

Contexte

- Robotic System est une association familiale, ayant pour objectif de participer aux compétitions de robotique
 - L'équipe a participé aux coupes de France, de Belgique, Suisse et d'île de France de Robotique de 2012 à 2015
 - 2013: 9^{ème} place en coupe de France et Meilleure équipe étrangère lors de la Coupe de Belgique
 - 2014: Vice champions de France, vice champions d'île de France
 - 2015: Vice champions de Belgique, vice champions de Suisse, vice champions d'île de France
- Chaque année, nous mettons en priorité les déplacements des robots par la réutilisation d'une base roulante et de l'électronique.

Puis nous venions ajouter les actionneurs pour interagir avec les éléments de jeux.

Généralement le robot était capable de marquer des points à la période de Noël.

Le reste de l'année était consacré à l'amélioration de la stratégie, au découpage des actions et au debuggage

Ce document a pour but de présenter la majorité des travaux qui ont été réalisés dans ce cadre

- Présentation de l'architecture électronique
- Présentation détaillée des cartes électroniques
- Introduction à l'application Temps réel multi-tâches

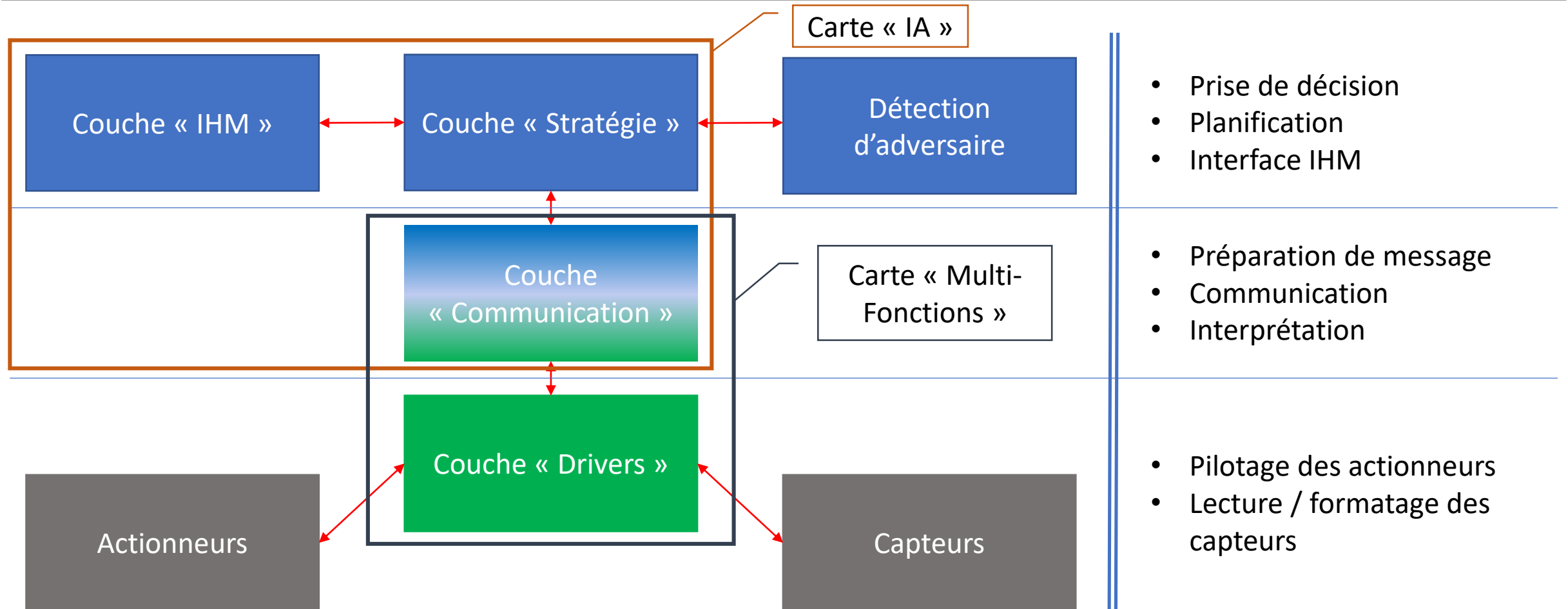


Idée générale

- L'électronique est conçue de façon à répartir les tâches entre différentes cartes
 - La partie prise de décision est portée par une carte dite « IA »
 - La gestion des actionneurs est déplacée sur une/des carte(s) dites « Multi-fonctions »
- Critères de choix de cette organisation
 - Ce choix a été fait pour limiter le nombre de cartes différentes à réaliser
 - Pouvoir les réutiliser d'une année à l'autre sans avoir une électronique spécifique à une édition
 - Etre le plus modulaire possible
 - Pouvoir servir dans le petit et le gros robot
 - Peu encombrante pour être positionnable au souhait dans les volumes autorisés
 - Utiliser le moins de code spécifique possible (le même programme pour toutes les cartes)
 - Une architecture similaire entre les deux robots

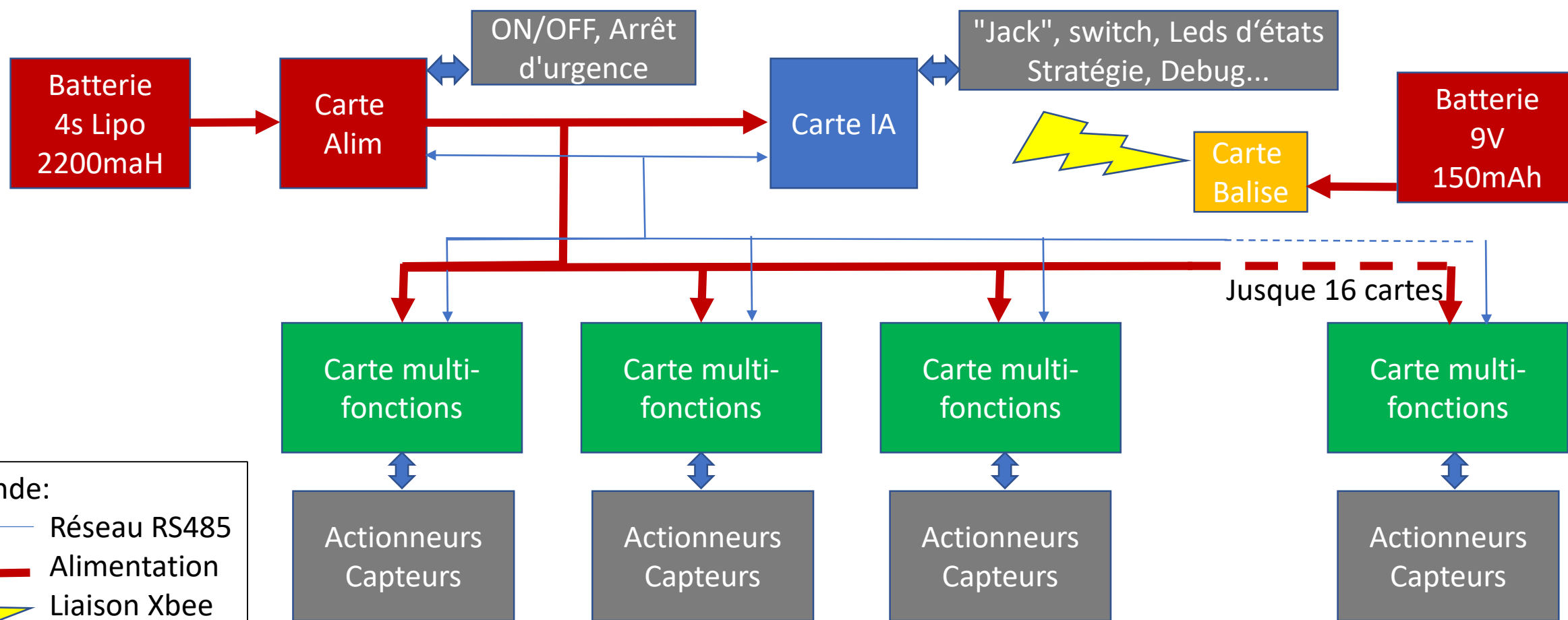
Présentation de l'architecture électronique

- Architecture générale



Présentation de l'architecture électronique

- Architecture générale



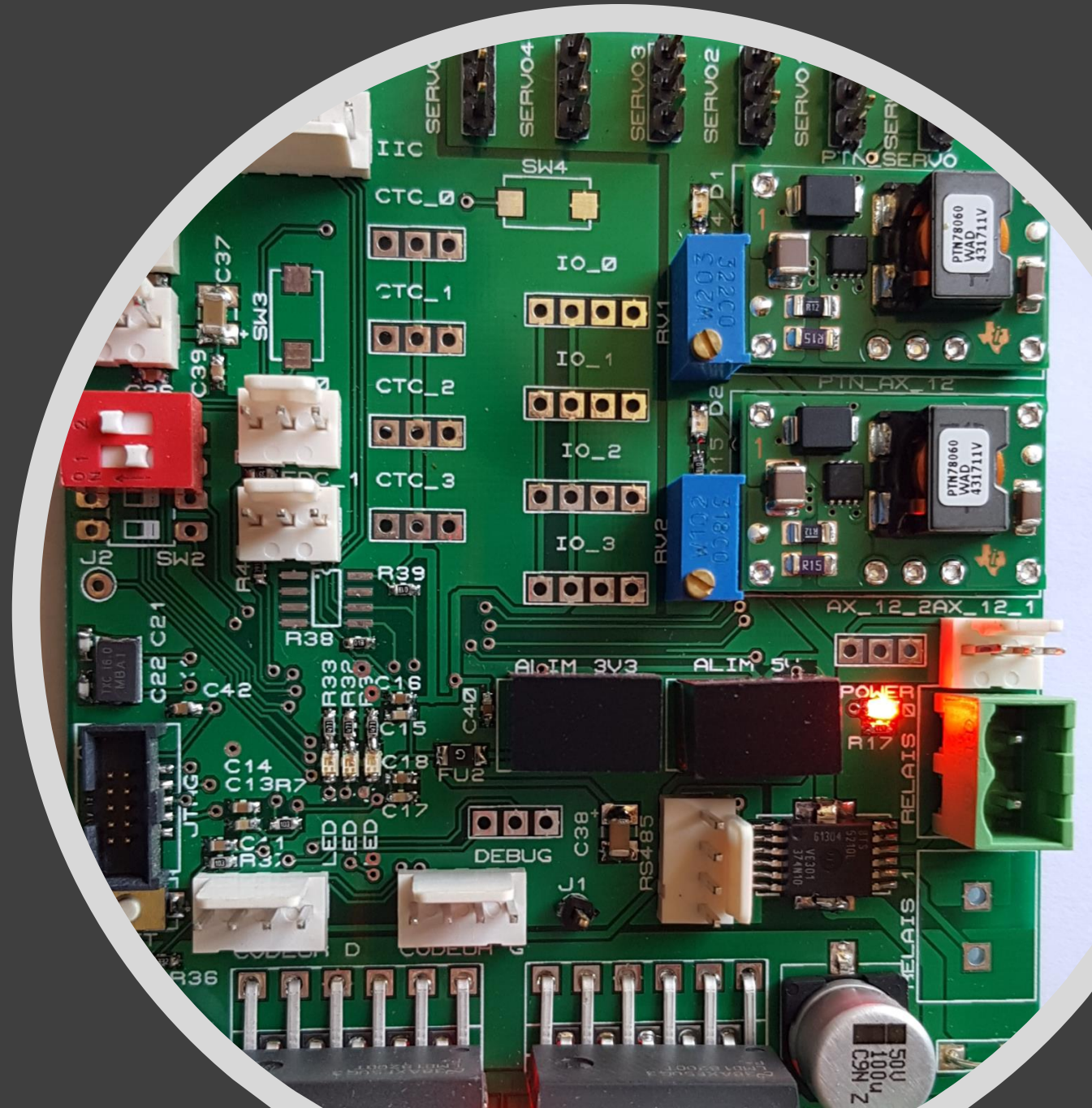
Présentation de l'architecture électronique

- Question time



Détails des cartes

Présentation de l'architecture électronique



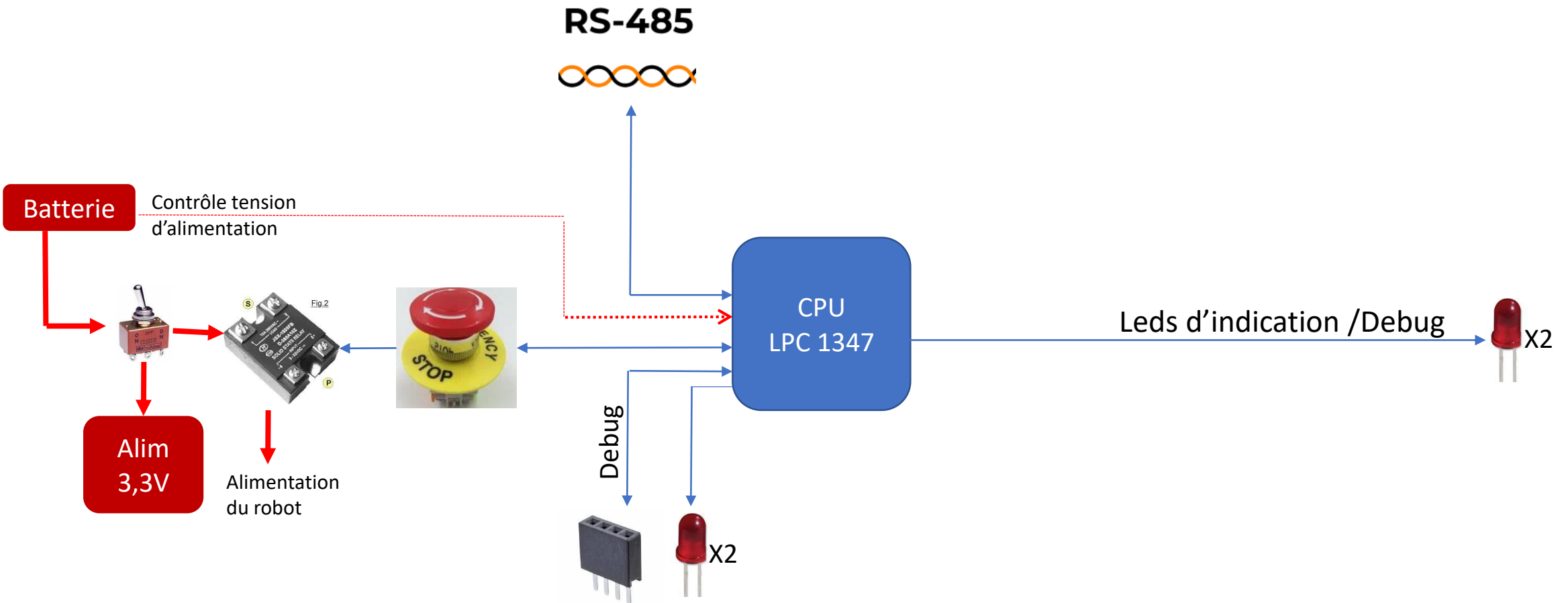
Détail des cartes

Cartes Alim

- Alimentation: 14 à 20 V
 - Avec fusible et protection contre les inversion de polarité
 - Création de 3.3V (1A)
 - 2 plans de masses pour l'isolation
 - Interrupteur ON/OFF principal
 - 4 sorties pour alimenter le robot
 - 1 micro contrôleur LPC 1347
 - Cortex-M3, 72MHz
 - 64KB de Flash, 12KB de RAM
 - 1 entrée analogiques
 - Mesure de tension de batterie
 - 1 arrêt d'urgence
 - Environnement de programmation gratuit (jusque 256KB)
 - Compatible FreeRTOS
- MOSFET (30A) ON/OFF piloté
 - 1 port RS485
 - 1 pin de debug
 - 2 leds déportées libre
 - 2 leds libres d'usage sur la carte
 - 2 leds libres d'usage déportées
 - Leds d'indication
 - Présence 3.3V
 - Un connecteur de programmation JTAG
 - Format 5*7cm
 - 4 trous de fixation

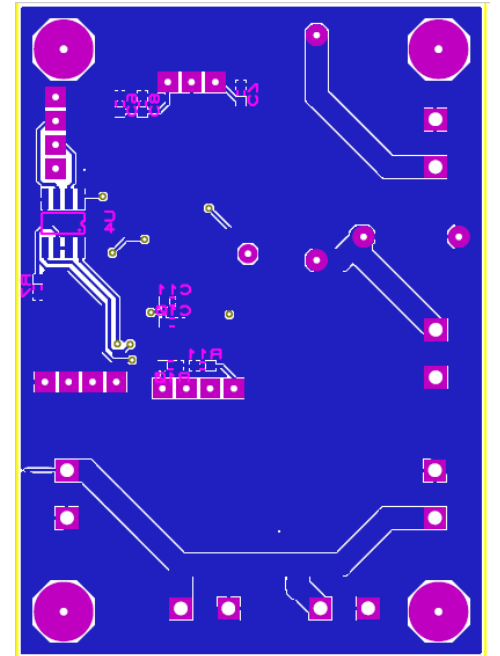
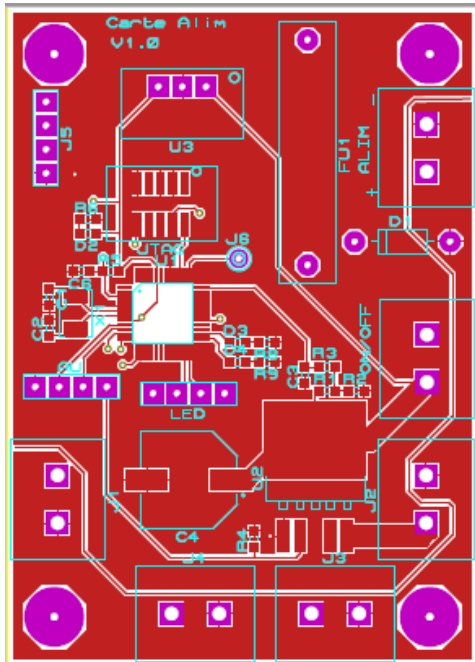
Détail des cartes

- Cartes Alim, blocks fonctionnels



Cartes Alim

- Gros plan sur les deux faces



Résumé

- Cartes Alim

- Les cartes IA présentes dans les deux robots en 2015 nous ont donné entière satisfaction

- Programmation facile
- Protection du robot et des batteries
- Fiabilité et encombrement réduit

- Aucun défaut relevé

Détail des cartes

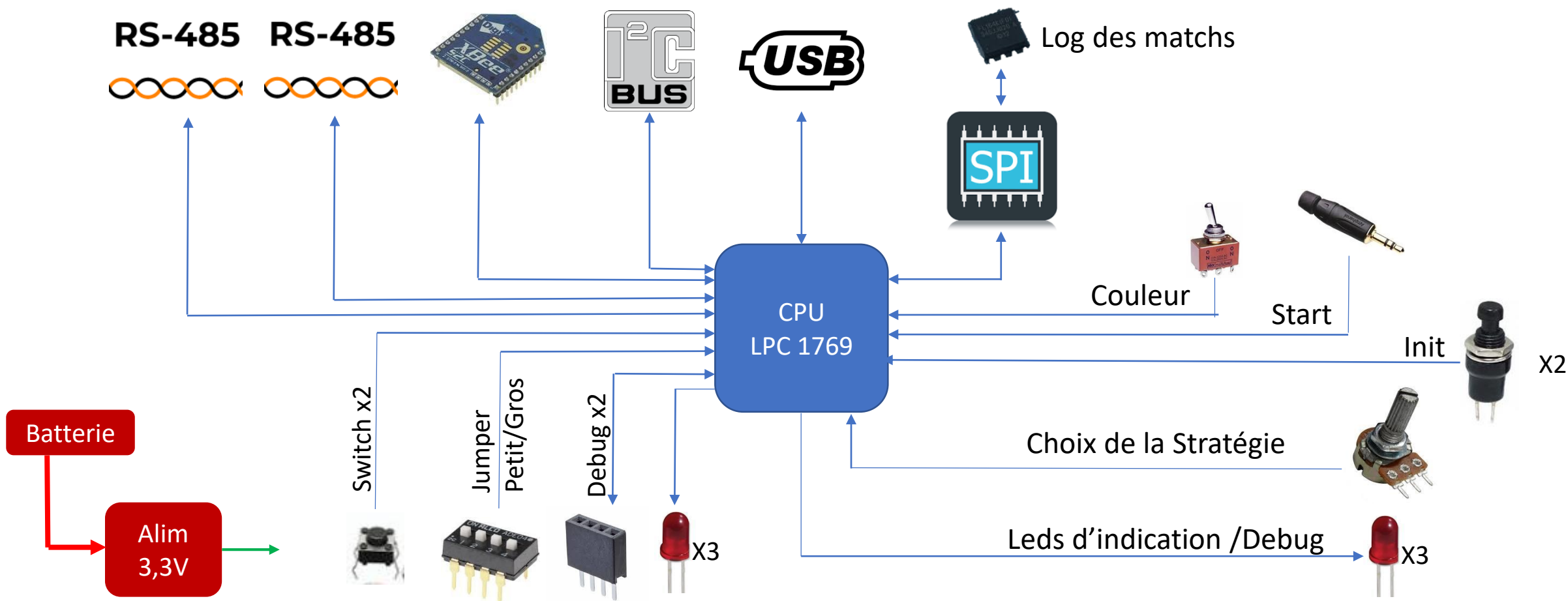
Cartes IA



- Alimentation: 14 à 20 V
 - Création de 3.3V (1A)
 - 2 plans de masses pour l'isolation
 - 1 micro contrôleur LPC 1769
 - Cortex-M3, 128MHz
 - 512KB de Flash, 64KB de RAM
 - 1 entrée analogiques
 - Potentiometre exterieur pour choix de stratégie
 - 4 entrées TOR
 - Jack
 - Couleur de jeu
 - Bouton poussoir
 - Autre
 - 2 boutons libres d'usage
 - 1 switch de reset
 - 1 jumper (Petit/Gros Robot)
 - Environnement de programmation gratuit (jusque 256KB)
 - Compatible FreeRTOS
- 2 port RS485
 - 1 port I2C
 - 1 port UART vers module Xbee
 - 1 port USB de debug
 - Une mémoire flash SPI (log des matchs)
 - 2 pins de debug
 - 3 leds libres d'usage sur la carte
 - 3 leds libres d'usage déportées
 - Leds d'indication
 - Présence 3.3V
 - Association du Xbee
 - Un connecteur de programmation JTAG
 - Format 5*7cm
 - 4 trous de fixation

