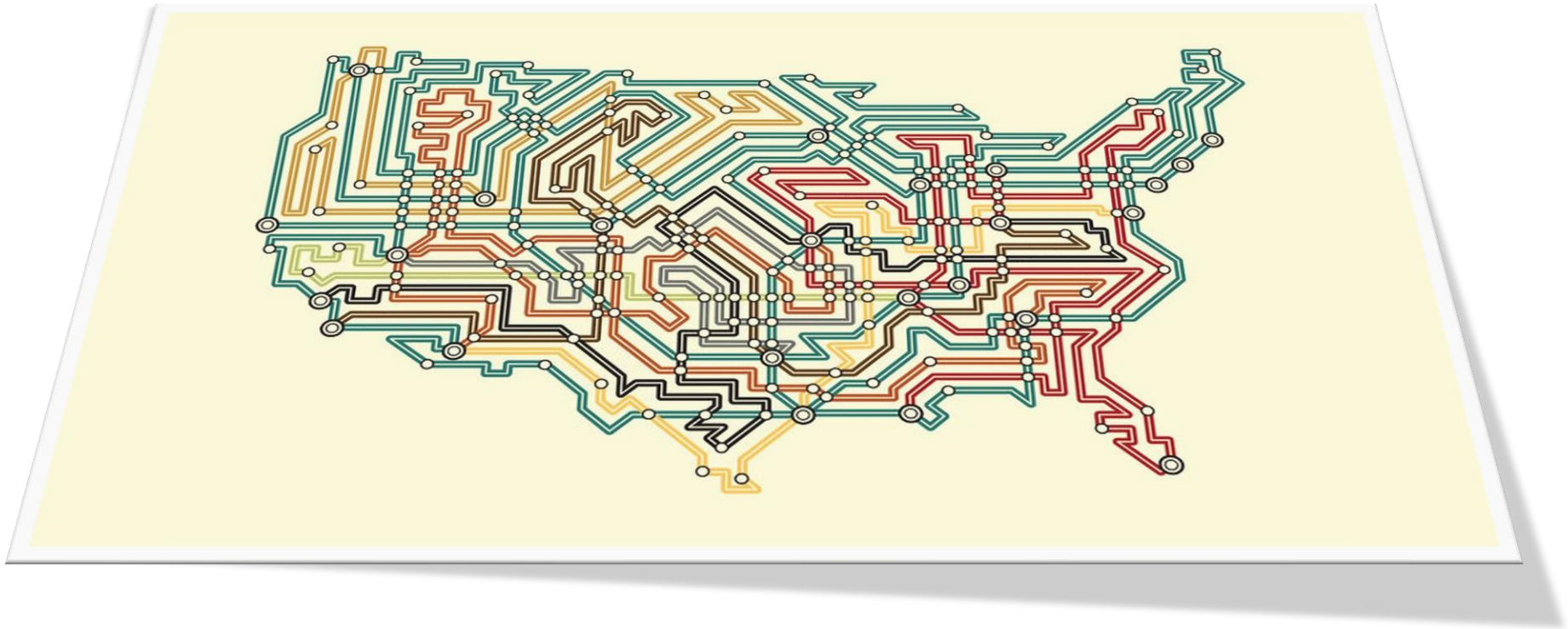
# OPTIMISATION DE LA NAVIGATION ROUTIÈRE À L'AIDE DE L'INTELLIGENCE ARTIFICIELLE



ZOUHRI YASSINE

# SOMMAIRE

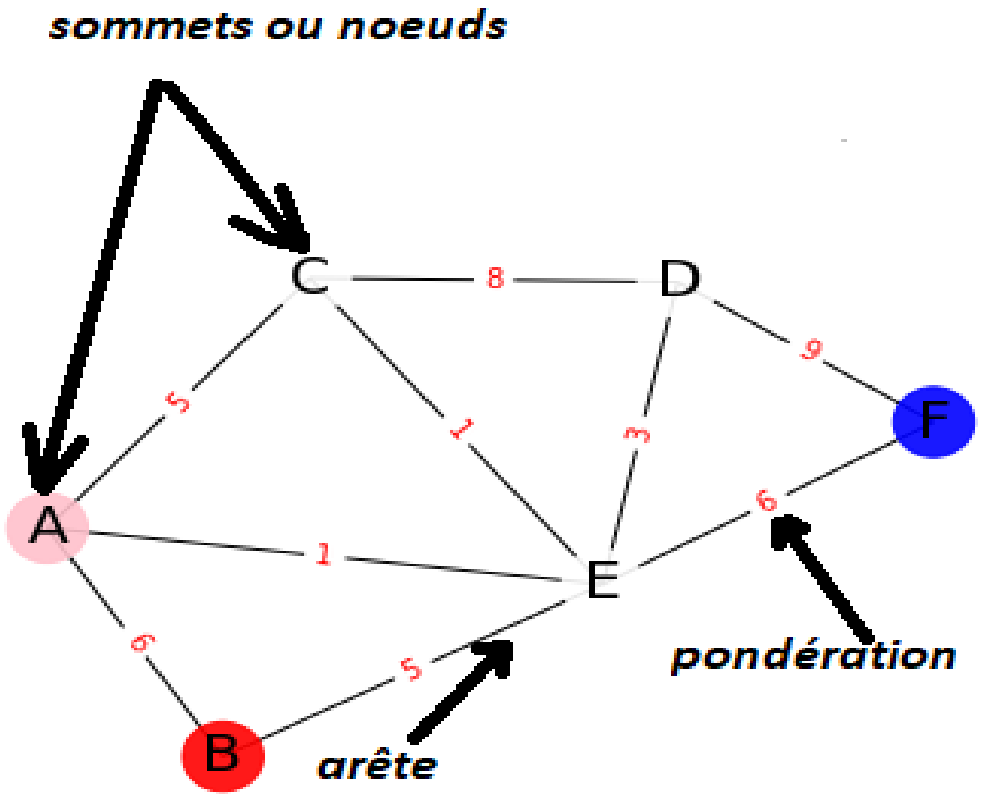
# INTRODUCTION

L'ascension de la population et des véhicules a conduit à une crise de transport pluridimensionnelle : perte de ressources matérielles et énergétiques (carburant), perte de temps , pollution ... D'où la légitimité de la recherche de solutions permettant l'optimisation de la navigation routière et le choix du meilleur itinéraire.

