Les marmottes au sommeil léger

... une idée farfelue de Marie Duflot-Kremer
créée en juin 2016

dernière modification le 25 août 2017

Le document que vous êtes en train de consulter n'est pas une référence très finalisée, ni un guide strict à suivre. Il regroupe des infos que vous pourriez trouver utiles si vous envisagez d'animer cette activité. Il contient l'histoire des marmottes, le déroulé de l'activité, l'explication côté informatique ainsi qu'un patron à imprimer pour réaliser l'activité. La diffusion de ce document est libre, vous pouvez suggérer des améliorations/enrichissements à `marie.duflot-kremer@loria.fr`. Si ce document vous a été utile, vous pouvez également me le signaler car j'envisage de mentionner sur ma page médiation les écoles/associations/etc. qui ont testé et approuvé l'activité. La page `https://members.loria.fr/MDuflot/` permet de trouver (section médiation/activités) une liste d'autres activités pour faire découvrir différents aspects de l'informatique, réalisables en grande majorité sans ordinateur.

Étape 1 — Le contexte

Un groupe de marmottes, moyennement satisfaites de leur terrier actuel, décide de concevoir un nouveau terrier et de le creuser avant l'hiver. Pour ce faire les marmottes doivent respecter trois règles.

1. A partir de l'entrée on peut construire deux couloirs, et au bout de chaque couloir on peut faire un embranchement vers deux autres, mais pas plus (au risque de faire s'écrouler l'édifice).
2. Les marmottes vont chacune occuper une salle différente (pour ne pas se réveiller les unes les autres) et forcément une salle qui est tout au bout d'un couloir. Pour des marmottes au sommeil léger il est inenvisageable de dormir dans une salle à un embranchement, car les marmottes qui seraient au-delà de cet embranchement leur marcheraient dessus en entrant/sortant, et cela ruinerait leur hibernation
3. Chaque marmotte se réveille un nombre précis de fois dans l'hiver, et s'il n'est pas grave de mettre assez loin de l'entrée une marmotte qui ne se réveille (et donc ne sort du terrier) qu'une fois dans l'hiver, c'est bien plus embêtant de mettre loin une marmotte qui va se réveiller 10 fois par exemple. Comme les pas de marmottes émettent de légères vibrations et que nos marmottes ont vraiment le sommeil léger, on va vouloir minimiser les déplacement de l'ensemble du groupe. Pour cela on va compter les déplacements de la façon suivante. Une marmotte dormant à 4 couloirs de l'entrée se réveillant 5 fois dans l'hiver va parcourir 4 x 5 = 20 couloirs aller et retour (pour simplifier on ne va compter que les allers). On va faire la somme des déplacements de toutes les marmottes et essayer de rendre cette somme la plus petite possible.

Étape 2 — Préparer le matériel

Avec le modèle de couloirs et de marmottes donné, on peut préparer un kit. Pour cela il vous faut une imprimante, un stylo, des ciseaux, une plastifieuse et des feuilles plastiques qui vont avec, et un peu de scratch adhésif (se trouve en mercerie). Ensuite il vous suffit de :

— tout imprimer,
— découper couloirs et marmottes (pour les marmottes laisser un peu de place autour, qui représente la salle dans laquelle elle dort),
— ecrire des 0 (branche de gauche) et des 1 (branche de droite) au dos des couloirs,

— choisir une phrase à compresser ne contenant pas plus de 8 caractères différents (sinon il nous faut plusieurs planches pour un kit) et compter la fréquence d'apparition des caractères (espace compris),
— noter au dos de chaque marmotte un des caractères et sa fréquence (par exemple A 6) et recopier la fréquence côté marmotte (c'est le nombre de fois où elle se réveille pendant l'hiver)
— plastifier le tout (marmottes + couloirs) et redécouper en laissant un peu de marge pour que cela tienne bien,
— placer de petits morceaux de scratch adhésif de la façon suivante :
  — des morceaux tout doux au dos du coude des couloirs et au dos des marmottes (sans cacher la lettre qui y est écrite !)
  — des morceaux grattants côté recto aux extrémités des couloirs

Pour un (des ?) exemple de phrase et de de nombres à écrire aller voir la section sur l'explication informatique.

