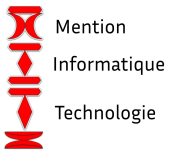

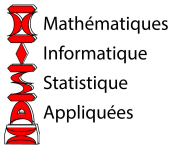

Université d'ANTANANARIVO
Domaine Sciences et Technologies
Mention Informatique et Technologie

Mémoire en vue de l'obtention du diplôme de Master 2 en
Mathématiques Informatique et Statistique Appliquées

Élaboration et implémentation d'un réseau de neurones résolvant ou approximant la solution d'un problème du type flot maximum coupe minimum dans un réseau en imitant les algorithmes existants

Présenté le 30 Mai 2023 par :
Taratra Dimbimandresy RAHARISON

Devant le jury composé de :

Président du Jury :	$M.$ *Arthur* RANDRIANARIVONY	Université d'Antananarivo
Examinateur :	$M.$ *Olivier* ROBINSON	Université d'Antananarivo
Encadreur pédagogique :	$M.$ *Tsinjo Tony* RAKOTOARIMALALA	Université d'Antananarivo

Remerciements

Je remercie l'Éternel qui par sa présence m'a accompagné durant mon parcours.

Je tiens à exprimer ma gratitude à tous ceux qui ont contribué à l'élaboration de ce mémoire.

J'exprime toute ma reconnaissance à Monsieur Tsinjo Tony RAKOTOARIMALALA, de m'avoir encadré, soutenu, et conseillé durant ce stage de recherche.

Je tiens à remercier le corps enseignant de la MISA, ainsi que celui du département de Mathématiques et Informatique, pour les connaissances et les conseils qu'ils ont prodigués lors de mon cursus universitaire.

Je suis reconnaissant aux membres du jury qui ont accepté d'évaluer mon travail.

Enfin, je ne saurais manquer d'exprimer ma gratitude envers ma famille pour leur soutien indéfectible, et à mes camarades à la MISA dont l'aide a contribué à l'aboutissement de ce mémoire.

Table des matières

Table des figures

Liste des tableaux

Acronymes

CLRS Cormen Leiserson Rivest Stein
CNN Convolutional Neural Network
ER Erdös-Rényi
GAN Generative Adversarial Network
GAT Graph Attention Network
GNN Graph Neural Network
MNN Memory Neural Network
MPNN Message-Passing Neural Network
ReLU rectified linear unit
RNN Recurrent Neural Network
ViT Vision Transformer

Introduction

Cette partie expose de manière générale ce qu'est le raisonnement algorithmique neuronal ; introduit au CLRS-30, un cadre de travail développé dans le but d'aider à faire avancer ce domaine ; explique les raisons du choix du problème de flot maximum coupe minimum ; et détaille les contributions apportées au CLRS-30.

Contexte et motivation

Raisonnement algorithmique neuronal et imitation algorithmique

Le raisonnement algorithmique neuronal est un domaine à l'intersection de l'algorithmique et de l'apprentissage profond, dont le but est de transférer les capacités de généralisations des algorithmes classiques aux algorithmes d'apprentissage, en particulier, aux réseaux de neurones [48]. Pour ce faire, on entraîne les réseaux de neurones à imiter le comportement des algorithmes classiques. Nous faisons en sorte que les réseaux de neurones puissent reproduire les opérations effectuées par les algorithmes classiques.

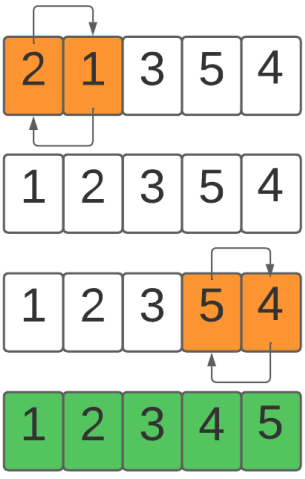

FIGURE 1 – Échanges de l'algorithme de tri à bulles

Prenons le cas de l'algorithme de tri à bulles, nous pouvons foire sur la figure 1 les échanges effectués par l'algorithme. Imiter l'algorithme de tri à bulles à l'aide d'un réseau de neurones, c'est entraîner ce réseau de neurones à reproduire les échanges effectués par l'algorithme de tri à bulles sur les éléments adjacents du tableau durant son exécution. Il ne s'agit donc pas de donner en entrées le tableau non-trié avec le résultat attendu, c'est-à-dire, le tableau trié, mais bien de permettre au réseau de neurones de générer les différentes étapes de l'algorithme. Cette approche d'imitation algorithmique est adoptée dans l'espoir que les méthodes de l'apprentissage profond puisse permettre la découverte de nouveaux algorithmes, en plus de permettre aux réseaux de neurones de mieux généraliser face à des données en dehors de leurs données d'entraînements [53]. En effet, l'utilisation de méthodes de l'apprentissage automatique pour découvrir de nouveaux algorithmes a déjà donné des fruits. En Octobre 2022, Fawzi et al. ont découvert, à l'aide de l'apprentissage profond par renforcement, un algorithme de multiplication de matrice efficace, plus performant que l'algorithme à deux niveaux de Strassen **pour les matrices 4 x 4 dans un corps fini** [16].

CLRS-30

Pour permettre aux chercheurs de développer le domaine du raisonnement algorithmique neuronal, les chercheurs de Deepmind, Veličković et al., ont développé le CLRS-30 [47]. Il s'agit d'un cadre de travail, venant avec les outils nécessaires pour imiter des algorithmes classiques, notamment, cinq architectures de réseaux de neurones différentes :

— Deep Sets [56]
— Graph Attention Networks [51]
— Graph Attention Networksv2 [6]
— Message-passing Networks [20]
— Pointer Graph Networks [50]
— Memory Networks [43]

Ces architectures appartiennent à ce qu'on appelle les réseaux de neurones sur les graphes ou GNN. Cette catégorie de réseaux de neurones a été choisie pour ses performances sur les tâches de raisonnement (test QI, problèmes élémentaires en physique) [55].

Dans leur étude, Veličković et al. ont imité 30 algorithmes classiques tirés du livre CLRS [9], laissant de côté le problème du flot maximum en guise d'amélioration future. Le problème du flot maximum est un problème d'optimisation combinatoire dont les applications sont nombreuses, on peut citer par exemple la gestion du réseau routier, la gestion des réseaux d'irrigation et de distribution de l'eau, l'allocation de ressource. Formulé par Harris E. T. et Ross F.S. en 1955 [27], ce problème a été longtemps étudié, et de nombreux algorithmes ont été développés pour le résoudre, chacun avec leurs forces et leurs faiblesses en terme de compléxité. Plus récemment (2017,2019, et 2020), il y a eu un intérêt croissant dans le développement de techniques basées sur l'apprentissage automatique, notamment, les GNN, dans la résolution de problèmes d'optimisation combinatoire [19], [33], [34], incluant le problème du flot maximum. Cependant, les approches imitant les algorithmes de flot maximum existants sont quasi-inéxistantes. C'est ainsi que nous avons décidé d'imiter un algorithme de flot maximum à l'aide des outils du CLRS-30.

Le problème du flot maximum dans un réseau de transport, bien qu'enfreignant les règles fixés par Veličković et al. [47], peut tout de même être imité en utilisant la correspondance avec le problème de la coupe minimum, grâce au théorème de Ford-Fulkerson [17]. Ces règles s'expliquent du fait que la représentation des nombres par les architectures rendent ambigue l'évaluation des performances des modèles [47]. Ces architectures s'en sortent mieux sur des décisions discrètes, par exemple le choix ou non d'un noeud dans un graphe.

Question de recherche

Le problème du flot maximum a été mis de côté par Veličković et al.[47], mais dans leurs travaux, les auteurs ont émis la possibilité d'imiter les algorithmes résolvant ce problème. Mais la question se pose, un GNN peut-il imiter efficacement le comportement d'un algorithme de flot maximum coupe minimum donné, et si oui, quelles architectures ont été les plus performantes ? Telle est la question autour de laquelle s'est formée cette étude. Dans cette étude, nous allons étudier les perfomances des différentes architectures de GNNs par rapport à l'imitation d'un algorithme de flot maximum coupe minimum.

Contributions

Durant ce stage de recherche, nous avons, en premier lieu, exploré ce qu'est le raisonnement algorithmique neuronal, sa finalité, comment ce domaine essaie de rapprocher l'algorithmique et l'apprentissage profond. Nous avons étudié en profondeur le concept d'alignement algorithmique, le concept sur lequel se base le raisonnement algorithmique neuronal, comment l'architecture d'un réseau de neurones pouvait "s'aligner" sur un problème algorithmique ou un algorithme.

Ensuite, nous avons étudié les outils utilisés dans ce domaine, notamment, les réseaux de neurones sur les graphes ou GNN, nous nous sommes intéressés au paradigme de cette catégorie de réseaux de neurones et avons exploré les principales architectures connues à ce jour, en particulier, les architectures présentes dans le CLRS-30. Nous avons mis en évidence les différences entre les architectures GNN du CLRS-30, plus particulièrement, comment chaque architecture s'attaquait à un problème. Nous avons exploré comment les GNN gèrent les données représentées en graphe, comment cette catégorie de réseaux de neurones travaille sur la représentation des graphes. Durant cela, nous avons constaté les travaux sur l'optimisation combinatoire et les GNN, ainsi que les performances des GNNs sur ces tâches, et ainsi montré la validité de l'utilisation des GNNs pour le problème du flot maximum.

Puis nous avons étudié le CLRS-30, experimenté et utilisé ce cadre de travail dans le cadre de l'imitation d'algorithmes. Nous nous sommes familiarisés avec les différents moyens d'utiliser le CLRS-30 :

— l'ajout de nouvelles données pour les algorithmes qui ont déjà été abordés par les auteurs
— l'ajout de nouvelles architectures GNNs
— l'ajout et l'imitation de nouveaux algorithmes

Nous avons choisi de contribuer au CLRS-30, en ajoutant à la liste des algorithmes imités, un algorithme de flot maximum coupe minimum, puisque cette catégorie d'algorithme a été laissé par les auteurs en tant qu'amélioration future. Nous avons choisi d'ajouter l'algorithme POUSSER-RÉÉTIQUETER, variante de l'étiquette la plus élevée [21], un algorithme de flot maximum. Dans cette entreprise, nous avons étudié le problème de flot maximum-coupe minimum, le théorème de Ford-Fulkerson liant ces deux problèmes. Nous avons étudié les principaux algorithmes notoires qui ont été développés pour résoudre ces problèmes, étudié leurs compléxités en temps.

Nous avons également étudié la génération de graphe aléatoire, les différentes méthodes et les algorithmes existants pour générer des graphes de manières aléatoires, en particulier, nous avons étudié plus en détail le modèle d'Erdös-Rényi et les modèles d'attachement préférentiel. La simplicité du modèle d'Erdös-Rényi nous a convaincu dans le choix du générateur

de graphes pour l'entraînement des GNN dans le contexte du problème de flot maximum-coupe minimum.

Méthodologie

Pour se faire, nous allons en premier lieu étudier et comparer les algorithmes de flot maximum coupes minimum existants, nous nous limiterons à ceux donnant des solutions exactes et non approximatives. Ensuite, nous générerons des graphes aléatoires pour tester l'implémentation de l'algorithme choisi. L'outil Google OR-Tools (`https://developers.google.com/optimization`), un logiciel créé par Google pour l'optimisation combinatoire, sera utilisé pour témoigner de la véracité de l'implémentation de l'algorithme POUSSER-RÉÉTIQUETER, variante de l'étiquette la plus élevée. Puis nous suivrons les instructions du CLRS-30 pour l'ajout de l'algorithme au cadre de travail, après cela nous entraînerons six GNN à simple processeur, un pour chaque processeur précédemment mentionné, puis nous comparerons leurs performances sur l'imitation de l'algorithme.

Organisation du document

Le premier chapitre présente l'état de la question sur le problème de flot maximum coupe minimum, les algorithmes de résolution, la génération de réseau aléatoire, l'apprentissage profond, et les GNNs :

- La section 1.1 introduit le problème du flot maximum coupe minimum.
- La section 1.2 parle des algorithmes qui ont été développés pour résourdre ce problème, de leurs forces et faiblesses, de leurs complexités en temps.
- La section 1.3 nous introduit à la notion d'apprentissage profond.
- La section 1.4 est axée sur les GNN.
- La section 1.5 présente les généralités sur la génération de réseaux aléatoires.

Le second chapitre traite de a méthodologie suivie durant le travail effectué : :

- La section 2.1 détaille les données utilisées avec le CLRS-30.
- La section 2.2 parle de l'entraînement et des tests du réseau neuronal.
- La section 2.3 présente les procédés de collecte et de traitement des données.
- La section 2.4 décrit en détail l'expérimentation.
- La section 2.5 fournit les détails sur l'évaluation des performances du réseau de neurones.

Le troisième chapitre présente les résultats de l'expérimentation :

- La section 3.1 présente les résultats.
- La section 3.2 expose l'analyse et l'interprétation des résultats.
- La section 3.3 discute des résultats par rapport aux questions de recherche.

Le quatrième chapitre conclue le document sur une discussion des résultats obtenus, parle des contributions et des futures travaux en perspective.

Chapitre 1

Analyse de la littérature

Nous explorerons dans cette partie l'état actuel de la recherche sur le problème de flot maximum coupe minimum, ensuite nous ferons un tour d'horizon rapide du domaine de l'apprentissage automatique, puis nous introduirons les réseaux de neurones sur les graphes (GNN) et les principes de ces réseaux. Après cela, nous parlerons du CLRS-30 et de ce qu'on entend par raisonnement algorithmique neuronal. Enfin, nous dicuterons de la génération de graphes et de réseaux aléatoires, et des méthodes que nous utliserons pour générer les données d'entraînement.

1.1 Le problème du flot maximum coupe minimum

Considérons le scénario suivant : une entreprise a importé une quantité significative de produits, ces produits sont dans des conteneurs à Toamasina. L'entreprise en question voudrait ramener les produits jusque dans ses entrepôts de stockage à Antananarivo, pour cela, elle utlise des camions. Les camions transitent par des petites villes intermédiaires : Moramanga, Manjakandriana, Ambatondrazaka, Anjozorobe. Les produits n'y font que passer et n'y sont pas déchargés. Les routes sont cependant soumises à certaines contraintes, elles ne supportent qu'un certain poids de marchandise par jour. Entre Toamasina et Moramanga ne peuvent transiter qu'au maximum 10 tonnes de marchandises par jour ; entre Toamasina et Ambatondrazaka, 3 tonnes par jour ; entre Moramanga et Manjakandriana, 7 tonnes par jour ; entre Moramanga et Anjozorobe 2 tonnes par jour ; entre Ambatondrazaka et Anjozorobe, 3 tonnes par jour ; entre Manjakandriana et Antananarivo, entre Manjakandriana et Ambatondrazaka, 1 tonne par jour, 5 tonnes par jour ; entre Anjozorobe et Antananarivo, 5 tonnes par jour.

Nous pouvons modéliser ce scénario à l'aide du graphe/réseau ci-dessous : figure 1.1

La question que l'on se pose est : quelle quantité maximale de marchandise en tonnes par jour transite sur chaque arc pour que tout arrive rapidement aux entrepôts ? Le problème du flot maximum dans un réseau de transport y apporte une réponse.

Le problème du flot maximum a été formulé par Harris E. T. et Ross F.S. [27], le but étant une meilleure compréhension des réseaux férovières. Ford et Fulkerson [17] ont ensuite affiné le problème et formulé un algorithme pour résoudre le problème. Ils ont également formulé le

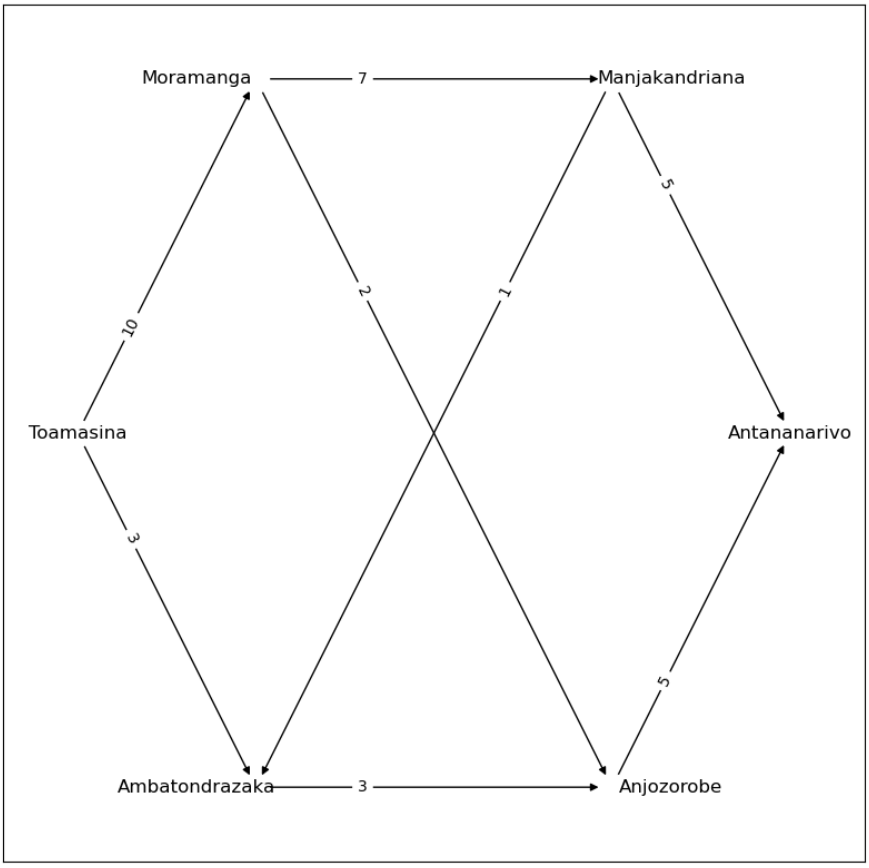

FIGURE 1.1 – Réseau de transport, exemple fictif

théorème de Ford-Fulkerson permettant de faire la correspondance entre coupe minimum et flot maximum.

