Alexandre LE

David ANATON

Warfare

Rapport de développement

1. Table des matières

2. Historique des modifications

Name	***Date*** *(dd/mm/yy)*	***Reason For Changes***	***Version***

3. Introduction

Ce rapport a pour vocation d'expliquer l'approche utilisée pour résoudre les 3 problèmes qui nous ont été posés.

Sans plus attendre, entrons directement dans le vif du sujet.

4. Mission 1 _ Remplissage automatisé de la soute d'un bateau.

4.1. Présentation du problème

Une soute de 9 emplacements doit être remplie avec des conteneurs de différentes contenances.

Chaque emplacement peut recevoir 4 conteneurs maximum.

Les conteneurs disponibles sont :

- Vivres :
  - Ces conteneurs sont à traiter en premier à l'arrivée du bateau et doivent être placés au sommet des piles.
- Artillerie :
  - Ces conteneurs sont les plus lourds et doivent être placés au bas des piles pour ne pas endommager les autres.
  - Du fait de leurs poids et pour ne pas déséquilibrer le bateau, nous devons limiter le nombre de conteneurs d'artillerie à 2 par pile.
- Munitions :
  - Ces conteneurs sont susceptibles d'exploser et ne doivent pas se trouver à plus d'un exemplaire dans chaque pile.
  - En cas d'attaque du bateau, il faut accéder à un exemplaire de ce conteneur rapidement. Pour cela, en placer 1 à l'emplacement le plus proche du cockpit.
- Combat léger :
  - En cas d'attaque du bateau, il faut accéder à un exemplaire de ce conteneur rapidement. Pour cela, en placer 1 à l'emplacement le plus proche du cockpit.

Le bateau se schématise de la façon suivante :

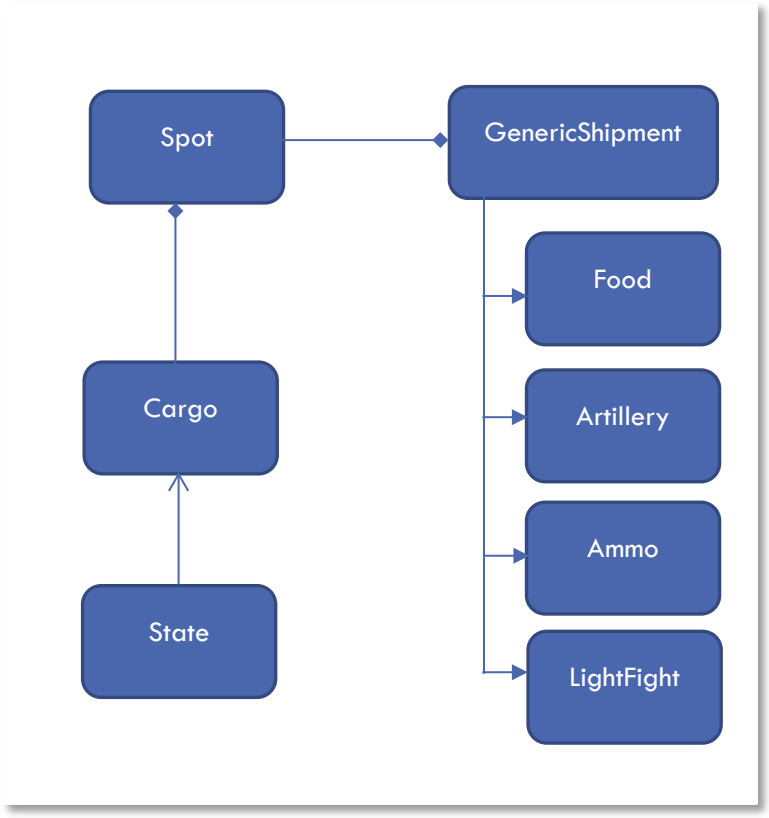
Cockpit	Emplacement 1	Emplacement 2	Emplacement 3
	Emplacement 4	Emplacement 5	Emplacement 6
	Emplacement 7	Emplacement 8	Emplacement 9

Dans la liste de conteneurs à embarquer, on trouve :

- 8 conteneurs d'Artillerie
- 11 conteneurs de vivres
- 9 conteneurs de munitions
- 8 conteneurs de combat léger

4.2. Modélisation

Voici la représentation UML de la structure des données que nous avons adoptée :