La plastification se fait ici une fois les éléments découpés pour que le plastique soit bien scellé sur les bords de chaque objet.

Étape 3 — Et maintenant, creusez

Pour commencer on va distribuer des kits et expliquer les règles ci-dessus. Les participants peuvent donc expérimenter et créer eux-mêmes leurs terriers, en essayant d'avoir le meilleur possible. L'animateur doit donc avoir résolu le problème avant et savoir dire au vu d'un terrier s'il est optimal ou non (la bonne nouvelle c'est qu'on peut lire d'abord la suite et avoir l'algo qui donne le nombre de déplacements minimal).

Pour compter le nombre de déplacements au total, on a deux solutions :
— pour chaque marmotte, calculer sa distance jusqu'à l'entrée, faire la multiplication avec le nombre de fois qu'elle se réveille dans l'hiver, puis faire la somme de tous ces résultats
— ... ou bien, on peut calculer pour chaque pièce de couloir combien de fois elle est traversée. Par exemple, pour un couloir qui relie deux marmottes se réveillant respectivement 5 et 2 fois, on note dans le coude du couloir la valeur $5 + 2 = 7$. Si un couloir relie maintenant une marmotte se réveillant 4 fois et le couloir précédent (qui porte donc la valeur 7), on note sur son coude la valeur $7 + 4 = 11$. On continue jusqu'à l'entrée du terrier, puis il reste juste à faire la somme de tous les chiffres écrits dans les couloirs (et pas à prendre le chiffre écrit dans le couloir le plus en haut, ce que font certains participants).

Le point positif est qu'on ne peut pas vraiment se "tromper" en faisant cette étape, on finit par avoir un terrier, et puis on essaie de le transformer pour l'améliorer.

Étape 4 — Comment trouver le meilleur terrier ?

Pour trouver le meilleur terrier, il y a une méthode infaillible, qui ne demande pas trop de calculs :
— on choisit les deux (ou deux parmi les) marmottes qui se lèvent le moins souvent, et on les relie par un morceau de terrier. Sur le coude on note le nombre de fois que ce morceau est emprunté (donc la somme des valeurs des deux marmottes, par exemple $2 + 3 = 5$)
— on recommence exactement la même chose, mais les deux marmottes reliées à l'étape précédente comptent maintenant pour une seule marmotte qui se réveillerait 5 fois
— on continue, jusqu'à ce que toutes les marmottes soient reliées en un seul terrier.
— et comme on a noté des chiffres sur les coudes de terriers au fur et à mesure on a juste à faire la somme de ces chiffres pour compter le nombre de déplacements

Étape 5 — Et hop, c'est de l'informatique

Oui bon c'est bien sympa tout ça mais quel est le rapport avec l'informatique ? C'est juste qu'on a suivi un algorithme pour faire le meilleur terrier ? Eh non, parce que cet algorithme n'aide pas que les marmottes (en fait il n'est même pas prouvé qu'il les aide, aucune marmotte n'a été dérangée pour créer cette activité) mais est utilisé dans le domaine de la compression de texte.

C'est là qu'on retourne les terriers (qui tiennent si on a bien mis les scratchs) et qu'on découvre la version informatique du problème. La structure que l'on obtient est celle d'un arbre, avec (oui les informaticiens sont des blagueurs) la racine en haut, les couloirs qui se nomment des branches, et les extrémités des branches appelées feuilles. Et dans cet arbre une marmotte correspond à une lettre, et son nombre de réveils au nombre de fois que la lettre apparaît dans un texte.

Par exemple dans la phrase BARBARA RASE BASILE LE BARBIER, il y a 6 A, 5 B, 4 espaces etc. qui seraient au dos de marmottes se réveillant, 6, 5, 4 fois etc.

