Olivier Bartheye, Guy Alain Narboni

Retour sur un exemple historique en Prolog III : le découpage d’un rectangle en carrés de tailles distinctes

Volume 5, n° 2-3 (2024), p. 95-119.

https://doi.org/10.5802/roia.74

© Les auteurs, 2024.

Cet article est diffusé sous la licence
Creative Commons Attribution 4.0 International License.
http://creativecommons.org/licenses/by/4.0/

La Revue Ouverte d’Intelligence Artificielle est membre du Centre Mersenne pour l’édition scientifique ouverte
www.centre-mersenne.org
e-ISSN : 2967-9672

Revue Ouverte d'Intelligence Artificielle
Volume 5, nº 2-3, 2024, 95-119

Retour sur un exemple historique en Prolog III : le découpage d'un rectangle en carrés de tailles distinctes

Olivier Bartheye[a], Guy Alain Narboni[b]

[a] CREA école de l'air et de l'espace, base aérienne 701. F-13300 Salon-de-Provence (France)
E-mail : olivier.bartheye@ecole-air.fr

[b] Implexe, 63 Avenue de la Pointe Rouge, 13008 Marseille (France)
E-mail : r-d@implexe.eu.

Résumé. — L'objet de cet article est de rappeler cinquante ans plus tard l'intérêt de la programmation logique par contraintes selon un cas d'usage historique et pédagogique : le remplissage d'un rectangle de taille inconnue par un ensemble de carrés de tailles inconnues mais distinctes. L'obligation de différentiation des tailles conduit à se passer de toute hypothèse de symétrie permettant de résoudre plus facilement ce problème.

Il a été prouvé par un mathématicien polonais, Zbigniew Moroń en 1925, que le plus petit ensemble de carrés distincts pavant parfaitement un rectangle est de cardinal 9.

Nous commenterons le comportement du programme Prolog III conçu par Alain Colmerauer qui utilise le domaine des nombres rationnels pour offrir une version automatisée de cette preuve en exploitant une propriété démontrée par le mathématicien Max Dehn, élève de David Hilbert, en 1903 : il ne peut y avoir de solution que si les rapports entre les tailles sont commensurables.

Le programme se lance dans une étude de cas systématique mais fortement combinatoire à la recherche de placements. Il accumule au cours de son cheminement un double ensemble de contraintes additives (selon la hauteur et la largeur du rectangle), donnant lieu à la résolution globale d'un système d'équations et d'inéquations linéaires. Mais tant que ce système reste sous-déterminé, aucune solution partiellement instanciée ne nous aide à entrevoir l'issue du calcul.

Cet article effectue en quelque sorte une rétro-ingénierie des inférences bidimensionnelles effectuées, en explicitant les types de données issus de la méthode de dissection géométrique du problème. La discussion formelle et informelle qui prend place montre à la fois la puissance et la concision déclarative de la programmation logique par contraintes. Elle témoigne aussi de la difficulté à suivre pas à pas les inférences de la belle machinerie opérationnelle qui en coulisse conduit au résultat.

Mots-clés. — Programmation en logique avec contraintes, Casse-tête mathématique, Prolog, Contraintes linéaires sur les rationnels.

---

1. Introduction

Nous proposons dans cet article d'expliquer pédagogiquement un programme très connu écrit en Prolog III par Alain Colmerauer : le remplissage d'un rectangle par des carrés de tailles toutes distinctes. Publié dans « Une introduction à Prolog III » [2, 3] puis repris dans les manuels d'utilisation de Prolog III et de Prolog IV [9], ce petit programme est à lui seul un manifeste. Il met en valeur tout le potentiel qu'apporte, de manière inédite à l'époque, l'ouverture au numérique d'un langage symbolique comme Prolog.

À la fin des années 80, Prolog III incarne la renaissance de la programmation en logique sous une forme nouvelle : la programmation par contraintes. Son innovation majeure est l'adjonction de contraintes linéaires sur les nombres rationnels qui permet de marier élégamment inférences qualitatives et inférences quantitatives dans le développement de systèmes experts d'aide à la décision.

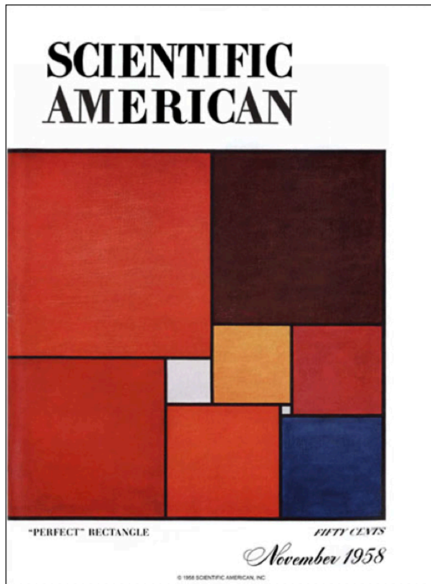
Peu après sa sortie, en 1991, Alain Colmerauer est invité de l'émission « Profession scientifique » de France Culture, animée par Philippe Boulanger et Emile Noël. À la question : « Quel problème avez-vous été content de résoudre ? », il répond de façon surprenante aux journalistes : « Un programme Prolog III qui découpe un rectangle de taille inconnue en des petits carrés de taille inconnue mais tous distincts. Un programme très bref mais très beau. »

Nous proposons ici au lecteur de se remémorer ou de découvrir ce problème plaisant et délectable qui fit la gloire de Prolog III.

1.1. Origines du puzzle mathématique

Quadriller de façon régulière un rectangle de dimensions entières par des carrés tous identiques s'effectue simplement en suivant une grille. *A contrario*, décomposer un rectangle en carrés inégaux sans perdre de surface est un casse-tête qui a de très rares solutions. Son apparence ludique cache un problème difficile, vieux de plus d'un siècle. Brillant élève de David Hilbert, le géomètre Max Dehn s'y intéressa peu après l'obtention de son doctorat. Il établit en 1903 que si le problème a une solution, alors les dimensions de toutes les figures qui la composent sont nécessairement dans un rapport rationnel (leurs côtés sont commensurables) [4]. Un rectangle solution sera toujours similaire à un rectangle de dimensions entières.

Il faut attendre 1925 et la publication du mathématicien Zbigniew Moroń pour obtenir les premiers exemples de rectangles divisés en carrés inégaux [8]. Le plus petit d'entre-eux est presque carré et comprend 9 carrés. On en trouve une très belle illustration en couverture d'un ancien numéro du magazine de vulgarisation *Scientific American* (figure 1.1) [7]. La rubrique « Jeux mathématiques » de la revue venait d'être prise en main par le regretté Martin Gardner. Elle offrait une tribune à l'un des pères de la théorie des graphes, William Tutte, qui expliquait comment, étudiant à Trinity College, il s'était attaqué à ce défi mathématique avec trois de ses camarades [10].



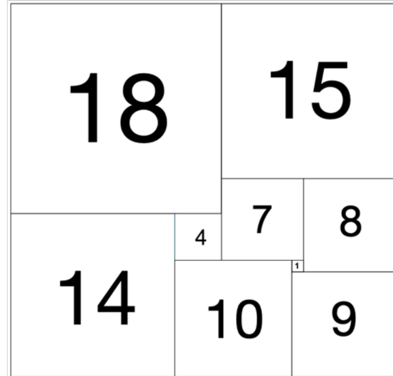


FIGURE 1.1 – Couverture de *Scientific American* montrant une solution à 9 carrés

Enfin, le rectangle peut être lui-même un carré ! Au moins 21 carrés sont nécessaires pour que la construction d'un « carré parfait » soit possible. Cette plus petite quadrature fut découverte en 1978 en utilisant une recherche par ordinateur [5].

Curieusement, ce qui est possible dans le plan devient impossible dans l'espace puisqu'on ne peut décomposer un cube en un nombre fini de cubes de tailles distinctes...

