# Optimisation opérationnelle des flux maritimes

## Solutions algorithmiques et application pratique

Lucas RODRIGUEZ

31 août 2019

**Résumé**

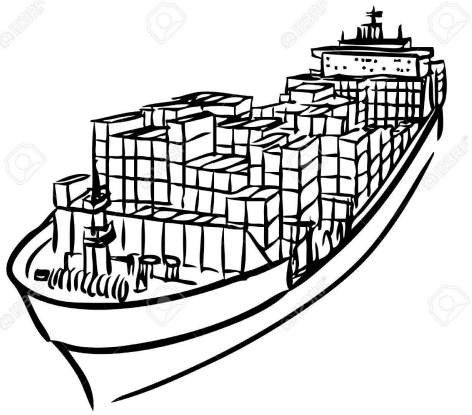
Ce projet s'inscrit dans une logique moderne relative à la prévision d'itinéraires à moindre coût. L'importance de la prévision et de l'optimisation en temps réel des trajets permet de répondre à des contraintes de nature diverses : économiques, écologiques et de respecter un cahier des charges temporel. Cette problématique est omniprésente dans un monde fortement mondialisé où les échanges maritimes occupent une place primordiale dans un espace aussi vaste que complexe : l'océan. L'objectif établi sera d'étudier et d'implémenter les fondements de solutions algorithmiques complètes permettant de déterminer le chemin à moindre coût envisageable par un navire. Nous nous intéresserons tout d'abord à l'algorithme de Dijkstra, puis A*. Enfin, nous raffinerons certaines méthodes pour rendre un résultat optimal en un minimum de temps sur des maillages importants.

**Remerciements** Je tiens à remercier Monsieur De Saint-Julien pour son aide précieuse apportée lors du développement de mon projet, mais également Pierre GARREAU, CTO de *Searoutes - MaritimeDataSystems* pour m'avoir permis d'appliquer une discrétisation efficace 'a la solution proposée.

**Mots-clés** *Optimisation numérique - Approche gloutonne - Modélisation mathématique - Théorie des graphes - Approche heuristique*

**Keywords** *Numeric optimization - Greedy algorithmic approach - Mathematical model - Theory of graph - Heuristic approach*

*Les implémentations ont été réalisées en* **Python** *(3.6.4) et* ***SQL*** *(SQLite 3.27.2).*



# Table des matières

# 1 Introduction

## 1.1 Motivation et placement thématique

Depuis une dizaine d'années, de nombreuses entreprises ont mis au point des systèmes permettant de fluidifier et optimiser le trafic maritime mondial. Ces solutions, souvent coûteuses, sont devenues des éléments indispensables pour dominer au mieux l'espace maritime. Les usages, qu'ils soient militaire, commercial ou même scientifique répondent à une problématique commune : « le gain de temps et de ressources »et font intervenir de nombreux acteurs, présents sur la scène maritime internationale :

- Zones de piraterie
- Zones de conflit
- Territoires aquatiques impraticables pendant des durées limitées
- Courants et vents contraires menant à une perte d'efficacité de la propulsion (rendement bas, perte haute)
- Conditions météorologiques défavorables
- Obstacles statiques et dynamiques

Chacun d'entre eux jouent un rôle qui ne peut être négligé. La difficulté sera alors de tous les prendre en compte au sein de la modélisation, afin que le résultat obtenu soit conforme à la réalité.

L'objectif de ce TIPE sera donc de mettre en œuvre des solutions algorithmiques dans le but d'optimiser ces flux maritimes, tout en adoptant une démarche stratégique cohérente permettant de n'omettre aucun acteur.

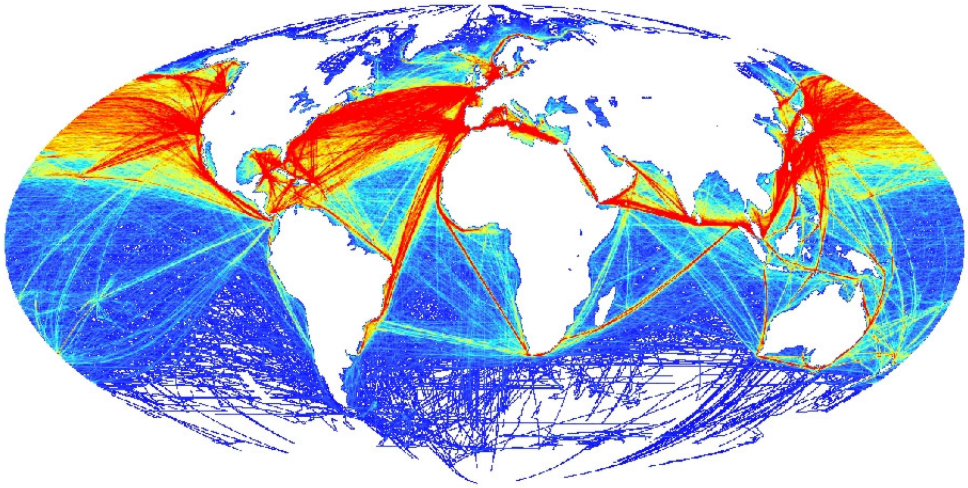


FIGURE 1 – Activité maritime commerciale mondiale (Février 2010)

## 1.2 Cahier des charges

Par « optimiser », on cherchera bien entendu à déterminer le chemin complet le moins coûteux pour un navire choisi, souhaitant à partir d'un point $s$ (point de départ), atteindre une cible $c$ (point d'arrivée). Cette question de minimisation du coût sera relative à une pondération des flux engendrée par l'action des différents acteurs. Voici un exemple concret :

**Exemple 1.** *On considère un bateau commercial souhaitant se rendre au port de Hong-Kong. Sa position initiale est Marseille. La route la plus cohérente serait de passer par le Canal de Suez. Cependant, des restrictions douanières ainsi que la présence de multiples zones de conflit représentent une zone fortement défavorable pour le navire. D'autres solutions sont possibles : Contourner le continent africain par le Sud ou l'Eurasie par le Nord. La deuxième solution semble cohérente mais présente un inconvénient : le territoire est impraticable en hiver, tandis que la première présente des zones de piraterie et des conditions météorologiques défavorables....*

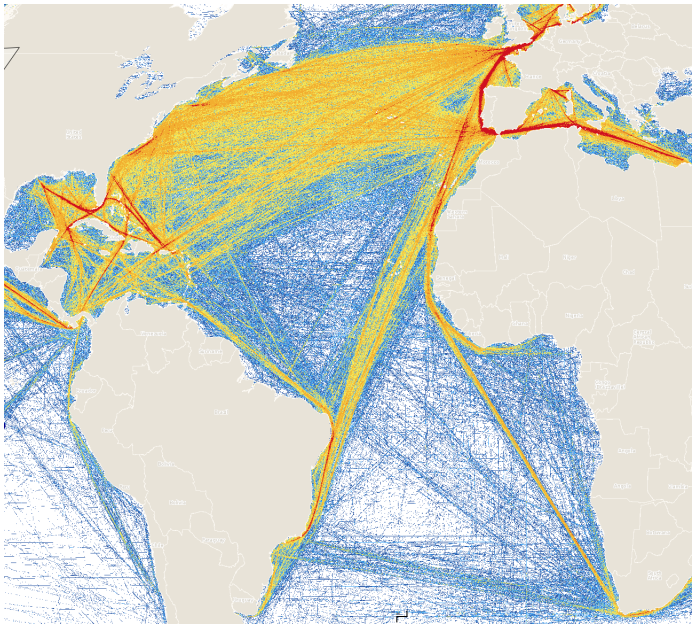


FIGURE 2 – Activité maritime commerciale de la face Atlantique (Février 2010) - Source : NCEAS

La solution algorithmique pourra prendre en compte ces différents facteurs et appliquer ainsi à chaque flux un coût en fonction de la distance géographique $s - c$, mais également de la mesure de l'influence de chacun des acteurs sur la traversée du flux, illustrant fidèlement la pertinence du flux.

## 1.3 Problématique générale

La problématique retenue est : « **Comment optimiser le trafic maritime global ?** ». Cependant, ce problème soulève d'autres questions auxquelles nous devrons apporter une réponse avant de poursuivre :

1. Comment numériser le plus fidèlement possible la situation réelle ? [**Etape de discrétisation**]
2. Quelles sont les structures de données les mieux adaptées au problème posé ? [**Etape de veille technique**]
3. Comment pourrait-on traiter et parcourir les données de façon efficace ? [**Etape d'initialisation**]
4. Comment trouver le chemin le moins coûteux ? [**Etape de traitement**]

Les trois premières questions relèvent du choix de la structure de données utilisée. En effet, c'est sur celle-ci que la discrétisation du modèle repose : l'étape la plus importante.

Pour mener à bien la modélisation, nous devons représenter les différents points majeurs (ports maritimes, détroits et points de cheminement), ainsi que les connexions qui s'opèrent entre eux : ces flux étant soumis aux acteurs cités plus hauts. La solution optimale retenue pour discrétiser notre modèle est l'utilisation de graphe orienté pondéré [1].

Cette structure de graphe se révèle très pertinente pour la détermination du chemin le moins coûteux [2].

## 1.4 Définitions et notations formelles

Cette partie sert de lexique pour les définitions de tous les termes utilisés dans cet exposé. Les notations les plus récurrentes seront soulignés dans la marge, pour en faciliter la lecture.

1. On utilisera des graphes orientés afin de distinguer nettement les 2 sens de circulation d'une route maritime. Ils seront de plus pondérés par les poids attribués suivant l'influence des acteurs.
2. Ce problème se nomme également : Problème du chemin le plus court (abrégé PCC). Cependant, vu que la distance géographique séparant 2 points n'est pas l'unique facteur de minimisation, le nom de chemin à moindre coût est plus approprié.

### 1.4.1 Définitions

**Définition 1** (Graphe orienté)**.** *Un graphe orienté fini $G = (V, E)$ est défini par :* $G$, $V$, $E$
- *l'ensemble $V = \{v_1, v_2, ..., v_n\}$ dont les éléments sont appelés* ***sommets***
- *l'ensemble $E = \{e_1, e_2, ..., e_m\}$ dont les éléments sont appelés* ***arcs***

*Généralement, on a $E \subset V \times V - \{(x, x) \mid x \in V\}$*
*Un arc $e$ de l'ensemble $E$ est défini par une paire ordonnée de sommets. Lorsque $e = (x, y)$, on dit alors que l'arc $e$ va de $x$ à $y$. On dit aussi que $x$ est l'extrémité initiale et $y$ l'extrémité finale.* $e$

Ici, tous les arcs possèdent la même distance entre chaque noeud ; l'ajout d'une valuation des arcs permet ainsi de répondre au cahier des charges.

