Étude de quelques algorithmes de joueurs artificiels participants à des jeux de stratégie en temps réel

Cocheril-Crèvecoeur Dimitri
13960

8 juin 2024

Plan

1. Présentation du problème
2. Création du moteur
3. Stratégies testées

StarCraft

Jeu de stratégie en temps réel :

- Jeux simultané
- Deux joueurs s'affrontent pour le contrôle d'une carte
- Gestion de ressources : minage, création d'unitées, de bases (stratégie)
- Combat entre unités (tactique)

Modèles existants

- IA de Google : *AlphaStar*
  Apprentissage supervisé puis par renforcement
- Robots ("bots") :

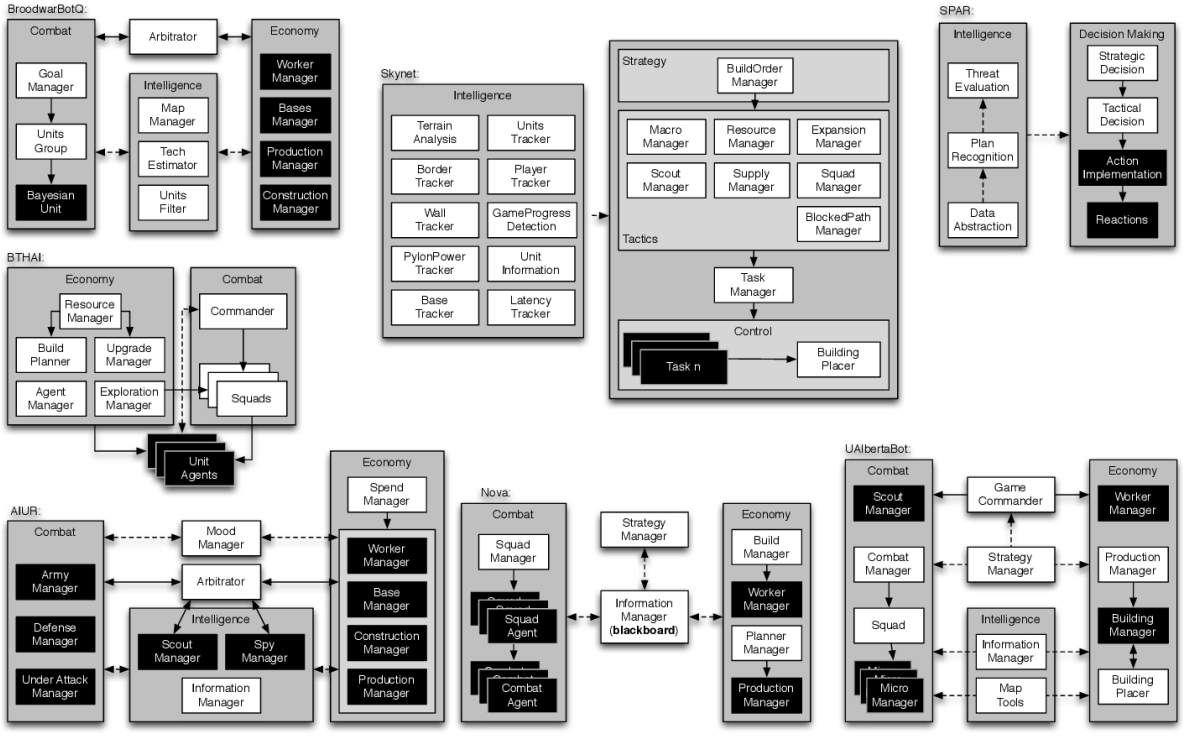


Figure – Figure extraite de la source 2

StarCraft

On se concentre ici sur la partie "combat" : Deux joueurs

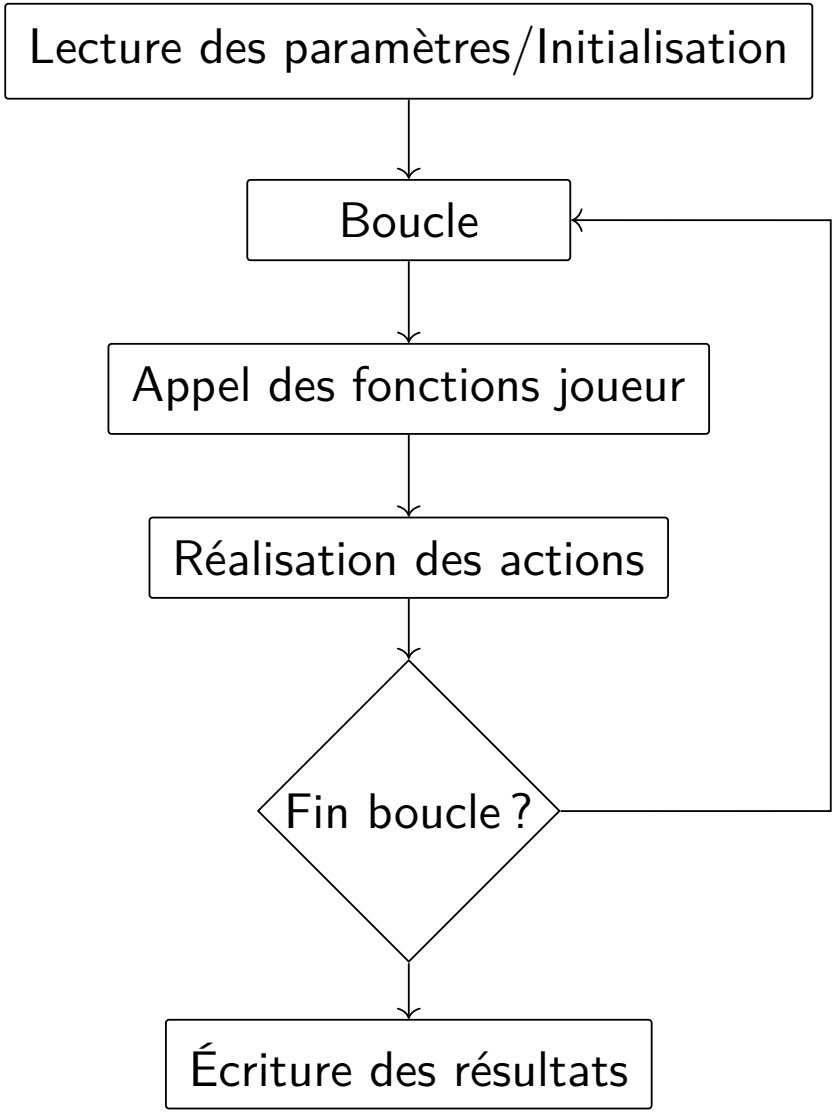
disposent d'unitées pouvant bouger et attaquer celles adverses Chaque joueur doit tuer

