Algorithmes Avancés - Travaux Pratiques

Théo KASZAK , Paul-André Jacques - ESGI B3 SRC

Ce projet implémente plusieurs algorithmes avancés de graphes, de tri et de structures de données dans le cadre des travaux pratiques d'algorithmie avancée.

Structure du Projet

```
.
├── main.py                 # Point d'entrée principal
├── matrice.json           # Données des graphes de test
├── algo/
│   ├── dfs.py             # Algorithmes de parcours en profondeur
│   ├── bfs.py             # Algorithmes de parcours en largeur
│   ├── dijkstra.py        # Algorithme de Dijkstra
│   ├── bellman_ford.py    # Algorithme de Bellman-Ford
│   ├── ford_fulkerson.py  # Algorithme de Ford-Fulkerson
│   ├── edmonds_karp.py    # Algorithme d'Edmonds-Karp
│   ├── quicksort.py       # Tri rapide déterministe et randomisé
│   ├── avl.py             # Implémentation d'arbres AVL
│   ├── sat.py             # Vérificateur SAT et résolution
│   └── tsp.py             # Heuristiques pour le TSP
└── README.md
```

Exercices Implémentés

Exercice 1 : Recherche et Parcours de Graphes (DFS/BFS)

Algorithmes implémentés :

- **DFS (Depth-First Search)** : Parcours en profondeur
- **BFS (Breadth-First Search)** : Parcours en largeur
- **Détection de cycles** avec DFS
- **Composantes connexes** avec BFS

Analyse de la complexité :

Complexité temporelle :

- **DFS** : O(V + E) où V = nombre de sommets, E = nombre d'arêtes
- **BFS** : O(V + E) où V = nombre de sommets, E = nombre d'arêtes

Complexité spatiale :

- **DFS** : O(V) pour la pile d'appels récursifs et l'ensemble des sommets visités
- **BFS** : O(V) pour la file d'attente et l'ensemble des sommets visités

Cas d'usage préférentiels :

- **DFS** est préférable pour :
    - La détection de cycles
    - La recherche de composantes fortement connexes
    - Les problèmes nécessitant d'explorer en profondeur (parcours d'arbres)
- **BFS** est préférable pour :
    - Trouver le plus court chemin dans un graphe non pondéré
    - Parcourir par niveaux
    - Trouver les composantes connexes

Exercice 2 : Algorithme de Dijkstra

Fonctionnalités :

- Calcul des plus courts chemins depuis une source
- Reconstruction des chemins optimaux
- Gestion des graphes pondérés positifs

Analyse de la complexité :

Complexité temporelle :

- $O((V + E) \log V)$ avec une file de priorité (heap)
- $O(V^2)$ avec une implémentation naïve

Complexité spatiale :

- $O(V)$ pour stocker les distances, parents et la file de priorité

Exercice 3 : Algorithme de Bellman-Ford

Fonctionnalités :

- Calcul des plus courts chemins avec poids négatifs possibles
- Détection de cycles de poids négatif
- Reconstruction des chemins

Analyse de la complexité :

Complexité temporelle :

- $O(V \times E)$ où V = nombre de sommets, E = nombre d'arêtes
- V-1 itérations sur toutes les arêtes

Complexité spatiale :

- $O(V)$ pour stocker les distances et les parents

Exercice 4 : Algorithmes de Flot Maximum

Algorithmes implémentés :

- **Ford-Fulkerson** : Méthode générale pour le flot maximum
- **Edmonds-Karp** : Implémentation spécifique de Ford-Fulkerson avec BFS

Analyse de la complexité :

Ford-Fulkerson :

- **Complexité temporelle** : $O(E \times f)$ où f = valeur du flot maximum
- **Complexité spatiale** : $O(V)$ pour la recherche de chemin

Edmonds-Karp :

- **Complexité temporelle** : $O(V \times E^2)$
- **Complexité spatiale** : $O(V)$ pour la recherche BFS

Edmonds-Karp est préférable quand :

- On a besoin d'une complexité polynomiale garantie
- Le graphe a des capacités importantes (flot maximum élevé)
- On préfère la stabilité de performance à l'optimisation cas par cas

Exercice 5 : Tri Rapide Randomisé

Implémentations :

- **Tri rapide déterministe** : pivot = dernier élément
- **Tri rapide randomisé** : pivot choisi aléatoirement