**Définition 2** (Graphe orienté pondéré)**.** *Un graphe orienté pondéré (ou valué) fini est un triplet $G = (V, E, \omega)$ avec l'application $\omega : E \to \mathbb{K}$, où $\mathbb{K} = \mathbb{R}$ ou $\mathbb{R}^+$ ou $\mathbb{Z}$ suivant la situation.* $\omega$
*On peut étendre cette valuation en posant :*

$$\forall (x, y) \in V^2,\ \omega(x, y) = +\infty \text{ si } (x, y) \notin E$$

**Remarque 1.** *Quelques formalismes utiles pour la suite :*
- *Un graphe orienté fini est également appelé* ***digraphe*** *(pour directed graph).*
- *$n = |V| = \text{card}(V)$ : le nombre de sommets de $G$.* $n$
- *$m = |E| = \text{card}(E)$ : le nombre d'arcs de $G$.* $m$

Par la suite, on notera les arcs de la manière suivante : $(x, y)$, $xy$, $x \rightsquigarrow y$ ou $< xy >$.

**Définition 3** (Adjacence, incidence, voisinage, degré)**.** *Soit $G = (V, E, \omega)$ un digraphe :*
- *Deux sommets $x$ et $y$ de $G$ sont* ***adjacents*** *si $xy \in E$.*
- *L'arc $xy$ est* ***incident*** *aux sommets $x$ et $y$.* $xy$
- *Le* ***voisinage*** *d'un sommet $x$ est défini par $\Gamma(x) = \{y \in V \mid xy \in E \text{ ou } yx \in E\}$.* $\Gamma(x)$
- *Le* ***degré*** *du sommet $x$ est alors $d(x) = |\Gamma(x)| = \text{card}(\Gamma(x))$.* $d(x)$

**Définition 4** (Voisinages extérieurs et intérieurs)**.** *Soit $x$ un sommet d'un digraphe $G = (V, E, \phi)$ :*
- *Le* ***voisinage extérieur*** *de $x$ est défini par $\Gamma^+(x) = \{y \in V \mid xy \in E\}$.* $\Gamma^+(x)$
  *Il s'agit de l'ensemble des successeurs de $x$.*
- *Le* ***voisinage intérieur*** *de $x$ est défini par $\Gamma^-(x) = \{y \in V \mid yx \in E\}$.* $\Gamma^-(x)$
  *Il s'agit de l'ensemble des prédécesseurs de $x$.*

*Ainsi, on définit le* ***voisinage*** *:* $\boxed{\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)}$.

**Définition 5** (Chemin, Chaîne et Longueur)**.** *Soit $G = (V, E)$ un graphe orienté :*
- *Un* ***chemin*** *d'un sommet $u$ vers un sommet $v$ est une séquence $\mu = (s_1, ..., s_k)$ de sommets avec : $u = s_1$, $v = s_k$ et $\forall i \in \{1, ..., k-1\}, s_i s_{i+1} \in E$.* $\mu$
  *On dira que le chemin contient les sommets $s_1, ..., s_k$ et les arcs : $(s_1, s_2), ..., (s_{k-1}, s_k)$.*
- *La* ***longueur d'un chemin*** *$l$ est le nombre d'arêtes que contient ce chemin.* $l$
- *Une* ***chaîne*** *de longueur $k$ est un parcours sans répétition d'arcs : $\mu =< s_1, ..., s_k >$.*
- *Un chemin est* ***élémentaire*** *s'il n'y a pas de répétition de sommets :*

$$\forall (s_i, s_j) \in \mu^2, s_i \neq s_j$$

Les chaînes élémentaires joueront un rôle essentiel dans la détermination du chemin le moins coûteux. En effet, un itinéraire passant par le même point ou par le même chemin n'est pas du tout pertinent et donc à proscrire. Dans notre situation, cela représenterait un cas extrêmement défavorable où la compagnie maritime accuserait de lourdes pertes de ressources.

### 1.4.2 Propriétés fondamentales

**Lemme 1** (Lemme de König)**.** *Soit un graphe orienté $G = (V, E)$, s'il existe un chemin/chaîne d'un sommet $u$ vers un sommet $v$, alors il existe un chemin élémentaire de $u$ vers $v$.*

*Démonstration.* Soit $x$ et $y$ : 2 sommets d'un graphe orienté $G = (V, E)$. Afin de prouver ce lemme, on va choisir un chemin particulier entre $x$ et $y$ et montrer qu'il est élémentaire. Un candidat pourrait être un chemin élémentaire semble être un plus court chemin.
Parmi tous les chemins reliant les 2 sommets $x$ et $y$ : on choisit

$$\mu = (s_1, s_2, ..., s_k)$$

avec $l(\mu) = k$ avec $x = s_1$ et $y = s_k$. On choisit $\mu$ de façon qu'il comporte le moins d'arcs possibles.
Supposons par l'absurde que $\mu$ n'est pas élémentaire. Il existe alors dans ce chemin un sommet $z$ apparaissant au moins 2 fois dans $\mu$.
On note $i$ et $j$ les 2 premiers indices de $z$, c'est-à-dire $s_i = z$ et $s_j = z$.
On a donc :

$$\mu = (s_1, ..., s_i, ..., s_j, ..., s_k)$$

Mais alors si on supprime le cycle entre $s_i$ et $s_j$, on obtient

$$\mu' = (s_1, ..., s_{i-1}, z, s_{j+1}, ..., s_k)$$

avec $l(\mu') < l(\mu)$. On aboutit à une contradiction avec notre sélection initiale de $\mu$ comme étant un plus court chemin. □

La notion de pondération portée par la fonction $\omega$ nous permet de définir la notion de distance (coût) dans un graphe.

**Définition 6** (Distance, Coût)**.** *Soit $G = (V, E, \omega)$, un graphe orienté. La **distance** ou **coût** $d$ d'un sommet $x$ à un sommet $y$ est la somme des poids des arcs empruntés par un plus court chemin entre ces deux sommets. Autrement dit, si un plus court chemin de $x$ à $y$ est $\mu = (s_1, ..., s_n)$ avec $x = s_1$ et $y = s_n$, alors :* $d(x, y)$

$$d(x, y) = \sum_{i=1}^{n-1} \omega(s_i, si + 1)$$

*On étend cette définition en posant $d(x, y) = +\infty$ s'il n'existe pas de chemin entre $x$ et $y$.*

**Remarque 2.** *On fera tout de même l'hypothèse lors de nos implémentations qu'il existe un chemin entre la source $s$ et la cible $c$.*

Nous nous proposons alors d'étudier et d'implémenter un algorithme pionnier dans la recherche de chemin à moindre coût : l'algorithme de Dijkstra. Cependant, un choix judicieux doit être pris quant à la structure de données utilisée.

## 1.5 Numérisation concrète de la situation

Afin de numériser la situation réelle de manière optimale, nous nous sommes tournés vers les graphes orientés pondérés. Il reste cependant à choisir le type de structure de données concrète à mettre en place pour implémenter ces graphes. Deux options s'offrent à nous :

— Les matrices d'adjacence [3]
— Les listes d'adjacence

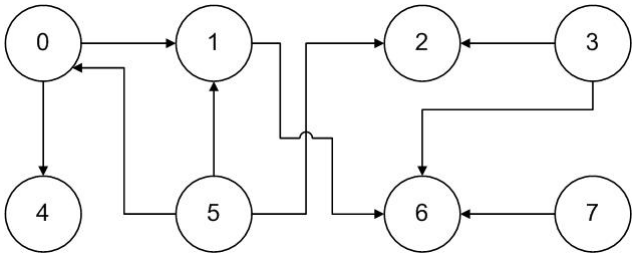
### 1.5.1 Matrices et listes d'adjacence

**Définition 7** (Matrice d'adjacence). *La matrice $M$ appelée **matrice d'adjacence de** $G$ est la matrice de $\mathcal{M}_n(\{0,1\})$ telle que : $\forall (i,j) \in \{1,...,n\}^2, m_{ij} = 1$ si $(i,j) \in E$ et $0$ sinon.* M

*En d'autres termes, il s'agit d'une matrice dont les lignes et les colonnes sont indexées par les sommets et dont les coefficients contiennent une valeur booléenne (soit 1, soit 0) indiquant l'existence ou l'absence d'arc entre les sommets correspondant à la ligne et la colonne du coefficient.*
*Par exemple, si $M \in \mathcal{M}_3(\{0,1\})$, $m_{23} = 1$ indique qu'il existe une arc allant du sommet 2 au sommet 3.*

**Définition 8** (Liste d'adjacence). *La liste d'adjacence `L` d'un graphe $G$ est une structure de données de type tableau, qui pour chaque sommet source de $G$, lui associe une liste contenant les sommets que l'on peut atteindre à partir de ce sommet source.* L
*La liste d'adjacence d'un graphe est donc numérisée par un tableau de tableaux.*

**Exemple 2.** *Prenons un exemple pour illustrer les 2 cas d'utilisation. Considérons le graphe $G = (V, E)$ suivant*[4] *:*




Ici, on a : $V = \{1,2,3,4,5,6,7\}$ et $E = \{(0,1),(0,4),(1,6),(3,2),(3,6),(5,0),(5,1),(5,2),(7,6)\}$, d'où $n = 7$ et $m = 9$.