Or, la ville - en particulier les grandes métropoles - étant la plus conçue par cette crise, il serait judicieux de penser à des mécanismes capables de minimiser le coût de transport (temps, trajet, énergie ...) et d'assurer une meilleure mobilité en milieu urbain.

# THÉORIE DES GRAPHES

# QU'EST CE QU'UN GRAPH ?



# COMMENT L'IMPLÉMENTER ?

**Méthode 1 :** A l'aide d'un dictionnaire :

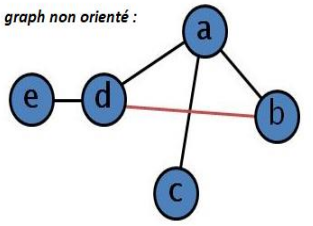
graph={'a':['b' , 'c' , 'd' ] , 'b':[ 'a' ,'e' ,'f' ] , 'c':['a' ,'b' ], 'd' :['c' ,'e' ], 'e':[ ], 'f':[ ]}

```
graph={'a':[('b',2),('c',3),('d',1)] ,.....  }
```

***Méthode 2 :*** A l'aide d'une matrice d'adiacence :

```
graph=[
    [0, 2, 2, 0, 0],
    [3, 0, 0, 1, 1],
    [5, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 4, 0, 0, 0]
]
```

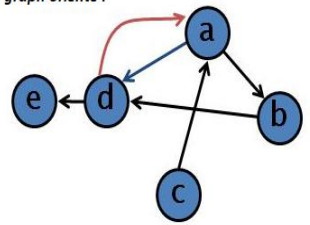
graph non orienté :




| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 1 | 1 | 1 | 0 |
| b | 1 | 0 | 0 | 1 | 0 |
| c | 1 | 0 | 0 | 0 | 0 |
| d | 1 | 1 | 0 | 0 | 1 |
| e | 0 | 0 | 0 | 1 | 0 |

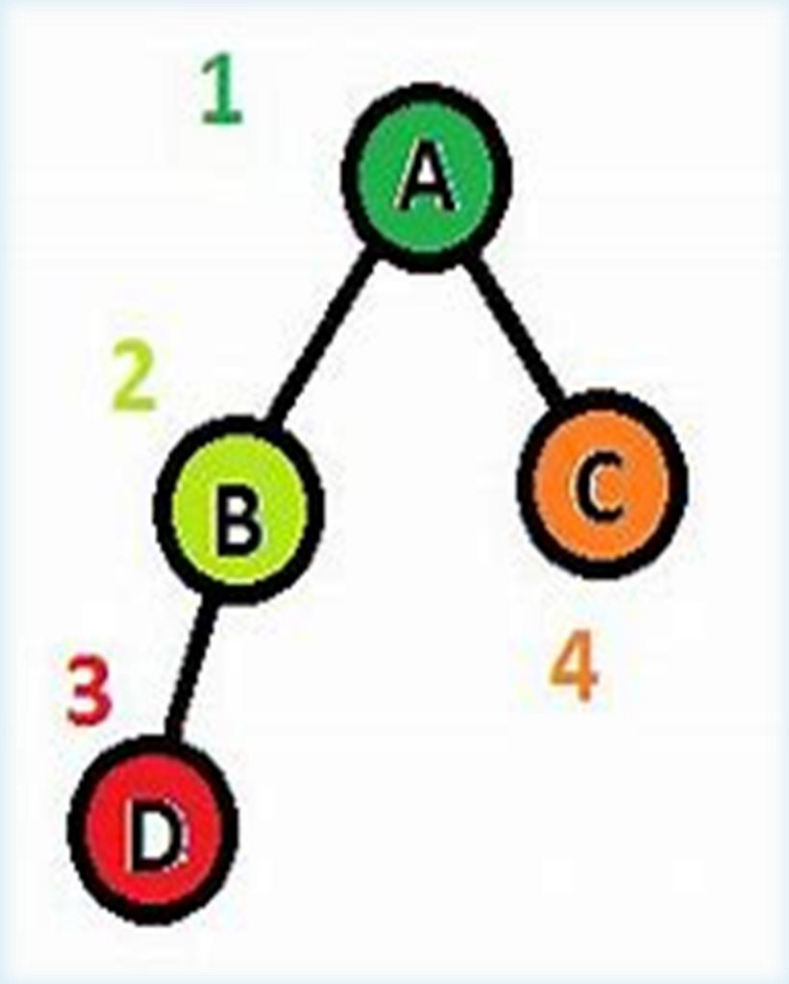
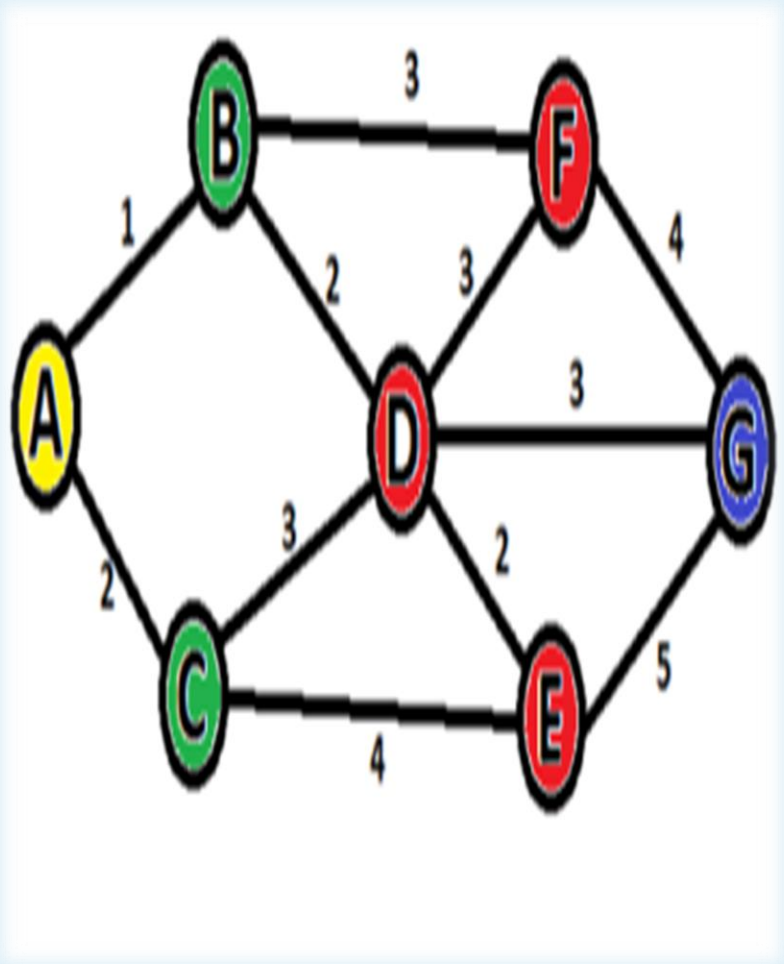
matrice d'adjacence symétrique

