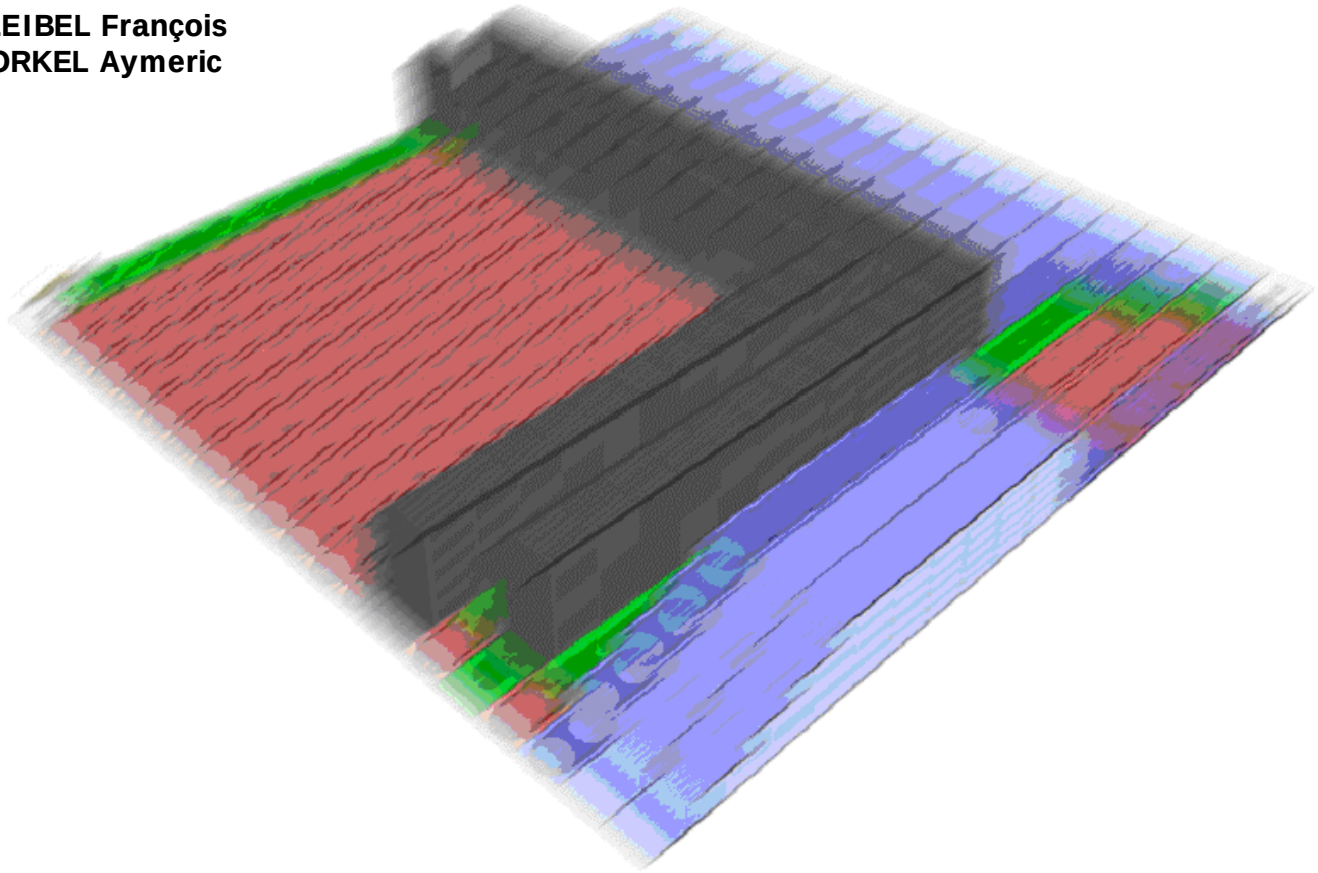
TPE 2004
BLEIBEL François
DORKEL Aymeric



Les algorithmes de l'intelligence artificielle

Quels algorithmes permettent à un robot de se déplacer de manière autonome dans son environnement ?

Introduction

L'intelligence artificielle est une compétence spécifique aux robots ou aux automatismes (« artificielle ») qui détermine sa faculté à imiter un comportement humain. Cette intelligence essaye d'imiter le fonctionnement d'un cerveau humain dans beaucoup de domaines, par exemple la parole ou le caractère, deux aspects spécifiques aux humains, mais aussi notamment la capacité du cerveau à reconnaître les objets ou personnes qui l'entourent. Elle a des applications tout d'abord dans l'informatique pure : une des utilisations de l'intelligence artificielle sert à l'apprentissage d'une voix par un ordinateur pour la reconnaissance vocale; les jeux vidéo utilisent aussi beaucoup une intelligence simulée pour donner au joueur l'impression de jouer avec des humains, donnant la capacité aux personnages contrôlés artificiellement de se « comporter » comme un humain, en adoptant un caractère, un langage ou encore des actions qui sont à l'origine de l'attitude du joueur envers eux, et qui imiteraient un dialogue ou un combat le plus réaliste possible. On peut aussi l'utiliser pour contrôler des chaînes de production mécanisées (industrie lourde ou robotique) : des machines « intelligentes » sont ainsi capables de deviner le moment où un défaut ou une fausse manoeuvre va survenir bien avant des capteurs traditionnels, soit par un apprentissage préalable ou soit en utilisant des algorithmes « classiques » (méthode la moins utilisée), le résultat source de réduction du temps d'arrêt de la production, faisant ainsi économiser du temps et de l'argent à l'entreprise qui les utilise.

Dans ce TPE, notre but est de découvrir et examiner les différentes solutions qui permettent à un robot quelconque de se déplacer dans son environnement (une maison ? une pelouse) pour aller chercher une balle rouge. Il devra pour ce faire éviter les obstacles qui se présenteraient à lui, et pourquoi pas, optimiser son trajet histoire d'économiser de l'énergie en se déplaçant jusqu'à la balle. Ce robot doit faire preuve de deux aptitudes « humaines » ou les imiter pour réaliser son objectif.

La première consiste à donner à un robot (ou personnage, dans une approche virtuelle) la capacité de se mouvoir « intelligemment » dans un environnement. Ici l'intelligence mise en œuvre aura pour but de se mouvoir d'un point à un autre en évitant les obstacles. On pourrait penser que c'est une intelligence assez « simple » ou en d'autres termes, évidente : nos trajets de tous les jours nous semblent simples; contourner des obstacles, ou aller chercher une chaise derrière une table ne relève pas du défi pour nous les humains. Mais dans le cadre d'un robot, avec pour seule « intelligence » un microprocesseur et des capteurs, c'est beaucoup plus ardu : il se pose alors le problème d'une détermination optimale (c'est-à-dire la plus courte) du trajet, ceci en n'ayant pas forcément la vision spatiale et immédiate de l'environnement que nous avons (c'est-à-dire sans forcément connaître la position des obstacles à l'avance : nous examinerons les deux cas), et même alors, les « points de passage » (du trajet) doivent être calculés de manière précise en minimisant la distance parcourue, mettant en œuvre des algorithmes aussi complexes que variés, dont les utilisations sont décrites dans la partie II.

La seconde aptitude humaine que nous allons étudier consiste à reconnaître les objets, c'est-à-dire leurs formes et/ou leurs couleurs. Les yeux qui nous permettent de reconnaître une balle quand nous en voyons une (en nous basant sur notre expérience et les informations de nos sens) ont leurs équivalents robotiques. Ces capteurs CCD ou autres caméras numériques embarqués sur notre robot couplées à un système de commande (le cerveau électronique du robot) lui permettent de reconnaître les caractéristiques (la forme et la couleur d'une balle rouge, par exemple) des objets. Le robot peut alors identifier certains objets et prendre des décisions en temps réel basées sur sa reconnaissance de l'environnement.

Ces deux aptitudes sont les plus utilisées dans les applications modernes de l'intelligence artificielle : la première (détermination du trajet) sert dans les logiciels de cartes routières ou les ordinateurs de bord, afin de conseiller l'utilisateur sur la meilleure route à prendre. La seconde peut servir dans la reconnaissance des personnes (un robot - chien qui reconnaîtrait son maître par exemple) ou pour guider le déplacement d'un robot vers un objet, comme dans notre cas.

Nous allons donc étudier les algorithmes et autres méthodes nécessaires pour réaliser notre objectif. Le robot doit *reconnaître la balle et en déterminer sa position*, pour pouvoir ensuite *se déplacer jusqu'à elle*. Ce TPE est donc divisé en trois parties. La première, *Reconnaissance d'images* traite des algorithmes, capteurs, et tout ce qui est nécessaire pour que le robot puisse reconnaître notre balle et en déterminer sa position. La seconde partie, intitulée *Détermination Intelligente de Trajet*, parle des algorithmes utilisés pour que le robot trouve (et optimise) son chemin en se déplaçant jusqu'à la balle. Enfin, la dernière partie, *Les réseaux neuronaux*, explique les mécanismes de fonctionnement d'une des plus récentes et plus compliquées découvertes de l'intelligence artificielle.

Sommaire

Reconnaissance et localisation d'objets à partir d'images numérisées

Les systèmes robotisés utilisant des informations visuelles provenant de capteurs optiques, du type de ceux utilisés dans des caméras ou appareils photo, sont nombreux (ex : systèmes de contrôle industriel, robots d'exploration spatiale, …). Mais comment sont-ils capables d'interpréter ce qu'ils « voient » pour interagir avec leur environnement ? A partir d'un exemple, nous allons essayer d'illustrer différentes méthodes permettant à un robot de repérer un objet pour aller le ramasser, nous utiliserons pour cela une balle rouge.

Mise en situation

Nous disposons d'une image, provenant du capteur d'un robot (voir ci-dessous). Nous allons repérer la balle rouge qu'il devra ramasser, après en avoir déterminé la position.

Pour simplifier le problème, nous définissons qu'un seul objet sphérique rouge (la balle) sera visible par le robot, et qu'il ne sera pas masqué à plus de 25% par un obstacle.

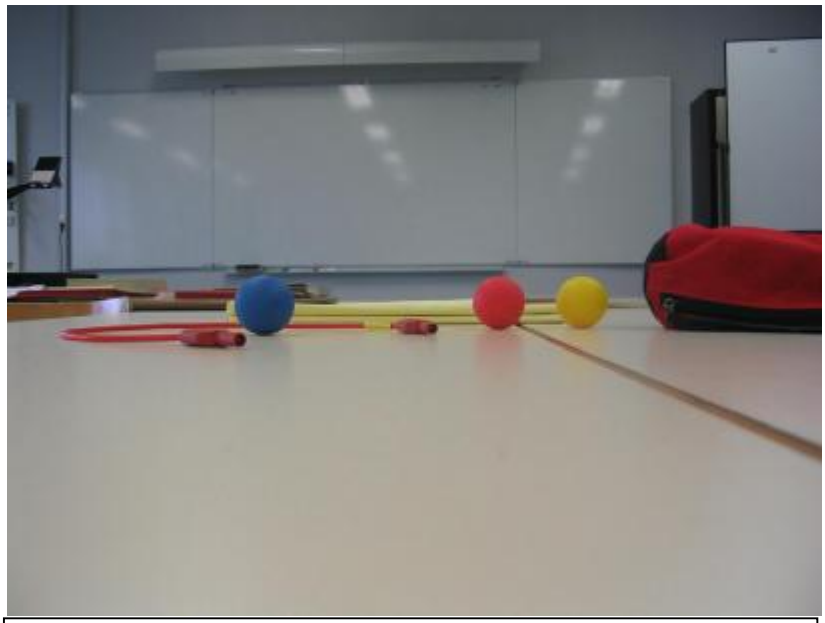
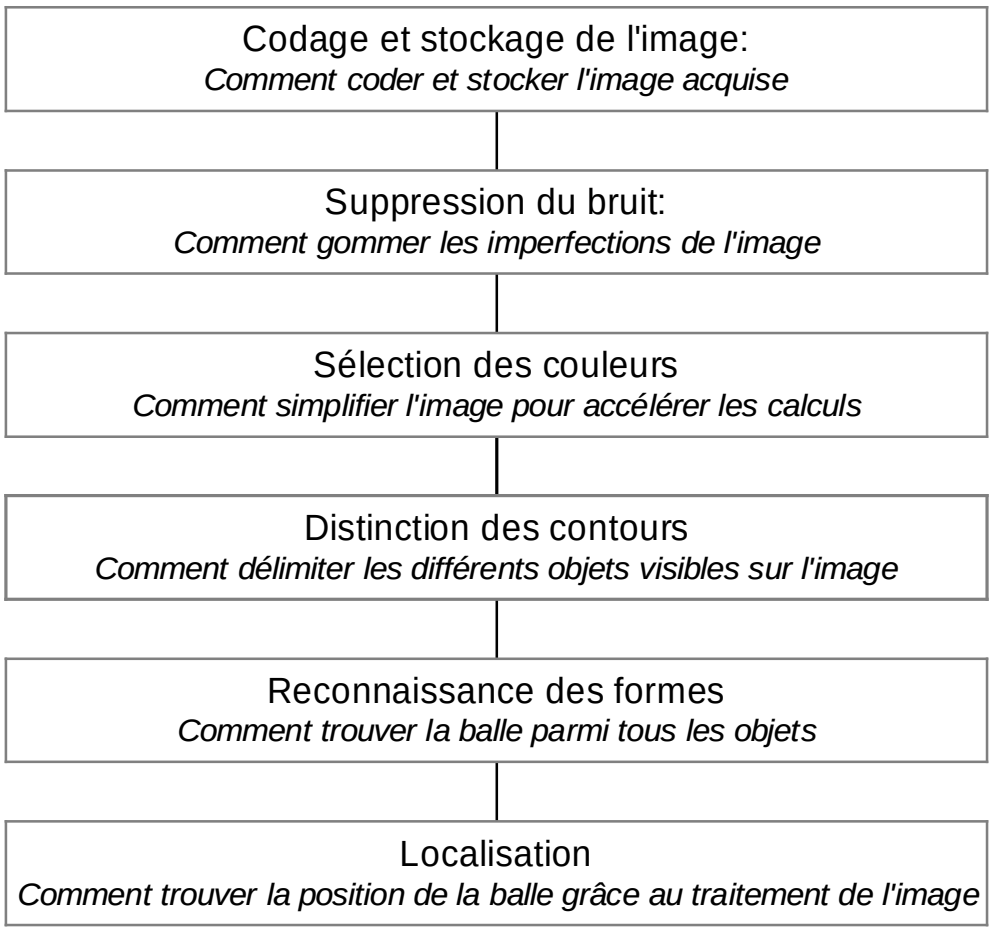


Image provenant du capteur du robot

Processus permettant d'aboutir à une solution



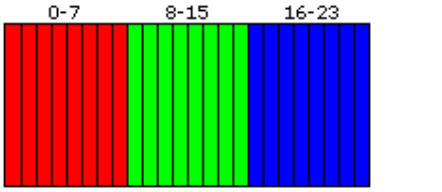
A. Codage et stockage de l'image

1) Acquisition et codage

Dans la majorité des cas, le système réceptionne les données provenant d'un capteur CCD (*Charge Coupled Device* ou *Dispositif à Transfert de Charges* en français). Ce type de cellule photo-sensible est utilisé dans la plupart des appareils photos et caméras numériques. Il délivre les données de l'image sous la forme d'un tableau de pixels. Pour chaque pixel, les informations de luminosité et de couleur sont codées puis stockées dans une entité de taille définie. Dans la majorité des cas, on utilise des codes couleurs dits « 24 bits ».

Une couleur est stockée sur 24 bits, soit 3 octets, comme le montre le schéma ci-dessous.

On stocke l'intensité de chacune des composantes primaires (rouge, vert et bleu) sur un octet, soit 256 niveau de luminosité différents. Cela permet d'obtenir 256^3, soit 16 777 216 nuances de couleurs distinctes.



Stockage d'une couleur sur 24 bits

La dimension de la structure de stockage dépend de plusieurs facteurs. Elle est proportionnelle à la qualité du capteur et à la qualité requise. Une image en noir et blanc utilise des couleurs stocké sur 1 bit (2 états, noir et blanc), une image en niveaux de gris utilise des couleurs stockées sur 16 bits. Les images couleurs utilisent généralement des structures de stockages de 16, 24 ou 32 bits. Dans ce TPE, nous utiliserons des couleurs 24 bits, car elles sont très pratiques (1 octet, soit 256 états, par couleur primaire).

2) Stockage : la matrice

En mathématique, une matrice est un tableau de valeurs. Nous allons donc considérer qu'une image peut être définie par trois matrices, une pour chaque couleur primaire (rouge, vert, bleu). Les valeurs initiales de celles-ci seront des entiers, compris entre 0 et 255.

Pour obtenir la couleur d'un pixel d'abscisse x et d'ordonné y d'une matrice M, nous utilisons la syntaxe suivante : Couleur = M(x, y)

Exemple de matrice M (valeurs entre 1 et 9) :

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

En grisé, la case de coordonné (3,2). Nous pouvons donc écrire M(3,2) = 9.
La case de coordonnée 0,0 se situe en haut à gauche.

3) Modifications de base

Au fur et à mesure de l'avancement des processus de recherche de la position de la balle, les algorithmes utilisés vont modifier les valeurs des trois matrices initiales. Après certaines étapes, leurs contenus ne seront plus directement visualisables, car ils ne correspondront plus à des niveaux de couleur compris entre 0 et 255. Elles pourront contenir aussi bien des nombres négatifs que des valeurs supérieures à 255.

Afin de pouvoir visualiser la progression de la recherche, nous adapterons les valeurs de ces matrices, en fonction de la situation. Nous pourront soir remplacer les valeurs négatives par leur opposé, soit par une valeur nulle. Les valeurs trop élevées seront tronquées.

B. Elimination du bruit

Les caméras contiennent des circuits électroniques associés à des capteurs optiques. Le bruit qui est induit par tout composant électronique génère des pixels dont la couleur ne correspond pas à la réalité, qui forment un bruit.

Les algorithmes de reconnaissance de formes réagissent très mal à la présence de bruit, car ils le considèrent comme de petits objets. Il faut donc supprimer ce bruit, ou l'atténuer. Pour cela, nous utiliserons des filtres, qui mettent en œuvre des principes mathématiques qui peuvent être par exemple la convolution.

Afin de pouvoir juger de l'efficacité des filtres utilisés, nous allons utiliser un extrait de la photo prise par le robot, auquel nous allons ajouter un bruit artificiel. Nous pourrons ensuite le traiter avec les différents filtres présentés.

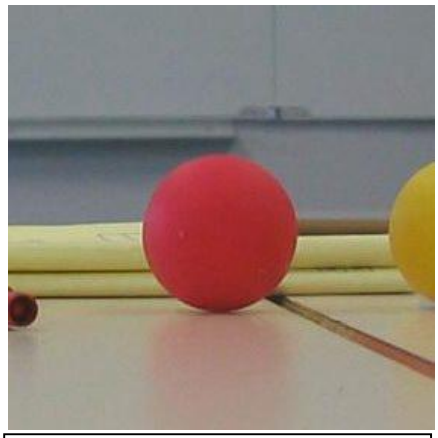


Image de base

Pour bruiter l'image, nous attribuons des couleurs aléatoires à 10 000 pixels aléatoirement choisis. Voici le résultat :

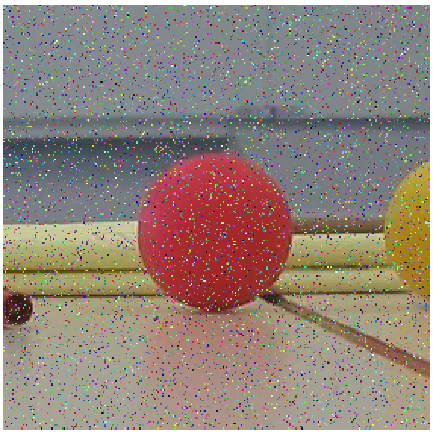


Image aléatoirement bruitée

1) Principe de convolution

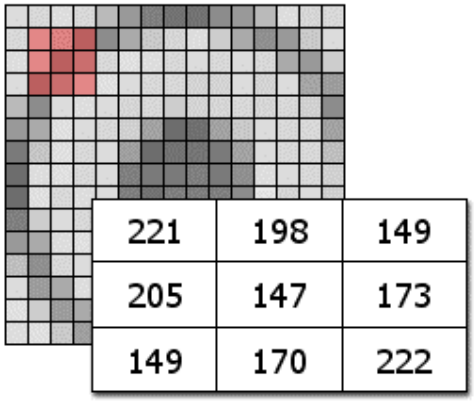
La convolution est une opération mathématique que l'on applique aux matrices. Elle permet de générer une matrice fille (image traitée) à partir d'une matrice mère (image à traiter), et d'une matrice « noyau » (fonction de la finalité du filtre).

Soit une matrice fille N, une matrice mère P, et une matrice noyau K, de dimensions latérales impaires i_{max} et j_{max}, dont la valeur de coordonnée (0,0) est le centre.

Voici la formule permettant d'effectuer une convolution :

$$N_{y_x} = \sum_{i=-\frac{i_{Max}-1}{2}}^{\frac{i_{Max}-1}{2}} \sum_{j=-\frac{j_{Max}-1}{2}}^{\frac{j_{Max}-1}{2}} K(i,j) * P(x+j, y+i)$$

Exemple de convolution :



221	198	149
205	147	173
149	170	222

-1	0	1
-2	0	2
-1	0	1

Sur la gauche, en arrière plan, un dessin représentant la matrice mère. En rouge, la partie dont les valeurs chiffrées sont données dans le tableau numérique de gauche. Sur la droite, la matrice « noyau ».
Calculons la valeur du centre du carré rouge après la convolution :

N(x,y) = (-1 x 222) +
(0 x 170) +
(1 x 149) +
(-2 x 173) +
(0 x 147) +
(2 x 205) +
(-1 x 149) +
(0 x 198) +
(1 x 221) = 63

Pour effectuer une convolution, on utilise les pixels voisins de chaque pixel. On ne peut donc pas appliquer cette technique aux pixels du bord de la matrice mère. La matrice fille est donc plus petite que la matrice mère (plus la taille de la matrice noyau est importante, plus cette différence sera grande).

Il ne faut pas oublier que ces opérations sont très coûteuses en calculs : pour une image carrée, de 100 pixels de côté, avec une matrice noyau de dimensions 3x3, il faut déjà faire :

(98 *[côté valide de la matrice mère]*)²
x 9*[Nombre de valeurs de la matrice « noyau »]*
= 86.436 multiplications et autant d'additions.

(pour une image de meilleure qualité, 800x600 pixels, avec une plus grande matrice noyau, de 11x11 par exemple, il faudrait 112 796 200 opérations !)

2) Filtres linéaires : élimination basée sur la convolution

Les filtres linéaires sont des filtres relativement basiques, utilisant uniquement le principe de convolution précédemment décrit. Nous allons en étudier deux, le filtre dit « moyenneur » et celui dit « gaussien ».

a) Filtre moyenneur

Cet algorithme calcule, pour chaque pixel, la couleur moyenne d'une zone l'entourant. La convolution utilise donc une matrice noyau dont la somme vaut 1, pour ne pas modifier l'image. Il donne ensuite la couleur résultant du calcul au pixel étant traité.

Généralement, on essaye d'appliquer un filtrage qui a le même effet dans toutes les directions, ce que l'on nomme filtrage isotopique. Pour les matrices noyaux de grande taille, la zone utilisée sera circulaire.

Exemples de matrices :

Matrice 3x3 : $\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$

Il s'agit ici d'une matrice de base, elle permet un calcul rapide et un filtrage correct.

Matrice 7x7 :

0	0	0	1/29	0	0	0
0	1/29	1/29	1/29	1/29	1/29	0
0	1/29	1/29	1/29	1/29	1/29	0
1/29	1/29	1/29	1/29	1/29	1/29	1/29
0	1/29	1/29	1/29	1/29	1/29	0
0	1/29	1/29	1/29	1/29	0	0
0	0	0	1/29	0	0	0

Cette matrice noyau de plus grande taille nécessite un temps de calcul plus long et doit de préférence être utilisée avec des images de hautes résolution, où les imperfections pourraient êtres de plus grandes taille (en nombre de pixels pollués).

D'un point de vu pratique, l'algorithme moyenneur n'a pas besoin d'une convolution, il existe une façon plus simple de le mettre en œuvre.
Pour chaque pixel valide, nous effectuons l'opération en boucle suivante :

- **Calcul de la somme des valeurs des couleurs du pixel et de ses voisins**
- **Division par le nombre de pixels pris en compte**
- **Application de la nouvelle couleur au pixel traité**

(les pixels situés au bord le l'image n'ont pas le nombre de voisins requis pour être traités, ils ne sont donc pas « valides »)

Exemple pratiques :

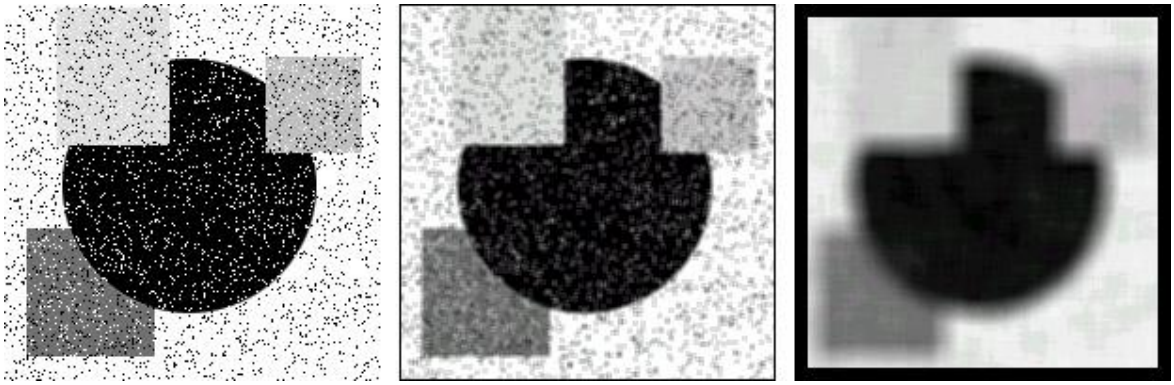


Image avec bruit, filtrée avec « noyau » à 9 pixels, filtrée avec « noyau » à 225 pixels.

On note que plus la matrice noyau est grande, plus le bruit est atténué. En contrepartie, plus elle est grande, plus l'image devient floue.

Avec l'image provenant de notre robot :

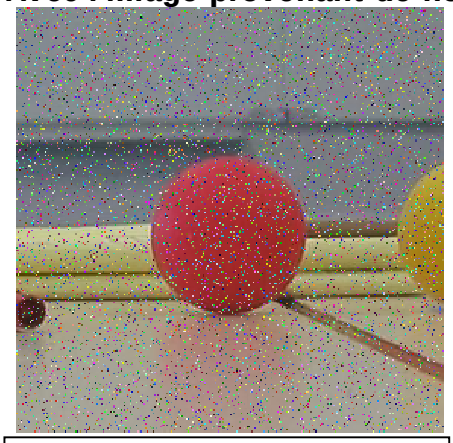


Image aléatoirement bruitée 10 000 pixels de couleur modifiée

Image nettoyée avec un filtre moyenneur, matrice noyau de côté 3

On constate facilement que le résultat est de très mauvaise qualité. Le bruit a été à peine atténué.

