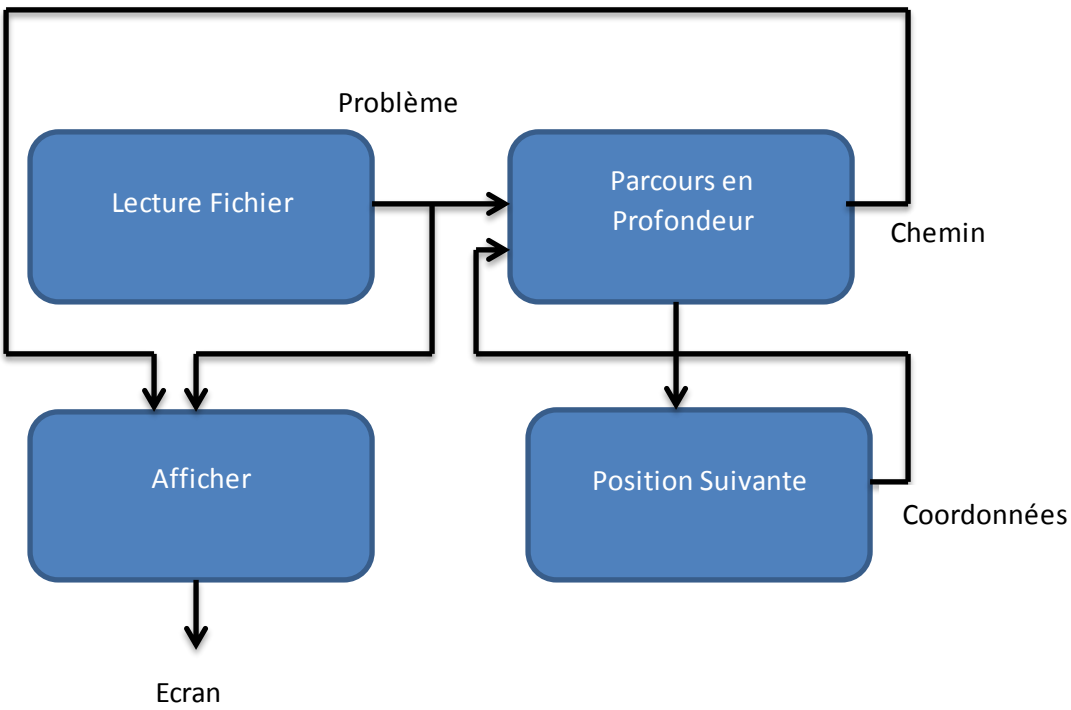
Nono le Robot

TOUS LES CODES SONT SUR : https://github.com/Guilhem117/NonoLeRobot

Schéma Bulle



Description du problème

	D	|		
+		|	_	
+	+			A

Légende :

D → Position de départ
A → Position d'arrivée
+, -, | → Murs
→ Position possible

Objet : Labyrinthe.

Objectif : Le robot commence à la position D et doit se déplacer jusqu'à atteindre la position A.

Déplacements possibles : haut, Bas, Gauche, Droite (Sauf si mur)

Algorithmes :

pos_suiv (auteur Jordan) : Renvoie les positions suivantes possibles à une position donnée

Recherche_Largeur (auteur Steven)

Recherche_Profondeur (auteur Guilhem & Jordan) avec optimisation en fonction de la distance