Matrice d'adjacence de $G$ :

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Liste d'adjacence de $G$ :

```
0 : [1, 4]
1 : [6]
2 : []
3 : [1, 6]
4 : []
5 : [0, 1, 2]
6 : []
7 : [6]
```

**Remarque 3** (Analogie sur les matrices). *Dans le cas des graphes pondérés, la matrice d'adjacence sert également de matrice de coût (ou de distance) : les coefficients de la matrice représentent la présence mais également la valeur du coût d'un arc entre 2 sommet.*

**Remarque 4** (Analogie sur les listes). *Afin d'implémenter les coûts des arcs, les listes d'adjacence de chaque sommet seront composées des couples de la forme : (extrémité finale, coût de l'arc).*

### 1.5.2 Étude comparative des 2 solutions

Afin de trouver la structure la plus efficace, voici une liste complète des principaux avantages et inconvénients de ces 2 choix :

3. Il existe aussi les matrices d'incidence où un coefficient $m_{ij}$ représente un sommet pointé par les arrêtes $i$ et $j$. Cette solution n'étant pas pertinente, on ne s'y intéressera pas plus dans cet exposé.
4. Ce graphe est disponible sur Internet : https://fr.wikipedia.org/wiki/Matrice_d%27adjacence.

| | Matrice d'adjacence | Liste d'adjacence |
|---|---|---|
| Avantages | Simple : facile à mettre en oeuvre et à maintenir à jour<br>Vérification de l'existence d'un arc en temps constant : $O(1)$<br>Insertion en temps constant : $O(1)$ | Compacité très avantageuse (espace mémoire plus souple : $O(n+m)$<br>$\Rightarrow$ Privilégié pour faible densité de graphe<br>Parcours des successeurs : $O(d^+(x))$ |
| Inconvénients | Complexité spatiale : $O(n^2)$<br>Densité faible : gênant car on stocke des coefficients inutiles (les `INF` pour les absences d'arcs)<br>Parcours des prédécesseurs et successeurs d'un sommet : $O(n)$ | Existence des successeurs $\Rightarrow$ parcours des successeurs : $O(d^+(x))$<br>Parcours des prédécesseurs : $O(m)$ |
| Approche | Tourné vers les arcs (flux) | Tourné vers les sommets (ports) |

TABLE 1 – Étude comparative des 2 structures de données

**Remarque 5.** *Dans le cas des matrices d'adjacence, les coefficients nuls peuvent représenter des coûts nuls. La valeur 0 n'étant plus adéquate pour signaler l'absence d'arc entre 2 sommets, et par analogie avec la définition de $\omega$, on utilisera une valeur particulière notée* `INF`*. Pour l'implémenter, on utilisera soit une valeur inatteignable :* $2 << 63 - 1$ *ou bien des valeurs non numériques propre au langage de programmation :* `float("inf")`*, ou* `float("NaN")`*.*

On remarque que chacun de ces 2 modèles de données présente des avantages et des inconvénients quant à l'utilisation du graphe en question.

La liste d'adjacence privilégie une approche *plus centrée sur la gestion des sommets* tandis que la matrice d'adjacence sera plus *tournée vers les arcs* les reliant.

**Définition 9** (Densité d'un graphe)**.** *La densité $\Delta$ d'un graphe orienté $G = (V, E)$ est définie par :* $\Delta$

$$\Delta = \frac{|E|}{|V|^2} = \frac{m}{n^2}$$

*Elle est considérée comme faible si inférieure ou égale à 0.3.*

**Remarque 6** (Comparaison spatiale des 2 méthodes)**.** *Pour un graphe creux (c'est-à-dire dont la densité est faible), une liste d'adjacence semble être le choix le plus cohérent. En effet, il est beaucoup plus pertinent de ne pas stocker tous ces* `INF` *inutiles (représentant l'absence d'arcs dans le graphe). De plus la place requise par une liste d'adjacence est proportionnelle à $m+n$, alors que cette place est de l'ordre de $n^2$ pour le choix d'une matrice d'adjacence*[5]*. La représentation par liste d'adjacence est tolérable uniquement pour des graphes à faible densité.*

De plus, si on a pour ambition de simuler une large partition du trafic maritime mondial, le choix des matrices d'adjacence sera non viable pour la suite des opérations (relativement à sa forte complexité spatiale ainsi qu'à des contraintes temporelles développées ci-après).

**Exemple 3** (Application pratique)**.** [6] *Si on prend un nombre de ports égal à 1500 ($n = 1500$) et qu'on prend au maximum* 500000 *flux possibles, on obtient une densité $\Delta \simeq 0,22$. La densité étant faible, le choix de la liste d'adjacence semble se confirmer.*

5. Il est possible de diminuer ce coût spatial avec les tables de hachage : voir https://openclassrooms.com/fr/courses/1241676-les-tables-de-hachage.

6. L'examen des bases de données géographiques vendues dans le commerce montre clairement que les graphes routiers sous-jacents au commerce maritime ont des caractéristiques notables : grande dimension (entre 5000 et 30000 sommets), une faible densité (pour un sommet $i$ donné, le degré moyen $\overline{d(i)}$ est environ égal à 3. De plus, les coûts des flux occupent une plage de valeurs assez restreinte. *Ils sont très généralement quasi-symétriques et quasi-planaires à une échelle assez grande (régionale ou nationale).*

On ne privilégiera pas l'aspect temporel à l'aspect spatial de l'algorithme. En effet, la rapidité d'exécution est l'une des problématiques majeures dans notre situation ; le résultat doit être renvoyé rapidement à l'utilisateur afin d'ajuster au mieux sa prise de décision et limiter les prises de risques. De plus, le traitement des données doit s'opérer sans une grande dépendance de capacité spatiale. Le choix des listes d'adjacences en fait donc un bon compromis.

| Critères | Matrice d'adjacence | Liste d'adjacence |
|---|---|---|
| Complexité spatiale | $O(n^2)$ | $O(n+m)$ * |
| Accès à un sommet | $O(1)$ | $O(1)$ |
| Parcours des sommets | $O(n)$ | $O(n)$ |
| Parcours des arcs | $O(n^2)$ | $O(n+m)$ |
| Existence d'un arc | $O(1)$ | $O(d^+(x))$ ** |

TABLE 2 – Comparaison détaillée des complexités spatiales et temporelles

* On somme les longueurs des listes de la liste d'adjacence, d'où le résultat.
** L'existence d'un arc $(x, y)$ se traduit par $y$ dans `L[x]`.

# 2 Une première approche : l'algorithme de Dijkstra

*Jusqu'à la fin de l'exposé, $s$ représente le nœud source et $c$ le sommet cible.* $s$, $c$

## 2.1 Introduction

### 2.1.1 Méthode naïve

**Méthode brute** Afin de déterminer le chemin le moins coûteux, la méthode classique serait de tester tous les chemins possibles partant de $s$ vers $c$. Pour des valeurs convenables de $n$, cela marcherait mais la tâche serait très vite pénible pour un humain. La machine quant à elle peinerait à réaliser cette tâche pour de grandes valeurs de $n$. Cette méthode est donc à éviter car les complexités spatiales et temporelles de cette « méthode »seraient considérables ; on se tourne alors vers des algorithmes dits « gloutons ».

### 2.1.2 Description des situations

L'algorithme de Dijkstra est un algorithme itératif « glouton », opérant selon le principe suivant : à chaque passage dans la boucle, il fait le choix optimum local. Dans certains cas, cette approche permet d'atteindre un optimum global, mais dans la plupart des cas, il s'agit d'une heuristique [7]

Dans la recherche d'un plus court chemin entre deux sommets $s_i$ et $s_j$, trois cas peuvent se présenter :

1. Il n'existe pas de chemin de $s_i$ vers $s_j$
2. Il existe des chemins de $s_i$ vers $s_j$ mais pas de plus court chemin
3. Il existe un plus court chemin de $s_i$ vers $s_j$

**Définition 10.** *Un circuit de longueur négative est appelé* ***circuit absorbant****.*

**Proposition 1.** *Dans un graphe orienté pondéré $G = (V, E, \omega)$, il existe un plus court chemin entre tout couple de sommets si et seulement s'il n'existe pas de circuit absorbant dans $G$.*

*Démonstration.* Si un chemin entre deux sommets $x$ et $y$ possède un circuit absorbant, alors $\delta(x, y) = -\infty$.
En effet, dès que l'on parcourt un circuit absorbant, on diminue la distance du chemin. On peut donc diminuer ce coût à l'infini. En augmentant infiniment le nombre de tours dans le circuit, on obtient bien $\delta(x, y) = -\infty$. □

7. Heuristique : méthode de calcul aboutissant à une solution réalisable renvoyée en un temps acceptable. Cette solution n'est pas nécessairement optimale ou exacte.

### 2.1.3 Définition fondamentale

Il est important de préciser la distance d'un plus court chemin.

**Définition 11** (Coût d'un plus court chemin)**.** *Le **coût d'un plus court chemin** entre deux sommets $s_i$ et $s_j$, noté $\delta(s_i, s_j)$ est défini par :* $\delta$

$$\delta(s_i, s_j) = \min\{d(\mu) \mid \mu \text{ chemin de } s_i \text{ à } s_j\}$$

*De plus, $\delta(s_i, s_j) = +\infty$ s'il n'existe pas de chemin entre $s_i$ et $s_j$ (analogie avec la définition de la distance)*

Si $d$ est notre fonction d'évaluation de chaque sommet de $G$, on va chercher à la minimiser à chaque itération.

## 2.2 Étude fonctionnelle de l'algorithme de Dijkstra

### 2.2.1 Description de l'algorithme

**Définition 12.** *Le problème du chemin à moindre coût à origine unique $s$ à cible unique $c$ consiste à calculer le réel $\delta(s, c)$.*

A l'heure actuelle, il n'existe pas d'algorithmes permettant de déterminer **directement** cette information. Il faut donc calculer $\forall s_j \in V, \delta(s, s_j)$.

En plus de déterminer le coût du chemin déterminé numériquement, nous voudrons obtenir la composition du dit chemin, c'est-à-dire la liste ordonnée des sommets à emprunter. Pour cela, nous allons utiliser une structure arborescente notée $\Pi$ (`parents` en PYTHON) . $\Pi$

$$\forall s_j \in V, \Pi(s_j) = s_i$$

si $s_i s_j$ est un arc présent dans l'arborescence. ($s_i$ étant le prédécesseur de $s_j$) On remonte ainsi $\Pi$ à l'envers pour trouver la composition du plus court chemin de $s$ à $c$

**Proposition 2.** *L'algorithme de Dijkstra fonctionne avec des valuations positives ou nulles **uniquement**, d'où : $\omega : E \longrightarrow \mathbb{R}^+ \cup \{+\infty\}$.*

*Démonstration.* L'utilisation de valuations négatives entraine, dans la plupart des cas, la création de circuits absorbants, tronquant ainsi les résultats. □

L'idée centrale de l'algorithme de Dijkstra est d'associer à chaque sommet $s_i \in V$, une valeur $d(s_i)$ stockée dans une liste `distances`. Ainsi, `distances[s_i]` représente le coût du plus court chemin entre $s_0 = s$ et $s_i$ (c'est-à-dire $\delta(s_0, s_i)$).
A l'initialisation :

- `distances[s_0]` = `distances[s]` = 0 car $\delta(s_0, s_0) = 0$ (par convention)
- $\forall s_i \in V - \{s_0\}$ , `distances[s_i]` = $+\infty \gg \delta(s_0, s_i)$

Au fur et à mesure de la progression de l'algorithme, ce dernier diminue automatiquement les valeurs de `distances[s_i]` jusqu'à ce qu'on ne puisse plus les diminuer, c'est-à-dire jusqu'à l'étape où on obtient `distances[s_i]` $= \delta(s_0, s_i)$.

Pour minimiser les valeurs de `distances`, on va itérativement examiner chaque sommet $s_j \in \Gamma^+(s_i)$ (successeurs de $s_i$) et voir si on ne peut pas améliorer `distances[s_j]` en passant par $s_i$. Cette étape est appelée : le **relâchement** de l'arc $s_i s_j$.

