



Assistant constructeur de deck

2017 - 2018

Bellamy Lola

Bourgeois Ducournau Guillaume

Master 2 - Informatique

Développement d'applications réparties

Index

Introduction

Objectifs

Choix des technologies

Langage principal

Base de données

Interface graphique

Implémentation

Structures et objets

Evaluation du poids des arêtes

Recherche des meilleurs associations

Evaluation du deck

Interface graphique

Pour aller plus loin

Déséquilibre des couleurs

Autres paramètres d'évaluation

Introduction

Avec une première édition sortie en 1993, Magic est le jeu de cartes à jouer et collectionner le plus ancien dans son genre. Avec plus de 200 éditions et 17 000 cartes aujourd'hui, il s'agit d'un jeu complexe où chaque joueur doit sélectionner un ensemble de cartes qui composeront son deck.

Face à de si nombreuses possibilités, il peut être avantageux de disposer d'une aide pour le choix de ses cartes. C'est sur un outil visant à assister un joueur dans la construction de son deck que porte le projet associé à ce rapport.

En effet, l'utilisateur sera libre de construire son deck comme il le souhaite tout en bénéficiant d'un ensemble de propositions et de statistiques visant à le guider pour obtenir un deck dont les cartes qui le composent sont à la fois efficaces quand associées et assez diversifiées pour obtenir un deck équilibré et viable.

Nous aborderons donc dans ce rapport les différentes étapes qui ont mené au développement de ce projet en commençant par en énoncer les objectifs. Nous évoquerons la façon dont nous avons choisi chaque technologie présente ainsi que la façon dont nous les utilisons afin d'implémenter la solution.

Chaque étape de ce rapport contiendra non seulement les résultats auxquels nous sommes arrivés mais également les démarches mises en place pour y parvenir.

Objectifs

1. Mise en place d'algorithmes permettant l'évaluation de la valeur d'un lien entre plusieurs cartes en fonction d'un paramètre donné.
2. Création d'un graphe global permettant de représenter de manière évaluée les liens existant entre les cartes.
3. A partir du graphe précédent, rechercher les meilleures associations de cartes, tous paramètres confondus.
4. Evaluation du deck à chaque choix de carte en fonction de statistiques visant à l'obtention d'un deck final équilibré.

Choix des technologies

I. Langage principal

Le premier choix de technologie n'a pas posé de problèmes, orientés par nos professeurs nous nous sommes directement dirigés sur du C++. En effet, nous savions dès le départ que l'application allait demander de la performance, notamment à cause des nombreux calculs qui allaient être effectués et du grand nombre de données à traiter. Le C++ se prête parfaitement à ce genre d'utilisation car il est connu pour être rapide et performant. C'est d'ailleurs pourquoi il est très utilisé dans le développement d'outils financiers ou dans la programmation de jeux vidéos. Il offre également un grand nombre de fonctionnalités et une bonne portabilité.

L'autre solution envisagée aurait pu être un programme 100% en PHP, en commençant par la récupération des informations dans la base de données (voir ci-dessous), puis en traitant ces dernières et enfin en offrant une interface web. Cela aurait simplifié la première étape et la dernière étape mais nous n'aurions pas profité des mêmes performances.

Il s'est trouvé que le choix du C++ nous a également été très utile par la suite grâce à ses structures de données telles que les maps ou les vectors mais aussi grâce à ses bibliothèques, et plus particulièrement la bibliothèque boost. Cette dernière nous offre des structures de données supplémentaires comme les multimap ou les bimap, qui sont des sortes de "map dérivées" et qui allient performances d'accès et tri automatique. Elles nous ont été très utiles lorsque nous avons voulu faire des opérations sur les cartes parfois en fonction de leurs poids et parfois en fonction de leur identifiant.

