Eirbot
Coupe de France de robotique

Equipe Eirboat

1A 2019-2020

Table des matières

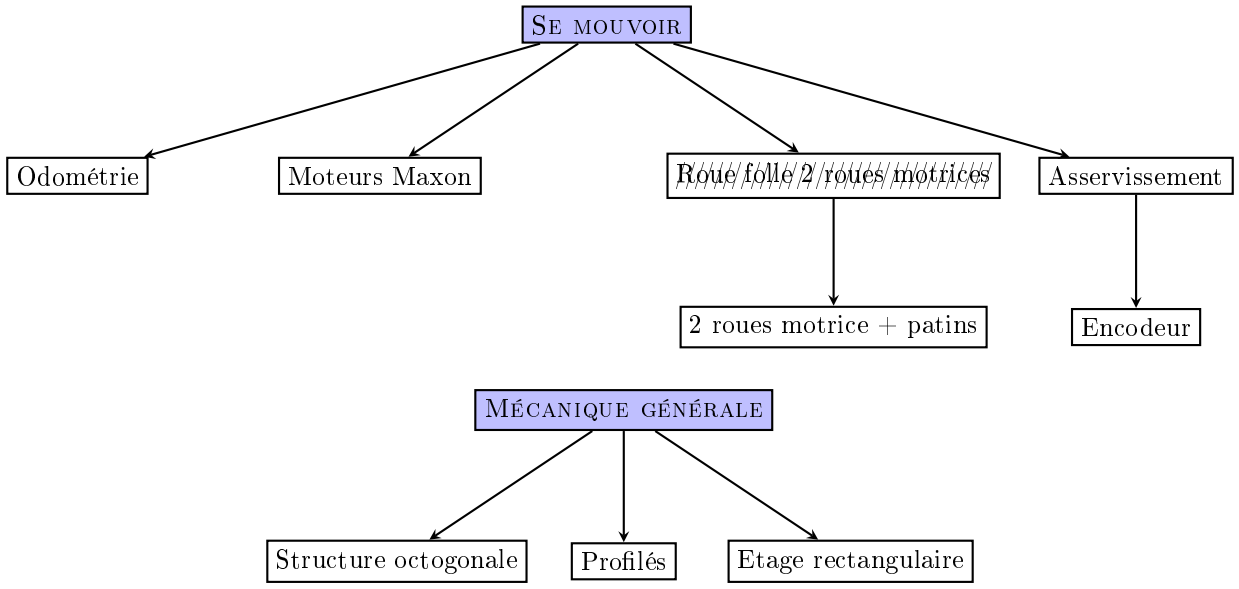
Deuxième partie

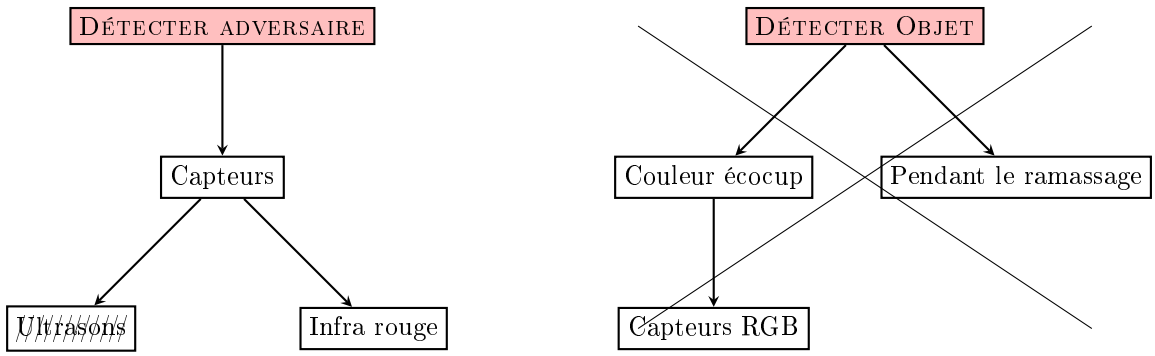
Description des projets