1.1.1 Réseau de transport et flot

Un réseau de transport est un graphe orienté et connexe $G = (S, A)$, où chaque arc, $(u, v) \in A$, a une capacité $c(u, v) \geq 0$. Si $(u, v) \notin A$, on a, $c(u, v) = 0$, nous avons également : $|A| \geq |S| - 1$. Deux sommets de ce graphe se distinguent, la source **s** et le puits **t**. En général, la source n'a que des arcs sortants, et le puits que des arcs entrants, mais ce n'est pas toujours le cas. On peut noter par N = (G, c, s, t) le réseau de transport.

Un flot de G est une fonction à valeurs réelles $f : S \times S \to \mathbb{R}$, satisfaisant les conditions suivantes :

— **Contrainte de capacité** : Pour tout $u, v \in S$, on exige
$0 \leq f(u, v) \leq c(u, v)$.

— **Conservation du flot** : Pour tout $u \in S - \{s, t\}$, on exige
$\sum_{v \in S} f(u, v) = \sum_{w \in S} f(w, u)$.

La quantité $f(u, v)$ est appelé, flux du sommet u vers le sommet v. La **valeur** d'un flot se définie par :

$$|f| = \sum_{v \in S} f(s, v) - \sum_{v \in S} f(v, s) \tag{1.1}$$

Généralement, pour cette formulation, on exige également que si $(u, v) \in A$, $(v, u) \notin A$. Lorsque nous avons en même temps $(u, v) \in A$, et $(v, u) \in A$, nous choisissons l'un de ces arcs, par exemple (v, u), nous la décomposons en créant un sommet w de la manière suivante : $(v, w), (w, u)$, ensuite, les capacités des arcs (v, w) et (w, u) sont mises à celle de l'arc (v, u) d'origine, autrement dit : $c(v, w) = c(w, u) = c(v, u)$

Une formulation équivalente du flot, initiée par Dinitz [12] et fortement utilisée par Sleator [41] se définit comme suit :

Un flot de G est une fonction à valeurs réelles $f : S \times S \to \mathbb{R}$, satisfaisant les conditions suivantes :

— **Contrainte de capacité :** $f(v, w) \leq c(v, w)$ pour tout $(v, w) \in S \times S$

— **Antisymétrie :** $f(v, w) = -f(w, v)$ pour tout $(v, w) \in S \times S$

— **Conservation du flot** : Pour tout $v \in S - \{s, t\}$, on exige
$\sum_{u \in S} f(u, v) = 0$.

La **valeur** d'un flot se définie par :

$$|f| = \sum_{v \in S} f(v, t) \tag{1.2}$$

Cette notation ne met aucune condition sur les arcs antiparallèles, nous pouvons avoir $(u, v) \in A$, et $(v, u) \in A$, en effet la condition d'antisymétrie assure que seul l'un des arcs (u,v) ou (v,u) ait une quantité positive de flot. De plus dans cette formulation, si (u,v) $\notin A$, $c(u, v) = 0$.

Le **problème du flot maximum** est de trouver un flot de valeur maximum.

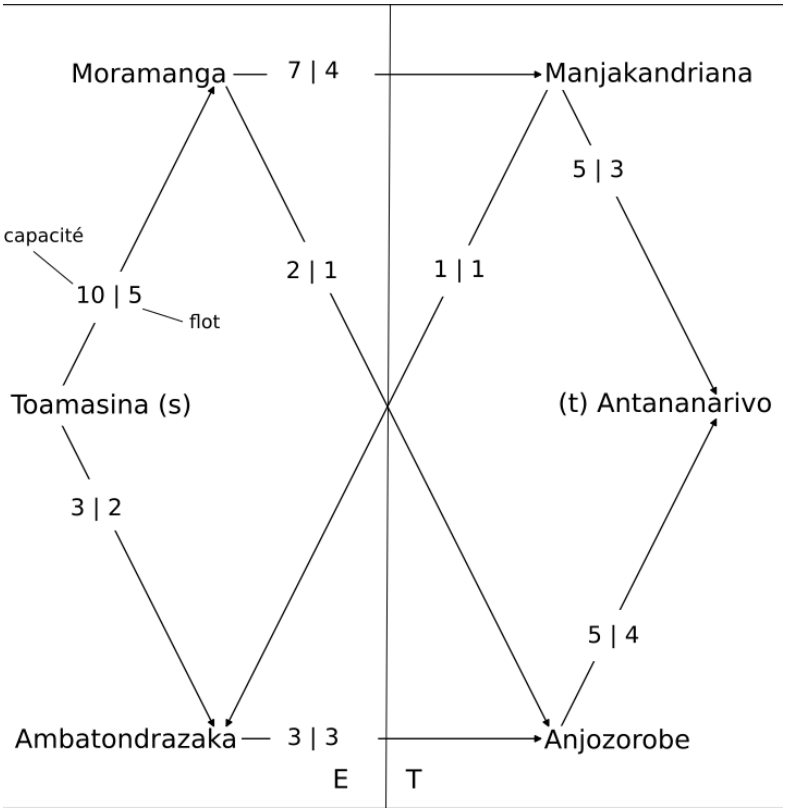

FIGURE 1.2 – Une coupe (E,T) dans le réseau de transport,
$E = \{Toamasina(s), Ambatondrazaka, Moramanga\}$,
$T = \{Anjozorobe, Manjakandriana, Antananarivo(t)\}$

1.1.2 Le problème de la coupe minimum

Une coupe (E, T), d'un réseau de transport $G = (S, A)$ est une partition de S dans E et $T = S - E$ telle que $s \in E$ et $t \in T$.

Suivant la notation en (1.1) :
si f est un flot, alors le flot net à travers la coupure (E, T) est défini par

$$f(E,T) = \sum_{u \in E} \sum_{v \in T} f(u,v) - \sum_{u \in E} \sum_{v \in T} f(v,u) \tag{1.3}$$

. La capacité de la coupe (E, T) est :

$$c(E,T) = \sum_{u \in E} \sum_{v \in T} c(u,v) \tag{1.4}$$

Une **coupe minimum** d'un réseau est une coupe dont la capacité est minimale rapportée à toutes les coupes du réseau.

Le théorème de Ford-Fulkerson [17] énonce que la valeur maximale du flot est égale à la capacité de la coupe minimale, mais aussi que lorsqu'il n'y plus de chemins améliorants dans

le réseau résiduel, le flot est maximum.

1.2 Les algorithmes résolvant le problème du flot maximum coupe minimum

1.2.1 La méthode de Ford-Fulkerson

La première méthode formulée pour la résolution du problème de flot maximum a été développée par Ford et Fulkerson, il s'agit de la méthode de Ford-Fulkersonn [17], on parle plus de méthode que d'algorithme car la méthode peut faillir pour un réseau dont les capacités sont réelles, et sa complexité dépend fortement du flot [9]. Elle se base sur les concepts de réseau résiduel et de chemins améliorants.

Réseau résiduel et arcs résiduels

Soit N = (G, c, s, t) un réseau de transport.

Capacité résiduelle Nous utiliserons en premier lieu la formulation (1.1). Étant donné un flot f, on définit la capacité résiduelle comme étant la fonction $c_f : V \times V \to \mathbb{R}$:

$$c_f(v,w) = \begin{cases} c(v,w) - f(v,w), & \text{si } (v,w) \in A \\ f(w,v), & \text{si } (w,v) \in A \\ 0 & \text{sinon} \end{cases} \tag{1.5}$$

Un seul de ces cas se présentera car la formulation (1.1) exige l'abscence d'arcs antiparallèles.

Pour la notation (1.2), celle qui suit [41], on a uniquement :

$$c_f(v,w) = c(v,w) - f(v,w) \tag{1.6}$$

Réseau résiduel Étant donné un réseau de transport (G,c,s,t), où G = (S,A), et un flot f, le réseau résiduel de G induit par f est $G_f = (S, A_f)$ où :

$$A_f = \{(u,v) \in S \times S : c_f(u,v) > 0\} \tag{1.7}$$

En voici un exemple, partons de la figure 1.1, et considérons le flot sur la figure de la page suivante. Le symbole | délimite la capacité de l'arc et le flot de l'arc. La capacité est à la gauche de |, le flot à sa droite (figure 1.2).

Construisons maintenant le réseau résiduel induit par f (figure 1.3).

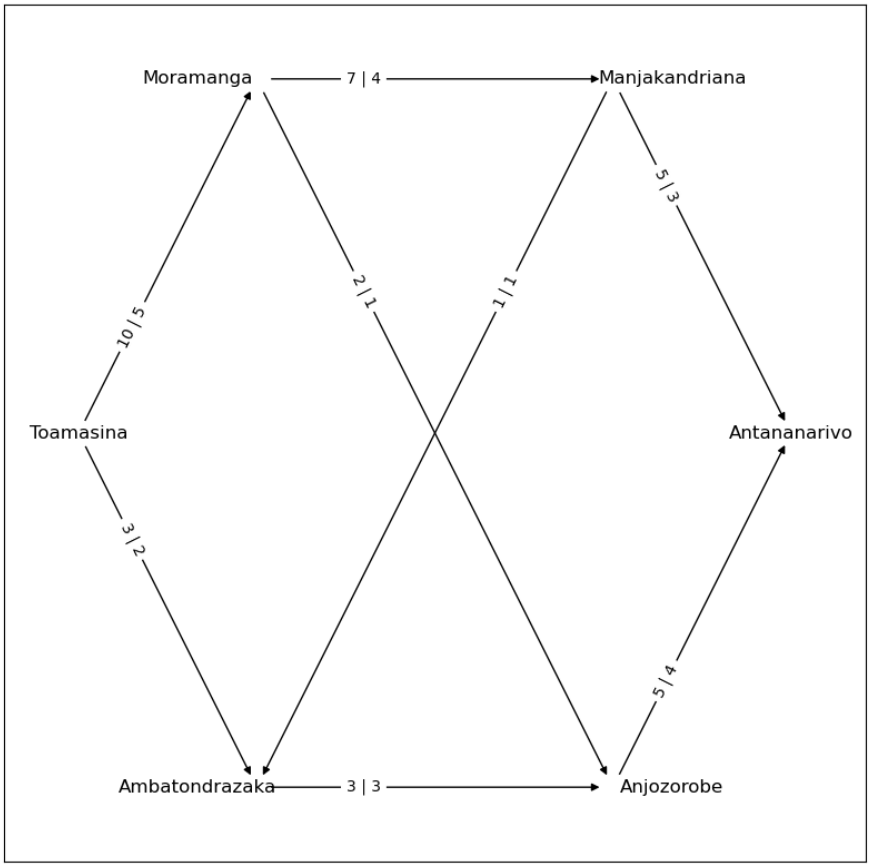

FIGURE 1.3 – Flot dans le réseau de transport de la figure 1.1

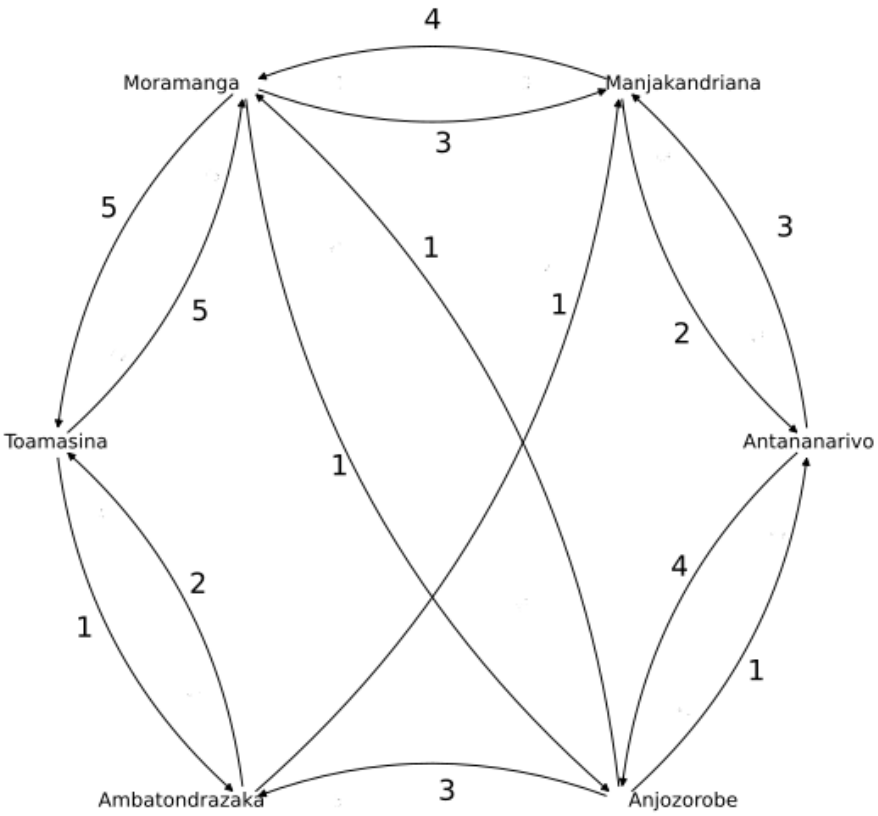

FIGURE 1.4 – Réseau résiduel induit par le flot

Chemin améliorant

Soit N = (G, c, s, t) un réseau de transport, et f, $f : S \times S \to \mathbb{R}$, un flot. Suivant Sleator [32], un chemin améliorant par rapport à f est un chemin de s à t où les arcs avançants (u, v) satisfont : $f(u, v) < c(u, v)$, et les arcs reculants (v, u) satisfont : $f(v, u) > 0$

Il existe une autre définition, mais pour cela, définissons d'abord ce qu'est un **chemin** en théorie des graphes.

Soit G = (S,A), un graphe, avec S l'ensemble des sommets, et A, l'ensemble des arcs. On appelle chemin p, une séquence de sommets $v_1, v_2, v_3, ..., v_n$, telle que chaque paire de sommets est reliée par un arc, c'est-à-dire, pour tout i=1,2,...,n-1, $(v_i, v_{i+1}) \in A$.

Lorsque G est un graphe orienté, une notion supplémentaire s'ajoute, celle de chemin orienté.

Un chemin p est dit orienté si chaque sommets qui le compose est relié à son voisin dans la séquence, autrement dit, pour tout i=1,2,...n-1, il existe un arc orienté $(v_i, v_{i+1}) \in A$. Il est à noter également que les sommets et les arcs d'un chemin sont distincts, un sommet n'apparaît qu'une seule fois dans la séquence, de même pour un arc.

Pour reprendre la définition d'un chemin améliorant : un chemin améliorant est un chemin élémentaire de s à t dans le réseau résiduel.

Énoncé de la méthode de Ford-Fulkerson

Nous allons exposer cette méthode à l'aide de la version tirée du CLRS [9].

Algorithme 1 : FORDFULK

Données : $G = (S, A)$, s, t
Résultat : flot f

```
pour (u, v) ∈ G.A faire
    (u, v).f ← 0 ;
tant que il existe un chemin p de s à t dans le réseau résiduel G_f faire
    c_f(p) ← min{c_f(u, v) : (u, v) est dans p } ;
    pour (u, v) dans p faire
        si (u, v) ∈ A alors
            (u, v).f ← (u, v).f + c_f(p) ;
        sinon
            (v, u).f ← (v, u).f − c_f(p) ;
```

Cette méthode se base sur le théorème de Ford-Fulkerson, en effet, le théorème de Ford-Fulkerson dit que s'il n'y a plus de chemins améliorants dans le réseau résiduel, c'est que le flot f est maximum.

Il a été dit précédemment que la complexité en temps de cette méthode était pseudo-polynomial. En effet, en supposant que les arcs ont des capacités entières, on a une compléxité de $O(|A||f_{max}|)$ pour un réseau de transport N = (G, c, s, t), de graphe orienté G = (S,A), de capacité c, de source s et de puits t. Ici $|f_{max}|$ désigne la valeur maximale du flot.