Pourquoi le code ASCII n'est pas efficace ? Le but de la compression est de trouver un codage du texte, en binaire parce que dans un ordinateur tout est stocké en binaire, qui soit le plus court possible, et qui soit facile à coder et décoder. En général pour représenter un texte non compressé, on utilise le code ASCII ou l'une de ses extensions. Mais même rien qu'en ASCII on a 8 chiffres binaires (8 bits) pour stocker un caractère. Si ce n'est pas grave que le code du "w" ou du "k" soient longs en français, par contre le "e" qui apparaît plus souvent on aimerait bien que son code soit plus court, et c'est précisément ce que fait le codage de Huffman : donner à chaque lettre un code binaire tel que si on met bout à bout le code de toutes les lettre du texte, on a une version codée la plus courte possible.

Sur notre structure de terrier on peut facilement lire le code de chaque lettre : on part de la racine du terrier et on lit le bit écrit sur chaque couloir que l'on emprunte (cf section matériel). Par exemple si pour aller de l'entrée à la lettre A on prend à gauche puis à droite puis à droite, le code de A est 011.

Justification des contraintes Les trois contraintes que l'on a imposées sur les terriers se justifient quand on veut faire de la compression de texte. La preuve ci-dessous :

- tout d'abord minimiser le nombre de déplacements dans le terrier correspond exactement à minimiser la taille du texte en binaire. On aura autant de chiffres binaires pour coder tout le texte qu'on avait de déplacement de marmottes dans les couloirs (si on a compté les allers et pas les retours),
- ensuite on n'autorisait pas plus de deux nouveaux couloirs partant de l'entrée ou du bout d'un couloir. Cela correspond au binaire. Comme on a deux valeurs à mettre sur les couloirs à chaque étape (0 et 1), on ne peut avoir que deux couloirs qui partent,
- enfin il était interdit de mettre une marmotte dans une salle d'où partent de nouveaux couloirs, car ce la la réveillerait. Côté informatique cette propriété que l'on impose (les marmottes en bout de couloirs) correspond à un codage préfixe (voir ci-dessous).

Codage préfixe Imaginez que je donne le code 0 à la lettre E, 01 à la lettre S et 10 à la lettre T. Pour coder ce n'est pas difficile : ETES donne 0 puis 10 puis 0 puis 01 soit 010001. Par contre pour décoder c'est franchement plus difficile. Si je lis 10010001 je sais que ça commence par un T, et puis il me reste 010001 que je peux décoder en ETES ou SEES. Sur cet exemple je sais que TETES est un mot et TSEES pas, mais sur un long texte je vais au mieux perdre énormément de temps, au pire avoir plusieurs possibilités et ne pas savoir comment décoder.

Le problème est que 0 est à la fois le code de E et le début du code de S, donc si je vois un 0 je ne sais pas si je peux décoder E tout de suite ou prendre le prochain bit et les décoder ensemble. Si au contraire on impose un codage préfixe, on interdit que le code d'une lettre soit le début du code d'une autre. Du coup on va commencer à la racine, suivre les branches en fonction des 0 et 1 dans le texte compressé, et dès qu'on arrive sur une lettre on la note (comme on ne peut pas continuer plus loin car la branche s'arrête là, on est sûr d'avoir lu le code de cette lettre) et on repart de la racine en continuant à lire les chiffres binaires. Le décodage/décompression se fait donc très rapidement et sans hésitation.

Et les codes qui ne sont pas préfixes, on peut les jeter ? Non, pas tous. Si on regarde le code Morse par exemple, on a le E (un point) qui est préfixe du I (deux points). La différence c'est qu'en Morse on fait une légère pause entre deux lettres qui permet de savoir quand une lettre s'arrête et ainsi différencier un double e d'un i. En informatique tous les 0 et les 1 sont rangés les uns à la suite des autres, sans pause, et c'est pour cela qu'on a besoin d'un codage préfixe.

Efficacité de l'algorithme de Huffman Pour donner un exemple d'efficacité, sur le texte intégral de Cyrano de Bergerac, le gain obtenu en passant du texte (codé en UTF-8) non compressé à la version

compressée avec la méthode ci-dessus est de 40%. On a donc diminué quasiment de moitié la taille du texte, rien qu'avec un algorithme explicable à des enfants. Pas mal non ?