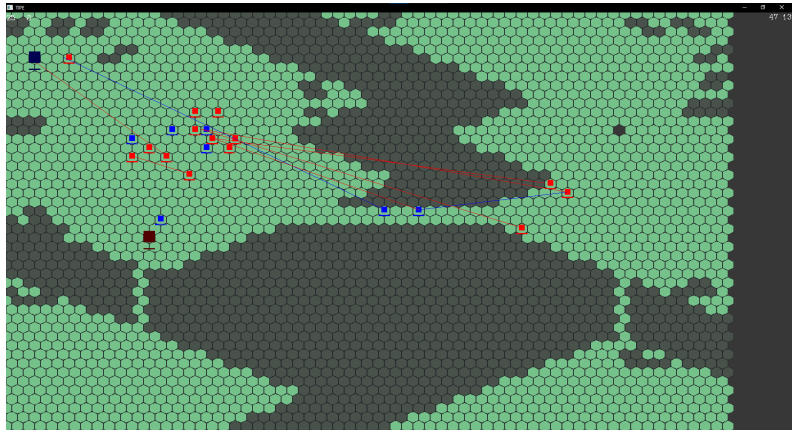
toute les unitées ennemies

Problème

1. Facteur de branchement entre 10^{50} et 10^{200}
2. Durée typique d'une partie : 25 minutes donc $25 \times 60 \times 24 = 36000$ états

Développement moteur du jeu



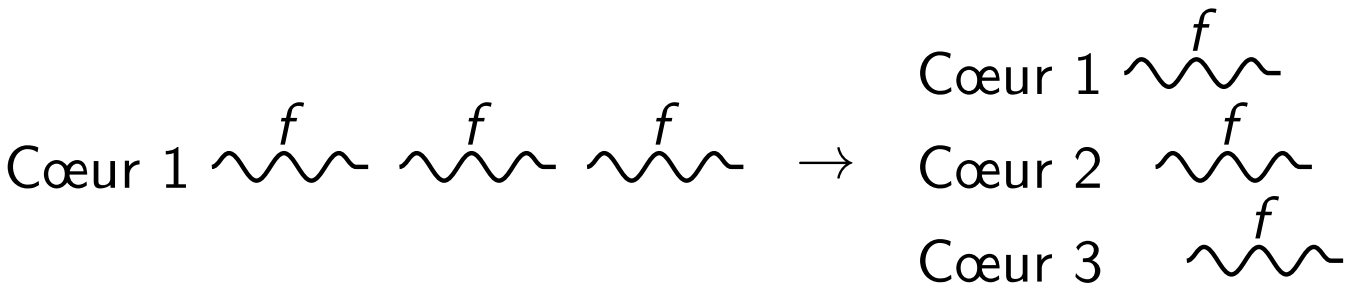


```
void game_class::play() const
{
    for (const auto p : players_)
    /*Pour chaque joueur...*/
    {
        p->moves_get(this, state_);
        /*..on choisit les actions..*/
    }
    state_->moves_make(map_);
    /*..et on effectue les actions.*/
}
```

Problème de la recherche de chemin

- Optimisation compliquée :
  On utilise un A* pondérée à 5 (trouvé empiriquement) :
  comme le A* normal mais on ajoute un poids à l'heuristique
- Fonctions parallélisables
  On alloue les espaces dynamiquement au lieu d'utiliser le tas pour éviter la concurrence

Parallélisation :



