définie de manière précise. Chaque cas doit ainsi être traité de manière rigoureuse et sans ambiguïté. De manière générale, les étapes sont constituées d'actions élémentaires autorisées. Ce qui est le cas par exemple avec les jeux d'instructions dans un langage informatique, une étape ne peut être définie qu'avec des mots autorisés par ce langage. Dans notre exemple, à l'étape de recherche du reste, le lecteur est supposé comprendre le sens de l'opération de division euclidienne entre deux entiers m et n, cela veut aussi dire qu'à tout instant, au cours de cette étape, il faut que m et n soit des entiers auquel cas la division euclidienne n'est pas bien définie. Ici c'est toujours le cas, puisque c'est vrai à l'initialisation et que l'étape de réduction ne change pas la nature de n et m.

— *Entrée.* Un algorithme peut avoir un certain nombre d'entrées. Une entrée est une quantité qui est donnée à l'initialisation avant même que l'algorithme ne commence ou bien, de manière dynamique au cours de l'exécution. Ces entrées sont choisies dans des ensembles spécifiques. Dans notre exemple, les deux entrées sont n et m qui sont deux éléments de l'ensemble des entiers naturels $\mathbb{N}$.

— *Sortie.* Un algorithme a au moins une sortie. Une sortie est une quantité qui a une relation particulière avec les entrées. Dans le cas de l'algorithme d'Euclide, la sortie est le PGCD des entrées. Pour prouver que c'est bien cette sortie que nous obtenons, il suffit de voir que le PGCD de m et n et le PGCD de n et r sont les mêmes.

— *Efficacité.* Les opérations effectuées par l'algorithme doivent être efficaces, dans le sens où elles doivent être suffisamment basiques pour pouvoir être effectuées en un temps fini par quelqu'un utilisant juste un crayon et une feuille de papier. Les opérations utilisées dans l'algorithme d'Euclide ne consistent qu'à faire la division entre deux entiers, tester si une variable est nulle, et modifier des variables par des nouvelles valeurs entières. De ce fait ces opérations sont efficaces dans ce sens, puisque les nombres entiers peuvent être écrits sur un papier sous une forme finie, et nous connaissons une méthode elle même finie pour diviser deux nombres entre eux. Si nous remplacions cette division d'entier par une division sur des nombres réels arbitraires qui peuvent avoir un nombre de décimal infini, l'opération deviendrait alors inefficace puisqu'il est impossible de représenter ces nombres de manière finie sur un papier.

Pour revenir sur notre analogie de recette de cuisine, cette dernière possède les propriétés de finitude, d'entrée et de sortie. Elle peut cependant manquer de définition rigoureuse, par exemple, si l'instruction consiste à rajouter une poignée de sel, la définition d'une poignée de sel est ambigüe. Un algorithme doit pouvoir être exécuté par une machine et il faut donc faire preuve d'une certaine rigueur lorsque nous écrivons un algorithme.

3.1.2 Théorie de la complexité

Un des éléments importants dans plusieurs de ces définitions est qu'un algorithme doit pouvoir résoudre un problème avec efficacité. Nous devons donc définir des méthodes pour pouvoir mesurer l'efficacité d'un algorithme, et c'est ce que la théorie de la *complexité* nous permet de faire. Le but de la théorie de la complexité est ainsi de permettre de quantifier les performances d'un algorithme. Concrètement, nous nous intéressons au nombre d'opérations élémentaires effectuées par l'algorithme au cours de son exécution. Ce nombre d'opérations doit de plus dépendre de la taille de l'entrée, par exemple, si l'entrée est une séquence, nous pouvons considérer son nombre d'éléments comme la taille de l'entrée. Bien souvent, nous nous intéresserons au comportement asymptotique de ce nombre d'opérations exprimé sous forme d'une fonction de coût $f(n)$ quand la taille de l'entrée n est grande. En pratique, les fonctions de coûts qui vont nous intéresser permettent d'obtenir des estimations de la **durée d'exécution** ou de la **mémoire utilisée** d'une implantation de l'algorithme sur une machine RAM dont nous discuterons par la suite dans la section 3.2.

Il y a trois grands types de comportement qui sont principalement étudiés en théorie de la complexité :

- L'analyse dans **le meilleur des cas**, qui comme son nom le laisse à penser, consiste à regarder le comportement de l'algorithme dans le cas où il reçoit l'entrée qui lui est la plus favorable.
- L'analyse dans **le pire cas**, qui à l'opposée du premier type consiste à regarder le comportement de l'algorithme lorsque celui-ci reçoit l'entrée qui lui est la moins favorable.
- L'analyse dans le **cas moyen**, qui revient à faire la moyenne de tous les coûts sur l'ensemble des entrées possibles. Bien souvent, nous considérons que l'entrée est générée selon une distribution qui nous donne ainsi sa probabilité d'apparition. Nous définissons alors la complexité dans le cas moyen en regardant le comportement asymptotique de l'espérance du coût. Par défaut, nous choisissons souvent la distribution uniforme ce qui revient à la définition d'origine. Il peut cependant être intéressant de considérer d'autres distributions, et c'est ce que nous faisons par la suite dans le Chapitre 6.

Remarque. Il est possible de faire une analogie avec les statistiques. Le pire cas est ainsi un maximum de la fonction de coût, sur toutes les entrées nous savons que nous ne dépasserons pas le coût du pire cas. De manière similaire, le meilleur cas est un minimum, pour toute entrée nous avons un coût au moins aussi grand que le meilleur cas. Évidemment, le cas moyen est une moyenne par définition. Bien qu'en pratique il est possible de rencontrer peu fréquemment le pire cas, l'analyse de ce dernier peut rester important pour des applications critiques qui jouent le rôle d'une garantie sur le temps d'exécution de l'algorithme une fois implanté sur une ressource critique.

La notation de Landau

Nous allons faire un rappel sur la notation de Landau qui est grandement utiliser en théorie de la complexité. Nous allons, par la suite, nous intéresser à des fonctions définies sur les entiers positifs à valeurs réelles et croissantes.

Majoration, notation $\mathcal{O}$ Nous l'avons utilisée précédemment pour discuter de la complexité du tri rapide. La notation $\mathcal{O}$ permet de définir une majoration d'une fonction par une autre à une constante près. Plus formellement, on dit qu'une fonction $f(n) = \mathcal{O}(g(n))$, s'il existe un rang n_0 et une constante $c \in \mathbb{R}$, tels que pour tout $n > n_0$, on a l'inégalité $f(n) < c \times g(n)$ qui est vérifiée.

Minoration, notation Ω À l'opposé de la majoration, il est possible de définir une minoration d'une fonction par une autre à une constante près. Dans ce cas, on dit que $f(n) = \Omega(g(n))$ si et seulement si $g(n) = \mathcal{O}(f(n))$.

Borne, notation Θ On dit qu'une fonction est bornée par une autre fonction à une constante près si cette première est à la fois majorée et minorée par cette deuxième. Autrement dit, nous avons $f(n) = \Theta(g(n))$ si et seulement si $f(n) = \Omega(g(n))$ et $f(n) = \mathcal{O}(g(n))$.

Négligeable, notation o On dit que la fonction f est négligeable devant la fonction g si pour toute constante réelle $c \in \mathbb{R}$, il existe un rang $n_0 \in \mathbb{N}$ tel que pour tout $n > n_0$ l'inégalité $f(n) < c \times g(n)$ est vérifiée. Nous notons alors que $f(n) = o(g(n))$. Nous pouvons également remarquer que, si $f(n) = o(g(n))$, alors $f(n) = \mathcal{O}(g(n))$.

Domine, notation ω On dit que f domine g si l'on a $g(n) = o(f(n))$. On note alors que $f(n) = \omega(g(n))$. Nous pouvons ainsi faire une remarque similaire à celle qui a été faite pour la notation o, à savoir que si $f(n) = \omega(g(n))$, alors $f(n) = \Omega(g(n))$.

Équivalent, notation $\sim$ On dit que f est équivalent à g, si et seulement si, $f(n) = g(n) + o(g(n))$. On note alors $f(n) \sim g(n)$. Une fois de plus, nous pouvons faire une remarque similaire aux précédentes, si $f(n) \sim g(n)$ alors $f(n) = \Theta(g(n))$.

D'après les trois remarques précédentes, on peut dire que ces trois premières définitions sont des cas particuliers plus faibles de ces trois dernières définitions. Lorsque l'on cherche à faire l'analyse sur la complexité d'un algorithme, nous procéderons souvent ainsi, premièrement, nous choisirons le type de comportement qui nous intéresse comme par exemple le pire cas, ensuite nous choisissons une fonction de coût comme, par exemple, le nombre de comparaisons total pendant l'exécution de l'algorithme, et enfin, nous utilisons la notation de Landau pour comparer cette fonction avec des fonctions plus simples.

3.1.3 Méthodes d'analyse

Toujours dans le cadre de la théorie de la complexité, nous allons voir différentes méthodes pour déterminer des bornes. Avec l'argument d'adversaire, nous pouvons obtenir des bornes inférieures sur l'ensemble des algorithmes d'un problème spécifique. La méthode d'analyse amortie consiste à considérer les coûts moyens d'une opération dans une structure. Nous utilisons une méthode d'analyse amortie pour faire notre analyse de l'algorithme TimSort dans le chapitre 4.

Argument d'adversaire

Dans cette section, nous aborderons la méthode d'argument d'adversaire qui permet de déterminer une borne inférieure à un problème. Cette méthode nous permet donc de démontrer rigoureusement, que tout algorithme qui trouve une solution à ce problème doit avoir une complexité qui est supérieure à cette borne.

Illustrons le principe de fonctionnement avec l'exemple de la bataille navale. Pour rappel, une partie de bataille navale se compose en une première étape où nous plaçons des bâteaux dans un espace quadrillé. La deuxième étape est la partie en elle-même où chaque joueur essaye de deviner la position de la flotte de son adversaire en tirant un missile à une position à tour de rôle. Après un tir, l'adversaire doit alors dire si le joueur a fait un tir à blanc, touché un bâteau, ou couler un bâteau si ce dernier a subi un tir sur toutes ses positions. Un adversaire malhonnête pourrait faire durer la partie en trichant. Au lieu de positionner ses bateaux dès le début de la partie, il va le faire à la volée au cours de la partie en fonction des tirs du joueur opposé. Tant que ses réponses restent cohérentes, ce joueur ne peut pas se rendre compte que son adversaire est un tricheur.

C'est exactement le fonctionnement d'un *argument d'adversaire*. L'idée étant qu'un adversaire va mentir à l'algorithme en construisant l'entrée à la volée, dans le but de faire trainer le jeu et faire poser le plus de questions à ce dernier. L'algorithme n'a donc pas accès directement à l'entrée et ne peut donc pas faire d'observation directement. L'adversaire est l'intermédiaire entre l'algorithme et l'entrée. Bien entendu, si l'algorithme recevait l'entrée que l'adversaire génère à la volée, il prendrait le même temps. De cette façon, si nous montrons qu'un adversaire peut faire durer ce jeu de questions pour un certain temps indépendamment de l'algorithme contre lequel il joue, nous obtenons alors une borne inférieure, puisqu'il existe pour tous ces algorithmes au moins une entrée qui aura le coût correspondant à cette borne.

Essayons de formaliser un peu tout ce que nous venons de dire jusqu'à présent. Dans un argument d'adversaire, un jeu est joué entre un algorithme et un adversaire. Dans ce jeu, l'algorithme doit générer la ou les sorties en faisant certaines observations sur l'entrée. Il n'a cependant pas directement accès à cette dernière et doit alors poser des *questions* à l'adversaire. Le but de l'algorithme est alors de poser le moins de questions possibles afin de déterminer la sortie le plus rapidement possible. Le but de l'adversaire est, au contraire, de faire traîner le jeu le plus longtemps possible en faisant en sorte que l'algorithme pose un maximum de questions. L'adversaire peut mentir aux questions posées par l'algorithme mais il doit rester cohérent dans ses réponses. Par exemple, dans le cas où on a des algorithmes de tri pour une séquence $X = \langle x_1, \ldots, x_n \rangle$, si l'algorithme demande si $x_1 < x_2$ et $x_2 < x_3$ et que l'adversaire répond "oui" à ces deux questions, il n'a pas le droit de répondre "non" si on lui demande en troisième question si $x_1 < x_3$, car par transitivité ça doit être vraie. Bien entendu, il faut faire attention aux questions qu'un algorithme peut poser. Nous considérons que les questions appartiennent à un ensemble de questions élémentaires autorisées. De cette façon, toujours dans notre exemple d'algorithme de tri, il est interdit de poser la question "est-ce que la séquence est triée ?". Les questions doivent refléter la borne que nous cherchons à établir. Dans le cas d'un algorithme de tri par comparaisons, nous n'autoriserons, par exemple, que les questions de comparaisons entre deux éléments de la séquence d'entrée. Si l'adversaire a la garantie de pouvoir faire durer le jeu pendant m questions,

alors m est une borne inférieure au problème, et ainsi tout algorithme qui essaye de résoudre ce problème doit nécessairement poser au moins autant de questions pour résoudre le problème. Si par exemple, pour un problème donné, l'ensemble des questions est défini par toutes les comparaisons possibles entre deux éléments dans une séquence d'entrée, alors un algorithme qui n'utilise que ces comparaisons pour résoudre ce problème a une complexité qui est au moins de m.