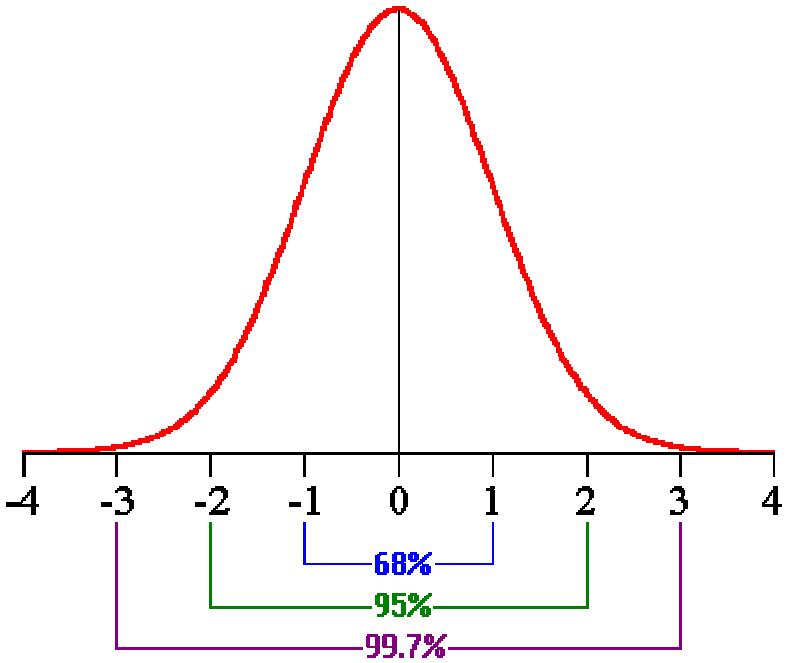
Cet algorithme a un très grand défaut : il rend les images floues. En effet, dans les zones de changement de couleur, on retrouve un effet de fondu, dont l'importance est proportionnelle à la taille de la matrice « noyau ». Ce filtre applique également cet effet aux pixels considérés comment étant du bruit, ce qui l'atténue mais ne le supprime pas totalement.

Le filtre moyenneur est extrêmement simple à mettre en place et ne demande pas trop de ressources lors de son fonctionnement. Il est rarement utilisé, lorsque un très faible bruit doit être atténué.

b) Filtre gaussien

Le filtre gaussien, qui doit son nom au célèbre mathématicien ayant définit la courbe caractéristique représentée ci-dessous, est très proche du filtre moyenneur, à cela près qu'il ne prend pas autant en compte les pixels éloignées du pixel à traiter que ceux qui lui sont directement voisins. C'est une sorte de moyenne pondérée, dont les poids sont définis par la courbe de Gauss. Plus un pixel est proche du centre de la matrice noyau, plus sa couleur est prise en compte.

Exemple de courbe de Gauss :

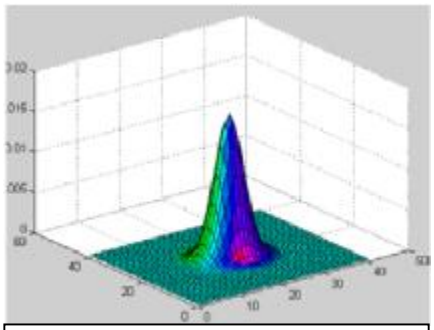


Courbe de Gauss

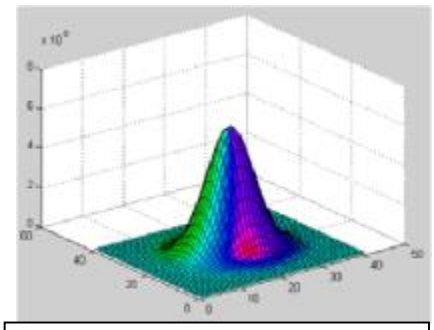
Si on nomme K la matrice noyau, le calcul des poids se fait selon la formule suivante :

$$K(x,y) = \frac{1}{2p s^2} \exp\left(-\frac{x^2+y^2}{2s^2}\right)$$

La constante s **permet de facilement régler le degré de filtrage, comme le montre les différentes représentations tridimensionnelles des poids de la matrice noyau suivantes :**



$s = 2$



$s = 3$

Ce filtre permet une prise en compte progressive de la couleur des pixels voisins du pixel à traiter, en fonction de leur distance à celui-ci, contrairement au filtre moyenneur, qui définit une zone de pixels pris également en compte.

Malheureusement, ce filtre ajoute également un effet de flou à l'image, et ne supprime pas totalement les pixels parasites, mais se contente d'atténuer le bruit. Il est relativement peu utilisé.

Les filtres linéaires, basées sur les convolutions de matrices ne sont pas très efficaces. Le bruit n'est qu'atténué, et l'image devient flou, ce qui va compliquer la reconnaissance de contours. Seuls des filtres non-linéaires pourront résoudre ce problème.

3) Filtres non-linéaires

Les filtres non linéaires utilisent tous des tests conditionnels, ils sont par exemples parfois basés sur des seuillages ou des classements, ce qui les rend non linéaires.

a) Filtre médian

Le filtre médian considère les pixels entourant le pixel à traiter, généralement sur un disque, pour obtenir un filtrage isotopique, et en trouve la valeur médiane, qui sera donnée au pixel à traiter.

Dans une série, la médiane correspond à la valeur qui sépare la série en deux ensemble de même taille. C'est donc une valeur appartenant à la série, contrairement à la moyenne, qui est une valeur calculée. Cela évite les effets de flou.

Exemple d'application :

Zone de l'image, avec sur fond bleu, le pixel à traiter.
Seuls les pixels sur fond coloré sont pris en compte pour la médiane.

7	10	12	7	9	9	9
12	12	12	7	9	9	10
9	9	10	10	7	10	12
10	9	9	12	10	9	12
12	12	12	10	7	7	10
10	7	10	12	12	10	7
10	7	9	12	10	9	12

Voici la liste ordonnée des 29 valeurs prises en comptes :
7, 7, 7, 7, 7, 7, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 10, 12, 12, 12, 12, 12, 12, 12, 12, 12

La valeur sur fond gris représente la médiane de la série. Elle sera appliquée au pixel à traiter.

Résultats :

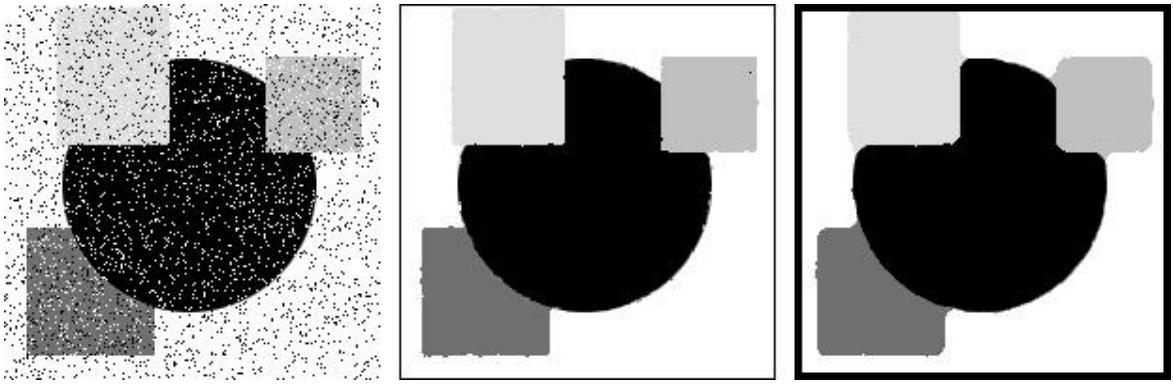


Image avec bruit, filtrée avec médiane sur 9 pixels, filtrée avec médiane sur 81 pixels.

Résultat avec un extrait de l'image de notre robot :

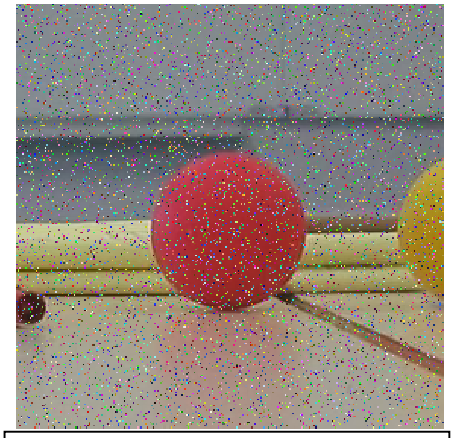


Image aléatoirement bruitée
10 000 pixels de couleur modifiée

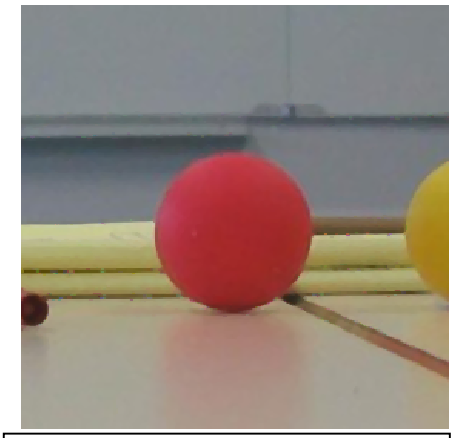
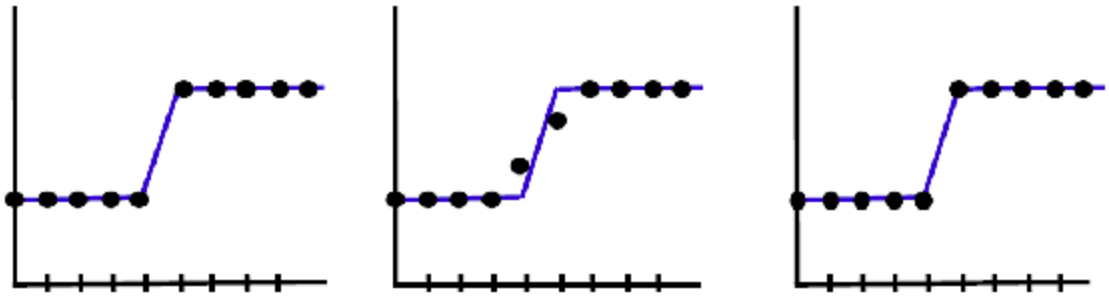


Image nettoyée avec un filtre médian, sur 9 pixels

Le filtrage médian a permis de retrouver une image nette, débarrassée de la majorité du bruit anciennement présent, sans effet de flou.

On comprend très bien pourquoi ce filtre n'ajoute pas d'effet de flou à l'image, comme le montre la comparaison ci-dessous, qui représente en changement de couleur traité avec un filtre moyenneur et un filtre médian. En abscisse, une dimension (par exemple une ligne d'une image) et en ordonné, les couleurs.



La médiane permet de donner des limites nettes aux formes, et supprime totalement le bruit. Malheureusement, le temps de calcul nécessaire est plus long, une moyenne est plus rapide à calculer qu'une médiane. Le filtre médian reste plutôt simple à implanter, et a un très bon rapport temps de calcul nécessaire / netteté du résultat.

b) Filtre seuillé

Le principal défaut des filtres précédemment détaillés réside dans le fait qu'ils lissent toute l'image, aussi bien dans les zones de couleur unie où se trouvent un bruit qu'il faut éliminer que dans les zones de transitions (contours) qu'ils ont tendance à rendre flou, ou à déformer. Le principe de moyenne seuillée est une fonction que l'on peut coupler avec un filtre linéaire, par exemple avec un filtre moyenneur, gaussien ou médian, afin d'éviter ce problème.

Principe :

L'algorithme calcule la variance sur une zone autour de chaque pixel. La variance est une grandeur mathématique permettant d'estimer la dispersion d'une série de valeurs.

Si elle est faible (inférieure à un seuil), nous sommes dans une zone relativement homogène, donc dans une même forme, et nous pouvons appliquer le filtre afin de supprimer les imperfections. Par contre, si la variance est élevée, nous nous trouvons dans une zone de transition (par exemple un contour délimitant deux formes distinctes) et nous n'appliquons pas le filtrage, pour ne pas détériorer la netteté du contour.

Dans le cas où nous utilisons un filtre à moyenne seuillée, le filtrage par moyenne ne rend plus les images floues, mais atténue tout de même les imperfections de l'image.

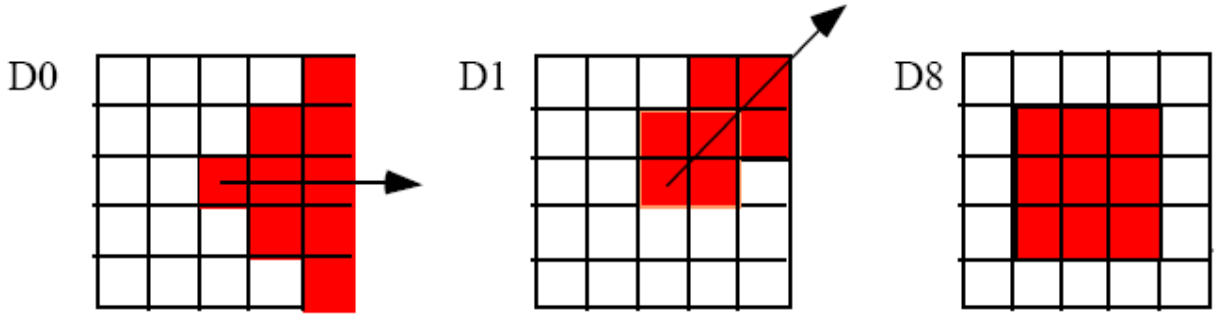
c) Filtre pondéré

Le filtre pondéré est une évolution du filtre seuillé. Lorsque ce dernier agissait de façon binaire, et définissait s'il faut ou s'il ne faut pas appliquer de filtrage pour un pixel en fonction de la variance de la zone l'entourant, le filtre pondéré applique le filtrage inversement proportionnellement à cette variance. Plus la variance est faible (zone plutôt homogène) plus le filtrage est efficace. Plus la variance est élevé (zone de contour) plus le filtrage est discret.

Le filtre pondéré permet un filtrage en fonction de l'image, adapté à chaque situation. Il ne rend pas les contours flous, mais gomme les imperfections des zones homogènes.

d) Filtre de Nagao

Le filtre de Nagao, aussi appelé filtre à sélection de voisinage, est l'ultime évolution des filtres seuillés et pondérés. Il va appliquer un filtrage en fonction de la variance sur des formes prédéfinies, des gabarit, correspondant aux contours les plus souvent rencontrés. Il va donc supprimer le bruit présent sur l'image et renforcer les contours présents.



Chaque carré de 5 pixels de côté est divisé en 9 domaines, nommés D0 à D8. D2, D4 et D6 sont déduits de D0 par rotations de 90°, de même pour D3, D5 et D7, provenant de rotations de D1.

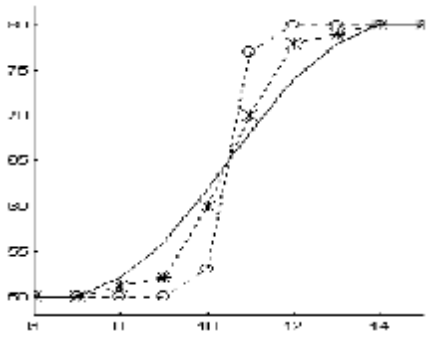
Lorsque l'on applique le filtre de Nagao à un pixel, on calcule la variance de chaque domaine Di de la zone carrée l'entourant. Celui pour lequel elle sera la plus faible ne sera traversé par aucun contour. Nous allons donc appliquer un filtre secondaire (médian ou moyenneur par exemple) en utilisant uniquement les valeurs du domaine sélectionné.

Par exemple, si lors du traitement d'un pixel, le domaine dont la variance est la plus faible est le domaine D1 et que le filtre secondaire est un filtre moyenneur, nous appliqueront la couleur moyenne de tous les pixels en rouge sur le schéma du domaine D1, représenté ci-dessus.

Conclusion sur le filtrage anti-bruit

Le choix d'un filtre anti-bruit va dépendre de plusieurs critères :

- ***La puissance de calcul disponible :*** **si les opérations doivent êtres exécutées par un ordinateur embarqué dans un robot de petite taille, le matériel ne dispose pas forcément d'une grande vitesse de calcul ou de beaucoup de mémoire.**
- ***La vitesse de calcul requise :*** **Le choix du filtre ne sera pas le même si les résultats doivent êtres obtenus « en temps réel », par exemple pour un système nécessitant un déplacement rapide, ou à une fréquence moins élevée, par exemple pour un système de surveillance.**
- ***Le type de bruit :*** **des filtres relativement simples (comme le médian appliqué sur un carré de 3 pixels de côté) seront suffisants pour nous débarrasser totalement d'un bruit impulsionnel (pixels de couleur incohérente par rapport au reste de l'image, pouvant provenir d'interférences dans l'électronique d'acquisition) mais ne gèreront peut-être pas correctement un bruit plus important (comme de petites taches sur l'objectif du capteur, d'une surface bien plus grande qu'un pixel)**
- ***Le résultat requis :*** **parfois, une atténuation du bruit peut suffire, quand dans d'autres cas, les moindres interférences pourrait s'avérer très gênantes. Cela dépend des traitements appliqués à l'image après le filtrage anti-bruit.**



Ci-contre, une comparaison du filtre de loyenneur (ligne), du filtre médian (étoiles) et du filtre de Nagao (cercles) sur une zone de transition. En abscisses, une position sur l'image, en ordonnée, une couleur (par exemple en niveaux de gris). On vois que le filtre de Nagao rend les contours très nettes (très forte courbe) quand les deux autres sont plus progressifs.

Voici un comparatif des filtres présentés les plus importants :

Filtre	Points forts	Points Faibles
Moyenneur	Calculs très rapides Fonctionnement très simple	Atténue le bruit sans le supprimer Ajoute un effet de flou
Median	Calculs assez rapides Aucun effet de flou	Peut légèrement déformer les contours
Seuillé	Filtrage non-systématique	Temps de calcul élevé
Nagao	Qualité de filtrage élevée	Temps de calcul élevé (le plus long)

Les filtres anti-bruit sont très variés, et adaptés à de nombreuses situations. Le filtre le plus simple, le moyenneur, sera de préférence implanté dans les systèmes autonomes ne nécessitant pas une grande précision et ne disposant pas d'une grande capacité de claculs, alors que le filtre le plus poussé, le filtre de Nagao, ne sera utilisé que dans des cas bien particuliers, comme un traitement de photo par exemple.

C. Sélection

Dans notre cas, nous savons que l'objet que nous recherchons est rouge. Il n'est donc pas nécessaire de s'inquiéter des autres couleurs présentes sur notre image. La sélection consiste à ne plus que s'occuper des zones de l'image ayant la couleur recherchée.

Nous allons modifier les couleurs de l'image, en coloriant en blanc les parties correspondant à nos critères et en noir celles ne nécessitant pas notre attention. Il s'agit d'une « binérisation » (au final il ne reste que deux états distincts)

Pour effectuer cette sélection, nous allons utiliser d'une part les composantes primaires de chaque couleur (quantité de rouge, de vert, de bleu, variant pour chacune entre 0 et 255), et d'autre part, la proportion de chaque couleur primaire dans la couleur finale.

Voici l'image que nous allons binériser :

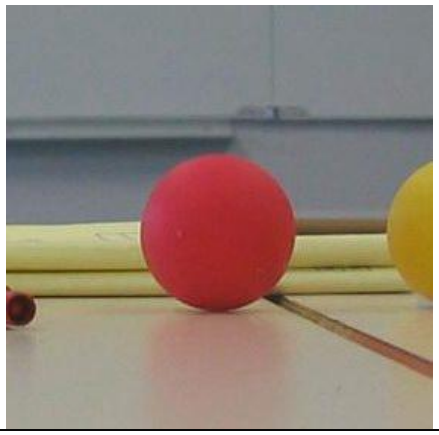


Image de base après filtrage antibruit

Etant donné que la balle que nous recherchons est rouge, les conditions que nous pouvons fixer sont :

- **La part de rouge minimale dans la couleur**
- **Le niveau minimum de rouge**
- **Le niveau maximum de bleu**
- **Le niveau maximum de vert**

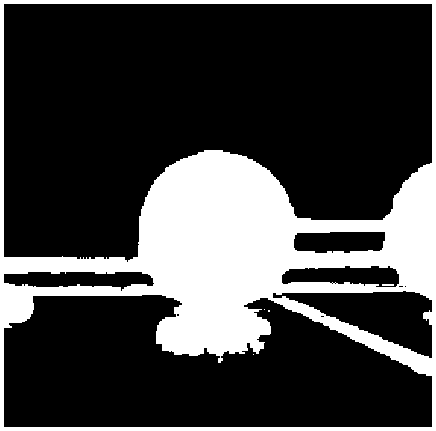
Cette sélection permettra une nette accélération des calculs de détection de contours (au lieu de travailler sur trois matrices dont les valeurs sont échelonnées entre 0 et 255, nous obtiendront une matrice binaire).

Première tentative

Condition pour qu'un pixel soit sélectionné : sa composante rouge doit représenter au moins 40% de la somme des valeurs des 3 couleurs primaires. Soit R la composante Rouge, B la Bleu, et V la verte :

$$\frac{R}{R+V+B} \geq 0.4 \quad \textbf{avec} \quad \begin{array}{l} R \in ¥, R \in [0,255[\\ V \in ¥, V \in [0,255[\\ B \in ¥, B \in [0,255[\end{array}$$

Résultat :



On vois que le contour de la balle a été correctement délimité (étant donné que l'arrière plan au niveau de la balle n'est pas de la couleur sélectionnée), mais des zones superflues ont été sélectionnées

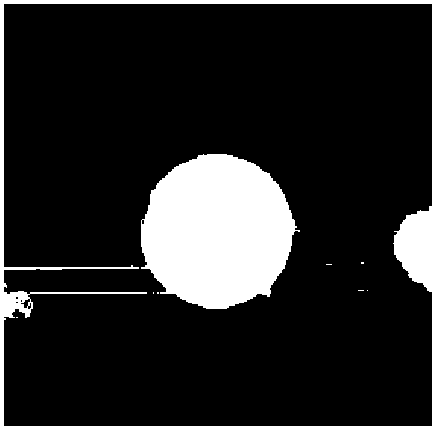
- **Zones sombres de la photo (lignes brunes)**
- **Partie de la balle jaune**
- **Reflet de la balle rouge sur le support (en dessous d'elle)**

Seconde tentative

Condition pour qu'un pixel soit sélectionné : sa composante rouge doit représenter au moins 50% de la somme des valeurs des 3 couleurs primaires (soit 10% de plus).

$$\frac{R}{R+V+B} \geq 0.5 \quad \textbf{avec} \quad \begin{array}{l} R \in ¥, R \in [0,255[\\ V \in ¥, V \in [0,255[\\ B \in ¥, B \in [0,255[\end{array}$$

Résultat :



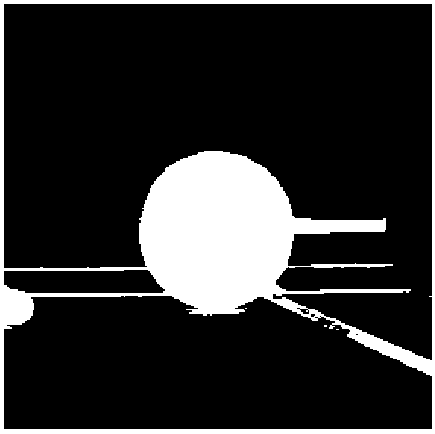
Le reflet de la balle n'est plus sélectionné, et les lignes brunes non plus. Par contre, il reste toujours une partie de la balle jaune, et les bords de la balle rouge ont étés écorchées, ce qui risque de compliquer la reconnaissance de formes.

Troisième tentative

Conditions pour qu'un pixel soi sélectionné : sa composante rouge doit représenter au moins 40% de la somme des valeurs des 3 couleurs primaires et les composantes vertes et bleues ne doivent pas avoir de valeurs supérieurs à 110 (sur 255).

$$\frac{R}{R+V+B} \geq 0.5 \quad V < 110 \quad B < 110 \qquad \textbf{avec} \qquad R \in ¥, R \in [0,255[\quad V \in ¥, V \in [0,255[\quad B \in ¥, B \in [0,255[$$

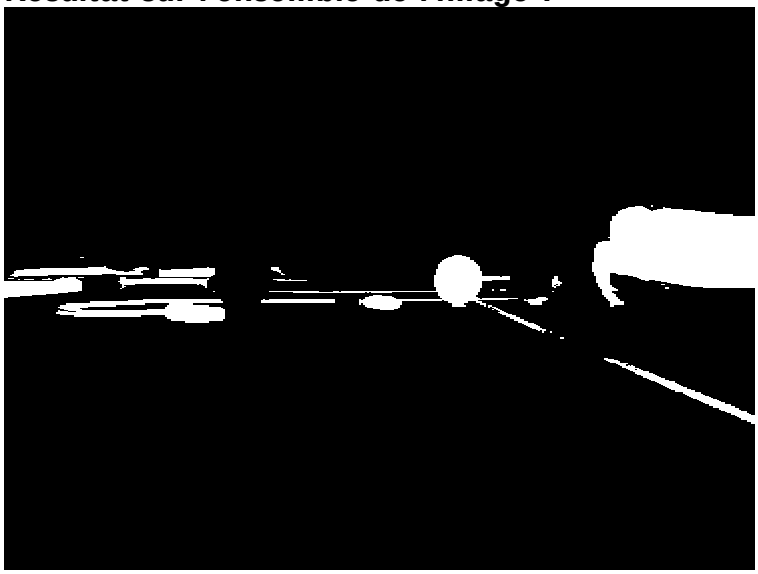
Résultat :



On peut très clairement voir que la balle jaune n'a pas été sélectionnée. La sélection des lignes horizontales est sans importance car elles seront éliminées par la reconnaissance de formes.

Les conditions choisies lors de ce dernier essai sont donc satisfaisantes pour binairiser cette image. Nous pourrions probablement encore les affiner, mais cela limiterait trop la capacité d'adaptation de l'algorithme, celui ci risquerait de mal réagir lors d'un changement d'éclairage par exemple.

Résultat sur l'ensemble de l'image :



D. Détermination de contours

Pour pouvoir reconnaître des formes sur une image, il faut tout d'abord en distinguer les contours. Ces derniers sont généralement caractérisés par une ligne de contraste, pouvant être accompagnée d'un changement de couleur. Dans notre cas, nous venons de binériser notre image. Les contours seront donc les passages du blanc au noir, et vis-versa.

Il existe différentes méthodes utilisées pour déterminer des contours. Nous allons détailler celles basées sur la convolution. Il s'agit de dérivations, car elles mettent en valeur des changements de couleur.

1) Par convolution

Pour détecter un pixel délimitant deux formes, il faut déterminer s'il appartient à une ligne de contraste. Pour cela, il faut comparer la couleur de ses pixels voisins. Cela peut être fait grâce à une convolution et une matrice noyau adaptée.

Nous allons chercher les contrastes horizontaux et verticaux séparément, ce qui permettra de définir l'orientation des contours.

La matrice noyau qui, après une convolution avec l'image, permettra de mettre en valeur le plus simplement possible les variations verticales de couleur se nomme masque de Prewitt.

-1	0	1

Si le pixel à droite de celui traité est de la même couleur que celui à gauche du pixel traité, et que cette couleur a une valeur G, le résultat de la convolution sera : -1xG + 1xG = 0. Par contre, si ces deux pixels n'ont pas la même couleur, le résultat sera différent de 0, et proportionnel au contraste autour du pixel traité. La ligne de contraste étant située entre deux séries de pixels, elles sera matérialisée par ce masque sur les pixels l'entourant. Son épaisseur sera donc de deux pixels.

Cette matrice est très simple, et ne permet pas d'obtenir une bonne appréciation de la direction du contour. La ligne de contraste peut facilement former un angle de plus ou moins 45° avec la verticale. Pour être plus précis, nous allons utiliser les matrices de Sobel.

-1	0	1
-2	0	2
-1	0	1

Contraste horizontale

-1	-2	-1
0	0	0
1	2	1

Contraste vertical

Ces matrices prennent en compte plus de pixels, et donnent un meilleur résultat, permettant de déduire la direction de la ligne de contraste plus précisément.

