Baseball des couleurs - Une petite étude théorique

Christophe BAL : projetmbc@gmail.com

Version du 2017-01-21

Mentions « légales »

Table des matières

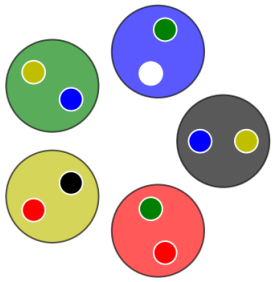
---

1 Origine du jeu

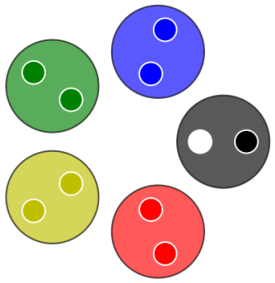
L'auteur de ces modestes lignes a pris connaissance du jeu « *le baseball des couleurs* » le jeudi 5 janvier 2017 lors d'une formation d'« *Informatique Débranchée* » proposée par l'Académie de Grenoble. Ce jeu est inspiré du « *jeu de l'orange* » proposé dans le livre « *L'informatique sans ordinateur - Programme d'activités d'éveil pour les élèves à partir de l'école primaire* » de Tim Bell, Ian H. Witten et Mike Fellows qui a été adapté à l'utilisation en classe par Robyn Adams et Jane McKenzie (Septembre 2009 - 2e éd. juin 2014) [1].

2 Les règles

On considère cinq îlots colorés différemment que nous nommerons « *bases* ». Chacune de ces bases contient deux trous pour y accueillir deux jetons. Pour chaque couleur $\mathcal{C}$ sauf une, disons $\mathcal{C}''$, on a deux jetons de couleur $\mathcal{C}$. Par contre, on a juste un jeton de couleur $\mathcal{C}''$. La partie commence en mettant au hasard deux jetons sur chaque base sauf une qui ne contiendra qu'un seul jeton. Voici un exemple où c'est le jeton noir qui est seul, et c'est sur la base bleue qu'il n'y a qu'un seul jeton.



Le seul mouvement autorisé est le déplacement d'un jeton vers le trou libre à condition que le dit jeton et le trou soient dans des bases « *directement* » voisines. Dans l'exemple ci-dessus, on ne peut donc bouger que l'un des jetons des bases bleue et rouge. Le but du jeu est d'obtenir la configuration suivante où chaque jeton a retrouvé sa base (la position du trou dans la base noire est sans importance).



Si vous ne connaissez pas ce jeu nous vous conseillons, avant de lire la suite de ce document, d'essayer de trouver une méthode (il en existe plusieurs possibles et peut-être que vous trouverez d'autres points de vue que ceux que nous proposerons plus bas). Pour cela, il suffit d'expérimenter en vous fabriquant des bases et des jetons carrés, ce qui est très vite fait.

Remarque : dans ce jeu nous avons à tout moment accès à toutes les informations mais on pourrait très bien imaginer n'avoir accès qu'aux informations présentes sur trois bases voisines (concrètement, on mettrait les bases sur un plateau tournant avec au-dessus un cache ne laissant apparaître que trois bases). Il faut savoir qu'en informatique l'on peut avoir ce type de contrainte : par exemple, les routeurs qui dirigent les informations sur le réseau physique de l'internet n'ont pas une connaissance globale du réseau et pourtant ils arrivent à trouver d'assez bonnes solutions pour acheminer des données.

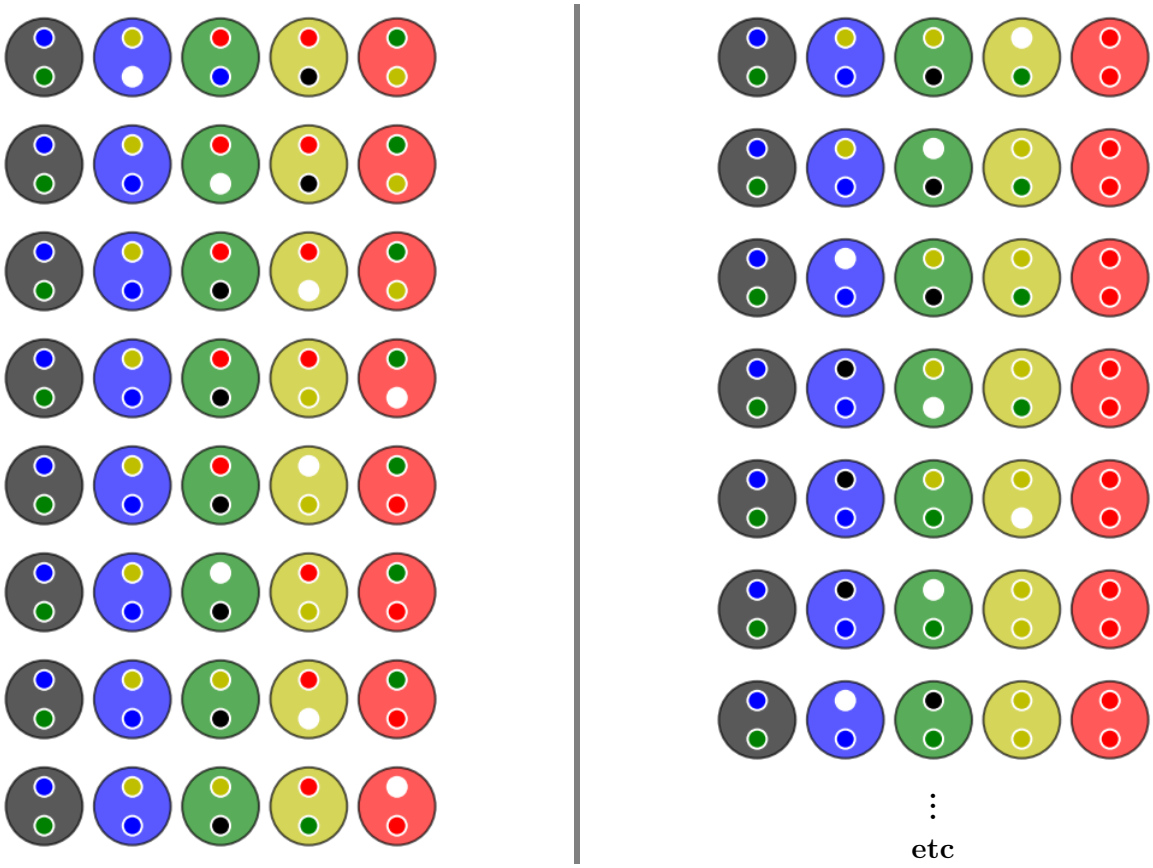
1. Voir `http://csunplugged.org/books/`.

3 Premier contact : la méthode *« une base à la fois »*

Avant de chercher une solution optimale, commençons déjà par en trouver au moins une qui fonctionne[2]. Laissant de côté le problème de la solution la plus efficace, nous avons assez vite les deux idées simples suivantes.

1. Casser le cercle pour représenter les bases en ligne comme ci-dessous où il est important de bien mettre à gauche la base noire, celle n'ayant qu'un seul jeton de même couleur une fois le problème résolu.
2. S'interdire tout mouvement de la base rouge, la plus à droite, à la base noire, la plus à gauche.

