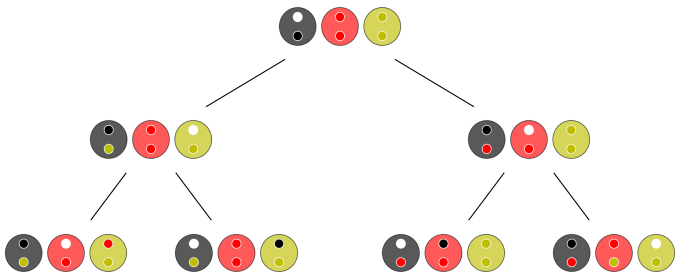
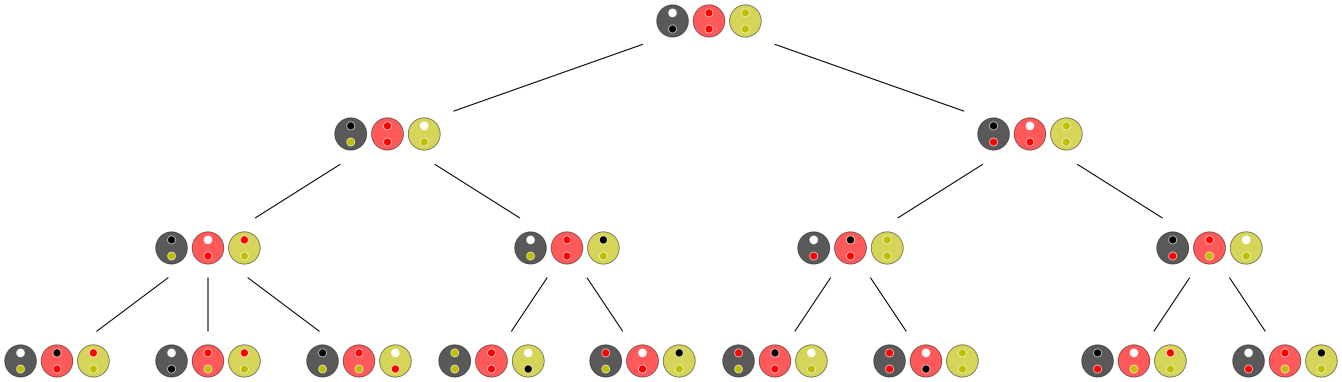
0.1 Un algorithme donnant toujours la solution optimale

L'idée est toute simple à comprendre. On commence en partant de la configuration gagnante pour trouver toutes les nouvelles configurations obtenues en faisant un seul mouvement. À partir de ces nouvelles configurations, on en recherche d'autres nouvelles 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, et 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 à gauche perd sa symétrie.



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 comme ceci est justifié dans l'annexe. Donc représenter l'arbre complet pour 5 bases sur une feuille A4, même avec l'aide d'un programme, ne sera pas possible.

? ? ? ?
idée que l'on en peut pas faire mieux, et traductiona lgorithmqiue, qui est un préalable pour programmer cette méthod qui n'est pas utilisable par humain (sauf à utiliser une armée de servants)
Voici l'algorithme qui va nous donner le moins possible de déplacements pour arriver à la configuration gagnante. Nous allons utiliser 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".

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

Début

```
// Construction de toutes les configurations tout en gardant en mémoire le moins possible
// de déplacements pour accéder à chaque nouvelle configuration construite.
```
? ? ?

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

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

? ? ?

```
// ????
```

Les instructions étant sans ambiguïté, nous allons pouvoir nous attaquer très sérieusement à la validation des propriétés de « *finitude* » et de « *résolution* ».

Démonstration. ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? □

Remarque : ? ? ? ?