

TECHNIQUES ALGORITHMIQUES ET PROGRAMMATION



Cyril Gavoille
LaBRI
Laboratoire Bordelais de Recherche
en Informatique, Université de Bordeaux
gavoille@labri.fr

23 septembre 2019
– 151 pages –



Ce document est publié sous *Licence Creative Commons « Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International (CC BY-NC-SA 4.0) »*. Cette licence vous autorise une utilisation libre de ce document pour un usage non commercial et à condition d'en conserver la paternité. Toute version modifiée de ce document doit être placée sous la même licence pour pouvoir être diffusée. <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.fr>

Licence 3 : Techniques Algorithmiques et Programmation

Objectifs : Introduire, aux travers d'exemple de problèmes simples, diverses approches algorithmiques, les programmer et les tester sur machines. Les approches abordées sont :

- Formule close ;
- Exhaustive (*Brute-Force*) ;
- Récursive ;
- Programmation dynamique ;
- Heuristique ;
- Approximation ;
- Gloutonne (*Greedy*) ;
- Diviser pour régner (*Divide-and-Conquer*) ;

Nous programmerons en **C** avec un tout petit peu d'OpenGL/SDL pour plus de graphismes. Les concepts techniques et les objets que l'on croisera seront : les algorithmes, la complexité, les graphes, les distances, les points du plan, ...

Pré-requis : langage **C**, notions algorithmiques, notions de graphes

Quelques ouvrages de références :

- *Programmation efficace*
Christoph Dürr et Jill-Jênn Vie
ELLIPSES 2016
- *The Algorithm Design Manual (2nd edition)*
Steven S. Skiena
SPRINGER 2008
- *Algorithm Design*
Robert Kleinberg et Éva Tardos
PEARSON EDUCATION 2006
- *Introduction à l'algorithmique (2e édition)*
Thomas, H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein
DUNOD 2001
- *Algorithms (4th edition)*
Robert Sedgewick et Kevin Wayne
ADDISON-WESLEY 2011

Table des matières

1	Introduction	1
1.1	Tchisla	2
1.2	Des problèmes indécidables	5
1.3	Recherche exhaustive	8
1.4	Rappels sur la complexité	17
1.4.1	Compter exactement?	18
1.4.2	Pour résumer	22
1.5	Notations asymptotiques	22
1.5.1	Exemples et pièges à éviter	23
1.5.2	Complexité d'un problème	25
1.5.3	Sur l'intérêt des problèmes de décision	26
1.6	Algorithme et logarithme	27
1.6.1	Propriétés importantes	29
1.6.2	Et la fonction $\ln n$?	33
1.7	Morale	34
	Bibliographie	37
2	Partition d'un entier	39
2.1	Le problème	39
2.2	Formule asymptotique	40
2.3	Récurrence	42
2.4	Programmation dynamique	48
2.5	Mémorisation paresseuse	50
2.6	Morale	55
	Bibliographie	56

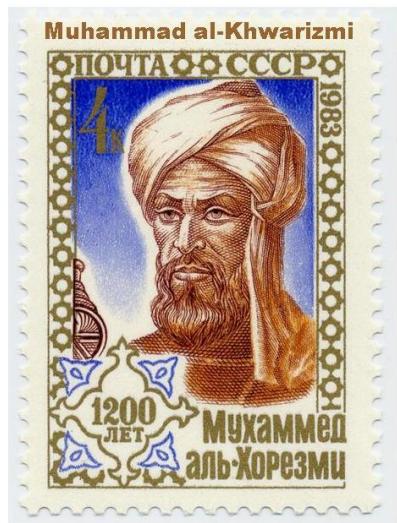
3 Voyageur de commerce	57
3.1 Le problème	57
3.2 Approche exhaustive	59
3.3 Programmation dynamique	61
3.4 Approximation	67
3.4.1 Algorithme glouton: un principe général	68
3.4.2 Problème d'optimisation	69
3.4.3 Autres heuristiques	72
3.4.4 Inapproximabilité	73
3.4.5 Cas euclidien	75
3.4.6 Une 2-approximation	76
3.4.7 <i>Union-and-Find</i>	80
3.4.8 Algorithme de Christofides	87
3.5 Morale	90
Bibliographie	91
4 Navigation	93
4.1 Introduction	93
4.1.1 <i>Pathfinding</i>	93
4.1.2 <i>Navigation mesh</i>	94
4.1.3 Rappels	96
4.2 L'algorithme de Dijkstra	97
4.2.1 Propriétés	99
4.2.2 Implémentation et complexité.	101
4.3 L'algorithme A [*]	105
4.3.1 Propriétés	107
4.3.2 Implémentation et complexité	110
4.3.3 Plus sur A [*]	110
4.4 Morale	112
Bibliographie	113
5 Diviser pour régner	115
5.1 Introduction	115

5.2 Trouver la paire de points les plus proches	118
5.2.1 Motivation	118
5.2.2 Principe de l'algorithme	118
5.2.3 L'algorithme	123
5.2.4 Complexité	123
5.2.5 Différences entre n , $n \log n$ et n^2	125
5.2.6 Plus vite en moyenne	126
5.3 Multiplication rapide	128
5.3.1 L'algorithme standard	128
5.3.2 Approche diviser pour régner	129
5.3.3 Karastuba	133
5.4 <i>Master Theorem</i>	136
5.4.1 Exemples d'applications	137
5.4.2 Explications	137
5.4.3 D'autres récurrences	140
5.5 Calcul du médian	140
5.5.1 Motivation	140
5.5.2 Tri-rapide avec choix aléatoire du pivot	142
5.5.3 Médian	142
5.6 Morale	142
Bibliographie	143

CHAPITRE

1

Introduction



|| *May the Algorithm's Force be with you.*
— Bernard Chazelle ¹

Sommaire

1.1 Tchisla	2
1.2 Des problèmes indécidables	5
1.3 Recherche exhaustive	8
1.4 Rappels sur la complexité	17
1.5 Notations asymptotiques	22
1.6 Algorithme et logarithme	27
1.7 Morale	34
Bibliographie	37

Mots clés et notions abordées dans ce chapitre :

- formule close
- indécidabilité
- instance, problème
- notation asymptotique
- recherche exhaustive, grammaire
- fonction logarithme, série géométrique

1. Voir <https://www.cs.princeton.edu/~chazelle/pubs/algorithm.html>.

1.1 Tchisla

Pour illustrer les notions du cours nous allons considérer un problème réel, volontairement complexe.

Tchisla (du russe « Числа » qui veut dire « nombre ») est une application (voir la figure 1.1) que l'on peut trouver sur *smartphone* et tablette. La première version est sortie en 2017. Le but du jeu est de trouver une expression arithmétique égale à un entier $n > 0$ mais utilisant uniquement un chiffre $c \in \{1, \dots, 9\}$ donné. L'expression ne peut comporter que des symboles parmi les dix suivants :

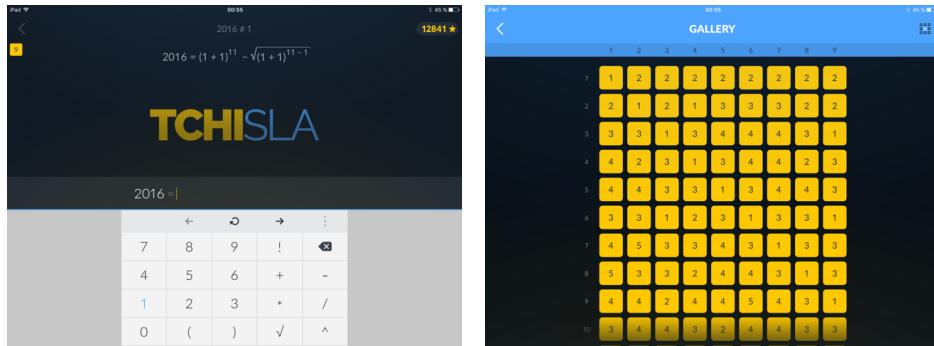
$$c + - * / ^ \sqrt{ } ! ()$$


FIGURE 1.1 – Capture d'écran de l'application Tchisla.

L'objectif est de trouver l'expression comportant le moins de fois le chiffre c , et on note $f_c(n)$ cette valeur. Par exemple, $10 = 4 + 4 + \sqrt{4}$ ce qui fait que $f_4(10) \leq 3$. En fait on ne peut pas faire mieux, si bien que $f_4(10) = 3$. On en déduit alors par exemple que $11 = 10 + 1 = 4 + 4 + \sqrt{4} + 4/4$ et donc $f_4(11) \leq 5$. Cependant, $11 = 44/4$ ce qui est optimal², et donc $f_4(11) = 3$. La figure 1.1 montre que

$$2016 = (1+1)^{11} - \sqrt{(1+1)^{11}-1}$$

et on ne peut pas faire mieux si bien que $f_1(2016) = 9$, mais aussi elle montre les premières valeurs de $f_c(n)$ pour $n = 1 \dots 10$ (lignes) et $c = 1 \dots 9$ (colonnes).

Formule close? Ce qui nous intéresse c'est donc de calculer $f_c(n)$ pour tout c et n , et bien sûr de trouver une expression correspondante avec le nombre optimal de chiffres c . Il semble que les premières valeurs de n ne laissent pas apparaître de formule évidente. La première colonne de la figure 1.1 de droite donne les dix premières valeurs qui sont :

2. On peut trouver sur Internet les solutions optimales pour tous les entiers jusqu'à quelques milliers. Dans un article scientifique [Tan15] donne les solutions optimales jusqu'à 1 000 mais sans les symboles $\sqrt{ }$ et $!$. On y apprend par exemple que $37 = ccc/(c+c+c)$ quel que soit le chiffre c .

n	1	2	3	4	5	6	7	8	9	10
$f_1(n)$	1	2	3	4	4	3	4	5	4	3

Et la table ci-après donne les dix premières valeurs de n produisant des valeurs croissantes pour $f_1(n)$. Encore une fois elle ne laisse apparaître aucun paterne particulier.

$f_1(n)$	1	2	3	4	5	6	7	8	9	10
n	1	2	3	4	8	15	28	41	95	173

En fait, comme le montre le graphique de la figure 1.2, les 200 premières valeurs de $f_1(n)$ sont visiblement difficiles à prévoir. Même si les valeurs ont l'air « globalement croissantes » avec n , on remarque qu'à cause des expressions comme

$$11 \quad 11! \quad 11!! \quad 11!!! \quad 11!!!! \quad \dots$$

il y a qu'en même une infinité de valeurs de n pour lesquelles $f_1(n) = 2$.

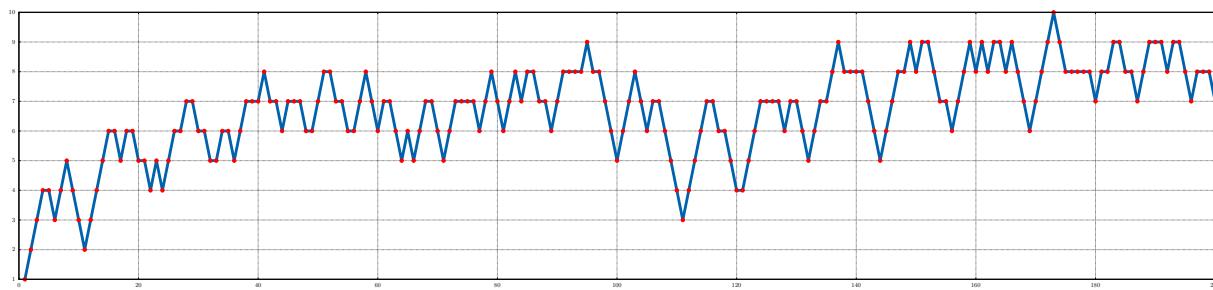


FIGURE 1.2 – Les 200 premières valeurs pour $f_1(n)$.

La fameuse encyclopédie en ligne³ des suites d'entiers ne répertorie pas cette suite-là. Certaine valeur semble plus difficile à trouver que d'autre. Pour s'en convaincre, essayez de déterminer par exemple $f_4(64)$? On a facilement $64 = 4*4*4 = 4^4/4$, mais pouvez-vous trouver une expression avec seulement deux 4? Ou encore une expression correspondant à $f_6(27) = 4$?

Parfois (et même très souvent) pour d'autres problèmes il n'y a pas de formule directe, ou plus précisément de *formules closes*, c'est-à-dire une formule arithmétique comportant un nombre fini d'opérations arithmétiques liée aux paramètres (ici c et n). Une somme ou un produit infini ne constitue pas une formule close.

Par exemple, les racines des équations polynomiales de degré 5 ne possède pas de formules closes dans le cas général. C'est un résultat issu de la théorie de Galois. Pour les calculer, on a recours à d'autres techniques, comme l'approximation et le calcul numérique. Dans pas mal de cas on peut obtenir un nombre de chiffres significatifs

3. <https://oeis.org/>

aussi grand que l'on veut. Mais le temps de l'algorithme de résolution s'allonge avec le nombre de chiffres souhaités, ce qui n'est pas le cas lorsqu'on dispose d'une formule directe.

Bien sûr, pour certaines équations polynomiales on peut exprimer les racines de manière exacte comme $x^6 = 2$. Dans ce cas il y a $2^{1/6}$ comme solution mais pas seulement⁴.

Un algorithme? S'il n'y a pas de formule close pour le calcul de $f_c(n)$, on peut alors rechercher un *algorithme*, un procédé automatique et systématique de calcul (une recette) donnant une solution à chaque entrée d'un problème donné. Notons qu'une formule close n'est qu'un algorithme (particulièrement simple) parmi d'autres.

Mais qu'entendons nous par *problème*? C'est tout simplement la description des *instances*⁵ et des sorties attendues, c'est-à-dire la relation entre les entrées et la sortie. Cette notion est partiellement capturée par le prototype d'une fonction `C` comme

```
int f(int c,int n)
```

Le problème présenté ici pourrait être formalisé ainsi, où Σ est l'alphabet des 10 symboles évoqués plus haut :

TCHISLA

Instance: Un entier $n > 0$ et un chiffre $c \in \{1, \dots, 9\}$.

Question: Déterminer une expression arithmétique de valeur n composée des symboles de Σ comportant le moins de fois le chiffres c .

Malheureusement, pour le problème TCHISLA, et donc le calcul de $f_c(n)$, trouver un algorithme n'est pas si évident que cela. Et parfois la situation est plus grave que prévue. Pour certains problèmes, il n'y a ni formule ni algorithme !

On parle de problème *indécidable* — il serait plus juste de dire *incalculable* — lorsqu'il n'y a pas d'algorithme permettant de le résoudre.

4. Dans \mathbb{C} , elles sont en fait toutes de la forme $2^{1/6} \cdot (\cos(2k\pi/6) + i \sin(2k\pi/6))$ où $k \in \mathbb{N}$. On parle de racines de l'unité et elles peuvent être représentées par six points du cercle de rayon $2^{1/6}$ centré à l'origine et régulièrement espacés. On observe alors que deux des six racines tombent sur la droite réelle : $2^{1/6}$ ($k=0$) et $-2^{1/6}$ ($k=3$).

5. Pour un algorithme on parle plutôt d'*entrées*. On réserve le terme d'*instances* pour un problème.

1.2 Des problèmes indécidables

Une équation *diophantienne* est une équation à coefficients entiers dont on s'intéresse aux solutions entières (si elles existent), comme par exemple celle-ci :

$$(x - 1)! + 1 = xy \quad (1.1)$$

Elles ont été étudiées depuis l'antiquité du nom de Diophante d'Alexandrie, mathématicien grec du IIIe siècle. Par exemple, le théorème de Wilson établit que l'équation (1.1) possède une solution si et seulement si $x > 1$ est premier. Dit autrement l'ensemble des $x > 1$ qui font parties des solutions décrit exactement l'ensemble des nombres premiers.

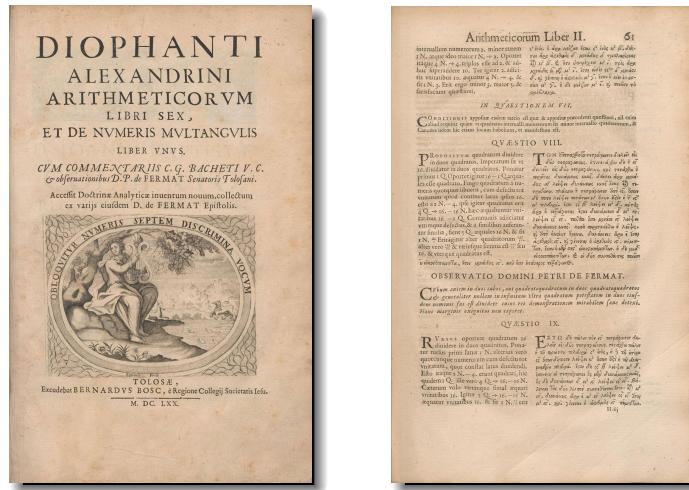


FIGURE 1.3 – Édition de 1670 de l'ouvrage de Diophante. Le texte reprend la note de 1621 (traduite ici en latin) de Pierre Fermat concernant le problème II.VIII à propos de l'équation diophantienne $x^p + y^p = z^p$: « *Au contraire, il est impossible de partager soit un cube en deux cubes, soit un bicarré en deux bicarrés, soit en général une puissance quelconque supérieure au carré en deux puissances de même degré : j'en ai découvert une démonstration véritablement merveilleuse que cette marge est trop étroite pour contenir.* » Source Wikipédia.

Ces équations, souvent très simples, sont parfois parfois extrêmement difficiles à résoudre. Le dernier théorème de Fermat en est un exemple (cf. figure 1.3). Il aura fallut 357 ans d'efforts pour démontrer en 1994, grâce à Andrew Wiles, que l'équation

$$x^p + y^p = z^p \quad (1.2)$$

n'a pas de solutions entières non nulles dès que $p > 2$. Pour $p = 2$, les solutions (x, y, z) sont appelées « triplets pythagoriciens », comme $(3, 4, 5)$. Ce n'est pas plus facile pour les systèmes d'équations diophantiennes comme

$$\begin{cases} 7xy^2 + 5y = 3z \\ 3xyz - 4y^2 = 7 \end{cases} \Leftrightarrow (7xy^2 + 5y - 3z) \cdot (3xyz - 4y^2 - 7) = 0$$

En fait, c'est même pire. Le problème de savoir si un système d'équations diophantiennes polynomiales avec au plus 11 variables possède ou non une solution (entière) est indécidable. Remarquons qu'on peut toujours se ramener à une seule équation diophantienne polynomiale.

Il faut bien distinguer les problèmes indécidables avec les problèmes sans solution. On mélange souvent les notions d'instance et de problème. Une équation diophantine est ici une instance. Elle possède ou ne possède pas de solution entière. Et pour certaines d'entre elles, c'est facile de le décider : il suffit d'appliquer un théorème. Mais le problème est de trouver une procédure systématique qui, pour *toute* équation diophantine (soit toute instance), détermine s'il existe ou pas de solution entière. Personne ne peut prétendre avoir trouvé un tel algorithme, car cet algorithme n'existe pas ! Et on sait prouver qu'il n'existe pas.

Le problème suivant est bien connu pour être indécidable.

HALTE

Instance: Un programme f avec une entrée x .

Question: Est-ce que $f(x)$ s'arrête ou boucle indéfiniment ?

Encore une fois, il est clair que chaque programme sur une entrée donnée s'arrête au bout d'un moment ou alors boucle indéfiniment. Il n'y a pas d'intermédiaire. Seulement, il n'existe pas d'algorithme qui à coup sûr peut déterminer pour tout programme f et entrée x , si l'évaluation de $f(x)$ s'arrête.

Le point commun de ces problèmes indécidables, et ce qui les rend si difficiles à résoudre, c'est qu'on arrive pas à dire si tous les cas ont été examinés et donc à s'arrêter sur la bonne décision. On peut parfaitement lister les suites d'entiers (x, y, z, p) avec $p > 2$ ou simuler l'exécution du programme $f(x)$. Si l'équation (1.2) est satisfait ou si le programme s'arrête, on pourra décider car on va s'en apercevoir. Mais sinon, comment décider ? Faut-il continuer la recherche ou bien s'arrêter et répondre qu'il y a pas de solution ou que le programme boucle ? En fait, on a la réponse pour l'équation (1.2) grâce à un théorème (Andrew Wiles). Mais on a pas de théorème pour chaque équation diophantienne possible !

Mais comment montrer qu'un problème n'a pas d'algorithme ? Supposons qu'on dispose d'une fonction `halte(f, x)` permettant de dire si une fonction `f` écrit en `C` du type `void f(int x)` s'arrête sur l'entier `x` (`=1`) ou boucle pour toujours (`=0`). La fonction `halte()`, aussi compliquée qu'elle soit, implémente donc un certain algorithme censé être correct qui termine toujours sur la bonne réponse.

Considérons le programme `loop()` ci-dessous faisant appel à la fonction `halte()` :

```

int halte(void (*f)(void*), int x); // f=pointeur de fonction

void loop(int x){
    if(halte(loop, x)) for(;;); // ici loop(x) va boucler
    return; // ici loop(x) se termine
}

```

Parenthèse. En C le nom des fonctions est vu comme un pointeur représentant l'adresse mémoire où elles sont codées en machine. On peut donc passer en paramètre une fonction simplement en spécifiant son nom sans le préfixer avec &, la marque de référencement qui permet d'avoir l'adresse où est stocké un objet. Mais ce n'est pas faux de mettre ! Ainsi on peut écrire indifféremment `halte(loop,x)` ou `halte(&loop,x)`.

La question est de savoir se qui se passe lorsqu'on fait un appel à `loop(0)`, par exemple ? D'après son code, `loop(x)` terminera sur le paramètre `x` si et seulement si `halte(loop,x)` est faux, c'est-à-dire ssi `loop(x)` boucle ! C'est clairement une contradiction, montrant que la fonction `halte(f,x)` ne peut pas être correcte pour toute fonction `f()` et paramètre `x`.

Parenthèse. Un autre exemple bien connu est la complexité de Kolmogorov. Notée $K(n)$, elle est définie sur les entiers naturels comme ceci : $K(n)$ est le nombre de caractères du plus court programme, disons écrit⁶ en C, qui affiche l'entier n et qui s'arrête. La fonction $K(n)$ n'est pas calculable par un algorithme.

Pourquoi ? Supposons qu'il existe un tel algorithme capable de calculer la fonction $K(n)$. Cet algorithme est une suite finie d'instructions, et donc peut-être codé par une fonction `K()` écrites en C dont le code comprend un total de `disons k` caractères. Ce programme est ainsi capable de renvoyer la valeur $K(i)$ pour tout entier i .

Considérons le programme `P()` ci-dessous faisant appel à la fonction `K()` et qui affiche le plus petit entier de complexité de Kolmogorov au moins n :

```

int K(int); // fonction dont le code est défini quelque part

void P(int n){
    int i=0;
    while(K(i)<n) i++;
    print("%d",i); // ici K(i)≥n
}

```

Même si cela semble assez clair, montrons que ce programme s'arrête toujours. Il s'agit d'un argument de comptage. Soit $f(n)$ le nombre de programmes C ayant moins de n ca-

6. On peut montrer que la complexité $K(n)$ ne dépend qu'à une constante additive près (la taille d'un compilateur par exemple) du langage considéré.

ractères⁷. Par définition de $K(i)$, chaque entier i possède un programme de $K(i)$ caractères qui affiche i et s'arrête. Ces programmes sont tous différents⁸. Lorsque i va atteindre $f(n)$, les $f(n) + 1$ entiers de l'intervalle $[0, f(n)]$ auront été examinées. Et tous ne peuvent pas prétendre avoir un programme qui affiche un résultat différent et qui fasse moins de n caractères. Donc $K(i) \geq n$ pour au moins un certain entier $i \in [0, f(n)]$ ce qui montre bien que $P(n)$ s'arrête toujours (à cause du test `while(K(i)<n)` qui va devenir faux). L'entier affiché par $P(n)$, disons i_n , est le plus petit d'entre eux. Et bien évidemment $K(i_n) \geq n$ à cause du `while`.

La fonction `P()` à proprement parlée fait 56 caractères, sans compter les retours à la ligne qui ne sont pas nécessaires en C. Il faut ajouter à cela le code de la fonction `K()` qui par hypothèse est de k caractères. Notons que la fonction `P()` dépend de n , mais la taille du code de `P()` ne dépend pas de n . Idem pour `K()`. Pour s'en convaincre, il faut imaginer que chaque `int` aurait pu être représenté par une chaîne de caractères `char*` donnant la liste de ses chiffres. Donc le paramètre `n` et la variable `i` ne sont que des « pointeurs » dont la taille ne dépendent pas de ce qu'ils pointent. Dans notre calcul de la taille du programme, ils ne font qu'un seul caractère, ce qui ne dépend en rien de la valeur qu'ils représentent. Donc au total, le code qui permet de calculer $P(n)$ fait $56+k$ caractères, une constante qui ne dépend pas de n .

On a donc construit un programme $P(n)$ de $56+k$ caractères qui a la propriété d'afficher un entier i_n tel que $K(i_n) \geq n$ et de s'arrêter. En fixant n' importe quel entier $n > 56 + k$ on obtient une contradiction, puisque :

- (1) $P(n)$ s'arrête et affiche l'entier i_n tel que $K(i_n) \geq n$;
- (2) $P(n)$ fait $56 + k < n$ caractères ce qui implique $K(i_n) < n$;

L'hypothèse qui avait été faite, et qui se révèle fausse, est qu'il existe un algorithme (un programme d'une certaine taille k) qui calcule la fonction $K(n)$.

1.3 Recherche exhaustive

On n'a pas formellement montré que le calcul de $f_c(n)$ ne possède pas de formule close, ni que le problème TCHISLA est indécidable. Pour être honnête, la question n'est pas tranchée, même si je pense qu'un algorithme existe. Une des difficultés est que la taille de l'expression arithmétique qu'il faut trouver pour atteindre $f_c(n)$ n'est pas bornée en fonction de $f_c(n)$. Par exemple,

$$n = 3!!!!$$

7. Bien que la valeur exacte de $f(n)$ n'a ici aucune espèce d'importance, on peut quand même en donner une valeur approximative. Si on se concentre sur les programmes écrit en caractères ASCII sur 7-bits, alors il y a au plus 2^{7t} programmes d'exactement t caractères. En fait beaucoup d'entre-eux ne compilent même pas, et très peu affichent un entier et s'arrêtent. Il y a des programmes de $t = 0, 1, 2, \dots$ jusqu'à $n - 1$ caractères, d'où $f(n) \leq \sum_{t=0}^{n-1} 2^{7t} = 2^{7n} - 1$.

8. On utilise ici « l'arrêt » dans la définition de $K(n)$. Sinon, le même programme pourrait potentiellement afficher plusieurs entiers s'il ne s'arrêtait pas. Par exemple, le code suivant de 34 caractères affiche n'importe quel entier i , n'est-ce pas ? `int i=0;for(;;) printf("%d", i++);`

est un nombre gigantesque de 62 chiffres et pourtant $f_3(n) = 1$. Et on pourrait ajouter un nombre arbitraire d'opérateurs unaires de la sorte. Et que dire du nombre suivant ?

$$n = 3!!!!/3!!!!$$

Se pose aussi le problème de l'évaluation. On pourrait être amener à produire des nombres intermédiaires de tailles titaniques, impossibles à calculer alors que n n'est pas si grand que cela. Il n'est même pas clair que l'arithmétique entière suffise comme

$$n = (\sqrt{\sqrt{3}}^{(3!!/3!)}) = 14\,348\,907$$

où les calculs intermédiaires pourraient ne pas être entiers ou rationnels ...

On va donc considérer une variante du problème plus simple à étudier.

TCHISLA2

Instance: Un entier $n > 0$ et un chiffre $c \in \{1, \dots, 9\}$.

Question: Déterminer une expression arithmétique de valeur n composée des symboles de Σ comportant le moins de caractères possibles.

L'instance est bien la même. La différence avec TCHISLA est donc qu'on s'intéresse maintenant non plus au nombre de symboles c mais au nombre total de symboles de l'expression arithmétique. Il n'y a aucune raison que les solutions optimales pour TCHISLA2 le soit aussi pour TCHISLA. Par exemple,

24 = (1+1+1+1) !	#1=4	longueur=10
= 11+11+1+1	#1=6	longueur=9

Cependant, pour résoudre TCHISLA2, on peut maintenant appliquer l'algorithme dont le principe est le suivant :

|| **Principe.** On liste toutes les expressions par taille croissante, et on s'arrête à la première expression valide dont la valeur est égale à n .

Comme on balaye les expressions de manière *exhaustive* et par taille croissante, la première que l'on trouve sera nécessairement la plus petite. Cet algorithme ne marche évidemment que pour la version TCHISLA2. Car comme on l'a déjà remarqué, pour la version originale, on ne sait pas à partir de quelle taille s'arrêter.

Cette approche, qui s'appelle « recherche exhaustive » ou « algorithme *brute-force* » en Anglais, est en fait très générale. Elle consiste à essayer tous les résultats possibles, c'est-à-dire à lister tous les résultats envisageables et à vérifier à chaque fois si c'est une solution ou non. En quelques sortes on essaye de deviner la solution et on vérifie qu'elle est bien valide.

Attention ! L'exhaustivité porte sur l'espace des solutions, sur ce qu'il faut trouver et donc, en général⁹, sur les sorties. Pas sur les entrées ! Par exemple, si le problème est de trouver trois indices i, j, k d'éléments d'un tableau T tels que $T[i] + T[j] = T[k]$, alors la recherche exhaustive ne consiste pas à parcourir tous les éléments de T (=l'entrée) mais à lister tous les triplets (i, j, k) (=les solutions) et à vérifier si $T[i] + T[j] = T[k]$. [Exercice. Trouver un algorithme en $O(n^3)$, puis $O(n^2 \log n)$. Améliorer le en $O(n^2)$ si les éléments sont des entiers naturels $< n^2$.]

Parenthèse. En toute généralité, la structure qui représente une solution et qui permet de vérifier si c'est une solution s'appelle un certificat positif. Dans l'exemple du tableau ci-dessus, le certificat est un triplet d'indices (i, j, k) vérifiant $T[i] + T[j] = T[k]$. Pour TCHISLA2 c'est une expression sur un alphabet de dix lettres. La méthode exhaustive se résume alors à lister tous les certificats positifs possibles. Généralement on impose qu'il puisse être vérifié en temps raisonnable, typiquement en temps polynomial en la taille des entrées, ce qui impose aussi que leurs tailles est polynomiales. Les cours de Master reviendront sur ces notions.

Pour être sûr de ne rater aucune expression, on peut lister tous les mots d'une longueur donnée et vérifier si le mot formé est une expression valide. C'est plus simple que de générer directement les expressions valides. En quelques sortes on fait une première recherche exhaustive des expressions valides parmi les mots d'une longueur donnée, puis parmi ces expressions on en fait une deuxième pour révéler celles qui s'évaluent à n (cf. le schéma de la figure 1.4).

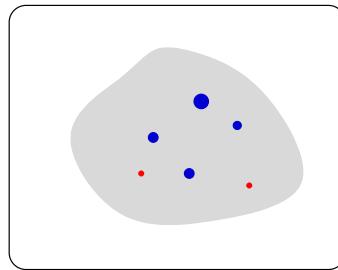


FIGURE 1.4 – Le rectangle représente l'ensemble des mots de taille $\leq 2n + 3$, ensemble qui contient forcément la solution. Parmi ces mots il y a les expressions valides (zone grise). Parmi cette zone, celles qui s'évaluent à n (disques colorés). Et enfin, parmi ces disques ceux de plus petite taille (en rouge). Il peut y en avoir plusieurs.

On va coder une expression de taille k par un tableau de k entiers avec le codage suivant pour chacun des dix symboles :

$$\begin{array}{ccccccccc} c & + & - & * & / & ^ & \sqrt{} & ! & () \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array}$$

9. Mais pas toujours ! comme les problèmes de décisions, dont la sortie est Vrai ou Faux. Par exemple, savoir un graphe possède un cycle Hamiltonien. La sortie est booléenne alors que les solutions possibles sont plutôt un cycle du graphe.

Ainsi l'expression $(c+c)/c$ sera codé par le tableau $T[] = \{8, 0, 1, 0, 9, 4, 0\}$. Chaque expression se trouve ainsi numérotée. Bien sûr certains tableaux ne représentent aucune expression valide, comme $\{8, 0, 1, 0, 9, 4, 1\}$ censé représenter $(c+c)/+$.

Générer tous les tableaux de k chiffres revient à maintenir un compteur. Et pour passer au suivant, il suffit d'incrémenter le compteur.

$T[]$	expression	validité
...
8010940	$(c+c)/c$	✓
8010941	$(c+c)/+$	✗
8010942	$(c+c)/-$	✗
...
8010949	$(c+c)/)$	✗
8010950	$(c+c)^c$	✓
8010951	$(c+c)^+$	✗
...

Pour rappel, l'algorithme d'incrémantation (c'est-à-dire qui ajoute un à un compteur), peut être résumé ainsi :

Principe. Au départ on se positionne sur le dernier chiffre, celui des unités. On essaye alors d'incrémenter le chiffre sous la position courante. S'il n'y a pas de retenue on s'arrête. Sinon, le chiffre courant passe à 0 et on recommence avec le chiffre précédent.

Pour pouvoir donner une implémentation concrète, on supposera déjà programmées les deux fonctions suivantes qui s'appliquent à un compteur T de k chiffres décimaux :

- `int Next(int T[], int k)` qui incrémente le compteur et puis retourne faux ($=0$) lorsque le compteur a dépassé sa capacité maximum, c'est-à-dire qu'il est revenu à $0\dots 0$. Ainsi, si $T[] = \{9, 9, 9\}$, alors `Next(T, 3) == 0` avec en retour $T[] = \{0, 0, 0\}$. Cette fonction s'implémente facilement, comme expliqué précédemment. Il s'agit d'un simple parcours du tableau T , et donc sa complexité est ¹⁰ $O(k)$.
- `int Eval(int T[], int k, int c)` qui renvoie la valeur n de l'expression T de taille k dans laquelle le code 0 correspond au chiffre c . Si l'expression n'est pas valide ou si $n \leq 0$ on renvoie 0, en se rappelant que le résultat est censé vérifier $n > 0$. L'évaluation d'une expression valide se fait par un simple parcours de T en utilisant une pile (cf. cours/td d'algorithmique de 1ère et 2e année). Il est facile de le modifier de sorte que pendant l'évaluation on renvoie 0 dès qu'une erreur se produit

10. La complexité est en fait constant en moyenne car l'incrémantation du i -ème chiffre de T se produit seulement toutes les 10^i incrémantations. Donc sur le total des 10^k incrémantations, 10^k nécessitent le changement du chiffre numéro 0 (le dernier); 10^{k-1} nécessitent le changement du chiffre 1; 10^{k-2} nécessitent le changement du chiffre 2; ... soit un total de changements de $\sum_{i=0}^{k-1} 10^{k-i} = \sum_{i=1}^k 10^i < 10^{k+1}/9$. En moyenne, cela fait donc $<(10^{k+1}/9)/10^k = 10/9 < 2$ chiffres à modifier.

traduisant une expression non valide (comme un mauvais parenthésage lors d'un dépilement, des opérandes ou opérateurs incorrects lors de l'empilement, etc.). Sa complexité est $O(k)$.

D'où le programme qui résout TCHISLA2.

```
int tchisla2(int c,int n){
    int T[2*n+3],k=1; // n=(c+...+c)/c, k=taille
    T[0]=0;           // plus petite expression de taille k=1
    for(;;k++)        // une condition vide est toujours vraie
        do if(Eval(T,k,c)==n) return k; // fin si s'évalue à n
        while(Next(T,k)); // passe à l'expression suivante
}
```

La fonction renvoie en fait la longueur k de la plus courte expression. L'expression elle-même se trouve dans les k premières cases de T . La ligne `int T[2*n+3]` se justifie par le fait que $n = (c+...+c)/c$ (la somme ayant n termes) qui est une expression valide de valeur n de $2n + 3$ symboles. On peut donc toujours résoudre le problème par une expression d'au plus $2n + 3$ symboles.

On pourrait se demander si on ne peut pas trouver une expression générale en fonction de c qui soit plus courte. Cette une question intéressante à part entière abordée ci-après, et qui n'est pas au programme.

Complexité. La boucle `for(;;k++)` s'exécutera au plus $2n + 3$ fois, puisque comme expliqué précédemment la plus petite expression valide a au plus $2n + 3$ symboles. Et le nombre de fois qu'on exécute les fonctions `Eval()` et `Next()`, qui prennent chacune un temps $O(k)$, est au plus 10^k , soit le nombre d'expressions de taille $k = 1, 2, 3, \dots, 2n + 3$. Au final, la complexité est¹¹ :

$$\sum_{k=1}^{2n+3} (k \cdot 10^k) < (2n+3) \cdot \sum_{k=1}^{2n+3} 10^k = O(n) \cdot 10^{2n}.$$

Il est inutile de programmer un tel algorithme (quoique?), car pour $n = 9$, le nombre d'opérations est déjà de l'ordre de 10^{18} , et ce même en oubliant le terme $O(n)$. Comme on le verra page 60, dès que le nombre d'opérations élémentaires dépasse $10^9 \times 10^9$ il faut compter 30 ans de calcul sur un processeur 1 GHz ... Certes, un ordinateur plus puissant¹² (cadence plus élevée et multi-cœurs) pourrait sans doute venir à bout du cas

11. Le calcul de $\sum 10^k$ peut-être majorer sans formule en remarquant que cette somme représente un nombre s'écrivant avec $2n + 3$ « 1 » et terminé par un « 0 ». C'est le nombre $10^{2n+3} + \dots + 1000 + 100 + 10 = 111 \dots 1110$. Elle est donc $< 10^{2n+4} = 10000 \cdot 10^{2n} = O(1) \cdot 10^{2n}$.

12. En 2018, les *smartphones* les plus puissants (processeurs A12 d'Apple) affichaient une cadence de 2.4 GHz environ avec 5 000 milliards d'opérations/secondes, ce qui ramène le temps de calcul à moins de 3 jours.

$n = 9$ en un temps raisonnable, plus rapidement que 30 ans de calcul. Mais si on passe à $n = 10$, on multiplie par 100 le temps puisque $10^{2(n+1)} = 100 \cdot 10^{2n}$. Notez bien qu'en pratique, il n'y a pas de différence entre un programme de complexité trop grande et un programme qui boucle (et donc incorrect).

Si l'on pense que pour $n = 9$, l'instance du problème n'est jamais que 2 chiffres (un pour n et un pour c), la complexité exprimée en fonction de la taille de l'entrée est vraiment mauvaise. Mais c'est toujours ça, car pour TCHISLA on ne dispose d'aucun algorithme !

Parenthèse. On pourrait se demander si notre recherche exhaustive n'est pas trop « exhaustive », c'est-à-dire si on ne cherche pas la solution dans un ensemble démesurément trop grand.

Pour la réduire, on pourrait remarquer qu'on peut se passer de parenthèses, en utilisant la notation Polonaise inversée : pour les opérateurs binaire, on note les opérandes avant l'opérateur, comme factoriel dans le cas unaire. Par exemple : $(c+c*c)^c$ devient $ccc**+c^$. On gagne deux symboles : (et). Le terme exponentiel passe donc de 10^{2n} à 8^{2n} . Mais bon, même avec 8^{2n} , pour $n = 10$ on dépasse déjà 10^{18} . Et ce n'est pas non plus exactement le même problème puisque la sortie n'est plus vraiment une expression arithmétique standard. Et rien ne dit qu'en rajoutant les parenthèses cela correspond à la plus petite. Cependant l'astuce pourrait être utilisée pour la version originale TCHISLA puisqu'on ne se soucie que du symbole c dont le nombre reste identique dans les deux cas.

En fait il est possible de ne générer que les expressions arithmétiques valides (la partie grisée de la figure 1.4) au lieu d'utiliser un compteur. Pour cela il faut décrire les expressions par une grammaire et utiliser des outils de génération automatique qui peuvent alors produire en temps raisonnable chaque expression valide. Une sorte de fonction `Next()` améliorée donc.

La description d'une grammaire ressemblerait à quelque chose comme ceci¹³ :

$$\begin{array}{lcl} \lambda & \rightarrow & c \mid \lambda c \\ o & \rightarrow & + \mid - \mid * \mid / \mid ^ \\ E & \rightarrow & \lambda \mid (A)o(B) \mid (A)! \mid \sqrt(A) \\ A,B & \rightarrow & E \end{array}$$

Le problème est que, même si la fonction `Next()` ne générât que des expressions valides, le nombre d'expressions resterait très grand. Pour le voir, considérons l'expression $(c+\dots+c)/c$ de valeur n . Elle possède $2n+3$ symboles dont n opérateurs : $n-1$ additions et 1 division. Chacun des ces opérateurs peut être arbitrairement remplacé par $+, -, *, /, ^$ produisant à chaque fois une expression parfaitement valide. Les $n-1$ additions peuvent être remplacées aussi par le chiffre c, soit six symboles interchangeables. Chacune de ces expressions valides devra être évaluée a priori car la plus petite peut se trouver parmi elles. Il y a $n-1$ premiers symboles à choisir parmi six et le dernier parmi cinq. Ce qui fait déjà

13. Cela n'est pas parfait, car on génère des parenthèses inutilement comme les expressions $(c+c)+(c+c)$ ou $\sqrt(22)$.

$6^{n-1} \cdot 5$ expressions valides possibles dont au moins une s'évalue à n , sans compter les façons de mettre des paires de parenthèses, les opérateurs unaires¹⁴ ... Bref, le nombre d'expressions valides est intrinsèquement exponentiel en la taille de l'expression. Notez aussi que dès que $n \geq 24$, $6^{n-1} \cdot 5 > 10^{18}$... soit 30 ans de calcul.

Et si `tchisla2()` était efficace? Certes le nombre d'expressions valides est inexorablement exponentiellement en la taille de l'expression recherchée, mais rien ne dit que la recherche exhaustive ne va pas toujours s'arrêter sur une expression de taille très courte? L'analyse précédente est basée sur le fait que la taille de l'expression la plus courte ne peut pas dépasser $2n + 3$. La complexité est donc au plus exponentielle en $2n + 3$. Mais c'est peut-être beaucoup plus petit? Si c'est le cas, l'algorithme exhaustif `tchisla2()` pourrait finalement se révéler relativement efficace en pratique. En effet, l'analyse de la complexité ne change pas l'efficacité réelle du programme.

Essayons de trouver une expression de valeur n de taille la plus courte possible, indépendamment de c . Notons $k(n)$ cette taille et $e(n)$ une expression de valeur n correspondant à cette taille.

Bien sûr $k(n) \leq 2n + 3$ avec l'expression $e(n) = (c + \dots + c)/c$. Pour tenter d'en trouver une plus courte, et donc de trouver un meilleur majorant pour $k(n)$, on va décomposer n en une somme de puissance de deux. L'idée est que dans une telle somme, le nombre de termes est assez faible ainsi que les exposants. On va alors pouvoir ré-appliquer récursivement la construction aux exposants. Voici ce que cela donne pour quelques puissances de deux :

n	$k(n)$	$e(n)$
2^0	3	c/c
2^1	7	$(c+c)/c$
2^i	$12 + k(i)$	$((c+c)/c)^{e(i)}$

Prenons un exemple d'une telle décomposition. Par exemple $n = 5$:

$$\begin{aligned} 5 &= 2^2 + 2^0 = ((c+c)/c)^{e(2)} + c/c \\ &= ((c+c)/c)^{((c+c)/c)} + c/c \end{aligned}$$

Pour la taille de l'expression, il vient (il s'agit d'un majorant) :

$$\begin{aligned} k(5) &\leq k(2^2) + 1 + k(2^0) \\ &\leq 12 + k(2) + 1 + 3 \\ &\leq 16 + k(2) = 16 + 7 = 23 \end{aligned}$$

Bon, c'est pas terrible car on a vu que $k(5) \leq 2 \cdot 5 + 3 = 13$. Mais cela donne une formule de récurrence sur $k(n)$ qui va se révéler intéressante quand n est assez grand.

14. On peut remplacer par exemple $c+c$ par $+ \sqrt{c}$ ou $+c!$.

Voici un programme récursif calculant $k(n)$ en fonction des $k(2^i)$ et donc des $k(i)$ avec $i > 0$. En C, pour tester si n possède le terme 2^i dans sa décomposition, il suffit de faire un et-logique entre n et 2^i . Ce programme ne sert strictement à rien pour `tchisla2()`. Il peut servir en revanche à son analyse.

```

int k(int n){ // il faut n>0
    int i=0,p=1,s=0,t=0; // p=2^i, s=taille, t=#termes
    for(i=0;p<=n;i++,p*=2)
        if(n&p){ // teste le i-ème bit de n
            if(i==0) s+=3;
            else s+=min(12+k(i),2*p+3); // le meilleur des deux
            if(t>0) s++; // ajoute '+' s'il y a déjà un terme
            t++;
        }
    return s;
}

```

La valeur de `i` lorsque la boucle `for(i=...)` se termine correspond au nombre de bits dans l'écriture binaire de n . Notons ce nombre $L(n)$. Dans le paragraphe 1.6, on verra que $L(n) = \lceil \log_2(n+1) \rceil = O(\log n)$.

On peut alors donner un majorant sur la taille $k(n)$, car au pire sont présentes les $L(n)$ puissances de deux (sans oublier les '+') entre les $L(n)$ termes) :

$$\begin{aligned}
k(n) &\leq L(n)-1 + \sum_{i=0}^{L(n)-1} k(2^i) \leq L(n)-1 + k(2^0) + \sum_{i=1}^{L(n)-1} k(2^i) \\
&\leq L(n)-1 + 3 + \sum_{i=1}^{L(n)-1} (12 + k(i)) \leq L(n)+2 + \sum_{i=1}^{L(n)-1} (12 + 2i + 3) \\
&\leq L(n)+2 + 15 \cdot (L(n)-1) + 2 \sum_{i=1}^{L(n)-1} i \leq 16 \cdot L(n) - 13 + (L(n)-1) \cdot L(n) \\
&\leq L^2(n) + 15 \cdot L(n) - 13 = O(\log^2 n).
\end{aligned}$$

Il s'agit bien sûr d'un majorant grossier, puisqu'on n'utilise ni la récursivité ni le fait qu'on peut prendre le minimum avec $2n+3$. Pour $n=63$, ce majorant donne 113 alors que $2n+3=129$. C'est donc mieux. En utilisant le programme ci-dessus pour $k(63)$, on obtient 95. On a aussi, en utilisant le programme, que $k(n) < 2n+3$ dès que $n > 31$.

Nous avons précédemment vu que la complexité de `tchisla2()` était exponentielle en la taille k de l'expression recherchée. Bien sûr $k \leq k(n)$. La discussion ci-dessus montre donc que cette complexité est en fait plus petite, de l'ordre de :

$$10^{k(n)} = 10^{O(\log^2 n)} = n^{O(\log n)}.$$

En fait, l'exposant est plus petit que $O(\log n)$. Mais la décomposition en puissances de deux, et surtout l'analyse ci-dessus, ne permettent pas de conclure que la complexité est en $n^{O(1)}$.

Parenthèse. *En fait, une analyse plus fine de la récurrence découlant de la décomposition en puissance de deux permet de montrer que*

$$k(n) \leq O\left(\prod_{i=1}^{\log^* n} \log^{(i)} n\right) = o\left(\log n \cdot \log^2(\log n)\right)$$

où $\log^* n$ et $\log^{(i)} n$ sont des fonctions abordées page 135. En pratique $\log^* n \leq 5$ pour toute valeur de n aussi grande que le nombre de particules dans l'Univers.

On pourrait se demander si d'autres décompositions ne mèneraient pas à des expressions plus courtes encore, et donc à un meilleur majorant pour $k(n)$. On pourrait ainsi penser aux décompositions en carrés. Plutôt que de décomposer n en une somme de $\log n$ termes $2^{e(i)} = ((c+c)/c)^{e(i)}$, on pourrait décomposer en somme de $e(i)^2 = (e(j))^{e((c+c)/c)}$. Le théorème de Lagrange (1770) affirme que tout entier naturel est la somme d'au plus quatre carrés. (C'est même trois carrés sauf si n est de la forme $4^a \cdot (8b-7)$.) On tombe alors sur une récurrence du type :

$$k(n) \leq 4 \cdot k(\sqrt{n}) + O(1)$$

car chacune des quatre valeurs qui est mise au carré est bien évidemment au plus \sqrt{n} . La solution est alors $k(n) = O(4^{\log \log n})$ car en déroulant i fois l'équation on obtient $k(n) \leq 4^i \cdot k(n^{1/2^i}) + O(4^i)$. En posant $i = \log \log n$ il vient $k(n) = O(\log^2 n)$. C'est donc moins bien que pour les puissances de deux, ce qui n'est pas si surprenant.

En fait, il a été démontré en 2014 dans [GRS14, Théorème 1.6] que la plus courte expression de valeur n , en notation polonaise inversée (donc les parenthèses ne comptent pas) et utilisant seulement les symboles $1 + * ^$, était de taille au plus $t = 6 \log_2 n$. Ceci est démontré en considérant la décomposition de $n = \prod_i p_i^{\alpha_i}$ en facteurs premiers p_i , plus exactement décomposant $n = \prod_i (1 + (p_i - 1))^{\alpha_i}$ et en décomposant ainsi récursivement les α_i et les $p_i - 1$. En ajoutant les parenthèses (soit 4 caractères de plus par opérateur binaire) et en remplaçant le chiffre 1 par c/c (soit 2 caractères de plus par chiffres) on obtient une expression valide de valeur n et de taille au plus¹⁵ 4t démontrant que

$$k(n) \leq 24 \cdot \log_2 n .$$

Finalement, la fonction `tchisla2()` a une complexité de l'ordre de $10^{k(n)} = 10^{24 \log_2 n} = n^{24 \log_2(10)} \approx n^{80}$. On a progressé, mais pas tellement car pour $n = 9$ cela donne $10^{80} = 10^{62} \cdot 10^{18}$ soit 10 avec 62 zéros fois 30 ans ... La question sur l'efficacité de `tchisla2()` reste entière.