Algorithme des positions

```
/**
 * Renvoie les positions possibles pour le déplacement du robot
 * @Auteur Jordan PRADEL
 */
Fonction Coordonnee[] pos_suiv (E\ problem, E\ posActuelle, E\ fichierMap) {

        Paramètres :
           problem: pointeur sur structure Problem // Contient le contexte
           posActuelle : pointeur sur structure Coordonnee // Coordonnées de la position actuelle
           fichierMap : fichier // Fichier contenant les infos de la map

        Variables :
           possibilites : Coordonnee[] // Tableau des déplacements possibles
           c : Coordonnee // coordonnée à tester

        Début :

           // déplacement gauche
           c.num_ligne ← posActuelle.num_ligne
           c.num_col ← (posActuelle.num_col) -1
           SI deplacementPossible(problem, c, fichierMap)
                 possibilites[0] ← c

            // déplacement haut
           c.num_ligne ←(posActuelle.num_ligne) -1
           c.num_col ← posActuelle.num_col
           SI deplacementPossible(problem, c, fichierMap)
                 possibilites[1] ← c

           // déplacement droit
           c.num_ligne ← posActuelle.num_ligne
           c.num_col ← (posActuelle.num_col) +1
           SI deplacementPossible(problem, c, fichierMap)
                 possibilites[2] ← c

           // déplacement bas
           c.num_ligne ← (posActuelle.num_ligne) +1
           c.num_col ← posActuelle.num_col
```

```
        SI deplacementPossible(problem, c, fichierMap)
            possibilites[3] ← c
        Tri_Distance(possibilites, problem);
        RETOURNER possibilites;
    Fin
}
```

```
/**
 * Teste si un déplacement est possible
 * Si c'est le cas, renvoie 1, sinon 0
 * @Auteur Jordan PRADEL
 */
Fonction entier deplacementPossible(E\ problem, position, fichierMap) {
        Paramètres :
           problem: pointeur sur structure Problem // Contient le contexte
           position : Coordonnee // Coordonnées de la position à tester
           fichierMap : fichier // Fichier contenant les infos de la map
        Variables :
        Début :
           // Déplacement impossible
            SI (position.num_ligne < 0 OU position.num_ligne > nbLignes
                    OU position.num_col < 0 OU position.numCol > nbCol)
                RETOURNER 0
           // Déplacement possible
           SI problem→ carte[position.num_ligne][position.num_col] = ' '
                RETOURNER 1
           SINON // Obstacle
                RETOURNER 0
        Fin
}
```

Algorithme de Parcours en largeur

```
/**
  *Permet d'obtenir le parcours demandé par un probléme à l'aide d'un algorythme de parcours *en largeur.
  *@author salandini
  *@param m, le problème contenant le sommet de dép/de fin et la map.
  *@retour P la pile contenant les sommet constituant le parcours
  */
Fonction RechercheLarg(E/ Propleme m , S/ Pile P)
Variables
        Pile P;
        File F;
        Booleen marques[m.nbLignes*m.nbColonnes];
        Entier i;
        Entier[] suivants;

F ← créerFile ()
P ← créerPile()
AjouterElem(F,s)
AjouterElem(P,s)
Tant que non FileVide(F) faire
        V ← RetirerElem(F)
        Si v = m.sommetfin alors
                Retour
        Fin si
              posSuivante ←  pos_suiv(M,V)
        Pour i de 0 à 4 faire
                U ← * pos_suiv(m,v) +i
                numU ← NumSommet(M,u)
                Si non Marque[numU] ET posSuivante[numU-1]=0 alors
                        Marque[numU] ← vrai
                        AjouterElem(F,u)
                        AjouterElem(P,u)
                Fin si
        Fin pour
        Si DegreExt(M,v) = 0 OU toutMarque(M,v,Marque)
                RetirerElem(P)
        Fin Si
Fin Tant Que
```

FIN

Algorithme du Parcours en Profondeur

```
/**Calcule un chemin qui va de la position de départ à la position d'arrivée
 *@author SERENE Guilhem
 *@param p, le probleme contenant la carte du labyrinthe (et ses obstacles), la position d'arrivée et la position de départ
 *@return P, une pile contenant les sommets constituant le parcours
 *          ou une pile vide si le chemin n'a pas pu être trouvé
 */
Fonction Recherche_Profondeur(E/ Problem p) Retourne Pile
Variables
        Pile chemin;
        Booleen marques[p.nbLignes*p.nbColonnes];
        Booleen posSuivant;
        Entier i;
        Entier[] suivants;
Debut

        chemin ← Init_Pile();
        chemin.Sommet ← p.pos_depart;

        Tant Que (chemin.Sommet != p.pos_arrivee ET PileVide(chemin)) Faire
                Ajouter_Element(chemin, chemin.Tete);
                marques[chemin.Sommet];

                suivants = pos_suiv(p, chemin.Sommet);
                pasSuivant ←  Faux;
                i ←  0;
                Tant que (i < 4 ET pasSuivant) Faire
                        /* Obstacle n'existe pas ou est déjà marquée */
                        Si (suivants[i] = 0 ET marque[i] !=suivant[i]) Alors
                                chemin.Sommet ← suivants[i];
                                posSuivant← Vrai;
                        Fin Si
                Fin Tant que

                Si Non (posSuivant) Alors
                        Retirer_Elem(chemin);
                Fin Si;
```

```
        Fin Tant Que

        retourne chemin;

Fin
```