II. Base de données

Le deuxième choix a été celui de la base de données et deux solutions s'offraient à nous :

- Créer un script PHP qui récupérerait les informations dont nous avons besoin puis les exporterait pour être traitées dans notre programme en C++
- Récupérer directement la base de données dans notre programme en C++ sans intermédiaire

Nous avons choisi la deuxième solution car nous n'avions pas encore fait ce genre d'opérations en C++, c'était pour nous l'occasion d'apprendre de nouvelles choses et nous voulions nous astreindre d'intermédiaires et pouvoir tout gérer dans le même programme.

Utilisant Visual Studio, il nous a donc fallu installer MySQL Workbench, et faire la connexion à la base de données avec MySQL C++ Connector. Ce fut fastidieux au départ pour des raisons de compatibilité et surtout de prise en main de l'ajout de bibliothèques externes dans un projet Visual Studio.

III. Interface graphique

Plus tard dans le projet il nous a fallu créer une interface graphique pour pouvoir bien visualiser le deck et les cartes qui nous sont proposés. La création d'une interface graphique n'était pas sensée être un sujet majeur du projet mais cela a eu son importance tout de même car nous nous devions d'être efficaces et pouvoir l'implémenter rapidement tout en travaillant avec un outils assez souple pour nous permettre de rajouter des fonctionnalités selon nos besoins. Nous avons donc pris un certain temps pour tester les solutions qui s'offraient à nous et découvrir les avantages et inconvénients de chacune d'elles :

- Application de bureau Windows sous Visual Studio :
 - Prise en main fastidieuse
 - Grand manque de documentation
 - Pas d'interface graphique pour gérer les éléments graphiques
- Bibliothèque SFML :
 - Simple bibliothèque à ajouter au projet
 - Documentation fournie et en français
 - Tout est à gérer dans le code
 - Pas d'éléments graphiques comme les boutons ou labels... Donc les zones cliquables sont à gérer avec la position du curseur par exemple
 - Plutôt adapté à la création de mini-jeux en C++
- Projet entièrement sous Qt :
 - Toutes les fonctionnalités de Qt disponibles
 - Interface graphique pour gérer les éléments graphiques
 - Bonne documentation
 - Projet à recréer sous Qt... Et donc changement de la bibliothèque MySQL C++ Connector pour une autre compatible avec Qt ou implémentation d'une solution en PHP pour récupérer la base de données comme évoqué ci-dessus
- Projet Visual C++ CLI/C++ sous Visual Studio :
 - Bibliothèques compatibles
 - Interface graphique pour gérer les éléments
 - Langage différent : le CLI/C++ est différent du C++, il fallait donc réécrire une bonne partie du code et surtout prendre en main ce nouveau langage
- Projet Qt sous Visual Studio grâce à un plugin



- Difficulté à trouver le bon plugin et la bonne configuration pour l'installation
- Nous pouvions exporter facilement le code que nous avions déjà vers ce nouveau projet
- Disposition d'une interface graphique intuitive pour ajouter les éléments dont nous avons besoin comme les boutons ou les labels pour le texte et les images
- Documentation très fournie et communauté active

Au final nous avons choisi la dernière solution qui s'est avérée être un très bon choix, car nous avons certes passé du temps sur le choix de la technologie mais ensuite la prise en main des fonctionnalités de Qt fut très rapide et nous avons pu proposer une première version de l'interface graphique du programme dans les délais. Nous avons ensuite pu améliorer cette interface pour qu'elle propose plus de fonctionnalités.

Implémentation

I. Structures et objets

a. Cards

❖ Objectif :

Les données sur les cartes récupérées depuis la base de données doivent être stockées afin d'être comparées pour l'évaluation des liens entre les cartes.

❖ Réflexion :

La base de données contenant 18155 cartes différentes, la quantité d'informations est importante mais nous devons stocker tout ce dont nous avons besoin : les couleurs, éditions, blocks, types, sous-types et capacités de chaque carte.