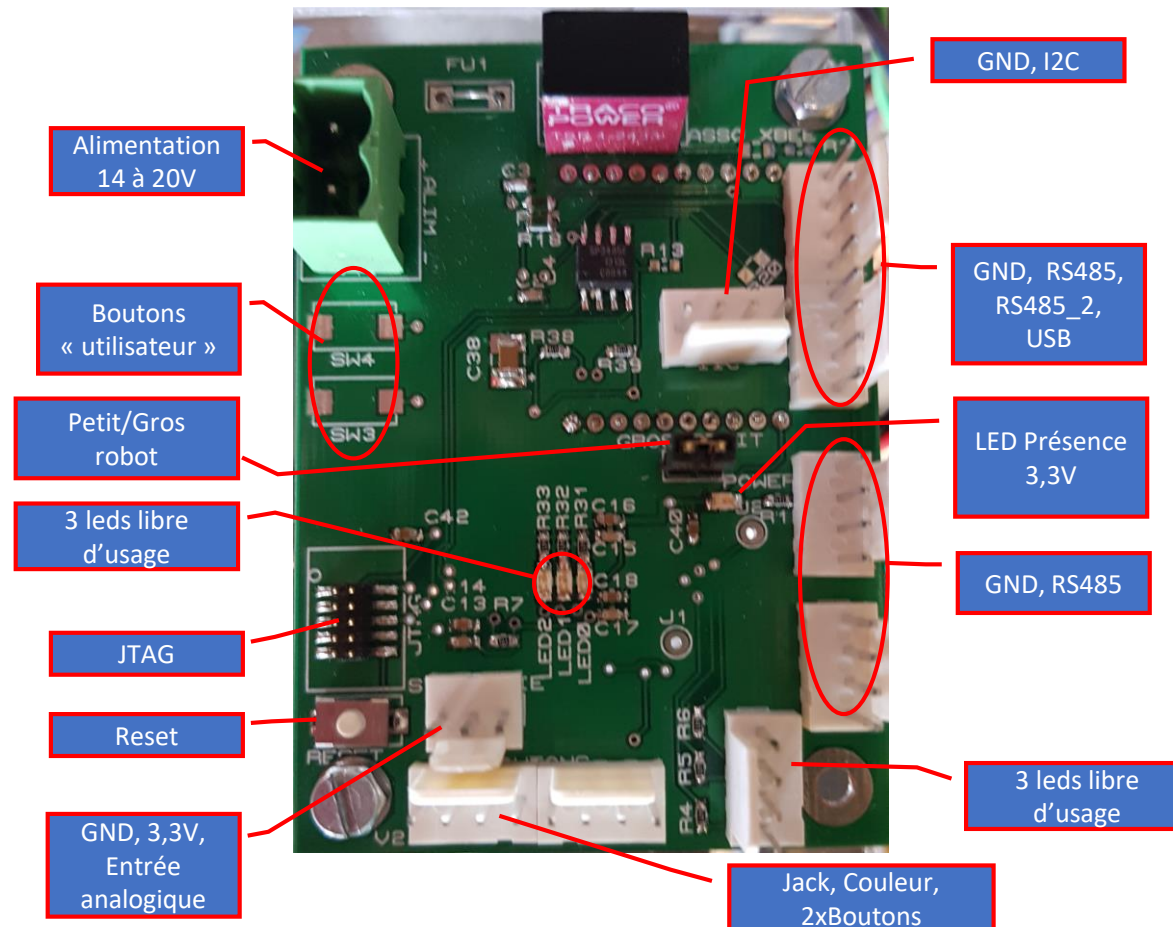
Détail des cartes

- Cartes IA, blocks fonctionnels



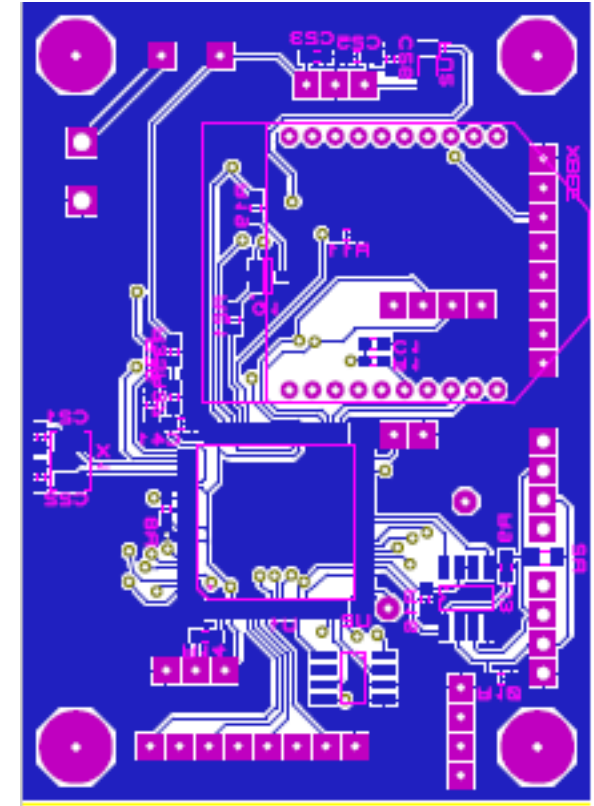
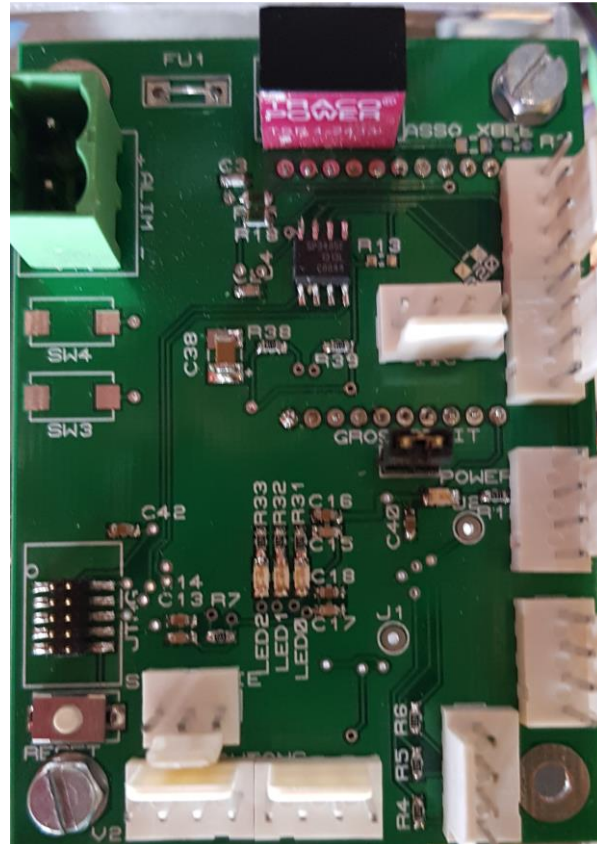
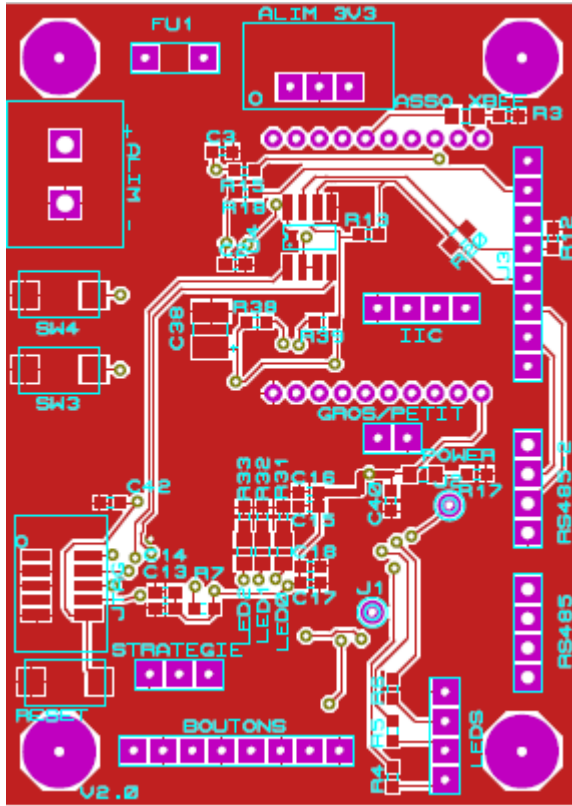
Détail des cartes

- Cartes IA




Cartes IA

- Gros plan sur les deux faces



Résumé

- Cartes IA

- 
- Les cartes IA présentes dans les deux robots en 2015 nous ont donné entière satisfaction
 - Puissance de calcul sans faille (Pathfinding, gestion de la stratégie, IHM, communication Inter-Robot + balises)
 - Format compact
 - Nombreux moyens de communication
 - Enregistrement des matchs en interne (jusque 3 matchs sauvegardés en flash) avec possibilité de relecture sur l'IHM
 - Une seule programmation pour les deux Robots (le jumper sur la carte permettant de faire la différence)
 - Un choix facile de stratégie (en quantité « illimitée »), pas besoin de reprogrammer au dernier moment

- 
- Aucun défaut relevé

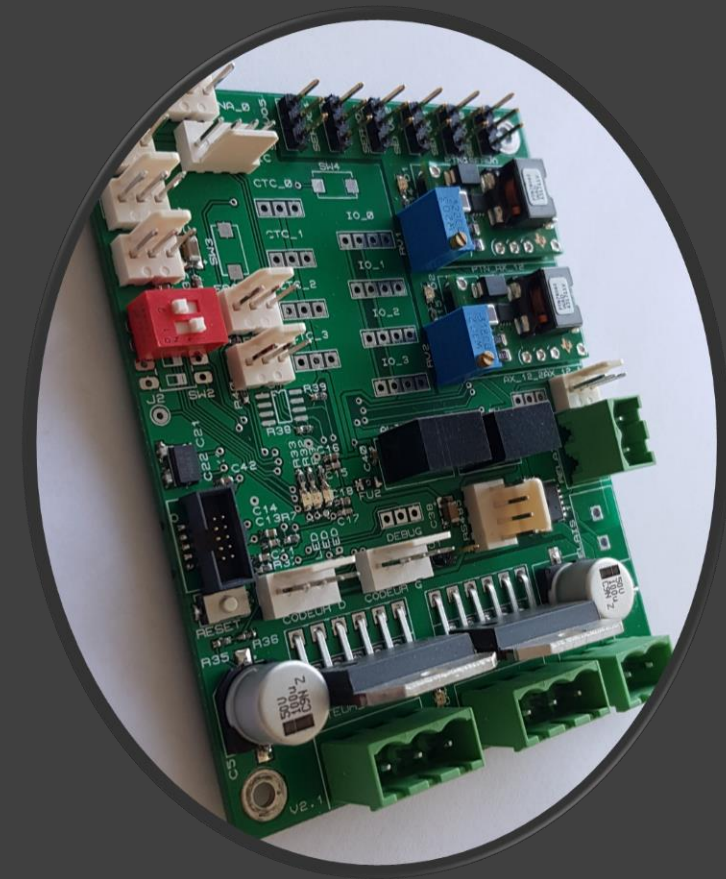
Carte IA

- Question time



Détail des cartes

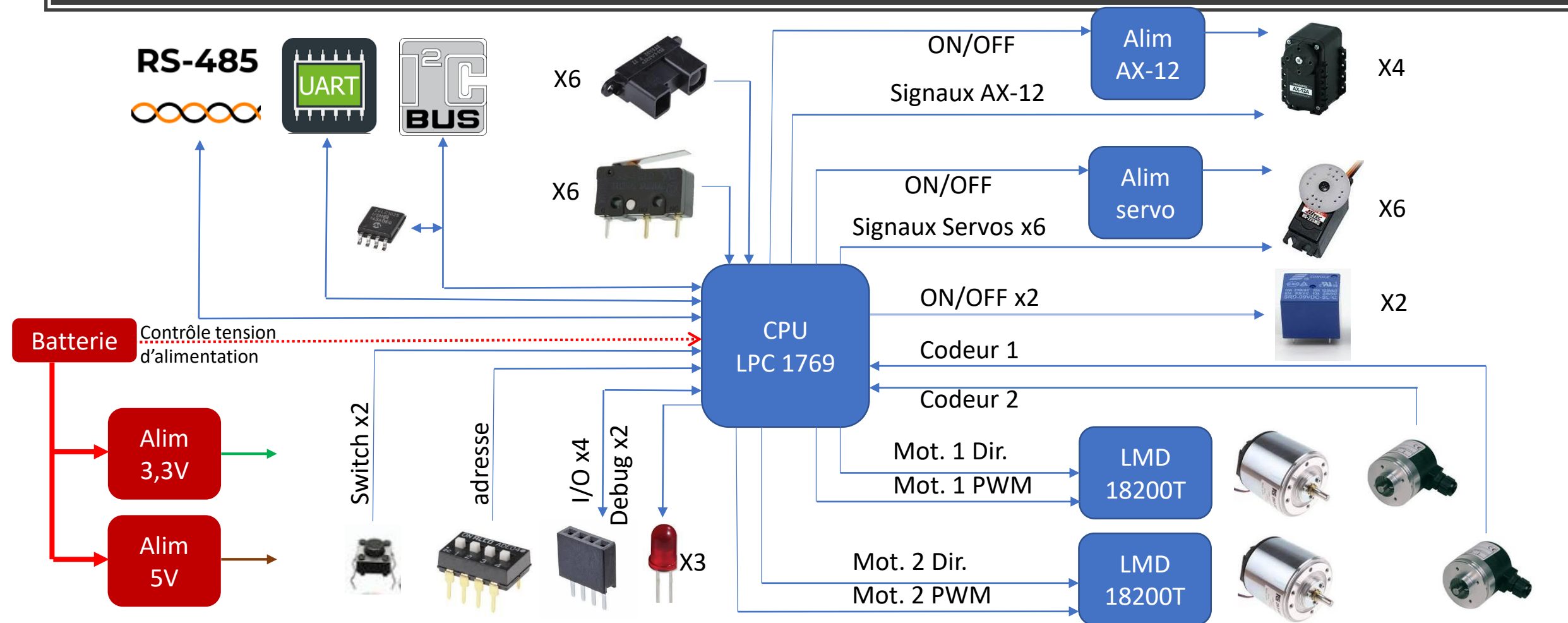
Cartes multi-fonctions



- Alimentation: 14 à 20 V, avec mesure de tension d'alimentation
 - Création de 3.3V et 5V (1A chacun)
 - 2*4 plans de masses pour l'isolation
 - 1 micro contrôleur LPC 1769
 - Cortex-M3, 128MHz
 - 512KB de Flash, 64KB de RAM
 - 2 contrôleurs de moteurs
 - 2 entrées de codeurs en quadrature avec alimentation 5V
 - 2 sorties TOR: 16V, 1.8A
 - 4 entrée analogiques avec alim 5V, diviseur de tension et filtrage PB
 - Alimentation (réglable et débrayable) et pilotage de 6 servos
 - Alimentation (réglable et débrayable) et pilotage de 4 servos type AX-12
 - 6 entrées TOR (contacteurs) avec alim 3.3V
 - 4 entrées/ sorties TOR avec alimentations 3.3V et 16V
 - Environnement de programmation gratuit (jusque 256KB)
 - Compatible FreeRTOS
 - Potentiellement autonome
- 2 boutons libres d'usage
 - 1 switch de reset
 - 4 switchs d'adresse
 - 1 port RS485
 - 1 port I2C avec alimentation 5V
 - 1 port UART
 - Une eeprom I2C
 - 2 pins de debug
 - 3 leds libres d'usage
 - Leds d'indication
 - Présence alim de puissance
 - Présence 5V et 3.3V
 - Alim des servos
 - Alim des AX12
 - Un connecteur de programmation JTAG
 - Format 7*10cm
 - 4 trous de fixation

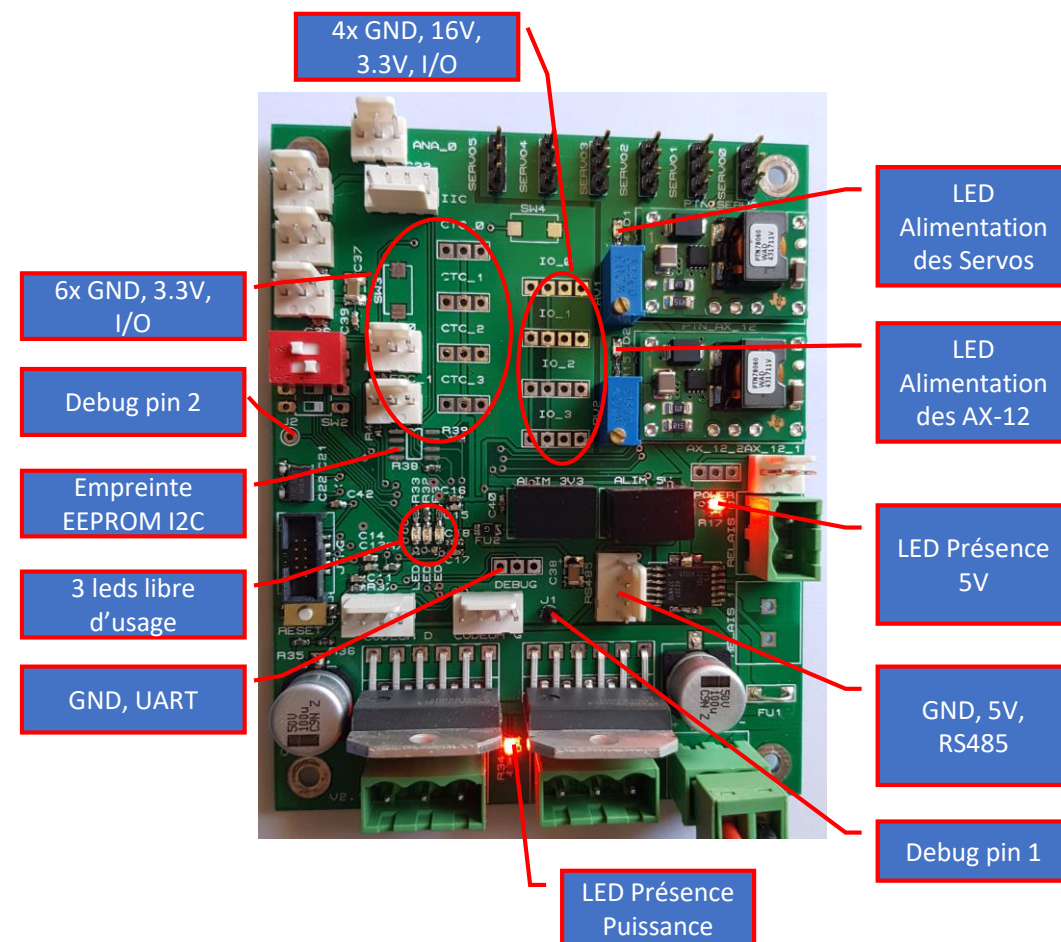
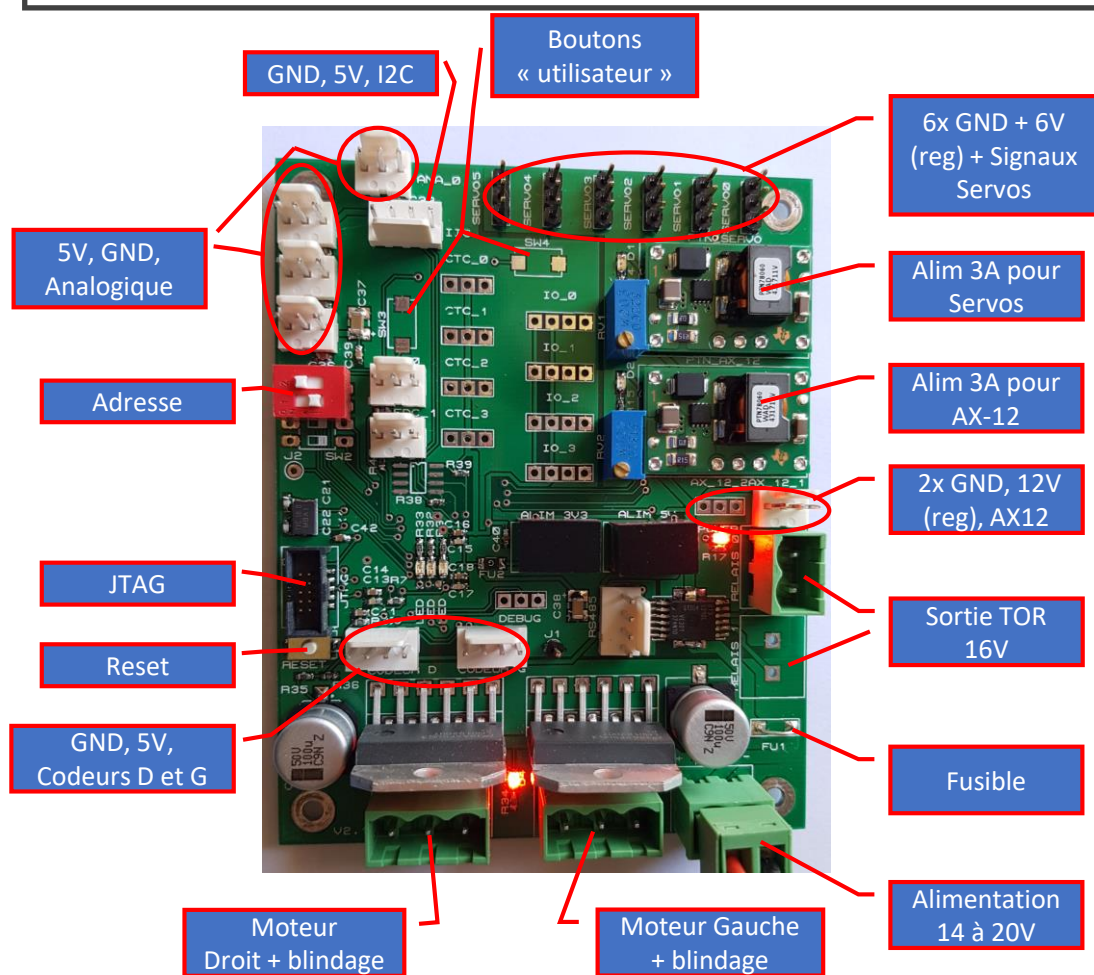
Détail des cartes

- Cartes multi-fonctions, blocks fonctionnels



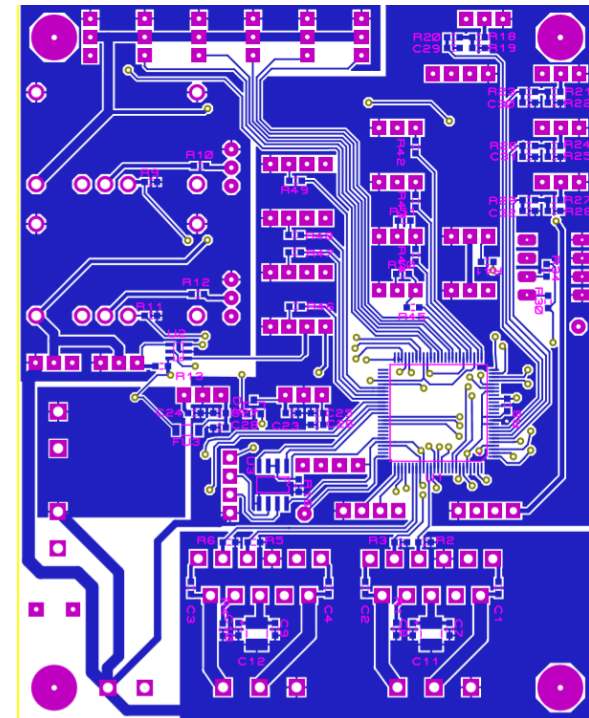
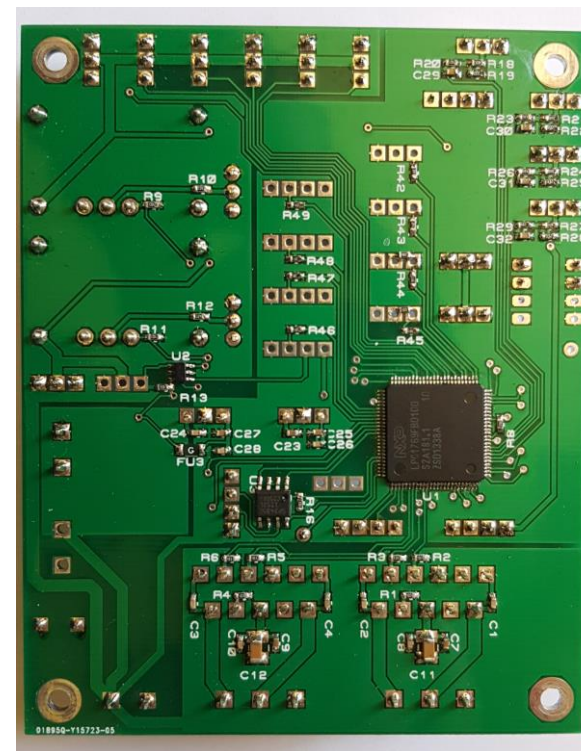
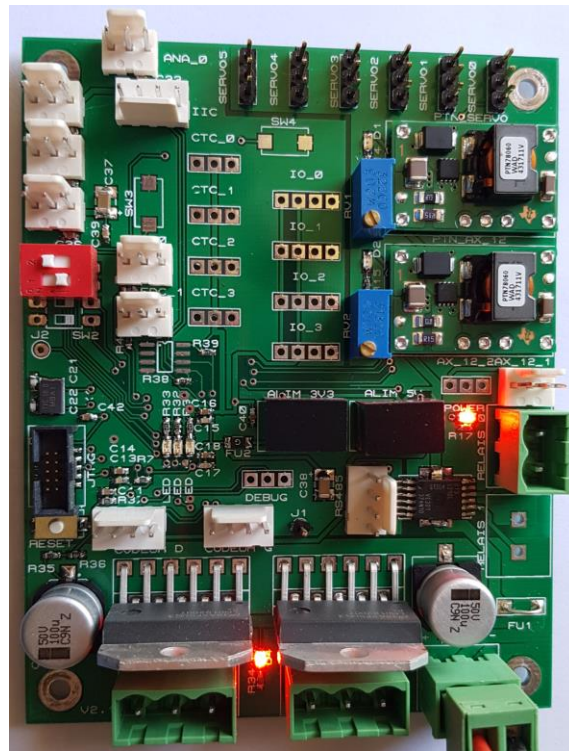
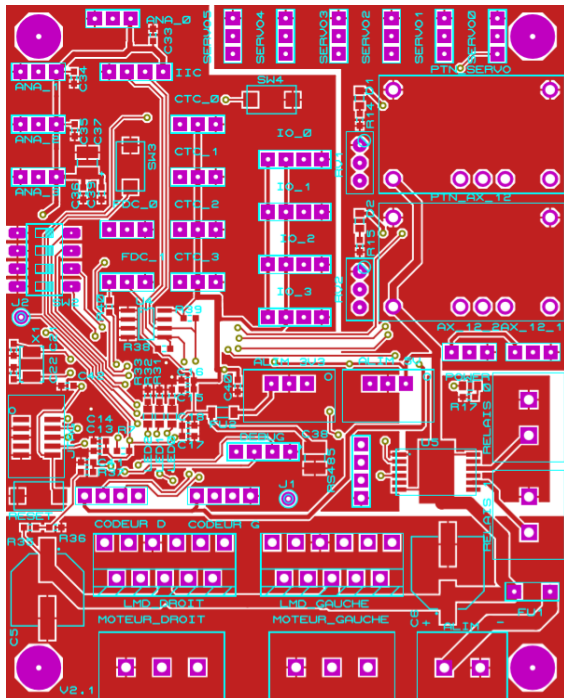
Détail des cartes

- Cartes multi-fonctions



Cartes multi-fonctions

- Gros plan sur les deux faces



Résumé

- Cartes multi-fonctions

- Ces cartes multi-fonctions se sont révélées performantes et fiables, voici les points forts et faibles rencontrés:
 - La puissance de calcul est au rendez-vous, l'asservissement et le pilotage des servos est amplement assuré
 - Le format compact permet de les positionner facilement dans les deux robots
 - Le design commun permet de n'avoir qu'un seul et même code pour chacune des cartes
 - La possibilité d'étendre le bus permet de passer de 1 carte sur le petit robot à 2 cartes sur le gros pour couvrir les besoin sans rien avoir à développer d'autre
 - Possibilité de n'utiliser qu'une seule carte pour faire fonctionner un robot
- A l'issue de la V2.1 (ici présentée en photo), quelques erreurs ont été constatées:
 - Il manquait 2 résistances de pull-up sur les bit d'adresse des cartes (les deux autres sont équipée en pull-up internes)
 - Les sorties « Timers » utilisées pour piloter les moteurs ne sont pas le meilleur choix pour générer des PWM avec ces micro-contrôleurs, une optimisation du code/hard est à revoir.