Après la méthode de Ford-Fulkerson, Edmonds et Karp, ainsi que Dinitz ont proposé des améliorations, basées toutes deux sur le chemin le plus court dans le réseau résiduel, mais se distinguant par leurs détails et leurs compléxités en temps.

1.2.2 L'algorithme d'Edmonds-Karp

Comme souligné précédemment, l'algorithme d'Edmonds-Karp [14], choisi le chemin améliorant le plus court de s à t dans le réseau résiduel, pour cela, les arcs résiduels sont considérés de distance unitaire, et un parcours en largeur est ensuite effectué à partir de la source pour trouver le chemin le plus court jusqu'à t.

Énoncé de l'algorithme d'Edmonds-Karp

Algorithme 2 : Edmonds-Karp

Données : $G = (S, A)$, s, t
Résultat : flot f

pour $(u, v) \in G.A$ **faire**
  $(u, v).f \leftarrow 0$;
tant que *il existe un chemin de s à t dans le réseau résiduel* G_f **faire**
  $p \leftarrow$ chemin de s à t le plus court dans le réseau résiduel G_f;
  $c_f(p) \leftarrow min\{c_f(u, v) : (u, v)$ est dans p } ;
  pour (u, v) *dans p* **faire**
    si $(u, v) \in A$ **alors**
      $(u, v).f \leftarrow (u, v).f + c_f(p)$;
    sinon
      $(v, u).f \leftarrow (v, u).f - c_f(p)$;

L'algorithme d'Edmonds-Karp a une compléxité de $O(|S||A|^2)$, et peut donner une solution aux réseaux à capacités réelles.

1.2.3 L'algorithme de Dinitz

Yefim Dinitz a développé un algorithme basé sur les mêmes idées qu'Edmonds-Karp [11], à la différence près qu'au lieu de ne considérer qu'un seul plus court chemin, il considère tous les plus courts chemins existants [12].

L'algorithme de Dinitz se base sur quatres principes : le réseau résiduel déjà vu précédemment, le **réseau en couches** le **graphe de niveau**, et les **flots bloquants**. Voyons les définitions formelles de graphe de niveu, réseau en couches et flot bloquant. Pour le graphe orienté du réseau de transport, il s'assure que les arcs (u,v) et (v,u) existent, et ajoutent l'une d'entre elles avec une capacité nulle si ce n'est pas le cas.

Graphe de niveau

Soit f un flot, N = (G, c, s, t), un réseau de transport de graphe G = (S,A), de capacité c : S x S $\rightarrow \mathbb{R}$, de source s, et de puits t. Soit $G_f = (S, A_f)$, le réseau résiduel induit par f. Pour rester consistant, soit $\delta_f(s, u)$, la distance minimale de u par rapport à la source s dans le réseau résiduel G_f, les arcs sont de distances unitaires.

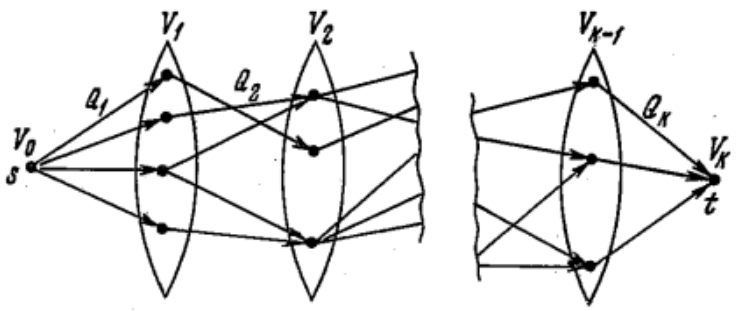

FIGURE 1.5 – Réseau en couches [11]

Le graphe de niveau de G_f est le graphe $G_L = (S, A_L)$ où
$A_L = \{(u, v) \in A_f : \delta_f(s, v) = \delta_f(s, u) + 1\}$.

Réseau en couches

Le réseaux en couche est un graphe orienté où les sommets sont répartis en couches et où toutes les arêtes vont des sommets d'une couche aux sommets de la couche suivante. L'idée de base de l'algorithme de Dinitz est d'utiliser les réseaux en couches pour trouver efficacement des chemins d'augmentation dans le réseau résiduel, ce qui permet d'accroître le flux dans le graphe.

Pour représenter mathématiquement un réseau en couches, nous pouvons définir un graphe $G = (S, A)$ avec un sommet source s et un sommet puits t. Nous définissons également un ensemble de couches $L_0, L_1, \ldots, L_k$ telles que $s \in L_0$ et $t \in L_k$. Ensuite, pour chaque couche i, nous définissons l'ensemble des sommets S_i et l'ensemble des arêtes A_i qui vont des sommets de S_i aux sommets de S_{i+1}. Nous pouvons l'écrire comme suit [11] :

$$S_i = \{v \in S : \text{ v est dans la couche i}\} \tag{1.8}$$

$$A_i = \{(u, v) \in A : u \in S_{i-1}, v \in S_i\} \tag{1.9}$$

On observe sur l'image que Dinitz a utilisé la notation V_i au lieu de S_i, et Q_i au lieu de A_i (figure 1.5).

Flot bloquant

Soit N = (G, c, s, t) un réseau de transport de graphe orienté G = (S, A), de source s, de puits t, et de foncion de capacité c. Soient f un flot, et h un flot sur le réseau résiduel G_f. On dit que h est un flot bloquant si pour chaque chemin améliorant le plus court, il existe un arc qui est saturé par h ($\exists a = (u, v), c(u, v) = h(u, v)$), et si chaque arc $a, h(a) > 0$, appartient à un chemin améliorant le plus court. [10].

Énoncé de l'algorithme de Dinitz

Tiré de l'énoncé de l'algorithme de Dinitz dans le livre de Jungnickel : raphs, Networks and Algorithms (Algorithms and Computation in Ma- thematics). Berlin, Heidelberg : Springer [32], et très légèrement modifié à l'aide d'une pile pour enregistrer le chemin le plus court, voici un énoncé de l'algorithme de Dinitz.

Remarques :

— $d_{in}(v)$ est le degré entrant de v, c'est-à-dire, $|\{a \in A : a = (u, v)\}|$

— a^- désigne le sommet commençant un arc, exemple : (u,v) = a, $a^- = u$ et $a^+ = v$.

Algorithme 3 : BLOCKFLOW

Données : $G = (\bigcup_{i=0}^{i=k} L_i, \bigcup_{i=1}^{i=k} A_i)$,
$N' =$
(G, c, s, t) un réseau de transport en couches avec les couches $L_0, L_1, \ldots, L_k$,
construit à partir du réseau N.
f , flot utilisé pour construire les couches

Résultat : g flot bloquant

```
1  pour a ∈ A faire
2      g(a) ← 0 ;
3  répéter
4      Q ← ∅;
5      v ← t, b ← ∞ ;
6      pour i = k à 1 faire
7          choisir un arc a_i = (u, v) ∈ A_i;
8          pousser a_i dans Q;
9          b ← min{c(a_i), b}, v ← u;
10     pour i = 1 à k faire
11         a_i ← Q.retirer l'arc au sommet de la pile;
12         g(a_i) ← g(a_i) + b, c(a_i) ← c(a_i) − b;
13         si c(a_i) alors
14             oublier a_i de A;
15     pour i = 1 à k faire
16         pour v ∈ L_i faire
17             si d_in(v) = 0 alors
18                 oublier v et tous les arcs a tels que a^- = v;
19 jusqu'à t ∉ L_k;
```

Algorithme 4 : Dinitz [10]

Données : $G = (S, A)$,
$N = (G, c, s, t)$ un réseau de transport
Résultat : f flot maximum

```
1 f ← 0, G_f ← G;
2 tant que G_f a un chemin de s à t faire
3     construire N', le réseau en couche à l'aide d'un parcours en largeur sur G_f;
4     h ← BLOCKFLOW(G, N');
5     f ← f + h mettre à jour G_f;
```

L'algorithme de Dinitz a une compléxité de $O(|S|^2|A|)$ pour un réseau de transport N = (G, c, s, t), avec G = (S, A).

1.2.4 Préflots et algorithmes "pousser-réétiqueter"

Les algorithmes de préflots font partie des plus rapides dans la litérature [9], ils se basent sur trois principes : le principe de préflot, introduit par Karzanov en 1974 [21], un concept similaire à celui de flot à une excéption près ; l'excédent de flot, une relaxation dûe au préflot ; et la fonction de hauteur, un concept liée à la distance par rapport au puits t. Goldberg et Tarjan dans leur article [21], introduisent l'algorithme générique de Goldberg-Tarjan, ainsi que les diverses modifications de cet algorithme pour améliorer la compléxité en temps. En premier lieux, voyons les définitions des principes des algorithmes de préflots.

Préflot

Suivant la notation de Sleator [41], on définit un préflot comme une correspondance $f : S \times S \to \mathbb{R}$, satisfaisant les conditions de contraintes de capacités et d'antisymétrie, ainsi que la version plus faible de la conservation du flot suivante :

— Pour tout $v \in S - \{s\}$, on exige
$\sum_{u \in S} f(u, v) \geq 0$.

Nous n'exigeons plus que le flux qui entre d'un sommet soit égal au flux qui en sort, le flux entrant doit être au moins plus grand que celui qui en sort, on autorise un excédent.

Excédent de flot

Soit f un préflot, on définit l'excédent de flot en un sommet v comme la valeur :

$$e(v) = \sum_{u \in S} f(u, v) \tag{1.10}$$

On dit qu'un sommet v déborde si $e(v) > 0$.

Capacité résiduelle

Étant donné un préflot f, on définit la capacité résiduelle comme étant la fonction $c_f : V \times V \rightarrow \mathbb{R}$:

$$c_f(v, w) = c(v, w) - f(v, w) \tag{1.11}$$

Il s'agit de la même définition que pour le flot.

Fonction de hauteur

Soit N = (G, c, s, t) un réseau de transport, de graphe orienté G = (S,A), de fonction de capacité c, de source s et de puits t. Soit f un préflot de G. la fonction de hauteur est une fonction $h : S \rightarrow N$ satisfaisant les conditions suivantes :

1. $h(s) = |S|$ et $h(t) = 0$
2. pour tout arc résiduel $(u, v) \in A_f$, on a :
   $h(u) \leq h(v) + 1$

Énoncé de l'algorithme générique de Goldberg-Tarjan

L'algorithme générique de Goldberg-Tarjan se basent sur deux opérations élémentaires : POUSSER et RÉÉTIQUETER. Voici une traduction du pseudo-code de ces opérations vu dans l'article de Goldberg et Tarjan [21].

Algorithme 5 : POUSSER

Données : (u, v)
Admissibilité : u est débordant, $c_f(u, v) > 0$, et $h(u) = h(v) + 1$;
Action : envoyer $\delta = min(e(u), c_f(u, v))$ unités de flots de u à v ;
$f(u, v) \leftarrow f(u, v) + \delta$;
$f(v, u) \leftarrow f(v, u) - \delta$;
$e(u) \leftarrow e(u) - \delta$;
$e(v) \leftarrow e(v) + \delta$;

Algorithme 6 : RÉÉTIQUETER

Données : u
Admissibilité : u est débordant, pour tout v $\in S$, $c_f(u, v) > 0 \implies h(u) \leq h(v)$;
$d(u) \leftarrow min\{d(v) + 1 : (u, v) \in A_f\}$;

L'algorithme générique de Goldberg-Tarjan s'énonce comme suit :

Algorithme 7 : GOLDBERG-TARJAN

Données : $N = (G, c, s, t)$
Résultat : f
`/* Initialisation du préflot */`
pour $\forall (u, v) \in (S - \{s\}) \times (S - \{s\})$ **faire**
  $f(u, v) \leftarrow 0$;
  $f(v, u) \leftarrow 0$;
pour $\forall u \in S$ **faire**
  $f(s, u) \leftarrow c(s, u)$;
  $f(u, s) \leftarrow -c(s, u)$;
`/* Initialisation de la fonction de hauteur et des excédents */`
$h(s) \leftarrow |S|$;
pour $\forall u \in S - \{s\}$ **faire**
  $h(u) \leftarrow 0$;
  $e(u) \leftarrow f(s, u)$;
`/* Boucle principale */`
tant que $\exists$ *opération élémentaire admissible* **faire**
  Choisir une opération admissible et l'exécuter;

L'algorithme générique de Goldberg-Tarjan a une compléxité de $O(|S|^2|A|)$.

Dans leur article, Goldberg et Tarjan ont mentionné des améliorations possibles à leur algorithme, notamment l'utilisation de structures de données supplémentaires, telles que l'utilisation des listes d'incidence ($I(u) = \{(u, v) \in A$ ou $(v, u) \in A : v \in S\}$) pour chaque sommet u dans S, couplée à l'utilisation soit d'une file FIFO pour la version FIFO de l'algorithme POUSSER-RÉÉTIQUETER, soit d'une file à priorité pour la version de l'étiquette la plus élevée, avec l'étiquette comme priorité des sommets (notée Q) [21]. La litératture considère la version de l'étiquette la plus élevée comme la plus performante, surtout en pratique, Goldberg et Tarjan ont initialement donné une complexité de $O(|S|^3)$ à cette version, mais cette analyse a été affiné par Cheriyan et Maheshwari [8], et ils ont démontré une complexité de $O(|S|^2|A|^{1/2}$. Ils ont également montré que la version FIFO avait une complexité de $O(|S|^3)$.

Énoncé de la version de l'étiquette la plus élevée

Il s'agit d'une traduction et d'une modification de la version FIFO de l'algorithme de Goldberg-Tarjan [21].

En annexe 3.4 sera donné la version avec la coupe minimum accompagné de l'implémentation en python de cet algorithme. Voilà en ce qui concerne les algorithmes les plus populaires pour la résolution du priblème du flot maximum. Pour résumer tout cela, voyons à l'aide de ce tableau un récapitulatif de chaque algorithme, de leurs forces et de leurs faiblesses.

Algorithme 8 : POUSSER/RÉÉTIQUETER

Données : u
Admissibilité : u est débordant ;
$u, v \leftarrow$ l'arc actuellement considéré dans la liste d'incidence de u ;
si *POUSSER(u,v) est admissible* **alors**
    POUSSER(u,v) ;
    si $v \notin Q$ *et* $v \neq s, t$ **alors**
        insérer v dans Q avec une priorité h(v);
sinon
    si *(u,v) n'est pas le dernier arc dans la liste d'incidence de u* **alors**
        remplacer (u,v) par le prochain arc dans la liste d'incidence de u ;
    sinon
        faire du premier arc dans la liste d'incidence de u l'arc actuel à considérer;
        RÉÉTIQUETER(u) ;

Algorithme 9 : DÉCHARGER

Données : u
Admissibilité : Q $\neq \emptyset$; retirer le sommet u de Q;
faire
    POUSSER/RÉÉTIQUETER(u) ;
jusqu'à ce que *e(u) = 0 ou d(u) a augmenté*;
si $e(u) > 0$ **alors**
    insérer u dans Q avec une priorité d(u);

Algorithme 10 : ÉTIQUETTE-LA-PLUS-ÉLEVÉE

Données : $N = (G, c, s, t)$, avec G = (S,A) un graphe orienté symétrique
I, listes d'incidence pour chaque sommet de G.
Résultat : f

`/* Initialisation du préflot */`
pour $\forall (u, v) \in (S - \{s\}) \times (S - \{s\})$ **faire**
    $f(u, v) \leftarrow 0$;
    $f(v, u) \leftarrow 0$;
pour $\forall u \in S$ **faire**
    $f(s, u) \leftarrow c(s, u)$;
    $f(u, s) \leftarrow -c(s, u)$;
`/* Initialisation de la fonction de hauteur et des excédents */`
$h(s) \leftarrow |S|$;
$Q \leftarrow \emptyset$;
pour $\forall u \in S - \{s\}$ **faire**
    $h(u) \leftarrow 0$;
    $e(u) \leftarrow f(s, u)$;
    `/* Ajout des modifications pour DÉCHARGER */`
    faire du premier arc dans $I(u)$, l'arc actuel à considérer pour u;
    si $e(u) > 0$ *et* $u \neq t$ **alors**
        insérer u dans Q avec une priorité d(u);
`/* Boucle principale */`
tant que $Q \neq \emptyset$ **faire**
    $u \leftarrow$ premier sommet de Q ; `/* sommet avec la plus grande priorité */`
    DÉCHARGER(u);

1.2.5 Récaputulatif

TABLE 1.1 – tableau récapitulatif sur les algorithmes de flot maximum

Algorithme	Compléxité en temps	Forces	Faiblesses
Ford-Fulkerson	$O(\|A\|\|f_{max}\|)$	— énoncé simple.	— pseudo-polynomial ; — peut faillir pour des capacités réelles ;
Edmonds-Karp	$O(\|S\|\|A\|^2)$	— fortement polynomial — capable de gérer des capacités réelles	— ne considère qu'un seul chemin améliorant le plus court à la fois ; — moins performant sur des réseaux denses (|A| élevé) ;
Dinitz	$O(\|S\|^2\|A\|)$	— fortement polynomial ; — capable de gérer des capacités réelles ; — considère tous les chemins les plus courts à la fois — performant sur des réseaux creux.	— moins performants sur des réseaux avec de nombreux sommets.
Goldberg-Tarjan (générique)	$O(\|S\|^2\|A\|)$	— fortement polynomial ; — capable de gérer des capacités réelles ; — performant sur des réseaux creux.	— plus gourmand en mémoire à cause du nombre de variables (ajout des excédents et des hauteurs pour les sommets).
Goldberg-Tarjan (étiquette la plus élevée)	$O(\|S\|^2\|A\|^{1/2})$	— fortement polynomial ; — capable de gérer des capacités réelles ; — particulièrement intéressant pour les réseaux creux ; — considéré comme le plus performant en pratique ;	— plus gourmand en mémoire à cause du nombre de variables et des structures de données supplémentaires (pile Q, graphe symmétrique G,excédents et hauteurs des sommets).

