École Polytechnique

PSC

Surveillance de zone par réseau de drones

Rapport intermédiaire

Paul Mortamet
Thibault Vignon
Louis Proffit
Louis Stefanuto
X2019

Tuteur :
M. Vincent Jeauneau

Coordinateur :
M. Emmanuel Haucourt

25 Janvier 2021

Table des matières

Introduction

Le but de ce rapport intermédiaire est avant tout de dresser un bilan à mi-parcours de l'avancée de notre Projet Scientifique Collectif débuté en septembre dernier.

Ce projet s'inscrit initialement dans la continuité de celui d'un groupe de la promo X18 travaillant sur le même sujet : **l'optimisation de la surveillance de zone sensible par réseau de drones.**

D'après le retour d'expérience de nos ainés, leur approche du problème s'était révélée trop large et finalement peu frucuteuse. De nos échanges est donc resorti l'importance de mieux définir les ambitions du projet. Nous avons donc fait le choix de nous concentrer sur la répartition des drones et de laisser l'étude des autres "phases" de la surveillance (détection, poursuite, neutralisation) à d'éventuels groupes ultérieurs. En d'autres termes, **travailler ensemble sur une problématique plus précise plutôt que chacun sur des problèmes trop décorrélés.**

A l'exception de l'utilisation de l'algorithme A* pour les premières visualisations, nous avons donc fait **le pari de repartir de zéro** et d'explorer de nouvelles techniques de résolution du problème développées dans ce rapport.

Un autre point qui nous est paru trop peu développé l'année précédente est la génération de résultats visuels. Un de nos gros chantiers était et reste donc de mettre en place une interface graphique plus visuelle que le quadrillage statique du groupe X18.

Enfin, il nous est paru vital de nous assurer que nos réponses aux problèmes demeurent au maximum à l'image de la menace à laquelle elles doivent répondre, c'est-à-dire **modulables**. En effet, si les attaques de drones demeurent aujourd'hui aussi craintes, c'est principalement à cause de la diversité de formes qu'elles peuvent adopter. Nous veillons donc constamment à penser ou revoir nos codes afin que le changement de l'environnement (ajout de drone, d'intrus, ajout de checkpoint ...), puisse avoir lieu à tout instant, facilement, intuitivement et si possible directement depuis l'interface.

Nous avons donc décidé de nouvelles orientations : implémentation de davantage d'algorithmes de recherche de chemin et de répartition sur zone, développement plus poussé de nos interfaces graphiques... Ce rapport intermédiaire de PSC détaille nos avancées sur ces sujets, les problèmes rencontrés au cours de notre travail, et une description de nos objectifs pour la suite du PSC.

1 Travail effectué

1.1 Première approche du problème

Notre première approche du problème consiste à disposer des checkpoints sur la zone selon un ordre souhaité et à trouver des chemins entre chaque objectif. L'idée est inspirée du fonctionnement souvent adopté pour surveiller une zone : un intinéraire de patrouille entre les points sensibles.

Avec un algorithme de recherche de chemin, on cherche alors le plus court chemin entre 2 checkpoints successifs. On fusionne ensuite les tronçons et on obtient alors un premier parcours de ronde. A tout moment, si jamais un nouveau checkpoint est ajouté, le chemin est mis à jour.

Nous avons principalement utilisé cette méthode conjointement avec l'algorithme de recherche de plus court chemin A*. Les résultats sont satisfaisants : pas d'erreur sur le chemin (pas de traversée d'obstacle) et temps de calcul plus rapide que le rafraichissement de l'affichage d'une frame.

1.2 Parcourir un ensemble de checkpoints le plus rapidement possible

1.2.1 Algorithme de parcours de graphe A*

Nous avons tout d'abord cherché à traiter le problème de calculer un chemin aussi court que possible passant une fois par chacun des checkpoints et prenant en compte les obstacles. Pour ce faire, nous nous sommes assez naturellement tournés vers l'algorithme A*. Cet algorithme calcule le chemin optimal entre deux nœuds d'un graphe pondéré. La différence entre A* et l'algorithme de Dijkstra est que l'algorithme A* utilise une heuristique pour guider sa recherche vers les nœuds les plus prometteurs. L'heuristique que nous avons utilisée est la somme de la distance déja parcourue et de la distance euclidienne jusqu'au but : en effet, malgré les obstacles, il est souvent efficace de se diriger dans la direction du checkpoint recherché.

Nous avons implémenté A* en Java sur notre grille de cases, initialement en stockant les nœuds à visiter dans une simple liste chaînée, puis dans une file de priorité triée selon l'heuristique pour gagner en rapidité. Une fois notre A* fonctionnel et relativement rapide, il nous restait à résoudre un problème du voyageur de commerce pour relier nos n checkpoints de manière efficace. Hors de question de tester l'ensemble des $(n-1)!$ configurations possibles, car cela limiterait drastiquement nos options en termes de nombre de checkpoints. Nous avions donc le choix entre une méthode gloutonne comme le hill-climbing ou diverses métaheuristiques basées sur des populations, comme les algorithmes génétiques ou les algorithmes inspirés des colonies de fourmis. Nous avons choisi d'implémenter un algorithme génétique.

1.2.2 Algorithme génétique pour résoudre le TSP

La structure des algorithmes génétiques est toujours plus ou moins la même : on commence avec une population initiale de solutions générées aléatoirement. Dans notre cas, une solution est un ordre dans lequel on visite les n checkpoints avant de revenir au checkpoint initial. On exécute ensuite un certain nombres d'étapes où on conserve les meilleures solutions et on en génère de nouvelles solutions à partir de celles dont on dispose : cette phase de 'reproduction' des solutions est celle qui donne son nom à l'algorithme. Cette succession d'étapes doit en théorie permettre de converger vers l'optimum.

