Rapport de Projet : Matching sur Grille

Alexandre Houard & Octave Hedarchet
ENSAE Paris

April 2, 2025

Contents

1 Introduction

Le problème du matching dans les graphes est un problème fondamental en informatique et en théorie des graphes. Il consiste à trouver un ensemble de paires d'éléments qui satisfont certaines contraintes, généralement dans le but de minimiser ou maximiser un critère donné, comme le coût. Ce problème est transversal à divers domaines, notamment dans les réseaux de transport, dans l'appariement de données dans les systèmes de recommandation ou l'optimisation de processus industriels.

Dans ce contexte, l'objectif principal de ce rapport est de traiter de la résolution du problème de correspondance dans un cadre bidimensionnel, représenté sous forme de grille, où les cellules doivent être appariées selon certaines règles de coût et de validité. Nous explorons plusieurs algorithmes de résolution qui permettent de traiter ce problème de manière optimale, en minimisant le coût des appariements tout en respectant des contraintes spécifiques, telles que l'adjacence des cellules et l'interdiction de certaines paires en fonction de leurs caractéristiques.

Ce rapport présente une analyse approfondie des défis techniques, des choix algorithmiques et des structures de données utilisés dans l'implémentation de notre système de matching sur grille. Le projet comprend plusieurs algorithmes de résolution (Greedy, Ford-Fulkerson, Hongrois) appliqués à un problème d'appariement de cellules sur une grille colorée.

1.1 Contexte et Motivation

L'objectif de ce projet de programmation était de résoudre un problème de matching biparti sur une grille colorée, où des cellules doivent être appariées selon des règles de compatibilité de couleurs et de positions. Ce type de problème a des applications dans divers domaines comme l'allocation de ressources, la planification et l'optimisation combinatoire.

Le rapport détaille d'abord des choix algorithmiques utilisés (méthode de Ford Fulkerson, algorithme hongrois, etc.) pour résoudre ce problème. Ensuite, il présente une évaluation de la performance des algorithmes en utilisant l'argument de leurs complexité. Puis, les résultats de ce projet sont présentés avant de proposer une extension sous forme de jeu interactif.

1.2 Objectifs

Les objectifs principaux de ce projet étaient:

- Modéliser une grille colorée comme un problème de matching biparti
- Implémenter différents algorithmes de résolution (Greedy, Ford-Fulkerson, Hongrois)
- Comparer les performances et la qualité des solutions obtenues
- Fournir une interface visuelle pour observer les matchings générés
- Développer une extension interactive permettant de jouer à ce jeu

1.3 Aperçu du Rapport

Ce rapport ne se veut aucunement exhaustif quant au code du projet, son fonctionnement et son implémentation. L'objectif est avant tout de fournir un aperçu des éléments saillants au fil du projet, en mettant l'accent sur les défis techniques rencontrés et les solutions apportées.

1.4 Outils et Méthodologie

1.4.1 Git

Afin de collaborer au mieux ensemble, nous avons utilisé git tout au long du projet. Même si nous n'étions pas habitués à l'utiliser pour travailler en équipe mais surtout lors de projets individuels pour profiter du versionnage, ce dernier nous a été utile.

Afin de suivre les meilleures pratiques en terme de versionnage de code, nous travaillions avec 3 branches : la branche main, la branche Alexandre et la branche Octave. Les modifications n'étaient jamais faites sur le main mais toujours sur les branches respectives avant de merger les changements avec une pull-request. Nous aurions d'ailleurs apprécié qu'une courte introduction à git nous soit faite au vu de l'importance de cette technologie.

1.4.2 IDE et Linting

Comme indiqué, nous utilisions tous les deux Visual Studio Code. Afin de garantir un code propre, agréable à lire et à comprendre, nous avons utilisé successivement Pylint puis Ruff. Ce choix de changement de formater et de Linter a été avant tout motivé par la vitesse de Ruff qui nous donnait des conseils et avertissements au fur et à mesure de l'écriture du code là où pylint avait un peu plus de latence, ce qui pouvait être gênant dans l'itération successive du code.