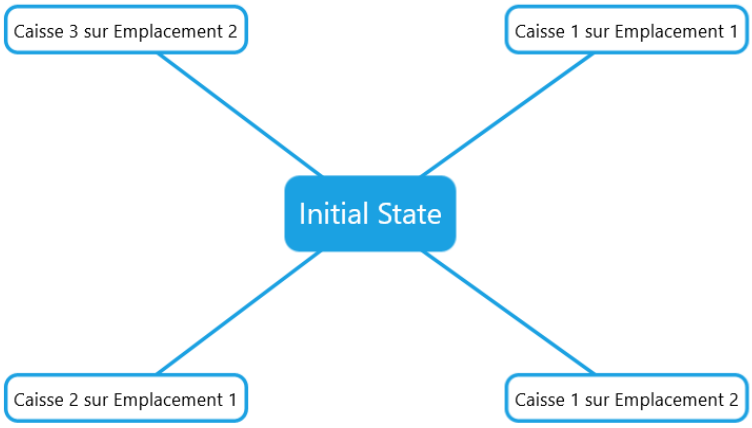


4.3. Algorithme

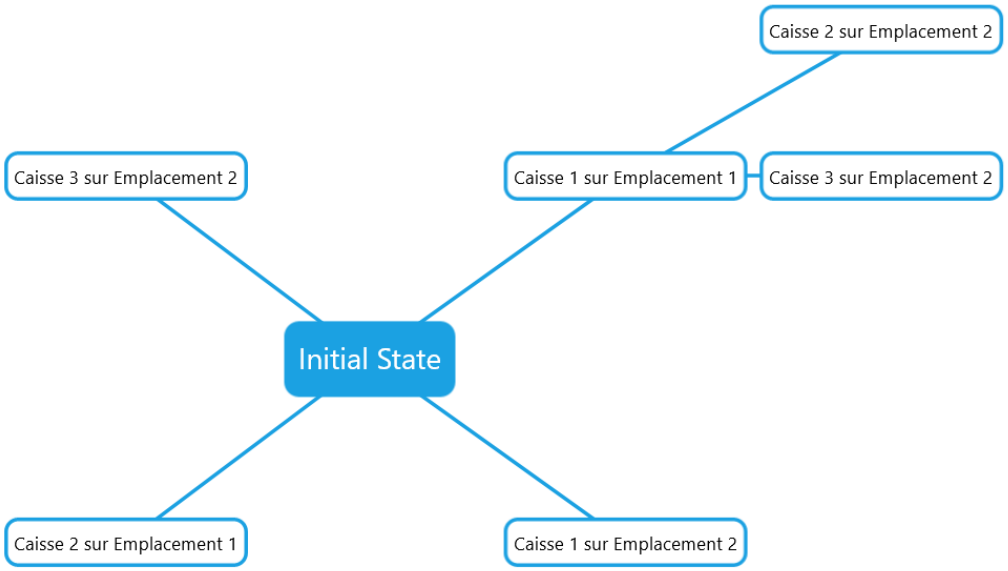
Nous avons opté pour un algorithme DFS (Depth First Search). Ce dernier favoriser la recherche en profondeur d'une solution.

Voici le principe de cet algorithme (de façon schématique, bien sûr).

1) A partir de l'état initial, on génère les états possibles.

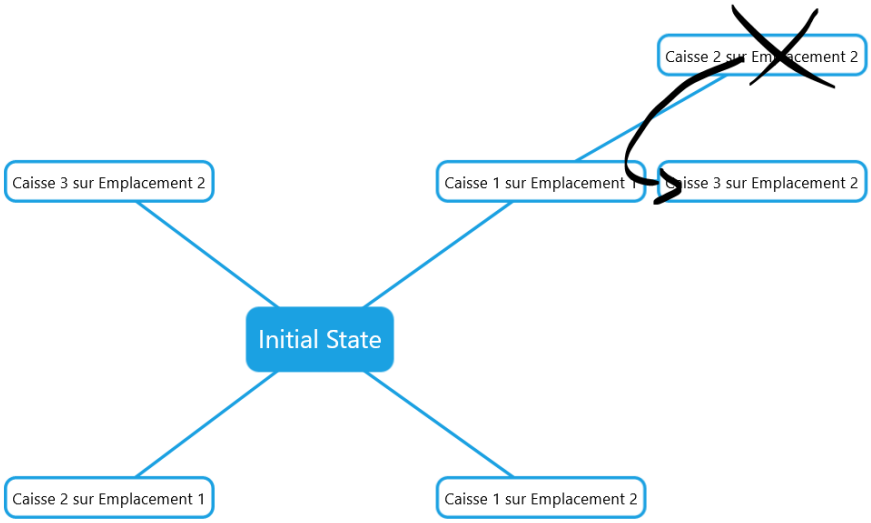


2) On choisit le premier état qui arrive et on trouve les états qui peuvent être atteints à partir de ce dernier :

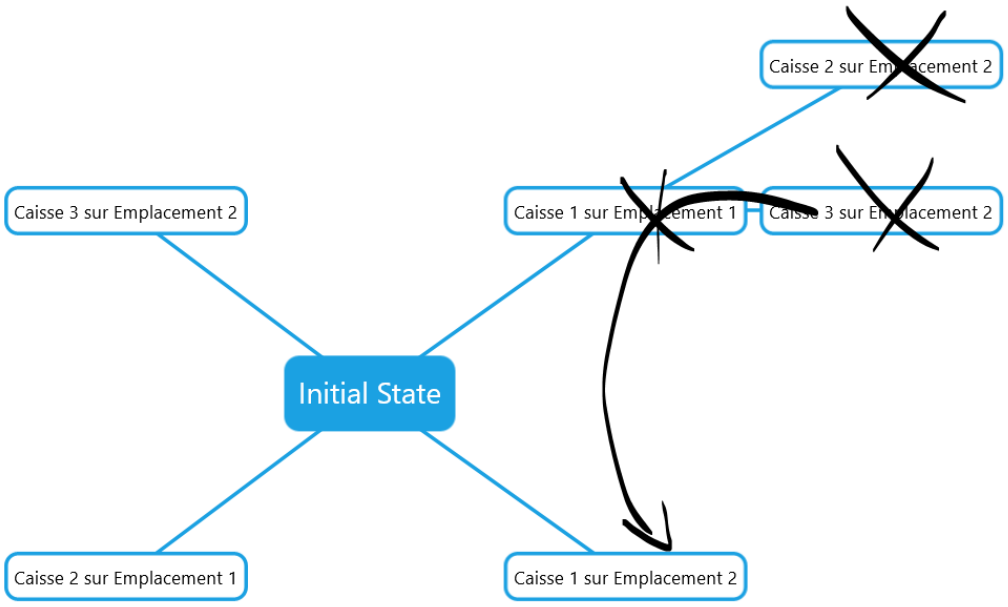


3) On choisit le premier enfant rencontré et développe ses enfants. Dans notre cas, Caisse 1 est sur l'emplacement 1 et Caisse 2 sur l'emplacement 2. Caisse 3 ne pouvant pas se placer sur une caisse de type Caisse 1 ou Caisse 2, le nœud où l'on est se