graph orienté :




| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 | 0 |
| b | 0 | 0 | 0 | 1 | 0 |
| c | 1 | 0 | 0 | 0 | 0 |
| d | 1 | 0 | 0 | 0 | 1 |
| e | 0 | 0 | 0 | 0 | 0 |

# COMMENT LE PARCOURIR?

| Parcours en longueur | Parcours en largeur |
| --- | --- |
|  |  |

# IMPLÉMENTATION

```python
def bfs(graph,debut,marquer):
    f=[debut]
    while f:
        v=f.pop(0)
        print(v)
        if not v in marquer:
            marquer+=[v]
            f=f+graph[v]
            print(graph[v])
            print(f)
            print(marquer)
    return marquer
```

```python
def voisins_non_visités(G,s,marquer):
    non_visités=[]
    for i in range(len(G)):
        if G[s][i]==1 and marquer[i]==False :
            non_visités.append(i)
    return non_visités
def bfs_adj(G,s,marquer):
    file=[s] , chemin=[]
    while True :
        a=file.pop(0)
        if marquer[a] == False:
            marquer[a]=True
            chemin.append(a)
        l = voisins_non_visités(G,a,marquer)
        if l!=[]:
            for i in l:
                file.append(i)
    return chemin
```

```python
def dfs(graph , debut , marquer):
    marquer+=[debut]
    for sommet in graph[debut]:
        if not sommet in marquer :
            marquer = dfs(graph , sommet , marquer)
    return marquer
graph={'a':['b','c','d'],'b':['a','e','f'],'c':['a','b']}
marquer=[]
debut='a'
print(dfs(graph,debut,marquer))
```

```python
def dfs(graph, s, marquer):
    marquer[s] = True
    print(s, end=" ")

    for i in range(len(graph[s])):
        if graph[s][i] == 1 and not marquer[i]:
            dfs(graph, i, marquer)


def voisins_visités(graph):
    marquer = [False] * len(graph)
    for i in range(len(graph)):
        if not marquer[i]:
            dfs(graph, i , marquer)
```

# A LA RECHERCHE DU PLUS COURT CHEMIN :

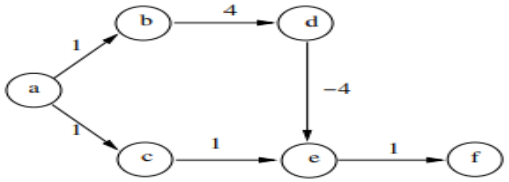
# DANS UN GRAPHE STATIQUE :

# DIJKSTRA

```python
import numpy as np
#La fonction plus_court sélectionne le sommet non visité ayant la distance la plus courte parmi l'ensemble de sommets.
def plus_court(dist, visités):
    #initialisation
    min_dist = np.inf
    min_indice = -1
    for i in range(len(dist)):
        if dist[i] < min_dist and not visités[i]: #Cela signifie qu'une distance plus courte a été trouvée pour le sommet correspondant à l'indice i et ce sommet n'est pas encore visité
            #mise à jour
            min_dist = dist[i]
            min_indice = i
    #Après avoir parcouru tous les sommets, la fonction retourne min_indice, qui correspond à l'indice du sommet non visité ayant la distance la plus courte.
    return min_indice

def dijkstra(adj, s):
    n = len(adj)
    dist = [np.inf] * n  # Tableau des distances les plus courtes
    père = [None] * n  # Tableau des prédécesseurs
    visités = [False] * n  # Tableau pour suivre les sommets visités

    dist[s] = 0  # Distance du sommet de départ à lui-même est de 0
    père[s] = s  # Prédécesseur du sommet de départ est lui-même

    while True:
        u = plus_court(dist, visités)  # Sélectionne le sommet non visité avec la distance la plus courte
        if u == -1:  # Si tous les sommets ont été visités ou il n'y a plus de sommets non visités
            break  # Sort de la boucle

        visités[u] = True  # Marque le sommet comme visité

        #maintenant la  fonction de relaxation
        for v in range(n):
            if not visités[v] and adj[u][v] != 0 and dist[u] + adj[u][v] < dist[v]:
                # Si le sommet v n'est pas visité, il y a une arête entre u et v et une distance plus courte est trouvée
                dist[v] = dist[u] + adj[u][v]  # Met à jour la distance la plus courte
                père[v] = u  # Met à jour le prédécesseur

    return dist, père
```

# BELLMAN-FORD

- L'algorithme de Dijkstra ne marche pas quand le graphe contient des arcs dont les coûts sont négatifs.




- Pour aller de a à f, l'algorithme de Dijkstra va trouver le chemin < a, c, e, f >, de poids 3, alors que le chemin < a, b, d, e, f > est de poids 2.
- Contrairement à Dijkstra, chaque arc va être relâché plusieurs fois : on relâche une première fois tous les arcs ; après quoi, tous les plus courts chemins de longueur 1, partant de s0, auront été trouvés. On relâche alors une deuxième fois tous les arcs ; après quoi, tous les plus courts chemins de longueur 2, partant de s0, auront été trouvés… et ainsi de suite… Après la kième série de relâchement des arcs, tous les plus courts chemins de longueur k, partant de s0, auront été trouvés.
- L'algorithme de Bellman-Ford fonctionne sous réserve que le graphe ne contienne pas de circuit absorbant (un cycle dans le graphe où la somme des poids des arêtes du cycle est négative).