2 Modélisation du problème

2.1 Grille et graphe bipartite

La modélisation du problème comme un problème de matching biparti a constitué le premier défi conceptuel. Il a fallu transformer une grille 2D en un graphe biparti où:

- Les cellules de parité paire (i+j est pair) forment un ensemble de sommets
- Les cellules de parité impaire forment l'autre ensemble
- Les arêtes représentent les paires valides selon des règles de compatibilité de couleurs

La matrice `MatriceCouleurOk` définit ces règles de compatibilité entre couleurs:

```
1 BlancCombinaisonOk = [1, 1, 1, 1, 0]
2 RougeCombinaisonOk = [1, 1, 1, 0, 0]
3 BleuCombinaisonOk = [1, 1, 1, 0, 0]
4 VertCombinaisonOk = [1, 0, 0, 1, 0]
5 NoirCombinaisonOk = [0, 0, 0, 0, 0]
```

Listing 1: Matrice de compatibilite des couleurs

Une paire entre deux cellules n'est valide que si l'entrée correspondante dans cette matrice est 1, ce qui représente une règle métier complexe à intégrer dans les algorithmes de matching.

3 Choix algorithmiques

Dans ce projet, plusieurs algorithmes fondamentaux ont été utilisés pour résoudre le problème. Ces algorithmes incluent principalement l'algorithme Hongrois, l'algorithme de Ford-Fulkerson et une approche gloutonne. Ces choix ont été motivés par la nature du problème de correspondance et les besoins d'optimisation associés. Dans cette section, nous allons détailler les caractéristiques de ces algorithmes, leur fonctionnement, et pourquoi ils ont été sélectionnés.

3.1 Le greedy solver

Le greedy solver n'a pas posé de difficultés particulières dans son implémentation. En revanche, nous avons été très surpris de sa complexité algorithmique, de l'ordre de $O(p^2)$ où p est le nombre de paires valides.

```
def run(self) -> list[list[tuple[int, int]]]:
    # Tri des paires: O(p log p)
    all_pairs_sorted = self.grid.all_pairs().copy()
    all_pairs_sorted.sort(key=self.grid.cost)

    # Pour chaque paire, on filtre les paires incompatibles: O(p^2)
    while len(all_pairs_sorted) > 0:
        filtered_list = []
        for pair in all_pairs_sorted:
            if pair[0] not in chosen_cells and pair[1] not in chosen_cells:
                filtered_list.append(pair)
        # ...
```

Listing 2: Implementation du solveur glouton

Elle s'explique aisément par la méthode employée et est inhérente à l'implémentation d'un *greedy algorithm*. À chaque étape, il nous a fallu recalculer les paires possibles, ce qui était très coûteux.

Nous avions envisagé le fait de potentiellement retirer des éléments de la liste mais cela aurait été équivalent à recréer une liste à chacune des étapes, chose pire du point de vue du nombre d'opérations que de simplement rajouter des éléments à la liste "paires possibles" comme nous l'avons fait.

Nous avons décidé de ne pas plus explorer une telle approche dans la mesure où l'algorithme greedy présente des limites qui lui sont propres : une succession d'optima locaux (paire la moins coûteuse à chaque étape) ne garantit pas un optimum global à la fin (le matching n'est pas nécessairement optimal).

3.2 Ford-Fulkerson

L'Algorithme de Ford-Fulkerson fut un premier pas dans l'implémentation d'algorithmes de matching. Il propose une méthode efficace afin de résoudre le problème mais sa portée est assez limitée dans la mesure où il ne s'applique concrètement que dans le cas où la valeur de toutes les cellules est égale à 1.

3.2.1 Fonctionnement de l'algorithme de Ford-Fulkerson

L'algorithme repose sur la modélisation du problème de correspondance comme un problème de flux dans un réseau. Le graphe est divisé en deux sous-ensembles : un ensemble de cellules de "source" (les cellules paires) et un ensemble de "puits" (les cellules restantes). L'algorithme cherche à maximiser le flux entre ces deux ensembles, ce qui correspond à maximiser le nombre de paires valides formées dans le problème de correspondance.

Voici comment il fonctionne :

- Création du graphe de résidu : Chaque paire de cellules possibles est considérée comme un arc entre deux nœuds du graphe. Le flux maximal est déterminé en augmentant progressivement les flux possibles entre les nœuds du graphe.
- Recherche de chemins augmentants : À chaque itération, on recherche un chemin augmentant du nœud source au nœud puits, c'est-à-dire un chemin où il est possible d'augmenter le flux.

