LINMA1691: Théorie des graphes
Devoir 4 : Minimum cost maximum flow

1 Contexte

A échelle macroscopique, le problème d'optimisation de la production d'énergie et de sa distribution depuis les lieux de production principaux jusqu'aux consommateurs est de taille. Il est important de toujours fournir à chaque consommateur la quantité d'énergie dont il a besoin pour une journée tout en minimisant le coût total lié à cet approvisionnement. Vous allez planifier au mieux la production et l'acheminement d'énergie dans un modèle simplifié.

Dans ce devoir, on va considérer deux types de producteurs d'énergie :

- **Les producteurs verts** : Ces producteurs correspondent à des champs éoliens ou de panneaux solaires. Ils sont considérés comme ayants un coût de production marginalement nul. Cependant, ils ont une capacité de production limitée et insuffisante pour couvrir la consommation totale. Celle-ci est déterminée à l'avance via l'utilisation des services météo ainsi que du machine learning.
- **Les centrales à gaz** : Celles-ci sont plus flexibles et vont nous permettre de compenser les manques du solaire et de l'éolien. Par contre elles n'ont pas un coût de production marginalement nul, car il faut acheter et brûler du gaz. Le coût marginal de production du GWh n'est pas linéaire pour une centrale dû à une baisse d'efficacité avec la quantité produite. On modélisera cette baisse d'efficacité par une fonction linéaire par morceaux et convexe. Un exemple d'une telle évolution de coût se trouve sur la Figure 1.

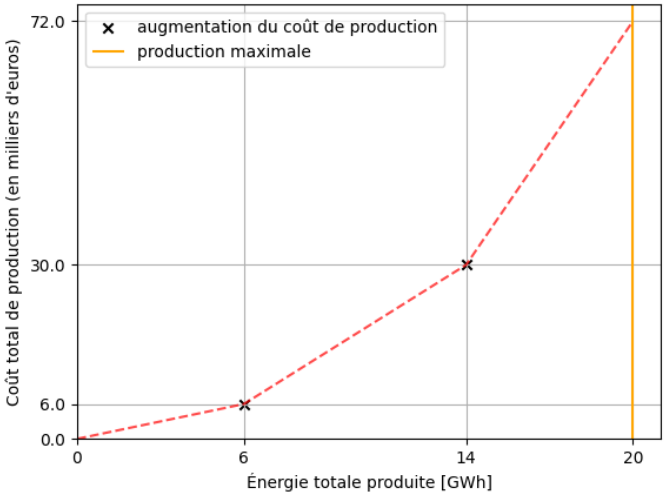

FIGURE 1 – Exemple d'évolution de coût de production pour une centrale à gaz

L'énergie produite circule via un réseau de lignes à haute tension reliant les producteurs et consommateurs entre eux. Il existe aussi des lignes entre producteurs et entre consommateurs. Chacune de ces lignes possède une capacité maximale de GWh qu'elle peut transférer sur une heure, ainsi qu'un coût associé et proportionnellement linéaire avec l'utilisation de cette ligne.

Ce coût est une manière simplifiée de prendre en compte les pertes sur l'énergie qui transite sur la ligne. On supposera ici que ces pertes peuvent être compensées via des acteurs locaux qui réinjecte l'énergie manquante. Ce phénomène ne sera pas modélisé autrement que par le coût associé à l'utilisation de la ligne, donc pas de perte dans le flux énergétique à modéliser.

Les lignes fonctionnent dans les deux sens.

2 Outils mathématiques

2.1 Problème

Vous modéliserez cette situation comme un problème de min cost max flow. Ce problème est une variante proche du problème de flot maximal vu au cours. Le problème est de déterminer un flot maximal sur le graphe sous les conditions que vous connaissez déjà (entrée de flot à la source, le flot sort au puits, respecte les capacités mises sur les arêtes, ...), mais en plus ce flot doit minimiser un coût lié à l'utilisation de chaque arête.

Formellement, à chaque arête, $e \in E$, du graphe on associe un coût $\text{cost}(e)$ et une capacité $\text{capa}(e)$. On défini le flot, f, qui associe à chaque arête son flot $f(e)$. Sous les contraintes classiques sur le flot, on contraint f à maximiser la quantité de flot et on minimise $\sum_{e \in E} f(e)\text{cost}(e)$ sous cette contrainte.