Une démonstration par argument d'adversaire consiste donc à bien définir le jeu qui est joué, ainsi que les questions qui sont autorisées. Notre but est alors de décrire la stratégie utilisée par un adversaire et de montrer combien de temps il est capable de faire traîner le jeu indépendamment de l'algorithme contre lequel il joue. Nous devons ensuite montrer que, pour cette stratégie les réponses données sont toujours cohérentes. Voyons donc dès à présent des exemples d'arguments d'adversaires.

Problème de recherche du minimum : Nous avons en entrée une séquence de taille n, $X = \langle X_1, \ldots, X_n \rangle$ et nous avons une relation d'ordre sur ces éléments que l'on peut donc comparer. Nous désirons obtenir en sortie l'élément minimal de la séquence. Dans ce jeu, les seules questions autorisées sont celles où l'on compare deux éléments de la séquence. La stratégie utilisée par l'adversaire consiste à construire la séquence d'entrée $X = \langle X_1, \ldots, X_n \rangle$ au fur et à mesure qu'il donne des réponses aux questions de l'algorithme. Au commencement, la séquence est initialisée avec uniquement la valeur 0 pour tous ses éléments. Lorsque le joueur pose une question de type "Est-ce que $X_i < X_j$?" pour $i, j \in [n]$, l'adversaire a alors deux cas possibles :

1. Soit $X_i = X_j = 0$, dans ce cas il répondra systématiquement par "oui" à la question du joueur. Il faut ensuite mettre à jour la séquence X. Pour ce faire, nous ajoutons 1 à toutes les valeurs non-nulles de X. Nous fixons également $X_j = 1$.
2. Soit $X_i \neq X_j$, il suffit alors de répondre correctement à la question en fonction des valeurs de X_i et de X_j. Pour ce cas, il n'y a aucune mise à jour à effectuer dans la séquence X.

Nous pouvons alors remarquer que cette stratégie permet à l'adversaire d'être cohérent dans ses réponses. Ces deux cas conservent l'ordre entre les éléments. Dans le deuxième cas, nous ne faisons aucune mise à jour et donc il est évident que l'ordre entre les éléments est inchangé. Remarquons en particulier qu'un nombre qui est de valeur nulle est alors plus petit que tous les autres éléments qui ont déjà été modifiés par la mise à jour du premier cas. Soit X_i et X_j les éléments qui ont été comparés dans le premier cas, nous savons donc que $X_i = X_j$. Remarquons que pour tout $k \in [n]$ si $X_k > X_j = 0$ alors après la mise à jour, nous avons X_k qui devient $X_k + 1$ et donc $X_k + 1 > 1$, X_k reste donc plus grand que X_j après la mise à jour. De plus, comme nous ajoutons la même quantité à toutes les valeurs non nulles, l'ordre reste identique entre ces valeurs après mise à jour.

Maintenant que nous avons vu que cette stratégie d'adversaire fonctionne, nous pouvons obtenir une borne inférieure au problème de la recherche du minimum.

Proposition 1. *Le problème de la recherche d'un élément minimum dans une séquence d'entrée S de taille n admet une borne inférieure en $n-1$ comparaisons.*

Démonstration. L'algorithme peut trouver une solution lorsqu'il ne reste plus qu'un seul 0 dans le tableau de l'adversaire. En effet, s'il ne reste qu'un seul 0, si l'algorithme demande à l'adversaire si cet élément est plus petit qu'un autre, ce dernier répondra systématiquement "oui" d'après le deuxième cas de notre stratégie. Cet élément est donc plus petit que tous les autres dans la séquence et est ainsi le minimum que l'algorithme peut donner en solution. Si il reste au moins deux 0 dans le tableau, il est impossible de conclure pour l'algorithme, puisqu'il existe au moins deux éléments dans le tableau dans la séquence dont l'algorithme ne peut déterminer lequel est le plus petit. La condition du 0 unique dans le tableau est donc nécessaire et suffisante. Remarquons que l'unique façon pour l'algorithme de retirer des 0 dans le tableau de l'adversaire est de forcer le premier cas, lorsque l'on compare $X_i = X_j = 0$. De plus, il ne peut retirer qu'un seul 0 par question de cette façon. Il est alors nécessaire pour l'algorithme de forcer ce cas au moins $n-1$ fois afin que le tableau de l'adversaire ne contienne plus qu'une unique valeur à 0, ce qui demande alors nécessairement de poser au moins autant de questions. Nous pouvons donc conclure la preuve, quel que soit l'algorithme qui joue contre cet adversaire, il est nécessaire de faire au moins $n-1$ comparaisons. □

Remarque. Si nous cherchons à trouver le maximum au lieu du minimum, nous pouvons utiliser le même argument. En effet, chercher le maximum dans une séquence $X = \langle X_1, \ldots, X_n \rangle$ revient à chercher le minimum dans $X' = \langle -X_1, \ldots, -X_n \rangle$. Il suffit donc que l'adversaire réponde aux questions de l'algorithme en générant X' à la place de X.

Problème de recherche du minimum et du maximum : Nous avons, une fois de plus, une séquence d'éléments que nous pouvons ordonner. Cette fois-ci, nous voulons avoir en sortie l'élément minimum et maximum de cette séquence. Ce problème est proche du précédent, mais nous voulons également obtenir dans le même temps ce maximum. Comme la recherche est simultanée, la borne inférieure n'est pas nécessairement une somme des deux bornes, c'est ce que nous allons montrer avec cet exemple.

Les questions autorisées pour ce jeu sont les mêmes que pour le jeu précédent. Une fois de plus, nous allons définir une stratégie d'adversaire pour déterminer une borne inférieure à ce problème. Pour maintenir la cohérence, l'adversaire va, premièrement, construire une séquence de la taille de l'entrée n. Ce tableau sera rempli premièrement par des symboles dont nous allons voir la signification. Le premier symbole est $+$ qui signifie que l'élément est potentiellement le maximum. Le deuxième symbole est $-$ qui signifie que l'élément est potentiellement le minimum. Enfin, le dernier symbole est $\pm$ qui signifie que l'élément peut à la fois être le minimum ou le maximum. Au début, comme l'algorithme ne sait absolument rien sur l'entrée, chaque élément peut alors potentiellement être soit le minimum soit le maximum. Nous initialisons donc la séquence par tous ses éléments égaux à $\pm$. Nous allons également retenir un compteur c initialisé par la valeur 1. Au cours du jeu, si il devient certain qu'un élément n'est ni le maximum ni le minimum de la séquence, une valeur dépendant du compteur c lui sera alors attribuée. Regardons maintenant dans le détail les différents cas de questions que l'algorithme pourrait poser. Nous donnons pour tous ces cas la réponse à donner et les mises à jours nécessaires à effectuer dans la séquence

$X_i \setminus X_j$	$-$	$+$	$\pm$	cst.
$-$	non	oui	oui	oui
$+$	non	oui	non	non
$\pm$	non	oui	oui	oui
cst.	non	oui	oui	$X_i < X_j$

(a) Réponses à $X_i < X_j$

$X_i \setminus X_j$	$-$	$+$	$\pm$	cst.
$-$	$X_i \leftarrow -c$ $c \leftarrow c+1$	r.à.m.	$X_j \leftarrow +$	r.à.m.
$+$	r.à.m	$X_i \leftarrow c$ $c \leftarrow c+1$	$x_j \leftarrow -$	r.à.m.
$\pm$	$X_i \leftarrow +$	$X_i \leftarrow -$	$X_i \leftarrow -$ $X_j \leftarrow +$	$X_i \leftarrow -$
cst.	r.à.m.	r.à.m.	$X_j \leftarrow +$	r.à.m.

(b) Modifications après réponse

FIGURE 3.1 – La stratégie à employer par l'adversaire dans un jeu de recherche simultanée du minimum et du maximum dans une séquence $X = \langle X_1, \ldots, X_n \rangle$. Pour certains $i, j \in [n]$ nous donnons la stratégie à suivre lorsque l'algorithme pose la question $X_i < X_j$. Le tableau (a) donne les réponses à donner pour tous les cas tandis que le tableau (b) donne les modifications à faire après avoir donner la réponse. Le terme r.à.m. dans le tableau (b) signifie qu'il n'y a rien à modifier.

X ainsi que du compteur c. La stratégie et les mises à jours à effectuer sont résumées dans les deux tableaux de la figure 3.1.

Nous allons regarder dans le détail la troisième ligne de ces tableaux, où nous faisons la comparaison $X_i < X_j$ pour certains $i, j \in [n]$ avec $X_i = \pm$. Le but de l'adversaire est de faire en sorte de laisser un maximum de "doutes" à l'algorithme et c'est ce que nous arrivons à faire avec cette stratégie. Ainsi, quand $X_j = -$, l'adversaire doit répondre par "non" ce qui signifie que $X_j \leq X_i$, donc X_j reste potentiellement un minimum mais X_i ne peut être qu'un maximum puisqu'il est forcément plus grand que X_i, d'où la mise à jour $X_i \leftarrow +$. Si l'adversaire avait choisi de répondre par "oui" à la place, alors nous aurions $X_i < X_j$, et donc X_j ne peut plus être un minimum, mais en plus, X_i ne peut plus être un maximum. Nous aurions donc ainsi eu à faire deux modifications et de la même façon, nous pouvons considérer que l'adversaire donne deux fois plus d'informations à l'algorithme, ce qui n'est pas une bonne idée quand le but est de faire durer la partie. Le raisonnement est similaire pour la colonne suivante où $X_j = +$. Pour la colonne suivante, $X_j = \pm$, quelle que soit la réponse, nous avons une symétrie évidente et nous donnons donc la même quantité d'informations à l'algorithme. Nous avons donc choisi de répondre par "oui" et de faire les modifications nécessaires (i.e. $X_i \leftarrow -$ et $X_j \leftarrow +$). La dernière colonne est le cas où X_j est une constante. Pour ce cas, peu importe la réponse, nous perdons l'indétermination qui dit que X_i est potentiellement le minimum ou le maximum. Une fois encore, nous répondons par défaut "oui", et par conséquent X_i ne peut plus être le maximum d'où la modification $X_i \leftarrow -$.

Nous allons maintenant donner l'intuition pour montrer que l'adversaire reste cohérent dans ses réponses. Nous avons déjà vu dans l'exemple précédent, que les modifications étaient cohérentes avec les réponses. Remarquons qu'il y a deux cas pour lesquels nous assignons une constante à un élément X_i. Dans les deux cas nous comparons X_i avec un X_j qui a le même symbole simple $+$ ou $-$. Pour le cas où $X_i = -$, nous fixons alors $X_i = -c$, tandis que pour le cas où $X_i = +$, nous le fixons à $X_i = c$. Tout élément qui est donc potentiellement minimum par l'attribution du symbole $-$ peut alors être considéré comme un nombre strictement négatif tandis que tout élément qui est potentiellement un maximum par l'attribution du symbole $+$ peut alors être considéré comme un nombre strictement positif. Cette considération devient vraie au moment où le symbole a été attribué et restera vraie jusqu'à la fin du jeu. De ce fait si $X_i = -$ et si $X_j = +$ la réponse "oui" reste cohérente jusqu'à la fin du jeu. Comme les valeurs attribuées ne changent plus jusqu'à la fin du jeu, la comparaison entre deux constantes est elle aussi inchangée. Enfin, comme nous incrémentons le compteur c pour les deux cas $X_i = X_j = -$ et $X_i = X_j = +$, l'ordre est préservé entre les éléments jusqu'à la fin du jeu. Par exemple si $X_i = X_j = -$, $X_i = -c$ après modification. Si plus tard, nous faisons une comparaison entre $X_j < X_k$ pour un certain $k \in [n]$, avec $X_j = X_k = -$, alors le compteur c aura déjà été incrémenté au moins une fois et donc nous aurons à ce moment $X_i > -c$ donc après modification de X_j par $-c$, nous aurons toujours et ce jusqu'à la fin du jeu que $X_j < X_i$, la réponse reste donc cohérente.