L'ajout d'une contrainte va nous permettre de donner un algorithme simple à décrire mais aussi facile à valider. Par contre, nous devinons bien que nous nous interdisons de résoudre rapidement le problème. La méthode que nous allons présenter va consister à remplir la cinquième base, puis la quatrième... en cherchant juste à rapatrier les jetons manquants. Voici des premiers mouvements possibles où vous noterez au passage qu'une fois une base remplie à droite, celle-ci n'est plus jamais utilisée (les étapes évoluent dans la colonne de gauche puis dans celle de droite).



Pour définir précisément comment fonctionne notre méthode, nous le faisons via l'algorithme suivant (l'écriture formelle employée est simple à comprendre).

2. Les problèmes d'optimisation d'algorithme se traitent toujours dans un second temps. Cette règle s'applique aussi constamment lorsque l'on programme.

Donnée : une configuration en ligne quelconque de début de jeu
Résultat : une configuration en ligne où tous les jetons sont rentrées dans leur base

Début

Aucune base n'est isolée pour le moment (nous allons vite voir ce que cela signifie).

Tant Que *la configuration contient un jeton qui n'est pas dans sa base* **:**

$\mathcal{D}$: la base non remplie la plus à droite.
$Coul_{\mathcal{D}}$: la couleur de la base $\mathcal{D}$.

`// Les deux jetons de couleur` $Coul_{\mathcal{D}}$ `peuvent être dans la même base.`
j : un jeton de couleur $Coul_{\mathcal{D}}$ le plus à droite possible en dehors de la base $\mathcal{D}$.
$\mathcal{J}$: la base du jeton j.

`// Deux contraintes.`
Ctr_1 : ne pas passer par d'éventuelles bases isolées.
Ctr_2 : ne pas bouger l'autre jeton de couleur $Coul_{\mathcal{D}}$ excepté si les deux jetons sont dans la même base.

En respectant les deux contraintes Ctr_1 et Ctr_2,
- si besoin, amener le trou dans la base à droite de la base $\mathcal{J}$,
- puis utiliser le trou pour amener j dans la base $\mathcal{D}$.

Si *la base $\mathcal{D}$ est complète* **:**

Isoler la base $\mathcal{D}$ (en la décalant un peu plus à droite par exemple).

Nous devons vérifier la validité de cet algorithme c'est à dire vérifier trois choses.

1. **Non Ambiguïté :** les actions proposées doivent être sans ambiguïté.
2. **Finitude :** les actions à faire seront toujours en nombre fini.
3. **Résolution :** une fois toutes les actions effectuées, nous devons obtenir une configuration où tous les jetons sont rentrées dans leur base.

Le contrat de « *non ambiguïté* » est rempli même si une certaine liberté est laissée pour déplacer le trou ou le jeton [3] à condition de ne pas visiter d'éventuelles bases isolées et sans bouger un éventuel jeton déjà bien placé dans la base la plus à droite non encore complète. En pratique, la « *non ambiguïté* » n'est jamais justifiée, par contre les deux derniers points doivent toujours faire l'objet d'une démonstration comme nous allons le faire tout de suite.

Démonstration. La preuve va s'appuyer sur deux résultats très simples dont on va donner des énoncés un peu formels mais avec des preuves visuelles très simples.

Fait n°1 : on peut toujours déplacer le trou de la base où il est vers une base voisine $\mathcal{V}$ en laissant fixe un jeton de son choix dans la base $\mathcal{V}$.

En effet, considérons le cas suivant où l'on veut déplacer le trou vers la troisième base en ne touchant pas au jeton rouge (les choix faits ne nuisent pas à la généralité du raisonnement pour un déplacement vers la droite).

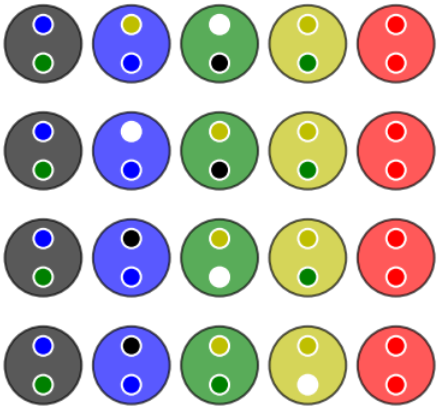
Il suffit de procéder comme suit (c'est évident).

3. Nous avons ici un bel exemple d'algorithme « *non déterministe* » en ce sens qu'en lançant l'algorithme plusieurs fois sur la même configuration initiale, on ne passera pas forcément par les mêmes étapes intermédiaires pour résoudre le jeu.

Le cas où la base voisine est à gauche se traite de façon analogue (on peut aussi utiliser un argument de type « *symétrie* »).

Fait n°2 : avec les notations de l'algorithme, on suppose le trou être dans la base $\mathcal{J}_d$ à droite de la base $\mathcal{J}$ du jeton j à déplacer, et que cette base $\mathcal{J}_d$ n'est pas la base $\mathcal{D}$ à remplir. Dans ce cas, on peut placer le jeton j dans la base $\mathcal{J}_d$ et le trou dans la base à droite de $\mathcal{J}_d$ en choisissant la place du trou.

La preuve est bien plus simple à comprendre que l'affreux énoncé ci-dessus (on se demande bien qui a pu rédiger un truc pareil). Ci-dessous, le jeton à déplacer est le jaune dans la deuxième base bleue, et nous choisissons de ne pas toucher au jeton jaune de la quatrième base jaune (le lecteur notera la généralité de la méthode proposée).



Finitude et résolution : commençons pas démontrer que l'algorithme commence par remplir la base rouge la plus à droite.

Si la base rouge est déjà remplie, aucune action n'est requise et le résultat est vrai. Supposons donc que nous ayons au moins un jeton rouge dans l'une des quatre premières bases. Nous pouvons alors suivre les instructions de l'algorithme comme suit.

- Grâce au fait n°1, il est effectivement possible de placer le trou dans la base à droite de celle du jeton rouge à déplacer, et ceci sans faire bouger un éventuel jeton rouge déjà dans la base rouge.
- Ensuite, le fait n°2 nous permet d'amener le jeton rouge à déplacer dans l'avant-dernière base jaune et le trou dans la base rouge, de nouveau sans faire bouger un éventuel jeton rouge déjà dans la base rouge.
- Il ne reste plus qu'à déplacer notre jeton rouge à la place du trou.

Si avant de faire ces opérations la base rouge contenait déjà un jeton rouge, nous avons rempli cette base, sinon l'algorithme nous fera remplir cette base dans un second temps (lors de la prochaine boucle `Tant Que`).