*(Code en ANNEXE 2)*

# A*

- L'algorithme de Dijkstra permet, à partir d'un sommet s , de calculer les distances et les plus courts chemins de s jusqu'à tous les sommets t du graphe. Mais il arrive souvent qu'on cherche un tel chemin pour une seule destination t fixée. Dans ce cas ce n'est pas utile de calculer tous les autres chemins. . .
- On pourrait alors reprendre le code de l'algorithme de Dijkstra, et l'interrompre dès qu'on rencontre le sommet t voulu. Cela va réduire le temps de calcul, mais ce n'est pas encore suffisant : avant de trouver le sommet t, on aura forcément parcouru tous les sommets t' tels que :

$$\delta(s, t') < \delta(s, t).$$

- on doit introduire une heuristique : une fonction qui nous donne une approximation de la distance qu'il nous reste à parcourir pour finir le chemin jusqu'à t.
- une heuristique est admissible si elle ne surapproxime jamais la distance dans G entre deux sommets :

$$\forall (v, t) \in S , h(v, t) \leq \delta(v, t)$$



*(Code en ANNEXE 3)*

# DANS UN GRAPHE DYNAMIQUE:

# QUELQUES DÉFINITIONS

- ***Graphe dynamique stochastique:*** Soit G = (N,A) un graphe orienté et pondéré. G est appelé graphe dynamique stochastique si le poids d'un arc (i,j) ∈ A est déni par une distribution de probabilité discrète et que cette dernière dépende du temps. D'où, le poids de l'arc (i,j) est déni par la fonction p(i,j,t) avec t la date de départ de i.

- ***Graphe dynamique avec intervalles :*** Soit un graphe dynamique G = (N , A). Pour chaque intervalle de temps, le poids de chaque arc est supposé connu avec incertitude. Néanmoins, l'intervalle dans lequel il varie est supposé connu. Par suite, pour chaque intervalle de temps, à un arc du graphe (i,j) ∈ A est associé un intervalle du type $\left[a_{ij}^{t}, b_{ij}^{t}\right]$ avec $0 < a_{ij}^{t} \leq b_{ij}^{t}$ . Donc, à l'instant t, le poids de l'arc (i,j), noté p(i,j,t), est compris entre $a_{ij}^{t}$ et $b_{ij}^{t}$ .

- ***Graphe FIFO avec intervalles :*** Un graphe dynamique avec intervalle est dit graphe FIFO avec intervalles si tous ses arcs vérient la condition FIFO.(first in first out ) :

$$\forall t, t' \geq 0, t \leq t' \Rightarrow t + p(i,j,t) \leq t' + p(i,j,t')$$

# RESOLUTION SUIVANT UN CRITÈRE D'OPTIMISME

- On modélise les «préférences d'un décideur » par la notion du facteur d'optimisme .
- ***DEFINITION:*** Soit le facteur d'optimisme $\alpha$ . $\alpha \in R$ et il est tel que $0 \leq \alpha \leq 1$. Dans le cas le plus optimiste, $\alpha = 0$. Dans le cas le plus pessimiste $\alpha = 1$.
- Pour représenter le poids d'un arc grâce au facteur d'optimisme, une variable aléatoire $X_{ij}^t$ est utilisée. Cette variable aléatoire décrit la distribution des poids possibles d'un arc dans l'intervalle $\left[a_{ij}^t, b_{ij}^t\right]$ . $X_{ij}^t$ peut être une variable aléatoire discrète ou continue.
- ***DEFINITION:*** Le poids de l'arc $(i,j)$ à l'instant t, noté $p(i,j,t)$, est déni comme suit :

  Si $p(i,j,t) \in \mathbb{R}_+^*$ alors $p(i,j,t)$ est le réel vériant la propriété $P(X_{ij}^t \leq p(i,j,t)) = \alpha$ ;
  Si $p(i,j,t) \in \mathbb{N}^*$ alors $p(i,j,t) = E(x)$ avec $x$ est le réel vérifiant la propriété $P(X_{ij}^t \leq x) = \alpha$

*(si le décideur est optimiste alors* $\alpha = 0$. *Par suite,* $p(i,j,t) = a_{ij}^t$ *car* $P(X_{ij}^t \leq a_{ij}^t) = 0$. *Si, par contre le décideur est pessimiste alors* $\alpha = 1$. *Par conséquent,* $p(i,j,t) = b_{ij}^t$ *car* $P(X_{ij}^t \leq b_{ij}^t) = 1$. *)*

***Proposition :*** Si X est une variable aléatoire continue sur [a,b] alors il existe un réel p ∈ [a,b] unique tel que P(X ≤ p) = α avec Cune constante , par contre s'elle est discrète , cela n'est pas toujours vérifié .

***( démo en ANNEXE 4)***

***Hypothèse :*** La variable aléatoire $X_{ij}^{t}$ est une variable aléatoire continue sur $\left[a_{ij}^{t}, b_{ij}^{t}\right]$ . De plus, elle est supposée connue pour tous les arcs et pour tous les intervalles de temps. ( s'elle suit un loi équiprobable par exemple , on obtient $p(i,j,t) = a_{ij}^{t} + (b_{ij}^{t} - a_{ij}^{t})\alpha$ ) ***( voir ANNEXE 5)***

***C/c :*** on obtient par la suite les poids dynamiques p(i,j,t) , qui permettent de calculer le plus court chemin en adaptant l'algorithme **Dijkstra** au cas d'un graphe dynamique avec intervalles et en adoptant ces définitions :

Chaque chemin entre i et j est noté Ch(i,j). L'ensemble de ces chemins est noté ECh(i,j) = {Ch(i,j)}. Formellement, Ch(i,j) = (u0,u1, . . . ,un) avec :

_ u0 = i, un = j

_ ∀uk,uk+1 ∈ Ch(i,j), (uk,uk+1) ∈A

_ P(i,j) est le poids de l'arc reliant i et j

G = (N,A) un graphe dynamique . Soit CD la fonction de coût d'un chemin dans G. $CD^{t_0}$ ((u0,u1, . . . ,um)) indique le coût du chemin (u0,u1, . . . ,um) en partant du nœud u0 à t0. La valeur de $CD^{t_0}$ ((u0, . . . ,um)) est définie comme suit :

_ $CD^{t_0}$ ((u0, . . . ,um−1)) + p(um−1,um,CDt0 ((u0, . . . ,um−1))) Si m > 0

_t0 Si m = 0

Le plus court chemin de i ∈ N à j ∈ N en partant de i à l'instant t0 est noté $PCC^{t_0}$ (i,j) = Ch(i,j) avec $CD^{t_0}$ (Ch(i,j)) = $\min_{P_{\epsilon Ech(i,j)}} \{CD^{t_0}(P)\}$. Au cas où plusieurs chemins avec un coût minimal existent, un parmi eux est choisi aléatoirement.

