Compte Rendu de Mi-Projet de Recherche

Pierre Baumann, Ivan Mulot-Radojcic, Damien Assire

18 mars 2024

Résumé

Dans ce compte-rendu, nous allons présenter l'état actuel de notre projet de recherche sur le thème "L'efficacité des flots de liens engendrés par des marcheurs sur un réseau viaire".

Nous expliquerons dans un premier temps notre interprétation du sujet. Ensuite, nous allons documenter le travail que nous avons effectué, notemment les structures de données et algorithmes que nous avons implémentés pour représenter informatiquement les réseaux viaires et la simulation des attaques. Puis, nous allons aborder le travail restant à effectuer, et les perspectives que nous envisageons pour la suite du projet.

Table des matières

1 Introduction **2**

2 Génération du graphe **2**
2.1 Transformation du graphe en Matrice creuse 3

3 Génération des attaques **3**
3.1 Attaques dynamiques 4
3.1.1 Attaque aléatoire 4
3.1.2 Attaque mouvante 4
3.2 Attaques statiques 4
3.2.1 Attaque par centralité intermédiaire 5

4 Simulation **5**
4.1 Structure Générale du programme C 5
4.2 Structures de données internes du programme C 5
4.3 L'algorithme de "Floyd Warhsall Temporel" 6
4.4 Calcul de la mesure de robustesse 6
4.5 Pseudocode de l'algorithme de Floyd Warshall Temporel et de l'initialisation des matrices de distances : 6

5 Travail restant **7**
5.1 Lancement des simulations 7
5.2 Analyse des résultats 7

6 Perspectives **7**
6.1 Optimisation du programme 7
6.1.1 Parallélisation des calculs 7
6.1.2 Moins de calculs 8
6.2 Nouvelles attaques 9
6.3 Modification de la mesure 9

7 Conclusion **9**

1 Introduction

Les réseaux sont des concepts primordiaux dans le monde moderne, le réseau routier, les réseaux sociaux, les réseaux de neurones, internet, ect... Ces réseaux peuvent être modélisés par des graphes orientés pondérés $G = (V, E)$ où V est l'ensemble des sommets, E est l'ensemble des liens $e = (u, v, w)$ $u, v \in V, w \in \mathbb{N}$ qui est la pondération du lien de u à v, noté u $\xrightarrow{\text{w}}$ v. De nombreux outils mathématiques et informatiques ont été développés pour trouver les manières les plus efficaces de les concevoir.

Cependant, ces objets sont bien réels, et peuvent donc subir des perturbations extérieures qu'il est important de prendre en compte. Dans ce projet, nous allons prendre l'exemple de manifestants bloquant les routes d'une ville, qui entraîne naturellement des perturbations dans le réseau routier. Il est donc important de pouvoir savoir comment ces blocages vont impacter le trafic. Nous proposons de modéliser ce réseau perturbé par un graphe évoluant dans le temps dont les perturbations retirent des liens, que nous appellerons GT

Nous regarderons cette évolution de manière discrète avec des pas de temps t_i allant de 0 à $t_{\max}$, avec GT_{t_i} l'état du graphe temporel à l'instant t. $t_{\max}$ étant calculé en fonction du diamètre de la ville

$$diametre = \max_{\forall u \forall v \in V} \{\text{longueur}(PlusCourtChemin(u, v))\} \text{ [1]}$$

Nous appelerons le blocage d'un lien u $\xrightarrow{\text{w}}$ v à un temps t_i par un obstacle le fait de rendre inaccessible le lien u $\xrightarrow{\text{w}}$ v de GT_{t_i}, et une attaque A l'ensemble des blocages de liens à chaque pas de temps. Nous avons donc

$$GT = \{GT_i, \forall i \in [0, t_{\max}]\} \equiv (G, A)$$

Le but du projet est de mesurer l'évolution d'une mesure proposée pour savoir si un graphe est robuste face à une attaque appelée l'efficacité [2]

$$\varepsilon = \int_{t,t'} \sum_{u,v \in V} \frac{1}{distance_{u,v}(t, t')}, \text{ avec } t' > t$$

Nous importerons les villes attaquées depuis une bibliothèque de données de rues nommées OpenStreetMap [3], puis le paquet python OSMnx [4] pour les convertir en graphes que nous pourrons manipuler. Nous simulerons ensuite des attaques sur ces villes, avec différentes stratégies, et avec un nombre de liens attaqués variable, que nous appellerons le budget de l'attaque, où chaque lien a un prix. Pour finir, nous calculerons l'impact de ces attaques sur l'efficacité de la ville, afin de pouvoir les comparer.