Une fois la base rouge remplie, celle-ci est isolée. Ceci implique que l'algorithme va travailler sur une version à quatre bases du jeu. Pour conclure, il suffit de reprendre le raisonnement ci-dessus non plus avec la base rouge mais avec la base jaune. Puis ensuite l'algorithme travaillera avec trois puis enfin deux bases et l'on raisonnera à chaque fois de la même façon. Ceci achève de montrer les propriétés de finitude et de résolution. □

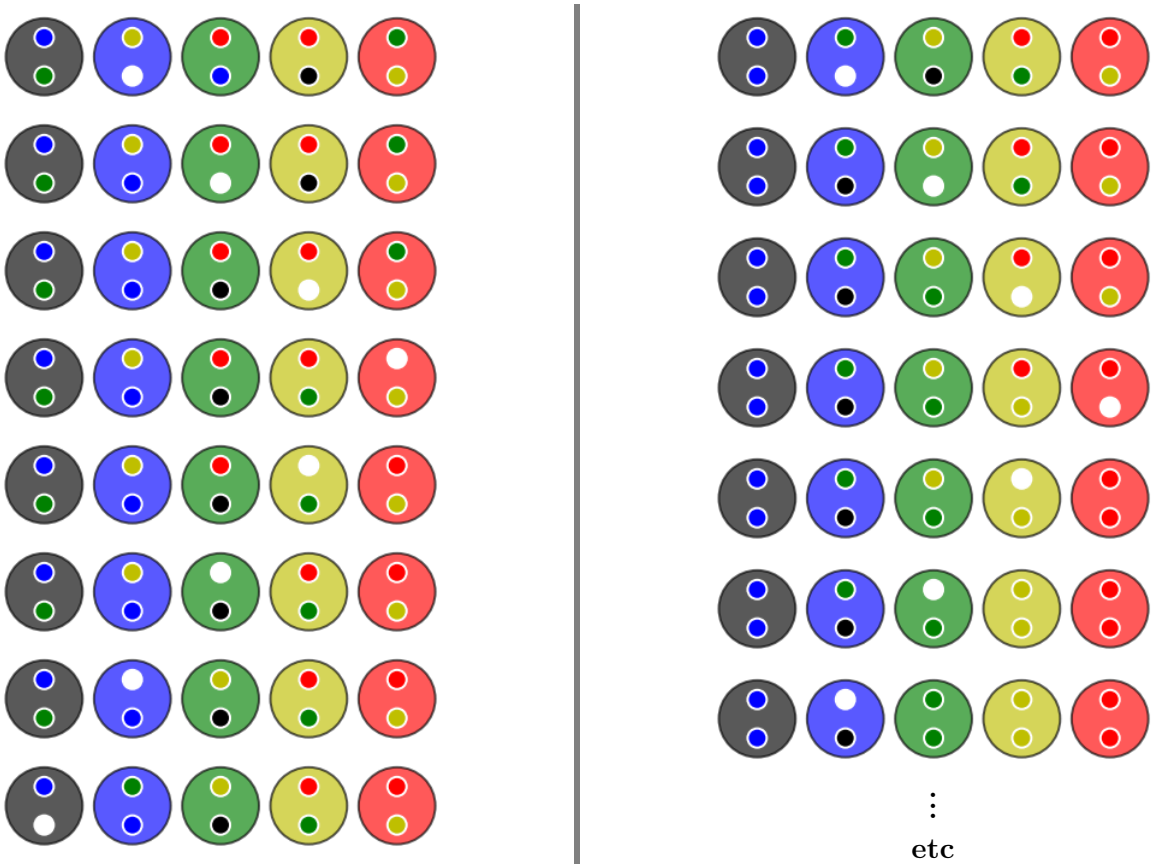
Remarque : vous noterez que la démonstration précédente est valable pour un nombre quelconque $n \geqslant 2$ de bases (les plus tatillons pourront faire un raisonnement par récurrence).

4 Soyons opportuniste : la méthode *« on avance au mieux »*

La très grosse maladresse de la méthode « *une base à la fois* » est de ne pas chercher à profiter de chaque déplacement du trou pour arranger la situation. Nous allons juste considérer cette idée pour proposer une nouvelle méthode de résolution [4] qui elle aussi commence par mettre les bases en ligne de la base noire vers la rouge, et interdit tout mouvement de la base rouge à la base noire.

1. On ordonne les couleurs, pour cela nous dirons que lorsque l'on va de la gauche vers la droite, on passe de couleurs froides à des couleurs chaudes.
2. On commence par déplacer le trou vers la base rouge la plus chaude. Lors de ce déplacement, il faut déplacer à chaque fois le jeton le plus froid vers la gauche. Si les deux jetons de la base où va aller le trou sont de même chaleur, on choisit au hasard l'un des deux jetons.
3. Une fois arrivé à la base rouge, on bouge le trou vers la base noire la plus froide. Lors de ce déplacement, ce sont les jetons les plus chauds qui migrent vers la droite. Si l'on tombe sur deux jetons de même chaleur, on applique la même tactique que ci-dessus.
4. On continue les deux opérations 2 et 3 tant que toutes les bases ne sont pas complètes.

Voici le début de l'application de cette méthode.



Intuitivement, on sent bien que l'on ne va pas entrer dans une boucle infinie mais encore faut-il passer de l'intuition à une preuve irréfutable. Pour cela donnons d'abord une version plus formelle de notre nouvel algorithme de résolution (pour ordonner les couleurs nous allons les numéroter afin de simplifier les explications tout en raisonnant de façon très générale).

4. Cette méthode a été proposée par les personnes en charge de la formation d'« *Informatique Débranchée* » proposée par l'Académie de Grenoble le jeudi 5 janvier 2017. Voir la section « *Origine du jeu* ».

Donnée : une configuration en ligne quelconque de début de jeu
Résultat : une configuration en ligne où tous les jetons sont rentrées dans leur base

Début

On numérote de 0 à 4 les bases de la gauche vers la droite.
On associe chaque couleur au numéro de la base de la dite couleur.

```
// d donne la direction que doit suivre le trou.
//   • d = 1 pour un déplacement vers la droite.
//   • d = -1 pour un déplacement vers la gauche.
```
$d \leftarrow 1$

Tant Que *la configuration contient un jeton qui n'est pas dans sa base* **:**

t : numéro de la base où est le trou.

Si $(d\,,t) = (1\,,4)$ **:**

$d \leftarrow (-1)$

Sinon Si $(d\,,t) = (-1\,,0)$ **:**

$d \leftarrow 1$

Sinon:

```
// v est le numéro de la base voisine de celle du trou où l'on doit mettre ce dernier.
```
$v \leftarrow t + d$

Si *d = 1* **:**

min : minimum des numéros des couleurs des jetons dans la base n°v.
Déplacer un jeton de couleur *min* de la base n°v vers la base n°t.

Sinon:

max : maximum des numéros des couleurs des jetons dans la base n°v.
Déplacer un jeton de couleur *max* de la base n°v vers la base n°t.

Les instructions sont clairement non ambigües. Prouvons les propriétés de « *finitude* » et de « *résolution* ».

Démonstration. À chaque configuration $\mathscr{C}$, on associe une liste $L(\mathscr{C})$ de nombres comme suit (des exemples visuels sont donnés un peu plus bas).

1. À chaque jeton on associe le numéro de sa couleur, et l'on considère le trou comme un jeton associé à l'entier (-1).
2. Commençons alors par considérer les jetons de la base n°0.
   (a) Si les deux jetons sont de la même valeur x, on définit la liste $L(\mathscr{C}) = [x\,,x]$ notée en utilisant des crochets. Dans une liste, l'ordre d'écriture est important et il peut y avoir des répétitions.
   (b) Si les deux jetons sont de valeurs différentes x et y, on définit la liste $L(\mathscr{C}) = [x\,,y]$ si $x < y$, et $L(\mathscr{C}) = [y\,,x]$ sinon.

