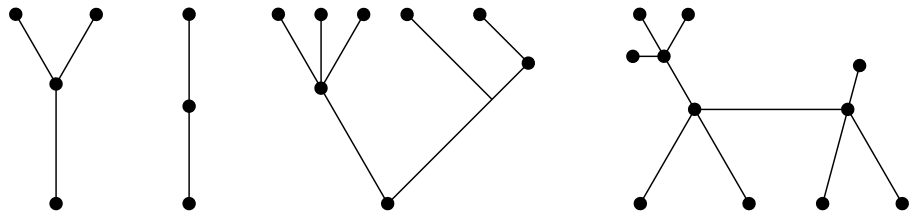
Lycée Loges

Chapitre 11

Théorie des Graphes

Simon Dauguet
simon.dauguet@gmail.com

22 mai 2025



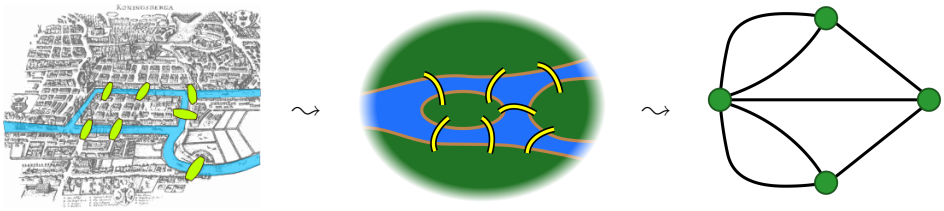
THE LAST TREE IS PARTICULARLY KNOWN FOR IT'S BARK

Table des matières

1 Généralités sur les graphes

Les graphes permettent de modéliser de très nombreuses situations : les réseaux routier ou informatique, schématiser des plans, décrire différents états d'un jeu, etc. Dans tous nos exemples, deux sommets d'un graphe ne seront reliés qu'au maximum par un arc (ou arête) : on étudiera donc les graphes simples auxquels les boucles sont ajoutées.

Il est usuel de retracé le début de la théorie des graphes avec le problème des 7 ponts de Königsberg, dont Léonard Euler donne une réponse en 1735.



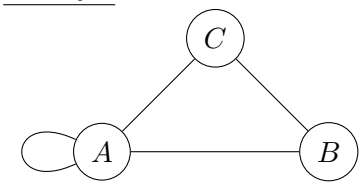
1.1 Graphes non orientés

1.1.1 Définition et représentation graphique

Définition 1.1 (Graphe non-orienté) :
Un graphe non-orienté est un ensemble de sommets (*vertex*), reliés par des arêtes (*edge*). Autrement dit, un graphe non-orienté est la donnée d'un couple $(\mathcal{S}, \mathcal{A})$ où $\mathcal{S}$ est l'ensemble des sommets et $\mathcal{A}$ est l'ensemble des arêtes.

Une arête sera noter $\{S_i, S_j\}$ ou $S_i{-}S_j$. Si l'arête relie le même sommet (notée $\{S_i\}$ ou $S_i{-}S_i$), il s'agit d'une *boucle*.

Exemple 1.1 :



Il s'agit d'un graphe non-orienté complet avec une boucle $A{-}A$.
Les sommets sont : A, B, C
Les arêtes sont : $\{A\}$; $\{A, B\}$; $\{A, C\}$ et $\{B, C\}$
ou également notées $A{-}A$; $A{-}B$; $A{-}C$ et $B{-}C$

Définition 1.2 (Degré d'un sommet, Sommets adjacents, Chaîne, Cycle, Longueur d'une chaîne) :
Le *degré d'un sommet* est le nombre d'arête relié à ce sommet, noté $d(S)$. En cas de boucle, on comptera deux.

Un sommet S_i est *adjacent* à un sommet S_j s'il existe une arête reliant le sommet S_i au sommet S_j.

Une *chaîne* entre un sommet S_i et un sommet S_j est une succession d'arêtes consécutives permettant de relier S_i à S_j

Un *cycle* est une chaîne partant et arrivant au même sommet. S'il ne passe jamais deux fois par la même arête il est qualifié de *cycle simple* ; s'il ne passe jamais deux fois par le même sommet (excepté l'extrémité) il est qualifié de *cycle élémentaire*.

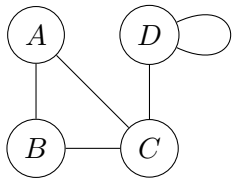
La *longueur d'une chaîne* entre deux sommets S_i et S_j est égal au nombre d'arêtes formant cette chaîne.

Exemple 1.2 :
Pour le graphe de l'exemple précédent, le degré de A vaut $d(A) = 4$, tandis que celui de B vaut $d(B) = 2$.

Tous les sommets sont adjacents l'un des autres.

Il existe une cycle simple de longueur 4, $(A-A-B-C-A)$, et un cycle élémentaire de longueur 3 $(A-B-C-A)$.

Exercice 1 :
Décrire succinctement le graphe suivant :



Remarque :
Nous ne considérerons que des *graphes simples*, c'est-à-dire des graphes dont chaque paire de sommets est relié par 0 ou 1 arête au maximum. Les graphes dont au moins deux sommets sont reliés par plusieurs arêtes s'appellent des *multigraphes*.

Définition 1.3 (Graphe connexe, Composantes connexes) :
Un graphe non orienté $G = (\mathcal{S}, \mathcal{A})$ est *connexe* si, pour toute paire de sommets (S_i, S_j) de $\mathcal{S}$, il existe une chaîne relation S_i à S_j. Autrement dit, un graphe non orienté est connexe, si n'importe quel sommet est relié par une chaîne à tous les autres sommets. Un graphe est connexe s'il est "d'un seule tenant", s'il ne peut être séparé en plusieurs morceaux non reliés.

Une *composante connexe* de G est une sous-graphe de G (donc un sous-ensemble de sommets) maximal pur l'inclusion dont tous les sommets sont reliés par des chaînes (*i.e.* une composante connexe est une sous-graphe connexe de G le plus grand possible).

Proposition 1.1 :
Soit $G = (\mathcal{S}, \mathcal{A})$ un graphe non-orienté, la relation "les sommets S_i, S_j sont reliés par une chaîne" définie une relation d'équivalence. Les classes d'équivalence forment alors les composantes connexes du graphe.

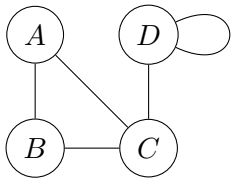
Démonstration :

- on considère que S_i est par défaut relié à lui-même,
- si S_i est relié à S_j, par symétrie des arêtes, S_j est relié à S_i,
- si S_i est relié à S_j et S_j à S_k, en mettant bout à bout les deux chaînes, les sommets S_i et S_k sont reliés.

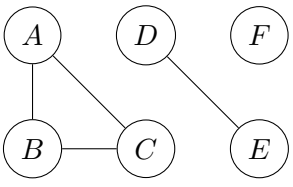
□

Remarque :
Un graphe non orienté est donc connexe s'il ne possède qu'une seule classe d'équivalence pour cette relation d'équivalence.

