Révisions d'informatique commune

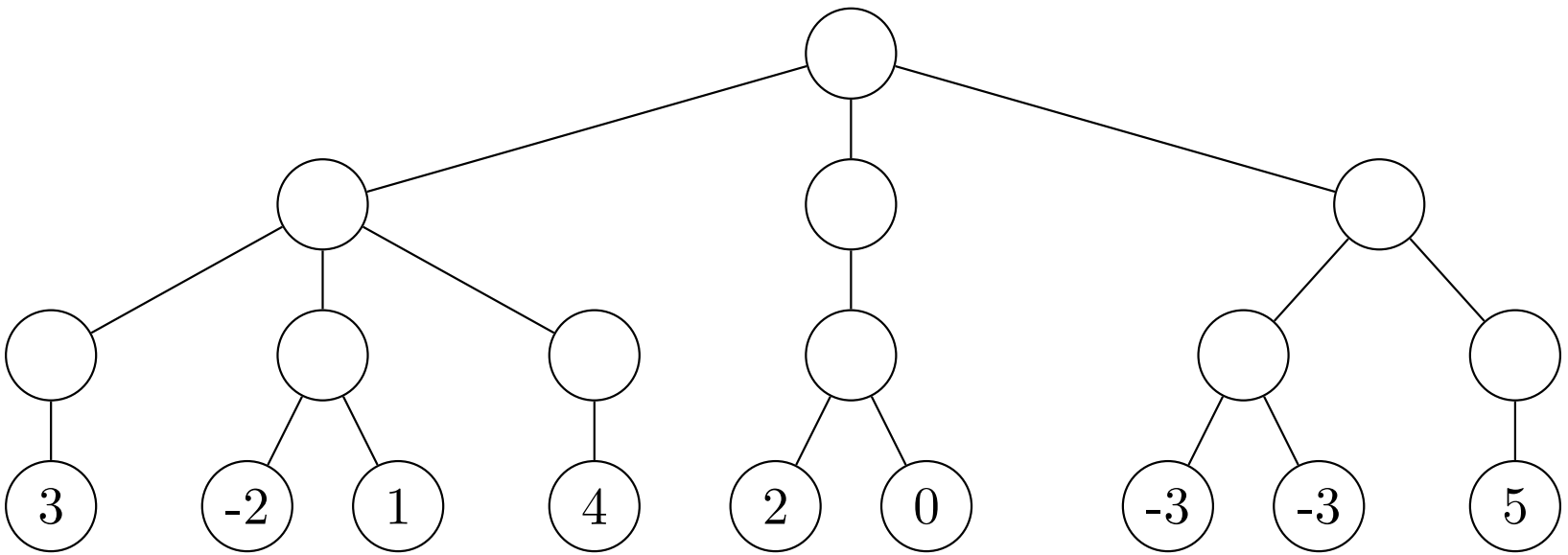
Quentin Fortier

April 5, 2023

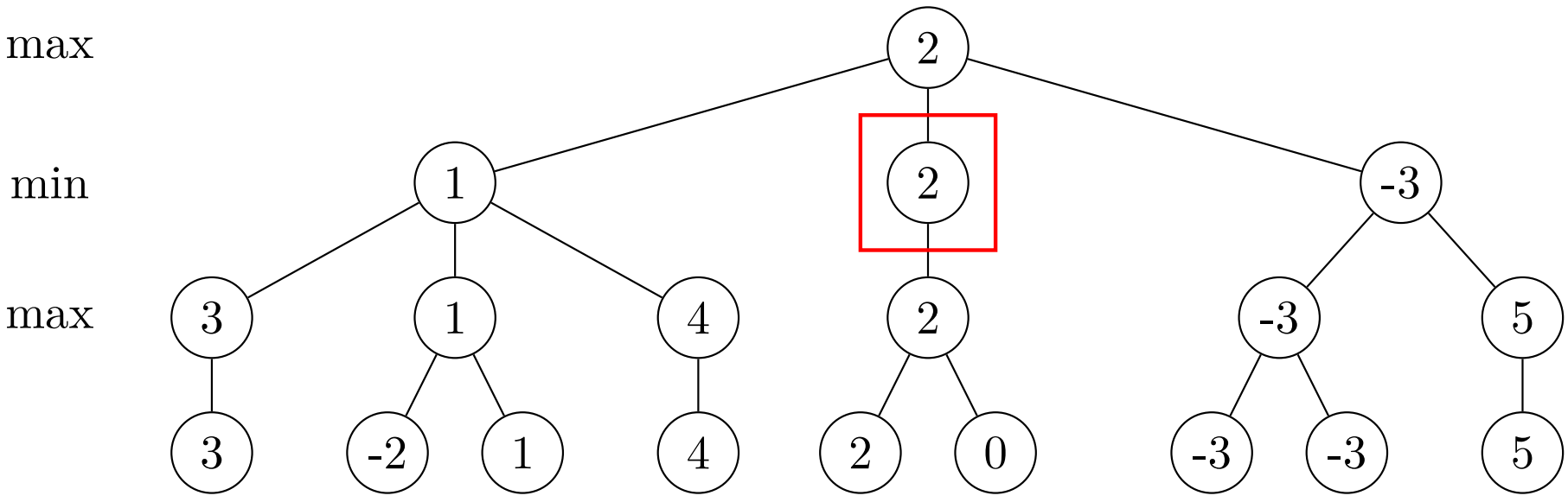
Minmax

Question

Compléter l'arbre mixmax ci-dessous, où le joueur qui joue en premier souhaite maximiser l'heuristique.



Minmax



Complexité

Réponse de nombreux élèves au dernier devoir :

```
def normalisation(M):
    n, p = len(M), len(M[0])
    for i in range(n):
        for j in range(p):
            M[i][j] = (M[i][j] - moyenne(M, j)) / ecart_type(M, j)
    return M
```

Complexité : $O(np)$.

Question

Quels sont les problèmes ?

Complexité

Vous pouvez indiquer la complexité de chaque ligne de code, dans une couleur différente :

```
def normalisation(M):
    n, p = len(M), len(M[0])
    M2 = grille(n, p)
    for j in range(p): # O(np)
        mu = moyenne(M, j) # O(n)
        sigma = ecart_type(M, j) # O(n)
        for i in range(n): # O(n)
            M2[i][j] = (M[i][j] - mu) / sigma
    return M2
```

Complexité

Vous pouvez indiquer la complexité de chaque ligne de code, dans une couleur différente :

```
def normalisation(M):
    n, p = len(M), len(M[0])
    M2 = grille(n, p)
    for j in range(p): # O(np)
        mu = moyenne(M, j) # O(n)
        sigma = ecart_type(M, j) # O(n)
        for i in range(n): # O(n)
            M2[i][j] = (M[i][j] - mu) / sigma
    return M2
```

Conseil : écrire des commentaires concis sur des variables ou fonctions dont le rôle n'est pas évident.

Complexité

- Justifier tous les calculs de complexité, même brièvement.
- Prendre en compte la complexité des appels de fonctions.
- Une complexité $\mathrm{O}(n)$ est dite linéaire.
- Une complexité $\mathrm{O}(n^2)$ est dite quadratique.
- Une complexité $\mathrm{O}(a^n)$ avec a constante est dite exponentielle.