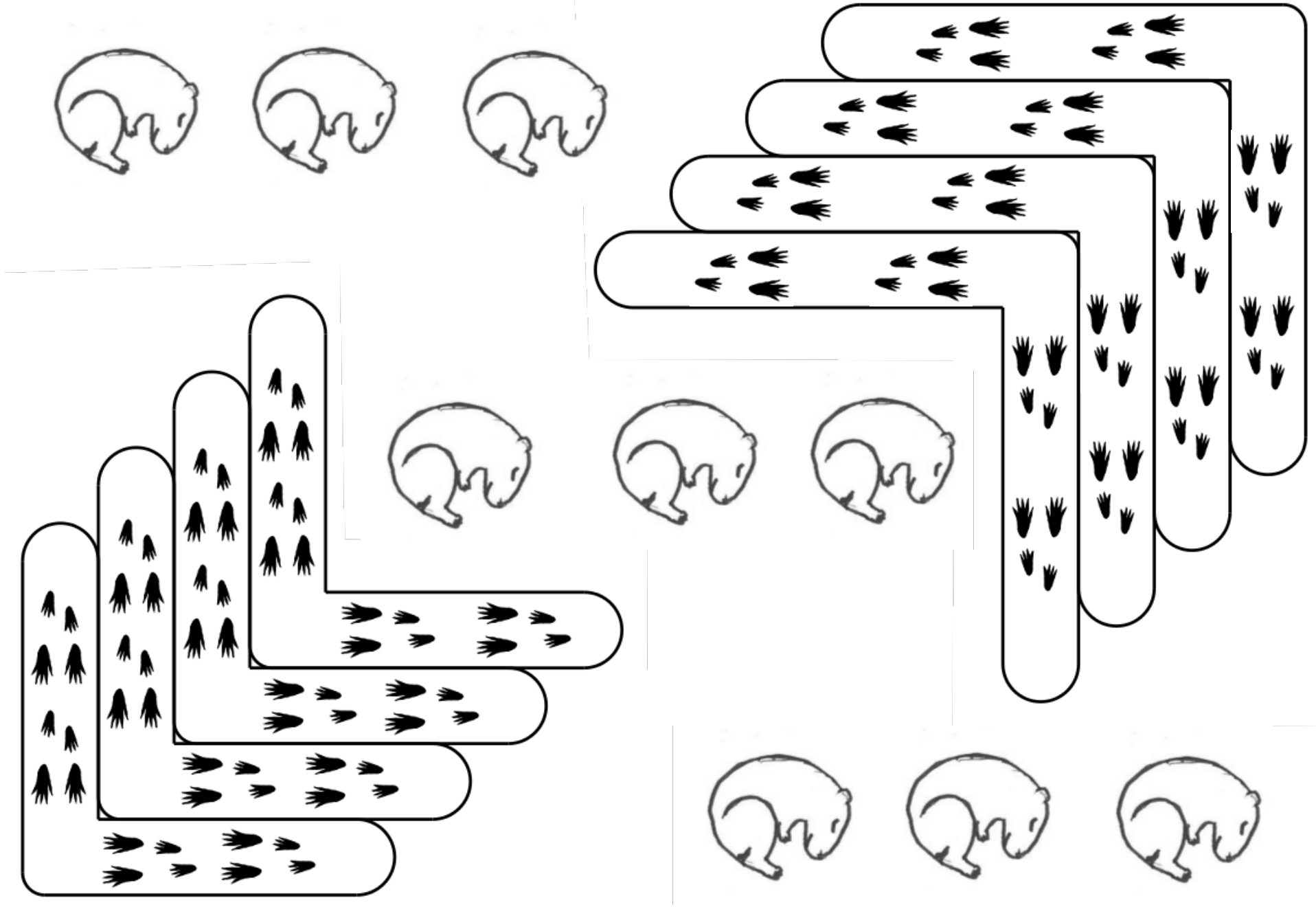
Oui mais notre interlocuteur, comment sait-il quel arbre on a choisi ? Il y a deux solutions : soit on fait un arbre une bonne fois pour toutes, en se basant par exemple sur les fréquences d'apparition de lettres dans une langue donnée, on se l'échange une fois et on travaille avec cet arbre tout le temps. Soit pour chaque texte envoyé on fait un nouvel arbre afin d'avoir le code optimal pour ce texte, mais dans ce cas il faut transmettre le texte... plus l'arbre. Ca prend un peu plus de place, mais pour de grands textes comme celui de Cyrano cité en exemple c'est presque négligeable.