15. S'il y a b opérateurs binaires, il y a au plus $t - b$ symboles 1. Donc la transformation ajoute au plus $4b + 2(t - b) = 2t + 2b$ symboles. Or $b \leq t/2$, ce qui ajoute au plus $3t$ symboles.

1.4 Rappels sur la complexité

Il est important de bien distinguer deux concepts qui n'ont rien à voir.

- La complexité.
- Les notations asymptotiques.

Les notations telles que O, Ω, Θ sont de simples écritures mathématiques très utilisées en analyse qui servent à simplifier les grandeurs asymptotiques, comme les complexités justement mais pas que. Elles n'ont *a priori* strictement rien à voir avec la complexité, et d'ailleurs ont peut faire de la complexité sans ces notations. Par exemple, la page Wikipédia à propos de la [formule de Stirling](#) est remplie de notations asymptotiques comme

$$\ln(n!) = n \ln n - n + \frac{\ln n}{2} + O(1) \quad (1.3)$$

alors que $\ln(n!)$ n'a *a priori* rien à voir avec la complexité et les algorithmes. On rappellera les définitions de O, Ω, Θ dans la section [1.5](#).

La *complexité* est une mesure qui est appliquée à un algorithme ou un programme et qui s'exprime en fonction de la taille des entrées ou des paramètres.

Comme la taille est très souvent notée par un entier $n \in \mathbb{N}$, la complexité est la plupart du temps une fonction de n (et bien souvent croissante). En général on s'intéresse à mesurer une certaine ressource consommée par l'algorithme lorsqu'il s'exécute, comme le nombre d'instructions, l'espace mémoire, le nombre de comparaisons, etc. L'idée est de classer les algorithmes par rapport à cette complexité. Il y a donc plus plusieurs complexités. Les plus utilisées sont quand même la complexité en temps et en espace.

La *complexité en temps* est le nombre d'*opérations élémentaires* maximum exécutées par l'algorithme pour toute entrée (donc dans le pire des cas).

Le terme *temps* peut être un peu trompeur. On ne parle évidemment pas ici de seconde ou de minute, une complexité n'a pas d'unité. C'est un nombre de quelques choses. On parle de complexité en temps (et pas de complexité d'instructions) car on admet que chaque opération élémentaire s'exécute en temps unitaire, si bien que le temps d'exécution est effectivement donné par le nombre d'opérations élémentaires exécutées^{[16](#)}.

¹⁶. La réalité est un peu plus compliquée. Par exemple, le temps de lecture d'un mot mémoire peut dépendre du niveau de cache où il se trouve. Donc on considère que c'est le temps de l'opération élémentaire la plus lente qui s'exécute en temps borné (disons unitaire). Si bien que le temps est alors majoré par la complexité en temps dans le pire des cas.

La *complexité en espace* est le nombre de *mots mémoires* maximum utilisés durant l'exécution de l'algorithme pour toute entrée (donc dans le pire des cas).

Il faut en théorie se mettre d'accord sur ce qu'est une opération élémentaire et un mot mémoire. C'est le modèle de calcul. La plupart du temps un mot mémoire (ou registre) est une zone consécutive de mémoire comportant un nombre de bits suffisant pour au moins contenir un pointeur sur les données. Par exemple, si l'entrée d'un problème est une chaîne binaire S de taille n , alors les entiers i de $[0, n]$ pouvant représenter des indices de S pourront être stockés entièrement sur un mot mémoire, ce qui est bien pratique. Dans ce cas la taille des mots est au moins de $\lceil \log_2 n \rceil$ bits. [Question. Pourquoi?] Voir le paragraphe 1.6.

La taille d'une entrée (comme ici la chaîne binaire S) est exprimée en nombre de bits ou en nombre de mots (comme par exemple un tableau de n entiers de $[0, n]$). Pour résumer, une entrée de taille n doit pouvoir être stocker sur un espace mémoire de taille n , et donc comporter au plus n mots mémoires.

Les opérations élémentaires sont les opérations de lecture/écriture et de calcul simple sur les mots mémoires, parmi lesquelles les opérations arithmétiques sur les entiers de¹⁷ $[0, n]$. On considère aussi comme opération élémentaire l'accès à un mot mémoire dont l'adresse est stockée dans un autre mot mémoire. Dans notre exemple, $S[i]$ pourra être accédé (lu ou écrit) en temps unitaire. On parle de modèle RAM (*Random Access Memory*).

En gros, on décompose et on compte les opérations que la machine peut faire en temps unitaire (langage machine), et c'est tout.

1.4.1 Compter exactement?

Calculer la complexité d'un algorithme est une tâche souvent jugée difficile. Cela ne vient pas de la définition de la complexité, mais tout simplement de la nature des objets mesurés : les algorithmes.

Calculer le nombre d'opérations élémentaires exécutées est évidemment très difficile puisqu'il n'y a déjà pas de méthode systématique pour savoir si ce nombre est fini ou pas : c'est le problème de la HALTE qui est indécidable. En fait, un théorème (celui de Rice) établit que pour toute propriété non triviale¹⁸ définie sur un programme n'est pas décidable. Des exemples de propriété sont : « Est-ce que le programme termine par une erreur? » ou bien « Peut-on libérer un pointeur précédemment allouée? »

17. Les opérations arithmétiques sur des entiers plus grands, comme $[0, n^2]$, ne sont pas vraiment un problème. Elles prennent aussi un temps constant en simulant l'opération avec des couples d'entiers de $[0, n]$.

18. Une propriété triviale d'un programme P serait une propriété qui donnerait toujours la même réponse quelque soit le programme P , ou alors quelque soit l'entrée x . Par exemple, « Quelle est la longueur d'un programme? » est une propriété triviale car elle ne dépend pas de l'entrée.

En fait, il n'est même pas la peine d'aller voir des algorithmes très compliqués pour percevoir le problème. Considérons le programme suivant¹⁹ renvoyant le nombre d'étapes nécessaires pour atteindre la valeur 1 par la suite définie par

$$n \mapsto \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{sinon} \end{cases}$$

```
int Syracuse(int n){
    int k=0;
    while(n>1){
        n = (n&1)? 3*n+1 : n/2;
        k++;
    }
    return k;
}
```

Trouver la complexité en temps de `Syracuse(n)` fait l'objet de nombreuses recherches²⁰, voir aussi l'ouvrage figure 1.5. En fait, on ne sait pas si la boucle `while` s'arrête toujours au bout d'un moment, c'est-à-dire si sa complexité est finie ou pas, ou dit encore autrement, si la suite des valeurs de `n` finit toujours par atteindre 1.

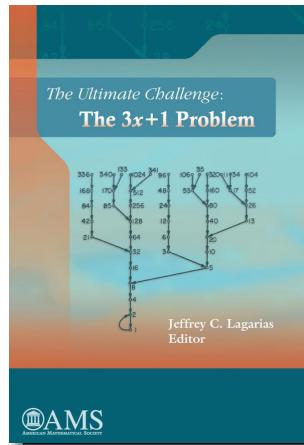


FIGURE 1.5 – Ouvrage consacré au problème « $3x + 1$ ».

En passant, si dans `Syracuse(n)` on remplace `3*n+1` par `a*n+b`, alors il est indécidable de savoir si, étant donnés les paramètres (a, b) , la fonction ainsi généralisée termine toujours. Encore une fois, pour certaines valeurs comme $(a=2, b=0)$, $(a=3, b=-1)$

19. On peut aussi faire récursif en une seule ligne :

int Syracuse(int n){ return (n>1)? 1+Syracuse((n&1)? 3*n+1 : n/2) : 0; }

20. https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

ou ($a=1, b=-1$) on sait répondre²¹. Mais aucun algorithme n'arrivera jamais à donner la réponse pour tout (a, b).

Le mathématicien Paul Erdős a dit à propos de ce problème (qu'on appelle aussi conjecture de Collatz, conjecture d'Ulam ou encore problème « $3x + 1$ » qui a été vérifiée²² pour tout $x < 5 \cdot 10^{18}$:

|| « *Les mathématiques ne sont pas encore prêtes pour de tels problèmes* ».

Beaucoup de problèmes très difficiles peuvent se formuler en simple problème de complexité et d'analyse d'algorithme, comme la conjecture de Goldbach (1742) qui dit :

|| « *Tout nombre entier pair supérieur à trois est la somme de deux nombres premiers.* ».

Face à ceci, il y a deux attitudes :

- Pessimiste : la complexité c'est compliquée ! c'est sans espoir.
- Optimiste : on peut espérer produire des algorithmes qui défis les mathématiques ! qui finalement marchent sans qu'on puisse dire et comprendre pourquoi.

Dans la pratique, on n'aura pas à traiter des cas si complexes, en tout cas cette année. Cependant, compter exactement le nombre d'opérations élémentaires ou de mots mémoires est souvent difficile en pratique même pour des algorithmes relativement simple.

La première difficulté est qu'on ne sait pas toujours quelles sont les opérations élémentaires qui se cachent derrière le langage, qu'il soit compilé ou interprété. Le compilateur peut cacher certaines opérations/optimisations, et l'interpréteur peut réaliser des tas d'opérations sans le dire (gestion de la mémoire²³ par exemple) ! Ensuite, le nombre d'opérations peut varier non seulement avec la taille de l'entrée, mais aussi avec l'entrée elle-même. Si l'on cherche un élément pair dans un tableau de taille n , le nombre d'opérations sera très probablement dépendant des valeurs du tableau. Certains langages aussi proposent de nombreuses instructions qui sont non élémentaires, comme certaines opérations de listes en Python.

Essayons de calculer la complexité dans l'exemple suivant :

21. Cela boucle trivialement pour ($a=2, b=0$) et $n=2$. Pour ($a=3, b=-1$) et $n=7$ la suite devient 7, 20, 10, 5, 14, 7, ... ce qui boucle donc aussi. Pour ($a=1, b=-1$) la suite ne fait que décroître, donc la fonction s'arrête toujours.

22. En juin 2018 une « preuve » non confirmée a été encore annoncée [Sch18].

23. On peut considérer que `malloc()` prend un temps constant, mais que `calloc()` et `realloc()` prennent un temps proportionnel à la taille mémoire demandée.

```
int T[n];
for(i=0; i<n; i++)
    T[i++] = 2*i-1;
```

Parenthèse. Il n'est pas vraiment pas conseillé d'utiliser une instruction comme $T[i] = 2*i-1$; La raison est qu'il n'est pas clair si la seconde occurrence de *i* (à droite du $=$) à la même valeur que la première. Sur certains compilateurs, comme `gcc` on aura $T[0]=-1$ car l'incrémentation de *i* à lieu après la fin de l'instruction (définie par le ; final). Pour d'autres²⁴, on aura $T[0]=1$. Ce qui est sûr est que l'option de compilation `-Wall` de `gcc` produit un warning²⁵. Mais est-ce bien sûr qu'on n'écrira pas en dehors des indices $[0, n[$?

Compter exactement le nombre d'opérations élémentaires n'est pas facile. Que se passe-t-il vraiment avec `int T[n]`? Combien y-a-t'il d'opérations dans la seule instruction $T[i] = 2*i-1$? Une incrémentation, une multiplication, une soustraction, une écriture, donc 4?²⁶ On fait aussi à chaque boucle une incrémentation, un saut et une comparaison (dans cet ordre d'ailleurs). Soit un total de 7 instructions par boucle. Et combien de fois boucle-t-on? $n/2$ ou plutôt $\lfloor n/2 \rfloor$? Donc cela fait $7 \cdot \lfloor n/2 \rfloor$ opérations plus le nombre d'instructions élémentaires pour `int T[n]` et $i=0$ (en espérant que le nombre d'instructions pour `int T[n]` ne dépende pas de *n*). Bref, même sur un exemple très simple, cela devient vite assez laborieux d'avoir un calcul exact.

En fait, peu importe le nombre exact d'opérations élémentaires. Avec un processeur 1 GHz, une opération de plus ou de moins ne fera jamais qu'une différence se mesurant en milliardième de secondes, soit le temps que met la lumière pour parcourir 30 cm.

Dans cet exemple, on aimerait surtout dire que la complexité de l'algorithme est linéaire en *n*. Car ce qui est important c'est que si *n* double, alors le temps doublera. Cela reste vrai que la complexité soit $7\lfloor n/2 \rfloor$ ou $4n - 1$. Et cela restera vrai, très certainement, quelque soit le compilateur ou le langage utilisé. Si la complexité était en n^4 , peu importe le coefficient devant n^4 , lorsque *n* double, le temps est multiplié par $2^4 = 16$. En plus, que peut-on dire vraiment du temps d'exécution puisque qu'un processeur cadencé à 2 GHz exécutera les opérations élémentaires deux fois plus vite qu'un processeur à 1 GHz.

Enfin, ce qui importe c'est la complexité *asymptotique*, c'est-à-dire lorsque la taille *n* de l'entrée est grande, tend vers l'infini.

En effet, quand *n* est petit, de toutes façons, peu importe l'algorithme, cela ira vite.

24. Comme celui en ligne https://www.tutorialspoint.com/compile_c_online.php

25. Message qui est : `warning: unsequenced modification and access to 'i'`

26. On calcule peut-être aussi l'adresse de `T+i` sauf si le compilateur s'aperçoit que `T` est une adresse constante. Dans ce cas il saura gérer un pointeur `p = T` qu'il incrémentera au fur et à mesure avec `p++`.

Ce qui compte c'est lorsque les données ont des tailles importantes. C'est surtout là qu'il faut réfléchir à l'algorithme, car tous ne se valent pas. Différents algorithmes résolvant le même problème sont alors comparés selon les valeurs asymptotiques de leur complexité. Ce n'est évidemment qu'un critère. Un autre critère, plus pratique, est celui de la facilité de l'implémenter correctement. Mais c'est une autre histoire.

1.4.2 Pour résumer

La complexité mesure généralement le nombre d'opérations élémentaires exécutées (complexité en temps) ou le nombre de mots mémoires utilisés (complexité en espace) par l'algorithme. Elle s'exprime en fonction de la taille de l'entrée. C'est n en général, mais pas toujours!. Dans la plupart des cas on ne peut pas exprimer la complexité exactement. On s'intéresse donc surtout à sa valeur asymptotique, c'est-à-dire lorsque n tend vers l'infini, car on souhaite éviter une réponse de Normand. Par exemple, à la question de savoir²⁷ :

« Lequel des algorithmes a la meilleure complexité entre $10n + 5$ et $n^2 - 7n$? »

Car en toute logique la réponse devrait être : « cela dépend de n ». Si $n < 18$, alors $n^2 - 7n < 10n + 5$. Sinon c'est le contraire. Le comportement de l'algorithme autour de $n = 18$ n'est finalement pas ce qui nous intéresse. C'est pour cela qu'on ne retient que le comportement asymptotique. Lorsque n devient grand,

$$\frac{10n + 5}{n^2 - 7n} \xrightarrow[n \rightarrow +\infty]{} 0.$$

Quand n devient grand, un algorithme de complexité en cn finit par gagner (complexité inférieure) sur celui de complexité $c'n^2$, peu importe les constantes c et c' , et peu importe les termes de second ordre. Cela se voit aussi sur les graphes des deux fonctions. Au bout d'un moment, l'une des deux courbes est au-dessus de l'autre, et pour toujours.

1.5 Notations asymptotiques

Les notations O, Ω, Θ servent à exprimer plus simplement les valeurs asymptotiques. Encore une fois, cela n'a rien à voir *a priori* avec la complexité. D'ailleurs, dans le chapitre suivant on l'utilisera pour parler de tout autre chose que la complexité. Il se trouve qu'en algorithmique on est particulièrement intéressé à exprimer des valeurs asymptotiques pour les complexités.

Soient f, g deux fonctions définies sur \mathbb{N} .

27. Peu importe qu'il s'agisse de temps, d'espace ou autre.

On dit que « $f(n)$ est en grand- O de $g(n)$ », et on le note $f(n) = O(g(n))$, si

$$\exists n_0 \in \mathbb{N}, \exists c > 0, \quad \forall n \geq n_0, \quad f(n) \leq c \cdot g(n)$$

Cela revient à dire qu'asymptotiquement et à une constante multiplicative près $f(n)$ est « au plus » $g(n)$.

On dit que « $f(n)$ est en $\Omega(g(n))$ », et on le note $f(n) = \Omega(g(n))$, ssi $g(n) = O(f(n))$. Ce qui revient à dire qu'asymptotiquement et à une constante multiplicative près que $f(n)$ est « au moins » $g(n)$. Enfin, on dit que « $f(n)$ est en $\Theta(g(n))$ », et on le note $f(n) = \Theta(g(n))$, ssi $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.

Il est très important de se souvenir que $f(n) = O(g(n))$ est une notation dont le but est de simplifier les énoncés. C'est juste pour éviter d'avoir à dire « lorsque n est suffisamment grand » et « à une constante multiplicative près ». Les notations servent à peu près à rien lorsqu'il s'agit de démontrer des formules précises. La définition ($\exists n_0 \in \mathbb{N}, \exists c > 0, \dots$) doit rester la priorité lorsqu'il s'agit de manipuler des asymptotiques.

1.5.1 Exemples et pièges à éviter

Souvent on étend la notation simple $f(n) = O(g(n))$ en la composant avec d'autres fonctions. Par exemple, lorsqu'on écrit $f(n) = 2^{O(n)}$ c'est pour dire qu'on peut remplacer l'exposant $O(n)$ par quelque chose $\leq cn$ pour n assez grand et pour une certaine constante $c > 0$. On veut donc exprimer le fait que $f(n) \leq 2^{cn}$ pour n assez grand et une certaine constante $c > 0$.

Si maintenant on écrit $f(n) = 1/O(n)$ c'est pour dire que le terme $O(n)$ est au plus cn pour une certaine constante $c > 0$ et pour n assez grand. Cela revient donc à dire que $f(n) \geq 1/(cn)$ ou encore que $f(n)$ est au moins $1/n$ à une constante multiplicative près. D'ailleurs cela montre que si $f(n) = 1/O(n)$ alors $f(n) = \Omega(1/n)$, notation plus claire qui est à privilégier dans ce cas.

De manière générale, lorsqu'on écrit $f(n) = h(O(g(n)))$ pour une certaine fonction h , c'est une façon d'écrire un asymptotique sur la composition $h^{-1} \circ f$, puisque $h^{-1}(f(n)) = O(g(n))$. Pour l'exemple précédent, si $f(n) = 1/O(n)$, alors $1/f(n) = O(n)$ ce qui signifie que $1/f(n) \leq cn$ et donc que $f(n) \geq 1/(cn)$ pour n assez grand et une certaine constante $c > 0$.

Il y a quelques pièges ou maladresses à éviter avec les notations asymptotiques.

- Il faut éviter d'utiliser la notation asymptotique les deux cotés d'une égalité, dans le membre de droite et le membre de gauche, comme par exemple $O(A) = \Omega(B)$. Car il y a alors confusion entre le « = » de l'équation et le « = » de la notation asymptotique. D'ailleurs dans certains ouvrages, surtout francophiles, on lit parfois $f(n) \in O(g(n))$. Si cela évite ce problème, cela n'évite pas les autres.

- Il faut éviter de composer plusieurs asymptotiques, comme par exemple, $f(n) = 1/\Omega(2^{O(n)})$. Ou encore de composer l'asymptotique $O()$ avec une fonction décroissante, comme $f(n) = O(n)^{-1/2}$. Dans ce cas il vaut mieux écrire $f(n) = \Omega(1/\sqrt{n})$, une expression plus facile à décoder. Notons en passant que $O(n)^{-1/2}$ n'est pas pareil que $O(n^{-1/2})$. Même chose pour $\Omega()$ qu'il faut éviter de composer avec une fonction décroissante, qui *in fine* change le sens des inégalités $\dots \leq cn$ ou $\dots \geq cn$ dans les définitions de $O()$ et $\Omega()$.
- Il faut éviter de manipuler les asymptotiques dans des formules de récurrences, comme on va le montrer dans l'exemple ci-après. En particulier, il faut éviter d'écrire $f(n) = O(1) + \dots + O(1) = O(1)$ car une somme de termes constants ne fait pas toujours une constante. Cela dépend du nombre de termes dans l'addition !

Pour illustrer un des derniers pièges de la notations grand- O , montrons tout d'abord la propriété suivante :

Si $a(n) = O(1)$ et $b(n) = O(1)$, alors $a(n) + b(n) = O(1)$.

En effet, $a(n) = O(1)$ implique qu'il existe c_a et n_a tels que $\forall n \geq n_a$, $a(n) \leq c_a$. De même, $b(n) = O(1)$ implique qu'il existe c_b et n_b tels que $\forall n \geq n_b$, $b(n) \leq c_b$. Donc si $a(n) = O(1)$ et $b(n) = O(1)$, alors $\forall n \geq \max\{n_a, n_b\}$, $a(n) + b(n) \leq c_a + c_b$. On a donc montré qu'il existe n_s et c_s tels que $\forall n \geq n_s$, $a(n) + b(n) \leq c_s$. Il suffit pour cela de prendre $n_s = \max\{n_a, n_b\}$ et $c_s = c_a + c_b$. Donc $a(n) + b(n) = O(1)$.

Considérons maintenant $f(n)$ la fonction définie par le programme suivant :

```
int f(int n){
    if(n<2)  return n;
    int a=f(n-1), b=f(n-2);
    return a + b;
}
```

En appliquant la propriété précédente, montrons par récurrence que $f(n) = O(1)$. Ça paraît clair : (1) lorsque $n < 2$, alors $f(n) < 2 = O(1)$; (2) si la propriété est vraie jusqu'à $n - 1$, alors on en déduit que $a = f(n - 1) = O(1)$ et $b = f(n - 2) = O(1)$ en appliquant l'hypothèse. On en déduit donc que $a + b = f(n) = O(1)$ selon la propriété précédemment démontrée.

Visiblement, il y a un problème, car $f(n) \approx 1.618^n$ est le n -ième nombre de Fibonacci, et $f(n)$ n'est certainement pas bornée par une constante. En fait, le même problème survient déjà avec un exemple plus simple encore : montrer par récurrence que la fonction f définie par $f(0) = 0$ et $f(n) = f(n - 1) + 1$ vérifie $f(n) = O(1)$.

[*Question. D'où vient l'erreur ? De la propriété, du point (1) ou du point (2) ?*]

1.5.2 Complexité d'un problème

Souvent on étend la notion de complexité d'un algorithme donné à celle d'un problème donné. On dit qu'un problème Π a une complexité $C(n)$ s'il existe un algorithme qui résout toutes les instances de Π de taille n avec une complexité $O(C(n))$ et que tout autre algorithme qui le résout a une complexité $\Omega(C(n))$. Dit plus simplement, la complexité du meilleur algorithme possible vaut $\Theta(C(n))$.

Par exemple, on dira que la complexité (en temps) du tri par comparaisons est de $n \log n$ ou est en $\Theta(n \log n)$. Le tri fusion atteint cette complexité et aucun algorithme de tri par comparaison ne peut faire mieux. Il existe des algorithmes de tri de complexité inférieure ... et qui, bien sûr, ne sont pas basé sur des comparaisons.

Parenthèse. Revenons sur la complexité des algorithmes de tri. Les algorithmes de tri par comparaisons nécessitent $\Omega(n \log n)$ comparaisons (dans le pire des cas). Ils ont donc une complexité en temps en $\Omega(n \log n)$.

En effet, le problème du tri d'un tableau non trié A à n éléments revient à déterminer la²⁸ permutation σ des indices de A de sorte que $A[\sigma(1)] < \dots < A[\sigma(n)]$. L'ensemble des permutations possibles est $N = n!$. Trouver l'unique permutation avec des choix binaires – les comparaisons – ne peut être plus rapide que celui de la recherche binaire dans un ensemble de taille N . Ce n'est donc rien d'autre que la hauteur²⁹ minimum d'un arbre binaire à N feuilles. Comme un arbre binaire de hauteur h contient au plus 2^h feuilles, il est clair qu'on doit avoir $2^h \geq N$, soit $h \geq \log_2 N$. Sous peine de ne pas pouvoir trouver la bonne permutation dans tous les cas, le nombre de comparaisons est donc au moins (cf. l'équation(1.3)) :

$$\log_2 N = \log_2(n!) = n \log_2 n - O(n).$$

Comme on l'a dit précédemment, il est possible d'aller plus vite en faisant des calculs sur les éléments plutôt que d'utiliser de simples comparaisons binaires. Il faut alors supposer que de tels calculs sur les éléments soient possibles en temps constant. Généralement, on fait la simple hypothèse que les éléments sont des clefs binaires qui tiennent chacune dans un mot mémoire. Trier n entiers pris dans l'ensemble $\{1, \dots, n^4\}$ entre dans cette catégorie. Les opérations typiquement autorisées sont des additions et des opérations logiques entre mots binaires (\vee, \wedge, \neg , etc.), les décalages binaires et parfois même la multiplication.

Le meilleur algorithme de cette catégorie, l'algorithme de Han [Han04], prend un temps de $O(n \log \log n)$ pour un espace en $O(n)$. Un précédent algorithme probabiliste, celui de Thorup [Tho97][Tho02] conçu sept ans plus tôt et qui n'utilise pas de multiplication, donnait les mêmes performances mais seulement en moyenne. Cela veut dire que l'algorithme trie correctement dans tous les cas mais en temps moyen $O(n \log \log n)$, cette moyenne dépendant des choix aléatoires de l'algorithme, pas de l'entrée. Les deux mêmes auteurs [HT02] ont ensuite produit un algorithme également probabiliste avec un temps et espace moyen en $O(n \sqrt{\log \log n})$ et $O(n)$ respectivement. C'est la meilleure complexité connue pour le tri de clefs binaires. On ne sait toujours pas s'il est possible ou non de trier en temps linéaire.

28. La permutation est unique si les éléments sont distincts.

29. Ici la hauteur est le nombre d'arêtes d'un chemin allant de la racine à une feuille quelconque.

1.5.3 Sur l'intérêt des problèmes de décision

Un problème de décision est un problème qui attend une réponse « oui » ou « non ». Le problème de la HALTE qu'on a vu au paragraphe 1.2 est un problème de décision, contrairement au problème TCHISLA dont la réponse n'est pas binaire mais une expression arithmétique. À première vue, un problème de décision est moins intéressant. Quel est donc l'intérêt des problèmes de décision tel que le suivant ?

CHEMIN HAMILTONIEN

Instance: Un graphe G .

Question: Est-ce que G possède un chemin Hamiltonien ? c'est-à-dire un chemin passant une et une seulement fois par chacun de ses sommets.

Ce problème est réputé difficile, mais en pratique on s'intéresse rarement au problème précis de savoir s'il existe ou pas un chemin passant par tous les sommets d'un graphe. Au mieux on aimeraient construire ce chemin plutôt que de savoir seulement qu'il existe. Alors, à quoi peut bien servir ce type de problème d'école ?

On pratique on s'intéresse plutôt, par exemple, au score maximum qu'on peut faire dans un jeu de plates-formes³⁰ (cf. figure 1.6) ou à un problème d'optimisation du gain que l'on peut obtenir avec certaines contraintes. N'oublions pas que le jeu n'est jamais qu'une simulation simplifiée de problèmes bien réels.

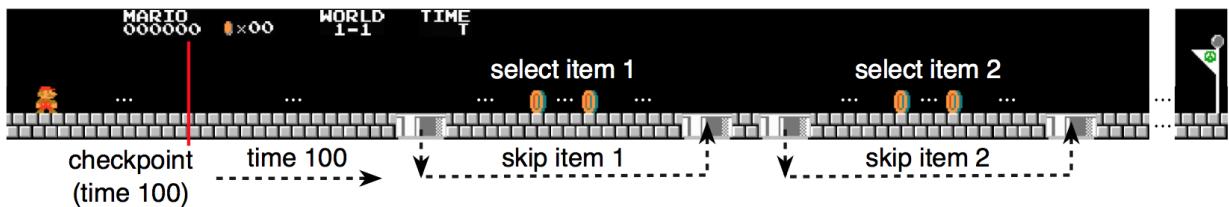


FIGURE 1.6 – Dans un jeu vidéo de type SUPER MARIO BROTHERS il s'agit souvent de choisir le bon chemin pour maximiser son score ou minimiser son temps de parcours. (Source [DVW16].)

Un autre exemple est le nombre maximum de cartes de UNO que l'on peut poser à la suite³¹ dans une poignée ... Ces problèmes reviennent à trouver le plus long chemin possible dans un graphe. Chaque sommet est l'une des positions possibles, et chaque arête représente un mouvement possible entre deux positions.

Il est alors vain de chercher un algorithme efficace général (ou une IA) pour ces problèmes, car si on le pouvait alors on pourrait déterminer si un graphe possède ou

30. Il existe des travaux théorique très sérieux et récent sur le jeu SUPER MARIO BROTHERS comme [DVW16].

31. On rappelle qu'au UNO on peut poser des cartes à la suite si elles sont de la même couleur ou de numéro consécutif.

pas un chemin hamiltonien simplement en répondant à la question : peut-on faire un score de n ?³²

Cela nous indique qu'il faut raffiner ou reformuler le problème, s'intéresser non pas au problème général du chemin le plus long, mais de trouver par exemple le chemin le plus long dans des graphes particuliers (comme des grilles) ou alors une approximation du plus long chemin.

Bien souvent, il arrive que le problème réel qu'on s'est posé contienne comme cas particulier un problème d'école. L'intérêt des problèmes de décisions qui sont réputés difficiles est alors de nous alarmer sur le fait qu'on est probablement parti dans une mauvaise voie pour espérer trouver un algorithme efficace.

1.6 Algorithme et logarithme

La fonction logarithme est omniprésente en algorithmique. Certes, `algorithme` et `logarithme` sont des anagrammes, mais ce n'est pas la seule raison ! Il y a des raisons plus profondes. Pour le comprendre, on va revenir sur les propriétés principales de cette fonction. Si ces propriétés ont déjà été vues en terminale, pour la suite du cours, elles doivent être maîtrisées³³.

Voici un petit exemple qui montre pourquoi cette fonction apparaît souvent en algorithmique. Considérons le code suivant³⁴ :

```
int f(int n){
    int p=1;
    while(n>1) n /= ++p;
    return n;
}
```

Peu importe ce que calcule précisément `f(n)`. Le fait est que ce genre de boucles simplissimes, ou ses variantes, apparaissent régulièrement en algorithmique, le bloc d'instructions à répéter pouvant souvent être plus complexe encore. Si l'on se pose la question de la complexité de cette fonction (ce qui revient ici à déterminer la valeur finale de `p` qui représente, à un près, le nombre de fois qu'est exécutée la boucle `while`)

32. Pour UNO, ce n'est pas aussi immédiat car le graphe d'une poignée de UNO n'est pas absolument quelconque comme il le pourrait pour un jeu de plates-formes. Il s'agit cependant du *line-graph* d'un graphe cubique. Or le problème reste NP-complet même pour ces graphes là (cf. [DDU⁺¹⁰]).

33. Le niveau en math exigé dans ce cours est celui de la terminale ou presque.

34. L'instruction `n /= ++p` signifie qu'on incrémente `p` avant d'effectuer la division entière `n/p`.

alors la réponse, qui ne saute pas aux yeux, est :

$$\Theta\left(\frac{\log n}{\log \log n}\right).$$

Trois « log » pour une boucle `while` contenant une seule instruction et les deux opérateurs `+1` et `/`. Il ne s'agit pas de retenir ce résultat, dont le sketch de preuve hors programme se trouve en notes de bas de page ³⁵, mais d'être conscient que la fonction logarithmique est plus souvent présente qu'il n'y paraît à partir du moment où l'on s'intéresse aux comportement des algorithmes.

En fait, en mathématique on a souvent à faire à la fonction $\ln x$ (ou sa fonction réciproque $\exp(x) = e^x$), alors qu'en algorithmique c'est le logarithme en base deux qui nous intéresse surtout.

Définition 1.1 *Le logarithme en base $b > 1$ d'un nombre $n > 0$, noté $\log_b n$, est la puissance à laquelle il faut éléver la base b pour obtenir n . Autrement dit $n = b^{\log_b n}$.*

Par exemple, le logarithme de mille en base dix est 3, car $1000 = 10^3$, et donc $\log_{10}(1000) = 3$. Plus généralement, et on va voir que cela n'est pas un hasard, le logarithme en base dix d'une puissance de dix est le nombre de zéros dans son écriture décimale. Bien évidemment $10^{\log_{10} n}$ fait ... n , c'est la définition.

La figure 1.7 montre l'allure générale des fonctions logarithmiques.

Il découle immédiatement de la définition 1.1 :

- La solution x de l'équation $n = b^x$ est $x = \log_b n$.
- C'est la réciproque ³⁶ de la fonction puissance de b .
- $\log_b(1) = 0$, $\log_b(b) = 1$, et $\log_b n > 0$ dès que $n > 1$.
- La fonction $\log_b n$ croît très lentement lorsque n grandit.

La croissance de la fonction logarithmique provient de la croissance de la fonction puissance de $b > 1$. On a $b^x < b^{x'}$ si et seulement si $x < x'$. Ensuite, cette croissance est lente dès que $n \geq b$. Pour que $x = \log_b n$ augmente de 1 par exemple, il faut multiplier n par b , car $b^{x+1} = b \cdot n$. On y reviendra.

Sauf mention contraire, on supposera toujours que $b > 1$ (cf. la définition 1.1), car l'équation $n = b^x$ n'a évidemment pas en générale de solution pour x lorsque $b = 1$. Et donc $\log_1 n$ n'est pas défini.

35. Il n'est pas très difficile de voir que l'entier p , après l'exécution du `while`, est le plus petit entier tel que $p! \geq n$. Il suit de cette remarque que $n > (p-1)!$. En utilisant le fait que $\ln(p!) = p \ln p - \Theta(p)$ (cf. l'équation (1.3)), on en déduit que $\ln n \sim p \ln p$. Il suit que $\ln \ln n \sim \ln p + \Theta(\ln \ln p)$ et donc que $p \sim \ln n / \ln p \sim \ln n / \ln \ln n$.

36. On dit aussi « fonction inverse » même si c'est potentiellement ambigu.

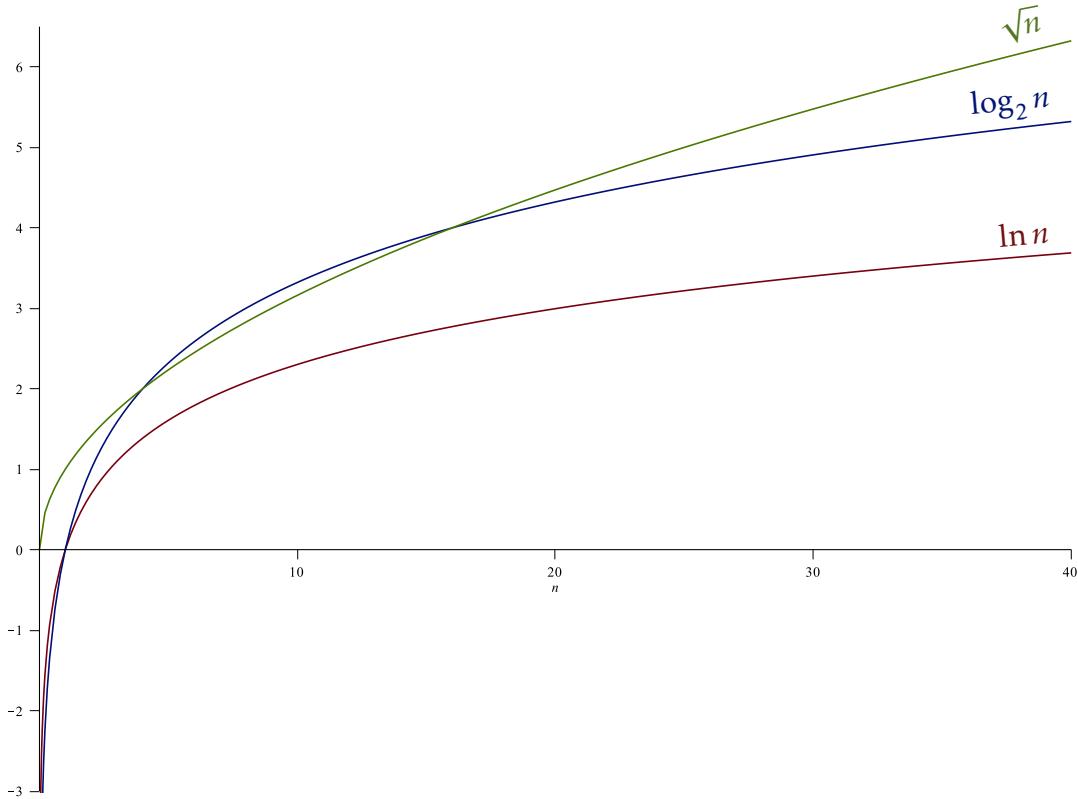


FIGURE 1.7 – Courbes des fonctions $\ln n$, $\log_2 n$, et \sqrt{n} . Bien que toutes croissantes, les fonctions logarithmiques ont un taux de croissance bien plus faible que la fonction : $1/n$ vs. $1/\sqrt{2n}$.

1.6.1 Propriétés importantes

La propriété vraiment importante qu'il faut retenir est :

Proposition 1.1 *Le nombre $\lceil \log_b n \rceil$ est le nombre de chiffres dans l'écriture de $n - 1$ en base b . C'est aussi le plus petit nombre de fois qu'il est nécessaire de diviser n par b pour obtenir un ou moins.*

Comme expliqué précédemment, il faut que $b > 1$.

Donc le logarithme de n est un peu la « longueur » de n . Par exemple :

- Pour $b = 10$ et $n = 10^3$. Alors $n - 1 = 999$ s'écrit sur 3 chiffres décimaux.
- Pour $b = 2$ et $n = 2^4$. Alors $n - 1 = 15 = 1111_{\text{deux}}$ s'écrit sur 4 chiffres binaires.
- Pour $b = 2$ et $n = 13$. Alors $n - 1 = 12 = 1100_{\text{deux}}$ s'écrit sur $\lceil \log_2(12) \rceil = \lceil 3.5849... \rceil = 4$ chiffres.

Comme on peut facilement « éplucher » les chiffres d'un nombre n écrit en base b en répétant l'opération $n \mapsto \lfloor n/b \rfloor$, grâce à l'instruction `n /= b` qui en C effectue la division

euclidienne, on déduit de la première partie de la proposition 1.1 le code suivant pour calculer $\lceil \log_b n \rceil$:

```
int log_b(int n, int b){    // n>0 et b>1
    n--; int k=1;           // écrit n-1, au moins un chiffre
    while(n>=b) n /= b, k++; // tant qu'il y a plus d'un chiffre
    return k;                // k = #chiffres de n-1 en base b
}
```

Attention! Contrairement à ce que semble affirmer la proposition 1.1, une boucle comme

```
while(n>1) n /= b;
```

n'est pas répétée $k = \lceil \log_b n \rceil$ fois avant de sortir avec $n \leq 1$. La raison est que cette boucle n'effectue pas l'opération $n \mapsto n/b^k$, mais plutôt

$$n \mapsto \underbrace{\lfloor \dots \lfloor \lfloor n/b \rfloor / b \rfloor \dots / b \rfloor}_{k \text{ fois}}.$$

Évidemment, quand n est une puissance entière de b , c'est bien la même chose. On peut montrer³⁷ que, malgré les nombreuses parties entières, le résultat n'est pas très loin de n/b^k . C'est en fait égal à 1 près.

Pour démontrer la proposition 1.1, on va se servir de la propriété qui est elle aussi très importante à connaître car elle revient (très) souvent : la somme des $n+1$ premiers termes d'une suite géométrique $(q^i)_{i \in \mathbb{N}}$ de raison³⁸ $q \neq 1$:

$$1 + q + q^2 + \dots + q^n = \sum_{i=0}^n q^i = \frac{q^{n+1} - 1}{q - 1}. \quad (1.4)$$

La forme plus générale

$$q^m + q^{m+1} + \dots + q^n = \sum_{i=m}^n q^i = q^m \sum_{i=0}^{n-m} q^i = q^m \cdot \frac{q^{n-m+1} - 1}{q - 1}$$

s'obtient trivialement depuis d'équation (1.4) en factorisant par le premier terme « q^m ». On retient dans cette dernière formule que « q^m » est le premier terme et « $n - m + 1$ » le

37. Cf. le fait 6.1 du chapitre 6 du cours d'[Algorithmique distribuée](#).

38. La formule ne marche pas si $q = 1$. Bien évidemment, dans ce cas $\sum_{i=0}^n 1^i = n + 1$.

nombre de termes de la somme. Notons d'ailleurs que la formule (1.4) est triviale³⁹ à démontrer par récurrence, le problème étant de se soutenir de la formule à démontrer.

Par exemple $1 + 2 + 4 + 8 + \dots + 2^h = (2^{h+1} - 1)/(2 - 1) = 2^{h+1} - 1 = 111\dots 1_{\text{deux}}$ donne le nombre de sommets dans un arbre binaire complet de hauteur h . On a aussi $1 + 10 + 100 + \dots + 10^h = (10^{h+1} - 1)/9 = 999\dots 9_{\text{dix}}/9 = 111\dots 1_{\text{dix}}$. Notons que dans ces deux exemples, le résultat est un nombre qui s'écrit avec h chiffres identiques qui sont des uns.

Preuve de la proposition 1.1. Le nombre k de divisions par b à partir de n nécessaire pour avoir 1 ou moins est le plus petit entier k tel que $n/b^k \leq 1$. Par la croissance de la fonction logarithme,

$$\frac{n}{b^k} \leq 1 \Leftrightarrow n \leq b^k \Leftrightarrow \log_b n \leq k.$$

Le plus petit entier k vérifiant $k \geq \log_b n$ est précisément $k = \lceil \log_b n \rceil$ qui est donc l'entier recherché.

Montrons maintenant que $\lceil \log_b n \rceil$ est aussi le nombre de chiffres pour écrire $n - 1$ en base b . On va d'abord montrer que pour tout entier $k \geq 1$, le nombre $b^k - 1$ s'écrit avec k chiffres⁴⁰. D'après la formule (1.4) avec $q = b$ et $n = k - 1$, on a :

$$\begin{aligned} b^k - 1 &= (b - 1) \cdot (1 + b + b^2 + \dots + b^{k-1}) \\ &= \boxed{b - 1} b^{k-1} + \boxed{b - 1} b^{k-2} + \dots + \boxed{b - 1} b^2 + \boxed{b - 1} b + \boxed{b - 1} \end{aligned}$$

Cette dernière somme comprend k termes de la forme $(b - 1) \cdot b^i$ qui représentent précisément les k chiffres de l'entier $b^k - 1$ écrit en base b . Chacun de ces chiffres vaut $b - 1$ ce qui montre que $b^k - 1$ est le plus grand nombre qui s'écrit en base b avec k chiffres. Il suit que b^k s'écrit avec $k + 1$ chiffres.

Soit k l'entier tel que $b^{k-1} < n \leq b^k$. On vient de voir que $b^k - 1$ s'écrit avec k chiffres. Donc $n - 1 \leq b^k - 1$ s'écrit avec au plus k chiffres. Mais on a vu aussi que b^k s'écrit avec $k + 1$ chiffres. Donc $n - 1 \geq b^{k-1}$ s'écrit avec au moins $(k - 1) + 1 = k$ chiffres. Il suit que $n - 1$ s'écrit avec exactement k chiffres. Par la croissance du logarithme, $b^{k-1} < n \leq b^k \Leftrightarrow k - 1 < \log_b n \leq k$, ce qui revient à dire que $k = \lceil \log_b n \rceil$. Cela termine la preuve de la proposition 1.1. \square

Notons que la proposition 1.1 est encore une autre façon de se convaincre que la fonction $\log_b n$ croît très lentement. En effet, $\log_{10}(100) = 2$ et $\log_{10}(1\ 000\ 000\ 000) = 9$ seulement.

La fonction logarithme possède d'autres propriétés importantes découlant de la définition 1.1, comme celles-ci :

39. C'est lié au fait que $[(q^{n+1} - 1)/(q - 1)] + q^{n+1} = (q^{n+1} - 1 + q \cdot q^{n+1} - q^{n+1})/(q - 1) = (q^{n+2} - 1)/(q - 1)$.

40. Un exemple qui ne démontre rien : $10^3 - 1 = 999$ s'écrit sur 3 chiffres décimaux.

Proposition 1.2

- $\log_b n = (\log_a n)/(\log_a b)$ pour toute base $a > 1$.
- $\log_b(x \cdot y) = \log_b x + \log_b y$ pour tout $x, y > 0$.
- $\log_b(n^\alpha) = \alpha \log_b n$ pour tout $\alpha \geq 0$.
- $\log_b n \ll n^\alpha$ pour toute constante $\alpha > 0$.

Pour le premier point. Calculons le nombre $b^{\log_a n / \log_a b}$ en remplaçant la première occurrence de b par $b = a^{\log_a b}$, par définition de $\log_a b$. Il vient :

$$b^{\log_a n / \log_a b} = (a^{\log_a b})^{\log_a n / \log_a b} = a^{(\log_a b)(\log_a n) / \log_a b} = a^{\log_a n} = n.$$

On a donc à la fois $b^{\log_a n / \log_a b} = n$ et $n = b^{\log_b n}$, c'est donc que $\log_a n / \log_a b = \log_b n$.

Le premier point a pour conséquence que lorsque b est une constante devant n (c'est-à-dire $b = O(1)$), les fonctions $\log_b n$, $\log_2 n$, $\log_{10} n$ ou même comme on va le voir $\ln n$, sont toutes équivalentes à une constante multiplicative près. On a par exemple $\log_2 n = \log_{10} n / \log_{10} 2 \approx 3.32 \log_{10} n$ et $\ln n = \log_2 n / \log_2 e \approx 0.69 \log_2 n$.

Dans les notations asymptotiques faisant intervenir des logarithmes, on ne précise pas la base (si celle-ci est une constante). On note donc simplement $O(\log n)$ au lieu de $O(\log_2 n)$, $O(\log_{10} n)$ ou encore $O(\log_\pi n)$.

De la même manière, on n'écrit jamais quelque chose du type $O(2n+1)$ ou $O(3 \log n)$, l'objectif de la notation asymptotique étant de simplifier les expressions. La remarque s'applique aussi aux notations $\Omega()$ et $\Theta()$.

Pour le deuxième point. D'une part $x \cdot y = b^{\log_b(x \cdot y)}$ par définition de $\log_b(x \cdot y)$. D'autre part

$$b^{\log_b x + \log_b y} = b^{\log_b x} \cdot b^{\log_b y} = x \cdot y = b^{\log_b(x \cdot y)}$$

et donc $\log_b x + \log_b y = \log_b(x \cdot y)$.

Pour le troisième point. Il peut se déduire directement du précédent seulement si α est un entier. L'argument est cependant similaire aux précédents. D'une part $n^\alpha = b^{\log_b(n^\alpha)}$ par définition de $\log_b(n^\alpha)$. D'autre part

$$n^\alpha = (b^{\log_b n})^\alpha = b^{(\log_b n)\alpha} = b^{\alpha \log_b n}$$

et donc $b^{\log_b(n^\alpha)} = b^{\alpha \log_b n}$. On a donc que $\log_b(n^\alpha) = \alpha \log_b n$.

Pour le quatrième point. On note « $f(n) \ll g(n)$ » pour dire que $f(n)/g(n) \rightarrow 0$ lorsque $n \rightarrow +\infty$. C'est pour dire que $f(n)$ est significativement plus petite que $g(n)$. En math, on le note aussi $f(n) = o(g(n))$. En utilisant la croissance de la fonction logarithme et le changement de variable $N = \log_b \log_b n$:

$$\begin{aligned} \log_b n &\ll n^\alpha \Leftrightarrow \\ \log_b \log_b n &\ll \log_b(n^\alpha) = \alpha \log_b n \Leftrightarrow \\ \log_b \log_b \log_b n &\ll \log_b(\alpha \log_b n) \Leftrightarrow \\ \log_b \log_b \log_b n &\ll \log_b \alpha + \log_b \log_b n \Leftrightarrow \\ \log_b N &\ll N + \log_b \alpha \end{aligned}$$

Comme b et α sont des constantes, $\log_b \alpha = O(1)$ est aussi une constante (éventuellement négative si $\alpha < 1$). Lorsque n devient grand, N devient grand aussi. Clairement $\log_b N \ll N - O(1)$, la « longueur » de N (cf. la proposition 1.1) étant significativement plus petite que N quand N est grand. D'où $\log_b n \ll n^\alpha$. Notez que ce point implique par exemple que $\log_2 n \ll \sqrt{n}$ ($\alpha = 0.5$).

1.6.2 Et la fonction $\ln n$?

Une base b particulière procure à la fonction logarithme quelques propriétés remarquables. Il s'agit de la base e , la constante due à Euler :

$$e = 1 + \frac{1}{1} + \frac{1}{1 \times 2} + \frac{1}{1 \times 2 \times 3} + \frac{1}{1 \times 2 \times 3 \times 4} \dots = \sum_{i \geq 0} \frac{1}{i!} = 2.718281828459\dots$$

On a aussi

$$\left(1 + \frac{1}{n}\right)^n \xrightarrow[n \rightarrow +\infty]{} e.$$

On note $\boxed{\log_e n = \ln n}$ et on l'appelle le *logarithme naturel* ou encore *logarithme népérien* de n (de Néper). La réciproque de $\ln n$ est donc e^n , la fonction exponentielle classique.

Le premier point de la proposition 1.2 permet de montrer, en posant comme deuxième base $a = e$, que :

$$\log_b n = \frac{\ln n}{\ln b}.$$

Il se trouve que la fonction $\ln n$ est la seule fonction définie pour $n > 0$ qui s'annule en zéro et dont la dérivée vaut $1/x$. Autrement dit, on a :

$$\ln n = \int_1^n \frac{1}{x} dx$$

ce qui s'interprète comme la surface sous la courbe $1/x$ pour $x \in [1, n]$. Voir la figure 1.8 ci-après.