Nous allons maintenant utiliser cette stratégie d'adversaire pour démontrer la proposition suivante :

Proposition 2. *Le problème de la recherche simultanée du minimum et du maximum dans une séquence d'entrée S de taille n admet une borne inférieure en $\frac{3n}{2} + \mathcal{O}(1)$ comparaisons.*

Démonstration. Le terme en $\mathcal{O}(1)$ ne sert qu'à capter la parité de n, ainsi, pour plus de simplicité, nous allons donc considérer ici uniquement le cas où $n > 0$ est pair. L'algorithme peut découvrir le minimum et le maximum de la séquence, uniquement quand il ne reste plus qu'un unique élément dont le symbole est $-$ et un unique élément dont le symbole est $+$ et plus aucun élément $\pm$. En effet, c'est une condition suffisante, puisque dans ce cas il est évident que l'élément avec le signe $-$ est le minimum de la séquence et l'élément avec le symbole $+$ est le maximum. C'est également une condition nécessaire, si il reste au moins deux éléments avec soit le symbole $-$ soit le symbole $+$, alors si on prend deux de ces éléments de même symbole nous ne savons pas les comparer et il est donc nécessaire de poser la question à l'adversaire. De même, si il reste au moins un élément avec un symbole $\pm$, alors il est impossible de savoir si cet élément est potentiellement un minimum ou un maximum. Il faut donc faire une comparaison de cet élément avec un autre élément quelconque dans la séquence.

Comme nous pouvons le remarquer sur la Figure 3.1, quand nous comparons un élément dont le symbole est $\pm$, ce dernier ne sera pas directement remplacé par une constante mais soit par $-$ soit par $+$. L'algorithme doit donc faire en sorte d'éliminer tous ces symboles, il ne peut parvenir qu'à en éliminer au plus 2 en faisant des comparaisons entre deux éléments X_i et X_j, pour certains $i, j \in [n]$, tels que $X_i = X_j = \pm$. De cette façon comme n est pair, il est nécessaire de faire au moins $\frac{n}{2}$ comparaisons pour parvenir à éliminer tous ces symboles $\pm$. Notons par n_+ le nombre d'éléments qui se verront attribuer le

symbole + au cours du jeu. Nous notons respectivement par n_- le nombre d'éléments qui se verront attribuer le symbole $-$ au cours du jeu. Nous avons bien évidemment $n = n_- + n_+$. Pour éliminer un symbole $-$, l'unique façon d'y parvenir est de comparer $X_i = X_j = -$ pour certains $i, j \in [n]$. Ce type de comparaisons n'élimine qu'un seul des deux symboles et il est donc nécessaire de faire au moins $n_- - 1$ de ces comparaisons. De la même façon, pour éliminer les symboles $+$, il faut forcer les comparaisons entre $X_i = X_j = +$ pour certains $i, j \in [n]$. Il faut donc faire au moins $n_+ - 1$ de ces comparaisons. Pour parvenir à trouver une solution, l'algorithme va donc avoir à faire au moins

$$\frac{n}{2} + (n_- - 1) + (n_+ - 1) = \frac{3n}{2} - 2$$

comparaisons, ce qui nous permet de conclure. □

Cette borne est la meilleure possible, nous verrons dans le chapitre 5 qu'il existe un algorithme qui dans le pire cas atteint cette borne.

Problème de la fusion de listes triées Nous avons en entrée deux listes d'éléments comparables triées dans le même ordre. En sortie, nous souhaitons obtenir une seule liste contenant tous les éléments de ces deux listes et cette liste doit être triée dans le même ordre que les deux listes à l'entrée. Ce problème a de multiples applications, mais la plus connue est, sans doute, l'utilisation en tant que sous-routine dans les algorithmes de type *tri fusion* où l'on appelle récursivement l'algorithme pour trier plusieurs sous-listes qui sont ensuite fusionnées par cette routine. Une autre application connue, mais qui n'apparaît pas directement quand on lit ce problème, est qu'il est possible de faire une recherche d'un élément dans une liste triée L. En effet, si l'on transforme l'élément en une liste de taille 1 alors elle est forcément triée, nous pouvons alors faire en sorte de fusionner cette liste de taille 1 avec la liste L, la position où se trouve alors cet élément nous donne un encadrement et si l'élément est dans L alors, soit l'élément précédent, soit le suivant seront de même valeur. Par la suite, nous considérerons que tous les éléments des deux listes sont distincts. Cette considération est sans conséquence, nous pourrions imposer un système de priorité entre les deux listes et l'ordre d'apparition des éléments dans les deux listes pour aider à la décision et avoir un ordre pour lequel on peut considérer que tous les éléments sont distincts. Nous allons maintenant définir une stratégie d'adversaire pour ce problème et déduire une première borne inférieure. Par la suite, nous posons $X = \langle x_1, \ldots, x_k \rangle$ et $Y = \langle y_1, \ldots, y_p \rangle$ les listes à l'entrée respectivement de tailles k et p. La stratégie d'adversaire consiste à maintenir un graphe orienté $G = (V, E)$ dont chaque sommet représente un élément d'une des deux listes d'entrées et dont chaque arc $(x, y) \in E$ signifie que l'on sait que $x < y$. À l'initialisation, le graphe est constitué de deux composantes connexes correspondant à deux arbres unaires. Chaque arbre représente alors une de ces deux listes. Ces deux composantes apparaissent car nous savons que les deux listes sont triées. La Figure 3.2 représente la configuration initiale pour $k = 4$ et $p = 3$.

Remarque. À tout moment ce graphe est acyclique. En effet, si il existait un cycle nous aurions une contradiction avec la propriété de transitivité de la relation d'ordre.

Dans ce jeu, les questions autorisées ne concernent que la comparaison entre deux éléments, le premier élément doit appartenir à la première liste, le deuxième élément doit appartenir à la seconde liste. Nous ignorons donc les questions entre deux éléments d'une même liste pour lesquels la réponse est triviale de par l'hypothèse que ces deux listes sont triées. Nous noterons par la suite pour tout $x \in V$ et $y \in V$ l'ensemble de tous les chemins allant de x jusqu'à y par $\Gamma(x, y)$. Nous noterons par la suite la fonction de longueur $\lambda : \cup_{x,y \in V} \Gamma(x, y) \mapsto \mathbb{N}$, qui à un chemin entre deux sommets x et y de V associe son nombre de sommets. En particulier, l'algorithme peut trouver une solution, si et seulement si, il existe un chemin C entre deux sommets x et y tel que $\lambda(C) = p+k$. Cette condition revient à dire qu'il existe un chemin dans le graphe qui passe par tous les sommets, nous prouvons par la suite que cette condition est nécessaire et suffisante pour que l'algorithme soit capable de trouver une solution.

Lemme 3. *L'algorithme peut trouver une solution, si et seulement si, il existe un chemin passant par tous les sommets du graphe.*

Démonstration. Remarquons que, si nous avons dans ce graphe un chemin de taille ℓ, $C = \langle c_1, \ldots, c_\ell \rangle$, nous savons par définition que C est trié. Ainsi, si il existe un chemin passant par tous les sommets du graphe, ce chemin définit la liste triée que nous voulons obtenir en sortie. Considérons maintenant que nous avons trouvé une solution, mais qu'il n'existe aucun chemin passant par tous les sommets du graphe. Il existe alors au moins deux sommets $x \in V$ et $y \in V$ pour lesquels il n'existe aucun chemin qui passent par ces deux sommets. Or, comme nous avons une solution, nous connaissons l'ordre entre x et y. Par construction du graphe, il doit donc exister un chemin reliant x à y, ce qui est une contradiction. □

Nous pouvons nous servir de ce lemme pour faire en sorte de retenir des informations sur le graphe. Pour chaque sommet $x \in V$, nous notons par $\pi(x) = \max\{\lambda(C) \mid \forall y \in V, \forall C \in \Gamma(y, x)\}$ la longueur du plus long chemin dans le graphe qui s'arrête à x. De la même façon, nous notons par $\tau(x) = \max\{\lambda(C) \mid \forall y \in V, \forall C \in \Gamma(x, y)\}$ la longueur du plus long chemin dans le graphe qui commence par x. Nous avons mis ces valeurs en étiquette de chaque sommet de la configuration initiale de la Figure 3.2 sous la forme $(\pi(x), \tau(y))$ pour tout sommet $x \in V$. Le lemme précédent dit alors que l'algorithme peut obtenir une solution uniquement lorsqu'il existe un sommet $x \in V$ tel que $\tau(x) = k + p$.

Remarque. Par définition de π et de τ, nous avons que pour tout sommet $x \in V$, $\pi(x) + \tau(x) = q_x + 1$, où $q_x = \max\{\lambda(C) \mid \forall y \in V, \forall z \in V, \forall C = \langle \ldots, x, \ldots \rangle \in \Gamma(y, z)\}$ est la longueur du plus long chemin passant par x. La preuve se fait par l'absurde, si l'on suppose que cette relation est fausse alors, on aurait une contradiction sur le fait qu'au moins $\pi(x)$ ou $\tau(x)$ ne serait pas un maximum.

Lorsque l'algorithme pose une question $x < y$, pour deux éléments x, y dans nos deux listes, voici comment l'adversaire doit répondre. Il répond "oui" si $\pi(x) + \tau(y) \leq \pi(y) + \tau(x)$ et ajoute l'arc (x, y) dans le graphe. Dans le cas contraire il répond "non" et ajoute l'arc (y, x) dans le graphe. Il est alors nécessaire de mettre à jour les valeurs retenues pour les fonctions π et τ pour chaque sommet. L'opération à appliquer pour faire ces mises à jour est la suivante, soit deux sommets u et v de V tels que $(u, v) \in E$.

— Si $\tau(u) < \tau(v) + 1$ alors on modifie $\tau(u)$ par $\tau(v) + 1$.

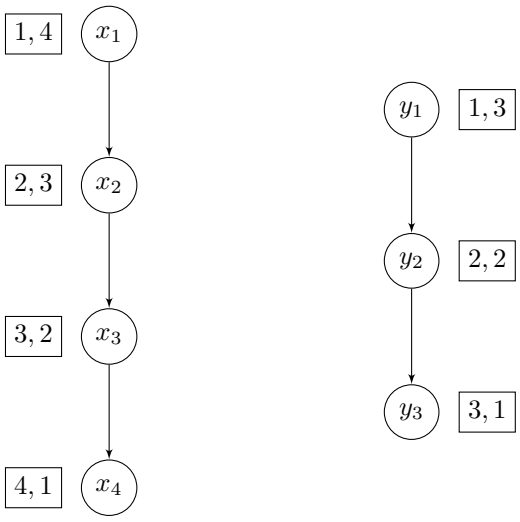

FIGURE 3.2 – Cette figure donne la représentation, stockée par l'adversaire, de l'entrée pour le problème de la fusion de listes triées au début du jeu. À côté de chaque sommet x sont affichées la valeur $\pi(x)$ à gauche et la valeur $\tau(x)$ à droite. Par exemple $\pi(x_2) = 2$ et $\tau(x_2) = 3$.

— Si $\pi(v) < \pi(u) + 1$ alors on modifie $\pi(v)$ par $\pi(u) + 1$.

Cette opération est réitérée jusqu'à stabilité, c'est-à-dire lorsque plus aucune des contraintes ci-dessus est vraie. Si l'algorithme pose la question $x \geq y$, il suffit de répondre le contraire de la question $x < y$ et d'exécuter les mêmes opérations décrites, ci-dessus, en fonction de la réponse à $x < y$. Ces opérations mettent correctement à jour les valeurs de π et de τ. Regardons le cas de la modification de τ, le cas de π est similaire. Nous regardons pour chaque arc entre (u, v) quel est le plus grand chemin qui commence par u. Après l'ajout d'un nouvel arc, le meilleur chemin est alors soit rester le même, soit il s'agit du chemin qui passe par l'arc (u, v) et qui ensuite passe par le meilleur chemin commençant par v.

Nous avons illustré un exemple de partie entre l'algorithme classique où les deux plus petits éléments des deux listes sont comparés avec notre stratégie d'adversaire. Cet exemple est montré en Figure 3.1.

Proposition 3. *Après une comparaison entre deux éléments, la longueur du plus long chemin dans le graphe est au plus augmentée de* 1.

Définition 12. Nous appellerons par la suite une comparaison qui, lorsqu'elle est effectuée par l'algorithme, fait augmenter la taille du plus long chemin, une comparaison utile.