- Augmentation du flux : Une fois un chemin trouvé, le flux sur ce chemin est augmenté, ce qui augmente le nombre total de correspondances.

```
def ford_fulkerson(self) -> int:
    max_flow = 0

    # Recherche de chemins augmentants: O(V*E) dans le pire cas
    path = self.find_augmenting_path()
    while path:
        # Mise a jour du graphe residuel: O(E)
        # ...
        max_flow += min_capacity
        path = self.find_augmenting_path()
```

Listing 3: Implementation de Ford-Fulkerson

Malheureusement, il n'est pas évident de le généraliser dans la mesure où le fait que la valeur de toutes les cellules soit égale à 1 garantissait le respect de la contrainte "une cellule ne peut faire partie que d'au plus une paire".

Dans le cas général, nous avons décidé de ne pas approfondir une généralisation de l'implémentation du SolverFulkerson pour se concentrer sur l'algorithme Hongrois.

3.3 Algorithme Hongrois

L'implémentation de l'algorithme hongrois a représenté le défi technique le plus significatif:

- Sa mise en œuvre nécessite une compréhension profonde de l'algèbre linéaire
- Les chemins augmentants et les potentiels duaux sont conceptuellement difficiles
- L'optimisation pour maintenir une complexité $O(n^3)$ demande une attention particulière

```
def _find_augmenting_path(self, cost: np.ndarray, u: np.ndarray, v: np.ndarray,
                          path: np.ndarray, row4col: np.ndarray, current_row: int)
                              -> tuple:
    min_value = 0
    num_remaining = cost.shape[1]
    remaining = np.arange(cost.shape[1])[::-1]

    SR = np.full(cost.shape[0], False, dtype=bool)
    SC = np.full(cost.shape[1], False, dtype=bool)

    shortest_path_costs = np.full(cost.shape[1], np.inf)
    sink = -1

    # Boucle de recherche du chemin augmentant
    while self._find_short_augpath_while_cond(...):
        # Complexe implementation detaillee...
```

Listing 4: Coeur de l'algorithme hongrois

Notre implémentation est un mélange de l'algorithme Hongrois classique et de l'approche Primal-Dual, permettant d'obtenir des solutions optimales pour tous les cas de test.

4 Analyse de la complexité algorithmique

4.1 Algorithme Greedy (SolverGreedy)

Complexité temporelle: $O(p^2)$ où p est le nombre de paires valides

L'algorithme glouton présente l'avantage d'être simple et rapide pour des grilles de petite taille, mais il peut produire des solutions sous-optimales, comme le montrent les résultats des benchmarks où il n'atteint des solutions optimales que dans 50% des cas sur les petites grilles et 11,1% sur les grilles moyennes.

4.2 Algorithme de Ford-Fulkerson (SolverFulkerson)

Complexité temporelle: $O(V \cdot E^2)$ où V est le nombre de sommets et E le nombre d'arêtes
L'algorithme construit un graphe résiduel où:

- Une source est connectée à toutes les cellules de parité paire
- Toutes les cellules de parité impaire sont connectées à un puits
- Les cellules de parité paire sont connectées aux cellules adjacentes de parité impaire

Cette représentation permet d'appliquer l'algorithme de flux maximum pour résoudre le problème de matching biparti. La fonction `find_augmenting_path` avec une complexité de O(V+E) est appelée potentiellement O(V·E) fois, ce qui donne une complexité globale de $O(V \cdot E^2)$.

4.3 Algorithme Hongrois (SolverHungarian)

Complexité temporelle: $O(n^3)$ où n est la dimension de la matrice de coût

```
def linear_sum_assignment(self, cost: np.ndarray, maximize: bool = False) ->
    tuple[np.ndarray, np.ndarray]:
    # Implementation de l'algorithme hongrois
    # ...
    # Recherche de chemins augmentants avec mise a jour des potentiels
    for current_row in range(cost.shape[0]):
        cost, u, v, path, row4col, col4row = self._lsa_body(
            cost, u, v, path, row4col, col4row, current_row
        )
```

Listing 5: Appel principal de l'algorithme hongrois

L'algorithme hongrois est le plus efficace pour trouver des solutions optimales, comme le montrent les benchmarks où il atteint 100% de solutions optimales sur toutes les tailles de grilles. Sa complexité cubique le rend toutefois plus coûteux pour les très grandes grilles.