Une des propriétés intéressantes est que la somme des inverses des n premiers entiers, notée H_n et appelée série Harmonique, est presque égale à $\ln n$ (ce qui se comprend aisément d'après la figure 1.8) :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \sum_{i=1}^n \frac{1}{i} = \ln n + O(1).$$

En particulier $\ln n \sim H_n$. On peut être même plus précis avec l'asymptotique $H_n = \ln n + \gamma + O(1/n)$ où $\gamma = 0.577 215 664\dots$ est la constante d'Euler-Mascheroni⁴¹. On observe encore une fois que la fonction $\ln n$ croît lentement, puisque $H_{n+1} = H_n + 1/(n+1)$ et donc $\ln(n+1) \approx (\ln n) + 1/(n+1)$.

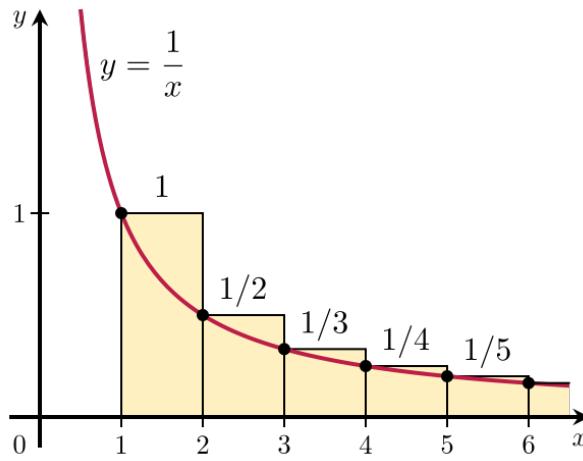


FIGURE 1.8 – Approximation de $\ln n$ par H_n (source Wikipédia). On voit par exemple que $H_5 - 1 < \int_1^5 \frac{1}{x} dx < H_4$. En fait H_4 s'interprète comme la somme des colonnes colorées 1, 2, 3, 4 et $H_5 - 1$ comme celles de colonnes 2, 3, 4, 5. Par concavité de $1/x$, on peut majorer chaque colonne par le milieu entre la colonne et sa suivante, ce qui donne un encadrement encore plus serré $H_5 - 1 < \int_1^5 \frac{1}{x} dx < \frac{1}{2} \sum_{i=1}^4 \left(\frac{1}{i} + \frac{1}{i+1}\right) = \frac{1}{2}(H_4 + H_5 - 1) = \frac{1}{2}(H_5 - \frac{1}{5} + H_5 - 1) = H_5 - \frac{1}{2}(\frac{1}{5} + 1) = H_5 - 1 + \frac{2}{5}$. Et donc $(\frac{1}{2} + \dots + \frac{1}{5}) < \ln 5 < (\frac{1}{2} + \dots + \frac{1}{5}) + \frac{2}{5}$ soit l'encadrement $1.283 < \ln 5 < 1.683$. En fait, $\ln 5 = 1.609\dots$. De manière générale on en déduit l'encadrement de taille < 0.5 qui est $H_n - 1 < \ln n < H_n - 1 + \frac{n-1}{2n}$ pour tout entier $n > 1$.

1.7 Morale

- En informatique les problèmes sont définis par la relation entre les entrées et les sorties (décrivées généralement sous la forme d'une question). L'ensemble des

⁴¹. On connaît très peu de chose sur cette constante. On ne sait pas par exemple si c'est un nombre rationnel ou pas.

entrées possibles s'appellent les instances du problème.

- Les algorithmes résolvent des problèmes, tant dit que programmes en donnent une implémentation. Une instance particulière peut être résolue sans qu'un algorithme existe pour le problème. Par exemple, le problème de la HALTE n'a pas d'algorithme. Pourtant, on connaît la réponse pour beaucoup d'entre eux.
- Quand on programme en C un algorithme qui résout un problème, les entrées et sorties du problème (en fait leurs types) sont partiellement capturés par le prototype d'une fonction qui implémente l'algorithme, comme `double puissance(double, int)`.
- Pour certains problèmes on peut essayer de trouver une formule close liant les paramètres d'entrées aux sorties, comme par exemple la formule liant les coefficients a, b, c d'un polynôme de degré deux et ses deux racines. On peut aussi tenter la recherche exhaustive (ou *brute-force*), une technique qui consiste à essayer tous les résultats possibles.
- Mais pour la plupart des problèmes intéressants il n'existe pas de telles formules, et pour certains on ne peut même pas envisager de recherche exhaustive, car, par exemple, l'ensemble de tous les résultats envisageables n'est pas de taille bornée (par une fonction de la taille de l'entrée⁴²). Pour certains problèmes il n'existe carrément pas d'algorithmes de résolution. Et ce n'est pas parce qu'on ne les a pas trouvés. C'est parce qu'on peut démontrer qu'il n'existe pas de tels algorithmes. Inutile alors d'essayer de trouver une fonction C, un programme ou une IA les résolvant. Ces problèmes là sont indécidables, comme par exemple le problème de la HALTE.
- Les problèmes de décisions très simples ne servent pas forcément à résoudre des problèmes pratiques (les problèmes pratiques sont souvent bien plus complexes). Ils servent en revanche à montrer que le problème réel qui nous intéresse est bien trop difficile, car il contient un problème d'école (de décision) réputé difficile comme cas particulier.
- La complexité est une mesure qui sert à comparer les algorithmes entre eux. Elle permet d'écartier rapidement un algorithme qui serait, quelle que soit son implémentation, une perte de temps car de complexité en temps ou en espace trop importante. Ainsi, un nombre d'instructions en 10^{18} pour un processeur 1 GHz est éliminatoire (>30 ans). En pratique, un programme de complexité en temps trop importante aura le même comportement qu'un programme qui boucle (et donc erroné).
- L'analyse de complexité (notamment sa qualité) ne change en rien l'efficacité d'un programme. Cette analyse, si elle est fine, sert à comprendre où et comment est

42. Par exemple, le problème de savoir si l'on peut dessiner un graphe sur le plan sans croisement d'arête ne se prête pas *a priori* à une recherche exhaustive car le nombre de dessins possibles n'est pas de taille bornée : \mathbb{R}^2 c'est grand ! même pour un graphe à n sommets. Il n'empêche, il existe des algorithmes linéaires pour le résoudre.

utilisée la ressource mesurée (temps ou espace). Il y des programmes qui marchent et on ne sait pas pourquoi.

- La complexité s'exprime toujours en fonction de la taille des entrées (généralement n). C'est un nombre qui n'a pas d'unité. C'est un nombre d'opérations élémentaires (=temps) ou de mots mémoires (=espace). Certaines instructions de certains langages, comme `printf()` ou `memcpy()` de la librairie standard `C`, ne sont pas élémentaires.
- Il n'y a pas de notation dédiée à la complexité en temps ou en espace. Les notations O, Ω, Θ n'ont pas de rapport direct avec la complexité. Ce sont des notations pour alléger les expressions mathématiques portant sur des valeurs asymptotiques. Elles évitent d'écrire $\exists n_0 \in \mathbb{N}, \forall c > 0, \dots$
- Il est difficile de calculer la complexité de manière exacte. On utilise plutôt des ordres de grandeurs et on l'évalue lorsque la taille n est « suffisamment grande » ($n \rightarrow +\infty$). On utilise alors souvent les notations asymptotiques pour simplifier l'écriture, notamment O, Ω, Θ . Ces notations sont parfois sources de pièges qu'il est bon de connaître.
- Le logarithme en base b de n ($=\log_b n$) est une fonction à connaître, surtout lorsque $b = 2$, car elle est omniprésente en algorithmique.

Bibliographie

- [DDU⁺10] E. D. DEMAINE, M. L. DEMAINE, R. UEHARA, T. UNO, AND Y. UNO, *UNO is hard, even for a single player*, in 5th International Conference Fun with Algorithms (FUN), vol. 6099 of Lecture Notes in Computer Science, Springer, June 2010, pp. 133–144. doi : [10.1007/978-3-642-13122-6_15](https://doi.org/10.1007/978-3-642-13122-6_15).
- [DVW16] E. D. DEMAINE, G. VIGLIETTA, AND A. WILLIAMS, *Super mario bros. is harder/easier than we thought*, in 8th International Conference Fun with Algorithms (FUN), vol. 49, LIPIcs, June 2016, pp. 13:1–13:14. doi : [10.4230/LIPIcs.FUN.2016.13](https://doi.org/10.4230/LIPIcs.FUN.2016.13).
- [GRS14] E. K. GNANG, M. RADZIWIELŁ, AND C. SANNA, *Counting arithmetic formulas*, European Journal of Combinatorics, 47 (2014), pp. 40–53. doi : [10.1016/j.ejc.2015.01.007](https://doi.org/10.1016/j.ejc.2015.01.007).
- [Han04] Y. HAN, *Deterministic sorting in $O(n \log \log n)$ time and linear space*, Journal of Algorithms, 50 (2004), pp. 96–10. doi : [10.1016/j.jalgor.2003.09.001](https://doi.org/10.1016/j.jalgor.2003.09.001). Also appears in STOC '02.
- [HT02] Y. HAN AND M. THORUP, *Integer sorting in $O(n \sqrt{\log \log n})$ expected time and linear space*, in 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, November 2002, pp. 135–144. doi : [10.1109/SFCS.2002.1181890](https://doi.org/10.1109/SFCS.2002.1181890).

- [Sch18] P. SCHORER, *A solution to the $3x + 1$ problem*, June 2018.
- [Tan15] I. J. TANEJA, *Single digit representations of natural numbers*, RGMIA Research Report Collection, 18 (2015), pp. 1–55.
- [Tho97] M. THORUP, *Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations*, in 8th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 1997, pp. 352–359.
- [Tho02] M. THORUP, *Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations*, Journal of Algorithms, 42 (2002), pp. 205–230. doi : [10.1006/jagm.2002.1211](https://doi.org/10.1006/jagm.2002.1211).

Sommaire

2.1	Le problème	39
2.2	Formule asymptotique	40
2.3	Réurrence	42
2.4	Programmation dynamique	48
2.5	Mémorisation paresseuse	50
2.6	Morale	55
	Bibliographie	56

Mots clés et notions abordées dans ce chapitre :

- nombre de partitions
- formule asymptotique
- programmation dynamique
- mémorisation paresseuse (mémoïsation)

2.1 Le problème

On s'intéresse à toutes les façons de partitionner un ensemble d'éléments indistinctables. Par exemple, il y a cinq façons de partager un ensemble de 4 billes :

- $\boxed{\text{oooo}}$: 1 paquet de 4 billes ;
- $\boxed{\text{ooo}} \boxed{\text{o}}$: 1 paquet de 3 billes et 1 paquet d'1 bille ;
- $\boxed{\text{oo}} \boxed{\text{oo}}$: 2 paquets de 2 billes ;
- $\boxed{\text{oo}} \boxed{\text{o}} \boxed{\text{o}}$: 1 paquet d'2 billes et 2 paquets d'1 bille ;
- $\boxed{\text{o}} \boxed{\text{o}} \boxed{\text{o}} \boxed{\text{o}}$: 4 paquets d'1 bille.

On représente plutôt les partitions d'un ensemble de n éléments comme les différentes façons d'écrire l'entier n (la cardinalité de l'ensemble) comme somme d'entiers non nuls (la cardinalité des parts). On parle de partition d'un entier.

Pour notre exemple précédent, les cinq partages possibles d'un ensemble de 4 billes reviennent à écrire :

$$\begin{aligned} 4 &= 4 \\ &= 3 + 1 \\ &= 2 + 2 \\ &= 2 + 1 + 1 \\ &= 1 + 1 + 1 + 1 \end{aligned}$$

Les éléments étant indistinguables, la somme $1 + 3$ représente la même partition que $3 + 1$. Par habitude on écrit les sommes par ordre décroissant des parts.

Pour simplifier un peu le problème, on va se contenter de compter le nombre de partitions d'un entier n , et on notera $p(n)$ ce nombre. Les premières valeurs sont :

n	1	2	3	4	5	6	7	8	9	10	...	50	...	100
$p(n)$	1	2	3	5	7	11	15	22	30	42	...	204 226	...	190 569 292

2.2 Formule asymptotique

Le n -ième nombre de Fibonacci, $F(n)$, possède une formule close (la formule de Binet). Il est bien connu que :

$$F(n) = \frac{\Phi^n - (1 - \Phi)^n}{\sqrt{5}} = \left\lfloor \frac{\Phi^n}{\sqrt{5}} \right\rfloor \quad \text{avec} \quad \Phi = \frac{1 + \sqrt{5}}{2} \approx 1.6180 \quad (2.1)$$

On note $\lfloor x \rfloor = \lfloor x + 0.5 \rfloor$ l'entier de plus proche de x , c'est-à-dire l'arrondi. La formule avec l'arrondi vient du fait ¹ que le terme $|(1 - \Phi)^n / \sqrt{5}| < 1/2$. Ces formules ne sont pas nécessairement très efficaces telles quelles car en pratique les calculs faisant intervenir les irrationnels (comme le nombre d'Or Φ) sont difficiles à représenter exactement en machine. Et donc les calculs sont vites entachés d'erreurs. Il n'empêche, une formule close est un bon point de départ pour la recherche d'un algorithme efficace.

Le nombre de partitions est très étudié en théorie des nombres. Par exemple, il a été montré en 2013 que $p(120\,052\,058)$, qui possède 12 198 chiffres, était premier. Il est aussi connu que, pour tout $x \in]0, 1[$:

$$\sum_{n=0}^{+\infty} p(n) \cdot x^n = \prod_{k=1}^{+\infty} \left(\frac{1}{1 - x^k} \right)$$

1. En fait, $(1 - \Phi)^n / \sqrt{5}$ vaut approximativement $+0.44, -0.27, +0.17, -0.10, \dots$ pour $n = 0, 1, 2, 3, \dots$ ce qui tend assez rapidement vers 0.

ce qui n'est malheureusement pas une formule close, à cause des sommes et produits infinis. De plus il faut choisir correctement x . Et puis c'est pas vraiment la somme infinie $\sum_{n=0}^{+\infty} p(n)$ qui nous intéresse ...

Il n'y a pas de formule close connue pour $p(n)$. Donc, contrairement à $F(n)$, il n'y aucun espoir de pouvoir calculer $p(n)$ à l'aide d'un nombre constant d'opérations arithmétiques. Il existe seulement des formules asymptotiques.

Hardy et Ramanujan [HR18] ont donné en 1918 l'asymptotique suivant :

$$p(n) \sim \frac{1}{4n\sqrt{3}} \cdot \exp\left(\pi\sqrt{2n/3}\right) \approx 2^{3.7\sqrt{n}}.$$

Parenthèse. L'erreur n'est que de 1.4% pour $n = 1000$, les trois premiers chiffres de $p(n)$ étant correctes à quelques unités près. Bien sûr, par définition de l'asymptotique, cette erreur diminue plus n augmente. Siegel a donné une autre formule asymptotique, publiée par Knopp [Kno81], plus complexe mais qui converge plus efficacement vers la vraie valeur que celle de Hardy-Ramanujan :

$$p(n) \sim \frac{2\sqrt{3}}{24n-1} \cdot \left(1 - \frac{6}{\pi\sqrt{24n-1}}\right) \cdot \exp\left(\frac{\pi}{6}\sqrt{24n-1}\right)$$

Intuitivement on retrouve la formule de Hardy-Ramanujan car le terme dans l'exponentielle $\frac{\pi}{6}\sqrt{24n-1}$ tends vers $\pi\sqrt{2n/3}$ quand n augmente. De même le coefficient du premier terme en $1/n$ tends vers $2\sqrt{3}/24 = 2\sqrt{3} \cdot \sqrt{3}/(4 \cdot 6 \cdot \sqrt{3}) = 1/(4\sqrt{3})$.

On dit que « $f(n)$ est asymptotiquement équivalent à $g(n)$ », et on note $f(n) \sim g(n)$, si

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 1$$

Lorsque $f(n) \sim g(n)$ cela ne signifie pas que l'écart *absolu* entre $f(n)$ et $g(n)$ est de plus en plus petit quand n devient grand, mais que l'écart *relatif* tend vers 0. L'écart absolu est la quantité $|f(n) - g(n)|$ alors que l'écart relative est $(f(n) - g(n))/g(n) = f(n)/g(n) - 1$.

Comme on l'a déjà dit, une fonction peut avoir plusieurs asymptotiques. Par exemple $n^2 + n + 1 \sim n^2 + n$, mais on a aussi $n^2 + n + 1 \sim n^2$. On pourrait dire ici que le premier asymptotique $n^2 + n$ converge plus efficacement vers $n^2 + n + 1$ que le deuxième en n^2 . En effet, en comparant les écarts absolus on a

$$1 = (n^2 + n + 1) - (n^2 + n) \ll (n^2 + n + 1) - n^2 = n + 1.$$

Notez qu'ici on compare des écarts qui sont des fonctions de n . En toute généralité, il n'y a pas de raison, comme dans cet exemple, d'en avoir un qui est toujours mieux que l'autre. Il pourrait se passer qu'un est meilleur jusqu'à un n_0 et qu'ensuite cela s'inverse, voir que cela oscille.

L'idée de la notion d'asymptotique est de ne retenir que le terme principal, le plus grand lorsque $n \rightarrow +\infty$, afin de simplifier l'expression. On peut montrer que si $f(n) \sim g(n)$ alors $f(n) = \Theta(g(n))$. Le contraire est faux, si l'on considère par exemple les fonctions n^2 et $2n^2$.

Une notion que l'on va croiser, et qui est reliée à celle d'asymptotique, est celle-ci.

On dit que qui énonce que « $f(n)$ est en petit- o de $g(n)$ », et on le note $f(n) = o(g(n))$, si

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$$

On peut vérifier que $f(n) \sim g(n)$ si et seulement si $f(n) = g(n) + o(g(n))$.

Comme le montrent les exemples précédents, une fonction comme $p(n)$ peut avoir plusieurs équivalents asymptotiques. En fait, pour ce chapitre, peu importe la finesse des asymptotiques sur $p(n)$. On retiendra surtout que $p(n)$ est exponentielle en \sqrt{n} , ce qui peut s'écrire :

$$p(n) = 2^{\Theta(\sqrt{n})}.$$

Parenthèse. Mais pourquoi peut-on écrire que $p(n) = 2^{\Theta(\sqrt{n})}$? Tout d'abord (et par définition), parce que $p(n) = 2^{O(\sqrt{n})}$ et $p(n) = 2^{\Omega(\sqrt{n})}$. Ensuite, pour toute constante c

$$e^{c\sqrt{n}} = (2^{\log_2 e})^{c\sqrt{n}} = 2^{(c \log_2 e)\sqrt{n}} = 2^{c'\sqrt{n}}$$

avec $c' = c \log_2 e$. (Voir le paragraphe 1.6.) Donc on a

$$e^{\Theta(\sqrt{n})} = 2^{\Theta(\sqrt{n})}.$$

On a également pour toutes constantes a, b, c

$$a \cdot n^b \cdot e^{c\sqrt{n}} = e^{\ln a} \cdot e^{b \ln n} \cdot e^{c\sqrt{n}} = e^{\ln a + b \ln n + c\sqrt{n}} = e^{\Theta(\sqrt{n})}$$

En combinant l'asymptotique $p(n) \sim a \cdot n^b e^{c\sqrt{n}}$ de la formule d'Hardy-Ramanujan, avec les constantes $a = 1/(4\sqrt{3})$, $b = -1$, et $c = \pi\sqrt{4/3}$, on déduit que $p(n) \sim 2^{\Theta(\sqrt{n})}$.

2.3 Récurrence

Une manière graphique de représenter une partition de n est d'utiliser une sorte de tableau où l'on entasse, à partir du coin inférieur gauche, n petits carrés en colonnes de hauteur décroissante. On appelle un tel tableau un diagramme de Ferrers.

Par exemple, sur la figure 2.1 la partition $12 = 5 + 3 + 2 + 1 + 1$ peut être représentée par le diagramme (a) et l'autre partition $12 = 3 + 3 + 2 + 2 + 2$ par le diagramme (b).

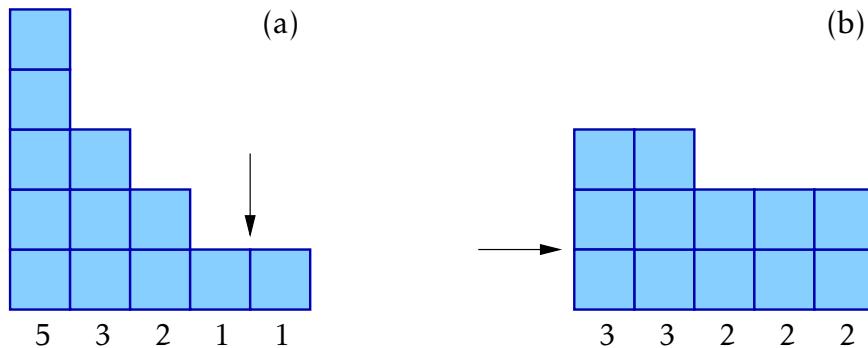


FIGURE 2.1 – Deux partitions de 12, chacune de 5 parts, représentées par des diagrammes de Ferrers. La partition (a) est de type 1, (b) de type 2.

Chaque colonne représente une part de la partition. Le nombre de parts est le nombre de colonnes. Notons que chaque partition est uniquement représentée par un diagramme (c'est lié au fait que les colonnes sont triées par hauteur). Inversement, chaque diagramme comportant n carrés organisés en colonnes décroissantes représente une seule partition de n . Donc compter le nombre de partitions revient à compter le nombre de tels diagrammes.

Parenthèse. La représentation en diagramme permet facilement de se convaincre que $p(n)$ est au moins exponentiellement en \sqrt{n} . Pourquoi ? On part d'un diagramme (k, k) où pour tout i , la colonne i est de hauteur i . Ce diagramme possède $n_0 = k(k+1)/2 \sim \frac{1}{2}k^2$ carrés répartis en k colonnes. Maintenant, en haut de chacune des k colonnes, on peut décider d'ajouter ou pas un carré. Cela crée à chaque fois un diagramme valide et différent. On construit ainsi 2^k diagrammes tous différents. Parmi eux beaucoup ont été obtenus en ajoutant exactement $\lfloor k/2 \rfloor$ carrés : $\binom{k}{\lfloor k/2 \rfloor}$ pour être précis. Ces diagrammes possèdent tous $n = n_0 + \lfloor k/2 \rfloor \sim \frac{1}{2}k^2$ carrés. Comme $n \sim \frac{1}{2}k^2$, on a que $k \sim \sqrt{2n}$. Le nombre de diagrammes à n carrés ainsi construits est donc

$$\binom{k}{\lfloor k/2 \rfloor} \sim 2^{k-o(k)} = 2^{\sqrt{2n}-o(\sqrt{n})}.$$

Autrement dit $p(n) = 2^{\Omega(\sqrt{n})}$.

Les diagrammes se décomposent facilement en éléments plus petits ce qui facilite leur comptage. Par exemple, si l'on coupe un diagramme de Ferrers entre deux colonnes, on obtient deux diagrammes de Ferrers. De même si on le coupe entre deux lignes (voir les flèches de la figure 2.1).

Les récurrences sont plus faciles à établir si l'on fixe le nombre de parts des partitions, c'est-à-dire le nombre de colonnes des diagrammes. Dans la suite, on notera $p(n, k)$ le nombre de partitions de n en k parts. Évidemment, le nombre parts k varie entre 1 et n , d'où

$$p(n) = p(n, 1) + \cdots + p(n, n) = \sum_{k=1}^n p(n, k).$$

Parmi les 5 partitions de $n = 4$, on a déjà vu qu'il n'y en a exactement deux avec deux parts : $4 = 2 + 2 = 3 + 1$. D'où $p(4, 2) = 2$. On peut vérifier qu'il a 13 diagrammes de Ferrers avec 12 carrés et 5 colonnes, d'où $p(12, 5) = 13$.

On peut classifier les partitions de n en k parts, qu'on nommera diagrammes (n, k) , en deux types : celles dont la plus petite part est 1 (type 1), et celles dont la plus petite part est au moins 2 (type 2). Le diagramme (a) est de type 1, et (b) de type 2. Évidemment, ces catégories sont disjointes : un diagramme est soit de type 1 soit de type 2. On peut donc compter séparément les diagrammes de chaque type et faire la somme :

$$p(n, k) = p_1(n, k) + p_2(n, k).$$

C'est une technique classique pour compter des objets : on les décompose en plus petit morceaux et/ou on les classe en catégories plus simples à compter ou à décomposer.

Supposons (par récurrence !) qu'on a réussi à construire tous les diagrammes « plus petits » que (n, k) , c'est à dire tous les diagrammes (n', k') avec $n' \leq n$ et $k' \leq k$. Et bien sûr $(n', k') \neq (n, k)$, un des deux paramètres doit être strictement plus petit.

Construire tous les diagrammes de type 1, à partir des diagrammes plus petits, est facile car on peut toujours les couper juste avant la dernière colonne. On obtient alors un diagramme $(n - 1, k - 1)$ avec un carré et une colonne de moins. Inversement, si à un diagramme $(n - 1, k - 1)$ on ajoute une colonne de hauteur un, on obtient un diagramme (n, k) de type 1. Par conséquent, il y a autant de diagrammes (n, k) de type 1 que de diagrammes $(n - 1, k - 1)$. Dit autrement, $p_1(n, k) = p(n - 1, k - 1)$.

On peut construire les diagrammes de type 2 à l'aide de diagrammes plus petits en les coupant juste au dessus de la première ligne. On obtient alors un diagramme avec k carrés de moins mais encore k colonnes puisque toutes les colonnes étaient initialement de hauteur au moins deux. On obtient donc un diagramme $(n - k, k)$. L'inverse est aussi vrai : à partir d'un diagramme $(n - k, k)$ on peut construire un diagramme (n, k) de type 2 en le surélevant d'une ligne de k carrés. Par conséquent, il y a autant de diagrammes (n, k) de type 2 que de diagrammes $(n - k, k)$. Dit autrement, $p_2(n, k) = p(n - k, k)$.

En sommant les diagrammes (n, k) de type 1 et de type 2, on a donc montré la relation de récurrence :

$$p(n, k) = p(n - 1, k - 1) + p(n - k, k)$$

valable pour tous les entiers tels que $1 < k < n$ (à cause de « $k - 1$ » et de « $n - k$ » dans le terme de droite qui doivent être > 0). Si $k = 1$ ou $k = n$, alors $p(n, k) = 1$ [[Question. Pourquoi ?](#)], et bien sûr $p(n, k) = 0$ si $k > n$ [[Question. Pourquoi ?](#)].

De cette récurrence, on en déduit immédiatement l'algorithme et le programme suivant :

```

int p(int n,int k){
    if(k>n) return 0;
    if(k==1 || k==n) return 1;
    return p(n-1,k-1) + p(n-k,k);
}

int partition(int n){
    int k,s=0;
    for(k=1;k<=n;k++) s += p(n,k);
    return s;
}

```

Parenthèse. Le programme termine bien car, même si chacun des paramètres ne diminuent pas toujours strictement, la somme des paramètres elle, diminue strictement. De manière générale, pour montrer qu'un programme récursif termine bien, il suffit d'exhiber une fonction de potentielle dépendant des paramètres d'appels, qui est bornée inférieurement et qui décroît strictement au cours des appels. On parle de bel ordre.

Arbre des appels.

L'arbre des appels d'une fonction est un arbre dont les noeuds représentent les paramètres d'appels et les fils les différents appels (éventuellement récursifs et/ou composés²) lancés par la fonction. L'exécution de la fonction correspond à un parcours en profondeur de l'arbre depuis sa racine qui représente les paramètres du premier appel.

Voici un exemple (cf. figure 2.2) de l'arbre des appels pour $p(6,3)$. Par rapport à la définition ci-dessus, on s'est permis d'ajouter aux noeuds l'opération (ici $+$) ainsi que les valeurs terminales aux feuilles (ici 0 ou 1), c'est-à-dire les valeurs renvoyées lorsqu'il n'y a plus d'appels récursifs. Les valeurs terminales ne font pas partie des noeuds de l'arbre des appels.

Les valeurs terminales permettent, à l'aide de l'opération attachés aux noeuds, de calculer progressivement à partir des feuilles la valeur de retour de chaque noeuds et donc de la valeur finale à la racine. L'évaluation de $p(6,3)$ produit en fait un parcours de l'arbre des appels. Lorsque qu'un noeud interne a été complètement évalué, sa valeur

2. Un appel composé est un appel correspondant à la composition de fonctions, comme dans l'expression $f(g(n))$ ou encore $f(f(n/2)*f(n/3))$ (dans ce dernier cas c'est un appel composé et récursif avec trois fils). Il peut arriver que l'arbre des appels ne puisse pas être construit à l'avance, mais seulement lors de l'exécution.

(de retour) est renvoyée à son parent qui poursuit le calcul. *In fine* la racine renvoie la valeur finale au programme appelant.

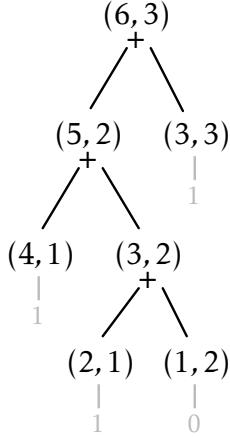


FIGURE 2.2 – Arbre des appels pour $p(6,3)$. Il comporte 7 nœuds dont 4 feuilles, chacune ayant une valeur terminale (0 ou 1 en gris). Le nombre de valeurs terminales à 1 est bien sûr de $3 = p(6,3)$.

L’arbre des appels pour $\text{partition}(6)$ est composé d’une racine avec (6) connectée aux fils $(6,1), (6,2), \dots, (6,6)$ étant eux-mêmes des arbres d’appels (cf. figure 2.3).

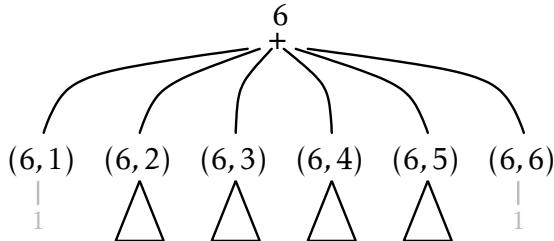


FIGURE 2.3 – Arbre des appels pour $\text{partition}(6)$. Pour obtenir l’arbre complet il faudrait développer les quatre sous-arbres comme sur la figure 2.2.

Complexité en temps. Calculons la complexité en temps de $\text{partition}(n)$. La première chose à dire est que cette complexité est proportionnelle aux nombres d’appels à la fonction $p(n,k)$. Ce nombre est aussi le nombre de nœuds dans l’arbre des appels de $\text{partition}(n)$, qui vaut un plus la somme des nœuds des arbres pour $p(n,1), p(n,2), \dots, p(n,n)$.

Le nombre d’appels exacts n’est pas facile à calculer, mais cela n’est pas grave car c’est la complexité qui nous intéresse. Donc une valeur asymptotique ou a une constante multiplicative près fera très bien l’affaire. Intuitivement ce nombre d’appels n’est pas loin de $p(n,k)$, puisque c’est ce que renvoie la fonction en calculant des sommes de

valeurs qui sont uniquement 0 ou 1, les deux cas terminaux de `p(n,k)`. En fait $p(n,k)$ est précisément le nombre de feuilles de valeur 1.

L'arbre étant binaire, le nombre de nœuds recherchés est 2 fois le nombre de feuilles moins un. Or chaque feuille renvoie 0 ou 1. De plus il y a jamais de feuilles sœurs renvoyant toutes les deux 0. C'est du au fait qu'une feuille gauche ne peut jamais renvoyer 0. En effet, il faudrait avoir $n - 1 < k - 1$ (fils gauche $(n - 1, k - 1)$ renvoyant 0) ce qui ne peut arriver car son père (n, k) aurait été une feuille ! (car si $n - 1 < k - 1$ c'est que $n < k$). Donc le nombre de feuilles est au plus deux fois le nombre de valeurs 1.

Ainsi, le nombre de feuilles de l'arbre est au plus $2p(n,k)$, et le nombre de nœuds au plus $2 \cdot (2p(n,k)) - 1 < 4p(n,k)$. En utilisant la formule asymptotique sur $p(n)$, on déduit que la complexité en temps de `partition(n)` est donc au plus proportionnelle à

$$\sum_{k=1}^n 4 \cdot p(n,k) = \Theta(p(n)) = 2^{\Theta(\sqrt{n})}.$$

Opérations arithmétiques sur de grands entiers. En fait on a supposé que l'opérations arithmétiques (ici `+`) sur les nombres $p(n,k)$ était élémentaires. Ce n'est vrai que si les nombres sont des entiers pas trop grands, s'ils tiennent sur un mot mémoire (`int` ou `long`). En fait, ce n'est pas le cas ici car au bout d'un moment les nombres sommés deviennent très grands. En toute rigueur, il faudrait alors effectuer les opérations de somme sur des tableaux de chiffres, et remplacer l'opération `S=A+B` par une fonction ressemblant à `Sum(int A[], int B[], int S[], int length)`.

D'après la formule asymptotique, des tableaux de `length` = $O(\sqrt{n})$ chiffres suffisent. [Question. Pourquoi?] Il faudrait donc multiplier la complexité en temps vue précédemment par ce facteur $O(\sqrt{n})$ puisque la somme de deux tableaux de taille `length` se fait trivialement en temps $O(\text{length})$. Notons toutefois que cela ne change pas grand chose car $O(\sqrt{n}) \cdot 2^{\Theta(\sqrt{n})} = 2^{\Theta(\sqrt{n})}$ [Question. Pourquoi?].

Complexité exponentielle? Par rapport à la taille de l'entrée du problème du calcul de partition de n , la complexité est en fait doublement exponentielle (2^{2^x} pour un certain x). Pourquoi? Il s'agit de calculer le nombre $p(n)$ en fonction de l'entrée n . L'entrée est donc un entier sur $k = \lceil \log_{10} n \rceil = \Theta(\log n)$ chiffres. Donc, une complexité en temps de $2^{\Theta(\sqrt{n})}$, en fonction de k est en fait une complexité en temps de $2^{2^{\Theta(k)}}$ car $\sqrt{n} = 2^{(\log n)/2} = 2^{\Theta(k)}$ et $\Theta(2^{\Theta(k)}) = 2^{\Theta(k)}$. La complexité en temps de la version récursive est donc doublement exponentielle!

Calculs inutiles. En fait, on passe son temps à calculer des termes déjà calculés. Pour le voir, il faut repérer des appels (ou nœuds) identiques dans l'arbre des appels. Cependant, dans l'arbre des appels de `p(6,3)` il n'y a aucune répétition !

Pour voir qu'il y a quand même des calculs inutiles, il faut prendre un exemple plus grand que précédemment. En fait, avec un peu de recul, il est clair qu'il doit y avoir des appels identiques pour $\text{partition}(n)$ et même pour $p(n, k)$. La raison est que le nombre de nœuds différents n'est jamais que n^2 , car il s'agit de couples (n', k') avec $n', k' \in \{1, \dots, n\}$. Or on a vu que l'arbre possédait $\Theta(p(n))$ nœuds ce qui est asymptotiquement bien plus grand que n^2 . Donc les mêmes nœuds apparaissent plusieurs fois, nécessairement. Il y a même un nœud qui doit apparaître $\Omega(p(n)/n^2) = \Omega(2^{\Theta(\sqrt{n})-\log_2(n^2)}) = 2^{\Theta(\sqrt{n})}$ fois !

La figure 2.4 montre qu'à partir de n'importe quel nœud (n, k) on aboutit à la répétition des nœuds $(n - 2k, k - 2)$, à condition toutefois que $n - 2k$ et $k - 2$ soient > 0 . Évidemment ce motif se répète à chaque nœud si bien que le nombre de calculs dupliqués devient rapidement très important.

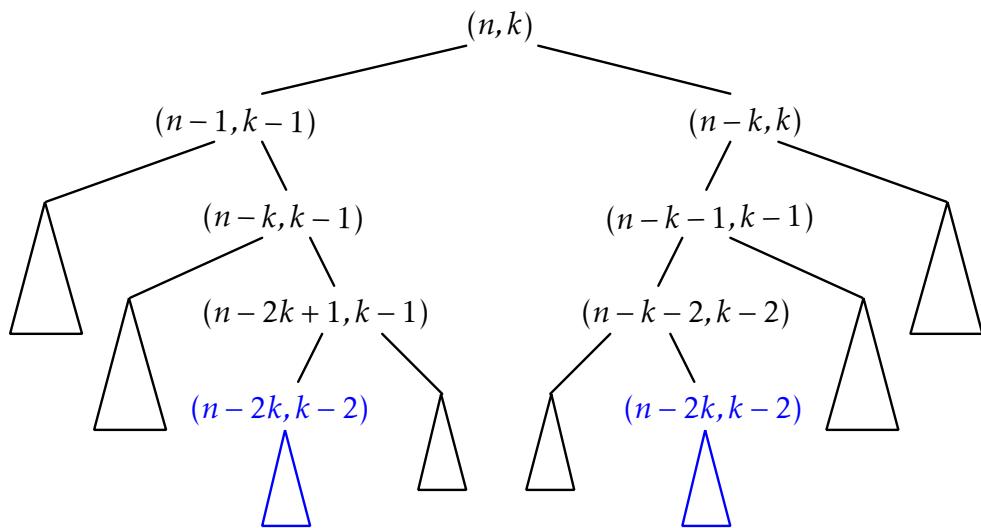


FIGURE 2.4 – Arbre d'appels pour le calcul de $p(n, k)$. En notant G/D les branches gauche/droite issues d'un nœud, on remarque que les branches GDDG et DGDD, si elles existent, mènent toujours aux mêmes appels.

2.4 Programmation dynamique

La programmation dynamique est l'implémentation améliorée de la version récursive d'un algorithme. Au lieu de faire des appels récursifs, on utilise la *mémorisation* qui économise ainsi des calculs (et du temps) au détriment de l'espace mémoire. On utilise une table où les valeurs sont ainsi calculées « dynamiquement » en fonction des précédentes.

On va donc utiliser une table $P[n][k]$ similaire à la table 2.1 que l'on va remplir progressivement grâce à la formule de récurrence. Pour simplifier, dans la table P on

$p(n, k)$	k								total $p(n)$
n	1	2	3	4	5	6	7	8	
1	1								1
2	1	1							2
3	1	1	1						3
4	1	2	1	1					5
5	1	2	2	1	1				7
6	1	3	3	2	1	1			11
7	1	3	4	3	2	1	1		15
8	1	4	5	5	3	2	1	1	22

TABLE 2.1 – Le calcul de la ligne $p(n, \cdot)$ se fait à partir des lignes précédentes. Ici $p(8, 3) = p(7, 2) + p(5, 3)$.

n'utilisera pas l'indice 0.

```

int partition(int n){
    int P[n+1][n+1], i, k, s=0; // indices 0,1,...,n
    for(i=1; i<=n; i++){ // pour chaque ligne
        P[i][1]=P[i][i]=1;
        for(k=2; k<i; k++) // pour chaque colonne
            P[i][k] = P[i-1][k-1] + P[i-k][k];
    }
    for(k=1; k<=n; k++) s += P[n][k];
    return s;
}

```

La complexité en temps est $O(n^2)$... si n n'est pas trop grand.

Plus rapide encore. Il existe d'autres formules de récurrence donnant des calculs encore plus performants. Par exemple,

$$\begin{aligned}
p(n) &= (p(n-1) + p(n-2)) - \\
&\quad (p(n-5) + p(n-7)) + \\
&\quad (p(n-12) + p(n-15)) - \\
&\quad (p(n-22) + p(n-26)) + \\
&\quad \dots
\end{aligned}$$

De manière plus synthétique la formule de récurrence s'exprime comme :

$$p(n) = \begin{cases} 1 & \text{si } n = 1 \\ \sum_{i \geq 1} (-1)^{i-1} \cdot (p(n - i \cdot (3i \pm 1)/2)) & \text{si } n > 1 \end{cases}$$

Il faut bien sûr que l'argument $n - i \cdot (3i \pm 1)/2 \geq 1$ puisque $p(n)$ n'est défini que pour $n \geq 1$. On en déduit alors que la somme comprend seulement $2\sqrt{2n/3}$ termes environ. C'est le degré maximum de l'arbre des appels. [Exercice. Quelle serait la complexité de la fonction récursive résultant de cette formule ? En utilisant la programmation dynamique et donc une table, quelle serait alors sa complexité ?]

[Exercice. Considérons la fonction `k(n)` décrite page 7. Montrez qu'il y a des calculs inutiles. Proposez solution de programmation dynamique.]

2.5 Mémorisation paresseuse

Dans une fonction récursive, on peut toujours éviter les calculs redondant en utilisant de la mémoire supplémentaire. C'est le principe de la programmation dynamique à l'aide d'une table auxiliaire comme vu précédemment, mais cette dernière impose de parcourir judicieusement la table. Il faut donc réfléchir un peu plus, et le programme résultant est souvent assez différent de la fonction originale. (Pour s'en convaincre comparer la fonction `p(n)` récursive page 45 et `partition(n)` itérative page 49.) Modifier abondamment un code qui marche est évidemment une source non négligeable d'erreurs.

Dans cette partie on va donc envisager de faire de la programmation dynamique mais sans trop réfléchir à comment remplir la table.

Principe. On stocke au fur et à mesure le résultat de chaque appel (ainsi que l'appel lui-même) sans trop se soucier de l'ordre dans lequel ils se produisent. Et si un appel avec les mêmes paramètres réapparaît, alors on extrait de la table sa valeur sans refaire de calculs.

Ainsi, en laissant la fonction gérer ses appels dans l'ordre d'origine, on modifiera au minimum le code d'origine tout en espérant un gain en temps.

L'idée est donc de modifier le moins possible la fonction d'origine en utilisant une mémorisation avec le moins d'efforts possibles. Si l'arbre des appels est « suffisamment » redondant (de nombreux appels sont identiques), alors cette méthode de mémorisation « paresseuse » aboutira à un gain en temps certain. On appelle parfois cette technique la *mémoïsation* d'une fonction.

Attention ! Pour que cette méthode fonctionne il est important que la valeur de la fonction ne dépende que des paramètres de l'appel. Il ne doit pas y avoir d'effets de bords via une variable extérieure (globale) à la fonction par exemple. Généralement, on ne peut pas appliquer la technique de mémoïsation à une fonction déjà mémoisée.

Exemple avec une simple table (1D). Pour commencer, voici une illustration de ce principe pour le calcul des nombres de Fibonacci³. La version d'origine est `fibo()`. Pour construire la version avec mémoïsation, `fibo_mem()`, on utilise une table `F[]` qui restera dans la mémoire `static` à travers les différents appels récursifs. Avant le calcul récursif, on teste simplement si la valeur souhaitée est déjà dans la table `F[]` ou non.

```
long fibo(int n){ // version d'origine
    if(n<2) return n; // fibo(0)=0, fibo(1)=1
    return fibo(n-1)+fibo(n-2);
}

long fibo_mem(int n){ // version mémoisée
    static long F[]={0 ... 99}=-1; // initialisation en gcc
    if(n<2) return n;
    if(F[n]<0) F[n]=fibo_mem(n-1)+fibo_mem(n-2); // déjà calculée?
    return F[n];
}
```

Parenthèse. Dans cette implémentation on a utilisé une variable locale `static long F[]` qui est allouée et initialisée à -1 dans la mémoire statique (et donc pas sur la pile) au moment de la compilation. Ce tableau n'est accessible que localement par la fonction qui l'a déclarée mais le contenu est préservé entre les différents appels comme une variable globale. On parle parfois de variable locale globale. Dans cet exemple, on aurait très bien pu déclarer `F[]` en dehors de `fibo_mem()` comme variable globale.

La différence de code entre les deux fonctions est minime alors que l'amélioration de la complexité est exponentielle ! Quand on pense qu'on aurait déclarer `F[]` en dehors du corps de la fonction, le code de `fibo_mem()` est presque identique à celui de `fibo()`. D'ailleurs certains langages comme Python permettent de faire automatiquement cette transformation. C'est le principe de *décoration* disponible à partir de la [version 3.2](#).

```
@lru_cache(maxsize=None)
def fibo(n):
    if n<2: return n
    return fibo(n-1)+fibo(n-2)
```

La complexité de `fibo(n)` est $2^{\Theta(n)}$, l'arbre des appels étant essentiellement un arbre binaire presque complet de hauteur n . [[Question. Pourquoi n'est-il pas complet ?](#)] L'autre argument qui montre qu'elle exponentielle en n est qu'elle calcule un nombre exponen-

3. Le 100e nombre de Fibonacci tient sur pas moins de 70 bits, soit plus grand que ce que peut contenir le type `long` (64 bits) ce qui explique la taille maximale pour `F[]` dans sa déclaration.

iel⁴ en n avec seulement des additions et les constantes positives < 2 (cas terminal), soit 0 et 1.

La complexité de `fibo_mem(n)` est cependant seulement de $O(n)$. Le gain est donc important. Pour le voir il faut construire l'arbre des appels pour constater qu'il ne comporte que $2n - 1$ nœuds (cf. figure 2.5).

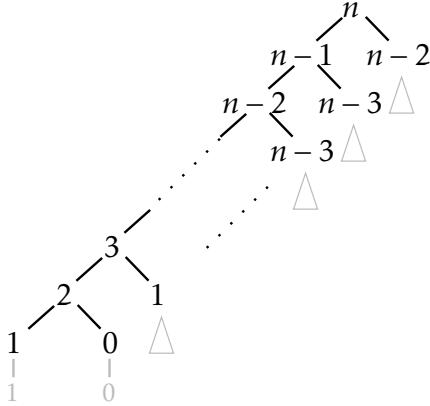


FIGURE 2.5 – Arbre des appels de `fibo_mem(n)` avec ses $2n - 1$ nœuds, les parties grisées ne faisant pas partie de l'arbre. L'exécution, comme pour `fibo(n)`, consiste à parcourir l'arbre selon un parcours en profondeur, sauf que les sous-arbres grisés ne se développent pas. Ils correspondent à une simple lecture dans la table $F[]$.

Parenthèse. On peut aussi faire en $O(\log n)$ avec une technique différente. On pose $\vec{F}_n = \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix}$ le vecteur composé des n -ème et $(n-1)$ -ème nombres de Fibonacci. On remarque alors que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{F}_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} F(n) + F(n-1) \\ F(n) \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \vec{F}_{n+1}.$$

Et donc

$$\vec{F}_{n+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{F}_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot F_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix}.$$

En posant $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, on en déduit que

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a+b \\ c \end{pmatrix}$$

d'où $c = F(n)$. En utilisant l'exponentiation rapide, soit

$$x^n \mapsto \begin{cases} (x \cdot x)^{n/2} & \text{si } n \text{ est pair} \\ x \cdot x^{n-1} & \text{sinon} \end{cases}$$

4. On a vu dans l'équation (2.1) que $F(n) \approx \Phi^n / \sqrt{5} \approx 1.61^n$.

on peut calculer le coefficient c et de $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ après au plus $2\log n$ multiplications de matrices 2×2 , ce qui fait une complexité en $O(\log n)$ pour le calcul de $F(n)$.

Bien sûr, on sait tous comment calculer en temps $O(n)$ les nombres de Fibonacci sans table auxiliaire. Il suffit de faire une simple boucle du type

```
for (u=0, v=1, i=2; i<n; i++) t=u, u=v, v+=t;
```

Mais le code est relativement différent de `fibo()`. Il est fortement basé sur le fait qu'il suffit de mémoriser les deux valeurs précédentes pour calculer la prochaine. Le fait que le code avec une boucle `for()` soit assez différent de l'original tend à montrer que la transformation n'est peut être pas si générique que cela. On peut légitimement se demander s'il est possible de faire de même pour toute fonction similaire, c'est-à-dire un code équivalent sans table auxiliaire utilisant une boucle à la place d'appels récursifs pour toute fonction ayant disons un paramètre entier et deux appels récursifs ?

Autant la technique de mémorisation paresseuse, on va le voir, est assez générale, autant la simplification par une simple boucle sans table auxiliaire n'est pas toujours possible. Pour s'en convaincre considérons la fonction :

```
long f(int n){
    if(n<2) return n;
    long u=f(n-1);
    return u+f(u%n);
}
```

Peut-on transformer la fonction `f()` ci-dessus avec une simple boucle et sans table auxiliaire ? [Exercice. Est-t-il bien sûr que cette fonction termine toujours?] Comme le suggère l'arbre des appels (cf. figure 2.6 à droite), le deuxième appel (fils droit) est difficile à prévoir puisqu'il dépend de la valeur de l'appel gauche. De plus ils peuvent être éloignés l'un de l'autre comme pour $f(28) = f(27) + f(2) = 1124$.

En fait, la fonction d'Ackermann que l'on rencontrera page 77 est un exemple bien connu de fonction comportant deux appels récursifs et deux paramètres entiers qu'il n'est pas possible de rendre itérative (sans table auxiliaire). La raison fondamentale à ceci est que la fonction d'Ackermann croît beaucoup plus rapidement que ce qu'il est possible de faire avec une (ou plusieurs) boucle(s) et un nombre constant de variables.

Avec une liste chaînée. [Cyril. Partie à terminer ...]