Carte Multi-Fonctions

- Question time



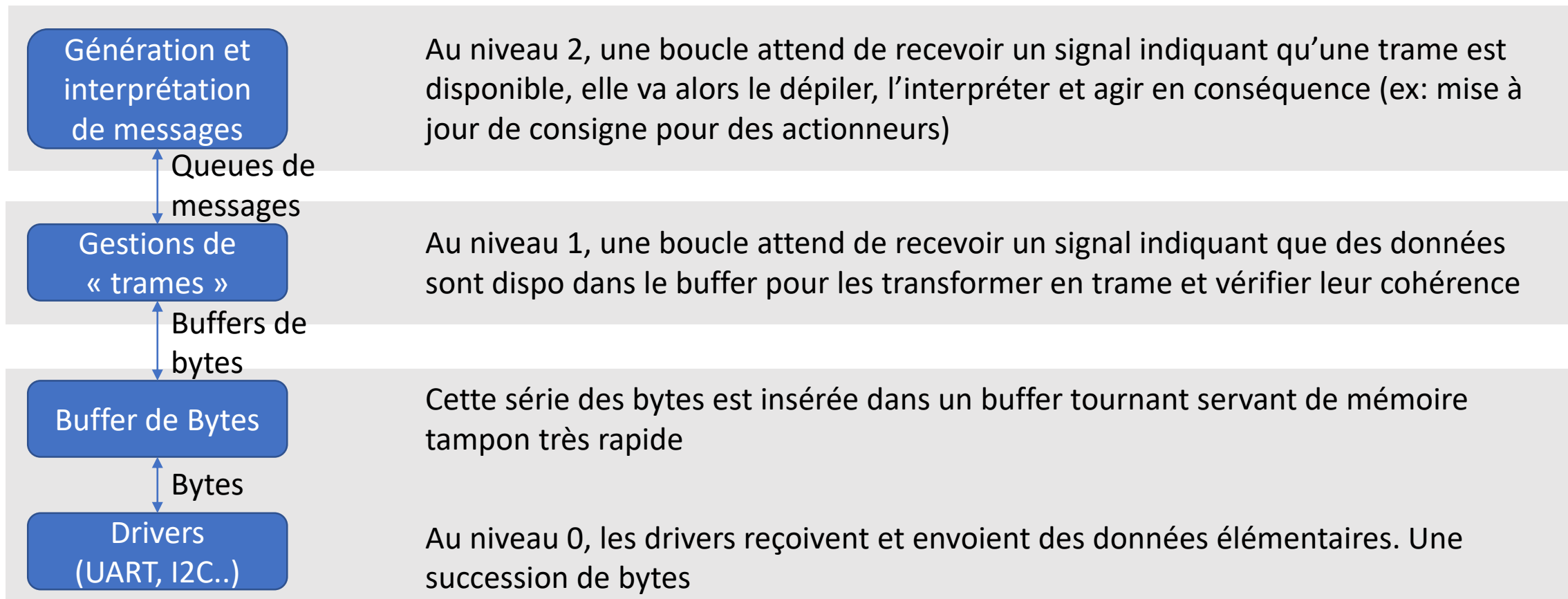
Firmware Multi-tâches Temps réel préemptif

- Généralités

- L'informatique des robots est répartie sur plusieurs cartes.
Chacune de ces cartes fait tourner un micrologiciel qui a pour but, suivant la carte, soit de s'occuper de la stratégie du robot, soit de piloter les actionneurs et lire les capteurs, ou encore de surveiller l'alimentation électrique.
- Les différentes cartes communiquent ensemble à l'aide d'un bus RS485 (un bus I2C est aussi à disposition)
- Afin de synchroniser toutes les actions un OS temps réel a été choisi, il s'agit de FreeRTOS 
 - Il permet de réaliser plusieurs boucles d'opérations (tasks) en parallèle et de passer rapidement de l'une à l'autre. Avec en plus la mise à disposition de différents moyens de communication entre ces tâches (Piles de messages, sémaphores....)
 - De cette façon, les différentes opérations ont été réparties sur plusieurs « niveaux » logiciels. En partant des pilotes de drivers (PWM, ADC, Timers, GPIO...) pour aller vers des couches de plus en plus abstraites
 - L'architecture électronique des cartes IA et multi-fonctions étant très similaires (même CPU, même I/O), une bonne partie du code est commune, accélérant ainsi le développement.

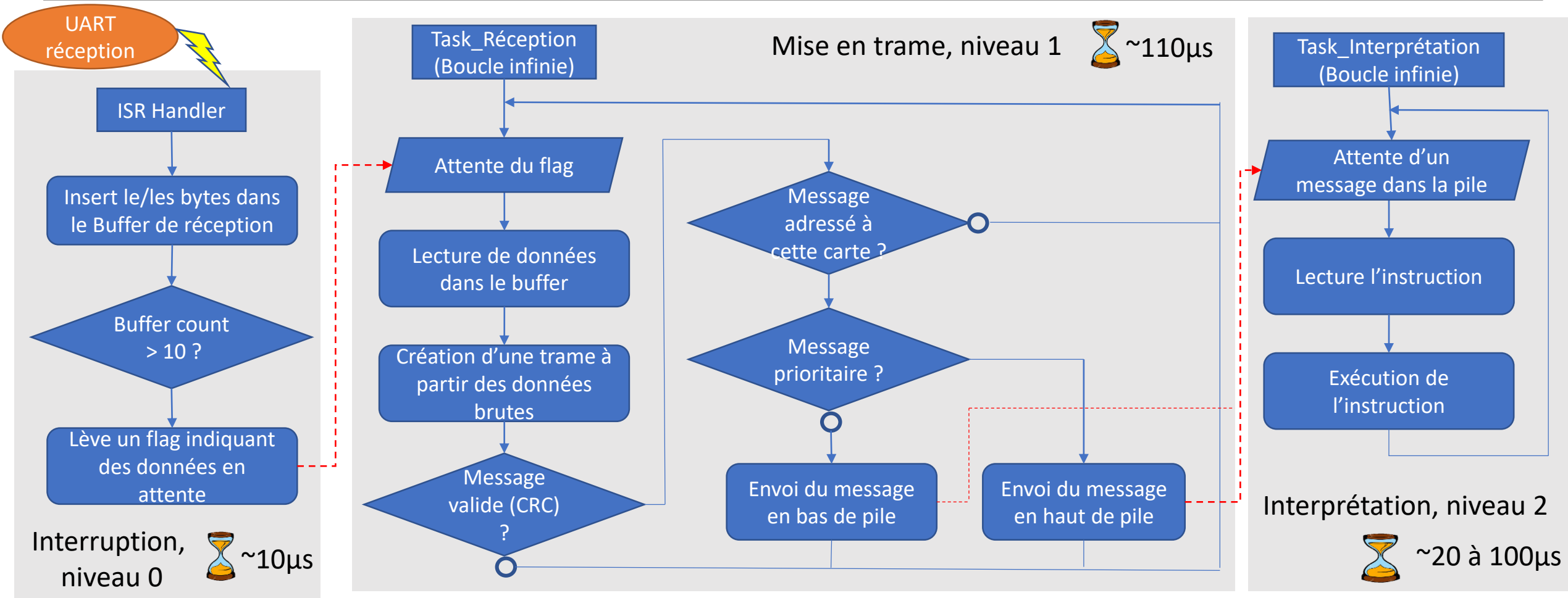
Firmware Multi-tâches Temps réel préemptif

- Une communication répartie en couches....



Firmware Multi-tâches Temps réel préemptif

- ...et des tâches en parallèle



Timing

Question-réponse

Séquence schématique illustrant le comportement lors de la réception d'un message (question) jusqu'à l'envoi de la réponse

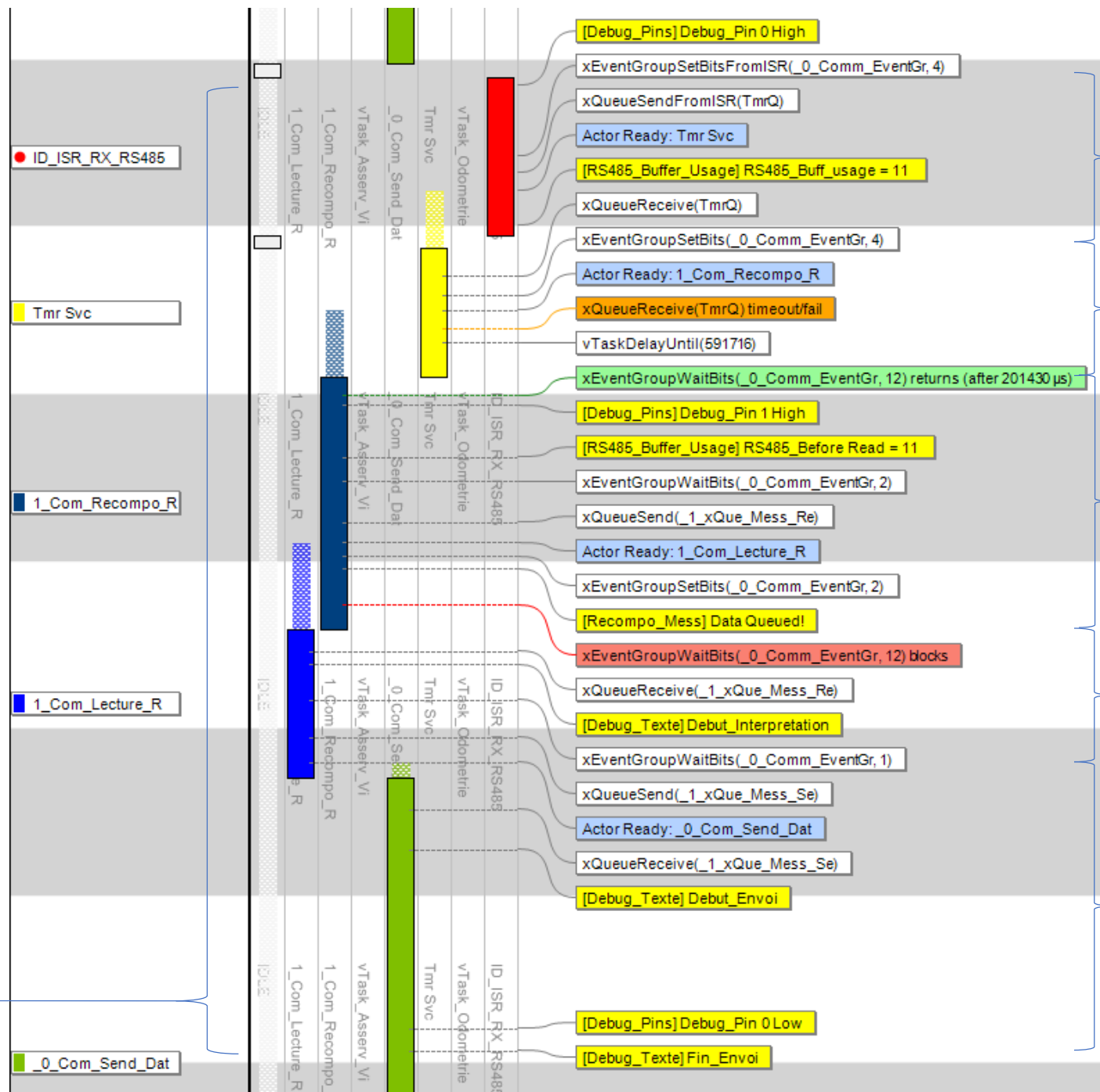


Timing

Question-réponse

Afin d'illustrer la séquence précédente dans le temps, voici un extrait issu de Tracalys[®] (outil d'analyse pour FreeRTOS) d'un échange question-réponse entre deux cartes. A lire de haut en bas.

Relevé de temps pris à l'analyseur logique
~200 à 500µs



Réception de la question en ISR

Levée du Flag

Formatage des données brutes en message à valeur ajoutée

Interprétation du message et préparation de la réponse

Envoi de la réponse

Le chef d'orchestre

- Généralités

- Le bus RS485 n'ayant pas de master/esclaves comme peut l'avoir un bus I2C, chaque carte est libre d'envoyer un message sur le bus au risque de le faire en même temps qu'une autre et ainsi de compromettre les deux messages.
- Pour palier à cette situation, il a été décidé que seule la carte IA pouvait décider d'envoyer des messages de son propre chef.
- Les autres ne feront que répondre à une/des question(s) ou pourront vider leur pile de messages à envoyer lorsque la carte IA leur permettra.
- L'IA interroge les autres cartes à tour de rôle puis leur laisse le temps de répondre, suivi d'un silence sur le bus, servant à l'ensemble des cartes à garder du temps de calcul pour exécuter les autres actions.

Firmware

- Question time



« Adressage des
actionneurs »



Adressage des actionneurs

- @servo

- Chaque actionneur ou capteur est vu sous 2 aspects:
 - Un premier au niveau de la carte multifonction qui le pilote
 - Un second au niveau de la carte IA
- Coté multifonction: ils sont connus par un numéro (ID local)
 - Les servos: de 0 à 5
 - Les AX-12: de 6 à 9 (qui correspond à leur ID sur le bus)
- Coté IA
 - Le numéro de la carte qui les pilote * 10 + numéro du servo (ID local) => ID Global
- Ainsi l'IA pilote les servos 10 à 19 sur la carte 1, indifféremment s'il s'agit d'un servo normal ou d'un AX-12
 - Les servos 20 à 29 sur la carte 2, 30 à 39 sur la 3...

Adressage des actionneurs

- Move servo

- Lorsque l'IA a besoin de faire bouger un servo, elle va envoyer un ordre sur le bus RS485 avec:
 - Instruction = `Deplacement_Servo`
 - Datas:
 - Nombre de servos à déplacer
 - Temps de déplacement souhaité
 - Tableau : `[servo_ID, destination, couple max]`
- Toutes les cartes multifonctions vont alors recevoir le message, l'interpréter, et déplacer les actionneurs qui les concernent (`servo_ID / 10 == adresse_carte`) dans le temps donné en consigne
- De cette façon plusieurs servos, connectés à plusieurs cartes peuvent être déplacés en même temps, de façon synchrone sans avoir à se préoccuper de la carte sur laquelle il est branché.
- Un seul message pour demander le déplacement de 12 servos simultanément
- L'IA ne se soucie pas de qui est connecté ou, ni de quel type de servo il s'agit, ce sont les cartes multifonctions qui décodent les infos, calculent les vitesses de chaque servo pour les déplacement dans le temps souhaité et envoient les infos aux servos suivant leur type

Move servo Exemple

Le message suivant crée par cet appel:

```
Move_3_Servos(12, 12546, 0, 17, 512, 200, 38, 125, 300, 1000);
```

```
Message = {  
    Instru = Deplacement_Servo;  
    Slave_Address = All_Boards;  
    length = 18; //1+2+1+2+2+1+2+2+1+2+2  
    Data =  
        3, //3 servos à déplacer  
        1000, //en 1000ms  
        12, 12546, 0, //servo 12, destination 12546, couple à 0  
        17, 512, 200, //servo 17, destination 512, couple à 200  
        38, 125, 300 //servo 38, destination 125, couple à 300  
};
```

Au pour conséquences de faire déplacer 3 servos de façon synchrone

- Le 3^{ème} servo de la carte 1 (Global_ID = 12) vers la destination 12546
- Le 2^{ème} AX-12 de la carte 1 (Global ID = 17), vers la destination 512 avec le couple max = 200
- Le 3^{ème} Ax-12 de la carte 3 (Global ID = 38), vers la destination 125 avec le couple max = 300
- Le tout en 1000ms

Move servo Réception

Du point de vue des cartes multifonction:

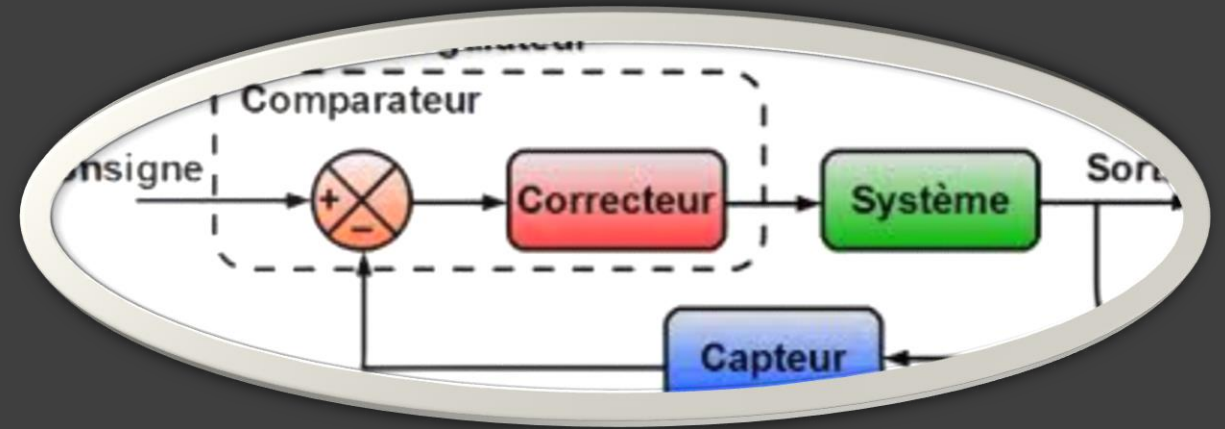
```
Void Rx_Dplacement_Servo(message)
{
    int nombre_servo = message.nombre_servos;
    //Pour chaque servo du message
    for(int i = 0; i < message.nombre_servos; i++)
    {
        if(message.servo[i].id / 10 == address_carte)
        {
            //Ce servo est piloté par cette carte
            if(message.servo[i].id % 10 <= 5)
            {
                //Servo
                MaJ_Desti_Servo(message.servo[i].id % 10, message.servo[i].destination, message.temps_deplacement);
            }else
            {
                //AX-12
                speed = AX12_Calculate_speed(message.servo[i].id % 10, message.servo[i].destination,
message.temps_deplacement);
                MaJ_AX12_speed(message.servo[i].id % 10, speed);
                MaJ_AX12_torque(message.servo[i].id % 10, message.servo[i].torque);
                MaJ_AX12_desti(message.servo[i].id % 10, message.servo[i].destination);
            }
        }
    }
}
```


Adressage des actionneurs

- Question time



« Asservissement »



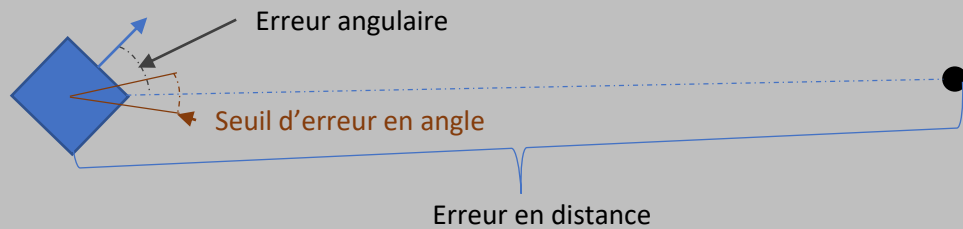
L'asservissement

- Généralités

- L'asservissement en position des robots est assuré par une cascade de PID. Les retours sont assurés par les codeurs montés sur des roues séparées des roues de propulsion qui permettent de mesurer les vitesses et positions du robot sur l'aire de jeu.
- Une première série de PID assure un asservissement en rotation afin d'orienter le robot vers sa destination, et en distance pour faire avancer (ou reculer) le robot jusqu'à sa cible.
- Ces PID sont seuillés afin de donner au robot un profil de vitesse en trapèze
- Les sorties de ces PID donnent des consignes de vitesses (en avance et angulaire) qui sont alors données à la seconde série de PID qui vont piloter les moteurs afin de respecter ces consignes en vitesses.
 - Les avantages de ce fonctionnement, sont d'avoir un mouvement assez régulier et une vitesse maîtrisée du robot mais aussi de conserver une consigne de vitesse même à faible distance de la cible et donc de réduire fortement les erreurs de fin de trajectoire. Une erreur entre la position souhaitée et celle calculée par l'odométrie inférieure au mm est atteignable.
- Un seuil d'erreur angulaire est aussi appliqué afin de permettre au robot de commencer à avancer avant d'être totalement aligné avec sa cible.
 - Au-delà de ce seuil, la vitesse en rotation est maximale, celle en avance est nulle.
 - En deçà du seuil, lorsque l'erreur angulaire tend vers 0, la vitesse en rotation tend vers 0 et la vitesse en avance tend vers son maximum

L'asservissement

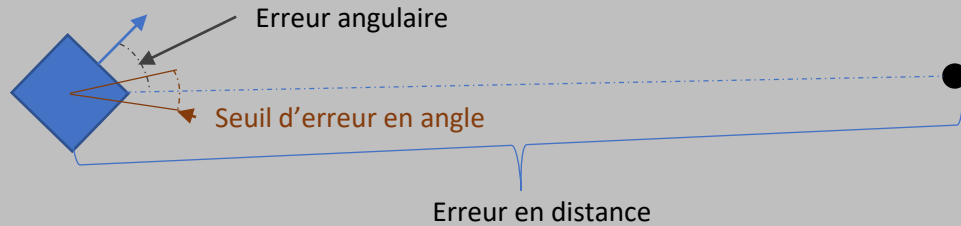
- Illustration



Initialement, l'erreur angulaire est $>$ au seuil
Le robot ne peut pas avancer, il tourne pour
s'orienter vers sa cible

L'asservissement

- Illustration



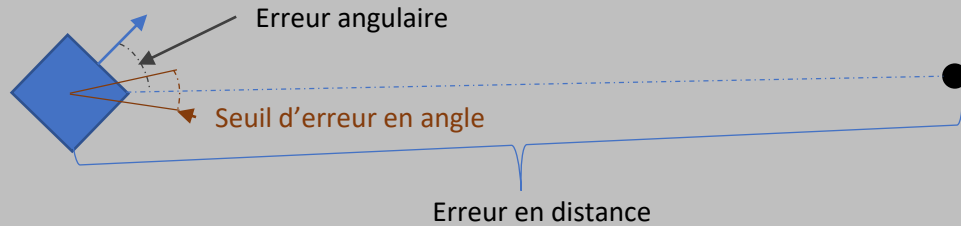
Initialement, l'erreur angulaire est $>$ au seuil
Le robot ne peut pas avancer, il tourne pour s'orienter vers sa cible



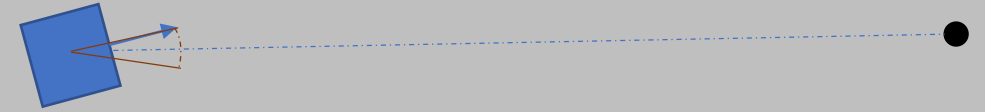
Lorsque l'erreur angulaire devient $=$ au seuil
le robot commence à avancer, sa vitesse est inversement proportionnelle à l'erreur en angle

L'asservissement

- Illustration



Initialement, l'erreur angulaire est $>$ au seuil
Le robot ne peut pas avancer, il tourne pour s'orienter vers sa cible



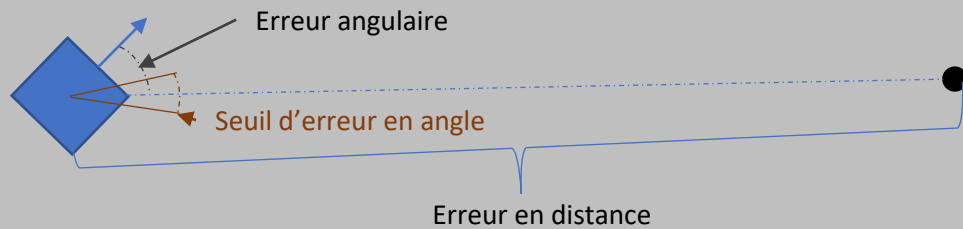
Lorsque l'erreur angulaire devient = au seuil
le robot commence à avancer, sa vitesse est inversement proportionnelle à l'erreur en angle



Au fur et à mesure que le robot avance, l'erreur angulaire diminue, de fait il accélère

L'asservissement

- Illustration



Initialement, l'erreur angulaire est $>$ au seuil
Le robot ne peut pas avancer, il tourne pour s'orienter vers sa cible



Lorsque l'erreur angulaire devient = au seuil
le robot commence à avancer, sa vitesse est inversement proportionnelle à l'erreur en angle

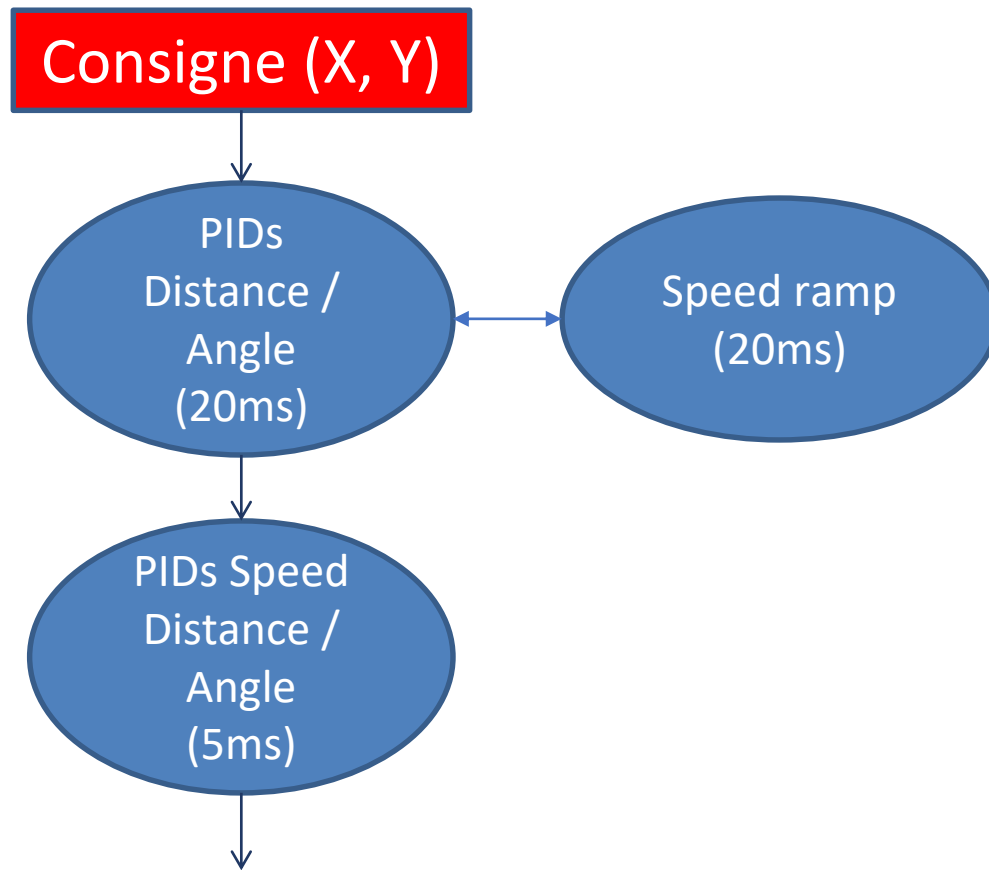


Au fur et à mesure que le robot avance, l'erreur angulaire diminue, de fait il accélère