Démonstration. Pour qu'une comparaison soit utile, il est nécessaire de comparer deux éléments x et y appartenant à deux chemins distincts. Posons q_1 et q_2 la longueur des plus longs chemins passant chacun respectivement par x et y. Nous pouvons supposer, sans nuire à la généralité de la preuve, que $q_1 \leq q_2$. Soit q la longueur des plus longs chemins dans le graphe. Par définition, nous avons alors $q_1 \leq q_2 \leq q$. Il existe alors $\alpha \in [0, q_1 - 1]$ tel que $\pi(x) = 1 + \alpha$

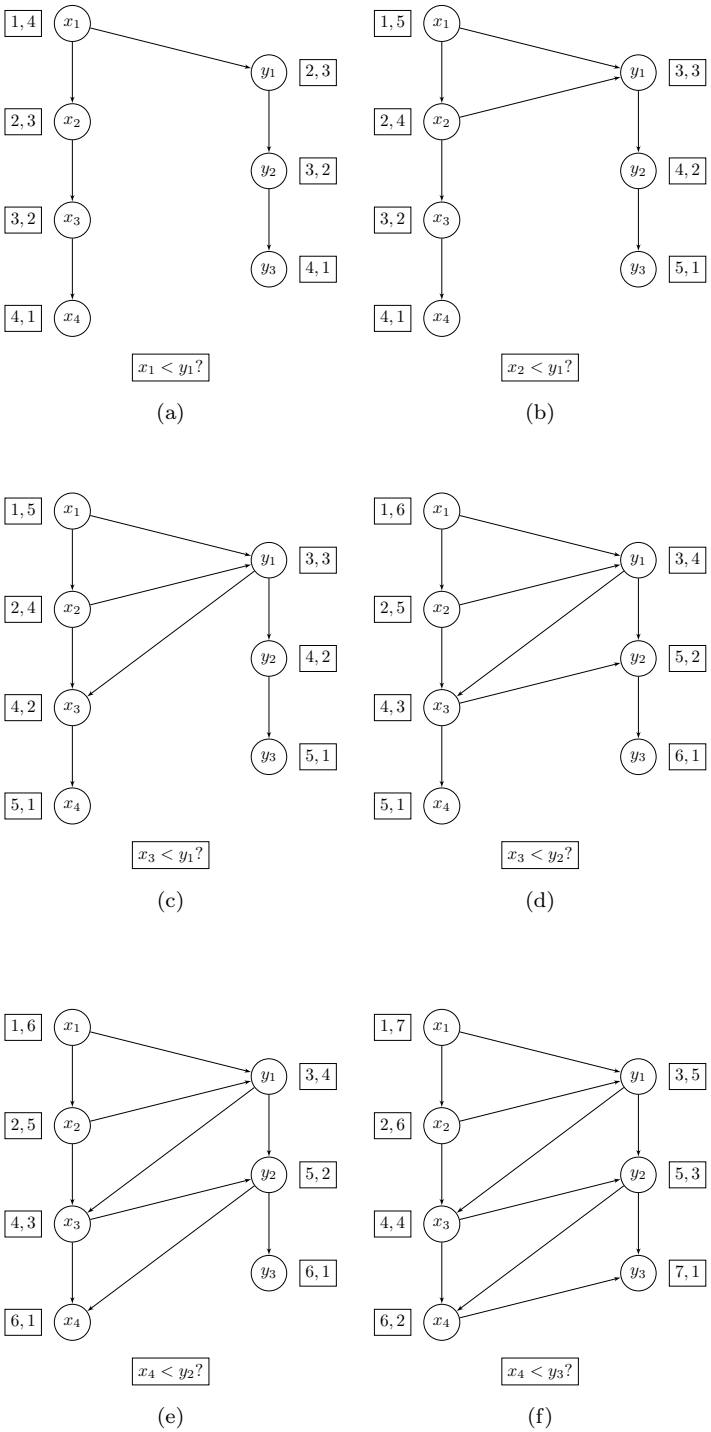

FIGURE 3.1 – Un exemple de jeu entre l'algorithme classique qui compare les deux éléments les plus petits des deux listes contre l'adversaire.

et $\tau(x) = q_1 - \alpha$. De même, il existe $\beta \in [0, q_2 - 1]$ tel que $\pi(y) = 1 + \beta$ et $\tau(y) = q_2 - \beta$. Après qu'une réponse à une question ait été donnée, la nouvelle longueur du chemin le plus long est donnée par

$$p' = \min(q_1 + 1 + \delta; q_2 + 1 - \delta) > p,$$

avec $\delta = \beta - \alpha$. Si $\delta < 0$, alors $p' = q_2 + 1 + \delta < q + 1$ ce qui est une contradiction à l'hypothèse que la comparaison est utile. Si $\delta > q_2 - q_1$, alors $p' = q_1 + 1 - \delta < q + 1$ ce qui est une autre contradiction à cette hypothèse. Le dernier cas est $\delta \in [0, q_2 - q_1]$, dans ce cas nous avons

$$q_1 + 1 \leq min(q_1 + 1 + \delta; q_2 + 1 - \delta) \leq q_2 + 1 \leq q + 1. \tag{3.1}$$

La comparaison est alors utile si $q_2 = q$ et l'on a alors une augmentation de la longueur du chemin le plus long de 1 ce qui est le résultat annoncé. □

Remarque. Si l'on regarde encore plus dans le détail le dernier cas, il est possible de voir qu'il est nécessaire d'avoir $q_1 = q_2 = q$ et donc ce cas se résume à avoir $\delta = 0$. Nous obtenons ainsi le lemme qui suit.

Lemme 4. *Si une comparaison entre x et y est utile, alors il existe deux chemins distincts de longueurs maximales dans le graphe, dont l'un passe par x et l'autre passe par y. De plus $\pi(x) = \pi(y)$ et $\tau(x) = \tau(y)$.*

Démonstration. On a vu précédemment qu'une comparaison est utile dans la preuve de la Proposition 3 uniquement pour $\delta \in [0, q_2 - q_1]$ et que dans ce cas l'Équation (3.1) est vérifiée. Comme $q_2 + 1 \leq q + 1$, il est alors nécessaire d'avoir $q_2 = q$. Pour avoir $\min(q_1 + 1 + \delta; q_2 + 1 - \delta) = q_2 + 1$, il y a potentiellement deux valeurs de δ qui sont $\delta = 0$ et $\delta = q_2 - q_1$. Supposons que $q_1 < q_2$, nous avons alors pour $\delta = q_2 - q_1$ que $q_1 + 1 + \delta = q_2 + 1$ et $q_2 + 1 - \delta = q_1 + 1$ ce qui donne $\min(q_1 + 1 + \delta; q_2 + 1 - \delta) = q_1 + 1$. Pour le cas $\delta = 0$, nous montrons de la même façon que $\min(q_1 + 1 + \delta; q_2 + 1 - \delta) = q_1 + 1$. Il y a donc contradiction sur le fait que la comparaison est utile et donc $q_1 \geq q_2$. Comme nous avons $q_1 \leq q_2$, nous obtenons alors que $q_1 = q_2$. Comme $q_1 = q_2$, nous avons alors que $\delta \in [0, 0]$ et ainsi $\delta = 0$. □

La Proposition 3 nous permet de donner une première borne inférieure.

Théorème 5. *Le problème de la fusion de deux listes triées de tailles p et k admet une borne inférieure en nombre de comparaisons définie par la relation* $\min(p, k)$.

Démonstration. À l'initialisation, le plus long chemin est de longueur $\max(p, k)$. L'algorithme peut trouver une solution lorsque le plus long chemin est de longueur $p + k = \max(p, k) + \min(p, k)$. Comme d'après la Proposition 3, une comparaison ne peut faire augmenter la longueur du plus long chemin d'au plus 1, il est nécessaire de faire au moins $\min(p, k)$ comparaisons utiles pour pouvoir atteindre cette longueur. □

Il doit être possible d'affiner ce raisonnement afin d'obtenir des bornes plus fines, mais, nous nous contenterons de cette borne, notre but étant ici de montrer un dernier exemple d'argument d'adversaire.

Analyse amortie

Dans le cadre de l'analyse amortie, nous nous intéressons à une suite de n opérations que l'on peut effectuer sur une structure de données particulière. Le terme amortie vient de l'idée que dans une suite de ces opérations, les plus coûteuses seront amorties par les moins coûteuses. C'est une notion que l'on retrouve en comptabilité lorsqu'une entreprise fait un investissement coûteux dans l'immédiat mais qui, s'il est comptabilisé sur une longue durée, comme si nous payons une partie de la totalité de sa somme tous les mois, sera amorti.

Définition 13. Soit une structure de données pour laquelle nous appliquons n opérations. Nous notons pour $i \in [n]$, son *coût réel* par c_i. Une analyse amortie consiste à associer à chacune de ces opérations un *coût amorti* $\widehat{c_i}$ de façon à avoir la propriété suivante qui est vérifiée

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i}. \tag{3.2}$$

Remarque. Cette relation n'impose pas qu'il existe une façon unique d'associée des coûts amortis à une opération. En effet, nous pourrions, par exemple, ajouter une quantité positive à chacun des coûts amortis et nous obtiendrions de nouveaux coûts pour lesquels la relation serait toujours vérifiée. De plus, cette relation montre directement que le coût amorti total donne un majorant du coût réel total de ces opérations. Si nous parvenons à démontrer que le choix des coûts amortis permet de vérifier l'équation, nous pouvons donc nous servir de l'analyse amortie pour en déduire des bornes supérieures de la totalité du coût réel de l'ensemble de ces opérations.

Par la suite, nous allons montrer deux méthodes qui permettent d'obtenir des coûts amortis valides au sens de l'Équation (3.2). La première méthode est appelée la *méthode de l'agrégat* et la deuxième est appelée la *méthode comptable*.

La méthode de l'agrégat : Cette méthode est probablement la plus simple et la plus directe pour faire une analyse amortie. Cette méthode ne nous permet pas de déterminer une borne supérieure au coût réel de l'ensemble des opérations effectuées sur la structure. Elle permet cependant d'avoir une première approche concrète d'analyse amortie. L'idée est d'associer à chacune des opérations le même coût amorti, à savoir le coût moyen du coût réel des n opérations. Plus formellement, nous associons pour tout $i \in [n]$,

$$\widehat{c_i} = \frac{\sum_{i=1}^{n} c_i}{n}.$$

Il est évident que pour ce cas l'Équation (3.2) est vérifiée puisque

$$\sum_{i=1}^{n} \widehat{c_i} = n\frac{\sum_{i=1}^{n} c_i}{n} = \sum_{i=1}^{n} c_i.$$

Voyons maintenant comment calculer le coût amorti par cette méthode sur un exemple. Nous allons nous intéresser à une structure de pile pour laquelle nous pouvons faire trois types d'opérations :

— L'empilement consiste à ajouter en haut de la pile un nouvel élément donné en paramètre.
— Le dépilement simple consiste à retirer l'élément en haut de la pile.
— Le dépilement multiple consiste à itérer d'un paramètre k l'opération de dépilement simple. Si k est plus grand que la taille de la pile, alors tous les éléments sont dépilés.

Il est facile d'obtenir des implantations de piles permettant de faire les deux premières opérations avec un coût en temps constant $\mathcal{O}(1)$. Comme l'opération de dépilement multiple consiste à itérer $\min(k, s)$ fois l'opération de dépilement simple, avec s la taille de la pile au moment où l'opération est exécutée. Nous avons que le coût en temps de cette opération est en $\mathcal{O}(\min(k, s)) = \mathcal{O}(s)$. Dans le pire cas, l'opération de dépilement multiple peut donc s'avérer onéreuse en temps de calcul. Si nous faisons n opérations à la suite en partant d'une pile vide initialement, nous avons qu'à tout moment $s \leq n$. Ainsi une première borne supérieure que l'on obtient consiste à considérer que l'on fait $\Theta(n)$ opérations de dépilement multiple. Dans ce cas, nous obtenons alors que le total des coûts en temps des n opérations est en $\mathcal{O}(n^2)$. Cette borne n'est cependant pas très fine, on peut sentir que l'opération de dépilement multiple ne peut pas être aussi coûteuse si elle est exécutée trop fréquemment, puisque la taille de la pile se retrouve en conséquence réduite. Remarquons qu'à chaque élément inséré dans la pile, cet élément ne peut être impliqué que pour au plus deux opérations, celle qui a empilé cet élément, et le dépilement simple provenant d'une des deux opérations de dépilement. De ce fait, comme nous avons vu que $s \leq n$, nous avons que le coût total en temps des n opérations est $\mathcal{O}(n)$. Ainsi le coût amorti de chacune de ces opérations est

$$\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1).$$

Il est intéressant de remarquer que dans ce cas le coût amorti de toutes ces opérations ne dépend pas de n et est constant. Du point de vue d'un utilisateur de la structure, nous pouvons alors considérer que quel que soit le nombre d'opérations que nous faisons sur cette pile, chacune de ces opérations s'effectuent en temps constant. Le coût de l'opération de dépilement multiple étant grossièrement amorti par son coût moyen.