# RESOLUTION SUIVANT UN CRITÈRE D'OPTIMISM DYNAMIQUE

- Maintenant , $\alpha$ deviendra dynamique : $\alpha(i,j,t)$ , et le poids d'un arc (i,j) sera défini comme dans la partie précédente
- Nous proposons de définir $\alpha(i,j,t)$ à l'aide de quatre paramètres : $\alpha_{profil}$, $\beta(i,j,t)$, $\gamma(i,j,t)$ et $\delta(i,j,t)$ :

  _ $\alpha_{profil}$ modélise les désirs du décideur

  _$\beta(i,j,t)$, $\gamma(i,j,t)$ et $\delta(i,j,t)$ ont pour rôle de corriger de $\alpha_{profil}$ :

➢ $\beta(i,j,t)$ indique l'influence de l'arc (i,j) sur le facteur d'optimisme (congestion , structure ou localisation du chemin

➢ $\gamma(i,j,t)$ tient en compte les erreurs d'estimation réalisées dans le passé

➢ $\delta(i,j,t)$ intègre tous les phénomènes résiduels influants sur le facteur d'optimisme(météo, probabilité d'un accident……)

$$\beta_{(i,j,t)} = \begin{cases} 100\% & \text{Si } (i,j) \in H1 \\ 110\% & \text{Si } (i,j) \in H2 \\ 120\% & \text{Si } (i,j) \in H3 \end{cases}$$

$$\gamma_{(i,j,t)} = \begin{cases} 100\% & \text{Si } t = 0 \\ \gamma_{(i,j,t-1)} \times \left(1 + \left(\dfrac{peff(i,j,t-1) - pest(i,j,t-1)}{b_{ij}^{t-1} - a_{ij}^{t-1}}\right)\right) & \\ sinon & \end{cases}$$

$$\delta_{(i,j,t)} = \begin{cases} 100\% & \text{Si beau temps} \\ 110\% & \text{Si pluie} \\ 120\% & \text{Si neige} \end{cases}$$

$$\alpha_{(i,j,t)} =$$

$$\min\{1, \alpha_{profil} \times (w1 \times \beta_{(i,j,t)} + w2 \times \gamma_{(i,j,t)} + w3 \times \delta_{(i,j,t)}\}$$

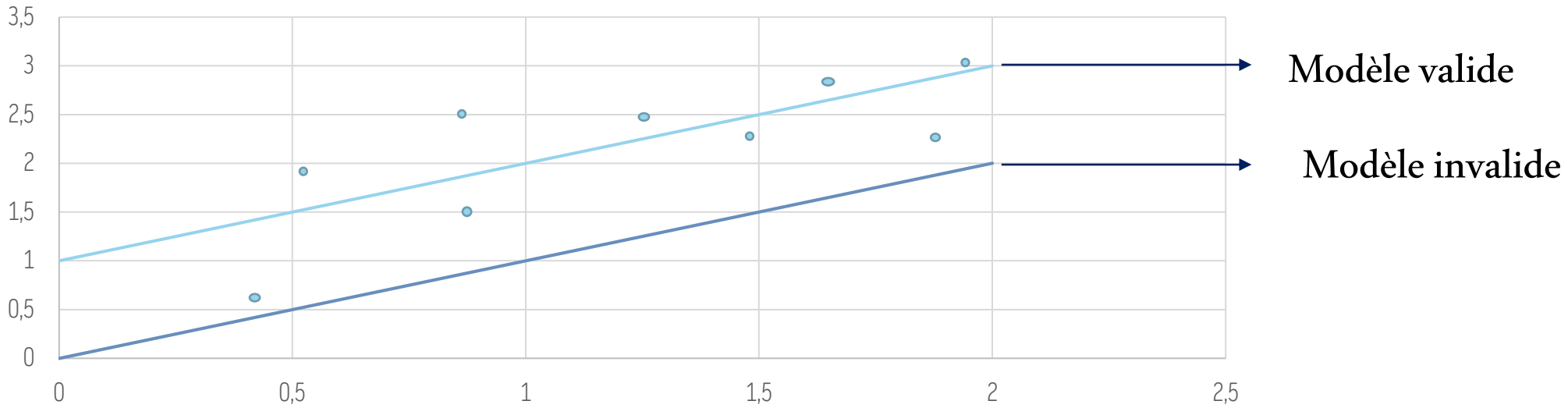
*Agrégation par somme pondérée*

# OPTIMISATION À L'AIDE DE L'INTELLIGENCE ARTIFICIELLE

# DÉSCENTE DU GRADIENT

*Objectif : approcher la loi de probabilité suivie par* $X_{ij}^{t}$

- Hypothèse : on suppose que la probabilité varie de manière linéaire . Cela nous permettra de définir le modèle comme étant une droite affine ***(f(x)=ax+b))*** . Généralement , on peut imposer n'importe seul modèle qu'on veule( loi normale , loi géométrique ….) selon les spécificités du problème .
- L'objectif est de trouver a et b tel que l'erreur (la distance ) par rapport au modèle soit minimale



## *Procédé :*

- Fonction du cout :

$$j(a,b) = \frac{1}{2n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$$

$$= \frac{1}{2n}\sum_{i=1}^{n}(ax_i + b - y_i)^2$$

- le cout doit etre minimal , on cherche donc a et b tel que :

$$\frac{\partial j(a,b)}{\partial a} = \frac{1}{n}\sum_{i=1}^{n} x_i(ax_i + b - y_i) = 0 ,$$

$$\frac{\partial j(a,b)}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} (ax_i + b - y_i) = 0$$