Complexité

- On multiplie les complexités de boucles imbriquées.
- On ajoute les complexités d'instructions les unes après les autres.

```
for i in range(n):
    for j in range(p):
        ...
```

$$\mathrm{O}(n) \times \mathrm{O}(p) = \mathrm{O}(np)$$

```
for i in range(n):
    ...
for i in range(p):
    ...
```

$$\mathrm{O}(n) + \mathrm{O}(p) = \mathrm{O}(n+p)$$

Complexité

Question

Quel est le problème avec cette fonction de recherche par dichotomie dans une liste triée ?

```
def dichotomie(L, x):
    n = len(L)
    if n == 0:
        return False
    m = n // 2
    if L[m] == x:
        return True
    if L[m] < x:
        return dichotomie(L[m+1:], x)
    return dichotomie(L[:m], x)
```

Complexité

Question

Quel est le problème avec cette fonction de recherche par dichotomie dans une liste triée ?

```
def dichotomie(L, x):
    n = len(L)
    if n == 0:
        return False
    m = n // 2
    if L[m] == x:
        return True
    if L[m] < x:
        return dichotomie(L[m+1:], x)
    return dichotomie(L[:m], x)
```

`L[i:j]` renvoie une liste contenant les éléments de L de l'indice i inclus à l'indice j exclus, en complexité $O(j - i)$.

Complexité

Version récursive avec des indices :

```
def dichotomie(L, x):
    def aux(i, j): # renvoie True si x est dans L[i:j]
        if i == j:
            return False
        m = (i + j) // 2
        if L[m] == x:
            return True
        if L[m] < x:
            return aux(m+1, j)
        return aux(i, m)
    return aux(0, len(L))
```

Complexité

Version itérative :

```
def dichotomie(L, x):
    i, j = 0, len(L)
    while i < j:
        m = (i + j) // 2
        if L[m] == x:
            return True
        if L[m] < x:
            i = m + 1
        else:
            j = m
    return False
```

Complexité

Question

Quelle est la complexité de la recherche par dichotomie sur une liste triée de taille n ?

Complexité

Question

Quelle est la complexité de la recherche par dichotomie sur une liste triée de taille n ?

À chaque itération, on divise la taille de la zone de recherche par 2.
Au bout de k itérations, la zone de recherche est donc au plus $\frac{n}{2^k}$.
La dichotomie s'arrête donc quand $\frac{n}{2^k} \leq 1$, c'est-à-dire quand $k \geq \log_2(n)$.
Il y a donc au plus $\lceil \log_2(n) \rceil$ itérations d'où une complexité $\boxed{\mathrm{O}(\log(n))}$.

Preuve

Définition

On dit qu'une fonction **termine** si elle ne peut pas faire de boucle infinie ou de récursion infinie.

Définition

Un **variant de boucle** est une quantité strictement monotone à chaque itération de boucle et qui permet de montrer que la boucle termine.

Preuve

Définition

On dit qu'une fonction **termine** si elle ne peut pas faire de boucle infinie ou de récursion infinie.

Définition

Un **variant de boucle** est une quantité strictement monotone à chaque itération de boucle et qui permet de montrer que la boucle termine.

Question

Quel variant de boucle peut-on utiliser pour montrer que la recherche par dichotomie termine ?

Preuve

Définition

On dit qu'une fonction est **correcte** si elle renvoie bien la valeur attendue.

Définition

Un **invariant de boucle** est une propriété qui est vraie avant et après chaque itération de boucle et qui permet de montrer que la boucle est correcte.

Preuve

Définition

On dit qu'une fonction est **correcte** si elle renvoie bien la valeur attendue.

Définition

Un **invariant de boucle** est une propriété qui est vraie avant et après chaque itération de boucle et qui permet de montrer que la boucle est correcte.

Question

Quel invariant de boucle peut-on utiliser pour montrer que la recherche par dichotomie est correcte ?

P : si `x` est dans `L` alors `x` est dans `L[i:j]`.

Preuve

Question

1. Sans utiliser de dictionnaire, écrire une fonction `somme(L, p)` déterminant en complexité linéaire s'il existe deux éléments de la liste triée `L` dont la somme est égale à `p`.
2. Montrer la terminaison de `somme` à l'aide d'un variant de boucle.
3. Montrer la correction de `somme` à l'aide d'un invariant de boucle.
4. Réécrire `somme` en utilisant un dictionnaire, pour une liste qui n'est pas forcément triée.

