TP – Algorithmique – S2

QUELENIS Enzo (B3 IW) & BONNAIRE Lucas (B3 SI)

Table des matières

Pour chacun des exercices, le code est disponible dans l'archive zip ou sur le repository Github https://github.com/CORT1N/esgi-algorithmic-works-2.

Exercice 1 : Recherche et parcours de graphes

1. Implémentation

Le code est disponible dans ex1/code.py.

Pour le graphe donné Figure 1 du sujet, retranscrit ici dans le config.json :

```
"ex1": {
    "graph": {
        "A": ["B", "C"],
        "B": ["D"],
        "C": ["E"],
        "D": ["F"],
        "E": [],
        "F": []
    }
},
```

Voici les résultats des algorithmes implémentés :

```
2025-06-19 11:08:27,363 - INFO - Parcours DFS récursif depuis A :
2025-06-19 11:08:27,363 - INFO - A
2025-06-19 11:08:27,363 - INFO - B
2025-06-19 11:08:27,363 - INFO - D
2025-06-19 11:08:27,364 - INFO - F
2025-06-19 11:08:27,364 - INFO - C
2025-06-19 11:08:27,365 - INFO - E
2025-06-19 11:08:27,365 - INFO - Parcours BFS depuis A :
2025-06-19 11:08:27,365 - INFO - BFS visit: A
2025-06-19 11:08:27,365 - INFO - BFS visit: B
2025-06-19 11:08:27,365 - INFO - BFS visit: C
2025-06-19 11:08:27,365 - INFO - BFS visit: D
2025-06-19 11:08:27,365 - INFO - BFS visit: E
2025-06-19 11:08:27,365 - INFO - BFS visit: F
```

2. Détection de cycles et recherche de composantes connexes

Voici les résultats obtenus pour le même graphe :

```
2025-06-19 11:08:27,365 - INFO - Détection de cycles (DFS) :
2025-06-19 11:08:27,365 - INFO - Aucun cycle détecté.
2025-06-19 11:08:27,365 - INFO - Composantes connexes (BFS) :
2025-06-19 11:08:27,365 - INFO -  Composante 1 : ['A', 'B', 'C', 'D', 'E', 'F']
```

On sait que l'algorithme de détection de cycles est fonctionnel car si l'on remplace le graphe actuel par un en possédant comme celui-ci :

```
"ex1": {
    "graph": {
    "A": ["B"],
    "B": ["C"],
    "C": ["A"]
    }
},
```

On obtient bien :

```
2025-06-19 11:12:38,597 - INFO - Détection de cycles (DFS) :
2025-06-19 11:12:38,597 - INFO - Cycle détecté dans le graphe.
```

3. Analyse de la complexité

a) Complexité temporelle

Concernant la complexité temporelle, DFS explore chaque nœud et chaque arête une fois.

Soit V le nombre de sommets (nœuds) du graphe.

Soit E le nombre d'arêtes du graphe.

La complexité temporelle du DFS est donc de $O(V+E)$.

Cela correspond au fait qu'on visite chaque sommet et on explore chaque arête adjacente une fois.

De la même manière, le BFS visite chaque sommet et chaque arête au plus une fois.

Sa complexité temporelle est donc elle aussi de $O(V+E)$.

b) Complexité spatiale

Pour le DFS, l'espace pour stocker le graphe est $O(V+E)$ et celui pour la pile d'appel récursive est au pire, la profondeur du graphe, c'est-à-dire $O(V)$ dans le cas d'un chemin linéaire de graphe en chaîne.

Donc sa complexité spatiale est $O(V+E)$.

Pour le BFS, l'espace pour stocker le graphe est le même et l'espace pour la file d'attente qui stocke les nœuds à visiter est au pire, toute une couche du graphe.

Ce qui veut dire que dans un graphe très large mais peu profond, avec peu de niveaux, la taille de la file peut atteindre $O(V)$.

Donc en conséquence, la complexité spatiale du BFS est elle aussi $O(V+E)$.

c) Dans quel cas préférer l'un ou l'autre ?

On peut choisir DFS lorsqu'on veut explorer profondément un chemin avant de revenir en arrière, dans les cas où la solution est susceptible d'être loin du point de départ, car il peut trouver la solution sans explorer tous les nœuds.

On peut au contraire choisir BFS lorsqu'on cherche le plus court chemin en nombre d'arêtes dans un graphe non pondéré car il explore couche par couche et garantit donc que la première fois qu'on atteint un nœud, c'est via le chemin le plus court possible.