En tenant compte de ces faits, nous avons choisi d'imiter l'algorithme de Goldberg-Tarjan, **variante de l'étiquette la plus élevée**. Nous utiliserons ses entrées, les trajectoires de ses variables et ses sorties en tant que données d'entraînement.

A présent, voyons rapidement ce qu'est l'apprentissage automatique, et faisons un tour d'horizon des avancées qui y ont eu lieu.

1.3 Apprentissage automatique

L'apprentissage automatique est un sous-domaine de l'intelligence artificielle qui se concentre sur la construction de systèmes capables d'apprendre à partir de données et de faire des prédictions ou de prendre des décisions sans être explicitement programmés [1]. Il a été largement appliqué dans divers domaines, notamment le traitement du langage naturel (NLP), la vision par ordinateur, la robotique et la finance, pour n'en citer que quelques-uns [31]. Elle compte quatres grandes famille de méthodes : l'apprentissage supervisé, l'apprentissage non-supervisé, l'apprentissage par renforcement et l'apprentissage profond.

1.3.1 Apprentissage supervisé

L'apprentissage supervisé est un type d'apprentissage automatique dans lequel l'algorithme est formé sur des données étiquetées, où chaque exemple a une entrée et une sortie [3]. L'objectif de l'algorithme est d'apprendre une correspondance entre les entrées et les sorties, de manière à pouvoir prédire la sortie pour de nouvelles entrées inédites. La classification est un domaine où ce type d'apprentissage est le plus utilisé. L'un des algorithmes d'apprentissage supervisé les plus populaires est la machine à vecteur de support (SVM) [45]. Il y a également les problèmes de régression, dont le but est de prédire une variable réelle à partir des données.

1.3.2 Apprentissage non-supervisé

L'apprentissage non-supervisé est un type d'apprentissage automatique dans lequel l'algorithme est formé sur des données non étiquetées et dont l'objectif est de trouver des modèles ou des structures cachés dans les données [28]. Le regroupement(clustering en anglais) est une technique courante d'apprentissage non supervisé dans laquelle l'algorithme regroupe des points de données similaires. L'un des algorithmes de regroupement les plus populaires est celui des k-moyennes, qui répartit les données en k groupes en fonction de leur distance aux centroïdes.

1.3.3 Apprentissage par renforcement

L'apprentissage par renforcement est un type d'apprentissage automatique dans lequel l'algorithme apprend à prendre des décisions en fonction du retour d'information de l'environnement [44]. L'objectif de l'algorithme est d'apprendre une politique qui maximise un signal de récompense. L'apprentissage par renforcement a été appliqué avec succès à des jeux comme Atari, ainsi qu'à la robotique [36]. Parmi les succès retentissant de cette approche, il y a AlphaZero et AlphaGo [40], faisant des avancées dans le jeu de Go et celui des échecs, mais aussi AlphaTensor [16], ayant permi la découverte d'un nouvel algorithme de mutliplication matricielle.

1.3.4 Apprentissage profond

L'apprentissage profond est un sous-ensemble de l'apprentissage automatique qui se concentre sur la construction de réseaux neuronaux à couches multiples, également connus sous le nom de réseaux neuronaux profonds [22]. L'apprentissage profond a connu un succès remarquable

dans diverses applications, notamment la reconnaissance d'images et de la parole, le traitement du langage naturel et les jeux. Les réseaux neuronaux convolutifs (CNN) et les réseaux neuronaux récurrents (RNN) sont deux types populaires de réseaux neuronaux profonds, les CNN étant principalement utilisés dans le traitement des images et des vidéos, et les RNN étant utilisés dans des tâches de prédiction de séquences telles que la modélisation du langage et la reconnaissance vocale [24]. Dans le domaine du traitement du langage, une avance majeur a été le développement des Transformers [46], qui se basent le principe d'attention, un concept permettant de donner des scores d'importance à des mots en fonction de son contexte. Ce principe a été appliqué avec succès au domaine de la reconaissance d'image, les ViT (Vision Transformer) [13], pouvant ainsi donner des scores d'attention aux subdivisions d'une image.

Ces dernières années, l'apprentissage automatique a connu des avancées significatives avec le développement de nouveaux algorithmes, architectures et applications. Une avancée notable est l'utilisation de réseaux adversaires génératifs (GAN) pour générer des images, des vidéos et des sons réalistes [23]. Une autre avancée est le développement de l'apprentissage par transfert, qui permet d'affiner les modèles pré-entraînés pour de nouvelles tâches, réduisant ainsi le besoin de grandes quantités de données étiquetées [38].

Une autre classe de réseau de neurones est la classe des réseaux de neurones sur les graphes our encore les réseaux neuronaux graphiques (GNN). Ils sont une classe de modèles d'apprentissage profond qui ont été développés pour traiter les données sous forme de graphes ou de réseaux.

L'un des premiers travaux sur les GNN est le réseau convolutif graphique (GCN) proposé par Kipf et Welling [35]. Le GCN est un type de GNN qui utilise des opérations de convolution de graphe pour agréger les informations des nœuds voisins dans le graphe. Ce modèle a été appliqué avec succès à diverses tâches telles que la classification des nœuds, la prédiction des liens et la classification des graphes.

Les réseaux neuronaux à passage de messages (MPNN), proposés par Gilmer et al. [20], constituent une autre classe populaire de GNN. Les MPNN sont basés sur le paradigme du passage de messages, où chaque nœud du graphe reçoit des messages de ses nœuds voisins et met à jour son état en conséquence. Ce modèle a été appliqué avec succès à diverses tâches telles que la prédiction des propriétés moléculaires et la découverte de médicaments.

Ces dernières années, l'utilisation des GNN pour les systèmes de recommandation a suscité un intérêt croissant. L'un des modèles les plus performants dans ce domaine est le GraphSAGE proposé par Hamilton et al. [26]. GraphSAGE est un GNN qui apprend la représentation d'un nœud en agrégeant les informations de ses nœuds voisins dans le graphe. Ce modèle a été appliqué avec succès à diverses tâches de recommandation telles que la recommandation de films et de musique.

Bref, l'aprrentissage profond a vu d'énormes progrès se former, aussi bien dans les classes de réseaux de neurones convolutifs (CNN), les réseaux de neurones récurrents (RNN), les réseaux de neurones génératifs (GAN), que les réseaux de neurones graphiques (GNN).

Parlons désormais plus en détail des réseau de neurones sur les graphes.

1.4 Réseau de neurones sur les graphes (GNN)

Nous rencontrons des graphes dans de nombreux problèmes, à titre d'exemple, l'une des manières les plus efficaces d'étudier les réseaux sociaux est de les représenter en tant que graphes. Un graphe est une collection de deux ensembles, l'ensemble des sommets ou nœuds,

et l'ensemble des arcs qui décrivent comment les sommets sont reliés entre eux. Un graphe permet donc de modéliser les relations entre des objets.

Comme il a été mentionné précédemment, les CNNs et les RNNs ont montré d'excellents résultats dans le domaine de la vision par ordinateur, le traitement du langage naturel et la reconnaissance vocale. Ces objets (images, textes, voix) ont une propriété géométrique commune, une structure de grille sous-jacente [7]. En effet, une image peut-être vue comme un objet avec une structure de grille dans un espace Euclidien. De même, un graphe est un objet géométrique dans un espace non-Euclidien.

Les réseaux de neurones sur les graphes sont donc une tentative de réitérer les exploits réussis par les CNNs et les RNNs mais cette fois sur des graphes [55].

Les GNNs ont été proposé par Scarselli et al. [39], et Gorsi et al. . Voyons la formulation suivante provient que Scarselli et al. ont proposé dans leur article [39] :

Soit G = (S,A) un graphe, avec S comme l'ensemble des sommets et A l'ensemble des arcs. L'ensemble des voisins d'un sommet n, noté : $ne[n]$, est l'ensemble des sommets qui sont relié à n par un arc($\{v \in S : (v,u) \in A$ ou $(u,v) \in A\}$). L'ensemble noté $co[n]$ est l'ensemble des arcs de A ayant n pour sommet ($\{a \in A : a = (v,u)$ ou $a = (u,v)\}$). Les nœuds et les arcs ont des propriétés qui leur sont rattachées, elles sont généralement représentées à l'aide de vecteurs réels. Les propriétés attachées au sommet n sont représentées par $l_n \in \mathbb{R}^{l_S}$, où l_S est le nombre de propriétés rattachées à chaque sommet. Pour les arcs, elles sont représentées par $l_{(n_1,n_2)} \in \mathbb{R}^{l_A}$, l_A le nombre de propriétés rattachées à chaque arc. Le but d'un GNN est d'apprendre la représentation d'un état $h_v \in \mathbb{R}^s$ pour chaque sommet v. Cette réprésentation d'état h_v est ensuite utilisé pour le calcul d'une sortie o_v (par exemple la distribution des propriétés prédites de chaque sommet).

1.4.1 Modèle

Comme précisé ultérieurement, un GNN apprend la représentation d'un état h_v afin de produire une sortie o_v. Pour cela, on utilise une fonction paramétrée f, appelée, fonction locale de transition ; et une fonction paramétrée g, appelée fonction de sortie locale.

$$h_v = f(x_v, x_{co[v]}, h_{ne[v]}, x_{ne[v]}) \tag{1.12}$$

$$o_v = g(h_v, x_v) \tag{1.13}$$

x désigne les caractéristiques d'entrées (les données), et h est l'état que l'on cherche à apprendre. Ainsi, on a :

— x_v sont les caractéristiques du sommet v ;
— $x_{co[v]}$ sont les caractéristiques des arcs ayant v comme sommet ;
— $h_{ne[v]}$ sont les états des sommets voisins de v ;
— $x_{ne[v]}$ sont les caractéristiques des sommets voisins de v ;

Soient H, O, X, X_S des matrices construites en empillant l'ensemble des états, des sorties, des caractéristiques et des caractéristiques de tous les sommets respectivement. En agrégeant toutes ses représentations, on a les équations suivantes :

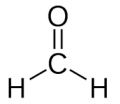

FIGURE 1.6 – CH_2O

$$H = F(H, X) \tag{1.14}$$

$$O = G(H, X_S) \tag{1.15}$$

F est la fonction globale de transition, G, la fonction globale de sortie. Ces fonctions sont les versions empillées des fonctions locales de transition et de sortie respectivement. F est supposée être une application contractance, et H est son point fixe. Le théorème de Banach sur les points fixes [18] nous permet de calculer les états de manière itérative :

$$H^{(t+1)} = F(H^{(t)}, X) \tag{1.16}$$

De manière un peu plus simple, les GNNs rassemblent les représentations que chaque nœud a de ses voisins, et utilisent également la représentation du nœud lui-même pour calculer le prochain état. Hamilton a exprimé ce procédé de la manière suivante [25] :

$$h_v^{(t+1)}) = MAJ^{(t)}(h_v^{(t)}, AGREGER^{(t)}(\{h_u^{(t)}, \forall u \in ne[v]\})) \tag{1.17}$$

$$= MAJ^{(t)}(h_v^{(t)}, m_{ne[v]}^{(t)}) \tag{1.18}$$

où MAJ : Mise à Jour, désigne le procédé pour mettre à jour les états ; et AGREGER désigne le procédé pour rassembler les représentations ; $m_{ne[v]}^{(t)}$ est le message obtenu à partir de l'agrégation des représentations du voisinage de v. Il a également souligné que ce qui distingue les différentes architectures de GNNs est la définition de ces procédés.

Les fonctions d'agrégation se doivent d'être invariantes par permutation. En effet, l'isomorphisme de graphe ne doit pas changer les propriétés du graphe, et donc toutes transformations isomorphes les conservent.

En ce qui concerne la fonction de perte, elle dépend des tâches que l'on confie aux GNN. Pour mieux comprendre, considérons l'exemple suivant : prenons la molécule CH_2O. Nous voyons que les atomes sont liés par des liaisons covalentes, pour les propriétés liées aux sommets, considérons les numéro atomiques. On peut modéliser les liaisons à l'aide du graphe de la figure 1.6 :

On a $x_C = 6$, $x_O = 8$, $x_{H^{(1)}} = 1$, $x_{H^{(2)}} = 1$ (il faut distinguer les deux atomes d'hydrogènes puisqu'il s'agit de sommets différents). Les liaisons ont des propriétés différentes également, elles peuvent être simples, doubles, ou triples : $x_{(C,O)} = 2$, $x_{(C,H^{(1)})} = 1$, $x_{(C,H^{(2)})} = 1$. Voilà, ce que l'on entend par propriétés des sommets et des arcs.

Voyons désormais quelques architectures de GNN populaires.

1.4.2 Réseaux de neurones sur les graphes convolutifs

Les réseaux de neurones sur les graphes convolutifs (GCN) ont été introduit par Kimpf et Welling [35] pour des tâches de classification semi-supervisées. L'expression combinée de sa fonction d'agrégation et de sa fonction de mise à jour est :

$$h_u^{(t)} = \sigma\left(W^{(t)} \sum_{v \in ne[u] \cup \{u\}} \frac{h_v}{\sqrt{|ne[u]||ne[v]|}}\right) \tag{1.19}$$

où W est la matrice des paramètres à apprendre et σ une fonction non-linéaire (tanh ou ReLU par exemple). Les GCN sont une tentatives de généraliser les convolutions à des objets géométriques non-Euclidiens.

1.4.3 Ensembles profonds

Les ensembles profonds (Deep Sets en anglais) ont été proposé par Zaheer et al. [56] pour résoudre des problèmes nécéssitant l'utilisation d'ensembles ou de toutes structurées invariantes par permutation (ce qui est le cas des graphes). [25] a formulé la fonction de message suivante :

$$m_{ne[u]} = MLP_\theta\left(\sum_{v \in ne[u]} MLP_\phi(h(v)\right) \tag{1.20}$$

MLP désigne un réseau de perceptrons mulitcouches, θ et ϕ sont des paramètres que l'on peut entraîner durant l'apprentissage.

1.4.4 Graph Attention Networks

Les GAT, introduits par Veličković et al. [51], utilisent le principe d'attention sur les voisinages pour souligner l'importance de chaque voisinage afin de les pondérer correctement lors de l'agrégation. Les messages sont modifiés comme tel :

$$m_{ne[u]} = \sum_{v \in ne[u]} \alpha_{u,v} h_v \tag{1.21}$$

où $\alpha_{u,v}$ est le score d'attention sur le voisin v de u lors de l'agrégation de u.

$$\alpha_{u,v} = \frac{exp(a^T[Wh_u \oplus Wh_v])}{\sum_{v' \in ne[u]} exp(a^T[Wh_u \oplus Wh'_v])} \tag{1.22}$$

où $\oplus$ désigne la concaténation.

1.4.5 Message-passing Networks

Gilmer et al. [20] ont introduit cette architecture. Bien qu'elle soit similaire au concept de base des GNN, elle se distingue par un paradigme plus élaboré concernant l'agrégation des messages aussi bien sur les sommets que sur les arcs. Permettant à chaque sommet d'en connâitre plus que par rapport au GNN basique.

1.5 CLRS-30 et raisonnement algorithmique neuronal

1.5.1 Raisonnement algorithmique neuronal

Le raisonnement algorithmique neuronal est un domaine visant à rapprocher l'algorithmique classique du domaine de l'apprentissage profond [49]. Le but étant de permettre à des réseaux de neurones d'apprendre le comportement d'algorithmes classiques, avec pour finalité l'apprentissage de nouvel algorithme [16], mais aussi le fait d'augmenter le pouvoir de généralisation des réseaux de neurones. Effectivement, les algorithmes classiques sont connus pour leurs fortes capacités de généralisation, sous la condition que leurs entrées satisfassent certaines conditions. Permettre à des réseaux de neurones de travailler sur les données d'entrées, de sorties ainsi que les données intermédiaires d'algorithmes permettraient ainsi d'attaquer le problème de généralisation et de mieux comprendre les difficultés des réseaux de neurones à généraliser pour des données en dehors de la distribution du jeu d'entraînement.