   En résumé, $L(\mathscr{C}) = [\alpha\,,\beta]$ avec α et β où sont les valeurs des jetons sur la base n°0.
3. Ensuite on passe aux jetons de la base n°1.
   (a) Si les deux jetons sont de la même valeur r, on « *augmente* » la liste $L(\mathscr{C})$ en lui adjoignant à droite la liste $[r\,,r]$. Ainsi si $L(\mathscr{C}) = [x\,,y]$ à la fin de l'étape n°2, on obtient ici $L(\mathscr{C}) = [x\,,y\,,r\,,r]$.
   (b) Si les deux jetons sont de valeurs différentes r et s, à la liste $L(\mathscr{C})$ on adjoint à droite soit la liste $[r\,,s]$ si $r < s$, soit $[s\,,r]$ sinon. Ainsi si $L(\mathscr{C}) = [x\,,y]$ à la fin de l'étape n°2, on obtient ici $L(\mathscr{C}) = [x\,,y\,,r\,,s]$ ou $L(\mathscr{C}) = [x\,,y\,,s\,,r]$ suivant les cas.
4. On fait de même avec la base n°2, puis la n°3, … *etc.*

En termes plus courants, dans chaque base on ordonne les jetons de la valeur la plus petite à la valeur la plus grande, puis on « *accole dans une liste* » les couples « *ordonnés* » ainsi formés. Voici des exemples.

$[2\ ,\ 2\ ,\ -1\ ,\ 3\ ,\ 1\ ,\ 3\ ,\ 0\ ,\ 4\ ,\ 1\ ,\ 4]$

$[2\ ,\ 4\ ,\ 0\ ,\ 3\ ,\ 2\ ,\ 3\ ,\ 1\ ,\ 4\ ,\ -1\ ,\ 1]$

Les listes de nombres de même taille sont comparables comme le sont les mots d'un dictionnaire, on parle d'ordre lexicographique. Par exemple, nous avons :

- $[\mathbf{2}\ ,\ \mathbf{2}\ ,\ \mathbf{3}\ ,1\ ,3\ ,0\ ,4\ ,1\ ,4] > [\mathbf{2}\ ,\ \mathbf{2}\ ,\ \mathbf{1}\ ,3\ ,0\ ,4\ ,1\ ,3\ ,4]$ en comparant le 3 et le 1 aux troisièmes positions qui sont les premières valeurs à être différentes (par contre peu importe ce qu'il y a aux positions suivantes).
- $[\mathbf{1}\ ,2\ ,2\ ,3\ ,3\ ,0\ ,4\ ,1\ ,4] < [\mathbf{2}\ ,2\ ,3\ ,1\ ,3\ ,0\ ,4\ ,1\ ,4]$ en comparant le 1 et le 2 aux premières positions (sans se soucier des nombres qui suivent).

Nous n'allons pas comparer les deux listes $L(\mathscr{C}_1)$ et $L(\mathscr{C}_2)$ associées à deux configurations $\mathscr{C}_1$ et $\mathscr{C}_2$. Nous allons considérer à la place les sous-listes $SL(\mathscr{C}_1)$ et $SL(\mathscr{C}_2)$ obtenues en retirant le seul (-1) présent dans les listes $L(\mathscr{C}_1)$ et $L(\mathscr{C}_2)$ respectivement (bien noter que les listes comparées via l'ordre lexicographique sont celles sans (-1) la valeur du trou).

Nous voilà armés pour faire notre démonstration sans encombre (mais pas sans réflexion).

Fait n°1 : si le trou n'est pas dans la base n°0 alors les mouvements indiqués par l'algorithme amène le trou dans la base n°0.

C'est évident à condition d'avoir bien noté que pour la configuration gagnante, le trou est dans la base n°0.

Fait n°2 : la configuration gagnante est la seule telle que le trou soit dans la base n°0 et telle que sa sous-liste soit minimale (c'est à dire qu'elle est inférieure ou égale à toutes les listes $SL(\mathscr{C})$ possibles).

Soit $\mathscr{G}$ la configuration gagnante. Il est clair que $SL(\mathscr{G}) = [0\ ,\ 1\ ,\ 1\ ,\ 2\ ,\ 2\ ,\ \dots\ ]$ et que le trou est dans la base n°0.

Soit $\mathscr{C}$ une configuration non gagnante. Si $SL(\mathscr{C}) \neq SL(\mathscr{G})$, il est évident que $SL(\mathscr{C}) > SL(\mathscr{G})$. Sinon si $SL(\mathscr{C}) = SL(\mathscr{G})$ c'est que le trou n'est pas dans la base n°0.

Fait n°3 : lorsque l'un des mouvements demandés par l'algorithme est effectué, on passe d'une configuration $\mathscr{C}_1$ à une configuration $\mathscr{C}_2$. On a alors la relation : $SL(\mathscr{C}_2) \leqslant SL(\mathscr{C}_1)$.

Considérons le cas où le trou se déplace vers la droite. Notons $L(\mathscr{C}_1) = [\ \dots\ , -1\ ,\ x\ ,\ y\ ,\ z\ ,\ \dots\ ]$ où les points de suspension indiquent des éventuelles paires de valeurs. En particulier, (-1) et x sont les valeurs du trou et du jeton partageant la même base, tandis que y et z sont celles de deux autres jetons dans la base voisine à droite (notons que forcément $y \leqslant z$). Nous avons deux situations.

1. **Cas 1 :** $x \leqslant y$.
   Dans ce cas, $L(\mathscr{C}_2) = [\ \dots\ ,\ x\ ,\ y\ ,\ -1\ ,\ z\ ,\ \dots\ ]$ d'où $SL(\mathscr{C}_2) = SL(\mathscr{C}_1)$.
2. **Cas 2 :** $y < x$.
   Dans ce cas, $L(\mathscr{C}_2) = [\ \dots\ ,\ y\ ,\ x\ ,\ -1\ ,\ z\ ,\ \dots\ ]$ d'où $SL(\mathscr{C}_2) < SL(\mathscr{C}_1)$.

Le cas d'un déplacement vers la gauche se traite de façon analogue.

Fait n°4 : si $\mathscr{C}$ est une configuration telle que $SL(\mathscr{C})$ ne soit pas minimale, l'algorithme fera apparaître à un moment ou à un autre une configuration $\mathscr{C}'$ telle que $SL(\mathscr{C}') < SL(\mathscr{C})$ (en fait, le dit moment arrivera avant un éventuel aller-retour « complet » du trou depuis sa base dans la configuration $\mathscr{C}$).

Comme $SL(\mathscr{C})$ n'est pas minimale, il existe au moins deux valeurs x et y telles que $x > y$ avec y situé après x dans $SL(\mathscr{C})$ lorsqu'on lit cette liste de gauche à droite. La condition $x > y$ implique que x et y ne sont pas dans la même base. Nous avons : $L(\mathscr{C}) = [\ \ldots\ ,\ x\ ,\ \ldots\ ,\ y\ ,\ \ldots\ ]$ où les points de suspension indiquent d'éventuelles valeurs.