La principale subtilité réside dans la manière dont on fait se "reproduire" les solutions entre elles. Il faut trouver un mécanisme qui préserve dans une certaine mesure la structure des deux solutions, tout en vérifiant la condition que chaque checkpoint doit être visité exactement une fois. Nous avons choisi le mécanisme illustré ci-dessous : on coupe la première "solution parent" à une place choisie au hasard. On forme la 'solution enfant' en conservant la portion du premier parent qui se trouve avant la coupe, puis en complétant les checkpoints manquants dans l'ordre dans lequel ils apparaissent dans la deuxième solution.

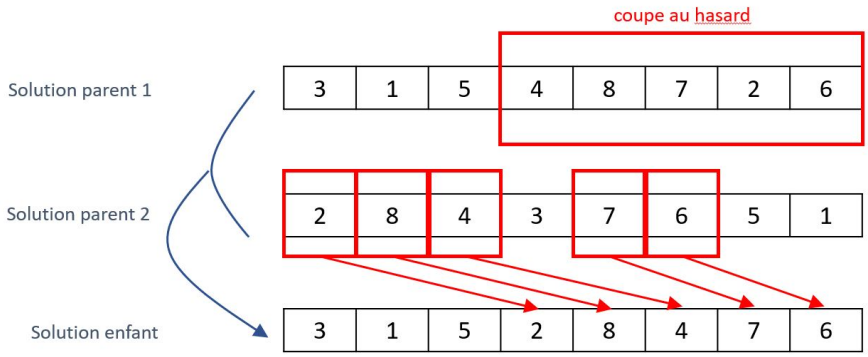

FIGURE 1 – Notre mécanisme de reproduction des solutions

Puisqu'on sélectionne à chaque étape les meilleures solutions, il est naturel de stocker la population de solutions sous la forme d'une file de priorité. Nous avons donc implémenté une file de priorité ordonnée selon la longueur totale de la solution. Pour disposer à chaque instant de celle-ci, au début de l'exécution nous exécutons $\frac{n^2}{2}$ fois l'algorithme A* pour remplir une matrice de coûts qui contient la distance à parcourir entre chaque paire de checkpoints.

1.2.3 Analyse de la performance de l'algorithme

Une fois que nous disposions de l'ensemble du back end (A* et algorithme génétique) et du front end (interface graphique), nous avons conduit de nombreux tests. D'abord des tests purement visuels (le parcours proposé semble-t-il efficace à première vue ?), qui ont permis de résoudre plusieurs bugs, et qui ont fini par donner des résultats satisfaisants. Puis des tests statistiques, à l'aide de la classe ScatterChart de JavaFX.

Nous avons codé une fonction qui génère une grille aléatoire avec un certain nombre de checkpoints et une certaine proportion d'obstacles, puis exécute l'algorithme de calcul de parcours sur cette grille avec diverses valeurs des paramètres de l'algorithme génétique : nombre d'étapes et taille de la population. Ci-dessous des tracés de la longueur de la solution trouvée, en fonction de la population à nombres d'étapes constant, puis en fonction du nombre d'étapes à population constante. On constate que la qualité de la solution s'améliore à peu près linéairement avec l'augmentation de la population, mais que l'augmentation du nombre d'étapes a un effet bien moins marqué. Ceci pourrait signifier que le mécanisme de 'reproduction' de notre algorithme génétique est peu efficace, et son amélioration pourrait être une des pistes de travail pour la suite.

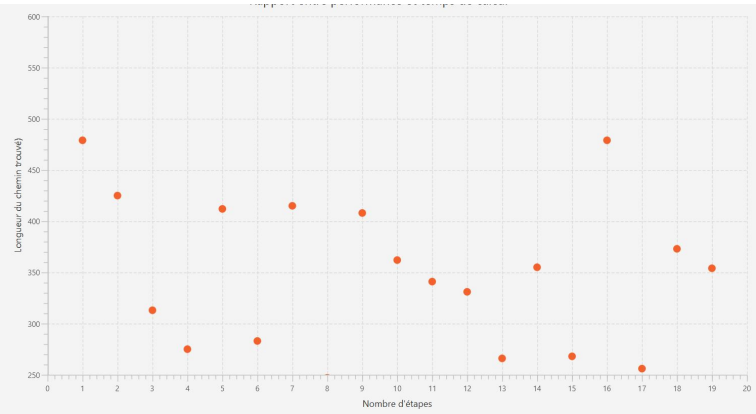

FIGURE 2 – Qualité de la solution en fonction du nombre d'étapes

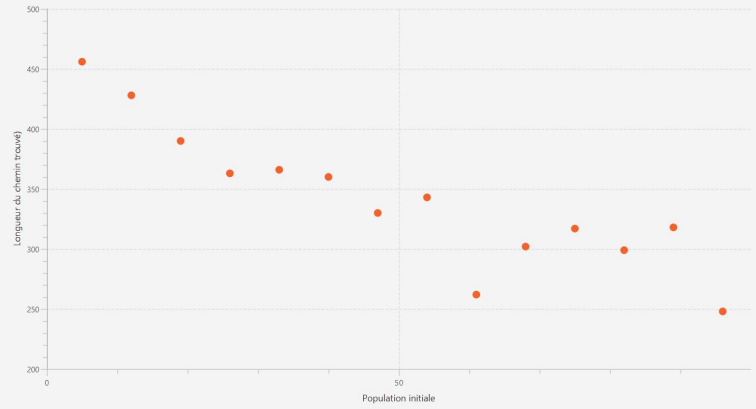

FIGURE 3 – Qualité de la solution en fonction de la population initiale

1.3 RRT

Dans l'optique de permettre aux drones de trouver un chemin entre deux checkpoints en évitant les obstacles, nous avons aussi développé un algorithme RRT (Rapidly Exploring Tree). La première version de cet algorithme est arrivée au bout d'une semaine mais comportait de nombreux points à améliorer.