Exercice 2 : Algorithme de Dijkstra

1. Implémentation

Fonction Dijkstra(graphe, noeud_depart)

    Initialiser dist : dictionnaire avec toutes les distances = +∞

    dist[noeud_depart] ← 0

    Initialiser parent : dictionnaire avec toutes les valeurs = None

    Initialiser une file de priorité (heap) avec (0, noeud_depart)

    Tant que la file de priorité n'est pas vide

        Extraire (distance_courante, noeud_courant) avec la plus petite distance

        Si distance_courante > dist[noeud_courant], alors

            Continuer (ignorer ce noeud car une meilleure distance a déjà été trouvée)

        Pour chaque (voisin, poids) dans les voisins de noeud_courant

            distance ← distance_courante + poids

            Si distance < dist[voisin], alors

                dist[voisin] ← distance

                parent[voisin] ← noeud_courant

                Ajouter (distance, voisin) dans la file de priorité

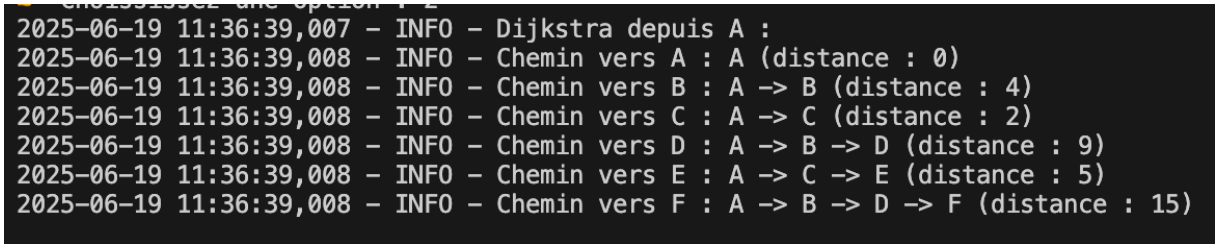
    Retourner dist, parent

Le code Python est disponible dans ex2/code.py

Pour le graphe pondéré donné Figure 2 du sujet, retranscrit ici dans le config.json :

```
"ex2": {
    "weighted_graph" : {
        "A": [["B", 4], ["C", 2]],
        "B": [["D", 5]],
        "C": [["E", 3]],
        "D": [["F", 6]],
        "E": [],
        "F": []
    }
},
```

Voici les résultats de l'algorithme implémenté :



2. Résolution de problèmes

Le problème de chemin le plus court est déjà géré dans la capture ci-dessus.

Nous ferons la comparaison des résultats avec l'algorithme de Bellman-Ford dans la partie d'après où nous en parlerons justement.

3. Analyse de la complexité

a) Complexité temporelle

Soit V le nombre de sommets (nœuds du graphe.

Soit E le nombre d'arêtes.

Cet algorithme utilise une file de priorité pour extraire le nœud avec la plus petite distance, donc chaque sommet est inséré au plus une fois dans la file, mais peut être inséré plusieurs fois si une distance meilleure est trouvée, par conséquent au pire $O(E)$ insertions.

L'extraction du minimum revient à $O(\log V)$.

Chaque mise à jour dans le tas revient aussi à $O(\log V)$.

Donc l'extraction au plus V fois : $O(V \log V)$.

Et l'insertion / la mise à jour au plus E fois : $O(E \log V)$.

En totalité, la complexité temporelle de l'algorithme de Dijkstra est donc $O((V + E) \log V)$.

b) Complexité spatiale

Le stockage du graphe est de $O(V + E)$, celui des dictionnaires dist et parent est de $O(V)$ et celui de la file au pire de $O(V)$ éléments donc la complexité spatiale totale de l'algorithme de Dijkstra est de $O(V + E)$.

Exercice 3 : Algorithme de Bellman-Ford

1. Implémentation

```
Fonction BellmanFord(graphe, noeud_depart)
    Initialiser dist : dictionnaire avec toutes les distances = +∞
    dist[noeud_depart] ← 0
    Initialiser parent : dictionnaire avec toutes les valeurs = None
    nodes ← liste des noeuds du graphe
    Pour i de 1 à (nombre_de_noeuds - 1) faire
        updated ← Faux
        Pour chaque noeud u dans nodes faire
            Pour chaque (voisin v, poids w) dans graphe[u] faire
                Si dist[u] + w < dist[v] alors
                    dist[v] ← dist[u] + w
                    parent[v] ← u
                    updated ← Vrai
        Si updated = Faux alors
            Sortir de la boucle
    Pour chaque noeud u dans nodes faire
        Pour chaque (voisin v, poids w) dans graphe[u] faire
            Si dist[u] + w < dist[v] alors
                Retourner None, None
    Retourner dist, parent
```

Le code Python est disponible dans ex3/code.py.