Afin de coordonner ces phases de relâchement, on doit opérer sur deux ensembles de sommets maintenus **disjoints** : $O \cup F = V$. $O$, $F$

- $O$ pour liste ouverte : contient les sommets à examiner
- $F$ pour liste fermée : contient les sommets dont on connait le PCC.

A chaque étape de l'algorithme, chaque sommet est dans un état : soit en cours d'examen (transféré de $O$ vers $F$), soit examiné (dans $F$), soit en attente d'examen (dans $O$ ou pas).

Reste à implémenter ces ensembles en PYTHON. La première idée est la suivante :

— `liste_ouverte` : tableau mutable de type `list`
— `liste_fermee` : tableau mutable de type `list`
— `distances` : un dictionnaire où chaque clé représente un sommet $i$ et sa valeur correspondante, $d(i)$
— `parents` : un dictionnaire où chaque clé représente un sommet $i$ et sa valeur correspondante, le successeur de $i$ dans $G$. (c'est la « numérisation » de l'arborescence $\Pi$)

La liste d'adjacence est représenté numériquement avec un dictionnaire [8]

```
G = {
    'a': [('b', 50), ('c', 60)],
    'b': [('d', 74), ('b', 140)],
    'c': [('d', 20)]
}
```

L'algorithme de DIJKSTRAfonctionne en suivant cette chronologie :

1. **Initialisation** On créé les listes ouverte, fermée et les dictionnaires de distances et des parents. On créé la liste vide du chemin final. On initialise la distance de la source à la source à 0, les autres distances infinies, les parents de chaque sommet étant eux-mêmes. La liste ouverte est initialisée à la source.

   *On boucle jusqu'à ce que la liste ouverte soit vide ou bien que le sommet en traitement soit la cible.*

2. **Sélection du sommet courant** On sélectionne le sommet $i$ de la liste ouverte ayant la distance minimale à la source. Autrement dit, on prend le sommet correspondant au minimum de $d$ restreint à la liste ouverte.

   *Si le sommet courant est le sommet cible, on sort de la boucle et on construit le chemin.*

3. **Transfert du sommet courant** On supprime le sommet courant de la liste ouverte et on l'insère dans la liste fermée.

4. **Relâchement des sommets successeurs** Pour chaque successeur du sommet courant, s'il a déjà était examiné, on passe au suivant sinon : si le successeur est dans la liste ouverte, on calcule le nouveau coût : $d(i)+\omega(i,j)$. Si le nouveau coût est inférieur à $d(j)$, alors on actualise le coût du successeur et son parent. S'il n'est pas dans la liste ouverte, on effectue le même protocole en l'ajoutant dans la liste ouverte.

5. **Construction du chemin** A la sortie de la boucle, on reconstruit le chemin. On part de la cible, on parcourt son parent, puis le parent du parent, jusqu'à arriver sur la source. On stocke ces sommets dans la liste `chemin` créée lors de l'initialisation et on renvoie cette liste ainsi que le coût global de l'itinéraire `distances[cible]` et le nombre de pts de cheminements `len(chemin)`.

### 2.2.2 Implémentation en Python

Afin de rendre l'implémentation plus compréhensible, elle est découpée en sous-fonction :

#### Fonction d'initialisation

```
def initialisation(G, source):
    """ Fonction d'initialisation """
    INF = float("inf")
    liste_ouverte = [source]
    liste_fermee = []
    distances = {sommet: INF for sommet in G}
    parents = {sommet: sommet for sommet in G}
```

8. Les dictionnaires se comportent comme les listes (même complexité pour les opérations élémentaires)

```
    chemin = []
    distances[source] = 0
    return liste_ouverte, liste_fermee, distances, parents, chemin
```

### Fonction de sélection du sommet courant

```
def selection_sommet_courant(liste_ouverte, distances):
    """ Fonction de sélection du sommet courant """
    sommet_courant = liste_ouverte[0]
    for k in range(0, len(liste_ouverte)):
        if distances[liste_ouverte[k]] < distances[sommet_courant]:
            sommet_courant = liste_ouverte[k]
    return sommet_courant
```

### Fonction de sélection des successeurs du sommet courant

```
def successeurs_sommet_courant(G, sommet_courant):
    """ Fonction retournant les successeurs du sommet courant dans G """
    return G[sommet_courant]
```

### Fonction de relâchement des sommets successeurs

```
def relachement_sommet(sommet_courant, s, liste_ouverte, liste_fermee, distances, parents
    ):
    """ Procédure de relâchement du sommet """
    nouveau_cout = distances[sommet_courant] + s[1]
    if s[0] in liste_ouverte:
        if nouveau_cout < distances[s[0]]:
            distances[s[0]] = nouveau_cout
            parents[s[0]] = sommet_courant
    else:
        liste_ouverte.append(s[0])
        distances[s[0]] = nouveau_cout
        parents[s[0]] = sommet_courant
```

### Fonction de construction du chemin

```
def construction_chemin(source, cible, liste_ouverte, parents, chemin):
    """ Fonction de construction du chemin trouvé """
    if len(liste_ouverte) == 0:
        return []
    else:
        n = cible
        while n != source:
            chemin.append(n)
            n = parents[n]
        chemin.append(source)
        chemin.reverse()
        return chemin
```

### Corps de l'implémentation

```
def dijkstra(G, source, cible):
    """ Fonction implémentant l'algorithme de Dijkstra """
    assert source != cible

    # Initialisation
    liste_ouverte, liste_fermee, distances, parents, chemin = initialisation(G, source)
    while len(liste_ouverte) > 0:
        # Sélection du sommet
        sommet_courant = selection_sommet_courant(liste_ouverte, distances)

```

```
11          if sommet_courant == cible:
12              break
13          # Transfert du sommet
14          liste_ouverte.remove(sommet_courant)
15          liste_fermee.append(sommet_courant)
16
17          # Etape de relâchement de chaque successeur du sommet courant
18          for s in successeurs_sommet_courant(G, sommet_courant):
19              if s[0] in liste_fermee:
20                  continue # passer au suivant
21
22              relachement_sommet(sommet_courant, s, liste_ouverte, liste_fermee, distances,
                    parents)
23
24      # renvoyer le chemin trouvé
25      return construction_chemin(source, cible, liste_ouverte, parents, chemin), distances[
            cible]
```

## 2.3 Exemples d'exécution

On a donc notre première version de l'algorithme, qui à partir d'un sommet source $s$ et d'un sommet cible $c$ donnés, peut déterminer et renvoyer le chemin le moins coûteux avec son coût correspondant.

**Exemples d'utilisation** Voici la trace d'exécution de `dijkstra` pour le digraphe $G = (V, E, \omega)$ suivant.

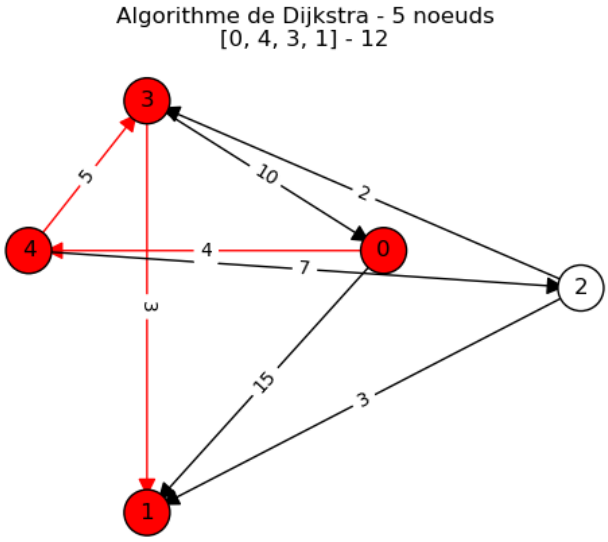



FIGURE 3 – Exemple d'exécution sur $G_1$ (sortie de l'algorithme)

## 2.4 Étude théorique de l'algorithme

### 2.4.1 Complexité

**Théorème 1** (Complexité de l'algorithme de Dijkstra)**.** *L'algorithme de Dijkstra possède une complexité temporelle $O(n^2)$ (quadratique) et une complexité spatiale en $O(n+m)$.*

*Démonstration.* $O$ est implémenté par une liste linéaire.

La déclaration des variables nécessite 4 variables (un compteur et 3 autres de contrôle), 5 tableaux et la liste d'adjacence de $G$. On a donc une complexité temporelle linéaire : $O(n)$ et spatiale linéaire également : $O(n+m)$.

L'initialisation est en $O(n)$.

Au niveau du traitement, l'étape de recherche du minimum consiste à parcourir un tableau jusqu'à trouver sa plus petite valeur. Elle est donc réalisée en temps linaire : $O(n)$.

A chaque itération, il faut chercher le sommet $s_i$ de $O$ ayant la plus petite valeur de $d$. Étant donné qu'il y a au maximum $n$ itérations, qu'au premier passage $O$ contient au plus $n-1$ éléments et qu'à chaque itération, on lui ôte un seul sommet, le nombre total d'opérations est :

$$\sum_{k=1}^{n} k = \frac{(n-1)(n-2)}{2} \sim_{+\infty} n^2$$

De plus, le relâchement de chaque arc étant unique, il y a $m$ relâchements d'où une complexité temporelle en $O(m)$ pour cette étape.

Le coût global des opérations sur le graphe $G = (V, E)$ est donc :

$$O(n^2 + n + m) = O(n^2)$$

. □

Ce résultat est cohérent avec notre implémentation. Cependant, il n'est guère satisfaisant. En effet, pour de grandes valeurs de $n$ (nombre de sommets), le temps d'exécution devient très vite considérable, impliquant des temps de réponse intolérables pour des utilisations spécifiques.

### 2.4.2 Correction & Terminaison

### 2.4.3 Avantages et inconvénients

On doit donc se tourner vers d'autres alternatives (abordées plus tard).

**Avantages** Cet algorithme permet d'obtenir les résultats attendus, cohérents avec la réalité et juste. Il permet de plus, de déterminer le chemin à prendre, facilitant la prise de décision du navire.
L'ajout de contraintes d'acteurs étant facile à mettre en place (en manipulant les pondérations), cette solution est envisageable pour de petits réseaux maritimes comportant quelques nœuds.