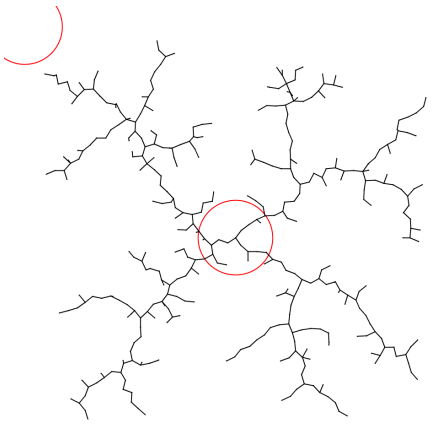

FIGURE 4 – L'algorithme RRT à l'épreuve sur une carte de 1000x1000 entre le point (500,500) et (100,900)

Nous avons ensuite voulu améliorer la vitesse d'exécution du programme ainsi que la qualité du chemin trouvé. Pour comprendre pourquoi et comment il faut d'abord comprendre comment fonctionne le RRT :

1. Tire un point P_{new} au hasard dans l'espace.
2. Trouve le point $P_{nearest}$ le plus proche de lui parmi ceux déjà dans l'arbre.
3. S'ils sont à une distance supérieure à une distance maximale d_{max} définie au départ, alors on rapproche le nouveau point tiré jusqu'à d_{max} de $P_{nearest}$.
4. Rajoute P_{new} dans l'arbre, relié à$P_{nearest}$.

Comme l'illustre la figure précédente, le RRT produit un chemin où les virages forment des angles autour de 90 degrés : on a donc un chemin peu optimisé et difficilement exécutable dans la réalité. De plus, on aimerait pouvoir avoir un algorithme qui continue à travailler pour essayer d'améliorer notre chemin. Pour pallier cela nous avons mis au point le code d'un algorithme RRT* qui reprend le principe du RRT mais ajoute simplement une étape, voici le fonctionnement :

1. Tire un point P_{new} au hasard dans l'espace.
2. Trouve le point le plus proche $P_{nearest}$ de lui parmi ceux déjà dans l'arbre.
3. S'ils sont à une distance supérieure à une distance maximale d_{max} définie au départ, alors on rapproche le nouveau point tiré jusqu'à d_{max} de $P_{nearest}$.
4. Inspecte dans un rayon r défini au départ tous les points autour de P_{new} et regarde si un d'eux pourraient le relier à la racine de l'arbre plus rapidement :
   (a) Si oui, rajoute P_{new} dans l'arbre relié au point trouvé.
   (b) Sinon, rajoute P_{new} dans l'arbre, relié à $P_{nearest}$.
5. Dans ce cercle de rayon r, si un des points pourrait passer par P_{new} pour raccourcir son chemin jusqu'à la racine alors on le fait.

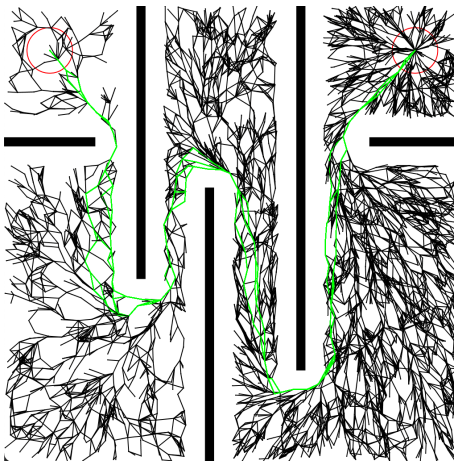

FIGURE 5 – RRT* avec obstacles, on obtient un arbre à l'aspect "filaire"

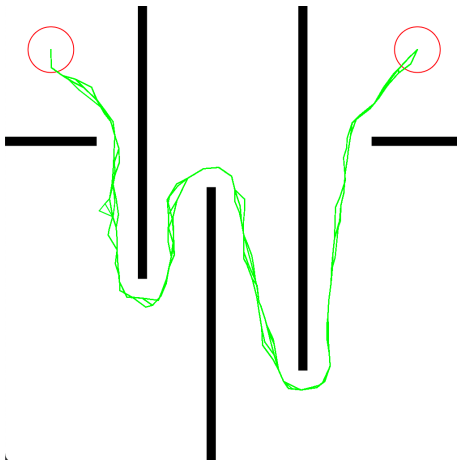

FIGURE 6 – RRT* en affichant les chemins trouvés. Le chemin est trouvé presque instantanément et des améliorations arrivent vite pour converger vers un chemin aux courbes lisses

Cette simple étape permet de « limer » l'aspect du chemin et de se débarrasser des « zigzags » du RRT.

Finalement, nous avons réalisé un RRT*-Informed qui permet d'accélérer la vitesse de convergence du programme vers une solution optimale. On gagne donc en vitesse et en qualité. Il repose sur le principe suivant : une fois qu'un chemin est trouvé entre les deux points, si ce chemin venait à être améliorer, ce ne serait que par l'ajout d'un point se situant dans une ellipse où :

— 1. Les foyers sont le point de départ et d'arrivée.
— 2. Le grand axe vaut la longueur du chemin précédemment trouvé.