Joueur aléatoire

Joueur "témoin" : choisis une action aléatoire, et l'effectue avant d'en choisir une autre aléatoirement

On effectue 10000 combats sur une carte vide entre deux joueurs aléatoires avec 50 unités chacun :

Victoires de joueur 0 : 4889
Victoires de joueur 1 : 5106
Égalités : 5

Stratégie naïve : attaque par puissance

Toutes nos unités attaquent l'unitée ennemie avec la plus grande puissance

Résultats contre joueur aléatoire avec 50 unités :

100% de victoire pour joueur aléatoire

Joueur MCTS

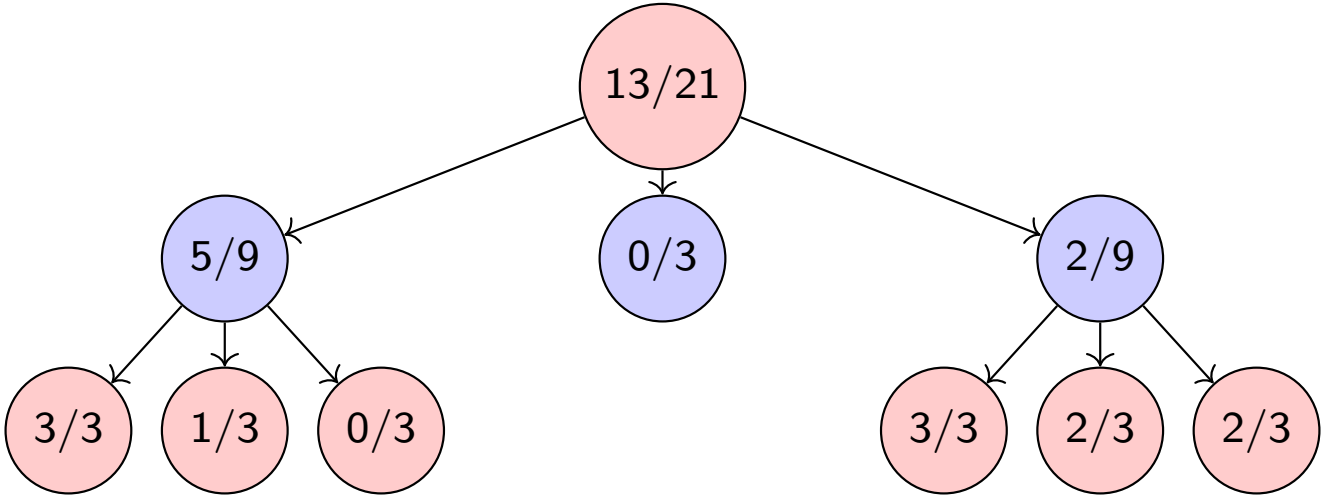
Trop grand nombre de possibilités à chaque tick : plus de 10^{50}

On utilise la recherche arborescente Monte-Carlo :

Exploration de l'arbre des possibilités en vue du choix d'une bonne solution

Utilisé en Go, en échecs et pour d'autres jeux au tour par tour.