1.2. Marseille, fin des années 80

C'est probablement le résultat de Dehn qui aiguillonna Alain Colmerauer. Depuis la sortie des premiers prototypes fonctionnels du langage, il était à l'affût d'exemples bien choisis pour promouvoir Prolog III et marquer les esprits. Le domaine du linéaire qui bénéficiait de procédures de décision complètes et raisonnablement efficaces ouvrait un champ d'applications fascinant.

« Colmé » avait coutume de challenger ses étudiants du Groupe d'Intelligence Artificielle de Luminy inscrits en cycle de recherche en leur demandant de trouver le programme Prolog le plus apte à résoudre l'un de ces exercices. La plupart du temps, compte tenu de la difficulté des problèmes, il surclassait ses infortunés étudiants en abattant quelques jours plus tard la solution optimale et définitive !

Le problème du découpage d'un rectangle en carrés de tailles distinctes n'y fit pas exception. Séduit par les contraintes sur les rationnels, l'un des deux auteurs qui était en DEA à Luminy en 1987 se souvient d'avoir essayé de se mesurer en Prolog III à ce problème totalement inaccessible aux autres langages de programmation. Il conçut

une solution limitée (bien en deçà de celle attendue) qui exigeait de fixer préalablement les deux dimensions du rectangle – en plus du nombre de carrés, avant de procéder à leurs empilements. Cependant, lorsque le couple de dimensions donnait un rectangle qui ne pouvait être décomposé, le programme bouclait indéfiniment...
Bref, la tentative de l'élève était loin de pouvoir rivaliser avec la solution du maître !

2. Le programme Prolog III

La question est de savoir s'il existe n carrés de dimensions distinctes pouvant s'assembler sans se chevaucher ni laisser d'espace entre eux de façon à former un rectangle. Et dans l'affirmative, de connaître leurs tailles et celle du rectangle produit. Cette recherche peut s'effectuer en accroissant progressivement la valeur de l'entier n jusqu'à produire une solution (et énumérer au besoin les suivantes).

Le programme Prolog III va ainsi chercher à répondre à une requête plus ciblée : pour n donné, existe-t-il une telle dissection d'un rectangle en n carrés ?

2.1. Le remplissage d'un rectangle par n carrés

On considère pour cela un rectangle quelconque que l'on représentera couché sur son plus grand côté A. S'il existe une solution au pavage de ce rectangle, alors il en ira de même pour tout agrandissement ou réduction de celui-ci. Sans perdre en généralité, on peut donc fixer sa hauteur à 1 et se ramener à la question du remplissage d'un rectangle de dimensions $A \times 1$ par des carrés tous distincts (A étant un nombre rationnel à déterminer).

Le nombre de carrés n étant fixé, nous avons $n+1$ inconnues : la dimension A (rapport largeur/hauteur), plus les n dimensions des carrés à consommer. Nous savons que dans toute solution ces inconnues auront des valeurs rationnelles. Prolog III (comme Prolog IV) saura donc les représenter. Il s'agit maintenant d'exhiber par programme un tel « rectangle parfait ».

2.2. Appel général

L'appel du programme se fait par la requête `remplir_rectangle(A,C)` où `C` est une liste de longueur égale au nombre n de carrés attendus. La variable `A` correspond à la largeur du rectangle dont la hauteur normalisée est 1. Dans le cas où une solution existe, la liste `C` donne les côtés des n carrés recherchés.

Observons les résultats obtenus en Prolog III avec les premières requêtes au programme :

```
1  > remplir_rectangle(A, [B]) ;
2  {A = 1, B = 1}
3  >
```

Listing 1 – Remplissage avec 1 carré

Si l'on teste le programme avec une liste réduite à un seul élément (`C = [B]`), on obtient pour seule réponse `B = A = 1`. La solution est un rectangle de dimension 1×1. Pour pouvoir paver un rectangle avec un seul carré, il faut et il suffit que ce rectangle soit un carré ! Les mathématiciens de Trinity College qui se sont penchés sur ce problème au siècle dernier ont exclu cette solution triviale. Ils ont défini l'« ordre » d'un rectangle parfait comme le plus petit cardinal d'une décomposition en carrés tous distincts qui ne soit pas réduite à un singleton. Mais ici le programme d'Alain Colmerauer n'impose pas au rectangle d'être distinct des carrés.

Il est évident que pour former un rectangle avec 2 carrés, il faut que ces carrés soient de même taille, ce qui est incompatible avec l'énoncé. Prolog III répond qu'il n'y a pas de solution à la requête :

```
1  > remplir_rectangle(A, [B1, B2]) ;
2  >
```

Listing 2 – Remplissage avec 2 carrés

Il n'existe donc pas de rectangle parfait d'ordre 2. Pas plus qu'il n'en existe d'ordre 3. De fait, la requête échoue pour toute tentative utilisant de 2 à 8 carrés. Si l'on admet par avance que notre programme est correct et complet, ceci démontre formellement qu'on ne peut trouver de rectangle parfait d'ordre inférieur à 9 !

Si l'on pose maintenant la question avec 9 carrés (la contrainte sur la taille de la liste apparaît entre accolades), on obtient 2 rectangles solutions avec leurs symétriques verticaux et horizontaux, soit en tout 8 configurations :

```
1  > remplir_rectangle(A, C) , {C = [_,_,_,_,_,_,_,_,_]} ;
2  {A = 33/32,
3   C = [9/32,5/16,7/16,1/4,1/32,7/32,1/8,9/16,15/32]}
4  {A = 69/61,
5   C = [28/61,16/61,25/61,7/61,9/61,5/61,2/61,36/61,33/61]}
6  {A = 69/61,
7   C = [25/61,16/61,28/61,9/61,7/61,2/61,5/61,36/61,33/61]}
8  {A = 33/32,
9   C = [7/16,5/16,9/32,1/32,1/4,1/8,7/32,9/16,15/32]}
10 {A = 69/61,
11  C = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]}
12 {A = 33/32,
13  C = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32]}
14 {A = 69/61,
15  C = [36/61,33/61,5/61,28/61,25/61,9/61,2/61,7/61,16/61]}
16 {A = 33/32,
17  C = [9/16,15/32,7/32,1/4,7/16,1/8,5/16,1/32,9/32]}
18 >
```

Listing 3 – Remplissage avec 9 carrés

Dimensions du rectangle	Dimensions des carrés
33×32	$1, 4, 7, 8, 9, 10, 14, 15, 18$
69×61	$2, 5, 7, 9, 16, 25, 28, 33, 36$

FIGURE 2.1 – Les deux solutions avec 9 carrés, aux symétries près

La première solution correspond au rectangle découvert il y a cent ans par Moroń (sans que l'on sache vraiment comment).

Pour finir, en accolant un carré de côté A à un rectangle parfait $A \times 1$ d'ordre n, on crée automatiquement un rectangle parfait d'ordre $n+1$. Si bien qu'à partir de l'ordre 9 toutes les requêtes ont en théorie une solution. En pratique, si l'on vise une recherche exhaustive, on observe très rapidement une explosion combinatoire.

2.3. Deux remarques

Avant d'en venir à la solution algorithmique proposée, nous pouvons relier le problème étudié à deux sous-problèmes dont l'abord paraîtra plus aisé au lecteur.

2.3.1. *Problème de dimensionnement*

Lorsque la disposition des carrés dans la solution est connue (comme dans la figure 1.1 dans laquelle on aura préalablement effacé les tailles), il est assez facile de retrouver les dimensions qui leur correspondent. C'est un joli exercice d'algèbre qui revient à résoudre un système de Cramer. Pour ce faire, il suffit de tracer tous les traits prolongeant les côtés des carrés et d'égaler les sommes des dimensions rencontrées à A dans le sens horizontal et à 1 dans le sens vertical.