Exemple 1.3 :



graphe connexe



graphe avec trois composantes connexes
(F est un sommet isolé)

Définition 1.4 (Graphe complet, Graphe bi-parti, Arbre, Forêt) :
Soit G est un graphe acyclique (sans boucle).

Le graphe G est dit *complet* si pour tout couple de sommets distincts un arc (ou arête) les relie, *i.e.* si tous les sommets est relié à tous les autres sommets du graphes.

Le graphe G est dit *bi-parti* s'il est possible de séparer les sommets en deux sous-ensembles disjoints $\mathcal{S}_1$ et $\mathcal{S}_2$ de sorte que toute arête relie un sommet de $\mathcal{S}_1$ à un sommet de $\mathcal{S}_2$.

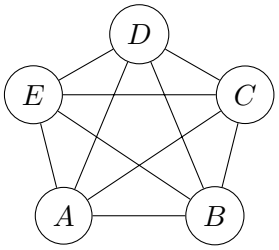
Un *arbre* est un graphe non orienté, connexe et sans cycle.

Un *arbre* dont chaque sommet possède au plus un sommet adjacent se nomme une liste.

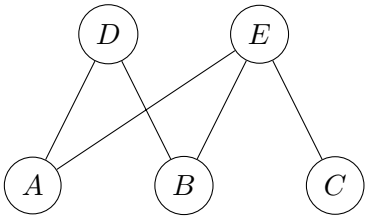
Une *forêt* est un graphe non orienté dont toutes les composantes connexes sont des arbres, *i.e.* une forêt est un ensemble d'arbre.

Exemple 1.4 :

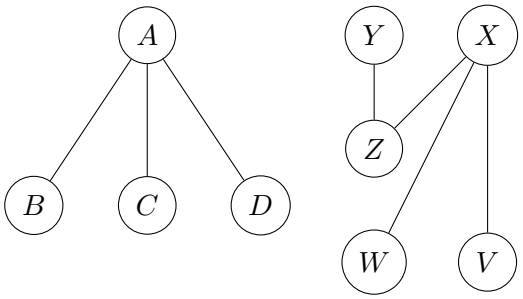
graphe complet



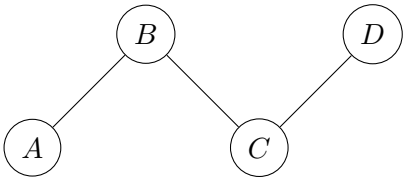
graphe bi-parti



forêt composée de 2 arbres



liste



1.1.2 Premières propriétés

Proposition 1.2 :
Soit $G = (\mathcal{S}, \mathcal{A})$ un graphe non-orienté, alors

$$\sum_{S \in \mathcal{S}} d(S) = 2|\mathcal{A}|.$$

Démonstration :
Par définition, le degré d'un sommet compte le nombre d'arêtes atteignant ce sommet. Or une arête est définie par deux sommets, ainsi

$$2|\mathcal{A}| = \sum_{S \in \mathcal{S}} d(S).$$

□

Théorème 1.3 :
Soit G est un graphe simple.
Le graphe G est bi-parti si, et seulement si, il ne contient aucun cycle de longueur impair.

Démonstration (partielle) *:*
Dans un graphe bi-parti un cycle est composé de sommets appartenant alternativement à l'ensemble $\mathcal{S}_1$ et $\mathcal{S}_2$. Si le cycle est de longueur impair, le 1er sommet du cycle appartient nécessairement au même ensemble que l'avant-dernier sommet de ce cycle. Ce qui contredit le fait que toutes les arêtes soient définies par un sommet de $\mathcal{S}_1$ et un sommet de $\mathcal{S}_2$. Les cycles ne peuvent donc être de longueur impair.

La réciproque est admise mais l'idée est de construire les ensembles $\mathcal{S}_1$ et $\mathcal{S}_2$ à partir d'un arbre couvrant chaque composante connexe de G. □

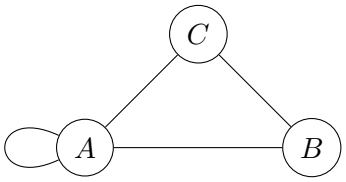
1.1.3 Représentation matricielle

Définition 1.5 (Matrice d'adjacence) :
Soit $G = (\mathcal{S}, \mathcal{A})$ avec $n = \mathrm{Card}(\mathcal{S})$ le nombre de sommets.
La *matrice d'adjacence* $(m_{i,j})_{ij} \in \mathcal{M}_n(\mathbb{N})$ du graphe G est définie par

$$\forall i, j \in \{1, \ldots, n\},\ m_{i,j} = \begin{cases} 1 & \text{si l'arête } \{S_i, S_j\} \text{ existe} \\ 0 & \text{sinon} \end{cases}$$

Remarque :
Dans le cas où les nœuds d'un graphe sont nommé à partir de l'alphabet, les sommets sont en général donné dans l'ordre alphabétique. Ça permet de s'assurer de ne pas nommer une arête deux fois, par exemple.

Exemple 1.5 :



a pour matrice d'adjacence $M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$

Proposition 1.4 (Matrice d'un graphe non orienté) :
Soit G un graphe non orienté et M sa matrice d'adjacence. Alors :

(i) M est symétrique.

(ii) Si une ligne i est remplie de zéro, alors la colonne i est remplie de 0 (par symétrie) et aucune arête ne sort du sommet S_i (le somme S_i est isolé).

Exercice 2 :
Considérons $\mathcal{S} = \{A, B, C, D, E\}$ et $\mathcal{A} = \{\{A, B\}, \{A, C\}, \{A, E\}, \{B, C\}, \{B, D\}, \{D, E\}\}$.
Dessiner le graphe $G = (\mathcal{S}, \mathcal{A})$ et préciser sa matrice d'adjacence.

Proposition 1.5 (Degré d'un nœuds) :
Soient $G = (\mathcal{S}, \mathcal{A})$ un graphe non-orienté et M sa matrice d'ajdacence et $n = \mathrm{Card}(\mathcal{S})$. Alors :

(i) $d(S_i) = \sum_{j=1}^{n} m_{i,j} = \sum_{j=1}^{n} m_{j,i}$

(ii) $\mathrm{tr}(M)$ est le nombre de boucles du graphe G.

Exemple 1.6 :
En prenant le graphe de l'exemple précédent, on a bien

- $d(A) = 3 = m_{1,1} + m_{1,2} + m_{1,3} = m_{1,1} + m_{2,1} + m_{3,1}$
- $d(B) = 2 = m_{2,1} + m_{2,2} + m_{2,3} = m_{1,2} + m_{2,2} + m_{3,2}$
- une seule boucle en A et $\mathrm{tr}(M) = 1$.

Théorème 1.6 :
Soit G un graphe ayant n sommets, $M \in \mathcal{M}_n(\mathbb{N})$ sa matrice d'adjacence, et $p \in \mathbb{N}^*$.
Alors, $\forall i, j \in \{1, \dots, n\}$, $[M^p]_{i,j}$ représente alors le nombre de chaînes de longueur p entre les sommets S_i et S_j.

