Rapport Algo TP

Icham Duret - Aurélien Aubriet

Exercice 1 :

2) Exercices sur la détection de cycles et la recherche de composantes connexes
a) Voici le code utilisé pour la recherche cyclé :

```python
def detect_cycle_dfs(graph):  1 usage  MaurrouViff
    visited = set()

    def dfs_cycle(node, parent):  MaurrouViff
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs_cycle(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs_cycle(node,  parent: None):
                return True
    return False

print("\nCycle détecté dans le graphe ?", detect_cycle_dfs(graph))
```

Voici ce que ça retourne :

```
Cycle détecté dans le graphe ? False
```

Ce qui veut dire que ça ne trouve aucun chemin cyclé.
b) Voici le code utiliser pour pour trouver toutes composantes connexes d'un graphe non orienté :

```python
def connected_components_bfs(graph):  1 usage  MaurrouViff
    visited = set()
    components = []

    for node in graph:
        if node not in visited:
            component = []
            queue = [node]
            while queue:
                current = queue.pop(0)
                if current not in visited:
                    visited.add(current)
                    component.append(current)
                    queue.extend(graph[current] - visited)
            components.append(component)
    return components

print("\nComposantes connexes :", connected_components_bfs(graph))
```

et ça retourne

```
Composantes connexes : [['E', 'C', 'A', 'B', 'D', 'F']]
```

3. Analyse de la complexité

(a) Complexité temporelle

DFS :

On visite chaque sommet et chaque arête une seule fois.

Complexité : O(V + E)

(V = nombre de sommets, E = nombre d'arêtes)

BFS :
Même complexité que DFS, car chaque nœud et chaque arête est traité une fois.
Complexité : O(V + E)

(b) Complexité spatiale

DFS :

Visited set : O(V)

Appels récursifs (pile d'appel) : O(V) au pire

Total : **O(V)**

BFS :

Visited set : O(V)

File (queue) : O(V) dans le pire cas

Total : **O(V)**

Exercice 2 :

Voici les plus courts chemins retournés par l'algorithme de Dijkstra.

{'A': 0, 'B': 4, 'C': 2, 'D': 9, 'E': 5, 'F': 15}

La complexité temporelle de Dijkstra est O((V + E) log V) (ou V est le nombre de sommets et E le nombre d'arêtes), la complexité spatiale est O(V) pour stocker les distances.

Dans le cas présent Dijkstra et Bellman Ford donnent les mêmes résultats (pas de poids négatif) mais Dijkstra est plus rapide

Exercice 3 :

1. a) Voici le pseudo-code de Bellman-Ford

```
Entrée : graphe G = (V, E), source s
 Initialiser la distance de chaque sommet à ∞ sauf la source à 0
 Pour i de 1 à |V| - 1 :
     Pour chaque arête (u, v) avec poids w :
        Si distance[u] + w < distance[v] :
           distance[v] = distance[u] + w
```

parent[v] = u
Pour chaque arête (u, v) avec poids w :
Si distance[u] + w < distance[v] :
=> Cycle de poids négatif détecté

d) Voici un exemple :

```
Distances depuis la source : {'C': 3, 'E': 0, 'D': 2, 'F': -4, 'B': 6, 'A': 0}
Parents : {'C': 'B', 'E': 'D', 'D': 'B', 'F': 'E', 'B': 'A', 'A': None}
Cycle de poids négatif : Non
```

2. a) Voici le code :

```
for edge in edges:
    u, v, w = edge["from"], edge["to"], edge["weight"]
    if distance[u] + w < distance[v]:
        has_negative_cycle = True
```

b)

Applications de la détection de cycles de poids négatif :

1. **Détection d'arbitrage en finance** :

   Dans les marchés de change, un cycle de poids négatif indique une opportunité d'arbitrage.

2. **Réseaux de transport** :

   Pour détecter des incohérences ou erreurs dans les données de temps/coûts.

3. **Planification de projets (PERT/CPM)** :

   Pour identifier des erreurs logiques (ex. tâche qui dépend d'elle-même indirectement).

4. **Algorithmes de jeux ou d'IA** :

   Pour détecter des stratégies auto-destructrices ou instables.