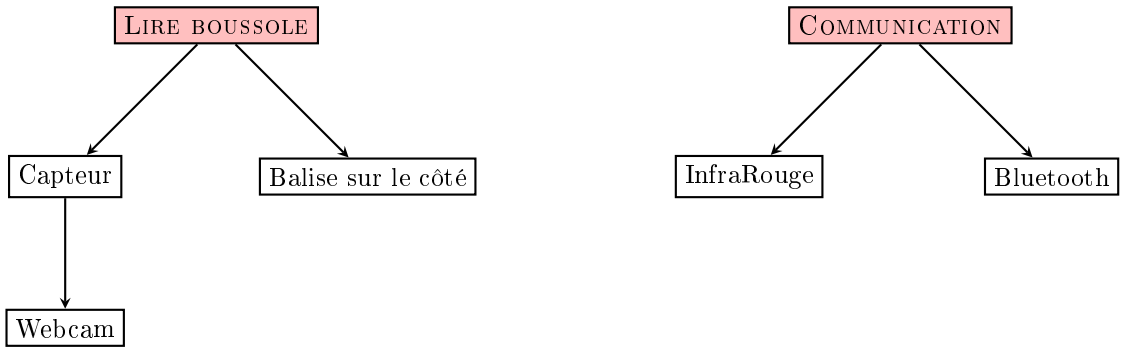
Chapitre 1

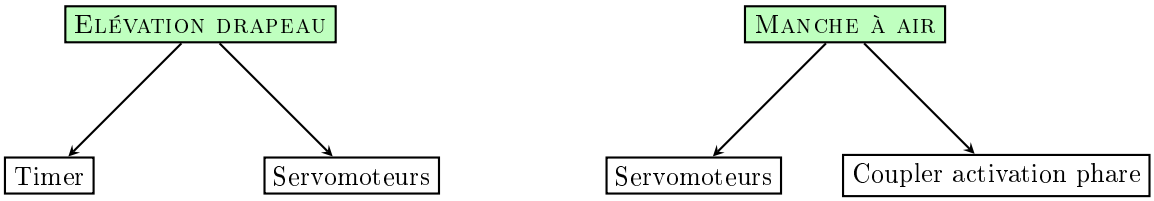
Description générale de l'organisation

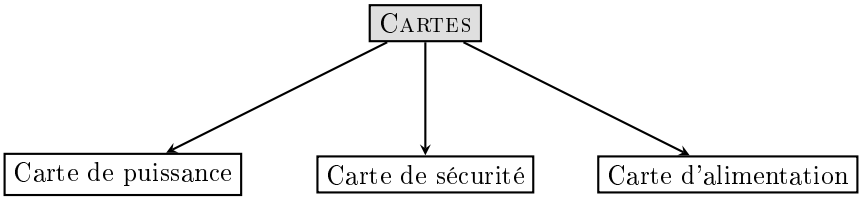
1.1 Arbres des tâches à réaliser par le robot