C'est dans cet effort qu'est naît le CLRS-30.

1.5.2 Introduction au CLRS-30

Le CLRS-30 [47] est un cadre de travail developpé dans le but de résoudre le raisonnement algorithmique neuronal. Pour se faire, il vient avec un réseau de neurone sur les graphes pouvant accueillir plusieurs architectures de GNNs. Le nom CLRS vient du nom des auteurs du livre : Introduction to Algorithms, 3rd Edition. MIT Press, 2009, de Cormen, Leiserson, Rivest et Stein. Le nombre 30 provient du fait qu'au moment de la sortie officielle du cadre de travail, on pouvait compter 30 algorithmes dans la liste des algorithmes étudiés et imités à l'aide du GNN du CLRS-30.

1.5.3 Liens entre GNNs et les algorithmes

Pourquoi des GNNs ? La réponse à cette question est parce que les algorithmes sont des objets qui manipulent des graphes en permanence. En effet, considérons par exemple le problème de tri, tiré du CLRS-30 [47].

Comme nous le voyons sur la figure, la liste peut être représentée par un graphe, les flèches vertes représentant la relation "est précédé par". Les sommets ont des propriétés (la valeur contenue en elle), et ce sont ces propriétés et les arcs que manipulent un algorithme de tri. Cela rappelle le paradigme des GNNs qui manipulent les représentaitons des sommets, de leurs propriétés, les représentaitons des arcs, la représentation du graphe. On dit que les GNNs s'aligne algorithmiquement sur le problème [54]. Il est donc normal de développer des GNNs pour la tâche d'imiter des algorithmes du fait de cette ressemblance.

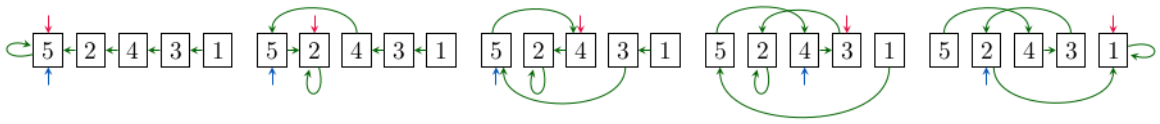

FIGURE 1.7 – Représentation en graphe d'un tableau et visualisation du tri [47]

1.5.4 Architecture du GNN

Le CLRS-30 travaille sur les sommets, les arcs et les graphes, en premier lieu, ces objets passent dans un encodeur pour que l'on puisse avoir une représentation de ces objets.

Considérons les entrées des sommets x_i, ceux des arcs e_{ij}, et le graphe g.

$$h_i = f_n(x_i)\ ,\ h_{ij} = f_e(e_{ij})\ ,\ h_g = f_g(g) \tag{1.23}$$

f_n, f_e, f_g sont des fonctions linéaires (couches linéaires). Ces états sont ensuites fournies aux différentes architectures présentes dans le CLRS-30, leurs fonctions messages et leurs fonctions d'agrégation. Il y a les architectures suivantes :

— Deep Sets [56]
— Graph Attention Networks [51]
— Graph Attention Networksv2 [6]
— Message-passing Networks [20]
— Pointer Graph Networks [50]
— Memory Networks [43]

Après traitement de ces processeurs, leurs sorties passent par un décodeur pour obtenir les prédictions.

1.6 Génération de réseaux aléatoires

Pour entraîner des modèles de réseaux neuronaux, nous avons besoin de données. Dans le cas actuel, nous avons besoin de générer des réseaux de transports.

Les réseaux et les graphes aléatoires ont été largement étudiés dans de nombreux domaines, notamment les mathématiques, l'informatique, la physique et les sciences sociales. Ces structures ont été utilisées pour modéliser une variété de systèmes complexes, y compris les réseaux sociaux, les réseaux biologiques, les réseaux de transport et les réseaux de communication. Dans cette revue de la littérature, nous nous concentrerons sur la génération de réseaux et de graphes aléatoires et sur leurs applications. Nous commencerons par présenter les concepts et modèles de base des réseaux et graphes aléatoires, puis nous passerons en revue certains des algorithmes les plus importants pour la génération de réseaux et de graphes aléatoires.

1.6.1 Concepts et modèles de génération de réseaux et graphes aléatoires

Le concept de réseaux et de graphes aléatoires a été introduit pour la première fois par Erdős et Rényi en 1959 [15]. Ils ont proposé un modèle simple pour générer des graphes aléatoires, connu sous le nom de modèle Erdős-Rényi (ER). Dans le modèle ER, un graphe est généré en connectant aléatoirement des paires de nœuds avec une probabilité fixe. Le modèle ER a été largement étudié et il a été démontré qu'il présente une transition de phase d'un graphe déconnecté à un graphe connecté lorsque la probabilité de connexion des nœuds augmente [4].

Un autre modèle important pour les réseaux et les graphes aléatoires est le modèle d'attachement préférentiel (PA), introduit par Barabási et Albert en 1999 [2]. Dans le modèle PA, les nœuds sont ajoutés au graphe un par un, et chaque nouveau nœud est connecté aux nœuds existants avec une probabilité proportionnelle à leur degré. Il a été démontré que le modèle PA présentait la propriété dite "sans échelle", où la distribution des degrés des nœuds suit une loi de puissance[37].

1.6.2 Algorithmes de génération de réseaux et de graphes aléatoires

Il existe plusieurs algorithmes permettant de générer des réseaux et des graphes aléatoires, notamment le modèle ER, le modèle PA et le modèle du petit monde. Le modèle du petit monde a été introduit par Watts et Strogatz en 1998 [52]. Dans ce modèle, un treillis régulier est recâblé avec une faible probabilité pour générer un graphe présentant à la fois une forte concentration et de courtes longueurs de chemin.

1.6.3 Applications des réseaux et graphes aléatoires

Les réseaux et les graphes aléatoires ont été utilisés pour modéliser une grande variété de systèmes complexes, notamment les réseaux sociaux, les réseaux biologiques, les réseaux de transport et les réseaux de communication. Dans les réseaux sociaux, les graphes aléatoires ont été utilisés pour modéliser la diffusion d'informations et de maladies, la formation d'amitiés et l'émergence de hiérarchies sociales. Dans les réseaux biologiques, les graphes aléatoires ont été utilisés pour modéliser les réseaux de régulation génétique, les réseaux d'interaction protéine-protéine et les réseaux métaboliques [10]. Dans les réseaux de transport, les graphes aléatoires ont été utilisés pour modéliser le flux de trafic et la conception des systèmes de transport [11]. Dans les réseaux de communication, les graphes aléatoires ont été utilisés pour modéliser le flux d'informations et la conception de protocoles de communication.

Après avoir tenu compte de ces faits, nous avons choisi le générateur de graphes aléatoires d'Erdős-Rényi du fait de sa simplicité et des exigences simples du CLRS-30 en terme de génération de graphes.

Chapitre 2

Méthodologie

2.1 Données d'entraînement du CLRS-30

Pour générer les données d'entraînement, deux étapes doivent être suivies :

1. générer les réseaux aléatoires à l'aide d'un échantillonneur
2. générer les trajectoires des variables de l'algorithme à imiter

2.1.1 Échantillonneur

Chaque algorithme du CLRS-30 doit être accompagné d'un échantillonneur. Pour pouvoir créer l'échantillonneur d'un algorithme, il convient de spécifier en premier lieu les variables utilisées au cours de l'algorithme.

Spécification des variables

Une variable donnée doit appartenir à une des phases suivantes :
— entrée (INPUT)
— indices (HINTS)
— sortie (OUTPUT)
Nous sommes habitués aux variables d'entrées et aux variables de sorties. Les variables d'indices eux sont une classe de variables définie par les auteurs du CLRS-30.

Indices (HINTS) :
pour mieux comprendre les indices, prenons l'exemple des algorithmes de tri. Fondamentalement, les algorithmes de tri prennent les mêmes données en entrée, et produisent évidemment les mêmes sorties (le tableau trié). Ce qui les différencient entre eux, ce sont les procédés de tri qu'ils utilisent, et donc leurs compléxités en temps. Pour distinguer les différents algorithmes, il est donc essentiel d'enregistrer l'état interne des variables intermédiaires, celles

qui ne sont ni en entrées ni en sortie. Ces variables constituent les indices. L'enregistrement de l'évolution des variables à chaque itération a été nommé par les auteurs "probing".

En plus de l'appartenance à une phase, il est essentiel de définir sur quelle partie du problème une variable agit. Comme on l'a dit ultérieurement, un problème algorithmique peut être vu comme un graphe. En tant que tel, les variables agissent soit sur les arcs, soit sur les sommets, soit sur le graphe. L'emplacement de ces actions définit ce qui est appelé la localisation d'une variable. Localisation :

- sommet (NODE)
- arc (EDGE)
- graphe (GRAPH)

Pour spécifier entièrement une variable, il reste à définir son type. Selon le type, une fonction de perte ainsi qu'une méthode d'encodage/de décodage différentes seront utilisées. Au sein du CLRS-30, une variable peut-être de cinq types différents :

- scalaire (scalar) : les nombres réelles à virgule flottante. Généralement, l'erreur quadratique moyenne sera utilisée pour les évaluer.
- catégorielle (categorical) : décrit l'appartenance à l'une de K classes possibles. La fonction de perte est l'entropie croisée sur les K classes.
- masque (mask) : décrit l'appartenance ou non à l'une des deux classes possibles. La fonction de perte sera l'entropie croisée binaire.
- "mask_one" : décrit l'appartenance ou non à l'une des deux classes (actif, inactif) possibles, avec en plus le fait qu'exactement un sommet peut-être "actif". La fonction de perte sera l'entropie croisée catégorielle.
- pointeur (pointer) : décrit l'appartenance à l'une des n classes, plus spécifiquement, pointe un des n sommets. La fonction de perte sera l'entropie croisée catégorielle.

Le triplet (phase, localisation, type) définit précisement le rôle d'une variable, pour compléter l'état interne, il reste à ajouter la valeur de la variable. Le quadruplet (phase, localisation, type, valeur) enregistre l'état interne d'une variable.

Spécifions les caractéristiques des variables utilisées pour l'algorithme de Goldberg-Tarjan, variante de l'étiquette la plus élevée :

- N = (G, c, s, t) : décomposé de la manière suivante
  - G : représenté à l'aide d'une matrice d'adjacence, avec les capacités comme poids des arcs, sa représentation et celle des capacités seront la matrice C, (phase : INPUT, localisation : EDGE, type : SCALAR) ;
  - s : le sommet source, (phase : INPUT, localisation : NODE, type : MASK_ONE) ;
  - t : le sommet puits, (phase : INPUT, localisation : NODE, type : MASK_ONE) ;
- S : coupe minimum, côté de s. (phase : OUTPUT, localisation : NODE, type : MASK) ;
- Q : file avec notion de priorité. (phase : HINT, localisation : NODE, type : MASK) ;
- active_edge : arc actif durant DÉCHARGER. (phase : HINT, localisation : NODE, type : MASK) ;
- f : préflot, représenté par une matrice carrée de même dimension que la matrice de capacités. (phase : HINT, localisation : EDGE, type : SCALAR) ;
- r_f : capacité résiduelle, matrice carrée de même dimension que la matrice de capacités. (phase : HINT, localisation : EDGE, type : SCALAR) ;
- e : excédent sur les sommets. (phase : HINT, localisation : NODE, type : SCALAR) ;
- d : hauteur des sommets. (phase : HINT, localisation : NODE, type : SCALAR) ;

Ces spécifications sont à entrées dans le fichier specs.py du CLRS-30.

Une fois les spécifications des variables terminées, on définit l'échantillonneur.

Échantillonneur : nous utiliserons les techniques de génération de graphes aléatoires utilisant le modèle d'Erdös-Rényi. L'échantillonneur est à définir dans le fichier samplers.py du CLRS-30. L'implémentation de l'échantillonneur peut être vue en annexe, avec tous les détails.

2.1.2 Génération des trajectoires

Une fois l'échantillonneur implémenté, il faut générer les trajectoires, pour cela, il faut implémenter l'algorithme de Goldberg-Tarjan, variante de l'étiquette la plus élevée, en choisissant des points de sauvegardes afin d'enregistrer l'état interne de chaque variable à chaque itération. Nous enregistrons en premiers lieu les variables d'entrée (INPUT), puis viennent les variables intermédiaires (HINTS), et enfin les variables de sorties (OUTPUT). L'enregistrement de l'état interne des variables trace les trajectoires suivies par chaque variable intermédiaire pour arriver aux données de sortie prévues. L'implémentation de l'algorithme est en annexe, il y sera également expliquée comment utiliser l'échantillonneur pour générer les trajectoires.

2.2 Entraînement et test des réseaux de neurones

Pour l'entraînement, nous avons suivi les recommandations du CLRS-30, mais dûes à des contraintes de mémoires, nous avons été obligé de réduire la taille des réseaux de transport, nous contentant de 1000 réseaux de quatre sommets et des trajectoires des variables dans ces 1000 réseaux pour la phase d'entraînement, 32 réseaux de 8 sommets et les trajectoires des variables pour ces 32 réseaux pour la phase de validation/évaluation, et enfin 32 réseaux de 16 sommets et les trajectoires des variables pour chaque réseau pour la phase de test.

La fonction de perte finale est la somme des différentes fonctions de pertes pour chaque variable et leurs spécifications.

En ce qui concerne l'évaluation du score d'une prédiction, comme dans notre cas, la sortie est de type masque, il s'agit d'une tâche de classification sur les sommets. On utilise alors le score micro-F1 :

$$Micro - F1 = \frac{VP}{VP + 0.5 \times (FP + FN)} \tag{2.1}$$

où VP signifie Vrai Positif, FP veut dire Faux Positif, et FN, Faux Négatif.

2.3 Génération et traitement des données

2.3.1 Génération des données

Comme dit dans le chapitre précédent, 1000 réseaux de transport de quatre sommets (1000 matrices d'adjacence/de capacités) sont générés à l'aide du modèle d'Erdös-Rényi. À l'aide d'un générateur de nombre pseudo-aléatoire, on génère la source s et le puits t pour chaque réseau. Ces 1000 réseaux, accompagnés de la source et du puits, sont ensuite fournis à l'implémentaiton de l'algorithme de Goldberg-Tarjan, variante de l'étiquette la plus élevée, pour générer les sorties et les trajectoires des variables du programme. Ensuite, 32 réseaux de

transport d'une taille de 8 sommets, leurs sources et leurs puits respectifs sont générés, puis donnés à l'algorithme. Enfin, 32 réseaux de transport d'une taille de 16 sommets, leurs sources et leurs puits respectifs sont générés, et donnés à l'algorithme.

2.3.2 Traitement des données

Une fois ces données générées, les trajectoires des variables sont ensuite traitées de la manière suivante :

1. les données sont séparées par phase : INPUT, HINTS, OUTPUTS ;
2. puis pour chaque phase, les données sont réparties selon la localisation pour chaque variable ;
3. pour chaque localisation(EDGE, NODE, GRAPH), les données de chaque variables sont empillées pour former la représentation des états.
4. c'est cette forme empillée qui est ensuite fournie au GNN pour l'entraînement. Nous n'avons pas tronqué les données mais décidé de fournir les matrices en entier.

2.4 Description de l'expérimentation

Nous avons créé une classe (functionalGNN), un GNN avec un seul processeur, basé sur l'exemple présent dans le CLRS-30 et celui exposé par Ibarz et al. [30]. De cette classe, nous avons instancié six GNNs, un pour chacun des processeurs suivant :

— MPNN
— Deep Sets
— GAT
— GATv2
— Memory Network
— PGN

Pour l'entraînement, nous itérerons sur 10 000 pas. Tous les 50 pas, nous évaluons le modèle sur les données de validation (les réseaux de 8 sommets). A la fin des 10 000 pas, nous effectuons un test sur les réseaux de 16 sommets et évaluons les scores.

L'entraînement a été réalisé sur un notebook de la plateforme Kaggle, sur deux GPUs NVIDIA T4. Les processeurs ont été implémenté en JAX [5] et Haiku[29]. JAX est un logiciel de calcul de dérivés optimisé pour accélérer les calculs à la mesure du possible selon le matériel disponible. Haiku quant à lui est une librairie de réseaux de neurones basé sur JAX, developpé, testé et maintenu par les chercheurs de chez Deepmind.

2.5 Évaluation des performances des réseaux de neurones

Tous les 50 pas, les réseaux de neurones sont évalués. Nous observons le score moyen calculé à partir des score Micro-F1 sur les 32 réseaux de transports. Les paramètres sont ensuite enregistrés si le score moyen bat le meilleur score précédent.