Grace à cela, une fois qu'un chemin est trouvé, les points sont tirés aléatoirement dans l'ellipse et non dans l'espace entier.

1.4 Limites de ces méthodes

Notre première approche, qui consiste à désigner les zones à surveiller par un ensemble de checkpoints puis à trouver un chemin entre ces checkpoints sur lequel les drones patrouilles, à cependant des limites :

- **Limite 1 :** Elle ne prend pas en compte le nombre de drones dont on dispose. Dans le cas extrême de deux zones à surveiller, éloignées, et deux drones, elle fait patrouiller les drones entre une zone et l'autre, alors qu'il suffirait d'en laisser un pour chaque zone.
- **Limite 2 :** Elle oblige à décrire la zone à surveiller de manière discrète : checkpoint ou pas checkpoint. Or les cas d'utilisation dans la vie réelle correspondent plus à des zones que des points. On peut certes décrire une zone en la remplissant de beaucoup de checkpoints, mais cela augmente considérablement la complexité du problème alors qu'il n'est pas fondamentalement plus "dur" qu'un problème avec un point par zone.
- **Limite 3 :** Elle oblige à décrire la zone à surveiller de manière binaire : à surveiller ou ne pas surveiller. Or une zone peut avoir plus besoin de surveillance qu'une deuxième, moins qu'une troisième… trouver un chemin entre les checkpoints ne prend pas en compte cette subtilité.

Face à ces limites, nous avons envisagé deux nouvelles méthodes de surveillance que sont la méthode des potentiels et la méthode de surveillance par conflit. L'une est à un stade suffisant pour être présentée ici, l'autre fera l'objet d'une description dans la troisième partie.

1.5 Méthode de surveillance par conflit

1.5.1 Première approche

Cette méthode vise à supprimer les limites précédemment identifiées. Elle se base pourtant sur une description similaire du problème, à savoir un ensemble de checkpoints qui désigne les points à surveiller. A un checkpoint C_i, on associe la fonction d'insatisfaction :

$$f_i : t \to f_i(t) \tag{1}$$

Intuitivement, f_i est croissante. Chaque checkpoint peut avoir sa propre fonction d'insatisfaction, ce qui permet de mieux prendre en compte le besoin en surveillance de chaque zone. Cela permet de contrecarrer la **limite 3**.

Face à ces checkpoints, chaque drone D_j prend sa décision, en fonction de l'état des checkpoints et de sa position : il assigne à chaque checkpoint un score $S_{i,j}$:

$$S_{i,j}(t) = g\left[d(C_i, D_j), f_i(t)\right] \tag{2}$$

Il choisit ensuite le checkpoint qui à le plus grand score et se dirige vers ce checkpoint.

Cette solution est assez naïve, et soulève immédiatement un nouveau problème : les drones ne se coordonnent pas et peuvent tout à fait se retrouver à faire chacun exactement la même chose. Pour les coordonnéer, nous faisons l'hypothèse (pas très contraignante) **H** que nous avons plus de checkpoints que de drones et nous adoptons le comportement suivant :

1. Chaque drone classe tous les checkpoints par intérêt décroissant
2. Chaque drone entre dans le processus de décision :
   (a) D_i choisit le checkpoint en première position dans sa liste (existe grâce à **H**).
   (b) Si le checkpoint n'est pas encore choisi : D_i choisit ce checkpoint
   (c) Si $D_{i'}$ a déjà choisi ce checkpoint :
   - Si D_i est plus près que $D\prime_i$, D_i choisit ce checkpoint, $D_{i'}$ retire le checkpoint de sa liste et ré-entre dans le processus de décision
   - sinon D_i retire le checkpoint de sa liste et entre dans le processus de décision

Pour montrer la terminaison de l'algorithme, posons n_i le nombre de checkpoints que D_i peut choisir et m le nombre de drones qui ont fait une décision. On a alors :

$$\sum_{\forall i} n_i + m \tag{3}$$

qui est une quantité positive strictement décroissante à chaque passage d'un dronedans le processus d'un drone. Nécessairement, donc, l'algorithme termine.

L'ordre dans lequel les drones prennent leurs décisions semble avoir une influence sur l'attribution finale, mais n'en à en réalité pas. On n'a donc pas à s'inquiéter de l'ordre dans lequel on fait prendre aux drones les décisions. Cependant, il est possible qu'un ordre judicieusement choisi limite le nombre de conflits et donc le temps de calcul pour faire une étape $t \to t+1$, nous nous intéresserons à cette question pour la suite du projet.

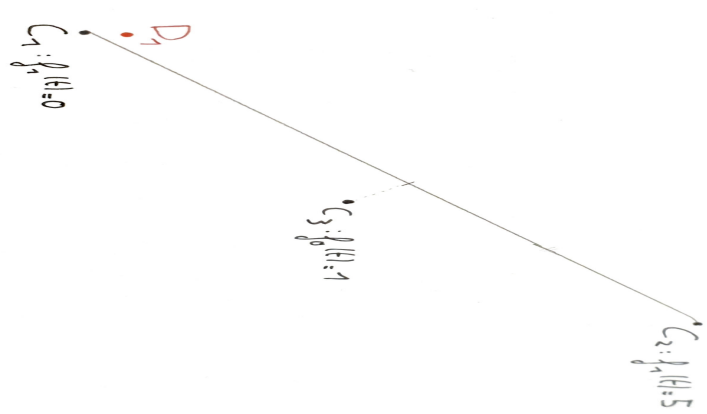

FIGURE 7 – Situation problématique

1.5.2 Approfondissement

Lors de notre première implémentation, nous avions fait la décision suivante : une fois qu'un drone à choisi sa destination, et résolu tous les conflits, il ne fait sa prochaine décision que lorsqu'il arrive à destination (ou lorsqu'un autre drone entre en conflit avec lui et le bat). Cela permet de faire des décisions plus rarement, donc d'augmenter la performance, et traduit intuitivement le fait que si un drone à choisi une cible, il va s'en approcher et donc ne fera qu'augmenter l'intérêt qu'il aurait à choisir cette destination à nouveau, si on lui demandait de faire une décision.