Pour le graphe pondéré négativement donné Figure 3, retranscrit ici dans le config.json :

```
"ex3": {
    "weighted_graph" : {
        "A": [["B", 6], ["C", 5]],
        "B": [["C", -2], ["D", -1]],
        "C": [["F", 3]],
        "D": [["E", 1]],
        "E": [["F", -4]],
        "F": []
    }
},
```

Voici les résultats de l'algorithme implémenté :

```
2025-06-19 11:55:44,177 - INFO - Bellman-Ford depuis A :
2025-06-19 11:55:44,178 - INFO - Chemin vers A : A (distance : 0)
2025-06-19 11:55:44,178 - INFO - Chemin vers B : A -> B (distance : 6)
2025-06-19 11:55:44,178 - INFO - Chemin vers C : A -> B -> C (distance : 4)
2025-06-19 11:55:44,178 - INFO - Chemin vers D : A -> B -> D (distance : 5)
2025-06-19 11:55:44,178 - INFO - Chemin vers E : A -> B -> D -> E (distance : 6)
2025-06-19 11:55:44,178 - INFO - Chemin vers F : A -> B -> D -> E -> F (distance : 2)
```

2. Détection de cycles de poids négatif

Dans le graphe pondéré négativement donné précédemment, il n'y avait pas de cycle négatif mais en créant cette situation avec celui-ci :

```
"ex3": {
    "weighted_graph": {
        "A": [["B", 1]],
        "B": [["C", -1]],
        "C": [["D", -1]],
        "D": [["B", -1]],
        "E": [["A", 2]],
        "F": []
    }
},
```

On obtient :

```
👉 Choississez une option : 3
2025-06-19 11:59:47,080 - INFO - Bellman-Ford depuis A :
2025-06-19 11:59:47,081 - ERROR - Cycle de poids négatif détecté dans le graphe !
```

Ce genre de détection est très utile et a des applications dans certains domaines critiques comme :

La finance ! En finance, les poids peuvent représenter des taux de change entre devises.

Donc un cycle de poids négatif pourrait correspondre à un arbitrage possible, une boucle de conversions qui rapporte du bénéfice sans risque, ce qui est littéralement de la triche.

Source : https://anilpai.medium.com/currency-arbitrage-using-bellman-ford-algorithm-8938dcea56ea

De manière plus globale, il est sûrement utilisé de beaucoup de manières pour détecter bugs et faille dans des systèmes tarifaires, ou d'horaires.

3. Analyse de la complexité

a) Complexité temporelle

Premièrement, on initialise les distances avec $O(V)$ avant de lancer la boucle principale exécutée $V - 1$ fois et pour chaque itération, on parcourt toutes les arêtes donc $O(E)$, ce qui nous donne $O(V \cdot E)$. Il reste la détection de cycle négatif qui elle aussi passe sur toutes les arêtes donc $O(E)$.

Ce qui nous donne en définitive une complexité temporelle pour l'algorithme de Bellman-Ford de $O(V \cdot E)$.

b) Complexité spatiale

L'algorithme utilise trois structure, le dictionnaire dist correspondant à une distance par nœud donc $O(V)$, le dictionnaire parent correspondant à un parent par nœud donc $O(V)$ aussi et enfin la structure du graphe, si en liste d'adjacence étant $O(V + E)$.

La complexité spatiale totale de l'algorithme de Bellman-Ford est donc $O(V + E)$.

Exercice 4 : Algorithmes de Ford-Fulkerson et Edmonds-Karp

1. Implémentation de Ford-Fulkerson

```
Fonction FordFulkerson(graphe_capacités, source, puits)
    graphe ← copie du graphe_capacités (pour représenter les capacités résiduelles)
    flux_max ← 0
    Définir Fonction DFS(parcours, visités, noeud)
        Si noeud = puits alors
            Retourner parcours (chemin trouvé)
        Ajouter noeud à visités
        Pour chaque voisin dans graphe[noeud]
            Si voisin non visité ET capacité[noeud][voisin] > 0 alors
                résultat ← DFS(parcours + (noeud, voisin), visités, voisin)
                Si résultat existe alors
                    Retourner résultat
        Retourner None (aucun chemin trouvé)
    Répéter indéfiniment :
        visités ← ensemble vide
        chemin ← DFS([], visités, source)
        Si aucun chemin trouvé alors
            Sortir de la boucle
        flot ← capacité minimale parmi les arêtes du chemin
        Pour chaque (u, v) dans le chemin faire
            Réduire capacité résiduelle de u → v par flot
            Augmenter capacité résiduelle de v → u par flot (rétroflux)
        Ajouter flot à flux_max
    Retourner flux_max
```

