MP2I
MPI

INFORMATIQUE

Cours et exercices corrigés

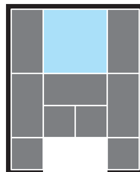
Thibaut Balabonski
Sylvain Conchon
Jean-Christophe Filliâtre
Kim Nguyen
Laurent Sartre

ellipses

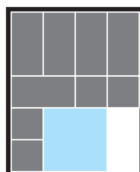
Chapitre 1

Avant-goût

C'est votre anniversaire et on vous offre « l'âne rouge », un magnifique casse-tête en bois composé de dix pièces que l'on peut déplacer à l'intérieur d'un cadre. Le cadre a une dimension 5×4 et deux cases sont laissées libres pour permettre le mouvement des pièces, à la manière d'un jeu de taquin. Initialement, les pièces sont disposées comme ceci :



La pièce de dimension 2×2, ici dessinée en bleu, est appelée l'« âne rouge » et donne son nom au casse-tête. L'objectif est de l'amener devant la « sortie » matérialisée en bas et au milieu du cadre. Ainsi, si par un mouvement successif des différentes pièces, on parvient à la configuration suivante



alors on a résolu le casse-tête. Mais il y a bien d'autres configurations gagnantes ! Essayez quelques déplacements à partir de la configuration de départ, et voyez si vous arrivez à vous approcher d'une solution. Vous observerez qu'à chaque étape on n'a qu'un tout petit nombre d'options possibles, mais que cela ne suffit pas à rendre le problème simple : essayer une voie, se retrouver coincé, revenir en arrière pour

essayer une nouvelle voie, tourner en rond… L'ensemble des positions possibles du jeu dessine un territoire finalement assez vaste, dans lequel l'orientation n'est pas évidente. On finirait par se poser la question :

- existe-t-il bien comme promis une solution à ce casse-tête ?

Cette question résiste un certain temps à l'exploration manuelle, et aucune théorie mathématique n'en donne directement la réponse. C'est là que l'informatique prend tout son sens : à l'aide d'algorithmes et de programmation, résoudre des problèmes que l'on ne saurait résoudre ni à la main, ni à l'aide de mathématiques. En l'occurrence, on peut confirmer l'existence d'une solution à l'aide d'un programme de quelques dizaines de lignes de code, écrit dans un langage de programmation répandu, et dont la durée d'exécution est plus courte qu'un clignement d'œil.

Dans cet ouvrage, nous donnons toutes les clés pour y parvenir. Nous verrons par exemple un algorithme de **parcours en profondeur**, qui permet ici d'explorer le labyrinthe que forment les configurations du jeu, en revenant en arrière lorsque l'on arrive à un cul de sac et en évitant d'explorer à nouveau des parties déjà vues. Cet algorithme permet ici de trouver, en deux dixièmes de seconde, une solution comportant 1171 coups ! Réaliser un tel algorithme nécessite, outre la connaissance d'un **langage de programmation**, l'utilisation de **structures de données** variées pour représenter et organiser les données manipulées. Pour représenter une configuration du jeu de sorte à pouvoir déterminer les prochains coups possibles, on utilise avantageusement une **liste** de blocs, chaque bloc étant lui-même décrit par une structure formée de quelques entiers : hauteur, largeur, coordonnées. Pour mémoriser l'ensemble des configurations qui ont déjà été vues et que l'on souhaite éviter d'explorer à nouveau, on peut cette fois convoquer une **table de hachage**.

Mais à peine l'existence d'une solution assurée, d'autres questions émergent :

- existe-t-il une solution plus courte ?
- quel est le nombre minimal de coups pour une solution ?

À nouveau, la réponse n'est pas immédiate, et résiste à l'analyse mathématique. Mais on peut enrichir notre programme pour y répondre. L'algorithme de **parcours en largeur** propose une manière différente d'explorer le jeu, en organisant les configurations à explorer dans une nouvelle structure de **file**. Ici, en une seconde de calcul cet algorithme permet de trouver une solution au casse-tête nettement plus courte, comportant 116 coups ! Mieux encore, des techniques de **raisonnement** sur les algorithmes et leurs propriétés permettent d'assurer que cette solution trouvée par le parcours en largeur est la plus courte possible.