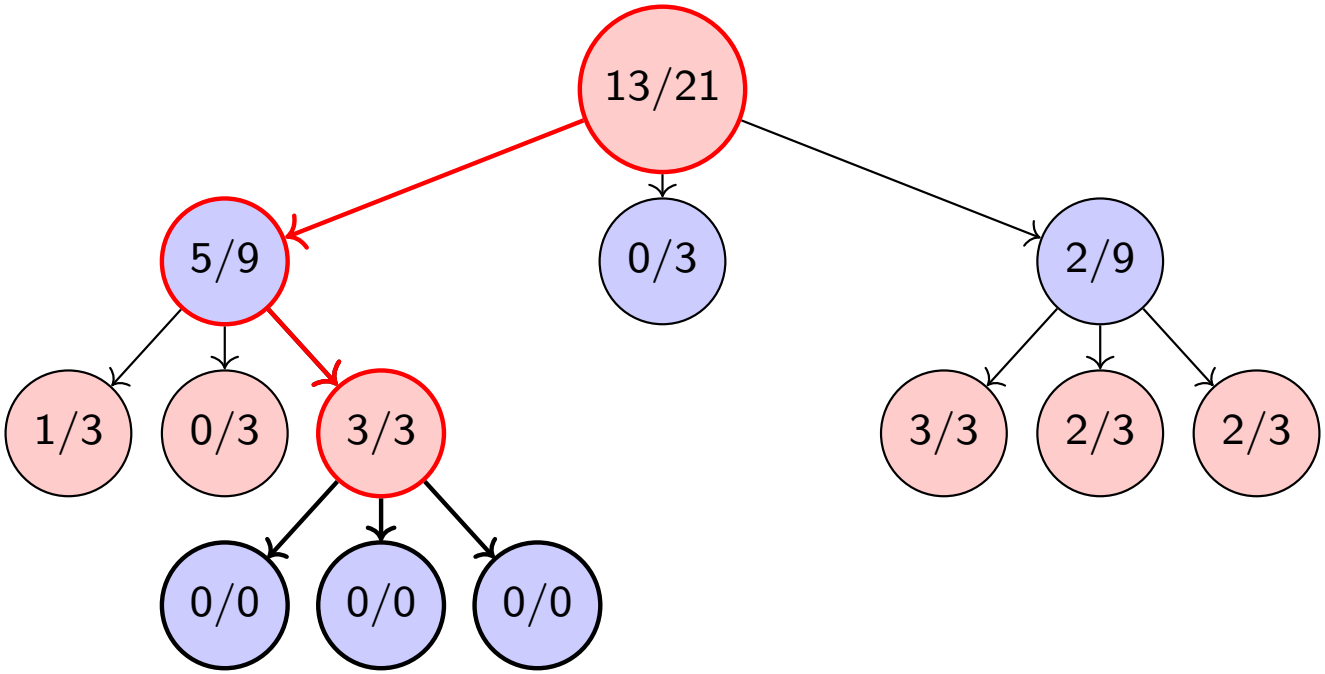
Utilisation MCTS



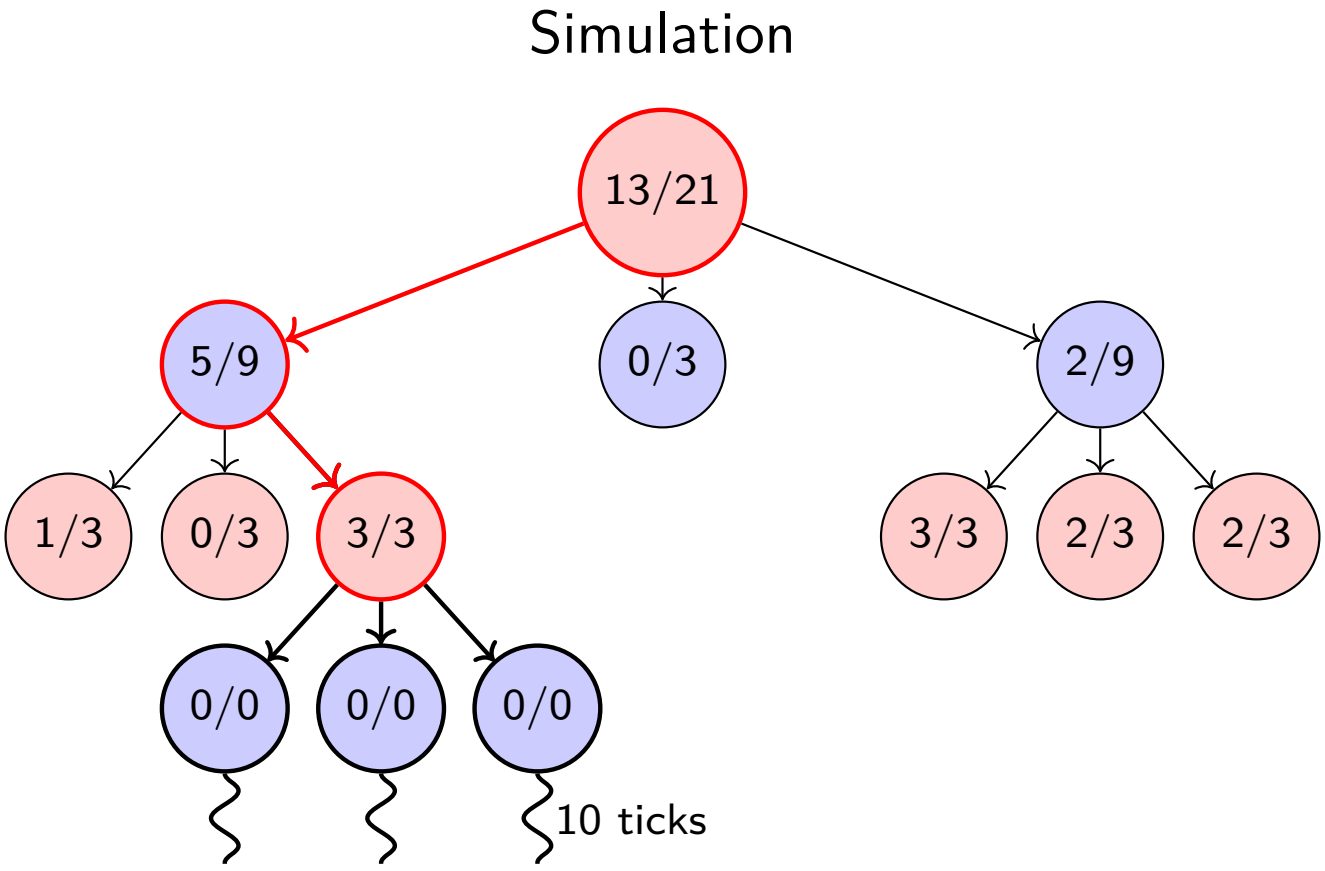
Utilisation MCTS

Utilisation de UCT pour la sélection : $\frac{w}{n} + c\sqrt{\frac{\ln N}{n}}$

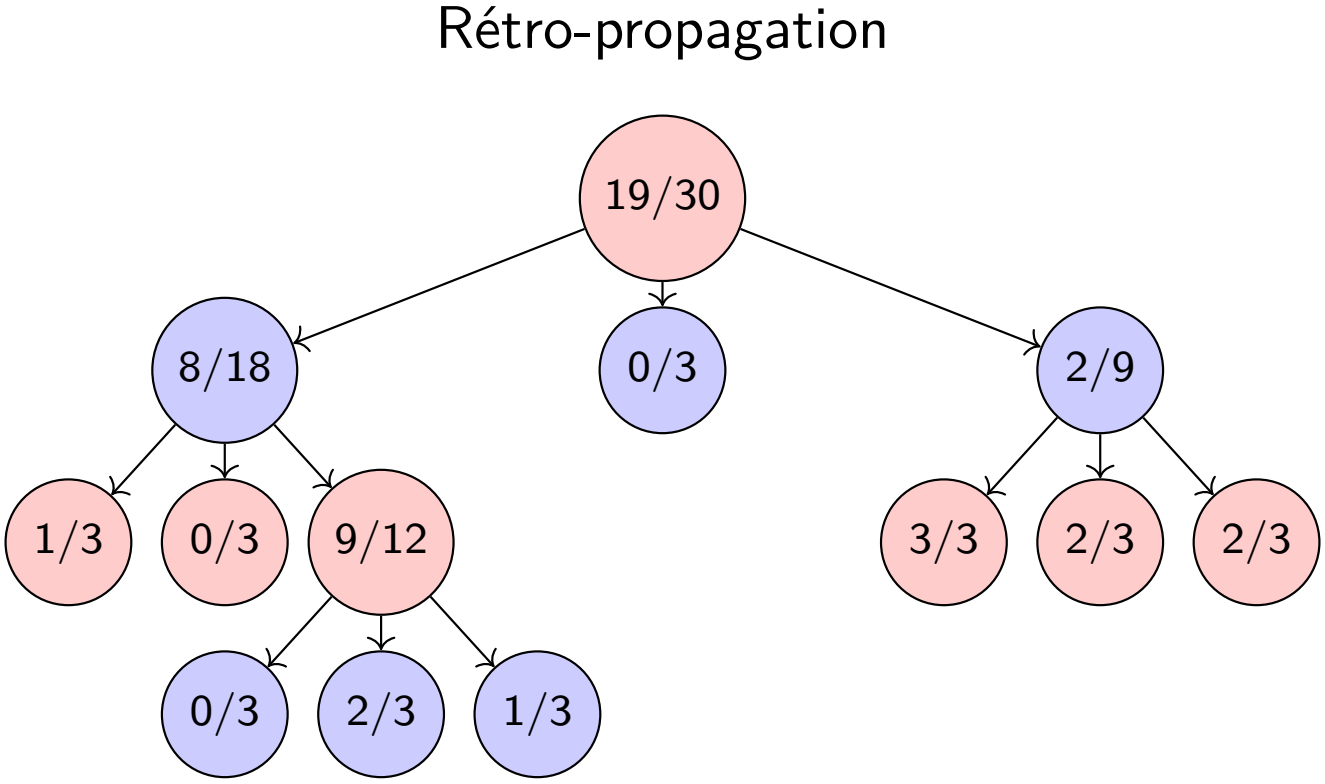
Selection et expansion



Utilisation MCTS



Utilisation MCTS

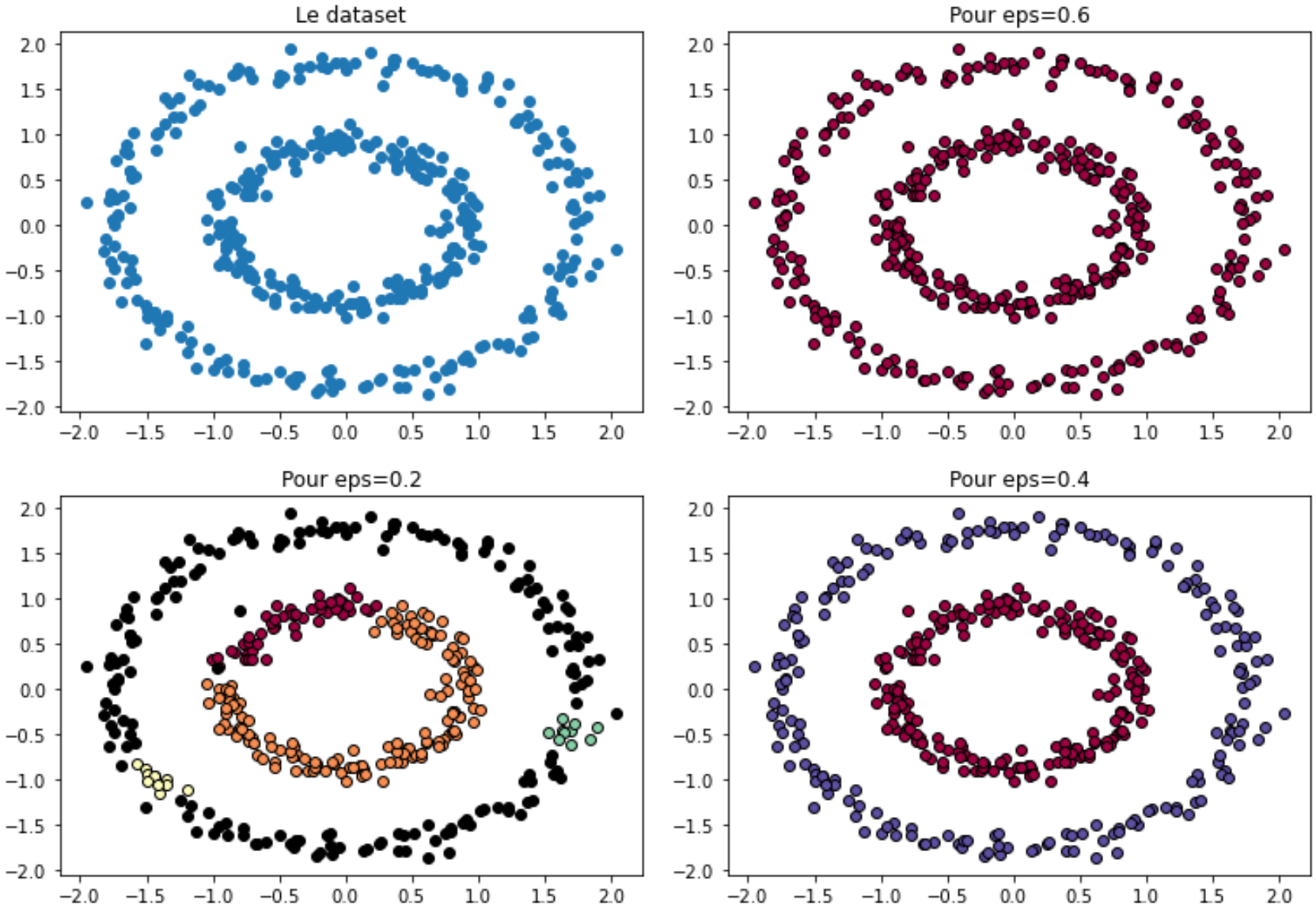