Le code Python est disponible dans ex4/code.py

Pour le réseau de capacités donné Figure 4, retranscrit ici dans le config.json :

```
"ex4": {
    "capacity_graph": {
        "A": {"B": 16, "C": 13},
        "B": {"D": 12, "E": 10},
        "C": {"F": 9},
        "D": {"E": 14},
        "E": {"F": 7},
        "F": {}
    }
},
```

Voici les résultats de l'algorithme implémenté :

```
2025-06-19 12:27:21,768 - INFO - Algorithme de Ford-Fulkerson (DFS) :
2025-06-19 12:27:21,769 - INFO - Chemin trouvé : [('A', 'B'), ('B', 'D'), ('D', 'E'), ('E', 'F')] avec flux = 7
2025-06-19 12:27:21,769 - INFO - Chemin trouvé : [('A', 'C'), ('C', 'F')] avec flux = 9
2025-06-19 12:27:21,769 - INFO - Flux maximal (Ford-Fulkerson) : 16
```

2. Implémentation de Edmonds-Karp

```
Fonction EdmondsKarp(graphe_capacités, source, puits)
    graphe ← copie du graphe_capacités (capacités résiduelles)
    flux_max ← 0
    Répéter indéfiniment :
        parent ← dictionnaire vide
        visités ← ensemble contenant la source
        file ← file contenant la source (BFS)
        Tant que la file n'est pas vide :
            u ← file.defiler()
            Pour chaque voisin v de graphe[u] :
                Si v non visité ET capacité u→v > 0 :
                    parent[v] ← u
                    visités ← visités ∪ {v}
                    file ← file ∪ {v}
                    Si v = puits :
                        Sortir de la boucle (chemin trouvé)
        Si le puits n'est pas dans parent :
            Sortir de la boucle principale (plus de chemin augmentant)
        chemin ← liste vide
        v ← puits
        Tant que v ≠ source :
            u ← parent[v]
            Ajouter (u, v) au chemin
            v ← u
        Inverser le chemin
        flot ← min(capacité[u][v] pour chaque (u, v) dans chemin)
        Pour chaque (u, v) dans chemin :
            capacité[u][v] ← capacité[u][v] - flot
            capacité[v][u] ← capacité[v][u] + flot
        flux_max ← flux_max + flot
        Afficher "Chemin trouvé : [chemin] avec flux = flot"
    Retourner flux_max
```

Pour le même réseau de capacités qu'avant, voici les résultats de l'algorithme implémenté :

```
2025-06-19 12:27:21,769 - INFO - Algorithme de Edmonds-Karp (BFS)
2025-06-19 12:27:21,769 - INFO - Chemin trouvé : [('A', 'C'), ('C', 'F')] avec flux = 9
2025-06-19 12:27:21,769 - INFO - Chemin trouvé : [('A', 'B'), ('B', 'E'), ('E', 'F')] avec flux = 7
2025-06-19 12:27:21,769 - INFO - Flux maximal (Edmonds-Karp) : 16
```

3. Analyse de la complexité

a) Complexité temporelle

Concernant Ford-Fulkerson en DFS, chaque appel peut explorer jusqu'à toutes les arêtes donc $O(E)$, et dans le pire des cas, on peut effectuer un nombre très elevé de cheminements, car le flot peut augmenter de 1 à chaque fois (ex : capacité = 1 partout) donc jusqu'à total flow = F, ce qui nous donne une complexité temporelle de $O(F . E)$.

Concernant Edmonds-Karp, il utilise BFS pour trouver les chemins augmentants les plus courts en nombre d'arêtes, et chaque arête peut être saturée au plus $O(V)$ fois, car chaque fois que le flot augmente, au moins une arête sur le plus court chemin est saturée ou recule.

Donc sachant que le nombre total d'augmentations représente $O(V . E)$ et que chaque BFS représente $O(E)$, on se retrouve à une complexité temporelle de $O(V . E^2)$.

b) Complexité spatiale

Les deux algorithmes utilisent des structures similaires comme le graphe résiduel, représentant une copie du graphe des capacités ayant une complexité de $O(V^2)$ si dense et $O(E)$ si sparse, la table parent servant à construire les chemins de complexité $O(V)$, et le visited ou la queue selon le DFS ou le BFS, tous deux d'une complexité $O(V)$.