❖ Solution :

Nous avons réduit au minimum l'espace mémoire nécessaire en tirant profit de types propres au c++. Chaque objet Card contient donc les paramètres suivants :

- id & multiverseid : permettent l'identification de la carte dans la majorité des structures que nous détaillerons plus loin
- colors : une séquence de bits représentant la ou les couleurs de la carte parmi les couleurs existantes (Noire, Blanche, Bleue, Rouge, Verte)
- blocs : vecteur d'int16 représentant les blocs d'éditions auxquels est associée la carte
- editions : vecteur d'int16 représentant les éditions auxquelles est associée la carte
- types : vecteur d'int8 représentant les types de carte parmi les suivants : Créatures, Artefact, Enchantement, Éphémère et Sort
- subtypes : vecteur d'int16 représentant les sous types de la carte, par exemple Vampire, Zombie, etc
- capacities : vecteur d'int8 représentant les capacités de la carte

b. Graph

❖ Objectif :

Il s'agit de l'objet autour duquel s'articule la totalité du programme puisqu'il contient les cartes et les valeurs des liens entre celles-ci. Ici encore, il s'agit d'une grande quantité d'informations. Le graphe étant complet, il s'agit de 329 604 025 liens à calculer et stocker.

❖ Réflexion :

Notre implémentation de base était trop volumineuse et comportait des faiblesses. En effet, nous avons créé un objet Edge qui contenait la valeur de chaque critère de l'arête pour représenter les liens entre les cartes.

❖ Solution :

Pour réduire la quantité d'information, nous avons supprimé les arêtes et gardé uniquement la matrice d'adjacence, allouée sous la forme d'un tableau mono dimensionnel afin de disposer d'une structure contiguë en mémoire.

L'objet Graph contient donc les paramètres suivants :

- Le nombre de cartes total.
- Un vecteur de Cards représentant la totalité des cartes.
- Un vecteur d'int8 représentant les valeurs des liens entre les cartes.
- Une bimap d'identifiants de cartes : permettant de faire le lien entre les identifiants des cartes et les identifiants associés sur la matrice d'adjacence. La bimap offre un accès en $O(1)$ ainsi qu'un tri des identifiants dans les deux sens

Pour réduire encore plus la taille de la matrice, nous aurions pu utiliser une matrice triangulaire étant donné que les informations sont dupliquées. Mais pour faire ceci nous aurions dû changer également notre implémentation.

II. Evaluation du poids des arêtes

Lien entre les couleurs de carte : nous avons établi une formule, basée sur le nombre de couleurs communes et faisant ainsi la différence entre deux cartes strictement identiques et celles possédant uniquement certaines couleurs communes. Cette formule est la suivante :

$$colorValue = nbColorsTotal - (MAX(nbColors1, nbColors2) - nbCommuns)$$

nbColorsTotal correspond au nombre de couleurs différentes sur la totalité des cartes, c'est à dire B, W, U, R, U, soit 5.

nbCommuns correspond au nombre de couleurs communes entre les deux cartes.

nbColors1 et **nbColors2** correspondent aux nombres de couleurs sur les deux cartes.

Lien entre les éditions et blocs de carte : Ces deux paramètres étant liées (une édition appartient à un bloc), une seule valeur les définit de la façon suivante :

$$editionValue = 2 \text{ s'il s'agit de la même édition}$$

$$editionValue = 1 \text{ s'il s'agit du même bloc}$$

$$editionValue = 0 \text{ rien en commun}$$

Lien entre les types, les sous types et les capacités : Ces trois valeurs sont calculées de la même façon :