Exemple :

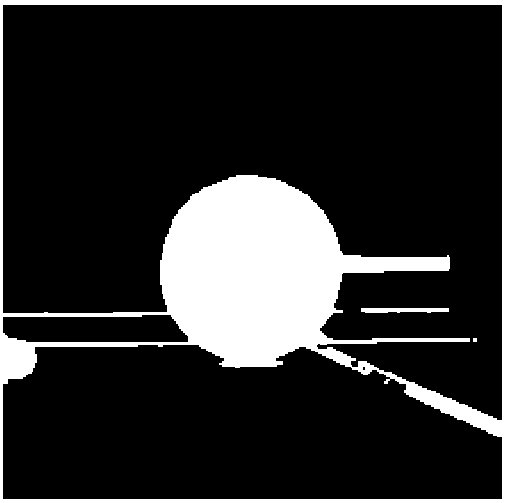
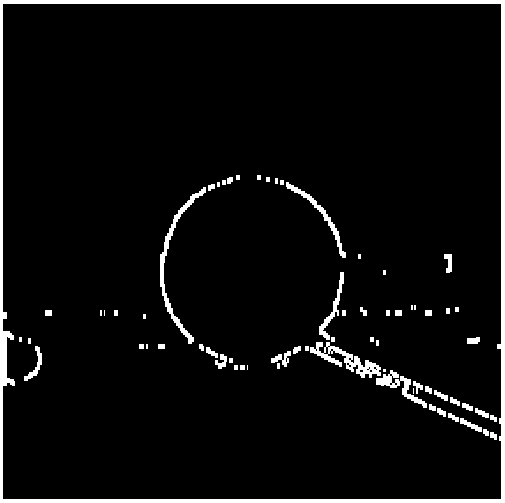
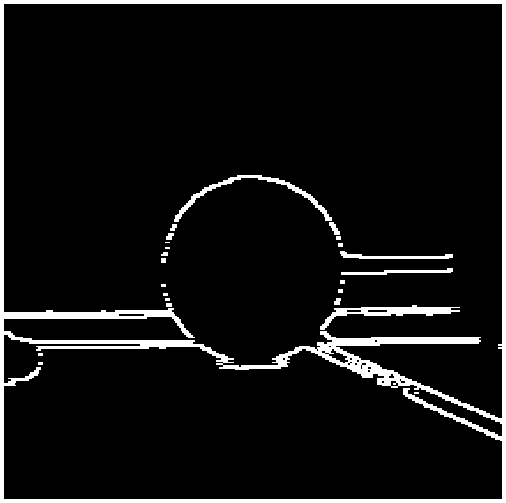


Image après filtrage anti-bruit et sélection

Résultats :



Contrastes horizontaux



Contrastes verticaux

Nous connaissons maintenant les contours de la balle rouge.

Maintenant que nous connaissons les variations de contrastes (donc les contours) sur deux dimensions, nous pouvons déduire pour chacun de leurs point le coefficient directeur de la tangente au contour en ce point, et donc le vecteur qui lui est normal. Pour cela, nous allons utiliser le théorème de Pythagore et la trigonométrie.

Soit g le vecteur défini grâce aux contrastes sur l'axe des abscisses (g_x) et sur l'axe des ordonnées (g_y). Soit g sa norme, et téta l'angle le séparant de l'axe des abscisses.

$$\mathbf{g} = \begin{bmatrix} g_x \\ g_y \end{bmatrix}$$

$$g = \sqrt{g_x^2 + g_y^2}$$

$$\theta = \tan^{-1}\left(\frac{g_y}{g_x}\right)$$

Les deux convolutions grâce aux matrices de Sobel nous ont servi à trouver les lignes de contraste de l'image (correspondant généralement à des contours de formes) et à les orienter.

2) Autres méthodes

Technique basée sur les gabarits

Dans le cas de détections de contours de formes bien particulières, on peut utiliser une technique basée sur des gabarits. Il s'agit d'un algorithme qui va tenter de placer différents gabarits sur l'image, en les adaptant grâce à des transformations géométriques de base (translations, rotations, changement de taille, …) afin de trouver les formes présentes sur l'image.
Cette solution est très peut utilisée, car elle ne permet la recherche de formes connues uniquement.

Codage de Freeman

Un contour est décrit par une succession de codes de Freeman (ce qui correspond à une chaîne de Freeman). Chaque code nous indique l'orientation du contour en un point. L'algorithme va chercher le pixel voisin le plus ressemblant de chaque pixel traité (pour former un contour), et attribuer un code au pixel traité en fonction de la position de ce voisin. La chaîne formée sera un contour.

Cette méthode est très compliquée, et il est impossible de la détailler simplement. Sa mise en œuvre nécessite une grande connaissance du traitement d'images.

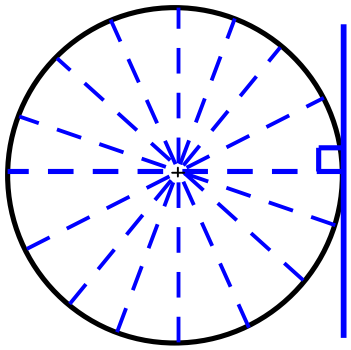
Il existe de nombreuses méthodes pour déterminer les contours d'une image, mais celles basées sur la convolution sont les plus simple à comprendre et à implanter dans une application (par exemple dans un logiciel de traitement d'images ou dans le système de commande d'un robot).

E. Reconnaissance de forme

Maintenant que nous disposons des contours des objets de la couleur de la balle que nous recherchons, il va falloir déterminer lequel est la balle.

La balle étant une sphère, quel que soit le point de vue à partir duquel nous la photographions, elle nous apparaîtra comme un disque. Son contour formera donc un cercle.

Le cercle possède une propriété géométrique unique : toutes les perpendiculaires à ses tangentes se coupent en un seul point, le centre du cercle.



Etant donné que nous connaissons toutes les orientations de tous les contours, nous allons tracer toutes les perpendiculaires des tangentes à ces contours, et le point où le plus grand nombre d'entre elles se coupent sera le centre du cercle.

En pratique, nous allons utiliser une « matrice additive ».

Procédure :

Nous créons une matrice possédant les dimensions de l'image. Il y aura donc une valeur dans cette matrice pour chaque pixel de l'image. Toutes les valeurs de cette matrice seront nulle au départ.

Pour chaque pixel appartenant à une ligne de contour de l'image :

Nous déterminons l'équation de la perpendiculaire à la tangente au contour en ce pixel

Nous incrémentons la valeur (lui additionnons 1) de toutes les cases de la matrice additive correspondant aux points par lesquelles cette droite doit passer.

Lorsque tous les contours on étés traités, les valeurs les plus élevées de la matrice additive correspondent aux points par lesquels passent le plus de perpendiculaire à des tangentes de contours. Ce seront donc des centres de cercles.

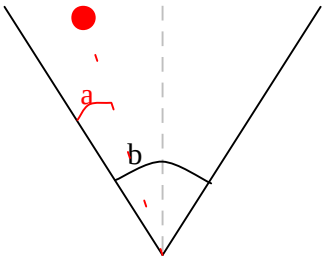
Nous avons donc trouvé le centre de la balle rouge.

F. Localisation de la balle

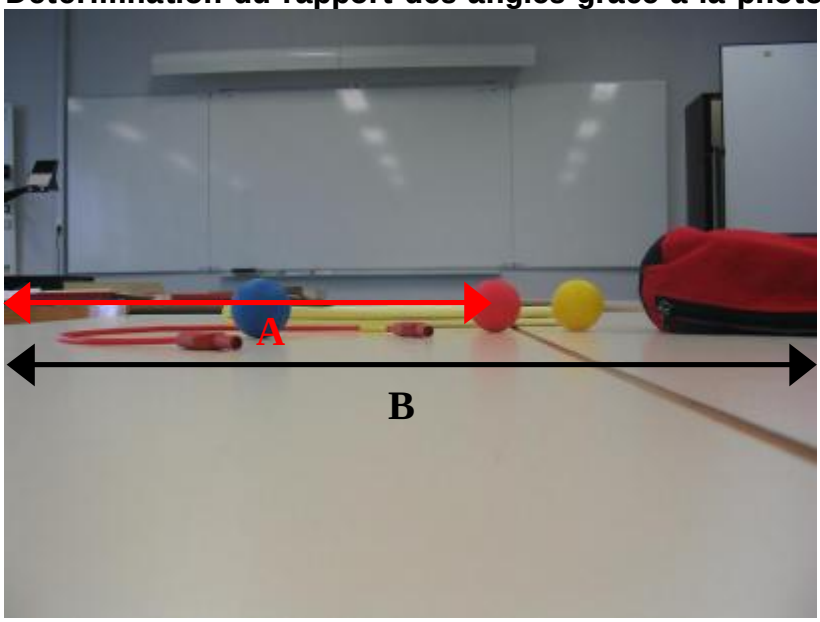
Nous désirons déterminer la position de la balle par rapport à celle de notre robot, afin qu'il puisse aller la chercher.

Grâce à une seule photo, le robot peut uniquement déterminer l'angle qui sépare la ligne de mire de sa prise de vue de la balle, étant donné qu'il connaît la position de la balle sur l'image qu'il reçoit de son capteur et que l'on peut déterminer l'angle de prise de vue de son capteur.

Ce schéma représente la prise de vue du robot, vu du dessus. En bas, le robot. Les deux trais noirs représentent les limites de son champ de vision. En rouge, la balle. Nous constatons que lorsque celle-ci se trouve dans le champ de vision du robot, nous pouvons déterminer l'angle (noté a) séparant la trajectoire du robot de la trajectoire menant à la balle. L'angle b est le demi angle de vision.



Détermination du rapport des angles grâce à la photo :



La flèche rouge représente la distance entre le bord de l'image et le centre la balle (dont nous avons précédemment déterminé la position). Longueur : A

La flèche noire représente la largeur de l'image. Longueur : B

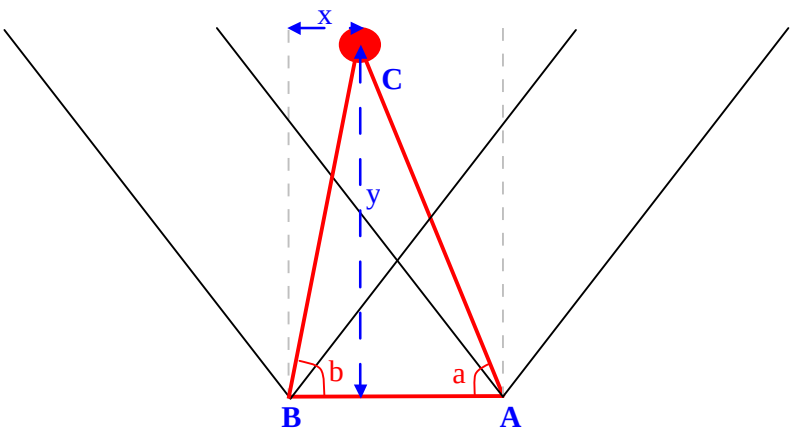
Nous établissons le rapport :

$$\frac{A}{B} = \frac{a}{b}$$

Grâce à une photo, nous avons déterminé un angle permettant de situer la balle. Le robot peut aller la chercher.

Avec deux photos prises côtes à côtes, nous allons pouvoir déterminé les coordonnées de la balle par rapport au robot, ce qui lui permettra d'une part de s'orienter pour aller la chercher, mais également d'en déterminer la distance.

Principe de trigonométrie



La balle se situe au point C. Les deux points de prises de vues se situent en A et B, en noir les limites des champs de visions.

La longueur AB est connue. Les angles a et b également.

Détermination des coordonnées x et y du centre de la balle :

$$\tan(a) = \frac{y}{AB - x}$$

$$\tan(b) = \frac{y}{x}$$

d'ou

$$y = (AB - x)\tan(a)$$

$$y = x\tan(b)$$

d'où

$$x\tan(b) = (AB - x)\tan(a)$$

$$x = \frac{AB\tan(a)}{\tan(a) + \tan(b)}$$

et

$$y = \frac{AB\tan(a)\tan(b)}{\tan(a) + \tan(b)}$$

Application pratique

Données de base :

- **Angle de vue de l'appareil photo utilisé :.Environ 70°**
- **Distance entre les 2 prises de vue : 10 cm**
- **Largeur des photos en pixels : 1600**

Position de la balle rouge :

- **A partir de la droite sur la photo de gauche : 634 pixels**
- **A partir de la gauche sur la photo de droite : 641 pixels**

Angles déduits :

- **a = 86.58°**
- **b = 84.88°**

Coordonnées de la balle par rapport à la prise de vue gauche :

- **X = 4 cm**
- **Y = 67 cm**

L'utilisation d'algorithmes pour ébruité l'image, pour déterminer les contours la composant, trouver une forme et la localiser a permis de trouver assez précisément la position de notre balle. Ces algorithmes sont donc très adaptés a ce problème, car ils permettent de trouver une solution rapidement.

Un robot peut donc parfaitement se déplacer en utilisant des informations visuelles.

Nous allons maintenant pouvoir utiliser ces informations afin de déterminer le trajet que devrai emprunter le robot afin de se rendre jusqu'à cette balle.

II. Détermination Intelligente de Trajet

Sommaire

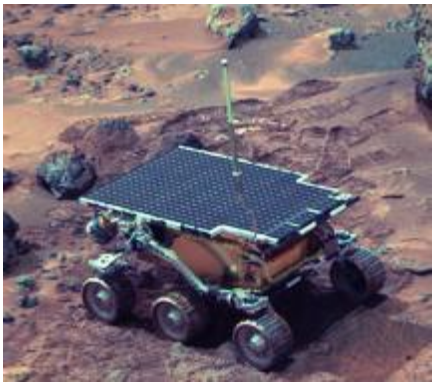
Introduction

Les algorithmes mettant en oeuvre la détermination intelligente de trajet ont beaucoup d'utilisations. Ils sont utiles dans la robotique, car ils peuvent servir à guider un robot sur un terrain difficile sans besoin d'une intervention humaine constante, ce qui s'avère utile quand le robot est sur une autre planète comme Mars, où certains types de terrains doivent être évités, mais à cause des distances extrêmes impliquées, le contrôler complètement à distance serait impossible (à cause des délais des transmissions). Ces algorithmes pourraient aussi être utiles si le robot devait opérer sous l'eau, où les ondes radio ne pourraient pas l'atteindre. Ils sont bien sûr utiles dans presque tous les cas où un véhicule doit se rendre quelque part, sans intervention humaine (par exemple la tondeuse et l'aspirateur intelligents, le robot qui imite un chien). Ces algorithmes servent aussi à trouver le plus court chemin pour conduire d'un point à l'autre sur une carte, ou le plus court chemin pour « diriger » un e-mail par un réseau d'ordinateurs. Notre but est d'appliquer cet algorithme à un robot qui, connaissant la position d'une balle [rouge], sera en mesure de planifier son déplacement « intelligent » jusqu'à cette balle.

La détermination intelligente de trajet est une application de l'intelligence artificielle et une caractéristique d'un robot qui lui permet de se déplacer « intelligemment » dans un environnement. Pour cette partie, l'intelligence mise en œuvre consiste à éviter des obstacles ou passer par certains endroits dans le but de se mouvoir d'un point source à un point destination de l'espace, et ceci en « optimisant » le trajet, c'est-à-dire en réduisant au maximum la longueur parcourue et en prenant en compte les différents types de sols sur lesquels le robot va se déplacer : ainsi, un robot-aspirateur-intelligent devra se construire peu à peu une carte « mentale » de la pièce dans laquelle il est, alors que lors de sa première visite dans cette pièce, il ne connaîtra rien ou presque des obstacles, alors qu'après un certain nombre de passages, il saura la position plus ou moins exactement des objets dans la pièce et pourra commencer à optimiser ses déplacements en fonction de ces positions, alors que le robot-chien de Sony (Aïbo) peut par exemple rejoindre le plus rapidement possible sa station de recharge quand ses batteries sont presque épuisées.

On voit donc ici deux cas distincts, qui mettent en œuvre des types d'algorithmes très différents : le premier cas suppose les coordonnées (ou positions des régions occupées) des obstacles connues, alors que dans un deuxième cas, les données de départ n'incluent pas l'environnement. Cette partie présentera uniquement les algorithmes qui fonctionnent avec les données de départ connues du robot, alors que la partie III présentera d'autres moyens pour résoudre le même problème.

Suivant les utilisations ou les données de départ, certains algorithmes pourront ou pas être utilisés, et certains seront plus efficaces que d'autres. Ce TPE va présenter les algorithmes les plus connus, comment les appliquer, ainsi que leurs avantages et leurs inconvénients.



Mars pathfinder : cet engin a été lancé sur mars afin d'étudier le sol de la planète rouge. Il se déplace de manière autonome par rapport à son environnement.

Les algorithmes de recherche de chemin

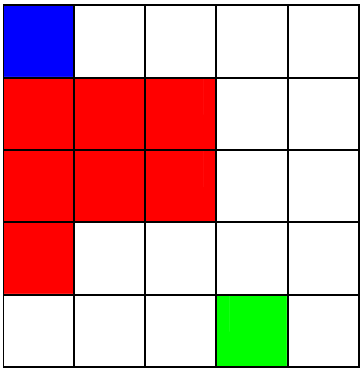
Note : La condition nécessaire aux algorithmes présentés dans cette partie est que le terrain (obstacles, élévations comme nous le verront plus tard) soit connu du robot (cette notion sera expliquée dans *Environnement et représentation*). Pour découvrir des algorithmes plus « souples », voir la partie III.

Données de départ : comment les obtenir ?

Pour connaître les coordonnées des obstacles, le robot doit être équipé de capteurs et d'une intelligence spéciale qui exploite les données des capteurs, pour construire graduellement ou en un coup ou une représentation interne de l'environnement qui l'entoure. Par exemple, une tondeuse automatique qui après un ou plusieurs passages, enregistrerait les positions et surfaces occupées des massifs de fleurs dans le but d'optimiser son « voyage de retour » construirait un représentation graduelle de l'environnement, qui deviendrait de plus en plus précise à mesure qu'elle ferait plus de passages et augmenterait ainsi son temps passé à recueillir des informations, alors qu'un système industriel utilisé dans un environnement inchangé peut être rempli dès le départ avec les positions exactes des objets qui l'entourent.

Environnement et représentation

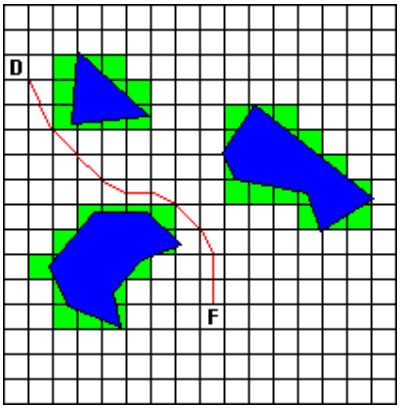
Le robot doit se déplacer du point de départ au point d'arrivée en évitant les obstacles :
Si les régions occupées par les obstacles sont connues, on peut représenter les obstacles / espaces libres par une matrice de dimensions arbitraires :



Exemple de matrice. Les obstacles sont représentés en rouge, le point d'arrivée en vert (de coordonnées 4 et 5) et le point de départ en bleu (de coordonnées 1 et 1).
(les cases sont repérées sous la forme (colonne, ligne) pour un repérage plus explicite).

Tous les algorithmes de cette partie se basent sur une matrice similaire qui représente un environnement numérisé. (le rectangle et le carré rouges pourraient par exemple représenter une table et une chaise).
Les dimensions de la matrice (nombre de lignes et de colonnes) dépendent du niveau de détail avec lequel on veut représenter les objets (obstacles) : avant de pouvoir représenter un objet de forme complexe, il faut le transformer en une série de carrés sur la grille pour qu'il soit représentable. Cette transformation ne fait pas l'objet de ce TPE, néanmoins il faut signaler que plus on agrandit les dimensions de la grille, plus l'objet sera représenté avec finesse, permettant au robot de ce déplacer plus près des obstacles pour gagner peut-être quelques mètres dans le déplacement final (suivant les différentes tailles de matrices).

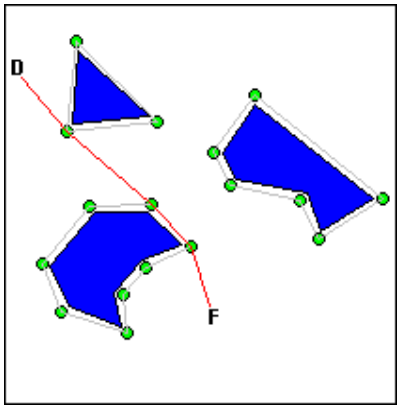
Autres représentations



Représentation matricielle.

Les cases vertes représentent les « zones de collision » où on ne peut pas passer.

Le chemin optimal est une succession de cases de la matrice.



Représentation polygonale.

Les points verts sont les points par lesquels ont peut passer.

La matrice n'est pas la seule forme de représentation de l'environnement : il en existe d'autres comme la représentation polygonale, qui peuvent diminuer la distance de déplacement.

Poids des cases

Une case peut être pondérée, dans le but de passer en priorité sur certaines zones du terrain : par exemple, pour le robot-chien [Aïbo], une case affectée du poids 1 serait un sol « normal » alors qu'une case avec un point 2 serait un tapis, où il peut quand même y marcher si l'éviter dans le trajet global constituerait une trop grosse dépense en terme de temps ou de distance, mais où il serait ralenti par exemple (il peut aussi être instruit spécifiquement par l'utilisateur d'éviter une zone très « fréquentée » lorsque cela est possible). On pourrait dans le même genre demander à la tondeuse automatique quand elle retourne à son point de départ d'éviter si possible des portions de terrain recouvertes de macadam, ou la laisser déterminer un plus court chemin lorsque le terrain n'est pas plat (les zones auraient un poids plus ou moins proportionnel à leur altitude, par exemple), et ainsi de suite. Cependant, tous les algorithmes ne prennent pas en compte le poids des zones.

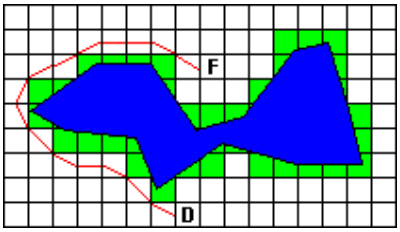
Temps de calcul

Sur un ordinateur, tous ces algorithmes ont un temps de calcul à peu près équivalent, à cause des performances des processeurs actuels et de la faible demande en puissance de calcul et du fait que l'utilisation de la mémoire pour de tels algorithmes est faible devant la taille des mémoires actuelles. Néanmoins, dans le cas des utilisations pour la cybernétique et la robotique, les processeurs sont largement moins puissants et les mémoires moins importantes ou non installées, et ces dispositifs sont suffisants pour le contrôle direct ou télécommandé du trajet. Mais quand on veut implémenter un mode de raisonnement plus compliqué, algorithmique, on doit parfois ajouter des mémoires externes ou adopter un microprocesseur plus puissant qui soit capable de gérer tout le robot en temps réel. Certains algorithmes de détermination de

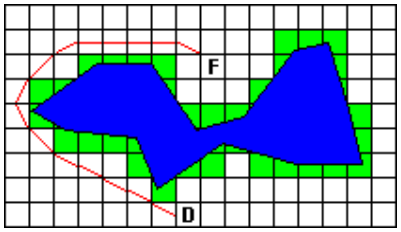
trajet demandent plus de puissance de calcul que d'autres, et certains plus de mémoire que d'autres.

Lissage du chemin

Le robot a besoin de points, de coordonnées précises par lesquelles passer : les cases de la matrices devront être utilisées pour transformer des zones de passages en points de passage, mais cette transformation peut laisser des zones de virage brusques. Le lissage permet de rendre le chemin pris par le robot le plus « naturel » possible et permet aussi d'éviter des changements de directions trop raides ou inutiles :

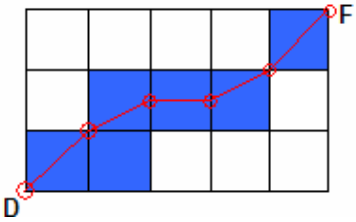


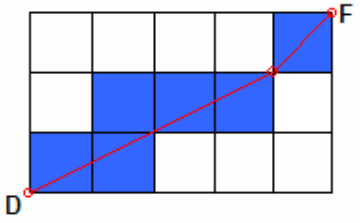
Chemin sans amélioration. Le chemin comporte 7 brisures.



Chemin avec amélioration. Le chemin comporte 4 brisures.

Il y a plusieurs méthodes pour lisser un chemin : on peut l'améliorer simplement en reliant des points situés sur des cases adjacentes (cf. illustration ci-dessus). Exemple :

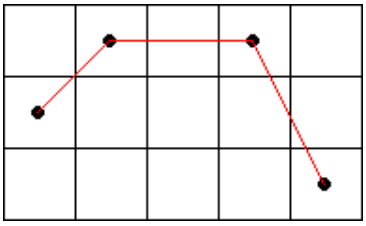




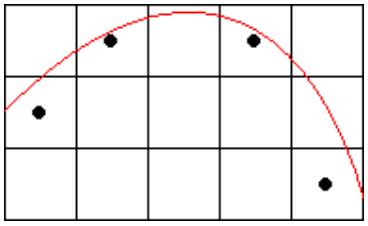
En bleu : zones de passage.
En rouge : points de passage.

A gauche, un chemin non lissé qui oblige le robot à tourner quatre fois. A droite le chemin a été lissé pour réduire au maximum les aspérités du trajet, et le robot ne tourne plus qu'une fois.

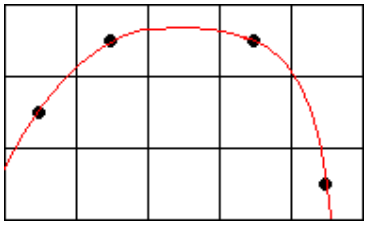
On peut aussi calculer des points intermédiaires à partir des points obligatoires, en utilisant les courbes de Bézier ou les courbes de Catmull-Rom. Mais il faut alors faire attention à ce que la courbe ne passe pas au-delà des cases marquées comme « praticables » (surtout pour les courbes de Bézier) :



Chemin calculé



Lissage par courbe de Bézier



Lissage par courbe de Catmull-Rom

Les courbes sont sources d'ordres plus compliqués pour le robot et nécessitent un système de déplacement qui puisse varier la direction pendant le déplacement. Mais si le robot présent de tels équipements, elles servent à réduire le temps que met le robot à se déplacer du point de départ au point final.

Les algorithmes

L'analyse précédente nous a permis de dégager les points importants qu'un algorithme doit présenter et qui permettront de les classer.

Nous avons donc :

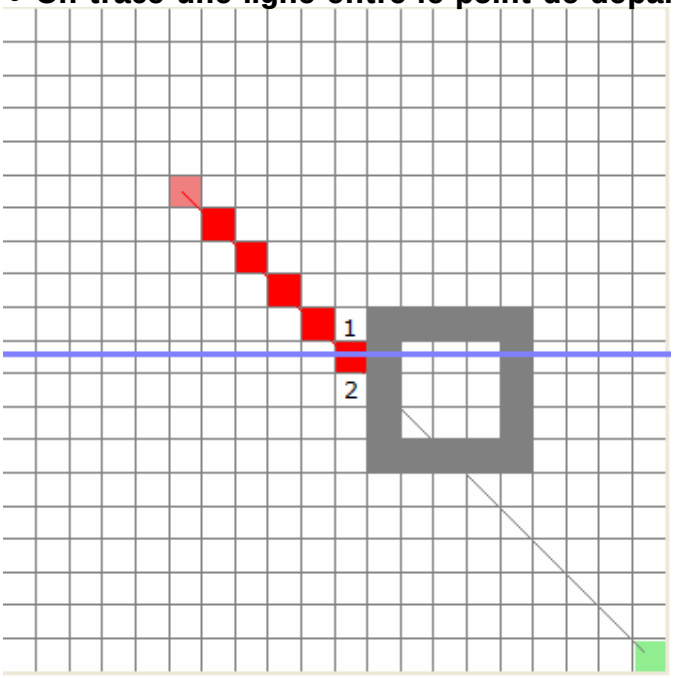
- **L'optimisation de la longueur du trajet (important)**
- **Le temps de calcul (important)**
  Le temps de calcul nécessaire à un algorithme est aussi appelé c*omplexité temporelle* de cet algorithme.
- **L'utilisation mémoire (moyennement important)**
  La taille de la mémoire nécessaire au fonctionnement d'un algorithme est appelée c*omplexité spatiale* de l'algorithme.
- **La prise en compte ou non du poids des cases (moyennement important).**

1.La ligne droite

C'est l'algorithme le plus simple, car c'est celui qu'on applique instinctivement : en général, on marche jusqu'à un obstacle avant de le contourner, par la gauche ou par la droite, alors qu'on connaît la direction approximative du point où l'on veut aller.
L'implémentation de cet algorithme pourrait s'effectuer uniquement avec des capteurs et un système de commande asservi, sans besoin d'informations précises sur l'environnement. Mais les positions précises des obstacles seront ici utilisées pour affiner le trajet et dans les calculs.