La méthode comptable : Cette méthode consiste à maintenir un invariant pour faire en sorte que pour tout $k \in [n]$, nous avons

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}.$$

En particulier, nous avons quand $k = n$, l'Équation (3.2) qui est vérifiée. À la différence de la méthode de l'agrégat, il est donc ici tout à fait possible d'avoir des coûts amortis différents par types d'opérations. Une façon de visualiser la méthode comptable, est de considérer que nous associons un compte en banque virtuel à notre structure de données. Lorsqu'une opération est effectuée, elle crédite ce compte d'un montant correspondant à son coût amorti. Simultanément, le compte est débité du coût réel de l'opération. L'invariant que l'on maintient revient à dire que ce compte en banque n'est jamais à découvert après chaque

opération. Lorsqu'une opération a un coût amorti plus petit que son coût réel, on dit que cette opération est *sous-facturée*. Une opération dont le coût amorti est plus grand que son coût réel est dit *sur-facturée*. En un sens, nous pouvons dire que les opérations sur-facturées payent pour les opérations sous-facturées.

Reprenons l'exemple précédent avec notre structure de pile. Nous avions fait la remarque qu'un élément dans la pile pouvait entraîner des coûts de deux façons différentes, à savoir quand il est ajouté et quand il est supprimé de la pile. Cette remarque peut nous guider pour le choix des coûts amortis de chaque opération. L'idée est de considérer que l'opération d'empilement va être sur-facturée, elle va payer son propre coût réel et payer en avance le coût réel du dépilement de l'élément qu'elle vient d'ajouter. Comme l'opération d'empilement coûte 1 et que le dépilement de cet élément coûte 1 également, le coût amorti de l'opération d'empilement est alors de 2. Comme la suppression d'un élément dans la pile a déjà été payée en avance par l'opération d'empilement, les opérations de dépilement simple et multiple n'ont alors plus rien à payer. Ces deux opérations ont donc un coût amorti de 0. Nous voyons bien ici, que pour cette attribution de coût amorti, il est impossible pour la structure d'être à découvert après chaque opération effectuée sur la structure. Nous avons donc à la fin l'Équation (3.2) qui est vérifiée. Comme le coût amorti maximum est de 2 pour toutes les opérations effectuées sur la pile, nous avons alors

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i} \leq \sum_{i=1}^{n} 2 \leq 2n.$$

Et donc, nous avons montré que le coût réel total est en $\mathcal{O}(n)$. Ce que nous venons de réaliser est très intéressant pour obtenir des bornes supérieures de la complexité d'un algorithme. Ici, nous avons démontré que le compte en banque de la structure n'est jamais à découvert. De ce fait, nous pouvons majorer le coût réel total par le coût amorti total d'après l'Équation 3.2. Il se trouve que le coût total amorti est simple à majorer ce qui nous permet d'obtenir une borne supérieure du coût réel. Ce schéma que nous venons d'utiliser peut être utilisé pour déterminer des bornes supérieures en lissant les coûts réels par les coûts amortis. Nous utiliserons, par ailleurs, la méthode comptable pour déterminer la complexité dans le pire cas de TimSort au chapitre 4.

3.2 La machine RAM

Lorsque nous souhaitons faire l'analyse de la complexité d'un algorithme, nous sommes confrontés à un problème. En effet, nous avons vu précédemment, que l'analyse d'algorithmes consistait à obtenir l'asymptotique de fonctions de coût qui permettent de, par exemple, donner une estimation du temps ou de la mémoire requise au cours de l'exécution d'une implantation de l'algorithme sur une machine réelle. On peut alors se demander en quoi la fonction choisie est pertinente, qu'est-ce qui nous permet, par exemple, dans le cas d'algorithmes de tri de considérer que le nombre de comparaisons ou que le nombre de déplacements mémoire donne une bonne estimation du temps d'exécution sur une machine réelle. La vérité est que, bien souvent lorsque l'on fait l'analyse d'algorithmes, nous considérons que son implantation est exécutée sur une machine abstraite à savoir le modèle RAM, qui est un acronyme pour Random Access

Machine.

Voyons concrètement comment fonctionne ce modèle. Une machine RAM est constituée d'une unité de logiques et d'arithmétiques ALU, qui est capable de faire toutes les opérations arithmétiques standards en une même durée de temps. Concernant sa mémoire, la machine est constituée d'un nombre illimité de registres qui sont eux-mêmes de taille infinie. Un registre peut être vu comme une case dans la mémoire permettant d'accueillir une valeur. Ce registre est également nommé avec, par exemple, un nombre entier, que l'on appelle son adresse. Un registre est donc ainsi caractérisé par un couple (**adresse**, **valeur**).

Remarque. Cette dernière considération peut être étendue à une machine où chaque registre a une taille fixée. Pour stocker un registre de plus grande taille, il suffit alors d'utiliser une collection de plusieurs registres, comme par exemple un tableau.

Cook et Reckhow [19] qui, à notre connaissance, ont posé les bases de la théorie de la complexité dans le cadre du modèle RAM, ont proposé que le temps d'accès d'un registre en lecture et en écriture soit logarithmique en la valeur du registre. L'utilisation du modèle RAM reste une bonne approximation de ce que font les ordinateurs modernes et reste donc un modèle pertinent pour analyser la complexité d'un algorithme. Il peut toutefois être intéressant de considérer des extensions de ce modèle pour affiner les résultats de ces analyses afin d'être plus proche de ces derniers.

En plus de faire des accès directs dans les registres, un programme fonctionnant sur une machine RAM peut également faire des accès indirects, c'est à dire accéder à un registre dont l'adresse est pointée par un autre registre. Le modèle RAM est fixé à un seul programme, mais il est également possible d'avoir une architecture de Von Neumann, où le programme est situé également dans la mémoire, nous parlons alors de modèle RASP. Nous avons donc avec le modèle RASP, une architecture abstraite qui s'approche du modèle des ordinateurs que nous utilisons dans la pratique, avec une mémoire à accès aléatoire.

Dans la pratique, les machines modernes ont des similitudes avec le modèle RAM mais ces dernières possèdent quelques caractéristiques supplémentaires. De ce fait, le choix des fonctions utiles de coût pour l'analyse en temps et en mémoire reste assez souvent pertinent pour comparer en complexité les algorithmes, même sur des machines réelles récentes. Nous ajoutons cependant un petit bémol à tout cela. Dans le cadre de l'analyse de la complexité, nous avons introduit la notation de Landau. Avec cette notation, nous avons également introduit une notion de variables cachées qui sont les constantes multiplicatives que nous utilisons pour borner la fonction de coûts par une autre fonction. Lorsque deux algorithmes sont de complexités similaires sous cette notation, il devient alors intéressant de regarder de plus près les valeurs de ces constantes pour comparer ces derniers. Ces constantes sont cependant dépendantes du modèle RAM. Il peut alors arriver qu'en pratique pour deux algorithmes dont on connaît les constantes cachées, qu'il y ait des surprises et que l'algorithme que nous pensions être le plus rapide, car sa constante cachée est la plus petite, soit au final plus lent que le deuxième. C'est par exemple ce qui se produit pour les algorithmes de recherche simultanée du minimum et du maximum dans une séquence et dont nous discuterons au chapitre 5. Les raisons de ces résultats surprenants proviennent alors généralement de l'incomplétude du modèle RAM concernant des caractéristiques modernes d'architectures qu'utilisent nos ordi-

nateurs actuels. Le choix de nos fonctions de coût ne serait alors plus suffisant pour comparer les algorithmes et il faudrait s'intéresser à des coûts associés à ces caractéristiques que nous ignorons dans le modèle RAM. Pour être rigoureux sur la comparaison de deux algorithmes dont les fonctions de coût d'intérêts dans le modèle RAM sont relativement du même ordre de grandeur, il nous faudrait en vérité faire l'analyse des implantations de ces derniers sur la machine utilisée en considérant que nous connaissons tout de ses caractéristiques. Bien qu'il soit difficile de procéder ainsi, nous allons dans les prochaines sections discuter de deux caractéristiques à savoir la *hiérarchie mémoire* et la *prédiction de branchement*. Nous verrons en quoi ces caractéristiques diffèrent des hypothèses considérées dans le modèle RAM et nous verrons comment affiner nos analyses en considérant de nouvelles fonctions de coût.

3.3 Les hiérarchies de mémoires

Nous allons donc commencer par discuter de la hiérarchie mémoire qui est, comme nous l'avons vu précédemment, une des caractéristiques d'architecture de nos machines actuelles. Si dans le modèle RAM, deux algorithmes font un nombre d'opérations du même ordre de grandeur, s'intéresser aux constantes cachées dans ce modèle peut s'avérer insuffisant pour les comparer. En effet, il est possible que l'algorithme qui est considéré comme le plus lent dans ce modèle soit le plus rapide en pratique sur une véritable machine. Une des raisons possibles est alors que cet algorithme gère mieux les mémoires et qu'il gagne considérablement du temps de cette façon. Pour ce genre de cas, le modèle RAM n'est donc plus suffisant pour comparer de tels algorithmes une fois implantés sur un ordinateur. Il est alors nécessaire de le compléter.

3.3.1 Concept et définition

Dans le modèle RAM, nous considérons qu'il existe une unique zone de mémoire dans laquelle est stockée un ensemble de registres. Si nous considérons que ces registres sont de tailles fixes, alors les temps d'accès dans ces derniers sont tous égaux. Cette zone constitue alors un unique niveau de mémoire. Nous avons donc dans le cas du modèle RAM, une unique zone pour de multiples registres. Dans nos machines actuelles, le fonctionnement est différent, nous avons en réalité plusieurs niveaux de mémoires pour plusieurs registres. Chaque niveau est paramétré par *une taille mémoire*, autrement dit sa capacité en quantité d'informations en bits qu'il peut stocker, et le *temps d'accès* à la lecture et à l'écriture. Pour chaque niveau, ces paramètres sont différents de tous les autres niveaux. En général, il existe une relation entre la taille mémoire et le temps d'accès. La taille mémoire d'un niveau est quasiment proportionnelle à son temps d'accès. Cela s'explique pour des raisons de coûts des matériaux utilisés qui sont plus chers pour les mémoires à accès rapide. Il y a donc moins de mémoire dont le temps d'accès est court sur une machine pour éviter que cette dernière ne soit trop chère à produire et à vendre. Le paramètre du temps d'accès implique alors qu'il existe un ordre hiérarchique entre les différents niveaux. Intuitivement, les niveaux qui ont un temps d'accès rapide, sont donc les plus petits en taille également et sont donc des ressources plus critiques. Il est alors important de savoir exploiter cette mémoire sur des données dont nous faisons très fréquemment des

accès.

Nous allons particulièrement nous intéresser ici aux niveaux de mémoires qui sont les plus proches du cœur d'un processeur puisque c'est principalement ces mémoires que nous utilisons dans la pratique lorsque nous implantons nos algorithmes. Nous allons donc dès à présent discuter de la mémoire cache et des niveaux les plus couramment existant dans la plupart de nos machines. Le processeur possède lui-même un ensemble de registres très restreint. Cet ensemble de registres forme le niveau de mémoire le plus bas possible avec le temps d'accès qui correspond à un cycle du processeur. Nous avons ensuite un premier niveau de cache, noté $L1$, à capacité faible mais rapide qui est partagé en deux zones, une zone pour mémoriser les instructions à exécuter et une zone pour les données. Les niveaux supérieurs sont ensuite en général partagés par les instructions et les données. Nous avons ainsi, pour la plupart des machines récentes, jusqu'à 3 niveaux supplémentaires $L2$, $L3$ et $L4$. Une fois atteint ce dernier niveau de cache, nous avons une mémoire bien plus grande qui est souvent la mémoire vive.

Il serait bien évidemment plus pratique d'avoir une mémoire comme pour le modèle RAM avec le meilleur temps d'accès mais c'est évidemment compliqué pour des raisons de prix. De la même façon, l'intérêt d'utiliser ce système de hiérarchie mémoire est de pouvoir gagner du temps, comparé à un modèle RAM qui n'utiliserait qu'un seul type de mémoire.