Horizontalement		Verticalement	
$x_{18} + x_{15} = A$	(2.1)	$x_{18} + x_{14} = 1$	(2.6)
$x_{18} + x_7 + x_8 = A$	(2.2)	$x_{18} + x_4 + x_{10} = 1$	(2.7)
$x_{14} + x_4 + x_7 + x_8 = A$	(2.3)	$x_{15} + x_7 + x_{10} = 1$	(2.8)
$x_{14} + x_{10} + x_1 + x_8 = A$	(2.4)	$x_{15} + x_7 + x_1 + x_9 = 1$	(2.9)
$x_{14} + x_{10} + x_9 = A$	(2.5)	$x_{15} + x_8 + x_9 = 1$	(2.10)

TABLE 2.1 – Système d'équations à résoudre

De l'exemple de Moroń, on tire ainsi un système linéaire à 10 équations et 10 inconnues dont la solution unique[1], exprimée en 32-ièmes d'unité, est :

$$A = 33, x_{18} = 18, x_{15} = 15, x_{14} = 14, x_{10} = 10, x_9 = 9, x_8 = 8, x_7 = 7, x_4 = 4, x_1 = 1.$$

[1]Sur le plan informatique, en règle générale, le calcul de solutions fractionnaires réclame une précision infinie afin que les pivots de l'élimination de Gauss ne soient pas entachés d'erreur.

Chaque groupe d'égalités peut être vu sous l'angle de la conservation d'un flot dans un réseau. La figure 2.2 donne une représentation graphique du premier (de valeur A).

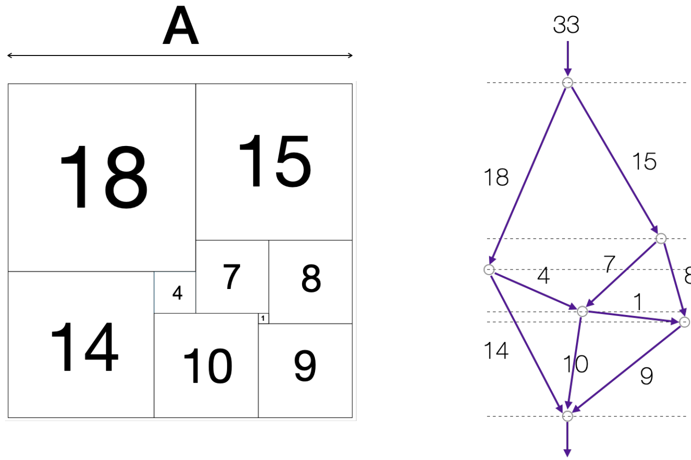


FIGURE 2.2 – Analogie avec les équations de conservation d'un flot

2.3.2. *Problème d'ordonnancement*

Inversement, lorsque les tailles des carrés et celle du rectangle sont connues, leur placement géométrique répond à un double problème d'ordonnancement de projet sous contraintes de ressources (RCPSP pour *Resource-Constrained Project Scheduling Problem* en anglais).

Le côté d'un carré indique à la fois la durée d'une tâche et sa consommation en ressources. Si l'on identifie l'axe du temps à l'abscisse, la largeur A correspond à la date de fin du projet et la hauteur 1 donne le niveau de ressources disponibles. Il n'y a pas de préemption possible. Les tâches (carrés) peuvent s'exécuter en parallèle, mais à aucun moment leur cumul ne peut dépasser la limite de capacité définie (rectangle). Si l'on identifie l'axe du temps à l'ordonnée, ces 2 valeurs sont permutées.

La modélisation se résume globalement à l'énoncé de 2 contraintes cumulatives. Mais résoudre un RCPSP est en soi un problème combinatoire difficile ! Dans les deux cas, il n'y a aucun temps mort (toutes les tâches sont sur un chemin critique) et les profils cumulatifs sont plats (toutes les inégalités sont saturées).

3. Le procédé constructif de remplissage

Le programme de remplissage du rectangle en Prolog III va à la fois agencer les carrés et les dimensionner. Alain Colmerauer propose ici une méthode exhaustive pour explorer toutes les façons de construire une solution.

Si l'on veut carreler la surface du rectangle, les côtés des carrés doivent être joints parallèlement aux axes, sans se chevaucher ni laisser d'interstice. Mais comment faire avec des carreaux de tailles irrégulières ? On ne peut former ni rangée ni colonne ! Toute l'astuce du procédé constructif est de parvenir à retrouver dans ce remplissage une certaine régularité.

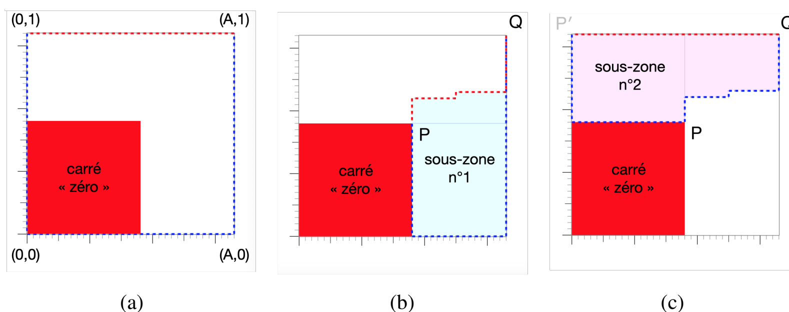
3.1. Principe des poupées russes

L'algorithme de placement va généraliser le problème à d'autres surfaces que le rectangle, pour le subdiviser ensuite en sous-problèmes de remplissage de « zones » issues de sa décomposition. Ces zones s'agenceront entre elles comme les pièces d'un puzzle pour finir par couvrir l'intégralité de la surface du rectangle. Elles auront une géométrie particulière pour pouvoir s'assembler. On autorisera à cet effet des déformations de la figure du rectangle, dans lesquelles ses côtés inférieur et supérieur pourront devenir des escaliers(2).

Par convention, le placement progressera de gauche à droite et de bas en haut, initialement à partir des bords du rectangle, puis des bords des carrés déjà placés. Toute zone non vide se verra décomposée en trois parties adjacentes :

- un carré « zéro », placé à l'origine de la zone(3)
- une première sous-zone, située à sa droite
- une seconde sous-zone, située au dessus.

À son tour, chaque nouvelle zone non vide sera à remplir de carrés. Si bien que dans le résultat final, zone et sous-zones s'imbriqueront comme des poupées russes.



(a) (b) (c)

Figure 3.1 – Tripartition du rectangle initial (a) en sous-zones (b) et (c) après placement du premier carré

(2)Les côtés gauche et droit restant verticaux.

(3)Ce carré pouvant à lui seul consommer l'intégralité de la surface de la zone.

3.2. Chemins remontants

En partant du coin supérieur droit d'un carré P dans un pavage solution, nous pouvons observer qu'il est toujours possible de rejoindre le coin supérieur droit du rectangle Q par une ligne brisée continue qui longe les côtés de carrés adjacents, sans jamais redescendre ni repartir à gauche[(4)]. Les tentatives de découpage du rectangle suivront de tels tracés irréguliers que nous appellerons *chemins remontants*. Ils apparaissent en pointillés sur la figure 3.1.

Nous définirons un chemin remontant comme une succession de marches horizontales (formant des plateaux de profondeur non nulle) et de contremarches verticales (de hauteur éventuellement nulle) qui remontent en escalier [(5)] vers le coin supérieur droit du rectangle pour s'y arrimer dans le plan. Pour cette raison, nous appellerons le point Q de coordonnées $(A, 1)$ *point d'ancrage*.

Le fait que la destination d'un chemin remontant reste constante va nous permettre, à rebours, de le représenter très simplement par une suite de distances horizontales et verticales. Nous en distinguerons 2 types suivant leur nombre de segments :

- les chemins remontants *pairs* : $H_1, V_1, \ldots, H_i, V_i, \ldots, H_k, V_k$ qui commencent par un plateau (segment horizontal H_1), où
  - les distances horizontales sont toujours strictement positives ($H_i > 0$)
  - les distances verticales peuvent être positives ou nulles ($V_i \geqslant 0$)
- les chemins remontants *impairs* : $V_0, H_1, V_1, \ldots, H_i, V_i, \ldots, H_k, V_k$ qui commencent par une contremarche (segment vertical V_0) [(6)], avec les mêmes conventions.

