Rapport APPROX

Delphine MORVAN, Antoine THOMAS, Louis LE QUELLEC, Pierre MAUNIER

22 juin 2024

Table des matières

1 Rappel du sujet

Dans le jeu vidéo *Teamfight Tactics* (TFT), les joueurs doivent placer des unités qui combattent pour eux. Chaque unité a une origine et une classe (ou deux), et rassembler plusieurs unités partageant l'un de ces traits confère des bonus. Dans la dernière version du jeu, une nouvelle classe appelée *Exalté* a été introduite. La particularité de cette classe est que ses membres sont déterminés de manière aléatoire à chaque partie, selon les règles suivantes :

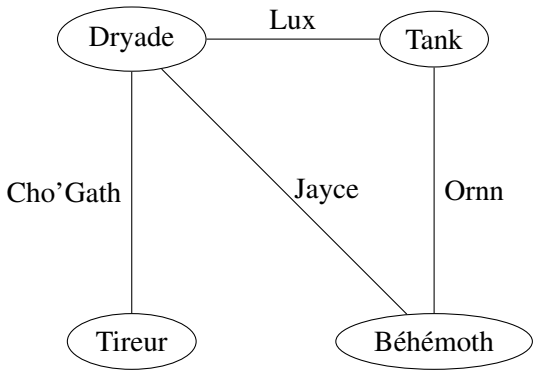
— Deux unités partageant une classe ou une origine ne peuvent pas être *Exaltées* simultanément.

— Une unité ayant déjà 3 traits ne peut pas être *Exaltée*.

L'objectif est de déterminer combien d'unités peuvent être *Exaltées* simultanément au maximum. Cependant, comme il s'agit d'un problème complexe, il est aussi nécessaire de trouver un algorithme d'approximation pour approcher ce nombre.

2 Modélisation

Afin de faire notre modélisation pour la version optimisée, nous avons choisi d'utiliser un graphe, qui aurait pour noeuds les différents traits possibles et comme arêtes les champions. Nous avons donc, par exemple, en prenant uniquement les 4 champions : *Cho'Gath*, *Lux*, *Jayce* et *Ornn*, le graphe suivant (non-représentatif d'un cas réel) :



Nous avons donc, pour chaque arête, une instance de la classe *Perso* qui possède le nom du champion et une liste de ses traits. Il existe aussi un pretty_print pour avoir les Perso sous la forme "Perso : nom_du_perso, traits : attribut1 attribut2". La création de ces instances est effectuée lors du parsing de notre input. Nous pouvons décrire cette fonction comme ceci : nous prenons ligne par ligne et nous essayons de voir combien de traits possède le personnage. S'il en possède déjà 3, ça ne sert à rien de le rajouter puisqu'il ne peut pas être exalté. Sinon on crée un objet de la classe Perso avec son nom et ses attributs que l'on ajoute dans une liste qui sera ensuite donnée à nos 2 algorithmes.

La version non-optimisé prendra la liste de personnage que renvoie notre fonction, et la version optimisée, à l'aide de cette même liste, va créer un graphe, comme celui ci-dessus.

3 Algorithmes

3.1 Brute force

Pour avoir le nombre maximal exact de champions exaltés, il faut générer toute les possibilités de sélection. Nous avions d'abord pensé à générer tous les sous-ensembles de champions possibles, puis trouver le plus grand

où ils n'avaient pas de trait en commun. * Cette solution était beaucoup plus longue et nous avons donc choisi de l'implémenter comme ci-dessous. Pour cela on parcours l'entrée et pour chaque champion de la liste on crée récursivement une sélection où le champion est et une autre où il n'est pas. Comme l'algorithme réalise deux appels récursifs pour chaque champion, il y aura 2^n sélections générées. La complexité de l'algorithme est donc :

$$O(2^n)$$

Avec n la taille de l'entrée.

3.1.1 Pseudo-code

Algorithm 1 Brute_force_max_champions

Require: *liste_unites* liste contenant les différents champions
Require: *liste_traits* liste contenant tous les traits déjà utilisé (vide au départ)
Require: *index* indice du champion actuel, d'abord à 0
  if *index* = longueur de *liste_unites* **then**
    return 0
  else
    if *liste_unites*[*index*] compatible avec *liste_traits* **then**
      nouveaux_traits = *liste_traits* ∪ traits de *liste_unites*[*index*]
      return max(1 + Brute_force_max_champions(*liste_unites*, *nouveaux_traits*, *index* + 1), Brute_force_max_champions(*liste_unites*, *liste_traits*, *index* + 1))
    else
      return Brute_force_max_champion(*list_unites*, *liste_traits*, *index* + 1)
    end if
  end if

3.2 Version optimisée

Pour la version optimisée nous nous sommes vite rendu compte qu'un des minimums requis était d'essayer de prioriser l'utilisation des traits les moins représentés parmi les champions. Pour cela une simple recherche du noeud au degré le plus faible était notre point de départ, et dans l'optique de garder la priorité définie, nous continuons avec le voisin de degré le plus faible. Une fois les traits sélectionnés le champion est sauvegarde et les noeuds ne pouvant être réutilisés, ils sont supprimés. Enfin nous répétons ces étapes jusqu'à l'épuisement des arcs du graphes : c'est-à-dire des champions sélectionables.
Voilà le pseudo-code associé :