Parmi tous les y possibles, on choisit celui qui est le plus à gauche possible dans $SL(\mathscr{C})$, c'est à dire celui qui est le plus prêt de x (dans ce cas, y est la plus petite valeur de sa base). Avec ce choix, les valeurs éventuelles w entre x et y dans $SL(\mathscr{C})$ vérifient toutes $y < x \leqslant w$. Ceci permet donc de choisir x et y voisins dans $SL(\mathscr{C})$ (avec des bases associés voisines différentes). Avec ce nouveau choix, $L(\mathscr{C}) = [\ \ldots\ ,\ g\ ,\ x\ ,\ y\ ,\ d\ ,\ \ldots\ ]$ où les points de suspension indiquent des éventuelles paires de valeurs. Notons que seul g peut être égal à (-1). Désignons par $\mathcal{B}_x$ et $\mathcal{B}_y$ les basses associées aux valeurs x et y.

Rappelons que tant que $SL(\mathscr{C})$ n'est pas minimale, la configuration $\mathscr{C}$ n'est pas une configuration gagnante et donc le trou se balade. Nous avons alors les situations suivantes.

1. **Cas 1 :** $g = -1$
   Si l'algorithme est dans une phase de déplacement vers la droite, alors comme dans la preuve du fait n°3, voir son cas 2, nous savons que la configuration suivante $\mathscr{C}'$ vérifie $SL(\mathscr{C}') < SL(\mathscr{C})$ ce qui prouve ici le fait n°4.
   Sinon le trou va aller vers la base n°0. Si lors de ces déplacements, l'une des configurations $\mathscr{C}'$ vérifie $SL(\mathscr{C}') < SL(\mathscr{C})$ alors le fait n°4 sera validé. Sinon le trou reviendra en direction de la base $\mathcal{B}_x$. Si lors de ces déplacements, l'une des configurations $\mathscr{C}'$ vérifie $SL(\mathscr{C}') < SL(\mathscr{C})$ alors le fait n°4 sera validé. Sinon nous nous retrouvons dans le sous-cas traité ci-dessus où l'on avait un déplacement du trou vers la droite avec $g = -1$, et de nouveau nous obtenons le résultat souhaité.
2. **Cas 2 :** $g \neq -1$ et le trou est à gauche de la base $\mathcal{B}_x$.
   Si l'algorithme est dans une phase de déplacement vers la droite, alors le trou va prendre la place de g et l'on retombe dans le cas 1. Si le déplacement se fait vers la gauche, on peut appliquer exactement le même raisonnement que dans le deuxième sous-cas du cas 1 traité ci-dessus.
3. **Cas 3 :** $g \neq -1$ et le trou est à droite de la base $\mathcal{B}_y$.
   L'algorithme va amener le trou dans la base voisine de $\mathcal{B}_y$ de sorte que l'on ait une configuration $\mathscr{C}'$ telle que $L(\mathscr{C}') = [\ \ldots\ ,\ g\ ,\ x\ ,\ y\ ,\ d\ ,\ -1\ ,\ \alpha\ ,\ \ldots\ ]$ avec $SL(\mathscr{C}')$ non minimal.
   Si lors de ces déplacements une configuration $\mathscr{C}''$ vérifie $SL(\mathscr{C}'') < SL(\mathscr{C})$, le fait n°4 est bien entendu vérifié. Sinon $L(\mathscr{C}')$ devient $[\ \ldots\ ,\ g\ ,\ x\ ,\ -1\ ,\ y\ ,\ ?\ ,\ ?\ ,\ \ldots\ ]$, puis $[\ \ldots\ ,\ -1\ ,\ g\ ,\ y\ ,\ x\ ,\ ?\ ,\ ?\ ,\ \ldots\ ]$ où l'on utilisé le fait que $y < x$. Pour cette dernière liste, la configuration $\mathscr{C}'''$ vérifie $SL(\mathscr{C}''') < SL(\mathscr{C})$ et c'est gagné !

Finitude et résolution : le fait n°1 nous permet de considérer le 1er moment où le trou est dans la base n°0. Si nous avons une configuration $\mathscr{C}$ gagnante, nous nous arrêtons comme demandé et il n'y a rien à prouver.

Sinon la configuration $\mathscr{C}$ est telle que $SL(\mathscr{C})$ ne soit pas minimale d'après le fait n°2. Grâce au fait n°4, nous savons que nous arriverons ensuite à une configuration $\mathscr{C}'$ telle que $SL(\mathscr{C}') < SL(\mathscr{C})$. Dès lors les faits n°3 et n°1 nous permettent d'affirmer que le trou va se retrouver dans base n°0 pour une configuration $\mathscr{C}''$ telle que $SL(\mathscr{C}'') \leqslant SL(\mathscr{C}') < SL(\mathscr{C})$.

En résumé, si le trou se retrouve dans la base n°0 pour une configuration $\mathscr{C}$ non gagnante, alors l'algorithme nous fera déplacer le trou jusqu'à le faire arriver de nouveau dans la base n°0 pour une nouvelle configuration $\mathscr{C}''$ telle que $SL(\mathscr{C}'') < SL(\mathscr{C})$.

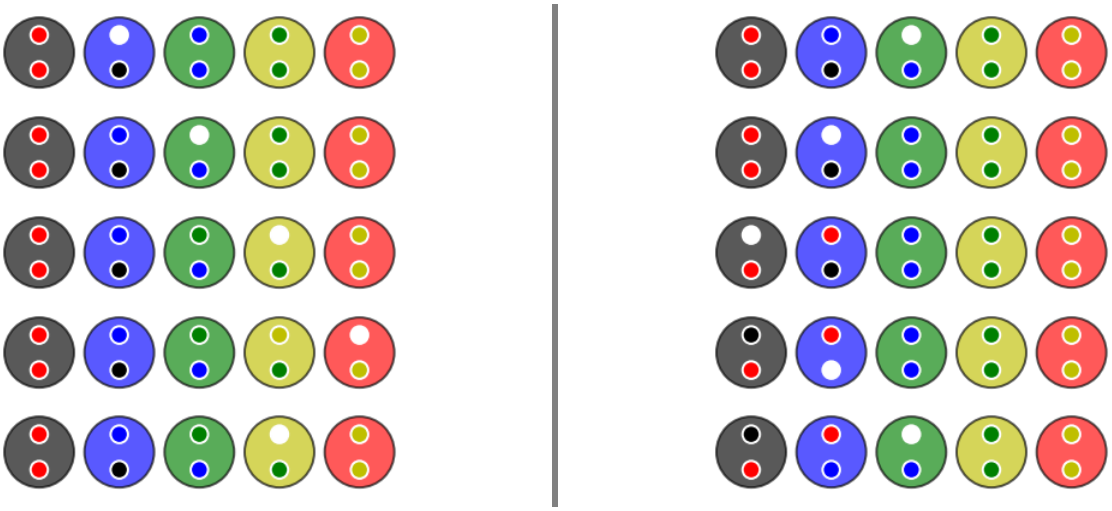
Pour conclure, il suffit de noter que le nombre de listes $SL(\mathscr{C})$ est fini. Dès lors il est impossible d'avoir une suite strictement décroissante de listes du type $SL(\mathscr{C})$. Ceci signifie que l'algorithme va en un nombre fini d'étapes amener le trou dans la base n°0 pour une configuration gagnante. □

Remarque : la démonstration s'adapte sans problème à un nombre quelconque $n \geqslant 2$ de bases.