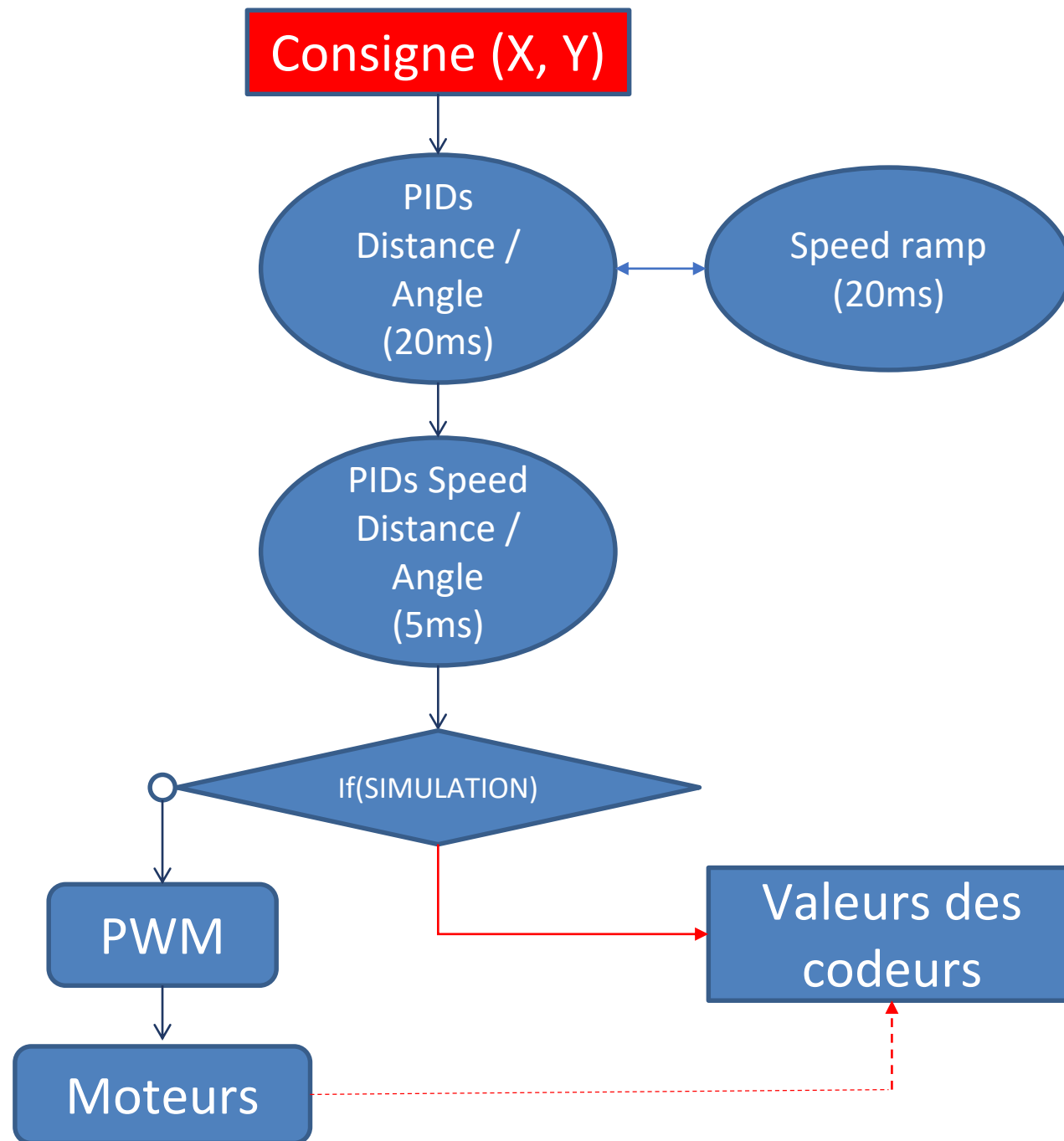


A l'arrivée, le robot n'a pas parcouru une ligne droite, mais une légère courbe qui tend à l'aligner vers sa cible en permanence

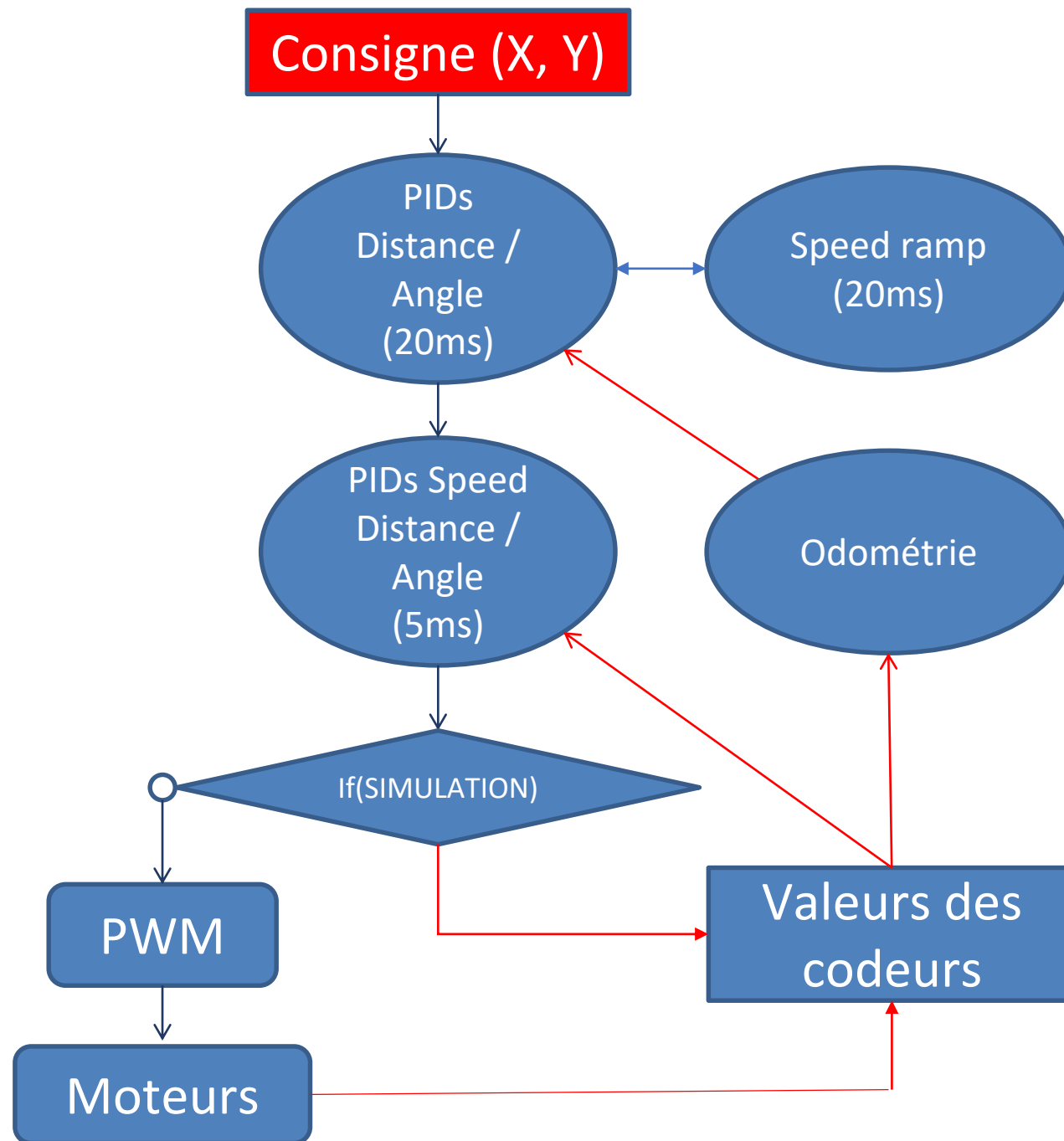
Asservissement



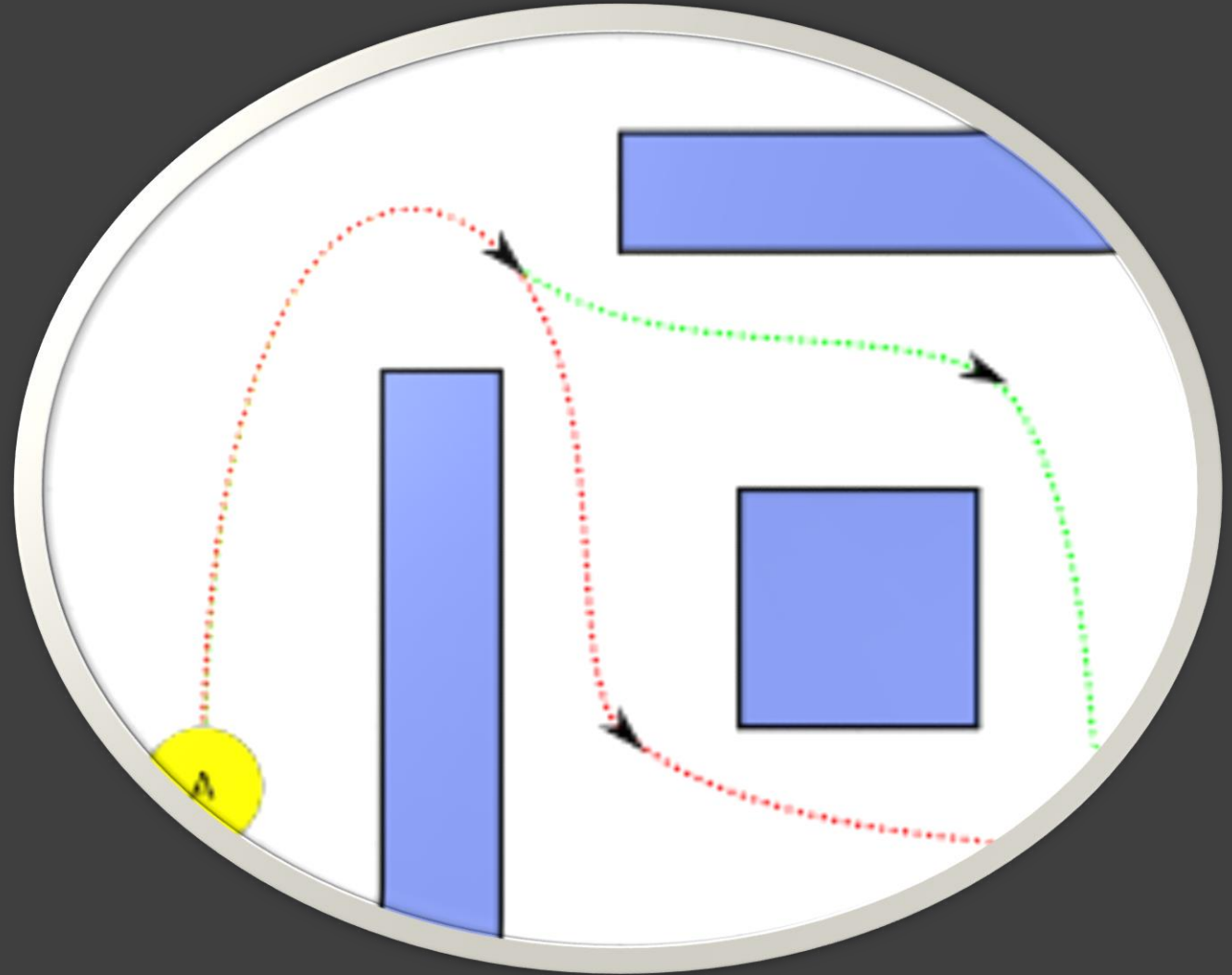
Asservissement



Asservissement



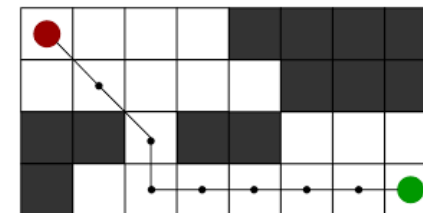
« Pathfinding »



Les déplacements

- Généralités

- Comme présenté dans ce document, les déplacements sont gérés par une carte « multi-fonctions »
 - Celle-ci va recevoir des ordres de déplacement du genre: Goto(X, Y) sans se préoccuper du pourquoi
 - Et retourner la position du robot (calculée par l'odométrie) à intervalles réguliers vers l'IA qui agira en fonction
- Une partie des coordonnées de déplacements sont rentrées à la main, d'abord de façon théorique (positions des éléments de jeu...) puis ajustées par itérations au cours des tests.
- Une autre partie de ces coordonnées va être calculée en temps réel par l'IA afin d'éviter les obstacles (fixes ou mobiles) sans que l'utilisateur ni les couches supérieures de stratégie n'aient à s'en préoccuper.
 - Pour cela, un algorithme de recherche de chemin est employé, il s'agit d'un A*.
 - Le principe est de discrétiser l'aire de jeu sous forme d'un damier. Certaines des cases seront considérées comme accessibles, d'autres comme bloquées (là où il y a des obstacles)
 - Une fois exécuté, l'algorithme nous renvoie une série de coordonnées qui correspondent aux points de passages à suivre pour rejoindre notre destination en évitant les obstacles.
(S'il existe un chemin, sinon il renvoi une erreur)

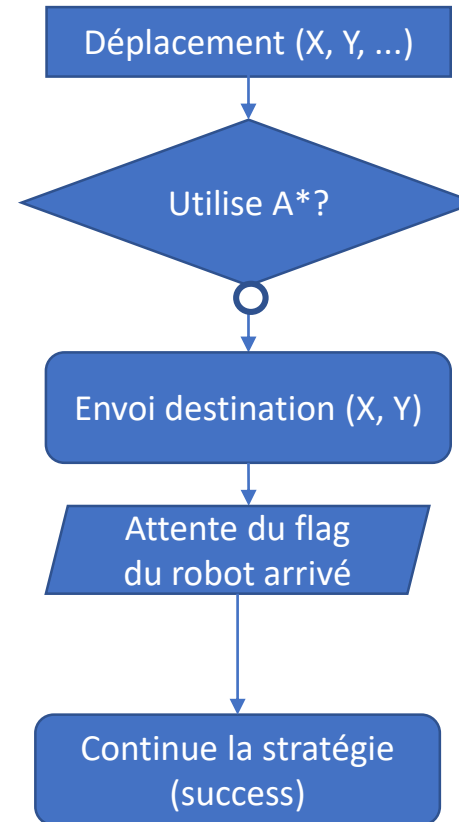


Les déplacements

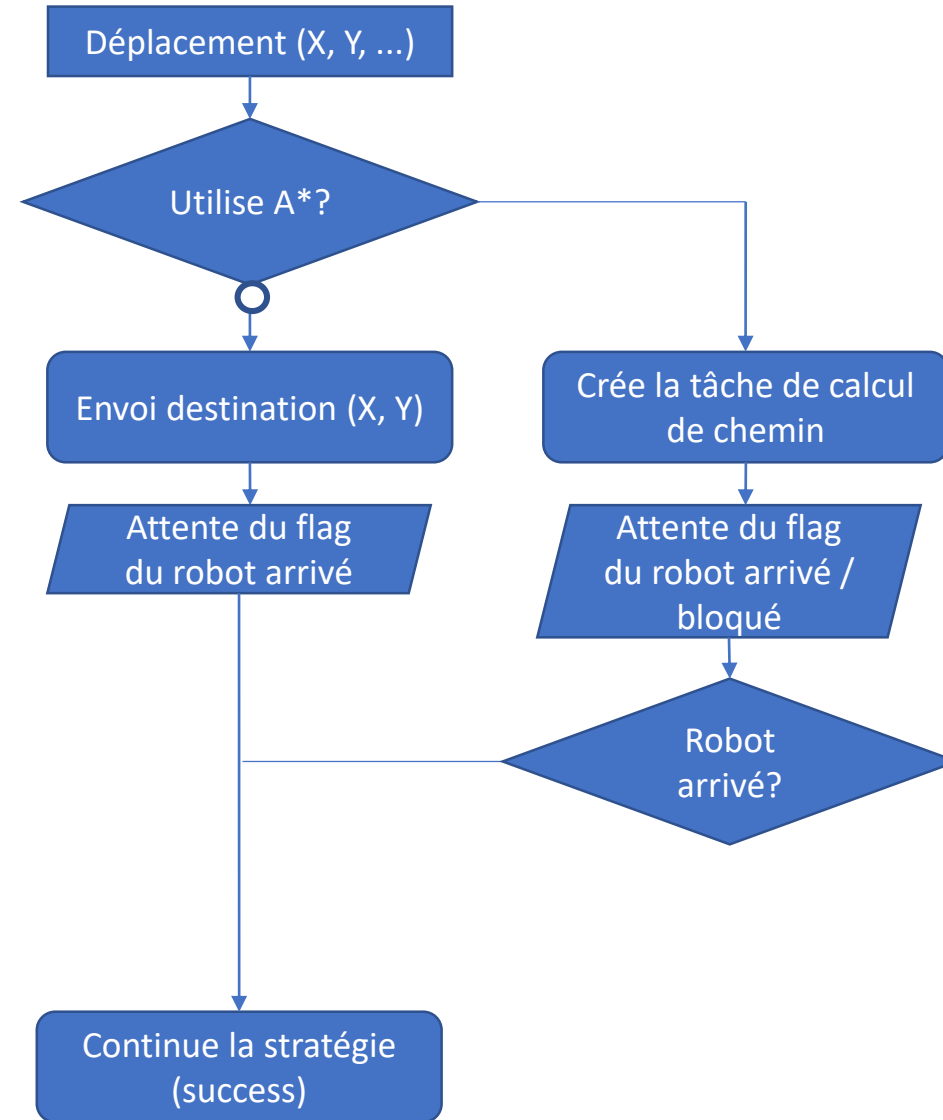
- Généralités

- L'utilisation d'un OS multitâches, permet d'effectuer différentes tâches en parallèle.
- De cette façon, une tâche principale de la stratégie va simplement demander au robot de se rendre à une destination et attendre que le robot y arrive (ou pas, s'il n'existe pas de chemin pour y aller).
 - Durant cette attente, une seconde tâche de l'OS va se charger d'exécuter l'algorithme, de chercher le meilleur chemin pour rejoindre la destination, et l'envoyer à la carte qui gère les déplacements (via une autre tâche en charge de la communication)
 - Lorsque le robot atteint sa destination, la tâche en charge du A* est supprimée, elle sera recréée au prochain déplacement