2.2 Algorithme

Un algorithme pour résoudre ce problème est très proche de l'algorithme de Ford-Fulkerson que vous connaissez déjà pour résoudre le problème de flot maximal. De la même manière, itérativement on va trouver un chemin augmentant et augmenter le flot sur celui-ci jusqu'à ne plus trouver de chemin augmentant. La seule différence est que l'on va choisir le chemin augmentant minimisant le coût par unité de flot. La recherche de ce chemin augmentant optimal revient à un problème de plus court chemin où les arêtes sont pondérées par leurs coûts. Les arêtes correspondant à un retour de flot ont le coût opposé (voir plus bas sur la construction du graphe résiduel).

Attention qu'ici un algorithme comme Dijkstra ne peut donc pas être employé. Car même si les coût initiaux sont positifs, des arêtes avec des poids négatifs vont apparaître dans le graphe résiduel. On utilisera ici l'algorithme de Bellman-Ford (pseudo-code plus bas) pour calculer le chemin augmentant optimal à chaque itération.

Une preuve d'optimalité de l'algorithme présenté peut être lue ici. L'idée en deux étapes est : un, que l'augmentation du flot tel que décrite ne crée pas de cycle à coût négatif dans le graphe résiduel ; deux, que l'inexistence d'un cycle de coût négatif dans le graphe résiduel implique que le flot minimise le coût. Combiner au fait que l'algorithme décrit trouve un flot maximal car il est une version de Ford-Fulkerson, on conclu que l'algorithme résout bien notre problème.

3 Énoncé

Ce devoir se déroule en trois parties. D'abord, vous allez devoir implémenter l'algorithme résolvant le problème de minimum cost maximum flow. Cette implémentation se fera dans la fonction `min_cost_max_flow` et les informations supplémentaires sur la manière de l'implémenter se trouvent dans la section 3.1.

Ensuite, vous allez devoir implémenter une fonction intermédiaire, `get_residual`. Celles-ci servira simplement à construire le graphe résiduel d'un graphe donné avec un flot initial nul. Le graphe résiduel, $G_r(V, E_r)$, d'un graphe dirigé, $G(V, E)$, avec capacités, capa ; coûts, cost ; et flot f, se définit comme suit.

Pour toute arête $e \in E$ allant du noeud u au noeud v, le graphe résiduel va contenir deux arêtes. La première, e', va du noeud u au noeud v et est telle que $\text{cost}(e') = \text{cost}(e)$ et $\text{capa}(e') = \text{capa}(e) - f(e)$. Et la deuxième, e'', va du noeud v au noeud u et est telle que $\text{cost}(e'') = -\text{cost}(e)$ et $\text{capa}(e'') = f(e)$.

Et enfin, vous allez devoir implémenter une fonction, `create_graph`, qui permet de modéliser le problème posé sous la forme d'un graphe dirigé avec un poids et une capacité sur chacune des arêtes ainsi qu'une source et un puits. Le minimum cost maximum flow sur le graphe représente la solution de notre problème de planification pour minimiser nos coûts tout en servant les consommateurs sur une heure.

3.1 Implémentation `min_cost_max_flow`

Implémentez l'algorithme décrit plus haut.

Pour vous aider dans votre tâche, vous pouvez vous inspirer du code fourni sur la page Wikipedia de l'algorithme de Ford-Fulkerson, **ici** [1].

Attention, la version que vous devez implémenter utilisera une liste d'adjacence et non une matrice d'adjacence comme dans le lien Wikipedia fourni. Une implémentation utilisant une matrice et non une liste d'adjacence, en plus d'être largement moins efficace, ne permettrait pas l'utilisation d'arêtes multiples qui pourraient être utiles dans le problème posé.

Vous devez aussi adapter le code pour chercher un chemin augmentant minimisant le coût via Bellman-Ford. Pour l'implémentation de Bellman-Ford, vous pouvez vous inspirer du pseudocode suivant :

```
distance := [infinity] * n
distance[s] := 0
Q := queue()
Q.add(s)
while Q is not empty :
    u := Q.pop()
    for each edge u->v incident to node u :
        if edge.cost + distance[u] < distance[v]:
            distance[v] := edge.cost + distance[u]
            parent[v] := u
            if v not in Q:
                Q.add(v)
```

La version présentée utilise l'opération `not in` sur une `Queue`. Cette opération a une complexité linéaire en le nombre d'éléments dans la structure de donnée. Corrigez ce défaut dans votre implémentation.

Deuxièmement, le pseudo-code ne retient pas quelle arêtes font parties du plus court chemin. Cette information est nécessaire pour votre algorithme résolvant le problème de min cost max flow. Corrigez ce deuxième problème.