Analyse de la complexité :

Complexité moyenne :

- $O(n \log n)$ pour les deux versions

Complexité dans le pire cas :

- **Déterministe** : $O(n^2)$ sur tableau déjà trié
- **Randomisé** : $O(n^2)$ mais très improbable

Le tri rapide randomisé est préférable quand :

- Les données peuvent être pré-triées ou partiellement triées
- On veut éviter les performances dégradées sur des patterns spécifiques
- On privilégie une performance moyenne stable

Exercice 6 : Arbres AVL

Opérations implémentées :

- **Insertion** avec rééquilibrage automatique

- **Suppression** avec rééquilibrage
- **Rotations** simples et doubles
- **Parcours infixe**

Analyse de la complexité :

Complexité temporelle :

- **Insertion** : O(log n)
- **Suppression** : O(log n)
- **Recherche** : O(log n)

Complexité spatiale :

- O(n) pour stocker l'arbre
- O(log n) pour la pile d'appels récursifs

Comparaison avec d'autres structures arborescentes :

- **vs BST non équilibré** : Garantit O(log n) au lieu de O(n) dans le pire cas
- **vs Arbre Rouge-Noir** : Plus strictement équilibré mais plus de rotations
- **vs B-arbres** : Mieux adapté pour la mémoire, B-arbres pour le stockage disque

Exercice 7 : Problèmes NP-complets et NP-difficiles

Implémentations :

- **Vérificateur SAT** : Vérification de satisfiabilité de formules booléennes
- **Résolution SAT** : Algorithme de backtracking pour trouver une assignation
- **Heuristique TSP** : Plus proche voisin pour le problème du voyageur de commerce
- **TSP optimal** : Solution par force brute pour comparaison

Concepts théoriques abordés :

NP-complet :

- Problèmes dans NP au moins aussi difficiles que tous les autres problèmes NP
- SAT est le premier problème prouvé NP-complet (théorème de Cook)
- Si P = NP, alors tous les problèmes NP-complets sont résolubles en temps polynomial

NP-difficile :

- Problèmes au moins aussi difficiles que les problèmes NP-complets
- Incluent des problèmes d'optimisation comme le TSP
- Peuvent ne pas être dans NP (problèmes de décision vs optimisation)

Analyse de la complexité :

Vérificateur SAT :

- **Complexité temporelle** : O(c × l) où c = nombre de clauses, l = taille moyenne des clauses
- **Complexité spatiale** : O(v) où v = nombre de variables

Résolution SAT (backtracking) :

- **Complexité temporelle** : O(2^n) dans le pire cas où n = nombre de variables
- **Complexité spatiale** : O(n) pour la pile de récursion

Heuristique TSP (plus proche voisin) :

- **Complexité temporelle** : $O(n^2)$ où n = nombre de villes
- **Complexité spatiale** : O(n)

TSP optimal (force brute) :

- **Complexité temporelle** : O(n!) où n = nombre de villes
- **Complexité spatiale** : O(n)

Applications pratiques :

- **SAT** : Vérification de circuits, planification automatique, intelligence artificielle
- **TSP** : Logistique, routage, optimisation de trajets

Résultats d'Exécution

Le programme teste tous les algorithmes sur des jeux de données prédéfinis et affiche :

- Les temps d'exécution en millisecondes
- Les résultats des algorithmes (chemins, distances, flots)
- Les démonstrations de rééquilibrage pour les arbres AVL

Exemples de sortie :

- **Graphe1** : 3 composantes connexes (A-B, C-D, E-F)
- **Graph_cycle** : Détection de cycle réussie
- **Dijkstra** : Plus courts chemins depuis A
- **Bellman-Ford** : Gestion des poids négatifs
- **Flot maximum** : 16 unités de A vers F
- **AVL** : Démonstration du rééquilibrage automatique
- **SAT** : Vérification de satisfiabilité sur formules booléennes
- **TSP** : Comparaison heuristique vs solution optimale

Utilisation

```
python3 main.py
```

Dépendances

- Python 3.x
- Modules standard : `json`, `time`, `random`, `heapq`, `collections`, `itertools`

Auteurs

JACQUES Paul-André, KASZAK Théo

Projet réalisé dans le cadre du cours d'Algorithmie Avancée - ESGI B3 2024-2025