- Le calcul étant lourd , on introduit la boucle de déscente du gradient : $\begin{cases} a = a - \alpha\frac{\partial j(a,b)}{\partial a} \\ b = b - \alpha\frac{\partial j(a,b)}{\partial b} \end{cases}$ où $\alpha$ est le pas de la descente. On arrete la boucle un fois que $a_{n-1} \approx a_n$ et $b_{n-1} \approx b_n$

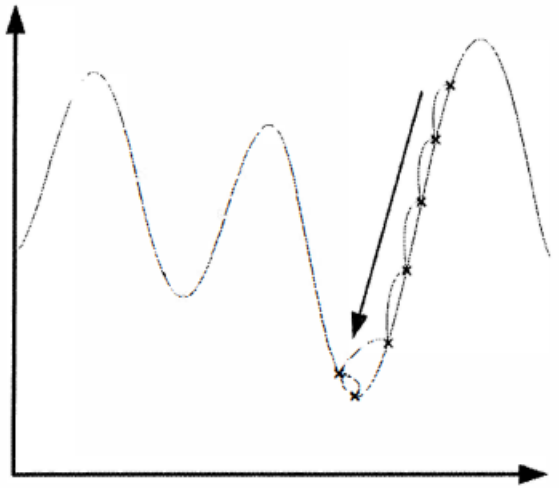
- Dans notre cas :

$$X=\begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_m \end{bmatrix} \quad , \quad y=\begin{bmatrix} y_1 \\ \vdots \\ \vdots \\ \vdots \\ y_m \end{bmatrix} \quad , \; f(x)=\begin{bmatrix} f(x_1) = ax_1 + b \\ \vdots \\ \vdots \\ \vdots \\ f(x_m) = ax_m + b \end{bmatrix}$$

- Sous forme matricielle , le modèle s'écrit f(x) = X . P avec :

$X=\begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ \vdots & \vdots \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix}$ et P = $\begin{bmatrix} a \\ b \end{bmatrix}$ et la function du coût J(P)= $\frac{1}{2m}\sum_{i=1}^{m}(XP - Y)^2$

- $\frac{\partial J(P)}{\partial P} = \frac{1}{m} X^t (XP - Y)$ avec $X^t = \begin{pmatrix} x_1 & \cdots\cdots & x_m \\ 1 & \cdots\cdots & 1 \end{pmatrix}$
- Descente du gradient : P = P - $\alpha \frac{\partial J(P)}{\partial P}$

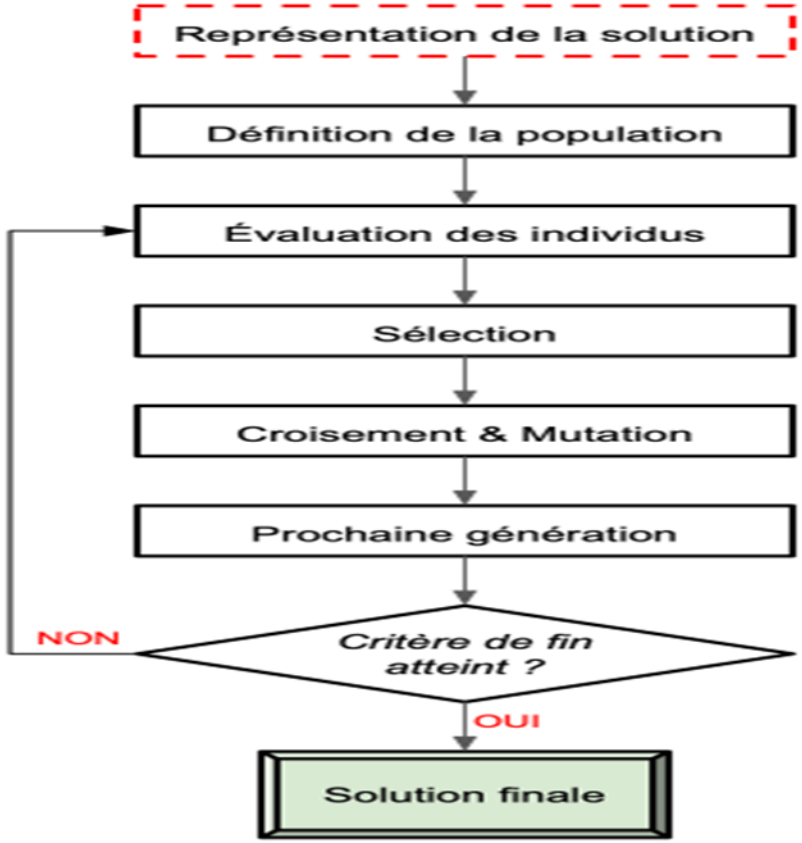


# ALGORITHME GÉNÉTIQUE

*Objectif : améliorr les résultats obbtenus à chaque itération*

*Procédé :*

- Définir les paramètres de l'algorithme.
- Générer une population initiale aléatoire de solutions.
- Évaluer chaque solution en fonction d'un critère de performance
- Répéter jusqu'à atteindre le critère d'arrêt Sélectionner.



- ***PARAMÈTRES DE L'ALGORITHME:***

_NOMBRE MAXIMUM D'ITÉRATIONS

_TAILLE DE LA POPULATION

_PROBABILITÉ DE MUTATION

_CRITÈRE D'ARRÊT ( UN NOMBRE D'ITÉRATIONS SANS AMÉLIORATION SIGNIFICATIVE)

- ***ÉVALUER CHAQUE SOLUTION EN FONCTION D'UN CRITÈRE DE PERFORMANCE:***

_ SÉLECTIONNER LES MEILLEURES SOLUTIONS DE LA POPULATION ACTUELLE EN FONCTION DE LEUR ÉVALUATION.

_GÉNÉRER DE NOUVELLES SOLUTIONS EN COMBINANT LES MEILLEURS ITINÉRAIRES DE LA POPULATION ACTUELLE, EN UTILISANT DES OPÉRATIONS TELLES QUE LE CROISEMENT (CROSSOVER) ET LA MUTATION.