Démonstration :
Pour $p = 1$, c'est évident. C'est la définition de M.

Soit $p \in \mathbb{N}^*$. Supposons que les coefficients de M^p représente le nombres de chaînes de longueur p entre les différents sommets de G.

Soit $i, j \in \{1, \dots, n\}$. Toute chaîne de longueur $p+1$ peut être découpé comme une chaîne de longueur p, puis une arête (*i.e.* une chaîne de longueur 1). Donc, toute chaîne de longueur $p+1$ entre S_i et S_j est composée par la succession d'une chaîne de longueur p entre S_i et S_k et d'une arête entre S_k et S_j si elles existent, avec un $k \in \{1, \dots, n\}$.

Par hypothèse de récurrence, le nombre de chaînes de longueur p reliant S_i à S_k est donné par $[M^p]_{i,k}$. Puisque $m_{k,j}$ vaut 1 si l'arête (S_k, S_j) existe, 0 sinon, le nombre de chaînes de longueur $p+1$ reliant S_i à S_j est donné par

$$\sum_{k=1}^{n} [M^p]_{i,k} \times m_{k,j} = [M^{p+1}]_{i,j}.$$

par définition du produit matriciel.

Donc, par principe de récurrence, $\forall p \in \mathbb{N}^*$, les coefficients de M^p représentent le nombres de chaînes de longueur p entre chaque sommets de G. □

Remarque :
Si $\mathcal{G}$ est un graphe non orienté ayant n sommets, et M sa matrice d'adjacence, alors $\mathcal{G}$ est connexe si, et seulement si, $\sum_{k=0}^{n-1} M^k \in \mathcal{M}_n(\mathbb{R}_+^*)$.

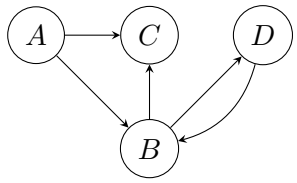
1.2 Graphes orientés

1.2.1 Définition et représentation graphique

Définition 1.6 :
Un graphe orienté est un ensemble de sommets (*vertex*), reliés par des arcs (*arc*). Autrement dit, un graphe orienté est la donnée d'un couple $(\mathcal{S}, \mathcal{A})$ où $\mathcal{S}$ est l'ensemble des sommets et $\mathcal{A}$ est l'ensemble des arcs, et $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ (donc un arc est un couple de deux sommet, c'est-à-dire avec une orientation).

Un arc est donc un couple de la forme (S_i, S_j) (ou $S_i \to S_j$). Si l'arc relie le même sommet, il s'agit d'une *boucle*.

Exemple 1.7 :



Il s'agit d'un graphe orienté non-complet. Les sommets sont : $\{A, B, C, D\}$ et les arcs sont : (A, B) ; (A, C) ; (B, C) ; (B, D) et (D, B) ou également notés $A \to B$; $A \to C$; $B \to C$; $B \to D$ et $D \to B$.

Définition 1.7 (Degré, Sommets adjacents, Chemin, Circuit, Longueur) :
Soit $G = (\mathcal{S}, \mathcal{A})$ une graphe orienté.

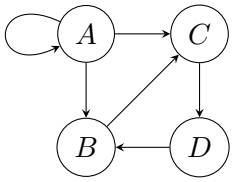
- Le *degré sortant* (resp. entrant) d'un sommet est le nombre d'arcs sortants (reps. entrants) de ce sommet, noté $d_+(S)$ (resp. $d_-(S)$)
- Le *degré* (ou valence) d'un sommet est la somme des degrés sortant et entrant de ce sommet, noté $d(S) = d_+(S) + d_-(S)$
- Un sommet S_i est *adjacent* à un sommet S_j s'il existe un arc reliant le sommet S_i au sommet S_j
- Un *chemin* entre un sommet S_i et un sommet S_j est une succession d'arcs consécutifs permettant de relier S_i à S_j
- Un *circuit* est un chemin partant et arrivant au même sommet. S'il ne passe jamais deux fois par le même arc il est qualifié de *circuit simple*. S'il ne passe jamais deux fois par le même sommet (excepté l'extrémité) il est qualifié de *circuit élémentaire*.
- La *longueur d'un chemin* entre deux sommets S_i et S_j est égal au nombre d'arcs formant ce chemin.

Exemple 1.8 :
Dans le graphe de l'exemple précédent, le degré sortant de B vaut $d_+(B) = 2$, tandis que le degré entrant vaut $d_-(B) = 2$, ainsi le degré de B vaut 4.

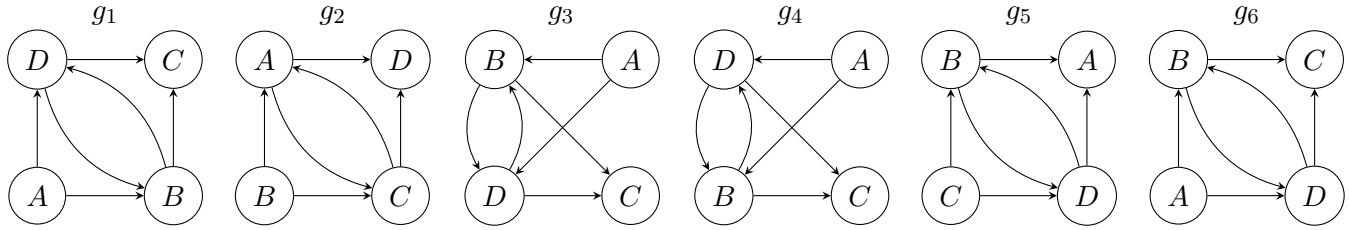
Le sommet C est adjacent au sommet A, tandis que le sommet B n'est pas adjacent au sommet C.

Il existe un chemin de A vers D de longueur 2 composé des arcs (A, B) et (B, D). Il n'existe pas de chemin de D vers A.

Exercice 3 :
Décrire succinctement le graphe suivant :



Exercice 4 :
Parmi ces graphes, lesquels sont identiques ?

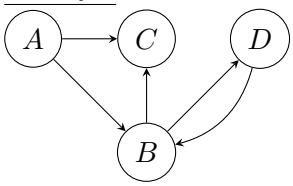


1.2.2 Représentation matricielle

Définition 1.8 :
Soit un graphe $G = (\mathcal{S}, \mathcal{A})$ avec $n = \mathrm{Card}(\mathcal{S})$ le nombre de sommets.
La matrice d'adjacence $M = (m_{i,j})_{1 \leq i,j \leq n} \in \mathcal{M}_n(\mathbb{N})$ du graphe G est définie par

$$\forall i, j \in \{1, \ldots, n\},\ m_{i,j} = \begin{cases} 1 & \text{si l'arc } (S_i, S_j) \text{ existe} \\ 0 & \text{sinon} \end{cases}$$

Exemple 1.9 :



a pour matrice d'adjacence $M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Proposition 1.7 (Ligne et colonne de 0 dans une matrice d'adjacence) :
Soit G un graphe orienté, n le nombre de sommets de G et M sa matrice d'adjacence. Soit $i, j \in \{1, \dots, n\}$.