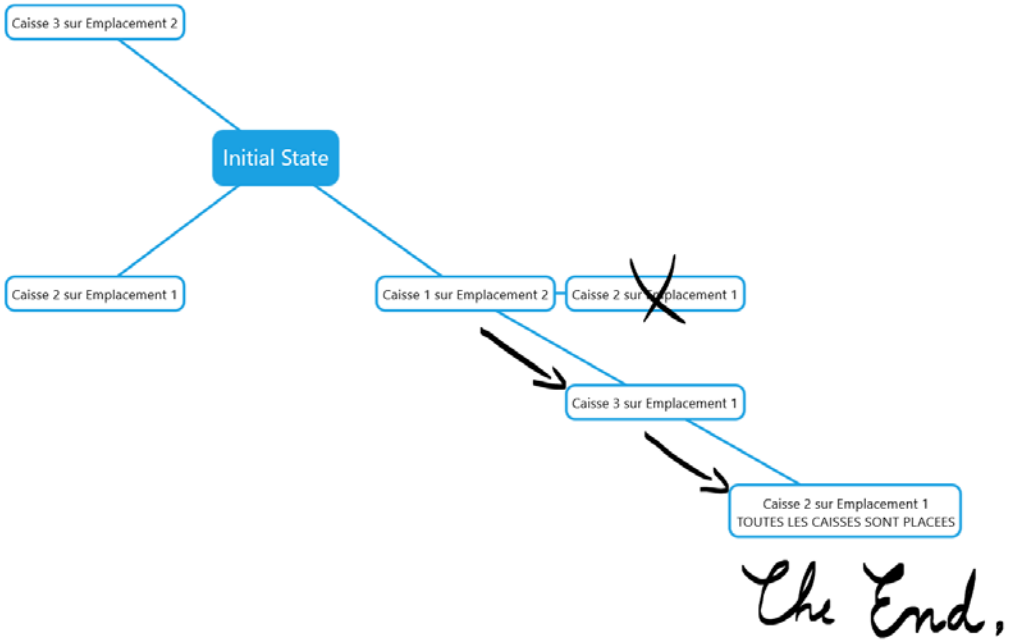
retrouve sans enfant. On passe donc à son voisin direct.



4) Supposons maintenant que le nœud où l'on est maintenant n'a toujours pas de descendant. Il n'a pas non plus de voisin. On remonte donc d'un niveau et passe au voisin de son précurseur.



5) Nous explorons maintenant les enfants du nœud où nous sommes et appliquons le même raisonnement jusqu'à trouver un nœud où toutes les caisses sont placées.



Ce type d'algorithme est particulièrement adapté lorsque le nombre de choix possibles à chaque niveau de l'arbre est très important et qu'il suffit de trouver une seule configuration validant certaines conditions (même s'il en existe plusieurs). Il devient indispensable lorsque la pertinence des nœuds parents ne peut pas être établie sans la connaissance des nœuds enfants.

4.4. Résultat

Après implémentation en Java, le programme donne la configuration suivante :

```
CARGO

Emplacement 1
	LightFight
	Ammo
	Artillery
	Artillery
Emplacement 2
	LightFight
	Ammo
	Artillery
	Artillery
Emplacement 3
	LightFight
	Ammo
	Artillery
	Artillery
Emplacement 4
	LightFight
	Ammo
	Artillery
	Artillery
Emplacement 5
	LightFight
	LightFight
	LightFight
	Ammo
Emplacement 6
	Food
	Food
	LightFight
	Ammo
Emplacement 7
	Food
	Food
	Food
	Ammo
Emplacement 8
	Food
	Food
	Food
	Ammo
Emplacement 9
	Food
	Food
	Food
	Ammo

REMAINING 0
```

5. Mission 2 – Transport des bateaux entre zones de combat

5.1. Présentation

Une fois les bateaux chargés il faut organiser leurs déplacements vers le champ de bataille.

Nous avons donc 4 bateaux à déplacer :