Preuve

En pratique, il est souvent fastidieux de prouver qu'une fonction est correcte... on peut aussi élaborer des tests pour vérifier le comportement sur des valeurs bien choisies.

Par exemple, tester `dichotomie(L, x)` sur différents cas possibles :

1. `L` est vide
2. `x` est le premier élément de `L`
3. `x` est le dernier élément de `L`
4. `x` est un élément quelconque de `L`
5. `x` n'est pas dans `L`

Représentation des nombres

Sur un ordinateur (dans la RAM ou sur le disque dur), on ne peut stocker que des 0 et des 1.

Définition

Un entier est stocké en utilisant un nombre p de bits fixé, par **complément à 2**, qui représente tous les entiers $n \in \{-2^{p-1}, ..., 2^{p-1} - 1\}$:

1. Si $n \geq 0$, on représente n par son écriture en base 2.
2. Si $n < 0$, on représente n par l'écriture en base 2 de $n + 2^p$

Représentation des nombres

Représentation par complément à 2 sur $p = 8$ bits :

0	1	1	1	1	1	1	1	$= 2^7 - 1 = 127$
0	1	1	1	1	1	1	0	$= 2^7 - 2 = 126$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	0	0	0	0	0	0	1	$= 1$
0	0	0	0	0	0	0	0	$= 0$
1	1	1	1	1	1	1	1	$= -1$
1	1	1	1	1	1	1	0	$= -2$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	0	0	0	0	0	0	1	$= -2^7 + 1 = -127$
1	0	0	0	0	0	0	0	$= -2^7 = -128$

On remarque que le 1er bit indique le signe.

Représentation des nombres

En Python, les entiers sont automatiquement redimensionnés si nécessaire donc il n'y a pas de limite sur la taille des entiers.

Représentation des nombres

Les flottants sont représentés en binaire selon le standard IEEE 754, qui utilise toujours 64 bits dont 52 pour les chiffres significatifs.

Représentation des nombres

Les flottants sont représentés en binaire selon le standard IEEE 754, qui utilise toujours 64 bits dont 52 pour les chiffres significatifs. Donc :

- Il y a une limite sur la taille des flottants.

```
>>> 2**1024.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

Représentation des nombres

Les flottants sont représentés en binaire selon le standard IEEE 754, qui utilise toujours 64 bits dont 52 pour les chiffres significatifs. Donc :

- Il y a une limite sur la taille des flottants.

```
>>> 2**1024.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

- Les calculs sur les flottants se font avec une précision limitée, avec des erreurs d'arrondis.

```
>>> 0.1 + 0.2
0.30000000000000004
```

Plutôt que tester l'égalité entre deux flottants (`if x == y`), il faut regarder si la différence est petite (`if abs(x - y) < 1e-10`).

Représentation des nombres

Question

Pourquoi le code suivant fait boucle infinie ?

```
s = 2**53
while s < 2**53 + 1:
    s += 0.1
```

Représentation des nombres

Question 3 Parmi les affirmations suivantes, indiquez celle ou celles qui sont vraies.

A) L'utilisation de nombres flottants peut provoquer des erreurs d'arrondis, mais celles-ci ne sont jamais graves car les erreurs d'arrondis sont minimes.

B) L'utilisation de nombres flottants peut provoquer de graves erreurs d'arrondis.

C) L'utilisation de nombres flottants ne provoque pas d'erreur d'arrondis.

D) Pour ne pas avoir d'erreur d'arrondis, il suffit de coder les flottants sur 64 bits plutôt que sur 32 bits.

Matrices

Question

Quel est le problème avec le code suivant ?
Comment le corriger ?

```
def creer_matrice(n, p):
    M = []
    L = []
    for i in range(p):
        L.append(0)
    for i in range(n):
        M.append(L)
    return M
```

Programmation dynamique

Question

On considère des listes L contenant une fois chaque entier de 1 à n, où $n = len(L)$.
Soit $s(L) = |\{i \mid L[i] \leq L[i+1]\}$ et $L_n(k)$ le nombre de listes L de taille n telles que $s(L) = k$.