3. a) Voici la complexité temporelle :
Notation :

VVV = nombre de sommets

EEE = nombre d'arêtes

Étapes :

Initialisation : O(V)O(V)O(V)

Boucle principale : (V−1)×E=O(VE)(V - 1) \times E = O(VE)(V−1)×E=O(VE)

Vérification de cycle : O(E)O(E)O(E)

Complexité totale : O(V×E)

b) Voici la complexité spatiale :
Dictionnaires distance et parent → O(V)O(V)O(V)

Liste des arêtes (edges) → O(E)O(E)O(E)

Exercice 4 :

Complexité des algorithmes

	Temporelle	Spatiale
Ford-Fulkerson	O(E · max_flow)	O(V)
Edmonds-Karp	O(V · E²)	O(V)

L'algorithme d'Edmonds-Karp est plus long, mais il garantit (grâce à l'utilisation du BFS) d'avoir le réel flot maximum contrairement à Ford-Fulkerson

Exercice 5 :

1. a) Voici le pseudo code du tri randomisé :

```
Fonction RandomizedQuickSort(A, low, high)
    Si low < high :
        pivotIndex ← Random(low, high)
        Échanger A[pivotIndex] avec A[high]
        pi ← Partition(A, low, high)
        RandomizedQuickSort(A, low, pi - 1)
        RandomizedQuickSort(A, pi + 1, high)

Fonction Partition(A, low, high)
    pivot ← A[high]
    i ← low - 1
    Pour j de low à high - 1 :
        Si A[j] < pivot :
            i ← i + 1
            Échanger A[i] avec A[j]
    Échanger A[i + 1] avec A[high]
    Retourner i + 1
```

c) Voici ce que ça affiche avec le tri randomisé :

```
Avant tri : [10, 7, 8, 9, 1, 5]
Après tri : [1, 5, 7, 8, 9, 10]
```

2. a)

```
Données de départ : [10, 7, 8, 9, 1, 5]

Tri déterministe : [1, 5, 7, 8, 9, 10]
Temps déterministe : 8e-06 s

Tri randomisé    : [1, 5, 7, 8, 9, 10]
Temps randomisé  : 1.2e-05 s
```

b)

Tri rapide déterministe

Avantages :

Plus facile à déboguer et à tester car il donne toujours le même résultat sur les mêmes données.

Plus prévisible pour les performances sur certaines entrées spécifiques.

Inconvénients :

Très sensible à l'ordre initial des données. Si les données sont déjà triées ou presque, le pivot mal choisi peut conduire au pire cas de complexité, c'est-à-dire $O(n^2)$.

Moins robuste sur des ensembles de données structurées ou répétitives.

Tri rapide randomisé

Avantages :

Très robuste : même si les données sont déjà triées, le pivot aléatoire empêche la dégradation en $O(n^2)$ dans la grande majorité des cas.

Les performances moyennes sont presque toujours proches de $O(n \log n)$, quelles que soient les données.

Inconvénients :

Moins facile à déboguer car l'ordre des comparaisons change à chaque exécution.

Les résultats (chemin de tri, non le résultat final) varient, ce qui peut gêner certains tests ou analyses.

3. a)

Complexité moyenne

La complexité moyenne du tri rapide (qu'il soit déterministe ou randomisé) est **O(n log n)**. Cela s'explique par le fait que, dans un scénario idéal, chaque pivot divise le tableau en deux sous-tableaux de tailles à peu près égales. À chaque niveau de récursion, on fait **n** opérations pour réorganiser les éléments autour du pivot, et il y a environ **log n** niveaux de récursion. Le coût total est donc proportionnel à **n log n**.
Dans le cas du tri rapide randomisé, la **moyenne est encore plus stable**, car le pivot est tiré au hasard, ce qui réduit le risque de mauvais découpage systématique.

b)

Complexité dans le pire cas

Dans le pire des cas, le tri rapide a une complexité de **O(n²)**. Cela se produit lorsque le pivot choisi ne divise pas bien le tableau : par exemple, si le plus petit ou le plus grand élément est choisi à chaque fois. Cela conduit à une profondeur de récursion maximale (n niveaux) et à une complexité quadratique due aux échanges et comparaisons.

Ce cas est **courant** avec le tri rapide déterministe si les données sont déjà triées ou presque triées.
En revanche, dans le tri rapide randomisé, ce cas devient **très rare** car le pivot est choisi aléatoirement : la probabilité de toujours mal tomber devient négligeable à grande échelle.

c)
Le tri rapide randomisé est préférable dans plusieurs situations pratiques. Il est particulièrement utile lorsque :

Les données sont déjà triées ou presque : cela évite de tomber dans le pire cas du tri déterministe.

Les données ont une structure régulière ou un motif répétitif, où un pivot fixe tomberait toujours sur une mauvaise position.

On veut garantir des performances **moyennes stables et robustes**, indépendamment de l'ordre d'entrée.

Le code est utilisé dans un contexte concurrent ou avec des entrées très variées, où on ne peut pas anticiper leur structure.

Exercice 6

Les rotations de l'arbre AVL permettent de le garder équilibré en faisant littéralement faire un tour à l'arbres pour "changer les valeurs de place".

La complexité des opérations est en O(log n) ce qui le rends très rapide par rapport par exemple à une liste chaînée qui à une complexité de O(n) pour l'accès.

La complexité spatiale pour le stockage est de O(n)

Exercice 7

1. a) Problèmes NP (Non Déterministe Polynomial)
Un problème est dit **NP** si on peut **vérifier rapidement (en temps polynomial)** une solution **proposée**. Cela ne signifie pas qu'on peut forcément **trouver** cette solution rapidement, mais si quelqu'un nous la donne, on peut la valider efficacement.