Cependant, cette décision s'avère mauvaise face à un certain type de situation comme en figure 7. Le drone se dirige vers C_2 alors que le fait de d'abord passer en C_3 ne lui fait perdre que très peu de temps, avant d'aller à C_2.

Nous sommes donc revenus en arrière, et fait prendre à chaque drone une décision à chaque instant. Dans la situation précédente, le drone ferait finalement le détour une fois qu'il se serait plus approché de C_3. Mais cela impose de faire un grand nombre de décisions. De plus, dans certaines situations, un drone peut se retrouver à osciller au milieu d'un ensemble de checkpoints qu'il se fait systématiquement "voler" lors d'un conflit avant qu'il ne puisse y arriver.

Une solution, qui permettrait d'éviter les deux problèmes, nous à été suggérée par notre tuteur. Dans une situation comme en figure 8

Le drone choisit le checkpoint dans l'ellipse qui est le plus proche de lui. C'est une manière de l'autoriser à faire un petit détour (donc éviter le problème 1), tout en conservant l'idée de mettre à jour plus rarement (donc éviter le problème 2). Nous essayerons de l'implémenter.

Dans une version optimisée, la complexité temporelle en le nombre de checkpoints est faible, donc on peut décrire une zone, continue, en plaçant beaucoup de checkpoints, discrets, sans pénaliser trop la résolution. Cette méthode évite donc partiellement la limite 2.

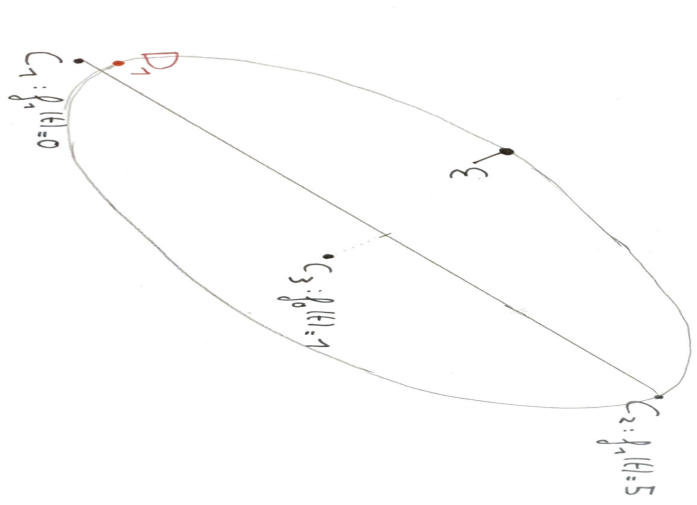

Figure 8 – Même situation avec l'ellipse

2 Organisation du travail et problèmes rencontrés

2.1 Notre organisation

Surveiller une zone par un réseau de drones est un problème ouvert et large. Ouvert car il n'existe pas de critère objectif pour caractériser la qualité d'une solution. Large puisqu'il inclus un certain nombre de phases :

- Faire patrouiller les drones dans une zone
- Détecter les menaces
- Intercepter les menaces
- ...

Sur conseil de notre coordinateur, nous avons décidé de nous concentrer sur la première phase. Cela permet d'avoir un sujet plus spécifique, et de prendre moins de risques de "hors sujet".

Cela devrait aussi nous permettre d'être créatifs dans les solutions envisagées et les implémentations possibles. Certains thèmes et certaines structures viennent souvent à l'esprit, comme les algorithmes de path-finding, ou le système de checkpoints. Mais nous essayons de ne pas nous y limiter (nous détaillerons ce risque dans la section suivante) et de chercher des solutions plus originales. Les solutions développées en partie 3 en témoignent.

Cela se traduit concrètement dans notre organisation de travail. Nous nous réunissons pour échanger sur les idées qui ont émérgé, ou que nous sommes en train d'étudier, à 2, à 3 ou à 4. Cela nous permet d'élargir notre pensée. En contrepartie, le travail de code et d'implémentation est essentiellement individuel.

Nous utilisons le créneau de notre emploi du temps du mercredi pour une réunion à 4 pour contrôler l'avancement des projets en cours, proposer des corrections pour ceux qui sont bloqués et suggérer de nouvelles idées.

Avec cette forme d'organisation, il nous faut être attentif au fait de ne pas laisser tomber les projets en cours de route dès lors qu'une nouvelle idée, qui semble toujours meilleure qu'elle n'est vraiment, est proposée par l'un ou l'autre.

Notre travail avance à un rythme qui nous convient et semble convenir à notre tuteur et notre coordinateur, nous n'avons donc pas prévu de changer d'organisation pour la suite du projet. Nous continuerons à régulièrement faire des comptes-rendus à notre coordinateur et à soliciter l'aide de notre tuteur.