1. Écrire une fonction calculant $s(L)$.
2. Donner une équation de récurrence pour $L_n(k)$.
3. En déduire un algorithme par programmation dynamique pour calculer $L_n(k)$.

Graphe

Définition

Un **graphe non orienté** est un couple $G = (V, E)$ où V est un ensemble de **sommets** et chaque **arête** $e \in E$ est un **ensemble** de deux sommets $e = \{u, v\}$. u et v sont les **extrémités** de e et on dit que u et v sont **voisins** (ou : adjacents).

Graphe

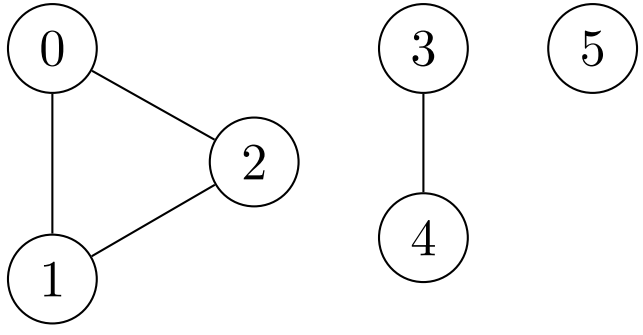
Définition

Un **graphe non orienté** est un couple $G = (V, E)$ où V est un ensemble de **sommets** et chaque **arête** $e \in E$ est un **ensemble** de deux sommets $e = \{u, v\}$. u et v sont les **extrémités** de e et on dit que u et v sont **voisins** (ou : adjacents).
Un graphe orienté est la même chose mais où les arêtes sont des couples (u, v), signifiant que l'on peut aller de u vers v seulement.

Graphe

Pour stocker un graphe en mémoire, on peut utiliser :

- Une **matrice d'adjacence** : `G[i][j]` vaut `1` si les sommets `i` et `j` sont adjacents, `0` sinon.
- Une **liste d'adjacence** : `G[i]` est la liste des voisins du sommet `i`.



Matrice d'adjacence

```
[[0, 1, 1, 0, 0, 0],
 [1, 0, 1, 0, 0, 0],
 [1, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0],
 [0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0]]
```

Liste d'adjacence

```
[[1, 2],
 [0, 2],
 [0, 1],
 [4],
 [3],
 []]
```

Graphe

Question

Écrire une fonction `degre(G, v)` calculant le degré d'un sommet `v` dans un graphe `G` :

1. Si `G` est une matrice d'adjacence.
2. Si `G` est une liste d'adjacence.

Parcours

Il y a deux façons de parcourir les sommets d'un graphe depuis un sommet de départ s :

- **En profondeur** : on visite le plus profondément suivant une branche et on revient en arrière quand on ne peut plus avancer.
- **En largeur** : on visite les sommets par distance croissante depuis s.

Parcours

Parcours en profondeur :

```
def dfs(G, s): # G est une liste d'adjacence
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

Question

Adapter le code ci-dessus pour savoir si un graphe est connexe.

Parcours

Parcours en largeur avec file q :

```
def bfs(G, s): # G est une liste d'adjacence
    visited = [False]*len(G)
    q = deque([s])
    while len(q) > 0:
        u = q.pop()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                q.appendleft(v)
```

Parcours

Application : recherche des distances (en nombre d'arêtes) depuis un sommet s.