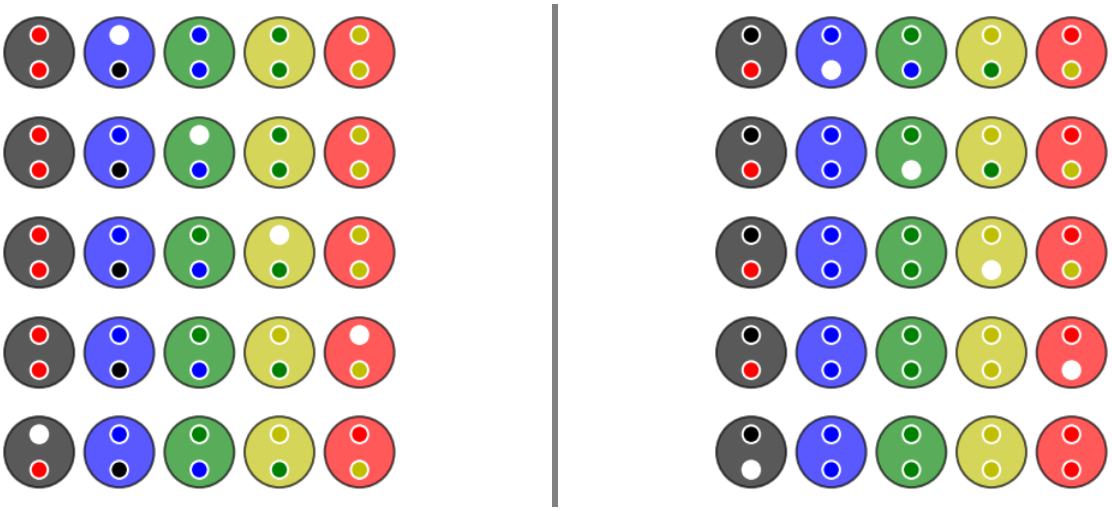
5 À la recherche d'une solution optimale

5.1 Où tentons-nous d'aller ?

Commençons par noter que la méthode « *on avance au mieux* » n'est pas la plus efficace possible comme le montre l'exemple suivant où six mouvements totalement inutiles sont effectués si bien que l'on n'a toujours pas gagné à la neuvième étape [5]. Il est important de noter ici que nous appliquons à la lettre les mouvements demandés par l'algorithme ! Il est sûrement évident pour le lecteur de voir l'inutilité des tous premiers mouvements.



Ici, on peut gagner en seulement 9 coups ! Voici les mouvements à faire [6]. Nous verrons bientôt que l'on ne peut pas faire mieux.



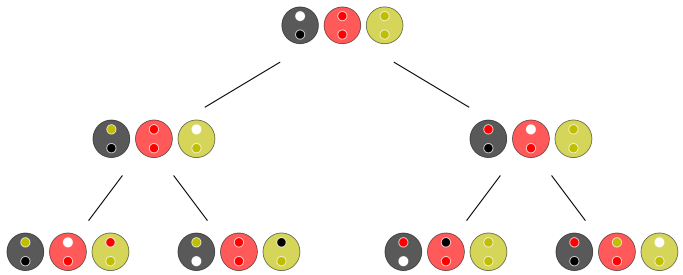
5. En fait, il faut 39 mouvements pour gagner avec la méthode « *on avance au mieux* », et la méthode « *une base à la fois* » en demande 25.

6. Notez au passage les enseignements que l'on peut tirer d'une configuration très, très particulière.

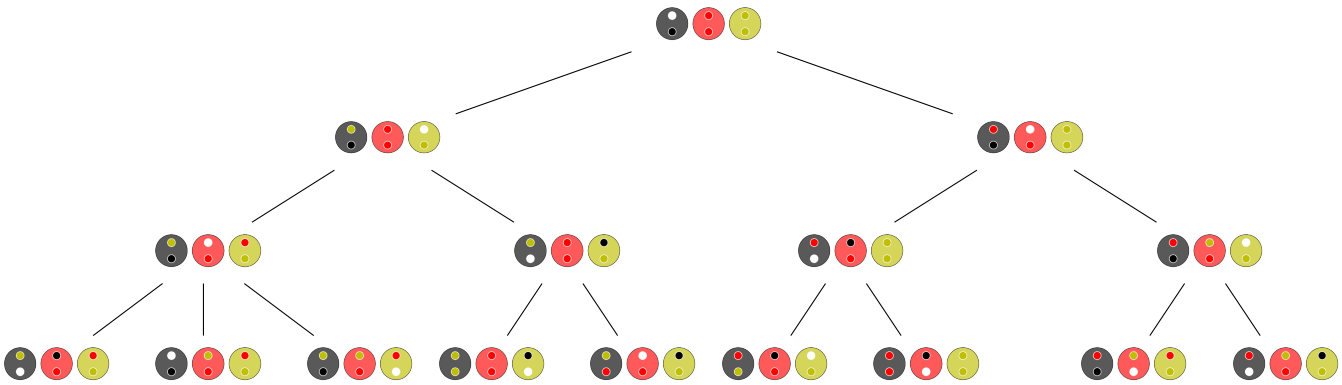
Précisions ce que nous cherchons à faire : nous voulons trouver une solution peu coûteuse. Très bien ! Mais dans ce cas, comment évalue-t-on ce coût ? Nous choisissons de chercher à minimiser le nombre de déplacements du trou (donc toute opération autre que le déplacement d'un jeton ne sera pas comptabilisée). Dans le premier cas ci-dessus, le coût est strictement plus grand que 9, tandis que pour le second, il vaut exactement 9. La section suivante va proposer un algorithme donnant à coup sûr la solution la moins coûteuse.

5.2 Un algorithme donnant toujours la solution optimale

Nous allons exploiter une idée toute simple à comprendre. Partons de la configuration gagnante pour trouver toutes les configurations obtenues en faisant un seul mouvement. À partir des précédentes nouvelles configurations, nous recherchons ensuite d'autres configurations obtenues en faisant un second mouvement. Ceci peut se résumer par l'arbre ci-dessous où une configuration $\mathcal{C}_1$ est reliée à une autre $\mathcal{C}_2$ uniquement si l'on peut passer de $\mathcal{C}_1$ à $\mathcal{C}_2$ en un seul mouvement. De plus, quand on descend dans l'arbre on ne garde que les nouvelles configurations.



Avec un mouvement de plus, nous avons l'arbre ci-dessous (qui n'est pas symétrique : voir en bas à gauche).



Avec de la patience, ou grâce à un programme, on peut fabriquer l'arbre complet (vous le trouverez en annexe). Notons que pour un jeu à cinq bases, il y a tout de même 11 010 configurations (ceci est justifié en annexe), donc représenter l'arbre complet pour 5 bases sur une feuille A3, même avec l'aide d'un programme, ne sera pas possible.

Il est évident que la méthode que nous employons va finir par trouver toutes les configurations tout en nous indiquant leur résolution en faisant le minimum de coups possible.

Notre démarche peut se traduire par l'algorithme ci-dessous où nous utilisons des dictionnaires qui sont des objets associant une valeur à une clé. Par exemple, `mon_dico = {"un": 1, "deux": 2}` admet pour clés `"un"` et `"deux"`, et nous notons `mon_dico<"un"> = 1` la valeur associée à la clé `"un"`. Nous utilisons aussi [ ] pour indiquer une liste vide prête à être remplie.