Principe de l'algorithme

• On trace une ligne entre le point de départ et le point d'arrivée :



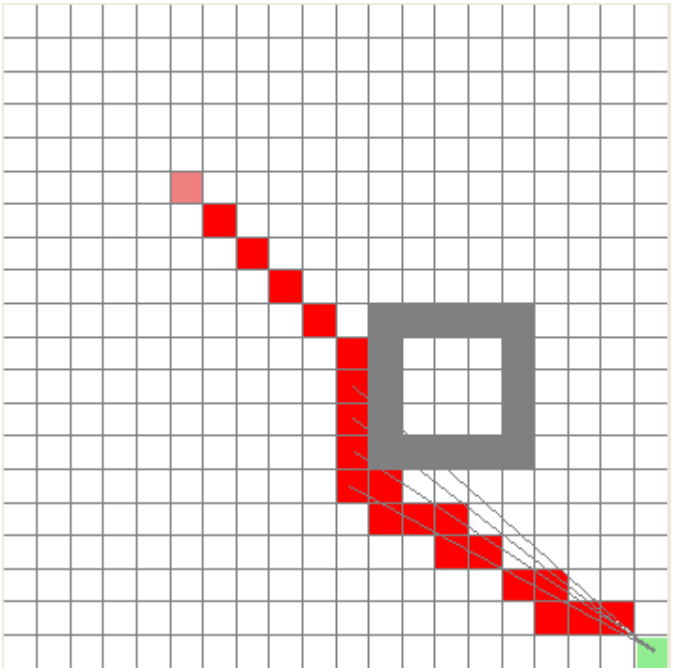
Matrice exemple
Point rouge : point de départ (D).
Point vert : point destination (A).
Cases grisées : obstacles.
Pour le calcul de distances et le tracé de lignes, on utilisera le point au centre des cases.

• On fait avancer le robot jusqu'à l'obstacle (cases rouges), puis on choisit une direction dans laquelle le déplacer par rapport à l'obstacle (cases repérées). Ici, le choix entre la case 1 et la case 2 peut se faire soit au hasard (on choisit toujours une des deux directions), soit en utilisant une méthode de prédiction : on peut par exemple séparer le terrain en deux parties et diriger le robot dans la partie comportant le moins de cases grisées (ligne bleu clair), en supposant que si il y a moins de cases grisées dans une partie, cela implique moins d'obstacles, donc un chemin plus court. Dans ce dernier cas, la case 1 serait choisie en priorité (5 cases complètes contre 9). On peut aussi mesurer la distance entre la case 1 et la case 2 au point d'arrivée :

| 1->A | = √181 = 13,45
| 2->A | = √145 = 12
(où | A -> B | signifiera « distance du point A au point B » dans ce TPE)

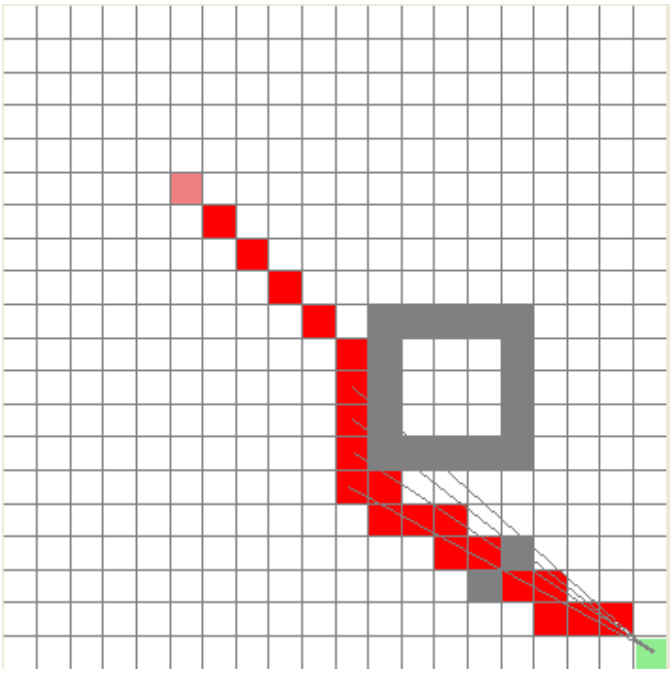
Dans ce cas, c'est la case 2 qui l'emporte.
On peut utiliser la séparation des parties quand les longueurs renvoyées par la seconde méthode sont égales, par exemple.

• On décide de faire bouger le robot à la case 2, et on recommence : on re-trace une ligne et on détermine une autre position pour le robot, et ainsi de suite. Dans cette situation, on arrive rapidement jusqu'au but :



Exemple de chemin complètement déterminé.

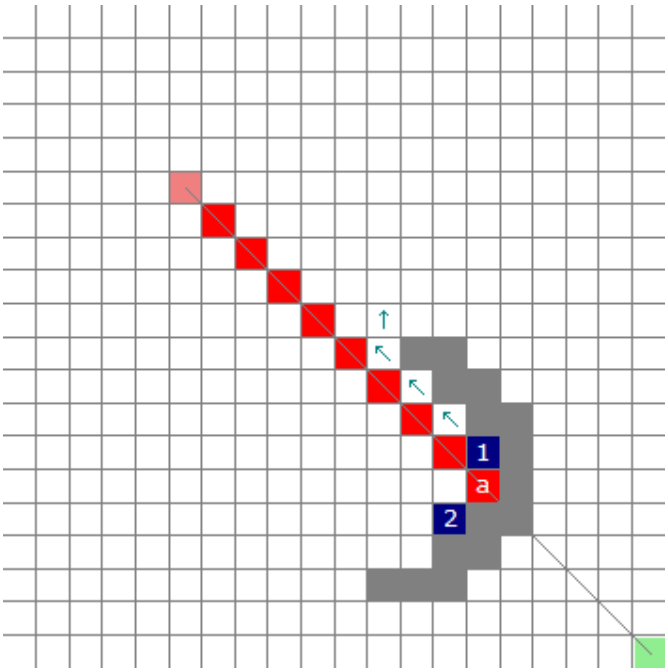
Remarque : Les cases en rouge sont celles sélectionnées par l'algorithme, c'est-à-dire celles par lesquelles la ligne passe directement. Mais le robot a une taille qui doit être prise en compte. Par exemple, examinons la situation suivante :



Situation problématique !

Ici, les deux obstacles rajoutés n'ont pas été évités car la dernière ligne est passée juste entre les deux : le robot se retrouverait bloqué s'il suivait le chemin décidé !
Pour remédier à cette situation, on peut par exemple sélectionner toutes les cases situées à une certaine distance (en fait, le rayon du cercle circonscrit au robot) de la ligne en parcourant celle-ci, au lieu de ne sélectionner que les cases « sous » la ligne.

Un autre problème se situe au niveau des obstacles concaves : imaginons la situation suivante :



Quand le robot arrive en a, il choisit entre la case 1 et la case 2. Supposons que l'algorithme lui fasse choisir la case 1; dans ce cas, l'examen des cases adjacentes désignera la case a comme successeur, qui désignera la case 1 et ainsi de suite ! La même situation est valable pour la case 2.
Pour remédier à ce problème, on peut « marquer » les cases déjà visitées sur le chemin pour qu'elles ne soient plus sélectionnées, mais cela dépasse le cadre de ce simple algorithme.

Résumé

1. **Soient A et B deux points. Le point A est défini comme le point de départ et B le point d'arrivée.**
2. **On trace une ligne entre le point A et le point B.**
3. **Si il y a un obstacle entre A et B, alors on marque toutes les cases traversées par la ligne jusqu'à l'obstacle, on choisit un successeur C pour le point A (successeur veut dire une case adjacente), puis on recommence l'étape 2 en définissant A = C.**
4. **Si il n'y a pas d'obstacle, alors on marque les cases traversées par la ligne, et l'algorithme se termine.**

Problèmes connus

- **Dans le cas d'un obstacle concave, si on n'utilise pas une méthode de mise en mémoire des cases visitées, alors l'algorithme peut rester « bloqué » entre deux cases.**
- **Une ligne qui passe exactement entre deux cases peut ne pas désigner ces deux cases comme points de passage, résultant en un blocage du robot, tout dépend de la manière dont on désigne les cases de passage à partir de la ligne.**

Avantages

La mise en place de cet algorithme est très simple, et il nécessite très peu de calculs.

Si il n'y a aucun ou peu d'obstacles (et qu'ils sont de faible taille), alors on arrivera très vite au but en un minimum de temps.
Les calculs nécessaires ne le sont que pour l'équation de la ligne et la définition des cases « sous » cette ligne, ainsi que le test d'un obstacle présent : il présente une faible complexité temporelle.
Enfin, il n'utilise quasiment pas de mémoire pour stocker ses données de fonctionnement : il a une faible complexité spatiale.

Inconvénients
Par contre, cet algorithme ne trouve pas forcément un chemin optimal, tout dépend de l'ordre dans lequel lui sont donnés les cases adjacentes (comme nous l'avons vu, cet ordre peut être défini à l'aide de calcul de distances ou comptage de cases, mais cela rajoute une surcharge de calcul pour le processeur). Il peut même ne pas trouver de chemin du tout, et on ne peut lui appliquer aucune méthode pour prendre en compte le poids des cases (qui se traduit par une élévation ou un différent type de terrain).

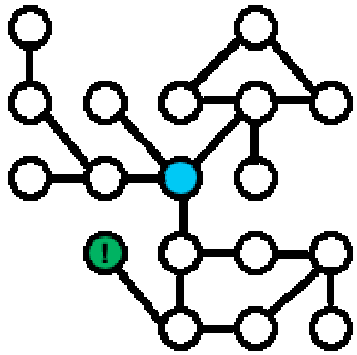
En résumé, cet algorithme est impensable à utiliser pour un robot qui se respecte. Il peut par contre être utilisé dans des robots plus petits et disposant de moins de puissance de calcul. On verra que cet algorithme servira par la suite, comme sous algorithme dans d'autres plus performants.

2.Depth First Search (DFS)

L'algorithme DFS (Depth First Search, « profondeur d'abord »), est à l'origine un algorithme de traversée de graphes. Son originalité est de parcourir tous les enfants des nœuds d'un graphe avant ses voisins (et d'être très simple). En quoi un algorithme de parcours de graphes pourrait – il être utile dans notre situation ?

<u>Analogie graphe / notre terrain</u>

Le DFS travaillant sur des graphes, il a fallu l'adapter pour pouvoir travailler avec notre matrice. Pour cela, nous avons défini une analogie entre les composants d'un graphe et les composants de notre matrice.



Un problème typique des graphes : on a un point de départ (bleu), un point d'arrivée (vert), et on cherche un chemin pour les joindre.

Un graphe est composé de nœuds connectés entre eux par des liens : sur la figure ci-dessus, les nœuds sont représentés par des disques et les liens qui les relient sont des lignes noires. Le problème que le DFS résout est le suivant : donnés un nœud source et un nœud cible, par quels nœuds doit-on passer, quelles routes doit-on emprunter pour joindre ces deux nœuds en utilisant les routes existantes ?
Par comparaison, notre problème est de déterminer par quelle(s) case(s) passer alors qu'on cherche à traverser la matrice d'une case à une autre.

On définit ainsi l'analogie suivante :

- **Les cases peuvent être assimilées à des nœuds ;**
- **Les liens entres les nœuds n'ayant pas d'équivalent direct, on donnera à chaque case huit liens au maximum la liant avec les cases adjacentes (une case est liée avec toutes ses cases adjacentes qui ne sont pas des obstacles) ;**
- **Un successeur d'un nœud A (c'est-à-dire un nœud B relié à ce nœud A) représente une case adjacente B à la case A.**
- **Les nœuds spéciaux, c'est-à-dire le nœud source et le nœud cible, seront assimilés à la case de départ et la case d'arrivée.**
- **Les poids permettant de pondérer les cases ne peuvent pas être pris en compte dans des algorithmes travaillant sur des graphes.**
- **Enfin, la distance pour aller d'un nœud à un autre ne correspond pas à la distance réelle « sur le terrain », et on ne peut la calculer qu'après avoir transformé les deux nœuds en cases et ensuite seulement peut-on mesurer la distance entre ces cases.**

Du fait de l'analogie précédente, on peut définir plusieurs objets dans notre programme : pour l'instant, les nœuds seront nos seuls objets. Les routes (liens)

n'ont pas besoin d'être représentées. Les objets que nous définirons auront des caractéristiques (par exemple pour un nœud, sa position dans le graphe) et des fonctions (toujours pour un nœud, par exemple une fonction pour connaître tous les nœuds adjacents), comme cela est expliqué ci-dessous.

Les structures de données

Pour représenter ces objets (nœuds, liens) dans le programme, il nous faut des structures de données qui regroupent les propriétés et les méthodes des objets. Par exemple, un nœud doit contenir :

- **Les propriétés X et Y, deux nombres entiers positifs qui contiennent un moyen de repérage de la case que ce nœud représente (je le rappelle, X est la colonne de la case dans la matrice et Y est la ligne correspondante).**
- **Une propriété Colored, représentant une variable booléenne (0 ou 1) permettant de savoir si le nœud a été « colorié » ou non (l'usage des nœuds coloriés sera expliqué plus en détail dans le fonctionnement de l'algorithme).**
- **Une méthode GetSuccessors qui permet d'obtenir tous les successeurs d'un nœud, c'est-à-dire les cases adjacentes.**
- **Une méthode (fonction) Compare pour comparer un nœud à un autre (ce qui paraît trivial pour nous mais est nécessaire au programme).**

Dans le programme, ces propriétés et méthodes seront représentées comme suit :

```
structure node
{
      entiers : X, Y à propriétés
      bool : Colored à propriétés
      liste : GetSuccessors() à fonction
      bool : Compare( node : B ) à fonction
}
```

(où `bool` signifie « variable booléeenne »)

La fonction GetSuccessors retourne une liste contenant tous les successeurs d'un nœud : voici l'algorithme permettant de trouver la position des cases adjacentes à une autre :

```
liste : GetSuccessors()
{
      entiers : Xs, Ys
      liste : Successeurs
      pour i = 1 jusqu'à i = -1
      {
            pour j = 1 jusqu'à j = -1
            {
                  Xs = i + X
                  Ys = j + Y
                  Si (Xs,Ys) est en dehors de la grille Alors passer ce nœud
                  Si (i,j) = (0, 0) Alors passer ce nœud
                  Si (Xs, Ys) est un obstacle Alors passer ce nœud
                  Successeurs : Ajouter (Xs,Ys)
            }
      }
      Renvoyer la liste Successeurs
}
```

On remarque que avec cet algorithme, les nœuds adjacents sont retournés dans l'ordre suivant :

8	**5**	**3**

7		2
6	4	1

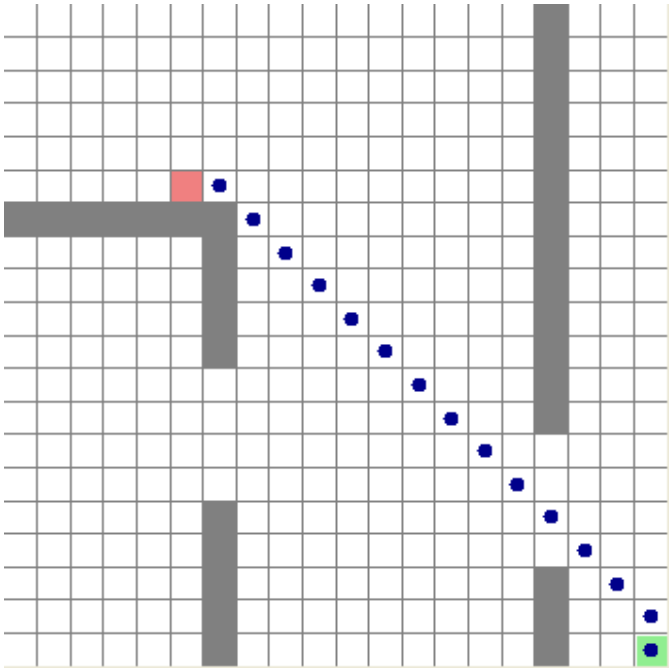
L'ordre dans lequel les nœuds sont placés a une importance cruciale pour le DFS et moindre pour les autres algorithmes. Globalement, on peut retenir que si le premier nœud retourné est situé dans la direction du nœud d'arrivée (souvenez vous que nœud équivaut à case), alors les algorithmes seront globalement plus rapides (et beaucoup plus efficace pour le DFS, cf. suite).

La fonction Compare d'un nœud A est très simple mais nécessaire au programme qui ne fait que des comparaisons entre valeurs numériques. Elle accepte comme variable d'entrée un autre nœud B, compare le nœud A(X, Y) auquel elle appartient et B(B.X, B.Y) et retourne 0 ou 1 suivant l'égalité ou non :

```
bool : Compare( node : B )
{
      Si (B.X = X) ET (B.Y = Y) à on compare les valeurs X et Y de B aux valeurs X
         et Y du nœud courant
      Alors Renvoyer la valeur 1
      Sinon Renvoyer la valeur 0
}
```

Principe de l'algorithme

Soit la situation suivante :



Rappel : la case rouge est le point de départ, la verte le point d'arrivée. Les cases grisées sont des obstacles (ici, on peut imaginer des murs et des portes, par exemple).

La manière de procéder est la suivante :

1. **Soient deux nœuds A_1 et B. On définit A_1(6, 6) le point de départ, et B(20, 20) le point d'arrivée. On définit n = 1**
2. **On colorie le nœud A_n sur la grille (ici, les nœuds colorés sont marqués à l'aide de cercles bleus).**
3. **On crée une liste contenant chacun des successeurs de An.**
4. **Pour chaque successeur A_{n+1} :**
   - **Si A_{n+1} est colorié, on examine le successeur suivant**
   - **Si A_{n+1} = B, c'est-à-dire que A_{n+1} est le point d'arrivée, alors on a trouvé le but.**

- **On incrémente n et on recommence(1) l'étape 2**
- **S'il n'y a plus de successeurs, on décrémente n et on revient(2) à l'étape 4**

Le nom depth-first vient du fait qu'on examine directement le premier successeur d'abord (on recommence à l'étape 2 sans avoir examiné tous les successeurs). Cet algorithme est récursif : au (1), on repasse à l'étape 2, c'est-à-dire qu'on réappelle l'algorithme avec un autre nœud en paramètre. Il faut imaginer l'étape 4 comme une boucle, et on ne sort (2) de la boucle que quand il y n'y a plus de successeurs à examiner. La couleur sert à ne jamais examiner deux fois le même nœud, en le marquant. Cela est mieux traduit par le programme de l'algorithme : (je trouve qu'un algorithme récursif est mieux expliqué au travers d'un programme)

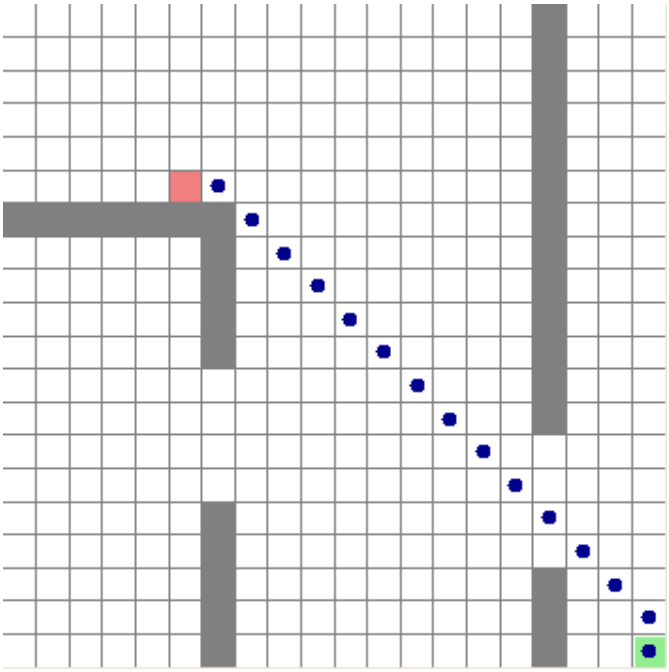
```
Départ de l'algorithme:

node : Start = node : (6, 6)
node : Goal = node : (20, 20)
liste : Chemin
bool : réussi
réussi = dfs ( Start )

Fonction dfs (récursive):
bool : dfs ( node : A )
{
      Colorier A
      liste : Successeurs
      Successeurs = A.GetSuccessors()
      Pour chaque node : C dans Successeurs
      {
            Si C est colorié Alors passer au successeur suivant
            Si C.Compare(Goal) = 1 Alors
            {
                  Chemin : Ajouter C
                  Arrêter la boucle
                  Retourner la valeur 1
            }
            bool : trouvé
            trouvé = dfs ( C )
            Si trouvé = 1 Alors Arrêter la boucle ET Retourner la valeur 1
      }
      Retourner la valeur 0
}
```

C.Compare(A) compare C à A et retourne 1 si C = A, sinon 0.
Ici, bool : réussi et bool : trouvé sont des variables booléennes qui contiennent la valeur de retour de la fonction dfs, qui déterminera si la recherche a réussi à trouver le nœud d'arrivée ou non.
Chemin est une liste qui contiendra, après la fin de l'algorithme, tous les nœuds que l'algorithme a traversé, dans l'ordre inverse, pour arriver au nœud final. Il suffira au robot de parcourir cette liste à partir de la fin pour retrouver la liste des cases sur lesquelles il doit passer.
La fonction retourne la valeur 0 quand il n'y a pas ou plus de successeurs à examiner.

La fonction dfs_1 est appelée pour la première fois avec le nœud de départ en paramètre (tel que A = Start = 6, 6). Ce nœud est examiné, donc colorié pour éviter de rappeler la fonction dfs sur ce nœud si jamais il était sorti comme successeur d'un nœud adjacent. Ses successeurs sont sortis et mis dans la liste Successeurs. On utilise ensuite la fonction dfs_2 sur le premier successeur, si il

n'est pas colorié et s'il n'est pas le goal ; la fonction appelée s'appelle sur le premier successeur du successeur, et ainsi de suite.
Quand on arrive à un bout de la carte par exemple, il est possible qu'un nœud n'ait pas de successeur car tous les nœuds adjacents ont été coloriés ou sont des obstacles. Dans ce cas, la fonction dfs_n retourne 0, ce qui permet à la fonction dfs_{n-1} qui l'a appelée de passer au successeur suivant, et ainsi de suite, ce qui permet de couvrir toute la carte par récurrence.
Si le goal est trouvé, il est ajouté à Chemin, et la fonction dfs_n retourne 1. La fonction dfs_{n-1} qui l'appelée, remarque que la fonction dfs_n a retourné la valeur 1, ajoute le successeur qu'elle examinait à la liste Chemin, puis retourne elle-même la valeur 1, ce qui permet à la fonction dfs_{n-2} d'ajouter son nœud à Chemin et ainsi de suite, jusqu'à dfs_1 qui examinait un des successeurs du nœud Start, et qui ajoute ce successeur à la liste Chemin.
Si le goal n'est jamais trouvé (par exemple si il est inaccessible), alors dfs_∞ n'aura pas de successeurs car ils seront tous déjà visités ou des obstacles ; dfs_∞ retourne 0. De proche en proche, les fonctions appelantes seront aussi à court de successeurs, et retourneront 0. On arrive ainsi jusqu'à dfs_1 qui retourne 0, permettant ainsi au programme principal de savoir que la recherche a échoué.
La liste Chemin permet de faire une trace des nœuds solutions menant de Start à Goal.
Elle contient les solutions dans l'ordre inverse, En commençant par un successeur de Start et en terminant par Goal.

Voyons comme exemple notre situation :



20, 20
20, 19
19, 18
18, 17
17, 16
16, 15
15, 14
14, 13
13, 12
12, 11
11, 10
10, 9
9, 8
8, 7
7, 6

ß dernier élément de Chemin

Ordre des nœuds retournés par GetSuccessor() :

8	5	3
7		2
6	4	1

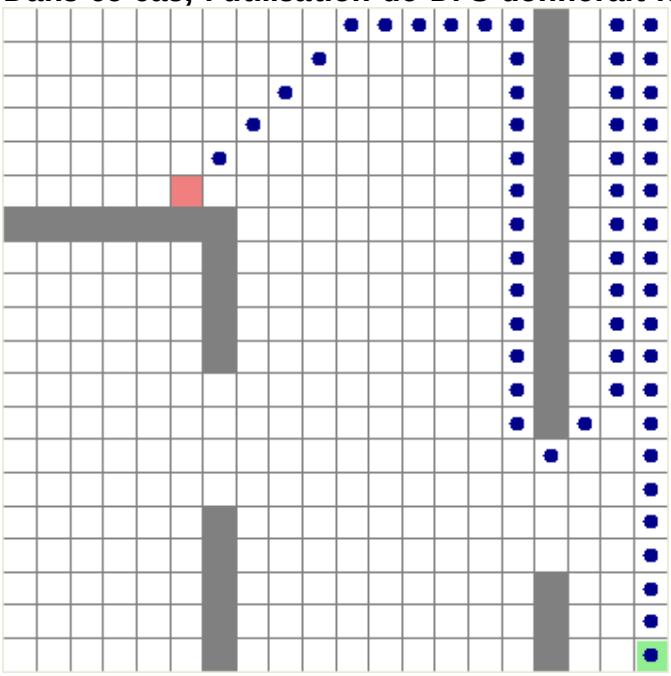
ß premier élément de chemin

Influence de la fonction GetSuccessors

Vu que la fonction dfs examine le premier successeur d'abord, l'ordre dans lequel ces successeurs apparaissent est d'une importance primordiale : ainsi, dans cet exemple, le nœud en bas à droite du nœud courant est retourné en premier, résultant en un chemin assez court. Mais imaginons que les nœuds soient retournés dans cet ordre :

6	4	1
7		2
8	5	3

Dans ce cas, l'utilisation de DFS donnerait le chemin suivant :



Un chemin incohérent

Comme on le voit, le fait que les nœuds situés en haut à droite soient retournés les premiers influence grandement le chemin pris par l'algorithme : en effet, au départ, le nœud situé en haut à gauche du premier est pris, puis celui en haut à gauche du second, et ainsi de suite.
On voit que cet algorithme garantit un chemin jusqu'à l'arrivée (si c'est possible, évidemment), mais pas forcément le plus court, et il peut chercher dans tous les nœuds de la carte jusqu'à trouver le nœud d'arrivée, même si celui-ci est situé juste à côté du nœud de départ !
Une amélioration de cet algorithme serait de mettre le premier nœud retourné par GetSuccessors() dans la direction approximative du nœud d'arrivée, ce qui aboutirait à une convergence plus rapide vers le goal, et éviterait certains chemins qui, pourtant triviaux « sur le terrain » seraient beaucoup plus compliqués une fois retournés par l'algorithme DFS.

Unité de calcul

Cet algorithme utilise la récursivité. Le problème d'une fonction récursive, c'est qu'elle peut s'exécuter indéfiniment (ou pendant une très longue période de temps) sans qu'il soit possible de l'interrompre, empêchant ainsi le robot de pouvoir faire d'autre actions (comme par exemple interpréter et récupérer les informations des capteurs) pendant qu'il réfléchis au chemin à prendre. C'est pour cela qu'on peut introduire la notion d'unité de calcul pour les algorithmes récursifs : le but est de « casser la récurrence », c'est-à-dire permettre à la fonction d'être appelée plusieurs fois, autant de fois qu'il faudra pour résoudre le problème posé, et faisant un certain nombre (réduit au maximum) de calculs à chaque appel, permettant à la fonction d'être appelée plusieurs fois, indépendamment, permettant au robot d'effectuer d'autres calculs avant et après. De plus, cette manière de procéder a d'autres avantages, facilitant notamment le déboguage (c'est-à-dire la recherche d'erreurs dans le programme) en permettant d'observer la recherche pas à pas, par exemple.
Dans notre exemple, il faudrait inventer une fonction DFS qui sauvegarderait sa « position » dans le calcul (c'est-à-dire les nœuds visités) à chaque appel, ce qui se révélerait extrêmement utile.

Résumé

Pour trouver un chemin dans un graphe ou une matrice à l'aide de l'algorithme DFS :

- **Il faut colorier les nœuds traversés**
- **On traverse chaque successeur de chaque nœud en le coloriant jusqu'au dernier successeur non colorié du dernier nœud non colorié, puis on remonte progressivement grâce à la récursivité, jusqu'au nœud de départ si on ne trouve pas le nœud d'arrivée. Si on le trouve, on remonte les nœuds en les ajoutant à une liste qui permettra de retrouver le chemin parcouru.**

Problèmes connus

Cet algorithme trouvera toujours un chemin si le nœud d'arrivée est à portée. Par contre, il est récursif, ce qui signifie qu'il faut séparer le code en unités de calcul pour qu'il soit réellement utilisable par un robot.