```
def distances(G, s):
    dist = [-1]*len(G)
    q = deque([(s, 0)])
    while len(q) > 0:
        u, d = q.pop()
        if dist[u] == -1:
            dist[u] = d
            for v in G[u]:
                q.appendleft((v, d + 1))
    return dist
```

Plus courts chemins dans un graphe pondéré : Dijkstra

Définition

Un graphe **pondéré** est un graphe $\vec{G} = (V, \vec{E})$ muni d'une fonction de poids $w : \vec{E} \longrightarrow \mathbb{R}$.
$w(u, v)$ est le **poids** de l'arête de u vers v.

Plus courts chemins dans un graphe pondéré : Dijkstra

Définition

Un graphe **pondéré** est un graphe $\vec{G} = (V, \vec{E})$ muni d'une fonction de poids $w : \vec{E} \longrightarrow \mathbb{R}$.
$w(u, v)$ est le **poids** de l'arête de u vers v.

Il est pratique de définir $w(u, v) = \infty$ s'il n'y a pas d'arête entre u et v.
En Python, on peut utiliser `float("inf")` pour représenter $+\infty$.

Plus courts chemins dans un graphe pondéré : Dijkstra

Définition

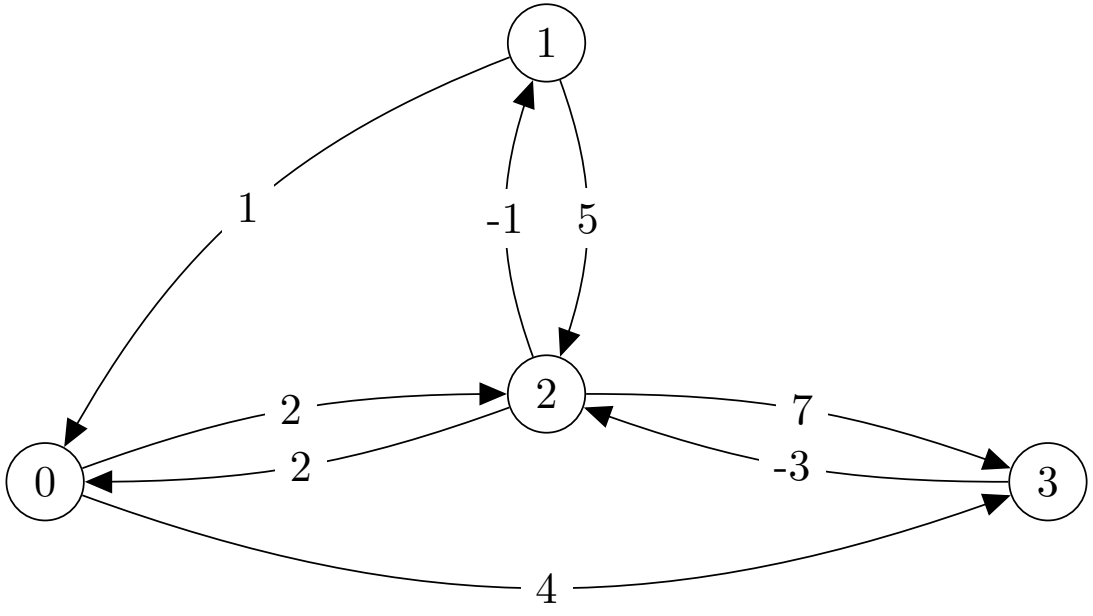
Un graphe **pondéré** est un graphe $\vec{G} = (V, \vec{E})$ muni d'une fonction de poids $w : \vec{E} \longrightarrow \mathbb{R}$.
$w(u, v)$ est le **poids** de l'arête de u vers v.

Il est pratique de définir $w(u, v) = \infty$ s'il n'y a pas d'arête entre u et v.
En Python, on peut utiliser `float("inf")` pour représenter $+\infty$.

Pour représenter un graphe pondéré, on utilisera une **matrice d'adjacence pondéré**, contenant $w(u, v)$ sur la ligne u, colonne v.

Plus courts chemins dans un graphe pondéré : Dijkstra

Exemple de graphe représenté par matrice d'adjacence pondérée :