Par exemple, le tracé (0,0) $(A,0)$ $(A,1)$ qui part de l'origine du système de coordonnées, longe le côté inférieur du rectangle, puis remonte en équerre le long de son côté droit, forme un chemin remontant pair de 2 segments : une marche de profondeur A précédant une contremarche de hauteur 1. Il sera codé par le couple $[A, 1]$. Quant au tracé $(A, 0)$ $(A, 1)$ qui remonte le côté droit du rectangle, il forme un chemin remontant impair. Son unique segment vertical donnera le singleton [1].

Notre définition impose qu'un chemin remontant se termine par un segment vertical, mais ce dernier peut être un artéfact de construction de hauteur nulle. De même, un segment vertical de hauteur nulle intercalé entre deux segments horizontaux décrit une « fausse marche » formée deux plateaux jointifs successifs de hauteur identique.

[(4)]Cela est évident si le carré est placé sur le côté droit ou sur le côté supérieur du rectangle. Si tel n'est pas le cas, le carré est nécessairement placé à gauche ou en dessous d'un autre carré. Il suffit alors d'en suivre un des contours pour remonter jusqu'a son coin supérieur droit et prolonger ainsi un chemin remontant conduisant au point $(A, 1)$.

[(5)]Les nez des marches successives ont des abscisses et des ordonnées croissantes.

[(6)]Certains chemins remontants impairs pouvant se réduire à cette contremarche.

3.3. Géométrie des zones

Les zones avec lesquelles fonctionne l'algorithme de remplissage peuvent être vues comme des conteneurs à ciel ouvert. Ces cavités vont toujours présenter :

- un point minimal, en bas à gauche, servant d'origine au placement
- un point maximal, en haut à droite, correspondant au point d'ancrage.

Indiqué en pointillés bleus sur la figure 3.1, le profil de la zone fait apparaître une fosse, suivie ou non à sa droite d'une ligne de gradins. Partant d'un point P du rectangle, ce profil comprend plus précisément :

(1) un dénivelé vertical strictement négatif, $D < 0$
(2) un plateau horizontal strictement positif, $H > 0$
(3) une butée verticale positive ou nulle, $V \geqslant 0$, constituant le premier élément d'un chemin remontant impair $[V \mid L]$.

Le triplet $[D, H, V]$ délimite la fosse proprement dite dont la surface à remplir vaut au minimum $D \times H$. La ligne brisée $[H, V \mid L]$ qui part de l'origine de la zone pour rejoindre le point d'ancrage [7] forme un chemin remontant pair de P à Q que nous appellerons *ligne de soubassement*.

Les zones à remplir auront ainsi un plancher étagé en terrasses. Une fois remplies, elles auront un plafond étagé de la même manière, ce qui permettra aux zones filles issues de la division tripartite d'une zone mère de s'emboîter parfaitement [8]. Le plafond d'une zone suivra donc un chemin remontant impair (visualisé par des pointillés rouges sur la figure 3.1) que nous appellerons *ligne de couverture*.

Après le dénivelé et la ligne de soubassement, la ligne de couverture achève de cerner complètement une zone.

De façon imagée, la définition d'une ligne de couverture est donnée par :

$$\uparrow[\overrightarrow{\ },\ \uparrow]^*$$

alors que le profil d'une zone à remplir suit l'expression régulière :

$$\downarrow[\rightarrow,\ \uparrow]^+$$

où * et $^+$ sont les opérateurs d'itération (de 0 à n et de 1 à n respectivement).

4. L'algorithme de remplissage

Nous conserverons dans cet exposé l'ancienne syntaxe de Prolog III pour l'élégance et la concision de sa formulation. Le lecteur trouvera en annexe la traduction complète du programme en Prolog IV. Trois choses sont à retenir :

(7) Pour les lecteurs qui ne seraient pas familiers de la syntaxe standard des listes en Prolog, lire : la liste dont les 2 éléments de tête sont, dans l'ordre, H et V, et dont la queue est la liste L.

(8) Par construction en effet, le niveau de remplissage d'une zone ne sera jamais inférieur à l'ordonnée de son point de départ P. Dans les faits, il pourra la dépasser en croissant par paliers sans sortir du rectangle.

- La flèche de réécriture `->` qui correspond au connecteur logique d'implication ($\Leftarrow$) sépare le prédicat défini (à gauche) de sa définition (à droite).
- Comme en mathématiques, les noms des variables (qui apparaissent ici en lettres majuscules) peuvent se terminer par les caractères prime, seconde ou tierce, comme par exemple `L,L',L'',L'''`.
- Les contraintes sur les variables (qui portent ici sur des nombres rationnels) sont regroupées en fin de règle, entre accolades.

Rappelons plus généralement :

- qu'un programme est constitué de règles dans lesquelles tous les objets manipulés ont une structure d'arbre (éventuellement réduite à un nombre)
- que le déroulement d'un programme vise à déterminer les valeurs des inconnues
- que le langage donne aux opérations mathématiques leur signification habituelle – la seule restriction pour Prolog III étant que les expressions numériques doivent être linéaires.

Comme le rappelle le manuel : « Programmer en Prolog III ne sera parfois rien d'autre qu'écrire les contraintes qui pèsent sur un arbre initialement inconnu, représenté par une variable. Le travail de Prolog III sera alors de résoudre ces contraintes et de dévoiler l'arbre en question. »

4.1. Point d'entrée

Le point d'entrée du programme de remplissage est le prédicat `remplir_rectangle` que nous avons invoqué précédemment. Ses arguments sont :

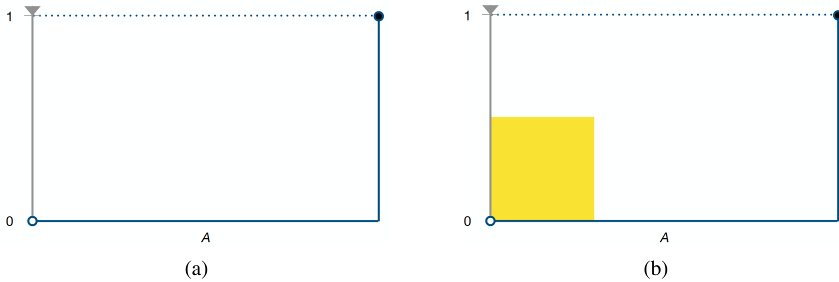
(1) l'inconnue A mesurant le grand côté du rectangle
(2) une liste C de n valeurs inconnues correspondant aux tailles des carrés B_i disponibles (leur nombre n étant fixé d'avance).

```
1  remplir_rectangle(A, C)    ->
2          carres_distincts(C)
3          remplir_zone([-1, A, 1], _, C, []) ,
4          { A ⩾ 1 } ;
```

Listing 4 – La règle `remplir_rectangle/2`

Le programme commence par contraindre A à être plus grand que 1 ($A \geqslant 1$). Puis il crée n carrés de tailles inconnues mais distinctes, ce qui permet d'identifier chaque carré à la longueur B de son côté. Enfin, il lance avec `remplir_zone/4` le remplissage du rectangle, ou plus précisément, de la zone définie par son profil (premier argument). Celui-ci présente un dénivelé de -1, un plancher de longueur A et une contremarche de hauteur 1. Sa ligne de couverture (second argument) ne pourra que suivre le côté

supérieur du rectangle – raison pour laquelle sa composition est ignorée [9]. Enfin, le dernier argument stipule qu'aucun carré de la liste C ne doit rester inutilisé : tous doivent être consommés pour construire la solution.



(a) (b)

Figure 4.1 – Rectangle à remplir (a) et pose du premier carré (b)
Le triangle pointant vers le bas indique le dénivelé; le cercle vide marque l'origine du placement dans la zone; le cercle plein représente le point d'ancrage.