Donc pour les deux, nous arrivons à une complexité de $O(V^2)$ ou $O(E)$ si liste d'adjacence.

c) Quand et lequel choisir ?

D'après les points précédents, on peut en déduire que Ford-Fulkerson en DFS sera plus performant et suffisant si le graphe a peu de sommets ou d'arêtes ou quand les capacités sont grandes, car le nombre de chemins augmentant sera donc faible et réduira les itérations.

Au contraire, Edmonds-Karp en BFS sera toujours polynomiale et donc sera préférable si on veut garantir la complexité, si le graphe est grand ou les capacités petites, et si on veut éviter les boucles très longues de DFS.

Exercice 5 : Tri rapide randomisé

1. Implémentation

Fonction QuickSortRandom(tableau)

Si la taille de tableau ≤ 1 :

Retourner tableau

pivot ← choisir un élément aléatoire de tableau

moins ← [éléments < pivot]

égaux ← [éléments == pivot]

plus ← [éléments > pivot]

Retourner QuickSortRandom(moins) + égaux + QuickSortRandom(plus)

Le code Python est disponible dans ex5/code.py.

2. Comparaison avec le tri rapide déterminist

Selon ces tableaux définis arbitrairement dans le config.json :

```
"ex5": {
    "test_arrays": [
        [10, 7, 8, 9, 1, 5],
        [3, 6, 2, 9, 4, 1],
        [5, 4, 3, 2, 1],
        [1, 2, 3, 4, 5],
        [],
        [42],
        [9, 8, 7, 6, 5, 4, 3, 2, 1]
    ]
},
```

Voici les résultats de cette comparaison d'algorithmes :

```
2025-06-19 14:32:37,945 - INFO - Test 1 avec le tableau : [10, 7, 8, 9, 1, 5]
2025-06-19 14:32:37,946 - INFO - Tri rapide déterministe : résultat = [1, 5, 7, 8, 9, 10], temps = 0.000551s
2025-06-19 14:32:37,946 - INFO - Tri rapide randomisé : résultat = [1, 5, 7, 8, 9, 10], temps = 0.000026s
2025-06-19 14:32:37,946 - INFO - Test 2 avec le tableau : [3, 6, 2, 9, 4, 1]
2025-06-19 14:32:37,946 - INFO - Tri rapide déterministe : résultat = [1, 2, 3, 4, 6, 9], temps = 0.000004s
2025-06-19 14:32:37,946 - INFO - Tri rapide randomisé : résultat = [1, 2, 3, 4, 6, 9], temps = 0.000014s
2025-06-19 14:32:37,946 - INFO - Test 3 avec le tableau : [5, 4, 3, 2, 1]
2025-06-19 14:32:37,946 - INFO - Tri rapide déterministe : résultat = [1, 2, 3, 4, 5], temps = 0.000004s
2025-06-19 14:32:37,946 - INFO - Tri rapide randomisé : résultat = [1, 2, 3, 4, 5], temps = 0.000010s
2025-06-19 14:32:37,946 - INFO - Test 4 avec le tableau : [1, 2, 3, 4, 5]
2025-06-19 14:32:37,946 - INFO - Tri rapide déterministe : résultat = [1, 2, 3, 4, 5], temps = 0.000004s
2025-06-19 14:32:37,946 - INFO - Tri rapide randomisé : résultat = [1, 2, 3, 4, 5], temps = 0.000010s
2025-06-19 14:32:37,946 - INFO - Test 5 avec le tableau : []
2025-06-19 14:32:37,946 - INFO - Tri rapide déterministe : résultat = [], temps = 0.000000s
2025-06-19 14:32:37,946 - INFO - Tri rapide randomisé : résultat = [], temps = 0.000000s
2025-06-19 14:32:37,946 - INFO - Test 6 avec le tableau : [42]
2025-06-19 14:32:37,947 - INFO - Tri rapide déterministe : résultat = [42], temps = 0.000001s
2025-06-19 14:32:37,947 - INFO - Tri rapide randomisé : résultat = [42], temps = 0.000000s
2025-06-19 14:32:37,947 - INFO - Test 7 avec le tableau : [9, 8, 7, 6, 5, 4, 3, 2, 1]
2025-06-19 14:32:37,947 - INFO - Tri rapide déterministe : résultat = [1, 2, 3, 4, 5, 6, 7, 8, 9], temps = 0.000009s
2025-06-19 14:32:37,947 - INFO - Tri rapide randomisé : résultat = [1, 2, 3, 4, 5, 6, 7, 8, 9], temps = 0.000017s
```

On peut voir ici, malgré la petite taille des jeux de données que le tri déterministe est légèrement plus rapide.