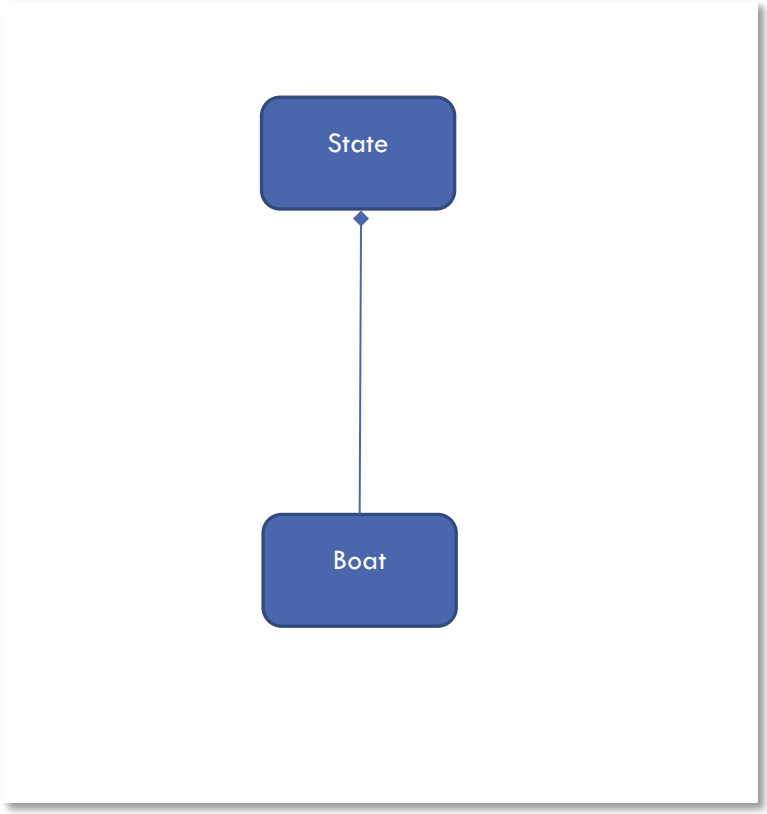
- Un XC21 qui effectue le trajet en 45 minutes.
- Un XC56 qui fait le trajet en 1 heure et 30 minutes.
- Un XC100 qui peut être sur place après 3 heures et 45 minutes.
- Un XC800 qui met 6 heures à rejoindre les festivités.

Le problème étant que chaque bateau ne peut traverser les zones adverses que s'il est accompagné d'une équipe de sécurité. Nous ne disposons que d'une unique équipe de sécurité et cette dernière peut protéger 2 bateaux à la fois.

Le but de ce problème est de minimiser la somme des temps de parcours de chaque convoi.

5.2. Modélisation

Voici la représentation UML de la structure des données que nous avons adoptée :



5.3. Algorithme utilisé

Nous avons opté pour un algorithme A*. Il permet de favoriser l'exploration du nœud le plus prometteur dans la recherche d'une solution.

En voici un schéma simplifié. Le but de l'exemple est de partir de init pour arriver à end sans passer par les cases noires. Le prix d'un déplacement horizontal/vertical est 1x 10 = 10. Le prix d'un déplacement

1) On souhaite trouver le plus court chemin entre init et end

				end
	init			

2) On marque init comme visité et on trouve les nœuds accessibles à partir de l'état initial. Dans chaque case bleue, on se souvient alors que le parent de la case en question est init.

				end
	init			

3) On évalue le coût de chaque coup (pour le coup, c'est cocasse) et, surtout, l'heuristique de chaque coup (i.e. : un minorant de la distance à laquelle le nœud est de la destination).
Prenons l'exemple de la case en haut à droite : pour y aller, on a dépensé une case en diagonale, soit F = prix_diagonale = 14. Son heuristique est de 2 diagonales soit H = 2 x prix_diagonale 28. S = H +F.

				end
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42		
F :10, H :52 S :62	init	F :10, H :38 S :48		

4) Le nœud promettant alors le plus court chemin est la case en haut à droite avec un coût de trajet total de 42 (c'est toujours la bonne réponse). C'est cette case qu'on visite.
Pour la case la plus en haut à gauche, toujours, le coût final est celui du nœud parent (14) + une horizontale = 24. Le reste du calcul est le même.

				end
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42	F :24, H :24 S :48	
F :10, H :52 S :62	init	F :10, H :38 S :48	F :28, H :34 S :62	

5) Maintenant, les nœuds les plus promettant sont à 48. On prend la plus petite heuristique.

				end
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42	F :24, H :24 S :48	F :34, H :20 S :54
F :10, H :52 S :62	init	F :10, H :38 S :48	F :28, H :34 S :62	F :38, H :30 S :68

6) A H et S égaux, peu importe, on choisit le premier.

				end
F :24, H :44 S :68				
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42	F :24, H :24 S :48	F :34, H :20 S :54
F :10, H :52 S :62	init	F :10, H :38 S :48	F :28, H :34 S :62	F :38, H :30 S :68

7) Remarquez qu'en visitant la nouvelle case, la case plus foncée a vu ses valeurs mises à jour car les valeurs trouvées en passant par la case actuelle sont plus petites que les valeurs trouvées précédemment. Ainsi, la case foncée verra son parent mis-à-jour.

				end
F :24, H :44 S :68				
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42	F :24, H :24 S :48	F :34, H :20 S :54
F :10, H :52 S :62	init	F :10, H :38 S :48	F :20, H :34 S :54	F :38, H :30 S :68

8) On continue le même algorithme jusqu'à atteindre l'état final.

F :34, H :40 S :74	F :38, H :30 S :68	F :48, H :20 S :68	F :58, H :10 S :68	end
F :24, H :44 S :68				
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42	F :24, H :24 S :48	F :34, H :20 S :54
F :10, H :52 S :62	init	F :10, H :38 S :48	F :20, H :34 S :54	F :30, H :30 S :60

9) Il ne reste plus qu'à remonter d'enfant en parent jusqu'à trouver le chemin total.

F :34, H :40 S :74	F :38, H :30 S :68	F :48, H :20 S :68	F :58, H :10 S :68	end
F :24, H :44 S :68				
F :14, H :48 S :62	F :10, H :38 S :48	F :14, H :28 S :42	F :24, H :24 S :48	F :34, H :20 S :54
F :10, H :52 S :62	init	F :10, H :38 S :48	F :20, H :34 S :54	F :30, H :30 S :60

Cet algorithme est particulièrement adapté dans la recherche d'un plus court chemin. Son avantage sur le Djikstra est qu'il n'explore que les chemins les plus prometteurs. Son heuristique impose que l'état solution soit unique (ou qu'on puisse assimiler les différents états finaux à un même état).