L'exemple précédent avec les nombre de Fibonacci est plutôt simpliste. L'appel ne comporte qu'un seul paramètre (ici un entier n), et on sait que la table a une taille

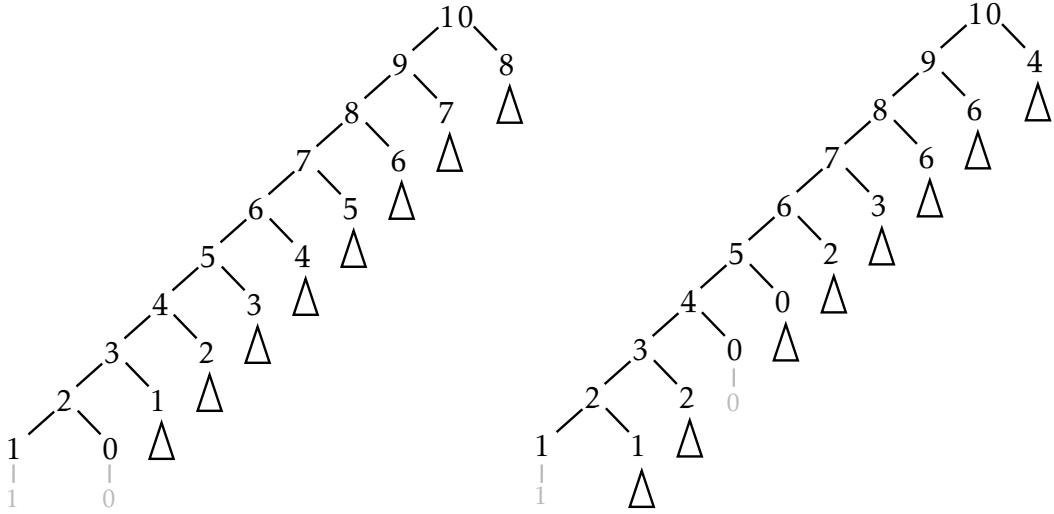


FIGURE 2.6 – Arbre des appels pour `fibo(10)=55` (à gauche) et `f(10)=38` (à droite). Pour `fibo()` on remarque que les appels qui se répètent sont à distance bornée dans l’arbre (relation oncle-neveu), ce qui montre qu’un nombre constant de variables dans une simple boucle suffit. En revanche, pour `f()`, la distance entre nœuds identiques est variable et plus importante, comme pour $f(10) = f(9) + f(4) = 24 + 14$, ce qui nécessite *a priori* un stockage bien plus important de variables (table) ou bien la répétition de calculs (réursifs).

maximum définie à l’avance (ici 100).

Considérons un exemple générique plus complexe du calcul hypothétique d’une certaine fonction récursive f ayant plusieurs paramètres entiers, disons deux pour fixer les idées, mais le principe s’applique dès qu’on a un nombre fixé de paramètres. Autrement, dit les nœuds dans l’arbre des appels sont des couples d’entiers comme sur la figure 2.4.

On modifie f de la façon suivante. À chaque fois qu’on calcule récursivement une valeur, comme dans `t=f(i,j)`, on cherche si (i,j) est déjà en mémoire. Si oui on renvoie la valeur et sinon on la calcule récursivement, l’ajoute à la chaîne et renvoie sa valeur.

...

On note T l’arbre des appels pour $f(i_0, j_0)$.

...

Si k est le nombre de nœuds différents, alors le temps de recherche et de mise à jour de la liste sera de $O(k^2)$, puisque la liste sera bien évidemment de taille au plus k .

Le parcours de l’arbre lors de tous les appels suit un parcours en profondeur. L’effet de la mémorisation est le suivant : si p est un nœuds que l’on visite et qui a déjà visité, alors tous les nœuds descendant de p ne seront pas visités. En quelque sorte, on visite T en supprimant (ou coupant) des sous-arbres en dessous des nœuds déjà visités. On visite donc plusieurs fois le même nœud, mais aucun de ces descendant si c’est la 2e

fois.

La question est alors : combien de nœuds visite-t-on dans T ?

Soit X l'ensemble des nœuds différents de T rencontrés lors du parcours (en profondeur) de T . Par définition de X , les voisins de X dans T et qui ne sont pas dans T ont déjà été rencontrés. Soit Y cet ensemble des voisins de X dans T . Il suit que $T' = T[X \cup Y]$ car d'après le processus aucun autre descendant de Y n'est dans T' .

On a $|X| = k$. On peut borner la taille de Y par $|Y| \leq |X| \cdot \deg(T)$ où $\deg(T)$ représente le nombre de fils maximum d'un nœuds de T , ce qui correspond aussi au nombre maximum d'appels récursifs de la fonction f .

[DESSIN D'ARBRE]

Avec un arbre équilibré. [Cyril. À finir.]

2.6 Morale

- La récursivité à l'aide de formules de récurrence permettent d'obtenir des programmes concis, rapide à développer et dont la validité est facile à vérifier.
- La complexité peut être catastrophique si l'arbre des appels contient des parties communes. On passe alors son temps à recalculer des parties portant sur les mêmes paramètres (c'est-à-dire les mêmes appels). C'est le cas lorsque la taille de l'arbre (son nombre total de nœuds) est beaucoup plus grand que le nombre d'appels différents (le nombre de nœuds qui sont différents). Pour la partition de n , il y a $2^{\Theta(\sqrt{n})}$ nœuds dans l'arbre, alors qu'il y a seulement n^2 appels différents possibles.
- La mémorisation permet d'éviter le calcul redondant des sous-arbres communs. Plus généralement, la programmation dynamique utilise des récurrences à travers une table globale indexée par les divers paramètres des appels.
- La programmation dynamique permet alors d'économiser du temps par rapport à l'approche récursive naïve. L'inconvénient est l'usage de mémoire supplémentaire (tables) qui, en cas de pénurie, peut être problématique. Car concrètement, en cas de pénurie, il faut soit repenser l'algorithme soit modifier la machine en lui ajoutant de la mémoire. Le manque de temps est peut être plus simple à gérer en pratique puisqu'il suffit d'attendre.
- Une difficulté dans la programmation dynamique est qu'il faut souvent réfléchir un peu plus, par rapport à la version récursive, quant au parcours de la table pour être certain de remplir une case en fonction des cases déjà remplies. La difficulté va apparaître au chapitre suivant au paragraphe 3.3. On peut y remédier grâce

à la mémorisation paresseuse, qui combine l'approche récursive et la mémorisation : on fait le calcul et les appels récursifs seulement si l'appel n'est pas déjà en mémoire.

- Un exemple de programmation dynamique déjà vu, autre le calcul du nombre de partitions d'un entier, est le calcul de plus courts chemins à partir d'un sommet dans un graphe. Tester tous les chemins possibles et prendre le plus courts est beaucoup trop couteux. Par exemple, entre deux sommets diagonaux d'une grille $n \times n$, il existe $\binom{2n}{n} \sim 2^{2n-o(n)}$ chemins de longueur $2n$ (cf. figure 2.7). Ce nombre dépasse le limite fatidique de 10^{18} dès que $n = 32$. À la place on utilise l'algorithme

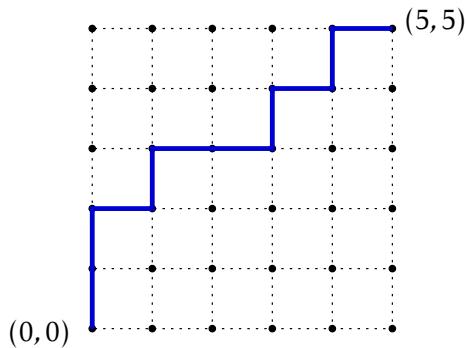


FIGURE 2.7 – Un plus court chemin dans une grille carrée entre sommets diagonaux. Ce chemin de longueur 10 est codé par le mot « $\uparrow\uparrow\rightarrow\uparrow\rightarrow\rightarrow\uparrow\rightarrow\uparrow\rightarrow$ » contenant 5 pas « montant » (\uparrow) et 5 pas « à droit » (\rightarrow). Pour $n = 5$, cela fait $\binom{10}{5} = 252$ chemins possibles. En effet, construire un chemin de longueur $2n$ entre les coins $(0,0)$ et (n,n) revient à choisir n « pas montant » parmi les $2n$, ce qui donne $\binom{2n}{n}$ possibilités.

de Dijkstra qui mémorise dans un tableau la distance (D) entre la source et tous les sommets à distance $\leq L$ (au début $L = 0$ et D ne contient que la source). Les distances $D[v]$ des sommets v situés à distance immédiatement supérieure, c'est-à-dire à distance $> L$, sont alors calculées à partir des sommets u de la table D par une formule de récurrence du type :

$$D[v] = \min_{u \in D, uv \in E} \{D[u] + d(u, v)\} .$$

Bibliographie

- [HR18] G. H. HARDY AND S. A. RÂMÂNUJAN, *Asymptotic formulæ in combinatory analysis*, in Proceedings of the London Mathematical Society, vol. 17, 2, 1918, pp. 75–115. doi : [10.1112/plms/s2-17.1.75](https://doi.org/10.1112/plms/s2-17.1.75).
- [Kno81] M. I. KNOPP, *Analytic Number Theory – Proceedings of a Conference Held at Temple University, Philadelphia, USA, May 12-15, 1980*, vol. 899 of Lecture Notes in Mathematics, Springer-Verlag, 1981. doi : [10.1007/BFb0096450](https://doi.org/10.1007/BFb0096450).

Sommaire

3.1 Le problème	57
3.2 Approche exhaustive	59
3.3 Programmation dynamique	61
3.4 Approximation	67
3.5 Morale	90
Bibliographie	91

Mots clés et notions abordées dans ce chapitre :

- problème d'optimisation
- inégalité triangulaire
- problème difficile
- algorithme d'approximation
- facteur d'approximation
- heuristique

3.1 Le problème

Un robot doit ramasser un ensemble d'objets en un minimum de temps et revenir au point de départ. L'ordre de ramassage n'a pas d'importance, seul le temps (ou la distance parcouru) doit être optimisé.

Une autre instance du même problème est celui où un hélicoptère doit inspecter un ensemble de plateformes *offshore* et revenir à son point de départ sur la côte. Il veut parcourir les plateformes en utilisant le moins de carburant possible. Une autre formulation est que l'hélicoptère possède une quantité de carburant C et il veut savoir s'il va pouvoir visiter toutes les plateformes avant de revenir.

La première formulation est un problème d'*optimisation* (la réponse est une valeur), alors que la seconde (avec un budget maximum C donné) est un problème de *décision*

(la réponse est « oui » ou « non »).

Dans la littérature et historiquement, on parle plutôt du problème du **VOYAGEUR DE COMMERCE**, TSP en Anglais pour *Traveler Salesman Problem*. Un commercial doit effectuer une tournée comprenant n villes et il faut déterminer l'ordre de visite qui minimise la longueur de la tournée (cf. la figure 3.1).

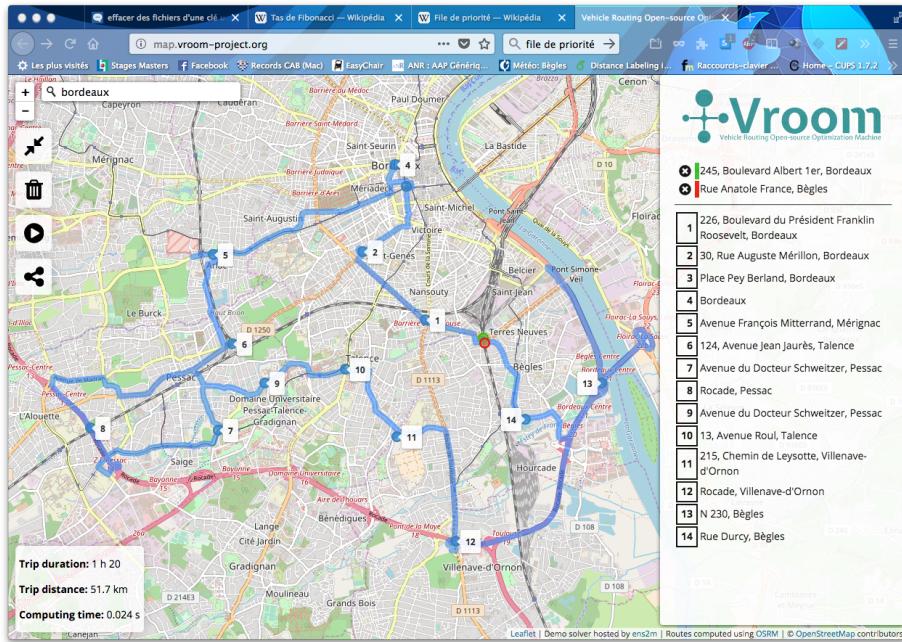


FIGURE 3.1 – L'application **Vroom** propose des solutions au problème du **VOYAGEUR DE COMMERCE** sur une carte routière réelle.

Formellement le problème est le suivant :

VOYAGEUR DE COMMERCE

Instance: Un ensemble V de points et une distance d sur V .

Question: Trouver une tournée de longueur minimum passant par tous les points de V , c'est-à-dire un ordre v_0, \dots, v_{n-1} des points de V tel que $\sum_{i=0}^{n-1} d(v_i, v_{i+1 \text{ mod } n})$ est minimum.

En fait, il existe plusieurs variantes du problème. Pour celle que l'on considérera, la plus classique, d est une distance. En particulier, c'est une fonction qui doit vérifier inégalité triangulaire dont on rappelle la définition.

Une fonction $d(\cdot, \cdot)$ vérifie l'*inégalité triangulaire* si $d(A, B) \leq d(A, C) + d(C, B)$ pour tout triplet d'éléments A, B, C .

Cette inégalité tire son nom du fait que dans un triangle la longueur d'un coté

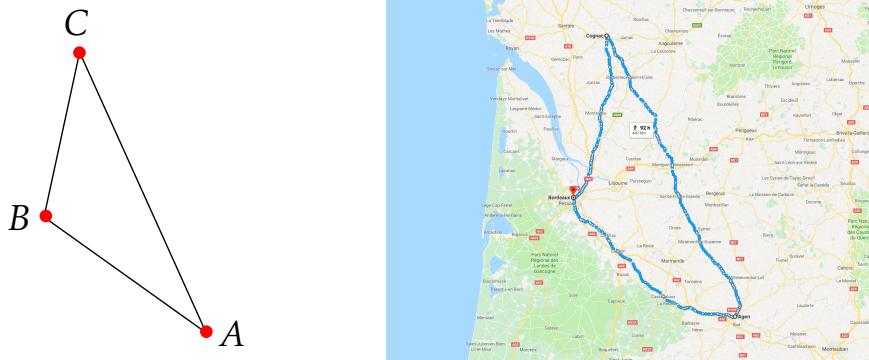


FIGURE 3.2 – Inégalité triangulaire.

est toujours plus petite (ou égale) que la somme des deux autres (voir figure 3.2). La distance euclidienne¹ vérifie l'inégalité triangulaire. Le trajet Agen-Bordeaux est plus court que le trajet Agen-Cognac-Bordeaux.

On parle ainsi de TSP « métrique » lorsque d vérifie l'inégalité triangulaire. Dans la version générale du TSP, c'est-à-dire lorsque d n'est plus forcément une distance vérifiant l'inégalité triangulaire, on doit ajouter que la tournée passe une et une seule fois par chacun des points.

Il existe aussi un TSP « asymétrique », lorsque $d(A, B) \neq d(B, A)$. Notons que dans le réseaux Internet l'inégalité triangulaire n'est, en général, pas respectée ; de même que la symétrie (c'est le « A » de l'ADSL). Les temps de trajet entre gares du réseau ferré ne vérifient pas non plus l'inégalité triangulaire. Le trajet Bordeaux → Lyon par la ligne traversant le Massif central est plus long (en temps) que le trajet Bordeaux → Paris-Montparnasse → Paris-Gare-de-Lyon → Lyon.

Il y a aussi la variante où les points sont les sommets d'un graphe avec des arêtes valuées et la distance est la distance dans le graphe. La tournée, qui doit visiter tous les sommets, ne peut passer que par des arêtes du graphe. Elle peut être amené à passer plusieurs fois par le même sommet. On parle de TSP « graphique ».

3.2 Approche exhaustive

La question de savoir s'il existe une formule close n'a pas vraiment de sens puisque le nombre de paramètres n'est pas borné (le nombre de points). On ne risque pas d'avoir une formule de taille bornée ...

1. En dimension deux, la distance euclidienne entre les points (x, y) et (x', y') vaut $\sqrt{(x' - x)^2 + (y' - y)^2}$, formule que l'on peut démontrer grâce au théorème de Pythagore. De manière générale, en utilisant les multiples triangles rectangles liés aux projections sur chacune des dimensions on montre facilement que la distance euclidienne en dimension $\delta \geq 1$ entre (x_1, \dots, x_δ) et (x'_1, \dots, x'_δ) vaut $\sqrt{\sum_{i=1}^{\delta} (x'_i - x_i)^2}$.

Pour l'approche exhaustive, il suffit généralement de commencer par se poser deux questions :

- (1) Quelle est la sortie attendue d'un algorithme qui résoudrait le problème ?
- (2) Comment faire pour savoir si la sortie est celle que l'on veut ?

Pour la question (1), c'est un ordre sur les n points que l'on cherche. Pour la question (2), c'est l'ordre qui minimise la longueur de la tournée. Visiblement, on peut calculer tout cela. On a donc un algorithme !

Principe. Générer tous les ordres possibles, calculer la longueur de chaque tournée et ne garder que la plus petite.

Complexité en temps. Le nombre d'ordres possibles sur n points est le nombre de permutations, soit $n!$. Une fois l'ordre des points fixé, le calcul de la tournée prend un temps $O(n)$ pour calculer la somme des n distances. Mettre à jour et retenir le minimum prend un temps constant. Au final, la complexité en temps de l'algorithme *brute-force* est :

$$O(n \cdot n!).$$

Notez bien qu'il n'y a pas de notation standard pour la complexité en temps d'un algorithme.

Combien de temps cela prendra-t-il en pratique ? La formule de Stirling donne l'asymptotique suivant :

$$n! \sim \left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}$$

En fait, pour tout $n > 0$, on a $n! > (n/e)^n$. [[Question. Peut-on le déduire de la formule de Stirling ?](#)] Rappelons que $e = \exp(1) = 2.718281828\dots$. Pour $n = 20$, cela nous donne un temps approximatif d'au moins $n \cdot n! > 20 \cdot (20/2.72)^{20} = 10^{18.63\dots} > 10^9 \times 10^9$. C'est donc 30 ans (1 milliard de secondes) sur notre processeur 1 GHz.

Des ordres de grandeurs importants à connaître. On reparlera des ordres de grandeurs plus tard au paragraphe [5.2.5](#), mais voici deux ordres de grandeurs qu'il faut avoir en tête :

- En un milliardième de seconde, soit la durée de 10^{-9} s ou encore 1 GHz, la lumière se déplace d'au plus 30 cm (et encore dans le vide, car dans le cuivre c'est 10% de moins). Ceci explique que les processeurs cadencés à plus d'1 GHz sont généralement de taille < 30 cm puisque sinon la communication est impossible dans le délai imparti.

- Un milliard de secondes, soit 10^9 s, correspond à une durée supérieure à 30 ans. Ainsi sur un ordinateur 1 GHz pouvant exécuter un milliard d'opérations élémentaires par seconde, il faudra que la complexité de l'algorithme soit $< 10^9 \times 10^9 = 10^{18}$.

3.3 Programmation dynamique

On va présenter l'algorithme de Held – Karp découvert indépendamment par Bellman en 1962 qui résout le problème du **VOYAGEUR DE COMMERCE**. En fait, il fonctionne même si d ne vérifie pas l'inégalité triangulaire et/ou n'est pas symétrique. On a juste besoin que $d(A, B) \geq 0$. D'ailleurs c'est la même chose pour l'algorithme *brute-force* qui pour fonctionner n'utilise ni la symétrie, ni l'inégalité triangulaire.

La formulation du problème semble indiquer qu'il n'y a pas vraiment d'alternative à chercher parmi toutes les tournées possibles celles de longueur minimum. Et pourtant ...

Observons d'abord que l'algorithme *brute-force* teste inutilement de nombreux cas. Supposons que parmi toutes les tournées possibles, on s'intéresse à toutes celles qui passent par les points $v_1, S_1, v_2, S_2, v_3, S_3, v_4, S_4, v_5$ (où les S_i sont des ensembles de points) comme représenté sur la figure 3.3. Elles doivent passer par v_1, \dots, v_5 mais sont libres de circuler dans chaque S_i par le point du haut ou du bas. Comme chaque S_i possède deux points, le nombre de chemins possibles est donc $2 \times 2 \times 2 \times 2 = 2^4 = 16$. L'approche *brute-force* va donc tester ces 16 chemins.

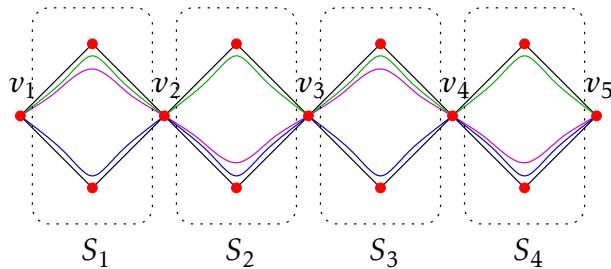


FIGURE 3.3 – Trois chemins parmi les 16 visitant l'ensemble de points $v_1, S_1, v_2, S_2, v_3, S_3, v_4, S_4, v_5$ dans cet ordre. Le chemin minimum visitant v_1, S_1, v_2 est calculé deux fois.

Cependant, si on avait commencé par résoudre (récursevivement?) le problème du meilleur des deux chemins allant de v_i à v_{i+1} et passant par S_i , pour chacun des 4 ensembles, alors on aurait eut à tester seulement $2 + 2 + 2 + 2 = 2 \times 4 = 8$ chemins contre 16 pour l'approche *brute-force*. L'écart n'est pas très impressionnant car les S_i ne contiennent que deux points. S'ils en contenaient 3 par exemple, la différence serait alors de $3 \times 4 = 12$ contre $3^4 = 81$ pour le *brute-force*.

L'algorithme par programmation dynamique est un peu basé sur cette intuition.

Comme pour la partition d'un entier, pour exprimer une formule de récurrence on a besoin de définir une variable particulière (comme $p(n, k)$ au lieu de $p(n)$).

La variable. Dans la suite, on supposera que la tournée recherchée commence, ou plutôt termine, au point v_{n-1} . Ce choix est arbitraire². Pour simplifier les notations, on notera $V^* = V \setminus \{v_{n-1}\} = \{v_0, \dots, v_{n-2}\}$ qui est donc l'ensemble des points de V sans le dernier.

Attention ! l'ordre v_0, v_1, \dots, v_{n-1} n'est pas ici la tournée de longueur minimum comme dans la formulation encadrée du problème. C'est simplement les indices des points d'origine. L'indexation des points est donc totalement arbitraire sans lien avec la solution.

L'algorithme de programmation dynamique repose sur la variable $D(t, S)$, définie pour tout sous-ensemble de points $S \subseteq V^*$ et tout point $t \in S$, comme ceci :

$$D(t, S) = \begin{cases} \text{la longueur minimum d'un chemin allant de } v_{n-1} \notin S \text{ à } t \in S \text{ et qui} \\ \text{visite tous (et seulement) les points de } S. \end{cases}$$

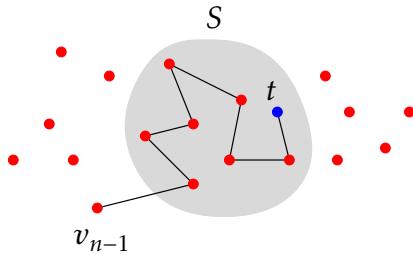


FIGURE 3.4 – Chemin de longueur minimum allant de v_{n-1} à t visitant tous les points de S .

Le « seulement » dans la définition précédente est nécessaire seulement si d ne vérifie pas l'inégalité triangulaire. Sinon, dans le cas du TSP métrique, le chemin de longueur minimum visitant tous les sommets de S ne peut emprunter de sommet en dehors de S (sauf v_{n-1}), car le chemin direct $x - y$ entre deux points de S est plus court (ou égal) que tout chemin $x - z - y$ avec $z \notin S$.

Notons $\text{OPT}(V, d)$ la solution optimale recherchée, c'est-à-dire la longueur minimum de la tournée pour l'instance (V, d) du VOYAGEUR DE COMMERCE. Il est facile de voir que

$$\text{OPT}(V, d) = \min_{t \in V^*} \{D(t, V^*) + d(t, v_{n-1})\}. \quad (3.1)$$

En effet, la tournée optimale part de v_{n-1} , visite tous les points de V^* pour se terminer en un certain point $t^* \in S^*$ avant de revenir en v_{n-1} (cf. la figure 3.5). Donc

2. Pour des raisons d'implémentation, on verra que c'est plus malin de choisir v_{n-1} que v_0 par exemple.

$\text{OPT}(V, d) = D(t^*, V^*) + d(t^*, v_{n-1})$. Or $D(t^*, V^*) + d(t^*, v_{n-1}) \geq \min_{t \in V^*} \{D(t, V^*) + d(t, v_{n-1})\}$ par définition du minimum. Et comme $D(t, V^*) + d(t, v_{n-1})$ représente la longueur d'une tournée, c'est que $\text{OPT}(V, t) = \min_{t \in V^*} \{D(t, V^*) + d(t, v_{n-1})\}$.

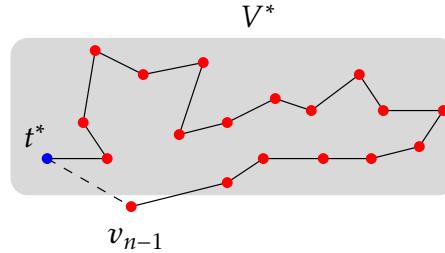


FIGURE 3.5 – Tournée optimale à l'aide d'un chemin minimum de v_{n-1} à t^* visitant tous les points de V^* .

Formule de récurrence. L'idée est de calculer $D(t, S)$ à partir de sous-ensembles strictement inclus dans S . Supposons que $v_{n-1} - s_1 - \dots - s_k - x - t$ soit un chemin de longueur minimum parmi les chemins allant de v_{n-1} à t et visitant tous les points de $S = \{s_1, \dots, s_k, x, t\}$. Sa longueur est précisément $D(t, S)$ par définition de la variable $D(t, S)$. L'observation élémentaire, mais cruciale, est que le sous-chemin $v_{n-1} - s_1 - \dots - s_k - x$ est aussi un chemin de longueur minimum de v_{n-1} à x visitant tous les points de $S \setminus \{t\}$. Il est donc de longueur $D(x, S \setminus \{t\})$. En effet, s'il y en avait un autre plus court, alors en rajoutant le segment $x - t$ on déduirait une longueur de chemin de $v_{n-1} - s_1 - \dots - s_k - x - t$ plus courte que $D(t, S)$. Notez qu'on a pas utilisé l'inégalité triangulaire ni la symétrie pour démontrer cette propriété.

De cette discussion, on en déduit la formule suivante, définie pour tout $S \subseteq V^*$ et tout $t \in S$:

$$D(t, S) = \begin{cases} d(v_{n-1}, t) & \text{si } |S| = 1 \\ \min_{x \in S \setminus \{t\}} \{D(x, S \setminus \{t\}) + d(x, t)\} & \text{si } |S| > 1 \end{cases} \quad (3.2)$$

On rappelle que $|S|$, lorsque S est un ensemble, représente la cardinalité de S (son nombre d'éléments). Notons que la condition « $|S| = 1$ » est équivalente à poser « $S = \{t\}$ » étant donné qu'on doit avoir $t \in S$.

Implémentation récursive. De l'équation (3.2), on déduit immédiatement l'implémentation triviale suivante, en supposant déjà définies quelques opérations de bases sur les ensembles comme `set_card`, `set_in`, `set_minus`, `set_create`, `set_free`. Pour simplifier, `V`, `n` et `d` sont des variables globales et ne sont pas passées comme paramètres.

```

double D_rec(int t, set S){ // Calcul récursif de D(t,S)
    if(set_card(S)==1) return d(V[t],V[n-1]); // si |S|=1
    double w=DBL_MAX; // w=+∞
    set T=set_minus(S,t); // T=S \ {t}
    for(int x=0;x<n-1;x++) // pour tout x ∈ S:
        if(set_in(x,T)) // si x ∈ T
            w=fmin(w,D_rec(x,T)+d(V[x],V[t])); // min_x(D(x,T)+d(x,t))
    set_free(T);
    return w;
}

```

```

double tsp_tour(){
    double w=DBL_MAX; // w=+∞
    set S=set_create(n-1); // crée S = {0,...,n-2} = V*
    for(int t=0;t<n-1;t++) // min_t(D(t,V*) + d(t,v_{n-1}))
        w=fmin(w,D_rec(t,S)+d(V[t],V[n-1]));
    set_free(S);
    return w;
}

```

Bien évidemment, il ne faut absolument pas implémenter l'algorithme de cette manière. L'arbre des appels est composé à la racine de $n-1$ branches (à cause du `for()` dans `tsp_tour()`), qui se subdivisent chacune en $n-2$ appels lors du premier appel à `D_rec()`, qui génère à son tour $n-3$ appels, puis $n-4$, etc. car le paramètre S (via T) diminue d'un point à chaque récursion. Le nombre total d'évaluations est au moins le nombre de feuilles de cet arbre qui vaut $(n-1)!$. Ce n'est donc pas vraiment mieux que l'approche exhaustive³.

On voit aussi que l'algorithme va passer son temps à recalculer les mêmes sous-problèmes. Chaque branche correspond à un choix du dernier points de la tournée : le point `t` dans `tsp_tour()` puis le point `x` dans `D_rec()`. Du coup il y aura, par exemple, six embranchements correspondant aux $3!$ façons de terminer la tournée par les points $v_{n-1}, v_{n-2}, v_{n-3}$ et qui vont tous faire un appel à, par exemple, `D_rec(n-4,{1,...,n-4})`. Donc `D_rec(n-4,{1,...,n-4})` est évalué six fois (au moins).

Une autre évidence de la présence de calculs inutiles est que les nœuds de l'arbre des appels sont des paires (t, S) où $t \in S$ et $S \subseteq V^*$. Le nombre d'appels distincts est donc au plus $|V^*| \cdot 2^{|V^*|} = (n-1) \cdot 2^{n-1}$ ce qui est bien plus petit que le nombre nœuds de l'arbre des appels qui est de $(n-1)!$ (au moins). [Question. Pourquoi?]

3. En pratique c'est sans doute plus lent car on va faire en plus autant de `malloc()` et de `free()` pour la construction de T dans les appels à `D_rec()`.

Mémorisation. On va donc utiliser une table $D[t][S]$ à deux dimensions pour stocker les valeurs $D(t, S)$ et éviter de les recalculer sans cesse. Pour simplifier l'implémentation on représentera un sous-ensemble $S \subseteq \{v_0, \dots, v_{n-1}\}$ directement par un entier de n bits, aussi noté S , chaque bit indiquant si $v_i \in S$ ou pas. Plus précisément, $v_i \in S$ ssi le bit en position i de S est mis à 1. Les positions commencent à 0 de sorte que l'entier 2^i représente tout simplement le singleton $\{v_i\}$.

Par exemple, si $S = \{v_3, v_2, v_0\}$ et $n = 5$, alors on aura :

$$S = \begin{matrix} v_4 & v_3 & v_2 & v_1 & v_0 \\ 0 & 1 & 1 & 0 & 1 \end{matrix} = \{v_3, v_2, v_0\} = 13_{\text{dix}}$$

On peut ainsi coder très facilement les opérations sur les ensembles de taille $n = 32$ ou 64 (dépendant de l'architecture), ce qui est amplement suffisant.

Les lignes de la table $D[t][S]$ représentent les points (t) et les colonnes les sous-ensembles (S). Voir la figure 3.6 pour un exemple avec $n = 5$ points. Comme S ne contient jamais $v_{n-1} = v_4$, il sera représenté en fait par un entier de $n - 1 = 4$ bits obtenu en supprimant le bit le plus à gauche qui vaut toujours 0.

s	$0001 \{v_0\}$	$0010 \{v_1\}$	$0011 \{v_1, v_0\}$	$0100 \{v_2\}$	$0101 \{v_2, v_0\}$	$0110 \{v_2, v_1\}$	$0111 \{v_2, v_1, v_0\}$	$1000 \{v_3\}$	$1001 \{v_3, v_0\}$	$1010 \{v_3, v_1\}$	$1011 \{v_3, v_1, v_0\}$	$1100 \{v_3, v_2\}$	$1101 \{v_3, v_2, v_0\}$	$1110 \{v_3, v_2, v_1\}$	V^*	
t	1	2	3	4	5	6	7	11	8	9	10	11	12	13	14	15
v_0																
v_1																
v_2																
v_3																

FIGURE 3.6 – Table $D[t][S]$. La colonne correspondant à l'ensemble vide ($s = 0000$) n'est pas représentée. La ligne correspondante à v_4 n'a pas besoin d'être dans la table. Les numéros de la dernière ligne indique dans quel ordre parcourir les colonnes pour remplir la table. Les cellules colorées n'ont pas à être calculées. [Question. Pourquoi?]

On remplit chaque case $D[t][S]$ de la table à l'aide de l'équation (3.2). La difficulté principale est de décider dans qu'elle ordre les remplir. La formule de récurrence précise que pour calculer $D(\cdot, S)$ il faut $D(\cdot, T)$ pour tous les sous-ensembles $T \subset S$ ayant juste un élément de moins. Il suffit donc de lister les ensembles par taille croissante. Par exemple, il faut remplir d'abord les colonnes 1, 2, 4, 8 (ensembles de taille 1), pour pouvoir remplir les colonnes 3, 5, 6, 9, 10, 12 (ensembles de taille 2). De plus, dans chaque

colonne il faut faire attention de ne remplir que les cases correspondant à des sommets de S (de couleur blanche).

Pour trouver cet ordre on utilisera une fonction `set_next(S,n)` permettant de générer le prochain ensemble juste suivant S qui a soit le même nombre d'éléments (en terme d'entier, c'est l'entier strictement supérieur ayant le même nombre de 1), soit juste un de plus. L'algorithme permettant d'implémenter cette fonction est le suivant : dans S on décale complètement à droite le premier bloc de 1 (celui contenant le bit de poids faible) sauf le bit le plus à gauche de ce bloc qui lui est décalé d'une position à gauche. Si on ne peut pas décaler ce bit à gauche (car arrivé en fin de mot) on renvoie $2^p - 1$ où p est le nombre de 1 désirés, celui de S plus 1. Ainsi les 16 entiers sur 4 bits sont parcourus dans l'ordre suivant, à partir de $S=1$ et en répétant `S=set_next(S,4)` :

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 9 \rightarrow 10 \rightarrow 12 \rightarrow 7 \rightarrow 11 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 0.$$

Récupérer la tournée. Pour déterminer la longueur $\text{OPT}(V,d)$ de la tournée optimale une fois la table calculée, il faut examiner la dernière colonne, celle correspondant à l'ensemble le plus grand soit $S = V^*$, et appliquer la formule de l'équation (3.1). Si l'on souhaite de plus extraire la tournée (l'ordre des points réalisant ce minimum), il faut stocker plus d'informations dans la table. Plus précisément, il faut mémoriser pour quel point x la longueur minimum de $D(t,S)$ a été atteinte, c'est-à-dire le sommet précédent t .

Complexité en espace. Le nombre de mots mémoire utilisés est, à un facteur constant près, majoré par le nombre de cases de la table qui est $(n-1) \cdot 2^{n-1}$. Donc la complexité en espace est $O(n \cdot 2^n)$.

Complexité en temps. L'algorithme se résume donc à remplir la table `D[t][S]` et à récupérer la longueur de la tournée grâce à la dernière colonne. Déterminer la tournée, en particulier le calcul de $\text{OPT}(V,d)$ grâce à l'équation (3.1), se fait en temps $O(n)$ une fois la table calculée.

On a vu que le nombre de cases de la table est $(n-1) \cdot 2^{n-1}$. Remplir une case nécessite le calcul d'un minimum $x \in S \setminus \{t\}$ se qui prend un temps $O(n)$ car il y n'a pas plus de n éléments x à tester. Donc le remplissage de toutes les cases prend un temps de $O(n^2 \cdot 2^n)$, même si on remarque que la moitié des cases de la table ne sont pas utilisées. (En fait chacune des lignes est utilisée à moitié puisqu'un point v_i est présent dans exactement la moitié des sous-ensembles $S \subset V^*$.) Il faut aussi ajouter le temps nécessaire pour passer d'une colonne à la suivante, c'est-à-dire tenir compte de la complexité de la fonction `set_next()`. Bien qu'en pratique elle prenne un temps $O(1)$ lorsque n ne dépasse pas la taille des mots mémoires, elle prendrait de manière générale un temps au plus $O(n)$ même si on implémentait les ensembles par de simples tableaux de taille n . Il y a 2^{n-1}

colonnes et donc autant d'appels à la fonction `set_next()`, ce qui prend un temps de $O(n \cdot 2^n)$ pour tous ces appels.

Au total, la complexité en temps de l'algorithme est de :

$$n^2 \cdot 2^n + n \cdot 2^n = O(n^2 \cdot 2^n).$$

En pratique. Pour $n = 20$, nous avons vu que l'approche exhaustive prenais 30 ans sur un ordinateur 1 GHz. Et en pratique, c'est plutôt des valeurs de $n = 10, 11$ ou 12 qu'il est possible de résoudre par l'approche exhaustive. Dans notre cas, la complexité en temps devient $n^2 \cdot 2^n \approx 20^2 \cdot 2^{20} < 2^9 \cdot 2^{20} < 10^9$ ce qui fait 1s sur le même ordinateur. En TP on va voir qu'effectivement $n = 20$ voir un peu plus est largement faisable en pratique. Si on avait 30 ans devant nous, alors on pourrait résoudre une instance de taille ... $n = 49$. Ceci justifie l'usage des entiers par le codage des ensembles de taille n .

Il s'agit du meilleur algorithme connu pour résoudre de manière exacte le **VOYAGEUR DE COMMERCE**. Notons en passant que c'est un problème ouvert de savoir s'il existe un algorithme de complexité en temps $c^{n+o(n)}$ avec $c < 2$ une constante (et $n = |V|$). La meilleure borne inférieure connue pour la complexité en temps est $\Omega(n^2)$, ce qui laisse une marge de progression énorme pour les chercheurs en informatique.

Mémorisation paresseuse. On peut se demander quelle serait la complexité d'une implémentation, avec une mémorisation paresseuse, disons à l'aide d'une liste chaînée, de la version récursive vue page 64.

Le nombre d'appels différents est, on l'a vu 64, est $k = O(n \cdot 2^n)$. La complexité, sans les appels récursifs, est $O(n)$. [Question. Pourquoi?] L'insertion dans une liste chaînée est linéaire, soit $O(k)$.

Pour résumer, on a donc k appels différents qui vont être insérés en un temps total $O(k^2) = O(n^2 \cdot 2^{2n})$. Au final cela fait $O(n^3 \cdot 2^{2n})$. Bien que plus rapide que l'approche naïve en $n!$, c'est bien moins efficace que le remplissage direct de la table `D[t][S]`.

3.4 Approximation

Le meilleur algorithme qui résout le **VOYAGEUR DE COMMERCE** prend un temps exponentielle en le nombre de points. Si on a besoin d'aller plus vite pour traiter de plus grandes instances, disons de $n \gg 100$ points, alors on doit abandonner la minimalité de la longueur de la tournée.

Une façon de calculer rapidement une tournée est par exemple d'utiliser l'algorithme dit « du plus proche voisin » : on part d'un point quelconque et à chaque étape on ajoute au chemin courant le point libre le plus proche du dernier point atteint. Une

fois le dernier point atteint on revient au point initial. La figure 3.7 illustre l'exécution d'une telle construction.

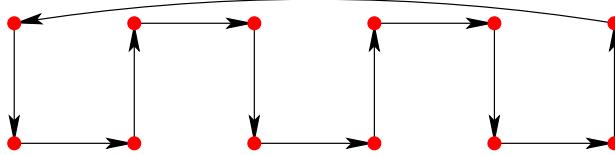


FIGURE 3.7 – Tournée produite par l'algorithme du plus proche voisin (en noir) qui est près de 50% plus longue que la tournée optimale. Si V est constitué de k carrés disjoints de côté 1 (ici $k = 3$), la tournée produite est de longueur $6k - 2$ alors que l'optimale est de $4k$.

Comme on peut le voir, le résultat ne donne pas nécessairement la tournée de longueur minimum. En contrepartie l'algorithme est très rapide. En effet, chaque point ajouté nécessite de comparer $O(n)$ distances, soit en tout une complexité en temps de $O(n^2)$. On pourrait construire encore plus rapidement une tournée. Par exemple en construisant aléatoirement la tournée, ce qui prend un temps optimal de $\Theta(n)$. [Question. Pourquoi est-ce optimal en temps?] Mais la longueur pourrait être $n/2$ fois plus longue, puisque la distance entre deux points est $\leq \text{OPT}(V, d)/2$ [Question. Pourquoi?], ce qui peut arriver comme illustrer sur la figure 3.8. Souvent on souhaite trouver un

– ALTERNANCE DE POINTS À DROIT POINTS À GAUCHE –

FIGURE 3.8 – Exemple de tournée aléatoire qui peut être presque deux $n/2$ fois plus longue que l'optimale.

compromis entre la qualité de la solution et le temps de calcul.

3.4.1 Algorithme glouton : un principe général

L'algorithme du plus proche voisin est aussi appelé l'algorithme *glouton* qui est en fait une méthode assez générale qui peut s'appliquer à d'autres problèmes.

L'algorithme glouton (*greedy* en Anglais) est une stratégie algorithmique qui consiste à former une solution en prenant à chaque étape le meilleur choix sans faire de *backtracking*, c'est-à-dire sans jamais remettre en cause les choix précédents.

Ce n'est pas une définition très précise, d'ailleurs il n'y en a pas. C'est une sorte de méta-heuristique qui peut se décliner en heuristiques le plus souvent très simples pour de nombreux problèmes.

Par exemple, pour les problèmes de type *bin packing*, qui consiste à ranger des objets pour remplir le mieux possible une boîte de capacité donnée, l'algorithme glouton se

traduit par l'application de la simple règle : « essayer de ranger en priorité les objets les plus gros. »

L'algorithme de Kruskal, pour calculer un arbre couvrant de poids minimum, est issu de la même stratégie : essayer d'ajouter en priorité les arêtes de plus petit poids. Cette stratégie est optimale pour l'arbre de poids minimum, pas pour *bin packing*.

Pour le **VOYAGEUR DE COMMERCE** la stratégie gloutonne consiste à construire la tournée en ajoutant à chaque fois le points qui minimise la longueur de la tournée courante, ce qui revient à prendre à chaque fois, parmi les points restant, celui le plus proche du dernier point sélectionné. C'est donc exactement l'algorithme du plus proche voisin discuté précédemment.

3.4.2 Problème d'optimisation

Les problèmes d'optimisations sont soient des minimisations (comme le **VOYAGEUR DE COMMERCE**) ou des maximisations (comme chercher le plus long chemin dans un graphe).

Pour une instance I d'un problème d'optimisation Π , on notera

- $\text{OPT}_{\Pi}(I)$ la valeur de la solution optimale pour l'instance I ; et
- $A(I)$ la valeur de la solution produite par l'algorithme A sur l'instance I .

Parfois on notera $\text{OPT}(I)$ ou même simplement OPT lorsque Π et I sont claires d'après le contexte. Pour simplifier, on supposera toujours que le problème Π est à valeurs positives⁴, c'est-à-dire que $\text{OPT}_{\Pi}(I) \geq 0$ pour toute instance I .

Un algorithme d'approximation a donc pour vocation de produire une solution de valeur « relativement proche » de l'optimal, notion que l'on définit maintenant⁵.

Définition 3.1 Une α -approximation pour un problème d'optimisation Π est un algorithme polynomial A qui donne une solution pour toute instance $I \in \Pi$ telle que :

- $A(I) \leq \alpha \cdot \text{OPT}_{\Pi}(I)$ dans le cas d'une minimisation ; et
- $A(I) \geq \alpha \cdot \text{OPT}_{\Pi}(I)$ dans le cas d'une maximisation.

La valeur α est le facteur d'approximation de l'algorithme A .

Parenthèse. Dans la définition précédente, on a écrit « algorithme polynomial » au lieu d'« al-

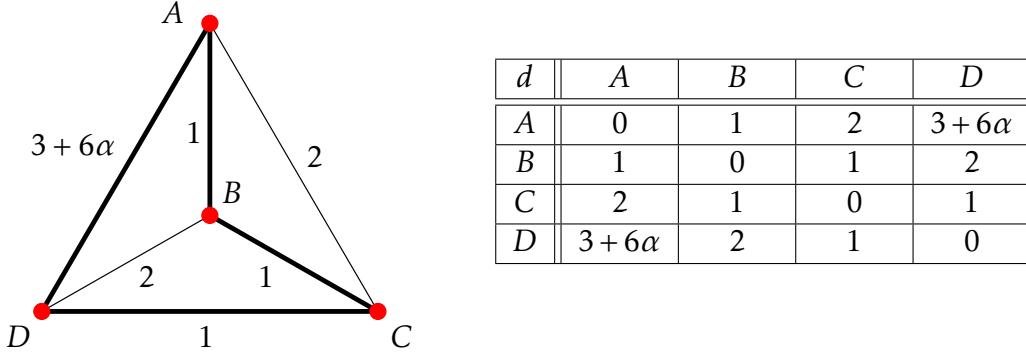
4. Sinon on peut toujours s'intéresser au problème similaire renvoyant la valeur opposée, la valeur absolue ou une translation de la valeur, quitte à transformer une maximisation en minimisation (ou le contraire).

5. La définition peut varier suivant le sens précis que l'on veut donner à « relativement proche ». Parfois on souhaite des approximations à un facteur additif près plutôt que multiplicatif. Parfois, on impose le facteur d'approximation seulement pour les instances suffisamment grandes, puisque pour les très petites, un algorithme exponentiel peut en venir à bout en un temps raisonnable (en fait en temps constant si la taille était constante).

gorithme de complexité en temps polynomial ». C'est un raccourci pour dire les deux : les complexités en temps et en espace sont polynomiales. Si on se permet de ne pas préciser, c'est parce que la complexité en temps est toujours plus grande que la complexité en espace. En effet, en temps t on ne peut jamais écrire que t mots mémoires. Donc imposer une complexité en temps polynomiale revient à imposer aussi une complexité en espace polynomiale.

Dans le cas d'une minimisation $\alpha \geq 1$ car $\text{OPT}(I) \leq A(I) \leq \alpha \cdot \text{OPT}(I)$, et pour une maximisation $\alpha \leq 1$ car $\alpha \cdot \text{OPT}(I) \leq A(I) \leq \text{OPT}(I)$. Remarquons qu'une 1-approximation est un algorithme exact polynomial.

L'algorithme glouton (c'est-à-dire l'algorithme du point le plus proche) est-il une approximation pour une certaine constante α ? À cause du contre exemple présenté sur la figure 3.7, on sait qu'il faut $\alpha \geq 1.5$. Mais *quid* de $\alpha = 1.6$ soit une garantie de 60% au-delà de l'optimal? Et bien non! Et pour le prouver on va montrer que ce n'est pas une α -approximation pour tout facteur $\alpha \geq 1$ donné. Considérons l'ensemble des points suivants $V = \{A, B, C, D\}$ ainsi que les distances données par la table :



La tournée produite par l'algorithme glouton à partir de A est

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$$

qui a pour longueur $\text{GREEDY}(V, d) = 6 + 6\alpha$. Il est facile de vérifier que la tournée optimale est $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$ qui a pour longueur $\text{OPT}(V, d) = 6$. En effet, c'est la tournée qui utilise les 4 plus courtes arêtes.

Le facteur d'approximation de l'algorithme glouton est donc au moins

$$\frac{\text{GREEDY}(V, d)}{\text{OPT}(V, d)} = \frac{6 + 6\alpha}{6} = 1 + \alpha.$$

Donc l'algorithme glouton n'est pas une α -approximation. On parle alors plutôt d'heuristique.

De manière générale, on nomme *heuristique* tout algorithme supposé efficace en pratique qui produit un résultat mais sans garantie sur la qualité par rapport à la solution optimale.

Parfois une heuristique peut être un algorithme d'approximation « qui s'ignore » : l'algorithme peut réellement avoir un facteur d'approximation constant, seulement on ne sait pas le démontrer ... Il peut aussi arriver qu'une heuristique ne soit même pas de complexité polynomiale (dans le pire des cas), mais très rapide en pratique.

Même sans garantie, une heuristique peut se révéler très efficace en pratique. C'est d'ailleurs pourquoi elles sont utiles et développées. Pour résumer, une heuristique est inutile en théorie mais bien utile en pratique, enfin si elle est « bonne ». Mais en l'absence de facteur d'approximation, on est bien évidemment un peu embêté pour donner un critère objectif pour comparer les heuristiques entre-elles ... En fait, il existe une mesure pour comparer les heuristiques : le *nombre de domination*. C'est le nombre de solutions dominées par celles produites par l'heuristique dans le pire des cas, une solution dominant une autre si elle est meilleure ou égale. Un algorithme exact a un nombre de domination maximum, soit⁶ $(n - 1)!/2$ pour le TSP, puisqu'il domine toutes les solutions. Il a été montré dans [GYZ02], que l'algorithme glouton a un nombre de domination de 1, pour chaque $n > 1$. Il arrive donc que l'heuristique produise la pire des tournées.

Dans l'exemple à $n = 4$ points qui fait échouer l'algorithme glouton, on remarquera que la fonction d est symétrique mais qu'une des distances (= arêtes du K_4) ne vérifie pas l'inégalité triangulaire. Plus précisément $d(A, D) > d(A, B) + d(B, D)$ dès que $\alpha > 0$. On peut se poser la question si l'algorithme glouton n'aurait pas un facteur d'approximation constant dans le cas métrique (avec inégalité triangulaire donc). Malheureusement, il a été montré en 2015 dans [BH15] que l'algorithme glouton, même dans le cas euclidien, a un facteur d'approximation de $\Omega(\log n)$, et que plus généralement pour le cas métrique est était toujours en $O(\log n)$.

Parenthèse. Notez bien l'usage de $\Omega(\log n)$ et de $O(\log n)$ de la dernière phrase. Elle signifie qu'il existe des ensembles de n points du plan pour lesquels le facteur d'approximation de l'algorithme glouton est au moins $c \log_2 n$, pour une certaine constante $c > 0$ et pour n assez grand. Mais qu'aussi, pour toute instance du TSP métrique (qui inclut le cas euclidien), le facteur d'approximation du même algorithme ne dépasse jamais $c' \log_2 n$ pour une certaine constante $c' \geq c$ et pour un nombre de points n assez grand. L'usage des notations asymptotiques permet ainsi de résumer fortement les énoncés lorsqu'elles sont correctement utilisées. Notez au passage qu'il est inutile de préciser la base du logarithme dans la notation $O(\log n)$ car $\log_b n = \log n / \log b$. Donc $\log_2 n$, $\log_{10} n$ ou $\ln n$ sont identiques à une constante multiplicative près. Traditionnellement on utilise $O(\log n)$ plutôt que $O(\ln n)$. Voir aussi le paragraphe 1.6.