(i) Si la ligne i est remplie de zéro, aucun arc ne sort du sommet S_i,

(ii) Si la colonne j est remplie de zéro, aucun arc ne rentre au sommet S_j,

Exercice 5 :
Considérons $\mathcal{S} = \{A, B, C, D, E\}$ et $\mathcal{A} = \{(A, B), (A, D), (A, E), (B, D), (B, E), (C, A), (C, B), (D, A)\}$. Dessiner le graphe $G = (\mathcal{S}, \mathcal{A})$ et préciser sa matrice d'adjacence.

Proposition 1.8 :
Soient $G = (\mathcal{S}, \mathcal{A})$ un graphe orienté et M sa matrice d'ajdacence. Alors

$$\forall i, j \in \{1, \dots, n\},\ d_+(S_i) = \sum_{k=1}^{n} m_{i,k} \quad \text{et} \quad d_-(S_j) = \sum_{k=1}^{n} m_{k,j}.$$

et $\mathrm{tr}(M)$ est le nombre de boucles du graphe G.

Démonstration :
Les deux premières assertions découlent de la définition de $m_{i,j}$

La troisième assertion découle de l'assertion "une boucle existe au sommet S_i ssi $m_{i,i} = 1$". □

1.3 Graphes pondérés

Souvent, on peut vouloir ajouter des informations pour caractériser des arcs/arêtes (distance entre deux villes sur une carte, débit entre deux serveurs, définition d'automates, etc.)

Définition 1.9 (Graphe pondéré) :
Un graphe est dit pondéré lorsqu'un poids (une valeur) est associée à chaque arcs (resp. arêtes). On associe alors à chaque arc (resp. arête) $a_{i,j}$ un poids $\omega(a_{i,j}) = e_{i,j}$.

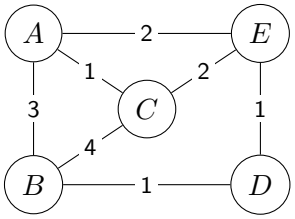
Un graphe pondéré peut être orienté, ou non.

Remarque :
En fait, on pourrait associer n'importe quoi à une arête. On parle alors de *graphe étiqueté*. Les graphes pondérés sont une sous-catégorie des graphes étiquetés : les graphes pondérés sont des graphes étiquetés par des nombres.

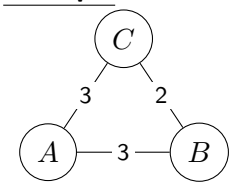
Remarque :
Dans un graphe pondéré, on peut associer un poids au chemin entre deux sommets S_i et S_j en faisant la somme des pondérations des arcs (resp. arêtes) formant le chemin.

Exemple 1.10 :

On considère $\mathcal{S} = \{A, B, C, D\}$ et $\mathcal{A} = \{\{A, B, 3\}; \{A, C, 1\}; \{A, E, 2\}; \{B, C, 4\}; \{B, D, 1\}; \{C, E, 2\}; \{D, E, 1\}\}$.
Le graphe $G = (\mathcal{S}, \mathcal{A})$ peut alors être représenté par :

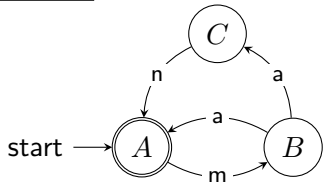


Exemple 1.11 :



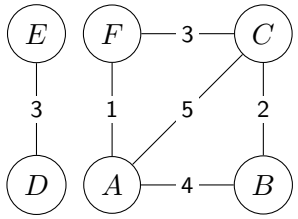
Il s'agit d'un graphe non-orienté pondéré.
La cycle élémentaire $A-B-C-A$ est de longueur 3 et de poids 8.

Exemple 1.12 :



Il s'agit d'un graphe orienté étiqueté.
Le circuit $A \to B \to A \to B \to C \to A$ permet de définir le mot `"maman"`.
Lorsqu'il est défini un sommet initial (flèche *start*) et un sommet final (double cercle), le graphe décrit un automate.

Exercice 6 :
Décrire succinctement le graphe suivant (ainsi que le cycle simple de poids maximal) :



Exercice 7 :
Définir un automate qui reconnaisse les mots ("chat", "chiot").

On peut définir une matrice d'étiquettes de sorte que les coefficients de cette matrice $(E_{i,j})_{i,j}$ soient l'étiquette de chaque arc (ou arête), et selon le contexte ($\emptyset$, 0, ∞, ...) si l'arc ou l'arête n'existe pas.

On peut également considérer que l'ensemble $\mathcal{A}$ est composé des triplets $(S_i, S_j, e_{i,j})$ ou $\{S_i, S_j, e_{i,j}\}$.

Exemple 1.13 :

La matrice étiquette de l'exemple précédent sera $E = \begin{pmatrix} \infty & 4 & 5 & \infty & \infty & 1 \\ 4 & \infty & 2 & \infty & \infty & \infty \\ 5 & 2 & \infty & \infty & \infty & 3 \\ \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & 3 & \infty & \infty \\ 1 & \infty & 3 & \infty & \infty & \infty \end{pmatrix}$

2 Implémentation Python

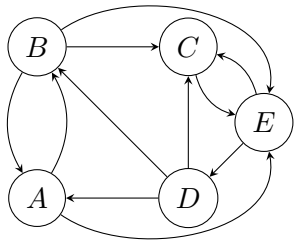
Il semble naturel d'implémenter les graphes à l'aide de leur matrice d'adjacence mais selon les cas, une représentation par liste d'arcs peut être plus pertinente.

2.1 Représentation par matrice ou liste d'adjacence

La représentation matricielle découle logiquement de la matrice d'adjacence du graphe. Néanmoins lorsque le graphe comporte de nombreux sommets et peu d'arcs ou arêtes (*i.e.* $\mathrm{Card}(\mathcal{S})^2 \gg \mathrm{Card}(\mathcal{A})$), la matrice comporte beaucoup de 0 (on parle de matrice creuse). Cette information occupe de la place mémoire inutile et on aura tendance à privilégier une représentation par listes d'adjacence.

Représenter un graphe par listes d'adjacence consiste à associer pour chaque sommets, les sommets qui lui sont adjacents.

Exemple 2.1 :



Considérant le graphe G. En associant le sommet A au nombre 0, B à 1, etc. le graphe G a pour matrice d'adjacence

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

Enfin on peut le représenter à l'aide du tableau des listes d'adjacence `T=[[1,4],[0,2,4],[4],[0,1,2],[2,3]]`.

Remarque :
Afin de connaître à coup sûr le nombre de sommets dans le graphe, il faut ajouter des listes vides dans le tableau des listes d'adjacence afin de vérifier $\mathrm{Card}(\mathcal{S}) = len(T)$. En effet, si le graphe est non-connexe, on pourrait perdre toutes les sommets isolés supérieurs au sommet le plus grand utilisé dans un arc (ou arête).

Exercice 8 :
Tracer le graphe défini par le tableau de listes d'adjacence `T=[[1,2,4],[],[3,4],[0,1,2],[]]`.

Exercice 9 :
Définir la fonction `mat2listes(mat: list) -> list`, qui renvoie le tableau des listes d'adjacence à partir de la matrice d'adjacence.