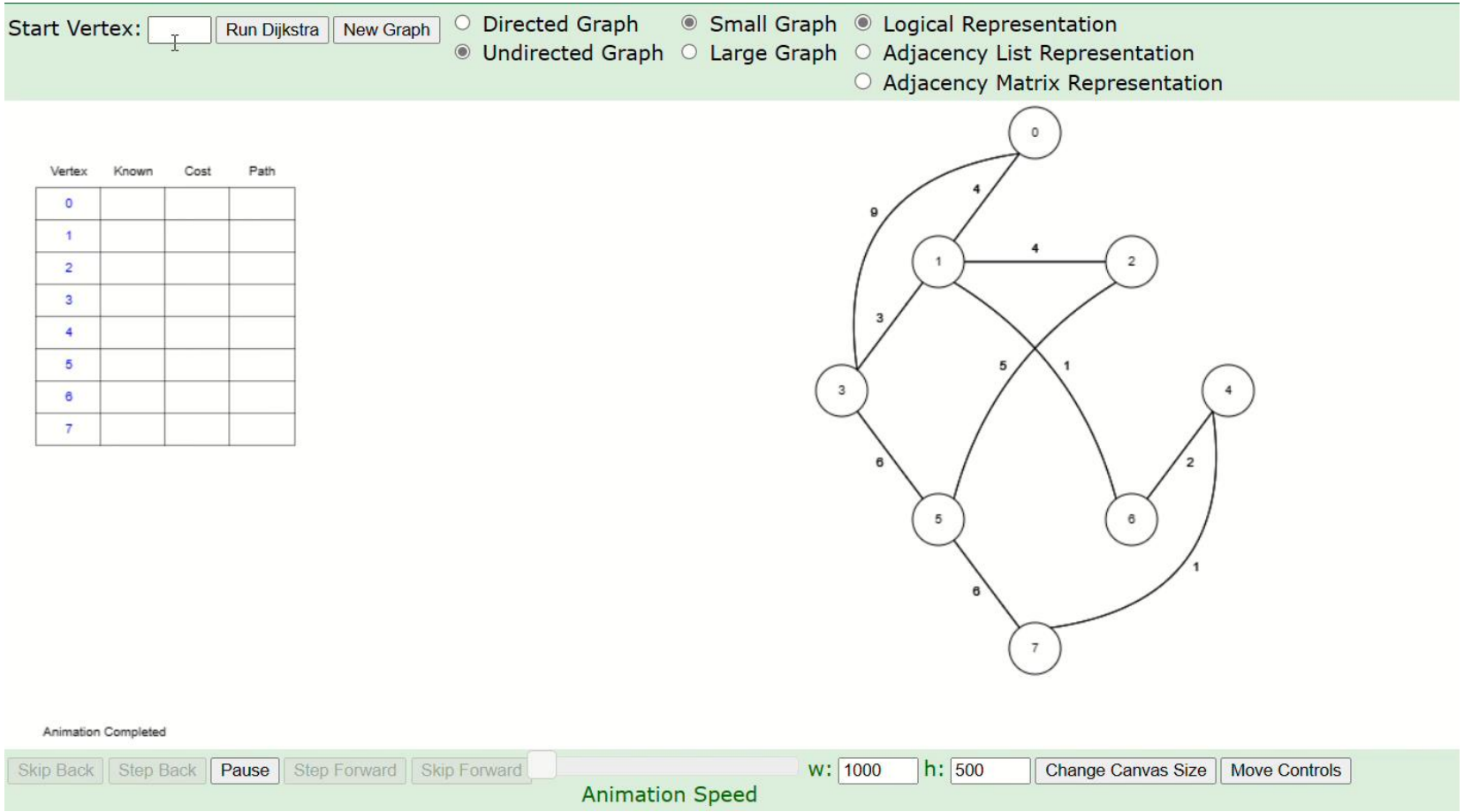
_ÉVALUER LES NOUVELLES SOLUTIONS GÉNÉRÉES.

_REMPLACER LA POPULATION ACTUELLE PAR LES MEILLEURES SOLUTIONS PARMI LES NOUVELLES SOLUTIONS ET LA POPULATION PRÉCÉDENTE



*Code en ANNEXE 6*

# ANNEXE 1:


Start Vertex:
Run Dijkstra
New Graph
Directed Graph
Undirected Graph
Small Graph
Large Graph
Logical Representation
Adjacency List Representation
Adjacency Matrix Representation
Vertex
Known
Cost
Path
0
1
2
3
4
5
6
7
Animation Completed
Skip Back
Step Back
Pause
Step Forward
Skip Forward
w:
1000
h:
500
Change Canvas Size
Move Controls
Animation Speed

# ANNEXE 2:

```python
import numpy as np
def bellman_ford(adj, s):
    n = len(adj)
    dist = [np.inf] * n  # Tableau des distances les plus courtes
    père = [None] * n  # Tableau des prédécesseurs

    dist[s] = 0  # Distance du sommet de départ à lui-même est de 0
    père[s] = s  # Prédécesseur du sommet de départ est lui-même

    for i in range(n - 1):
        for u in range(n):
            for v in range(n):
                if adj[u][v] != 0 and dist[u] + adj[u][v] < dist[v]:
                    # Si une distance plus courte est trouvée entre u et v
                    dist[v] = dist[u] + adj[u][v]  # Met à jour la distance la plus courte
                    père[v] = u  # Met à jour le prédécesseur

    # Vérification des cycles de poids négatifs(cycles absorbants)
    for u in range(n):
        for v in range(n):
            if adj[u][v] != 0 and dist[u] + adj[u][v] < dist[v]:
                # Si une distance plus courte est trouvée, cela signifie qu'il y a un cycle de poids négatif et une exception ValueError est levée
                raise ValueError("Le graphe contient un cycle de poids négatif")

    return dist, père
```