Nous connaissons donc maintenant la meilleure solution à notre casse-tête. Mais le jeu doit-il vraiment s'arrêter là ? Nous pouvons prolonger en essayant des variantes :

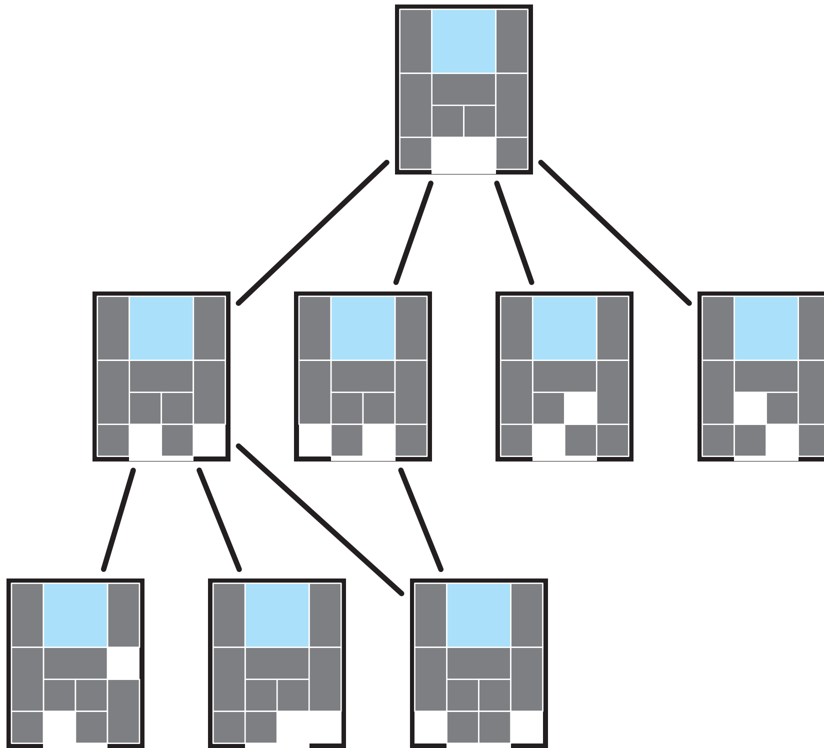


Figure 1.1 – Un tout petit morceau du graphe des états de l'âne rouge.

- peut-on encore atteindre une solution en partant de n'importe quelle configuration de départ ?
- et d'ailleurs, combien y a-t-il de configurations possibles ?
- peut-on résumer tout ceci dans une carte complète du jeu ? et si oui, quelle serait sa forme ?

Et on retourne à notre ordinateur, avec en main de nouveaux algorithmes et de nouvelles structures que vous découvrirez au fil des chapitres. Pour dénombrer toutes les configurations possibles du casse-tête, nous verrons un algorithme de **retour sur trace** qui nous permettra de déterminer qu'il y en a 65 880 au total, en trois secondes de calcul seulement. Mieux encore, on verra comment représenter une topographie complète des configurations du jeu sous la forme d'une structure de **graphe**, où toute configuration est reliée aux autres configurations atteignables en un coup. La figure 1.1 illustre un tout petit morceau de ce graphe. Au total, il contient 206 780 liaisons entre configurations. Une fois ce graphe construit, nous pourrons l'explorer avec différents algorithmes et déterminer en particulier qu'il ne se présente pas sous la forme d'un unique continent. On détecte au contraire 898 parties isolées les unes

des autres de tailles et de formes variées. Les deux plus grandes contiennent à elles seules près de 80% des configurations. D'autres sont réduites à de petits îlots solitaires ne comportant qu'une poignée de configurations. On y remarque surtout que certaines de ces parties ne contiennent aucune configuration gagnante ! Ainsi, on trouve des pans entiers du graphes depuis lesquels il serait impossible de résoudre le casse-tête.