3.3.2 Gestion des accès

Nous avons dit précédemment qu'il était nécessaire d'exploiter au maximum la mémoire de plus bas niveau pour gagner du temps, voyons donc dès à présent comment le processeur gère ces différentes mémoires pour optimiser les temps d'accès.

Lorsqu'il y a un accès, par exemple, la lecture d'une variable dans un programme en cours d'exécution, voici ce qui va se passer. Dans un premier temps, le processeur va essayer de chercher cette donnée dans un de ses registres, si il ne s'y trouve pas, il va alors chercher dans la mémoire de cache $L1$. De la même façon, si cette dernière n'est pas dans le cache $L1$, il va alors chercher dans le cache $L2$. Le processeur va ainsi parcourir la mémoire en cherchant dans les niveaux supérieurs jusqu'à parvenir à le trouver. Lorsque le processeur ne parvient pas à trouver la donnée dans un niveau, nous avons alors ce que nous appelons une *erreur de cache*. Lorsqu'il y a une erreur de cache, le processeur va déplacer la mémoire vers des niveaux de cache les plus bas exploitant ainsi un principe de *temporalité*. Ce principe fait une hypothèse sur la façon dont les données sont parcourues. Cette hypothèse est qu'une donnée qui a été fraîchement utilisée a de forte chance d'être utilisée à nouveau dans un avenir proche. Un exemple de données qui sont fréquemment utilisées sont les itérateurs qui vont être très fréquemment lus et mis à jour lors de l'exécution d'une boucle, il est alors naturel de vouloir accéder rapidement à ce genre de variables en les laissant dans le cache le temps de l'exécution de cette boucle. Après une erreur, le processeur rapatrie également les données appartenant au même bloc mémoire dans les niveaux les plus bas du cache. Il faut voir ici un bloc comme un ensemble de cases voisines dans la mémoire. Procéder de cette manière permet d'exploiter alors un deuxième principe qui est celui de *localité*. Ce principe fait lui aussi une hypothèse sur la façon dont les données sont parcourues. Cette hypothèse est

que les données au voisinage d'une donnée, qui a été fraîchement utilisée, ont de fortes chances d'être utilisées dans un futur proche. Un exemple où ce genre de situations se produisent est le cas d'un parcours linéaire d'un tableau, il est alors fréquent de parcourir les éléments du premier élément jusqu'au dernier. De ce fait, par principe de localité, le premier élément ainsi que ses voisins dans un même bloc vont être chargés dans le cache puis vient le bloc suivant et ainsi de suite. Quand nous parcourons un tableau de cette façon, nous produisons le moins d'erreurs de cache possibles. Il est possible d'observer ce phénomène sur nos machines en calculant, par exemple, la somme des éléments d'un tableau en procédant en un parcours du premier au dernier élément, et en choisissant une permutation au hasard pour décrire le parcours. Nous observons, en général, que le premier est souvent plus rapide que le dernier.

Nous discutions dans la section précédente de la pertinence de nos fonctions de coût. Il existe une littérature riche d'analyse d'algorithmes qui en plus de regarder les fonctions de coûts classiques utilisées avec le modèle RAM vont choisir d'étudier le coût en nombre d'erreurs de cache. Dans le modèle du cache, il existe divers paramètres à définir comme le nombre de niveaux de caches, la façon dont on crée des blocs et bien d'autres. Il existe de nombreux modèles de caches qui étendent le modèle RAM. Ces modèles simplifient le cas réel mais permettent, d'un autre côté, de faire de l'analyse de ce genre de complexités. Le lecteur intéressé par le domaine de l'analyse d'algorithmes dans le cadre d'un modèle de cache peut, par exemple, consulter l'article de LaMarca et Ladner [31] qui s'intéressent à l'influence du cache sur les performances de divers algorithmes de tri.

3.4 Le pipelining

Au début, les processeurs fonctionnaient de manière séquentielle. Un programme consiste en une suite d'instructions, et toutes ces instructions sont exécutées dans l'ordre par le processeur. Il est intéressant de noter que l'étymologie du mot ordinateur vient en particulier de cette notion d'ordonnancement des instructions dans un programme. Pour ces processeurs, il est donc nécessaire d'attendre que l'exécution d'une instruction soit terminée avant de pouvoir commencer à exécuter la suivante. Il était donc nécessaire d'attendre un certain nombre de cycles d'horloge du processeur entre l'exécution de deux instructions, ce temps correspondant au temps d'exécution de la première instruction. L'apparition des microprocesseurs à jeu d'instructions réduits RISC a permis d'introduire l'utilisation de *pipelines* qui fait en sorte d'exécuter une nouvelle instruction à chaque cycle d'horloge. Le jeu d'instructions étant réduit en un minimum de formats, il est possible de décomposer ces dernières en plusieurs étapes. De cette façon, il n'est plus nécessaire pour le processeur d'attendre la fin de l'exécution d'une instruction avant de pouvoir commencer à exécuter la suivante. La majorité des ordinateurs construits à l'heure où sont écrites ces lignes reprennent ce modèle de pipelining dont nous allons voir le détail de fonctionnement dès à présent.

Le pipeline reprend le concept de chaîne de montages et permet de paralléliser l'exécution des instructions. Concrètement, les instructions décomposées en micro-instructions sont exécutées par une unité du processeur correspondant à un **étage** du pipeline. Chacun de ces étages fonctionne indépendamment des

autres, et il est donc possible d'exécuter plusieurs micro-instructions de plusieurs instructions consécutives dans le programme pendant un même cycle d'horloge. Nous pouvons observer sur la Figure 3.2, qu'il y a un chevauchement dans l'exécution des instructions du programme.

3.4.1 Le pipeline RISC classique

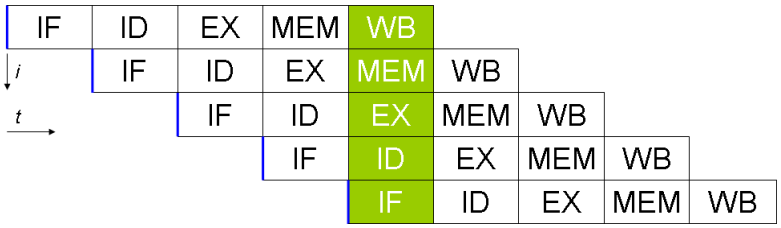

FIGURE 3.2 – Un exemple d'exécution de programme pour le cas d'un pipeline à 5 étages. L'axe des i représente l'ordre d'arrivée des instructions, tandis que l'axe t représente le temps. Par exemple au temps $t = 3$, l'instruction 1 est à l'étape d'exécution, l'instruction 2 est à l'étape de décodage de l'instruction, et l'instruction 3 est à l'étape de chargement de l'instruction. Les autres instructions sont toujours en attente. Cette image est issue de la page Wikipedia sur le pipelining.

Un exemple de processeur utilisant un pipeline de 5 étages est détaillé dans les cours de Patterson, le concepteur du processeur RISC, comme par exemple [27] et [36]. La Figure 3.2 représente une exécution d'une suite d'instructions pour ce modèle de pipeline. Voyons maintenant le détail des différents étages de ce dernier :

- **Instruction Fetch** Charge l'instruction pointée par le registre PC (Program Count) depuis la mémoire et met à jour ce pointeur. Ce registre PC doit toujours pointer sur l'adresse de la prochaine instruction à exécuter.
- **Instruction Decode** Pendant cette étape l'instruction est décodée et les opérandes sont lus dans les registres correspondants. C'est également durant cette étape que certaines conditions de branchements sont testées.
- **Execution** C'est pendant cette étape que l'opération est effectuée par l'ALU afin d'obtenir un résultat.
- **Memory Access** Dans cette étape, des opérations de lectures ou d'écritures dans la mémoire sont effectuées en fonction du type de l'instruction en cours d'exécution.
- **Write Back** Pendant cette étape, le résultat est écrit dans les registres donnés en paramètre à l'instruction.

Ce pipeline est un cas d'école et en pratique les processeurs contiennent bien plus d'étages que celui-ci. Le maximum dont nous ayons connaissance provient de l'Intel Pentium 4 Prescott avec un total de 31 étages. Les derniers processeurs d'Intel de la famille des i3, i5 et i7 semblent comptabiliser une vingtaine d'étages au moment où sont écrites ces lignes.

3.4.2 Les problèmes du pipelining

Des problèmes peuvent apparaître lors de l'utilisation d'un pipeline dans un programme. Les problèmes surgissent, par exemple, lorsqu'il y a des dépendances entre deux instructions consécutives. Il est possible d'avoir des problèmes de dépendances de données, où, par exemple, deux instructions consécutives vont écrire ou lire dans une même adresse mémoire. Un autre type de dépendances, qui nous intéressera par la suite, provient des instructions de branchement. Une instruction de branchement peut avoir à choisir la prochaine instruction après le résultat d'un test. Dans ce cas il est impossible de savoir l'instruction suivante avant l'issue de ce test.

Pour pallier ces problèmes, il existe différentes optimisations logicielles et matérielles qui peuvent être réalisées. Il est, par exemple, possible de placer une instruction au milieu de deux instructions qui sont dépendantes entre elles. L'instruction se trouvant au milieu pourra ainsi commencer à être exécutée sans avoir à attendre le résultat obtenu à la fin de l'exécution de la première instruction et va donc éviter l'apparition de *bulles*, où des étages n'ont rien à exécuter pendant une durée d'au moins un cycle du processeur. Il existe aussi des court-circuits entre les étages qui permettent d'éviter d'attendre les ré-écritures en mémoire de certaines données. Une autre optimisation est de dérouler les boucles afin d'obtenir plus d'instructions qui ne sont pas dépendantes entre-elles et d'utiliser la première méthode de réarrangement dans cette nouvelle boucle.

Dans le cas où nous avons une instruction de branchement, il y deux solutions dont nous avons connaissance. La première solution revient simplement à attendre que le test soit terminé avant de continuer. Cependant, cette solution peut s'avérer désastreuse vis-à-vis de la rentabilité du pipeline car il peut introduire un nombre important de bulles. La deuxième solution, dont nous allons discuter dans la section suivante, est d'anticiper le résultat que nous pensons être le plus probable et d'exécuter les instructions qui suivent ce résultat anticipé. Dans le cas où le résultat qui a été anticipé n'était pas le bon, le processeur doit alors annuler toutes les instructions exécutées en avance et rétablir le contexte juste avant l'instruction de branchement qui a causée cette erreur d'anticipation. Lorsqu'une erreur de ce type se produit, le temps perdu semble être à peu près équivalent au temps perdu de la première solution. L'unité dans le processeur qui permet d'anticiper le résultat d'une instruction de branchement est appelée un prédicteur de branchement et nous allons voir dans la section suivante les fonctionnements d'un ensemble de modèles de prédicteurs dont les variantes sont encore utilisées à ce jour sur nos processeurs actuels.

3.5 Prédiction de branchement

Comme vu précédemment, une des possibilités qu'a le processeur pour éviter de perdre du temps à attendre le résultat après une instruction de branchement, est d'essayer de prédire la branche qui sera exécutée. En cas d'erreur pour la prédiction du test, le processeur doit alors annuler ce qui a été exécuté en avance et recommencer à partir de l'autre branche. Quand une erreur de ce genre arrive,

le processus perd donc un temps relativement important [1] et on dit alors que le *prédicteur* utilisé par le processeur a fait une *erreur de prédiction*.

Un prédicteur est une machine à états, c'est en fonction de ces états que ce dernier prend sa décision. Par la suite, nous dirons lorsque le prédicteur anticipe que le résultat est vrai, qu'il est dans un état TAKEN, et lorsqu'il prédit que le résultat est faux, qu'il est dans un état NOT TAKEN. Dans cette section, nous ferons un parcours non-exhaustif des prédicteurs les plus connus et utilisés en pratique.

Données : Le premier élément de la suite u_0 et le nombre d'itération n
Résultat : u_n

```
1 pour i ← 1 à n faire
2     si u_{i-1} pair alors
3         u_i ← u_{i-1} / 2
4     sinon
5         u_i ← 3u_{i-1} + 1
6     retourner u_n
```

Algorithme 2 : Algorithme calculant le n-ème terme de la suite de Syracuse commençant par la valeur u_0.