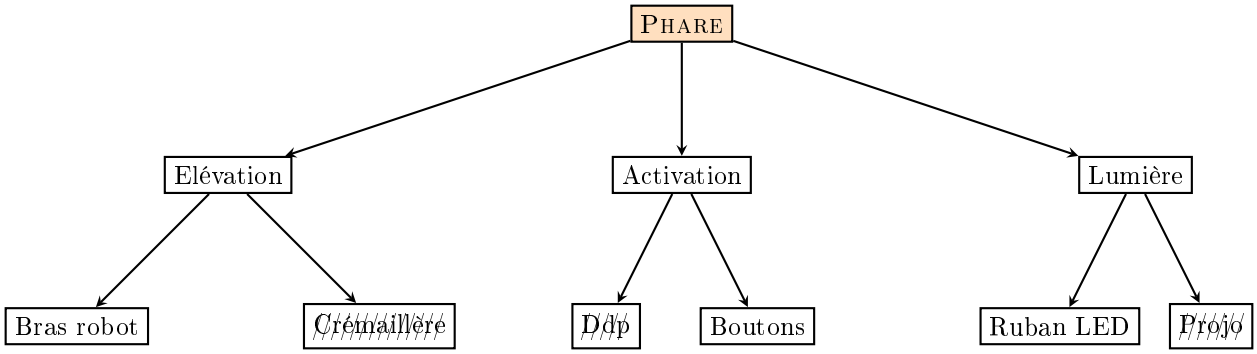




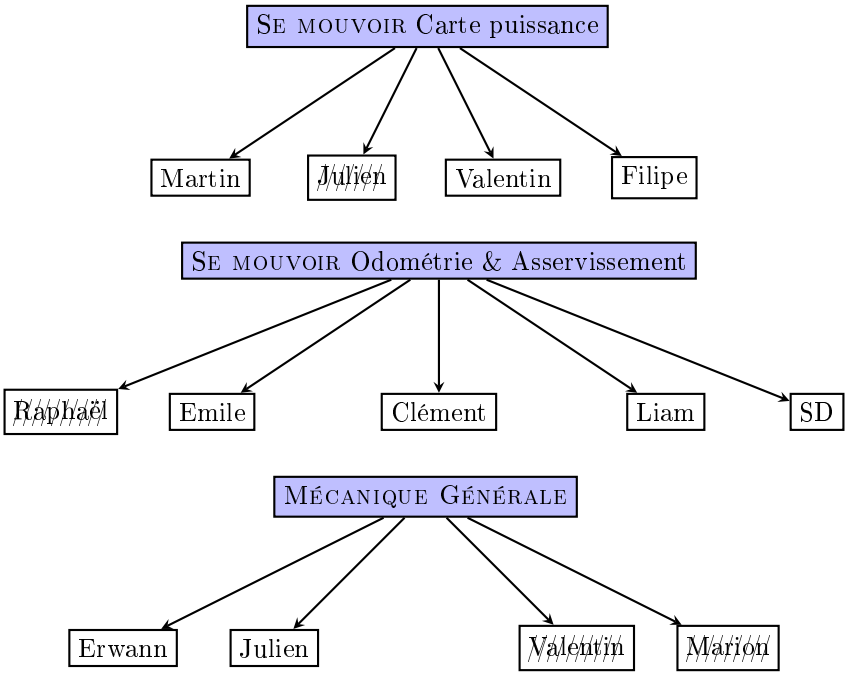

Lire boussole
Capteur
Balise sur le côté
Webcam
Communication
InfraRouge
Bluetooth

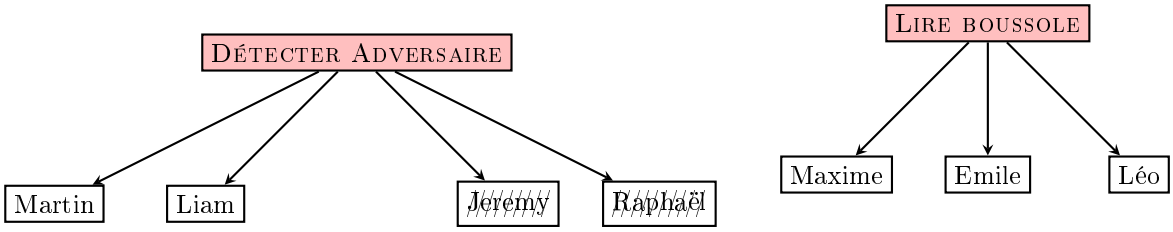

Elévation drapeau
Timer
Servomoteurs
Manche à air
Servomoteurs
Coupler activation phare