4.2. Conditionnement de la liste des carrés

La définition du prédicat `carres_distincts(C)` suit un schéma classique en programmation logique par contraintes : un double parcours de liste impose que les tailles des n carrés soient distinctes deux à deux. Au total, $n \times (n-1)/2$ contraintes de « non égalité » sont posées. Par ailleurs, ces tailles doivent être strictement positives. Chaque carré placé contribuera donc à couvrir une portion de la surface du rectangle.

```
1  carres_distincts([])       -> ;
2  carres_distincts([B | C]) ->
3        distincts_de(C, B)
4        carres_distincts(C) ,
5        { B > 0 } ;
6
7  distincts_de([], _)        -> ;
8  distincts_de([B' | C], B) ->
9        distincts_de(C, B)
10       { B ≠ B' } ;
```

Listing 5 – Les règles `carres_distincts/1` et `distincts_de/2`

La relation subsidiaire `distincts_de(C,B)` est naturellement vraie si la valeur B ne fait pas partie des valeurs de la liste C. On s'interdit ainsi de produire une solution dans laquelle deux carrés B et B' auraient des côtés de même longueur [10].

[9]Le blanc souligné indique que l'on ne se préoccupe pas de la valeur résultante.
[10]Pour la lisibilité du code, nous avons substitué le caractère ≠ au caractère ASCII # .

4.3. La récursion principale

Le cœur du programme est la procédure `remplir_zone(L,L',C,C')` composée d'une règle récursive et d'une règle d'arrêt.

```
1 remplir_zone([D | L], L''', [B | C], C'') ->
2       placer_carre(B, L, L')
3       remplir_zone(L', L'', C, C')
4       remplir_zone([D + B, B | L''], L''', C', C'') ,
5       { D < 0 } ;
```

Listing 6 – La règle récursive `remplir_zone/4`

En entrée, la description d'une zone est partiellement définie par son profil de conteneur (premier argument), lequel démarre par un dénivelé D négatif ($D < 0$) et se poursuit par une ligne de soubassement L. En sortie, cette description est complétée par la définition de sa ligne de couverture (second argument). On dispose également en entrée de la liste des carrés disponibles pour paver la zone (troisième argument). Après remplissage, il ne subsiste en sortie que la liste des carrés qui n'ont pas été utilisés (quatrième argument).

Pour remplir une zone de profil L, on commence par placer à son origine un premier carré, de dimension B inconnue au départ. La surface résiduelle peut être scindée en deux par un chemin remontant impair qui part du coin supérieur droit de ce carré. Cette frontière définit deux nouvelles zones, à remplir successivement :

(1) la première à droite du carré et en-dessous de cette frontière
(2) la seconde au-dessus du carré et à gauche de cette frontière.

Ce schéma aboutit à une procédure de placement doublement récursive.

Le placement du carré B définit le profil L' de la première sous-zone dont la ligne de couverture trace la frontière L''. En accolant les valeurs du dénivelé $D + B$ et du plateau B en tête de cette frontière on obtient le profil de la seconde sous-zone dont la ligne de couverture L''' établit celle de la zone toute entière.

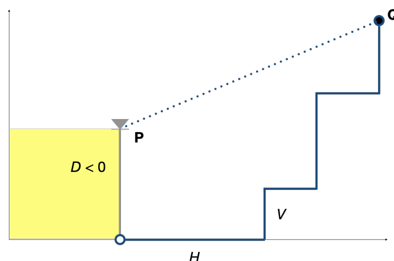


Figure 4.2 – Délimitation de la première zone (non vide) à remplir

Comme le fait remarquer Alain Colmerauer dans un commentaire concernant la ligne de couverture de la zone initiale : *cette ligne doit joindre deux points qui se*

Zone	Tracé délimitant le conteneur	Dénivelé	Ligne de soubassement
rectangle initial	(0, 1) (0, 0) (A, 0) (A, 1)	`-1`	`[A,1]`
à droite de B	(B, B), (B, 0) (A, 0) (A, 1)	`-B`	`[A-B,1]`
au dessus de B	(0, 1), (0, B) (B, B) ... (A, 1)	`-1+B`	`[B | L'']`

TABLE 4.1 – Description du rectangle et des zones issues de sa première partition

trouvent à la même hauteur. Compte tenu de ce qu'elle ne peut redescendre, ce sera forcément une ligne horizontale représentée par un escalier dont toutes les marches sont de hauteur nulle.

4.4. L'étude de cas pour le placement d'un carré

Venons-en maintenant au placement du carré qui constitue la partie non déterministe du programme. Par hypothèse, la fosse $[D, H, V]$ présente une surface non nulle à couvrir puisque D et H sont tous deux positifs. S'il n'y a plus de carré disponible, c'est peine perdue : la procédure `remplir_zone` échoue. On supposera donc qu'il reste au moins un carré B en stock.

Trois cas peuvent se présenter :

(1) le côté du carré est inférieur à la largeur du plateau sur lequel il doit reposer : $B < H$,
(2) le côté du carré est égal à la largeur du plateau qui le supporte : $B = H$,
(3) le côté du carré est supérieur à la largeur du plateau sur lequel il doit reposer : $B > H$.

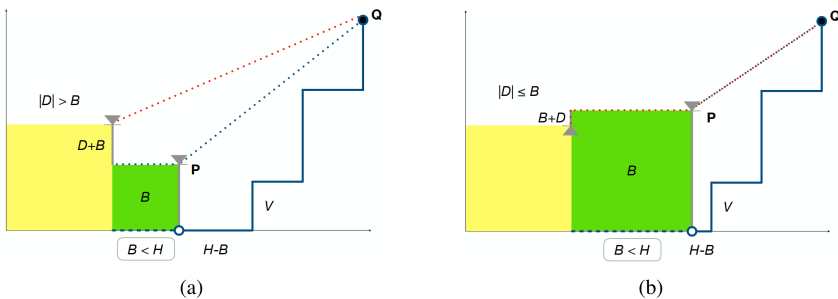


FIGURE 4.3 – Cas $B < H$:
sous-cas $|D| > B$ (à gauche) et $|D| \leqslant B$ (à droite)

Tous ces cas sont mutuellement exclusifs – ce qui garantit une recherche sans redondance ni omission.

Dans le cas numéro 1 où B est strictement inférieur à H, le placement du carré à l'origine de la zone laisse obligatoirement un vide à sa droite. La sous-zone de droite

présente dès lors une fosse à combler. Pour cela, il faut à nouveau faire appel à la règle `remplir_zone/4`.

Deux sous-cas sont à considérer pour la sous-zone du dessus, selon le rapport de B à la valeur absolue du dénivelé D de la zone en cours de remplissage.

- Si $B < |D|$ (figure 4.3(a)), alors la sous-zone située au dessus de B présente une fosse à combler, ce qui justifie un nouvel appel à `remplir_zone/4`.
- Si par contre $B \geqslant |D|$ (figure 4.3(b)), alors la sous-zone située au dessus de B ne présente pas de fosse à combler : son profil bas prend la forme d'une ligne de couverture. La zone peut être refermée en suivant cette même ligne (symbolisée par des pointillés rouges et bleus). Cette zone ayant une surface nulle, la récursion s'arrête.