L'un des avantages principaux du tri rapide randomisé est la sélection du pivot, qui est faite de manière aléatoire, contre souvent le premier, le dernier ou le milieu dans le déterministe. Cela implique pour ce dernier que si le tableau est mal ordonné, le temps du pire cas est plus probable d'être $O(n^2)$.

Cependant, le premier commentaire fait sur les performances ne sera plus si vrai si nous augmentons la taille du tableau d'entrée, et le déterministe aura plus de chances d'être plus long que le randomisé.

Par conséquent, il paraît préférable de choisir le déterministe pour les petits jeux de données simples, ou si l'on souhaite un résultat reproductible, et de choisir le randomisé quand les données sont grandes ou plus complexes pour des raisons de performances.

3. Analyse de la complexité

a) Complexité moyenne

Dans un cas typique, en moyenne, le pivot divisera équitablement le tableau à chaque étape, même si c'est aléatoire. Le nombre de comparaison suivra donc une récurrence $T(n) = 2T(n/2) + O(n)$ et aura donc une complexité moyenne de $O(n \log n)$.

b) Complexité dans le pire des cas

Dans le pire des cas, se produisant si le pivot choisit toujours l'élément le plus grand ou le plus petit, le tableau sera divisé en n – 1 et 0, menant à une récurrence $T(n) = T(n - 1) + O(n)$ et donc une complexité de $O(n^2)$. Précisons cependant que ce cas est rare, le choix du pivot étant aléatoire.

c) Quand est-il préférable ?

Comme dit plus haut pour des questions de performances, le tri rapide randomisé est préférable quand le jeu de données est grand et qu'il n'est pas trié. Il évite beaucoup plus le pire des cas que le déterministe. Il est aussi très utile quand on a besoin de rapidité et pas de stabilité. Il faut en revanche ne pas nécessiter de résultat reproductible.

Exercice 6 : Arbres AVL

1. Implémentation

Fonction AVL_Insert(noeud, clé):

Si noeud est vide:

Retourner un nouveau noeud AVL contenant la clé

Si clé < noeud.clé:

noeud.gauche ← AVL_Insert(noeud.gauche, clé)

Sinon:

noeud.droite ← AVL_Insert(noeud.droite, clé)

Mettre à jour la hauteur de noeud

équilibre ← hauteur(noeud.gauche) - hauteur(noeud.droite)

Si équilibre > 1 et clé < noeud.gauche.clé:

Retourner Rotation_Droite(noeud)

Si équilibre < -1 et clé > noeud.droite.clé:

Retourner Rotation_Gauche(noeud)

Si équilibre > 1 et clé > noeud.gauche.clé:

noeud.gauche ← Rotation_Gauche(noeud.gauche)

Retourner Rotation_Droite(noeud)

Si équilibre < -1 et clé < noeud.droite.clé:

noeud.droite ← Rotation_Droite(noeud.droite)

Retourner Rotation_Gauche(noeud)

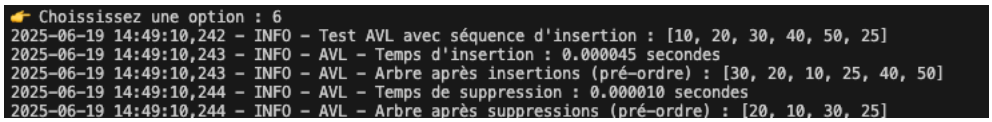
Retourner nœud

Le code Python est disponible dans ex6/code.py.

Selon ces séquences définies arbitrairement dans le config.json :

```
"ex6": {
    "insert_sequence": [10, 20, 30, 40, 50, 25],
    "delete_sequence": [40, 50]
},
```

Voici les résultat de l'algorithme implémenté :



Ils mettent aussi en avant les implémentations de la question suivante.

2. Tests de rééquilibrage

Pour tester le rééquilibrage de l'arbre, nous avons inséré la séquence montrée précédemment. Après chaque insertion, comme le montre la capture d'écran ci-dessus, l'arbre s'est automatiquement équilibré à l'aide des fonctions de rotations simple et doubles implémentées lorsque c'était nécessaire. De même lors de la suppression des éléments à supprimer. Cela garantit un facteur de balance entre +1 -1 à chaque nœud.