Chapitre 3

Résultats

3.1 Résultats des tests de l'implémentation personnelle de l'algorithme de l'étiquette la plus élevée face à Google OR-Tools

Nous avons testé l'implémentation de l'algorithme de Goldberg-Tarjan, variante de l'étiquette la plus élevée, face à l'implémentation de Google OR-Tools du même algorithme. Et sur les 10 000 réseaux de transports de taille 16 (16 sommets), nous avons eu 3 coupes différentes, cependant, la valeur de la coupe minimum, ainsi que la valeur du flot maximum ont été la même sur tous les réseaux. (`https://github.com/DimbyTa/CLRS_extended.git`)

3.2 Résultats sur les scores de validation et les scores de tests

Le tableau suivant présente les scores moyen pour les phases de validation :

TABLE 3.1 – Score moyen de validation

PGN	GATv2	MPNN	GAT	Deep Sets	Memory Network
99,40%+-3,88	99,34%+-3,16	99,18%+-4,00	97,23%+-3,25	96,00%+-3,62	95,25%+-5,58

Le tableau suivant présente les scores pour la phase de test :

TABLE 3.2 – Score de test

PGN	GATv2	MPNN	GAT	Deep Sets	Memory Network
87,38%	83,02%	73,08%	81,51%	81,17%	81,28%

Pour voir l'évolution à chaque pas, nous pouvons observer la figure suivante :

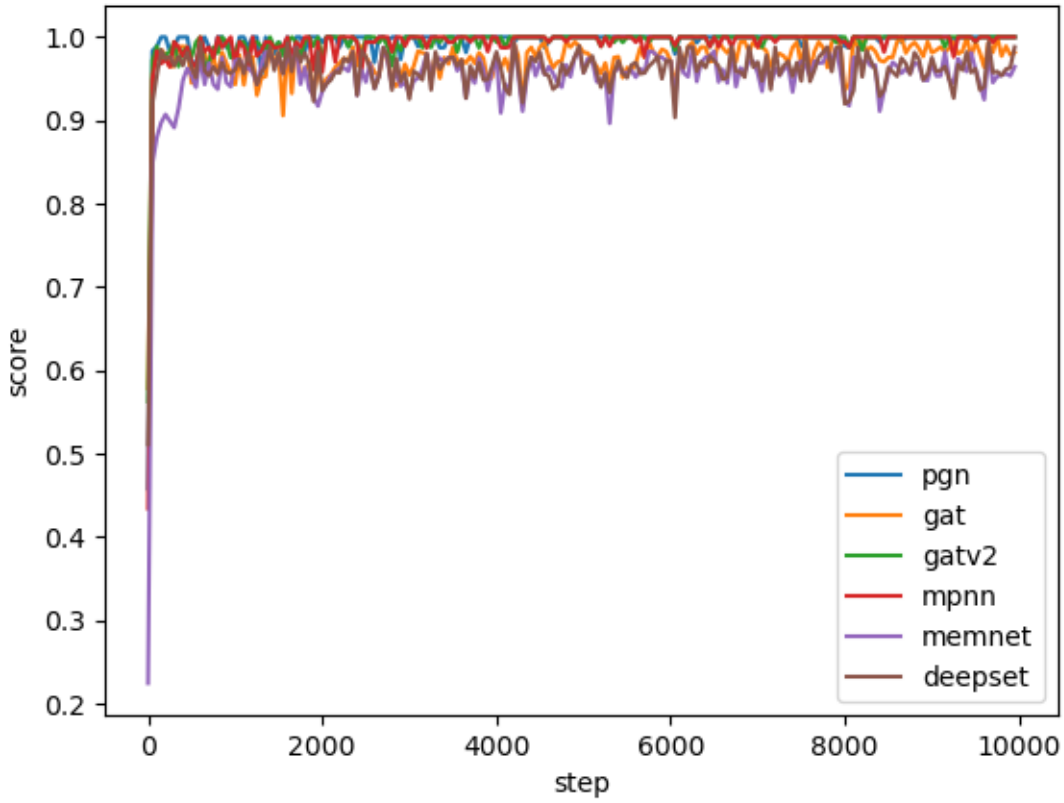

FIGURE 3.1 – Score de validation par pas (step)

Pour des descriptions plus détaillée, voici un tableau de synthèse statistique pour les scores de validation :

TABLE 3.3 – Synthèse statistique des scores de validation

	PGN	GATv2	MPNN	GAT	Deep Sets	Memory Network
Moyenne	0,994	0,993	0,9918	0,972	0,960	0,9525
Écart-type	0,038	0,031	0,040	0,032	0,036	0,055
Minimum	0,458	0.562	0,434	0,578	0,511	0,224
25%	1	0,993	0,993	0,965	0,953	0,945
50%	1	1	1	0,976	0,962	0,959
75%	1	1	1	0,987	0,974	0,970
Maximum	1	1	1	1	0,994	0,993

3.3 Analyse et interprétation

En général, le PGN a eu les meilleurs scores, suivi du GATv2, du MPNN, du GAT, du Deep Sets, et enfin du Memory Network.

Le PGN a été conçu spécialement dans le but d'apprendre le raisonnement algorithmique neuronal [50]. Ses capacités de généralisation font sa force puisqu'il a été conçu pour imiter des algorithmes, expliquant ses performances.

Les résultats du GATv2 peuvent s'expliquer par le fait que, par rapport au GAT, ses scores d'attention sont dynamiques. Elles tiennent compte des changements que peuvent subir les sommets, et changent en conséquence les scores d'attention reflétant les changements d'importance des sommets de l'algorithme de Goldberg-Tarjan, variante de l'étiquette la plus élevée, qui traite en priorité des sommets débordants avec l'étiquette la plus élevée.

Les scores du MPNN peuvent venir du fait que le paradigme de passage de messages s'aligne assez bien sur le problème du flot maximum, coupe minimum. Le passage de messages rassemblent les états des voisins d'un sommets, l'état du sommet, mais prend également en compte les arcs. Ce qui ressemble à ce que l'algorithme fait en prenant en compte l'excédent d'un sommet, sa hauteur, les capacités résiduelles des arcs menant aux voisins d'un sommet.

Comme dit précédement le GAT calcule des scores d'attention sur les sommets, et bien qu'elles soient statiques, elles suffisent à montrer que dans un voisinage, les sommets n'ont pas la même importance.

Pour les ensembles profonds ou Deep Sets, comme ils ont été développé pour travailler avec des ensembles, des objets qui sont invariants par permutation [56], ils sont adaptés pour travailler sur des graphes qui sont collections de sommets et d'arcs.

Pour les Memory Networks [43], leurs architectures se rapprochent plus des réseaux de neurones récurrents, et leurs principes sont de tenir en mémoire les représentations tant que cela ne dépasse pas la taille du tampon, afin de les évaluer face aux sorties. Cela pourrait expliquer qu'ils aient les scores les plus faibles, les structures dans les données sont certes tenues en mémoire, mais elles ne sont pas aussi bien exploitées que dans les autres architectures.

3.4 Réponses aux questions de recherche

En général, il a été possible d'imiter l'algorithme de Goldberg-Tarjan, variante de l'étiquete la plus élevée, sur des réseaux de transport de **taille quatre, huit et seize**. Les architectures qui s'en sont le mieux sorties sont le **PGN, le GATv2 et le MPNN**.

Conclusion

Notre travail a consisté à imiter le comportement de l'algorithme de Goldberg-Tarjan, variante de l'étiquete la plus élevée, à l'aide de six architectures GNNs à simple processeur. Nous avons employé six processeurs :

— Deep Sets [56]
— Graph Attention Networks [51]
— Graph Attention Networksv2 [6]
— Message-passing Networks [20]
— Pointer Graph Networks [50]
— Memory Networks [43]

Nous avons pu contribuer au CLRS-30 [47] en ajoutant un algorithme de plus à la liste des 30 algorithmes disponibles, et avons également créé des données supplémentaire au sujet des réseaux de transport et du problème de flot maximum coupe minimum. Nos résultats ont montré que le PGN, le GATv2, et le MPNN s'en sortent le mieux pour ce type de problème, mais il ne faut pas oublier que la taille des réseaux a été considérablement réduite par rapport aux études dans le CLRS-30, 4 au lieu de 16 pour l'entraînement, 8 au lieu de 32 pour la validation, et 16 au lieu de 64, pour les tests. Le but des études sur le raisonnement algorithmique neuronale est d'apprendre de nouveaux algorithmes, il serait intéressant de voir comment ces architectures s'en sortent sur des problèmes connexes, comme le problème du coût minimal dans un réseau de transport, ou le problème de circulation. Tester ces architectures sur des réseaux plus grands et ainsi tester au maximum les capacités de généralisation de ces architectures est une perspective future possible. Avec le matériel adéquat, il est tout à fait possible de le faire. On pourrait également voir si l'apprentissage par transfert peut être utilisé pour appliquer ce qui a été appris avec le problème du flot maximum coupe minimum sur des problèmes lui ressemblant. Comprendre pourquoi certaines architectures s'en sortent bien et d'autres moins bien sur le plan théorique et non plus empirique permettraient de répondre à des questions telles que : quelle classe de problème telle architecture pourrait apprendre à résoudre efficacement ? Ce sont tous des sujets probables de recherches futures, et le domaine du raisonnement algorithmique neuronale est encore émergent. L'importance du problème de généralisation des réseaux de neurones sur des données hors de la distribution du jeu d'entraînement peut être intensivement étudiée à l'aide du CLRS-30 et des réseaux de neurones sur les graphes, et les autres domaines des réseaux de neurones en bénéficieraient aussi.

Bibliographie

[1] E. ALPAYDIN, *Introduction to Machine Learning, fourth edition* (Adaptive Computation and Machine Learning series). MIT Press, 2020, ISBN : 9780262043793.

[2] A.-L. BARABÁSI et R. ALBERT, "Emergence of Scaling in Random Networks," *Science*, t. 286, n° 5439, p. 509-512, 1999. DOI : 10.1126/science.286.5439.509. eprint : https://www.science.org/doi/pdf/10.1126/science.286.5439.509.

[3] C. BISHOP, *Pattern Recognition and Machine Learning* (Information Science and Statistics). Springer New York, 2016, ISBN : 9781493938438.

[4] B. BOLLOBÁS, *Random Graphs* (Cambridge Studies in Advanced Mathematics). Cambridge University Press, 2001, ISBN : 9780521797221.

[5] J. BRADBURY, R. FROSTIG, P. HAWKINS et al., *JAX : composable transformations of Python+NumPy programs*, version 0.3.13, 2018.

[6] S. BRODY, U. ALON et E. YAHAV, "How Attentive are Graph Attention Networks ?" In *International Conference on Learning Representations*, 2022.

[7] M. M. BRONSTEIN, J. BRUNA, T. COHEN et P. VELIČKOVIĆ, *Geometric Deep Learning : Grids, Groups, Graphs, Geodesics, and Gauges*, 2021. arXiv : 2104.13478 [cs.LG].

[8] J. CHERIYAN et S. N. MAHESHWARI, "Analysis of Preflow Push Algorithms for Maximum Network Flow," *SIAM Journal on Computing*, t. 18, n° 6, p. 1057-1086, 1989. DOI : 10.1137/0218072. eprint : https://doi.org/10.1137/0218072.

[9] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST et C. STEIN, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009, ISBN : 978-0-262-03384-8.

[10] *CS6820 : Algorithms,The Dinitz, Hopcroft-Karp, and Push-Relabel Algorithms*, URL : https://www.cs.cornell.edu/courses/cs6820/2020fa/handouts/dinitz.pdf. Last visited on 2023/05/03, 2020.

[11] Y. DINITZ, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation," *Soviet Math. Dokl.*, t. 11, p. 1277-1280, jan. 1970.

[12] Y. DINITZ, "Dinitz' Algorithm : The Original Version and Even's Version," in *Theoretical Computer Science : Essays in Memory of Shimon Even*. Berlin, Heidelberg : Springer-Verlag, 2006, p. 218-240, ISBN : 3540328807.

[13] A. DOSOVITSKIY, L. BEYER, A. KOLESNIKOV et al., *An Image is Worth 16x16 Words : Transformers for Image Recognition at Scale*, 2021. arXiv : 2010.11929 [cs.CV].

[14] J. EDMONDS et R. M. KARP, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *J. ACM*, t. 19, nº 2, p. 248-264, avr. 1972, ISSN : 0004-5411. DOI : 10.1145/321694.321699.

[15] P. L. ERDOS et A. RÉNYI, "On the evolution of random graphs," *Transactions of the American Mathematical Society*, t. 286, p. 257-257, 1984.

[16] A. FAWZI, M. BALOG, A. HUANG et al., "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, t. 610, nº 7930, p. 47-53, oct. 2022, ISSN : 1476-4687. DOI : 10.1038/s41586-022-05172-4.

[17] L. FORD et D. FULKERSON, *Flows in Networks* (Princeton Landmarks in Mathematics and Physics). Princeton University Press, 2015, ISBN : 9781400875184.

[18] "Front Matter," in *An Introduction to Metric Spaces and Fixed Point Theory*. John Wiley Sons, Ltd, 2001, p. i-x, ISBN : 9781118033074. DOI : https://doi.org/10.1002/9781118033074.fmatter. eprint : https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118033074.fmatter.

[19] M. GASSE, D. CHETELAT, N. FERRONI, L. CHARLIN et A. LODI, "Exact Combinatorial Optimization with Graph Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, H. WALLACH, H. LAROCHELLE, A. BEYGELZIMER, F. D'ALCHÉ-BUC, E. FOX et R. GARNETT, éd., t. 32, Curran Associates, Inc., 2019.

[20] J. GILMER, S. S. SCHOENHOLZ, P. F. RILEY, O. VINYALS et G. E. DAHL, *Neural Message Passing for Quantum Chemistry*, 2017. DOI : 10.48550/ARXIV.1704.01212.

[21] A. V. GOLDBERG et R. E. TARJAN, "A New Approach to the Maximum-Flow Problem," *J. ACM*, t. 35, nº 4, p. 921-940, oct. 1988, ISSN : 0004-5411. DOI : 10.1145/48014.61051.

[22] I. GOODFELLOW, Y. BENGIO et A. COURVILLE, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[23] I. J. GOODFELLOW, J. POUGET-ABADIE, M. MIRZA et al., *Generative Adversarial Networks*, 2014. arXiv : 1406.2661 [stat.ML].

[24] A. GRAVES, *Generating Sequences With Recurrent Neural Networks*, 2014. arXiv : 1308.0850 [cs.NE].

[25] W. L. HAMILTON, "Graph Representation Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, t. 14, nº 3, p. 1-159,

[26] W. L. HAMILTON, R. YING et J. LESKOVEC, *Inductive Representation Learning on Large Graphs*, 2018. arXiv : 1706.02216 [cs.SI].

[27] T. E. HARRIS et F. S. ROSS, "Fundamentals of a Method for Evaluating Rail Net Capacities," Santa Monica, California, rapp. tech., 1955.

[28] T. HASTIE, R. TIBSHIRANI et J. FRIEDMAN, *The Elements of Statistical Learning : Data Mining, Inference, and Prediction, Second Edition* (Springer Series in Statistics). Springer New York, 2009, ISBN : 9780387848587.

[29] T. HENNIGAN, T. CAI, T. NORMAN et I. BABUSCHKIN, *Haiku : Sonnet for JAX*, version 0.0.9, 2020.

[30] B. IBARZ, V. KURIN, G. PAPAMAKARIOS et al., *A Generalist Neural Algorithmic Learner*, 2022. arXiv : 2209.11142 [cs.LG].

[31] M. I. JORDAN et T. M. MITCHELL, "Machine learning : Trends, perspectives, and prospects," en, *Science*, t. 349, nº 6245, p. 255-260, juill. 2015.

[32] D. JUNGNICKEL, *Graphs, Networks and Algorithms (Algorithms and Computation in Mathematics)*. Berlin, Heidelberg : Springer-Verlag, 2004, ISBN : 3540219056.

[33] N. KARALIAS et A. LOUKAS, “Erdos Goes Neural : an Unsupervised Learning Framework for Combinatorial Optimization on Graphs,” in *Advances in Neural Information Processing Systems*, H. LAROCHELLE, M. RANZATO, R. HADSELL, M. BALCAN et H. LIN, éd., t. 33, Curran Associates, Inc., 2020, p. 6659-6672.

[34] E. KHALIL, H. DAI, Y. ZHANG, B. DILKINA et L. SONG, “Learning Combinatorial Optimization Algorithms over Graphs,” in *Advances in Neural Information Processing Systems*, I. GUYON, U. V. LUXBURG, S. BENGIO et al., éd., t. 30, Curran Associates, Inc., 2017.

[35] T. N. KIPF et M. WELLING, *Semi-Supervised Classification with Graph Convolutional Networks*, 2017. arXiv : 1609.02907 [cs.LG].

[36] V. MNIH, K. KAVUKCUOGLU, D. SILVER et al., *Playing Atari with Deep Reinforcement Learning*, 2013. arXiv : 1312.5602 [cs.LG].

[37] M. NEWMAN, “Power laws, Pareto distributions and Zipf’s law,” *Contemporary Physics*, t. 46, nº 5, p. 323-351, sept. 2005. DOI : 10.1080/00107510500052444.