Avantages

La convergence de cet algorithme peut être très rapide, si les successeurs sont donnés dans le bon ordre (en fait la convergence la plus rapide de tous les algorithmes présentés, comme nous le verrons plus tard).

Inconvénients

Cet algorithme ne retourne pas toujours un chemin optimal : en effet, si les successeurs sont donnés dans le mauvais ordre (ou plutôt dans un mauvais ordre), alors le chemin obtenu sera praticable (il respecte la condition de parcours, du nœud de départ vers le nœud d'arrivée), mais n'est pas forcément le meilleur (même souvent aberrant pour des cas triviaux). Il faudrait en effet proposer un système de modification automatique de l'ordre des successeurs pour obtenir un algorithme vraiment efficace. De plus, cet algorithme ne tient pas compte des poids des cases, et il serait très difficile de le modifier pour lui en faire tenir compte. Il examine aussi un très (trop) grand nombre de nœuds si les successeurs ne sont pas retournés dans le bon ordre.

En résumé, cet algorithme présente un trop grand nombre de calculs pour être retenu dans un contexte robotique, car il demande trop de puissance de calcul pour être optimisé.

3.Breadth First Search (BFS)

L'algorithme Breadth First Search, communément appelé BFS, est aussi un algorithme de traverse de graphes, comme le DFS. L'originalité de cet algorithme est qu'il traverse d'abord les voisins d'un nœud avant ses enfants. De plus, le BFS utilise des listes pour garder la trace des nœuds qu'il doit traverser (cela sera expliqué plus tard).
Il utilise aussi la coloration des cases pour éviter de réexaminer les cases qu'il a déjà examinées.
En tant qu'algorithme de traversée de graphes, il réutilise le principe des nœuds déjà évoqué avec l'algorithme depth first search.

Les listes

Avant de parler du premier algorithme qui utilise les listes, je vais d'abord parler des listes elles – mêmes.

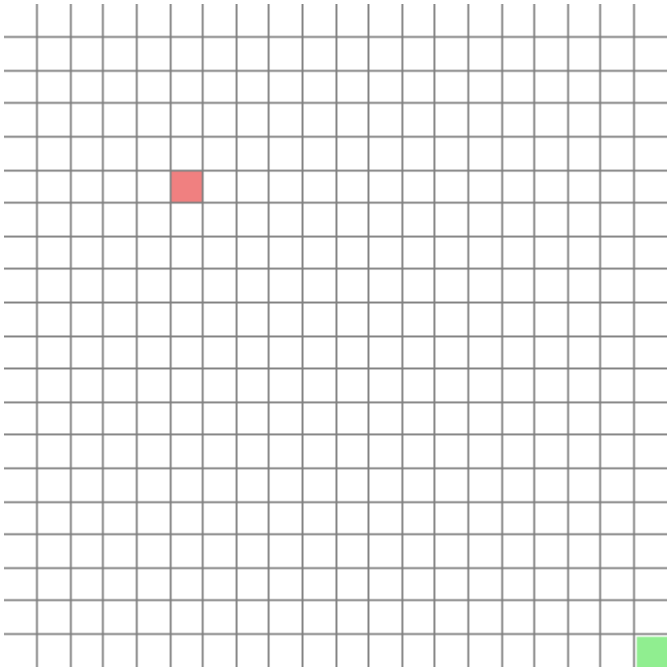
20, 20
20, 19
19, 18
18, 17
17, 16
16, 15
15, 14
14, 13
13, 12
12, 11
11, 10
10, 9
9, 8
8, 7
7, 6

Un exemple de liste : la liste de coordonnées (donc de nœuds, un nœud étant complètement et de manière unique défini par ses coordonnées) du 2). On peut classer ses nœuds en utilisant un paramètre défini pour chaque nœud, ici, par exemple on pourrait utiliser la distance d'un nœud au nœud de départ pour les classer suivant leur proximité du nœud de départ.

Les listes sont un problème à part entière pour les mathématiciens, les « algoriciens » qui se sont penchés sur différents problèmes, notamment d'optimisation pour le *tri* des listes. Il faut savoir que lorsqu'on veut classer les éléments d'une liste en fonction d'une valeur caractéristique de chaque élément (on pourrait classer une liste de nœuds d'après leur distance au point d'arrivée, par exemple), le tri peut prendre beaucoup de temps selon la méthode (un algorithme à part entière !) utilisée.
Ces algorithmes ne font pas l'objet de ce TPE, néanmoins, l'algorithme que nous utiliserons pour classer nos listes (à partir du 4.- dijkstra) sera l'algorithme « *quick sort* ».

Fonctionnement

Nous avons donc le problème classique :



Le Problème Classique (sans obstacles).

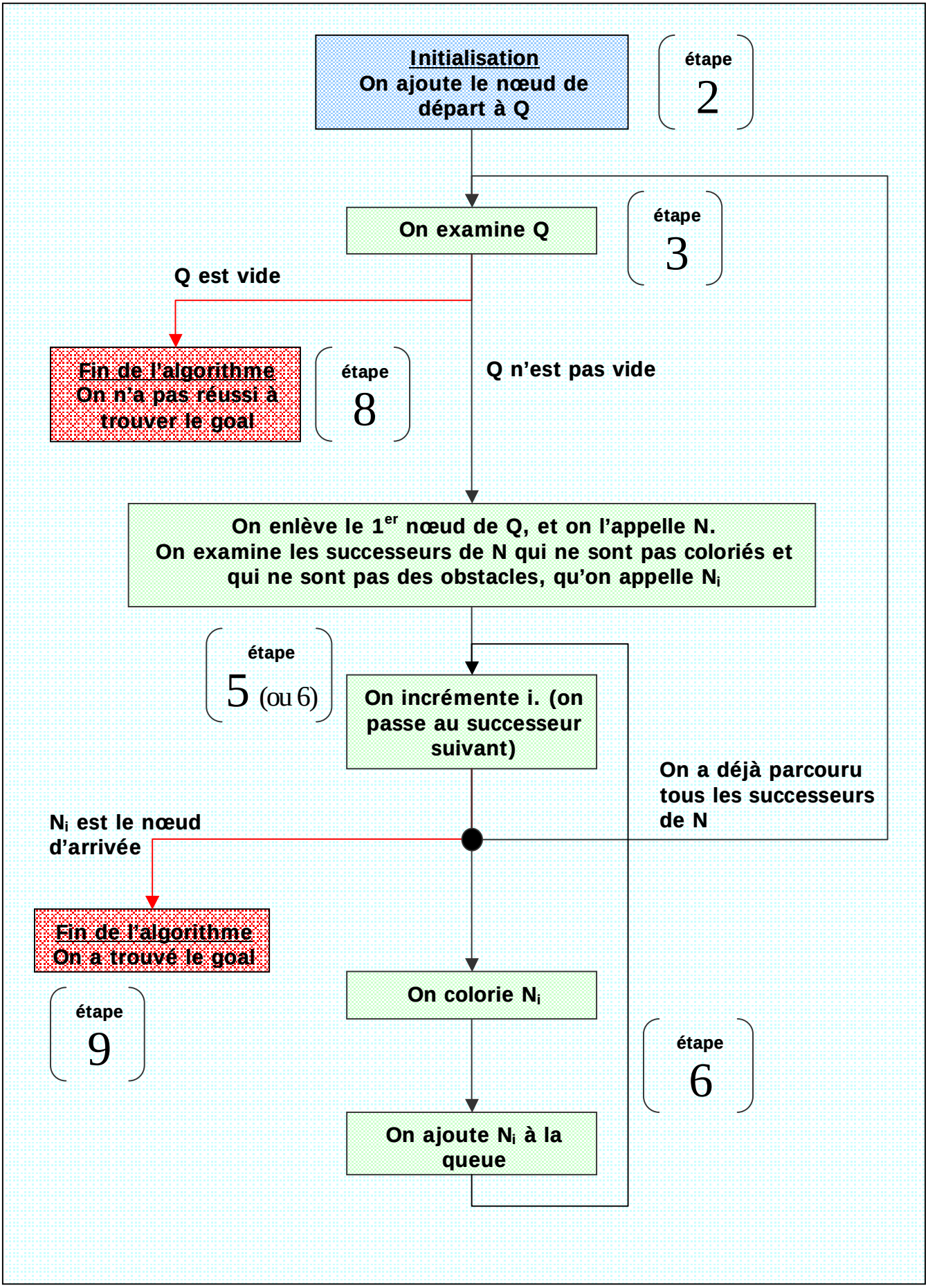
Le système breadth first search, contrairement au système DFS, examine d'abord les nœuds adjacents au nœud de départ, c'est à dire qu'il regarde d'abord si un des nœuds entourant un nœud défini n'est pas le nœud d'arrivée avant de passer à l'enfant de ce nœud (rappel : un nœud entourant un autre nœud est appelé un enfant ou un successeur de ce dernier). Ce système est mieux expliqué dans la procédure à suivre :

Initialisation

1. **Soit une queue Q. (pour l'instant, considérez qu'une queue est une liste comme décrite dans Les listes, donc à laquelle on peut ajouter ou enlever des noeuds).**
2. **Initialisation : on ajoute le nœud de départ à Q. Ce nœud est donc situé en position 1 dans la liste Q (si on ajoutait un autre nœud, il serait en position 2, etc. Si on retirait le nœud numéro 1, le nœud numéro 2 s'il existe prendrait sa place de deviendrait à son tour le nœud numéro 1).**
   Début de l'algorithme
3. **Si Q n'a aucun élément (la liste est vide), alors on passe à l'étape 8.**
4. **On retire le nœud situé en position 1 de Q, et on l'appelle N.**
5. **On détermine les successeurs de N, qu'on note N_1, N_2, etc.**
6. **Pour chaque successeur N_i, on regarde d'abord si il est colorié ou si c'est un obstacle, auquel cas on passe au successeur suivant (étape 6 avec i incrémenté).**
   Si il n'est ni colorié ni obstacle, alors regarde si c'est le nœud d'arrivée, si oui, on passe à l'étape 9. Si ce n'est pas le nœud d'arrivée, on ajoute ce successeur à la queue Q et on le colore sur le graphe (on colorie la case correspondante).
7. **On passe à l'étape 3 (retour au début).**
   Fins de l'algorithme

8. **Cette étape est atteinte quand la liste est vide : alors l'algorithme n'a pas réussi a trouver un chemin vers le goal en examinant tous les nœuds (le goal n'est pas accessible).**
9. **On a réussi à trouver un chemin vers le goal.**

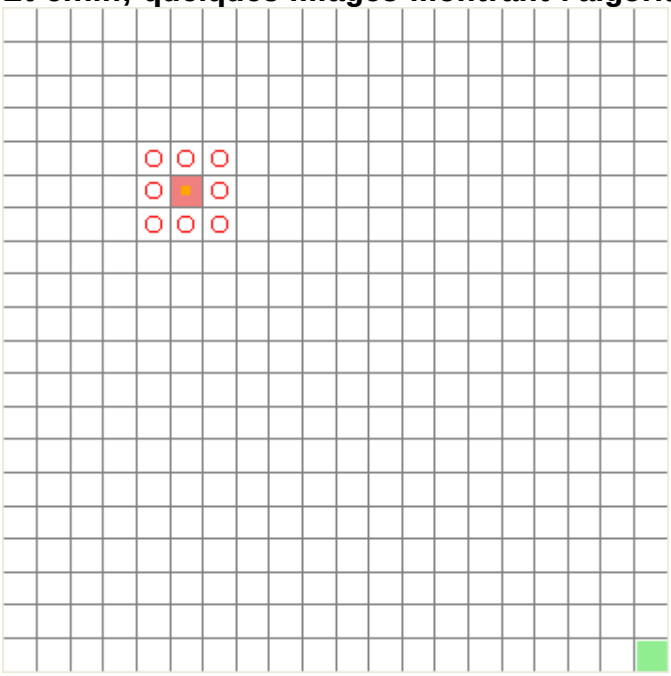
Cet algorithme est mieux décrit par le diagramme suivant que par du code :



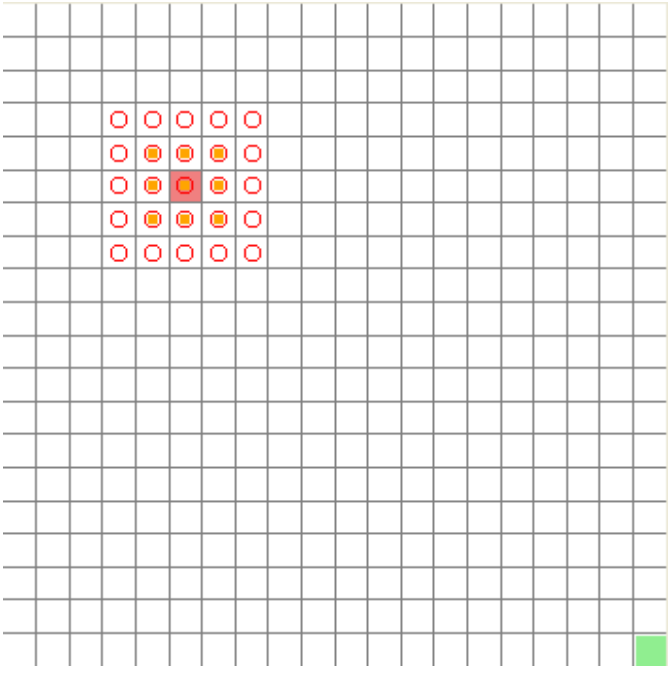
(il ne faut pas oublier pour autant que le code est nécessaire ! mais dans ce cas, le code est très largement semblable au code de l'algorithme DFS décrit avant,

car il réutilise les mêmes structures de données et les mêmes fonctions que les algorithmes de parcours de graphe).
Cet algorithme peut sembler récursif, à cause de la boucle de retour allant de l'étape 5 à l'étape 3, mais en fait il n'en est rien : la queue garde trace de tous les nœuds restants à traverser, évitant ainsi une fonction récursive ; il est donc très facile de mettre cette algorithme sous la forme unité de calcul.

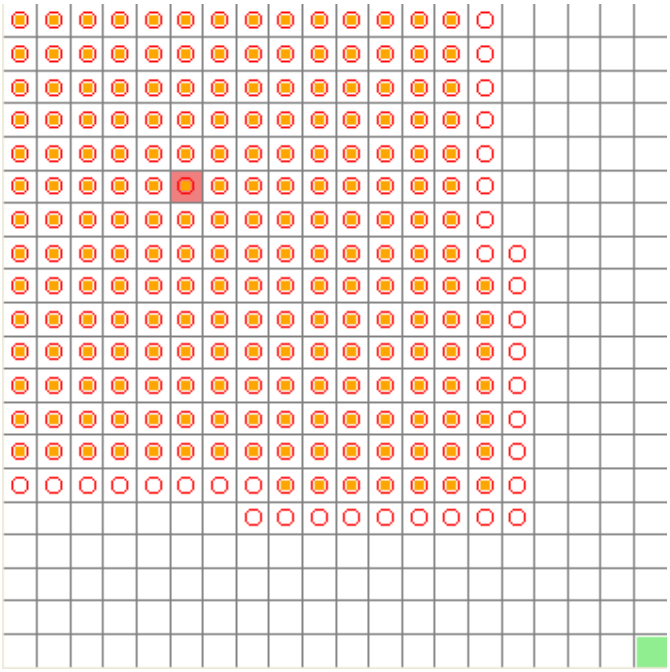
Et enfin, quelques images montrant l'algorithme en action :



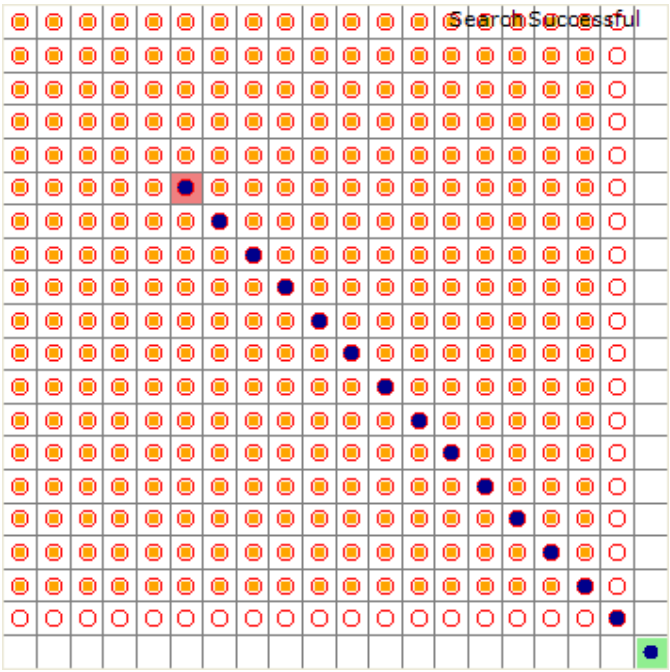
<u>Début d'une recherche</u>.
Les cercles rouges représentent les nœuds coloriés. On voit que ce sont les successeurs du nœud de départ.



L'algorithme examine les successeurs des successeurs. Les nœuds marqués d'un point orange sont les nœuds enlevés de la queue.



Une fois que tous les nœuds situés dans un coin de la carte ont été examinés, ils n'ont plus de successeurs et n'ont donc pas de descendance active (c'est-à-dire dans la liste Q), alors que les nœuds situés « en bas à droite » du nœud de départ ont toujours des enfants « productifs », c'est-à-dire qui ont eux-mêmes des enfants qui vont finir par atteindre le point d'arrivée.
Il faut s'imaginer que les nœuds situés en ce moment dans la queue sont tous les nœuds situés à la lisière des cases coloriées.



La recherche est terminée.
On voit que l'algorithme a quasiment examiné tous les nœuds de la carte. Les nœuds restants dans Q sont les nœuds non coloriés (blancs) qui ne seront jamais examinés car l'algorithme a trouvé le nœud d'arrivée avant de les examiner.
Les nœuds coloriés en bleu représentent le chemin conseillé par l'algorithme. (voir ci-dessous comment obtenir ce chemin).

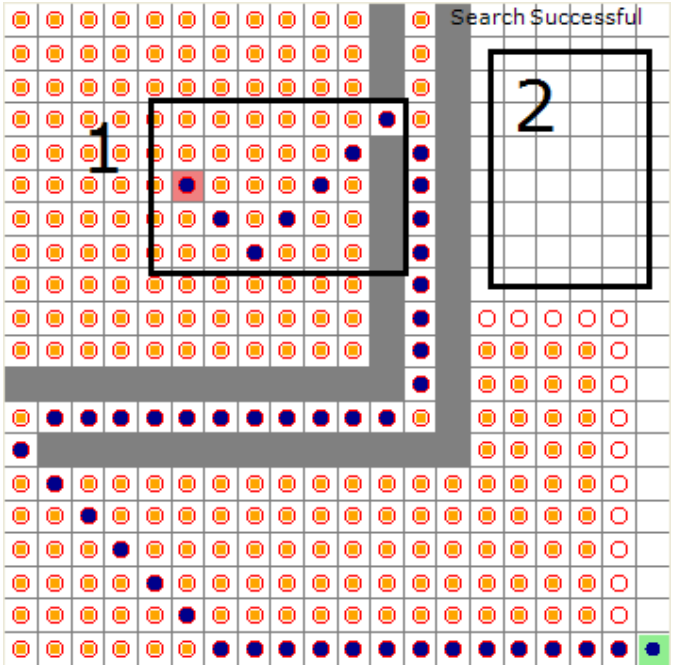
<u>Et le chemin à suivre ?</u>

On voit que l'algorithme se « répand » sur des cercles concentriques (qui ont pour centre le point de départ, plus exactement) jusqu'à ce que le nœud d'arrivée se trouve sur un de ces cercles. La question est : une fois qu'on a trouvé le nœud d'arrivée, comment remonter au nœud de départ ? La technique est simple : il faut ajouter une propriété qu'on appelle Parent à chaque nœud. Quand on examine le nœud N_i, on définit le parent de N_i comme étant N ; en effet, quoiqu'un nœud puisse avoir plusieurs successeurs ou enfants, il n'a qu'un seul parent. Chaque nœud traversé se trouve donc assigné d'un parent, qui est le nœud adjacent à N_i <u>le plus proche du point de départ</u> (à cause des cercles concentriques). Le nœud d'arrivée, quand il est traversé, se trouve alors affublé d'un parent, et c'est en remontant les parents des parents du nœud d'arrivée que l'on arrive jusqu'au nœud de départ, et qu'on a donc le chemin <u>le plus court</u>, qu'il suffit de remettre à l'endroit (on ne peut obtenir le chemin dans l'autre sens, car

un nœud a plusieurs successeurs) pour avoir fini la détermination intelligente de trajet avec l'algorithme breadth first search.

<u>Un cas intéressant</u>

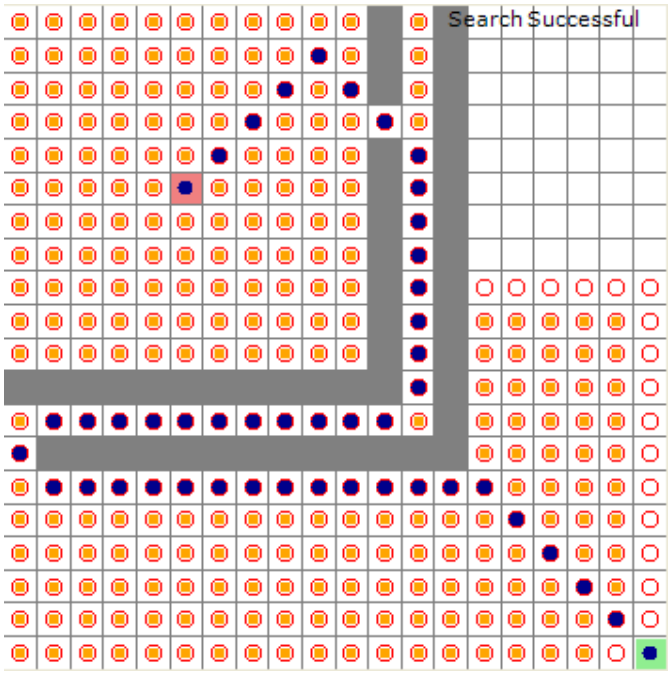
Examinons maintenant cette image : un chemin complètement résolu grâce à l'algorithme BFS. Cette fois ci, la carte comporte des obstacles, ce qui fait apparaître quelques faits intéressants sur la recherche :



Tout d'abord, le point 2. On voit qu'au point 2, les cases sont restées vierges, c'est-à-dire que les successeurs n'ont pas été parcourus : cela est du au fait que l'algorithme s'arrête une fois qu'il a trouvé le nœud d'arrivée. Sur les parties avec beaucoup d'obstacles, les successeurs sont plus rares, ce qui permet à l'algorithme d'avancer plus vite sur ces parties (car moins de successeurs à examiner) alors que les parties avec un terrain vierge d'obstacles seront « scannées » intégralement par l'algorithme.

Au point 1, on voit un trajet qui semble bizarre et qui ne semble pas être le plus court : cela est du à la fonction <u>GetSuccessors</u> qu'on a vu au 2). Cet algorithme ne peut considérer les cases pondérées ; il est donc impossible de lui faire « comprendre » qu'aller sur une case située en diagonale d'un nœud parent est plus long qu'aller sur une case située horizontalement ou verticalement de son fils. Ce chemin est donc équivalent en nombre de cases que le chemin optimal.

Pour connaître la réelle influence de GetSuccessors, j'ai changé l'ordre des nœuds retournés par cette fonction. Le chemin obtenu est sur la page suivante à



Deuxième solution.
Devinez l'ordre des successeurs !

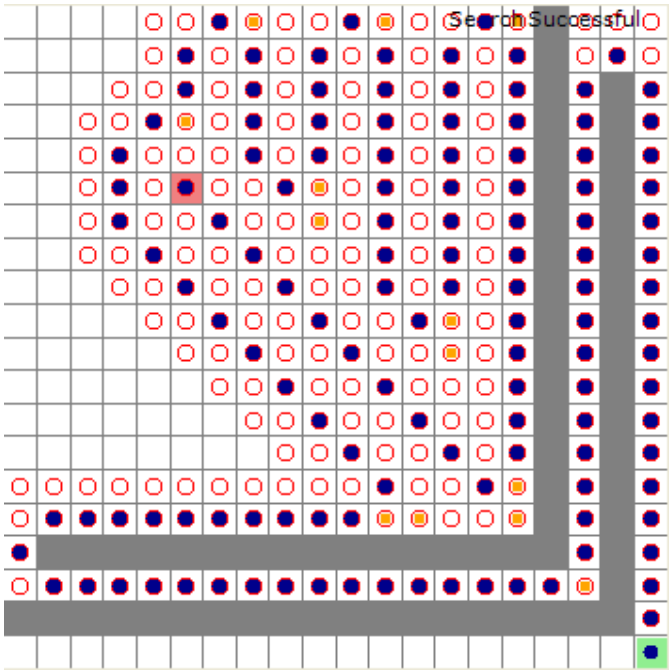
On voit que le chemin a changé, mais reste <u>équivalent en nombre de nœuds</u> traversés.

On sait que l'algorithme examine chaque nœud en position 1 dans la queue d'abord et ajoute donc ses successeurs en premier à la liste ; c'est donc les siens qui seront examinés en premier plus tard, et ainsi de suite. On peut ainsi deviner que le premier successeur dans la deuxième solution devait être le successeur en haut [à droite] du parent, contrairement à la première où ce devait être celui en bas [à droite].

<u>Pourquoi une queue ?</u>

Une queue est une liste où on retire le premier élément d'abord, celui qui est « le plus bas » dans la liste. Mais quelle est donc l'utilité de cette contrainte dans cet algorithme ? Pourquoi ne pas enlever un nœud situé au milieu, ou à la fin de cette liste ?

Pour le montrer, voici le chemin retourné par une variante du BFS qui examinerait les derniers nœuds de la queue :



Un chemin incohérent.
Je parie que ce n'est pas l'algorithme utilisé par Mars Pathfinder !

On voit que le chemin n'est pas optimal (digne des pires cauchemars de DFS !). Ce phénomène est dû au fait qu'on retire le dernier élément dans la liste. Mais que représente ce dernier élément ? c'est en fait l'élément ajouté le plus tard dans la liste, donc le nœud le plus éloigné du point de départ (en comptant l'éloignement en nœuds) : chaque successeur ajoutant son nœud, c'est seulement le nœud du dernier successeur qui aura ajouté des nœuds qui sera examiné et qui ajoutera ses nœuds, et ainsi de suite. Résultat : le chemin obtenu est celui qui est éloigné du plus de nœuds possible du chemin optimal. Peu adapté quand on pense que le but de cet algorithme est de fournir le chemin le plus court ! En résumé, une queue est indispensable au fonctionnement de cet algorithme.

Unité de calcul

Cet algorithme est déjà quasiment sous la forme unité de calcul (je rappelle que l'unité de calcul est la fonction principale d'un algorithme non récursif) : en effet, la boucle principale (étapes 3 à 6) se sert du premier nœud tiré de la queue au lieu de se servir d'un nœud défini par la répétition précédente (voir DFS pour un exemple d'algorithme récursif). On peut dire que, s'il existait une fonction bfs (par analogie avec la fonction dfs qui elle est récursive), elle ne prendrait aucun argument (contrairement à la fonction dfs), c'est-à-dire qu'elle est indépendante de l'appel précédent de bfs – la queue servant de seule liaison entre les différents appels de cette fonction.

Résumé

Pour trouver son chemin grâce à l'algorithme BFS :

- **On ajoute le nœud de départ à une queue Q**
- **Tant qu'il y a des nœuds dans Q :**
  - **On enlève le premier nœud de Q, qu'on appelle N**
  - **On détermine et on colorie les successeurs de N**
  - **Si l'un d'eux est le goal, on a trouvé le chemin. Sinon, on continue jusqu'à ce qu'il n'y ait plus de nœuds (condition d'échec).**

Problèmes connus

Cet algorithme ne souffre pas des mêmes problèmes que les précédents algorithmes. On peut néanmoins signaler la difficulté qu'on peut avoir à retrouver le chemin pris par l'algorithme jusqu'au goal (il faut créer une nouvelle propriété « parent » pour chaque nœud).