6. Il y a $n!$ permutations possibles. Mais pour chaque permutation, il y n points de départs possibles et deux sens de parcours, chacun de ses choix produisant une tournée équivalente.

3.4.3 Autres heuristiques

Il existe de nombreuses autres heuristiques. Une famille parmi elles est appelées *optimisations locales*. On part d'une solution (une tournée), et on cherche dans son proche « voisinage » (par exemple en échangeant quelques arêtes) s'il n'y a pas une meilleure solution (plus courte donc). Et on recommence tant qu'il y a un gain. C'est la base des *méthodes de gradient* pour l'optimisation de fonction qu'on ne traitera pas ici.

Pour le **VOYAGEUR DE COMMERCE** cela correspond aux heuristiques 2-OPT ou 3-OPT qui consistent à flipper⁷ deux ou trois arêtes. Aussi étrange que cela puisse paraître le temps de convergence vers l'optimal local peut prendre un nombre d'étapes exponentielles⁸ même dans le cas de la distance Euclidienne [ERV07]. Cependant sur des points aléatoires du carré unité $[0, 1]^2$ on observe un nombre moyen de flips effectués de l'ordre de $O(n \log n)$ et une tournée calculée de longueur entre 4% et 7% plus longue que la tournée optimale. En fait il a été prouvé que cet excès moyen est borné par une constante, ce qui n'est pas vrai dans le cas général. On peut aussi prouver que le nombre moyen de flips est polynomial⁹. Il peut arriver que l'heuristique 2-OPT soit plus longue d'un facteur $\Omega(\log n / \log \log n)$ sur des instances Euclidiennes particulières [CKT99] et même d'un facteur $\Theta(\sqrt{n})$ pour des instances vérifiant seulement l'inégalités triangulaires (voir [CKT99]). Ceci est valable uniquement si l'adversaire peut choisir la (mauvaise) tournée de départ. Évidemment, si la tournée de départ est déjà une 2-approximation comme vue précédemment, la tournée résultante par 2-OPT ne pourra être que plus courte. L'heuristique 2-OPT calculée à partir d'une tournée issue de l'algorithme glouton donne en pratique de très bon résultat.

Une autre heuristique est celle des « économies » ou encore l'heuristique *savings* de Clarke et Wright. On construit $n - 1$ tournées qui partent de v_0 et qui sont $v_0 - v_i - v_0$ pour tout $v_i \in V^*$. Puis, $n - 2$ fois on fusionne deux tournées $v_0 - v_{a_1} - \dots - v_{a_p} - v_0$ et $v_0 - v_{b_1} - \dots - v_{b_q} - v_0$ en une plus grande qui évite un passage par v_0 , soit $v_0 - v_{a_1} - \dots - v_{a_p} - v_{b_1} - \dots - v_{b_q} - v_0$. Par rapport au total des longueurs des tournées en cours, on économise $d(v_{a_p}, v_0) + d(v_0, v_{b_1}) - d(v_{a_p}, v_{b_1})$. On fusionne en priorité la paire de tournées qui économisent le plus de distance.

Une heuristique proche, dite d'« insertion aléatoire » (ou *random insertion*) donne aussi de très bon résultats, et est relativement rapide à calculer. On départ on considère une tournée avec un seul point choisi aléatoirement. Puis $n - 1$ fois on étend la tournée courante en insérant un point w choisi aléatoirement hors de la tournée à la place de l'arête $u - v$ qui minimise l'accroissement de distance $d(u, w) + d(w, v) - d(u, v)$. Une variante consiste à choisir w non pas aléatoirement mais comme celui qui minimise l'accroissement. C'est alors une 2-approximation — on se ramène au calcul de l'arbre de

7. Initialement introduite par Georges A. Croes en 1958.

8. Plus précisément $\Omega(2^{n/8})$.

9. Plus précisément, c'est au plus $O(n^{4+1/3} \log n)$ dans le cas Euclidien, mais on est pas encore capable de prouver si c'est moins de n^2 par exemple.

poids minimum par l’algorithme de Prim suivi d’un parcours en profondeur —, mais elle est $O(n)$ fois plus lente à calculer.

L’heuristique de Lin et Kernighan, plus complexe, est basée sur une généralisation des flips ou k -Opt. C’est elle qui permet d’obtenir les meilleurs résultats en pratique. Une très bonne heuristique (aussi rapide que 2-Opt) consiste à flipper trois arêtes dont deux sont consécutives. On parle de 2.5-flip.

Une heuristique bien connue lorsque $V \subset \mathbb{R}^2$ consiste à calculer la tournée *bitonique* optimale, qui apparaît la première fois dans [CLRS01][Édition 1990, page 354]. Une tournée est bitonique si elle part du point le plus à gauche (celui avec l’abscisse la plus petite), parcourt les points par abscisses croissantes jusqu’au point le plus à droite (celui avec l’abscisse la plus grande) et revient vers le point de départ en parcourant les points restant par abscisses décroissantes. Une autre définition équivalente est que toute droite verticale ne coupe la tournée en au plus deux points. La figure 3.9 présente un exemple. On peut montrer [*Exercice. Pourquoi?*] que la tournée bitonique optimale est sans croisement. De plus, c’est la tournée qui minimise la somme des déplacements verticaux. L’intérêt de cette notion est qu’il est possible de calculer la tournée bitonique optimale en temps $O(n^2)$, et ce par programmation dynamique.

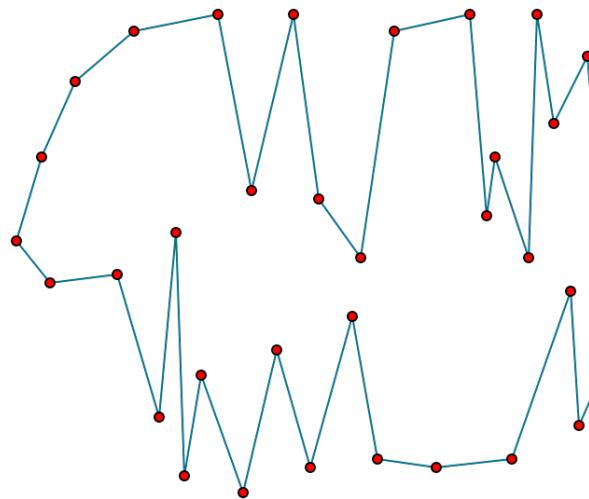


FIGURE 3.9 – Tournée bitonique optimale. Source Wikipédia.

Bien que très efficace, aucune de ces heuristiques ne permet en toute généralité de garantir un facteur d’approximation constant. On pourra se référer à l’étude complète sur le VOYAGEUR DE COMMERCE par [JM97].

3.4.4 Inapproximabilité

Non seulement le problème du VOYAGEUR DE COMMERCE est difficile à résoudre, mais en plus il est difficile à approximer. On va voir en effet qu’aucun algorithme polyno-

mial¹⁰ (et donc pas seulement l'algorithme glouton !) ne peut approximer à un facteur constant le problème du VOYAGEUR DE COMMERCE dans toute sa généralité, sauf si les classes de complexité P et NP sont égales¹¹, problème notablement difficile à 1 M\$.

Pour le voir on peut transformer une instance d'un problème réputé difficile en une instance du VOYAGEUR DE COMMERCE de sorte que la tournée optimale (et même une approximation) donne une solution au problème difficile initial. Cela s'appelle une *réduction*. On va utiliser un problème de la classe NP-complet. Ce sont des problèmes de NP qui sont réputés difficiles : on ne connaît pas d'algorithme de complexité polynomiale mais on arrive pas à démontrer qu'il n'y en a pas. Donc sauf si P=NP, on ne peut pas trouver un algorithme efficace pour le VOYAGEUR DE COMMERCE car sinon il permettrait de résoudre un problème réputé difficile.

Le problème difficile (NP-complet) que l'on va considérer est celui du CYCLE HAMILTONIEN, un problème proche du problème CHEMIN HAMILTONIEN rencontré au paragraphe 1.5.3, consistant à déterminer si le graphe possède un cycle, dit Hamiltonien, passant une et une seule fois par chacun de ses sommets.

On va construire une instance du VOYAGEUR DE COMMERCE à partir d'un graphe H donné à n sommets. L'ensemble des points est $V_H = V(H)$ et la distance d_H définie par (voir la figure 3.10) :

$$d_H(v_i, v_j) = \begin{cases} 1 & \text{si } v_i v_j \in E(H) \\ n^2 & \text{sinon} \end{cases}$$

La paire (V_H, d_H) est une instance particulière du VOYAGEUR DE COMMERCE.

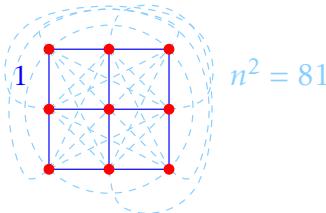


FIGURE 3.10 – Construction d'une instance (V_H, d_H) pour l'approximation du VOYAGEUR DE COMMERCE à partir d'une instance H du problème CYCLE HAMILTONIEN qui est NP-complet. Ici H est une grille 3×3 . Les arêtes de H sont valuées 1, les autres n^2 .

Considérons une α -approximation du VOYAGEUR DE COMMERCE, un algorithme noté A , où α est une constante $< n$. Il est assez clair que¹² $A(V_H, d_H) < n^2$ si et seulement si H possède un cycle Hamiltonien.

10. C'est-à-dire de complexité en temps et/ou en espace polynomial.

11. P est l'ensemble des problèmes possédant un algorithme déterministe de complexité en temps polynomial, alors que NP est celui des problèmes possédant un algorithme non-déterministe de complexité en temps polynomial.

12. On rappelle que $A(I)$ est la valeur de la solution pour l'instance I renvoyée par l'algorithme A , ici une longueur de tournée, cf. le paragraphe 3.4.2.

En effet, si H possède un cycle Hamiltonien, l'algorithme d'approximation devra renvoyer une tournée de longueur au plus $\alpha n < n^2$ puisque la longueur optimale est dans ce cas n . Et si H ne l'est pas, la tournée renvoyée par A contiendra au moins une paire de points visités consécutivement v_i, v_{i+1} ne correspondant pas à une arête de H , donc avec $d_H(v_i, v_{i+1}) = n^2 > \alpha n$.

On a donc réduit le problème du **VOYAGEUR DE COMMERCE** à celui du **CYCLE HAMILTONIEN**. Or ce dernier est réputé difficile : il ne possède pas d'algorithme polynomial, sauf si P=NP. Il suit que l'algorithme A n'est pas une α -approximation : soit A n'est pas polynomial, soit le facteur d'approximation est $> \alpha$. Le problème du **VOYAGEUR DE COMMERCE** est inapproximable.

Parenthèse. *Le problème CYCLE HAMILTONIEN est difficile même si le graphe est le sous-graphe d'une grille, c'est-à-dire obtenu par la suppression de sommets et/ou d'arêtes d'un grille. Cependant il est polynomial si la sous-grille n'a pas de trous, c'est-à-dire que les sommets qui ne sont pas sur le bord de la face extérieur sont tous de degré 4 (cf. [UL97]).*

Le problème CYCLE HAMILTONIEN est aussi difficile que le problème CHEMIN HAMILTONIEN. On peut le voir grâce à un autre type de réduction. Soit H le graphe dont on veut savoir s'il possède un chemin Hamiltonien. Pour chaque paire de sommets $\{u, v\}$ de H on crée un graphe H_{uv} obtenu en ajoutant l'arête $u - v$ à H et en testant si $H_{u,v}$ possède un cycle Hamiltonien. Si on obtient une réponse négative pour toutes les paires, c'est que H n'a pas de chemin Hamiltonien [Question. Pourquoi?]. Si on obtient une réponse positive pour un certain $H_{u,v}$, alors H contient un chemin Hamiltonien que l'on peut obtenir à partir du cycle dans H_{uv} et en supprimant l'arête $u - v$ ou alors n'importe quelle autre du cycle si $u - v$ n'était pas dans le cycle. Donc si on avait un algorithme polynomial pour le cas du cycle, on en aurait un aussi pour le cas du chemin (un peu plus lent, certes, mais encore polynomial). Donc le cas du cycle ne peut pas être plus facile que celui du chemin.

On pourrait arguer que la réduction précédente produit une instance du **VOYAGEUR DE COMMERCE** qui ne satisfait pas l'inégalité triangulaire. Cela ne prouve en rien, par exemple, qu'il n'y a pas d'algorithme efficace dans le cas du TSP métrique. En fait, il a été démontré dans [Kar15] que le TSP métrique ne peut être approché (en temps polynomial) à un facteur moins de $1 + 1/122$, sauf si les classes P et NP sont confondues, ce qui est $< 1\%$ de l'optimal. À titre de comparaison, le meilleur algorithme d'approximation connu pour le TSP métrique a un facteur d'approximation de 1.5, soit 50% de l'optimal. Cela laisse donc une grande marge d'amélioration. Des résultats d'inapproximabilité sont aussi donnés dans [Kar15] pour les variantes TSP asymétrique et TSP graphique mais qui restent en dessous des 2%.

3.4.5 Cas euclidien

Lorsque les points sont pris dans un espace euclidien de dimension δ , c'est-à-dire $V \subset \mathbb{R}^\delta$, et que d correspond à la distance euclidienne, alors il existe un algorithme d'approximation réalisant le compromis temps vs. approximation suivant :

Théorème 3.1 ([Aro98][Mit99]) Pour tout $\varepsilon > 0$, il existe une $(1 + \varepsilon)$ -approximation pour le problème du VOYAGEUR DE COMMERCE qui a pour complexité en temps

$$n \cdot (\log n)^{O\left(\frac{1}{\varepsilon} \sqrt{\delta}\right)^{\delta-1}}.$$

On parle parfois de schéma d'approximation polynomial, car le facteur d'approximation $1 + \varepsilon$ peut-être choisi arbitrairement proche de 1 tout en gardant un temps polynomial, ε et δ étant ici des constantes.

Notons que, par exemple, pour $\delta = 2$ et $\varepsilon = 0.1$ (soit le plan avec au plus 10% de l'optimal), la complexité en temps est de seulement $n \cdot (\log n)^{O(1)}$ ce qui est moins que le nombre de distances soit $\Theta(n^2)$. [Question. Pourquoi?] Ce résultat a valu à Arora et Mitchell le **Prix Gödel** en 2010. L'algorithme est réputé très difficile à implémenter, et en pratique on continue à utiliser des heuristiques.

3.4.6 Une 2-approximation

On va montrer que l'algorithme suivant est une 2-approximation. Il est plus général que l'algorithme d'Arora-Mitchell car il s'applique non seulement au cas de la distance euclidienne, mais aussi à toute fonction d vérifiant l'inégalité triangulaire. Il est aussi très simple à implémenter.

En Anglais « arbre couvrant de poids minimum » se dit *Minimum Spanning Tree* (MST). Rappelons qu'il s'agit de trouver un arbre couvrant dont la somme des poids de ses arêtes est la plus petite possible.

Algorithme APPROXMST(V, d)

Entrée: Une instance (V, d) du VOYAGEUR DE COMMERCE.

Sortie: Une tournée, c'est-à-dire un ordre sur les points de V .

1. Calculer un arbre couvrant de poids minimum T sur le graphe complet défini par V et les arêtes valuées par d .
 2. La tournée est définie par l'ordre de visite des sommets selon un parcours en profondeur d'abord de T .
-

Exemple. Voir la figure 3.11.

Il est clair que l'algorithme APPROXMST renvoie une tournée puisque dans l'ordre de première visite les points sont précisément visités une et une seule fois. Pour montrer que l'algorithme est une 2-approximation, il nous faut démontrer deux points :

1. sa complexité est polynomiale ; et
2. son facteur d'approximation est au plus 2.

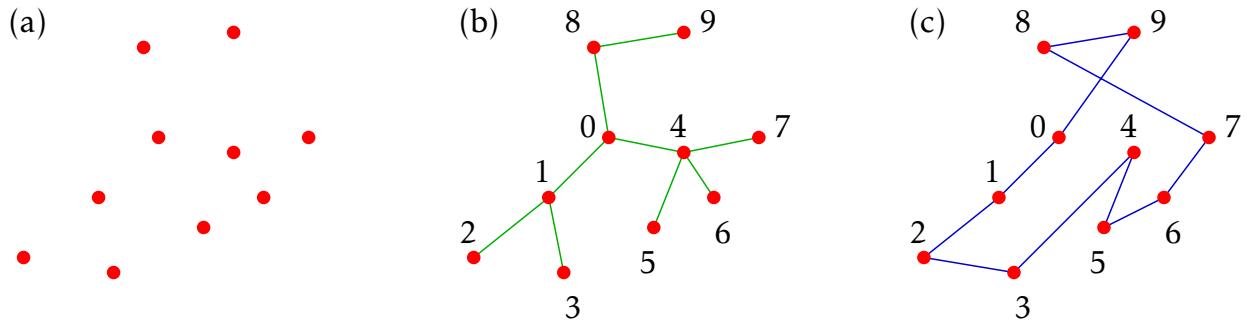


FIGURE 3.11 – (a) Ensemble de points sur lequel est appliqué APPROXMST ; (b) L’arbre couvrant de poids minimum et un parcours en profondeur ; (c) La tournée correspondante.

Complexité. Calculer un arbre couvrant de poids minimum prend un temps de $O(m \log n)$ pour un graphe ayant m arêtes et n sommets. Avec Prim (et une bonne structure de données) c’est $O(n^2)$ et pour Kruskal c’est $O(m \log n)$ que l’on peut faire simplement. Le graphe est complet, donc $m = n(n-1)/2 = O(n^2)$ arêtes. Donc la première étape prend un temps $O(n^2)$ avec Prim ou $O(n^2 \log n)$ avec Kruskal.

Le parcours en profondeur prend un temps linéaire en le nombre de sommets et d’arêtes du graphe. Ici le graphe est un arbre sur n sommets, et donc $n - 1$ arêtes. Cette étape prend donc un temps $O(n)$.

On a donc montrer :

Proposition 3.1 *L’algorithme APPROXMST a pour complexité $O(n^2)$.*

Parenthèse. En fait il existe des algorithmes plus efficaces que Prim et Kruskal pour calculer un arbre couvrant de poids minimum pour un graphe à n sommets et m arêtes. Les plus rapides, dus à [Cha00] et [Pet99], ont une complexité en $O(n + m \cdot \alpha(m, n))$ où $\alpha(m, n)$ est la fonction inverse d’Ackermann¹³. Cette fonction croît extrêmement lentement. Pour toutes valeurs raisonnables de m et n (disons inférieures au nombre de particules de l’Univers), $\alpha(m, n) \leq 4$. Plus précisément¹⁴, $\alpha(m, n) = \min\{i : A(i, \lceil m/n \rceil) > \log_2 n\}$ où $A(i, j)$ est la fonction d’Ackermann définie par :

- $A(1, j) = 2^j$, pour tout $j \geq 1$;
- $A(i, 1) = A(i-1, 2)$, pour tout $i > 1$;
- $A(i, j) = A(i-1, A(i, j-1))$, pour tout $i, j > 1$.

L’algorithme résultant est réputé pour être terriblement compliqué. Il existe aussi des algorithmes probabilistes dont le temps moyen est $O(m + n)$.

Classiquement, la fonction d’Ackermann est plutôt définie ainsi :

13. Voir aussi [ici](#) pour une définition alternative plus simple.

14. Il y a parfois des variantes dans les définitions de $\alpha(m, n)$: $\lfloor m/n \rfloor$ au lieu de $\lceil m/n \rceil$, ou encore n au lieu de $\log_2 n$. Ces variantes simplifient souvent les démonstrations, c’est-à-dire les calculs, mais au final toutes les définitions restent équivalentes à un terme additif près.

- $A(0, j) = j + 1$, pour tout $j \geq 0$;
- $A(i, 0) = A(i - 1, 1)$, pour tout $i > 0$;
- $A(i, j) = A(i - 1, A(i, j - 1))$, pour tout $i, j > 0$.

On a alors :

- $A(1, j) = 2 + (j + 3) - 3$
- $A(2, j) = 2 \times (j + 3) - 3$
- $A(3, j) = 2^{\wedge} (j + 3) - 3$
- $A(4, j) = 2^{\wedge} \dots^{\wedge} 2 - 3$ avec $n + 3$ deux empilés
- ...

La variante sur la fonction présentée au-dessus évite un terme additif -3 de la version classique de $A(i, j)$.

Facteur d'approximation. C'est le point difficile en général. Il faut relier la longueur de la tournée optimale à la tournée construite par l'algorithme.

Rappelons que le *poids* d'un graphe arête-valué (G, ω) est la valeur notée $\omega(G) = \sum_{e \in E(G)} \omega(e)$, c'est-à-dire la somme des poids de ses arêtes.

Proposition 3.2 La longueur de la tournée optimale pour l'instance (V, d) est plus grande que le poids de l'arbre de poids minimum couvrant V . Dit autrement, $\text{OPT}(V, d) > d(T)$.

Preuve. Soit T l'arbre couvrant de poids minimum calculé à l'étape 1 de l'algorithme. Le poids de T , vu comme un graphe arête-valué (T, d) , vaut $d(T)$ puisque le poids de chaque arête de T est la distance donnée par d entre ses extrémités.

À partir de n'importe quelle tournée pour (V, d) , on peut former un cycle arête-valué (C, d) dont le poids $d(C)$ correspond précisément à la longueur de la tournée (la somme des poids des arêtes qui vaut la distance d ici). Si on supprime une arête e quelconque de C , alors on obtient un arbre couvrant, $C \setminus \{e\}$ n'ayant pas de cycle et couvrant toujours tous les sommets. En particulier, pour le cycle C^* correspondant à la tournée optimale, il suit que :

$$d(C^*) > d(C^* \setminus \{e\}) = d(C^*) - d(e) = \text{OPT}(V, d) - d(e) \geq d(T)$$

puisque T est un arbre de poids minimum et que $d(C^*) = \text{OPT}(V, d)$. On a donc montré que $\text{OPT}(V, d) \geq d(T) + d(e) > d(T)$.

Parenthèse. On peut raffiner un peu plus cette inégalité et donner une meilleure borne inférieure sur $\text{OPT}(V, d)$, ce qui est toujours intéressant pour évaluer les performances des heuristiques :

$$\text{OPT}(V, d) \geq d(T) + d(e^+) \tag{3.3}$$

où e^+ est l'arête juste plus lourde que l'arête la plus lourde de T . Autrement dit, dans l'ordre croissant des arêtes du graphe (ici une clique), si e_i était la dernière arête ajoutée à T , alors

$e^+ = e_{i+1}$. En effet, lorsqu'on forme un arbre à partir de C^* , plutôt que de choisir n'importe qu'elle arête e , on peut choisir d'enlever l'arête e^* la plus lourde de C^* . L'arête e^* ne peut être dans le MST [Question. Pourquoi?]. Elle est aussi forcément plus lourde que la plus lourde du MST, c'est-à-dire $d(e^*) \geq d(e^+)$. [Question. Pourquoi?] On a vu (en posant $e = e^*$) que $\text{OPT}(V, d) - d(e^*) \geq d(T)$, ce qui implique $\text{OPT}(V, d) \geq d(T) + d(e^*) \geq d(T) + d(e^+)$ et prouve l'équation 3.3. Évidemment $d(T) + d(e^*)$ est une meilleure borne inférieure, mais e^* est par essence difficile à calculer (il faudrait connaître C^*), contrairement à e^+ .

□

Il reste à majorer la longueur de la tournée renvoyée par l'algorithme en fonction de $d(T)$. Pour cela, on a besoin de l'inégalité triangulaire.

Proposition 3.3 *La tournée obtenue par le parcours de T est de longueur au plus $2 \cdot d(T)$. Dit autrement, $\text{APPROXMST}(V, d) \leq 2 \cdot d(T)$.*

Preuve. Soit $v_0 - v_1 - \dots - v_{n-1} - v_0$ la tournée renvoyée par l'algorithme. Notons P_i le chemin dans T entre v_i et son suivant v_{i+1} (modulo n). L'inégalité triangulaire permet d'affirmer que la longueur du segment $v_i - v_{i+1}$ vaut au plus $d(P_i)$, le poids du chemin P_i . La longueur de la tournée renvoyée par l'algorithme est donc au plus $\sum_{i=0}^{n-1} d(P_i)$. Pour montrer que cette somme est au plus $2 \cdot d(T)$, il suffit d'observer que chaque arête e de T appartient à au plus deux chemins parmi P_0, \dots, P_{n-1} .

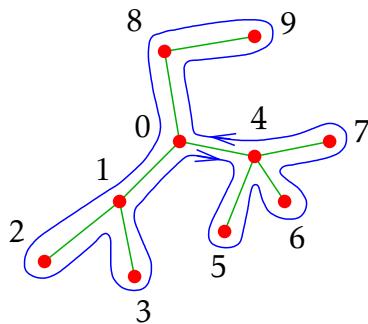


FIGURE 3.12 – Parcours de la face extérieure de l'arbre T , chaque arête étant parcourue exactement deux fois. L'arête $0 - 4$ appartient aux chemins P_3 et P_7 .

Pour le voir (cf. figure 3.12), on peut d'abord redessiner l'arbre T , éventuellement en réordonnant les fils autour de certains sommets, de sorte que parcours DFS de T corresponde à un parcours de sa face extérieure. Cela ne change évidemment pas la longueur des chemins P_i dans T . La suite des chemins P_0, \dots, P_{n-1} constitue alors comme un simple parcours de la face extérieure de T (les anglo-saxons parlent aussi de parcours eulérien¹⁵), chaque arête étant visitée en descendant (par exemple si P_i est

15. De manière générale pour un graphe G , c'est le plus petit cycle visitant chacune des arêtes de G , une arête pouvant être traversée plusieurs fois en présence de sommets de degré impair.

une arête $v_i - v_{i+1}$ de T) ou en remontant (si P_i part d'une feuille). \square

La combinaison des propositions 3.2 et 3.3 permet de conclure que la longueur de la tournée produite par l'algorithme est au plus deux fois l'optimale. En effet, on vient de voir que $\text{APPROXMST}(V, d) \leq 2 \cdot d(T)$ et que $d(T) < \text{OPT}(V, d)$. On en déduit donc que $\text{APPROXMST}(V, d) < 2 \cdot d(T)$. Avec la proposition 3.1, on a donc montré que :

Proposition 3.4 *L'algorithme APPROXMST est une 2-approximation.*

3.4.7 Union-and-Find

Bien que Prim soit plus rapide dans le cas où $m = \Theta(n^2)$, les détails de son implémentation sont plus nombreux que ceux de Kruskal. Nous allons détailler l'implémentation de l'algorithme de Kruskal, en particulier la partie permettant de savoir si une arête forme un cycle ou pas, et donc si elle doit être ajoutée à la forêt courante. Rappelons que dans cet algorithme (qui est glouton), on ajoute les arêtes par poids croissant, sans former de cycles, jusqu'à former un arbre couvrant.

Le problème général sous-jacent est de maintenir les composantes connexes d'un graphe (ici une forêt) qui au départ est composé de sommets isolés et qui croît progressivement par ajout d'arêtes (cf. figure 3.13).

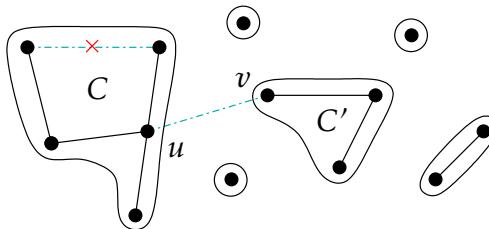


FIGURE 3.13 – Maintient des composantes connexes d'un graphe à l'aide d'une forêt couvrante : on ajoute une arête seulement si elle ne forme pas de cycle.

Pour cela on va résoudre un problème assez général de structure de données qui est le suivant. Dans ce problème il y a deux types d'objets : des *éléments* et des *ensembles*. Par rapport à notre problème de composantes connexes, les éléments sont les sommets et les ensembles les composantes connexes. L'objectif est de pouvoir réaliser le plus efficacement possibles les deux opérations suivantes (voir la figure 3.14 pour un exemple) :

- FUSIONNER deux ensembles donnés ;
- TROUVER l'ensemble contenant un élément donné.

Muni d'une telle structure de données, l'algorithme de Kruskal peut se résumer ainsi :

– EXEMPLE {a} {b} {c} {d} {e} {f} –

FIGURE 3.14 – Exemple de fusions d'ensembles d'éléments. Les ensembles seront codés par des arbres enracinés d'éléments.

Algorithme KRUSKAL(G)

Entrée: Un graphe arête-valué G .

Sortie: Un arbre couvrant de poids minimum.

1. Initialiser $T := (V(G), \emptyset)$.
 2. Pour chaque arête uv de G prise dans l'ordre croissant de leur poids :
 - (a) TROUVER la composante C de u et la composante C' de v ;
 - (b) si $C \neq C'$, ajouter uv à T et FUSIONNER C et C' .
 3. Renvoyer T .
-

La structure de données qui supporte ces opérations s'appelle *Union-and-Find*. Comme on va le voir, elle est particulièrement simple à mettre en œuvre et redoutablement efficace.

La structure de données représente chaque ensemble par un arbre enraciné, les nœuds étant les éléments de l'ensemble. L'ensemble est identifié par la racine de l'arbre. Donc trouver l'ensemble d'un élément revient en fait à trouver la racine de l'arbre le contenant.

On code un arbre enraciné par la relation de parenté, un tableau `parent[]`, avec la convention que `parent[x]=x` si `x` est la racine. On suppose qu'on a un total de n éléments qui sont, pour simplifier, des entiers (`int`) que l'on peut voir comme les indices des éléments. Au départ, tout le monde est racine : on a n éléments et n singltons.

```
// Initialisation Union-and-Find (v1)
int parent[n];
for(x=0; x<n; x++) parent[x]=x;
```

Pour trouver l'ensemble contenant un élément, on cherche la racine de l'arbre auquel il appartient. Pour la fusion de deux ensembles, identifiés par leur racine (disons `x` et `y`), on fait pointer une des deux racines vers l'autre (ici `y` pointe vers `x`). Ce qui donne, en supposant pour simplifier que `parent[]` est une variable globale :

```
// Union-and-Find (v1)
void Union(int x, int y){
    parent[y]=x;
}
int Find(int x){
    while(x!=parent[x]) x=parent[x];
    return x;
}
```

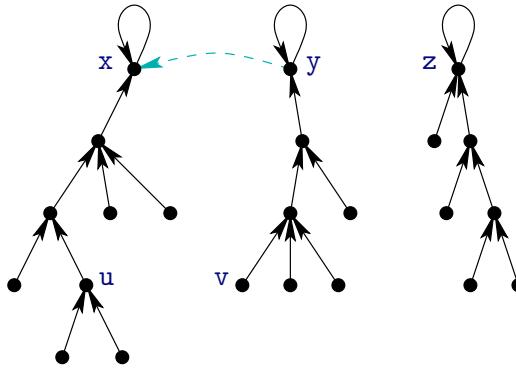


FIGURE 3.15 – Fusion des ensembles $x=Find(u)$ et $y=Find(v)$, tous deux représentés par des arbres enracinés, avec $Union(x,y)$.

Attention ! Pour le problème du maintien des composantes connexes d'un graphe, l'arbre enraciné qui représente l'ensemble n'a rien à voir avec la composante connexe elle-même (qui dans KRUSKAL se trouvent être aussi un arbre). Les arcs des arbres codant les ensembles ne correspondent pas forcément à des arêtes de la composante. Dans la figure 3.15, on va fusionner la composante de u avec celle de v à cause d'une arête $u-v$. On obtiendra une nouvelle composante représentée par un arbre ayant un arc entre x et y , mais sans arc entre u et v .

Notez bien qu'on fait la fusion d'ensembles et pas d'éléments. Par exemple pour fusionner l'ensemble contenant u avec l'ensemble contenant v , il faut d'abord chercher leurs racines respectives, ce qui se traduit par (cf. aussi la figure 3.15) :

```
Union(Find(u), Find(v));
```

En terme de complexité, `Union()` prend un temps constant, et `Find(x)` prend un temps proportionnel à la profondeur de x , donc au plus la hauteur de l'arbre. Malheureusement, la hauteur d'un arbre peut être h après avoir réalisé seulement h fusions comme le suggère l'exemple suivant :

```

Union(Find(x0),Find(x1)); // x1 profondeur 1
Union(Find(x1),Find(x2)); // x2 profondeur 2
Union(Find(x2),Find(x3)); // x3 profondeur 3
...

```

Pour un graphe à n sommets, cela aboutit à un arbre (en fait un chemin) de racine x_0 et de profondeur $n - 1$:

$$\circlearrowleft x_0 \leftarrow x_1 \leftarrow x_2 \leftarrow x_3 \leftarrow \cdots \leftarrow x_{n-1}$$

Il n'est pas très difficile de voir que le temps cumulé de ces $n - 1$ opérations `Union()`, chacune d'elles comprenant deux `Find()`, est de l'ordre de n^2 , ce qui n'est pas très efficace. On va faire beaucoup mieux grâce aux deux optimisations suivantes.

Première optimisation. Cette optimisation, dite du *rang*, consiste à fusionner le plus petit arbre avec le plus grand (en terme de hauteur). L'idée est que si on rattache un arbre peu profond à la racine du plus profond, alors la hauteur du nouvel arbre ne changera pas (cf. figure 3.16). Elle ne changera que si les arbres sont de même hauteur. Pour cela on ajoute donc un simple tableau `height[]` permettant de gérer la hauteur de chacun des arbres qu'il va falloir maintenir.

```

// Initialisation Union-and-Find (v2)
int parent[n], height[n];
for(x=0; x<n; x++) parent[x]=x, height[x]=0;

```

Parenthèse. De manière générale en algorithmique, l'augmentation de l'espace de travail peut permettre un gain de temps. C'est par exemple le cas en programmation dynamique. Cela a cependant un coût : celui de la mise à jour des informations lors de chaque opération. Et c'est nécessaire ! En effet, s'il n'y avait pas besoin de mise à jour, c'est que l'information n'était pas vraiment utile. Pour les structures de données, il y a donc un compromis entre le temps de requête (c'est-à-dire la complexité en temps des opérations supportées par la structure de données), la taille de la structure de données et le temps de mise à jour qui inévitablement va s'allonger en fonction de la taille.

```
// Union (v2)
void Union(int x, int y){
    if(height[x]>height[y]) parent[y]=x; // y → x
    else{
        parent[x]=y; // x → y
        if(height[x]==height[y]) height[y]++;
    }
}
```

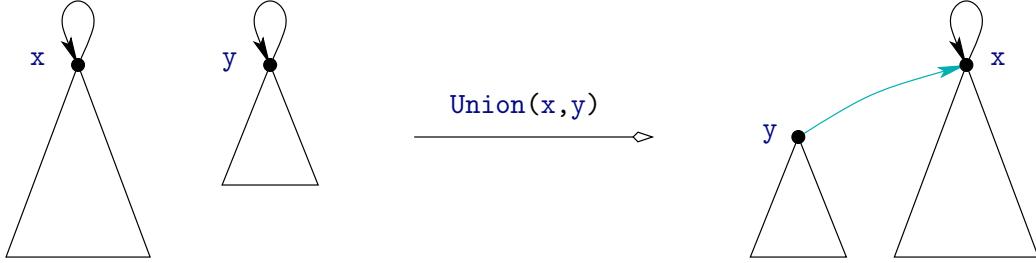


FIGURE 3.16 – Opération `Union(x,y)` avec optimisation du rang : c'est l'arbre le plus petit qui se raccroche au plus grand.

Notez que ce qui nous intéresse ce n'est pas la hauteur de chacun des sommets, mais seulement des arbres (ici les racines) ce qui permet une mise à jour plus rapide. La complexité de l'opération `Union()`, bien que plus lente, est toujours constante. Le gain pour `Find()` est cependant substantiel.

Proposition 3.5 *Tout arbre de profondeur h possède au moins 2^h éléments.*

Preuve. Par induction sur h . Pour $h = 0$, c'est évident, chaque arbre contenant au moins $1 = 2^0$ élément. Supposons vraie la propriété pour tous les arbres de hauteur h . On obtient un arbre de profondeur $h + 1$ seulement dans le cas où l'arbre est obtenu par la fusion de deux arbres de profondeurs h . Le nouvel arbre contient, par hypothèse, au moins $2^h + 2^h = 2^{h+1}$ éléments. \square

Il est clair qu'un arbre possède au plus n éléments. Donc si un arbre de hauteur h possède k éléments, alors on aura bien évidemment $2^h \leq k \leq n$ ce qui implique $h \leq \log_2 n$. Donc chaque arbre a une hauteur $O(\log n)$ ce qui fait que la complexité de `Find()` est $O(\log n)$. C'est un gain exponentielle par rapport à la version précédente !

Deuxième optimisation. La seconde optimisation, appelée *compression de chemin*, est basée sur l'observation qu'avant de faire `Union()` on fait un `Find()` (en fait deux). Lors du

`Find()` sur un élément u qui a pour effet de parcourir le chemin de u vers sa racine, on en profite pour connecter directement à la racine chaque élément du chemin parcouru. C'est comme si le chemin de u à sa racine avait été compressé en une seul arête ramenant tous les sous-arbres accrochés à ce chemin comme fils de la racine. Voir la figure 3.17.

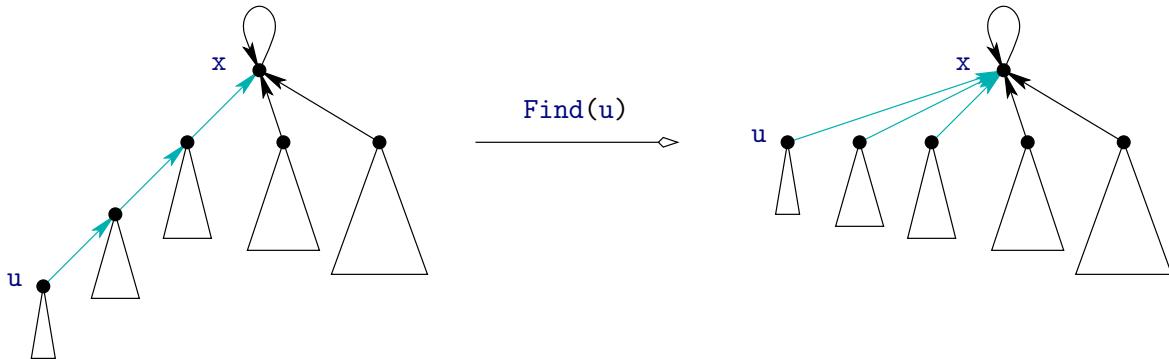


FIGURE 3.17 – Opération `Find(u)` avec compression de chemin.

Cela n'affecte pas la complexité de l'opération de `Find(u)`, il s'agit d'un parcours que l'on effectue de toute façon, mais va affecter significativement la complexité des `Find()` ultérieurs puisque l'arbre contenant u a été fortement raccourci. Voici le code (final) :

```
// Find (v2)
int Find(int x){
    if(x!=parent[x]) parent[x]=Find(parent[x]);
    return parent[x];
}
```

On remarque que `height[x]` n'est pas modifié par `Find()`. Ce n'est donc plus forcément la hauteur de l'arbre enraciné en x , mais est simplement un majorant. Ce n'est en fait pas gênant. Ce qui compte c'est que les `Find()` aient un coût faible. Avec cette optimisation, il devient alors très difficile d'avoir des arbres de grande profondeur. Car pour qu'un élément soit profond, il faut l'avoir ajouté, donc avoir fait un `Find()` sur cet élément, ce qui raccourcit l'arbre. On peut montrer :

Proposition 3.6 *Lorsque les deux optimisations « rang » et « compression de chemin » sont réalisées, la complexité de m opérations de fusion et/ou de recherche sur n éléments est de $O(n + m \cdot \alpha(m, n))$ où $\alpha(m, n)$ est la fonction inverse d'Ackermann¹⁶.*

On ne démontrera pas ce résultat. Notons simplement que $\alpha(m, n) \leq 4$ pour toutes valeurs réalistes de m et de n (jusqu'à plus de 10^{500} soit beaucoup plus que le nombre

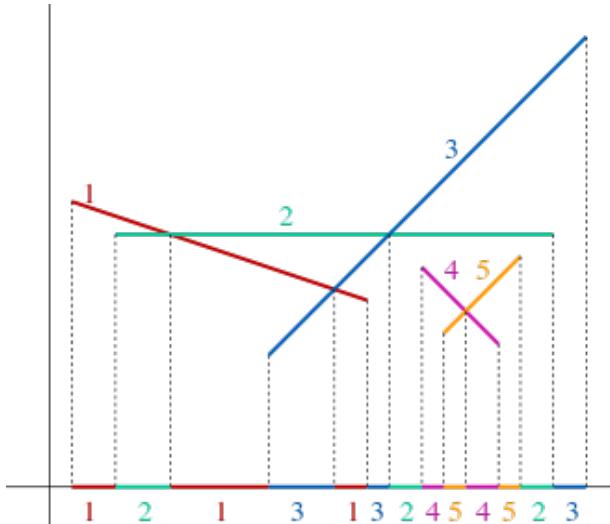
16. Ce résultat, ainsi que la fonction inverse Ackermann aussi définie page 77, est expliqué [ici](#).

de particules de l'univers estimé à 10^{80}). Ce résultat est très proche du temps optimal : évaluer m opérations coûte $\Omega(m)$ et initialiser chaque ensemble potentiel coûte $\Omega(n)$. Au total cela prend un temps d'au moins¹⁷ :

$$\max \{\Omega(n), \Omega(m)\} = \Omega(\max \{n, m\}) = \Omega(n + m).$$

On dit aussi parfois que la *complexité amortie* des opérations de fusion et de recherche est de $O(\alpha(m, n))$ dans la mesure où la somme de m opérations est $O(m \cdot \alpha(m, n))$, une fois l'initialisation de la structure de données effectuée en $O(n)$.

Parenthèse. La fonction inverse d'Ackermann apparaît aussi dans le contexte de la géométrique discrète, où il s'agit de déterminer dans un arrangement de n segments de droite du plan la complexité de l'enveloppe basse, c'est-à-dire le nombre de sous-segments que verrait un observateur basé sur l'axe des abscisses (voir la figure 3.18). Cette complexité intervient dans les algorithmes permettant de déterminer cette enveloppe basse.



[Cyril. Refaire le dessin en dessinant l'enveloppe basse.]

FIGURE 3.18 – La complexité de l'enveloppe basse de n segments du plan (ici $n = 5$) est $\Theta(n \cdot \beta(n))$ avec $\beta(n) = \min \{i : A(i, i) \geq i\}$, $A(i, j)$ étant la fonction d'Ackermann rencontrée page 77. Cette complexité est presque linéaire car $\beta(n) \leq \alpha(n^2, n) + O(1)$ et $\alpha(m, n) \leq 4$ pour toutes valeurs réalistes de m et n . Cette complexité correspond au nombre d'intervalles des sous-segments projetés sur l'axe des abscisses. Source Wikipédia.

17. La dernière équation est due au fait que $\max \{x, y\} \geq (x + y)/2$.

3.4.8 Algorithme de Christofides

Il s'agit d'une variante de l'algorithme précédent, due à Nicos Christofides en 1976, et qui donne une 1.5-approximation. C'est actuellement le meilleur algorithme d'approximation pour le TSP métrique.

L'algorithme utilise la notion de *couplage parfait* de poids minimum. Il s'agit d'apparier les sommets d'un graphe arête-valuée (G, ω) par des arêtes indépendantes de G (deux arêtes ne pouvant avoir d'extrémité commune). Bien sûr, pour pouvoir appairer tous les sommets, il faut que G possède un nombre pair de sommets. Un *couplage parfait* est ainsi une forêt couvrante F où chaque composante est composée d'une seule arête¹⁸.

Parmi tous les couplages F de (G, ω) il s'agit de trouver celui de poids $\omega(F)$ minimum. Un couplage parfait de poids minimum, s'il en existe un¹⁹, peut être calculé en temps $O(n^3)$. Ce dernier algorithme est complexe et ne sera pas abordé dans ce cours.

Algorithme APPROXMST2(V, d)

Entrée: Une instance (V, d) du VOYAGEUR DE COMMERCE.

Sortie: Une tournée, c'est-à-dire un ordre sur les points de V .

1. Calculer un arbre couvrant de poids minimum T sur le graphe complet défini par V et les arêtes valuées par d .
 2. Calculer l'ensemble I des sommets de T de degré impair.
 3. Calculer un couplage parfait de poids minimum F pour le graphe induit par I .
 4. La tournée est définie par un circuit eulérien du multi-graphe $T \cup F$ dans lequel on ignore les sommets déjà visités.
-

On rappelle qu'un *circuit eulérien* d'un multi-graphe, c'est-à-dire d'un graphe possédant éventuellement plusieurs arêtes entre deux sommets, est un circuit permettant de visiter une et une fois chacune des arêtes d'un graphe. Cela est possible si et seulement si tous les sommets du graphes sont de degrés²⁰ pairs.

Exemple. La figure 3.19 représente l'exécution de l'algorithme APPROXMST2(V, d), sur la même instance que l'exemple de la figure 3.11. La tournée n'est pas exactement la même.

18. Un couplage (donc pas forcément parfait) est une forêt couvrante où les composantes sont réduites à un sommet ou une arête

19. Même si G a un nombre pair de sommet, il pourrait ne pas avoir de couplage parfait, comme une étoile à trois feuilles par exemple.

20. Le degré d'un sommet est le nombre d'arêtes incidentes à ce sommet, ce qui peut donc être inférieur au nombre de voisins en présence d'arêtes multiples.

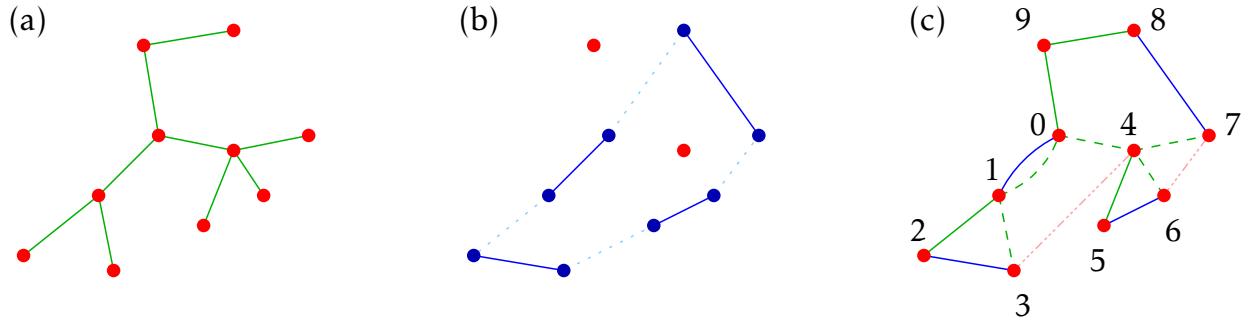


FIGURE 3.19 – (a) Arbre couvrant T de poids minimum ; (b) Couplage parfait F de poids minimum pour les points I (en bleu) correspondant aux sommets de degrés impairs de T ; (c) Multi-graphe $T \cup F$ et la tournée résultante : les chemins de T en pointillé vert ($3 \rightarrow 1 \rightarrow 0 \rightarrow 4$ et $6 \rightarrow 4 \rightarrow 7$) sont court-circuitées par les arêtes roses.

Validité. On peut se convaincre de la validité de l’algorithme en remarquant :

- Le couplage parfait existe bien car $|I|$ est pair (rappelons que dans tout graphe, il existe un nombre pair de sommet de degré impair) et que le graphe induit par I est une clique.
- L’ajout du couplage parfait F à T produit un multi-graphe où tous les sommets sont de degré pairs, puisqu’on ajoute exactement une arête incidente à chaque sommet de degré impair de T .
- Le circuit eulérien de $T \cup F$ visite au moins une fois chacun des sommets de V , puisque toutes les arêtes sont visitées et que T couvre V .

Facteur d’approximation. La longueur de la tournée renvoyée par l’algorithme, $\text{APPROXMST2}(V, d)$, est au plus la somme des poids des arêtes du circuit eulérien de $T \cup F$. Cela peut être moins car on saute les sommets déjà visités, ce qui grâce à l’inégalité triangulaire produit un raccourcis.

On a donc $\text{APPROXMST2}(V, d) \leq d(T \cup F) = d(T) + d(F)$. On a déjà vu que $d(T) < d(C^*) = \text{OPT}(V, d)$. Soit C_I la tournée optimale pour l’instance (I, d) , donc restreinte aux sommets I . Clairement $d(C_I) \leq d(C^*)$ puisque $I \subseteq V$.

Remarquons qu’à partir de la tournée C_I on peut construire deux couplages parfaits pour I : l’un obtenu en prenant une arête sur deux, et l’autre en prenant son complémentaire. Le plus léger d’entre eux a un poids $\leq \frac{1}{2}d(C_I)$ puisque leur somme fait $d(C_I)$. Il suit que le couplage parfait F de poids minimum pour I est de poids $d(F) \leq \frac{1}{2}d(C_I)$.

En combinant les différentes inégalités on obtient que :

$$\begin{aligned}
 \text{APPROXMST2}(V, d) &\leqslant d(T) + d(F) < d(C^*) + \frac{1}{2} \cdot d(C_I) \\
 &= d(C^*) + \frac{1}{2}d(C^*) = \frac{3}{2} \cdot d(C^*) \\
 &= \frac{3}{2} \cdot \text{OPT}(V, d)
 \end{aligned}$$

ce qui montre que le facteur d'approximation est de 1.5.