5 Structures de données

5.1 Représentation de la grille

La classe `Grid` utilise plusieurs structures complémentaires:

```
def __init__(self, n: int, m: int, color: list[list[int]] = None, value: list[
    list[int]] = None) -> None:
    self.n = n
    self.m = m
    self.color = color if color else [[0 for j in range(m)] for i in range(n)]
    self.value = value if value else [[1 for j in range(m)] for i in range(n)]
    self.colors_list: list[str] = ["w", "r", "b", "g", "k"]
    self.cells_list: list[Cell] = []
    self.cells: list[list[Cell]] = []
```

Listing 6: Initialisation de la grille

Cette double représentation (tableaux 2D et objets `Cell`) permet:

- Un accès direct par coordonnées via `color[i][j]` et `value[i][j]`

- Une manipulation orientée-objet via `cells[i][j]`
- Un accès séquentiel via `cells_list`

Les paires sont représentées comme des tuples de tuples `((i1,j1), (i2,j2))`, ce qui facilite l'accès direct aux informations de cellules tout en maintenant leur relation.

5.2 Représentation du graphe pour le matching

La classe `SolverFulkerson` utilise un dictionnaire pour représenter le graphe d'adjacence:

```
def adjacency_graph_init(self) -> None:
    self.residual_graph["source"] = {}
    self.residual_graph["sink"] = {}

    # Construction du graphe...
    for i in range(self.grid.n):
        for j in range(self.grid.m):
            cell_id = f"cell_{i}_{j}"
            self.residual_graph[cell_id] = {}
```

Listing 7: Initialisation du graphe d'adjacence

Cette structure offre une flexibilité pour les mises à jour dynamiques du graphe résiduel pendant l'exécution de l'algorithme de Ford-Fulkerson.

La classe `SolverHungarian` utilise des tableaux numpy pour la matrice de coût:

```
# Construction de la matrice de cout
cost_matrix = np.zeros((max_dim, max_dim))
for u, v in pairs:
    # Calcul des poids...
    cost_matrix[even_to_idx[u], odd_to_idx[v]] = weight
```

Listing 8: Construction de la matrice de cout

Ces tableaux numpy permettent des opérations matricielles optimisées essentielles pour l'algorithme hongrois.

6 Résultats Expérimentaux

6.1 Protocole Expérimental

Nous avons testé nos trois algorithmes sur différentes tailles de grilles:

- Petites grilles: grid0x.in (environ 6 cellules)
- Grilles moyennes: grid1x.in (environ 20-30 cellules)
- Grandes grilles: grid2x.in (environ 100 cellules)

Pour chaque algorithme et chaque grille, nous avons mesuré:

- Le temps d'exécution
- Le score obtenu
- La déviation par rapport au score optimal
- Le nombre de paires formées

6.2 Présentation des Résultats

6.2.1 Résultats sur petites grilles

```
================================================================================
Small Grid Benchmark Results
================================================================================
solver,grid,grid_size,time,score,best_score,deviation,pairs,quality
SolverGreedy,grid00.in,2x3,0.000s,14,12,16.67%,3,Suboptimal
SolverFulkerson,grid00.in,2x3,0.000s,14,12,16.67%,3,Suboptimal
SolverHungarian,grid00.in,2x3,0.000s,12,12,0.00%,3,Optimal
```

Listing 9: Extrait des resultats sur petites grilles

Statistiques des solveurs:

- SolverFulkerson: 66.7% de solutions optimales
- SolverGreedy: 50.0% de solutions optimales
- SolverHungarian: 100.0% de solutions optimales

6.2.2 Résultats sur grilles moyennes

Statistiques des solveurs:

- SolverFulkerson: 66.7% de solutions optimales
- SolverGreedy: 11.1% de solutions optimales
- SolverHungarian: 100.0% de solutions optimales

6.2.3 Résultats sur grandes grilles

Pour les grandes grilles (grid2x.in), seul l'algorithme hongrois a été testé en raison de sa supériorité. Par exemple, sur grid21.in:

```
Completed SolverHungarian on grid21.in: Score=1686, Best=1686, Time=93.646s
```

Listing 10: Resultat sur une grande grille

6.3 Analyse des résultats

Ces résultats montrent que:

1. **L'algorithme hongrois** est systématiquement le plus précis, trouvant des solutions optimales dans 100% des cas testés.
2. **L'algorithme de Ford-Fulkerson** offre un bon compromis, avec 66,7% de solutions optimales.
3. **L'algorithme glouton** est généralement le plus rapide mais produit souvent des solutions sous-optimales, particulièrement sur les grilles moyennes où son taux de réussite chute à 11,1%.