5.4. Résultat

Après implémentation, on obtient les instructions dans l'invite de commande d'Eclipse.

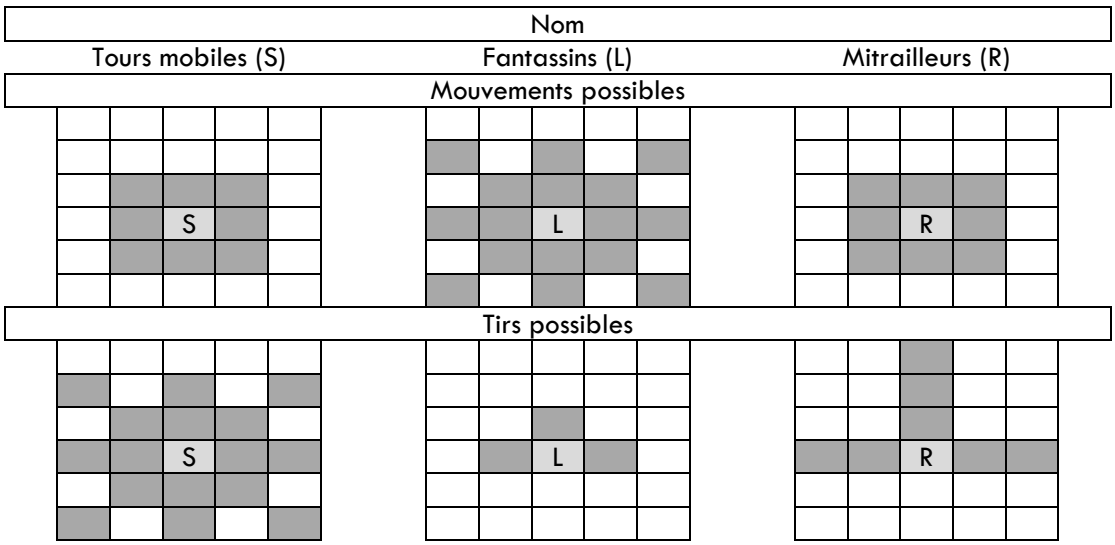
```
Etape 1
XC21     -->
XC56     -->
XC100    -->
XC800    -->
F:0 H:0

Etape 2
XC100              <-- XC56
XC800              <-- XC21
F:90 H:405

Etape 3
XC21     -->           XC56
XC100    -->
XC800    -->
F:135 H:450

Etape 4
XC21                <-- XC56
                    <-- XC800
                    <-- XC100
F:495 H:90

Etape 5
XC21     -->            XC800
XC56     -->            XC100
F:585 H:90

Etape 6
                    <-- XC800
                    <-- XC100
                    <-- XC56
                    <-- XC21
F:675 H:0
```

6. Mission 3 – Automatisation des phases de combats

6.1. Présentation

Un fois les soldats en place et les conteneurs livrés sur le champ de bataille, on s'occupe de gérer les déplacements et tirs de nos atouts pour limiter les pertes alliées et se débarrasser des forces ennemies.

Nous avons donc 3 types de ressources à notre disposition :

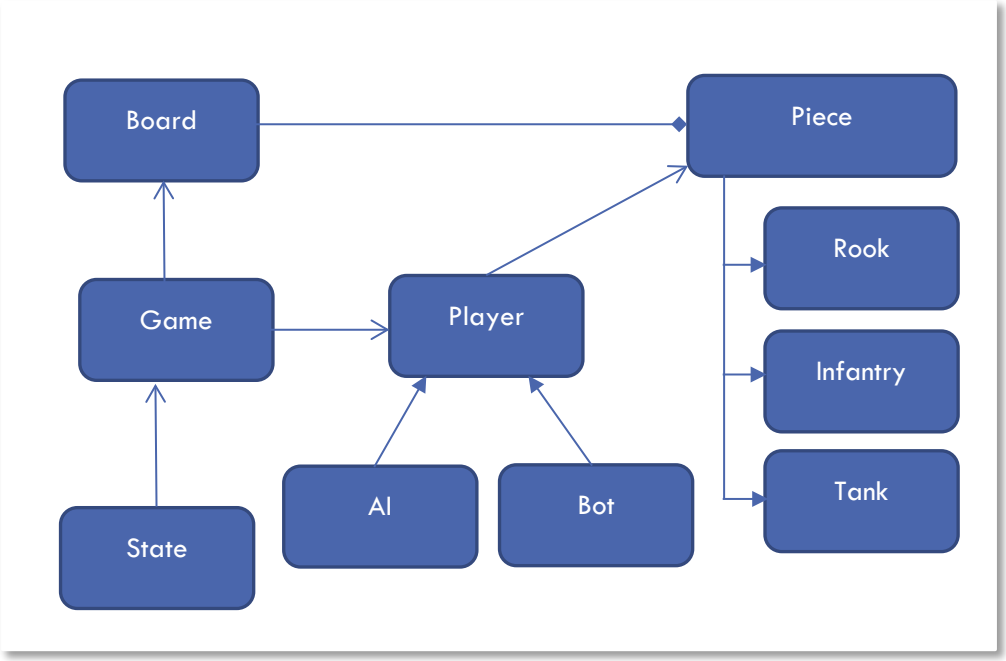


Le champ de bataille peut être initialement modélisé de la façon suivante :