```
G = [
    [0, float("inf"), 2, 4],
    [1, 0, 5, float("inf")],
    [2, -1, 0, 7],
    [float("inf"), float("inf"), -3, 0]
]
```

Plus courts chemins dans un graphe pondéré : Dijkstra

Algorithme de Dijkstra

Entrée : $\vec{G} = (V, \vec{E})$ un graphe pondéré orienté et $s \in V$.
Sortie : Une liste contenant $d(s, v)$, pour tout $v \in V$.

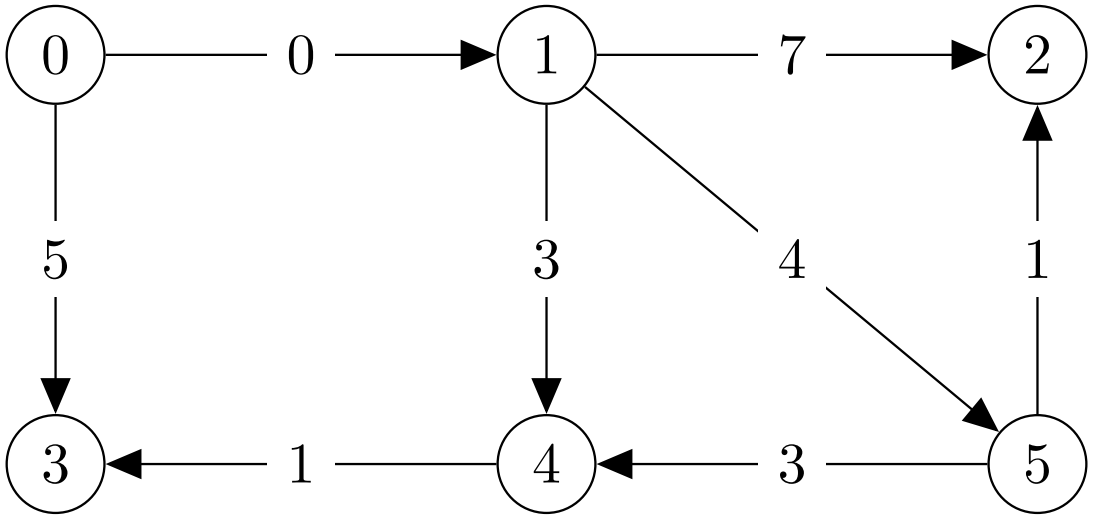
```
Initialement : q contient tous les sommets
               dist[s] = 0
               dist[v] = float("inf"), si v != s

Tant que q est non vide:
    Extraire u de q tel que dist[u] soit minimum
    Pour tout voisin v de u:
        Si dist[u] + w(u, v) < dist[v]:
            dist[v] = dist[u] + w(u, v)
```

Plus courts chemins dans un graphe pondéré : Dijkstra

Exercice

Appliquer l'algorithme de Dijkstra depuis $s = 0$ sur le graphe suivant, en mettant `dist[v]` à côté de chaque sommet `v` :



Plus courts chemins dans un graphe pondéré : Dijkstra

On suppose avoir une file de priorité telle que :

- `PriorityQueue()` renvoie une file de priorité vide.
- `q.add(x, p)` rajoute un élément `x` avec la priorité `p`.
- `q.take_min()` supprime et renvoie l'élément de priorité minimum.
- `q.update(x, p)` met à jouer la priorité de `x`.

Ici, chaque élément est un sommet dont la priorité est la distance estimée depuis le sommet de départ.

Plus courts chemins dans un graphe pondéré : Dijkstra

```python
def dijkstra(G, s):
    n = len(G)
    dist = n*[float("inf")]
    dist[s] = 0
    q = PriorityQueue()
    for v in range(n):
        q.add(v, dist[v])