Concernant l'observation des rotations en question lors de l'insertion de certains éléments comme le 30 ou le 25, des rotations simples et doubles ont été effectuées pour maintenir l'équilibre.

Elles ont un impact direct sur la hauteur de l'arbre, en limitant sa croissance et donc en garantissant des opérations plus efficaces.

Par exemple, l'insertion de 30 après 10 et 20 provoque une rotation gauche, et l'insertion de 25 provoque une rotation double gauche-droite.

3. Analyse de la complexité

a) Complexité temporelle des opérations

Concernant l'insertion, le temps moyen et le pire cas font $O(\log n)$ car chaque insertion nécessite au plus un chemin de la racine jusqu'à la feuille, suivi d'éventuelles rotations.

Concernant la suppression, le temps moyen et le pire cas font $O(\log n)$ pour les mêmes raisons que l'insertion.

b) Complexité spatiale des opérations

Chaque nœud d'un arbre AVL contient une clé, deux pointeurs pour la gauche et la droite et un entier représentant la hauteur du nœud. De cette manière, l'espace mémoire est $O(n)$ pour n nœuds.

Les opérations d'insertion et de suppression utilisent la récursivité, mais la profondeur maximale d'appel est limitée à $O(\log n)$ grâce à l'équilibrage.

c) Comparaison des performances avec d'autres structures de données arborescentes

Nous avons décidé d'utiliser pour une comparaison un arbre BST.

Après implémentation, voici les résultats sur le même jeu de données :

```
2025-06-19 14:49:10,242 - INFO - Test AVL avec séquence d'insertion : [10, 20, 30, 40, 50, 25]
2025-06-19 14:49:10,243 - INFO - AVL - Temps d'insertion : 0.000045 secondes
2025-06-19 14:49:10,243 - INFO - AVL - Arbre après insertions (pré-ordre) : [30, 20, 10, 25, 40, 50]
2025-06-19 14:49:10,244 - INFO - AVL - Temps de suppression : 0.000010 secondes
2025-06-19 14:49:10,244 - INFO - AVL - Arbre après suppressions (pré-ordre) : [20, 10, 30, 25]
2025-06-19 14:49:10,244 - INFO - Test BST avec séquence d'insertion : [10, 20, 30, 40, 50, 25]
2025-06-19 14:49:10,244 - INFO - BST - Temps d'insertion : 0.000011 secondes
2025-06-19 14:49:10,244 - INFO - BST - Arbre après insertions (pré-ordre) : [10, 20, 30, 25, 40, 50]
2025-06-19 14:49:10,244 - INFO - BST - Temps de suppression : 0.000005 secondes
2025-06-19 14:49:10,244 - INFO - BST - Arbre après suppressions (pré-ordre) : [10, 20, 30, 25]
2025-06-19 14:49:10,244 - INFO - Comparaison des performances (en secondes) :
2025-06-19 14:49:10,244 - INFO - Insertion AVL : 0.000045 | BST : 0.000011
2025-06-19 14:49:10,244 - INFO - Suppression AVL : 0.000010 | BST : 0.000005
```

Les deux dernières lignes comparant les résultats temporels entre les deux nous permettent de voir que le BST est plus rapide dans ce cas précis, mais surtout car il ne réalise aucune rotation ni rééquilibrage.

De plus, le BST devient déséquilibré à droite sur un si petit jeu de données, ce qui laisse entendre que ses performances se dégraderaient beaucoup sur de grandes données.

Nous pouvons en conclure que le BST peut être utile sur de petites données car il est plus rapide mais peut vite se fragiliser sur de grandes données, ce qui permet de dire que l'AVL est préférable dès que le volume de données augmente, si l'on veut garder de la stabilité.

Exercice 7 : Problèmes NP-complets et NP-difficiles

1. Théorie

Un problème est dit NP (Non-deterministic Polynomial time) s'il peut être résolu en temps polynomial par une machine non-déterministe, ou plus clairement si une solution proposée peut rapidement être avérée par un algorithme.

Un problème NP-complet lui l'est s'il remplit plusieurs conditions. Premièrement, il est NP, puis il est au moins aussi difficile que tous les autres problèmes NP, c'est-à-dire que tout problème NP peut être réduit à lui en temps polynomial. L'un de ceux vu en cours cette année est le problème du sac à dos.