Étape 6 — Et en vrai, comment ça se passe ?

Cette activité a pour moi plusieurs avantages. Le côté ludique des marmottes aide à se projeter et entrer dans l'activité, mais ce n'est pas tout.

- tout d'abord l'algorithme de compression présenté n'est pas un algorithme jouet conçu exprès pour ça mais un véritable algorithme de compression, inventé par David Albert Huffman en 1952, alors doctorant au MIT,
- et cet algorithme a la particularité d'être optimal, c'est à dire que si on donne un code binaire à chaque lettre on ne peut pas faire mieux que le taux de compression obtenu avec l'algorithme de Huffman,
- et enfin, et cela ne gâche rien, l'algorithme de Huffman est toujours utilisé, plus de 60 ans plus tard. On ne l'utilise pas seul, mais les formats compressés que l'on connaît : zip pour notamment les fichiers texte, mp3 pour le son, mp4 pour la vidéo etc, utilisent l'algorithme de Huffman. Il se composent en général d'une première étape, plus complexe et dépendante du type d'information à compresser (image, son, texte), suivie par une application du codage de Huffman.

Un dernier point. Si l'algorithme de Huffman est optimal, pourquoi fait-on une autre étape de compression aux fichiers texte ? En fait le codage de Huffman est le meilleur quand on impose d'associer un code à chaque lettre. Si on veut faire mieux, l'idée est, avant d'appliquer Huffman, de décider à quels blocs du texte on va associer des codes. En français par exemble, les digrammes (blocs de 2 lettres) "es", "en" ou "le" sont les plus fréquents. Il est donc plus efficace de choisir un code pour "es" plutôt que de le coder avec le code de "e" plus celui de "s". Et c'est en codant plusieurs lettres d'un coup que l'on peut aller plus loin que l'algorithme de Huffman dans la compression de textes.



Étape 7 — Mais pourquoi est-ce optimal ?

Cette section a été ajoutée parce qu'on me posait la question. Si vous ne voulez pas spécialement vous la poser, vous pouvez tout à fait vous en passer. Si vraiment ça vous intéresse, lisez ce qu'il y a ci-dessous, mais je vous aurai prévenus !

1. Tout d'abord on montre ce que ça fait dans un arbre d'échanger deux feuilles. Mettons qu'on a une lettre de poids 5 à la profondeur 3 et qu'on l'échange avec une lettre de poids 2 à la profondeur 6. Leur coût avant (la place que cela prenait pour coder 5 fois la lettre de code de taille 3 + 2 fois la lettre dont le code est de longueur 6) était 5 x 3 + 2 x 6 = 15 + 12 = 27, et après le coût devient 5 x 6 + 2 x 3 = 36. La différence "après - avant" est donc 5 (=le coût de la lettre qui s'éloigne de l'entrée)- 2 (=le coût de celle qui se rapproche), le tout x 3 (l'écart de profondeur entre les deux positions), ici (5 - 2) x 3 = 9
2. En se basant sur cela, on peut prouver que, si notre ensemble contient au moins deux lettres, il existe un arbre optimal pour lequel les deux lettres de poids minimum (ou deux parmi les lettres de poids minimum) $l1$ et $l2$ sont reliées ensemble. Si on a un arbre optimal pour lequel ce n'est pas le cas, prenons une lettre $l3$ qui est le plus loin de la racine. On a alors deux cas :
   — soit $l3$ est reliée à une autre lettre $l4$ à la même hauteur
   — soit $l3$ n'est reliée avec personne (il n'y a aucune lettre à l'autre bout de son couloir), auquel cas on peut enlever le couloir et raccrocher la lettre $l3$ une place plus haut dans l'arbre. C'est impossible car ça donnerait un arbre de poids total plus petit et on a supposé que notre arbre était optimal.