Le choix de l'algorithme dépend donc du contexte d'utilisation:

- Pour des solutions optimales: l'algorithme hongrois
- Pour un bon compromis performance/qualité: l'algorithme de Ford-Fulkerson
- Pour des approximations rapides: l'algorithme glouton

7 Extension : jeu interactif

Nous avons rajouté au projet un jeu interactif où les joueurs sélectionnent des cellules adjacentes pour former des paires, dans le but de minimiser la masse totale de la solution de correspondance. L'interface graphique est réalisée avec la bibliothèque PyGame, et plusieurs algorithmes de résolution sont implémentés pour offrir différentes options de jeu, notamment des jeux solo, "contre un autre joueur" et "contre une IA".

Le jeu utilise un tableau à double entrée représentant la grille de cellules où chaque cellule a une couleur et une valeur associée. Les joueurs interagissent avec cette grille, en sélectionnant des cellules adjacentes pour former des paires valides et minimiser le coût global.

7.1 Modes de jeu

Le jeu propose plusieurs modes de jeu :

- **Mode solo** : Le joueur joue seul, essayant de minimiser le coût de ses paires.
- **Mode 2 joueurs** : Deux joueurs s'affrontent pour former des paires avec les cellules adjacentes.
- **Mode contre un ordinateur** : Le joueur affronte une intelligence artificielle, avec la possibilité de choisir entre différents algorithmes (Greedy, Fulkerson, ou Hongrois).

Cette extension ludique permet non seulement de démontrer l'application pratique des algorithmes développés, mais offre également un moyen interactif de comprendre le problème du matching sur grille.

A votre tour de jouer !

8 Bonnes pratiques de développement

8.1 Organisation du code

Le code est organisé selon le principe de séparation des préoccupations:

- La classe `Grid` gère la représentation des données
- La hiérarchie de classes `Solver` fournit différentes implémentations d'algorithmes
- Le module principal orchestre le flux de travail

```
class Solver:
    """Base class with common functionality"""

class SolverEmpty(Solver):
    """Empty implementation for testing"""

class SolverGreedy(Solver):
    """Greedy implementation"""

class SolverFulkerson(Solver):
    """Ford-Fulkerson implementation"""

class SolverHungarian(Solver):
    """Hungarian algorithm implementation"""
```

Listing 11: Hierarchie des solveurs

8.2 Documentation et gestion des erreurs

Le code inclut une documentation complète avec docstrings et annotations de complexité:

```
def is_pair_forbidden(self, pair: list[tuple[int, int]]) -> bool:
    """
    Returns True if the pair is forbidden and False otherwise.
    A bit more complex and relevant than simply checking if one of the cells is
        black.

    Parameters:
    -----------
    pair: list[tuple[int, int]]
        A pair of cells represented as a list of two tuples [(i1, j1), (i2, j2)]
        where (i1, j1) are the coordinates of the first cell and
        (i2, j2) are the coordinates of the second cell

    Returns:
    --------
    bool
        True if the pair is forbidden, False otherwise

    Raises:
    -------
    IndexError
        If either cell's coordinates are out of bounds

    Time Complexity: O(1)
        Constant time as it only involves direct index lookup and matrix access.
    """
```

Listing 12: Exemple de documentation

La gestion des erreurs est implémentée à travers des vérifications de limites et des validations d'entrée:

```
if i < 0 or i >= self.n or j < 0 or j >= self.m:
    raise IndexError("Cell coordinates out of bounds")
```

Listing 13: Exemple de gestion d'erreur

8.3 Tests et benchmarks

Le projet inclut une suite de tests complète:

```
def test_benchmark_small_grids(self):
    """
    Benchmark all solvers on small grid files and generate a performance report.
    """
    # Get small grids (grid0x.in)
    small_grids = [name for name in self.best_scores.keys()
                   if re.match(r'grid0[0-5]\.in', name)]
```

Listing 14: Exemple de test de benchmark

Ces tests vérifient non seulement la fonctionnalité mais aussi la performance, permettant une comparaison systématique des différents algorithmes.