2.2 Algorithmes de parcours

Un algorithme de parcours consiste à passer d'un sommet courant à un autre par un arc ou une arête. Chaque arc ou arête n'est utilisable qu'une seule fois.

À partir de là, on peut classer les sommets selon trois états qu'ils auront durant le parcours.

- le sommet n'a pas encore été visité.
- le sommet a été visité mais mais tous ses voisins n'ont pas encore été visités. Auquel cas, on considérera que le sommet n'a pas été traité.
- le sommet est traité (visité et tous ses voisins ont été visités).

2.2.1 Parcours en largeur

Présentation :

Le parcours en largeur (développé par Zuse en 1945) consiste à visiter tous les voisins du sommet courant avant de passer aux voisins du voisin. Ainsi ce parcours permet de connaître naturellement l'ensemble des sommets à une distance 1 du sommet initial, puis ceux à distance 2, puis 3, etc. (on fait une liste en "cercle concentrique" à partir du sommet de départ de tous les sommets du graphes). Généralement, les voisins des sommets visités sont ajoutés dans une file (*FIFO : First In, First Out*).

Intérêt :

Il est utilisé pour détecter les cycles, pour déterminer la distance par rapport au sommet initial, et également pour déterminer si un graphe est connexe.

Pseudo-Code :

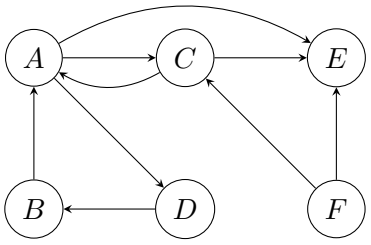
En pseudo-code, l'algorithme en largeur se décrit ainsi :

- ◇ marquer le sommet initial comme visité
  ajouter le sommet initial à la file des sommets visités
- ◇ tant que la file contient des sommets, définir le premier sommet entré dans la file comme sommet courant, s'il n'a pas déjà été traité, et le défiler
  - pour tous les voisins du sommet courant
    - si le voisin n'a pas été traité
      - les marquer comme visités
      - les ajouter à la file des sommets visités en notant le sommet courant comme prédécesseur
    - sinon
      - marquer le sommet courant comme traité
      - traiter le sommet courant
- ◇ récupérer le traitement effectué sur les sommets traités

Complexité temporelle :

La complexité temporelle est en $O(|\mathcal{S}| + |\mathcal{A}|)$.

Exemple 2.2 :



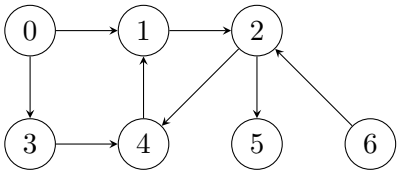
À partir du sommet A, compléter le déroulé de l'algorithme du parcours en largeur (en sélectionnant les sommets voisins dans l'ordre alphabétique).

Sommet courant	File des sommets visités	Sommets traités
A	C, D, E	A
C	D, E, A, E	A, C
D	E, A, E, B	A, C, D
E	A, E, B	A, C, D, E
A	E, B	A, C, D, E
E	B	A, C, D, E
B	A	A, C, D, E, B
A		A, C, D, E, B

Les sommets traités A, C, D, E et B sont les sommets accessibles depuis le sommet initial A.

Exercice 10 :

Écrire le parcours du graphe en largeur suivant en partant du sommet 0.



Code Python :

Remarque :
Pour pouvoir faire la liste des sommets visités, une file est le plus adaptée. Il n'y a pas vraiment de tels objets naturellement dans python. On va donc utiliser des listes mais que l'on va penser et utiliser comme des files.

```python
def parcoursLargeur(adjacences:list, depart:int) -> list :
    """
    adjacences : matrice d'adjences du graphes
    depart : indice du noeud de départ
    """
    n = len(adjacences) # nombres de sommets
    sommetsAVisites=[] # listes des sommets a visités
    sommetCourant = depart # sommet courant analysé
    sommetsTraites = [depart] # liste des sommets traités
    for k in range(n) : # Création de la liste des sommet a vistés initiale à partir du départ
        if adjacences[sommetCourant][k] != 0 :
            sommetsAVisites = [k] + sommetsAVisites
    while sommetsAVisites != [] :
        sommetCourant = sommetsAVisites.pop() # prend le premier sommet à visités
        if sommetCourant not in sommetsTraites :
            sommetsTraites.append(sommetCourant)
            for k in range(n) :
                if adjacences[sommetCourant][k] != 0 :
                    sommetsAVisites = [k] + sommetsAVisites
    return(sommetsTraites)
```

Exercice 11 :
Proposer une fonction `parcoursLargeurDist(adj:list, dep:int) -> list` qui modifie la fonction `parcoursLargeurDist` pour donner la liste des sommets atteignables depuis le sommet de départ `dep` et le distance au sommet initiale (on renverra une liste de couple (sommet, distance)).

2.2.2 Parcours en profondeur

Présentation :

Le parcours en profondeur (dont les prémices furent l'algorithme de Trémaux vers 1880) consiste à visiter le graphe le plus loin que l'on puisse, puis si on ne peut plus continuer, rebrousser chemin et recommencer sur un chemin non visité dès que cela est possible. On part donc d'un sommet courant et l'on visite un de ses voisins qui deviendra le sommet courant. Dès qu'il n'y a plus de voisins, on remonte au sommet précédent pour choisir un autre voisin.

Généralement, les voisins des sommets visités sont ajoutés dans une pile (*FILO : First In, Last Out*). L'utilisation d'une pile pour les sommets visités est une des différences majeurs avec le parcours en largeur.

Intérêt :

Le parcours en profondeur est utilisé pour détecter les cycles, découvrir un labyrinthe, et également pour déterminer si un graphe est connexe.

Pseudo-Code :

À partir d'un sommet initial, l'algorithme en profondeur se décrit ainsi :

◇ marquer le sommet initial comme visité
ajouter le sommet initial à la pile des sommets visités

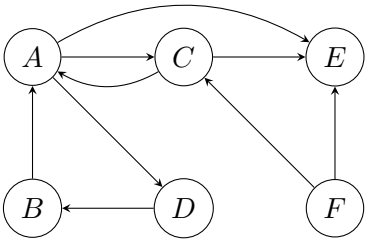
◇ tant que la pile contient des sommets, définir le dernier sommet entré dans la pile comme sommet courant, s'il n'a pas déjà été traité, et le dépiler
  pour tous les voisins du sommet courant
    si le voisin n'a pas été traité
      les marquer comme visités
      les ajouter à la pile des sommets visités en notant le sommet courant comme prédécesseur
    sinon
      marquer le sommet courant comme traité
      traiter le sommet courant

◇ récupérer le traitement effectué sur les sommets traités

Remarque :
Parfois, on peut vouloir traiter le sommet courant avant d'ajouter les sommets voisins, et/ou marquer les arcs/arêtes comme visitées afin de pouvoir modifier le traitement des sommets voisins.

Complexité temporelle :
La complexité temporelle est en $O(|\mathcal{S}| + |\mathcal{A}|)$.