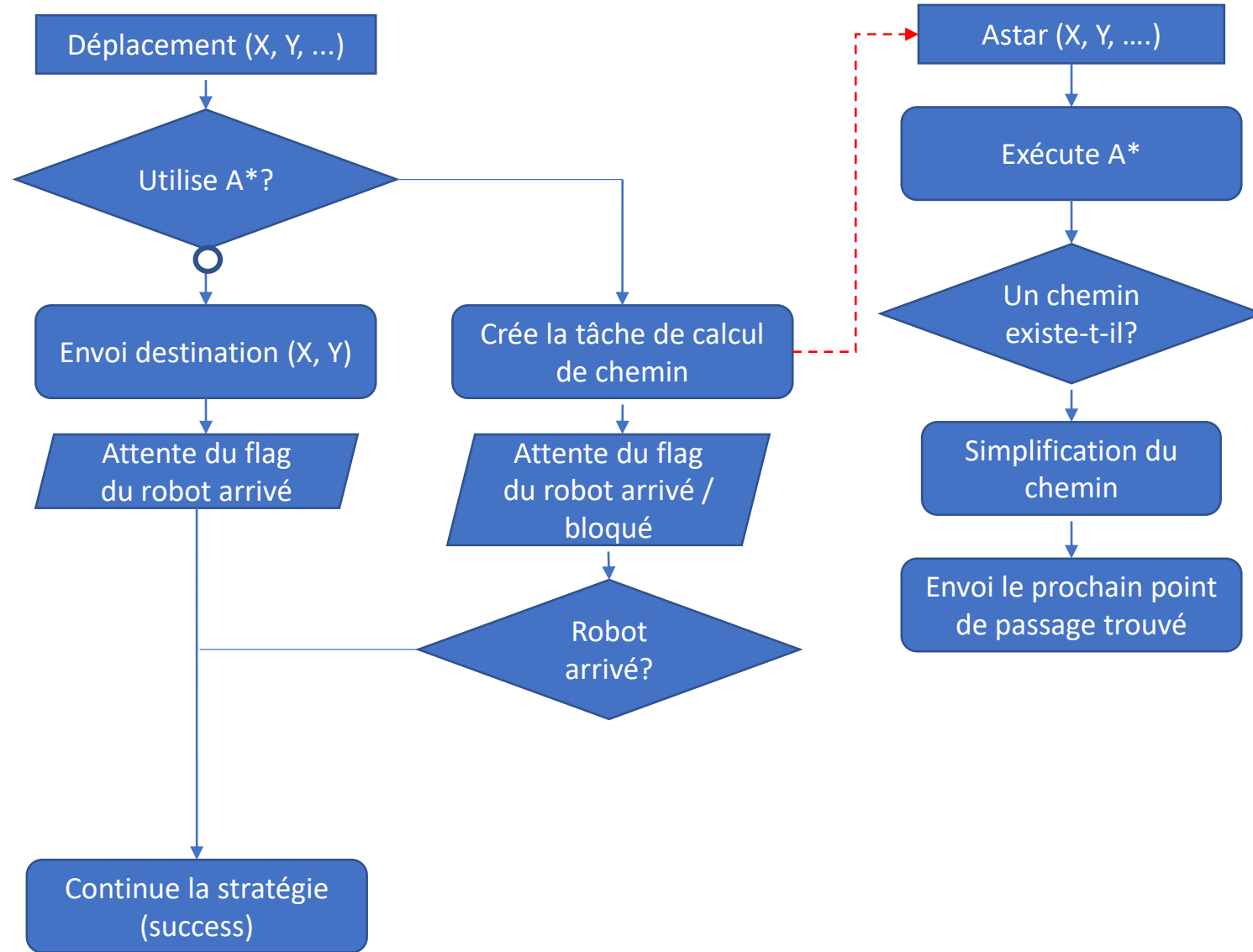
Tâches de Déplacements



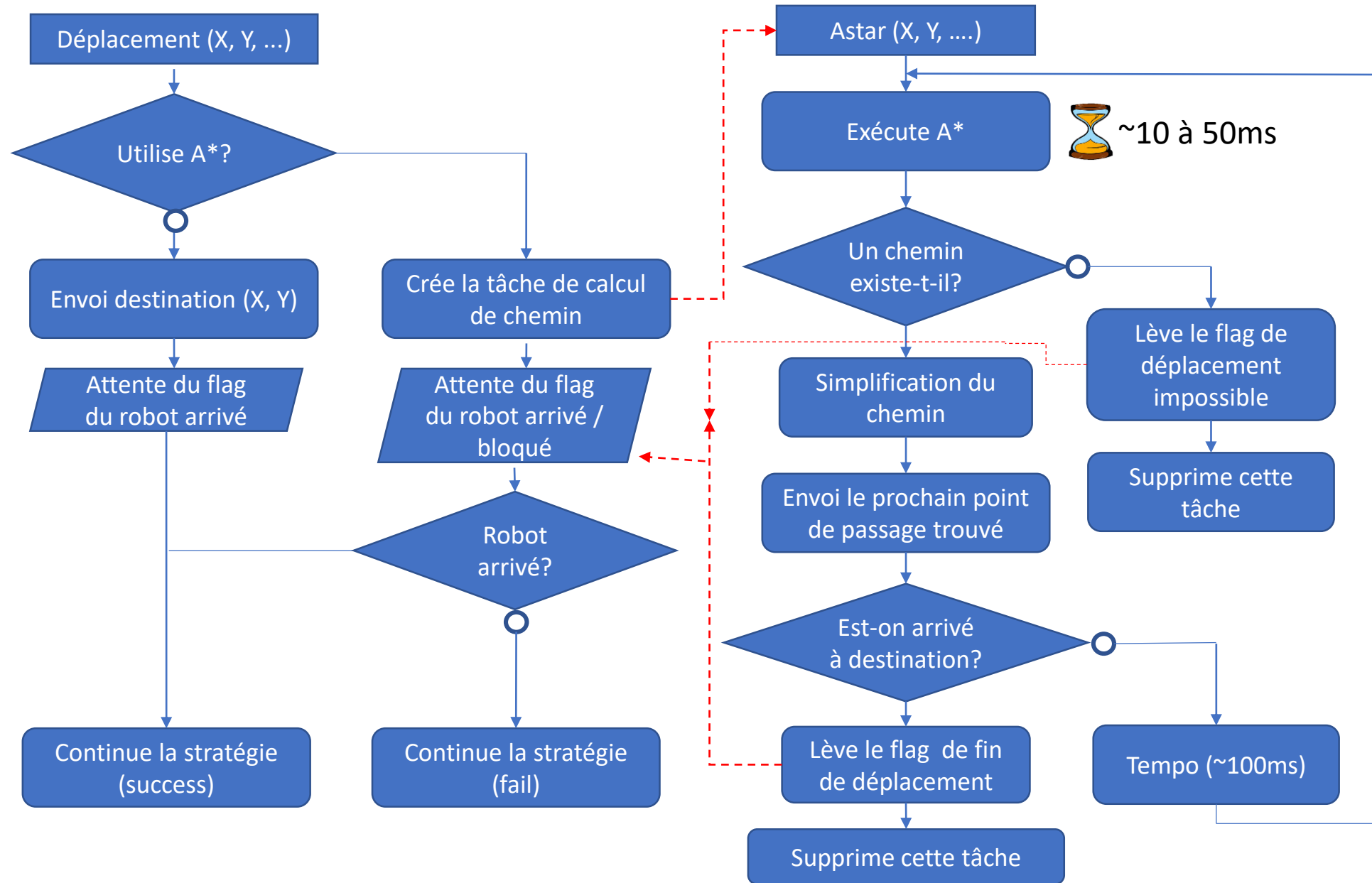
Tâches de Déplacements



Tâches de Déplacements



Tâches de Déplacements



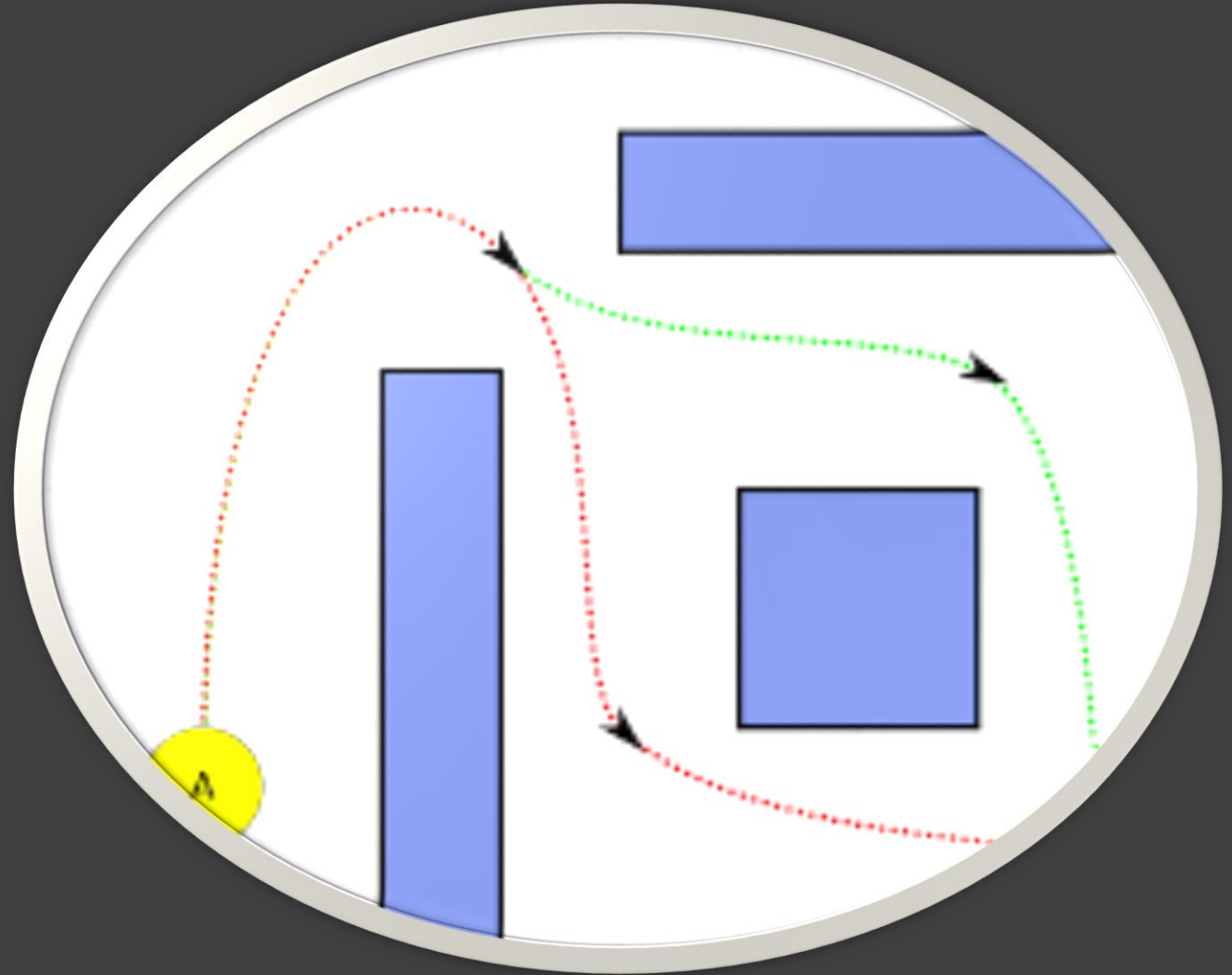
Les déplacements

- Question time

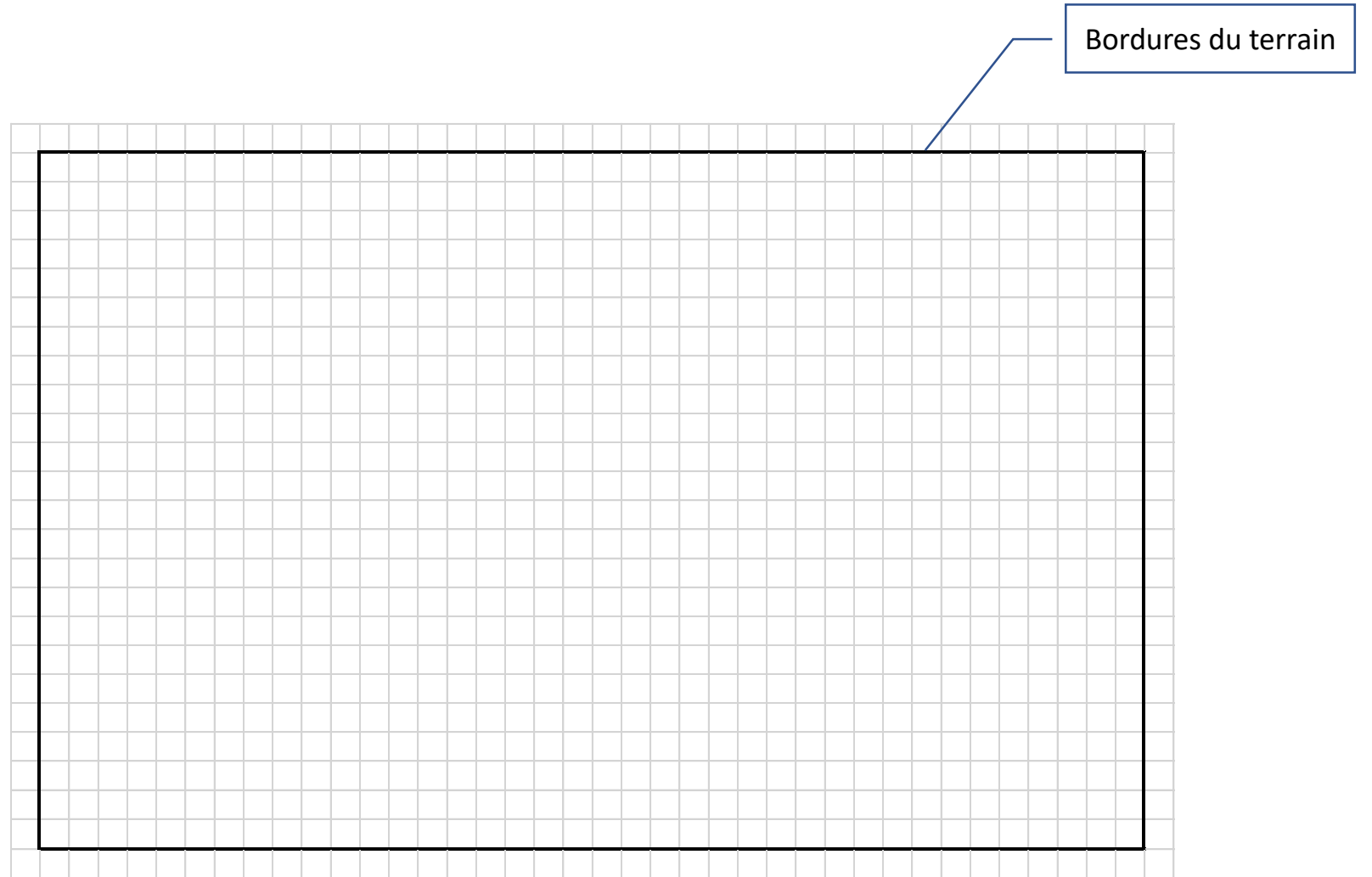


« Pathfinding »

Simplification du trajet

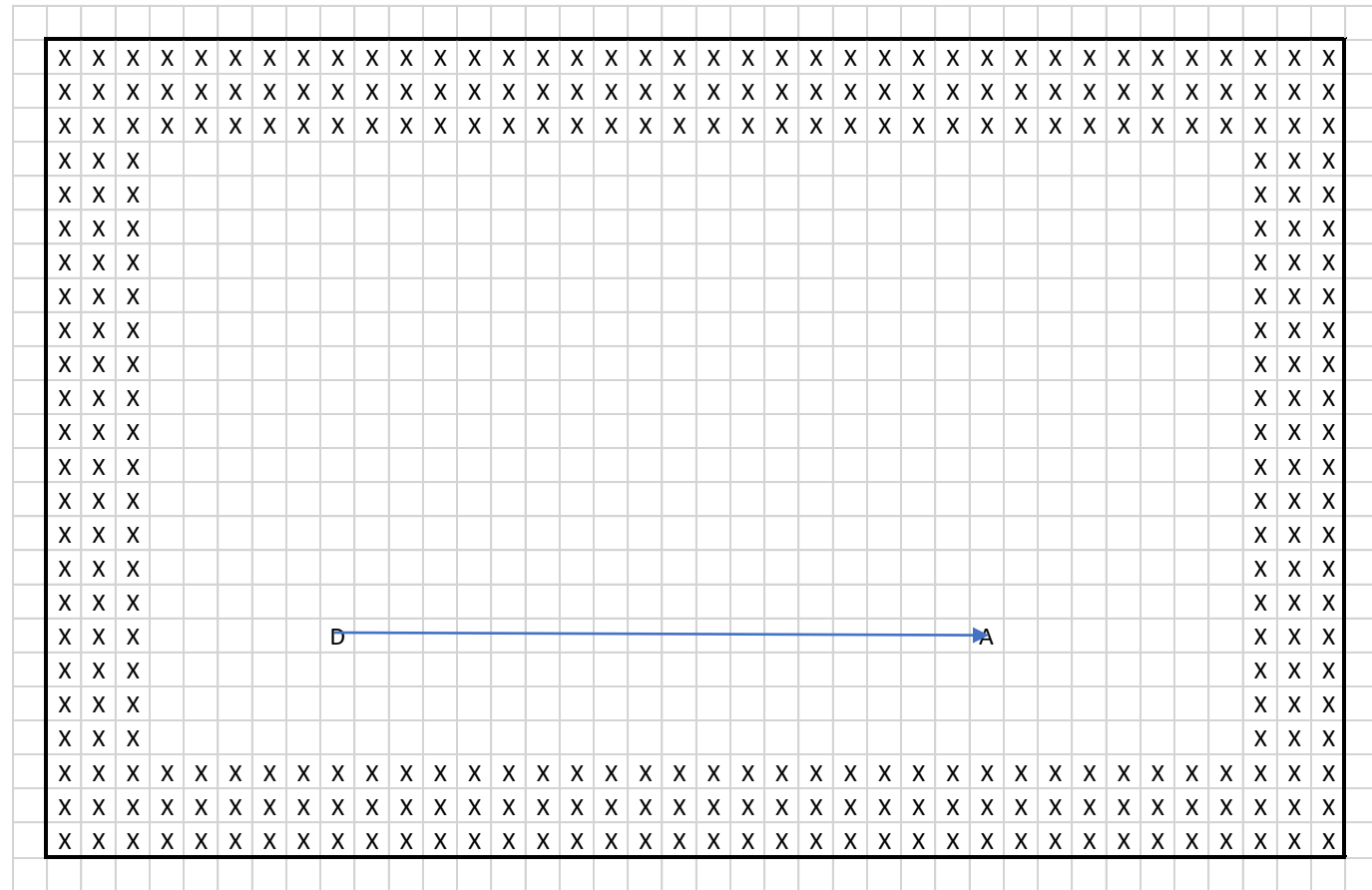


Tâches A*



On part d'une aire de jeu libre d'obstacles et découpée en damier (pas de 50 ou 100mm)

Tâches A*

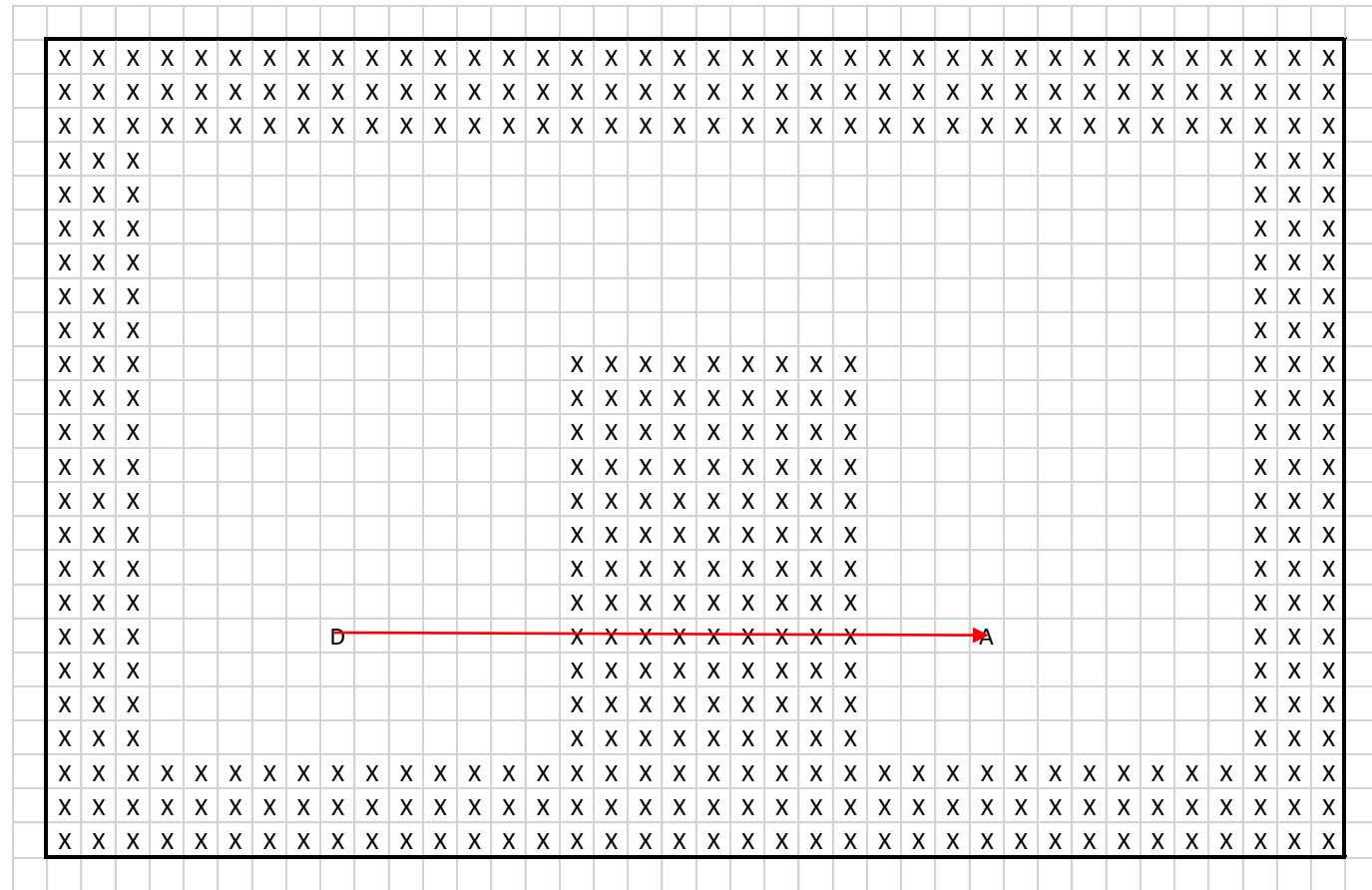


On y ajoute des obstacles (ici les bordures du terrain).

Pour cela on définit certaines cases du damier comme étant inaccessibles (l'image et le damier ne sont pas à l'échelle du terrain réel).

Dans ce cas simple, un déplacement direct est possible.

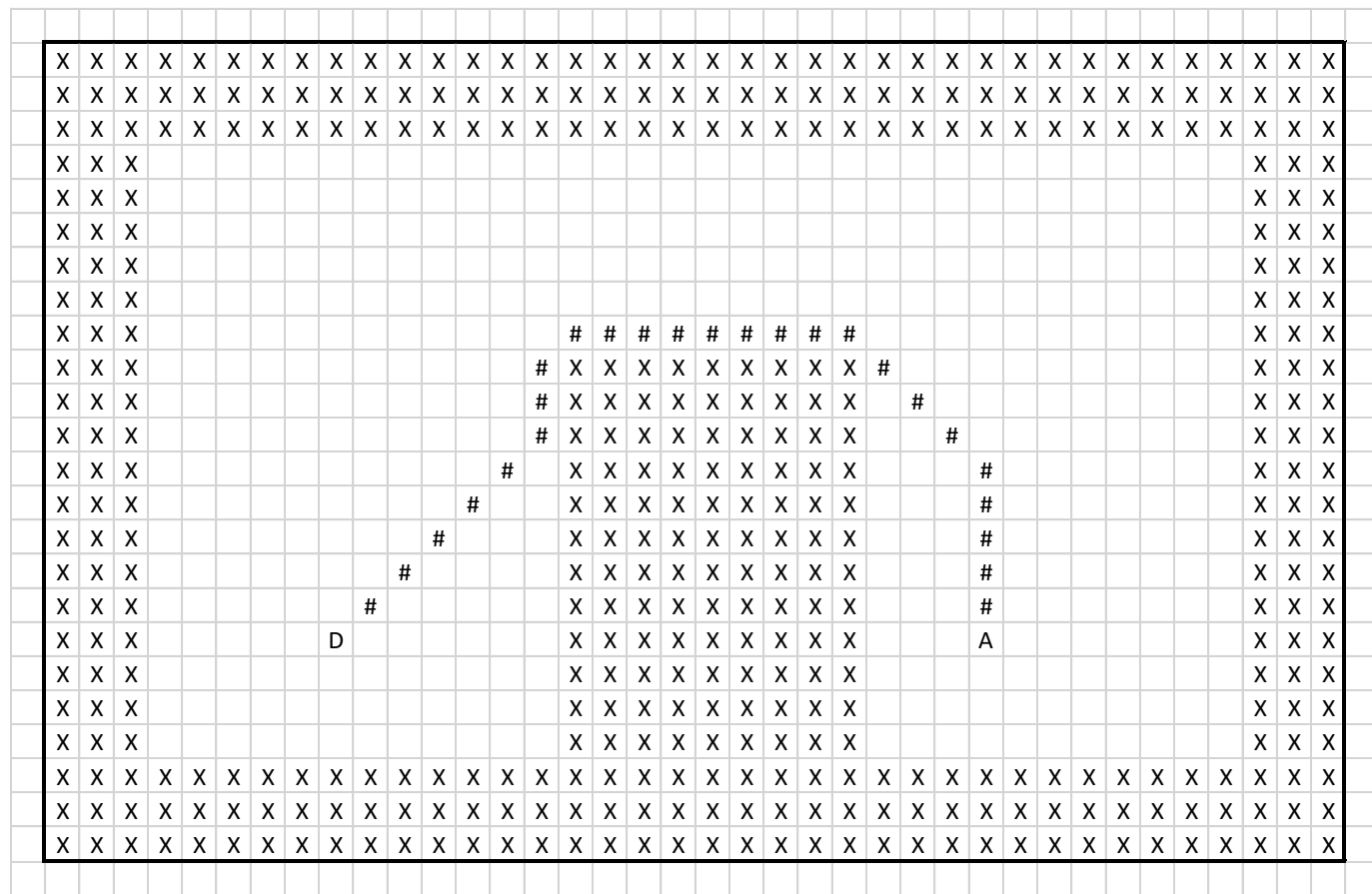
Tâches A*



Admettons que l'on y ajoute d'autres obstacles.

On remarque alors qu'un déplacement direct n'est plus possible

Tâches A*

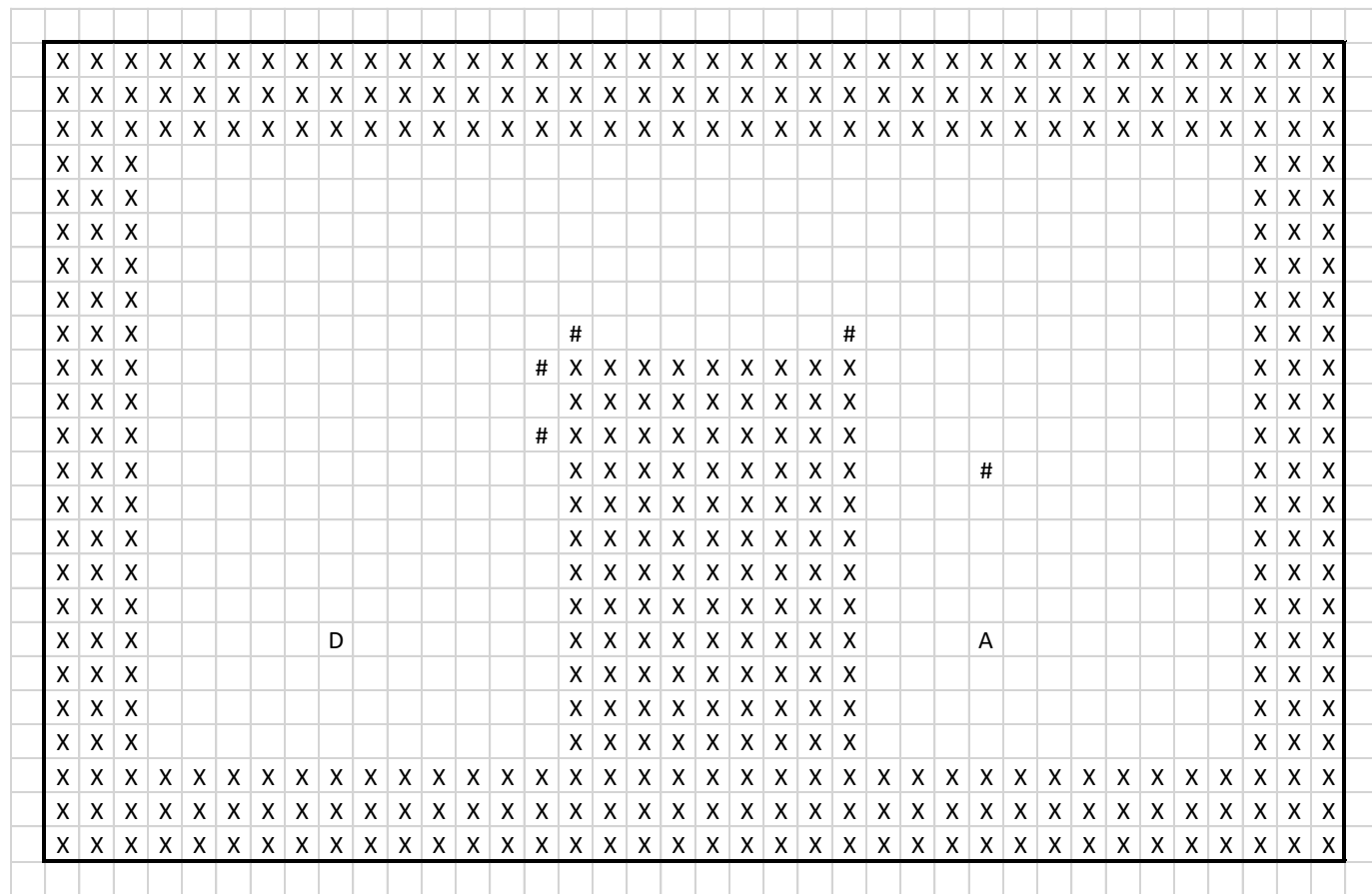


L'algorithme A* va alors calculer un chemin permettant de rejoindre notre destination, tout en évitant les obstacles (si un tel chemin existe, sinon il renverra une erreur).

Il est alors possible de suivre ce chemin, point à point.