[38] S. J. PAN et Q. YANG, “A Survey on Transfer Learning,” *IEEE Transactions on Knowledge and Data Engineering*, t. 22, p. 1345-1359, 2010.

[39] F. SCARSELLI, M. GORI, A. C. TSOI, M. HAGENBUCHNER et G. MONFARDINI, “The Graph Neural Network Model,” *IEEE Transactions on Neural Networks*, t. 20, nº 1, p. 61-80, 2009. DOI : 10.1109/TNN.2008.2005605.

[40] D. SILVER, T. HUBERT, J. SCHRITTWIESER et al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, 2017. arXiv : 1712.01815 [cs.AI].

[41] D. D. K. SLEATOR, “An o(Nm Log n) Algorithm for Maximum Network Flow,” AAI8108998, thèse de doct., Stanford, CA, USA, 1981.

[42] G. STRANG, *Linear Algebra and Learning from Data*. Wellesley-Cambridge Press, 2019, ISBN : 9780692196380.

[43] S. SUKHBAATAR, A. SZLAM, J. WESTON et R. FERGUS, *End-To-End Memory Networks*, 2015. DOI : 10.48550/ARXIV.1503.08895.

[44] R. SUTTON et A. BARTO, *Reinforcement Learning, second edition : An Introduction* (Adaptive Computation and Machine Learning series). MIT Press, 2018, ISBN : 9780262352703.

[45] V. VAPNIK, *The Nature of Statistical Learning Theory* (Information Science and Statistics). Springer New York, 2013, ISBN : 9781475732641.

[46] A. VASWANI, N. SHAZEER, N. PARMAR et al., *Attention Is All You Need*, 2017. arXiv : 1706.03762 [cs.CL].

[47] P. VELIČKOVIĆ, A. P. BADIA, D. BUDDEN et al., *The CLRS Algorithmic Reasoning Benchmark*, 2022. DOI : 10.48550/ARXIV.2205.15659.

[48] P. VELIČKOVIĆ et C. BLUNDELL, "Neural algorithmic reasoning," *Patterns*, t. 2, nº 7, p. 100 273, 2021, ISSN : 2666-3899. DOI : https://doi.org/10.1016/j.patter.2021.100273.

[49] P. VELIČKOVIĆ et C. BLUNDELL, "Neural algorithmic reasoning," *Patterns*, t. 2, nº 7, p. 100 273, juill. 2021. DOI : 10.1016/j.patter.2021.100273.

[50] P. VELIČKOVIĆ, L. BUESING, M. C. OVERLAN, R. PASCANU, O. VINYALS et C. BLUNDELL, *Pointer Graph Networks*, 2020. DOI : 10.48550/ARXIV.2006.06380.

[51] P. VELIČKOVIĆ, G. CUCURULL, A. CASANOVA, A. ROMERO, P. LIÒ et Y. BENGIO, *Graph Attention Networks*, 2017. DOI : 10.48550/ARXIV.1710.10903.

[52] J. WU et D. J. WATTS, "Small Worlds : The Dynamics of Networks between Order and Randomness," *SIGMOD Rec.*, t. 31, nº 4, p. 74-75, déc. 2002, ISSN : 0163-5808. DOI : 10.1145/637411.637426.

[53] L.-P. XHONNEUX, A.-I. DEAC, P. VELIČKOVIĆ et J. TANG, "How to transfer algorithmic reasoning knowledge to learn new algorithms ?" In *Advances in Neural Information Processing Systems*, M. RANZATO, A. BEYGELZIMER, Y. DAUPHIN, P. LIANG et J. W. VAUGHAN, éd., t. 34, Curran Associates, Inc., 2021, p. 19 500-19 512.

[54] K. XU, J. LI, M. ZHANG, S. S. DU, K.-i. KAWARABAYASHI et S. JEGELKA, *What Can Neural Networks Reason About ?* 2019. DOI : 10.48550/ARXIV.1905.13211.

[55] K. XU, J. LI, M. ZHANG, S. S. DU, K.-i. KAWARABAYASHI et S. JEGELKA, *What Can Neural Networks Reason About ?* 2020. arXiv : 1905.13211 [cs.LG].

[56] M. ZAHEER, S. KOTTUR, S. RAVANBAKHSH, B. POCZOS, R. SALAKHUTDINOV et A. SMOLA, *Deep Sets*, 2017. DOI : 10.48550/ARXIV.1703.06114.

Annexe A

Goldberg-Tarjan, variante de l'étiquette la plus élevée

A.1 Pseudo-code pour obtenir la coupe

Algorithme 11 : ÉTIQUETTE-LA-PLUS-ÉLEVÉE

Données : $N = (G, c, s, t)$, avec G = (S,A) un graphe orienté symétrique
I, listes d'incidence pour chaque sommet de G.
Résultat : coupe minimum S,T

`/* Initialisation du préflot */`
pour $\forall (u, v) \in (S - \{s\}) \times (S - \{s\})$ **faire**
    $f(u, v) \leftarrow 0$;
    $f(v, u) \leftarrow 0$;
pour $\forall u \in S$ **faire**
    $f(s, u) \leftarrow c(s, u)$;
    $f(u, s) \leftarrow -c(s, u)$;
`/* Initialisation de la fonction de hauteur et des excédents */`
$h(s) \leftarrow |S|$;
$Q \leftarrow \emptyset$;
pour $\forall u \in S - \{s\}$ **faire**
    $h(u) \leftarrow 0$;
    $e(u) \leftarrow f(s, u)$;
    `/* Ajout des modifications pour DÉCHARGER */`
    faire du premier arc dans $I(u)$, l'arc actuel à considérer pour u;
    si $e(u) > 0$ *et* $u \neq t$ **alors**
        insérer u dans Q avec une priorité h(u);
`/* Boucle principale */`
tant que $Q \neq \emptyset$ **faire**
    $u \leftarrow$ premier sommet de Q ; `/* sommet avec la plus grande priorité */`
    DÉCHARGER(u);
$S \leftarrow \emptyset$;
$T \leftarrow \emptyset$;
$hauteur\_cible \leftarrow -1$;
pour *hauteur de 1 à n-1* **faire**
    si *hauteur* $\notin h$ **alors**
        hauteur_cible $\leftarrow$ hauteur;
        sortie de boucle ;
si *hauteur_cible* $\neq$ *-1* **alors**
    pour $v \in S : h(v) >=$ *hauteur_cible* **faire**
        mettre v dans S;
    pour $v \in S : h(v) >$ *hauteur_cible* **faire**
        mettre v dans T;

A.1.1 Implémentation en Python de cet algorithme

```python
import numpy as np

def adjacencyToIncidenceList(A):
    """
    takes an adjacency matrix
    returns a dict representing the incidence list of each vertex
    """
    Incidence_list = dict()
    tp_list = []
    for v in range(A.shape[0]):
        Incidence_list[v] = []

    for v in range(A.shape[0]):
        w_s = np.argwhere(A[v,:]>0)
        for w in w_s:
            if (v,w[0]) not in Incidence_list[v]:
                Incidence_list[v].append((v,w[0]))
            if (w[0],v) not in Incidence_list[w[0]]:
                Incidence_list[w[0]].append((w[0],v))
        #Incidence_list[v] = [tp_list,-1]# -1 for current edge data structure
        #tp_list = []
    return Incidence_list
```

Listing A.1 – code pour obtenir la liste d'incidence à partir de la matrice d'adjacence

```python
def push(v, w, f, r_f, e):
    """
    v,w: edge through which we will push some flow
    f: preflow (np 2D array)
    r_f: residual capacity (np 2D array)
    e: excess vector (np array)
    returns f,r_f,e
    """
    f = f.copy()
    r_f = r_f.copy()
    e = e.copy()
```

```
    delta = min(e[v], r_f[v,w])
    f[v,w] += delta
    f[w,v] -= delta
    r_f[v,w] -= delta
    r_f[w,v] += delta
    e[v] -= delta
    e[w] += delta
    return f,r_f,e
```

Listing A.2 – code de l'opération POUSSER

```
def relabel(v, r_f, d):
    """
    relabelling operation
    v: vertex
    r_f: residual capacity (np 2D array)
    d: label
    returns d
    """
    w_s = np.argwhere(r_f[v,:]> 1e-6)
    d = d#.copy()
    if len(w_s) > 0:
        mini = np.min(d[w_s])
        #print(mini)
        d[v] = mini + 1
    return d
```

Listing A.3 – code de l'opération RÉÉTIQUETER

```
def procedure(A, C, Q, v, s, t, f, r_f, e, d, current_edge, rel):
    """
    A: dict of the incidence lists of each vertex, with G being a symmetric
    digraph
    C: capacity matrix
    Q: priority queue
    v: removed vertex with highest label priority in Q
    s: source node
    t: sink node
    f: preflow
```

```
    r_f: residual capacity
    e: excess vector
    d: label vector
    rel: a boolean variable associated with the highest label push algorithm

    returns f, r_f, e, d, rel
    """
    A = A.copy()
    Q = Q.copy()
    f = f.copy()
    r_f = r_f.copy()
    e = e.copy()
    d = d.copy()
    current_edge = current_edge.copy()
    rel = rel
    # Current edge in A_v
    v,w = A[v][current_edge[v]]
    #print(v,w)

    inside_queue = False
    # Adding for underflow
    if(C[v,w] - f[v,w] > 1e-6 and (d[v]==d[w]+1)):

        #print("e before", e)
        f, r_f, e = push(v, w, f, r_f, e)
        #print("e after", e)

        for priority,vertex in Q:
            if(w==vertex):
                inside_queue = True
        if((not inside_queue) and w != s and w != t):

            Q.append((-d[w],w))
            Q.sort(key=lambda vertex: vertex[0])#sort by priority

    elif e[v] > 1e-6:

```

```
47          if current_edge[v] < len(A[v])-1:
48
49              current_edge[v] = current_edge[v] + 1
50
51          else:
52
53              d = relabel(v, r_f, d)
54              #print(d)
55              rel = True
56              # first edge as current edge
57              current_edge[v] = 0
58
59      return A,Q,f,r_f,e,d,current_edge,rel
```

Listing A.4 – code de l'opération DÉCHARGER

```
1  def HLFlow(A, C, s, t):
2      """
3      Highest Label flow push algorithm, Goldberg and Tarjan 1988
4      A: Incidence lists
5      C: capacity matrix
6      s: source node
7      t: sink node
8      return S,T cut
9      """
10     # Initialization of all variables
11     #preflow
12     n,m = C.shape
13     f = np.full((n,m), 0.)
14     for v in range(n):
15         f[s,v] = C[s,v]
16         f[v,s] = -C[s,v]
17     # labels and excess
18     d = np.array([0 for i in range(n)])
19     e = np.array([0.0 for i in range(n)])
20     d[s] = n
21     for v in range(n):
22         if v!= s:
```

```
            e[v] = f[s,v]

    # residual capacity
    r_f = C - f
    for v in range(n):
        if v != s:
            r_f[s,v] = 0.
            r_f[v,s] = C[v,s] + C[s,v]
    # priority queue and current edge for each A_v
    Q = []
    current_edge = np.array([0 for i in range(n)])
    for v in range(n):
        if v!= s:
            # make the first edge the current edge
            current_edge[v] = 0
            # pushing in Q
            if e[v] > 0 and v != t:
                Q.append((-d[v],v))
    Q.sort(key=lambda vertex: vertex[0])# sort by priority
    while len(Q) > 0:
        # remove vertex of highest priority
        priority,v = Q.pop(0)
        rel = False
        A,Q,f,r_f,e,d,current_edge,rel = procedure(A, C, Q, v, s, t, f, r_f, e, d,
 current_edge,rel)
        while(e[v] > 0. and rel==False):
            A,Q,f,r_f,e,d,current_edge,rel = procedure(A, C, Q, v, s, t, f, r_f, e
, d, current_edge, rel)
        #if e[v] > 0 and d[v] < n and -d[v] < priority:
        if e[v] > 0 :
            Q.append((-d[v],v))
            Q.sort(key=lambda vertex: vertex[0])# sort by priority

    S = []
    T = []
    target_height = -1
    for height in range(1,n):
```

```
58          if height not in d:
59              target_height = height
60              #break
61      if target_height != -1:
62          for vertex in np.argwhere(d > target_height):
63              S.append(vertex[0])
64
65          for vertex in np.argwhere(d < target_height):
66              T.append(vertex[0])
67      else:
68          S = []
69          T = []
70      return S,T
```

Listing A.5 – code de la variante de l'étiquette la plus élevée

Annexe B

Réseaux de neurones

B.1 Généralités

Il y a plusieurs manières de voir un réseau de neurones :
— il s'agit d'un réseau fait de plusieurs couches de "neurones" ;
— c'est un graphe de calcul où transite des matrices multidimensionnelles (des tenseurs) ;
— un approximateur universel qui peut théoriquement apprendre toute fonction dans les tâches supervisées.

Ces différents points de vues permettent d'appréhender les différentes manières de faire dans le développement des réseaux de neurones. Dans cette partie, nous allons voir plus en détail le réseau MLP, les méthodes d'entraînement des réseaux de neurones, et la rétropropagation.

B.2 Réseau de perceptrons multicouches

Considérons une entrée x parmi les données X dont nous avons à disposition. Un perceptron est une fonction linéaire composée à une fonction non-linéaire qui prend en entrée x et produit une sortie y, de la manière suivante :

$$y = \sigma(Wx) \tag{B.1}$$

La matrice W est la matrice des poids, elle est apprise à l'aide des données, quant à σ, il s'agit d'une fonction linéaire. Généralement, il s'agit d'une fonction sigmoïde $\left(f(z) = \frac{1}{1+\exp(-z)}\right)$, de la fonction ReLu(x) = max (0,x), ou de la fonction tanh. Notons $L_i(x) = W_i x$ Un réseau de perceptrons multicouches est représenté de la manière suivante :

$$y = \sigma(...\sigma(L_2(\sigma(L_1(x))))) = F(x; w) \tag{B.2}$$

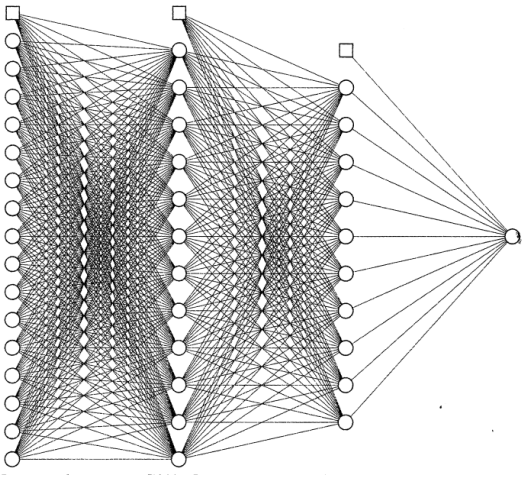

FIGURE B.1 – Réseau de neurones à plusieurs couches [42]

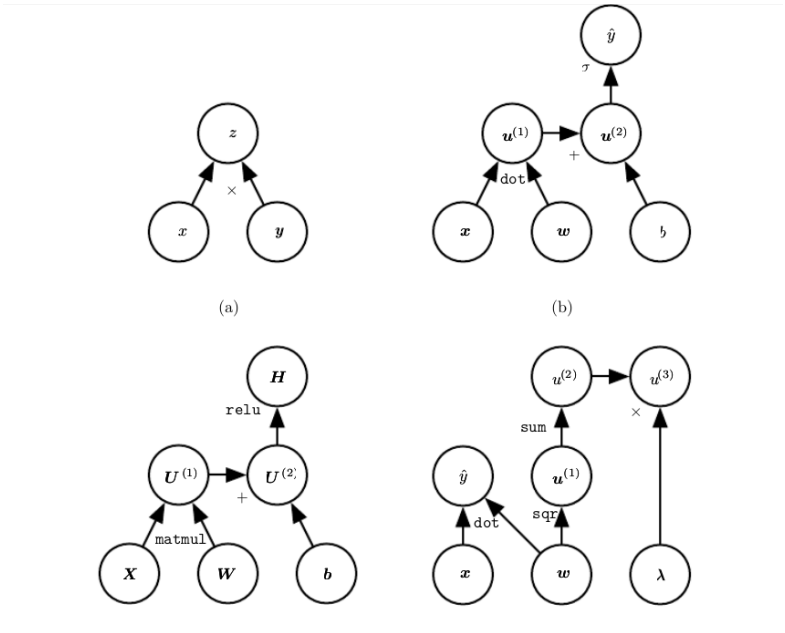

FIGURE B.2 – Graphe de calcul[22]

Il s'agit d'une composition multiple du perceptrons, chaque composition est une "couche" du réseau. La première couche est appelée couche d'entrée et la dernière couche de sortie. Les couches entre la première et la dernière sont appelées les couches cachées. L'ensemble de données spécifie pour chaque exemple une étiquette uniquement pour la couche de sortie. L'ensemble de données ne montre pas les valeurs que doivent prendre les couches cachées d'où leur nom. Quand on parle d'architecture d'un réseau de neurones, on parle alors du nombre de couches que nous allons utiliser, de la manière dont chaque perceptron communique avec les autres perceptrons, du choix de la fonction non-linéaire également appelée fonction d'activation. Le but de l'apprentissage est donc d'apprendre les paramètres w dans F(x ;w).