					S			S				S				S			S					
			R																	R				
					L		L		L		L		L		L		L		L					
		R								L		L		L							R			
		R								L		L		L							R			
					L		L		L		L		L		L		L		L					
			R																	R				
					S			S				S				S			S					

6.2. Modélisation

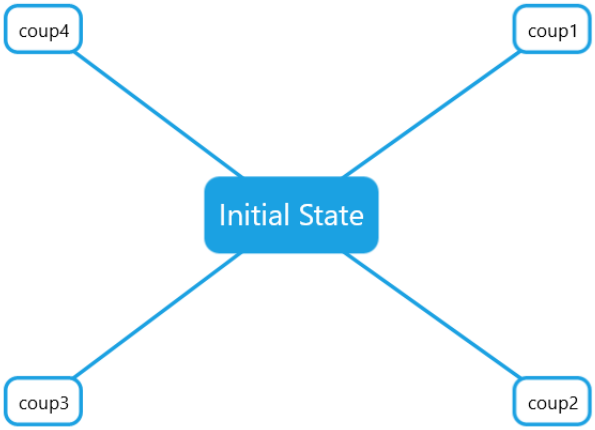
Voici le diagramme UML de la représentation des données adoptée :



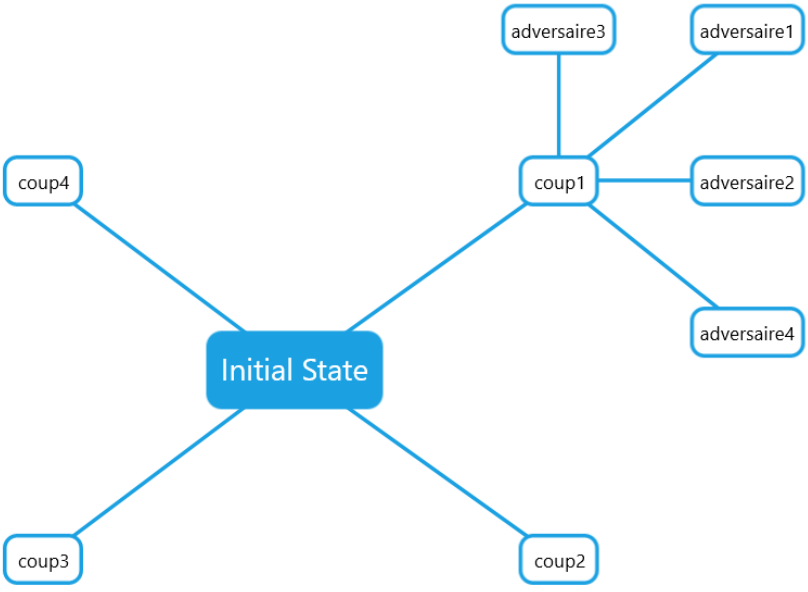
6.3. Algorithme utilisé

On suppose que le jeu auquel nous jouons est jeu non-coopératif synchrone à information complète (on a toutes les informations à disposition) et à somme nulle (le score d'un joueur est l'inverse du score de l'adversaire). Cette dernière hypothèse sera validée par le choix de notre valorisation des états. Le choix de l'algorithme à utiliser a été imposé. Nous avons donc utilisé un algorithme Minimax. Cet algorithme se base sur un DFS mais prend en compte l'aspect « jeu » du problème. Voici le détail de son fonctionnement :

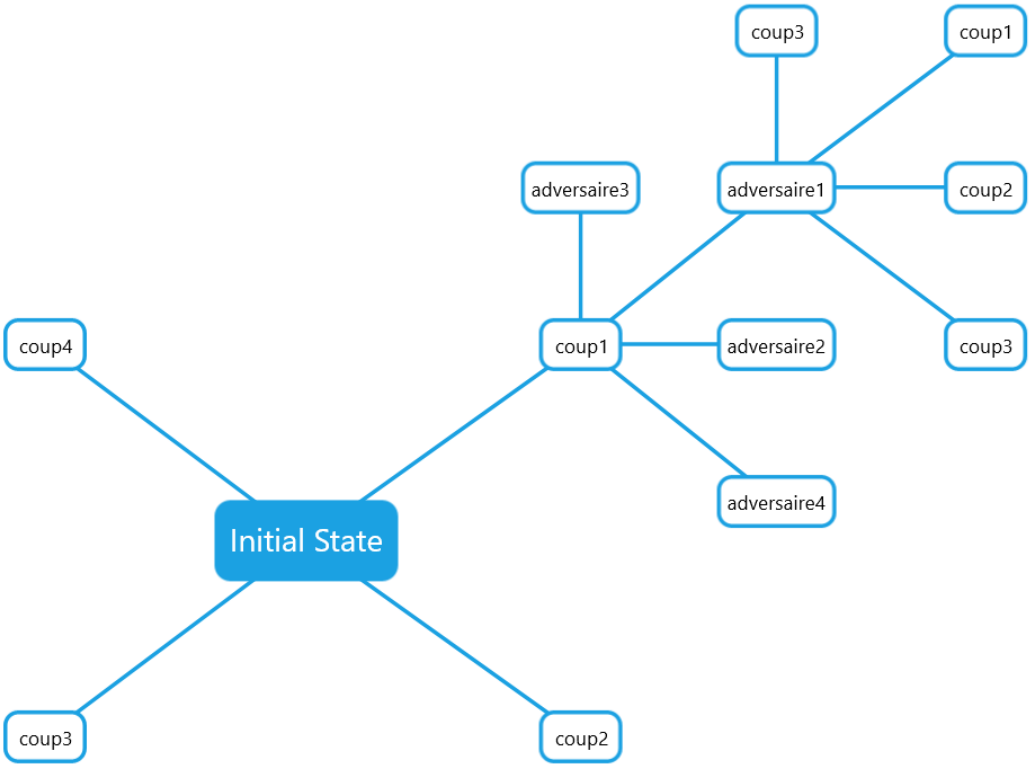
1) On génère tous les fils de l'état initial.