3.1.1 Input de la fonction

Cette fonction prendra 3 arguments en entrée :

- `s`, l'indice du noeud représentant la source dans le graphe construit.
- `t`, l'indice du noeud représentant le puit dans le graphe construit.
- `graph_residual`, une liste de `Edge` représentant la liste d'adjacence d'un graphe résiduel. La classe `Edge` vous est fournie dans le canevas.

1. L'implémentation disponible sur la page Wikipedia est celle d'Edmonds-Karp, une variante de Ford-Fulkerson utilisant BFS pour trouver les chemins augmentants.

3.1.2 Output de la fonction

Cette fonction retournera deux informations :

```
minimum_cost, maximum_flow = min_cost_max_flow(s, t, graph_residual)
```

Celles-ci représentent respectivement le flot maximum et le coût minimal d'un flot maximum entre les noeuds `s` et `t` du graphe résiduel donné en argument.

3.2 `get_residual`

Se référer à la description donnée plus haut.

3.2.1 Input de la fonction

— `graph`, une liste de `Edge` représentant la liste d'adjacence d'un graphe.

3.2.2 Output de la fonction

— `graph_residual`, une liste de `Edge` représentant la liste d'adjacence du graphe résiduel du graphe donné en input à la fonction pour un flot nul.

3.3 `create_graph`

3.3.1 Input de la fonction

— `capacities`, une matrice symétrique représentant les capacités des lignes à haute tension du réseau en GWh [2]. `capacities[i][j]` est un entier qui vaut la capacité de la ligne à haute tension allant du lieu `i` au lieu `j`.
— `costs`, une matrice représentant les coûts liés aux pertes sur les lignes à haute tension [3]. `costs[i][j]` vaut le coût lié aux compensations des pertes sur la ligne à haute tension allant du lieu `i` au lieu `j` en euro par Gwh.
— `green_sources`, un dictionnaire dont les clés sont les indices des lieux représentants un producteur vert sur le réseau et dont les valeurs sont leurs capacités de production en GWh.
— `gas_centrals`, un dictionnaire dont les clés sont les indices des lieux représentants une centrale à gaz et dont les valeurs sont une liste de tuples représentants les coordonnées des points où on observe une modification de la pente de la courbe représentant l'évolution du coût marginal de production de la centrale à gaz en question. Par exemple, si la centrale de l'exemple donné ci-dessus avait l'indice `i`, on aurait :

```
gas_centrals[i] = [(0,0), (6,6), (14,30), (20,72)]
```

— `consumers`, un dictionnaire dont les clés sont les indices des lieux représentants un consommateur sur le réseau et dont les valeurs sont les demandes correspondantes en GWh.

3.3.2 Output de la fonction

— `s`, l'indice du noeud représentant la source dans le graphe construit.
— `t`, l'indice du noeud représentant le puit dans le graphe construit.
— `graph`, une liste de `Edge` représentant la liste d'adjacence du graphe construit.

2. `capacities` est une matrice symétrique puisque les lignes à haute tension fonctionnent dans les deux sens
3. elle est symétrique également

3.4 Jeux de tests

Exécutez `test.py` pour lancer des tests sur votre code. Votre fonction résolvant le problème de min cost max flow est d'abord testée indépendemment. Ensuite, votre modélisation et la fonction créant le graphe résiduel sont testées en les imbriquant avant votre résolution du min cost max flow. Seuls les résultats finaux, c'est-à-dire la quantité de flot avec le coût atteint, sont testés.

La complexité asymptotique dans le pire cas de l'algorithme décrit est $O(V^2E^2)$ ($O(VE)$ itérations de trouver puis augmenter un chemin augmentant, et $O(VE)$ pour une application de Bellman-Ford). Le temps d'exécution en pratique peut tout de même être bien meilleur. Nous testerons votre algorithme avec $1e3$ noeuds et arêtes en ordre de grandeur sur un graphe fourni avec `test.py`.

4 Soumission

Soumettez votre code en un fichier `.py` sur Inginious avec votre binôme habituel. Les tests se font sur l'entièreté de votre code (la modélisation et la résolution algorithmique). Le résultat est binaire et il faut donc que vos deux parties soient correctes.

- **Échéance : S14 vendredi 22 décembre à 22h**
- **Questions :** Contactez `hugo.vandermosten@student.uclouvain.be` ***et*** `brieuc.pinon@uclouvain.be` (en copie).