    while not q.is_empty():
        u = q.take_min()
        for v in range(n):
            d = dist[u] + G[u][v]
            if v in q and d < dist[v]:
                q.update(v, d)
                dist[v] = d
    return dist
```

Algorithme A^*

La plupart du temps, on ne cherche pas les plus courts chemins à tous les autres sommets, mais seulement à un autre sommet fixé.

Exemples :

- GPS : Trouver un chemin le plus court d'une ville à une autre.
- Jeu vidéo : Déplacer un personnage d'un point à un autre.

Algorithme A^*

La plupart du temps, on ne cherche pas les plus courts chemins à tous les autres sommets, mais seulement à un autre sommet fixé.

Exemples :

- GPS : Trouver un chemin le plus court d'une ville à une autre.
- Jeu vidéo : Déplacer un personnage d'un point à un autre.

Dans ce cas, l'**algorithme A*** peut être plus efficace.

Algorithme A^*

But : Trouver un plus court chemin du sommet s au sommet t dans un graphe G.

Principe de l'algorithme A* :

1. Définir une fonction h telle que $h(v)$ soit une estimation de la distance de v à t.
2. Modifier l'algorithme de Dijkstra en utilisant `dist[u] + G[u][v] + h(v)` comme priorité, au lieu de `dist[u] + G[u][v]`.

Algorithme A^*

Modifications de A* par rapport à Dijkstra :

```
def astar(G, s, t, h):
    n = len(G)
    dist = n*[float("inf")]
    dist[s] = 0
    q = PriorityQueue()
    for v in range(n):
        q.add(v, dist[v])
    while not q.is_empty():
        u = q.take_min()
        if u == t:
            return dist[t]
        for v in range(n):
            d = dist[u] + G[u][v]
            if v in q and d < dist[v]:
                q.update(v, d + h(v))
                dist[v] = d
```

Algorithme A^*

Choix possibles de l'heuristique h pour A^* :

- Fonction nulle ($h = 0$) :

Algorithme A^*

Choix possibles de l'heuristique h pour A^* :

- Fonction nulle ($h = 0$) : On obtient Dijkstra.

Algorithme A^*

Choix possibles de l'heuristique h pour A^* :

- Fonction nulle ($h = 0$) : On obtient Dijkstra.
- Distance au sommet d'arrivée ($h(v) = d(v,t) - d(s,v)$) :

Algorithme A^*

Choix possibles de l'heuristique h pour A^* :

- Fonction nulle ($h = 0$) : On obtient Dijkstra.
- Distance au sommet d'arrivée ($h(v) = d(v,t) - d(s,v)$) : Choisit à chaque étape le sommet v minimisant $d(v,t)$.
  $\Longrightarrow$ Explore seulement les sommets le long d'un plus court chemin.

Algorithme A^*

Choix possibles de l'heuristique h pour A^* :

- Fonction nulle ($h = 0$) : On obtient Dijkstra.
- Distance au sommet d'arrivée ($h(v) = d(v,t) - d(s,v)$) :
  Choisit à chaque étape le sommet v minimisant $d(v,t)$.
  $\Longrightarrow$ Explore seulement les sommets le long d'un plus court chemin.

Malheureusement, on ne connaît pas $d(v,t)$. On utilise donc souvent une heuristique qui approxime la distance au sommet d'arrivée, mais plus facile à calculer.

Algorithme A^*

Choix possibles de l'heuristique h pour A^* :

- Fonction nulle ($h = 0$) : On obtient Dijkstra.
- Distance au sommet d'arrivée ($h(v) = d(v,t) - d(s,v)$) :
  Choisit à chaque étape le sommet v minimisant $d(v,t)$.
  $\Longrightarrow$ Explore seulement les sommets le long d'un plus court chemin.

Malheureusement, on ne connaît pas $d(v,t)$. On utilise donc souvent une heuristique qui approxime la distance au sommet d'arrivée, mais plus facile à calculer.
Par exemple, si les sommets sont des points dans $\mathbb{R}^2$ et $t = (t_1, t_2)$:

- Distance euclidienne ($h(x,y) = \sqrt{(x_1 - t_1)^2 + (x_2 - t_2)^2}$).
- Distance de Manhattan ($h(x,y) = |x_1 - t_1| + |x_2 - t_2|$).

Autres révisions

Revoir aussi :

- SQL
- Dictionnaire