Il est clair que l'algorithme APPROXMST2 est de complexité polynomiale. L'étape la plus coûteuse (qui est aussi la plus complexe) est celle du calcul du couplage parfait de poids minimum en $O(n^3)$. La complexité totale de l'algorithme étant ainsi de $O(n^3)$. On a donc montré que :

Proposition 3.7 *L'algorithme APPROXMST2 est une 1.5-approximation.*

Parenthèse. *On peut construire une instance critique pour l'algorithme APPROXMST2, c'est-à-dire d'une instance où le facteur d'approximation est atteint (ou approché). En effet, ce n'est parce qu'on a prouvé que le facteur d'approximation est un certain α qu'il existe des instances où ce facteur est atteint.*

On choisit un nombre n impair de points formant $\lfloor n/2 \rfloor$ triangles équilatéraux de longueur unité comme le montre la figure 3.20.

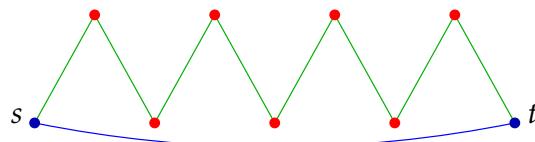


FIGURE 3.20 – Instance critique pour APPROXMST2 composé de $\lfloor n/2 \rfloor$ triangles équilatéraux unités. L'arbre T est en vert et le couplage F en bleu.

L'arbre de poids minimum T est le chemin de s à t parcourant tous les points, et donc $d(T) = n - 1$. Il n'y a alors que s et t qui sont de degré impair dans T . Donc le couplage parfait F est réduit au segment $s - t$ dont le coût est le nombre de triangles (chaque base valant 1), soit $\lfloor n/2 \rfloor$. La tournée produite par APPROXMST2 a un coût de $d(T) + d(C) = (n - 1) + \lfloor n/2 \rfloor = n - 1 + n/2 - 1/2 = 1.5n - 1.5$. Or la tournée optimale, obtenue en formant l'enveloppe convexe des n points, est de coût n . (On ne peut pas faire moins, la distance minimal entre deux points quelconques étant 1.) Le facteur d'approximation sur cette instance, $1.5 - O(1/n)$, approche aussi près que l'on veut 1.5.

3.5 Morale

- Le problème du VOYAGEUR DE COMMERCE (TSP) est un problème difficile, c'est-à-dire qu'on ne sait pas le résoudre en temps polynomial. Il est aussi difficile à approximer dans sa version générale, mais pas lorsque la fonction de distance d vérifie l'inégalité triangulaire.
- Il peut-être résolu de manière exacte par programmation dynamique, mais cela requière un temps exponentiel en le nombre de points.
- Lorsque la méthode exacte ne suffit pas (car par exemple n est trop grand) on cherche des heuristiques ou des algorithmes d'approximation censés être bien plus rapide, au moins en pratique.
- Il existe de nombreuses heuristiques qui tentent de résoudre le TSP. L'algorithme du point le plus proche (qui est un algorithme glouton) et l'algorithme 2-flip (qui est un algorithme d'optimisation locale) en sont deux exemples. Il en existe beaucoup d'autres.
- Un algorithme glouton n'est pas un algorithme qui consomme plus de ressources que nécessaire. Cette stratégie algorithmique consiste plutôt à progresser tant que possible sans remettre en question ses choix. En quelque sorte un algorithme glouton avance sans trop réfléchir.
- Les algorithmes d'approximation sont de complexité polynomiale et donnent des garanties sur la qualité de la solution grâce au facteur d'approximation. Le meilleur connu pour le TSP métrique, c'est-à-dire lorsque d vérifie l'inégalité triangulaire, est de 1.5, une variante de l'algorithme basé sur le DFS d'un arbre couvrant de poids minimum (MST).
- Pour être efficace, les algorithmes doivent parfois mettre en œuvre des structures de données efficaces, comme *union-and-find* qui permet de maintenir les composantes connexes d'un graphe en temps linéaire en pratique.
- On peut parfois optimiser les structures de données, et donc les algorithmes, en augmentant l'espace de travail, en utilisant par exemple un tableau auxiliaire pour la compression de chemin dans *union-and-find*. Le prix à payer est le coût du maintien de ces structures auxiliaires. De manière générale, il y a un compromis entre la taille, le temps de mise à jour de la structure de données et le temps de requête. Augmenter l'espace implique des mises à jour de cet espace, mais permet de réduire le temps de requêtes.

Bibliographie

- [Aro98] S. ARORA, *Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems*, Journal of the ACM, 45 (1998), pp. 753–782. doi : [10.1145/290179.290180](https://doi.org/10.1145/290179.290180).

- [BH15] J. BRECKLINGHAUS AND S. HOUGARDY, *The approximation ratio of the greedy algorithm for the metric traveling salesman problem*, Operations Research Letters, 43 (2015), pp. 259–261. doi : [10.1016/j.orl.2015.02.009](https://doi.org/10.1016/j.orl.2015.02.009).
- [Cha00] B. CHAZELLE, *A minimum spanning tree algorithm with inverse-Ackermann type complexity*, Journal of the ACM, 46 (2000), pp. 1028–1047. doi : [10.1145/355541.355562](https://doi.org/10.1145/355541.355562).
- [CKT99] B. CHANDRA, H. J. KARLOFF, AND C. A. TOVEY, *New results on the old k-Opt algorithm for the traveling salesman problem*, SIAM Journal on Computing, 28 (1999), pp. 1998–2029. doi : [10.1137/S0097539793251244](https://doi.org/10.1137/S0097539793251244).
- [CLRS01] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction à l’algorithmique (2e édition)*, DUNOD, 2001.
- [ERV07] M. ENGLERT, H. RÖGLIN, AND B. VÖCKING, *Worst case and probabilistic analysis of the 2-opt algorithm for the TSP*, in 18th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2007, pp. 1295–1304.
- [GYZ02] G. GUTIN, A. YEO, AND A. ZVEROVICH, *Traveling salesman should not be greedy : domination analysis of greedy-type heuristics for the TSP*, Discrete Applied Mathematics, 117 (2002), pp. 81–86. doi : [10.1016/S0166-218X\(01\)00195-0](https://doi.org/10.1016/S0166-218X(01)00195-0).
- [JM97] D. S. JOHNSON AND L. A. McGEOCH, *The traveling salesman problem : A case study in local optimization*, 1997. Local Search in Combinatorial Optimization, E. H. L. Aarts and J. K. Lenstra (eds.), John Wiley and Sons, London, 1997, pp. 215–310.
- [Kar15] M. KARPINSKI, *Towards better inapproximability bounds for TSP : A challenge of global dependencies*, in Electronic Colloquium on Computational Complexity (ECCC), vol. Report No. 97, June 2015.
- [Mit99] J. S. B. MITCHELL, *Guillotine subdivisions approximate polygonal subdivisions : A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems*, SIAM Journal on Computing, 28 (1999), pp. 1298–1309. doi : [10.1137/S0097539796309764](https://doi.org/10.1137/S0097539796309764).
- [Pet99] S. PETTIE, *Finding minimum spanning trees in $O(m\alpha(m,n))$* , Tech. Rep. TR-99-23, University of Texas, October 1999.
- [UL97] C. UMANS AND W. LENHART, *Hamiltonian cycles in solid grid graphs*, in 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 1997, pp. 496–505. doi : [10.1109/SFCS.1997.646138](https://doi.org/10.1109/SFCS.1997.646138).

Sommaire

4.1	Introduction	93
4.2	L'algorithme de Dijkstra	97
4.3	L'algorithme A*	105
4.4	Morale	112
	Bibliographie	113

Mots clés et notions abordées dans ce chapitre :

- Intelligence Artificielle (IA)
- *pathfinding, navigation mesh*
- algorithme de Dijkstra
- algorithme A*

4.1 Introduction

4.1.1 Pathfinding

La recherche de chemin (*pathfinding* en Anglais) est l'art de trouver un chemin entre deux points : un point de départ s (pour *start* ou *source* en Anglais) et une cible t (pour *target*). C'est un domaine à part entière de l'IA en Informatique.

Il existe de nombreux algorithmes de *pathfinding*, et on ne va pas tous les étudier : algorithme en faisceau (on explore qu'un nombre limité de voisins), algorithme *best-first* (on explore en premier le « meilleur » voisin déterminé par une heuristique), etc. On peut aussi se servir de ces algorithmes pour chercher une solution dans un espace plus ou moins abstrait. On les utilise principalement en robotique, pour les systèmes de navigation GPS et les jeux vidéos.

Pour les jeux vidéos, les algorithmes de *pathfinding* sont utilisés le plus souvent pour diriger les personnages non-jouables, c'est-à-dire les *bots* ou les IA, qui *in-fine* sont ani-

mées par des algorithmes exécutés par une machine (cf. figure 4.1). On utilise des algorithmes pour calculer les trajets car, pour des raisons évidentes de stockage, il n'est pas possible de coder *in extenso* (c'est-à-dire en « dur » dans une table ou un fichier) chaque déplacement $s \rightarrow t$ possibles¹. Parce que ces algorithmes sont particulièrement efficaces, ils sont aussi utilisés pour des jeux temps-réels² ou encore des jeux en-lignes multi-joueurs massifs où chaque joueur peut en cliquant sur l'écran déplacer automatiquement son personnage vers le point visé.

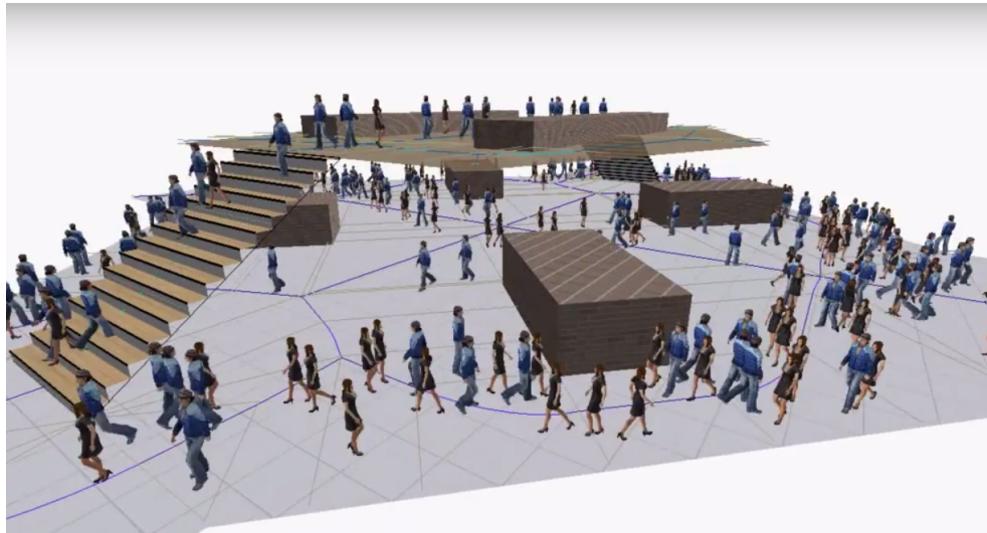


FIGURE 4.1 – Animation de *bots*.

4.1.2 *Navigation mesh*

Dans un jeu vidéo il faut spécifier par une structure abstraite où peuvent se déplacer les personnages. C'est le *navigation mesh*. Il s'agit d'un graphe. Les sommets sont les points d'intérêts ou point de cheminement (*waypoints* en Anglais) avec des coordonnées 2D ou 3D, et les arêtes interconnectent les points d'intérêts, le plus souvent en définissant un *tuilage*. Ce *tuilage* est à base de triangulations, de grilles ou d'autres types de maillage du plan, voir de l'espace, plus ou moins dense (figure 4.2). Bien sûr ce graphe est invisible au joueur qui doit avoir l'impression de naviguer dans un vrai décor. Il y a un compromis entre la taille du graphe (densité du maillage qui impacte les temps de calcul) et le réalisme de la navigation qui va en résulter.

On ne parlera pas vraiment des algorithmes qui, à partir d'une scène ou d'un décor, permettent de construire le *navigation mesh* (figure 4.2). Ce graphe est la plupart du

1. C'est parce qu'il y aurait des centaines de millions (n^2) de trajectoires à stocker pour une *map* avec quelques milliers (n) de points d'intérêts.

2. La notion de *temps-réel* se réfère au fait que le temps de réponse de la machine est limité de manière absolue et garantie, en pratique à quelques fractions de secondes.

temps déterminé (au moins partiellement) à la conception du jeu et non pas lors d'une partie, car cela peut être coûteux en temps de calcul. En terme de stockage, ce par contre relativement négligeable surtout en comparaison avec les textures par exemple.

Les algorithmes de *pathfinding* s'exécutent sur ce graphe, une structure qui reste cachée aux utilisateurs.

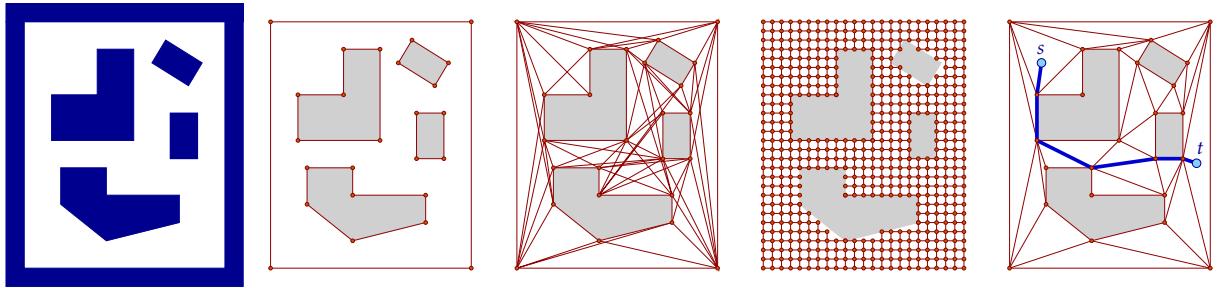


FIGURE 4.2 – Constructions de *navigation meshes*. De gauche à droite : décor 2D, graphe de contour, graphe de visibilité (une arête connecte des points d'intérêts visibles), maillage carré (ici des 4-voisinages), triangulation avec source et cible.

L'algorithme qui va nous intéresser est celui qui se chargent de trouver les chemins entre deux points d'intérêts $s \rightarrow t$ du *navigation mesh*. Dans la grande majorité des jeux, il s'agit d' A^* , un algorithme de *pathfinding* particulièrement efficace. Il s'agit d'une extension de l'algorithme de Dijkstra.

Bien sûr il y a de nombreux algorithmes qui gèrent la navigation des *bots* dans un jeu vidéo. Ceux, par exemple, chargés de la planification des paires $s_i \rightarrow t_i$ en fonction de l'environnement et des événements (objets mouvant ou autres *bots*), mais aussi pour rendre plus réaliste certaines trajectoires (un *bot* qui suivrait un plus court chemin trop tortueux peut paraître trop artificiel et nécessiter un redécoupage, par exemple en fonction de la visibilité du personnage). Il y a encore les algorithmes chargés de rendre réaliste le déplacement du personnage le long du chemin déterminé par A^* : adoucir les angles entre deux arêtes successives du chemin (cf. figure 4.3), aller en ligne droite au lieu de suivre les zig-zags d'une triangulation (comme sur la figure 4.2), etc.

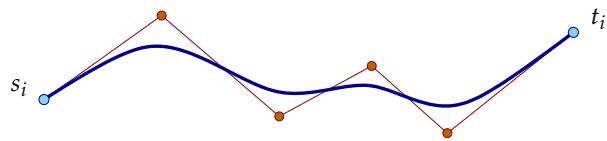


FIGURE 4.3 – Adoucissement d'un chemin, ici à l'aide d'une courbe *B-spline* cubique. On peut utiliser aussi des trajectoires « répulsives » évitant la collision avec les obstacles.

Il y a aussi les algorithmes qui déterminent le déplacement du personnage à l'intérieur d'une tuile vers le point d'intérêt le plus proche (et à la fin du dernier point

d'intérêt et de la cible). D'ailleurs une façon de faire est de modifier localement et temporairement le *navigation mesh* en ajoutant dans la tuile de la source (et de la cible) un point d'intérêt connecté à tous les points d'intérêts du bord de la tuile. Dans la suite nous supposerons que les déplacements planifiés $s_i \rightarrow t_i$ concernent des points d'intérêts (=sommets) du *navigation mesh* (=graphe) qui est fixe. C'est l'entrée du problème dont A^* est une solution.

4.1.3 Rappels

Il est important de bien distinguer les termes « poids des arêtes », « coût d'un chemin », « plus court chemin » et « distance », qui sont des notions proches mais différentes.

Soit G est un graphe, pas forcément symétrique, arête-valué par une fonction de poids ω . Par exemple, dans le cas d'un graphe géométrique, où les sommets sont des points du plan, $\omega(e)$ peut correspondre à la longueur de l'arête e , c'est-à-dire la distance euclidienne séparant ses extrémités. Mais dans un graphe général on parle plutôt de poids pour éviter la confusion avec la notion de longueur propre aux graphes géométriques.

Le *coût* d'un chemin C allant de u à v dans G est tout simplement la somme des poids de ses arêtes :

$$\text{coût}(C) = \sum_{e \in E(C)} \omega(e).$$

On dit que C est un chemin de *coût minimum* si son coût est le plus petit parmi tous les chemins allant de u à v dans G . Dans ce cas on dit aussi que C est un plus court chemin. La *distance* entre u et v dans G , notée $\text{dist}_G(u, v)$, est le coût d'un plus court chemin allant de u à v .

On peut utiliser aussi le terme de « coût » pour une arête e , à la place de « poids », car on peut très bien considérer e comme un chemin particulier joignant ses extrémités dont le coût est précisément $\omega(e)$ d'après la définition précédente.

Dans le chapitre 3 concernant le voyageur de commerce, nous avions utilisé le terme de *longueur minimum* plutôt que de coût minimum d'un chemin. C'était parce que le poids des arêtes correspondait à une longueur, la distance euclidienne entre les points extrémités de l'arête du graphe complet.

Attention ! Même si e est une arête, ce n'est pas forcément un plus court chemin. Par définition de la distance, si x, y sont voisins, alors $\text{dist}_G(x, y) \leq \omega(x, y)$. Cependant, l'arête $x - y$ peut représenter un trajet assez tortueux par exemple, si bien qu'un autre chemin alternatif, évitant $x - y$, pourrait avoir un coût strictement inférieur. Techniquement parlant, on a pas forcément l'inégalité triangulaire pour (G, ω) .

Évidemment, si l'objectif est de calculer des plus courts chemins, de telles arêtes ne sont pas très utiles et peuvent être supprimer du graphe en pré-traitement. Après ce traitement, l'inégalité triangulaire sera alors respectée. [*Exercice. Pourquoi ?*] Par contre, s'il faut trouver un chemin de coût maximum il faut les garder.

Si S est un sous-ensemble de sommets de G , on notera $N(S)$ l'ensemble des voisins de S dans G . Dit autrement, $N(S) = \bigcup_{s \in S} N(s)$ où $N(s)$ est l'ensemble des voisins de s dans G .

4.2 L'algorithme de Dijkstra

L'algorithme A* étant une extension Dijkstra, il est important de comprendre les détails de ce dernier. On va le présenter sous une version un peu modifiée. À l'origine, l'algorithme de Dijkstra calcule un plus court chemin entre un sommet source s et tous les autres accessibles depuis s dans un graphe G . Ici l'algorithme s'arrête dès qu'un sommet cible t donné est atteint. Pour fonctionner, l'algorithme suppose des poids positifs ou nuls, mais pas forcément symétrique. Il est possible d'avoir $\omega(u, v) \neq \omega(v, u)$. Par exemple, la montée d'un escalier est plus coûteuse que sa descente. Ou encore la vitesse de déplacement en vélo sur une route droite et plate peut être affectée par le sens du vent. Il n'y a pas d'autres hypothèses sur les poids. En particulier l'algorithme reste correct et calcule les plus courts chemins même si l'inégalité triangulaire n'est pas respectée, c'est-à-dire même s'il existe trois arêtes formant un triangle x, y, z avec $\omega(x, z) > \omega(x, y) + \omega(y, z)$.

Principe. On fait croître un sous-arbre du graphe depuis la source s en ajoutant progressivement les feuilles. La prochaine feuille à être ajoutée est choisie parmi le voisinage de l'arbre³ de sorte qu'elle minimise le coût du nouveau chemin ainsi créé dans l'arbre.

Il faut donc retenir que l'algorithme de Dijkstra est un algorithme glouton. On sélectionne le sommet le plus proche, c'est-à-dire celui qui minimise le coût du chemin créé, et on ne remet jamais en question ce choix. Comme le montre la figure 4.4, le choix de ce sommet peut être indépendant de certaines arêtes ce qui montre que l'algorithme ne peut être correct si des poids < 0 sont autorisés⁴.

Dans l'algorithme P représentera l'ensemble des sommets de l'arbre, et Q représentera la *frontière* de P , c'est-à-dire l'ensemble des sommets en cours d'exploration (voir la figure 4.4) qui sont aussi des voisins de P .

3. C'est l'ensemble des voisins des sommets de l'arbre dans G .

4. On peut tout de même arriver à trouver les distances correctes si G possède un seul arc uv de poids négatif (et sans cycle absorbant). Il faut calculer deux fois DIJKSTRA dans le graphe $G' = G \setminus \{uv\}$: l'un depuis s et l'autre depuis v . Ensuite, pour chaque sommet x , on sélectionne le plus court chemin entre celui qui ne prend pas uv et de coût $\text{dist}_{G'}(s, x)$, et celui qui passe par uv de coût $\text{dist}_{G'}(s, u) + \omega(u, v) + \text{dist}_{G'}(v, x)$.

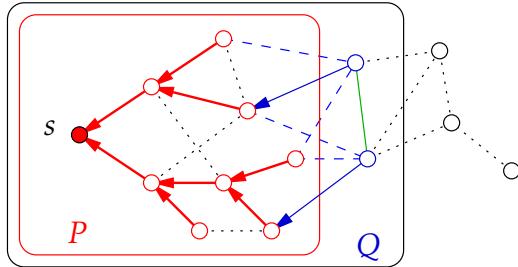


FIGURE 4.4 – Principe de l'algorithme de Dijkstra pour un graphe arête-valué (poids non représenté). Le choix du plus proche sommet $u \in Q$ est indépendant des arêtes internes à Q (arête verte). Il ne dépend pas non plus de la cible (non représentée). Les flèches liant les sommets de P représentent la relation $u \rightarrow \text{parent}[u]$.

Algorithme DIJKSTRA (modifié)

Entrée: Un graphe G , potentiellement asymétrique, arête-valué par une fonction de poids ω positive ou nulle, et $s, t \in V(G)$.

Sortie: Un plus court chemin entre s et t , une erreur s'il n'existe pas.

1. Poser $P := \emptyset$, $Q := \{s\}$, $\text{coût}[s] := 0$, $\text{parent}[s] := \perp$
 2. Tant que $Q \neq \emptyset$:
 - (a) Choisir $u \in Q$ tel que $\text{coût}[u]$ est minimum et le supprimer de Q
 - (b) Si $u = t$, alors renvoyer le chemin de s à t grâce à la relation $\text{parent}[u]$: $t \rightarrow \text{parent}[t] \rightarrow \text{parent}[\text{parent}[t]] \rightarrow \dots \rightarrow s$
 - (c) Ajouter u à P
 - (d) Pour tout voisin $v \notin P$ de u :
 - i. Poser $c := \text{coût}[u] + \omega(u, v)$
 - ii. Si $v \notin Q$, ajouter v à Q
 - iii. Sinon, si $c \geq \text{coût}[v]$ continuer la boucle
 - iv. $\text{coût}[v] := c$, $\text{parent}[v] := u$
 3. Renvoyer l'erreur : « le chemin n'a pas été trouvé »
-

Traditionnellement l'algorithme n'est pas présenté exactement de cette façon. D'abord, ici on s'arrête dès que la destination t est atteinte. Alors que dans l'algorithme d'origine on cherche à atteindre tous les sommets accessibles depuis s .

Ensuite, l'ensemble Q n'est pas explicité dans l'algorithme d'origine. Généralement on pose $\text{coût}[s] := 0$ et $\text{coût}[u] := +\infty$ pour tous les autres sommets u , si bien que les sommets de Q sont les sommets $u \notin P$ avec $\text{coût}[u] < +\infty$. L'algorithme classique s'écrit plutôt :

2. Tant qu'il existe un sommet $u \notin P$:
 - (a) Choisir $u \notin P$ tel que $\text{coût}[u]$ est minimum

L'avantage d'avoir l'ensemble Q est pour l'implémentation. Les tables $\text{coût}[\cdot]$ et $\text{parent}[\cdot]$ n'ont besoin d'être calculées *que* pour les sommets qui sont ajoutés à Q . Si t est proche de s , très probablement l'algorithme ne visitera pas tous les sommets du graphe. On consomme donc potentiellement beaucoup plus de mémoire et de temps si l'on initialise $\text{coût}[u] := +\infty$ pour tous les sommets, alors que l'initialisation à l'étape 1. prend ici un temps constant.

4.2.1 Propriétés

Il faut bien distinguer $\text{coût}[u]$, qui est la valeur d'une table pour le sommet u calculée par l'algorithme, et le coût d'un chemin C , notion mathématique notée $\text{coût}(C)$ correspondant à la somme des poids de ses arêtes. Bien évidemment, il va se trouver que $\text{coût}[u] = \text{coût}(C)$ où C est un plus court chemin de s à u , soit $\text{dist}_G(s, u)$ d'après les rappels de la section 4.1.3. Mais il va falloir le démontrer ! car c'est *a priori* deux choses différentes. D'ailleurs on verra plus tard que pour A^* $\text{coût}[u]$ n'est pas forcément le coût d'un plus court chemin.

Les deux propriétés suivantes sont immédiates d'après l'algorithme. En fait, elle ne dépendent pas du choix de u dans l'instruction 2(a) et seront donc communes avec l'algorithme A^* . On remarque que les tables $\text{coût}[u]$ et $\text{parent}[u]$ ne sont définies que pour les sommets de $P \cup Q$. De plus, à l'étape 2(a), $Q = (\{s\} \cup N(P)) \setminus P$.

Propriété 4.1 *S'il existe un chemin de s à t dans G , alors l'algorithme le trouve.*

En effet, on peut vérifier facilement que si le chemin n'est pas trouvé, alors tous les sommets accessibles depuis s ont été ajoutés à Q . Dit autrement, si t est accessible depuis s , l'algorithme finira par ajouter t à Q et trouvera le chemin. Il réalise en fait un parcours de la composante connexe de s dans le cas symétrique. Dans le cas asymétrique, t peut ne pas être accessible depuis s et pourtant être dans la même composante connexe comme dans l'exemple 4.5). La propriété 4.1 ne dépend en rien de la valuation ω des arêtes.

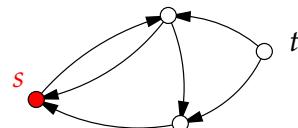


FIGURE 4.5 – Exemple de graphe asymétrique où t n'est pas accessible depuis s .

Propriété 4.2 Si $u \in P \cup Q$, le coût du chemin $u \rightarrow \text{parent}[u] \rightarrow \text{parent}[\text{parent}[u]] \rightarrow \dots \rightarrow s$ vaut $\text{coût}[u]$. De plus tous les sommets du chemin, sauf peut-être u , sont dans P .

C'est lié au fait que $\text{coût}[v]$ est construit en 2(d).iv par l'ajout à $\text{coût}[u]$ du poids $\omega(u, v)$ entre v et son père $u \in P$, ce qui de proche en proche constitue la somme des poids des arêtes du chemin de s à v .

La propriété suivante, que l'on va démontrer, dépend du choix de u dans l'étape 2(a).

Proposition 4.1 Soit u le sommet sélectionné à l'étape 2(a). Alors $\text{coût}[u] = \text{dist}_G(s, u)$.

On déduit de cette proposition que $\text{coût}[u] = \text{dist}_G(s, u)$ pour tout $u \in P \cup \{t\}$ puisque tous les sommets de P proviennent de l'ajout des sommets issus de l'étape 2(a), de même si $u = t$. En la combinant avec la propriété 4.2, on en déduit que le chemin défini par la table $\text{parent}[\cdot]$ pour tout sommet de $P \cup \{t\}$ est un plus court chemin. Dit autrement $\text{coût}[u]$ représente effectivement le coût d'un plus court chemin entre s et u .

Preuve. Pour démontrer par contradiction la proposition 4.1, on va suppose qu'il existe un sommet u sélectionné à l'étape 2(a) ne vérifiant pas l'énoncé, donc avec $\text{coût}[u] \neq \text{dist}_G(s, u)$. Comme $\text{coût}[u]$ est le coût d'un chemin de s à u (propriété 4.2), c'est que $\text{coût}[u] > \text{dist}_G(s, u)$.

Sans perte de généralité, on supposera que u est le premier sommet pour lequel $\text{coût}[u] > \text{dist}_G(s, u)$. Dans la suite, les ensembles P et Q correspondent aux ensembles définis par l'algorithme lorsque le sommet u est sélectionné en 2(a).

Tous les sommets sélectionnés en 2(a) avant u se trouvent dans P . Donc $\text{coût}[x] = \text{dist}_G(s, x)$ pour tout $x \in P$. Notons que $s \in P$ (et donc $P \neq \emptyset$), car $\text{coût}[s] = 0 = \text{dist}_G(s, s)$ et donc $u \neq s$.

Soit C un plus court chemin de s à u , et soit u' le premier sommet en parcourant C de s à u qui ne soit pas dans P (cf. figure 4.6). Ce sommet existe car $s \in P$ et $u \notin P$. Comme $Q \subset \{s\} \cup N(P)$, c'est que $u' \in Q$. À ce point de la preuve $u' = u$ est parfaitement possible.

Comme étape intermédiaire, nous allons montrer que $\text{coût}[u'] \leq \text{dist}_G(s, u')$.

Lorsque u est choisi, tous les arcs du type $w \rightarrow u'$ avec $w \in P$ ont été visité à cause de l'instruction 2(d). À cause de l'instruction 2(d).iii on a :

$$\text{coût}[u'] = \text{coût}[\text{parent}[u']] + \omega(\text{parent}[u'], u') = \min_{w \in P} \{\text{coût}[w] + \omega(w, u')\}. \quad (4.1)$$

Soit v' le prédécesseur de u' sur C , en parcourant C de s à u' . Par construction de u' , $v' \in P$. Il est possible d'avoir $v' \neq \text{parent}[u']$ comme illustré par la figure 4.6. À cause de l'équation (4.1), et puisque $v' \in P$,

$$\text{coût}[u'] \leq \text{coût}[v'] + \omega(v', u').$$

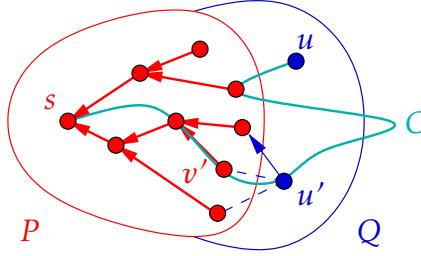


FIGURE 4.6 – Illustration de la preuve de la proposition 4.1. NB : Les flèches représentent la relation de parenté $w \rightarrow \text{parent}[w]$, pas les arcs du graphe. Le plus court chemin C de s à u peut pénétrer plusieurs fois P et Q , et ne pas suivre l'arborescence dans P à cause de poids nuls.

Notons $C[x, y]$ la partie du chemin C allant de x à y , pour tout $x, y \in C$. L'observation est que $C[x, y]$ est un plus court chemin entre x et y , car C est un plus court chemin. Dit autrement,

$$\text{coût}(C[x, y]) = \sum_{e \in E(C[x, y])} \omega(e) = \text{dist}_G(x, y).$$

(NB : ici $\text{coût}()$ est la valeur mathématique, pas $\text{coût}[]$.) Comme $v' \in P$, $\text{coût}[v'] = \text{dist}_G(s, v') = \text{coût}(C[s, v'])$ puisque $s, v' \in C$ qui est un plus court chemin. Or l'arc (v', u') appartient à C . On a donc :

$$\text{coût}[u'] \leq \text{coût}[v'] + \omega(v', u') = \text{coût}(C[s, v']) + \omega(v', u') = \text{coût}(C[s, u']) = \text{dist}_G(s, u').$$

On a donc montré que $\text{coût}[u'] \leq \text{dist}_G(s, u')$.

On a donc $\text{coût}[u'] \leq \text{dist}_G(s, u') \leq \text{dist}_G(s, u)$ mais aussi, par hypothèse, $\text{dist}_G(s, u) < \text{coût}[u]$. Il suit que $\text{coût}[u'] < \text{coût}[u]$, ce qui contredit le choix de u à l'étape 2(a) comme étant le sommet de Q de coût minimum. Par conséquent $\text{coût}[u] = \text{dist}_G(s, u)$, ce qu'on voulait montrer. \square

On remarquera que la preuve de la proposition 4.1 n'utilise pas l'inégalité triangulaire des poids. On utilise seulement le fait que le sous-chemin d'un plus court chemin est un plus court chemin.

4.2.2 Implémentation et complexité.

File de priorité. Généralement on implémente l'ensemble Q par une *file de priorité* (*priority queue* en Anglais). C'est une structure de données qui permet :

- de tester si la file est vide ;
- d'ajouter à la file un élément et sa priorité ; et
- d'extraire de la file l'élément de plus haute priorité.

Pour être plus précis, une clé c est associée à chaque élément v permettant de déterminer la priorité de l'élément. L'élément de plus haute priorité est celui avec la plus petite clé. C'est donc le couple (v, c) qui est inséré dans la file. Pour DIJKSTRA, la clé est $c = \text{coût}[v]$ si bien que l'élément de plus haute priorité est celui de coût minimum.

Des variantes plus sophistiquées de file de priorité permettent en plus d'augmenter la priorité d'un élément déjà dans la file en diminuant (=décrémenter) sa clé. C'est malheureusement plus complexe à programmer car il faut gérer, à chaque mise à jour de la file, la position de chaque élément dans la file.

Dans DIJKSTRA on remarque que l'on :

- parcourt chaque arc au plus une fois, ce qui coute $O(m)$;
- extrait de Q au plus une fois chacun des sommets, ce qui coute $O(n \cdot t_{\min}(n))$;
- ajoute au plus chacun des sommets à Q , ce qui coute $O(n \cdot t_{\text{add}}(n))$; et
- modifie les coûts au plus autant de fois qu'il y a d'arcs, ce qui coute $O(m \cdot t_{\text{dec}}(n))$.

Ici $t_{\min}(n)$, $t_{\text{add}}(n)$, $t_{\text{dec}}(n)$ sont les complexités en temps des opérations d'extraction du minimum, d'ajout et de modification de la clé d'un élément d'une file de taille au plus n .

Les sommets de P se gèrent par un simple marquage qui coute au total un temps et un espace en $O(n)$. Au total la complexité en temps de DIJKSTRA est donc

$$O(m + n \cdot t_{\min}(n) + n \cdot t_{\text{add}}(n) + m \cdot t_{\text{dec}}(n)).$$

Notons que les différentes tables, y compris la file, ne contiennent que des sommets distincts et donc occupent un espace $O(n)$.

Mise à jour paresseuse. En fait on peut se passer d'implémenter la décrémentation de clé dans la mesure où on est pas trop limité en espace. Au lieu d'essayer de décrémenter la clé c en $c' < c$ d'un élément v déjà dans la file, on peut simplement faire une mise à jour de manière paresseuse : on ajoute à la file un nouveau couple (v, c') (cf. figure 4.7). Cela n'a pas de conséquences dans la mesure où l'on extrait à chaque fois l'élément de clé minimum. C'est donc (v, c') , et sa dernière mise à jour, que l'on traitera en premier. Il en va de même en fait pour toute modification de clé, qu'elle soit une incrémentation ou décrémentation. Si plus tard on souhaite augmenter c' en $c'' > c'$, alors on ajoute (v, c'') à la file. Dans tous les cas, c'est la valeur minimum qui sera extraite en premier.

Cependant on crée, par cet ajout, le problème que le couple initial (v, c) va plus tard être extrait de la file, ce qui n'était pas possible auparavant. Dans DIJKSTRA on peut résoudre ce problème grâce à l'ensemble P , puisqu'une fois extrait, un sommet se retrouve dans P et n'a donc pas besoin d'être traité de nouveau.

Pour mettre en œuvre cette mise à jour paresseuse, il suffit de modifier l'algorithme DIJKSTRA ainsi.

(c) Si $u \in P$, continuer la boucle, sinon l'ajouter à P

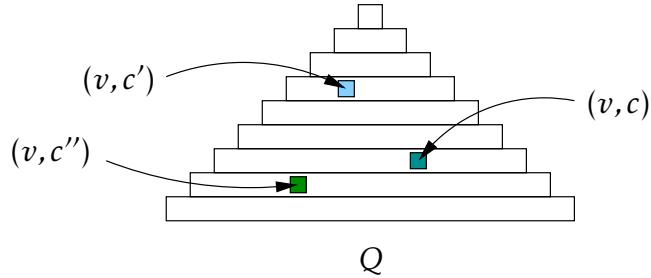


FIGURE 4.7 – Mise à jour paresseuse des clés d'une file Q , ici implémentée par un tas minimum. C'est la copie de v avec la plus petite clé (ici c') qui sera extraite en premier.

qui remplace l'ajout simple de u à P en 2(c). Dans « continuer la boucle » il faut comprendre revenir au début de l'instruction 2 du « Tant que $Q \neq \emptyset$ », ce qui en C se traduit par un simple `continue`. Il faut également réorganiser les instructions en 2(d) correspondant à la mise à jour des coûts des voisins v de u :

- | | | |
|--|-----------|--|
| <ul style="list-style-type: none"> ii. Si $v \notin Q$, ajouter v à Q iii. Sinon, si $c \geq \text{coût}[v]$ continuer la boucle iv. $\text{coût}[v] := c$, $\text{parent}[v] := u$ | \mapsto | <ul style="list-style-type: none"> ii. $\text{coût}[v] := c$, $\text{parent}[v] := u$ iii. ajouter v à Q |
|--|-----------|--|

Notez qu'au passage le code se simplifie (vive la paresse!) et surtout évite le test « $v \in Q$ » qui n'est pas adapté aux files.

L'autre inconvénient de cet ajout systématique est qu'on peut être amené à ajouter plus de n éléments à la file. Mais cela est au plus $O(m)$ car le nombre total de modifications, on l'a vu, est au plus le nombre d'arcs. L'espace peut donc grimper à $O(m)$. Mais, on va le voir, cela n'affecte pas vraiment la complexité en temps⁵ et vaut donc :

$$O(m + m \cdot (t_{\min}(m) + t_{\text{add}}(m))) .$$

Implémentation par tas. Une façon simple d'implémenter une file de priorité est d'utiliser un tas (*heap* en Anglais). Avec un tas classique implémenté par un arbre binaire quasi-complet (qui est lui-même un simple tableau), on obtient⁶ $t_{\min}(m) = O(\log m) = O(\log n)$ et $t_{\text{add}}(m) = O(\log m) = O(\log n)$ [Question. Pourquoi $O(\log m) = O(\log n)$?]. Ce qui donne finalement

$$O(m \cdot \log n) .$$

5. En plus les *navigations meshes* à base de triangulations du plan possèdent $m < 3n$ arêtes [Question. Pourquoi?]. Et puis il faut partir gagnant (surtout vrai avec A^*) : on espère bien évidemment trouver t avant d'avoir parcourir les m arcs du graphe!

6. C'est la suppression du minimum qui coute $O(\log m)$. Le trouver à proprement parler est en $O(1)$.

Cependant, il existe des structures de données pour les tas qui sont plus sophistiquées, notamment le tas de **Fibonacci**. Il permet un temps moyen par opérations – on parle aussi de *complexité amortie* – plus faible que le tas binaire. Il existe même une version, appelée tas de *Fibonacci strict* [SBLT12], avec $t_{\text{dec}}(n) = t_{\text{add}}(n) = O(1)$ et $t_{\min}(n) = O(\log n)$, soit une complexité dans le pire des cas et pas seulement amortie. La complexité finale tombent alors à $O(m + n \log n)$. On peut montrer que c'est la meilleure complexité que l'on puisse espérer pour **DIJKSTRA**. Mais ce n'est pas forcément le meilleur algorithme pour le calcul des distances dans un graphe !

Parenthèse. *Le principe consistant à prendre à chaque fois le sommet le plus proche implique que dans DIJKSTRA les sommets sont parcourus dans l'ordre croissant de leur distance depuis la source s . Si, comme dans la figure 4.8, la source s possède $n - 1$ voisins, le parcours de ses voisins selon l'algorithme donnera l'ordre croissant des poids de ses arêtes incidentes. En effet, l'unique plus court chemin entre s et v_i est précisément l'arête $s - v_i$. Ceci implique une*

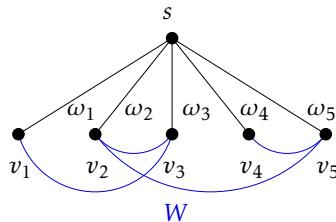


FIGURE 4.8 – Exemple de graphe avec $n = 6$ sommets et $m = 9$ arêtes où l'algorithme **DIJKSTRA** depuis s permet de trier les poids $\omega_i = \omega(s, v_i)$ des $n - 1$ arêtes incidentes à s , en supposant que le poids des autres arêtes vérifient $\omega(v_i, v_{i'}) = W > \max_i \{\omega_i\}$.

complexité d'au moins $\Omega(n \log n)$ pour **DIJKSTRA**, car il faut se souvenir que trier $k = n - 1$ nombres nécessitent au moins $\log_2(k!) = k \log_2 k - \Theta(k) = \Omega(n \log n)$ comparaisons. D'un autre côté la complexité est au moins le nombre total d'arêtes, m . Car, si toutes les arêtes et leurs poids ne sont pas examinés, l'algorithme pourrait se tromper. Il suit que la complexité de **DIJKSTRA** est au moins⁷

$$\max \{m, n \log n\} \geq \frac{1}{2}(m + n \log n) = \Omega(m + n \log n).$$

En utilisant une structure de données adéquate (notamment un tas de **Fibonacci**), **DIJKSTRA** peut effectivement être implémenté pour atteindre la complexité de $O(m + n \log n)$. Cependant, on ne peut pas en déduire que **DIJKSTRA** est l'algorithme ayant la meilleure complexité permettant de calculer les distances à partir d'une source donnée. Car rien n'indique

7. Attention ! Il y a ici deux arguments menant à deux bornes inférieures sur la complexité en temps. Schématiquement, l'un dit qu'il faut au moins 1h, tant que l'autre dit qu'il faut au moins 2h. On ne peut pas conclure directement que l'algorithme doit prendre 3h, mais seulement au moins le maximum des deux bornes inférieures. Rien ne dit, par exemple, qu'on ne peut pas commencer à trier les poids pendant qu'on examine les arêtes. Cependant, $\max \{x, y\} = \Omega(x + y)$.

que le principe du sommet le plus proche soit le meilleur. En fait, un algorithme de complexité optimale $O(n + m)$ [Question. Pourquoi est-ce optimal?] a été trouvé par [Tho99]. Bien sûr, cet algorithme ne parcourt pas les sommets par ordre croissant des distances depuis s .

4.3 L'algorithme A*

DIJKSTRA n'est pas vraiment adapté pour chercher une seule cible donnée. C'est un peu comme si on partait d'une île perdue en radeau pour rejoindre le continent et qu'on décrivait une spirale grandissante autour de l'île jusqu'à toucher un point quelconque de la terre ferme. Avec A* on estime le cap, puis on le suit avec plus ou moins de précision, en le ré-évaluant au fur et à mesure. Bien sûr il faut pouvoir estimer ce cap. En absence de cap, DIJKSTRA nous laisse dans la brume !



FIGURE 4.9 – Rejoindre le continent depuis une île perdue, avec ou sans cap.

L'algorithme A* a été mis au point en 1968 par des chercheurs en intelligence artificielle, soit presque 10 ans après l'article de Dijkstra présentant son célèbre algorithme [Dij59]. C'est une extension de l'algorithme de Dijkstra. Plusieurs versions ont été présentées : A1, puis A2 et au final A*.

Principe. Il est identique à celui de DIJKSTRA sinon que le choix glouton du sommet u se fait selon une valeur notée $\text{score}[u]$ qui tient compte, non seulement de $\text{coût}[u]$, mais aussi de l'estimation de la distance entre u et la cible t .

L'algorithme est donc paramétré par une fonction $h(x, t)$ qui est une heuristique sur la distance entre un sommet quelconque x et la cible t . Pour qu'A* calcule un plus court chemin, il faut que l'heuristique vérifie une condition supplémentaire qui sera détaillée plus tard.

C'est donc essentiellement l'ordre dans lequel les sommets de Q sont sélectionnés qui différentie l'algorithme A* de celui de DIJKSTRA. L'idée est qu'en visitant d'abord certains sommets plutôt que d'autres, grâce à l'heuristique h , on va tomber plus rapidement sur la cible que ne le ferait DIJKSTRA. Dans l'absolu, c'est-à-dire dans le pire

des cas, A* n'est pas meilleur que DIJKSTRA, les complexités sont les mêmes. C'est en pratique, sur des graphes particuliers, qu'A* se révèle supérieur.

Algorithme A*

Entrée: Un graphe G , potentiellement asymétrique, arête-valué par une fonction de poids ω positive ou nulle, $s, t \in V(G)$, et une heuristique $h(x, t)$ estimant la distance entre les sommets x et t dans G .

Sortie: Un chemin entre s et t dans G , une erreur s'il n'a pas été trouvé.

1. Poser $P := \emptyset$, $Q := \{s\}$, $\text{coût}[s] := 0$, $\text{parent}[s] := \perp$, $\text{score}[s] := \text{coût}[s] + h(s, t)$
 2. Tant que $Q \neq \emptyset$:
 - (a) Choisir $u \in Q$ tel que $\text{score}[u]$ est minimum et le supprimer de Q
 - (b) Si $u = t$, alors renvoyer le chemin de s à t grâce à la relation $\text{parent}[u]$:
 $t \rightarrow \text{parent}[t] \rightarrow \text{parent}[\text{parent}[t]] \rightarrow \dots \rightarrow s$
 - (c) Ajouter u à P
 - (d) Pour tout voisin $v \notin P$ de u :
 - i. Poser $c := \text{coût}[u] + \omega(u, v)$
 - ii. Si $v \notin Q$, ajouter v à Q
 - iii. Sinon, si $c \geq \text{coût}[v]$ continuer la boucle
 - iv. $\text{coût}[v] := c$, $\text{parent}[v] := u$, $\text{score}[v] := c + h(v, t)$
 3. Renvoyer l'erreur : « le chemin n'a pas été trouvé »
-

Sont encadrées les différences avec DIJKSTRA. Ainsi dans A* le choix du sommet u est déterminé non pas par son $\text{coût}[u]$ mais par son $\text{score}[u] = \text{coût}[u] + h(u, t)$. Comme on l'a déjà indiqué, les propriétés 4.1 et 4.2 ne reposent pas sur le choix du sommet u en 2(a). Elles sont donc communes avec celles de DIJKSTRA, et donc :

- si un chemin de s à t existe, A* le trouvera ; et
- le coût du chemin $u \rightarrow \text{parent}[u] \rightarrow \dots \rightarrow s$ vaut $\text{coût}[u]$ pour tout $u \in P \cup Q$.

On peut mesurer la différence de performances entre DIJKSTRA et A* dans le cas de graphes basés sur des grilles 2D (cf. figure 4.10). Pour une cible séparée d'une distance r de la source, DIJKSTRA visitera, à l'issue d'une recherche circulaire, environ r^2 sommets centrés autour de la source. Alors qu'A*, avec la distance vol d'oiseau comme heuristique h , visitera de l'ordre de r sommets, ce qui est évidemment le mieux que l'on puisse espérer. La différence est loin d'être négligeable en pratique.

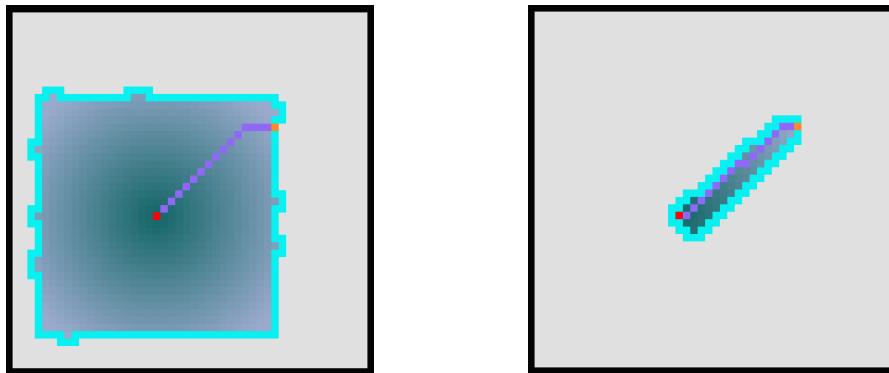


FIGURE 4.10 – Sur le plan DIJKSTRA (à gauche) visite un nombre quadratique de sommets en la distance, alors que A* (à droit) un nombre linéaire. Le plan est représenté ici par une grille avec 8-voisinage et l'heuristique est la distance vol-d'oiseau dans cette grille.

4.3.1 Propriétés

Les principales propriétés spécifiques à l'algorithme A* sont les suivantes :

Propriété 4.3 *Si $h(x, t) = 0$, alors A* est équivalent à l'algorithme DIJKSTRA, et donc calcule un plus court chemin entre s et t .*

C'est évident puisqu'on remarque que si $h(x, t) = 0$, alors $\text{score}[u] = \text{coût}[u]$ tout au long de l'algorithme A*, rendant les deux algorithmes absolument identiques.

Ce qui fait la force de l'algorithme A* est la propriété suivante qu'on ne démontrera pas (le terme « monotone » est expliqué juste après) :

Propriété 4.4 ([DP85]) *Tout algorithme qui calcule un chemin de s à t , sur la base de la même heuristique monotone h , visite au moins autant de sommets que A*.*

En fait, le nombre de sommets visités peut dépendre de l'ordre des sommets dans le tas si plusieurs sommets de Q sont de score minimum. Un algorithme gérant différemment les cas d'égalités pourrait visiter moins de sommets. Cependant, il existe un ordre des sommets du tas qui fait qu'A* ne visite pas plus de sommets que le meilleur algorithme possible.