Pour comprendre le fonctionnement de ces prédicteurs, rien ne vaut un exemple, et nous allons considérer leurs fonctionnements sur quelques exécutions de l'Algorithme 2 qui consiste en un calcul du n-ème terme de la suite de Syracuse paramétrée par une valeur initiale choisie u_0. La condition à la ligne 2 va ainsi produire une suite de résultats qui seront soit vrais, représenté par la lettre V, soit faux, représenté par la lettre F. Il est alors possible d'écrire cette séquence sous la forme d'un mot $w(n, u_0)$, par exemple $w(5, 42) = VFVVV$, $w(7, 1024) = VVVVVVV$ et $w(17, 52692) = VVFVVVVVFVFVVVFVF$. Pour chaque prédicteur p, nous allons regarder son nombre d'erreurs de prédiction produites à la ligne 2 que nous notons $\Upsilon_p(n, u_0)$.

3.5.1 Le prédicteur statique

Un premier prédicteur assez primitif mais qui a l'avantage de n'utiliser aucune mémoire auxiliaire est le prédicteur statique. Celui-ci choisit toujours la même branche, son état est ainsi fixé.

Exemple. *Pour le prédicteur statique en état TAKEN, nous avons* $\Upsilon_{staticV}(5, 42) = 1$, $\Upsilon_{staticV}(7, 1024) = 0$ *et* $\Upsilon_{staticV}(17, 52692) = 5$. *Pour le prédicteur statique en état NOT TAKEN, nous avons* $\Upsilon_{staticF}(5, 42) = 4$, $\Upsilon_{staticF}(7, 1024) = 7$ *et* $\Upsilon_{staticF}(17, 52692) = 12$

3.5.2 Le prédicteur 1-bit

Le prédicteur 1-bit est un prédicteur qui, comme les autres prédicteurs que nous verrons par la suite, entre dans la catégorie des prédicteurs dynamiques. Ces derniers, contrairement aux prédicteurs statiques, utilisent une mémoire auxiliaire afin de retenir des informations sur les derniers résultats d'un test

1. Ce temps peut varier en fonction de l'architecture de la machine, et de sa façon de gérer l'erreur. Pour un processeur Intel Core i7, par exemple, une erreur de prediction peut coûter environ 15 cycles d'horloge (voir [27]).

dans le but de déterminer des régularités qui pourraient permettre, une fois exploitées, d'améliorer la prédiction. Ainsi, le principe du prédicteur 1-bit est d'utiliser un compteur qui prend pour mémoire 1-bit. Ce bit peut ainsi être utilisé pour retenir le résultat du précédent test. Le prédicteur prédira alors que le prochain test produira la même issue. Ce prédicteur peut également être représenté sous la forme d'un automate comme on peut le voir pour la Figure 3.3. Les prédicteurs dynamiques sont plus flexibles que les prédicteurs statiques, ce qui leur permet de plus facilement gérer des cas où le test va produire un certain résultat avec une proportion supérieure à 50% de l'ensemble des tests effectués.

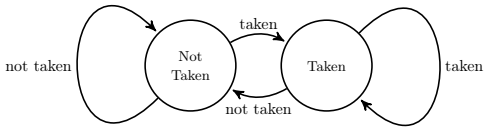

FIGURE 3.3 – prédicteur 1-bits sous forme d'automate.

Exemple. *Supposons que le prédicteur* 1*-bit soit initialement dans l'état TAKEN. La condition va produire une erreur de prédiction à partir du premier test qui sera faux, puis à chaque alternance entre un résultat vrai et un résultat faux. Nous allons représenter, en gras, les lettres correspondant aux résultats qui ont été mal prédits pour différentes valeurs de* $w(n, u_0)$*. Nous avons* $w(5, 42) = V\boldsymbol{F}\boldsymbol{V}VV$*,* $w(7, 1024) = VVVVVVV$ *et* $w(17, 52692) = VV\boldsymbol{F}\boldsymbol{V}VVVV\boldsymbol{F}\boldsymbol{V}\boldsymbol{F}\boldsymbol{V}VV\boldsymbol{F}\boldsymbol{V}\boldsymbol{F}$*. Ainsi, nous avons* $\Upsilon_{1-bit}(5, 42) = 2$*,* $\Upsilon_{1-bit}(7, 1024) = 0$ *et* $\Upsilon_{1-bit}(17, 52692) = 9$*.*

3.5.3 Les prédicteurs 2-bits

Une simple amélioration du prédicteur 1-bit consiste à doubler l'espace mémoire utilisé et ainsi d'obtenir un prédicteur utilisant 2-bits de mémoire pour stocker des informations sur les résultats précédents. Avec cette mémoire supplémentaire, le prédicteur sera en général plus robuste et permet d'éviter de changer d'état lorsque des variations apparaissent avec faible fréquence comme, par exemple, les résultats faux que l'on voit dans $w(17, 52692)$. Nous allons voir ici deux prédicteurs 2-bits que l'on retrouve souvent dans la littérature : le compteur saturé et le compteur qui fait un saut après deux erreurs de prédiction consécutives.

Compteur saturé

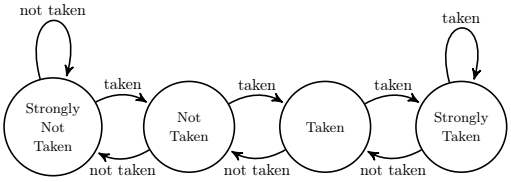

FIGURE 3.4 – prédicteur 2-bits saturé sous forme d'automate.

Le premier modèle de prédicteur 2-bits consiste en un compteur saturé, qui va associer aux valeurs 0 et 1 l'état NOT TAKEN, et pour les valeurs 2 et 3 l'état TAKEN. Soit x la valeur du compteur avant un test. Si le test est vrai, le compteur prend la valeur $min(x, x+1)$. Sinon, le compteur prend la valeur $max(x, x-1)$. Le compteur va donc s'incrémenter ou se décrémenter jusqu'à atteindre les valeurs extrêmes 0 ou 3, d'où son nom de compteur saturé. Ces valeurs extrêmes 0 et 3 sont respectivement appelé STRONGLY NOT TAKEN et STRONGLY TAKEN car on sait que l'issue prédite par ces états s'est produite au moins deux fois consécutivement. Le compteur 2-bits saturé peut également, comme pour le prédicteur 1-bit, être représenté sous forme d'automate (voir Figure 3.4).

Exemple. *En considérant que l'état initial du prédicteur est STRONGLY TAKEN. Nous reprenons les mêmes exemples utilisés pour le prédicteur* 1*-bit et la même convention qu'une lettre en gras est une des lettres qui provoque une erreur de prédiction. Nous avons alors* $w(5,42) = V\boldsymbol{F}VVV$, $w(7,1024) = VVVVVVV$ *et* $w(17,52692) = VV\boldsymbol{F}VVVV\boldsymbol{F}V\boldsymbol{F}VVV\boldsymbol{F}V\boldsymbol{F}$. *Ainsi nous avons,* $\Upsilon_{2-bit-sat}(5,42) = 1$, $\Upsilon_{2-bit-sat}(7,1024) = 0$ *et* $\Upsilon_{2-bit-sat}(17,52692) = 5$. *Nous remarquons que ces résultats sont identiques à ceux du prédicteur statique pour l'état TAKEN. Ce prédicteur reste cependant bien plus flexible, si on remplaçait tous les* V *par des* F *et tous les* F *par des* V *dans cet exemple, alors il ferait moins d'erreurs de prédictions que le statique pour l'état TAKEN.*

Compteur avec saut

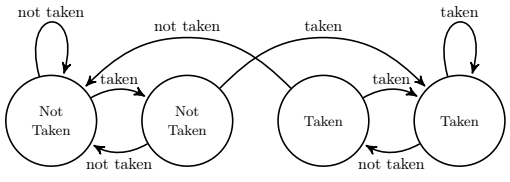

FIGURE 3.5 – prédicteur 2-bits avec saut sous forme d'automate.

Le deuxième modèle de prédicteur 2-bits est un compteur qui utilise les mêmes états que le premier, mais ne fait pas les transitions entre ces états de la même façon. Les transitions sont les mêmes sauf pour les états TAKEN et NOT

TAKEN. Il est encore une fois possible de représenter ce modèle à l'aide d'un automate et c'est ce que nous faisons dans la Figure 3.5. Les transitions qui ont été remplacées peuvent ainsi être vues comme des sauts dans la représentation en automate. L'idée de ces sauts est que le prédicteur a fait deux fois la même erreur de prédire un mauvais résultat, il y a donc de fortes chances que le prochain résultat soit encore le même, d'où l'intérêt de se mettre dans l'état le plus fort qui prédit cette dernière issue.

Exemple. *Nous allons utiliser le même exemple que le compteur saturé mais cette fois-ci en considérant que l'état initial est STRONGLY NOT TAKEN. Nous avons alors* $w(5,42) = \boldsymbol{VF}\,\boldsymbol{VV}V$, $w(7,1024) = \boldsymbol{VV}VVVVV$ *et* $w(17,52692) = \boldsymbol{VVF}VVVVV\,\boldsymbol{F}V\,\boldsymbol{F}VVV\,\boldsymbol{F}V\,\boldsymbol{F}$. *Ainsi* $\Upsilon_{2-bit-saut}(5,42) = 3$, $\Upsilon_{2-bit-saut}(7,1024) = 2$ *et* $\Upsilon_{2-bit-saut}(17,52692) = 7$.

3.5.4 Prédicteur 3-bit

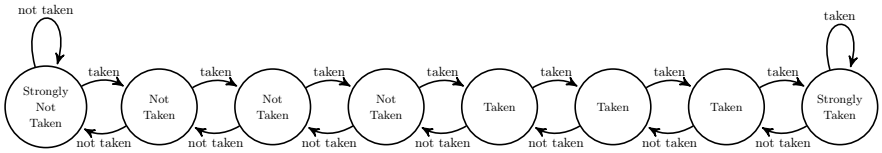

FIGURE 3.6 – prédicteur 3-bits saturé sous forme d'automate.

Il est encore une fois possible d'obtenir des prédicteurs différents en utilisant de la mémoire supplémentaire. Il existe bien évidemment un compteur 3-bits saturés fonctionnant sur le même principe que le 2-bits saturés mais qui possède deux fois plus d'états, celui-ci peut être représenté sous forme d'automate (voir Figure 3.6).

Exemple. *Reprenons notre exemple en considérant que, l'état initial est NOT TAKEN. Nous avons alors* $w(5,42) = \boldsymbol{VF}\boldsymbol{V}VV$, $w(7,1024) = \boldsymbol{V}VVVVVV$ *et* $w(17,52692) = \boldsymbol{VV}\boldsymbol{F}VVVVV\,\boldsymbol{F}V\,\boldsymbol{F}VVV\,\boldsymbol{F}V\,\boldsymbol{F}$. *Ainsi* $\Upsilon_{3-bit-sat}(5,42) = 3$, $\Upsilon_{3-bit-sat}(7,1024) = 1$ *et* $\Upsilon_{3-bit-sat}(17,52692) = 6$.

3.5.5 Prédicteur global

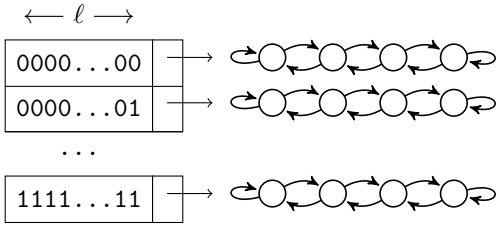

FIGURE 3.7 – Un prédicteur global avec une table d'histoires de table de taille 2^ℓ et des prédicteurs 2-bits saturés associé pour chaque histoire.

Les prédicteurs qui ont été présentés jusqu'à présent sont tous des prédicteurs locaux, c'est à dire qu'un prédicteur va être associé uniquement à une seule instruction de branchement. Cependant, ces prédicteurs ont le désavantage de ne pas tenir compte du comportement répétitif qui peut être associé à un branchement comme par exemple le fameux cycle triviale $4-2-1$ dans l'Algorithme 2 qui va produire le même motif VVF. Un autre désavantage est que ces prédicteurs ne peuvent pas prendre en compte les éventuelles dépendances qu'il peut y avoir entre les différentes instructions de branchements à l'exécution d'un algorithme.