9 Fonctionnalités visuelles

9.1 La fonction plot

Afin d'afficher la grille de manière similaire à ce qui avait été fait dans le descriptif du projet, nous avons opté pour la librairie matplotlib. Nous étions initialement partis pour un affichage dans le terminal (TUI / CLI).

Il nous a fallu utiliser plusieurs astuces afin d'afficher d'une part le cadrillage de la grille et de l'autre les chiffres en abcisse et en ordonnée placés au niveau des ticks qu'il nous a fallu décaler.

10 Difficultés rencontrées

10.1 Représentation duale des données

La représentation duale des données (tableau 2D et objets Cell) a nécessité une synchronisation constante:

```
def cell_init(self) -> None:
    for i in range(self.n):
        self.cells.append([])
        for j in range(self.m):
            self.cells[i].append(Cell(i, j, self.color[i][j], self.value[i][j]))

    self.cells_list = [
        Cell(i, j, self.color[i][j], self.value[i][j])
        for i in range(self.n)
        for j in range(self.m)
    ]
```

Listing 15: Initialisation des cellules

Cette dualité offre de la flexibilité mais introduit de la redondance dans le code et augmente le risque d'incohérences.

10.2 Implémentation de l'algorithme hongrois

La fonction `_find_augmenting_path` dans la classe `SolverHungarian` a été particulièrement délicate à implémenter correctement. L'algorithme hongrois requiert:

- Une compréhension fine des conditions de KKT (Karush–Kuhn–Tucker)
- Une gestion complexe des potentiels duaux
- Une optimisation constante pour maintenir la complexité théorique

11 Conclusion

Pour conclure, ce projet nous a permis d'approfondir notre compréhension des problèmes de correspondance dans les graphes, en particulier dans le contexte de la recherche de solutions optimales via des algorithmes bien établis tels que l'algorithme Hongrois et l'algorithme de Ford-Fulkerson.

L'algorithme Hongrois, avec son approche d'optimisation linéaire, nous a montré comment résoudre efficacement des problèmes de correspondance à coût minimal dans des graphes bipartites, en garantissant une solution optimale dans des temps raisonnables. D'autre part, l'algorithme de Ford-Fulkerson, basé sur le principe de flux maximal, nous a offert une perspective différente en modélisant le problème comme un réseau de transport, ce qui nous a permis d'explorer une approche fondée sur les graphes de flux pour trouver des correspondances maximales.

Ces algorithmes, bien que théoriquement solides, ont nécessité une mise en œuvre rigoureuse et un travail approfondi pour les adapter à notre problème spécifique. Nous avons dû gérer

de multiples contraintes, comme les couleurs et les valeurs des cellules, tout en optimisant le processus de sélection des paires. Le projet a ainsi renforcé notre capacité à appliquer des concepts théoriques à des problèmes pratiques, tout en nous permettant de mieux comprendre les nuances et les défis de l'optimisation combinatoire.

En outre, ce projet a été une véritable expérience de travail en équipe. Chaque membre a apporté ses compétences, tant en programmation qu'en analyse algorithmique, pour faire avancer le projet de manière cohérente et efficace. La collaboration étroite et la rigueur dans le développement et les tests des algorithmes ont été essentielles pour surmonter les défis techniques et garantir la réussite de l'implémentation. Ce travail collectif a enrichi notre approche du problème et nous a permis de tirer des leçons importantes sur la gestion de projets complexes et la collaboration dans le développement logiciel.

Remerciements

Nous tenions à remercier M. Benomar qui nous a accompagné au fil de la réalisation de ce projet. Sans lui, nous aurions eu beaucoup de mal à nous approprier les notions de théorie des graphes nécessaires à la résolution dudit problème.

Nous tenions aussi à remercier M. Galiana pour les explications claires quant à la mise en place d'un environnement python fonctionnel.