L'heuristique h est *monotone* si $h(x, t) \leq \omega(x, y) + h(y, t)$ pour tout voisin y de x . Elle sous-estime la distance si $h(x, y) \leq \text{dist}_G(x, y)$ pour toutes les paires de sommets x, y où h est définie.

La monotonie est une sorte de version « faible » d'inégalité triangulaire pour h (cf. figure 4.11). Elle est différente car elle s'applique spécifiquement pour t et un voisin y

de x , au lieu de s'appliquer pas sur un triplet x, y, z quelconque de sommets. Cependant on retombe sur l'inégalité triangulaire $h(x, t) \leq h(x, y) + h(y, t)$ si l'on impose que $h(x, y) \geq \omega(x, y)$ pour chaque arête $x-y$, ce qui n'est pas une grosse contrainte car c'est l'estimation de la distance entre sommets distant qui est difficile, pas ceux qui sont directement connectés. La meilleure estimation qu'on puisse espérer est $h(x, t) = \text{dist}_G(x, t)$. Mais évidemment on dispose rarement d'une telle heuristique puisque $\text{dist}_G(\cdot, t)$ est ce qu'on cherche à calculer.

Si x et y sont connectés par un chemin C , et plus forcément une arête, alors la monotonie de h , appliquée sur chaque arête de C , implique la formule plus générale :

$$h(x, t) \leq \text{coût}(C) + h(y, t). \quad (4.2)$$

Une heuristique peut sous-estimer la distance sans être monotone. Par contre une fonction monotone sous-estime nécessairement la distance si $h(t, t) \leq 0$. [Exercice. Pourquoi?]

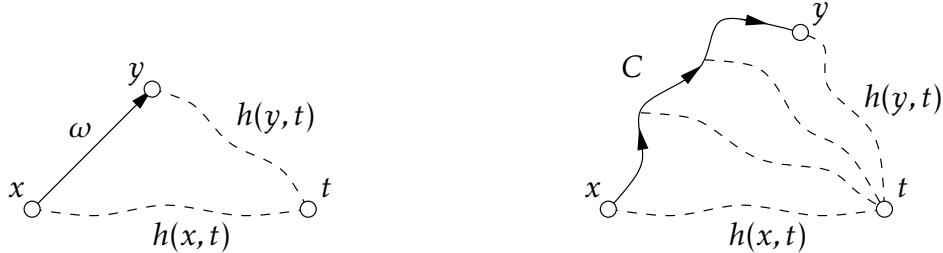


FIGURE 4.11 – Monotonie pour une arête $x - y$: $h(x, t) \leq \omega(x, y) + h(y, t)$. Généralisation à un chemin C de x à y : $h(x, t) \leq \text{coût}(C) + h(y, t)$.

L'heuristique définie par $\forall x \in V(G), h(x, t) = 0$ est monotone, de même que $h(x, t) = K$ où K est n'importe quelle constante indépendante de x . [Question. Pourquoi?] Mais, comme on va le voir, c'est aussi le cas de toute fonction de distance (et donc vérifiant l'inégalité triangulaire) qui sous-estime la distance dans le graphe. Typiquement, la distance « vol d'oiseau » vérifie l'inégalité triangulaire et bien sûr sous-estime la distance dans les graphes à base de grilles qui ne peut être que plus longue (cf. la figure 4.12). Elle est donc monotone. En effet, si h vérifie l'inégalité triangulaire et sous-estime la distance, alors $h(x, z) \leq h(x, y) + h(y, z)$ et $h(x, y) \leq \text{dist}_G(x, y)$ pour tout x, y, z . En particulier, si $x - y$ est une arête, $\text{dist}_G(x, y) \leq \omega(x, y)$, et on retrouve que $h(x, t) \leq \omega(x, y) + h(y, t)$ en posant $z = t$, ce qui montre que h est monotone.

On a vu que DIJKSTRA correspond à A* avec l'heuristique $h(x, t) = 0$ qui se trouve être monotone, et donc A* calcule un plus court chemin pour cette heuristique là. C'est en fait une caractéristique générale d'A*. On va montrer que :

Propriété 4.5 Si h est monotone, alors le chemin trouvé par A* est un plus court chemin. Plus précisément, le sommet u sélectionné à l'instruction 2(a) d'A* vérifie $\text{coût}[u] = \text{dist}_G(s, u)$.

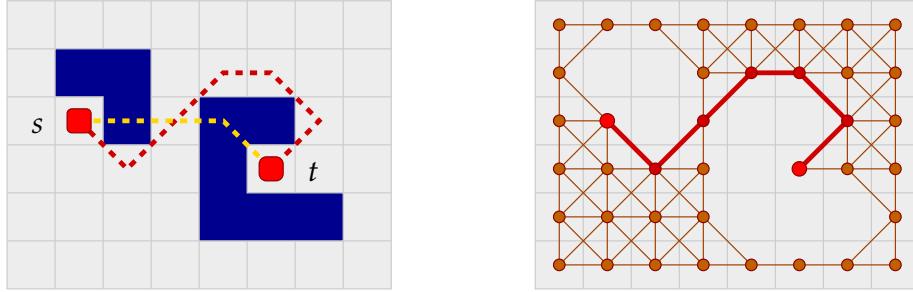


FIGURE 4.12 – La distance vol d'oiseau sous-estime la distance. Ici le graphe est un *navigational mesh* issu d'un maillage carré avec un 8-voisinage dans lequel on a enlevé les nœuds correspondant aux obstacles. La distance vol d'oiseau (en pointillé jaune) vaut dans l'exemple $\max\{|x_s - x_t|, |y_s - y_t|\} = 4$ au lieu de 6 pour le plus court chemin dans le graphe (en rouge).

Preuve. La preuve ressemble beaucoup à la preuve de la proposition 4.1, et reprend les mêmes notations (cf. figure 4.13). Donc u est toujours le premier sommet choisi en 2(d) tel que $\text{coût}[u] > \text{dist}_G(s, u)$. C'est notre hypothèse. La différence étant que u est choisi comme le sommet de Q de *score* minimum, et non pas comme celui de *coût* minimum.

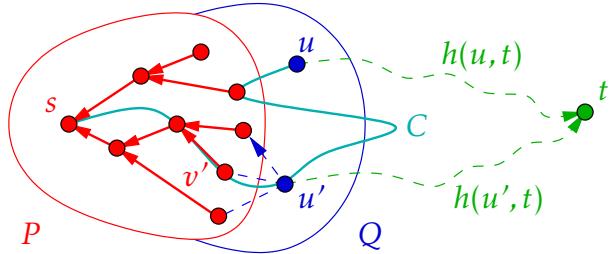


FIGURE 4.13 – Illustration de la preuve de la proposition 4.5.

On a montré dans la preuve de la proposition 4.1, en considérant le sommet u' sur le plus court chemin C de s à u , que :

$$\text{coût}[u'] \leq \text{dist}_G(s, u') . \quad (4.3)$$

Ceci reste valable puisque la preuve est basée sur la mise à jour de la table $\text{coût}[\cdot]$ en 2(d) pour les sommets de Q (voir aussi équation (4.1)) et la définition de C qui ne dépendent pas de $\text{score}[\cdot]$.

Dans la preuve de DIJKSTRA, l'équation (4.3) conduisait à $\text{coût}[u'] < \text{coût}[u]$ puisque $\text{dist}_G(s, u') \leq \text{dist}_G(s, u) < \text{coût}[u]$ par hypothèse sur u . C'était une contradiction car u était supposé être le sommet de coût minimum. Or ici pour A*, u est le sommet de score minimum où h intervient. Il faut conclure différemment.

Appliquons la propriété de monotonie de h sur chaque arête du chemin $C[u', u]$.

D'après l'équation (4.2) (voir aussi la figure 4.11) :

$$h(u', t) \leq \text{coût}(C[u', u]) + h(u, t).$$

Du coup,

$$\begin{aligned} \text{score}[u'] = \text{coût}[u'] + h(u', t) &\leq \text{dist}_G(s, u') + h(u', t) \quad (\text{par définition de score[]}) \\ &\leq \text{dist}_G(s, u') + \text{coût}(C[u', u]) + h(u, t) \quad (\text{par monotonie de } h) \\ &\leq \text{coût}(C[s, u']) + \text{coût}(C[u', u]) + h(u, t) \\ &\leq \text{coût}(C[s, u]) + h(u, t) \quad (\text{par définition de } C) \\ &\leq \text{dist}_G(s, u) + h(u, t) \quad (\text{par hypothèse sur } u) \\ &< \text{coût}[u] + h(u, t) = \text{score}[u]. \end{aligned}$$

L'inégalité $\text{score}[u'] < \text{score}[u]$ contredit le fait que u a été choisi comme le sommet de Q de score minimum. Donc $\text{coût}[u] = \text{dist}_G(s, u)$ ce qui termine la preuve. \square

4.3.2 Implémentation et complexité

La complexité et l'implémentation d'A* sont similaires à celles de DIJKSTRA, sinon qu'on implémente Q par un tas minimum pour la valeur $\text{score}[\cdot]$ au lieu de $\text{coût}[\cdot]$ comme vu au paragraphe 4.2.2. Mettre à jour le coût et le score d'un sommet peut se faire de manière paresseuse comme dans DIJKSTRA comme vu au paragraphe 4.2.2, en ajoutant systématiquement v à Q , même si v était déjà dans Q et même si le nouveau coût n'est pas meilleur que celui de la dernière version de v dans Q .

4.3.3 Plus sur A*

- La version présentée page 106 n'est pas la version originale d'A*. Dans sa version originale, l'instruction (d) devrait être :

(d) Pour tout voisin v de u :

L'effet est que des voisins déjà dans P peuvent être revisités, modifiant potentiellement leurs coûts et leurs scores. En les remettant dans Q on peut espérer trouver un chemin plus court au prix d'un temps d'exploration plus long. Cela complexifie l'analyse⁸, mais surtout cela n'est pas nécessaire si l'heuristique h est monotone, puisque dans ce cas A* (version du cours) calcule le chemin le plus court possible.

8. Notamment cela rend complexe l'analyse de la taille du tas avec une gestion paresseuse de la mise à jour du score.

Donc l'algorithme A* présenté dans le cours est une version un peu simplifiée et optimisée pour le cas des heuristiques monotones. La version originale d'A* est donc intéressante que lorsque h n'est pas monotone. En fait on peut montrer qu'il calcule un plus court chemin dès que h sous-estime la distance, une propriété plus faible que la monotonie.

- On peut parfois accélérer le traitement, dans le cas des graphes symétriques, en lançant deux exécutions d'A*: une de $s \rightarrow t$ et une de $t \rightarrow s$. Le chemin est construit dès qu'un sommet visité en commun a été trouvé. [Cyril. À finir.]
- On peut implémenter le parcours en profondeur (ou *DFS* pour *Depth-First Search*) à l'aide de A*. Pour cela, le coût des arêtes du graphe est fixé à 1. Puis, on remplace le terme $h(v, t)$ dans l'instruction 2(d).iv par un compteur (initialisée à $2m$ au départ) qui est décrémentée à chaque utilisation, si bien que c'est le premier sommet découvert qui est prioritaire. Cela revient aussi à dire que l'heuristique h décroît avec le temps d'exécution de l'algorithme. On obtient de meilleures performances en programmant directement un parcours *DFS*.
- L'algorithme A* peut également être utilisé pour calculer une α -approximation du plus court chemin entre s et t (cf. la définition 3.1 au chapitre 3). Si l'heuristique h est telle que $h(x, t)/\alpha$ est monotone pour une certaine constante $\alpha \geq 1$, alors A* trouve un chemin entre s et t (s'il existe) de longueur au plus $\alpha \cdot \text{dist}_G(s, t)$. Pour s'en convaincre, il suffit de réécrire, dans la preuve de la proposition 4.5, les inéquations page 110 comparant $\text{score}[u']$ à $\text{score}[u]$, en utilisant l'hypothèse que u est le premier sommet tel que $\text{coût}[u] > \alpha \cdot \text{dist}(s, u)$ et qu'ainsi⁹ $\text{coût}[u'] \leq \alpha \cdot \text{dist}(s, u')$, la monotonie de h/α impliquant que $h(u', t) \leq \alpha \cdot \text{coût}(C[u', u]) + h(u, t)$.

Donc ici $\alpha \geq 1$ est le facteur d'approximation sur la distance. L'espoir est que, grâce à une heuristique plus élevée, A* privilégie plus encore les sommets proches de la cible et visite ainsi moins de sommets. C'est très efficace s'il y a peu d'obstacles entre s et t . Voir la figure 4.14.

- L'algorithme A* n'est pas seulement utilisé pour le déplacement de *bots* et les jeux vidéos. Il sert d'heuristique pour l'exploration d'un espace de solutions. Le graphe représente ici des possibilités ou des choix, et il s'agit de trouver une cible dans cet espace (cf. [DRS07]). Un exemple est de savoir si un système peut atteindre un état cible donné à partir d'un état de départ donné, et de trouver un tel chemin. Donc ici le graphe n'est pas forcément entièrement donné dès le départ. Il est construit au fur et à mesure de l'exploration, les voisins d'un sommet u n'étant construits que si u est exploré. Bien sûr, la difficulté est dans la conception de l'heuristique h .

9. Il vaut se servir du fait que le prédecesseur v' de u sur C vérifie $v' \in P$ et donc que $\text{coût}[v'] \leq \alpha \cdot \text{dist}(s, v')$. Ensuite, à cause de la mise à jour des coûts, on en déduit que $\text{coût}[u'] \leq \text{coût}[v'] + \omega(v', u) \leq \alpha \cdot \text{dist}(s, v') + \text{dist}(v', u') \leq \alpha \cdot \text{dist}(s, u')$ car $v' - u'$ appartient à un plus court chemin et que $\alpha = 1$.

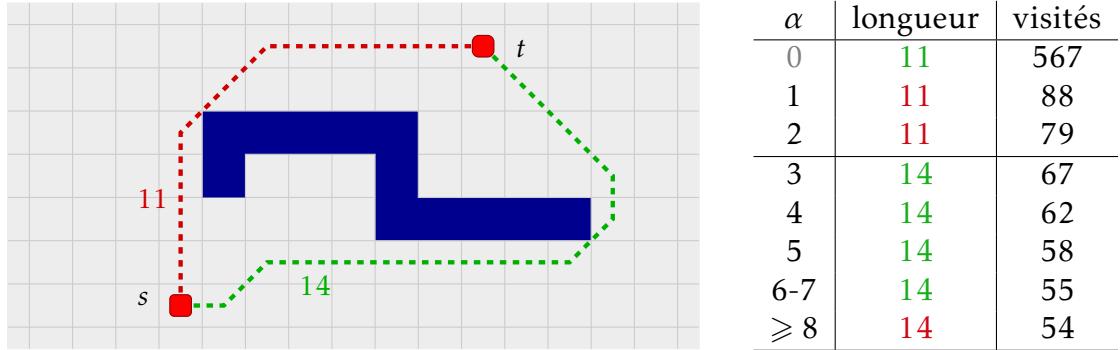


FIGURE 4.14 – Performances d’A* pour l’heuristique $h(x, t) = \alpha \cdot \delta(x, t)$ différentes valeurs entières d’ α , où $\delta(x, t)$ est la distance vol-d’oiseau pour les grilles avec un 8-voisinage. Seule une partie de la grille est représentée. Deux types de chemins sont trouvés. On remarque que dans le meilleur des cas ($\alpha = 2$) l’algorithme est capable de trouver le plus court chemin en visitant seulement 79 sommets, 7 fois moins qu’avec DIJKSTRA ($\alpha = 0$).

4.4 Morale

- Les *navigation meshes* sont des graphes issus de maillage de terrains 2D ou 3D pour simplifier les déplacements possibles des personnages artificiels (et autres *bots*) animés par des IA qui sont *infine* pilotée par des algorithmes exécutés par une machine. La requête principale est celle de recherche de chemin « court » entre deux points du *navigation mesh*.
- On peut faire mieux que DIJKSTRA en pratique en tenant compte de la cible, car les choix qu’il prend sont indépendants de la destination. Au contraire, A* profite d’informations sur la destination encodée par une heuristique qui peut être plus ou moins précise. C’est évidemment général : toute information supplémentaire peut être exploitée par l’algorithme pour être plus performant.
- Il faut distinguer le problème que résout un algorithme, et l’implémentation de l’algorithme. Il y a plusieurs implémentations possibles de DIJKSTRA, pas toutes équivalentes en termes de complexité. L’implémentation de DIJKSTRA qui a la plus faible complexité atteint $\Theta(m + n \log n)$. Cette borne est suffisante grâce aux tas de Fibonacci, et elle est nécessaire à cause du parcours des sommets par ordre croissant de distance depuis la source. Cependant, ce n’est pas la meilleure complexité pour le problème ! Il existe un algorithme en $O(m + n)$ qui calcule les distances de une source vers tous les autres.
- La ressource critique pour ce type d’algorithme, comme beaucoup d’autres en fait, est la mémoire utilisée, ce qui correspond au nombre de sommets visités. Pour chaque heuristique fixée, A* est l’algorithme de recherche de chemins qui visite le moins de sommets possibles.
- On peut se servir de A* pour approximer la distance avec une garantie sur le facteur d’approximation avec un choix judicieux de l’heuristique h .

- L'algorithme A* ne sert pas qu'à gérer le déplacement de *bots*. Il peut servir aussi à trouver des solutions dans un espace des « possibles », espace décrit implicitement par une fonction d'adjacence plutôt qu'explicitement par un graphe avec son ensemble de sommets et d'arêtes. Il est souvent utilisé comme brique de base en Intelligence Artificielle pour la résolution de problèmes d'optimisation.

Comme exemple on peut citer le problème du *Rubik's Cube*, où il s'agit à partir d'une configuration arbitraire de trouver un chemin « court » permettant d'atteindre la configuration gagnante où toutes les faces sont d'une seule couleur. Ici les sommets sont les configurations et le voisinage défini par les configurations accessibles par une rotation du cube (il y en a 18). Il n'est pas envisageable d'explorer, et encore moins de construire, le graphe des $n \approx 4.3 \times 10^{19}$ configurations. Même en explorant un milliard (10^9) de configurations par secondes il faudrait au moins 43 milliards de secondes pour parcourir seulement les sommets (pour les arêtes c'est 18 fois plus ...), soit plus de $43 \times 30 = 1\,290$ années de calculs.

Bibliographie

- [Dij59] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 (1959), pp. 269–271. doi : [10.1007/BF01386390](https://doi.org/10.1007/BF01386390).
- [DP85] R. DECHTER AND J. PEARL, *Generalized best-first search strategies and the optimality of A**, Journal of the ACM, 32 (1985), pp. 505–536. doi : [10.1145/3828.3830](https://doi.org/10.1145/3828.3830).
- [DRS07] H. DINH, A. RUSSELL, AND Y. SU, *On the value of good advice : the complexity of A* search with accurate heuristics*, in 22nd National Conference on Artificial Intelligence (AAAI), vol. 2, AAAI Press, July 2007, pp. 1140–1145.
- [SBLT12] G. STØLTING BORDAL, G. LAGOGIANNIS, AND R. E. TARJAN, *Strict Fibonacci heaps*, in 44th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 2012, pp. 1177–1184. doi : [10.1145/2213977.2214082](https://doi.org/10.1145/2213977.2214082).
- [Tho99] M. THORUP, *Undirected single-source shortest paths with positive integer weights in linear time*, Journal of the ACM, 46 (1999), pp. 362–394. doi : [10.1145/316542.316548](https://doi.org/10.1145/316542.316548).

Sommaire

5.1	Introduction	115
5.2	Trouver la paire de points les plus proches	118
5.3	Multiplication rapide	128
5.4	<i>Master Theorem</i>	136
5.5	Calcul du médian	140
5.6	Morale	142
	Bibliographie	143

Mots clés et notions abordées dans ce chapitre :

- la paire de points les plus proches
- algorithme de Karatsuba
- complexité définie par formule de récurrence
- *Master Theorem*

5.1 Introduction

Diviser pour régner (*divide-and-conquer* en Anglais) est une technique permettant de construire des algorithmiques récursifs. La stratégie consiste à découper le problème en sous-problèmes similaires (d'où l'algorithme récursif résultant) dans l'espoir d'affaiblir ou de casser la difficulté du problème initial.

L'expression provient du latin « *divide ut regnes* » et tire ses origines de l'antiquité. Une stratégie militaire (ou politique) bien connue consiste, afin d'affaiblir un groupe d'individus adversaires, à le diviser en plus petit espérant ainsi les rendre impuissant.

Les algorithmes produits ne sont pas forcément les plus efficaces possibles. Ils peuvent même parfois se révéler n'être pas plus efficaces que l'approche naïve. On a déjà vu de tels exemples, les algorithmes récursifs étant parfois franchement inefficaces

(cf. section 2.3). Cependant la technique gagne à être connue puisqu'elle peut mener à des algorithmes non triviaux auxquels on n'aurait peut-être pas pensé sinon.

L'archétype d'un algorithme résultant de cette approche est sans doute le *tri-fusion*. On découpe le tableau en deux sous-tableaux que l'on tri chacun récursivement. Ils sont ensuite recombinés (d'où le terme de *fusion*) pour obtenir un tableau entièrement trié. Voici le code :

```
// tri récursif de T[i..j]
void merge_sort(int *T, int i, int j){
    if(j-i<2) return;
    int m=(i+j)/2;
    merge_sort(T, i, m);
    merge_sort(T, m, j);
    fusion(T, i, m, j);
}
```

Il faut noter que c'est en fait la fusion qui trie le tableau. Les appels récursifs n'ont pas pour effet d'ordonner les éléments. Au mieux ils modifient les indices *i* et *j* ce qui virtuellement découpe le tableau en sous-tableaux de plus en plus petits. La fonction de comparaison de deux éléments (l'instruction « *T[p] < T[q]* ? » ci-après¹) n'est présente que dans la fonction *fusion()* dont le code est rappelé ci-dessous :

```
// fusion de T[i..m[ avec T[m..j[ en T[i..j[
void fusion(int *T, int i, int m, int j){
    int k=i, p=i, q=m, *r; // r = pointeur vers p ou q
    while(k<j){
        if(p==m) r=&q;
        else if(q==j) r=&p;
        else r=(T[p]<T[q])? &p : &q;
        A[k++]=T[(*r)++];
    }
    memcpy(T+i, A+i, (j-i)*sizeof(*A));
}
```

Remarque sur l'implémentation. Dans le code précédent de la fonction *fusion()*, il est supposé qu'on dispose d'un tableau auxiliaire *A* de la même taille que *T*. Bien sûr, on aurait peut aussi mettre en début de *fusion()* un *int *A = malloc(...)* et un *free(A)*

1. C'est cette instruction qu'il faudrait changer pour effectuer un tri selon une fonction de comparaison *fcmp()* quelconque à la *qsort()*. En fait, pour une fonction de comparaison absolument quelconque il faudrait déclarer le tableau *T[]* comme un tableau de *void** et définir une taille *sizeof()* des éléments du tableau et les copier dans *A[]* à l'aide de *memcpy()*.

avant de quitter la fonction. Cependant la fonction `fusion()` va être appelée $O(n)$ fois, soit autant de `malloc()` et de `free()` ce qui peut représenter un délai non négligeable sur le temps d'exécution. On peut donc faire comme ceci² :

```
static int *A;

// appel à merge_sort()
void sort(int *T, int n){
    A=malloc(n*sizeof(*A));
    merge_sort(T, 0, n);
    free(A);
}
```

L'approche du tri-fusion est efficace car il est effectivement plus rapide de trier un tableau à partir de deux tableaux déjà triés. Cela ne prend qu'un temps linéaire. Lorsque les sous-tableaux ne contiennent plus qu'un élément, alors les fusions opèrent. Pour analyser le temps consommé par toutes les fusions de l'algorithme, il est plus simple de grouper les fusions selon leur *niveaux*. Au plus bas niveau ($=0$) sont fusionnés les tableaux à un élément pour former des tableaux de niveau supérieur ($=1$). Puis sont fusionnés les tableaux de niveaux i (ou inférieur) pour former des tableaux de niveau $i + 1$. Comme la somme totale des longueurs des tableaux d'un niveau donné ne peut pas dépasser n , le temps de fusion de tous les tableaux de niveau i prend $O(n)$ pour chaque i . Si l'on découpe en deux parties égales³ à chaque fois, le niveau d'un tableau sera au plus $O(\log n)$ puisque sa taille doublera à chaque fusion. Au total la complexité est $O(n \log n)$.

Il est intéressant de remarquer que l'approche du tri-par-sélection, une approche naïve qui consiste à chercher le plus petit élément, de le mettre en tête et de recommencer sur le reste, est bien moins efficace : $O(n^2)$ comparaisons vs. $O(n \log n)$ pour le tri-fusion. On pourra se reporter au paragraphe 5.2.5 pour la comparaison des complexités n^2 et $n \log n$ en pratique.

Parenthèse. Construire un algorithme de tri de complexité $O(n \log n)$ itératif, donc non basé sur une approche récursive, n'est pas si simple que cela. Le tri-par-sélection, le tri-par-insertion⁴, et le tri-par-bulles⁵ sont des algorithmes itératifs de complexité $O(n^2)$. Même

2. Bien sûr, l'utilisation de la variable globale `A` n'est pas conseillé si `sort()` a vocation à être exécutée en parallèle.

3. Si n n'est pas une puissance de deux, il faut alors remarquer que la complexité en temps de l'algorithme ne sera pas plus grande que la complexité de trier $n' \in]n, 2n]$ éléments où cette fois n' est une puissance de deux, et $n' \log n' \leq 2n \log 2n = O(n \log n)$. En effet, on peut toujours, avant le tri, ajouter $n' - n$ éléments fictifs arbitrairement grand en fin de tableau. Après le tri, les n premiers éléments seront les éléments d'origine et triés.

4. On insère chaque élément à sa place dans le début du tableau comme le tri d'un jeu de cartes.

5. On échange les éléments qui ne sont pas dans le bon ordre.

le tri-rapide⁶ est récursif et de complexité $O(n^2)$, même si en moyenne la complexité est meilleure. Cf. le *tableau comparatif* des tris.

Cependant, le tri-par-tas échappe à la règle. Il n'est pas récursif, permet un tri en place, c'est-à-dire qu'il n'utilise pas de mémoire supplémentaire comme dans le tri-fusion, et a une complexité $O(n \log n)$. Le tableau T des n éléments à trier va servir de support à un tas maximum. La remarque est que les éléments d'un tas de taille k sont rangés dans k premières cases de T . Les $n - k$ cases suivantes sont libres pour le stockage des éléments de T qui ne sont pas dans le tas.

Dans une première phase les éléments sont ajoutés un à un au tas maximum jusqu'à le remplir. Cela prend un temps⁷ de $O(n \log n)$ pour les n insertions. Dans une seconde phase, on extrait successivement le maximum en le supprimant du tas et en remplaçant le tableau par la fin. Cela prend un temps de $O(n \log n)$ pour les n suppressions. Le tableau se trouve alors trié par ordre croissant en un temps total de $O(n \log n)$, et ceci sans avoir utilisé d'espace mémoire supplémentaire autre que le tableau lui-même.

5.2 Trouver la paire de points les plus proches

5.2.1 Motivation

Il s'agit de déterminer la paire de points les plus proches pris dans un ensemble donné de n points du plan. C'est un problème de géométrie discrète (*computational geometry*) qui s'est posée dans les années 1970 lorsqu'on a commencé à implémenter les routines de bases des premières cartes graphiques.

On a pensé pendant longtemps qu'aucun algorithme ne pouvait faire mieux qu'examiner chacune des paires de points, ce qui correspond à l'approche exhaustive. Il y a $\binom{n}{2} = n(n - 1)/2$ paires, ce qui donne une complexité en temps d'au moins $\Omega(n^2)$ pour l'approche exhaustive.

Et pourtant. On va voir que la technique « diviser pour régner » va produire un algorithme non trivial bien plus performant.

5.2.2 Principe de l'algorithme

Formellement, le problème s'énonce ainsi.

LA PAIRE DE POINTS LES PLUS PROCHES

Instance: Un ensemble $P \subset \mathbb{R}^2$ de n points du plan, $n \geq 2$.

Question: Trouver deux points distincts $p, p' \in P$ telle que $\text{dist}(p, p')$ est minimum.

-
- 6. On range les éléments par rapport à un pivot, et on recommence dans chaque partie.
 - 7. Une analyse plus fine permet de montrer que le temps total de n insertions est en fait $O(n)$.

Ici $\text{dist}(p, p')$ représente la distance euclidienne entre les points p et p' du plan. On étend cette notation aux ensembles, $\text{dist}(p, Q) = \min_{q \in Q} \text{dist}(p, q)$ représentant la distance entre un point p et un ensemble de points Q .

Diviser. L'idée est de partitionner les points de P en deux sous-ensembles, A et B . Si (p, p') est la paire recherchée, alors clairement soit :

- $(p, p') \in A^2$; ou bien
- $(p, p') \in B^2$; ou bien
- $(p, p') \in A \times B$.

On calcule alors récursivement $d_A = \min_{(a, a') \in A^2} \text{dist}(a, a')$ et $d_B = \min_{(b, b') \in B^2} \text{dist}(b, b')$ les distances minimum entre les paires de points de A^2 et B^2 . Reste ensuite à calculer d_{AB} , la distance minimum pour les paires de $A \times B$, afin de combiner le tout et d'obtenir la distance désirée (en fait la paire de points). Le calcul de d_{AB} est la partie difficile.

De prime abord, il semble que le calcul préliminaire de d_A et d_B n'aide pas vraiment pour le calcul de d_{AB} . En effet, le nombre de couples $(p, p') \in A \times B$ est⁸ $|A| \cdot |B| = \Omega(n^2)$ dans le cas équilibré où $|A| = |B|$. On a donc pas forcément avancé pour le calcul de d_{AB} , sinon, et c'est le point crucial, qu'on connaît la distance minimale à battre, soit $\min\{d_A, d_B\}$. C'est le petit détail qui va tout changer.

Plus en détails. Dans la suite on notera $\delta = \min\{d_A, d_B\}$ la plus petite des distances entre les paires de A^2 et B^2 . Pour tout sous-ensemble $Q \subseteq P$, on notera Q_x (resp. Q_y) la liste des points de Q ordonnée par abscisses x (resp. ordonnées y) croissant. On se servira du fait qu'une fois la liste P_x calculée, on peut calculer la liste Q_x par un simple parcours de P_x en temps $O(|P_x|)$. Idem pour Q_y à partir de P_y .

On supposera que P ne contient pas deux points avec la même abscisse. On peut s'en passer, mais cela complique la présentation et l'analyse de l'algorithme. Si jamais c'est le cas, on peut toujours effectuer une légère⁹ rotation des points pour se ramener à ce cas. La rotation ne change pas, en principe, la distance recherchée. Mais cela reste « en principe ». Le mieux est d'adapter correctement l'algorithme.

Soit p^* le point de rang $\lceil n/2 \rceil$ dans P_x . C'est l'élément *médian* de la liste ordonnée P_x : il y a autant d'éléments avant qu'après (à un près). On définit alors A comme l'ensemble des points de rang inférieur ou égale à celui de p^* dans P_x , et B l'ensemble des points de rang strictement supérieur à celui de p^* . Enfin, on pose L la ligne verticale passant¹⁰ par p^* et S l'ensemble des points de P qui sont à distance moins de δ de la ligne médiane L . Voir la figure 5.1.

8. On rappelle que $|A|$ représente la cardinalité de l'ensemble A , soit son nombre d'éléments.

9. La rotation « légère » exacte dépend en fait de la distance minimum entre deux points ... On peut alors s'en sortir avec une rotation aléatoire, et recommencer tant que cela échoue.

10. C'est ici qu'on a besoin qu'une ligne verticale donnée ne passe par qu'au plus un point de P .

Plus formellement :

$$\begin{aligned} A &= \{(x, y) \in P : x \leq x^*\} \\ B &= \{(x, y) \in P : x > x^*\} \\ L &= \{(x^*, y) : y \in \mathbb{R}\} \\ S &= \{(x, y) \in P : |x - x^*| < \delta\}. \end{aligned}$$

Notons que $|A|, |B| \leq \lceil n/2 \rceil$.

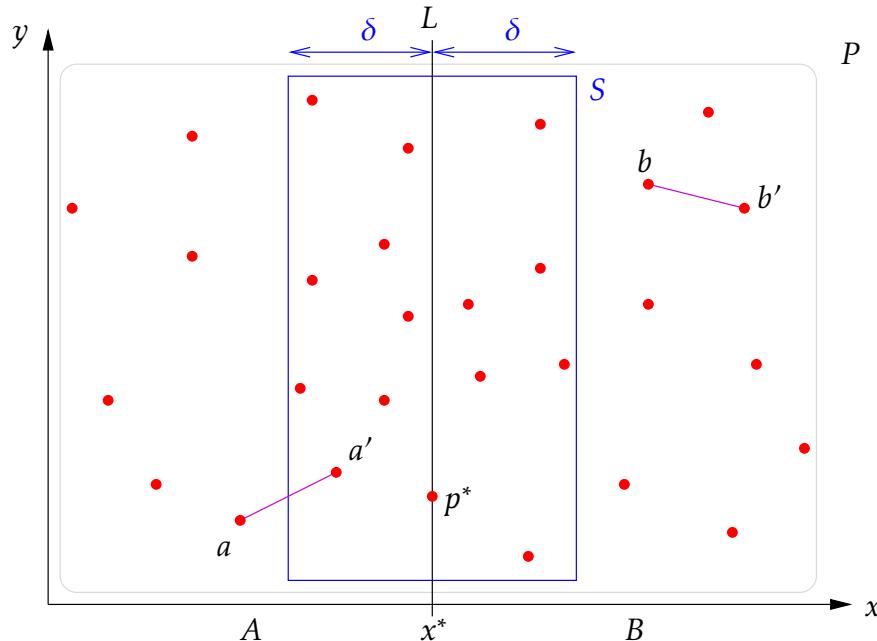


FIGURE 5.1 – Découpage de P en A et B selon le point médian p^* . Les paires (a, a') et (b, b') sont les paires de points les plus proches dans A et dans B [exemple presque réaliste].

L'algorithme repose sur les deux propriétés suivantes.

Propriété 5.1 *S'il existe $(a, b) \in A \times B$ tels que $\text{dist}(a, b) < \delta$, alors $a, b \in S$.*

Preuve. Si $a \notin S$, alors $\text{dist}(a, b) \geq \text{dist}(a, B) \geq \text{dist}(a, L) \geq \delta$: contradiction. De même, si $b \notin S$, alors $\text{dist}(a, b) \geq \text{dist}(b, A) \geq \text{dist}(b, L) \geq \delta$: contradiction. Conclusion : a et b sont tous des deux dans S . \square

Cette propriété seule n'aide pas beaucoup. Certes, pour calculer d_{AB} on peut se restreindre aux seules paires de S^2 . Malheureusement, il est parfaitement possible que $S = P$ (S contient tous les points), si bien que le calcul de d_{AB} peut se révéler aussi difficile que le problème initial. La propriété suivante va nous aider à calculer la paire de points les plus proches de S en temps $O(|S|)$ au lieu de $O(|S|^2)$.

Propriété 5.2 *S'il existe $(s, s') \in S^2$ tel que $\text{dist}(s, s') < \delta$, alors s et s' sont éloignés d'au plus 7 positions dans S_y .*

Dit autrement, si dans la liste $S_y = (s_0, \dots, s_i, \dots, s_j, \dots, s_{k-1})$ on a $\text{dist}(s_i, s_j) < \delta$, alors $j - i \leq 7$. En particulier, si les deux points les plus proches de S sont à distance $< \delta$, on les trouvera avec deux boucles comme par exemple :

```
for(i=0; i<k; i++)
    for(j=i+1; (j<=i+7)&&(j<k); j++){
        d=dist(S_y[i], S_y[j])
        if(d<d_min){ d_min=d; i_min=i; j_min=j; }
    }
```

Notez bien que les points d'indice après s_i dans S_y ne sont pas rangés suivant leur distance à s_i . Il est tout à fait possible d'avoir $\text{dist}(s_i, s_{i+7}) < \text{dist}(s_i, s_{i+1}) < \delta$. Par contre il est certain que $\text{dist}(s_i, s_{i+8}) \geq \delta$ de même que pour chaque s_j dès que $j > i + 7$. [Question. Que vaut d_{\min} si S_y ne contient pas au moins deux points?]

Preuve. Comme $\text{dist}(s_i, s_j) = \text{dist}(s_j, s_i)$, on va supposer sans perte de généralité que $i < j$ et donc s_i est en dessous de s_j . Pour tout $k \in \{0, 1, 2\}$, on pose H_k la ligne horizontale d'ordonnée $y(s_i) + k \cdot \delta/2$ où $y(s_i)$ est l'ordonnée de s_i . Donc H_0 est la ligne horizontale passant par s_i (voir la figure 5.2).

On va quadriller la partie du plan contenant S et au dessus de H_0 en boîtes carrées de côté $\delta/2$ de sorte que H_0 et L coïncident avec des bords de boîtes. Il ne faut pas que L coupe l'intérieur d'une boîte (voir la figure 5.2).

L'observation importante est que chaque boîte contient au plus un point de P . En effet, la distance la plus grande réalisable par deux points d'une même boîte a pour longueur la diagonale d'un carré de côté $\delta/2$, soit

$$\sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{2} \cdot \sqrt{2} = \frac{\delta}{\sqrt{2}} < \frac{\delta}{1.4} < \delta.$$

Donc si deux points p, p' sont dans une même boîte, ils sont à distance $\text{dist}(p, p') < \delta$. Or, cette boîte est incluse dans A ou dans B , L ne coupant l'intérieur d'aucune boîte. Ceci implique que leur distance doit être au moins $\min\{d_A, d_B\} = \delta$: contradiction.

D'après cette propriété, la zone du plan comprise entre H_0 et H_2 dans S contient au plus 8 points (incluant s_i) puisqu'elle ne comprend que 8 boîtes. En particulier s_{i+8} ne peut pas être compris entre H_0 et H_2 car $\{s_i, s_{i+1}, \dots, s_{i+8}\}$ contient 9 points. Il suit que $\text{dist}(s_i, s_{i+8}) \geq \text{dist}(H_0, H_2) \geq \delta$. Donc si $\text{dist}(s_i, s_j) < \delta$, on doit avoir $j < i + 8$, soit $j - i \leq 7$. \square

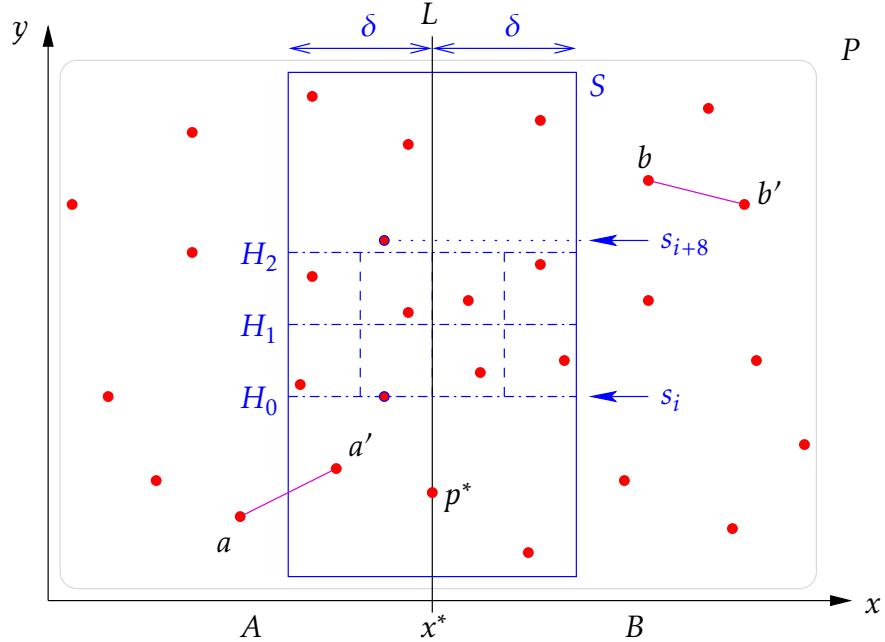


FIGURE 5.2 – Quadrillage de S en 8 boîtes de cotés $\delta/2$ pour s_i . Certaines boîtes pourraient être vides.

En fait, il est possible de placer 4 points dans un carré de côté δ avec la contrainte que les points soient à distance $\geq \delta$ les uns des autres, placés à l'intérieur du carré, et sans être sur la même ligne (H_i) ou même colonne (L), cf la figure 5.3. Ainsi, on peut placer s_{i+7} juste en dessous de H_2 de sorte qu'il soit au choix à distance $< \delta$ ou bien à distance $\geq \delta$ de $s_i \in H_0$ montrant l'optimatilité de la proposition 5.2.

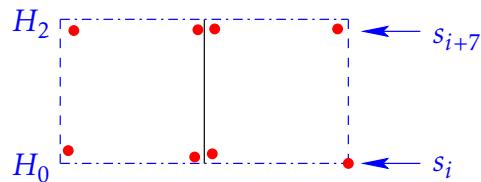


FIGURE 5.3 – Configuration montrant que le « 7 » de la proposition 5.2 peut être nécessaire.

5.2.3 L'algorithme

Algorithme PPPP(P)

Entrée: Un ensemble P de points du plan avec au moins deux points.

Sortie: La paire de points les plus proches.

1. Construire les tableaux P_x et P_y .
 2. Renvoyer $\text{PPPP}_{\text{REC}}(P_x, P_y)$.
-

Algorithme $\text{PPPP}_{\text{REC}}(P_x, P_y)$

Entrée: Deux tableaux de points (les mêmes) triés selon x et selon y .

Sortie: La paire de points les plus proches.

1. Si $|P_x| = 2$ ou 3 , renvoyer la paire de points de P_x les plus proches.
 2. Extraire le point médian de P_x , et soit x^* son abscisse.
 3. Soit $A = \{(x, y) \in P_x : x \leq x^*\}$ et $B = \{(x, y) \in P_x : x > x^*\}$. Construire les tableaux A_x, B_x, A_y et B_y à partir de x^*, P_x et P_y .
 4. Calculer $(a, a') = \text{PPPP}_{\text{REC}}(A_x, A_y)$ et $(b, b') = \text{PPPP}_{\text{REC}}(B_x, B_y)$
 5. Calculer $\delta = \min\{\text{dist}(a, a'), \text{dist}(b, b')\}$. Soit $S = \{(x, y) \in P_y : |x^* - x| < \delta\}$. Construire S_y à partir de P_y .
 6. Pour chaque point $s_i \in S_y$, calculer $\min\{\text{dist}(s_i, s_j) : j \in \{i + 1, \dots, i + 7\}\}$.
 7. Soit (s, s') la paire de points les plus proches calculée à l'étape 6, si elle existe¹¹. Renvoyer la paire de points les plus proches parmi (s, s') , (a, a') et (b, b') .
-

On a supposé que la ligne L du médian p^* ne contient qu'un seul point. Mais que se passe-t-il si cela n'est pas le cas ? Comment modifier l'algorithme pour qu'il marche efficacement dans tous les cas ?

5.2.4 Complexité

Soit $T(n)$ la complexité en temps de l'algorithme $\text{PPPP}_{\text{REC}}(P_x, P_y)$ lorsqu'il est appliqué à des tableaux P_x, P_y de n points chacun. La complexité en temps de l'algorithme PPPP appliqué à un ensemble P de n points est alors $O(n \log n) + T(n)$. En effet, $O(n \log n)$ est le temps nécessaire pour trier les n points (selon x et selon y) avec un algorithme de tri de cette complexité, comme le tri-fusion (cf. la section 5.1), auquel il faut ajouter le

11. Elle n'existe pas si $|S_y| = 1$.

temps $T(n)$ de calcul pour $\text{PPPP}_{\text{REC}}(P_x, P_y)$.

Il n'est pas difficile de voir que chaque étape, sauf peut-être l'étape 4 qui est récursive, peut être effectuée en temps $O(n)$ [Question. Pourquoi?]. On observe aussi que les tableaux A_x, A_y, B_x, B_y sont de taille au plus $\lceil n/2 \rceil$. Donc en incluant l'étape 4, on obtient que la complexité en temps de PPPP_{REC} vérifie l'équation :

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + O(n). \quad (5.1)$$

Afin d'éviter les pièges pointés dans la section 1.5.1, il est fortement conseillé de ne pas mettre de notation grand- O lors de la résolution d'une équation de récurrence. Cela tient au fait que si on « déplie » la récurrence, on aura un nombre non borné de termes en $O(\dots)$ ce qui généralement mène à des erreurs. Il se trouve que le facteur 2 devant $T(\lceil n/2 \rceil)$, de même que celui à l'intérieur de $T()$, est particulièrement important pour la complexité finale. Alors que celui dans le terme $O(n)$ l'est beaucoup moins. Mais tout ceci, on ne le saura qu'à la fin du calcul ...

Donc, pour une constante $c > 0$ assez grande, on a :

$$T(n) \leq \begin{cases} 2 \cdot T(\lceil n/2 \rceil) + cn & \text{si } n > 3 \\ c & \text{si } n \leq 3 \end{cases}$$

L'inéquation $T(n) \leq c$ pour $n \leq 3$ est tirée de l'algorithme qui termine en un temps constant dès que $n \leq 3$. En fait, on aurait pu écrire $T(3) \leq c'$ pour une certaine constante $c' > 0$, mais comme c est choisie « suffisamment grande » il n'est pas faux de mettre $T(3) \leq c$. En fait, pour le cas terminal, on peut écrire un peu ce qu'on veut car il est clair que lorsque n est constant, le temps de l'algorithme est aussi constant. Donc, si $n = O(1)$ alors $T(n) = O(1)$, peu importe que $n = 2, 3$ ou 100 .

On cherche donc à résoudre l'équation précédente. On verra dans la section 5.4 que la complexité $T(n)$ n'est pas influencée par les parties entières¹²

12. Une façon de le voir est qu'en temps $O(n)$ on peut ajouter des points sans modifier la solution [Question. Pourquoi?] et de sorte que le nouveau nombre de points soit une puissance de deux (et donc les parties entières peuvent être supprimées). Le nombre de points est au plus doublé [Question. Pourquoi?], ce qui n'a pas d'impact sur la complexité tant que celle-ci reste polynomiale [Question. Pourquoi?].

En dépliant¹³ la formule de récurrence, il vient :

$$\begin{aligned}
 T(n) &\leqslant 2 \cdot T(n/2) + cn \\
 &\leqslant 2 \cdot [2 \cdot T((n/2)/2) + c \cdot (n/2)] + cn \\
 &\leqslant 2^2 \cdot T(n/2^2) + cn + cn \\
 &\leqslant 2^2 \cdot [2 \cdot T((n/2^2)/2) + c \cdot (n/2^2)] + 2 \cdot cn \\
 &\leqslant 2^3 \cdot T(n/2^3) + cn + 2 \cdot cn \\
 &\quad \dots \\
 &\leqslant 2^i \cdot T(n/2^i) + i \cdot cn \quad \forall i > 0
 \end{aligned} \tag{5.2}$$

La dernière équation est valable pour tout $i \geqslant 1$. La récurrence s'arrête dès que $n/2^i \leqslant 3$. Lorsque $i = \lceil \log_2(n/3) \rceil$, on a $2^i \geqslant 2^{\log_2(n/3)} = n/3$ et donc $n/2^i \leqslant 3$, ce qui permet d'écrire :

$$\begin{aligned}
 T(n) &\leqslant 2^{\lceil \log_2(n/3) \rceil} \cdot T(3) + \lceil \log_2(n/3) \rceil \cdot cn \\
 &\leqslant 2^{(\log_2(n/3))+1} \cdot c + O(n \log n) \\
 &\leqslant 2c \cdot n/3 + O(n \log n) \\
 &\leqslant O(n) + O(n \log n) = O(n \log n).
 \end{aligned}$$

Remarquons que la constante c ne joue aucun rôle (on aurait pu prendre $c = 1$), ainsi que la constante « 3 » sur n dans le cas terminal.

Au final on a donc montré que la complexité en temps (dans le pire des cas) de l'algorithme PPPP est $O(n \log n) + T(n) = O(n \log n)$, soit bien mieux que l'approche exhaustive.

5.2.5 Différences entre n , $n \log n$ et n^2

Il est important de réaliser l'énorme différence en pratique entre un algorithme linéaire ($O(n)$), quasi-linéaire ($O(n \log n)$) ou quadratique ($O(n^2)$). Par exemple, considérons un jeu de données avec $n = 10^9$ points (un milliard), ce qui représente un fichier de `2*n*sizeof(double)` ≈ 16 Go. Les numérisations digitales au laser de grands objets 3D (statues, cavernes, etc.) dépassent largement cette taille¹⁴. Et supposons qu'on dispose d'une machine capable de traiter un milliard d'instructions élémentaires par secondes (soit une fréquence de $10^{-9} = 1$ GHz).

13. Déplier la récurrence permet de trouver la formule en fonction de i et de n . Pour être rigoureux, il faudrait le démontrer. Mais une fois qu'on a la formule, c'est très facile de le faire ... par récurrence justement! Appliquer l'hypothèse de récurrence revient à déplier la formule une fois de plus.

14. Pensez qu'un cube de données volumiques de simplement 1 000 points de cotés fait déjà un milliard de points (ou voxels).