**Inconvénients** Cependant, si nous avons pour ambition de simuler une grande partition du trafic maritime mondial, cet algorithme, tel qu'il est implémenté ne semble pas être un choix pertinent :

- Comme énoncé précédemment, la fonction de pondération doit être à valeurs positives, c'est-à-dire que $\omega$ est à valeur dans $\mathbb{K} \subset \mathbb{R}^+$. Autrement dit, on peut affirmer que $\omega$ est monotone croissante (en ajoutant des poids positifs ou nuls, le coût d'un chemin augmente en fonction de sa longueur). Une solution à cette restriction serait d'implémenter l'algorithme de Bellman-Ford [9] supprimant cette contrainte.
- L'efficacité de l'algorithme de DIJKSTRArepose sur *une mise en œuvre efficace de la procédure chargée de trouver le sommet à traiter en priorité*. Notre implémentation utilisant plusieurs listes semble être trop coûteuse en temps et gourmande en espace mémoire.

L'objectif serait donc d'améliorer cette étape en supprimant des étapes inutiles et en accélérant le traitement des sommets.

Pour cela, nous allons nous orienter vers d'autres solutions de structure de données : les files de priorité et leurs implémentations concrètes.

9. Algorithme de Bellman-Ford : voir https://fr.wikipedia.org/wiki/Algorithme_de_Bellman-Ford. Il permet, en plus de déterminer un PCC, la détection automatique de circuits absorbants d'un digraphe donné.

# 3 Une approche optimale : l'algorithme A*

## 3.1 Introduction

L'un des avantages de l'algorithme de DIJKSTRAest qu'il détermine en un temps convenable, pour la plupart des petites situations, le chemin le moins coûteux. Cependant, la fonction d'évaluation de l'algorithme, notée $d$ est la fonction qui a un sommet $i$ associe la distance de la source s au sommet $i$.

Autrement dit, à chaque étape, on détermine l'action la plus optimale en se concentrant sur la distance source - sommet courant.

Afin de raffiner cette méthode, on peut modifier cette fonction afin de rendre l'optimalité de la sélection du noeud encore meilleure.

Pour cela, nous allons étudier l'algorithme A*(prononcé « A star »). Ce dernier possède un fonctionnement analogue à celui de DIJKSTRA, à l'exception de sa fonction d'évaluation.

## 3.2 Description de l'algorithme A*

### 3.2.1 Fonction d'évaluation

L'algorithme attribue à chaque sommet du graphe : un coût $g$ et un coût $h$. Le coût global $f$ est la fonction d'évaluation de A*. Il résulte de la somme de $g$ et $h$.

$$\begin{array}{rccl} f & : & V & \to & \mathbb{R}^+ \\ & & n & \mapsto & f(n) = g(n) + h(n) \end{array}$$

— $g$ est équivalente à $d$ : il s'agit de la distance entre $s$ et $n \in V$.
— $h$ mesure la distance entre un sommet $n$ et la cible $c$

A chaque étape de l'algorithme, on vérifie ainsi la distance par rapport au départ du parcours, mais également la distance restante avant d'arriver à la cible.

### 3.2.2 Modification de la représentation (classes `Nœud` et `Grille`)

Pour implémenter au mieux cette nouvelle solution, on fait le choix de discrétiser le graphe orienté pondéré sous forme de maille carrée de dimension $N \times N$ où chaque case représente un nœud. Afin d'effectuer un repérage sur la maille, on le munit d'un système de coordonnées cartésiennes $(x, y)$. Chaque nœud (sommet) de la maille $G$ [10] possède ainsi un couple de coordonnées **qui lui est propre**.

PHOTO REPÉRE + MAILLE VIERGE

Chaque nœud $n$ possède différentes propriétés :
— des coordonnées cartésiennes $(x, y)$
— un poids $w$
— un booléen *walkable* indiquant si le nœud est « empruntable » ou si c'est un obstacle
— $g$ : distance entre $s$ et $n$
— $h$ : distance évaluée par une heuristique entre $n$ et $c$
— $f$ : somme de $g$ et $h$
— `parent` : nœud parent de $n$ dans la composition du PCC

Pour créer des nœuds, on va utiliser le paradigme de la programmation orienté objet (POO) permettant de créer des instances d'objets avec plusieurs propriétés et plusieurs méthodes :

```
1 class Noeud:
2     def __init__(self, x, y, w):
3         self.x = x
4         self.y = y
5         self.w = w
6         self.walkable = 0
7         if self.w != -1: # si le poids est -1, alors le noeud est un obstacle.
8             self.walkable = 1
9
```

10. Désormais, on aura $V = [\![0, N-1]\!]^2$

```
10          self.g = 0
11          self.h = 0
12          self.f = 0
13          self.parent = self # initialisé à lui-même
```

On peut ainsi créé un nœud comme ceci : `noeud1 = Noeud(10, 25, 50)`.

### 3.2.3 Heuristiques

Arrive maintenant une problématique cruciale : comment calculer le coût $h$ ? Cela amène à une discussion préliminaire : chaque sommet est adjacent à plusieurs autres nœuds. La question est de savoir combien. Le nombre de flux par nœud dépend du système de directions envisageables par l'algorithme. Une restriction au système *N - S - E - O* ne serait pas trop cohérent. Un navire peut se mouvoir dans plus de directions. Pour une grande précision, nous allons nous orienter vers un système *N - NE - E - SE - S - SO - O - NO*.

**Remarque 7.** *Nous verrons également, par la suite, une solution, supprimant toute restriction de direction. On pourra projeter un itinéraire suivant des directions inscrites dans* $[0, 2\pi]$.

Ainsi, pour un nœud intérieur de la grille : il possède 8 nœuds voisins (voisinage de 8), pour un noeud sur un bord, il en possède 5 et enfin, un coin 3.

Pour chaque nœud, on pourrait les implémenter en utilisant plusieurs listes d'attributs ou des dictionnaires. Cependant, on va utiliser une matrice de taille $N \times N$ où les coefficients seront des objets PYTHONde classe `Nœud`. On pourra alors très aisément accéder aux attributs de chaque nœud et les modifier.

Ainsi, vu que la grille est assimilable comme $\mathbb{R}^2$, on va pouvoir définir la notion de distance dans sa base orthonormée $(\overrightarrow{i}, \overrightarrow{j})$. Soit $n$ et $n'$, 2 nœuds distincts de la maille ayant comme coordonnées respectives : $(x, y)$ et $(x', y')$
On défini ainsi 2 distances suivant 2 cas :

**Définition 13** (Distance de Manhattan)**.** $\boxed{d(n, n') = |x - x'| + |y - y'|}$

**Définition 14** (Distance euclidienne)**.** $\boxed{d(n, n') = \sqrt{(x - x')^2 + (y - y')^2}}$

La distance de Manhattan est utilisée pour un déplacement suivant 8 directions, tandis que la distance euclidienne sera employée pour des déplacements dans toutes les directions
Ces distances constitueront les heuristiques de notre algorithme. Nous avons ainsi l'expression de $g$ et $h$. Nous pouvons donc aborder l'étape de relâchement, encore analogue à l'algorithme de DIJKSTRA.

### 3.2.4 Étape de relâchement

L'étape de relâchement est sensiblement la même à l'exception du nouveau coût noté $\phi$. Si $n$ est le nœud courant, et $j$ chacun de ces voisins,

— le nouveau coût $g(j)$ est égal au coût $g$ du parent du $j$
— le nouveau coût $h(j)$ est égale à la distance entre $j$ et $c$
— la nouvelle évaluation $f(j) = g(j) + h(j)$

Ceci est donc l'étape de relâchement de l'algorithme A*≪ classique ≫. Cependant, vu que nous travaillons avec des valuations par nœud, sur une grille à pondération non uniforme (chaque nœud a un poids différent) , il faut inclure la valuation dans le nouveau coût $\phi$, d'où :

$$\phi(j) = f(j) + \alpha \times w(j)$$

avec $\alpha \in [0, 1]$, **le coefficient d'amortissement**. C'est un nombre arbitraire fixé, permettant d'atténuer ou d'accentuer l'effet de la pondération sur l'évaluation des nœuds.

**Remarque 8.** *Ce nombre est assez difficile à fixer par le calcul. Sa valeur optimale a été trouvée* ***empiriquement*** *en exécutant l'algorithme sur plusieurs situations.*

**Remarque 9** (Manipulation de la distance euclidienne)**.** *On préférera manipuler le carré de la distance euclidienne afin d'opérer continuellement sur des entiers et non des flottants.*

### 3.2.5 Implémentation en Python

**Fonction de création de la matrice servant de base à la grille** Afin de créer une matrice avec des coefficients de type `Noeud`, on utilise cette fonction :

```
def creation_P(n, m, p):
    M = [[None for j in range(m)] for i in range(n)]
    for i in range(n):
        for j in range(m):
            if bernoulli(p) == 1:
                M[i][j] = rd.randint(1, 100)
            else:
                M[i][j] = -1
    return M
```

La fonction `bernoulli(p)` permet de remplir aléatoirement la grille avec $p$, la probabilité d'avoir une case empruntable.

**Classe `Grille` avec les méthodes de détermination des voisinages**

```
class Grille:
    def __init__(self, M):
        global NaN
        self.M = M
        self.n = len(M)
        self.m = len(M[0])
        self.grille = [[None for j in range(self.m)] for i in range(self.n)]
        for i in range(self.n):
            for j in range(self.m):
                self.grille[i][j] = NoeudPondere(i, j, M[i][j])

    def voisinage_4directions(self, noeud):
        noeud_x = noeud.x
        noeud_y = noeud.y
        voisins = []
        X = noeud_x - 1
        Y = noeud_y
        if (X >= 0 and X < self.n) and (Y >= 0 and Y < self.m):
            voisins.append(self.grille[X][Y])
        X = noeud_x + 1
        Y = noeud_y
        if (X >= 0 and X < self.n) and (Y >= 0 and Y < self.m):
            voisins.append(self.grille[X][Y])
        X = noeud_x
        Y = noeud_y - 1
        if (X >= 0 and X < self.n) and (Y >= 0 and Y < self.m):
            voisins.append(self.grille[X][Y])
        X = noeud_x
        Y = noeud_y + 1
        if (X >= 0 and X < self.n) and (Y >= 0 and Y < self.m):
            voisins.append(self.grille[X][Y])
        return voisins

    def voisinage_8directions(self, noeud):
        noeud_x = noeud.x
        noeud_y = noeud.y
        voisins = []
        for x in [-1, 0, 1]:
            for y in [-1, 0, 1]:
                if x == 0 and y == 0:
                    continue
                X = noeud_x + x
                Y = noeud_y + y
                if (X >= 0 and X < self.n) and (Y >= 0 and Y < self.m):
                    voisins.append(self.grille[X][Y])
        return voisins
```