NP-complet

Un problème est **NP-complet** s'il est à la fois :

> **Dans NP**, c'est-à-dire qu'on peut vérifier une solution rapidement.
>
> **Aussi difficile que tous les autres problèmes de NP**, ce qui signifie que **tous les autres problèmes NP peuvent se ramener à lui** (via une réduction polynomiale).

Si on trouve un algorithme polynomial pour un seul problème NP-complet, **tous les problèmes NP pourront être résolus rapidement**, ce qui est **la grande question P vs NP**.

Exemples de problèmes NP-complets :

> Problème SAT (satisfiabilité d'une formule booléenne).
>
> Problème du voyageur de commerce (TSP) en version décisionnelle.
>
> Problème du sac à dos.

NP-difficile

Un problème est **NP-difficile** s'il est **au moins aussi difficile que les problèmes NP**, **mais il n'est pas forcément dans NP**. Cela signifie qu'on ne peut **pas forcément vérifier rapidement une solution**, même si elle est correcte.

Certains problèmes d'optimisation ou de recherche peuvent être **plus complexes** que ceux qu'on peut juste vérifier rapidement.

Exemples de problèmes NP-difficiles :

TSP en version **optimisation** (trouver le plus court circuit, pas juste vérifier une solution).

Problèmes de jeux (comme échecs généralisés).

b)
Les concepts de **NP-complet** et **NP-difficile** sont fondamentaux en algorithmie pour plusieurs raisons :

1. **Comprendre les limites** : Ils aident à savoir **quand un problème est intrinsèquement difficile**, c'est-à-dire qu'il **n'existe probablement pas d'algorithme rapide** pour tous les cas généraux.
2. **Orienter la recherche d'algorithmes** : Pour les problèmes NP-complets, on ne cherche souvent pas une solution exacte en toutes circonstances, mais plutôt :

    Des **heuristiques** (rapides mais non garanties).

    Des **algorithmes d'approximation**.

    Des **algorithmes exacts mais lents**, à utiliser pour de petits cas.
3. **Réduction de problèmes** : En pratique, si un nouveau problème peut être réduit à un problème NP-complet connu, cela montre immédiatement qu'il est **au moins aussi difficile**, ce qui permet de choisir la bonne approche.
4. **Décision stratégique** : En algorithmie et en génie logiciel, savoir qu'un problème est NP-complet permet de **prévenir les attentes irréalistes** sur les performances.

2. c) Voici le TSP :

```
Itinéraire TSP (heuristique): ['A', 'B', 'D', 'C', 'A']
Distance totale : 80
```

Voici le SAT :

```
L'affectation satisfait la formule SAT ? True
```

3. a)

Vérificateur SAT

Le vérificateur SAT ne **résout pas** le problème, il vérifie simplement si une affectation donnée de variables **satisfait toutes les clauses**. Sa complexité est **linéaire en le nombre de clauses et de littéraux**, donc **très rapide** pour des formules de taille raisonnable.

Il est efficace tant que la solution candidate est fournie.
Mais pour **trouver** cette solution (ex : par recherche exhaustive), le coût explose car il faut tester **2^n affectations possibles** (où *n* est le nombre de variables).

Heuristique TSP (nearest neighbor)

L'heuristique du plus proche voisin (nearest neighbor) part d'une ville et choisit à chaque étape la ville non visitée la plus proche. Elle est rapide : **complexité en O(n²)**.

Elle donne **souvent une bonne solution**, mais **pas nécessairement la meilleure**. Dans certains cas (ex : agencements asymétriques), le résultat peut être loin de l'optimum.

Elle est donc utile pour obtenir rapidement une solution approximative dans des cas pratiques où le nombre de villes est grand.

b)

Approches exactes

Les approches exactes **garantissent** une solution optimale. Elles incluent :

- **Recherche exhaustive** (explorer toutes les possibilités),
- **Programmation dynamique** (par exemple, TSP par Held-Karp),
- **Retour arrière** ou **branch-and-bound**,
- **SAT solvers** modernes (ex : DPLL, CDCL).

Avantage : Solution garantie exacte.
Inconvénient : Temps d'exécution **exponentiel** dans le pire cas. Elles sont donc **inutilisables pour de grands problèmes** (ex. TSP avec 50+ villes).

Approches heuristiques

Les heuristiques **ne garantissent pas la meilleure solution**, mais cherchent une **bonne solution rapidement** :

- Pour le SAT : algorithmes gloutons, recherche locale, algorithmes génétiques.
- Pour le TSP : nearest neighbor, 2-opt, simulated annealing.

Avantage : Très rapides, **scalables**, souvent assez bons en pratique.
Inconvénient : Pas de garantie sur la qualité de la solution. Elles peuvent rater le meilleur chemin.