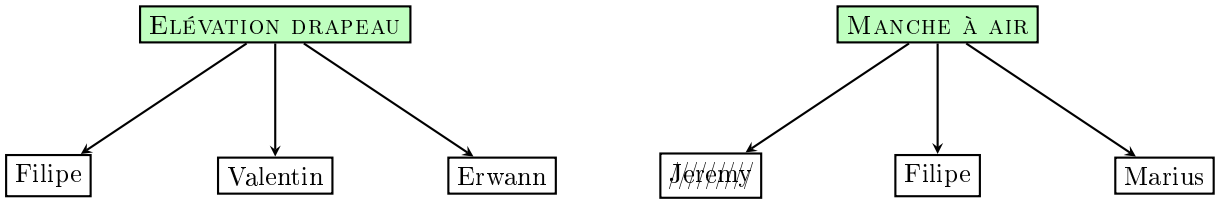

Cartes
Carte de puissance
Carte de sécurité
Carte d'alimentation

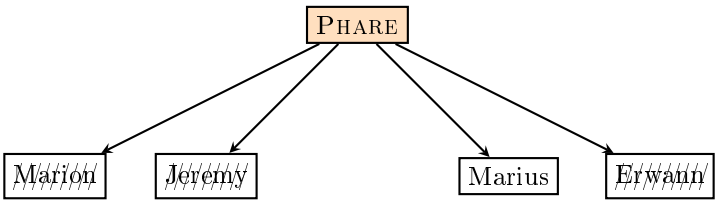

Phare
Elévation
Activation
Lumière
Bras robot
Crémaillère
Ddp
Boutons
Ruban LED
Projo