Le tableau de comparaison ci-après donne une idée des différents temps d'exécution en fonction de la complexité. Bien sûr le temps réel d'exécution n'est pas forcément exactement celui-ci. La complexité linéaire $O(n)$ correspond peut-être à l'exécution réelle de $10n$ instructions élémentaires ; Et puis il y a différents niveaux de caches mémoires qui influencent le temps d'exécution. Mais cela donne toutefois un bon ordre de grandeur. (Voir aussi le paragraphe 60).

complexité	temps
n	1 seconde
$n \log_2 n$	30 secondes
n^2	30 années

Si dans les années 70, alors qu'on pensait que n^2 était la meilleure complexité et que les ordinateurs étaient bien moins efficaces (les horloges étaient cadencées au mieux ¹⁵ à 1 MHz, c'est 1 000 fois moins qu'aujourd'hui), on avait demandé aux chercheurs en informatique si un jour on pourrait traité un problème d'1 milliard de points, ils auraient sans doute dit « non ». On parle ici de 30 000 ans à supposer que le problème puisse tenir en mémoire centrale, ce qui n'était pas possible à l'époque.

C'est donc les avancées algorithmiques qui permettent les plus grandes progressions, puisqu'à puissance de calcul égale on passe de 30 ans ¹⁶ à 30 secondes simplement en concevant un meilleur algorithme.

5.2.6 Plus vite en moyenne

En fait, le problème peut être résolu en temps $O(n)$ en utilisant un algorithme probabiliste. L'algorithme repose sur des tables hachages. Il est basé sur le tirage initial d'un ordre aléatoire des points. Le temps dépend de ce tirage initial, c'est donc une variable aléatoire, mais la paire les plus proches est correctement renvoyée. La complexité est donc ici une complexité moyenne calculée sur tous les tirages possibles. On parle d'algorithme *Las Vegas*. Les détails sont relativement complexes et on n'en parlera pas plus.

Parenthèse. Voici quelques détails supplémentaires. On utilise une grille virtuelle G_δ du plan où chaque case correspond à un carré de côté δ , chacune des cases d'indices (i, j) pouvant contenir une liste arbitraire de points. Il s'agit d'une table de hachage ¹⁷ où temps constant il est possible d'avoir accès à la liste associée à la case (i, j) , afin de la lire ou d'y ajouter un point. Donc $G_\delta[(i, j)]$ est une simple liste de points appartenant au carré $[i\delta, i\delta + \delta] \times [j\delta, j\delta + \delta]$. L'idée est de remplir cette table successivement avec les points p_1, p_2, \dots .

15. La fréquence des microprocesseurs était de 740 KHz pour l'Intel 4004 en 1971. Il faudra attendre 1999 pour atteindre 1 GHz avec l'Athlon, voir https://fr.wikipedia.org/wiki/Chronologie_des_microprocesseurs.

16. En fait c'est même plus de 31 ans.

17. C'est l'implémentation d'une telle table de hachage en temps constant qui est complexe.

Initialement $\delta = \text{dist}(p_1, p_2)$. Puis, on prend les points dans l'ordre p_1, p_2, \dots . À chaque point $p_t = (x_t, y_t)$ on calcule l'indice (i, j) de la case de G_δ où tombe p_t . Il s'agit de la case $(i, j) = (\lfloor x_t/\delta \rfloor, \lfloor y_t/\delta \rfloor)$. Ensuite on calcule la distance minimum d_t entre p_t et tous les points contenus dans la case (i, j) et ses 8 voisines $(i \pm 1, j \pm 1)$. On pose $d_t = +\infty$ si ces 9 cases sont vides. Si $d_t \geq \delta$, on ajoute simplement p_t à la liste $G_\delta[(i, j)]$ et on continue avec le point suivant p_{t+1} . Si le dernier point est atteint, δ est la distance cherchée. Si $d_t < \delta$, alors on efface la grille G_δ et on recommence le remplissage des points p_1, p_2, \dots dans une nouvelle grille de paramètre $\delta = d_t$.

Pour montrer que la complexité est $O(n)$ en moyenne, en supposant que les points sont ordonnés aléatoirement, il faut remarquer :

- (1) Si p_t est à distance $< \delta$ d'un des points précédents cela ne peut être qu'un point des 9 cases centrées en (i, j) . En effet la distance entre p_t et tout point de toute autre case est $\geq \delta$.
- (2) Comme observé précédemment page 121, chacun des 4 sous-carrés de coté $\delta/2$ d'une case de G_δ ne peut contenir qu'un seul point. Par conséquent, le calcul de d_t prend un temps constant après avoir extrait les 9 listes d'au plus 4 points chacune.
- (3) La diminution de δ , qui entraîne un redémarrage du remplissage à partir de p_1 , se produit sur des points d'indices t tous différents (en fait strictement croissants).

Ainsi, si le redémarrage se produit pour p_t , alors le coût sera de $O(t)$. Plus généralement, si cela se produit pour chaque p_t avec une certaine probabilité, disons $f(t)$, alors le coût total de l'algorithme sera de $\sum_{t=1}^n O(t \cdot f(t))$.

Pour montrer que ce coût est en $O(n)$, il suffit donc de montrer que $f(t) = O(1/t)$. La probabilité $f(t)$ recherchée est celle de l'événement où p_t se trouve être l'une des extrémités de la paire de points la plus proche parmi les t premiers points p_1, \dots, p_t . Ces t points forment $\binom{t}{2} = t \cdot (t - 1)/2$ paires de points. L'une des extrémités de la paire la plus proche étant p_t , cela laisse $t - 1$ possibilités pour l'autre extrémité, disons p_s avec $s \in [1, t]$. À cause de la permutation aléatoire initiale des n points, chacune des possibilités pour p_s se produit uniformément. Ainsi la probabilité qu'il existe p_s avec $s \in [1, t]$ telle que (p_s, p_t) soit la paire de points la plus proche parmi p_1, \dots, p_t est donc $(t - 1)/\binom{t}{2} = 2/t$, ce qui termine l'analyse de la complexité.

On peut également en temps $O(n)$ en moyenne, avec un algorithme *Las Vegas*, calculer les deux points les plus éloignés. L'implémentation est bien plus simple et ne nécessite pas la programmation de structures de données complexes comme les tables de hachage en temps constant. Le principe est de tirer un point $p \in P$ uniformément au hasard. On calcule le point p' le plus éloigné de p . On supprime ensuite de P tous les points du disque de diamètre $\text{dist}(p, p')$, sauf p et p' . Puis on met à jour l'ensemble P avec les points, et on recommence jusqu'à n'avoir plus que trois points ou moins. Avec de simples tableaux il est possible de réaliser chaque itération en temps $O(|P|)$. [Question. Pourquoi?] L'analyse de la complexité nécessite des détails supplémentaires.

Parenthèse. Les voici. La probabilité de supprimer au moins $|P|/7$ points est $1/2$, et ce quel que soit l'ensemble P de points. Pour voir, posons r la distance recherchée, c'est-à-dire la distance la plus grande entre deux points de P , ou encore le diamètre de P . Il existe donc

un point de P où le disque de rayon r contient tous les points de P . On peut vérifier que ce disque peut être couvert par au plus 7 disques de rayon $r/2$ (et donc de diamètre r). [Cyril. À FINIR.]

Pour simplifier l'analyse, on peut s'arranger pour supprimer à coup sûr au moins $|P|/7$ des points en recommençant à choisir un nouveau p si tel n'était pas le cas. Après un nombre d'essais en moyenne constant (en fait 2), $|P|/7$ des points sont effectivement supprimés de P , il en restera donc au plus $6|P|/7$.

On montre alors que la complexité moyenne de l'algorithme vérifie $T(n) = T(6n/7) + O(n)$ ce qui donne $T(n) = O(n)$. En effet, en utilisant la formule (1.4), en posant $q = 6/7$ et $O(n) \leq cn$ pour une certaine constante $c > 0$, on obtient :

$$\begin{aligned} T(n) &\leq cn + cn \cdot q + cn \cdot q^2 + \dots \leq cn \cdot \sum_{i=0}^n q^i \\ &\leq cn \cdot \frac{1-q^n}{1-q} < cn \cdot \frac{1}{1-q} \quad (\text{car } q < 1) \\ &< cn \cdot \frac{1}{1-6/7} = 7cn = O(n). \end{aligned}$$

5.3 Multiplication rapide

L'algorithme qu'on va présenter a été développé par Anatolii Alexevich Karatsuba en 1960 et publié en 1962. L'article d'origine a été écrit par Andreï Kolmogorov et Yuri Ofman, mais il a été publié sous les noms de Karatsuba et d'Ofman. Kolmogorov, ainsi que beaucoup d'autres, pensaient que l'algorithme naïf en n^2 était le meilleur possible. Kolmogorov a voulu rendre hommage ainsi à Karatsuba pour avoir résolu le problème du célèbre chercheur.

5.3.1 L'algorithme standard

Pour comprendre l'algorithme, rappelons l'algorithme standard, celui qu'on apprend à l'école élémentaire. Il est illustré par l'exemple suivant en base 10 et en base 2.

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ + 12 . \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times 1101 \\ \hline 1100 \\ 1100 . \\ + 1100 . . \\ \hline 1001100 \end{array}$$

L'algorithme est évidemment le même quelle que soit la base, un entier qu'on notera B et qu'on supposera ≥ 2 . Pour $B = 2$ l'algorithme se résume en fait à recopier ou décaler le premier opérande suivante que l'on a à multiplication par chiffre 1 ou 0. Lorsqu'on multiplie un nombre de n chiffres par un autre de m chiffres, le résultat est *a priori* un nombre de $n + m$ chiffres. C'est différent pour $B = 1$, le codage unaire se comportant différemment pour la multiplication. On supposera que B est une valeur fixée indépendante de la taille des nombres manipulés. C'est donc une constante.

Soit z un entier naturel de n chiffres en base B . Sa *représentation numérique* est la suite (z_{n-1}, \dots, z_0) de ces chiffres, des entiers $z_i \in [0, B[$, tels que :

$$z = z_{n-1} \cdot B^{n-1} + \cdots + z_2 \cdot B^2 + z_1 \cdot B + z_0 = \sum_{i=0}^{n-1} z_i \cdot B^i.$$

Le problème de la multiplication de grands entiers peut se formaliser ainsi :

MULTIPLICATION

Instance: Deux entiers x et y de n chiffres écrits en base $B \geq 2$

Question: Donner les $2n$ chiffres de $z = x \times y$ en base B

L'algorithme standard pour la multiplication de x par y consiste donc à considérer chacun des chiffres y_i de y , à calculer le produit partiel $(x \cdot y_i) \cdot B^i$, puis à l'ajouter à la somme courante qui contiendra à la fin le résultat souhaité.

Les grands nombres sont représentés en machine par un simple tableau d'entiers de $[0, B[$. Du coup un produit par une puissance de la base du type $p = z \cdot B^i$ n'est pas une opération arithmétique mais un simple décalage : il suffit d'écrire z au bon endroit dans le tableau de chiffres représentant p . D'ailleurs avec la méthode apprise à l'école on met un '.' à chaque nouveau chiffre pour simuler le décalage qui dans les faits ne coutent rien.

Complexité. Supposons que les nombres x, y ont n chiffres chacun. L'algorithme effectue n fois (pour chaque chiffres de y), une multiplication d'un nombre de taille n par un seul chiffre, ainsi qu'une addition de deux nombres d'au plus $2n$ chiffres. La multiplication par un chiffre et l'addition prennent un temps $O(n)$. Au total l'algorithme est donc en $O(n^2)$.

On a l'impression, tout comme Kolmogorov, que chaque chiffre de x doit « rencontrer » chaque chiffre de y , ce qui nécessite $\Omega(n^2)$ opérations. Et pourtant ...

5.3.2 Approche diviser pour régner

Pour simplifier, on va supposer que x et y comportent tous les deux n chiffres. Si tel n'était pas le cas, on complète par des zéros à gauche la représentation du plus court des deux nombres.

L'idée est de découper chaque suite de n chiffres en deux sous-suites : l'une comprenant les $m = \lceil n/2 \rceil$ chiffres les moins significatifs, et l'autre les $n - m = \lfloor n/2 \rfloor$ chiffres les plus significatifs. Puis on fait une multiplication par blocs. Par exemple, si $x = 1234$ et $y = 9876$ alors on découpe $x = \boxed{12} \boxed{34}$ et $y = \boxed{98} \boxed{76}$. Puis on fait une multiplication standard sur 2 chiffres en base $B = 100$.

$$\begin{array}{r}
 & \begin{array}{|c|c|}\hline 12 & 34 \\ \hline 98 & 76 \\ \hline \end{array} \\
 \times & \begin{array}{|c|c|}\hline 09 & 37 & 84 \\ \hline 12 & 09 & 32 & \dots \\ \hline \end{array} \\
 \hline
 + & \begin{array}{|c|c|}\hline 12 & 18 & 69 & 84 \\ \hline \end{array}
 \end{array}$$

Plus formellement, si $n > 1$, on découpe $x = \boxed{x^+} \boxed{x^-}$ avec $x^+ = (x_{n-1}, \dots, x_m)$ et $x^- = (x_{m-1}, \dots, x_0)$. Le bloc x^- possède exactement $m \geq 1$ chiffres alors que x^+ en possède $n - m \geq 1$. En choisissant $m = \lceil n/2 \rceil$, on a $n - m = \lfloor n/2 \rfloor$. Du coup x^- et x^+ ont tous les deux au plus m chiffres. On a alors :

$$x = \boxed{x^+} \boxed{x^-} = x^+ \cdot B^m + x^-$$

comme si on représentait x en base B^m et que x avait deux chiffres : $\boxed{x^+}$ et $\boxed{x^-}$. En découpant de manière similaire $y = \boxed{y^+} \boxed{y^-} = y^+ \cdot B^m + y^-$, le produit s'exprime alors :

$$\begin{aligned}
 x \times y &= \boxed{x^+} \boxed{x^-} \times \boxed{y^+} \boxed{y^-} = (x^+ \cdot B^m + x^-) \times (y^+ \cdot B^m + y^-) \\
 &= (x^+ \times y^+) \cdot B^{2m} + (x^+ \times y^- + x^- \times y^+) \cdot B^m + x^- \times y^-.
 \end{aligned}$$

On a donc remplacé un produit de deux nombres de n chiffres, par 4 produits d'au plus $\lceil n/2 \rceil$ chiffres, 4 sommes sur des nombres d'au plus $2n$ chiffres et 2 décalages (correspondant aux multiplications par B^{2m} et par B^m). Plus formellement, l'algorithme s'écrit :

 Algorithme $\text{MUL}_{\text{REC}}(x, y)$

Entrée: x, y deux entiers naturels de n chiffres écrits en base B .

Sortie: $x \times y$ sur $2n$ chiffres.

1. Si $n = 1$, renvoyer ¹⁸ $x_0 \cdot y_0$

2. Soit $m = \lceil n/2 \rceil$. Poser :

$$\begin{aligned} x^+ &= (x_{n-1}, \dots, x_m) \text{ et } x^- = (x_{m-1}, \dots, x_0) \\ y^+ &= (y_{n-1}, \dots, y_m) \text{ et } y^- = (y_{m-1}, \dots, y_0) \end{aligned}$$

3. Calculer :

$$p_1 = \text{MUL}_{\text{REC}}(x^+, y^+)$$

$$p_2 = \text{MUL}_{\text{REC}}(x^+, y^-)$$

$$p_3 = \text{MUL}_{\text{REC}}(x^-, y^+)$$

$$p_4 = \text{MUL}_{\text{REC}}(x^-, y^-)$$

$$a = p_2 + p_3$$

4. Renvoyer $p_1 \cdot B^{2m} + a \cdot B^m + p_4$

Dans l'algorithme ci-dessus aux l'étapes 3 et 4, on a noté de manière abusive par « + » l'addition de grands nombres, c'est-à-dire sur les tableaux. C'est une opération triviale non détaillée ici qui peut être réalisée par un simple parcours linéaire des tableaux.

Pour être rigoureux et respecter le format de l'entrée, il faudrait que lors des appels récursifs on soit certain que les nombres aient bien exactement le même nombre de chiffres. C'est le cas pour p_1 qui utilise deux opérandes de $\lfloor n/2 \rfloor$ chiffres et p_4 qui utilise deux opérandes de $\lceil n/2 \rceil$ chiffres. Mais ce n'est potentiellement pas le cas pour p_2 et p_3 où x^- et y^- pourraient comprendre un chiffre de plus que x^+ et y^+ . Il faut donc éventuellement ajouter un zéro à gauche pour x^+ et y^+ si tel n'était pas le cas.

Complexité. Soit $T(n)$ la complexité en temps de l'algorithme $\text{MUL}_{\text{REC}}(x, y)$ appliqué à des nombres de n chiffres. Toutes les opérations, sauf les appels récursifs, prennent un temps au plus $O(n)$. Les appels récursifs s'appliquent à des nombres d'au plus $m = \lceil n/2 \rceil$ chiffres. On peut donc borner le temps de chacun de ces appels récursifs par $T(\lceil n/2 \rceil)$, même si certains appels ne font que $T(\lfloor n/2 \rfloor)$, car clairement T est une fonction croissante. Donc $T(n)$ vérifie l'équation de récurrence :

$$T(n) = 4 \cdot T(\lceil n/2 \rceil) + O(n) \tag{5.3}$$

avec $T(1) = O(1)$. Cela ressemble beaucoup à l'équation (5.1) déjà rencontrée qui était $T(n) = 2 \cdot T(\lceil n/2 \rceil) + O(n)$. Malheureusement, on ne peut pas se resservir de la solution,

18. Si l'on devait exécuter l'algorithme à la main, le cas terminal consisterait simplement à lire le résultat dans une table de multiplication, qu'il faut malheureusement apprendre par cœur.

qui était $T(n) = O(n \log n)$, à cause du « 4 » qui change tout. Il faut donc recommencer l'analyse. On verra plus tard, qu'il y a un truc pour éviter de faire les calculs ...

Comme expliqué précédemment, il ne faut pas utiliser la notation asymptotique pour résoudre une récurrence. On a donc, pour une constante $c > 0$ suffisamment grande, les inéquations suivantes :

$$T(n) \leq \begin{cases} 4 \cdot T(\lceil n/2 \rceil) + cn & \text{si } n > 1 \\ c & \text{si } n \leq 1 \end{cases}$$

En négligeant la partie entière (cf. la section 5.4) et en dépliant la formule de récurrence, il vient :

$$\begin{aligned} T(n) &\leq 4 \cdot T(n/2) + cn \\ &\leq 4 \cdot [4 \cdot T((n/2)/2) + c \cdot (n/2)] + cn \\ &\leq 4^2 \cdot T(n/2^2) + (4/2) \cdot cn + cn \\ &\leq 4^2 \cdot [4 \cdot T((n/2^2)/2) + c \cdot (n/2^2)] + ((4/2) + 1) \cdot cn \\ &\leq 4^3 \cdot T(n/2^3) + ((4/2)^2 + (4/2) + 1) \cdot cn \\ &\dots \\ &\leq 4^i \cdot T(n/2^i) + \sum_{j=0}^{i-1} (4/2)^j \cdot cn \quad \forall i > 0 \end{aligned}$$

Ce qui est identique aux l'inéquations (5.2) en remplaçant le facteur « 2 » par le facteur « 4 » devant $T()$. En fait, de manière générale, si $T(n) \leq a \cdot T(n/b) + cn$ pour certains $a, b > 0$, alors :

$$T(n) \leq a^i \cdot T(n/b^i) + \sum_{j=0}^{i-1} (a/b)^j \cdot cn \quad \forall i > 0 \quad (5.4)$$

Ce qui s'obtient immédiatement en remplaçant $4 \rightarrow a$ et $2 \rightarrow b$ dans le calcul précédent.

Rappelons (cf. l'équation (1.4)) que :

$$\sum_{j=0}^{i-1} (4/2)^j = \sum_{j=0}^{i-1} 2^j = \frac{2^i - 1}{2 - 1} < 2^i .$$

On a précédemment vu que $n/2^i \leq 1$ lorsque $i = \lceil \log_2 n \rceil$, et la récurrence s'arrête sur le cas terminal $T(1)$. Cela donne :

$$\begin{aligned} T(n) &\leq 4^{\lceil \log_2 n \rceil} \cdot T(1) + 2^{\lceil \log_2 n \rceil} \cdot cn \\ &\leq 4^{(\log_2 n)+1} \cdot c + 2^{(\log_2 n)+1} \cdot cn \\ &\leq 4c \cdot 2^{\log_2 n} + 2n \cdot cn \\ &\leq 4c \cdot 2^{\log_2(n^2)} + O(n^2) \\ &\leq 4c \cdot n^2 + O(n^2) = O(n^2) . \end{aligned}$$

Le résultat est décevant, car on ne fait pas mieux que la méthode standard. Et en plus c'est compliqué. Mais seulement en apparence, les appels récursifs ayant tendance à compacter le code.

5.3.3 Karastuba

Pour que la méthode diviser pour régner donne de bons résultats, il faut souvent ruser. Couper naïvement en deux ne fait pas avancer la compréhension du problème, sauf si l'étape de « fusion » permet un gain significatif.

Pour le tri-fusion par exemple, c'est la fusion en temps $O(n)$ qui est maline. Pour la paire de points les plus proches c'est le calcul dans la bande S en temps $O(n)$ qui est rusé. Pour l'algorithme de Karastuba, l'idée est de faire moins d'appels récursifs quitte à perdre un temps $O(n)$ avant chaque appel.

Dans l'algorithme $\text{MUL}_{\text{REC}}(x, y)$ on utilise les quatre produits : $p_1 = x^+ \times y^+$, $p_2 = x^+ \times y^-$, $p_3 = x^- \times y^+$ et $p_4 = x^- \times y^-$. En y regardant de plus près on a en fait besoin de p_1 , p_4 et de $p_2 + p_3$.

L'idée est de remarquer que le produit

$$\begin{aligned} p &= (x^+ + x^-) \times (y^+ + y^-) &= x^+ \times y^+ + x^+ \times y^- + x^- \times y^+ + x^- \times y^- \\ &&= p_1 + p_2 + p_3 + p_4 \end{aligned}$$

contient les quatre produits souhaités. C'est en fait la somme. Donc si l'on calcule d'abord p_1 et p_4 , il suffit de calculer p pour avoir $p_2 + p_3 = p - (p_1 + p_4)$. Plus formellement, l'algorithme s'écrit :

Algorithme KARASTUBA(x, y)

Entrée: x, y deux entiers naturels de n chiffres écrits en base B .

Sortie: $x \times y$ sur $2n$ chiffres.

1. Si $n = 1$, renvoyer $x_0 \cdot y_0$

2. Soit $m = \lceil n/2 \rceil$. Poser :

$$\begin{aligned}x^+ &= (x_{n-1}, \dots, x_m) \text{ et } x^- = (x_{m-1}, \dots, x_0) \\y^+ &= (y_{n-1}, \dots, y_m) \text{ et } y^- = (y_{m-1}, \dots, y_0)\end{aligned}$$

3. Calculer :

$$\begin{aligned}p_1 &= \text{KARASTUBA}(x^+, y^+) \\p_4 &= \text{KARASTUBA}(x^-, y^-) \\a_1 &= x^+ + x^- \\a_2 &= y^- + y^+ \\p &= \text{KARASTUBA}(a_1, a_2) \\a &= p - (p_1 + p_4)\end{aligned}$$

4. Renvoyer $p_1 \cdot B^{2m} + a \cdot B^m + p_4$

Comme précédemment, les « + » et « - » dans les étapes 3 et 4 sont l'addition et la soustraction entre deux grands nombres qu'on suppose être des opérations connues. Comme pour l'addition, la soustraction s'effectue par un simple parcours linéaires des deux tableaux de chiffres.

Complexité. L'effet le plus notable, en comparant les deux algorithmes, est qu'on a remplacé un appel récursif par deux additions et une soustraction. C'est un petit détail qui va profondément changer la résolution de la récurrence dans l'analyse de la complexité.

Comme précédemment, on note $T(n)$ la complexité en temps de l'algorithme analysé, ici KARASTUBA appliqué à des nombres de n chiffres. Encore une fois, toutes les opérations, sauf les appels récursifs, prennent un temps $O(n)$. Deux appels utilisent des nombres d'au plus $m = \lceil n/2 \rceil$ chiffres (pour p_1 et p_4), cependant le calcul de $p = a_1 \times a_2$ utilise des nombres de $m + 1 = \lceil n/2 \rceil + 1$ chiffres. En effet, l'addition d'un nombre de n_1 chiffres avec un nombre de n_2 chiffres fait *a priori* $\max\{n_1, n_2\} + 1$ chiffres [Question. Pourquoi?]. Il suit que $T(n)$ vérifie l'équation de récurrence suivante :

$$T(n) = 3 \cdot T(\lceil n/2 \rceil + 1) + O(n) \quad (5.5)$$

ce qui en enlevant la notation asymptotique donne :

$$T(n) \leq \begin{cases} 3 \cdot T(\lceil n/2 \rceil + 1) + cn & \text{si } n > 1 \\ c & \text{si } n \leq 1 \end{cases}$$

pour une constante $c > 0$ suffisamment grande. En négligeant les constantes additives dans le paramètre de $T()$ (cf. la section 5.4) et en dépliant la formule de récurrence comme vue dans (5.4) avec $a = 3$ et $b = 2$, il vient directement :

$$T(n) \leq 3 \cdot T(n/2) + cn \leq 3^i \cdot T(n/2^i) + \sum_{j=0}^{i-1} (3/2)^j \cdot cn \quad \forall i > 0.$$

Rappelons (cf. l'équation (1.4)) que :

$$\sum_{j=0}^{i-1} (3/2)^j = \frac{(3/2)^i - 1}{(3/2) - 1} < 2 \cdot (3/2)^i.$$

On a précédemment vu que $n/2^i \leq 1$ lorsque $i = \lceil \log_2 n \rceil$, et la récurrence s'arrête sur le cas terminal $T(1)$. Cela donne :

$$\begin{aligned} T(n) &\leq 3^{\lceil \log_2 n \rceil} \cdot T(1) + 2 \cdot (3/2)^{\lceil \log_2 n \rceil} \cdot cn \\ &\leq 3^{\lceil \log_2 n \rceil + 1} \cdot c + 2 \cdot (3/2)^{\lceil \log_2 n \rceil + 1} \cdot cn \\ &\leq 3c \cdot 3^{\log_2 n} + 4c \cdot (3/2)^{\log_2 n} \cdot n \end{aligned}$$

On va utiliser le fait que¹⁹ $x^{\log_b y} = y^{\log_b x}$. Donc $3^{\log_2 n} = n^{\log_2 3}$ et $(3/2)^{\log_2 n} \cdot n = n^{\log_2(3/2)+1} = n^{\log_2(3/2)+\log_2(2)} = n^{\log_2(2 \cdot 3/2)} = n^{\log_2 3}$. D'où :

$$T(n) \leq 3c \cdot n^{\log_2 3} + 4c \cdot n^{\log_2 3} = O(n^{\log_2 3}) = O(n^{1.59})$$

car $\log_2 3 = 1.5849625\dots$

C'est significativement plus rapide lorsque n est grand. Dans le tableau de comparaison du paragraphe 5.2.5 qui compare différentes complexités et temps d'exécution pour $n = 10^9$, on passerait ainsi de 30 ans pour un algorithme en n^2 à 51 heures pour l'algorithme en $n^{\log_2 3}$. Bien sûr, il n'est pas dit qu'en pratique on soit amené à multiplier des nombres d'un milliard de chiffres. Cependant, pour des clés cryptographiques de l'ordre du Mo, $n = 10^6$ est plausible. Dans ce cas on passerait de 16 minutes à 3 secondes.

Parenthèse. Il existe des algorithmes encore plus rapides. Ils sont basés sur la transformée de Fourier rapide (FFT pour Fast Fourier Transform), donnant l'algorithme de Schönhage–Strassen de complexité $O(n \log n \log \log n)$ [SS71]. On ne le détaillera pas. En fait, on ne sait toujours pas s'il est possible de multiplier des entiers en temps linéaires. On pense que cela n'est pas possible, mais le passé montre qu'on s'est parfois trompé. Il est conjecturé que le meilleur algorithme possible doit avoir une complexité de $\Omega(n \log n)$. Mais cela n'est pas prouvé. Il existe une borne inférieure en $\Omega(n \log n)$ pour la version on-line de la multiplication, pour laquelle on impose que le k -ième chiffre du produit soit écrit avant la lecture

19. En effet, pour toute base $b > 1$, $x = b^{\log_b x}$. Donc $x^{\log_b y} = b^{(\log_b x)(\log_b y)} = b^{(\log_b y)(\log_b x)} = y^{\log_b x}$.

du $(k+1)$ -ième chiffre des opérandes [PFM74][vdH14]. Bien sûr, il n'y a aucune raison que le meilleur des algorithmes procède ainsi.

L'algorithme le plus rapide, due à [HvdH19] en 2019, a une complexité de $O(n \log n)$. Le précédent record était celui de [Für09] avec une complexité de $(n \log n) \cdot 2^{O(\log^* n)}$ où $\log^* n = \min\{i \geq 0 : \log^{(i)} n\}$ est une fonction qui croît extrêmement lentement. Plus formellement, $\log^* n = \min\{i \geq 0 : \log^{(i)} n\}$ avec $\log^{(i)} n = \log(\log^{(i-1)} n)$ et $\log^{(0)} n = n$ est l'itéré de la fonction \log . En pratique $\log^* n \leq 5$ pour tout n inférieur au nombre de particules dans l'Univers.

5.4 Master Theorem

Au travers des exemples de ce chapitre (et même avant), on a vu que la complexité en temps $T(n)$ d'un algorithme pouvait s'exprimer par une équation (ou inéquation) de récurrence. Bien souvent l'équation ressemble à ceci :

$$T(n) \leq a \cdot T(n/b + c) + f(n) \quad (5.6)$$

où $a \geq 1$, $b > 1$, $c \geq 0$ sont des constantes et $T(n)$ et $f(n)$ sont des fonctions croissantes. La constante a correspond aux nombres d'appels (ou branchements) récursifs, b est le nombre par lequel on divise le problème initial, et $f(n)$ le temps de fusion des solutions partielles. Enfin, c permet de gérer les parties entières supérieures ou inférieures. En fait il existe un théorème qui donne la forme générale de la solution, plus exactement l'asymptotique.

Théorème 5.1 (Master Theorem) Pour toute fonction entière $T(n)$ vérifiant l'équation (5.6) avec $\lambda = \log_b a$, alors :

1. Si $f(n) = O(n^{\lambda-\varepsilon})$ pour une constante $\varepsilon > 0$, alors $T(n) = \Theta(n^\lambda)$.
2. Si $f(n) = \Theta(n^\lambda)$, alors $T(n) = \Theta(n^\lambda \log n)$.
3. Si $f(n) = \Omega(n^{\lambda+\varepsilon})$ pour une constante $\varepsilon > 0$ et si $a \cdot f(n/b + c) \leq q \cdot f(n)$ pour une constante $q < 1$, alors $T(n) = \Theta(f(n))$.

Cela a l'air compliqué, mais on peut déchiffrer simplement le résultat comme suit. Comme on va le voir, l'exposant $\lambda = \log_b a$ est une valeur critique dans l'asymptotique de $T(n)$. Il y a trois cas, et dans chacun d'eux on compare $f(n)$ à n^λ .

Cas 1 : Si $f(n)$ est plus petite que n^λ , alors c'est n^λ qui « gagne », c'est-à-dire qui contribue le plus à l'asymptotique de $T(n)$.

Cas 3 : Si $f(n)$ est plus grande que n^λ , alors c'est $f(n)$ qui contribue le plus à l'asymptotique de $T(n)$, moyennant une certaine condition sur la croissance de f .

Cas 2 : Si $f(n)$ est de l'ordre de n^λ , alors $f(n)$ (ou bien n^λ) contribue $\Theta(\log n)$ fois à $T(n)$, d'où le facteur supplémentaire en $\log n$.

La comparaison formelle des fonctions $f(n)$ et n^λ , en fait des asymptotiques, se fait en jouant sur l'exposant avec ε . Mais l'idée est bien de dire : soit $f(n) \ll n^\lambda$, soit $f(n) \approx n^\lambda$, soit $f(n) \gg n^\lambda$.

5.4.1 Exemples d'applications

Dans ce cours, on a déjà vu trois exemples :

$$T(n) \leq 2 \cdot T(\lceil n/2 \rceil) + O(n)$$

On peut prendre $a = b = 2$, $c = 1$ (car $\lceil n/2 \rceil \leq n/2 + 1$) et $f(n) = \Theta(n)$. Alors $\lambda = \log_b a = 1$, et donc $n^\lambda = n$. Il vient que $T(n) = \Theta(n \log n)$.

$$T(n) \leq 4 \cdot T(\lceil n/2 \rceil) + O(n)$$

On peut prendre $a = 4$, $b = 2$, $c = 1$ et $f(n) = O(n)$. Alors $\lambda = \log_b a = 2$, et donc $n^\lambda = n^2$. Il vient que $T(n) = \Theta(n^2)$.

$$T(n) \leq 3 \cdot T(\lceil n/2 \rceil + 1) + O(n)$$

On peut prendre $a = 3$, $b = 2$, $c = 2$ et $f(n) = O(n)$. Alors $\lambda = \log_b a = \log_2 3$, et donc $n^\lambda < n^{1.59}$. Il vient que $T(n) = O(n^{1.59})$.

Une autre récurrence qu'on rencontre souvent, typiquement lors d'une recherche dichotomique, est la suivante :

$$T(n) \leq T(\lceil n/2 \rceil) + O(1)$$

On peut prendre $a = 1$, $b = 2$, $c = 1$ et $f(n) = O(1)$. Alors $\lambda = \log_b a = 0$, et donc $n^\lambda = 1$. Il vient que $T(n) = \Theta(\log n)$.

5.4.2 Explications

L'intuition derrière l'exposant $\lambda = \log_b a$ est la suivante. Lorsqu'on « déroule » i fois la récurrence de $T(n)$, il vient un terme en $a^i \cdot T(n/b^i)$, en négligeant la constante c dans $T(n/b^i + c)$. Les appels récursifs de l'algorithme, et donc la récurrence, s'arrêtent lorsque n/b^i devient constant, c'est-à-dire lorsque cette valeur est suffisamment petite. Et dans ce cas $T(n/b^i)$ est aussi constant, car alors l'algorithme travaille sur un problème de taille constante. Dans un premier temps, et pour simplifier, disons que $T(1) \leq 1$. Calculons le nombre minimum d'étapes i_0 pour avoir $n/b^{i_0} \leq 1$. D'après la proposition 1.1 (voir le paragraphe 1.6), on sait que $i_0 = \lceil \log_b n \rceil$. En effet,

$$n/b^{i_0} \leq 1 \Leftrightarrow n \leq b^{i_0} \Leftrightarrow \log_b n \leq i_0.$$

Donc après $i_0 = \lceil \log_b n \rceil$ appels récursifs l'algorithme s'arrête sur le cas terminal. Apparaît alors dans $T(n)$ un terme en :

$$\begin{aligned} a^{i_0} \cdot T(n/b^{i_0}) &\leqslant a^{i_0} \cdot T(1) \leqslant a^{\lceil \log_b n \rceil} < a^{(\log_b n)+1} \\ &\leqslant a \cdot a^{\log_b n} = a \cdot n^{\log_b a} = \Theta(n^\lambda) \end{aligned}$$

en utilisant la croissance de T , le fait que $a \geqslant 1$ et $b > 1$ sont des constantes et le fait que $x^{\log_b y} = y^{\log_b x}$ (cf. la note de bas de page ¹⁹). On se rend compte d'ailleurs que si on avait pris comme condition terminale $T(c_1) \leqslant c_2$ pour des constantes positives c_1, c_2 au lieu de $T(1) \leqslant 1$, alors on aurait eu le terme $c_2 \cdot a \cdot (n/c_1)^{\log_b a} = \Theta(n^\lambda)$ ce qui ne change pas la valeur asymptotique.

Bien sûr, il manque la contribution de $f(n)$ dans $T(n)$. En négligeant la constante c , c'est la somme :

$$f(n) + a \cdot f(n/b) + a^2 \cdot f(n/b^2) + \cdots + a^{i_0} \cdot f(1) = \sum_{i=0}^{i_0} a^i \cdot f(n/b^i). \quad (5.7)$$

On retrouve le dernier terme en $a^{i_0} \cdot f(1) = O(a^{i_0})$ car $f(1) = O(1)$, qui on l'a vu vaut $\Theta(n^\lambda)$. Le nombre de termes de la somme est $i_0 + 1 = \lceil \log_b n \rceil + 1 = \Theta(\log n)$. Suivant la croissance de $f(n)$, la somme peut valoir $\Theta(n^\lambda)$, $\Theta(n^\lambda \log n)$ ou encore $\Theta(f(n))$.

Si $f(n)$ est assez petit, alors on aura $\Theta(n^\lambda)$ à cause du dernier terme qui vaut $\Theta(n^\lambda)$. Si $f(n)$ est juste autour de n^λ alors on va avoir près le $i_0 = \Theta(\log n)$ termes de l'ordre de n^λ .

Dans le cas 3, en itérant la condition $a \cdot f(n/b) \leqslant q \cdot f(n)$ avec $q < 1$, et en supposant toujours que $c = 0$, on peut alors majorer :

$$\begin{aligned} a \cdot f(n/b) &\leqslant q \cdot f(n) \\ \Rightarrow a \cdot f((n/b)/b) &\leqslant q \cdot f(n/b) \\ \Rightarrow a^2 \cdot f(n/b^2) &\leqslant q \cdot a \cdot f(n/b) \leqslant q^2 \cdot f(n) \\ \Rightarrow a^i \cdot f(n/b^i) &\leqslant q^i \cdot f(n). \end{aligned}$$

Donc sous cette condition, la somme (5.7) se majore par (rappelons que $q < 1$ est une constante ²⁰) :

$$\sum_{i=0}^{i_0} a^i \cdot f(n/b^i) < \left(\sum_{i=0}^{+\infty} q^i \right) \cdot f(n) = \frac{1}{1-q} \cdot f(n) = O(f(n)).$$

La condition $a \cdot f(n/b) \leqslant q \cdot f(n)$ est technique mais pas restrictive en pratique. Par exemple, si $f(n) = n^p$ pour un certain exposant $p > \lambda$ assez grand, disons $p = 2\lambda =$

²⁰ 20. La formule qu'on utilise pour $\sum_{i=0}^{+\infty} q^i$ est la limite pour $n \rightarrow +\infty$ de la formule (1.4) déjà vue : $\sum_{i=0}^n q^i = (q^{n+1} - 1)/(q - 1)$. Comme $q < 1$, $q^{n+1} \rightarrow 0$, et on retrouve la limite $1/(q - 1)$.

$2\log_b a$. Alors la condition devient (en observant que $b^{\log_b a} = a$) :

$$a \cdot f(n/b) = a \cdot \left(\frac{n}{b}\right)^p = \frac{a}{b^p} \cdot n^p = \frac{a}{b^{2\log_b a}} \cdot n^p = \frac{1}{a} \cdot f(n)$$

ce qui donne bien $q < 1$ dès que $a > 1$.

Et si $c > 0$? Examinons le cas où $c > 0$. Précédemment (avec $c = 0$), en déroulant $i > 0$ fois la récurrence, les paramètres en n dans $T(n)$ et $f(n)$ évoluaient ainsi :

$$n \mapsto n/b \mapsto (n/b)/b = n/b^2 \mapsto \dots \mapsto n/b^i.$$

En tenant compte de c , ils évoluent en fait plutôt comme ceci :

$$\begin{aligned} n &\mapsto n/b + c \mapsto (n/b + c)/b + c = n/b^2 + c(1/b + 1) \\ &\mapsto (n/b^2 + c(1/b + 1))/b + c = n/b^3 + c(1/b^2 + 1/b + 1) \\ &\dots \\ &\mapsto n/b^i + c \sum_{j=0}^{i-1} 1/b^j \leq n/b^i + cb/(b-1). \end{aligned}$$

car on remarque²⁰ que $\sum_{j=0}^{i-1} 1/b^j < b/(b-1)$ car $b > 1$. Donc le terme que l'on obtient en déroulant i la récurrence est :

$$T(n) \leq a^i \cdot T(n/b^i + cb/(b-1)) + \sum_{j=0}^{i-1} a^j \cdot f\left(n/b^j + c \sum_{k=0}^{j-1} 1/b^k\right).$$

La remarque importante est que les termes supplémentaires $cb/(b-1)$ et $c \sum_k 1/b^k$ sont constants car b est une constante > 1 . Intuitivement, la différence avec le cas $c = 0$ sera donc *a priori* minime.

En prenant, comme précédemment, $i_0 = \lceil \log_b n \rceil$, on a $n/b^{i_0} \leq 1$ et mais aussi que $n/b^{i_0} + cb/(b-1) \leq 1 + cb/(b-1) = O(1)$ car $c \geq 0$ et $b > 1$ sont des constantes. Du coup la première partie devient

$$\begin{aligned} a^{i_0} \cdot T(n/b^{i_0} + cb/(b-1)) &\leq a^{\lceil \log_b n \rceil} \cdot T(1 + cb/(b-1)) \\ &= O(a^{\log_b n}) \cdot T(O(1)) = O(n^{\log_b a}) = O(n^\lambda). \end{aligned}$$

Ce qui n'a rien changé. Il en va de même pour le second terme. On peut vérifier par exemple que dans le cas 3, la condition $a \cdot f(n/b + c) \leq q \cdot f(n)$ implique $a^j \cdot f(n/b^j + \sum_k 1/b^k) \leq q^j \cdot f(n)$. Et donc que la majoration précédente ($\sum_{i=0}^{+\infty} q^i \cdot f(n) = O(f(n))$) reste valable.

On peut trouver la preuve complète et formelle du *Master Theorem* dans [CLRS01] par exemple.

5.4.3 D'autres récurrences

Bien sûr le *Master Theorem* ne résout pas toutes les récurrences. Par exemple $T(n) = a \cdot T(n - b) + f(n)$ qui se résout en $T(n) = \Theta(a^{n/b} \cdot \sum_{i=0}^{n/b} f(n - ib)) = O(a^{n/b} \cdot n \cdot f(n))$. Il faut $a, b > 0$. Le cas $a = 1$ avec $T(n) = \Theta(n \cdot f(n))$ est fréquent.

On peut évidemment imaginer des tas d'autres récurrences, comme $T(n) = T(\lceil 2\sqrt{n} \rceil) + f(n)$ ou encore $T(n) = a_1 \cdot T(n/b_1) + a_2 \cdot T(n/b_2) + f(n)$. Il n'y a alors plus forcément de formule asymptotique simple. On rencontre même parfois des récurrences du type $T(n) = T(T(n/2)) + 1$.

Bien évidemment il y a autant de récurrences que de programmes récursifs possibles.

5.5 Calcul du médian

5.5.1 Motivation

On s'en sert pour implémenter efficacement certains algorithmes de tri.

Parenthèse. *C'est quoi un algorithme naïf? En général, c'est un algorithme dont le principe est élémentaire, et pour lequel il est très simple de se convaincre qu'il marche. On donne donc priorité à la simplicité (simple à coder ou à exécuter à la main ou simple à montrer qu'il marche) plutôt qu'à l'efficacité. Il peut arriver qu'un algorithme naïf soit également efficace. Malheureusement, la règle générale est que pour être efficace il vaut mieux utiliser la « ruse ».*

Voici quelques algorithmes naïfs pour trier un tableau de n éléments. On va supposer qu'on trie par ordre croissant et par comparaisons – à l'aide d'une fonction donnée de comparaison, un peu la fonction `f()` passée en paramètre à `qsort()`, et on compte le nombre d'appels à cette fonction `f()`. Par exemple on souhaite trier des nombres réels, des chaînes de caractères, des entrées d'une base de données (combinaisons d'attributs comme sexe-age-nom). On exclue donc les tris par comptages, très efficaces selon le contexte, comme par exemple le tri de copies selon une notes entière de $[0, 20]$.

(1) « Tant qu'il existe deux éléments mal rangés on les échange. »

Il est clair qu'à la fin le tableau est trié, mais cela n'est pas très efficace. Il faut trouver une telle paire mal ordonnée et faire beaucoup d'échanges.

Trouver une paire peut nécessiter $\binom{n}{2} = \Theta(n^2)$ opérations si l'on passe en revue toutes les paires. Bien sûr on peut être un peu plus malin, en raffinant l'algorithme naïf, en observant que s'il existe une paire d'éléments mal rangés, alors il en existe une où les éléments sont consécutifs dans le tableau ce qui prend un temps $O(n)$ et pas $O(n^2)$ par échanges.

Et puis le nombre d'échanges peut-être grand. Combien ? Sans doute beaucoup si on ne prête pas attention à l'ordre dans lequel on opère les échanges. En effet, un élément donné peut au cours de l'algorithme se rapprocher puis s'éloigner (et ceci plusieurs fois) de sa position finale. En raffinant l'algorithme encore un peu on peut effectuer les échanges à la suite en se dirigeant vers le début du tableau. C'est un peu le tri-par-bulles qui est en $O(n^2)$.

- (2) « Chercher le plus petit élément et le placer au début, puis recommencer avec le reste du tableau. »

C'est l'algorithme du tri-par-sélection où l'on construit le tableau final trié progressivement élément par élément à partir de la gauche. Cette construction linéaire permet de se convaincre que le tableau est correctement ordonné à la fin de l'algorithme. La complexité est en $O(n^2)$ ce qui est atteint lorsque les éléments sont rangés dans l'ordre décroissant.

Ces algorithmes naturels ont la propriété de trier « en place ». Les éléments sont triés en effectuant des déplacements dans le tableau lui-même, sans l'aide d'un tableau auxiliaire. C'est une propriété clairement souhaitable si l'on pense que les algorithmes de tri sont utilisés pour trier des bases de données (très grand fichier Excel) selon certains attributs (colonnes). Pour des raisons évidentes de place mémoire, on ne souhaite pas (et souvent on ne peut pas), faire une copie de la base de données juste pour réordonner les entrées. Notons que les tris par comptage utilisent un espace mémoire auxiliaire (pour le comptage justement) qui dépend de l'intervalle des valeurs possibles (potentiellement grand donc).

En ce qui concerne les algorithmes efficaces, on pense à celui issu de la méthode « diviser pour régner », le tri-fusion évoqué en début de chapitre. On coupe en deux tableaux de même taille que l'on trie récursivement, puis on les fusionne. La récurrence sur la complexité est $T(n) = 2 \cdot T(n/2) + O(n)$ ce qui donne $O(n \log n)$. Mais l'algorithme nécessite un tableau auxiliaire.

Il y aussi le tri-rapide (*qsort*) : on choisit un élément particulier, le pivot, et on déplace les éléments avant ou après le pivot selon qu'ils sont plus petits ou plus grands que le pivot. Ce déplacement peut se faire « en place » en temps $O(n)$. Puis on récuse sur les deux tableaux de part et autre du pivot. En pratique il est efficace (avec un choix du pivot aléatoire) mais sa complexité dans le pire des cas est en $O(n^2)$ car le pivot pourrait ne pas couper en deux tableaux de taille proche, mais en un tableau de taille 1 et un tableau de taille $n - 2$ par exemple. La récurrence sur la complexité est alors $T(n) = T(n - 2) + O(n)$ ce qui fait malheureusement du $O(n^2)$. L'idéal serait de prendre comme pivot le médian car il a la propriété de couper précisément en deux tableaux de même taille. La récurrence devient alors celle du tri-fusion, soit une complexité de $O(n \log n)$... à condition de trouver rapidement le médian. L'équation de récurrence nous informe qu'on peut, sans changer la complexité finale, se permettre de dépenser un temps $O(n)$ pour le trouver. D'où l'intérêt du problème.

Parenthèse. Il existe d'autres algorithmes de tri en place et qui ont une complexité en temps $O(n \log n)$. Ils sont généralement plus complexes à implémenter et ne donnent pas de meilleures performances en pratique que le tri-rapide. Citons par exemple, l'implémentation rusée du tri-par-tas qui construit le tas dans le tableau lui-même. (L'étape de remplissage peut même être faite en temps linéaire !) On peut trouver quelques détails sur ce tri page 117.

5.5.2 Tri-rapide avec choix aléatoire du pivot

[Cyril. À finir.]

5.5.3 Médian

[Cyril. À finir.]

5.6 Morale

- La technique « diviser pour régner » permet de construire des algorithmes auxquels on ne pense pas forcément de prime abord.
- Par rapport à une approche naïve (souvent itérative), ces algorithmes ne sont pas forcément meilleurs. Leur complexité peut être aussi mauvaise.
- Pour obtenir un gain, il est nécessaire d'avoir recours à une « astuce » de calcul permettant de combiner efficacement les solutions partielles (fusion). Idéalement la fusion devrait être de complexité inférieure à la complexité globale recherchée.
- La complexité $T(n)$ suit, de manière inhérente, une équation récursive qu'il faut résoudre (asymptotiquement). Dans de nombreux cas elle est de la forme $T(n) = a \cdot T(n/b) + f(n)$ pour un algorithme qui ferait a appels récursifs (ou branchements) sur des sous-problèmes de tailles n/b , avec un temps de fusion $f(n)$ des a sous-problèmes.
- Des résultats généraux permettent d'éviter de résoudre les équations de récurrences en se passant aussi des problèmes de partie entière. Il s'agit du *Master Theorem*.

Bibliographie

[CLRS01] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms (second edition)*, The MIT Press, 2001.

- [Für09] M. FÜRER, *Faster multiplication algorithm*, SIAM Journal on Computing, 39 (2009), pp. 979–1005. doi : [10.1137/070711761](https://doi.org/10.1137/070711761).
- [HvdH19] D. HARVEY AND J. VAN DER HOEVEN, *Integer multiplication in time $O(n \log n)$* , Tech. Rep. hal-02070778, HAL, March 2019.
- [PFM74] M. S. PATERSON, M. J. FISCHER, AND A. R. MEYER, *An improved overlap argument for on-line multiplication*, in Complexity of Computation, R. M. Karp, ed., vol. VII, SIAM-AMS proceedings, American Mathematical Society, 1974, pp. 97–111.
- [SS71] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation großer Zahlen*, Computing, 7 (1971), pp. 281–292. doi : [10.1007/BF02242355](https://doi.org/10.1007/BF02242355).
- [vdH14] J. VAN DER HOEVEN, *Faster relaxed multiplication*, in 39th International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, July 2014, pp. 405–412. doi : [10.1145/2608628.2608657](https://doi.org/10.1145/2608628.2608657).