$$value = \text{nombre de points communs}$$

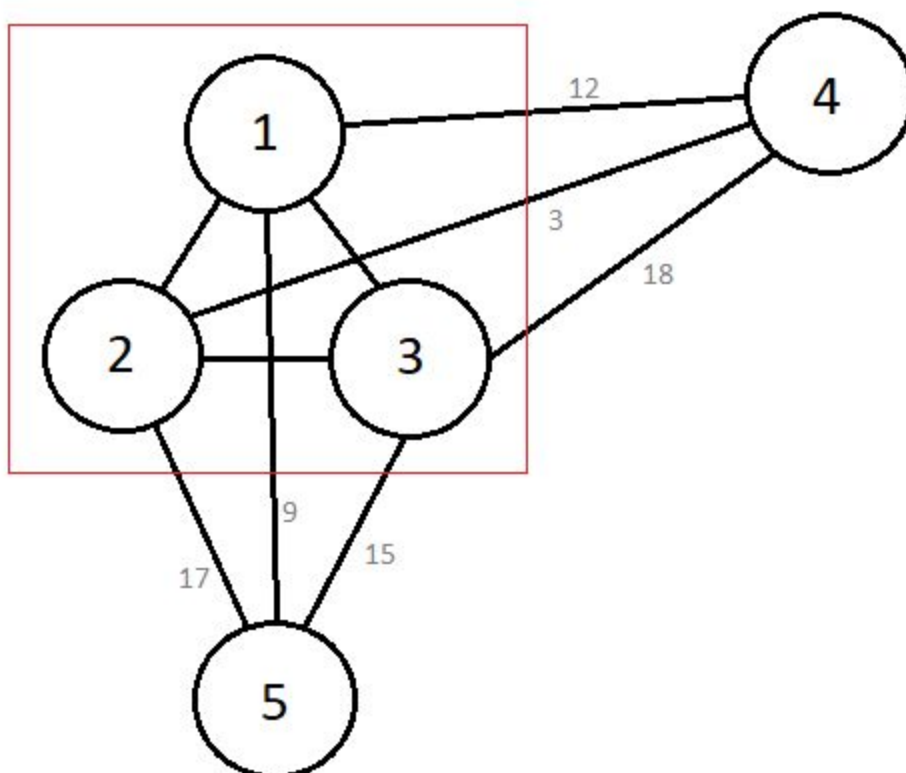
Total entre deux cartes : Le graphe généré est complet et construit à partir des liens dont les calculs ont été présentés ci-dessus. Pour chaque paramètre, un coefficient symbolisant son importance sert de multiplicateur pour le calcul du total. On a alors la formule suivante :

$$total = val_1 * coeff_1 + val_2 * coeff_2 + etc.$$

Dans le calcul final, les types et les capacités ont été multipliés par un coefficient de 0. En effet, il est primordial d'intégrer plusieurs types et capacités pour garantir l'équilibre du deck final.

III. Recherche des meilleurs associations

Une fois les valeurs des liens entre les cartes calculés selon les calculs présentés précédemment, nous avons implémenté trois algorithmes différents pour la recherche des meilleures propositions de cartes.



Nous utiliserons ce graph comme exemple pour expliquer les algorithmes. Ici, les cartes 1, 2 et 3 ont été sélectionnées. Les cartes 4 et 5 sont testées pour leur intégration ou non au deck.

a. Voisins les plus proches

Cet algorithme se déroule en deux étapes. La première consiste à sélectionner N cartes ayant le meilleur lien avec chaque carte déjà sélectionnée. Parmi ces liens, tout ceux ayant la même carte de destination sont additionnés. Les meilleurs résultats de ces additions sont les cartes sélectionnées.

Ainsi, selon l'exemple, pour $N = 2$, on sélectionne les liens entre les sommets [1;4], [3;4], [2;5] et [3;5]. On obtient alors les valeurs 15 et 17 pour le sommet 5 et 12 et 18 pour le sommet 4.

Ainsi, le lien entre l'ensemble des cartes sélectionnées et 4 est de 30 et de 32 pour 5. On proposera alors la carte 5 avant la carte 4.

b. Voisins les plus lourds

Cet algorithme est le plus simple des trois. En effet il s'agit uniquement de sélectionner les cartes qui ont le meilleur lien entre tous. Ainsi, dans l'exemple, la carte 4 passe avant la carte 5 puisqu'elle dispose du lien le plus fort sur l'ensemble du graph, c'est à dire 18 depuis 3.

c. Meilleures distances

C'est le principal algorithme que nous utilisons. Selon cet algorithme, toutes les liaisons sont prises en compte, même les plus petites. Ainsi, on additionne pour chaque sommet l'ensemble des poids des liens depuis les cartes sélectionnées.