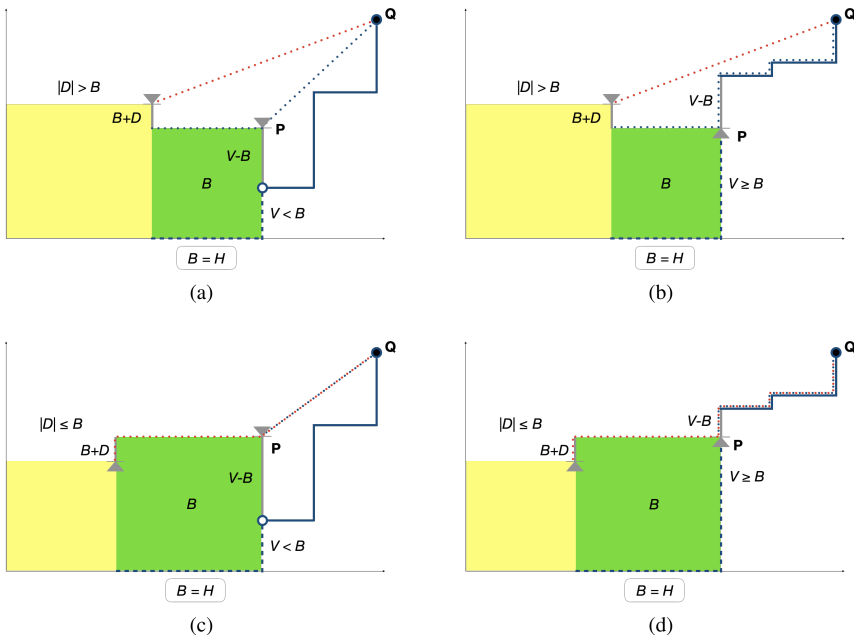


FIGURE 4.4 – Cas $B = H$ et ses 4 sous-cas

Dans le cas numéro 2 où B est égal à H, quatre sous-cas sont à considérer, suivant les valeurs relatives de B, V et D :

- Si $B < |D|$ et $B > V$ (figure 4.4(a)) : il y a présence de fosses à droite et au dessus de B. D'où une double récursion pour remplir les sous-zones de droite et du dessus.
- Si $B < |D|$ et $B \leqslant V$ (figure 4.4(b)) : il y a absence de fosse à droite mais présence d'une fosse au dessus de B. D'où arrêt de la récursion pour la sous-zone de droite (vide), mais poursuite de la récursion pour la sous-zone du dessus.

- Si $B \geqslant |D|$ et $B > V$ (figure 4.4(c)) : il y a présence d'une fosse à droite de B et absence au dessus. D'où poursuite de la récursion pour remplir la sous-zone de droite, mais arrêt de la récursion pour celle du dessus (vide).
- Si $B \geqslant |D|$ et $B \leqslant V$ (figure 4.4(d)) : il y a absence de fosse à droite et absence au dessus. D'où arrêt de la récursion pour les sous-zones de droite et du dessus (vides).

Enfin, pour que B dépasse de la largeur du plateau dans le cas numéro (3), il faut qu'il n'y ait pas de butée à sa droite. La seule hypothèse envisageable est que l'on se trouve en présence d'une fausse marche dont la hauteur est nulle. Si $V = 0$, on ne change rien au profil de la zone en remplaçant cette fausse marche par un plateau unique, en soudant entre eux les plateaux à niveau. Il suffit alors de relancer l'étude de cas avec cette configuration simplifiée.

Dans tous les autres cas, la procédure échoue.

4.5. Arrêt de la récursion

La récursion s'applique aussi longtemps qu'il reste une fosse à remplir. Elle s'interrompt lorsqu'on tombe sur une « zone vide » ayant déjà le profil d'une ligne de gradins.

À côté des vraies zones à remplir qui présentent une cavité non vide, il paraît judicieux d'élargir la définition de nos zones aux zones à laisser vides. On définira donc un « profil généralisé » de zone $[D \mid L]$ où le dénivelé D pourra avoir un signe quelconque.

Suivant le signe de ce dernier, on retrouvera :

- un profil standard de zone, commençant par un dénivelé négatif $D < 0$ (symbolisé par un triangle pointant vers le bas)
- un profil fictif de zone vide, commençant par un dénivelé positif ou nul $D \geqslant 0$ (symbolisé par un triangle pointant vers le haut).

Aussitôt ouverte, une zone vide sera reconnue comme factice. Elle tombera sous le coup de la règle d'arrêt de `remplir_zone`/4 qui la refermera à l'identique – sans consommer de carré – en identifiant son profil à sa ligne de couverture.

```
1 remplir_zone([D | L], [D | L], C, C)      -> ,
2       { D ⩾ 0 } ;
3 remplir_zone([D | L], L''', [B | C], C'') ->
4       placer_carre(B, L, L')
5       remplir_zone(L', L'', C, C')
6       remplir_zone([D + B, B | L''], L''', C', C'') ,
7       { D < 0 } ;
```

Listing 7 – Le programme `remplir_zone`/4 avec sa clause d'arrêt

Avec cette convention, la double récursion reste systématiquement de mise.

4.6. Traduction en règles de l'étude de cas

N'ayant plus à se soucier des conditions d'arrêt, l'étude de cas se simplifie jusqu'à l'épure. La procédure `placer_carre/3` résume par 3 règles les 3 cas précédents. Son premier argument est le carré B à placer dans la zone. Son second argument est la ligne de soubassement de la zone (dont la longueur est connue en entrée). Construit à partir des deux premiers arguments, son troisième argument donne en sortie le profil généralisé de la sous-zone de droite à remplir en premier.

```
1 placer_carre(B, [H | L], [-B, H - B | L]) -> ,
2       { B < H } ;
3 placer_carre(B, [H, V | L], [V - B | L])  -> ,
4       { B = H } ;
5 placer_carre(B, [H, 0, H' | L], L')       ->
6       placer_carre(C, [H + H' | L], L') ,
7       { B > H } ;
```

Listing 8 – Les trois règles de l'étude de cas

Les manipulations de listes traduisent les opérations à effectuer pour raccorder les lignes qui délimitent géométriquement ces zones. La table 4.2 résume ces calculs lorsque $D < 0$ est le dénivelé de la zone à remplir et $[H, V \mid L]$ sa ligne de soubassement.

La gestion de l'étude de cas nécessite pour son découpage au scalpel un solveur capable de faire la distinction entre inégalités strictes et inégalités larges, ce qui représente pour l'époque une innovation notable dans le champ des mathématiques appliquées. Avec ses contraintes linéaires d'inégalité, le modèle théorique et logique de Prolog III ouvre la voie. Il étend au calcul numérique les traitements de l'égalité et du différent que Prolog II avait rigoureusement établis.

Cas	Dénivelé généralisé	Ligne de soubassement	Ligne de couverture
$B < H$	`-B` < 0	`[H-B,V	L]`	`L''`
$B = H$	`V-B` $\geqslant 0$	`L`	`L''`

(a)

Cas	Dénivelé généralisé	Ligne de soubassement	Ligne de couverture
$B \leqslant H$	`D+B` $\geqslant 0$	`[B	L'']`	`L'''`

(b)

Table 4.2 – Délimitations des sous-zones de droite (a) et du dessus (b) (cf. clauses terminales de `placer_carre/3` et clause récursive de `remplir_zone/4`)

5. Le programme au complet

Tout compte fait, le programme tient en 10 règles.

```
1  % Point d'entrée :
2
3  remplir_rectangle(A, C)   ->
4        carres_distincts(C)
5        remplir_zone([-1, A, 1], _, C, []) ,
6        { A ⩾ 1 } ;
7
8  % Conditionnement de la liste des carrés :
9
10 carres_distincts([])      -> ;
11 carres_distincts([B | C]) ->
12       distincts_de(C, B)
13       carres_distincts(C) ,
14       { B > 0 } ;
15
16 distincts_de([], _)       -> ;
17 distincts_de([B' | C], B) ->
18       distincts_de(C, B)
19       { B ≠ B' } ;
20
21 % Algorithme de remplissage :
22
23 remplir_zone([D | L], [D | L], C, C)      -> ,
24       { D ⩾ 0 } ;
25 remplir_zone([D | L], L''', [B | C], C'') ->
26       placer_carre(B, L, L')
27       remplir_zone(L', L'', C, C')
28       remplir_zone([D + B, B | L''], L''', C', C'') ,
29       { D < 0 } ;
30
31 % Etude de cas pour le placement d'un carré :
32
33 placer_carre(B, [H | L], [-B, H - B | L]) -> ,
34       { B < H } ;
35 placer_carre(B, [H, V | L], [V - B | L])  -> ,
36       { B = H } ;
37 placer_carre(B, [H, 0, H' | L], L')       ->
38       placer_carre(C, [H + H' | L], L') ,
39       { B > H } ;
```

Listing 9 – Les dix règles Prolog III

Le placement dispose en tout de 3 budgets : le nombre de carrés (connu) et les dimensions horizontale et verticale du rectangle (de rapport A inconnu). Celles-ci contraignent les positions des coins inférieur gauche et supérieur droit des carrés placés qui ne peuvent sortir du rectangle initial.
La terminaison de l'algorithme est assurée par le fait que le remplissage d'une zone non vide consomme au moins un carré.