### Sélection du noeud courant

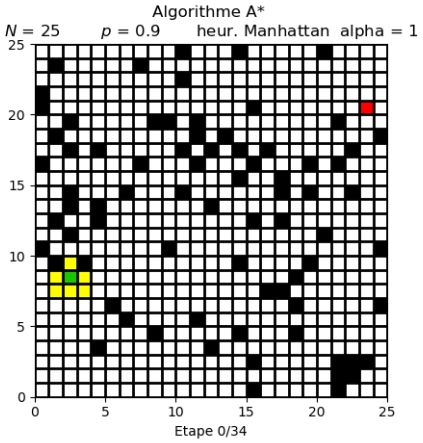
```
1  def heuristique_manhattan(noeudA, noeudB):
2      """ Fonction heuristique retournant la distance de Manhattan entre 2 objets de type
           noeud """
3      Ax, Ay = noeudA.x, noeudA.y
4      Bx, By = noeudB.x, noeudB.y
5      dist = abs(Bx - Ax) + abs(By - Ay)
6      return dist
7
8  def heuristique_euclidienne(noeudA, noeudB):
9      """ Fonction heuristique retournant la distance d'euclidienne entre 2 objets de type
           noeud """
10     Ax, Ay = noeudA.x, noeudA.y
11     Bx, By = noeudB.x, noeudB.y
12     dist = (Bx - Ax)**2 + (By - Ay)**2
13     return math.sqrt(dist)
14
15 def getCurrentNode(LISTE_OUVERTE):
16     """ Fonction renvoyant le noeud à traiter en priorité """
17     CURRENT = LISTE_OUVERTE[0]
18     for k in range(0, len(LISTE_OUVERTE)):
19         if LISTE_OUVERTE[k].f < CURRENT.f:
20             CURRENT = LISTE_OUVERTE[k]
21     return CURRENT
```

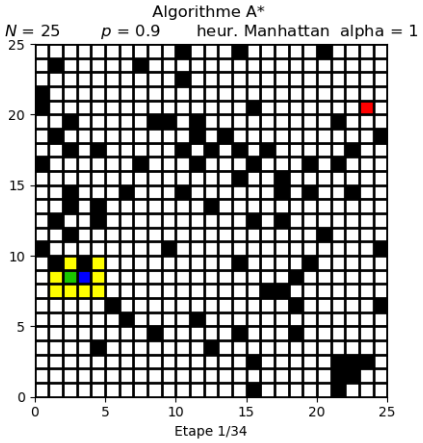
### Algorithme A*

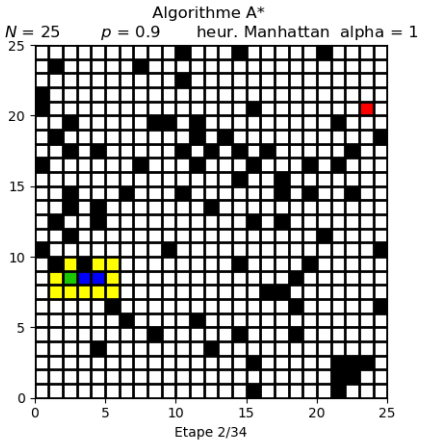
```
1  def algorithme_AP(G, s, c, alpha = 1):
2      """
3      Fonction appliquant l'algorithme A* à une grille G
4      en partant d'un noeud de départ s pour arriver à un noeud d'arrivée c
5
6      Entrées:
7          G: grille (objet de type Grille)
8          s: noeud de départ (objet de type Noeud)
9          c: noeud d'arrivée (objet de type Noeud)
10
11     """
12     # Déclaration des variables
13     LISTE_OUVERTE = []
14     LISTE_FERMEE  = []
15     CHEMIN_FINAL = []
16
17     source = s
18     cible = c
19     CURRENT = source
20     LISTE_OUVERTE.append(source)
21
22     # Tant que LISTE_OUVERTE n'est pas vide
23     while len(LISTE_OUVERTE) > 0:
24         CURRENT = getCurrentNode(LISTE_OUVERTE)
25         if CURRENT == cible:
26             break
27         LISTE_OUVERTE.remove(CURRENT)
28         LISTE_FERMEE.append(CURRENT)
29         voisins_CURRENT = G.voisinage_8directions(CURRENT)
30         for voisin in voisins_CURRENT:
31             if voisin in LISTE_FERMEE or voisin.walkable == 0:
32                 continue
33             new_g = voisin.parent.g
34             new_h = heuristique_euclidienne(cible, voisin)
35             new_f = new_g + new_h + alpha * voisin.w
36             if voisin in LISTE_OUVERTE:
37                 if new_g < voisin.g:
```

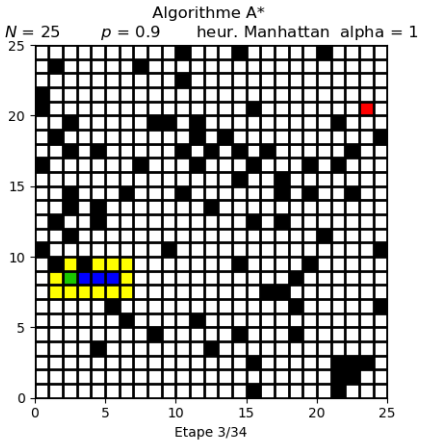
```
38                      voisin.parent = CURRENT
39                      voisin.g = new_g
40                      voisin.h = new_h
41                      voisin.f = new_f
42              else:
43                  LISTE_OUVERTE.append(voisin)
44                  voisin.parent = CURRENT
45                  voisin.g = new_g
46                  voisin.h = new_h
47                  voisin.f = new_f
48
49      if len(LISTE_OUVERTE) == 0:
50          return CHEMIN_FINAL
51      n = cible
52      while n != source:
53          CHEMIN_FINAL.append(n)
54          n = n.parent
55      CHEMIN_FINAL.append(source)
56      CHEMIN_FINAL.reverse()
57      return CHEMIN_FINAL
```