Donnée : une configuration quelconque de début de jeu
Résultat : la solution gagnante (en utilisant le moins de déplacements possible)

Début

$\mathcal{G}$ désigne la configuration gagnante.

`// "L'arbre" sera construit sous forme d'un dictionnaire.`
`//   • les clés sont les configurations possibles.`
`//   • chaque valeur donne la liste des configurations à suivre pour gagner`
`//     le plus rapidement possible (lecture de la liste de gauche à droite).`

$\mathcal{A} \leftarrow \{\mathcal{G} : [\,]\}$

`// Liste stockant les [n]ouvelles [c]onfigurations d'où partir.`

$NC \leftarrow [\,\mathcal{G}\,]$

Tant Que *NC n'est pas vide* **:**

$NC_{ap} \leftarrow [\,]$ va stocker les configurations d'où partir durant l'étape d'après.

Pour Chaque *configuration $\mathcal{C}$ dans NC* **:**

Pour Chaque *configuration $\mathcal{V}$ obtenue en un seul mouvement depuis $\mathcal{C}$* **:**

Si *$\mathcal{V}$ n'est pas une clé de $\mathcal{A}$* **:**

Ajouter $\mathcal{V}$ à NC_{ap}.

$COUPS$: liste obtenue en ajoutant $\mathcal{C}$ à gauche de la liste $\mathcal{A}\langle\mathcal{C}\rangle$.
Ajouter $\mathcal{V}$ à $\mathcal{A}$ avec $\mathcal{A}\langle\mathcal{V}\rangle = COUPS$.

$NC \leftarrow NC_{ap}$

`// Utilisation de l'arbre pour résoudre le jeu.`

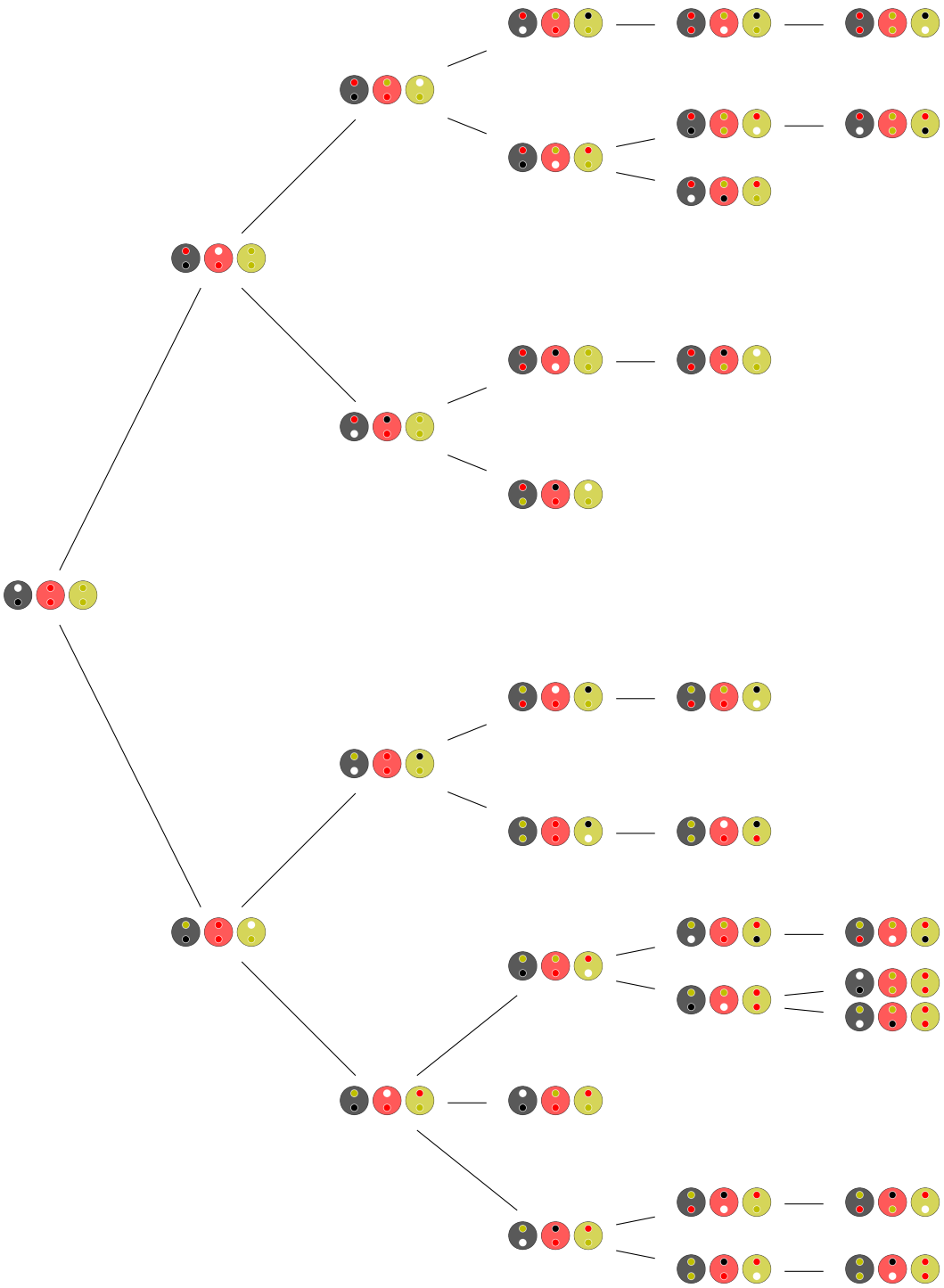
$\mathcal{C}_0$: la configuration à résoudre.
Obtenir successivement les configurations stockées de gauche à droite dans $\mathcal{A}\langle\mathcal{C}_0\rangle$.

Remarque : l'algorithme présenté dans cette section est facile à programmer mais il devient inutilisable par un humain dès le jeu à quatre bases qui permet d'avoir 480 configurations différentes (voir l'annexe pour le calcul de cette valeur).

6 Annexe

6.1 Toutes les configurations pour un jeu à trois bases

L'arbre ci-dessous [7] permet de gagner en un minimum de coups au baseball des couleurs pour trois bases (il y a 33 configurations différentes et dans le pire des cas on peut gagner en cinq coups). Pour cela, repérer la configuration à résoudre puis suivre les arêtes de la droite vers la gauche pour avoir les mouvements à faire.



7. Bien entendu, cet arbre a été obtenu via un programme et non à la main.

6.2 Calculer le nombre de configurations possibles

On peut se demander combien de configurations de jeux sont possibles. C'est ce que nous allons chercher à calculer. Soit $C(n)$ le nombre de configurations de jeux pour $n \geqslant 1$ base(s) (par exemple, $C(1) = 1$). Nous avons au total $(2n-1)$ jetons et n « *couleurs* » différentes. Nous dirons que le jeton associé au trou est le jeton noir. Enfin, nous allons noter $\mathcal{B}_1$, $\mathcal{B}_2$, ... , et $\mathcal{B}_n$ les bases « *rangées* » dans l'ordre initial de mise en place du jeu. Dans la suite, nous supposerons $n \geqslant 2$.

Commençons par examiner les configurations telles que la base $\mathcal{B}_1$ contienne le trou. Nous avons alors deux situations [8].