Exemple 2.3 :
À partir du sommet A, le déroulé de l'algorithme du parcours en profondeur (en parcourant les sommets voisins dans l'ordre inverse alphabétique) est le suivant :



Sommet courant	Pile des sommets à visités	Sommets traités
A	C, D, E	A
E	C, D	A, E
D	C, B	A, E, D
B	C, A	A, E, D, B
A	C	A, E, D, B
C	E	A, E, D, B, C
E		A, E, D, B, C

Là aussi, le sommet F n'a pas été visité, il n'est donc pas atteignable depuis le sommet A.

Remarque :
Le choix de l'ordre du parcours en profondeur (de la route en suivre en premier) est un choix et un autre

pourrait être fait. Dans l'exemple précédent, on fait les chemins $A \to E$, $A \to D \to B$ et $A \to C(\to E)$. On aurait pu choisir de commencer par le chemin commençant par C et l'explorer en entier en profondeur. Donc faire les chemins $A \to C \to E$ et $A \to D \to B$. L'avantage de la première version est de voir E comme étant à distance 1 de A. Mais il ne parcours pas en entier le chemin passant par C. La deuxième version permet de visiter tous les points atteignables depuis A avec le moins de chemins possibles (en seulement deux chemins au lieu de trois). Chaque version à ses avantages et ses inconvénients. Ce sont des choix.

Ces choix proviennent de la façon dont la pile des sommets à visités est remplies (par la gauche ou la droite ; par ordre alphabétique ou non etc).

Version récursive :

Cet algorithme possède une version récursive naturelle qui ne nécessite plus de pile des sommets visités puisque les appels récursifs fournissent cette pile.

```
parcoursProfondeur()
    marquer le sommet courant comme visité
    pour tout sommet voisin du sommet courant,
        s'il n'a pas déjà été parcouru à partir de ce sommet voisin,
        parcourir en profondeur le graphe à partir de ce voisin
```

Analysons sa terminaison et son hérédité.

- terminaison : si le sommet courant a été traité ou qu'il ne possède pas de voisin, on arrête (le parcours continuera avec le voisin d'un précédent sommet)
- hérédité : si le sommet possède des voisins non visités, on choisit l'un de ses voisins qui deviendra le sommet courant de parcours.

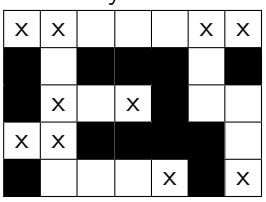
Code Python :

```python
def parcoursProfondeur(adj : list, dep:int) -> list :
    n = len(adj)
    sommetCourant = dep
    sommetsAVisites = []
    sommetsTraites = [dep]
    for k in range(n) :
        if adj[dep][k] != 0 :
            sommetsAVisites.append(k) # rajoute les nouveaux sommets à visiter à la fin de la pile.
    while sommetsAVisites != [] :
        sommetCourant = sommetsAVisites.pop() # Récupère le sommets à visité en haut de la pile et l'enlève
        if sommetCourant not in sommetsTraites :
            sommetsTraites.append(sommetCourant) # rajoute le nouveau sommet courant en haut de la pile des sommets traités
            for k in range(n) :
                if adj[sommetCourant][k] != 0 :
                    sommetsAVisites.append(k)
    return(sommetsTraites)
```

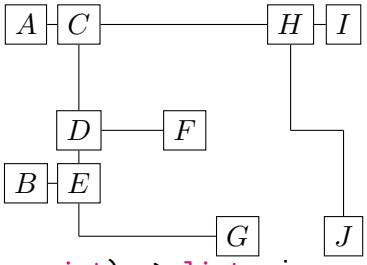
Exercice 12 :

Nous allons implémenter de manière récursive le parcours en profondeur afin de déterminer un chemin de sortie d'un labyrinthe. On considère le labyrinthe suivant dont l'entrée est en haut à gauche et la sortie en bas à droite. On suppose que l'on se déplace que verticalement ou horizontalement.
On peut alors représenter le labyrinthe en prenant pour sommet les entrées, sorties, intersections et cul-de-sac.

labyrinthe



graphe



Proposer une fonction `labyrinthe(adj:list, dep:int, arr:int) -> list` qui permet de donner un chemin allant de `dep` à `arr`.

2.2.3 Recherche de plus court chemin

2.2.3.1 Algorithme de Dijkstra

L'algorithme de Dijkstra a été créé en 1959 par Edsger Dijkstra. Il donne la liste de tous les plus court chemin dans un graphe pondéré à partir d'une source donnée. C'est une variation du parcours en largeur. On se déplace en cercle concentrique depuis la source, à la différence que le rayon des cercles correspond au poids total des chemins parcourus et plus en terme de nombres d'arcs.

L'algorithme peut se décrire en pseudo-code ainsi :

- associer à chacun des sommets un poids infini excepté pour le sommet initial de poids nul et définir le sommet initial comme sommet courant (*i.e.* $p(Si) = 0$ et $\forall S \in \mathcal{S} \setminus \{Si\},\ p(S) = \infty$)
- le traitement du sommet courant consiste en :
  pour chaque sommet i relié au sommet courant, qui n'a pas encore été traité
    si le poids du sommet courant plus le poids de l'arête reliant ses deux sommets est inférieur au poids du sommet i, changer le poids du sommet i par cette somme et retenir le sommet courant comme sommet antérieur au sommet i
  (*i.e.* $\forall S \in V\text{oisins}(Sc),\ p(S) = \min(p(S), p(Sc) + p(Sc - S))$)
- choisir pour sommet courant un sommet de poids minimal qui n'a pas encore été traité

  Recommencer les deux dernières étapes jusqu'à atteindre le sommet final.

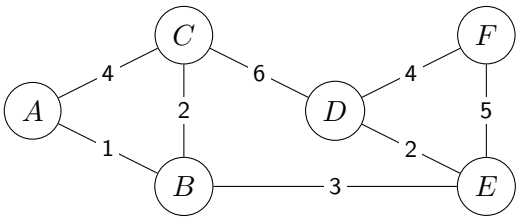
Si on note $\mathcal{S}'$ l'ensemble des sommets restant à traiter, p_{ci} le poids reliant le sommet courant et le sommet i, et w_i le poids minimal de la chaîne menant du sommet initial au sommet i dans $\mathcal{G}'$ (où $\mathcal{G}'$ est le sous-arbre en construction).

L'algorithme devient :

1. $\forall i$, $w_i \leftarrow +\infty$ sauf $w_0 \leftarrow 0$; $s_c \leftarrow s_0$ et $\mathcal{S}' = \mathcal{S} \setminus \{s_c\}$
2. pour chaque s_i relié à s_c tq $s_i \in \mathcal{S}'$
   si $w_c + p_{ci} < w_i$ alors $w_i \leftarrow w_c + p_{ci}$ et retenir s_c comme sommet antérieur à s_i
3. s_c tq $w_c = \min_{s_i \in \mathcal{S}'}(w_i)$ et $\mathcal{S}' = \mathcal{S}' \setminus \{s_c\}$
   Recommencer jusqu'à ce que $\mathcal{S}' = \emptyset$.