Au vu des boucles effectuées et de l'étude de leur complexité, nous avons réalisé que le pire cas possible était que les champions couvrent toutes les combinaisons de traits possibles. On pose n le nombre de traits retenus depuis l'entrée. Dans tous les cas, ayant n traits et en en retirant 2 à chaque recherche, trouver un premier noeud est une recherche de longueur $\frac{n}{2}$ dans les noeuds du graphe. Dans le pire cas, la sélection du voisin parcours elle aussi la totalité des noeuds moins le premier sélectionné soit $\frac{n}{2} - 1$. Heureusement pour nous ces recherches s'additionnent et ne se multiplient pas, la complexité à l'intérieur de la boucle est donc de $n - 1$. Le nombre d'arêtes initiales est de $n^2 - n$ et n en sont retirées par itération, il y a donc $n - 1$ itérations de recherche d'arête. Cette fois-ci cette complexité est multipliée, ce qui nous donne une complexité de $(n-1)^2$ soit

$$O(n^2)$$

Algorithm 2 Approx_max_champions

Require: G un graphe non-orienté (voir exemple dans la partie modélisation)
  result ← liste vide
  while *edges* in G **do**
    node ← le noeud qui possède le plus petit degré dans G
    other ← le noeud voisin de *node* avec le plus petit degré
    result ← le champion contenu dans le l'arête entre *node* et *other*
    remove *node* from G
    remove *other* from G
  end while
  return *result*

Le meilleur cas quant à lui est celui dans lequel chaque champion à 2 traits uniques, la recherche d'un deuxième noeud ne passera donc que dans une liste de longueur 1. La boucle a donc une complexité de $\frac{n}{2}+1$. On sait aussi qu'il y aura $\frac{n}{2}$ arêtes et qu'on en retire une seule par itération ce qui nous donne $\frac{n}{2}$ itérations de recherche d'arête. La complexité est donc au mieux de $\frac{n^2}{4}+\frac{n}{2}$ soit

$$O(n^2)$$

4 Limite et conclusion

4.1 Tests et résultats

4.1.1 Tests

Dans le cadre de nos tests, nous avons créé un script pour générer des données aléatoirement. Le script de génération de données a été conçu pour créer un ensemble de champions fictifs avec des traits aléatoires. Chaque champion est associé à une origine et une classe (ou deux) tirées au hasard parmi des listes prédéfinies (il existe aussi un script où on peut choisir la taille de ces listes). Le script prend en entrée le nombre de champions à générer et produit une sortie formatée qui détaille chaque champion et ses traits. Voici une explication détaillée du fonctionnement du script :

- Listes de Traits :
  Le script commence par définir deux listes de traits possibles : origins et classes. Les origines incluent des éléments comme "Fated", "Ghostly", et "Mythic", tandis que les classes incluent des éléments comme "Arcanist", "Sniper", et "Behemoth".
- Entrée Utilisateur :
  Le script demande à l'utilisateur d'entrer le nombre de champions à générer via input(). Cette entrée est stockée dans num_champions.
- Génération des Champions :
  Pour chaque itération : Un nom unique est généré pour le champion (e.g., Champ1, Champ2, etc.). Un nombre aléatoire de traits (entre 2 et 3) est attribué au champion. Les traits sont choisis de manière aléatoire, en s'assurant qu'il y ait au moins une origine et une classe parmi les traits sélectionnés. Un ensemble (set) est utilisé pour stocker les traits afin de garantir qu'il n'y ait pas de doublons.
- Formatage des Résultats :
  Après avoir généré tous les champions, le script formate les données pour chaque champion dans une chaîne de caractères selon la grammaire spécifiée : nom_champion trait1 trait2 ....

— Example de données générées :
Champ1 Fated Arcanist
Champ2 Ghostly Duelist
Champ3 Mythic Reaper Bruiser

4.1.2 Résultats

Nous avons généré, grâce à ce script, un input de taille 100, puis nous avons lancé nos deux algorithmes. Nous avons donc pour le brute force :

```
For the brute Force algorithm
Nombre de perso: 12

real    0m32.606s
user    0m32.555s
sys     0m0.051s
```

FIGURE 1 – Résultat de l'algorithme brute force

Puis pour la version optimisée :

```
First implementation of a new algorithm
Perso: Champ29, traits: ['Dryad', 'Heavenly']
Perso: Champ89, traits: ['SpiritWalker', 'Reaper']
Perso: Champ7, traits: ['Duelist', 'Arcanist']
Perso: Champ62, traits: ['Fortune', 'Fated']
Perso: Champ55, traits: ['Bruiser', 'Sage']
Perso: Champ22, traits: ['Trickshot', 'Umbral']
Perso: Champ95, traits: ['Dragonlord', 'Ghostly']
Perso: Champ50, traits: ['Warden', 'Behemoth']
Perso: Champ19, traits: ['Lovers', 'Altruist']
Perso: Champ56, traits: ['Artist', 'Invoker']
Perso: Champ24, traits: ['Sniper', 'Storyweaver']
Perso: Champ53, traits: ['Porcelain', 'Mythic']
Nombre de personnage: 12

real    0m0.247s
user    0m0.197s
sys     0m0.050s
```

FIGURE 2 – Résultat de l'algorithme optimisé

Nous avons testé notre algo optimisé avec 1500 champions, 10000 classes et 10000 traits et avons obtenu, après plusieurs générations d'input, au pire :

```
Nombre de personnage: 693

real    0m0.712s
user    0m0.688s
sys     0m0.020s
```

FIGURE 3 – Résultat de l'algorithme optimisé, deuxième test

Ce qui montre donc que notre algorithme est viable pour un grand nombre de champions et de liens entre ces champions.

4.2 Conclusion

Malgré une réflexion sur notre brute force, on remarque une très grosse différence de temps entre les deux algorithmes. Cependant, pour tous les tests que nous avons pu faire dû aux limitations du brute force, nous avons trouvé le même résultat entre les deux algorithmes à chaque fois. De plus, en comparant nos résultats obtenus avec d'autres groupes pour les tests plus conséquents, nous obtenions les mêmes résultats de nombre de champions sélectionnés. Nous sommes donc plutôt satisfait du résultat.