   Du coup $l4$ existe et si on échange $l1$ et $l3$ puis $l2$ et $l4$ (remarque : on n'est pas sûrs que $l1$ et $l2$ soient tous différents de $l3$ et $l4$ mais peu importe), comme $l1$ et $l2$ sont soit à la même profondeur soit moins loin de la racine, et que $l1$ et $l2$ ont un poids inférieur ou égal à $l3$ et $l4$, au pire le poids de l'arbre ne change pas, au mieux il diminue. Le nouvel arbre est donc optimal et $l1$ et $l2$ sont reliées ensemble.
3. Ensuite il faut se convaincre que, quand on a deux lettres $l1$ et $l2$ de poids respectif $p1$ et $p2$ qui sont reliées ensemble dans un arbre A, le poids de l'arbre (on le note $poids(A)$) est le même que $poids(A') + p1 + p2$, où A' est construit en enlevant à A les lettres $l1$ et $l2$ avec le couloir qui les relie, et en rajoutant au bout du couloir devenu vide une lettre de poids $p1 + p2$. En effet, si les lettres $l1$ et $l2$ sont à une profondeur n de la racine, le poids de A est égal à la somme du poids des autres lettres + $n \times p1 + n \times p2$ et le poids de A' est égal à la somme du poids des autres lettres + $(n-1) \times (p1 + p2)$.
4. Enfin on va montrer qu'un arbre créé grâce à l'algorithme de Huffman est optimal. Pour cela on le fait par induction, à savoir qu'on va montrer que c'est vrai pour un arbre à une lettre, puis prouver que si c'est vrai pour tout arbre à strictement moins de k lettres alors c'est aussi vrai pour les arbres de k lettres.
   — tout d'abord à une lettre c'est simple, on la met directement à la racine de l'arbre, pas besoin de couloir et le poids total est 0. C'est du coup optimal (mais d'un point de vue codage de texte ça n'a pas vraiment de sens). Avec deux lettres pas de choix non plus, on les relie ensemble et c'est réglé.
   — on suppose ensuite que la propriété est vraie pour tous les arbres jusqu'à $k-1$ lettres (on appelle cela notre hypothèse d'induction). Prenons maintenant un ensemble de k lettres et appelons A_H un arbre obtenu à partir de ces lettres en utilisant l'arbre de Huffman. Comme il a été créé en commençant par relier entre elles deux lettres se poids minimum (mettons l_1 et l_2), on va appeler maintenant A un arbre optimal ayant $l1$ et $l2$ reliés entre eux, vu qu'on a montré qu'il existe. D'après ce qu'il y a ci-dessus on sait que le coût de A est égal au coût de A' défini ci-dessus + $p1$ + $p2$. De même pour A_H son coût est égal à celui de A'_H (à savoir A_H à qui on a fait la même transformation que de A vers A')+$p1$ + $p2$. Or A'_H et A' sont deux arbres, de taille strictement plus petite que k, construits avec les mêmes lettres l'un et l'autre. Comme A'_H a été créé en suivant l'algorithme de Huffman (qui nous dit précisément qu'une fois deux lettres reliées ensemble on les considère comme une seule lettre dont le poids est la somme des

deux poids de départ) et est de taille $< k$, notre hypothèse d'induction nous dit que A'_H est optimal, et a donc un coût inférieur ou égal à celui de A' . On en déduit :

$$\begin{aligned} poids(A_H) &= poids(A'_H) + p_1 + p_2 \\ &\leq poids(A') + p_1 + p_2 \\ &\leq poids(A) \end{aligned}$$

On a donc le poids de A_H qui est inférieur où égal à celui de A. Comme A est par définition optimal, A_H l'est aussi et la preuve est finie.