Plusieurs exercices sont proposés dans cet ouvrage pour résoudre le problème de l'âne rouge et obtenir la plupart des valeurs énumérées ci-dessus. L'ensemble de ces exercices constitue un très bon projet. Il peut être adapté à beaucoup d'autres casse-têtes. On pourrait en outre continuer à énumérer de nouvelles questions (y a-t-il une configuration de départ « plus difficile », à partir de laquelle la solution minimale comporterait strictement plus que 116 coups ?), de nouveaux algorithmes, de nouvelles structures, et de nouvelles techniques de raisonnement.

21 p.121
128 p.489
147 p.596

Mais arrêtons-nous ici pour l'instant. Car il est maintenant temps pour vous d'entrer dans le vif du sujet, et d'explorer non pas un casse-tête en bois mais un autre vaste territoire, alliant science et technologie : l'informatique elle-même.

Chapitre 2

Notions d'architecture et de système

Nous donnons ici quelques rudiments d'architecture et de système. Cela ne saurait se substituer à un cours complet sur le sujet, mais cela aborde le minimum de connaissances requises par le reste de cet ouvrage.

2.1 Arithmétique des ordinateurs

Dans un ordinateur, toutes les informations (données ou programmes) sont représentées à l'aide de deux chiffres 0 et 1, appelés chiffres binaires ou *Binary Digits* en anglais (ou plus simplement *bits*).

Dans la mémoire d'un ordinateur (RAM, ROM, registres des micro-processeurs, etc.), ces chiffres binaires sont regroupés en *octets* (c'est-à-dire par « paquets » de 8, qu'on appelle *bytes* en anglais) puis organisés en *mots machine* (on dit *words* en anglais) de 2, 4 ou 8 octets (pour les machines les plus courantes). Par exemple, une machine dite de *64 bits* est un ordinateur qui manipule directement des mots de 8 octets ($8 \times 8 = 64$ *bits*) lorsqu'il effectue des opérations (en mémoire ou dans ses calculateurs).

Ce regroupement des *bits* en *octets* ou *mots machine* permet de représenter (et manipuler) d'autres données que des 0 ou des 1, comme par exemple des nombres entiers, des (approximations de) nombres réels, des caractères alpha-numériques ou des textes. Néanmoins, il est nécessaire d'inventer des *encodages* pour représenter ces informations.

2.1.1 Représentation des entiers

Encodage des entiers naturels. L'encodage le plus simple est celui des nombres entiers *naturels*. Il consiste simplement à *interpréter* un octet ou un mot machine comme un entier écrit en base 2. Dans cette base, les chiffres (0 ou 1) d'une séquence sont associés à un poids 2^i d'une puissance de 2 qui dépend de la position i des chiffres dans la séquence, de manière similaire à l'encodage en base 10. Par exemple, l'octet `01001101` peut être représenté en colonnes de la manière suivante.

séquence	0	1	0	0	1	1	0	1
positions	7	6	5	4	3	2	1	0
poids	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Vu comme un entier composé de huit chiffres binaires, cet octet correspond au nombre N calculé de la manière suivante :

$$\begin{aligned} N &= 0\times 2^7 + 1\times 2^6 + 0\times 2^5 + 0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 \\ &= 64+8+4+1 \\ &= 77. \end{aligned}$$

Plus généralement, une séquence $b_{n-1}b_{n-2}\dots b_1b_0$ de n *bits* b_i correspond au nombre N suivant.

$$N = \sum_{0\leqslant i<n} b_i \times 2^i.$$

Le chiffre b_{n-1} est appelé le chiffre ou le *bit* de *poids fort* et le chiffre b_0 est appelé le chiffre ou le *bit* de *poids faible*.

Cet encodage des entiers naturels par des séquences de n chiffres binaires permet donc de représenter tous les entiers de 0 à $2^n - 1$. On retiendra par exemple qu'un octet permet de représenter les entiers naturels de 0 à 255, ou encore que l'entier le plus grand représentable avec un mot de 16 *bits* est 65535.

Écriture en base 16. Une autre base fréquemment utilisée est la base 16, dite *hexadécimale*. Puisqu'il faut pouvoir écrire 16 chiffres hexadécimaux, on utilise les chiffres de 0 à 9 pour les 10 premiers, puis les lettres A, B, C, D, E et F pour les 6 derniers. La valeur de chaque lettre est donnée par le tableau de correspondance ci-dessous.