Résultats MCTS

Assez faible profondeur : prends trop de temps

Résultat médiocre : gagne 70% du temps contre joueur aléatoire

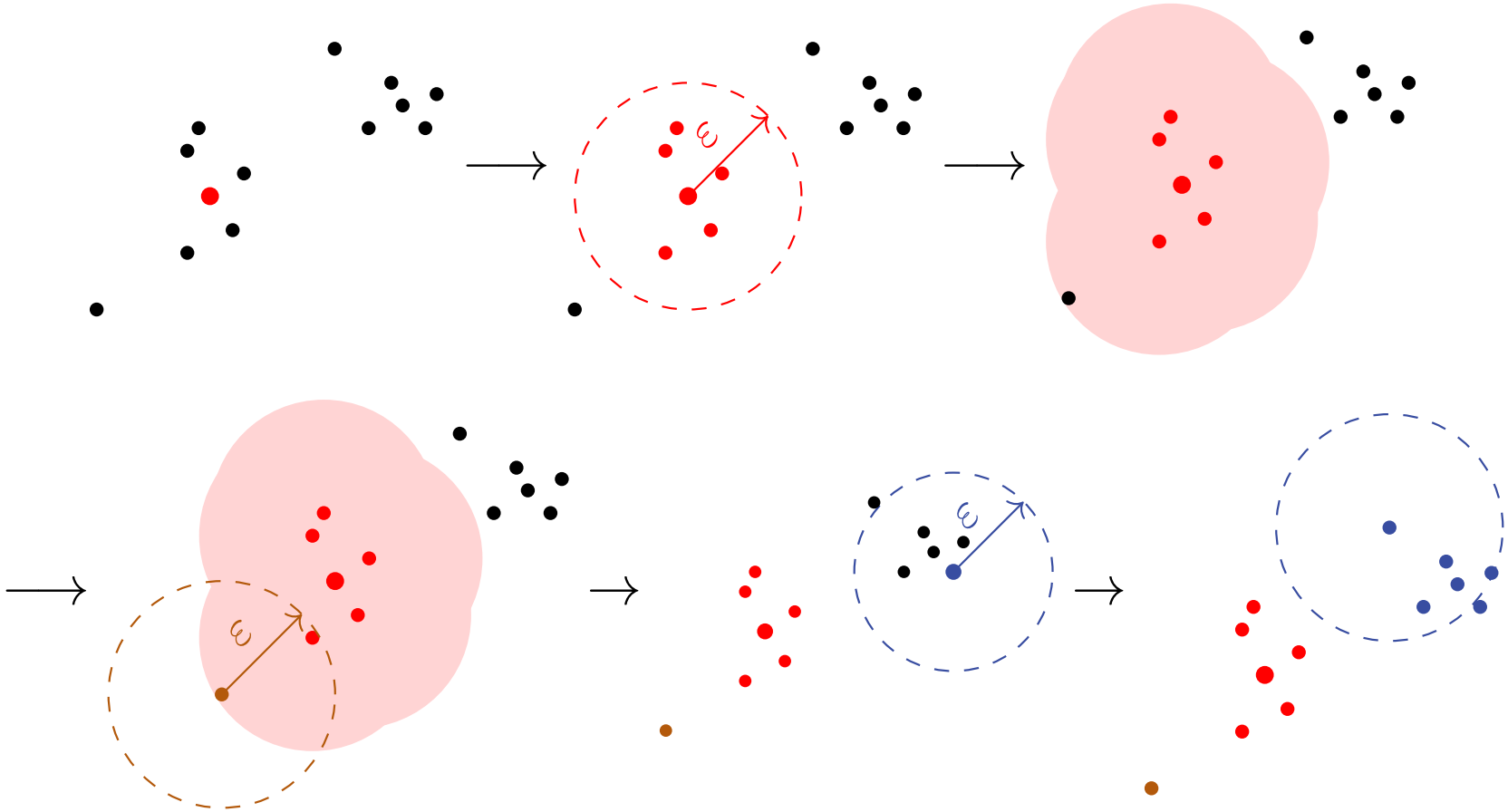