2.2 Directions prises et corrections

Au cours de ce projet, nous avons pu identifier un problème lié au sujet lui-même. Lorsque nous avons défini l'objectif, nous avons donné deux exigences relativement indépendantes :

— Attribuer aux drones un chemin pour surveiller une zone de manière la plus efficace possible.
— La zone peut contenir des obstacles que les drones doivent éviter.

La première exigence est l'objectif principal de notre PSC. La deuxième est plus accessoire, bien que relativement intuitive.

La décision d'ajouter cette deuxième nous vient sans doute du groupe de l'année dernière dont nous reprenons le sujet. Ils avaient basé leur recherches sur l'algorithme A* et travaillé essentiellement sur cette question des obstacles. Le problème est que cette deuxième exigence a beaucoup orienté notre travail dans un premier temps (développement de A* et ses variantes, RRT et ses variantes, structures de données optimisées pour la recherche de chemin…). Nous avons donc moins orienté nos recherches vers l'objectif principal que nous aurions dû. Le temps passé sur la recherche d'un unique chemin de patrouille pour tous les drones, qui évite les obstacles, n'a pas permis de progresser sur la recherche de solutions de coordination entre les drones. C'est donc seulement dans un second temps que nous avons commencé à nous orienter vers ce thème.

Néanmoins, les quelques algorithmes de pathfinding que nous avons développés semblent pouvoir traiter à peu près toutes les configurations possibles de recherche de chemin (continu, discret, forme des obstacles…), avec des performances acceptables. Nous chercherons bien sûr à les améliorer, mais nous pouvons maintenant les utiliser comme outils, et nous avons moins à nous soucier du problème des obstacles. Nous espérons que le temps "perdu" au départ sera "gagné" pour la suite.

2.3 La coordination dans un projet informatique

Chacun d'entre nous avait déjà eu l'occasion de mener un projet informatique, que ce soit sous forme d'un TD dans le cadre d'un cours (INF361, INF371, INF411, MAP433…), un TIPE en classes préparatoires, ou autre… Mais nous n'avions pas l'expérience de développement informatique en groupe sur un projet unique. Rapidement, des problèmes de coordination sont arrivés. Les différentes versions des codes ne sont plus les mêmes, certaines ont des fonctionnalités que d'autres n'ont pas, les versions ne sont pas compatibles entre elles, les échanges par mail des nouveaux fichiers nous amènent à confondre les versions et en conserver des obsolètes, le nombre de fichiers Java à aussi beaucoup augmenté.

Les problèmes de coordination arrivent également sur des aspects de contenu : quand un des membres du groupe développe une idée et l'implémente, les autres, qui n'ont pas forcément suivi à la lettre son travail peuvent ne pas comprendre l'implémentation réalisée. Il est donc souvent difficile de s'entraider sur des problèmes techniques d'implémentation.

Enfin, en menant un projet sur un sujet ouvert ou les idées arrivent le plus souvent en cours de route, nous avons eu souvent l'occasion de modifier un code existant pour y incorporer une nouvelle fonctionnalité. Mais si celle-ci s'avère ne pas être bonne, ou ne pas fonctionner, le fait de revenir en arrière prend beaucoup de temps, et, souvent, ne ramène pas au point de départ. Nous avons donc perdu beaucoup de temps à refaire fonctionner des codes qui fonctionnaient auparavant, ce qui est assez frustrant et pas très efficace.

Nous avons donc dû gagner en rigueur dans la coordination du travail, la gestion des fichiers, et la communication interne. Récemment, nous avons appris à utiliser le gestionnaire de versions Git, via GitHub, et nous espérons que cela résoudra certains des problèmes rencontrés.

3 Projets pour la suite

3.1 Comparaison des méthodes

Nous disposons désormais de divers algorithmes fonctionnels qui permettent de répartir les drones sur la carte, et d'un certain nombre d'idées à implémenter. Il s'agit maintenant de créer des méthodes de scoring qui soient valables pour nos différents algorithmes pour pouvoir les comparer entre eux. **Tester et comparer sera vraisemblablement un des prochains grands chantiers de notre projet.**

Les critères utilisées par ces méthodes sont subjectifs et peuvent avantager un algorithme ou un autre. Nous devrons donc faire attention à ne pas fixer les critères en fonction des solutions trouvées, mais bien l'inverse ! Cela nécessitera de prendre du recul sur le projet et peut-être explorer de nouvelles pistes.

3.2 TRA*

Les variantes de A* comportent un défaut majeur : elles fonctionnent sur un espace discrétisé. Avec une discrétisation assez fine, le graphe considéré par l'algorithme A* est très grand. On n'exploite pas la structure de l'espace qui pourrait simplifier le problème, par exemple en fusionnant les carrés voisins qui ne contiennent pas d'obstacle. Le graphe est alors réduit jusqu'à contenir exactement l'information de l'espace comme en figure 9.

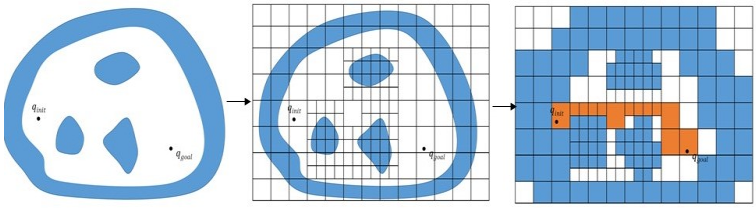

FIGURE 9 – A* sur un graphe optimisé

En poussant cette idée on arrive à l'algorithme de TRA* (TRiangulated A*) qui triangule l'espace pour en faire un graphe de mouvements possibles qui contient la stricte information nécessaire. Par la même occasion, on s'affranchit de la discrétisation du A* classique : on discrétise l'espace en fonction des obstacles et plus l'inverse.