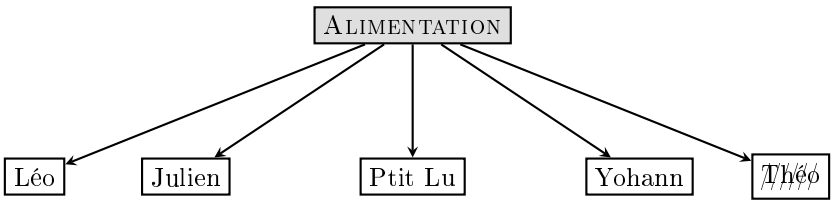
1.2 Répartition des tâches



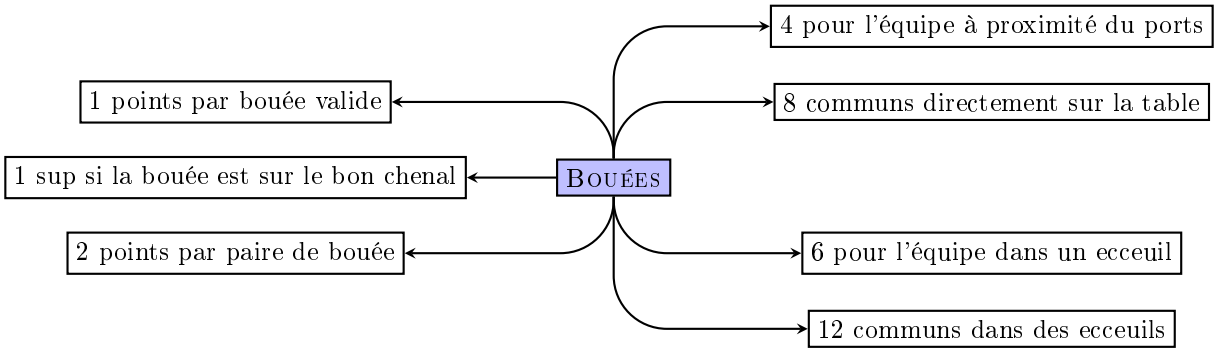


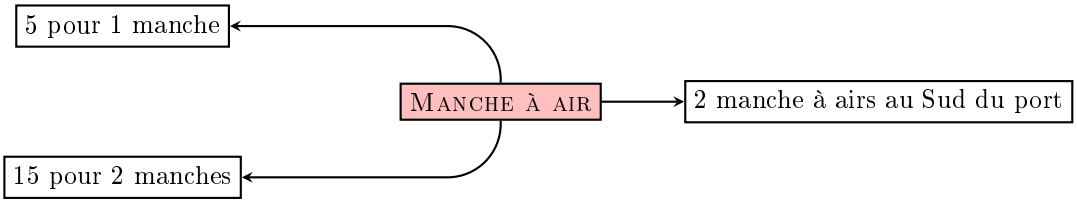


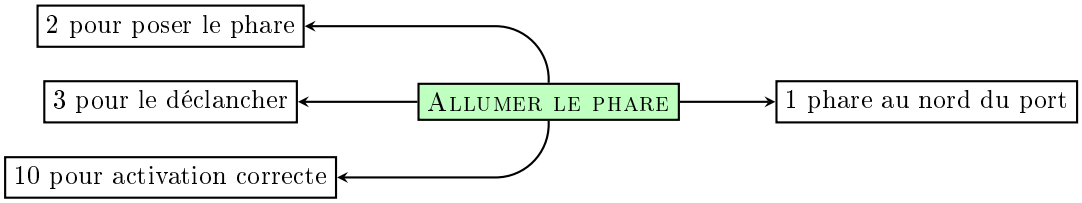


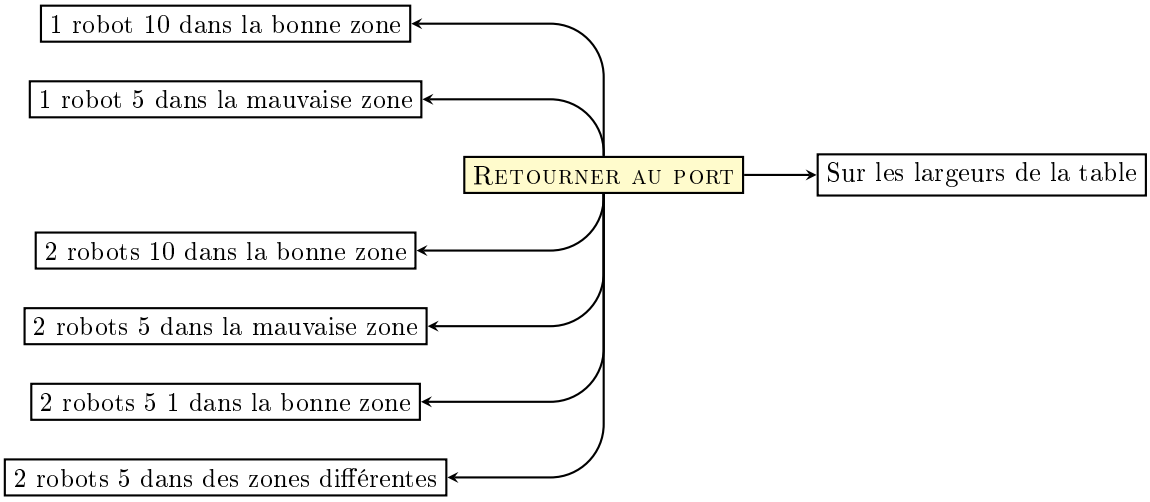

Alimentation
Léo
Julien
Ptit Lu
Yohann
Théo

1.3 Points pour la coupe









Chapitre 2

Mécanique

2.1 Mécanique générale du robot

2.2 Actionneurs

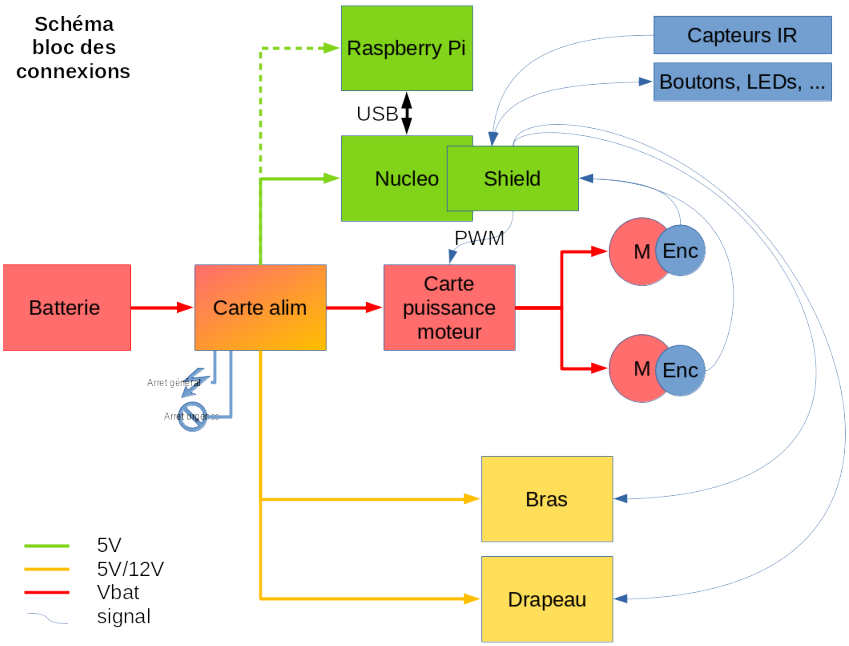
Chapitre 3

Electronique

3.1 Alimentation

Objectif. La carte d'alimentation doit distribuer l'énergie et convertir les tensions. Elle récupère l'énergie de la batterie et elle possède les caractéristiques suivantes :

- Moteurs : 3A/moteurs
- Logique 5V 1.5A
- Puissance 5V 3A
- Puissance 12V 3A



Les détails techniques sont disponibles dans le dossier alimentation sur github.

3.2 Puissance

3.3 Actionneur

Chapitre 4

Informatique

4.1 Asservissement

4.2 Stratégie

4.2.1 Modélisation de la table - Définition des obstacles

Objectif. Cette section de la stratégie est la première section, elle a pour but de définir l'environnement dans lequel le robot va évoluer. Cette définition de l'environnement sera cruciale pour la recherche de chemin. L'objectif est donc de créer un système permettant de définir des obstacles et des méthodes permettant de verifier si il y a des obstacles à un endroit.