Les prédicteurs globaux permettent de remédier à ce problème en ajoutant un niveau supplémentaire. Nous allons ici présenter un type de prédicteur global qui utilise une *table d'histoires* comme niveau supplémentaire. Une représentation de ce prédicteur est donnée dans la Figure 3.7. En premier lieu, ce prédicteur va garder en mémoire les ℓ derniers résultats des précédents tests, nous définissons ℓ comme *la taille de l'histoire* du prédicteur. De plus, nous écrivons h la trace de ces résultats, que nous appelons l'*histoire* du prédicteur. L'histoire h est alors un mot binaire où 0 signifie que l'on a eu un résultat pour lequel le branchement n'a pas été pris et 1 dans le cas contraire. Par exemple, si $\ell = 4$, $h = 1011$, alors cela signifie qu'au cours des 4 dernières instructions de branchements les résultats ont été dans l'ordre chronologique TAKEN, NOT TAKEN, TAKEN puis TAKEN. Pour chaque histoire possible, le prédicteur va alors prendre une décision, cette dernière dépendra ainsi des précédentes fois qu'elle a été rencontrée au cours de l'exécution. Concrètement la table d'histoires T possède une case pour chaque histoire possible h, à chaque case $T[h]$ est associée un autre prédicteur. Sur notre figure pour tout h, nous avons $T[h]$ qui est un prédicteur 2-bits saturé. Une conséquence de cette construction est que T est constituée de 2^ℓ cases. Sur notre figure, le prédicteur a alors besoin de $2^{\ell+2} + \ell$ bits en mémoire pour fonctionner.

Soit $h = h_1 h_2 \dots h_\ell$ l'histoire au moment où une nouvelle instruction de branchement est exécutée, nous résumons le fonctionnement de ce prédicteur en trois étapes :

1. Faire la prédiction du prédicteur $T[h]$.
2. Soit r le résultat du branchement, mettre à jour $T[h]$ en fonction de r.
3. Mettre à jour l'histoire par $h' = h_2 h_3 \dots h_\ell r$.

Exemple. *Nous reprenons encore une fois notre exemple, l'état initial est* $h = 00$ *et que pour tout* h *possible* $T[h]$ *est à l'état NOT TAKEN. Ici* $\ell = 2$ *et chaque case de la table contient un prédicteur 2-bits saturé. Nous avons alors* $w(5, 42) =$ ***VF VVV****,* $w(7, 1024) =$ ***VVV****VVVV et* $w(17, 52692) =$ ***VVF VVV VV FV F*** *VVV* ***FV F****. Ainsi* $\Upsilon_{global}(5, 42) = 4$*,* $\Upsilon_{global}(7, 1024) = 3$ *et* $\Upsilon_{global}(17, 52692) = 9$*. Dans cette exemple ce prédicteur a jusque-là fait moins bien que ceux que nous avons vus précédemment, cela peut s'expliquer car nos exemples sont trop courts. Si l'hypothèse de Syracuse est vraie, alors pour tout* $n \in \mathbb{N}$ *et pour un* m *très grand, il y a de fortes chances que ce prédicteur soit alors le meilleur parmi ceux que nous avons vus jusqu'à présent pour prédire le mot* $w(m, n)$*. En effet, après le temps d'envol de la suite, le motif se répète par le motif* VVF *qui est très bien anticipé par ce prédicteur global, contrairement aux autres que nous avons vus jusqu'à présent.*

3.5.6 Autres types de prédicteurs

Dans la pratique, il existe bien d'autres prédicteurs qui peuvent être des variantes de ceux qui ont été présentés ici, mais aussi des méthodes de prédictions différentes. Il existe, par exemple, des cas d'hybridations entre les prédicteurs globaux et locaux qui rajoutent un troisième niveau à la prédiction. Ce prédicteur va utiliser, par exemple, le prédicteur global avec table d'histoires et le prédicteur 3-bits saturés simultanément pour une même instruction de branchement. Lorsque cette instruction de branchement est exécutée, la prédiction se fait en deux étapes : une sélection du meilleur prédicteur dont le score est mesuré sur l'ensemble des fois où l'instruction a été exécutée ; la prédiction en elle-même obtenue à l'aide du prédicteur sélectionné. Il est aussi possible de faire des prédicteurs en utilisant les dernières connaissances en apprentissage artificiel. Ces prédicteurs sont peu applicables dans la pratique car beaucoup trop gourmands en ressources mais ils ont tout de même été étudiés.

De nouveaux prédicteurs sont encore créés de nos jours , comme par exemple dans le cadre d'un concours qui est réalisé et organisé par le journal JILP, tous les deux ans. Ce concours réuni des membres de compagnies réputées dans l'architecture et la fabrication de microprocesseurs. Le but de ce concours est de proposer sa propre implantation de prédicteurs de branchements. Ce concours présente plusieurs catégories, comme par exemple celle du prédicteur qui est le plus rapide sous des contraintes de mémoires, ceux qui utilisent l'apprentissage artificiel et bien d'autres. Ce concours est une vraie mine d'informations quant aux prédicteurs qui pourraient être utilisés dans l'avenir, le lecteur intéressé par ce dernier est invité à regarder cette référence [4]

3.5.7 Association d'une chaîne de Markov à un prédicteur :

Nous allons maintenant voir, comment nous pouvons associer une chaîne de Markov aux prédicteurs locaux que nous avons introduits dans cette section. L'approche d'utiliser des chaînes de Markov pour analyser des algorithmes sur leur nombre d'erreurs de prédiction a déjà été réalisée dans le passé, cela a, par exemple, été fait dans les articles [28] et [34]. Les prédicteurs de branchements sont des machines à états avec transition entre ces derniers, si nous rajoutons une probabilité de prendre les transitions entre ces états, il est alors assez évident que nous avons défini une chaîne de Markov.

Si nous nous plaçons dans un cas idéal, nous pouvons supposer qu'à chaque fois qu'une instruction de branchements est exécutée, la probabilité que le résultat du test soit vrai est toujours égale à une certaine quantité $p \in [0, 1]$. De cette façon, la probabilité de prendre une transition vers un état de type TAKEN est toujours p, tandis que la probabilité de prendre une transition vers un état de type NOT TAKEN est toujours $1 - p$. La chaîne de Markov $\mathcal{M}$, associée à un prédicteur dynamique va donc consister à prendre le même ensemble d'états que le prédicteur. Soit deux états x et y dans cet ensemble, si il existe une transition de x vers y dans le prédicteur, quand le résultat du test à l'instruction de branchement associée au prédicteur est vrai, alors $\mathcal{M}(x, y) = p$. S'il existe une transition de x vers y dans le prédicteur, quand le test est faux, alors $\mathcal{M}(x, y) = 1 - p$. Nous avons déjà vu dans l'introduction sur les chaînes de Markov dans la section 2.2.6 du chapitre 2, la représentation sous forme

d'automate de la chaîne de Markov du prédicteur 1-bit qui est donnée dans la Figure 2.1. Nous donnons, par ailleurs, dans la Figure 3.8, la représentation sous forme d'automate de la chaîne de Markov du prédicteur 2-bits saturé.

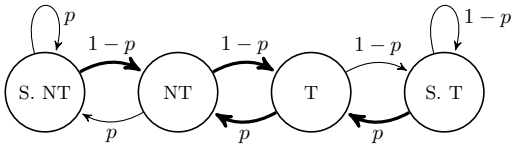

FIGURE 3.8 – La chaîne de Markov associée au prédicteur 2-bits saturé. Les transitions en gras correspondent à des erreurs de prédiction lorsqu'elles sont prises.

Pour tous les prédicteurs locaux que nous avons vus jusqu'à présent, il est évident que leurs chaînes de Markov sont irréductibles et apériodiques, de ce fait ils convergent vers une distribution stationnaire unique de par l'application du Théorème 3 et ce, indépendamment de la distribution initiale. Pour déterminer ces distributions, il suffit de déterminer le noyau de $\mathcal{M}^T - Id$ qui dépend de p. Nous avons déjà fait ce calcul pour le cas du prédicteur 1-bit et nous ne pensons pas qu'il est nécessaire de détailler les calculs pour les autres prédicteurs. Sous la distribution stationnaire, il est cependant intéressant de connaitre la probabilité de faire une erreur de prédiction. Ces probabilités sont directement données ci-après avec μ_1 qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 1-bit, μ_2 qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 2-bits saturé, μ_2' qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 2-bits avec sauts, et enfin μ_3 qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 3-bits saturé. Nous avons ainsi

$$\mu_1(p) = 2p(1-p), \tag{3.3}$$

$$\mu_2(p) = \frac{p(1-p)}{1-2p(1-p)}, \tag{3.4}$$

$$\mu_2'(p) = \frac{2p^2(1-p)^2 + p(1-p)}{1-p(1-p)}, \tag{3.5}$$

et enfin

$$\mu_3(p) = \frac{p(1-p)(1-3p(1-p))}{1-2p(1-p)(2-p(1-p))}. \tag{3.6}$$

Pour étendre le modèle RAM qui considère que le coût d'une instruction de branchement est toujours le même quel que soit le contexte dans lequel cette dernière est exécutée, il peut être intéressant d'étudier une nouvelle fonction de coût qui est le nombre total d'erreurs de prédiction qu'un algorithme peut effectuer au cours de son exécution. Bien entendu, ce nombre est dépendant du modèle de prédiction utilisé et il est donc important de fixer ce dernier pour pouvoir faire une étude. Les équations que nous venons de donner permettent de faire l'analyse de ce coût pour les modèles de prédicteurs locaux. Nous verrons au cours du chapitre 5 que nous pouvons aussi, dans certain cas, y parvenir

pour le prédicteur global avec table d'histoires. Bien que les prédicteurs utilisés en pratique sont différents des modèles que nous avons présentés, il s'agit bien souvent d'améliorations de ces derniers. Nous pouvons supposer que le choix d'un prédicteur est justifié par des résultats après une série de benchmarks pour vérifier leurs bons fonctionnements. Il n'est pas garanti que ces derniers fonctionnent toujours mieux que les modèles que nous avons présentés au cours de ce chapitre, mais nous pouvons espérer que ce sera souvent le cas ou bien que la différence de performance ne devrait pas être trop grande en nombre d'erreurs de prédiction. De ce fait, analyser le nombre d'erreurs de prédiction sur les modèles simples que nous avons vus dans ce chapitre devrait nous donner une bonne idée sur la performance réelle d'un algorithme une fois implanté sur un véritable ordinateur.

3.5.8 La simulation

Comme nous venons de le dire précédemment, les performances des prédicteurs utilisés en pratique ont tendance à être meilleures que les modèles que nous avons présentés dans cette section. Il peut cependant être intéressant de pouvoir vérifier nos résultats théoriques en faisant de la simulation de ces prédicteurs. De cette façon, lorsque nous annonçons un résultat pour un prédicteur en particulier, il nous sera possible de vérifier ce dernier en simulant ce-dit prédicteur.

C'est pour cette raison que nous avons développé une petite librairie Python pour pouvoir simuler ces prédicteurs et savoir exactement le nombre d'erreurs de prédiction qu'ils peuvent effectuer en exécutant des implantations en python de nos algorithmes. La librairie est donnée en Annexe A et repose sur la classe Predictor qui peut être étendue pour pouvoir créer ses propres prédicteurs. Les prédicteurs implantés sont le statique qui prédit toujours faux, le 1-bit, le 2-bits saturé, le 2-bits avec sauts, le 3-bits saturés et le prédicteur global avec une table des historiques qui associe un des prédicteurs locaux précédemment cités.

La simulation présente un autre avantage par rapport à l'utilisation de benchmarks pour tester nos implantations. Il n'est pas toujours nécessaire de conserver la structure de donnée sur laquelle travaille un algorithme. La simulation est alors moins gourmande en mémoire, ce qui est un avantage si nous voulons tester notre algorithme pour de très grandes entrées. Par exemple, si nous voulons tester un algorithme naïf, mais optimal d'après notre argument d'adversaire, de recherche d'un minimum dans une séquence. L'algorithme en question va simplement parcourir chaque élément de la séquence et retenir au fur et à mesure le minimum qu'il a rencontré dans une variable min. Pour cet algorithme, il est facile de simuler la probabilité qu'il ait à changer la valeur de sa variable au prochain élément rencontré. Dans le cadre de l'analyse de la prédiction de branchement, simuler cette probabilité nous est suffisant. Ainsi, si nous voulons simuler cet algorithme pour la mesure du nombre d'erreurs de prédiction, il nous suffit de stocker dans une variable min le minimum rencontré jusqu'à présent. À chaque étape, nous générons un nouvel élément x au hasard qui pourrait être dans la vraie séquence, et nous comparons x avec min. Les comparaisons effectuées sont ainsi les mêmes avec même probabilité sauf que dans le cas où nous simulerions entièrement l'algorithme il nous faudrait une mémoire de taille n, alors qu'ici nous n'avons besoin que d'une mémoire de taille constante. Avec cette méthode, il est alors possible d'aller au delà des tailles limites de nos

mémoires, ce qui nous permet d'obtenir les constantes cachées de nos complexités avec une plus grande précision. De plus, comme nous réduisons l'utilisation mémoire, nous pouvons également gagner en temps d'exécution de la simulation, dans notre exemple les variables utilisées tiennent dans des registres du processeur.