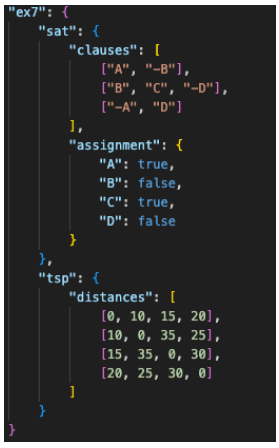
Enfin, un problème NP-difficile l'est si tous les problèmes NP peuvent être réduits à lui mais qu'il ne l'est pas lui-même. Autrement dit, il n'est pas garanti qu'on puisse avérer une solution rapidement, ou même qu'il en ait une décidable.

Ces notions jouent un rôle important en algorithmie, et en informatique car ils permettent premièrement de classer la difficulté d'un problème, deuxièmement d'orienter le choix des méthodes de résolution, troisièmement d'évaluer la faisabilité et les performances (pour ce point un exemple pour clarifier : savoir par exemple jusqu'à quelle taille de données un algorithme restera efficace), et enfin elle ont un lien avec la question la plus connue sur elles : être capable de prouver que P=NP, ce qui prouverait que tous les problèmes NP-complets sont résolubles, ce qui changerait radicalement des sous-domaines comme la cryptographie reposant sur des problèmes NP-difficiles comme la factorisation de grands nombres si l'on parle de RSA.

2. Implémentation

Le code Python est disponible dans ex7/code.py.

Selon ces jeu de données disponibles dans le sujet retranscrit dans le config.json :



Voici les résultats des algorithmes implémentés :

```
2025-06-19 16:35:38,817 - INFO - Vérification SAT :
2025-06-19 16:35:38,818 - INFO - Résultat : Non satisfiable
2025-06-19 16:35:38,818 - INFO - Temps SAT : 0.000121 s
2025-06-19 16:35:38,818 - INFO - Heuristique TSP (plus proche voisin) :
2025-06-19 16:35:38,818 - INFO - Chemin heuristique : [0, 1, 3, 2, 0]
2025-06-19 16:35:38,818 - INFO - Distance heuristique : 80
2025-06-19 16:35:38,819 - INFO - Temps heuristique : 0.000080 s
2025-06-19 16:35:38,819 - INFO - Chemin optimal (brute force) : [0, 1, 3, 2, 0]
2025-06-19 16:35:38,819 - INFO - Distance brute force : 80
2025-06-19 16:35:38,819 - INFO - Temps brute force : 0.000096 s
```

Comme vous pouvez le constater, afin de comparer nous avons aussi tenté de rajouter une méthode de bruteforce du chemin sur la partie TSP.

Concernant SAT, voici un exemple satisfiable :

```
"sat": {
    "clauses": [
        ["A", "-B"],
        ["B", "C"],
        ["-A", "D"]
    ],
    "assignment": {
        "A": true,
        "B": false,
        "C": true,
        "D": true
    }
```

Donnant :

```
👉 Choisissez une option : 7
2025-06-19 16:38:18,688 - INFO - Vérification SAT :
2025-06-19 16:38:18,688 - INFO - Résultat : Satisfiable
2025-06-19 16:38:18,689 - INFO - Temps SAT : 0.000015 s
```

3. Analyse et comparaison

a) Analyse des performances des vérificateurs et heuristiques

Dans le premier test, le vérificateur SAT a déterminé que la formule n'était pas satisfiable en 0.000018s, ce qui montre une excellente efficacité pour une formule de petite taille.

Pour le TSP, nous avons implémenté deux approches comme dit plus tôt, l'heuristique et l'exhaustif (le brute-force)

L'heuristique a aussi donné un résultat optimal mais beaucoup plus rapidement. Cependant, ce n'est pas garanti si on complexifie le cas.

Même pour un si petit cas, le bruteforce a montré un résultat 30 fois plus lent. Cela prouve que l'heuristique peut être très efficace pour obtenir une bonne solution rapidement même si elle n'est pas toujours la plus optimale.

b) Discussion concernant les approches exactes et heuristiques

Les approches exactes comme le brute-force pour le TSP ou l'énumération pour le SAT garantissent absolument la meilleure solution si elle existe. Cependant, leur complexité les rend lents, peu pratiques et peu performants pour les grands problèmes.

Les approches heuristiques elles, comme le TSP offrent rapidement une solution, mais pas toujours optimale. Dans des situations comme celles données par le sujet, elles sont plus utiles mais aussi dans des situations où le temps de réponse est crucial comme la navigation des véhicules autonomes, devant recalculer en temps réel le meilleur itinéraire avec le trafic et les obstacles devant eux.

Pour conclure, nous pensons qu'il est nécessaire de trouver un compromis en qualité de la solution et temps de calcul selon le contexte pratique de l'implémentation d'algorithmes de ce genre.