B.3 Descente de gradient

Un réseau de neurone est entrainé en utilisant la descente de gradient [22]. L'entrainement ressemble à un problème d'optimisation, à une exception près. Nous voulons minimiser une fonction de perte en espérant que la minimiser améliore également une certaine mesure de performance. Pour une fonction $f : \mathbb{R}^n \to \mathbb{R}$, la descente de gradient consiste à réduire $f(x)$ en variant x d'un petit pas dans la direction opposée du gradient de f. Ce pas noté ϵ est appelé taux d'apprentissage et est un hyperparamètre. La mise à jour s'effectue donc comme suit :

$$x' = x - \epsilon \nabla_x f(x) \tag{B.3}$$

Le problème avec la descente de gradient est que la mise à jour ne peut avoir lieu qu'après avoir vu tous les exemples de l'ensemble de données. Pour m observations, une traversée coute donc $O(m)$, généralement, la taille des données est énorme, et cela peut donc être très coûteux de calculer le gradient à partir de toutes les observations. Une variation de la descente de gradient, la descente de gradient stochastique, consiste à considérer uniquement un lot d'observations pour une mise à jour. Réduisant ainsi la complexité tout en assurant une bonne performance.

La fonction de perte d'un réseau de neurones est obtenu par la méthode de maximum de vraisemblance. C'est l'entropie croisée entre les données d'entrainement dont la distribution est notée $\hat{p}_{donnes}$ et la distribution de probabilité du modèle qu'on note p_{modele}. La fonction de perte prenant en entrée les paramètre θ est définie comme suit :

$$J(\theta) = -\mathbb{E}_{x,y\sim\hat{p}_{donnes}} \big[\log\ p_{modele}(y|x)\big] \tag{B.4}$$

En fonction de la distribution du modèle, on obtient une fonction de perte différente.

B.4 Rétropopagation

Une entrée x donne les informations initiales du réseau qui sont ensuite propagées à travers les couches cachées jusqu'à produire le scalaire $J(\theta)$. Ce procédé est appelé propagation. J représente la perte pour tous les exemples du lot tandis que L représente la perte pour un seul exemple.

Algorithme 12 : Propagation directe à travers un réseau pour un seul exemple x afin de calculer la valeur de la fonction de coût. Ω est utilisé pour la régularisation et θ représente l'ensemble d tous les paramètres.

Entrées : l, la profondeur du réseau ;

$W^{(i)}, i \in \{1, ..., l\}$, la matrice des poids du modèle ;

$b^{(i)}, i \in \{1, ..., l\}$, les biais du modèle ;

x, l'entrée du réseau ;

y, l'étiquette associée à x.

Résultat : $L(\hat{y}, y)$, le coût ou la perte

$h^{(0)} = x$

pour $k = 1$ **à** l **faire**

  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

  $h^{(k)} = f(a^{(k)})$

$\hat{y} = h^{(l)}$

retourner $L(\hat{y}, y) + \lambda \Omega(\theta)$

La rétropopagation [22] permet aux informations de passer dans la direction opposée pour calculer les gradients. Le calcul des gradients repose sur l'application récursive de la règle de la chaîne utilisée pour la dérivation de composition de fonctions. Ces gradients sont ensuite utilisés avec le taux d'apprentissage pour mettre à jour les paramètres du réseau.

Algorithme 13 : Rétropropagation d'un réseau de neurone

Entrées : $L(\hat{y}, y)$, le coût obtenu par propagation directe

$g = \nabla_{\hat{y}} L(\hat{y}, y)$

$h^{(0)} = x$

pour $k = l$ **à** 1 **faire**

  $g = g \odot f'(a^{(k)})$

  $\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

  $\nabla_{W^{(k)}} J = g \; h^{(k-1)\top} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

  $g = W^{(h)\top} g$

Ici, $\odot$ désigne la multiplication membre à membre.

Annexe C

CLRS-30

On peut trouver le dépôt git CLRS-30 à l'adresse suivante : `https://github.com/deepmind/clrs`. Nous avons choisi de contribuer au CLRS-30 en ajoutant un algorithme au cadre de travail. Pour cela, nous devons manipuler trois fichiers : clrs/_src/specs.py, clrs/_src/algorithms/-graphs.py, et clrs/_src/samplers.py.

C.0.1 Spécifications pour l'algorithme de l'étiquette la plus élevée

En anglais, il s'agit du Highest Label preFlow algorithm, HLFlow. Comme décrit dans les chapitres antérieurs, il faut préciser les spécifications de notre algorithme, décrire convenablement ses variables.

On ajoute une entrée au dictionnaire SPECS présent dans le fichier clrs/_src/specs.py :

```
'hlflow_mincut':{
        'pos': (Stage.INPUT, Location.NODE, Type.SCALAR),
        'C': (Stage.INPUT, Location.EDGE, Type.SCALAR),
        'adj': (Stage.INPUT, Location.EDGE, Type.MASK),
        's': (Stage.INPUT, Location.NODE, Type.MASK_ONE),
        't': (Stage.INPUT, Location.NODE, Type.MASK_ONE),
        'S': (Stage.OUTPUT, Location.NODE, Type.MASK),
        'Q': (Stage.HINT, Location.NODE, Type.MASK),
        'active_edge': (Stage.HINT, Location.NODE, Type.MASK),
        'f': (Stage.HINT, Location.EDGE, Type.SCALAR),
        'r_f': (Stage.HINT, Location.EDGE, Type.SCALAR),
        'e': (Stage.HINT, Location.NODE, Type.SCALAR),
        'd': (Stage.HINT, Location.NODE, Type.SCALAR),
```

```
    }
```

Listing C.1 – spécification de la variante de l'étiquette la plus élevée

Les variables pos et adj sont des variables qui doivent accompagner les algorithmes sur les graphes dans le CLRS-30, pos encode la position des sommets, et adj est une représentation binaire de la matrice d'adjacence.

C.0.2 Implémentation de l'algorithme suivant le cadre du CLRS-30

```
def HLFlow(A, C, s, t):
    """
    Highest Label flow push algorithm, Goldberg and Tarjan 1988
    A: Incidence lists
    C: capacity matrix
    s: source node
    t: sink node
    returns S,probes
    """
    n,m = C.shape

    chex.assert_rank(C, 2)
    probes = probing.initialize(specs.SPECS['hlflow_mincut'])
    C_pos = np.arange(C.shape[0])
    adj = probing.graph(np.copy(C))
    probing.push(
      probes,
      specs.Stage.INPUT,
      next_probe={
          'pos': np.copy(C_pos) * 1.0 / C.shape[0],
          'C': np.copy(C),
          'adj': adj,
          's': probing.mask_one(s, C.shape[0]),
          't': probing.mask_one(t, C.shape[0])
      })

    # Initialization of all variables
    #preflow

```

```
    f = np.full((n,m), 0.)
    for v in range(n):
        f[s,v] = C[s,v]
        f[v,s] = -C[s,v]
    # labels and excess
    d = np.array([0 for i in range(n)])
    e = np.array([0.0 for i in range(n)])
    d[s] = n
    for v in range(n):
        if v!= s:
            e[v] = f[s,v]

    # residual capacity
    r_f = C - f
    for v in range(n):
        if v != s:
            r_f[s,v] = 0.
            r_f[v,s] = C[v,s] + C[s,v]
    # priority queue and current edge for each A_v
    Q = []
    in_Q = np.zeros(n)
    current_edge = np.array([0 for i in range(n)])
    for v in range(n):
        if v!= s:
            # make the first edge the current edge
            current_edge[v] = 0
            # pushing in Q
            if e[v] > 1e-6 and v != t:
                Q.append((-d[v],v))
                in_Q[v] = 1
    Q.sort(key=lambda vertex: vertex[0])# sort by priority
    if len(Q) ==0:

        probing.push(
            probes,
            specs.Stage.HINT,
            next_probe={
```

```
                'Q': np.copy(in_Q),
            #'v': probing.mask_one(v, C.shape[0]),
            #'w': probing.mask_one(w, C.shape[0]),
                'active_edge': np.zeros(n),
                'f': np.copy(f),
                'r_f': np.copy(r_f),
                'e': np.copy(e),
                'd': np.copy(d)
            })

    while len(Q) > 0:
        # remove vertex of highest priority
        priority,v = Q.pop(0)
        in_Q[v] = 0
        rel = False
        probes,A,Q,in_Q,f,r_f,e,d,current_edge,rel = procedure(probes,A, C, Q,
    in_Q, v, s, t, f, r_f, e, d, current_edge,rel)


        while(e[v] > 1e-6 and rel==False):
            probes,A,Q,in_Q,f,r_f,e,d,current_edge,rel = procedure(probes,A, C, Q,
     in_Q, v, s, t, f, r_f, e, d, current_edge,rel)


        if e[v] > 1e-6 :
            Q.append((-d[v],v))
            in_Q[v] = 1
            Q.sort(key=lambda vertex: vertex[0])# sort by priority



    probing.push(
            probes,
            specs.Stage.HINT,
            next_probe={
                'Q': np.copy(in_Q),
            #'v': probing.mask_one(v, C.shape[0]),
```

```
102             #'w': probing.mask_one(w, C.shape[0]),
103                 'active_edge': np.zeros(n),
104                 'f': np.copy(f),
105                 'r_f': np.copy(r_f),
106                 'e': np.copy(e),
107                 'd': np.copy(d)
108             })
109     in_S = np.zeros(n)
110     target_height = -1
111     for height in range(1,n):
112         if height not in d:
113             target_height = height
114             break
115     if target_height != -1:
116         for vertex in np.argwhere(d > target_height):
117             #S.append(vertex[0])
118             in_S[vertex[0]] = 1
119 
120 
121     probing.push(probes, specs.Stage.OUTPUT, next_probe={'S': np.copy(in_S)})
122 
123     probing.finalize(probes)
124     return in_S,probes
125 
126 def hlflow_mincut(C: _Array, s: int, t: int) -> _Out:
127     """
128     Highest Label flow push algorithm, Goldberg and Tarjan 1988
129     """
130     A = adjacencyToIncidenceList(C)
131 
132     in_S,probes = HLFlow(A, C, s, t)
133     return in_S,probes
```

Listing C.2 – hlflow_mincut

Remarquons l'ajout de probing et de probes. Ce sont les objets nécessaires pour enregistrer l'état interne des variables.

C.0.3 Implémentation de l'échantillonneur

Il convient ensuite de créer un échantillonneur pour hlflow_mincut.
On ajoute une classe dans le fichier clrs/_src/samplers.py :

```
class HLFlowSampler(Sampler):
    """Highest Label Flow sampler for flow networks"""
    def _sample_data(
      self,
      length: int,
      p: Tuple[float, ...] = (0.17,),
      low: float = 1.,
      high: float = 5.,
    ):
      graph = self._random_er_graph(
        nb_nodes=length,
        p=self._rng.choice(p),
        directed=True,
        acyclic=False,
        weighted=True,
        low=low,
        high=high)
      source_node = self._rng.choice(length)
      sink_node = self._rng.choice(length)
      while source_node == sink_node:
          sink_node = self._rng.choice(length)
      return [graph, source_node, sink_node]
```

Listing C.3 – définition de l'échantillonneur

self.random_er_graph est le générateur de graph aléatoire basé sur le modèle d'Erdös-Réyini.

```
    def _random_er_graph(self, nb_nodes, p=0.5, directed=False, acyclic=False,
                         weighted=False, low=0.0, high=1.0):
    """Random Erdos-Renyi graph."""

    mat = self._rng.binomial(1, p, size=(nb_nodes, nb_nodes))
    if not directed:
      mat *= np.transpose(mat)
    elif acyclic:
      mat = np.triu(mat, k=1)
```

```
      p = self._rng.permutation(nb_nodes)  # To allow nontrivial solutions
      mat = mat[p, :][:, p]
    if weighted:
      weights = self._rng.uniform(low=low, high=high, size=(nb_nodes, nb_nodes))
      if not directed:
        weights *= np.transpose(weights)
        weights = np.sqrt(weights + 1e-3)  # Add epsilon to protect underflow
      mat = mat.astype(float) * weights
    return mat
```

Listing C.4 – Erdös-Réyini

Puis on ajoute une entrée dans le dictionnaire SAMPLERS.

```
    'hlflow_mincut': HLFlowSampler
```

Listing C.5 – dictionnaire SAMPLERS

Pour générer les données, il faut utiliser la méthode build_sampler. Ici par exemple, on construit l'échantillonneur pour l'algorithme de parcours en largeur (bfs).

```
sampler, spec = clrs.build_sampler(
    name='bfs',
    seed=42,
    num_samples=1000,
    length=16)

def _iterate_sampler(batch_size):
  while True:
    yield sampler.next(batch_size)

for feedback in _iterate_sampler(batch_size=32):
  ...
```

Listing C.6 – échantillonneur parcours en largeur

La première ligne définit l'échantillonneur, avec le nom de l'algorithme, la graine pour le générateur de nombre pseudo-aléatoire, le nombre d'échantillons dont on a besoin, la taille du problème (en terme de nombre de sommets ou la taille d'une liste pour d'autres algorithmes, ou la taille d'une chaîne de caractères).

C.0.4 Implémentation et entraînement des GNN

On peut trouver le notebook contenant la phase d'entraînement à ce dépôt git : `https://github.com/DimbyTa/CLRS_extended.git` .

Résumé

Ce mémoire étudie l'utilisation des réseaux de neurones sur les graphes (GNN) pour résoudre les problèmes de flot maximum coupe minimum en imitant l'algorithme de préflot variante de l'étiquette la plus élevée. L'étude commence par un examen des algorithmes de flot maximum classiques : Ford-Fulkerson, Edmonds-Karp, Dinic, Pousser-réétiqueter, et la variante de l'étiquette la plus élevée. Grâce à une analyse comparative de leurs complexités, l'algorithme de l'étiquette la plus élevée est identifié comme l'algorithme cible pour l'imitation à l'aide du GNN.

S'inspirant du raisonnement algorithmique neuronal, une architecture GNN est développée, incorporant des processeurs tels que Graph Attention Networks (GAT), GATv2, Message Passing Neural Networks (MPNN), Deep Sets, et Pointer Graph Networks (PGN). Ces processeurs ont déjà été étudiés pour leur capacité à imiter les algorithmes classiques, et cette recherche étend leur application au domaine des problèmes de flot maximum coupe minimum.

Pour évaluer les performances du GNN proposé, la génération de réseaux aléatoires est utilisée pour créer des ensembles de données permettant d'évaluer la mise en œuvre de l'algorithme de l'étiquette la plus élevée et d'entraîner le modèle GNN. L'algorithme est mis en œuvre et le GNN est entraîné sur les données générées. Les résultats obtenus par chaque processeur de l'architecture GNN sont analysés et comparés.

En combinant les connaissances des algorithmes de flot maximum classiques avec la puissance des GNN, cette étude vise à combler le fossé entre la théorie des graphes, l'algorithmique et les réseaux neuronaux. Le modèle GNN développé a le potentiel de fournir des solutions précises aux problèmes de flot maximum coupe minimum, ouvrant ainsi la voie à une exploration plus poussée de l'apprentissage de la représentation des graphes et de l'imitation algorithmique.

Mots-clés : GNN, raisonnement algorithmique neuronal, flot maximum coupe minimum

Abstract

This Master's thesis investigates the use of Graph Neural Networks (GNNs) to solve maxflow mincut problems by emulating the highest label preflow algorithm. The study begins with an examination of classical maxflow algorithms : Ford-Fulkerson, Edmonds-Karp, Dinic, Push-relabel, and highest label preflow. Through a comparative analysis of their time complexities, the highest label preflow algorithm is identified as the target algorithm for GNN emulation.

Drawing inspiration from Neural Algorithmic Reasoning, a GNN architecture is developed, incorporating processors such as Graph Attention Networks (GAT), GATv2, Message Passing Neural Networks (MPNN), Deep Sets, and Pointer Graph Networks (PGN). These processors have been previously explored for their ability to mimic classical algorithms, and this research extends their application to the realm of maxflow mincut problems.

To assess the performance of the proposed GNN, random network generation is used to create datasets for evaluating both the implementation of the highest label preflow algorithm and training the GNN model. The highest label preflow algorithm is implemented, and the GNN is trained on the generated data. Results obtained by each processor in the GNN architecture are analyzed and compared.

By combining insights from classical maxflow algorithms with the power of GNNs, this study aims to bridge the gap between graph theory, algorithmic and neural networks. The developed GNN model has the potential to provide accurate solutions to maxflow mincut problems, thereby paving the way for further exploration in graph representation learning and algorithmic imitation.

Keywords : GNN, neural algorithmic reasoning, maxflow mincut