Unités de mesure

Il est très courant en informatique de mesurer la capacité mémoire d'un disque dur, de la RAM d'un ordinateur ou d'un débit de données Internet avec une unité de mesure exprimée comme un multiple d'octets. Ces multiples sont traditionnellement des puissances de 10 et on utilise les préfixes « kilo », « mega », etc. pour les nommer. Le tableau ci-dessous donne les principaux multiples utilisés dans la vie de tous les jours.

Nom	Symbole	Valeur
kilooctet	ko	10^3 octets
megaoctet	Mo	10^3 ko
gigaoctet	Go	10^3 Mo
teraoctet	To	10^3 Go

Historiquement, les multiples utilisés en informatique étaient des puissances de 2. Pour ne pas confondre l'ancienne et la nouvelle notation, on utilise des symboles différents pour représenter ces multiples.

Symbole	Valeur	Nombre d'octets
Kio	2^{10} octets	1024
Mio	2^{10} Kio	1 048 576
Gio	2^{10} Mio	1 073 741 824
Tio	2^{10} Gio	1 099 511 627 776

lettre	valeur
A	10
B	11
C	12
D	13
E	14
F	15

De manière similaire aux bases 2 et 10, on peut représenter les séquences de chiffres hexadécimaux en colonnes, en indiquant position et poids des chiffres. Par exemple, la séquence `2A4D` correspond à la représentation en colonnes suivante.

séquence	2	A	4	D
positions	3	2	1	0
poids	16^3	16^2	16^1	16^0

La valeur de cette séquence correspond donc au nombre

$$\begin{aligned} N &= 2 \times 16^3 + 10 \times 16^2 + 4 \times 16^1 + 13 \times 16^0 \\ &= 2 \times 4096 + 10 \times 256 + 4 \times 16 + 13 \\ &= 10829. \end{aligned}$$

La base 16 est souvent utilisée pour simplifier l'écriture de nombres binaires. En effet, on peut facilement passer d'un nombre en base 2 à un nombre en base 16 en regroupant les chiffres binaires par 4. Par exemple, la séquence de *bits* 1010010111110011 correspond au nombre hexadécimal `A5F3`, comme on peut le voir simplement de la manière suivante.

$$\overbrace{1\,0\,1\,0}^{A}\;\overbrace{0\,1\,0\,1}^{5}\;\overbrace{1\,1\,1\,1}^{F}\;\overbrace{0\,0\,1\,1}^{3}$$

On note que la transformation inverse est aussi très simple puisqu'il suffit de traduire chaque chiffre hexadécimal avec 4 *bits* selon le tableau de correspondance suivant.

chiffre hexadécimal	*bits*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

chiffre hexadécimal	*bits*
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

On utilise parfois la notation xxx_b pour indiquer la base b de l'écriture d'un nombre xxx, par exemple, pour ne pas confondre 101_2 (qui vaut 5) et 101_{16} (qui vaut 257).

Boutisme. La représentation *en machine* des entiers naturels sur des mots de 2, 4 ou 8 octets se heurte au problème de l'ordre dans lequel ces octets sont organisés en mémoire. Ce problème est appelé le *boutisme*[1] (ou *endianness* en anglais). Prenons l'exemple d'un mot de 2 octets (16 *bits*) comme `4CB6`. Il y a deux organisations possible d'un tel mot en mémoire :

- Le *gros boutisme* (ou *big endian* en anglais), qui consiste à placer l'octet de poids fort en premier, c'est-à-dire à l'adresse mémoire la plus petite.

...	4C	B6	...

- Le *petit boutisme* (ou *little endian* en anglais), qui au contraire place l'octet de poids faible en premier.

...	B6	4C	...

1. Ce terme vient d'une référence au roman de Jonathan Swift « Les Voyages de Gulliver » [paru en 1726] dans lequel il caricature les guerres de religion en faisant s'affronter deux clans, les petits et les gros boutistes, qui se disputent au sujet du bout par lequel il fallait manger les œufs.