Algorithmique avancée : Projet

Hamza Bouihi, Mamadou Thiongane

7 mai 2024

Table des matières

1 Algorithmes implémentés

1.1 Algorithme glouton

1.1.1 Principe de l'algorithme

Le but d'un algorithme glouton est de choisir à chaque étape l'option **localement optimale**. Ici, cela revient à maximiser le nombre de blobs alliés à chaque coup joué. Pour cela, le principe de l'algorithme est le suivant.

Soit $\mathcal{S}$ la fonction qui, pour un mouvement m donné, calcul le nombre de blobs alliés une fois le mouvement m effectué et $\mathcal{M}$ l'ensemble des mouvements possibles pour le joueur.

Algorithm 1 Algorithme glouton

```
procedure GLOUTON                                   ▷ Au tour i ∈ ℕ
    s ← − MAX_INT
    mv ← null
    for chaque mouvement m ∈ M do
        tmp ← S(m)
        if tmp > s then
            s ← tmp
            mv ← m
    return mv
```

1.1.2 Implémentation

En pratique, nous avons quelque peu modifié l'algorithme lors de l'implémentation pour l'optimiser :

— Au lieu de compter le nombre de blobs alliés, il suffit de compter le nombres d'ennemis possiblement récupérés à chaque mouvement et de privilégier les duplications.

— La deuxième, est que nous avons créé un offset `neighbors` permettant d'éviter de passer sur la case sur laquelle nous nous trouvons lors de la vérification des voisins (chaque optimisation est bonne à prendre lorsque le nombre d'itérations est élevé)

1.1.3 Complexité

Considérons une grille de taille $n \times n$, $n \in \mathbb{N}$. L'optimisation détaillée précédemment permet de calculer $\mathcal{S}(m)$ en temps constant. Soit $S(m) = O(1)$. De plus, récupérer la liste des mouvements possible se fait en un seul parcours de la grille, de ce fait, si on note $\mathcal{C}(n)$, la complexité du calcul d'un coup sur la grille, alors nous avons :

$$\mathcal{C}(n) = O\left(n^2\right)$$

1.2 Algorithme min-max

1.2.1 Principe de l'algorithme

Blobwar peut se représenter comme un arbre, où les nœuds représentent des mouvements. Puisque les joueurs jouent alternativement, les nœuds d'un même niveau sont alternativement contrôlés par un même joueur, la racine étant contrôlée par le joueur Max.

Pour toute feuille f, on note $\mathcal{S}(f)$ le score du joueur Max si cette feuille est atteinte et on définit par induction une extension de la fonction $\mathcal{S}$, notée $\mathcal{E}$, de la manière suivante :

— Si n est une feuille, $\mathcal{E}(n) = \mathcal{S}(n)$

— Si n est contrôlé par le joueur Max, $\mathcal{E}(n) = \max\{\mathcal{E}(x) \mid x \prec n\}$
— Si n est contrôlé par le joueur Min, $\mathcal{E}(n) = \min\{\mathcal{E}(x) \mid x \prec n\}$

Si chaque joueur joue en essayant de maximiser (resp. minimiser) la quantité $\mathcal{E}$ précédente, le score final du joueur Max à la fin de la partie sera $\mathcal{E}(r)$, où r est la racine.

Mais l'arbre étant est beaucoup trop gros pour être exploré entièrement, on réduit la profondeur de recherche dans l'arbre à une certaine valeur p.

L'algorithme 2 calcule la valeur de $\mathcal{E}(n)$ pour un mouvement donné.

Algorithm 2 Algorithme min-max

```
function MinMax(n, p)
    if p = 0 ou n est une feuille then
        return S(n)
    if n est un nœud Max then
        return max{MinMax(f, p − 1) | f ≺ n}
    else
        return min{MinMax(f, p − 1) | f ≺ n}
```

On stocke ensuite, pour chaque tour, le mouvement du joueur Max maximisant $\mathcal{E}$.

1.2.2 Complexité

Considérons une grille de taille $n \times n$, $n \in \mathbb{N}$ et une profondeur $p \in \mathbb{N}$. Sachant que la fonction `computeValidMoves` et `estimateCurrentScore` ont une complexité de $O(n^2)$ et qu'une copie de la grille est faite pour simuler chaque mouvement (nous n'avons pas pu trouver une meilleure solution), si on note $\mathcal{C}(p)$ la complexité de calcul d'un coup du joueur et m_1 la taille de la liste des mouvements valides lors du premier appel, nous avons

$$C(p) = m_1 \left(C(p-1) + n^2\right) + n^2$$

soit, en notant m le plus grand élément de $\{m_1, m_2, ..., m_p\}$,

$$C(p) = \left(\prod_{k=1}^{p} m_k\right) \cdot \left(2n^2\right) + \left(\sum_{k=0}^{p-1} m_{k+1} \cdot n^2\right) = O(2n^2 m^p) = O(m^p)$$

1.3 Algorithme alpha-beta

La complexité de l'algorithme 2 étant en $O(24^p)$, on modifie cet algorithme pour réduire l'évaluation d'un grand nombre de nœuds.