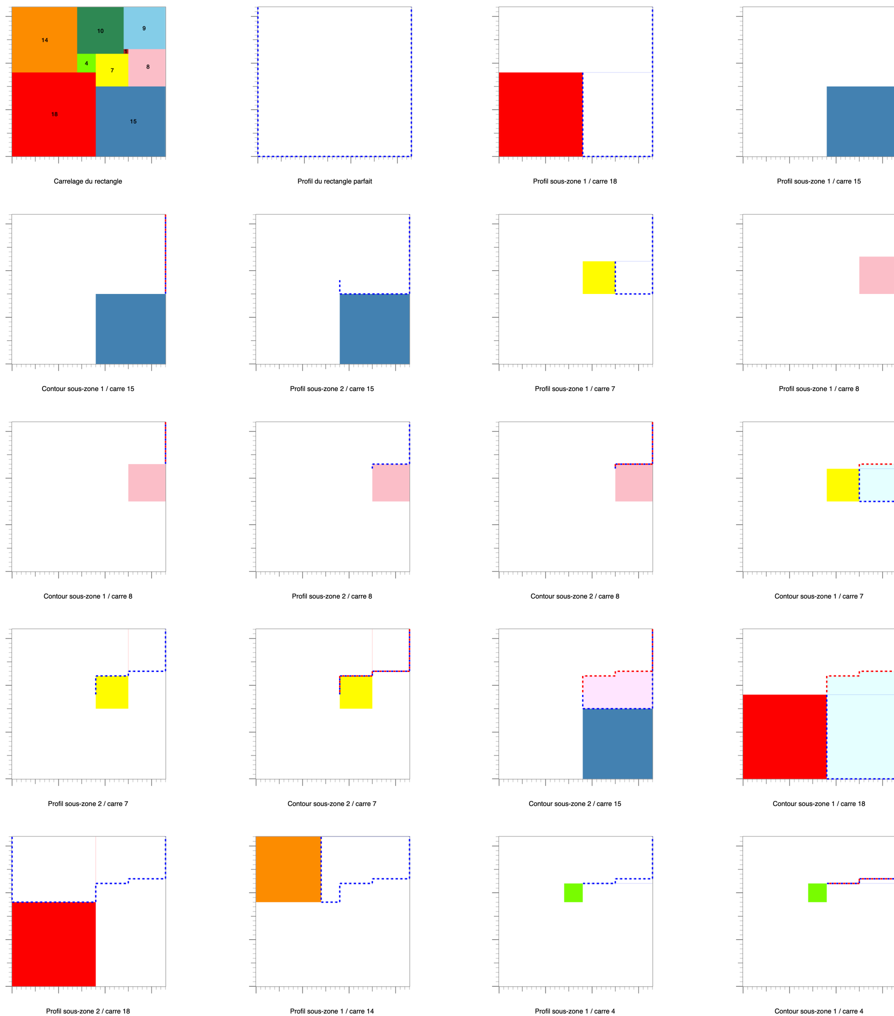


FIGURE 5.1 – Simulation d'exécution : planche 1/2

6. Trace d'une exécution

À partir de la réponse produite par le programme, retraçons, pour mieux en saisir la géométrie, la construction en poupées russes des zones qui la constituent.

Le déroulement présenté ici est celui qui aboutit à la solution à 9 carrés :

$$\{A = 33/32, C = [18/32, 15/32, 7/32, 8/32, 14/32, 4/32, 10/32, 1/32, 9/32]\}.$$

Figure 6.1 – Simulation d'exécution : planche 2/2

Le placement commence ici par les carrés les plus gros. Il donne l'image inversée du rectangle parfait de Moroń.

La lecture se fait de gauche à droite et de haut en bas. Chaque carré est repéré par sa couleur (dans l'ordre : rouge, bleu gris, jaune, rose, orange, vert pâle, vert sombre, brique, bleu ciel). La couleur de remplissage est le bleu pâle pour les premières sous-zones et le rose pâle pour les secondes.

Attention : l'exécution présentée ici est trompeuse ! Comme on indique au programme les valeurs des inconnues A et C, on ne voit plus tous les essais-erreurs qui conduisent à cette solution. Sans cela, la visualisation serait impossible.

En réalité, tant que les valeurs des dénivelés, des plateaux horizontaux et des seuils verticaux ne sont pas déterminées par le calcul, le contour d'une zone reste une définition paramétrique à ajuster. C'est la résolution du système linéaire engrangé qui fixe en dernier lieu ces quantités.

L'arborescence suivante décrit l'agencement des carrés dans cette solution :

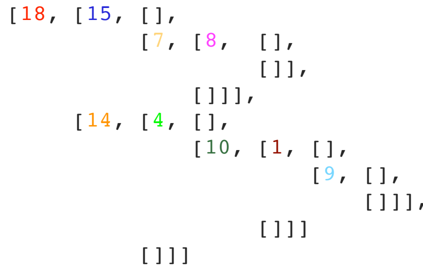
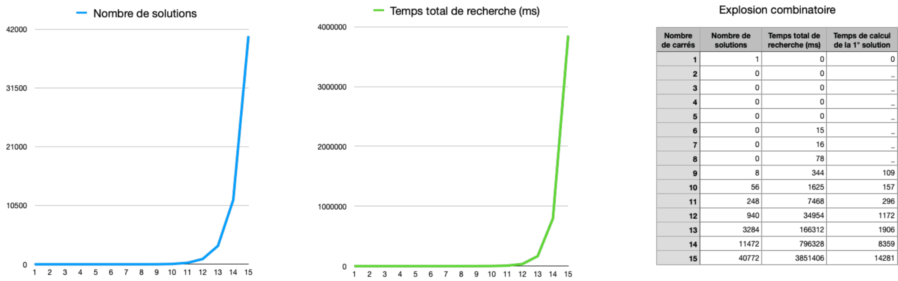


Figure 6.2 – Imbrication des zones

Sa lecture ligne par ligne donne le code de Bouwkamp $(18, 15)(7, 8)(14, 4)(10, 1)(9)$ du nom du professeur de mathématiques, conseiller des laboratoires de recherche de la société Philips, qui consacra sa vie à cataloguer les solutions de ce problème [1]. Dans cette représentation simplifiée, les parenthèses entourent les carrés contigus dont les côtés sont alignés horizontalement. Un balayage vertical, par plateaux successifs, produit leur séquencement.

7. Conclusion

De ce programme régulier et concis, Alain Colmerauer pouvait à juste titre être fier. Son esthétique de l'économie de moyens témoigne du haut degré d'exigence qui a toujours été le sien, comme ingénieur et comme scientifique, cet *ostinato rigore* pour reprendre la devise de Léonard de Vinci. Mais ce chef d'œuvre de minimalisme réclame un petit effort pour être compris !



Nombre de carrés	Nombre de solutions	Temps total de recherche (ms)	Temps de calcul de la 1° solution
1	1	0	0
2	0	0	_
3	0	0	_
4	0	0	_
5	0	0	_
6	0	15	_
7	0	16	_
8	0	78	_
9	8	344	109
10	56	1625	157
11	248	7468	296
12	940	34954	1172
13	3284	166312	1906
14	11472	796328	8359
15	40772	3851406	14281

FIGURE 7.1 – Temps de calcul en fonction de l'ordre du rectangle parfait

Nous espérons que nos explications – si longues pour un programme si bref – auront permis au lecteur de mieux percevoir la remarquable ingéniosité du procédé mis en œuvre par son auteur pour résoudre à l'ordre 9 le problème du rectangle parfait.