Avantages

Cet algorithme trouve toujours le plus court chemin, car il cherche en faisant des cercles concentriques dont l'un finit par « toucher » le goal. De plus, l'algorithme cherche dans toutes les directions à la fois.

Inconvénients

Le traçage du chemin (c'est-à-dire la détermination du chemin pris par l'algorithme une fois qu'il a trouvé le nœud d'arrivée) oblige à une augmentation de mémoire proportionnelle au nombre de nœuds traversés (la propriété « parent » ajoutée à chaque nœud) et oblige à des calculs (peu de calculs, mais présents quand même) pour donner un chemin utilisable par le robot.
Cet algorithme cherche beaucoup de nœuds pour arriver au nœud final (tout dépend de l'éloignement au nœud final, mais les cercles concentriques font

apparaître de plus en plus de nœuds aux fur et à mesure que l'on se rapproche du nœud d'arrivée), ceux –ci prenant une grande place en mémoire (dans la queue), ce qui peut être handicapant.
De plus, le fait qu'il ne tienne pas compte des poids entraîne quelques incohérences sur le trajet pris (car les éloignements sont considérés en nombre de cases et non en distances euclidiennes) et empêche le robot de considérer les élévations et autres altérations du terrain.

L'algorithme BFS est le premier algorithme que nous voyons et qui donne à coup sûr un des chemins les plus courts, mais il reste quelques inconvénients ennuyeux que d'autres algorithmes peuvent corriger. Entre autres, l'utilisation mémoire est vraiment un facteur handicapant, alors que les calculs demandés au microprocesseur du robot se limitent à la détermination des nœuds fils (GetSuccessors). C'est donc un algorithme qui pourrait être intégré dans un vrai robot. S'il n'y avait pas d'autres algorithmes plus performants…

4.Dijkstra

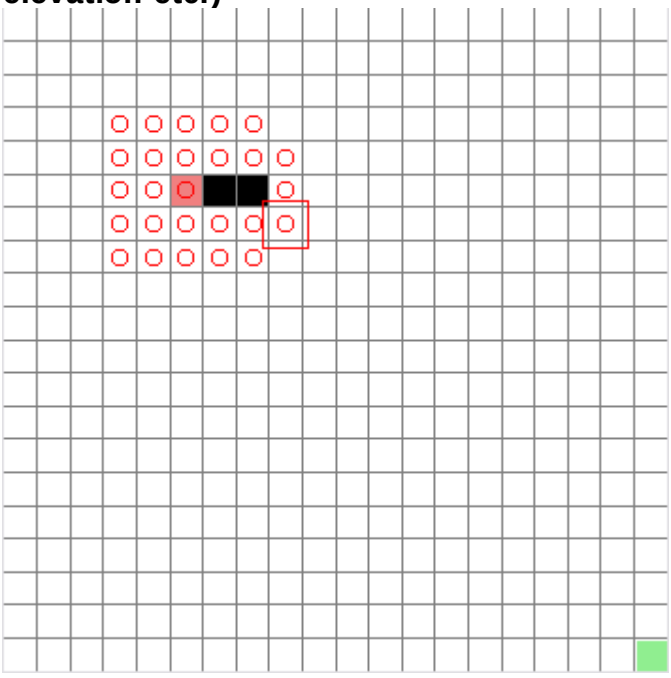
L'algorithme de dijkstra n'est pas, contrairement au BFS et au DFS, un algorithme de parcours de graphes, mais la technique qu'il explique peut être appliquée au principe des cases et des noeuds. Il est très similaire dans son comportement au BFS, dans le sens où quand il va explorer la carte (traverser les nœuds), il va considérer d'abord les voisins d'un nœud (voir l'explication au chapitre 3) BFS). Je vais expliquer cet algorithme pour faire la transition entre les techniques utilisées par les algorithmes de parcours de graphe et les spécificités de l'algorithme A*. C'est aussi là partir de l'algorithme de Dijkstra que le poids des nœuds sera enfin introduit et pris en compte dans les calculs.

Cet algorithme réutilise *le principe* de la coloration introduit à partir du DFS pour éviter de repasser deux fois sur une même case : la variante utilisée par dijkstra se sert d'une liste de nœuds, nommée communément *Closed* pour éviter de repasser sur des nœuds déjà visités (l'utilisation effective d'une liste pour cet usage sera explicitée plus tard).

Il utilise aussi la liste Open, qui est l'équivalent de la queue Q du BFS à quelques améliorations près (je rappelle que le but de cette liste était de stocker les positions des nœuds *à visiter*).

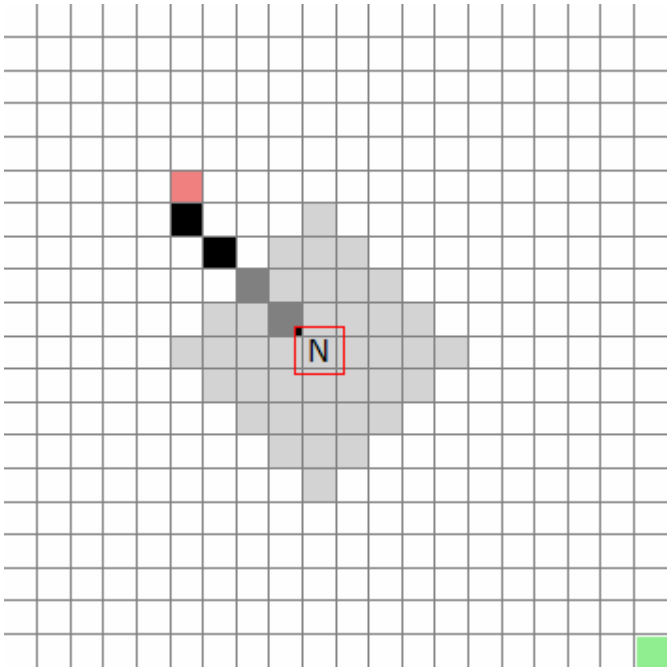
<u>L'intervention des poids</u>

On associe à chaque case (nœud) un poids qui correspond à une réalité physique (rappel : ces caractéristiques physiques peuvent être le type de terrain, une élévation etc.)



Une carte quelconque. Tous les poids de toutes les cases sont mis à 1 « par défaut ».

Examinons un exemple de début de recherche avec cet algorithme, et regardons plus particulièrement le nœud entouré en rouge. Le chemin provisoire déterminé par l'algorithme quand il arrive à ce nœud est indiqué en noir. On associe à ce nœud une valeur g, telle que g soit « la somme des poids des cases parcourues pour arriver à ce nœud ». Ici, les « cases parcourues pour arriver à ce nœud » sont les cases coloriées en noir, la case de départ (qui est comptée dans le calcul, bien qu'on puisse l'oublier) et notre nœud (entouré de rouge). Chaque case a un poids de 1 sur cette carte. Donc la valeur g de notre nœud sera 1+1+1+1 = 4, car pour arriver à ce nœud il a faut d'abord passer sur quatre cases pondérées de poids respectifs 1.



Une autre carte quelconque. Toutes les cases blanches ont un poids de 1, les cases grisées un poids de 2. Les cases noires sont des cases déjà traversées de poids 1, et les cases gris foncé des cases déjà traversées de poids 2.

Un autre exemple : on calcule la valeur g du nœud N ici. g = 1 (nœud de départ) + 1 + 1 + 2 + 2 + 2 (car N est sur une case de poids 2) = 9. A chaque nœud on peut associer une valeur g unique (car chaque nœud n'est examiné qu'une fois dans les algorithmes qui l'utilisent). On peut donc noter g(N) = 9 pour ce nœud.

Fonctionnement

Le fonctionnement de cet algorithme est très similaire au BFS, au sens où on regarde d'abord tous les successeurs d'un nœud avant un successeur particulier, et on garde la liste Q (qui s'appelle maintenant « Open »).

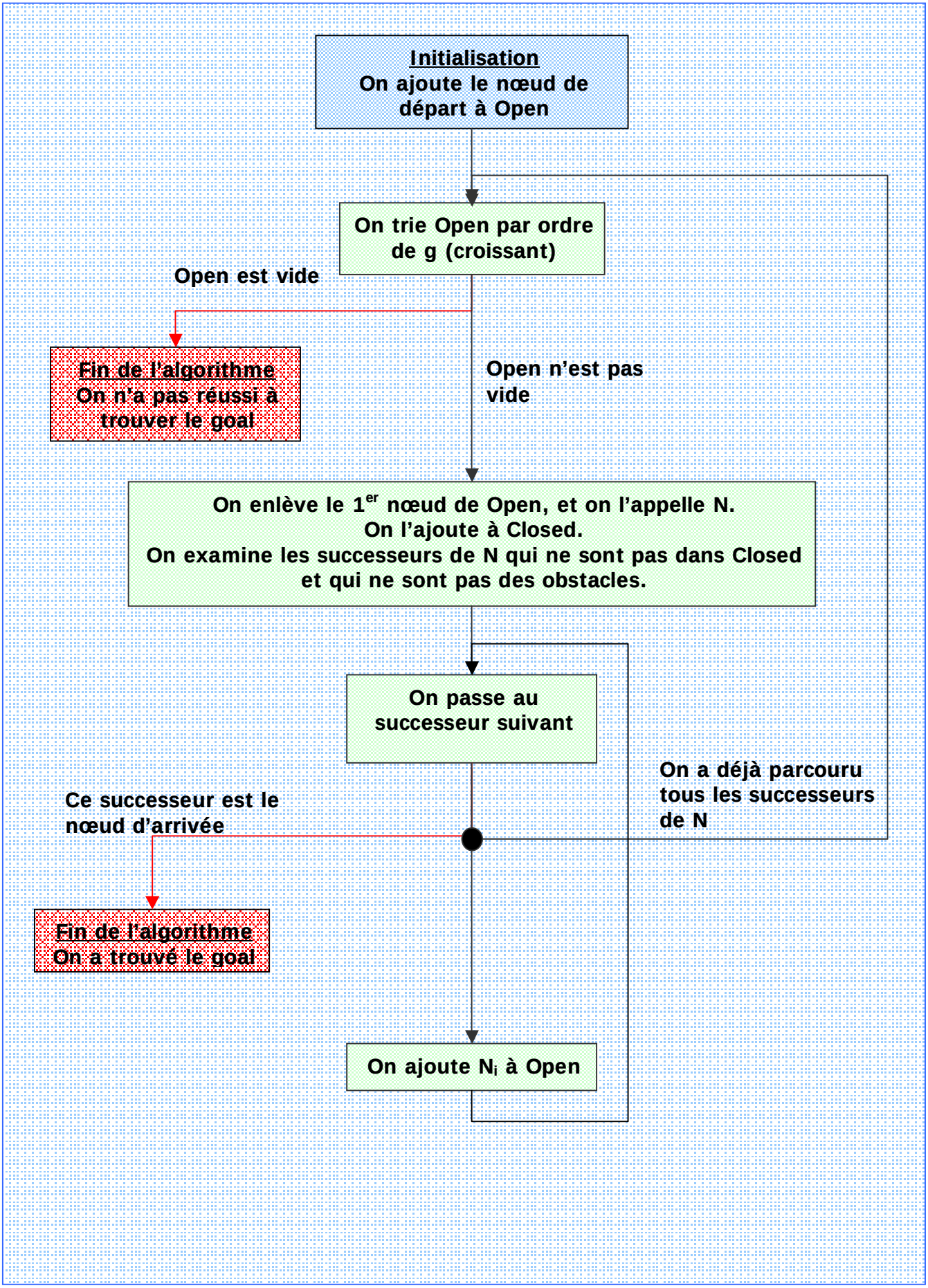
Initialisation

1. **On ajoute le nœud de départ à Open.**

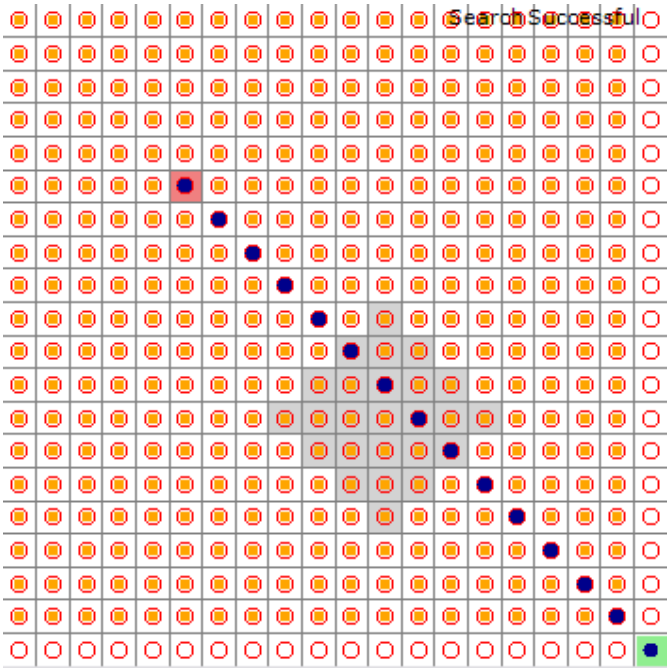
Début de l'algorithme

2. **Si Open est vide, alors l'algorithme a fini et on n'a pas trouvé de solution.**
3. **On trie la liste Open par ordre croissant de g. Ainsi, le nœud avec le g le plus faible sera enlevé (examiné) en premier.**
4. **On retire le premier nœud de Open, on l'appelle N.**
5. **On calcule le g de chaque successeur de manière récursive : g(successeur) = g(N) + poids(successeur). On ajoute chaque successeur à Open si il n'est pas déjà dans Closed (cette technique est équivalent au coloriage du DFS et BFS).**
   Si ce successeur est le nœud d'arrivée, alors on a trouvé le chemin. Sinon on passe au successeur suivant.
6. **On ajoute N à la liste Closed pour éviter de repasser dessus.**
7. **On passe à l'étape 2.**

On voit que le principe de fonctionnement est très semblable à celui de l'algorithme BFS, avec une exception notable : maintenant, les nœuds mis dans la liste Open sont triés par ordre croissant de g : ainsi, les nœuds avec les g les plus faibles, donc ceux qui doivent être privilégiés comme points de passage du robot sont placés en première position et sont donc examinés en premier. Mais Dijkstra examine quand même tous les nœuds, ce qui résulte en la création de cercles concentriques comme dans BFS.
Ci-dessous gît le diagramme régissant les actions de cet algorithme :

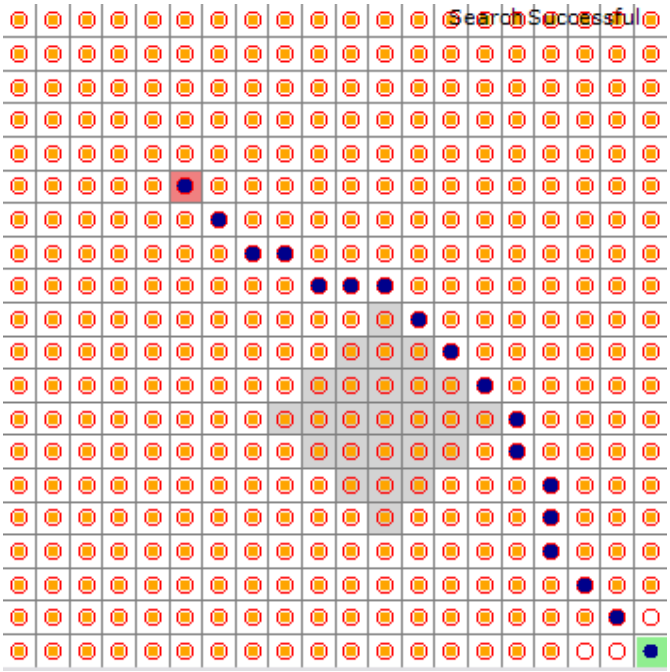

Initialisation
On ajoute le nœud de départ à Open
On trie Open par ordre de g (croissant)
Open est vide
Fin de l'algorithme
On n'a pas réussi à trouver le goal
Open n'est pas vide
On enlève le 1er nœud de Open, et on l'appelle N.
On l'ajoute à Closed.
On examine les successeurs de N qui ne sont pas dans Closed et qui ne sont pas des obstacles.
On passe au successeur suivant
On a déjà parcouru tous les successeurs de N
Ce successeur est le nœud d'arrivée
Fin de l'algorithme
On a trouvé le goal
On ajoute Ni à Open

Voyons pour commencer un exemple de recherche d'un chemin avec l'algorithme BFS :



Un chemin trouvé grâce à l'algorithme BFS. Les cases grisées sont les cases « à éviter si possible » de poids 2. Toutes les autres cases ont comme poids 1.

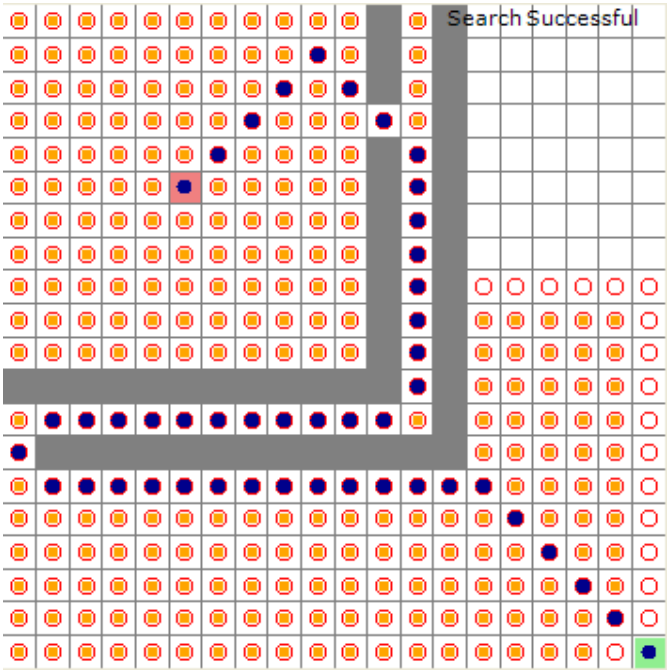
On voit bien ici que l'algorithme BFS n'a pas tenu compte du poids des cases et est passé sur celles-ci comme si de rien n'était. Gardons la même carte et faisons la même recherche grâce à dijkstra :



Un chemin trouvé grâce à l'algorithme de Dijkstra.

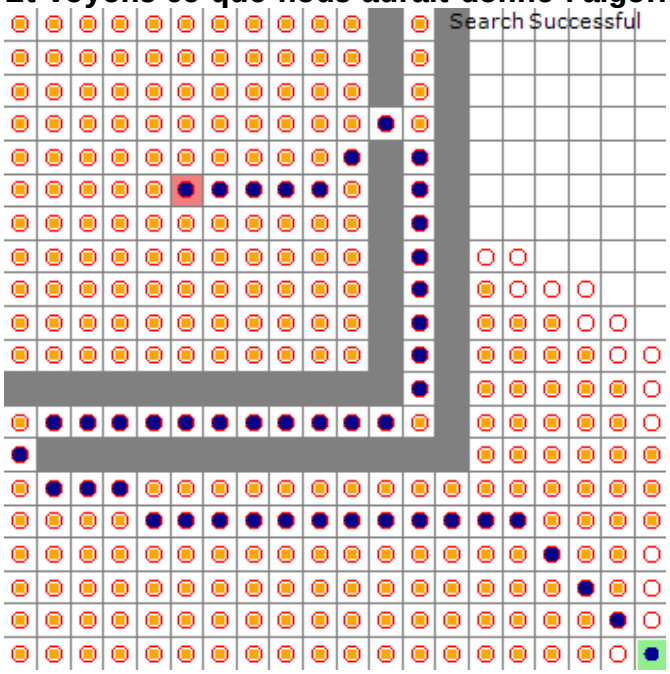
On voit bien qu'avec Dijkstra, l'algorithme évite naturellement les cases pondérées et trouve le plus court chemin vers son but.

On peut aussi reprendre un des problèmes du BFS, qui était celui-ci :



Une des recherches faites avec l'algorithme BFS. Notez les problèmes du chemin trouvé.

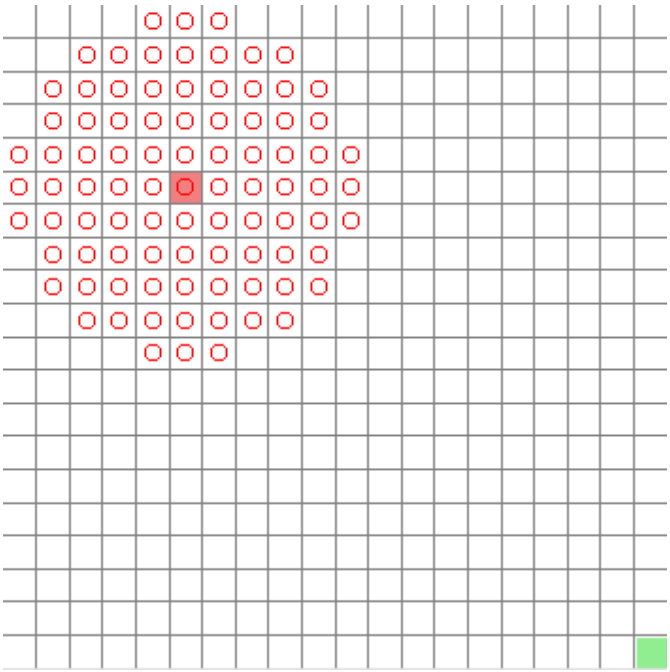
Et voyons ce que nous aurait donné l'algorithme de Dijkstra dans ce cas :



La même recherche, vue par Dijkstra.

On voit que Dijkstra nous donne le plus court chemin ; en effet, le fait de donner un poids supérieur à 1 aux cases situées en diagonale limite les déplacements diagonaux de l'algorithme qui leur préférera, si possible, un déplacement horizontal ou vertical. C'est donc cette petite astuce qui, grâce à la prise en compte des poids, permet d'optimiser le chemin.

Cet algorithme étant très semblable à BFS dans sa propagation sur la carte, je me contenterait de mettre un exemple montrant l'algorithme en action :



Sur cette image, on voit les cercles concentriques formés par Dijkstra autour du nœud de départ dans sa quête pour le nœud d'arrivée.

On peut remarquer accessoirement que les cercles concentriques qui sont formés « ressemblent » plus à des cercles qu'avec l'algorithme BFS (du fait de sa préférence pour les cases horizontales et verticales plutôt que diagonales).

L'algorithme garde donc en mémoire les poids des cases visitées, qui sont placées dans la queue. La queue sert à examiner en priorité les nœuds ayant les poids les plus faibles. Le principe et les chemins de cet algorithme sont très similaires à l'algorithme BFS.

L'algorithme de Dijkstra est facile à mettre sous la forme « unité de calcul » grâce à l'utilisation de la liste (cf. BFS), qui permet donc de diviser le temps demandé pour la détermination du trajet.

<u>Résumé</u>

Pour trouver un chemin avec l'algorithme de Dijkstra :

- **On ajoute le nœud de départ à une queue Open (qui contient les nœuds à visiter)**
- **Tant qu'il y a des nœuds dans Open :**
  - **On trie la liste open par ordre de g croissant**
  - **On enlève le premier nœud de Open, qu'on appelle N**
  - **On détermine les successeurs de N.**

    On calcule la valeur de g pour chaque successeur de N.

    On ajoute ce successeur de N à Closed.
  - **Si l'un d'eux est le goal, on a trouvé le chemin. Sinon, on continue jusqu'à ce qu'il n'y ait plus de nœuds (échec).**

Problèmes connus

Cet algorithme trouve toujours son chemin s'il existe : on ne note donc pas de problèmes pour cet algorithme, bien qu'on ait les mêmes difficultés à « remonter » dans le programme à partir du nœud d'arrivée pour déterminer exactement le chemin (cf. les « Problèmes connus » du BFS).

Avantages

Grâce à son déploiement en cercles concentriques, Dijkstra trouve toujours le chemin le plus court. A la différence du BFS néanmoins, l'utilisation du poids des cases permet de tenir compte des paramètres du terrain (hauteur, …) ainsi que de pondérer les cases situées en diagonale d'un nœud, permettant ainsi d'avoir le premier trajet vraiment optimisé que nous voyons.

Inconvénients

L'algorithme examine beaucoup de nœuds (même remarque que pour le BFS) à cause de l'expansion en cercle, non dirigée vers le nœud d'arrivée, qui conduit à « l'exploration » de presque tout le terrain avant de trouver le nœud d'arrivée. On peut néanmoins remarquer que la fonction g, principe de l'algorithme, permet d'éviter particulièrement les cases pondérées (le nombre de cases examinées est donc inférieur dans la plupart des cas à celui retourné par l'algorithme BFS).

Il y a aussi un risque important de fautes d'orthographe au niveau de son nom, ce qui explique que l'on peut lui préférer d'autres algorithmes, moins complexes à prononcer.

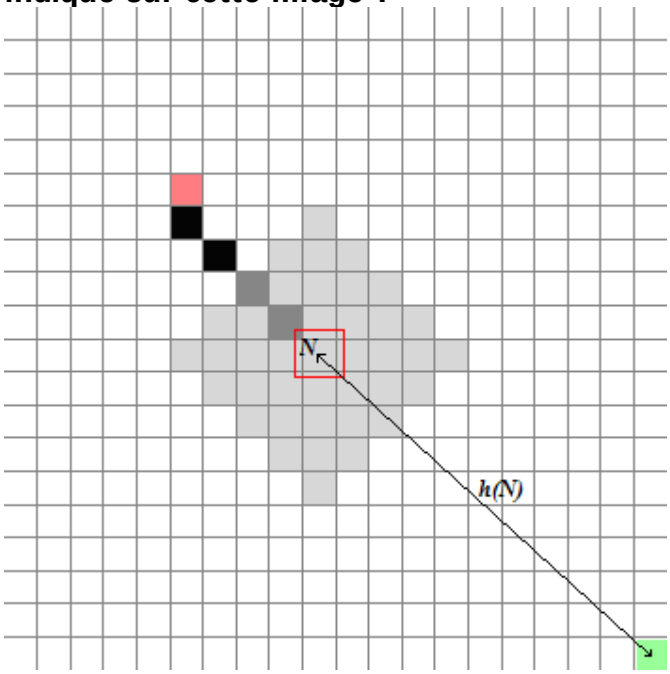
Cet algorithme est beaucoup plus gourmand au niveau des calculs que l'algorithme BFS, surtout à cause du tri des listes intensif à chaque nœud. Il demande en plus beaucoup de mémoire, car il faut garder trace des nœuds et de leur valeur g. Toutes ces contraintes font que l'algorithme est plutôt utilisé dans des systèmes grand public ou qui nécessitent un chemin optimisé (détermination de trajet sur des cartes routières, ordinateur de bord), mais il n'est pas destiné à la miniaturisation.

5.A* - « A star »

Cet algorithme est l'aboutissement des recherches sur l'algorithme de Dijkstra. Il réutilise les techniques du BFS et de Dijkstra pour les combiner et en faire l'algorithme le plus convergent de tous ceux que nous étudierons (c'est aussi le plus utilisé).
On garde la technique des deux listes, Open et Closed de dijkstra (rappel : Open contient les nœuds *à visiter*, Closed les nœuds *déjà parcourus*).
L'algorithme A* est le plus utilisé dans le monde actuel, car c'est le plus efficace. Il n'a pas été conçu spécifiquement pour être compatible avec les graphes. Néanmoins, et pour ne pas surcharger ce TPE de détails, nous considérerons uniquement la variante de cet algorithme qui lui permet de travailler avec des nœuds.

La fonction h : une distance

On associe à chaque nœud la distance de ce nœud à la case d'arrivée, comme indiqué sur cette image :



**Le segment qu'on voit en noir sur cette image est h(N).
(les cases noires et grisées n'ont aucune importance)**

On voit sur cette image la distance h du nœud N au nœud d'arrivée. Cette distance est unique pour chaque nœud/case de la carte, on dit donc qu'on associe à chaque nœud N la fonction h(N) qui représente cette distance, en unité de longueur (ici, h(N) vaudrait approximativement 14 en distance euclidienne).