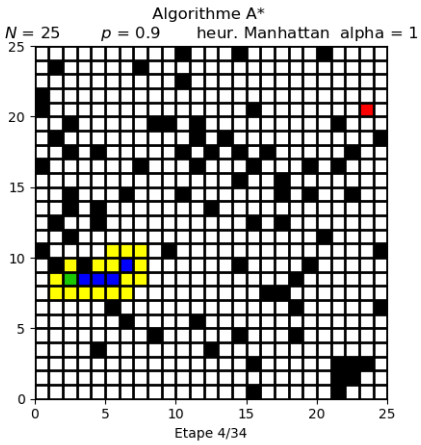
### Exemple d'exécution

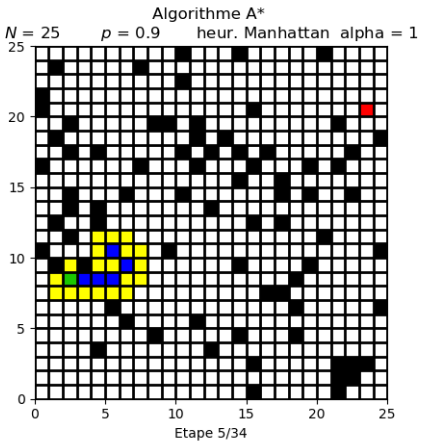


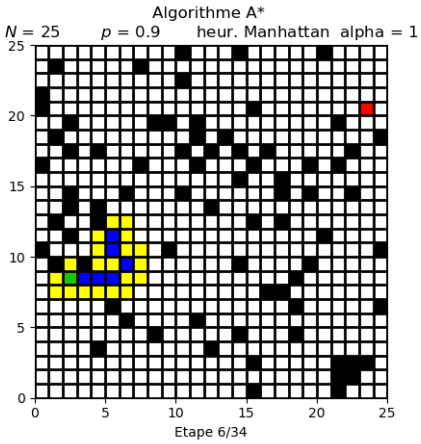



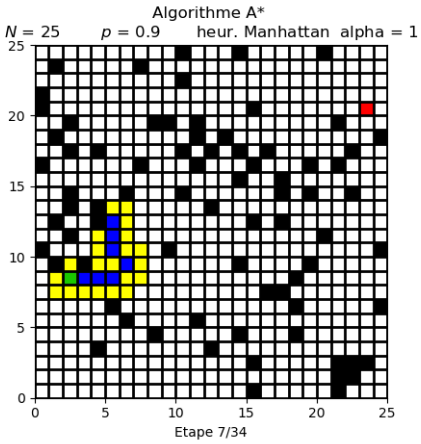



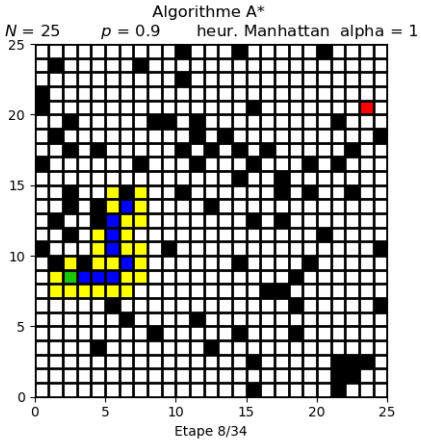



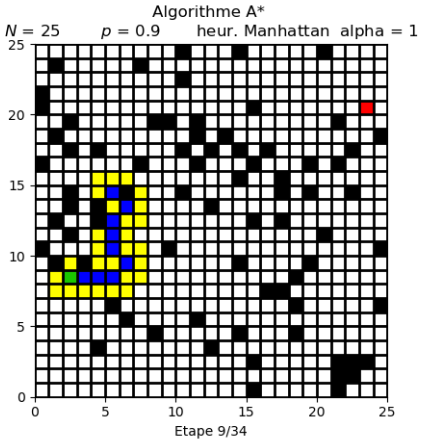

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 4/34

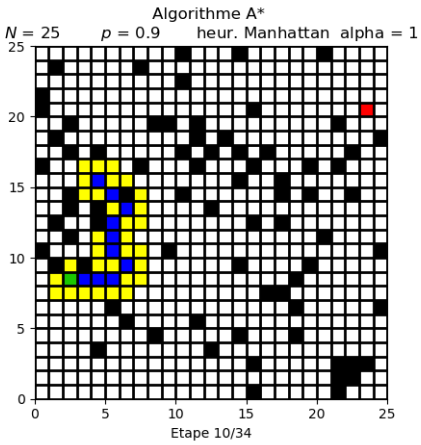

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 5/34

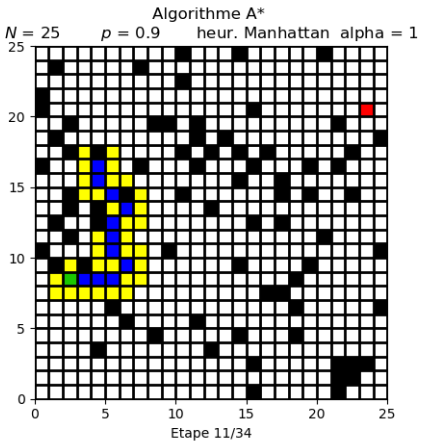

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 6/34

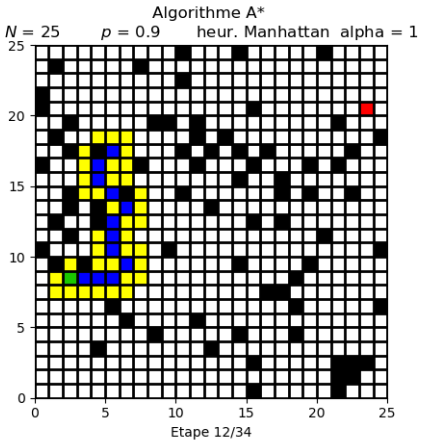

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 7/34

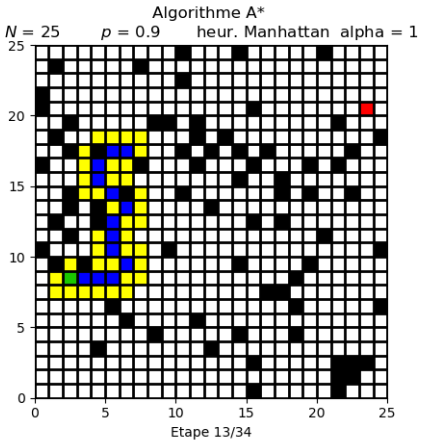

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 8/34

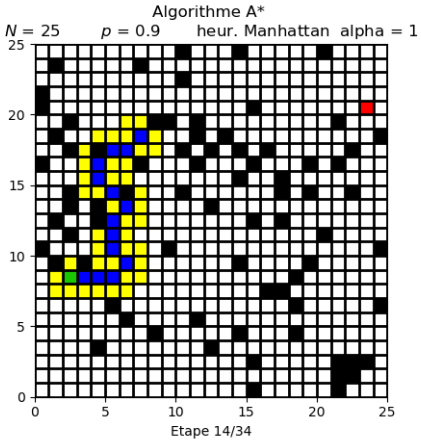

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 9/34

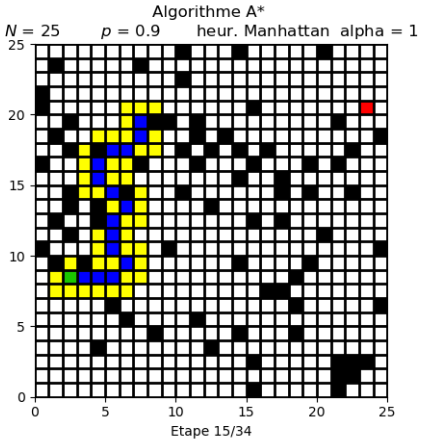

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 10/34

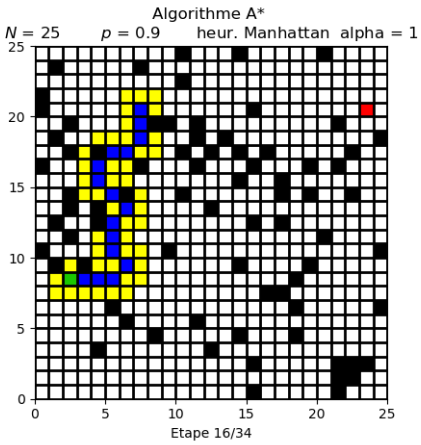

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 11/34

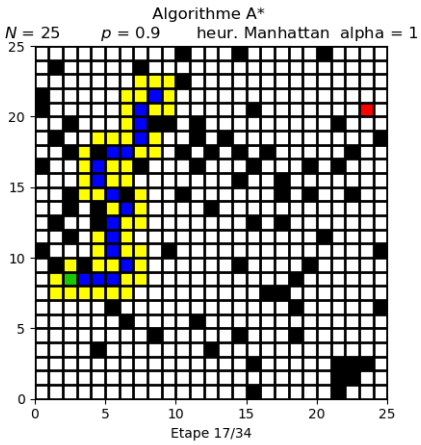

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 12/34

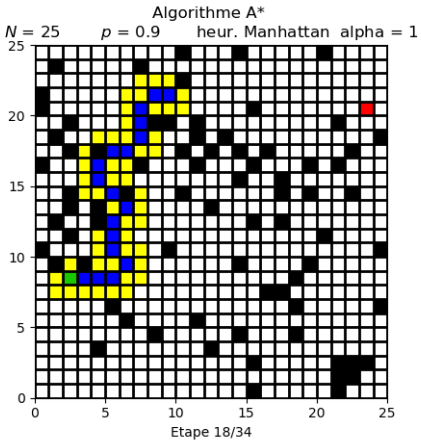

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 13/34

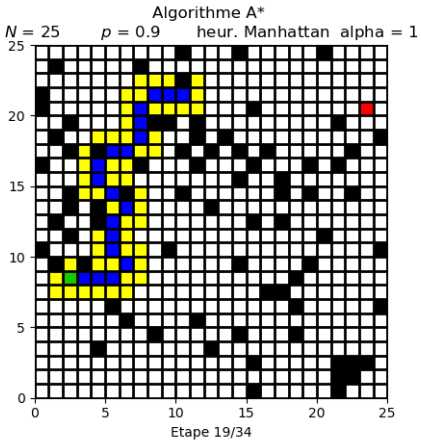

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 14/34

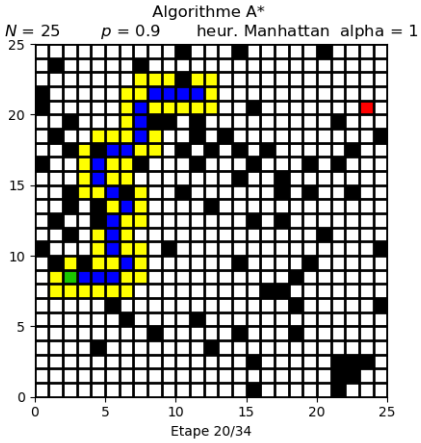

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 15/34

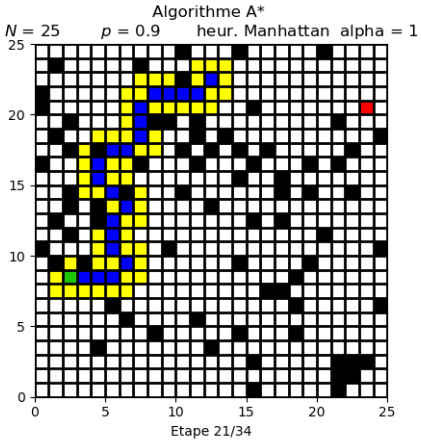

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 16/34

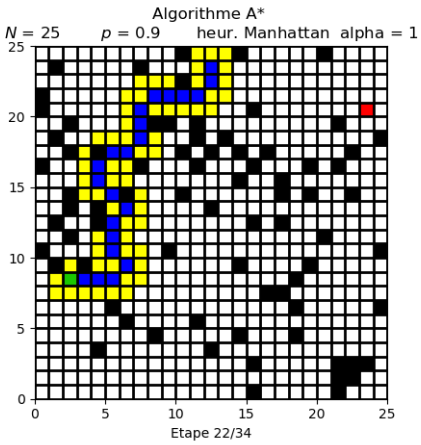

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 17/34

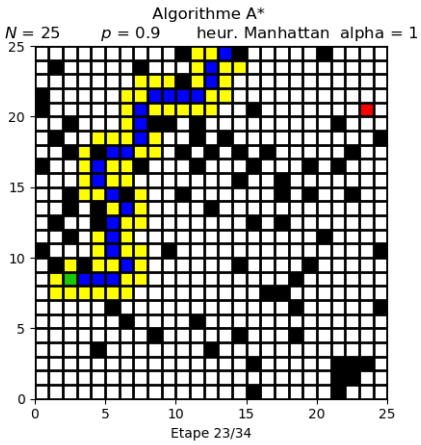

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 18/34

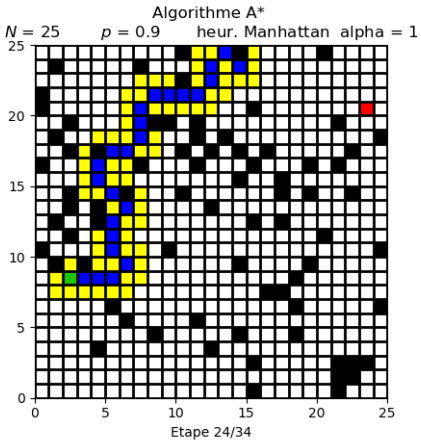

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 19/34

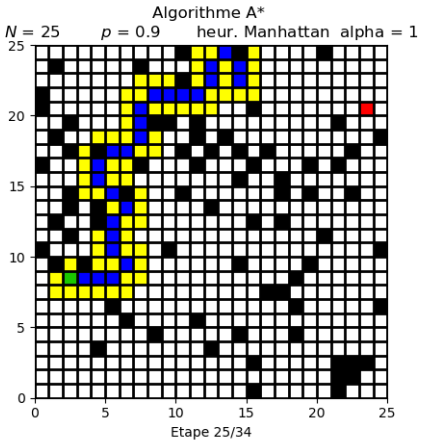

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 20/34

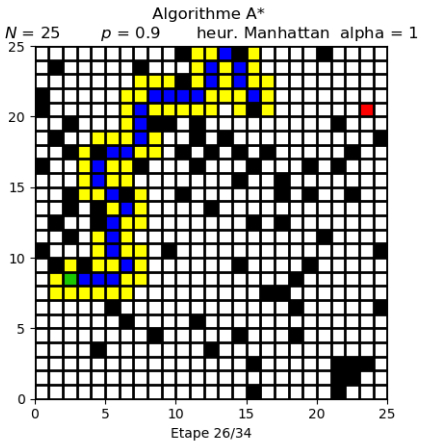

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 21/34

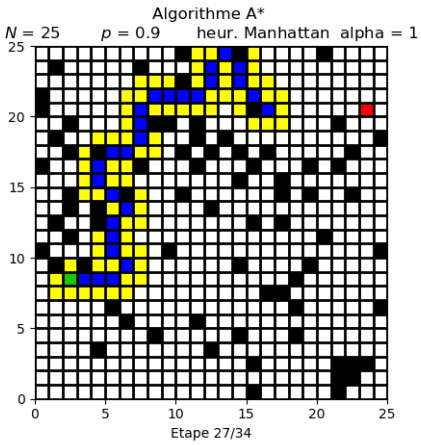

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 22/34

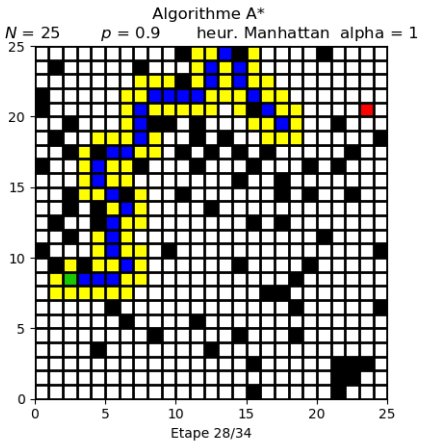

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 23/34

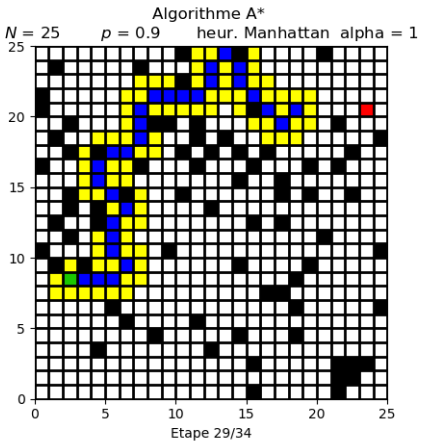

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 24/34

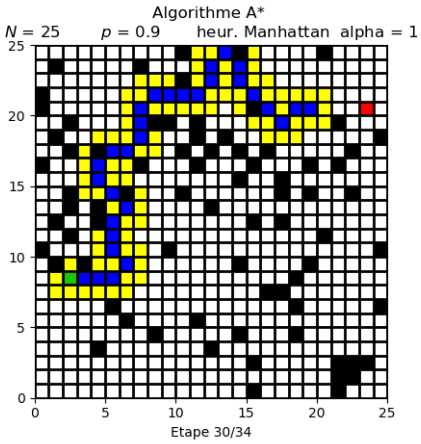

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 25/34

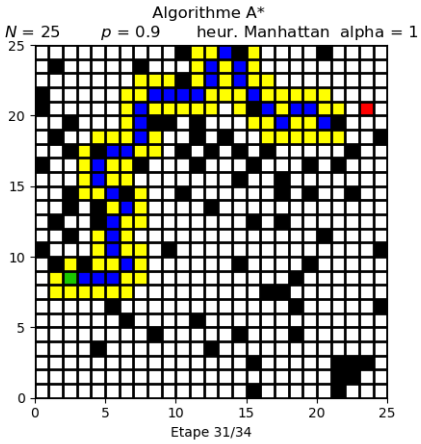

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 26/34

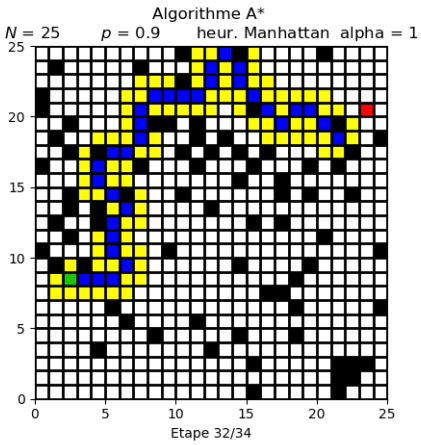

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 27/34

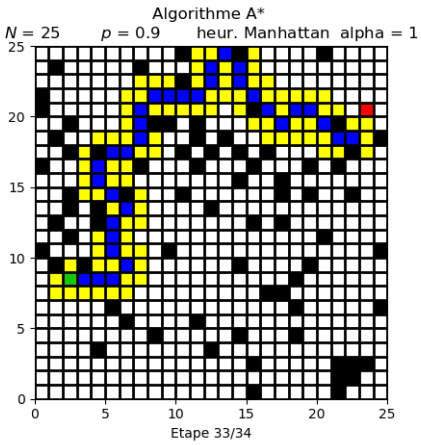

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 28/34

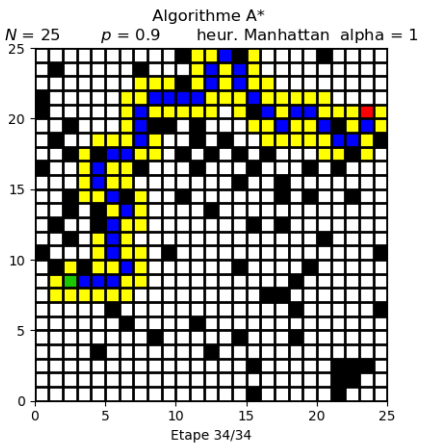

Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 29/34


Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 30/34


Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 31/34


Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 32/34


Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 33/34


Algorithme A*
N = 25 p = 0.9 heur. Manhattan alpha = 1
Etape 34/34

# 4 Amélioration de la méthode de sélection des nœuds

## 4.1 Introduction

*Tout d'abord, généralisons le concept de priorité.*

**Définition 15** (Priorité)**.** *Le sommet possédant la priorité maximale, dans notre situation, est celui qui est à la distance minimale de la source. Autrement dit, le sommet courant, qui sera sélectionné à chaque itération, est celui qui possède la priorité maximale.*

Afin de gérer au mieux les priorités, on va étudier les files de priorité.

## 4.2 Files et files de priorité

### 4.2.1 Files

Une **file** est une structure de données linéaire, dynamique et mutable (on peut modifier sa taille lors du traitement). Son fonctionnement est analogue à celui d'une file d'attente. Les premières entités sont rangées en premier dans la file (représentée généralement horizontalement) ; les suivantes se placent derrière les premières arrivées. Les premières à passer au traitement sont donc les premiers arrivés, d'où le nom « FIFO »(pour « First In, First Out »).

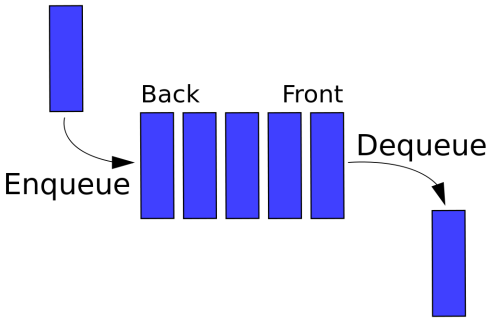



FIGURE 4 – Schéma du fonctionnement d'une file (de la gauche vers la droite)