1. $\mathcal{B}_1$ *contient aussi le jeton noir.* Dans ce cas, nous devons compter le nombre de façons différentes de placer des jetons à partir de $(n-1)$ paires de jetons de même couleur sur $(n-1)$ bases, sachant que sur chaque base l'ordre des jetons n'est pas important. Nous n'allons pas chercher à calculer ce nombre. Nous le notons juste $P(n-1)$ (les hypothèses sur les couleurs sont similaires à celles du jeu du baseball des couleurs sauf que l'on n'a plus de trou, ni de jeton noir associé au trou). Indiquons que l'on a clairement $P(1) = 1$.
2. $\mathcal{B}_1$ *ne contient pas le jeton noir.* Notant c la couleur du jeton dans la base $\mathcal{B}_1$, sur les $(n-1)$ bases restantes, il reste à placer $(n-2)$ paires de jetons de même couleur, ainsi que deux jetons de couleurs différentes « *sans jumeau* », à savoir les jetons de couleurs respectives c et noire. Interprétant le jeton de couleur c comme un trou, nous retombons sur une configuration d'un jeu de baseball des couleurs mais avec $(n-1)$ jetons. Ceci nous fait donc $C(n-1)$ possibilités associées à la couleur c.

D'après ce qui précède, il y a donc $P(n-1) + (n-1)C(n-1)$ configurations possibles telles que la base $\mathcal{B}_1$ contienne le trou. Dans $(n-1)C(n-1)$, le « *fois* $(n-1)$ » vient de ce que l'on a $(n-1)$ choix possibles pour la couleur c.

En répétant le raisonnement précédent, et en notant qu'il y a n choix de bases où placer le trou, nous arrivons à la relation $C(n) = n\,[P(n-1) + (n-1)C(n-1)]$. Nous réécrivons ceci comme suit :

$$C(n) = nP(n-1) + n(n-1)C(n-1) \tag{1}$$

Nous allons reprendre un raisonnement similaire à ce qui a été fait ci-dessus pour trouver une relation de récurrence pour tenter d'évaluer $P(n)$ le nombre de façons de placer des jetons à partir de n paires monochromes de jetons sur n bases [9]. De nouveau, nous allons d'abord raisonner sur la base $\mathcal{B}_1$. Deux cas sont possibles.

1. $\mathcal{B}_1$ *contient deux jetons de la même couleur* c. Dans ce cas, il y a $P(n-1)$ possibilités de placer les autres jetons sur les bases restantes. En effet, on a enlevé une paire monochrome de couleur c et la base $\mathcal{B}_1$ de couleur d. Si $c = d$, l'affirmation est évidente, sinon il suffit d'associer la couleur de jeton d à la couleur de base c pour les bases et les jetons restants.
2. $\mathcal{B}_1$ *contient deux jetons de couleurs différentes* c_1 *et* c_2. Dans les jetons restants, il y a juste deux jetons sans « *jumeau* », à savoir ceux de couleurs c_1 et c_2. Interprétant c_1 comme étant la « *couleur* » du trou, et c_2 celle du jeton associé au trou, nous avons alors une configuration de type baseball des couleurs pour les jetons et les bases restants. Nous avons donc $C(n-1)$ possibilités dans ce cas.

Dans le premier cas, pour la base $\mathcal{B}_1$, il y a n choix de couleurs, ce qui nous fait $nP(n-1)$ configurations avec une première base monochrome. Dans le second cas, pour la base $\mathcal{B}_1$ nous avons n choix pour la première couleur, et $(n-1)$ pour la seconde (car l'on veut deux couleurs différentes). Comme l'ordre ne compte pas, car tirer c_1 puis c_2, ou bien tirer c_2 puis c_1 nous donne à chaque fois la même base complétée, nous avons

8. Le lecteur notera que ce qui compte dans les raisonnements de cette section, c'est la couleur tirée et non le jeton tiré. C'est une subtilité importante à noter. Dans une première version de ce document, l'auteur avait raisonné sur les jetons, ce qui lui avait fait obtenir et « *démontrer* » une formule fausse. Cette erreur a été repérée très rapidement lors de la fabrication de l'arbre présenté dans la section précédente.

9. Il y a autant de couleurs différentes que de bases.

donc $\frac{n(n-1)}{2}$ façons de remplir la première base avec deux couleurs différentes. Nous en déduisons que nous avons $\frac{n(n-1)}{2}C(n-1)$ configurations avec une première base non monochrome. Nous arrivons finalement à la relation suivante.

$$P(n) = nP(n-1) + \frac{n(n-1)}{2}C(n-1) \tag{2}$$

En résumé, nous avons démontré les deux relations de récurrence suivantes pour $n \in \mathbb{N}^* - \{1\}$ avec les conditions initiales $C(1) = P(1) = 1$:

$$\begin{cases} C(n) = nP(n-1) + n(n-1)C(n-1) \\ P(n) = nP(n-1) + \dfrac{n(n-1)}{2}C(n-1) \end{cases} \tag{3}$$

Ceci n'est pas très difficile à programmer. En utilisant le logiciel SageMath directement en ligne à l'adresse suivante `https://sagecell.sagemath.org`, il suffit d'insérer le code de type Python suivant.

```
def C(n):
    if n == 1:
        return 1

    return n * P(n - 1) + n*(n-1)*C(n - 1)


def P(n):
    if n == 1:
        return 1

    return n * P(n - 1) + n*(n-1)/2*C(n - 1)


for k in range(2, 6):
    print "Pour k =", k, ":", C(k)
```

Ceci nous donne les résultats suivants.

- $C(2) = 4$ ce qui est calculable directement en imaginant les configurations possibles avec deux bases.
- $C(3) = 33$, $C(4) = 480$ et $C(5) = 11\,010$.

Par contre, si l'on veut par exemple calculer $C(36)$, il faut être un peu plus précautionneux car dans le code précédent certains calculs sont effectués plusieurs fois. Le code suivant permet d'obtenir instantanément $C(36)$ alors que celui ci-dessus fait ceci très, très, très lentement... L'astuce consiste à stocker tout ce qui est calculé et de regarder si une valeur à calculer a déjà été stockée en mémoire. Si c'est le cas, on récupère directement cette valeur, sinon on la calcule et on la stocke en mémoire en vue d'éventuelles utilisations à venir (nous faisons ceci « *à la main* » en utilisant des dictionnaires Python).

```
valsP, valsC = {}, {}


def C(n):
    if n == 1:
        return 1

    global valsC

    if n in valsC:
        return valsC[n]
```

```
    val      = n * P(n - 1) + n*(n-1)*C(n - 1)
    valsC[n] = val

    return val


def P(n):
    if n == 1:
        return 1

    global valsP

    if n in valsP:
        return valsP[n]

    val      = n * P(n - 1) + n*(n-1)/2*C(n - 1)
    valsP[n] = val

    return val


print C(36).n(10)
```

Ceci nous donne, sans temps de latence, $C(36) \approx 4{,}2 \times 10^{82}$ que l'on comparera à 10^{80} qui est une estimation du nombre d'atomes dans l'univers [10].

10. Notons que si l'estimation du nombre d'atomes de l'univers est juste, alors il est tout simplement impossible « *d'écrire* » tous les naturels de 1 à $C(36)$ en associant chaque naturel à un seul atome de l'univers. Vertigineux !