La fonction h : un heuristique

Qu'est-ce qu'un heuristique ? La définition algorithme d'un heuristique est un moyen de guider un algorithme plus grand, sans pour autant pouvoir remplacer l'algorithme à lui tout seul (définition volontairement simplifiée et adaptée à notre cas). Dans ce cas, notre heuristique est la fonction h, utilisée convenablement. Partons de l'algorithme de Dijkstra : cet algorithme choisissait en priorité les nœuds situés le plus près du nœud d'arrivée grâce à la fonction g. Pourquoi ne pas intégrer notre fonction h dans le choix des nœuds ? on pourrait en effet préférer de choisir les nœuds qui ont un h le plus faible, c'est-à-dire ceux qui sont les plus proches du but. Dans ce cas, la fonction h serait un heuristique, au sens où elle guiderait le sens de propagation de l'algorithme, au même titre que la fonction g. On pourrait donc imaginer une troisième fonction f qui associerait à chaque nœud une valeur f qui tienne compte de g et de h de ce

nœud, ce qui permettrait de classer les nœuds dans la liste *Open* par ordre croissant (ou décroissant) de f, pour avoir les nœuds les plus adaptés au chemin au début de la liste pour un examen plus rapide.

La fonction f

Nous avons donc une fonction f, qu'on a définie telle que :

$f = g + h$

On voit que avec cette fonction, plus un nœud est près du nœud de départ, plus son g est faible ; plus un nœud est près du nœud d'arrivée, plus son h est faible. Les nœuds dont les g et h sont les plus faibles font donc partie du chemin le plus court. Les nœuds dans la liste Open seront donc triés par ordre croissant de f, pour avoir les nœuds les plus susceptibles de fournir le chemin le plus court en haut de ces deux listes (ce qui permet de transformer ces listes en queues, c'est-à-dire qu'on retirera toujours l'élément le plus haut dans la liste).

Fonctionnement

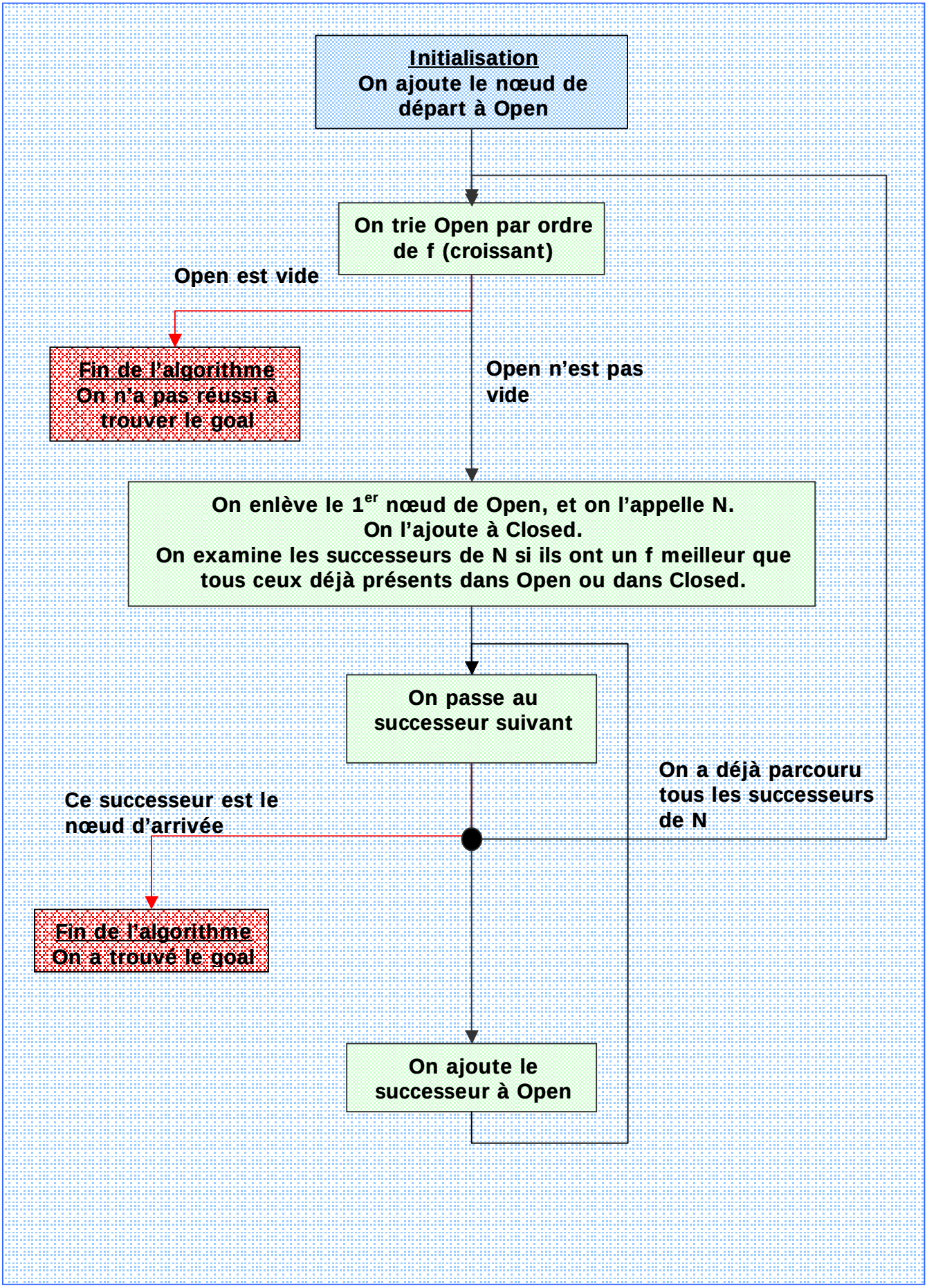
Initialisation

1. **On ajoute le nœud de départ à Open.**

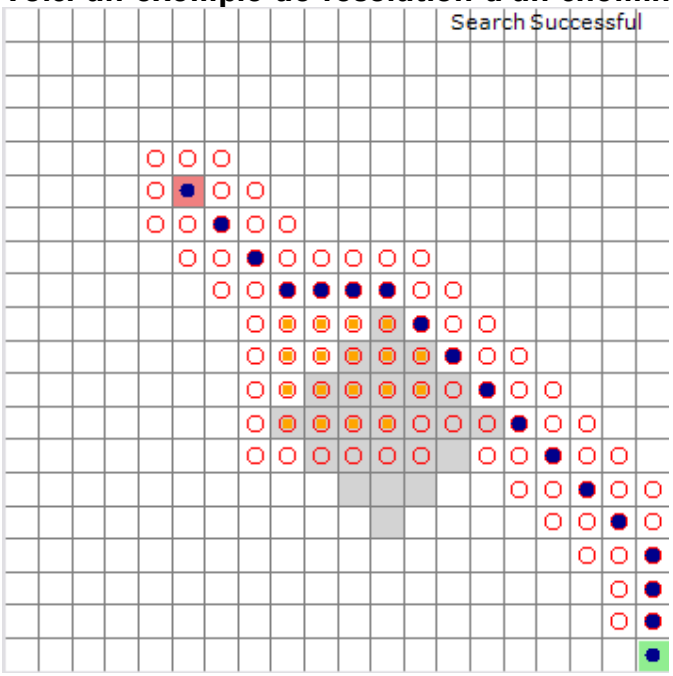
Début de l'algorithme

2. **Si Open est vide, alors l'algorithme a fini et on n'a pas trouvé de solution.**
3. **On trie la liste Open par ordre croissant de f (le nœud avec le f le plus faible sera examiné en premier).**
4. **On retire le premier nœud de Open, on l'appelle N.**
5. **On calcule le f de chaque successeur :**
   f(successeur) = g(N) + poids(successeur) + h(successeur)
6. **Si ce successeur est le nœud d'arrivée, alors on a trouvé le chemin.**
7. **Si il existe dans Closed ou dans Open un nœud X qui aurait déjà un f *inférieur* au f de ce nœud N (f(X) < f(N)), alors on passe au successeur suivant, car l'existence d'un tel nœud X prouve qu'il existe déjà de meilleures solutions qui ont déjà été examinées (ce nœud X existe dans Closed) ou qui vont l'être (ce nœud X existe dans Open), il n'est donc pas nécessaire d'examiner N.**
8. **On ajoute N à la liste Closed pour éviter de repasser dessus.**
9. **On passe à l'étape 2.**

On voit le mode de fonctionnement dans le diagramme suivant :

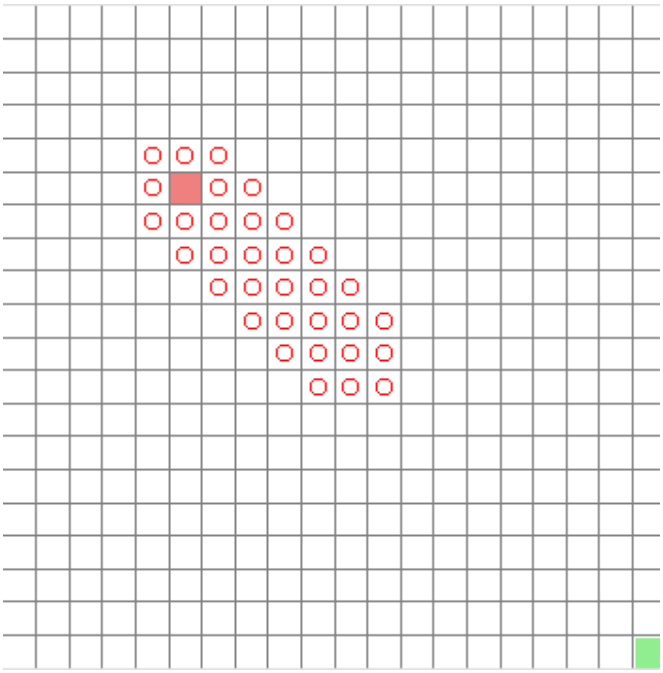

Initialisation
On ajoute le nœud de départ à Open
On trie Open par ordre de f (croissant)
Open est vide
Fin de l'algorithme
On n'a pas réussi à trouver le goal
Open n'est pas vide
On enlève le 1er nœud de Open, et on l'appelle N.
On l'ajoute à Closed.
On examine les successeurs de N si ils ont un f meilleur que tous ceux déjà présents dans Open ou dans Closed.
On passe au successeur suivant
On a déjà parcouru tous les successeurs de N
Ce successeur est le nœud d'arrivée
Fin de l'algorithme
On a trouvé le goal
On ajoute le successeur à Open

Voici un exemple de résolution d'un chemin avec l'algorithme A* :



Un chemin trouvé grâce à l'algorithme A*. Les cases grisées sont les cases « à éviter si possible » de poids 2. Toutes les autres cases ont comme poids 1. Les nœuds marqués par un cercle rouge sont passés directement dans Open alors que ceux remplis d'orange ont été d'abord par Closed (ils ont vraiment été examinés)

On voit clairement ici que, par rapport aux algorithmes de Dijkstra et BFS, l'algorithme A* ne parcourt pas toute la carte ; en effet les nœuds qui s'éloignent trop du « chemin optimal » sont rejetés (cercles rouges). Grâce à l'inclusion de la fonction g, l'algorithme préfère les cases non pondérées et les trajets en diagonale, comme l'algorithme de Dijkstra. Grâce à notre heuristique, l'algorithme n'a plus besoin de parcourir toute la carte pour trouver le nœud d'arrivée.



« le problème classique » : un début de recherche. Les cercles rouges correspondent aux nœuds situés dans Open (en cours d'examen).

On voit que sous l'influence de l'heuristique, l'algorithme forme une « ligne » dans sa recherche du nœud d'arrivée. La ligne droite reste le plus court chemin d'un nœud à un autre, et c'est la forme de propagation que A* adopte.

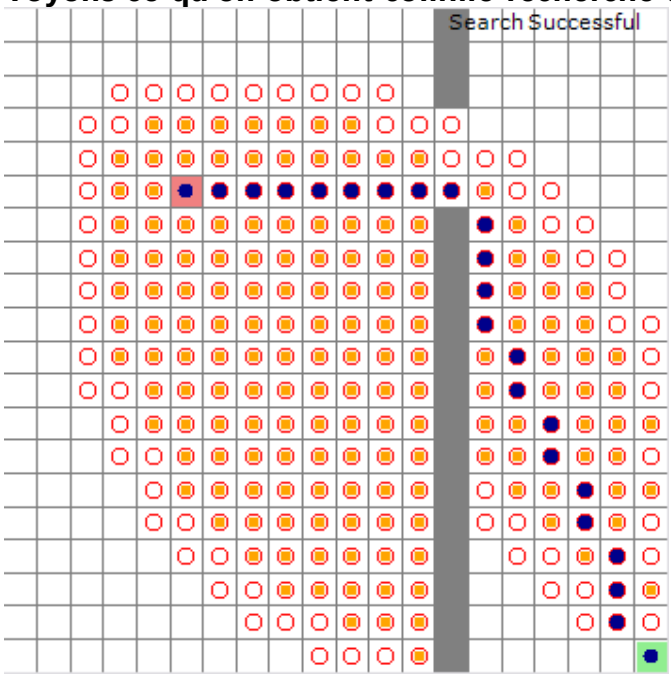
Je ne donnerai pas d'autres exemples de résolution de problèmes car cet algorithme donne les mêmes résultats que l'algorithme de Dijkstra (tout en examinant moins de nœuds).

L'influence des heuristiques

Supposons que nous ayons une case N dont le point situé au centre de cette case aurait comme coordonnées (X, Y). Notre case d'arrivée aurait comme coordonnées de son point central (X_A, Y_A). On veut calculer h(N), qui est la distance entre N et le point d'arrivée. Avec la formule de la distance euclidienne (théorème de Pythagore), on a :

$$h(N) = \sqrt{(X - X_A)^2 + (Y - Y_A)^2}$$

Voyons ce qu'on obtient comme recherche avec cet heuristique :



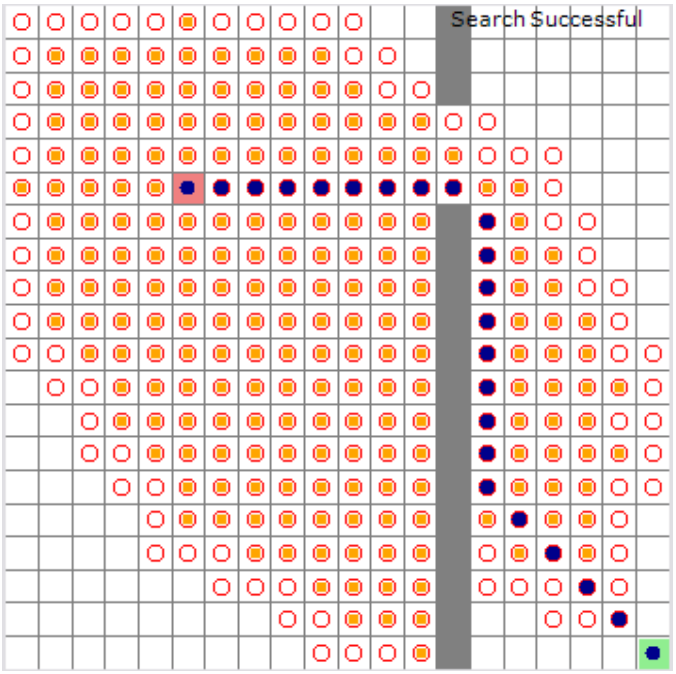
Heuristique : distance euclidienne

Visiblement, ce n'est pas un heuristique très efficace, car il dirige suffisamment mal l'algorithme A* pour entraîner l'exploration de dizaines de cases inutiles.

Mais il existe aussi d'autres heuristiques, c'est-à-dire ici d'autres moyens de calculer cette distance. L'une des plus connues est la distance maximum :

$$h(N) = \max(|X - X_A|, |Y - Y_A|)$$

Cet heuristique prend comme distance finale la différence de distance maximum sur X et sur Y, ce qui donne un chemin comme celui – ci (voir page suivante).



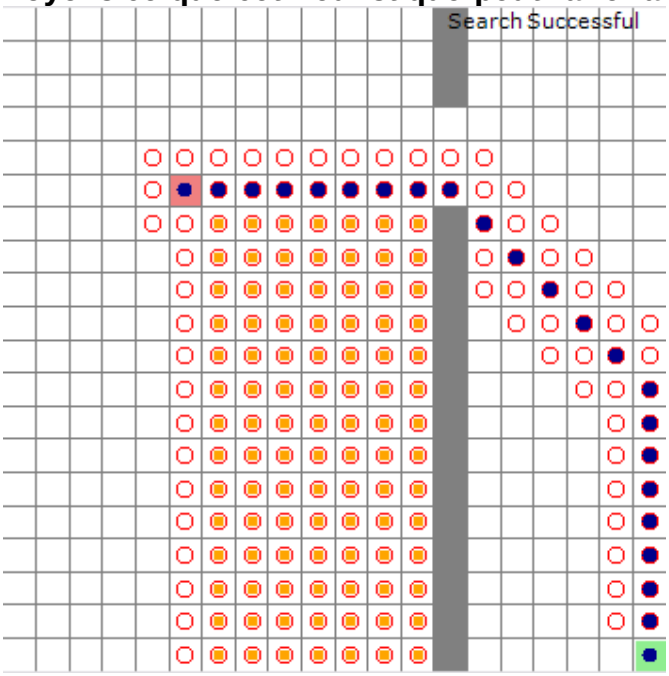
Heuristique : distance maximum

C'est encore pire que la distance euclidienne ! Cet heuristique fait traverser encore plus de nœuds à l'algorithme A*.

Il existe un dernier heuristique, nommé la « distance de Manhattan », qui se calcule comme ceci :

$$h(N) = |X - X_A| + |Y - Y_A|$$

Voyons ce que cet heuristique peut faire faire à notre A* :



Heuristique : distance de Manhattan

La différence est flagrante par rapport aux deux premiers heuristiques : la distance de Manhattan fait en effet parcourir beaucoup moins de nœuds à l'algorithme A* Je n'ai pas encore assez de connaissances mathématiques pour prouver pourquoi un heuristique produit de meilleurs résultats qu'un autre, mais il faut savoir que la fonction h, l'heuristique est censé retourner le *nombre de cases* qu'il faut parcourir pour se déplacer entre un nœud et le nœud d'arrivée. Or, on ne connaît ce nombre de cases que lorsqu'on a déterminé complètement le trajet ; il faut donc faire une approximation de ce nombre de cases. Le calcul

de la distance fournit une approximation correcte, mais la distance de Manhattan se rapproche plus du nombre de cases que la distance euclidienne, par exemple. Les algoriciens montrent que si il existait un heuristique qui retournerait exactement le nombre de cases entre un nœud et le nœud d'arrivée, alors A* passerait directement par le chemin le plus court, sans jamais examiner un seul nœud superflu. Malheureusement, et puisqu'un tel heuristique n'existe pas, on travaille avec des approximations.

Résumé

Pour trouver un chemin avec l'algorithme A* :

- **On ajoute le nœud de départ à une queue Open (qui contient les nœuds à visiter)**
- **Tant qu'il y a des nœuds dans Open :**
  - **On trie la liste Open par ordre de f croissant**
  - **On enlève le premier nœud de Open, qu'on appelle N**
  - **On détermine les successeurs de N.**

    On calcule la valeur de f pour chaque successeur, et on vérifie qu'il n'existe pas déjà un nœud avec un f inférieur dans Closed et Open. On ajoute ce successeur à Closed.
  - **Si l'un d'eux est le nœud d'arrivée alors on a trouvé le chemin.**

Sinon, on continue jusqu'à ce qu'il n'y ait plus de nœuds (condition d'échec).

Problèmes connus

L'algorithme A* ne souffre d'aucun problème connu. Il retourne un chemin qui est exactement le chemin le plus court.

Avantages

Cet algorithme tient compte des poids et examine le moins de nœuds possibles grâce à l'utilisation de l'heuristique qui lui permet une expansion en ligne droite, ce qui lui fait occuper une place en mémoire plus faible que les algorithmes de Dijkstra et BFS.

Inconvénients

A* peut prendre beaucoup de temps de calcul à cause du tri intensif des listes et surtout des calculs de distances (qui nécessitent en plus d'installer un coprocesseur arithmétiques pour les nombres à virgule dans les robots). Il prend donc plus de temps pour faire une recherche que l'algorithme de Dijkstra. Il faut aussi savoir choisir le bon heuristique (dans le cas d'une grille, c'est néanmoins la distance de Manhattan qui prédomine).

Conclusion

On a donc vu cinq algorithmes à travers ce TPE : la ligne droite, DFS, BFS, Dijkstra et A* (qui réutilise la ligne droite à travers la fonction h).
Il est temps maintenant, au vu des inconvénients et avantages de chaque algorithme, de les comparer pour leur utilisation *dans le but de notre TPE* : déplacer notre robot jusqu'à la balle rouge (qui a une position complètement déterminée par la reconnaissance d'images). Notre robot se rapprochant plutôt de la taille d'un robot-chien (cf. Aïbo, en introduction), il a les mêmes limites et capacités (qui rejoignent les limites courantes qu'on retrouve dans la robotisation) :

- **la détermination du trajet ne doit pas prendre trop de temps pour laisser au processeur le loisir de s'occuper d'autres parties du robot ;**
- **la mémoire est quasiment illimitée (tous ces algorithmes ne prennent que quelques méga-octets de mémoire au grand maximum, ce qui est facilement adaptable même sur un robot de cette taille) ;**
- **les poids et les obstacles doivent être pris en compte (notamment si l'utilisateur dit au robot d'éviter les tapis si possible – poids - et les objets fragiles à tout prix – obstacles -) ;**
- **le robot ne doit pas rester bloqué dans le trajet déterminé ; il est préférable que ce trajet soit le plus court, et si possible qu'il soit censé.**

A partir de ces critères, on peut donc classer les algorithmes à notre disposition :

lasse ment	Algorithme	Raisons
1	Dijkstra	Assez rapide, trajet largement optimisé qui ne bloque jamais le robot, prise en compte des poids.
2	A*	Relativement lent par rapport aux autres algorithmes.
3	BFS	Ne tient pas compte des poids. Quelques aberrations dans le trajet.
4	DFS	Trajet pas toujours optimal et parfois complètement à côté, ne tiens pas compte des poids.
5	Ligne droite	Ne tiens pas compte des poids, se bloque parfois dans le trajet.

C'est donc l'algorithme de Dijkstra que l'on retiendrait dans notre analyse pour notre robot.
Mais l'algorithme A* est de plus en plus répandu car il est très efficace dans des domaines variés (détermination intelligente de trajet pour les ordinateurs de

bord, utilisations dans les jeux vidéo) grands public, où la place n'est pas forcément comptée et où les limites technologiques sont sans cesse repoussées. Il remplace souvent l'algorithme de Dijkstra. L'algorithme BFS reste utilisé dans la miniaturisation, même s'il est difficile d'implanter un algorithme complet dans un trop petit espace. L'algorithme DFS quant à lui reste un algorithme de traversée de graphes, car il s'adapte peu à la détermination d'un trajet « optimal ». Enfin, la ligne droite n'est guère utile que comme heuristique, et ne peut guère être utilisée « comme telle ».

Il faut signaler que tous ces algorithmes se programment très facilement (cf. les fonctions de DFS) mais ne sont pas évolutifs : les algorithmes défendus dans la partie III sont beaucoup plus « souples » dans leur utilisation… pour découvrir comment un algorithme peut évoluer et en plus trouver un chemin optimal, tournez la page !

III. Les réseaux neuronaux

1.La théorie

Cette partie étant très théorique et peut être un peu « ennuyeuse », on peut lire uniquement le « f) Résumé sur les réseaux » à la fin de cette partie et passer aux parties suivantes en se reportant à la théorie quand (et si) cela est nécessaire

Note: faute de connaissances mathématiques suffisantes, nous n'avons pas pu réaliser les réseaux présentés ici, ce qui n'empêche heureusement pas de les comprendre.

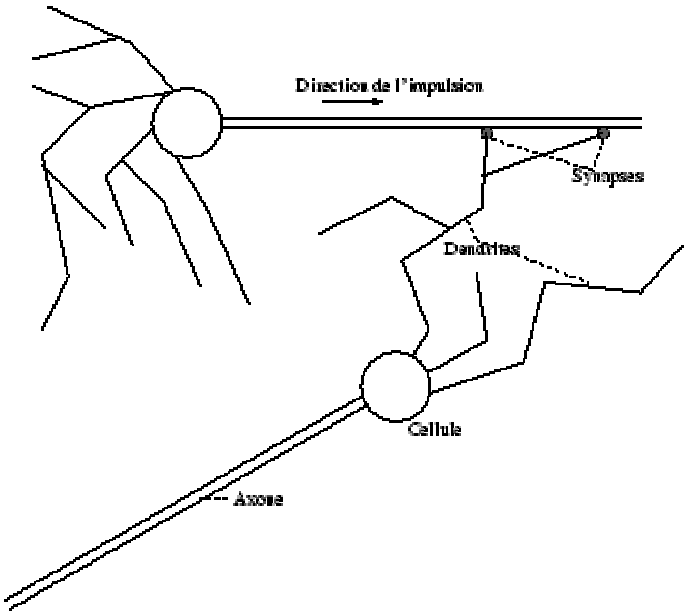
Comment l'homme fait-il pour raisonner, parler, calculer, apprendre, ...? Cette question a mobilisé les chercheurs en intelligence artificielle pendant plusieurs dizaines d'années. Deux types d'approches pour essayer de répondre à cette question de manière automatique ont été essentiellement explorés :

- **procéder d'abord à l'analyse logique des tâches relevant de la cognition humaine et tenter de les reconstituer par programme : ce sont les algorithmes ou les systèmes automatiques qui ont notamment été étudiés dans la partie II.**
- **puisque la pensée est produite par le cerveau, étudier comment celui-ci fonctionne. C'est cette approche qui a conduit à l'étude de réseaux de neurones formels.**

C'est le second point, les « réseaux de neurones » qui seront étudiés ici.

a) Le neurone biologique

Puisque les neurones artificiels s'inspirent des neurones du cerveau, je vais commencer par montrer le fonctionnement de notre cerveau :



Schématisation d'un neurone biologique. Les ronds sont les cellules neuronales. Les lignes simples sont des dendrites et les lignes doubles des axones (cf. ci-dessous).

Les *neurones* reçoivent les signaux qui sont des impulsions électriques par l'intermédiaire de *dendrites*, en fait des extensions ramifiées de la cellule et envoient l'information par de longs prolongements, les *axones*.

Les contacts entre deux neurones, de l'axone à une dendrite, se font par l'intermédiaire des *synapses*. Lorsqu'un potentiel d'action atteint la terminaison d'un axone, des neuromédiateurs sont libérés et se lient à des récepteurs post-synaptiques présents sur les dendrites. L'effet peut être excitateur ou inhibiteur.

Chaque neurone intègre en permanence jusqu'à un millier de signaux synaptiques.

b) Que permettent-ils de faire ?

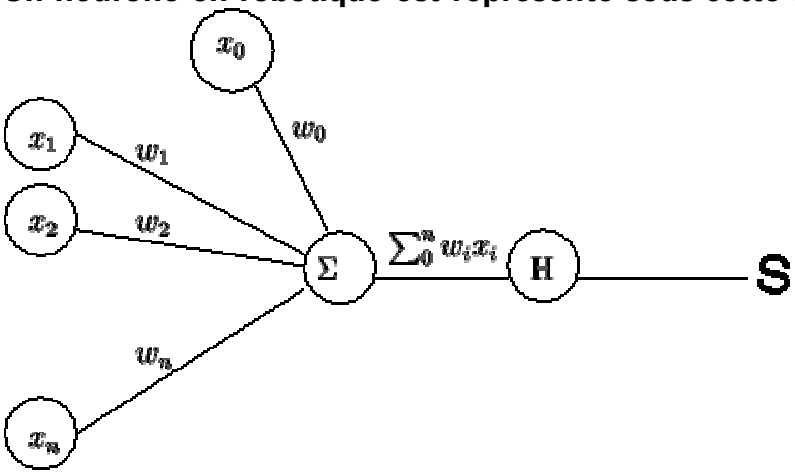
Les réseaux de neurones, aussi appelés réseaux neuro-mimétiques sont ce qu'il y a de plus efficace pour mettre en place la cognition dans un programme. Qu'est-ce que la cognition ? La cognition caractérise des outils tels que la reconnaissance de caractères, la reconnaissance vocale, ou encore la reconnaissance d'images. Les réseaux de neurones sont à la base de tous ces outils. Mais ce n'est pas tout : leurs capacités vont bien au delà de la simple cognition ; en effet ces réseaux peuvent également être utilisés pour la synthèse vocale, ou l'improvisation de mélodies (en temps réel !) appuyée par une musique déjà existante. Ils peuvent également être utilisés dans des cas comme les IDS (Intrusion Detection System), ou encore pour donner "vie" à des robots virtuels ou non. Le robot Mars Pathfinder qui roula sur le sol martien est un bon exemple : l'un des principes qui régissent ce robot sont les réseaux neuro-mimétiques.