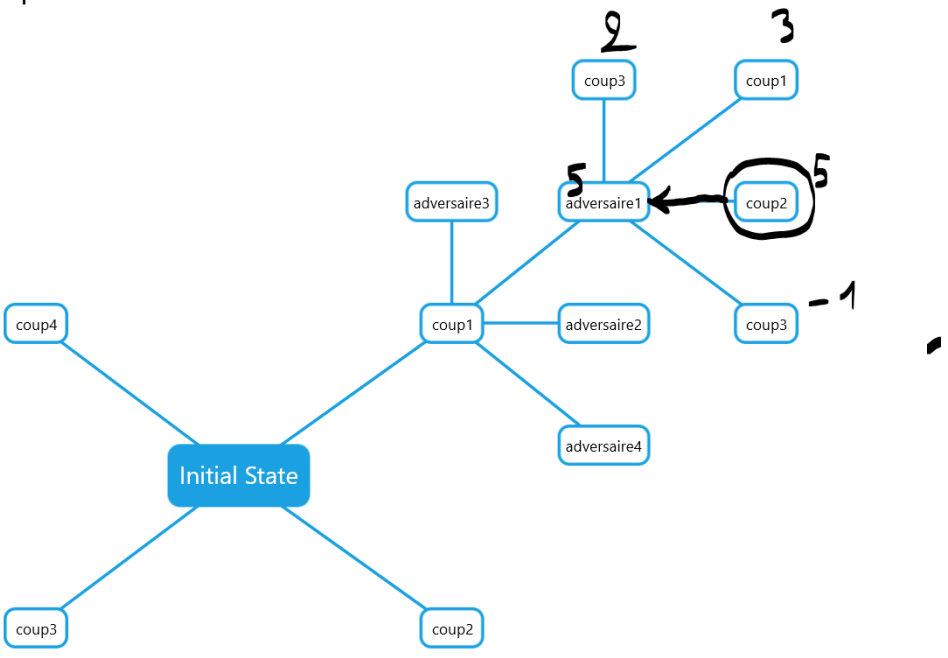
2) Comme pour le DFS, on génère le fils de son premier fils.



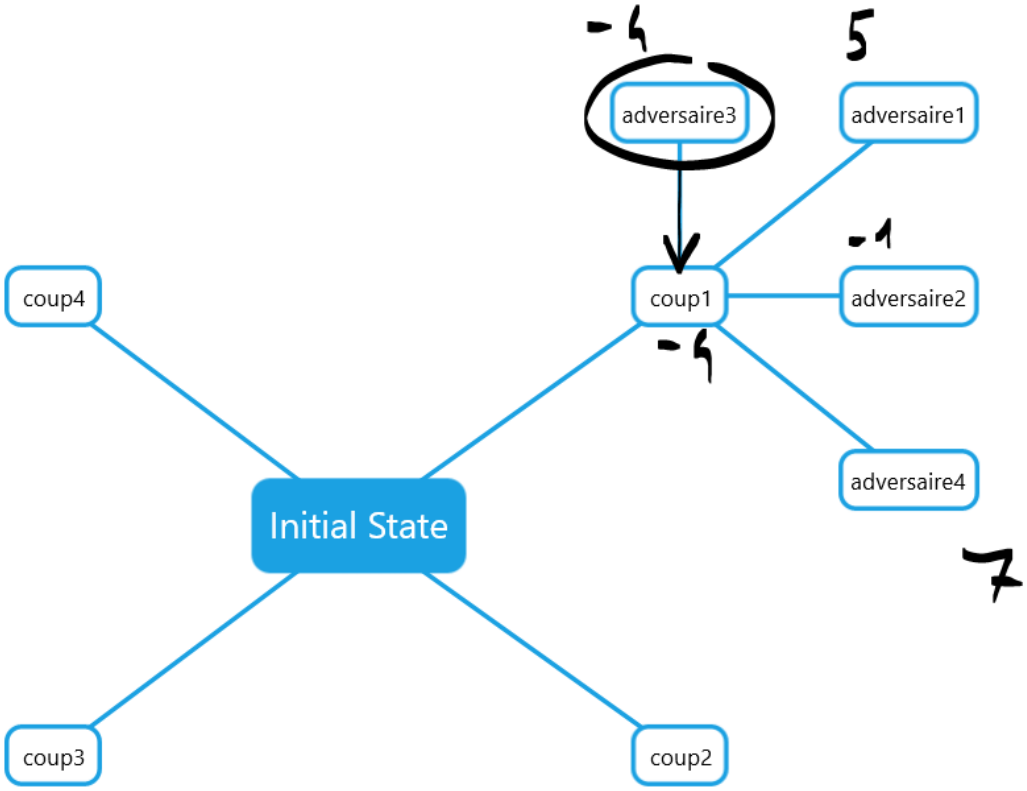
3) On continue jusqu'à arriver à la profondeur désirée (nous avons choisi 3).