En résumé, le programme se borne à décrire mathématiquement les différentes possibilités. Il laisse ensuite la machine Prolog dérouler l'étude de cas. Après tout, elle est conçue pour cela. Et la magie opère !

Tel quel cependant, le programme ne passe pas à l'échelle. L'ordre 21 est hors de sa portée pour espérer produire un carré parfait. Il faudra attendre la thèse de l'un de ses derniers étudiants, Ian Gambini, pour l'atteindre et le dépasser [6].

BIBLIOGRAPHIE

[1] S. E. ANDERSON, « Tiling by Squares – squared square finders », `www.squaring.net`.

[2] A. COLMERAUER, « Une introduction à Prolog III », *Annales des Télécommunications* **44** (1989), nº 5-6, p. 229–241, `alain.colmerauer.free.fr/alcol/ArchivesPublications/Prolog3/acmprolog3f.pdf`.

[3] ———, « An Introduction to Prolog III », *Communications of the ACM* **33** (1990), nº 7, p. 69-90.

[4] M. DEHN, « Über Zerlegung von Rechtecken in Rechtecke. », *Math. Ann.* **57** (1903), p. 314-332.

[5] A. J. W. DUIJVESTIJN, « Simple perfect squared square of lowest order », *Journal of Combinatorial Theory, Series B* **25** (1978), nº 2, p. 240-243.

[6] I. GAMBINI, « Quant aux carrés carrelés », thèse de doctorat, Université de la Méditerranée, Marseille, 2001, `alain.colmerauer.free.fr/alcol/ArchivesPublications/Gambini/carres.pdf`.

[7] M. GARDNER, « Mathematical Games », *Scientific American* **199** (1958), nº 5, p. 136-144.

[8] Z. MOROŃ, « O Rozkladach Prostokatow Na Kwadraty (On the Dissection of a Rectangle into Squares) », *Prezeglad Mat.* **3** (1925), p. 152-153.

[9] PROLOG HERITAGE, `www.prolog-heritage.org`, Manuels de référence des « Prolog de Marseille ».

[10] W. T. TUTTE, « The Quest of the Perfect Square », *The American Mathematical Monthly* **72** (1965), nº 2, p. 29-35.

8. Annexe : Transcription du programme en Prolog IV

```
1  % Point d'entrée :
2
3  remplir_rectangle(A, Carres) :-
4          gelin(A, 1),            % A >= 1
5          carres_distincts(Carres),
6          remplir_zone([-1, A, 1], _, Carres, []).
7
8  % Conditionnement de la liste des carrés :
9
10 carres_distincts([]).
11 carres_distincts([B | Carres]) :-
12         gtlin(B, 0),            % B > 0
13         distincts_de(Carres, B),
14         carres_distincts(Carres).
15
16 distincts_de([], _).
17 distincts_de([B1 | Carres], B) :-
18         dif(B, B1),
19         distincts_de(Carres, B).
20
21 % Algorithme de remplissage :
22
23 remplir_zone([D | L], [D | L], Carres, Carres) :-
24         gelin(D, 0).            % D >= 0
25 remplir_zone([D | L], CouvZone, [B | Cs1], Cs3) :-
26         ltlin(D, 0),            % D < 0
27         placer_carre(B, L, ProfilZone1),
28         remplir_zone(ProfilZone1, CouvZone1, Cs1, Cs2),
29         remplir_zone([D+B, B | CouvZone1], CouvZone, Cs2, Cs3).
30
31 % Etude de cas pour le placement d'un carré :
32
33 placer_carre(B, [H | L], [-B, H-B | L])  :-
34         ltlin(B, H).            % B < H
35 placer_carre(B, [H, V | L], [V-B | L])  :-
36         B = H.
37 placer_carre(B, [H1, 0, H2 | L], Profil) :-
38         gtlin(B, H1),           % B > H
39         placer_carre(B, [H1+H2 | L], Profil).
```

Listing 10 – Les dix règles en Prolog IV

Pour améliorer la lisibilité du code, le mode syntaxique `prolog4` est utilisé. Dans la règle `placer_carre` qui traite du cas d'égalité $B = H$ par exemple, cela permet d'interpréter $V - B$ non pas symboliquement comme un arbre, mais numériquement comme le résultat de l'opération de soustraction.

Avec le mode ISO strict de Prolog IV, l'écriture est un peu plus lourde. Il faut utiliser explicitement la formulation relationnelle et introduire à la place de l'expression fonctionnelle une variable intermédiaire, par exemple $VmoinsB$, définie par la contrainte linéaire `minuslin(VmoinsB,V,B)`.

En Prolog IV enfin, la contrainte de taille `size(9,C)` unifie C à une liste de 9 éléments. Avec elle, la requête s'exprime très simplement.

```
1  >> size(9, C), remplir_rectangle(A, C).
2
3  A = 33/32,
4  C = [9/32,5/16,7/16,1/4,1/32,7/32,1/8,9/16,15/32];
5
6  A = 69/61,
7  C = [28/61,16/61,25/61,7/61,9/61,5/61,2/61,36/61,33/61];
8
9  A = 69/61,
10 C = [25/61,16/61,28/61,9/61,7/61,2/61,5/61,36/61,33/61];
11
12 A = 33/32,
13 C = [7/16,5/16,9/32,1/32,1/4,1/8,7/32,9/16,15/32];
14
15 A = 69/61,
16 C = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61];
17
18 A = 33/32,
19 C = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32];
20
21 A = 69/61,
22 C = [36/61,33/61,5/61,28/61,25/61,9/61,2/61,7/61,16/61];
23
24 A = 33/32,
25 C = [9/16,15/32,7/32,1/4,7/16,1/8,5/16,1/32,9/32].
26
27 >>
```

Listing 11 – Les 8 solutions comprenant 9 carrés en Prolog IV

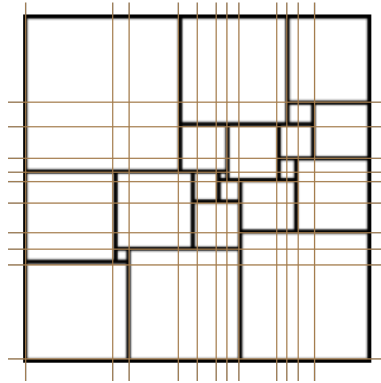


FIGURE 8.1 – Carré parfait d'ordre 21

ABSTRACT. — The purpose of this article is to recall, fifty years later, the interest of constraint logic programming according to a historical and pedagogical use case: filling a rectangle of unknown size with a set of squares of unknown but distinct sizes. The obligation to differentiate the sizes precludes the use of any symmetry hypothesis that would make this problem easier to solve.

It was proved by a Polish mathematician, Zbigniew Moroń in 1925, that the smallest set of distinct squares perfectly tiling a rectangle is of cardinal 9. We will comment on the behavior of the Prolog III program designed by Alain Colmerauer which uses the domain of rational numbers to offer an automated version of this proof. It exploits a property demonstrated by Max Dehn, a student of David Hilbert, in 1903: there can only be a solution if the size's ratios are commensurable.

The program embarks on a systematic but highly combinatorial case study in search of placements. Along the way, it accumulates a double set of additive constraints (according to the height and width of the rectangle), resulting in the global resolution of a system of linear equations and inequalities. But as long as this system remains underdetermined, no partially instantiated solution helps us to glimpse the outcome of the calculation.

By making explicit the data types that derive from the geometric dissection method, this article performs a kind of reverse engineering of the rules applied. The following formal and informal discussion shows both the power and the declarative conciseness of constraint logic programming. It also illustrates the difficulty of tracing the behind-the-scenes inferences of the smart operational machinery that leads to the result.

KEYWORDS. — Constraint Logic Programming, Mathematical Puzzle, Prolog, Linear constraints on rational numbers.

Manuscrit reçu le 27 mai 2024, accepté le 12 juillet 2024.