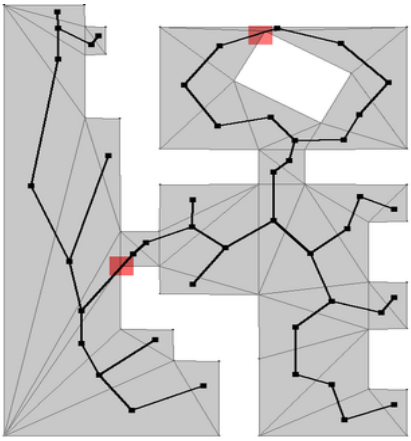

FIGURE 10 – Graphe utilisé par le TRA*

Une nouvelle contrainte apparaît : la distance entre les éléments de graphe n'est pas fixe. On peut passer par un triangle de plusieurs manières, et la distance que l'on à en parcourant ce triangle dépend du point par lequel on entre, et du point par lequel on sort. La calculer au cours des itérations de A* (notament avec l'algorithme du funnel) est assez compliqué, et nous n'avons pas réussi pour le moment. Nous souhaitons implémenter cette métode dans la suite du projet pour bénéficier de la meilleure version possible de l'agorithme A*. Notre tuteur nous conseillera sur ce point lors d'une visioconférence début février.

Notre version actuelle donne des résultats, comme en figure 11, il n'est pas toujours bon, que ce soit en termes de qualité (chemin non optimal) et de validité (n'évite pas les obstacles).

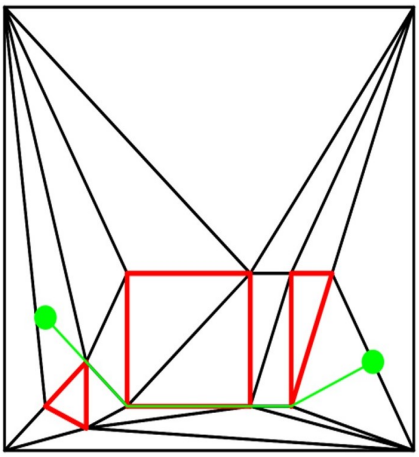

FIGURE 11 – Notre version de TRA*, quand elle marche

3.3 La méthode des potentiels

La méthode des potentiels est une méthode qui vise également à éviter les trois limites identifiées. Nous avons commencé à l'implémenter (visuel d'un exemple en figure 12) mais elle est encore à un stade peu avancé. Nous souhaitons nous concentrer dessus pour la suite de notre travail.

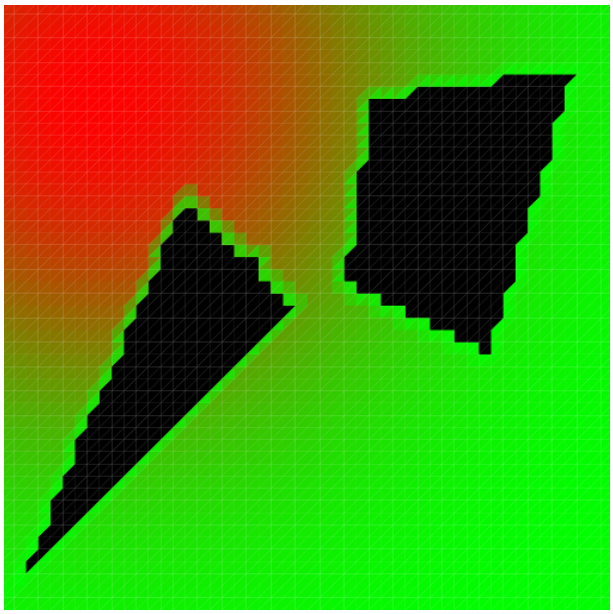

FIGURE 12 – Potentiel en nuances de couleurs, deux obstacles, une zone à surveiller

Dans cette méthode, la situation globale à un instant t est modélisée par un potentiel défini sur l'espace ou évoluent les drones. Le potentiel est une sorte de prolongement par continuité de la fonction d'insatisfaction de la méthode de recherche par conflit. La valeur du potentiel en un point traduit l'intérêt que ce point aurait à être visité. Pour des raisons de cohérence avec la physique (d'où nous tirons le mot potentiel), un potentiel bas traduit un

grand intérêt. Inversement, la valeur du potentiel en un obstacle est infiniment grande, ou du moins plus grande que celle de n'importe quel autre endroit de la carte ou un drone peut aller.

Avec cette carte, chaque drone n'a qu'à suivre le gradient du potentiel en sa position (ou plutôt son opposé), pour se diriger naturellement vers les zones de grand intérêt. Connaissant le potentiel, chaque drone peut donc prendre sa décision de manière indépendante, et très rapide.

Pour implémenter l'algorithme, nous devons discrétiser le potentiel. On le définit donc sur une grille qui couvre la zone étudiée. Le problème qui se pose avec un potentiel discret est le calcul du gradient en une position quelquonque. En s'inspirant du monde du jeu vidéo dans lequel les surfaces sont générées par des triangles, nous avons triangulé la grille (chaque carré de la grille est divisé en deux triangles en le coupant selon une des deux diagonales). Quelles que soient les valeur du potentiel aux sommets d'un triangle, on peut toujours définir le gradient sur ce triangle, ce qu'on n'aurait pas pu faire en gardant des carrés.