On introduit deux nouveaux paramètres α et β qui encadrent les valeurs intéressantes que peut prendre $\mathcal{E}(n)$ pour chaque nœud n, pour obtenir l'algorithme 3.

— Pour l'appel initial, il suffit d'utiliser $\alpha = -\infty$ et $\beta = +\infty$
— Pour un nœud Max n, on introduit un minorant m de la valeur $\mathcal{E}(n)$ (initialisée à $-\infty$). Cette valeur est actualisée à chaque calcul $\mathcal{E}(f)$ sur les fils de n. Si m dépasse β, on a une coupure β : la valeur $\mathcal{E}(n)$ dépassera la valeur intéressante maximale. Si m dépasse simplement α, on peut donner à α la valeur m, ce qui sera utile dans les appels récursifs sur les autres fils de n pour restreindre la plage de valeurs intéressantes.
— Symétriquement, pour un nœud Min n, on introduit un majorant M de la valeur $\mathcal{E}(n)$ (initialisée à $+\infty$). Cette valeur est actualisée à chaque calcul $\mathcal{E}(f)$ sur les fils de n. Si M descend en dessous de α, on a une coupure α. Si M descend simplement en dessous de β, on peut donner à β la valeur M.

Algorithm 3 Algorithme alpha-beta

```
function AlphaBeta(n, p, α, β)
    if p = 0 ou n est une feuille then
        return S(n)
    if n est un nœud Max then
        m ← −∞
        for chaque fils f de n do
            score ← AlphaBeta(f, p − 1, α, β)
            if score > m then
                m ← score
            if m ⩾ β then
                return m
            α ← max(α, m)
        return m
    else
        M ← +∞
        for chaque fils f de n do
            score ← AlphaBeta(f, p − 1, α, β)
            if score < M then
                M ← score
            if m ⩽ α then
                return M
            β ← min(β, M)
        return M
```

2 Évaluation de performances

2.1 Nombre maximal d'appels récursifs

Nous avons testé nos algorithmes min-max et alpha-beta avec plusieurs valeurs pour le nombre maximal d'appels récursifs sur la carte "standard" (nombre maximal de possibilités). Les valeurs maximales possibles pour ne pas avoir de *timeout* sont 4 (min-max) et 6 (alpha-beta).

2.2 Taux de victoires

Nous avons confronté nos différents algorithmes sur la carte "standard" et obtenu les résultats suivants.

	Glouton	Min-max(4)	Alpha-beta(6)
Glouton	–	0	0
Min-max(4)	100	–	0
Alpha-beta(6)	100	100	–

Table 1 – Pourcentage de victoires en fonction de la stratégie adverse

2.3 Bonus : Résultat du tournoi

Notre algorithme alpha-beta(6) nous a permis d'atteindre les quarts de finale du tournoi avec les résultats suivants :

— 5 victoires avec 10 points d'écart en moyenne
— 1 match nul
— 4 défaites avec 5 points d'écart en moyenne

3 Améliorations

Une piste d'amélioration envisagée pour rendre notre code plus performant est la suivante :

Si nous construisons le graphe $G = (V, E)$ tel que :

- Pour chaque blob allié, nous lui associons un sommet $v_i \in$V
- Nous ajoutons une arête $e_i = (u, v)$ à E si et seulement si $d(u, v) = 1$ sur la grille

Étant donné que la profondeur d'appels récursifs ne peut pas être élevée, il serait judicieux d'établir une classification des mouvements possibles de telle sorte que les mouvements prioritaires soient ceux qui tendent à augmenter la taille des composantes connexes tout en augmentant le nombre d'arêtes entre chaque sommet (de sorte à former des blocs de blobs $\rightarrow$ configuration où les blobs alliés sont le plus en sécurité) cela améliorerait grandement le taux de victoire de notre algorithme.

Enfin, paralléliser notre algorithme alpha-beta permettrait également de gagner du temps lors du calcul de notre meilleur coup et donc, d'augmenter la profondeur de calcul.