Exemple 2.4 :

Trouver la chaîne minimale de A à F considérant ce graphe



Ce qui donne le parcours suivant :

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
	$(1, A)$	$(4, A)$	∞	∞	∞
		$(3, B)$	∞	$(4, B)$	∞
			$(9, C)$	$(4, B)$	∞
			$(6, E)$		$(9, E)$

Et donc le chemin de poids minimum allant de A à F est $A \to B \to E \to F$ de poids 9.

Exercice 13 :

Déterminer la chaîne de poids minimal reliant A et F pour le graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ avec $\mathcal{S} = \{A, B, C, D, E, F\}$ et $\mathcal{A} = \{(A, B, 1), (A, C, 7), (A, F, 18), (B, C, 5), (B, D, 8), (B, F, 12), (C, E, 3), (D, E, 2), (E, F, 1)\}$.

Code Python :

```
def Dijkstra(si:int, sf:int, etiq:list) -> tuple :
    n = len(etiq) # nombre de sommets
    sommetsATraites = [k for k in range(n)]
    sommetsATraites.remove(si)
    poids = [float("inf") for k in range(n)]
    poids[si] = 0
    provenance = [-1 for k in range(n)]
    provenance[si] = si
    for s in sommetsATraites : # creation de la liste des poids et provenance a l'étape initiale
        if etiq[si][s] != float("inf") :
            poids[s] = etiq[si][s]
            provenance[s] = si
    while sommetsATraites != [] :
        # recherche des sommets de poids minimal dans les sommets à traités
        pc = float("inf") # poids du sommets courant
        sc = -1 # nouveau sommet courant
        for s in sommetsATraites :
            if poids[s] <= pc :
                pc = poids[s]
                sc = s # sc = sommets de poids minimal
        # sc, pc = sommet et poids de poids minimal parmi les sommets encore a traités
        sommetsATraites.remove(sc) # on va traité sc, donc on enlève sc des sommets à traités
        for s in sommetsATraites :
            if etiq[sc][s]+pc < poids[s] :
                poids[s] = etiq[sc][s]+pc
                provenance[s] = sc
    pf = poids[sf] # poids final
    chemin = [sf] # chemin parcouru
    s = sf
    while s != si :
        chemin = [provenance[s]] + chemin
        s = provenance[s]
    return(pf,chemin)
```

2.2.3.2 Algorithme A*

L'algoithme de Dijkstra donne tous les chemins démarrant de la source. Il est donc très gourmand en ressource et nécessite de faire beaucoup de calculs quand seulement un seul chemin nous intéresse. Inutile de faire la liste de tous les chemins depuis Paris vers toutes les villes pour connaître le meilleur itinéraire Paris-Lyon.

L'algorithme A* se propose de remédier à ce problème en "guidant l'algorithme pour se déplacer dans la bonne direction" (et donc ne pas parcourir certains chemins qui ne seront pas utile de toute façon).

Pour ça, on utilise une fonction *heuristique* qui donne une estimation de la distance entre chaque sommet et la destination pour essayer de guider le déplacement : on essaie de diminuer cette estimation de la distance pour se rapprocher de la destination.

> Définition 2.1 (Heuristique) :
> Heuristique (adj) : Qui sert à la découverte ; qui est propre à guider une recherche ou à vérifier une hypothèse. Méthode heuristique, qui procède par hypothèses provisoires, approches, trouvailles successives

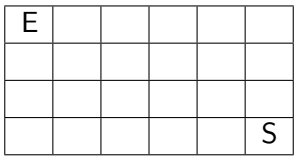
dans la résolution d'un problème. (Dictionnaire de l'académie française).

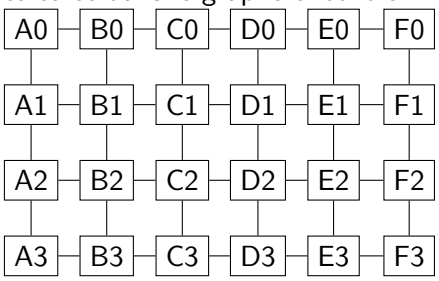
Dans le cas d'une heuristique dans un parcours de graphe, la quantité de l'heuristique correspondant à la destination doit être nulle.

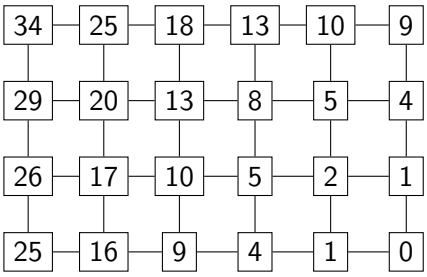
Exemple 2.5 :

Prenons un cas simple. Considérons le quadrillage suivant où l'on ne peut se déplacer que verticalement et horizontalement pour rejoindre l'entrée $(0, 0)$ à la sortie $(6, 4)$. Chaque case formera un sommet (de $A0$ à $F3$) de notre graphe. On supposera que tous les chemins ont le même poids 1.

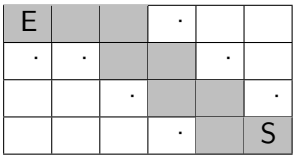
On prendra pour fonction heuristique le carrée de la distance euclidienne entre la case courante (i, j) et la case de sortie (*i.e.* la distance euclidienne, "en ligne droite" entre une case donnée et la sortie). Cette fonction heuristique a été pré-calculée dans le graphe ci-contre :







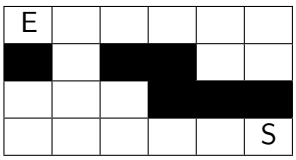
Le chemin le plus court est donc A0-B0-C0-C1-D1-D2-E2-E3-F3 de poids 8.
Chemin matérialisé ci-contre en gris clair (les cases pointées sont celles ajoutées à la file de priorité).

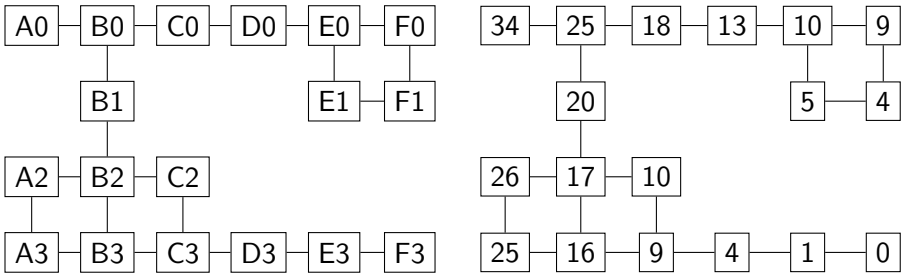


Le chemin se rapproche le plus possible de la diagonale à cause de l'heuristique (ligne droite). Seules 17 sommets ont été traités, ce qui est beaucoup plus efficace que l'algorithme de Dijkstra qui aurait entraîné un traitement de tous les sommets (puisque E et S sont les sommets les plus éloignés).