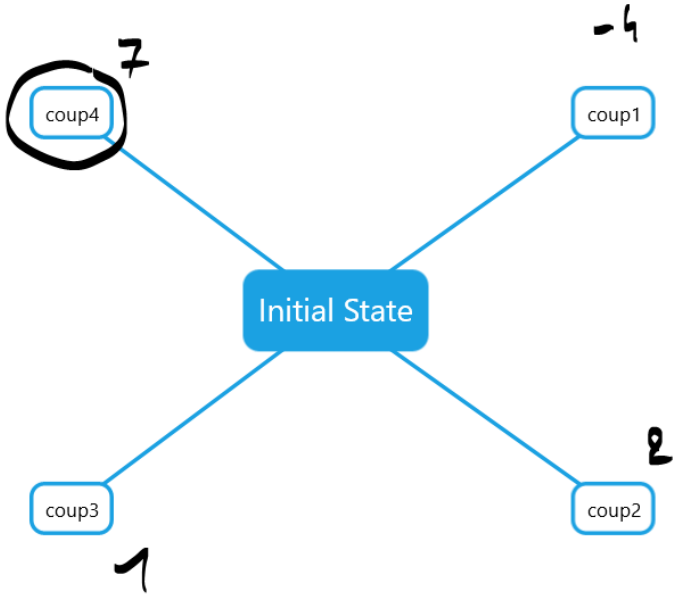
4) On calcule un score correspondant à chaque état généré. Dans notre cas, nous avons choisi le nombre de pièce et avons essayé d'ajouter une notion de territoire. Dans le cas où l'adversaire joue son coup 1, on pourra, au MAXIMUM, marquer 5 points. On marque donc cette valeur dans adversaire 1.



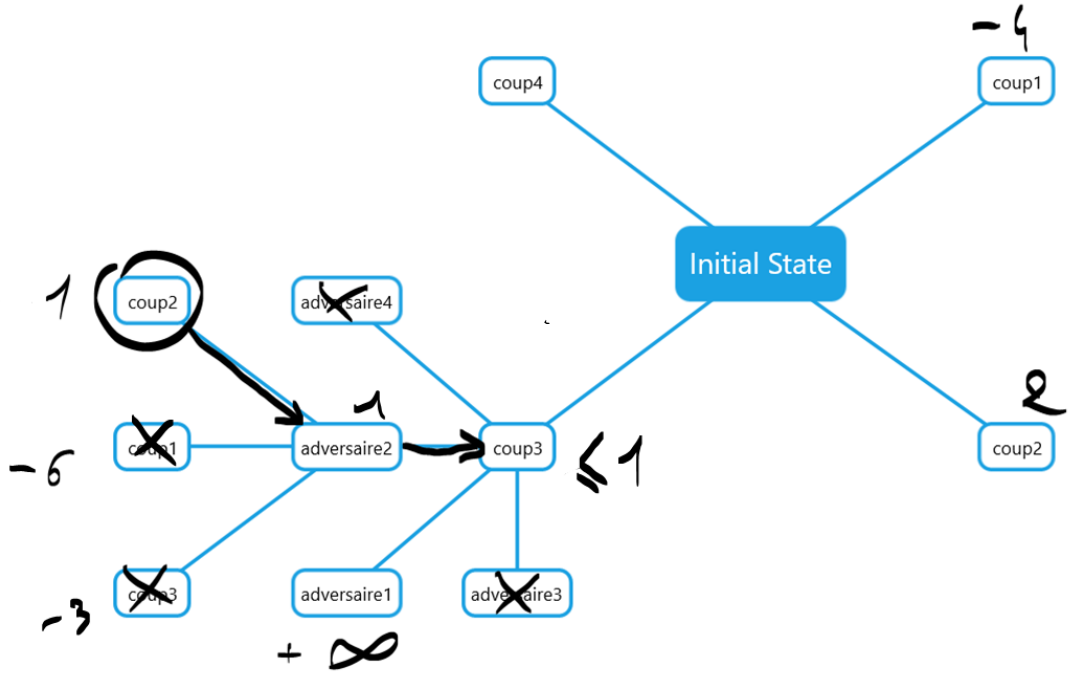
5) On applique le même raisonnement pour chaque nœud du niveau supérieur. On suppose ensuite que l'adversaire jouera son coup optimal. Notre score après son coup sera donc au MINIMAL égal à -4.



6) On alterne ensuite ces phases de minimisations et maximisations jusqu'à remplir tous nos choix de niveau 1. Il apparait ensuite de façon évidente que le meilleur coup à jouer est le coup 4 pour un score minimal de 7 (selon la qualité de jeu de l'adversaire)



7) Bonus : utilisation d'un couple alpha-beta. On peut également réduire grandement le temps de parcours de notre graphe. Pour cela, lors du remplissage du coup3, on peut remarquer que dès que l'on trouve une valeur inférieure à 2 (valorisation du coup 2 de niveau 1) ce dernier (ou un autre état avec une valorisation encore plus faible) sera choisi par l'adversaire. Donc le coup 3 nous sera toujours moins avantageux que le coup 2 et on n'explore même pas le reste de l'arbre pour le coup 3.



6.4. Résultats

Nous avons donc une IA qui gagne systématiquement toutes ses parties contre un robot jouant de façon aléatoire.

7. Conclusion

Merci pour votre attention.