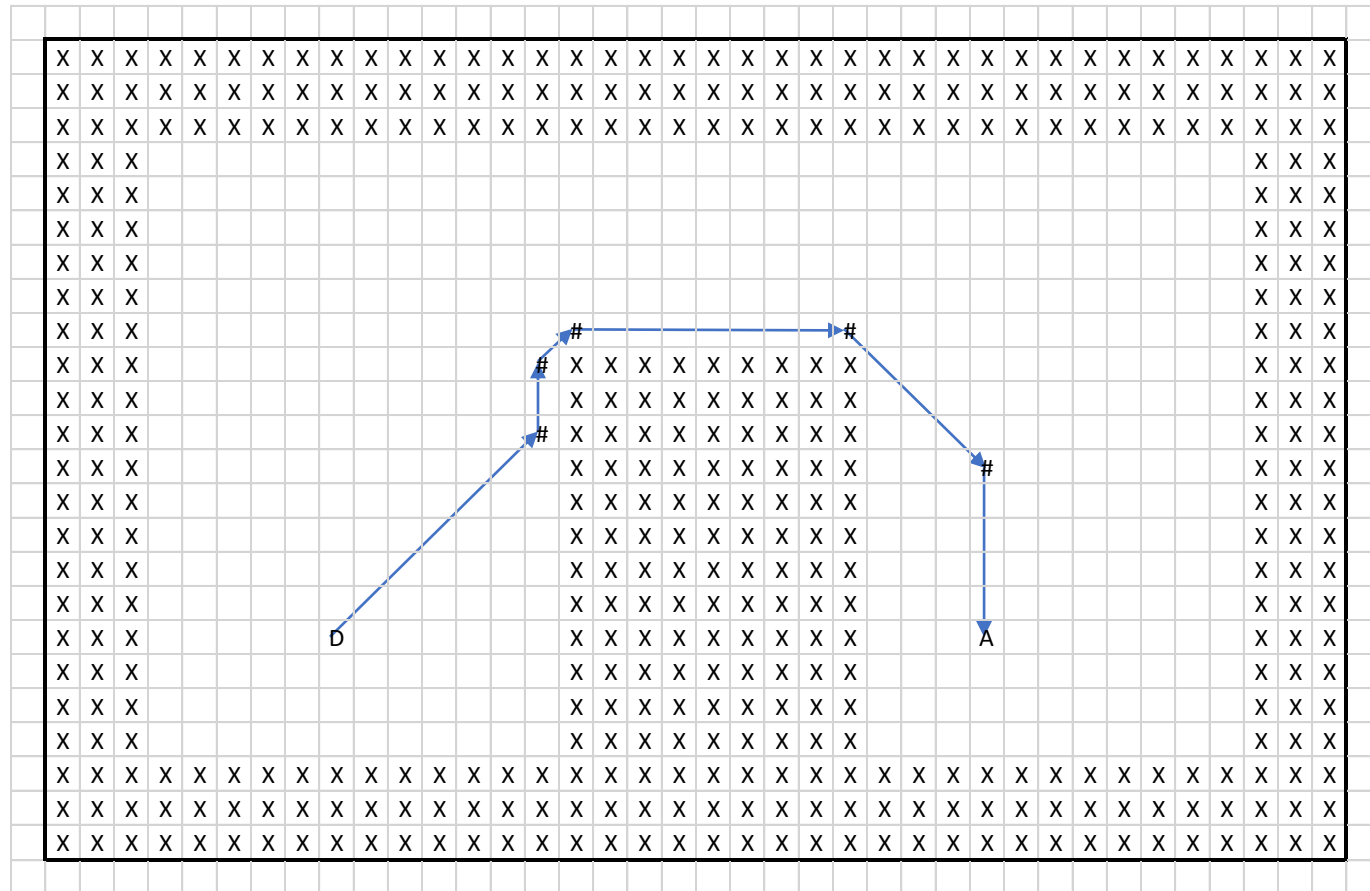
Avec comme inconvénient de faire plein d'arrêts/redémarrages et donc une perte de temps

Tâches A*



Un moyen rapide de simplifier ce chemin, consiste à supprimer les points intermédiaires qui sont alignés et ne garder que les points nécessitant un changement de direction

Tâches A*

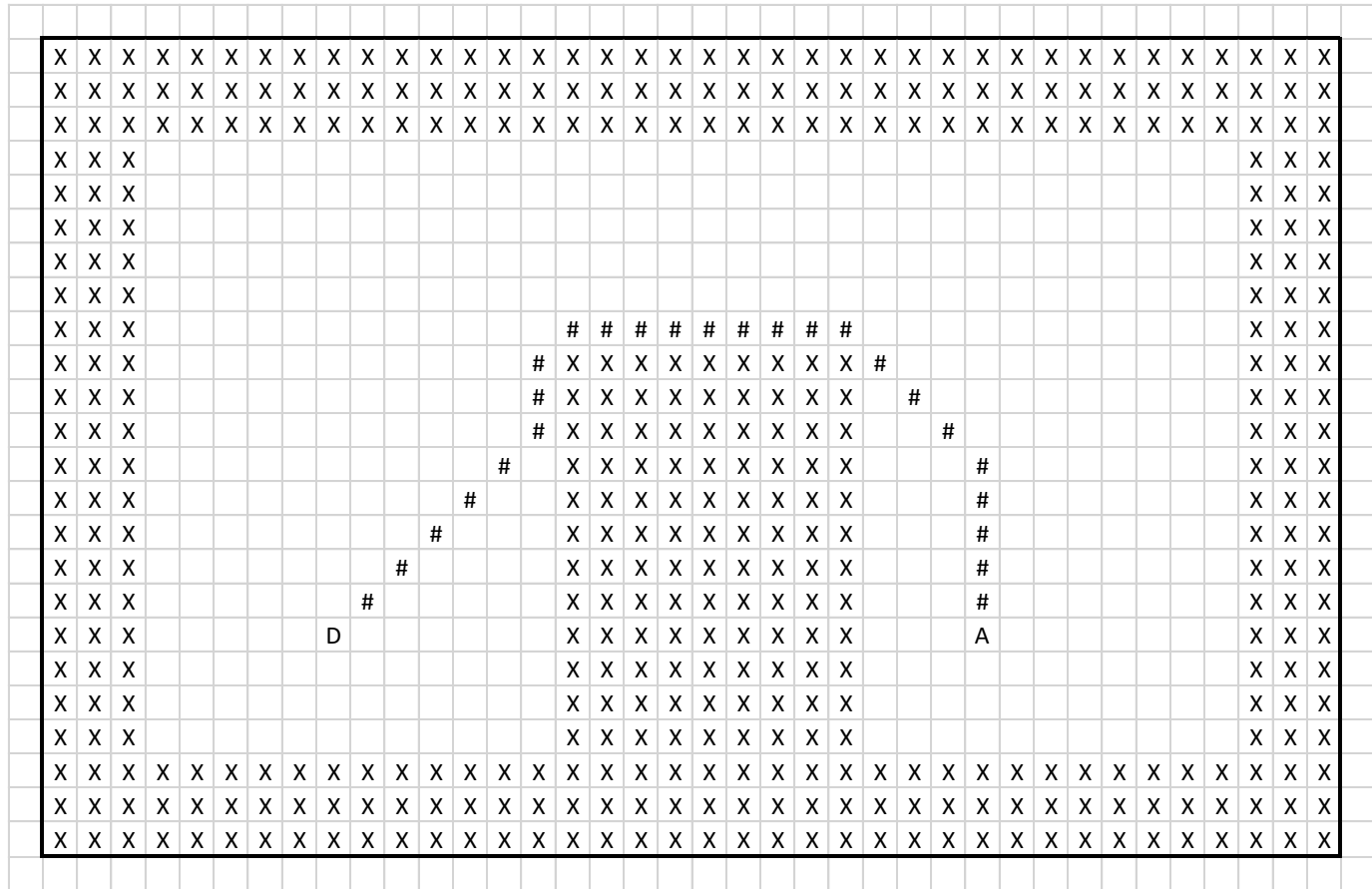


Ce qui donnerait cette trajectoire.

Elle répond au besoin initial d'aller d'un point A à un point B

Tâches A*

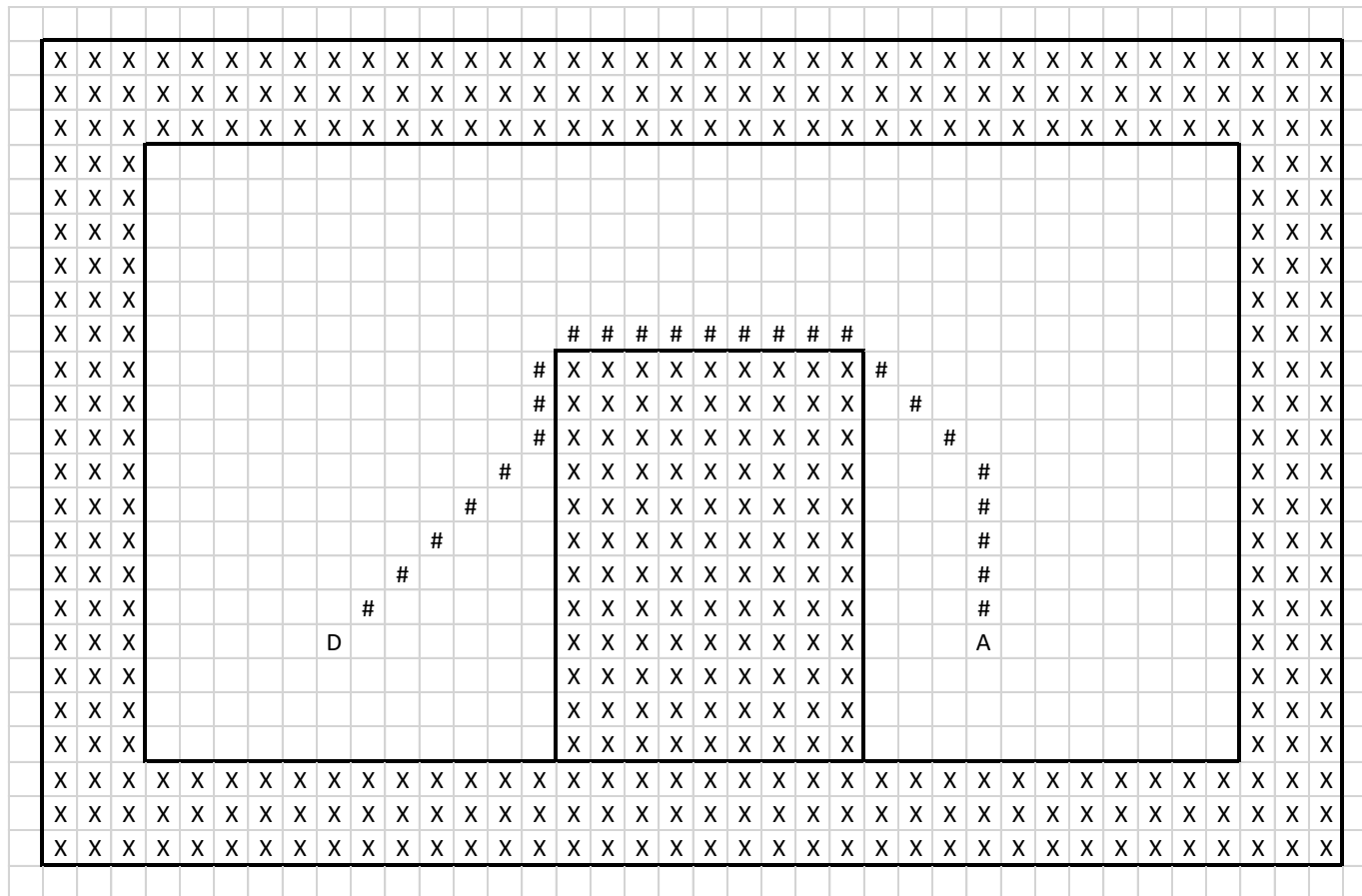
Simplification



Cependant, ce n'est pas le choix que nous avons fait.
Revenons un peu en arrière, avant la première simplification.

Tâches A*

Simplification

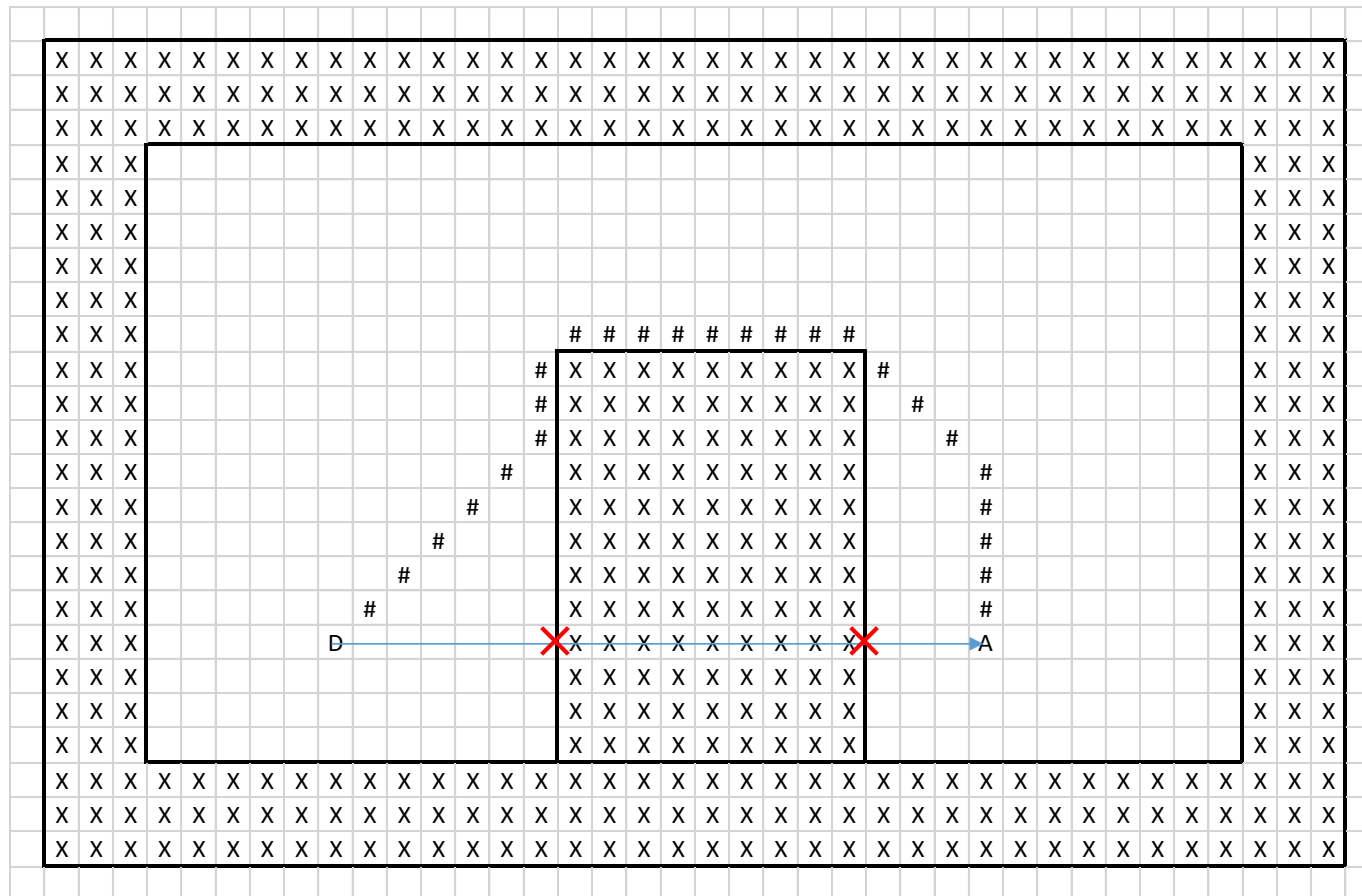


Et ajoutons quelques informations à nos obstacles.

En particulier, pour chacun d'eux, une liste de segments qui représentent les contours de ces obstacles

Tâches A*

Simplification



Puis on va tester s'il existe une intersection entre un segment qui reliera notre robot à sa destination et chacun des autres segments des obstacles.

Comme c'est le cas ici (en rouge), alors on remonte au nœud précédent et on recommence

Tâches A*

Simplification

pseudo-code

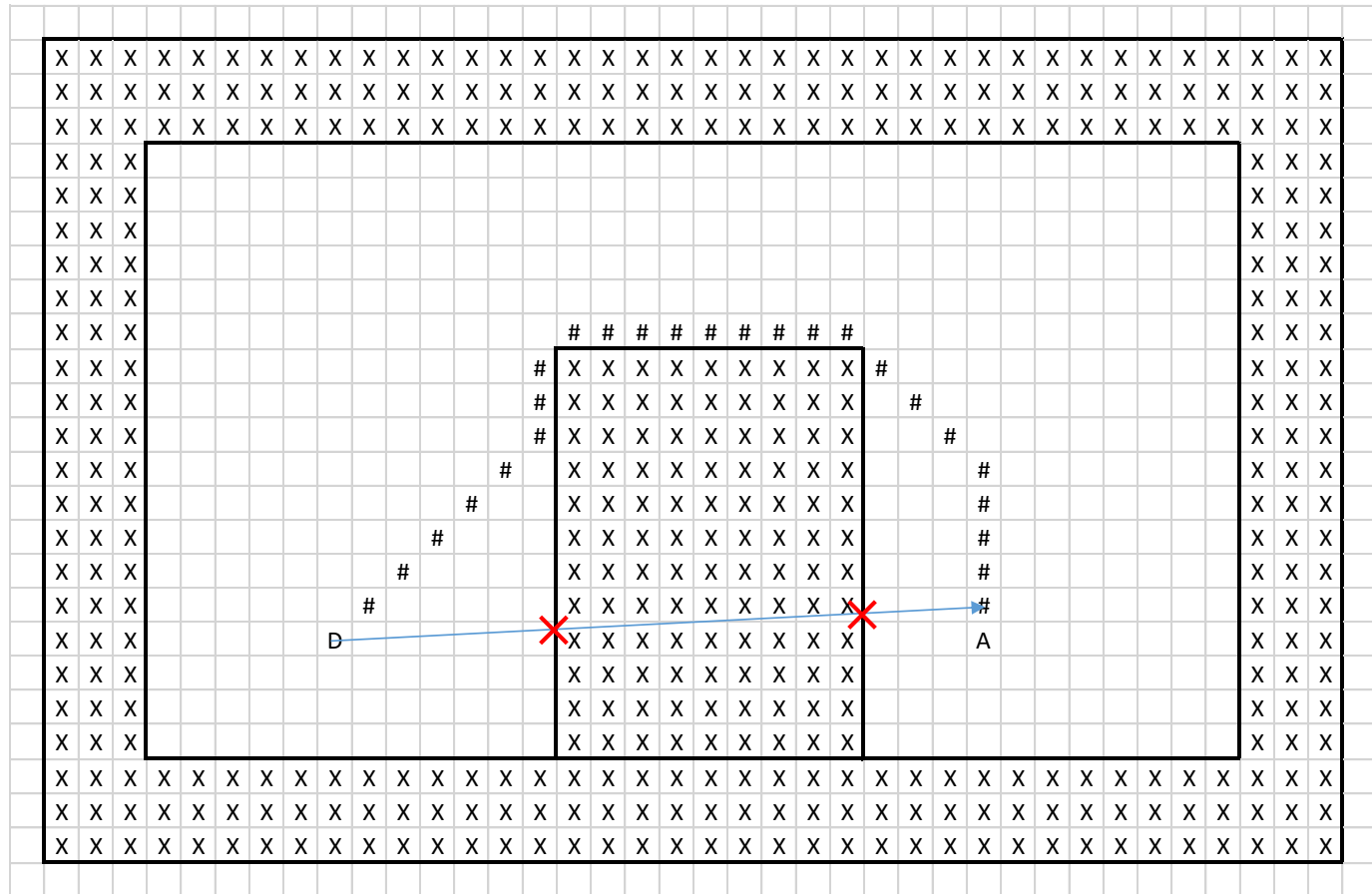
```
//Start with the final point
tested_destination = ASTAR_lastNode;

//Check if direct go is possible
For(int i = 0; i < segmentList.count; i++)
{
    segment seg = segmentsList[i];
    segment objective;
    objective.start = robot.position;
    objective.end = tested_destination;

    if(intersection_segments(seg, objective))
    {
        //There is an intersection
        //Change tested_destination to a closer one, and try again;
        tested_destination = tested_destination.parent;
        i = 0;
    }else
    {
        //No Intersection, this is the best destination
        return tested_destination;
    }
}
```

Tâches A*

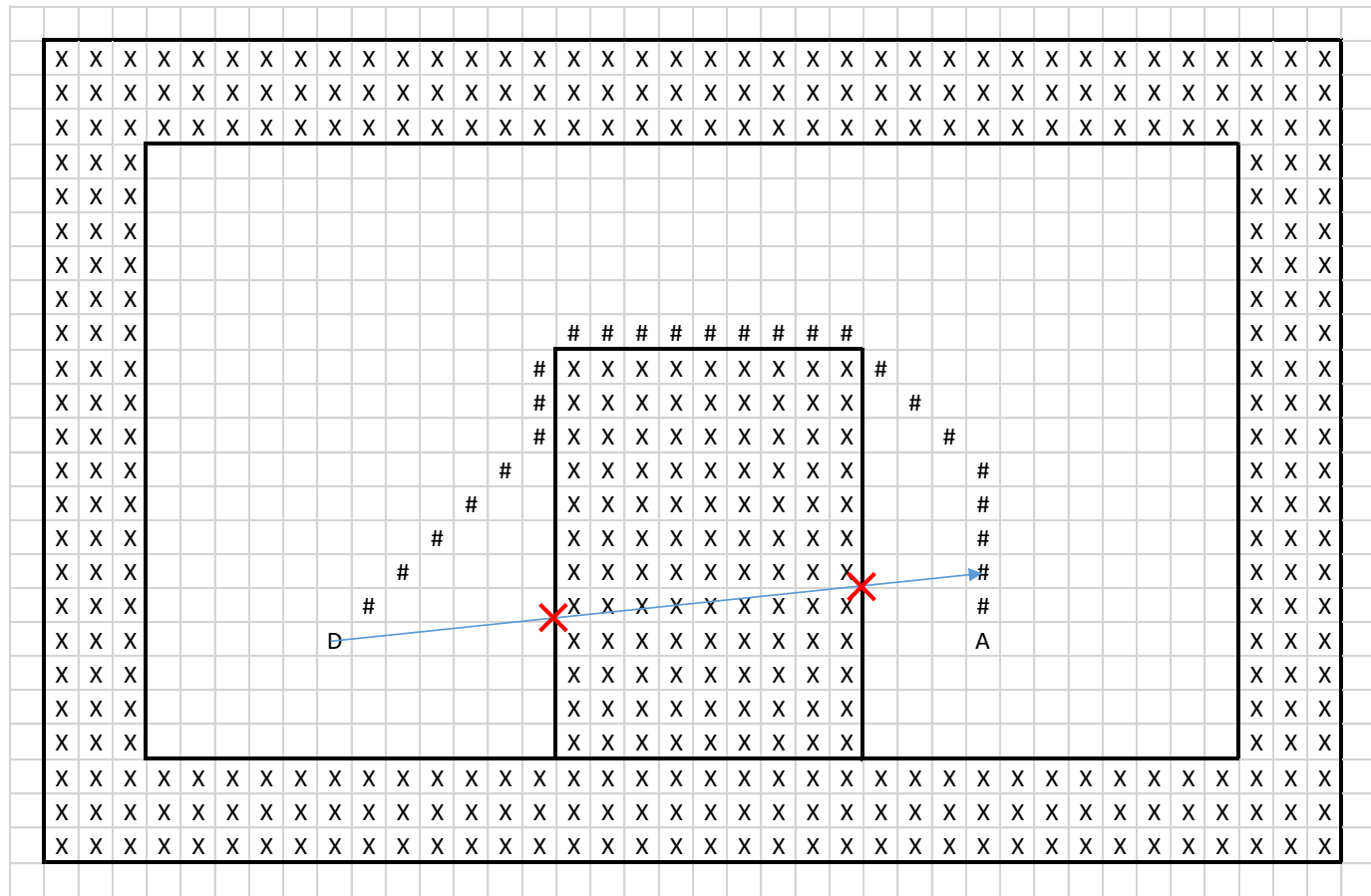
Simplification



Temps qu'il y a intersection, on teste avec un autre point plus en amont dans la trajectoire