Telle quelle, la méthode est inutilisable puisque chaque drone se dirige vers un minimum local de potentiel, donc une zone d'intérêt, et y reste. Il faut donc traduire le fait qu'un drone à surveillé une zone, en réhaussant le potentiel de la zone, pour décourager les drones de revenir en dans cette zone immédiatement. La difficulté, mais aussi la flexibilité de la méthode, se trouve dans cette mise à jour du potentiel à chaque itération. Elle impose deux exigences, liées :

— Chaque drone se déplaçant de manière indépendante, ils n'ont à priori aucune raison de se coordonner. Pour que ce soit le cas, il faut augmenter le potentiel sur toute la carte, pour que tous les drones soient « informés » de la présence des autres. Mais un problème se pose : en augmentant le potentiel là ou il passe, le drone risque d'aller en ligne droite en "surfant sur la vague" de potentiel qu'il crée lui-même. Il faut donc que les variations de potentiel dues à l'importance variable des zones soient plus importantes que celles dues à la vague que le drone crée. Mais si cette variation est trop grande c'est la coordination des drones qui est pénalisée et chacun est attiré par les zones les plus importantes indépendamment des positions des autres. Il faut donc trouver un bon équilibre pour mettre à jour le potentiel.
— Comme pour la méthode par conflit, une zone pas ou peu surveillée doit augmenter son insatisfaction (donc son potentiel), pour attirer les drones. Mais l'augmentation du potentiel dépend de l'intérêt de la zone, puisqu'une zone très importante exigera d'être visitée plus souvent qu'une zone moins importante.

Mettre à jour le potentiel en évitant ces deux écueils a posé beaucoup de problèmes d'ordres technique pas encore surmontés. Nous allons donc essayer de faire fonctionner cette méthode en retravaillant la mise à jour du potentiel.

3.4 Méthode par territoires d'animaux

3.4.1 Modèle

Lors de notre première visioconférence, notre tuteur chez MBDA Vincent Jeaunau nous avait encouragé à chercher de nouvelles idées à développer sur le projet. Nous avons pensé à nous inspirer du comportement animal et de se rapprocher du modèle de « territoires » que chaque agent s'octroie et adapte en fonction de la présente ou non d'autres agents aux alentours.

Nous avons donc codé un modèle où un nombre d'individus appelés « walkers » (marcheurs) se déplacent sur dans un espace en laissant des traces de leur passage. Ces traces durent un certain temps noté TAS pour « active scent time » (temps d'activité olfactive) au bout duquel les autres « walkers » peuvent à nouveau passer.

Ce modèle permet d'avoir un équilibre naturel des agents dans l'espace sans laisser de zones libres. En effet, de d'autres modèles de modélisation des territoires animaux permettent l'apparition de « no man's land » qui seraient une faille dans notre système de défense.

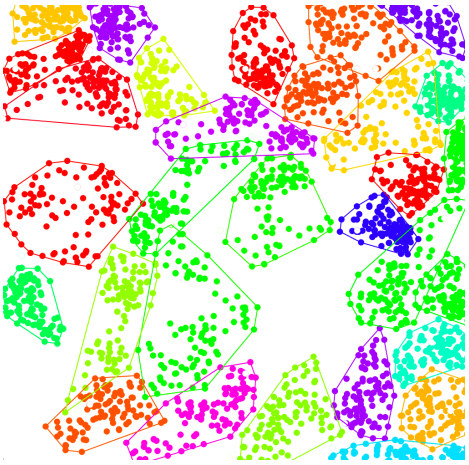

FIGURE 13 – Modélisation des territoires de 30 walkers sur une carte de taille 100

Cette partie de notre travail est très récente et ouvre la voie à de nombreuses possibilités d'utilisation puisque ce modèle brut est assez limité.

3.4.2 Diagrammes de Laguerre-Voronoï

Pour exploiter le modèles des territoires animaux, nous pensons développer un générateur de diagramme de Voronoï-Laguerre pour pouvoir donner un « poids » à chaque checkpoint en fonction de leur importance. Le modèle des territoires animaux servirait à donner à chaque drone son territoire dans lequel évoluer. Ils se déplaceraient en utilisant un des algorithmes de générateur de chemin entre deux points.

Nous voudrions pouvoir associer deux drones sur une même zone pour travailler en équipe, la construction du programme de territoires le permettrait facilement. Dans l'élaboration de nos différents codes, nous avons déjà développé certains algorithmes de triangularisation de Delaunay et d'enveloppe convexe qui faciliteraient notre travail sur les diagrammes de Voronoï-Laguerre.

Une fois générés, nous aurions un outil puissant et facilement modulable pour :

- 1. Attribuer des zones de défenses
  - a. Modulables
  - b. Classées par ordre d'importance
  - c. Naturellement définies en fonction des autres drones et checkpoints
- 2. Gérer des drones
  - a. Autonomes dans leurs déplacements malgré les obstacles
  - b. Avec la possibilité de les faire collaborer

Conclusion

Les résultats obtenus à mi-parcours sont encourageants : 2 interfaces graphiques utilisables, 4-5 algorithmes de tracés de chemin fonctionnels, des approches différentes (territoires d'animaux, potentiels, resolution de conflits, tracés puis suivi de parcours) sont en cours d'approfondissement ...

L'enjeu est désormais de finaliser ces différentes approches pour pouvoir les comparer sur des situations variées selon des critères qui restent à définir.

Dans tous les cas, nous tenons à remercier Vincent pour ses précieux conseils qui ont permis d'orienter nos recherches, d'accélerer la résolution de nos problèmes et de se pencher sur de nouvelles approches moins conventionnelles.