Selon l'exemple on obtient alors des poids de


$$12 + 18 + 3 = 33 \text{ pour } 4$$

$$17 + 15 + 9 = 41 \text{ pour } 5$$

On proposera alors la carte 5 avant la carte 4.

En ce qui concerne son implémentation, nous commençons par transformer les identifiants des cartes que nous lui transmettons en identifiants dans la matrice grâce à la bimap d'identifiants. Nous utilisons ensuite une map de multimap avec pour clef un entier qui sera l'identifiant de la carte.

Nous insérons donc dans la map toutes les cartes déjà présentes dans le deck, auxquelles nous associons une multimap qui sera composé des voisins de chaque carte avec le poids associé. Pourquoi une multimap ? Cette structure de données offre encore une fois un accès en $O(1)$ et permet d'avoir plusieurs fois les mêmes valeurs et en plus triées. Nous aurons donc une multimap de poids triés par



ordre croissant et à chaque poids est associé l'identifiant du voisin. La partie où on influence l'algorithme est développée plus bas. Ensuite pour chaque carte (qui sont les voisins) nous additionnons tous leurs poids. Ceci est ensuite transmis au programme pour qu'il retienne les voisins avec les poids les plus élevés pour faire des propositions à l'utilisateur.

IV. Evaluation du deck

La dernière partie du projet était l'évaluation du deck. Celle-ci devait nous permettre de récupérer des statistiques sur notre deck (que nous affichons dans l'interface graphique) et de pouvoir influencer l'algorithme en fonction de cette évaluation. La première étape de cette évaluation se trouve dans la fonction `generateStats()` qui est appelée à l'ouverture d'un deck ainsi qu'à chaque ajout de carte au deck. Nous calculons donc dedans le nombre de carte de chaque type et de chaque couleur et faisons la moyenne de leur coût en mana.

A savoir que pour le critère couleur nous partons sur le principe d'une répartition équilibrée du nombre de cartes de chaque couleur choisies par l'utilisateur. Si ce dernier choisit deux couleurs, nous partons sur une répartition 50% / 50% de chaque couleur tandis que s'il en choisit trois ce sera une répartition 33% / 33% / 33%.

Le coût en mana est la seule valeur qui est seulement indicative. En effet, nous trouvons qu'il n'était pas pertinent de proposer des cartes en fonction de ce critère car une fois à l'équilibre l'algorithme n'aurait proposé que des cartes du coût en mana souhaité. Par exemple, si nous avions souhaité un coût en mana de 2 et que nous avions eu une moyenne de 3 en début de deck, l'algorithme aurait d'abord proposé des cartes avec des coûts de 1 ou 2, puis une fois la moyenne à 2 il n'aurait plus proposé que des cartes de 2. Nous préférons laisser à l'utilisateur le choix de cette répartition au fur et à mesure qu'il constitue son deck. Ceci aurait pu faire également l'objet d'une liste de paramètres supplémentaires dans le fichier de configuration avec la répartition détaillée du nombre de carte pour chaque coût en mana que l'utilisateur souhaitait.

Une fois les statistiques calculées, nous allons envoyer des informations à l'algorithme de propositions pour qu'il soit influencé. Ces informations sont en fait la différence entre le pourcentage souhaité et le pourcentage actuel du deck. Cette différence est divisée par 2 pour le critère sort/créature et 3 pour la couleur. Si cette différence est négative elle est ramenée à 0. Ces valeurs ont été choisies de façon arbitraire après plusieurs tests sur l'évaluation effectués. Il n'y a donc pas eu de formule trouvée pour le calcul d'une évaluation en fonction des coefficients renseignés par l'utilisateur.