Tâches A*

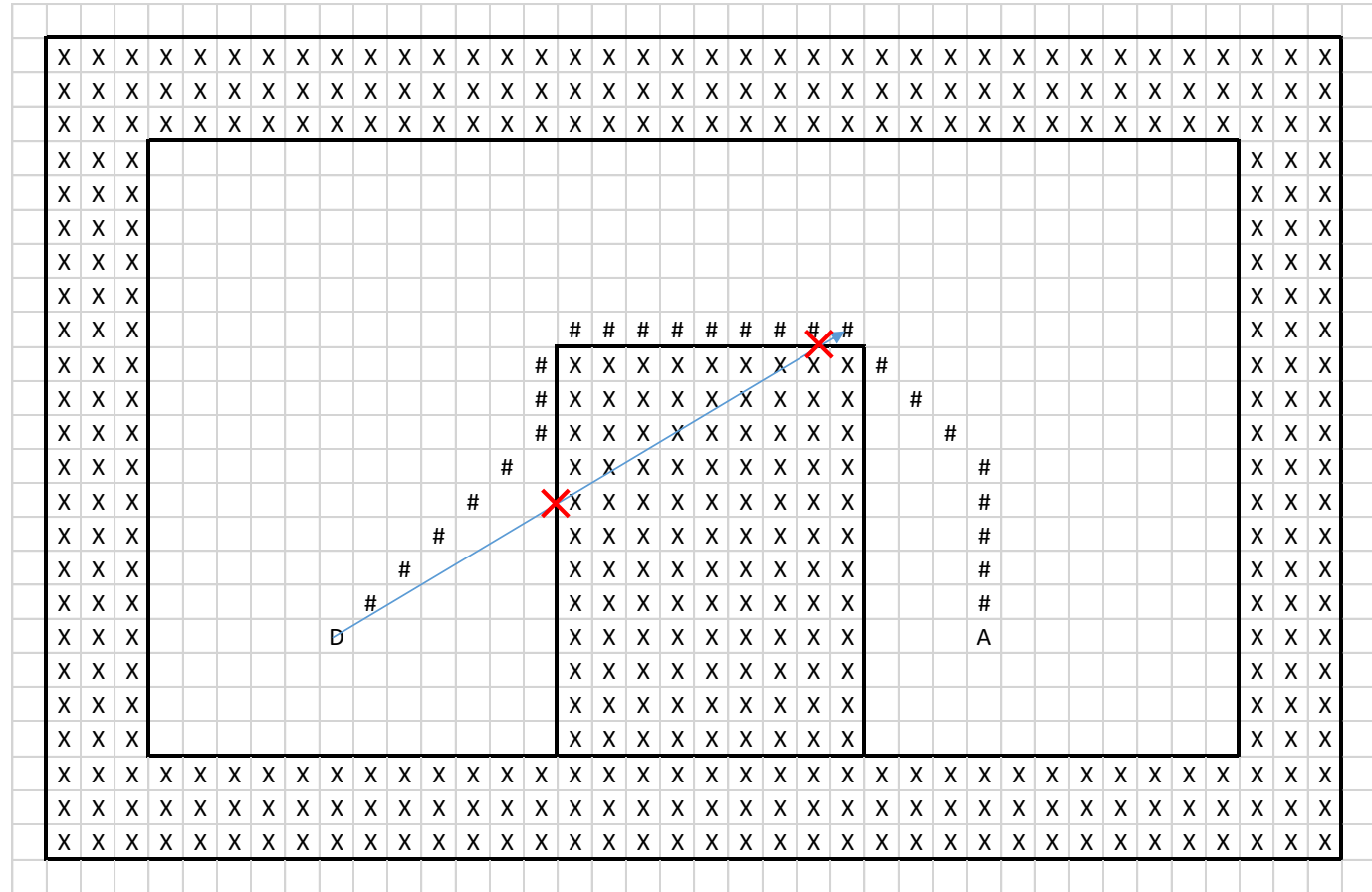
Simplification



On continue à remonter dans l'arborescence des noeuds

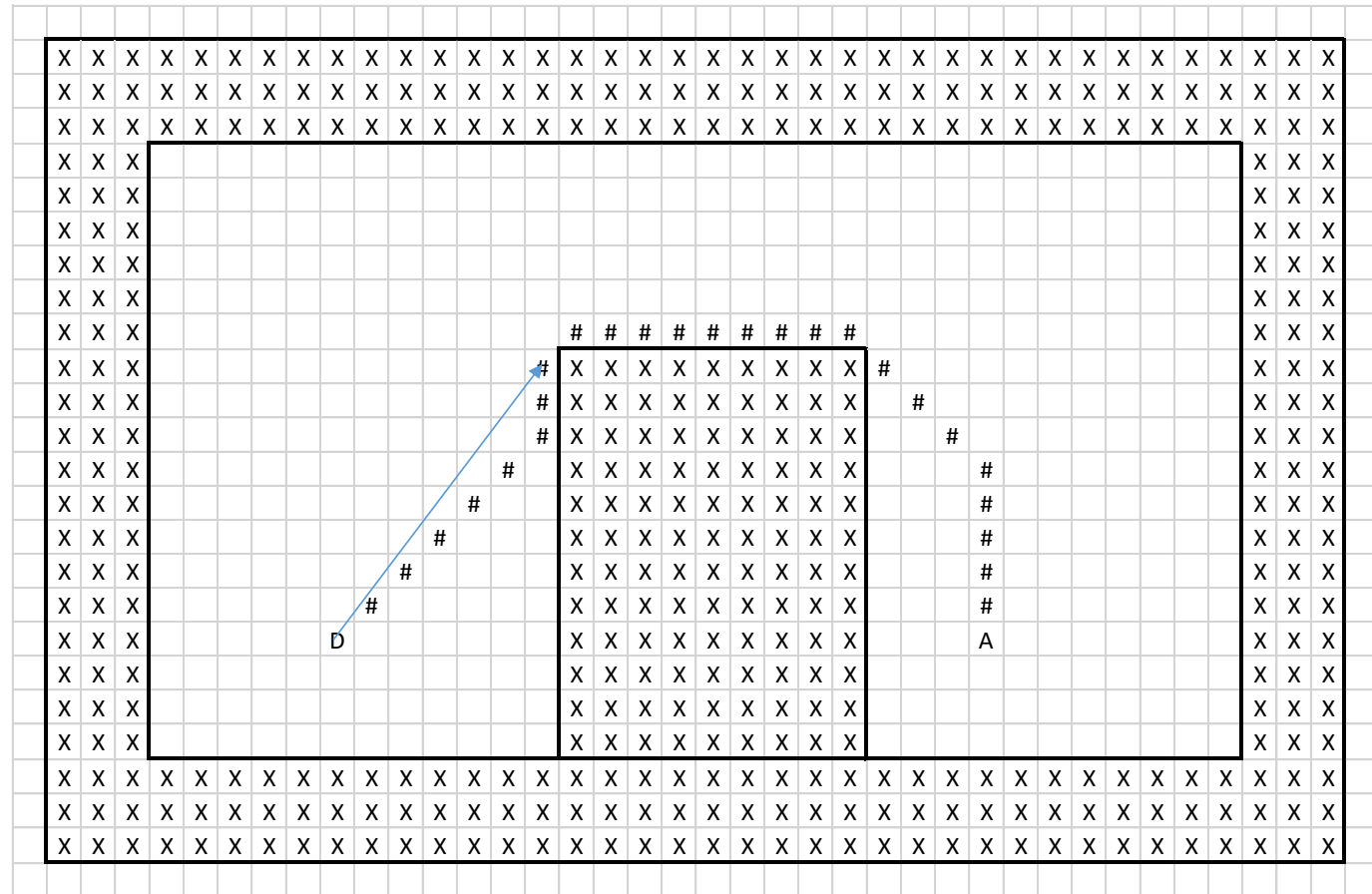
Tâches A*

Simplification



Tâches A*

Simplification



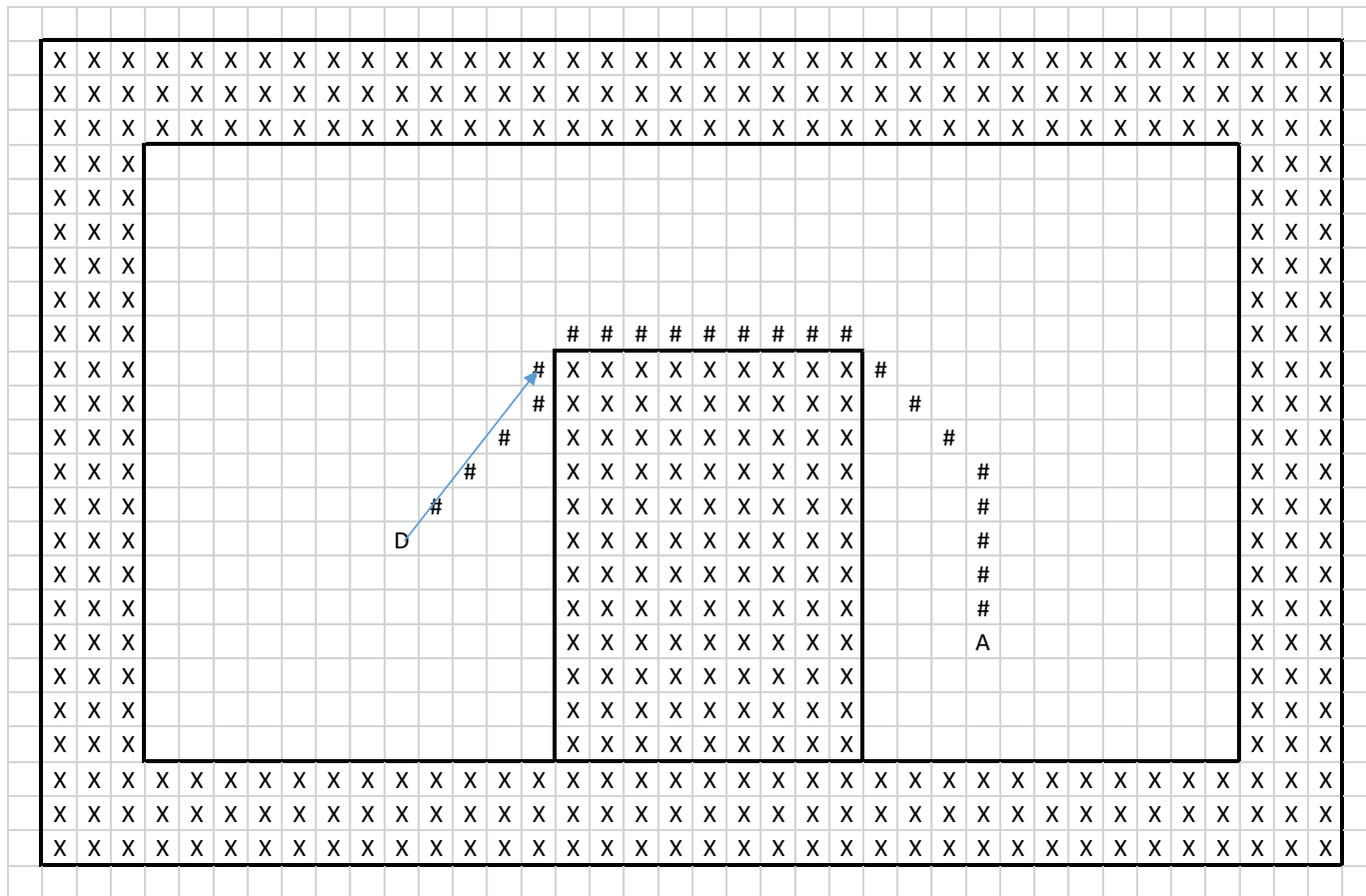
Jusqu'à trouver un point vers lequel le robot peut se rendre en ligne droite sans rencontrer d'obstacle.

Ce sera notre prochaine destination!

Nous ne calculerons pas les suivants, seul le prochain point nous intéresse, les autres on verra après

Tâches A*

Simplification

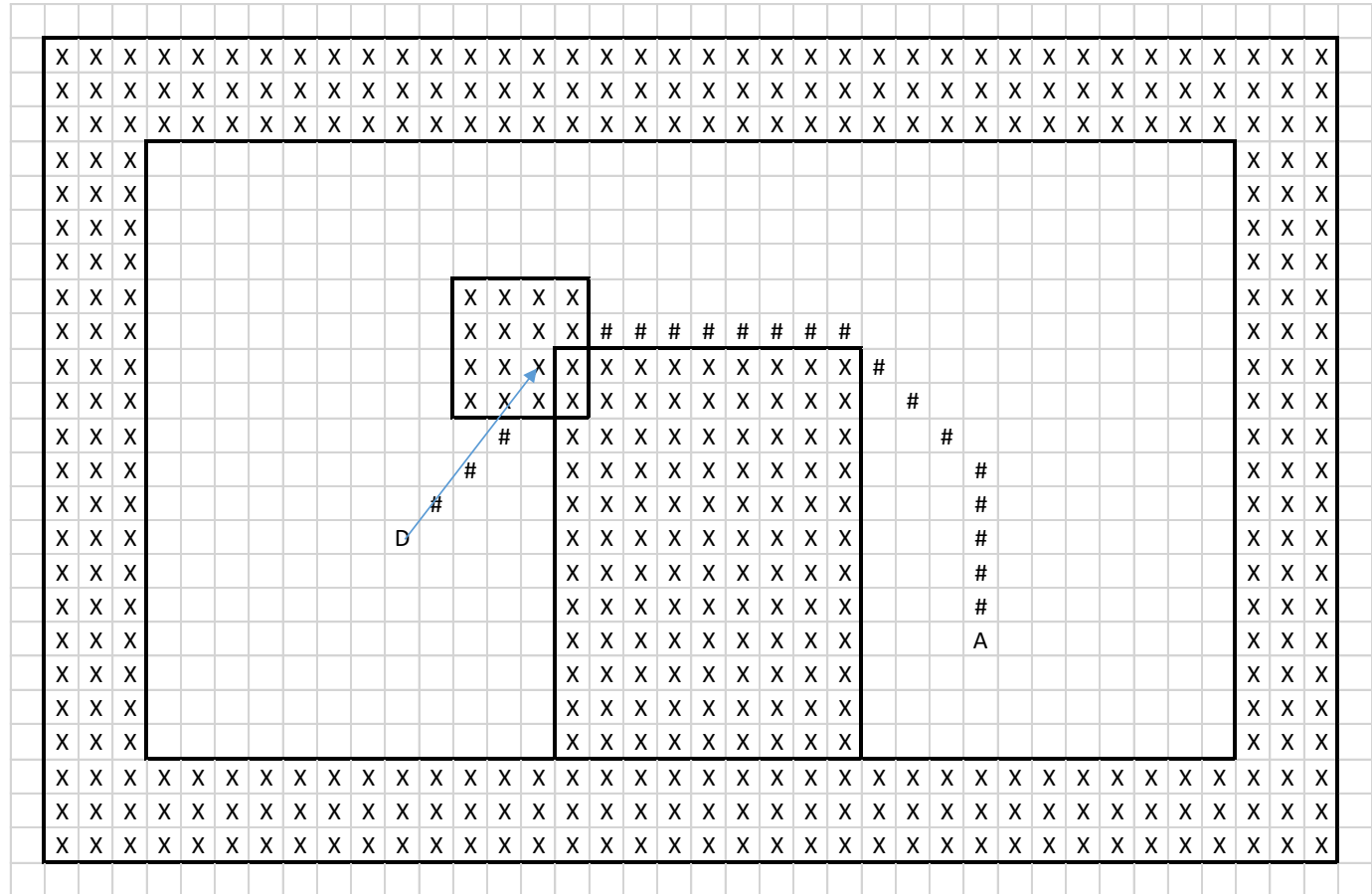


La destination trouvée, elle est envoyée à l'asservissement et le robot commence à s'y rendre. L'algorithme continuera à calculer un chemin optimal à intervalles réguliers (100ms)

Tâches A*

Evitement

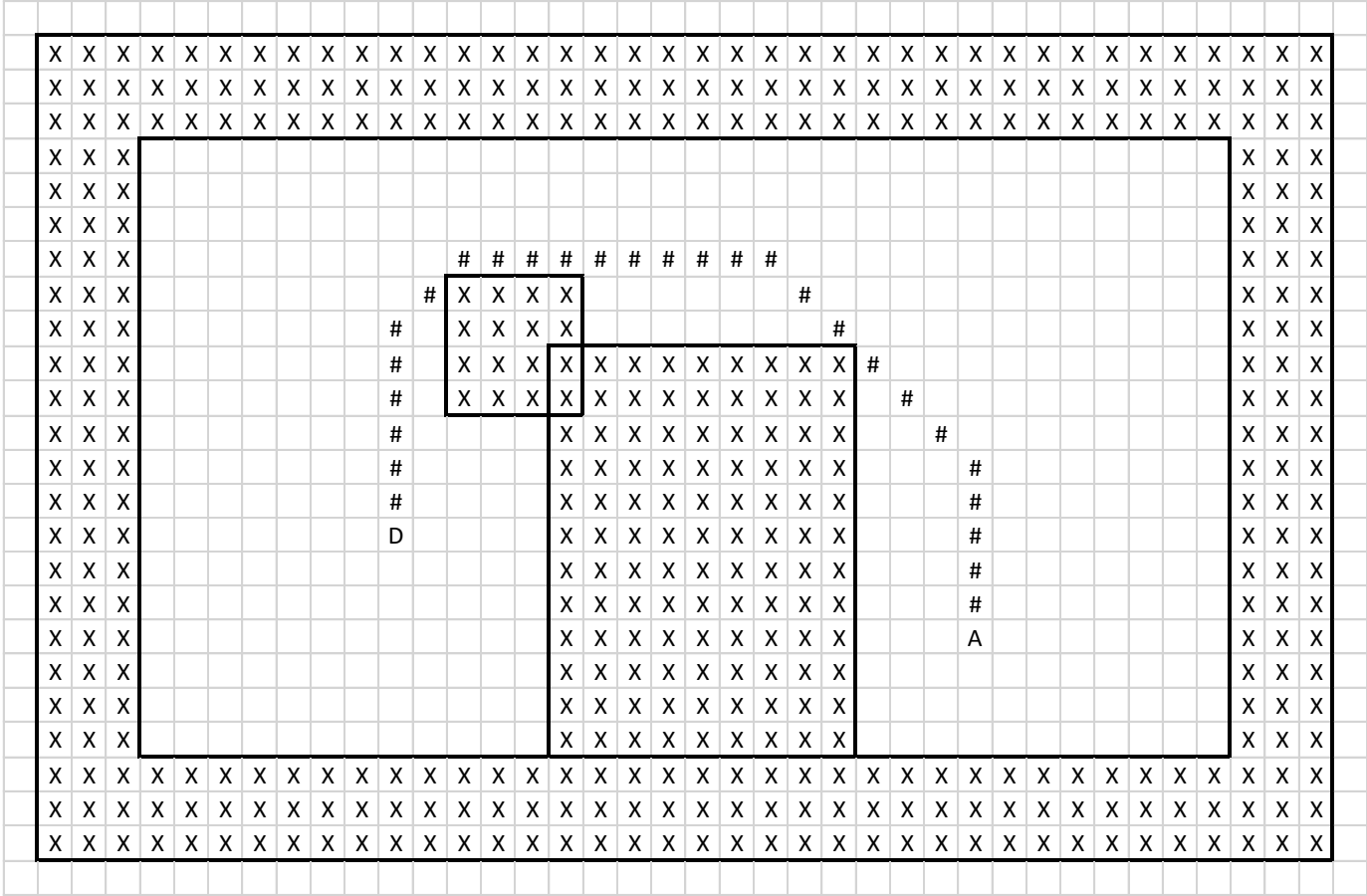
temps réel



Imaginons maintenant qu'un adversaire vienne se placer sur notre route.
Le chemin précédemment calculé n'est alors plus exploitable, on ordonne au robot de s'arrêter.

Tâches A*

Evitement temps réel

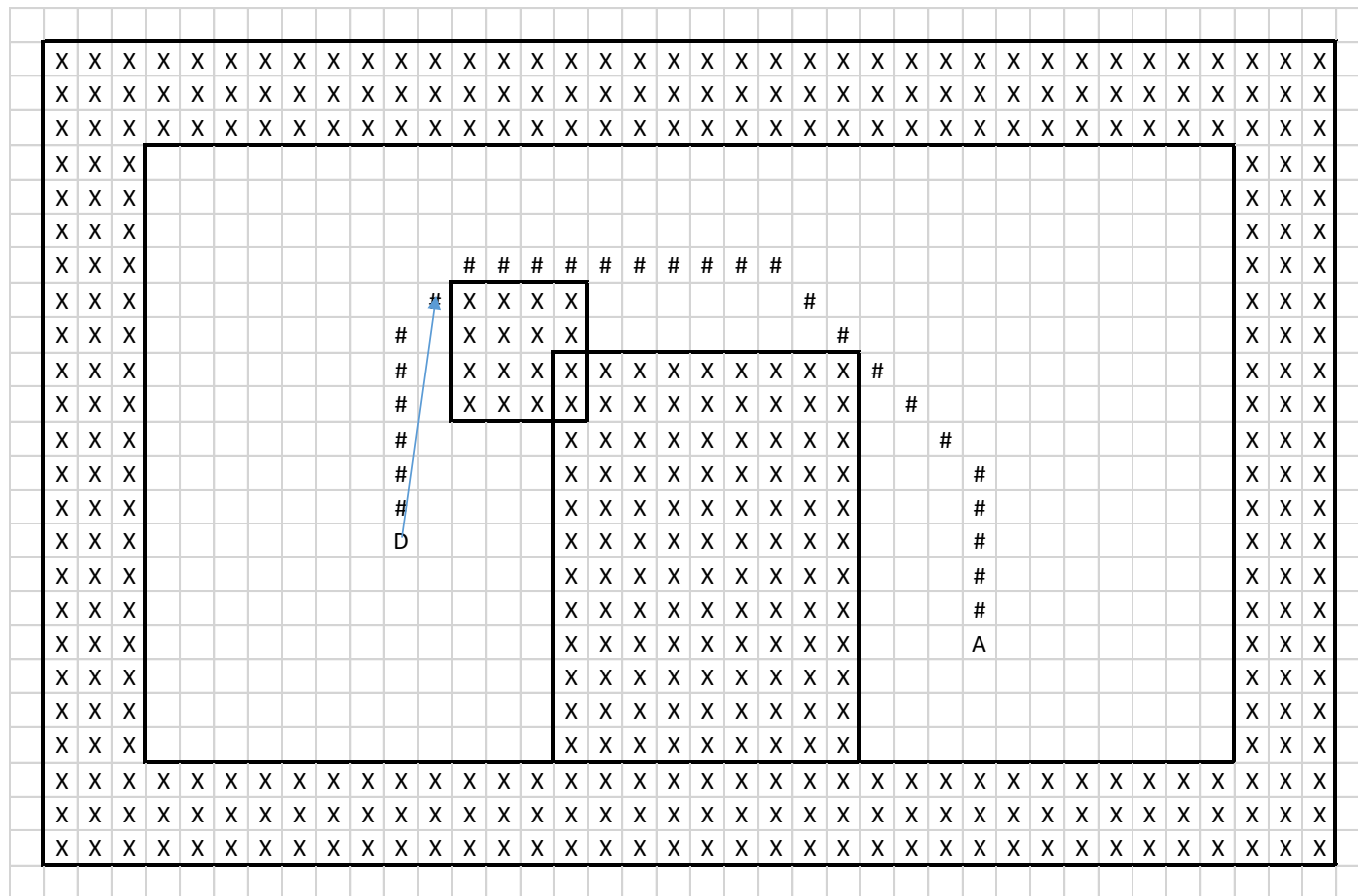


Une nouvelle trajectoire est calculée lors de la prochaine boucle du pathfinder
Au final, ce n'est pas un problème, puisqu'il existe un autre chemin qui contourne cet obstacle

Tâches A*

Evitement

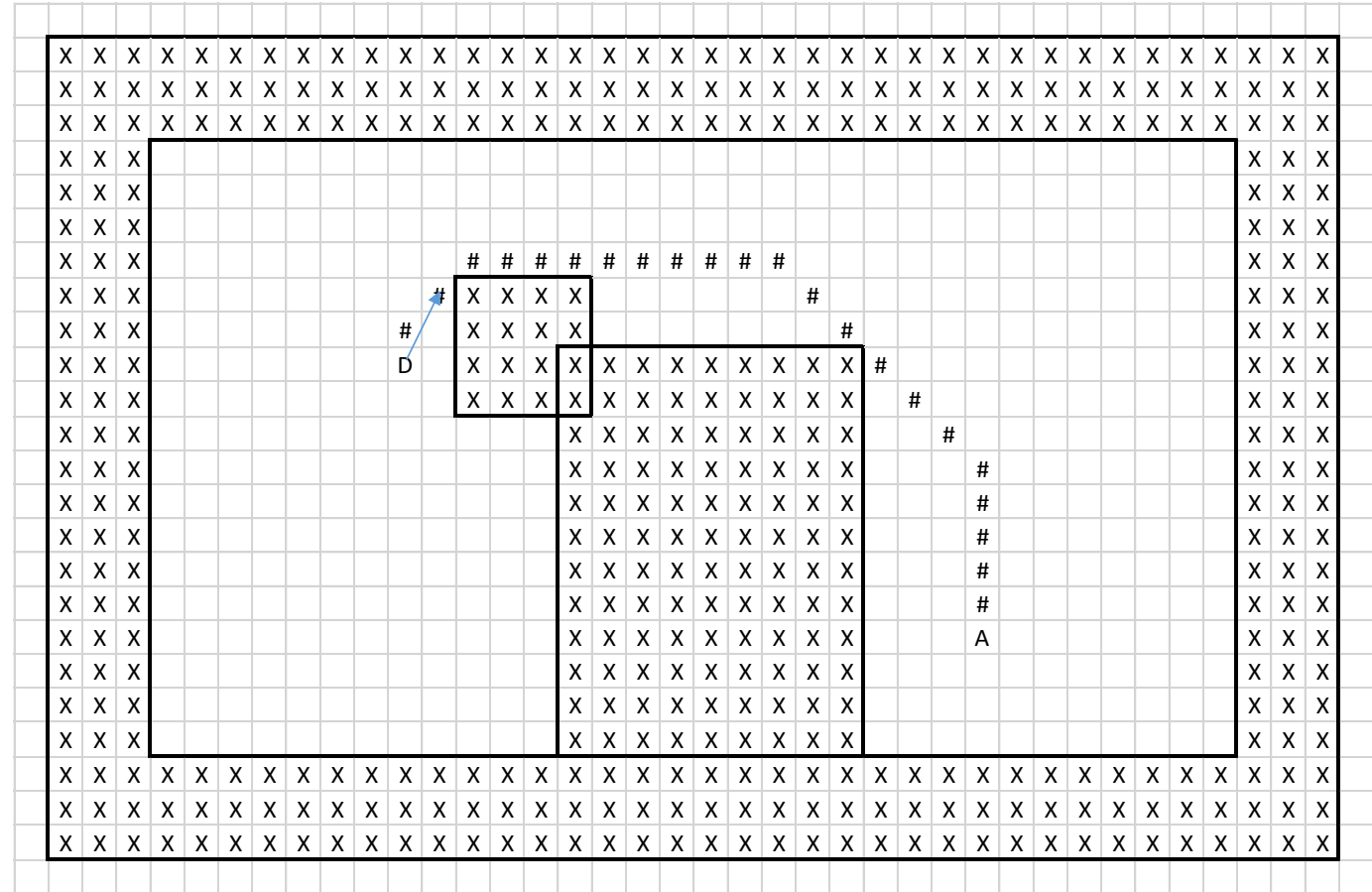
temps réel



De la même façon, on cherche le point de passage le plus avancé sur le nouveau chemin vers lequel nous rendre directement

Tâches A*

Evitement temps réel

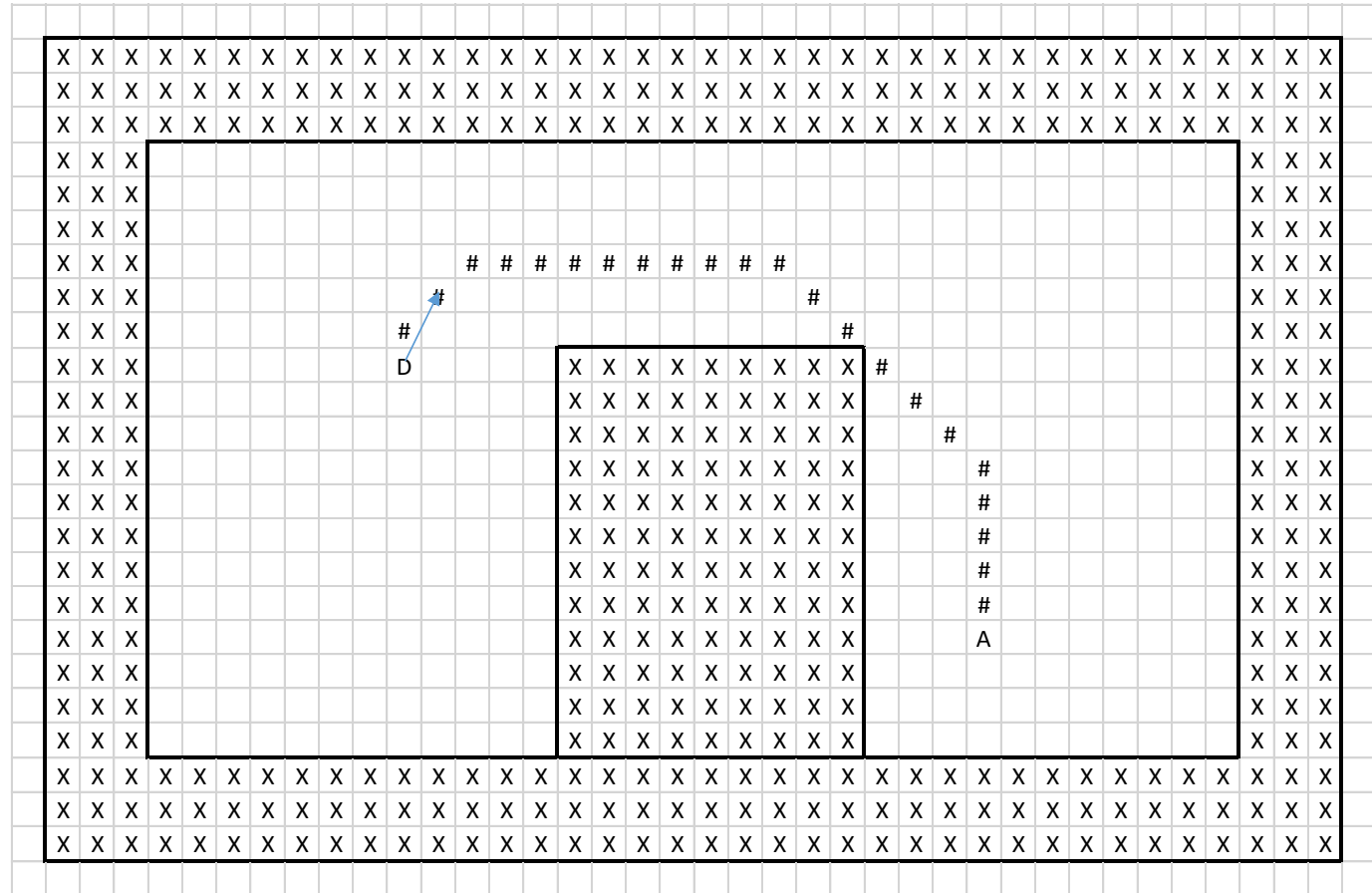


Et nous voilà partis à contourner cet obstacle...