Définition d'un obstacle. La structure permettant de définir un obstacle est relativement simple, elle contient la position du centre d'un obstacle, sa largeur et sa longueur. Avec cette définition tous nos obstacles sont rectangulaires. Par la suite, nous avons renseigné les positions de tous les obstacles et leurs types (eco cup, petit taquet, grand taquet). Nous mettons toutes ces informations dans un vecteur. Ce vecteur sera l'élément de base pour détecter les collisions entre notre robot et les obstacles.

Prévision de collision. L'idée est d'obtenir une méthode permettant de savoir si notre robot peut aller à un point donné sans toucher d'obstacles. Nous réalisons un test mathématique simple qui permet de nous donner en fonction de coordonées x, y, d'une forme d'obstacle et d'une liste contenant tous les obstacles si la case est valide ou non. Le fait de pouvoir tester les différentes formes d'obstacles nous permet dans des cas critiques de désactiver les tests sur les objets qui sont mobiles et ainsi nous sortir d'une situation délicate.

En somme nous disposons maintenant d'une modélisation simple de la table ce qui nous permettra de pouvoir calculer les déplacements du robot tout en évitant les obstacles. Nous ajoutons aussi une fonctionalité permettant de ne pas prendre en compte toutes les obstacles ce qui nous permet de prioriser l'évitemment de certains obstacles (nous préférons rentrer en collision avec une eco_cup qu'avec un robot).