Nous avons cependant pensé ajouter un coefficient d'influence mais celui-ci n'est pas utilisé pour les mêmes raisons c'est qu'il fallait trouver une formule cohérente.

Ces valeurs sont ensuite utilisées par l'algorithme de proposition. En ce qui concerne l'influence celui-ci ne fait qu'ajouter les valeurs transmises (différence de pourcentage divisé par 2 ou 3) au poids des arêtes dont les cartes correspondent aux critères.



Exemple :

Nous avons actuellement 60% de cartes créatures et 40% de cartes sorts dans le deck et nous souhaitons une répartition 50/50. La différence est donc de - 10, donc ramenée à 0 pour les créatures et 10 divisé par 2 pour les sorts. Nous transmettons donc 0 et 5 à l'algorithme. Tous les liens entre des cartes actuellement présentes dans le deck et des cartes de type sort se verra "renforcé" de 5.

Ceci est décuplé par l'algorithme que nous utilisons car il additionne les poids des arêtes pour chaque carte présente dans le deck. Donc si nous avons un deck actuellement composée de 10 cartes, une carte de type sort aura un poids final dans les propositions augmenté de 50.


La dernière étape de notre travail aurait été de trouver une formule fonctionnant à chaque fois en fonction des coefficients renseignés par l'utilisateur.

V. Interface graphique

Comme nous en avons parlé plus haut dans la partie choix des technologies, l'implémentation d'une interface graphique n'était pas une partie majeure du projet mais avait tout de même son importance et devait répondre à certains critères, le premier étant le temps de prise en main. Outre cela, il fallait que notre interface soit interactive et propose divers fonctionnalités, qui sont les suivantes :

- L'ouverture d'un fichier de configuration qui comporte :
 - Les différents coefficients de chaque critère (cela évite de recompiler le code à chaque test que nous faisons avec des paramètres différents)
 - Le numéro de l'algorithme que nous utilisons
 - La part en pourcentage des cartes sorts et créatures que nous voulons
 - La moyenne de mana désirée
 - Les couleurs que nous souhaitons avoir dans notre deck
- Générer le graphe en fonction des coefficients renseignés dans le fichier de configuration
- L'ouverture d'un deck, en soit un fichier texte qui contient les multiverseid des cartes qui composeront le début du deck et sur lesquels l'algorithme commencera à proposer d'autres cartes
- La sauvegarde de notre deck
- Le réinitialisation du deck ainsi que du graphe pour pouvoir recommencer sans avoir à fermer puis rouvrir le programme
- Un cadre qui contient les informations de configuration renseignées dans le fichier de configuration, soit le nombre de couleurs, la moyenne de mana et les pourcentages de cartes créatures et sorts
- Les statistiques du deck :
 - Le nombre de cartes actuellement présentes dans le deck
 - Le nombre de cartes de chaque couleur avec leurs pourcentages
 - Le nombre de cartes de chaque type créature et sort avec leurs pourcentages
 - La moyenne du coût en mana des cartes

L'implémentation de l'interface graphique a pu se faire efficacement et rapidement grâce à Qt qui propose un outils pour "glisser-déplacer" les éléments graphiques que nous souhaitons ajouter à notre interface, du moins pour la partie fixe de l'interface. Cependant il y a une partie variable qui se compose du nombre de propositions et de cartes que nous aurons dans notre deck. Car ces paramètres sont modifiables dans le programme par les `#define NB_PROPOSALS` et `#define DECKSIZE`. Nous ne pouvions pas mettre ces paramètres dans le fichier de configuration car les éléments graphiques sont chargés au démarrage de l'application, avant-même que l'on puisse charger une configuration. Avec du



recul nous aurions pu donner plus flexibilité à notre programme en faisant en sorte de pouvoir redimensionner le nombre de propositions et la taille du deck pendant l'exécution du programme, mais ce n'était pas une fonctionnalité prioritaire.