Tâches A*

Evitement

temps réel

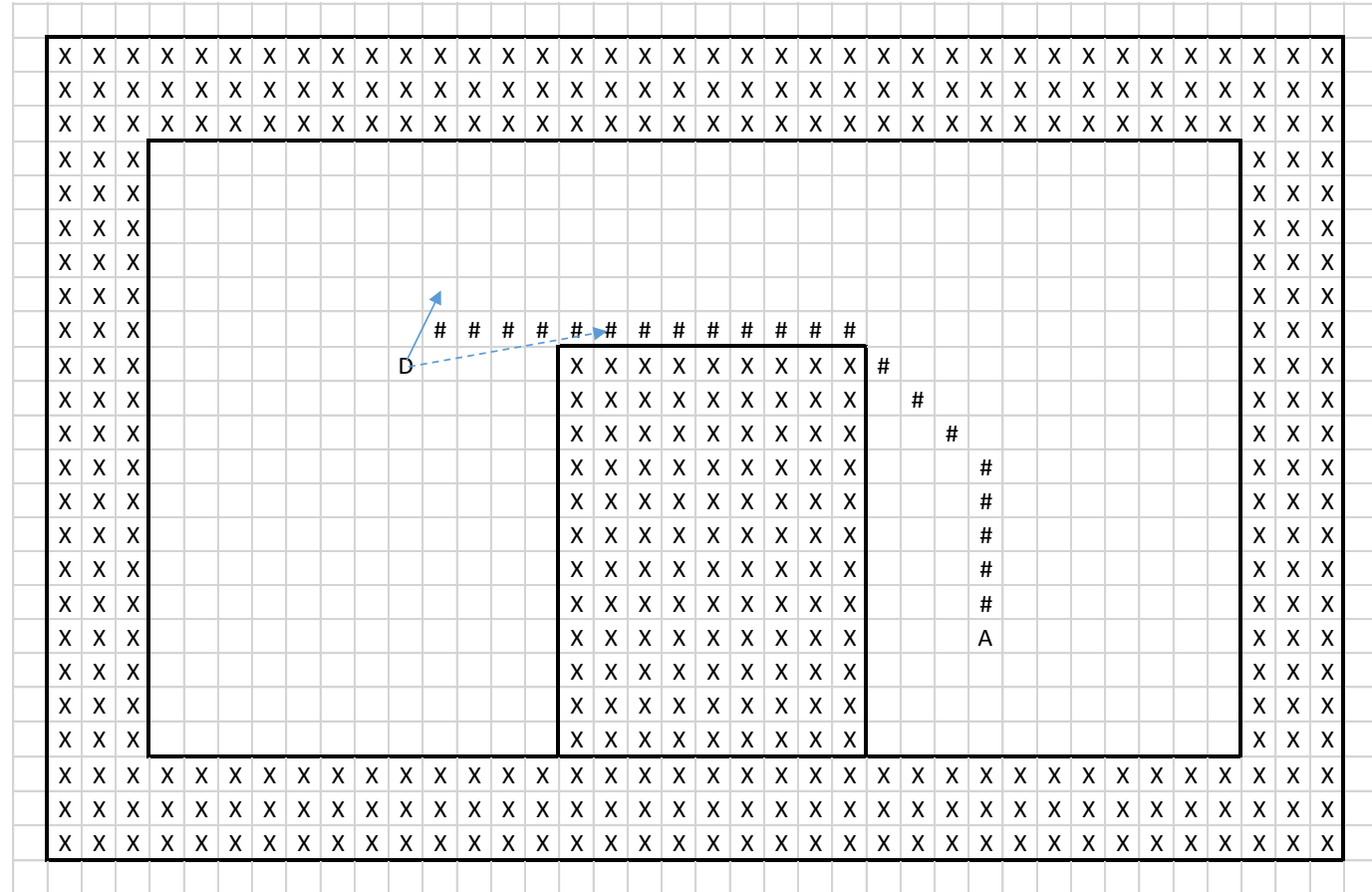


Alors que l'obstacle disparaît tout à coup (peut-être était-ce un faux robot téléporté pour l'homologation? (y'a des arbitres qui font ça! Si si!!!!) On avait pourtant dit à l'arbitre de ne pas téléporter son robot en bois!....(Il le font quand même)

Pas grave, on continue notre chemin comme planifié

Tâches A*

Evitement temps réel

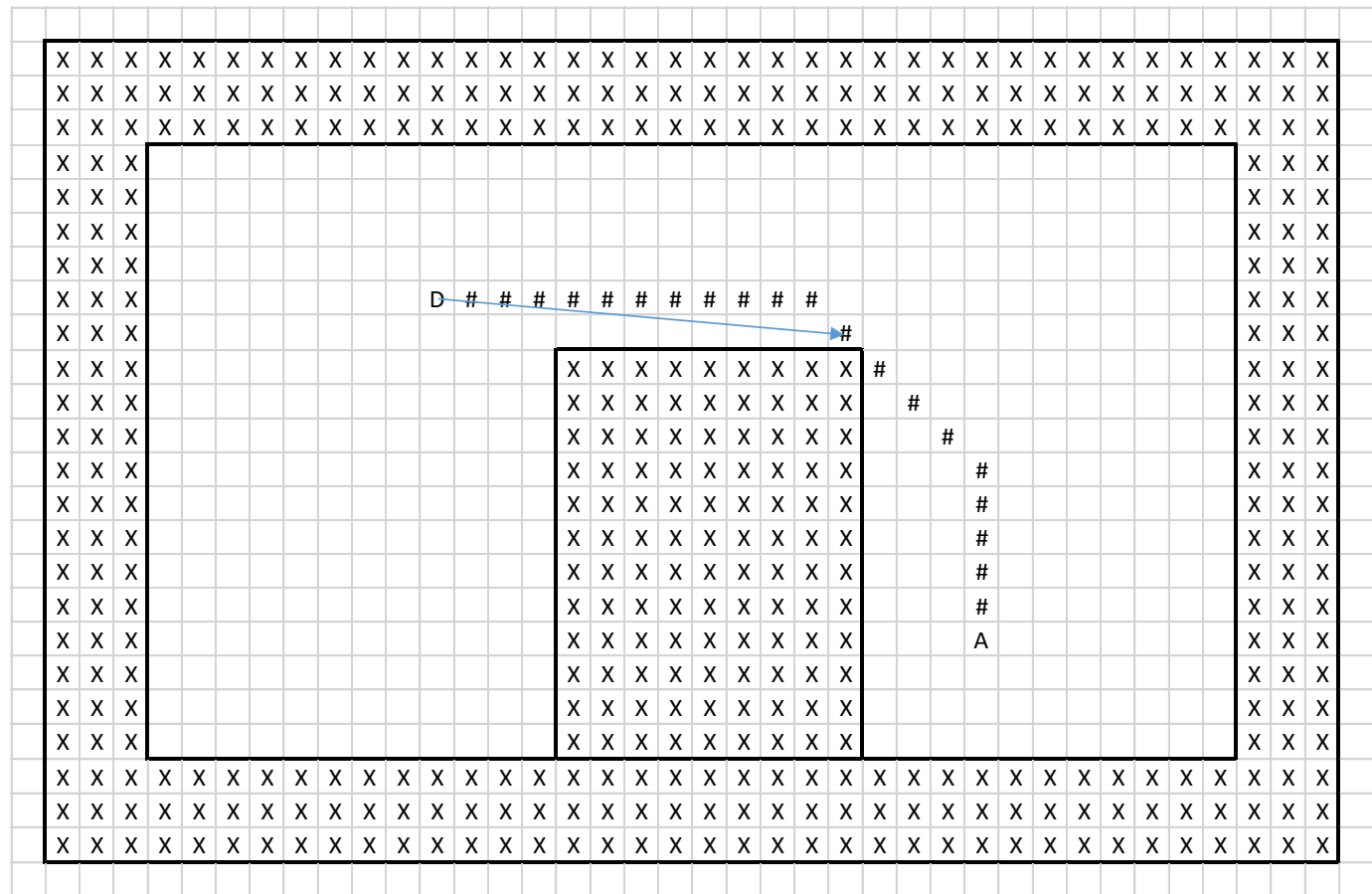


Le A* nous dit pourtant que ce n'est pas le chemin optimal pour rejoindre notre destination. Aucune importance, on conserve le cap temps que notre route actuelle ne rencontre pas d'obstacle.

Cela nous évitera d'osciller entre deux destinations si un robot adverse vient se frotter à nous

Tâches A*

Evitement temps réel



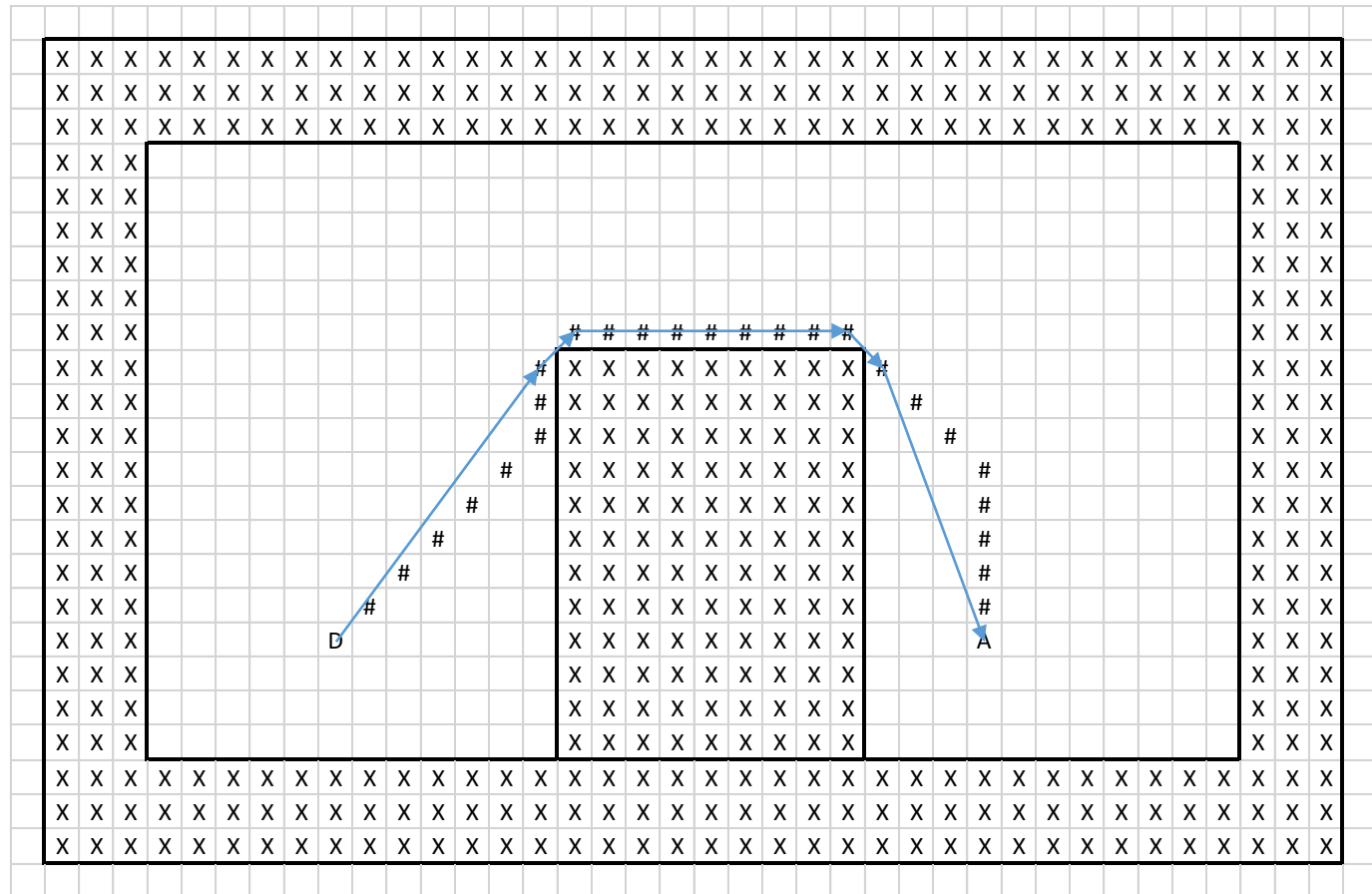
Une fois arrivé à notre point de passage précédent, il est temps de recalculer un nouveau chemin et de continuer vers notre destination.

Le processus recommence jusqu'à ce que nous atteignons notre destination ou qu'il n'y ai pas de chemin possible

Tâches A*

Evitement

temps réel



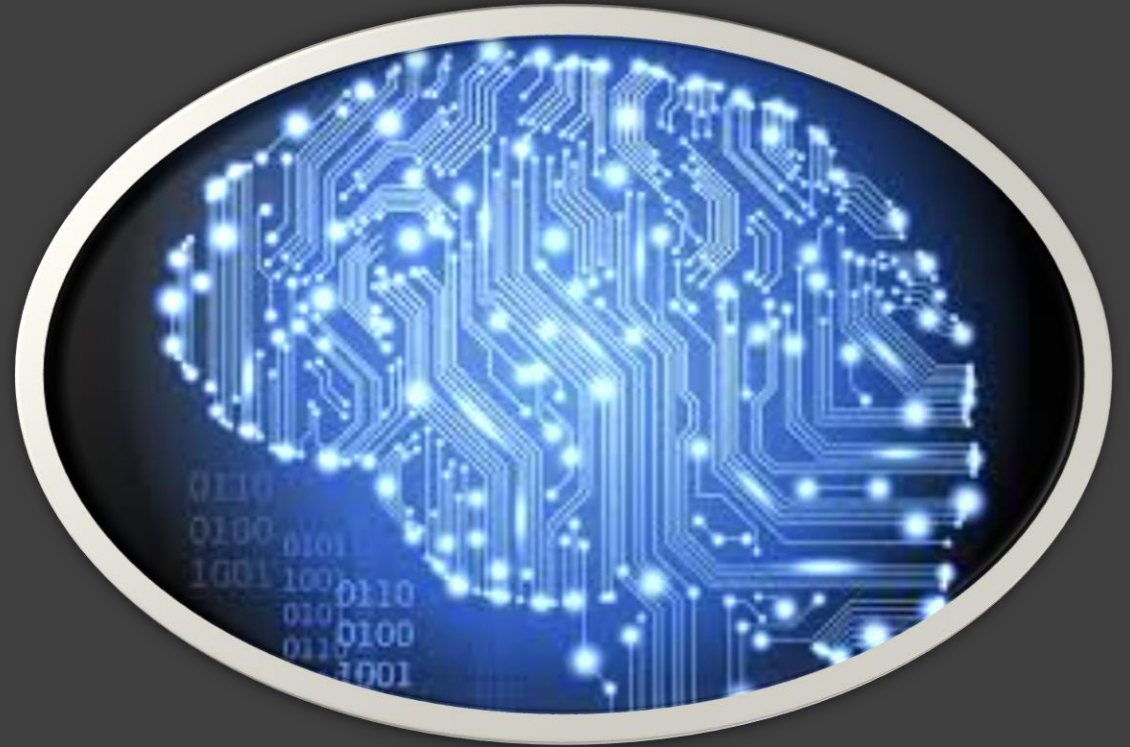
Pour résumer, s'il n'y a pas de changement en cours de déplacement, de la trajectoire initialement calculée (ici en #), le robot ne conservera que la trajectoire bleue comme version optimisée. On passe de 26 micro-trajets à 5 déplacements réels.

Le pathfinding

- Question time



« Stratégie »



Les actions de jeu

- Généralités

- La stratégie de chaque robot est découpée en plusieurs actions élémentaires, le système a été développé pour la coupe 2013 et conservé pour les suivantes.
- Il s'agit d'un tableau en mémoire comportant une liste d'actions sous forme d'une structure de données
 - Chaque action contient certaines infos qui vont permettre au système de choisir laquelle réaliser afin d'optimiser le match:
 - **Etat de l'action: impossible, en attente, en cours, en cours par l'autre robot, terminée**
 - Sert à décrire si l'action est à faire ou pas
 - **Avancement de l'action: step 0, 1, 2....**
 - Permet de reprendre une action déjà commencée sans forcément la reprendre à 0
 - **Nombre de points**
 - Le score qu'elle rapporte
 - **Priorité**
 - Permet de « forcer » un peu la main au système dans l'ordre de sélection des actions (pour au moins s'assurer de la première action)
 - **Coordonnées de départ**
 - Jeu de coordonnées sur le terrain permettant de savoir où se joue l'action
 - **Temps de match mini / maxi**
 - Définit un temps avant lequel l'action ne doit pas être exécutée, et un temps après lequel il est trop tard pour l'exécuter
 - **Quel robot doit exécuter cette action**
 - Les codes des robots étant communs, il faut dire à chaque robot, quelle action il est en mesure de réaliser ou pas
 - **Un pointeur**
 - Pointeur vers la fonction qui permet de réaliser cette action

Les actions de jeu

- Généralités

- Chaque fois que l'IA doit choisir une action à réaliser (soit parce que la précédente est terminée, soit parce qu'elle ne peut pas l'être et qu'il faut faire autre chose), la machine à état va parcourir le tableau des actions et évaluer celles qui méritent de l'être, c'est-à-dire:
 - Celles qui ne sont ni bloquées, ni terminées, ni prises par l'autre robot
 - Celles dont le temps mini de départ est atteint et dont le temps maxi ne l'est pas encore
- Pour toutes celles réalisables à cet instant, il va calculer via le A*, le coût du déplacement entre sa position actuelle et celle définie dans la structure de l'action.

Déjà pour savoir si l'action est atteignable et ensuite combien il lui coûtera de « distance » pour s'y rendre.
- Puis ce « coût » sera pondéré (coef < 0) et ajouté aux autres critères (nombre de points, priorité), eux aussi pondérés
- Chaque action recevra alors un score, le plus élevé à l'issue du process correspond alors à l'action à privilégier, son statut passera à « en cours » et elle sera exécutée temps qu'aucune information ne vienne l'interrompre (déplacement de l'adversaire, blocage...) ou qu'elle ne se termine.
- Une fois terminée ou interrompue, le système sélectionnera une autre action et le cycle recommence jusqu'à ce qu'il n'y ai plus d'action disponible ou que le temps de match soit écoulé.

Stratégie Sélection de l'action

```
int Choose_Best_Action(struct List_Actions Actions)
{
    int best_action_index = -1;
    int best_action_value = 0;

    for(int i = 0; i < Actions.number ; i++)    //For each action in the list
    {
        if(Actions[i].State == Terminee || Actions[i].State == Impossible) //Check if it's available for execution
            continue;

        if(Actions[i].Time_to_Start < Robot.Match_Time && Actions[i].Maximum_Time_to_Execute > Robot.Match_Time ) //is it time to start this action?
        {
            //This action is faisable, calculate the cost to go there
            Actions[i].Cost = Astar_Compute_Cost(Robot.Position, Actions[i].Start_Position);

            if(Actions[i].Cost < 0){
                //Action not reachable, try with an other one
                continue;
            }

            //calculate the "value" of this action
            int temp_value;
            temp_value = Actions[i].Cost * Kdistance;
            temp_value += Actions[i].Nombre_Points * Kpts;
            temp_value += Actions[i].Priority * Kprio;
            //Others criterias possibles (ie: secondary robot/Opponent position...)

            if(temp_value > best_action_value){
                //we've found a new best action to do
                best_action_value = temp_value;
                best_action_index = i;
            }
        }
    }

    //Now we know the best action to execute; -1 if no action available
    return best_action_index;
}
```

Stratégie Exemple d'action

```
bool Action_Bleu_Gobelets(struct Action* action){
    switch(action->Step){ //at which step of the action did we arrived last time?
        default:
        case 0:
            if(!Go_debut()){
                action->Time_to_Start = Robot.Match_Time + 5000; //we won't try this action again before 5 secondes
                action->State = En_Attente;
                return false; //Action not faisable
            }
            Ramasse_Gobelet1();
            Action_change_Step(action, action->Step + 1); //Increase the step, so that on next time we'll try this action, we won't grab the first element
            //no « break » because we don't want to stop the action
        case 1:
            if(!Go_Gobelet2()){ //Check if the move is accessible
                action->Time_to_Start = Robot.Match_Time + 5000; //we won't try this action again before 5 secondes
                action->State = En_Attente;
                return false; //Action not faisable
            }
            Ramasse_Gobelet2();
            Action_change_Step(action, action->Step + 1); //Increase the step, so that on next time we'll try this action, we won't grab the two firsts element
            //no « break » because we don't want to stop the action
        case 2:
            if(!Go_Zone_depose()){ //Check if the move is accessible
                action->Time_to_Start = Robot.Match_Time + 5000; //we won't try this action again before 5 secondes
                action->State = En_Attente;
                return false; //Action not faisable
            }
            Depose_Gobelets();
            action->State = Terminee;
            return true; //Action ended
    }
}
```

Les actions de jeu

- Interopérabilité des deux robots

Les deux robots utilisant le même code, ils partagent dans leur mémoire le même tableau d'actions. Leur liaison X-bee leur permet de s'échanger en temps réel, à la fois leurs positions respectives afin de s'éviter l'un l'autre. Mais aussi leurs choix concernant la stratégie et leur avancement sur les différentes actions.

De cette façon, lors qu'un robot choisit une action, celle-ci passe à l'état « En cours », le robot va en informer son équipier et lui dire « L'action numéro x est à passer à l'état «En cours sur l'autre robot » », elle ne sera alors plus disponible et il faudra en choisir une autre. Même chose pour les autres états et les « Steps » des actions qui sont échangés. Ainsi, chaque robot sait ce que l'autre est entrain de faire à chaque instant et peut agir en conséquence.

Les « Steps » peuvent être utilisés afin de synchroniser une action de coopération entre les deux robots. En 2015 (uniquement en test, non utilisé en match) nous avons fait en sorte que les deux robots choisissent une action de coopération presque en même temps. L'un d'entre eux, tenait un gobelet et le mettait à disposition de l'autre, sans le lâcher temps que le second ne l'avait pas attrapé. En scrutant les états « Steps » de l'action de l'autre robot, chacun pouvait savoir à quel moment était disponible le gobelet, quand il était saisi, relâché....sans utiliser de temporisations ni d'autres artifices de communication spécifiques.

La stratégie

- Question time