Sommet et poids courant	(sommet,poids,heuristique) à visiter
$(A0, 0)$	$(B0, \underbrace{1, 25}_{26})$, $(A1, \underbrace{1, 29}_{30})$
$(B0, 1)$	$(A1, 1, 29)$, $(C0, \underbrace{2, 18}_{20})$, $(B1, \underbrace{2, 20}_{22})$
$(C0, 2)$	$(A1, 1, 29)$, $(B1, 2, 20)$, $(D0, \underbrace{3, 13}_{16})$, $(C1, \underbrace{3, 13}_{16})$
$(C1, 3)$	$(A1, 1, 29)$, $(B1, 2, 20)$, $(D0, 3, 13)$, $(E0, \underbrace{4, 10}_{14})$, $(D1, \underbrace{4, 8}_{12})$
$(D1, 4)$	$(A1, 1, 29)$, $(B1, 2, 20)$, $(D0, 3, 13)$, $(E0, 4, 10)$, $(E1, \underbrace{5, 5}_{10})$, $(D2, \underbrace{5, 5}_{10})$
$(D2, 5)$	$(A1, 1, 29)$, $(B1, 2, 20)$, $(D0, 3, 13)$, $(E0, 4, 10)$, $(E1, 5, 5)$, $(E2, \underbrace{6, 2}_{8})$, $(D3, \underbrace{6, 4}_{10})$
$(E2, 6)$	$(A1, 1, 29)$, $(B1, 2, 20)$, $(D0, 3, 13)$, $(E0, 4, 10)$, $(E1, 5, 5)$, $(D3, 6, 4)$, $(F2, \underbrace{7, 1}_{8})$, $(E3, \underbrace{7, 1}_{8})$
$(E3, 7)$	$(A1, 1, 29)$, $(B1, 2, 20)$, $(D0, 3, 13)$, $(E0, 4, 10)$, $(E1, 5, 5)$, $(D3, 6, 4)$, $(F2, 7, 1)$, $(F3, 8, 0)$

Exemple 2.6 :
On considère le même quadrillage auquel des murs ont été ajoutés. Ces murs n'affectent par la fonction heuristique qui reste identique.

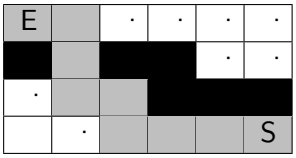




Déroulons l'algorithme A* :

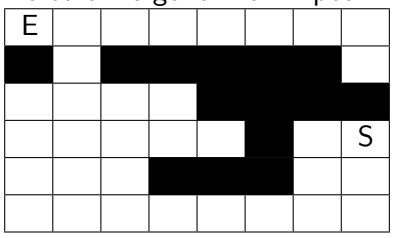
(sommet, poids)	(sommet, poids, heuristique)
$(A0, 0)$	$(B0, 1, 25)$
$(B0, 1)$	$(B1, 2, 20)$, $(C0, 2, 25)$
$(B1, 2)$	$(C0, 2, 25)$, $(B2, 3, 17)$
$(B2, 3)$	$(C0, 2, 25)$, $(C2, 4, 10)$, $(B3, 4, 16)$
$(C2, 4)$	$(C0, 2, 25)$, $(B3, 4, 16)$, $(C3, 5, 9)$
$(C3, 5)$	$(C0, 2, 25)$, $(B3, 4, 16)$, $(D3, 6, 4)$
$(D3, 6)$	$(C0, 2, 25)$, $(B3, 4, 16)$, $(E3, 7, 1)$
$(E3, 7)$	$(C0, 2, 25)$, $(B3, 4, 16)$, $(F3, 8, 0)$

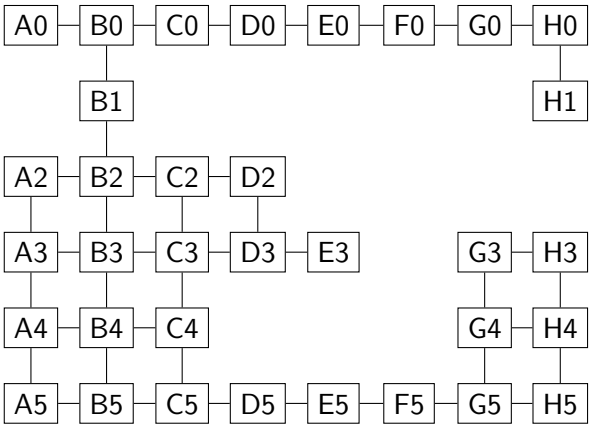
Le chemin le plus court est donc A0-B0-B1-B2-C2-C3-D3-E3-F3 de poids 8.
Chemin matérialisé ci-contre en gris clair (les cases pointées sont celles ajoutées à la file de priorité).



Exercice 14 :

Dérouler l'algorithme A* pour l'exemple suivant :





Code Python :

```
1  def Astar(si:int, sf:int, graph:list, heur:"function") -> tuple :
2      """
3      si : int - Sommet source
4      sf : int - Sommet destination
5      graph : list - Tableaux des adjacences avec poids
6      heur : "function" - Fonctions d'heuristique pour guider le chemin
7      """
8      sommets = [[si,0]] # Liste des sommets visités et de leur poids total
9      Avisiter = [] # liste des sommet à visiter de la forme (somme, poids total, sommet d'origine)
10     n = len(graph)
11     for k in range(n) : # Création des premiers sommets à visiter à partir de la source
12         if graph[si][k] < float("inf") :
13             Avisiter = Avisiter + [[k,graph[si][k],si]] # (sommet, poids total, origine)
14     sc = [si,0] # sommet courant
15     while sc[0] != sf : # tant que le sommet courant n'est l'arrivée
16         for k in range(n) : # faire la recherche de tous les sommets "suivant" à partir du sommet
   courant
17             if graph[sc[0]][k] < float("inf") :
18                 Avisiter = Avisiter + [[k,sc[1]+graph[sc[0]][k],sc[0]]]
19         # Recherche du prochain sommet courant (sommet avec le plus poids et heuristique
20         p = float("inf")
21         indSommet = -1 # indice du prochain sommet dans la liste des sommets à visiter
22         for k in range(len(Avisiter)) :
23             if Avisiter[k][1]+heur(Avisiter[k][0])<p :
24                 p=Avisiter[k][1]+heur(Avisiter[k][0])
25                 indSommet = k
26         sc = Avisiter[indSommet] # définition du nouveau sommet courant
27         sommets = sommets + [sc] # on rajoute le prohain sommet à être visité dans la liste des
   sommets visités
28         Avisiter.remove(Avisiter[indSommet]) # On enlève de la liste des sommets à visiter
   le prochain qui va être visité
29     # Création du chemin à parcourir pour aller du sommet initial à l'arrivée
30     chemin = [sf]
31     s = sf
32     while s != si : # on remonte les origines des sommets successifs visités
33         k=0
34         # Recherche de l'indice de sommets précédent du chemin dans la liste des sommets visités
35         while sommets[k][0] != s :
36             k = k+1
37         s = sommets[k][2] # nouveau sommet antécédent
38         chemin = [sommets[k][2]] + chemin # On rajoute l'antécédent au début dans le chemin
39     return(sommets[-1][1],chemin)
```