Cette représentation linéaire est très utile et permet de mieux cibler le problème de détermination du coût minimal. Cependant, dans sa forme générale (comme énoncée ci-dessus), elle n'est pas pertinente : l'ordre se fait selon les indices (la file retire le premier élément localisable par le premier pointeur, puis le second, etc). Notre problème ,quant à lui, de retourner les sommets dans l'ordre croissant des coûts minimaux. La relation d'ordre **inférieur ou égal** ($\leq$) s'applique donc sur la valeur des coûts des sommets présents dans la file et non sur leur ordre d'entrée/sortie dans la file.

Ce comportement peut être implémenté avec une structure de données dérivée des files : les **files de priorité**.

### 4.2.2 Files de priorité

Une **file de priorité** [11] est un type abstrait de données opérant sur un ensemble ordonné et muni des opérations primitives suivantes :

— Créer une file de priorité (constructeur)
— Insérer un élément (couple : identifiant/nom/valeur de l'élément et sa priorité respective)
— Retirer l'élément ayant la priorité la plus forte
— Tester si une file est vide

11. Les files de priorité sont utilisées dans plusieurs secteurs : ordonnancement des tâches d'un système complexe, application financière, contrôles maritimes et aériens.

Cette dernière opération est primordiale pour garantir l'arrêt du traitement !
Dans notre contexte, on ajoute à la liste des primitives, l'opération « Augmenter/Diminuer la plus grande/petite priorité (clé) ».

**Remarque 10.** *Les files de priorité sont donc très proches des files (et même des piles) de part la similitude de leurs primitives. Cependant, au lieu de ressortir les éléments dans un ordre dépendant de l'ordre dans lequel ils ont été insérés, on veut les ressortir dans un ordre qui dépend de leur priorité.*

En général, les éléments que l'on insère dans une file de priorité ont une valeur et une priorité associée. Cependant, d'un point de vue algorithmique la valeur de l'élément n'a aucune importance, seule la valeur de sa priorité est pertinente pour le choix du sommet à traiter.

*Pour maintenir un rendu final très compréhensible, on créé des classes explicites en implémentant suivant le paradigme de la programmation orienté objet (POO).*

**1ere implémentation : Liste chaînée (class `list` Python) avec coût sur l'insertion** La première idée est de réaliser cette implémentation simple en utilisant un objet de classe `list` en Python (c'est-à-dire un tableau). Pour cela, on va réaliser une nouvelle classe nommée `FPListe1`.
Pour cela, on va d'abord créé une liste vide (étape de construction) puis implémenter des méthodes correspondant aux 4 primitives exposées ci-dessus.

```
class FPListe1:
    def __init__(self):
        self.file = []

    def __str__(self):
        return '\n'.join([str(i) for i in self.file])

    def __repr__(self):
        return "FP"+"\n"+' ; '.join([str(i) for i in self.file])

    def taille(self):
        return len(self.file)

    def estVide(self):
        return self.taille() == 0

    def filer(self, sommet):
        id, distance = sommet
        if self.estVide():
            self.file.append(sommet)
        else:
            for k in range(self.taille()):
                if distance >= self.file[k][1]:
                    if k == (self.taille() - 1):
                        self.file.insert(k + 1, sommet)
                    else:
                        continue
                else:
                    self.file.insert(k, sommet)

    def defiler(self):
        return self.file.pop(0)
```

**2ème implémentation : Liste chaînée (class `list` Python) avec coût sur la suppression**

```
class FPListe2:
    def __init__(self):
        self.file = []

    def __str__(self):
```

```
6           return '\n'.join([str(i) for i in self.file])
7
8       def __repr__(self):
9           return "FP"+"\n"+' ; '.join([str(i) for i in self.file])
10
11      def taille(self):
12          return len(self.file)
13
14      def estVide(self):
15          return self.taille() == 0
16
17      def filer(self, valeur):
18          self.file.append(valeur)
19
20      def defiler(self):
21          try:
22              minimum = 0
23              for i in range(len(self.file)):
24                  if self.file[i] < self.file[minimum]:
25                      minimum = i
26              element = self.file[minimum]
27              del self.file[minimum]
28              return element
29          except IndexError:
30              print("La file de priorité est vide !")
```

### 4.3 Arbres binaires et tas

## 5 Conclusion

**Prise en compte d'un facteur clé : le temps réel strict**

**Réponse à la problématique**

**Synthèse personnelle**

**Ouverture, améliorations et travaux futurs**

## 6 Bibliographie et Annexes

## 7 Informations *SCEI*