Analyse théorique de l'algorithme génétique pour le problème Magic: The Gathering

Frédéric Carpentier-Lapointe et Imane Abdillahi

4 décembre 2019

Quelques notes importantes pour la suite :

- Nombre de cartes par paquet : C
- Nombre de paquets : S
- Population totale : P
- Nombre de parties d'un tournoi : R
- Nombre de mutations : M

1 Génération de la population initiale

```
fonction generateInitialPopulation(): Population
    Pour POPULATION faire
        population.ajouter(generateInitialInd)

fonction generateInitialIndividu(): Individu
    Pour i = 0 a nombre_paquet faire
        Pour j = 0 a carte_paquet faire
            paquet.ajouter(carte_aleatoire)

        scorePaquet <- calculeScorePaquet

fonction calculeScorePaquet()
    Pour x = 0 a carte_paquet faire
        valeur_carte <- valeur_carte + x
        Pour y = x a carte_paquet faire
            synergy <- synergy + synergy[x][y]
```

Tout d'abord, nous avons notre première fonction qui va initialiser notre population à l'aide d'individus aléatoires. Le nombre d'individus dépend d'une constante d'entrée de l'algoritme *POPULATION*. Par la suite, on entre dans

la fonction *generateInitialIndividu* qui contient deux boucles : le nombre de paquets à construire ainsi qu'une boucle qui va trouver le nombre de cartes de façon aléatoire pour remplir le paquet courant. À chaque fois que l'on rempli un paquet, on trouve son score avec la fonction *calculeScorePaquet* qui est deux boucles en fonction du nombre de cartes dans le paquet. On aura donc une complexité de $\Theta(P * S * C^2)$

2 Sélection

```
fonction selection(population, ind1, ind2):
    // Premier vainqueur
    index_ind1 <- indexAleatoire
    index_ind2 <- indexAleatoire
    index_ind1 <- tournoi(ind1,ind2)
    ind1 = population.individus[index_ind1]
    population.delete(ind1)

    // Deuxieme vainqueur
    index_ind1 <- indexAleatoire
    index_ind2 <- indexAleatoire
    ind2 <- tournoi(ind1, ind2)
    ind2 = population.individus[index_ind1]
    population.delete(ind1)

fonction tournoi(individu1, individu2):
    gagnant <- -1
    Pour i = 0 a PARTIES faire
        solInd1 = calculeSolutionIndividu(individu1)
        solInd2 = calculeSolutionIndividu(individu2)
        si (solInd1 > solInd2) alors
            gagnant = individu1
        sinon
            gagnant = individu2

fonction calculeSolutionIndividu(individu):
    min_score_paquet <- calculeScorePaquet(individu.paquets[0])
    Pour i = 0 a nombre_paquet faire
        score_temporaire = calculeScorePaquet(individu.paquets[i])
        si score_temporaire < min_score_paquet alors
            min_score_paquet = score_temporaire
    retourne min_score_paquet
```

Pour la sélection, nous sélectionnons tout d'abord deux individus aléatoires pour qu'ils s'affrontent. On appelle alors la fonction *tournoi* qui est une boucle effectuée R fois. Dans cette boucle, on calcule la solution des deux individus à

chaque itération. La méthode *calculeSolutionIndividu* est également une boucle sur le nombre de paquets S pour l'individu en question. Pour chaque paquet, on calcule son score avec la méthode utilisée dans la section précédente. Ainsi, on peut dire que cette partie de l'algorithme a une complexité de $\Theta(R * S^2 * C^2)$. Comme R est une constante connue, on peut l'ignorer.

3 Croissement

```
fonction croissement(ind1, ind2): Individu
    trierPaquet(ind1)
    trierPaquet(ind2)
    genesConserves <- choixNombreGenesAleatoire
    Pour i = 0 a genesConserves faire
        si (ind1.packs[i] > ind2.packs[i] alors
            nouvelIndividu[i] <- ind1.packs[i]
        sinon
            nouvelIndividu[i] <- ind2.packs[i]

    generePaquetRestantAleatoire(nouvelIndividu)
    retourner nouvelIndividu

fonction generePaquetRestantAleatoire(individiu)
    tant que individu.paquets.taille < nombre_paquet
        Pour i = 0 a carte_paquet faire
            paquet.ajouter(carteAleatoire)
            calculeScorePaquet(paquet)
```

Pour le croissement, on commence par trier aléatoirement nos deux individus selon le score des paquets. Pour les deux tris, on aura une complexité de $\Theta(S * log(S))$. Par la suite, on sélectionne un nombre aléatoire de gènes à conserver. On effectura alors la création du nouvel individu dans une boucle pour ce nombre fixe de gènes. Pour le nombre restants de paquets à construire, on génèrera aléatoirement des paquets tout en s'assurant de ne pas avoir de doublons. La génération de paquets est également une boucle while pour le reste des paquets à construire. On aura donc une complexité de $\Theta(S * log(S) * G * (S - G)C$. La variable G est une constante qui repreésente le nombre de gènes conservés.

4 Mutation

```
fonction mutation(population):
    Pour i = 0 a NbMutations faire
        individu = choisirAleatoirementIndividu
        mutationIndividu(individu)
```

```
fonction mutationIndividu(individu):
    paquet1, paquet2 = choisirDeuxPaquetsAleatoires()
    ancienPaquet1Score = calculerScorePaquet(paquet1)
    ancienPaquet2Score = calculerScorePaquet(paquet2)
    tant que !trouve et recherche < MaxSearch
        Pour i = 0 a nbCartesAleatoires faire
            echangerCarteEntrePaquet(i)

        nouveauPaquet1Score = calculerScorePaquet(paquet1)
        nouveauPaquet2Score = calculerScorePaquet(paquet2)

        si nouveauPaquet2Score > ancienPaquet1Score et nouveauPaquet2Score > ancienPaquet2Sc
            trouve = 1

        recherche++
```

Pour la mutaton, on effectura une boucle de M mutations sur notre population, et ce, de façon aléatoire. Chaque mutation est une boucle while jusqu'à ce que l'on trouve une mutation qui bat la performance actuelle des deux paquets choisis aléatoirement. La fonction *echangerCarteEntrePaquet* n'est qu'un accès et une assignation à une valeur d'un tableau. On aura également un nombre d'itérations limite (MaxSearch) pour arrêter notre recherche d'une meilleure mutation. Ainsi, dans cette partie, on aura une complexité de $\Theta(M * (2 * C)$. Puique le nombre de mutations M est une constante connue et déterminée, on peut l'ignorer dans le calcul de complexité, elle sera considéré comme une constante multiplicative du problème.

5 Sélection des meilleures solutions à conserver pour la prochaine génération

```
fonction selectionMeilleursIndividus(population): Population

    trierParScore(population)
    Tant que survivant < NbSurvivantsAConserver
        nouvellePopulation.ajouter(population.individus[survivant])
        survivant = survivant + 1
    retourner nouvellePopulation
```

Pour choisir les meilleurs individus à conserver, on commence par trier la population en ordre décroissant selon leur score (utilisant la fonction calculerScoreIndividu). Puisque le trie utilise la fonction calculerScoreIndividu, on obtiendra une complexité de $\Theta(S^2 * log(S))$. On remplira ensuite notre nouvelle population avec le nombre de survivants à conserver pour la prochaine itération de l'algorithme génétique. Cette partie a une complexité totale de

$\Theta(T * S^2 * log(S))$. T étant le nombre d'individus à conserver pour chaque itération de l'algorithme génétique. Puisque c'est une constante connue, on peut dire que la complexité sera de $\Theta(S^2 * log(S))$.