4.2.2 Recherche de Chemin - Algorithme A∗

Objectif. L'idée est d'implémenter un algorithme permettant de calculer un chemin pour notre robot. L'idée est qu'il puisse, d'un point donné aller à un autre point tout en évitant les obstacles. Ces obstacles seront soit défini directement (comme les gobelets) ou défini en fonction de la détection des robots adverses. Nous réutilisons la classe *world* pour l'implémentation des obstacles.

Description de l'algorithme de pathfinding A*. A* [1] commence à un nœud choisi. Il applique à ce dernier un cout initial, il estime ensuite la distance entre ce noeud et le but à atteindre. Le coût additionné à l'évaluation représentent le *cout euclidien* assignné au chemin menant à ce noeud. Le noeud est alors ajouté à une file d'attente prioritaire, appelée *open list*.

Premièrement l'algorithme récupère le premier noeud de l'*open list*. Si elle est vide, il n'y a aucun chemin du noeud initial à celui d'arrivé, l'algorithme est en erreur. Si le noeud est celui d'arrivé, l'algorithme va reconstruire [2] le chemin complet et renvoyer le résultat.

Ensuite, si le noeud n'est pas le noeud d'arrivée alors de nouveaux noeuds sont crées pour tous les noeuds contigus admissibles [3]. L'A∗ calcule ensuite son coût et le stocke avec le noeud. Ce coût est calculé à partir de la somme du coût de son ancêtre et du coût de l'opération pour atteindre ce nouveau noeud.

En parallèle l'algorithme conserve la liste des noeuds qui ont été vérifiés, c'est la *closed list*. Si un noeud nouvellement produit est déjà dans cette liste avec un coût égal ou inférieur, on ne fait rien.

Après, l'évaluation de la distance du nouveau noeud au noeud d'arrivée est ajoutée au coût pour former l'heuristique du noeud. Ce noeud est alors ajouté à la liste d'attente prioritaire, à moins qu'un noeud identique dans cette liste ne possède déjà une heuristique inférieure ou égale.

Une fois ces étapes effectuées pour chaque nouveau noeud contigu, le noeud original pris de la file d'attente prioritaire est ajouté à la liste des noeuds vérifiés. Le prochain noeud est alors retiré de la file d'attente prioritaire et le processus recommence.

Mise en oeuvre. Les détails techniques sont disponibles dans le dossier code, fichier navigation.
Reprennons l'idée générale de cette partie : implémentation d'une recheche de chemin. L'algorithme qui au coeur de cette recherche de chemin à déjà été présenté. Nous allons donc décrire la classe *Navigation*, le but de cette classe est de récupérer une destination souhaitée, de trouver le chemin (tout en évitant les obstacles) via l'A* et enfin d'effectuer les déplacements en appelant la classe d'Asservissement via le protocole de communication. Cette classe aura donc des noeuds comme attribut, (un noeud contient la position x, y, les différents cout $fcost, gcost, hcost$ et les parents du noeud $parentX, parentY$). Elle possède 3 méthodes principales, tout d'abord Astar qui permet d'executer l'Astar, cette méthode appelle MakePath qui a pour but de construire un vecteur contenant le chemin final. Cepedant à ce stade nous avons une description du chemin très précise (centimètre par centimètre), nous devons fournir quelque chose de moins précis à l'asservissement (au risque d'avoir un papybot). Pour cela nous avons la méthode NavigateToAsserv qui permet de transformer ce chemin en suite de segment ce qui nous permet de nous déplacer.

1. https://fr.wikipedia.org/wiki/Algorithme_A*
2. Cette reconstruction se fait grâce aux informations de la *closed list*
3. Dans notre cas si il est libre ou non

A ce stade nous avons un moyen de trouver notre chemin et faire deplacer un robot de manière assez brutale, il faudrait développer un algorithme de lissage de la trajectoire (pour avoir des changements de direction moins brutaux).

4.2.3 Détection des adversaires

4.2.4 Gestion des actionneurs

4.2.5 Boucle de jeu principale

4.2.6 Mise en place de tests

4.3 Protocole de communication Stratégie → Asservissement