La particularité de ces réseaux réside dans le fait qu'ils sont capables d'apprendre une fonction qui prend n variables en entrée, et retourne m variables en sortie (les variables étant des nombres réels). Ce sont de telles fonctions qui régissent les actions d'un robot : prenons par exemple Mars Pathfinder. Pour se déplacer, le « cerveau » du robot prend en entrée les informations de ses capteurs qui lui disent sa position, les obstacles, etc. et doit retourner en sortie des instructions pour ses moteurs, par exemple. Un réseau de neurones prend tous les signaux de capteurs en entrée et à partir de ses entrées et de ce qu'il a déjà en mémoire, est capable de donner les bons ordres à ses actionneurs : le réseau agit donc comme une fonction à plusieurs variables.

Pour l'instant, les réseaux neuronaux paraissent semblables aux systèmes experts (qui ne sont que des programmes écrits une fois pour toutes et incapables d'évoluer). Mais les réseaux neuro-mimétiques sont des usines à apprendre, elles évoluent sans cesse (pour peu qu'on leur demande) et si par exemple un jour, un paramètre venait à changer, contrairement aux systèmes experts, les réseaux neuro-mimétiques s'adapteraient automatiquement sans intervention humaine : c'est la magie de l'apprentissage, ce que sont capables de faire les neurones. Mais avant de passer à l'apprentissage proprement dit, voyons d'abord comment un neurone est représenté sous forme robotique.

c) Le neurone virtuel

Un neurone en robotique est représenté sous cette forme :



Le neurone virtuel

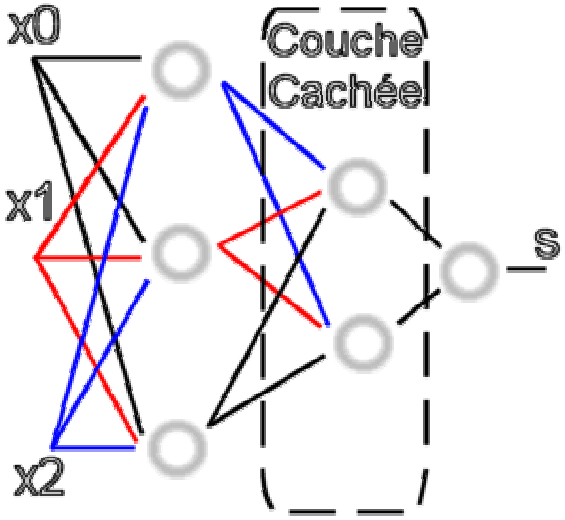
On voit le neurone (le cercle central) qui réalise l'intégration des signaux, des entrées notées x_n. A chaque entrée correspond un poids. Ce poids est utilisé dans le traitement de l'information ; en effet, le neurone réalise la somme pondérée de toutes les entrées : $N = x_1 * w_1 + x_2 * w_2 + ... + x_n * w_n$.

On fait ensuite passer la valeur N à travers une fonction H. H peut être une fonction de seuil, tel que H(N) peut prendre les valeurs 0 ou 1 par exemple, suivant les valeurs de N, ou peut encore être une fonction plus compliquée (le choix de la fonction est fait suivant les sorties que doit donner le réseau et ne fait pas partie de ce TPE – on peut considérer pour l'instant que la fonction H est une fonction de seuil).

Le neurone est l'unité de base des réseaux neuronaux, c'est lui qui « réalise les calculs ». Mais seul, il est inutile. C'est pour cela qu'on associe plusieurs neurones dans des réseaux de neurones.

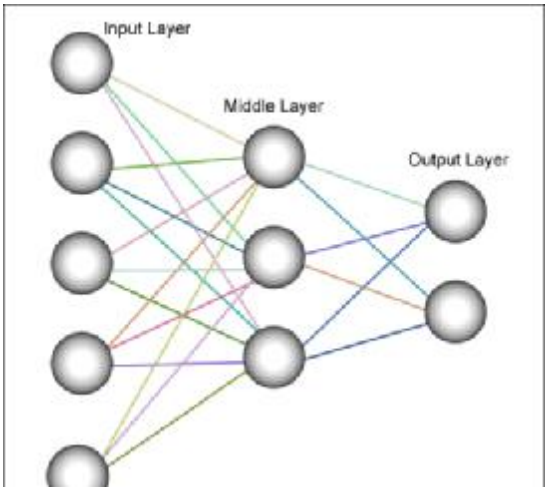
d) Les réseaux

Un réseau de neurones est formé de la liaison de plusieurs neurones agencés en couches, comme présenté sur cette figure :



On voit la forme d'un réseau de neurones multi-couches.

On voit donc un réseau formé de trois couches de neurones. La première couche est constituée de 3 neurones et s'appelle la couche d'entrée. Tous les neurones de cette couche prennent leurs entrées de trois variables x0, x1 et x2. Leurs sorties sont reliées aux neurones de la couche secondaire, la couche cachée. Les neurones de la couche cachée prennent leurs entrées des neurones de la couche d'entrée et leurs sorties sont combinées pour calculer la sortie finale du réseau S. On voit que ce réseau modélise une fonction qui prend 3 variables en entrée et en retourne une en sortie (nous verrons plus en détail dans les parties suivantes ce que peuvent être les variables d'entrée et de sortie, mais ce sont toujours des nombres réels). A chaque trait sur la figure ci-dessus est associé une liaison avec un poids (cf. le schéma d'un neurones plus haut).



**Un autre exemple de réseau, avec un nombre différent de couches et de neurones.
Ce réseau peut prendre 5 variables en entrée et retourner deux sorties.**

Les poids affectés aux connections permettent de relier les entrées aux sorties du réseau par le calcul effectué par chaque neurone.

Lorsqu'on présente aux neurones d'entrées différentes valeurs, le calcul successif des sorties de neurones de toutes les couches permet de calculer les sorties correspondantes. A chaque liste d'entrées correspond donc une liste de sorties que le réseau peut calculer.
Mais comment déterminer ces poids dans le but de faire une approximation de la fonction que l'on veut reproduire avec le réseau ? Et pourquoi parle t-on des capacités d'*évolution* de ces réseaux ?

e) L'apprentissage

On a vu précédemment qu'un réseau était capable d'approximer n'importe quelle fonction, pour peu qu'on lui mette les poids nécessaires aux neurones appropriés. Pour la plupart des fonctions que l'on peut représenter par de tels réseaux, on dispose d'un pack d'échantillons (un échantillon étant une liste d'entrées et la liste de sorties correspondantes). Il suffit de présenter au réseau un exemple (un échantillon) d'entrées et les sorties correspondantes pour qu'il l' « apprenne ». Que se passe t'il quand un réseau a appris l'échantillon ? Il suffit alors de lui présentées comme valeurs d'entrée les entrées présentes dans l'échantillon pour qu'il sorte après calcul une approximation des valeurs de sorties qui étaient dans l'échantillon : le réseau a *appris* cet échantillon. Le réseau est capable d'apprendre plusieurs échantillons ; ce sont les neurones qui réalisent cette fonction mémoire, car ils enregistrent les poids correspondants. Plus il y a de neurones dans un réseau et plus on pourra enregistrer d'échantillons différents tout en gardant une précision de sortie suffisante. On peut apprendre le même échantillon plusieurs fois pour augmenter cette précision. Quand tous les échantillons ont été présentés et que les valeurs de sorties sont satisfaisantes, le réseau réalise alors l'approximation de la fonction considérée. Le principal avantage des réseaux sur les systèmes classiques est qu'ils peuvent donner des sorties satisfaisantes pour des entrées qui n'ont jamais été données en exemple. Leur second avantage est qu'ils peuvent apprendre en temps réel ; imaginons par exemple un réseau qui s'occuperait de la surveillance d'une chaîne de production. Les capteurs envoient leurs données au réseau (les capteurs sont des entrées) qui détermine à l'aide de ces informations si un défaut (« présence défaut » est une sortie du réseau) est près de se produire sur la chaîne et l'arrête alors. Si un défaut est quand même signalé par des capteurs externes ou par un opérateur (par l'intermédiaire de l'arrêt d'urgence, par exemple) se produit alors que le réseau ne l'avait pas prévu, le réseau apprend alors que les entrées qu'il avait au moment du défaut correspondent effectivement à une possibilité de défaut et qu'il doit être signalé grâce à la sortie correspondante. Ainsi, alors que les concepteurs du réseau n'avaient pas pensé ou pas prévu un défaut dans ces conditions, le réseau l'apprend et sera capable de le signaler si le même cas se répète, ce qui permet de prédire les erreurs futures et d'éviter l'interruption prolongée de la production, dans ce cas.
Il existe bien évidemment des algorithmes qui permettent de trouver les poids et de les modifier à partir des échantillons, mais ils ne font pas l'objet de ce TPE.

f) Résumé sur les réseaux neuronaux

- **Un « réseau neuronal » est constitué de plusieurs neurones (des unités de calcul simples) reliés entre eux. Ces neurones sont agencés en couches.**
- **L'association des neurones permet de simuler une fonction (en faire une approximation) à plusieurs paramètres, c'est-à-dire de calculer des valeurs de sortie voulues à partir de valeurs d'entrée.**

- **Plus il y a de neurones, plus le réseau peut mémoriser de paires {valeurs d'entrées / valeurs de sortie}.**
- **Un réseau peut apprendre cette fonction, si on lui présente des exemples. Un exemple étant un échantillon de valeurs d'entrées et les valeurs de sorties que l'on veut faire calculer au réseau quand on lui présente ces entrées. Il peut apprendre en temps réel de nouveaux exemples si on lui présente les bons échantillons.**
  Il peut ainsi simuler une intelligence humaine (par le mécanisme d'action-réaction : à un état correspondent des actions appropriées).

2. Application à la reconnaissance d'images

Les réseaux neuromimétiques sont couramment utilisés dans des problèmes utilisant des images. Le plus souvent, ils sont impliqués dans des applications de reconnaissance de caractères (OCR), dont l'objectif est de transformer une écriture manuscrite en données informatiques. Leur capacité d'apprentissage leur permet de s'adapter à une écriture humaine, afin de pouvoir la « lire ».

Nous pouvons également les utiliser pour déterminer la position d'une balle sur une image, même si cela peut sembler moins évident. En effet, nous n'avons trouvé aucune application pratique effectuant une recherche similaire grâce à un réseau de neurones, mais cela reste tout à fait faisable.

Il faudra tout d'abord simplifier l'image. Un traitement basique permettra de diminuer le nombre de couleurs et de supprimer le bruit. Même si cela n'est pas forcément nécessaire, cela ne pourra que simplifier la tâche du réseau de neurones.

La taille de l'image définira la dimension de la couche d'entrée du réseau. On peut donc diminuer la résolution de l'image, afin de restreindre le réseau.

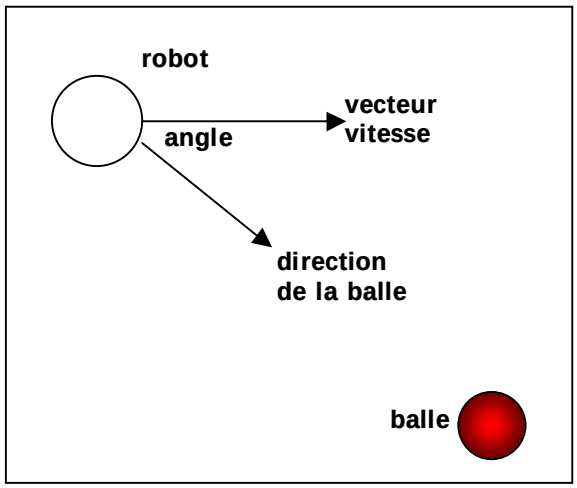
Entrées du réseau : les pixels de l'image et les informations sur l'objet recherché
Sorties : la position de la balle

Il est probablement possible de fournir comme informations sur l'objet recherché une image de cercle rouge au réseau. Celui ci tentera donc de la localiser sur l'image.

Nous ne pouvons donner plus d'informations sur l'utilisation d'un réseau neuronal pour déterminer la position d'un objet sur une image, car cela fait partie des sujets actuellement étudiés par les développeurs de logiciels commerciaux de traitement d'image. Nous ne disposons donc que de très peu d'informations à ce sujet.

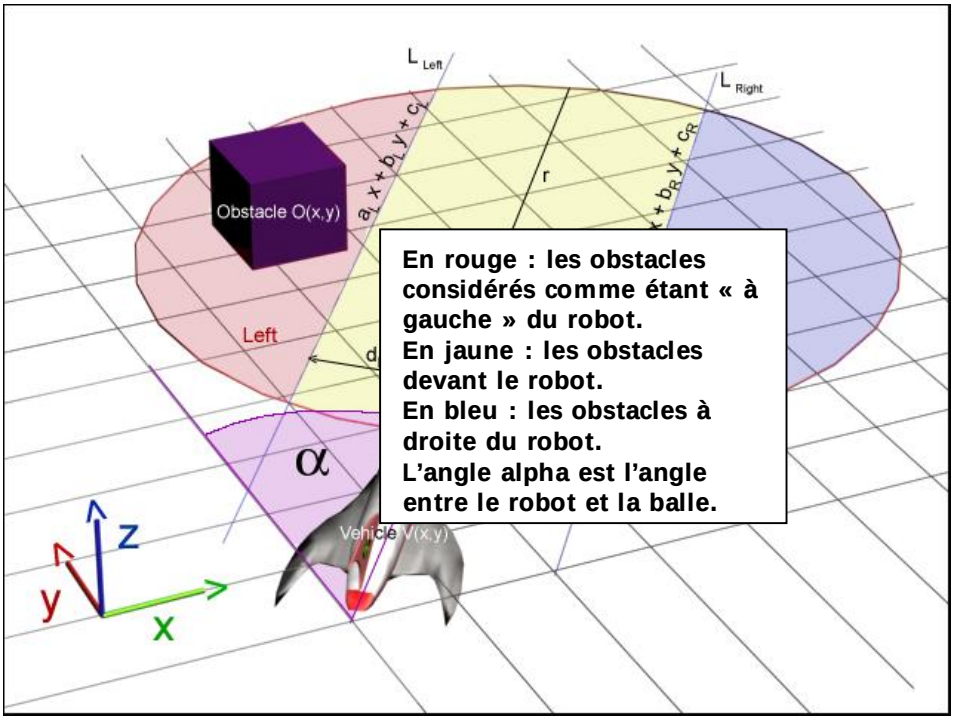
3. Application à la Détermination Intelligente de Trajet

Dans la partie II, le robot avait en « tête » une représentation précise de l'environnement qui l'entourait. Les algorithmes qui lui permettaient de trouver son chemin jusqu'à son but (la baballe) faisaient grand usage de cette représentation interne pour optimiser et trouver le chemin. Mais si on ne connaît pas l'environnement (par manque d'informations, de capteurs, de temps etc.), on peut utiliser un réseau neuronal pour déplacer le robot à travers l'environnement en direction de la balle.



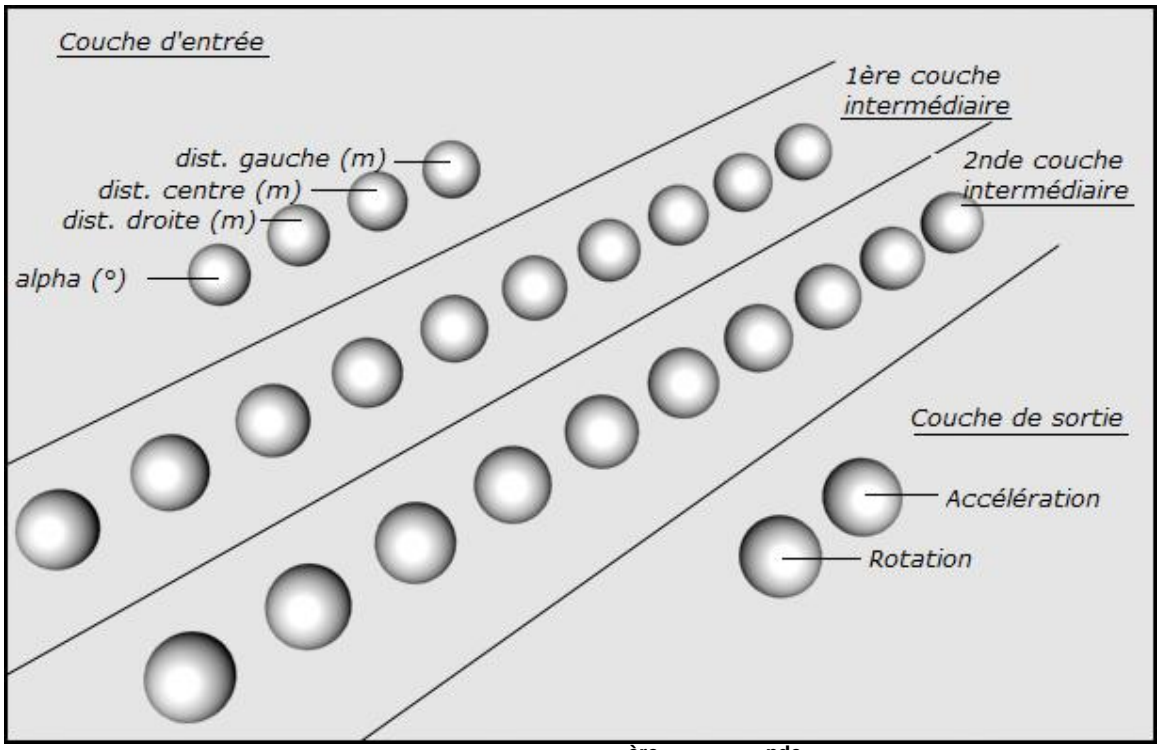
a) Les entrées-sorties

Nous dirons donc que nous disposons d'une info : la position de la balle par rapport au robot (qui peut être modélisée par un angle entre le vecteur direction du robot et le vecteur robot -> balle). Une autre information (les informations vont former les entrées du réseau) sera formée par la position d'un obstacle. On décide de prendre en compte les obstacles situés devant le robot et plus particulièrement devant à gauche, devant à droite et devant tout droit, comme indiqué sur la figure suivante :



On va donc ajouter trois informations d'entrée au réseau. La première sera la distance de l'obstacle le plus proche situé sur la gauche (en mètres, par exemple). La seconde, la distance de l'obstacle le plus proche situé au centre, et la troisième, la distance de l'obstacle le plus proche situé à droite. Par exemple, dans la situation présentée ci-dessus, l'échantillon d'entrée serait {45°, 0.5, 1.0, 1.0}. Le premier chiffre est l'angle formé entre la balle et la direction qu'a prise

le robot. Le second est la distance du robot à l'obstacle O. Il n'y a d'obstacle ni au centre ni à droite ; on met donc la distance maximale (située en l'occurrence à 1m du robot) pour les deux dernières valeurs, ce qui signifie « pas d'obstacle pour l'instant ».
Comme sorties pour le robot, on choisira de retenir deux valeurs importantes pour le circuit de commande : *rotation* et *accélération*. La rotation sera une valeur comprise entre 0 et 1, 0 correspondant à tourner à fond à gauche, 0.5 à continuer tout droit et 1 à tourner à fond à droite.
L'accélération sera aussi une valeur comprise entre 0 et 1, 0 étant freiner à fond, 0.5 continuer à la même vitesse et 1 accélérer à fond (il y a toujours une vitesse « de croisière » qui bloquera l'accélération).
On peut donc construire notre réseau neuronal comme ceci :



Le nombre de neurones utilisés pour la 1ère et la 2nde couches intermédiaires a été décidé après plusieurs essais.
A chaque seconde (ou intervalle de temps encore plus court), on met en entrée les valeurs données par les capteurs, et on laisse le réseau calculer les sorties correspondantes, qui seront traitées par le robot et utilisées pour actionner les moteurs.

b) L'entraînement

Avant de pouvoir utiliser le réseau, il faut l'entraîner avec différents exemples. Voici les valeurs qu'on lui présenterait en exemple : (chaque ligne est un exemple)

Neurones d'entrée Distance à l'obstacle			Sorties associées	
Gauche	**Centre**	**Droite**	**Accélération**	**Rotation**
Aucun obstacle	Aucun obstacle	Aucun obstacle	Accélération maximum	Tout droit
Moitié de la distance maximum	Aucun obstacle	Aucun obstacle	Accélérer un peu	Un peu vers la droite

Aucun obstacle	Aucun obstacle	Moitié de la distance maximum	Accélérer un peu	Un peu vers la gauche
Aucun obstacle	Moitié de la distance maximum	Aucun obstacle	Ralentir	Un peu vers la gauche
Moitié de la distance maximum	Aucun obstacle	Moitié de la distance maximum	Accélérer	Tout droit
Touche presque l'obstacle	Touche presque l'obstacle	Touche presque l'obstacle	Freiner à fond	A gauche
Moitié de la distance maximum	Moitié de la distance maximum	Moitié de la distance maximum	Garder la vitesse	Un peu vers la gauche
Touche presque l'obstacle	Aucun obstacle	Aucun obstacle	Ralentir	A fond à droite
Aucun obstacle	Aucun obstacle	Touche presque l'obstacle	Ralentir	A fond à gauche
Aucun obstacle	Touche presque l'obstacle	Aucun obstacle	Freiner à fond	A gauche
Touche presque l'obstacle	Aucun obstacle	Touche presque l'obstacle	Accélération maximum	Tout droit
Touche presque l'obstacle	Touche presque l'obstacle	Aucun obstacle	Freiner à fond	A fond à droite
Aucun obstacle	Touche presque l'obstacle	Touche presque l'obstacle	Freiner à fond	A fond à gauche
Obstacle proche	Obstacle proche	Obstacle très proche	Garder la vitesse	A gauche
Obstacle très proche	Obstacle proche	Obstacle proche	Garder la vitesse	A droite
Touche presque l'obstacle	Obstacle très proche	Obstacle très proche	Ralentir	A fond à droite
Obstacle très proche	Obstacle très proche	Touche presque l'obstacle	Ralentir	A fond à gauche
Touche presque l'obstacle	Obstacle proche	Obstacle loin devant	Garder la vitesse	A droite
Obstacle loin devant	Obstacle proche	Touche presque l'obstacle	Garder la vitesse	A gauche
Obstacle très proche	Obstacle proche	Obstacle proche de la moitié	Garder la vitesse	A fond à droite
Obstacle proche de la moitié	Obstacle proche	Obstacle très proche	Ralentir	A fond à gauche

Et les échantillons pour gérer le déplacement vers la balle :

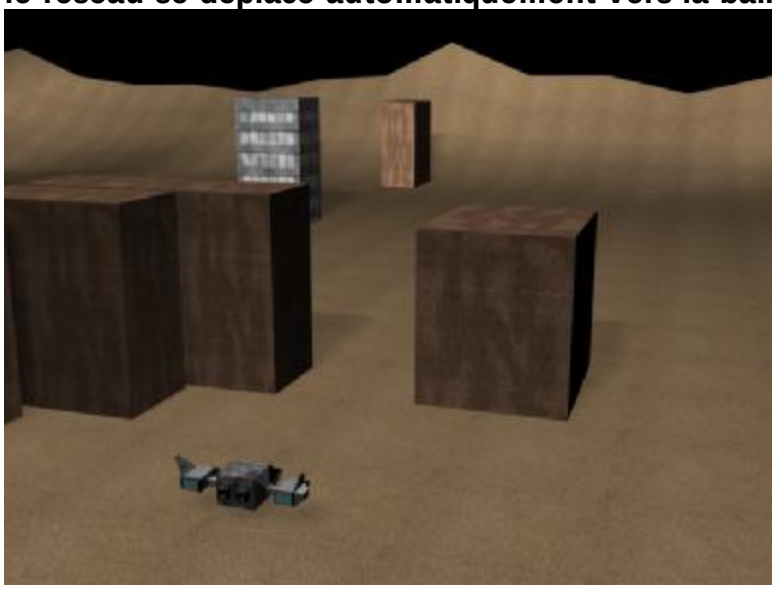
Neurones d'entrée Angle en degrés	Sorties associées	
Position relative de la balle	Rotation	Rotat
Complètement à gauche	A fond à gauche	
A gauche	A gauche	
Tout droit	Tout droit	
A droite	A droite	
Complètement à droite	A fond à droite	

On associe maintenant ces échantillons à des valeurs numériques :

Neurones d'entrée Distance à l'obstacle			Sorties associées	
Gauche	Centre	Droite	Accélération	Rotation
1.0	1.0	1.0	1.0	0.5
0.5	1.0	1.0	0.6	0.7
1.0	1.0	0.5	0.6	0.3
1.0	0.5	1.0	0.3	0.4
0.5	1.0	0.5	0.7	0.5

…	…	…	…	…

Ces couples d'actions – réactions correspondent à ce que ferait un être humain confronté aux mêmes situations.
Une fois que l'on a entraîné ce réseau avec les valeurs, le robot ainsi conduit par le réseau se déplace automatiquement vers la balle en évitant les obstacles :



Le plus gros problème posé par l'utilisation des réseaux neuronaux dans ce cas est le temps de calcul, qui est très grand avec le nombre de connexions entre neurones, chaque connexion demandant le calcul d'une somme pondérée par un neurone. Dans notre cas, nous avons en tout 260 connexions en reliant tous les neurones entre eux, ce qui fait une « grande » somme de calculs pour un processeur embarqué sur un robot.

Les réseaux neuronaux sont encore à un stade de « test » dans leurs applications ; en effet, les formules qui les régissent sont tellement compliquées que peu de chercheurs arrivent à comprendre complètement et concevoir de tels réseaux. Néanmoins, ils ont beaucoup d'applications dans plusieurs domaines qui se situent surtout, pour l'industrie, dans la *prévention* de problèmes ou d'évènements (défauts dans une chaîne de production, …) qui pourraient survenir sur un système.

Conclusion

Nous avons vu dans ce TPE combien les algorithmes actuels sont performants et adaptés pour effectuer des opérations relativement simples, comme localiser et aller chercher une balle. Ils sont également très compétents dans beaucoup d'autre domaines, comme la détection de défaut (par exemple dans les chaînes de fabrication industrielles), la simulation de comportement humain (très utilisée dans les jeux vidéos, par exemple dans un jeu d'échecs) ou encore l'assistance pouvant être fournie à un utilisateur (système de visée sur les avions de chasse ou de repérage sur les avions de ligne). L'utilisation des nombreux algorithmes existants permet de simplifier notre vie, en nous débarrassant des tâches les plus répétitives et en nous assistant pour les plus complexes.

Beaucoup de types d'algorithmes que nous n'avons pas présenté ont étés conçus pour répondre à des problèmes précis, comme par exemple les algorithmes experts, basés sur la comparaison à des situations préenregistrées et par exemple utilisés pour créer une intelligence artificielle dans un jeu d'échecs, ou les algorithmes génétiques, basés sur le principe d'évolution définit par le biologiste Darwin, utilisés principalement pour des optimisations.

Les algorithmes neuronaux font partie des domaines de l'intelligence artificielle qui évoluent actuellement le plus rapidement car beaucoup de recherches leur sont consacrées, et leurs possibilités sont énormes. Leur capacité unique à l'apprentissage fait d'eux les algorithmes pouvant le plus évoluer. Qui sait ce pour quoi on pourra les utiliser dans une dizaine d'années ...

Robot martien utilisant des algorithmes de pointe

Sources

www.GameDev.net http://prografix.games-creators.org

www.Generation5.org

www.ProgramamtionWorld.com

http://galileo.cpe.ma.cx

www.c2i.fr

www.gamasutra.com

www.vieartificielle.com

http://www.dearesin.cicrp.jussieu.fr/dearesin/module7/
http://glutro.free.fr/projet/nasp.htm
http://www.jautomatise.com
http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html
http://apiacoa.org/software/pathfinding/
http://www.sbcomputing.de/heni/libkdegames/pathdoc/astar_search.html
http://www.policyalmanac.org/games/aStarTutorial.htm