# ANNEXE 3 :

```python
import numpy as np

def retirer_min_a_star(F, ds, t, h):
    # Retire le nœud avec la plus petite valeur de ds[v] + h(v, t) dans F
    if not F:
        return None

    im = 0
    dm = np.inf

    for i in range(len(F)):
        v = F[i]
        dh_v = ds[v] + h(v, t)
        if dh_v < dm:
            dm = dh_v
            im = i

    return F.pop(im)

def chemin(s, t, pred):
    # Reconstruit le chemin final à partir de s jusqu'à t en utilisant pred
    courant = t
    L = [t]
    while courant != s:
        courant = pred[courant]
        L.append(courant)
    return L[::-1]
```

```python
def a_star(G, s, t, h):
    # Algorithme A* pour trouver le chemin le plus court entre s et t dans le graphe G

    if s not in G or t not in G:
        # Vérifie si les nœuds source et cible sont présents dans le graphe
        return "Les nœuds source et cible ne sont pas présents dans le graphe."

    ds = {}
    for u in G:
        ds[u] = np.inf
    ds[s] = 0

    F = [s]  # Liste des nœuds à évaluer
    pred = {}  # Dictionnaire pour stocker les prédécesseurs de chaque nœud visité

    while F:
        u = retirer_min_a_star(F, ds, t, h)
        if u == t:
            # Si le nœud actuel est le nœud cible, le chemin le plus court a été trouvé
            return ds[t], chemin(s, t, pred)

        for v in G[u]:
            if v not in F:
                # Ajoute le nœud voisin v à la liste des nœuds à évaluer
                F.append(v)

            new_dist = ds[u] + G[u][v] + h(v, t)
            if new_dist < ds[v]:
                # Met à jour la distance la plus courte vers le nœud voisin v
                ds[v] = new_dist
                pred[v] = u  # Met à jour le prédécesseur du nœud voisin v

    return "Pas de chemin de s à t"
```

# ANNEXE 4 :

Soient X une variable aléatoire continue sur [a,b], f sa densité de probabilité et $\alpha$ une constante:

$P(X \leq p) = \int_{-\infty}^{y} f(x)dx$

$= \int_{a}^{y} f(x)dx$

En posant $F(y) = \int_{a}^{y} f(x)dx - \alpha$ définie sur [a,b] ,

sachant que :

F est une fonction continue sur [a,b] car elle est dérivable sur [a,b] ;

$F(a) = -\alpha \leq 0$ car $0 \leq \alpha \leq 1$ ;

$F(b) = 1 - \alpha \geq 0$ car $0 \leq \alpha \leq 1$.

Puis que :

F est strictement monotone sur [a,b] car f est strictement positive sur[a,b].

On a d'après le théorème des valeurs intermédiaires et le théorème de bijection :

$\exists!\ p \in [a,b]$ tel que $F(p) = 0$. Donc, $\exists!\ p \in [a,b]$ tel que $P(X \leq p) = \alpha$

contre-exemple :

Soit X une variable aléatoire discrète qui ne peut prendre que les valeurs a ou b ($X \in \{a,b\}$) avec P(a) = 0.5 et P(b) = 0.5. Si nous considérons $\alpha = 0.75$,

$\nexists\ p \in [a,b]$ tel que $P(X \leq p) = \alpha$. En effet,

$P(X \leq y) = 0.5\ \forall y \in [a,b[$ et $P(X \leq y) = 1$ pour $y = b$.

# ANNEXE 5:

**Étude d'un exemple** Soit $X_{ij}^t$ $i,j \in N$ une variable aléatoire continue qui suit une loi équiprobable. Soit $f$ la densité de probabilité associée à cette variable aléatoire. Étant donné que $X_{ij}^t$ suit une loi équiprobable et qu'elle est définie sur $[a_{ij}^t, b_{ij}^t]$ alors $f(x) = \frac{1}{b_{ij}^t - a_{ij}^t}$ $\forall x \in [a,b]$. $p(i,j,t)$ est défini tel que $P(X_{ij}^t \leq p(i,j,t)) = \alpha$ d'où

$$\begin{aligned}
P(X_{ij}^t \leq p(i,j,t)) &= \alpha \\
\Rightarrow \int_{-\infty}^{p(i,j,t)} f(x)dx &= \alpha \\
\Rightarrow \int_{a_{ij}^t}^{p(i,j,t)} f(x)dx &= \alpha \\
\Rightarrow \int_{a_{ij}^t}^{p(i,j,t)} \frac{dx}{b_{ij}^t - a_{ij}^t} &= \alpha \\
\Rightarrow \frac{p(i,j,t) - a_{ij}^t}{b_{ij}^t - a_{ij}^t} &= \alpha \\
\Rightarrow p(i,j,t) &= a_{ij}^t + (b_{ij}^t - a_{ij}^t)\alpha
\end{aligned}$$

D'où pour cet exemple, le poids d'un arc est défini par la fonction $p(i,j,t) = a_{ij}^t + (b_{ij}^t - a_{ij}^t)\alpha$. Dans le cas le plus optimiste, $\alpha = 0$ d'où $p(i,j,t) = a_{ij}^t$. Dans le cas le plus pessimiste, $\alpha = 1$ d'où $p(i,j,t) = b_{ij}^t$.

# ANNEXE 6

```
def mutate(solution, mutation_rate):
    mutated_solution = list(solution)  # Copie de la solution initiale

    for i in range(len(mutated_solution)):
        if random.random() < mutation_rate:
            # Appliquer une mutation à l'élément i de la solution
            mutated_solution[i] = generate_random_element()  # Générer un
nouvel élément aléatoire

    return tuple(mutated_solution)
```

```
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child = parent1[:crossover_point] + parent2[crossover_point:]
    return child
```

```
import random
population_size = 100
mutation_rate = 0.01
num_generations = 100
# Fonction pour évaluer une solution
def fitness(solution):
    # Calculer la valeur de fitness de la solution
    pass

# Fonction pour générer une solution aléatoire
def generate_random_solution():
    # Générer une solution aléatoire
    pass

# Algorithme génétique
population = [generate_random_solution() for _ in range(population_size)]

for generation in range(num_generations):
    # Évaluation de la population
    fitness_scores = [fitness(solution) for solution in population]
```

```
# Sélection des parents pour la reproduction
    parent_indices = random.choices(range(population_size), weights=fitness_scores, k=2)
    parents = [population[i] for i in parent_indices]

# Croisement (crossover) et mutation pour créer une nouvelle génération
    offspring = []
    for _ in range(population_size):
        parent1, parent2 = random.choices(parents, k=2)
        child = crossover(parent1, parent2)
        mutated_child = mutate(child, mutation_rate)
        offspring.append(mutated_child)

    # Remplacement de la population par la nouvelle génération
    population = offspring

# Sélection de la meilleure solution de la dernière génération
best_solution = max(population, key=fitness)

# Affichage de la meilleure solution et sa valeur de fitness
print("Meilleure solution trouvée :", best_solution)
print("Valeur de fitness :", fitness(best_solution))
```