Rassemblement des unités : DBSCAN



Rassemblement des unités : DBSCAN

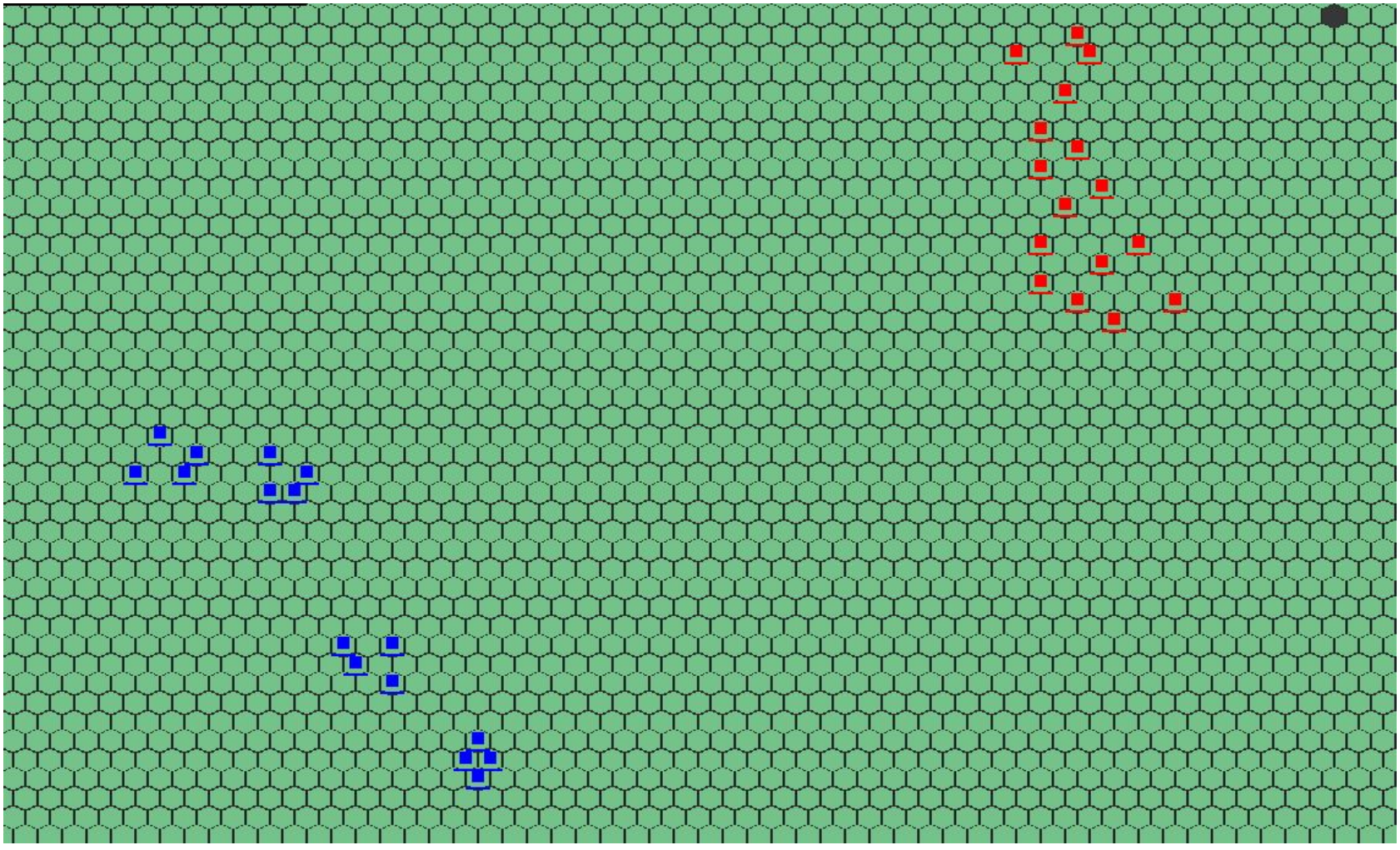
Deux paramètres : ε et *MinPts*

Ici avec $MinPts = 3$:



Points frontière et centraux du cluster 1 et du cluster 2

Application du DBSCAN



Monter en abstraction

- On génère des armées : DBSCAN
- On applique le MCTS à ces armées
- On accélère le MCTS grâce à la loi de Lanchester

$$\begin{cases} \frac{dA}{dt}(t) = -\alpha B(t) \\ \frac{dB}{dt}(t) = -\beta A(t) \end{cases}$$

avec $A(t)$ et $B(t)$ le nombre d'unitées dans les armées A et B à l'instant t
α et β les coefficients d'attritions de Lanchester de A et B :

$$\begin{cases} \alpha = \frac{DPF(B,A)}{HP(A)} \\ \beta = \frac{DPF(B,A)}{HP(B)} \end{cases}$$

Intensitée du combat définie par $I = \sqrt{\alpha\beta}$

On peut alors approximer le vainqueur :

$$\frac{|A(0)|}{|B(0)|} \begin{cases} > R_\alpha \text{ victoire de A} \\ = R_\alpha \text{ égalité} \\ < R_\alpha \text{ victoire de B} \end{cases}$$

avec $R_\alpha = \sqrt{\frac{\alpha}{\beta}}$ et $R_\beta = \sqrt{\frac{\beta}{\alpha}}$
Unités restantes à la fin :

$$\begin{cases} A(T) = \sqrt{A(0)^2 - \frac{\alpha}{\beta}B(0)^2}, B(T) = 0 \text{ si A gagn} \\ B(T) = \sqrt{B(0)^2 - \frac{\beta}{\alpha}A(0)^2}, A(T) = 0 \text{ si B gagn} \end{cases}$$

```
std::vector<hex_tile*> map_class::path_a_star(hex_tile* start, hex_tile* end)
    const
{
        constexpr int a_weight = 5;
        const auto list_closed = new bool*[size_y_];
        const auto weight_g = new double*[size_y_];
        const auto weight_h = new double*[size_y_];
        const auto parent = new hex_tile**[size_y_];
        for (int i = 0; i < size_y_; i++)
        {
                list_closed[i] = new bool[size_x_]();
                weight_g[i] = new double[size_x_]();
                weight_h[i] = new double[size_x_]();
                parent[i] = new hex_tile*[size_x_]();
        }
        vector<hex_tile*> list_open;
        hex_tile* current_node = start;
        weight_g[current_node->index_y()][current_node->index_x()] = 0;
        weight_h[current_node->index_y()][current_node->index_x()] =
             current_node->distance(end) * a_weight;
        list_open.push_back(current_node);
        make_heap(list_open.begin(), list_open.end(), compare(weight_g, weight_h
             ));
        int index = 0;
        while (!list_open.empty())
        {
                index++;
                current_node = list_open.front();
                list_open.erase(list_open.begin());
                list_closed[current_node->index_y()][current_node->index_x()] =
                     true;
                const double cost_g = weight_g[current_node->index_y()][
                     current_node->index_x()] + 1.0;
                if (list_closed[end->index_y()][end->index_x()])
                {
                        break;
                }
```

```
if (constexpr double depth = 200.0; cost_g > depth)
{
        break;
}
for (int dir = 0; dir < 6; dir++)
{
        hex_tile* children = neighbor(current_node->q(),
             current_node->r(), dir);
        if (children == nullptr)
        {
                continue;
        }
        if (!children->passable())
        {
                continue;
        }
        if (list_closed[children->index_y()][children->index_x()
             ])
        {
                continue;
        }
        if (find(list_open.begin(), list_open.end(), children)
             == list_open.end())
        {
                weight_g[children->index_y()][children->index_x
                     ()] = cost_g;
                weight_h[children->index_y()][children->index_x
                     ()] = (children->distance(end)) * a_weight;
                list_open.push_back(children);
                push_heap(list_open.begin(), list_open.end(),
                     compare(compare(weight_g, weight_h)));
                parent[children->index_y()][children->index_x()]
                      = current_node;
        }
```

```
                    else if (weight_g[children->index_y()][children->index_x
                        ()] + weight_h[children->index_y()][children->
                        index_x()] > cost_g + weight_h[children->index_y()
                        ][children->index_x()])
                    {
                        weight_g[children->index_y()][children->index_x
                            ()] = cost_g;
                        parent[children->index_y()][children->index_x()]
                             = current_node;
                        make_heap(list_open.begin(), list_open.end(),
                            compare(compare(weight_g, weight_h)));
                    }
                }
            }
            vector<hex_tile*> path;
            if (list_closed[current_node->index_y()][current_node->index_x()])
            {
                while (current_node != start && current_node != nullptr)
                {
                    path.push_back(current_node);
                    current_node = parent[current_node->index_y()][
                        current_node->index_x()];
                }
                reverse(path.begin(), path.end());
            }
            for (int i = 0; i < size_y_; i++)
            {
                delete[] list_closed[i];
                delete[] weight_g[i];
                delete[] weight_h[i];
                delete[] parent[i];
            }
            delete[] list_closed;
            delete[] weight_g;
            delete[] weight_h;
            delete[] parent;
            return path;
        }
```