Un point intéressant tout de même de l'utilisation de Qt est la gestion de plusieurs boutons avec une seule fonction callback. En effet, lorsque nous créons un bouton sous Qt nous lui associons un signal, qui peut être un clique, un relâchement de clique ou autre, ainsi qu'un slot qui est une fonction callback associée au signal. Il faut ensuite tout connecter : le bouton, le signal, le conteneur du bouton, la fonction. Pour que quand un signal est déclenché sur un bouton, la fonction associée soit exécutée.

Cela a été simple pour les boutons d'ouverture de deck, de configuration de réinitialisation etc mais nous avons des besoins plus complexes pour l'ajout d'une carte au deck. Il fallait donner à chaque bouton l'identifiant de la carte à laquelle il était associé. Nous avons donc surchargé la classe QPushButton pour lui ajouter un identifiant. Ensuite Il fallait associer une fonction à plusieurs boutons avec en plus le passage d'un paramètre (l'identifiant de la carte). Dans ce cas nous avons utilisé un QSignalMapper pour associer tous les boutons et leur signal à la map, puis associer la map à la fonction callback pour l'ajout d'une carte au deck. Comme ceci lorsqu'un signal sur un bouton de la map est déclenché, la fonction est exécutée, et ceci nous permet également de passer en paramètre l'identifiant !

Pour aller plus loin

I. Déséquilibre des couleurs

Suite aux premières simulations, nous avons remarqué un nombre important de cartes bicolores. Après étude de nos algorithmes nous en avons trouvé la raison. En effet, pour 5 cartes noires et 5 cartes blanches sélectionnées, les liens en fonction des couleurs sont plus importants avec les cartes bicolores noires/blanches qu'avec les cartes uniquement noires ou blanches.

Il ne s'agit cependant pas d'un effet souhaitable pour l'équilibre du deck. Pour parer à ce résultat, il est possible de changer la façon de calculer les liens. En effet, en divisant par le nombre de carte de couleur identique, on équilibre les résultats. Cette division se fait de la façon suivante :

Soit un début de deck constitué de 5 cartes blanches, 4 cartes noires et 1 carte bicolore noire/blanche. Soit des cartes à tester 1, 2, 3, respectivement blanche, noire et bicolore. On a les évaluations suivantes :

$$\text{Deck} \rightarrow A = 5 * 5 + 1 * 4 = 29$$

$$\text{Deck} \rightarrow B = 4 * 5 + 1 * 4 = 24$$

$$\text{Deck} \rightarrow C = 5 * 4 + 4 * 4 + 1 * 5 = 41$$

On peut alors diviser par le nombre de cartes pour obtenir les résultats suivants :

$$\text{Deck} \rightarrow A = 29 / 6 = 5$$

$$\text{Deck} \rightarrow B = 24 / 5 = 6$$

$$\text{Deck} \rightarrow C = 41 / 10 = 4$$

On remarque aussi que ce calcul permet d'équilibrer les quotas de carte par couleur. Il s'agit cependant d'une solution nécessitant une refonte complète de notre façon de calculer les liens. Les poids des arêtes étant calculés à la création du graph, l'évaluation des couleurs seules n'est pas stockée jusqu'à construction des propositions.

II. Autres paramètres d'évaluation

Les données disponibles étant limitées, l'évaluation des liens ne se fait que sur des paramètres les plus basiques. Le texte principal de chaque carte n'est pas interprété ainsi que les paramètres plus subjectifs tels que la popularité. En effet celle-ci est basée sur les ensembles de cartes apparaissant lors de compétitions et donc éprouvées par les meilleurs.

Il en va de même pour le coût en mana dont la moyenne dans un deck dépend de la façon de jouer. Si certains préfèrent la vitesse du faible coût, d'autre patienteront pour frapper fort plus tard.

Ce sont autant de paramètres qui ajouteraient des points de vue plus variés à l'évaluation et permettraient ainsi d'atteindre un meilleur équilibre pour le deck final.