2 Génération du graphe

Le graphe récupéré par OSMnx est sous la forme d'un multigraphe orienté : pour tout couple de sommets $(u, v) \in E$ le nombre de liens allant de u à v n'est pas limité à 1. Cela est notamment dû à des rues parallèles. Dans ce cas, $e \in E$ est tel que $e = (u, v, k, w)$ où $u, v \in V$, $w \in \mathbb{N}$ est la pondération du lien, et $k \in \mathbb{N}$ est un indice qui permet de différencier les multiples liens de u à v.

Afin de faciliter la manipulation du graphe, nous avons créé une classe *CityGraph* qui prends comme paramètre le nom d'une ville et propose différentes méthodes le manipuler ou récupérer les informations du graphe. Elle permet également de l'enregistrer dans un fichier pour le programme de simulation.

1. Cette formule ne marche que si le graphe est connexe, sinon il faut prendre le maximum des diamètres des composantes connexes.
2. L'équation présente la version continue de la mesure, notre version discrète utilisera une somme à la place de l'intégrale.
3. https://www.openstreetmap.org
4. https://osmnx.readthedocs.io/en/stable/

2.1 Transformation du graphe en Matrice creuse

Nous avons choisi de représenter le graphe sous la forme d'une matrice creuse en format CSR (Compressed Sparse Row). Sous cette représentation, le graphe est représenté par trois tableaux :

— ROW_INDEX qui contient les bornes des intervalles de valeurs liés aux liens partant d'un noeud u.

— COL_INDEX et V qui contiennent les sommets adjacents des noeuds u et le poids des liens. Pour un noeud u d'indice i, ses liens sortants se situent dans les sous-tableaux correspondants aux indices

$$[\![ROW\_INDEX[i], ROW\_INDEX[i+1]-1]\!]$$

Si u n'a aucun lien sortant, $i = i+1$ et le sous-tableau est vide.

Nous transformons le graphe au moment de l'écriture des fichiers. Pour cela, nous utilisons les fonctions des librairies NetworkX et SciPy afin d'obtenir les tableaux ROW_INDEX et COL_INDEX. Le tableau V doit est calculé via l'algorithme 1 car :

1. Nous utilisons des distances temporels et non la distance physique entre les deux noeuds.
2. La ville est représentée sous la forme d'un multigraphe alors que les simulations sont faites sur un graphe avec au plus un lien d'un sommet u à v.

Algorithm 1 Calcul de V

Require : G : multi-graphe, ROW_INDEX, COL_INDEX
  for $i \in [\![0, G.NumNodes]\!]$ **do**
    for $j \in ROW\_INDEX[i, i+1]$ **do**
      $weigth \leftarrow \emptyset$
      for all $e \in G.edges[i][COL\_INDEX[j]]$ **do**
        $weigth.append\left(\frac{e.length}{e.maxspeed}\right)$
      end for
      $V[j] \leftarrow min(weigth)$
    end for
  end for

L'algorithme 1 parcourt l'ensemble des couples $(u, v) \in V^2$ tels qu'il existe au moins un lien de u à v. Après, il parcourt tout les liens de u à v et calcul le minimum des distances temporelles, donnés par la formule

$$\Delta t = \frac{d}{v}$$

Où la durée Δt est la distance temporelle, d la longueur du lien et v la vitesse maximale. Certaines données nécessaires aux calculs peuvent êtres manquantes et sont remplacées par des valeurs par défaut pour la vitesse max.

3 Génération des attaques

À chaque pas de temps t_i, nous allons attaquer le graphe 1 en bloquant des liens jusqu'à un certain budget. Nous laissons le choix du prix de chaque lien à l'utilisateur, soit une constante, 1 dans notre cas, soit une fonction qui détermine le prix en fonction de la largeur de la rue bloquée. Avec la largeur de la rue qui correspond à son nombre de voies de circulation.

Nous avons choisi de séparer la gestion des attaques des stratégies d'attaques. Pour la gestion, nous avons la classe Attaque, avec 2 primitives :

— Bloquer un lien à un temps donné

— Retranscrire l'attaque dans un fichier pour la simulation

Comme ça, les stratégies peuvent se focaliser sur l'algorithme de choix des liens à supprimer, et la classe Attaque se charge de faire le reste.

Nous classons les stratégies d'attaques en deux catégories : les attaques dynamiques et les attaques statiques.

3.1 Attaques dynamiques

Les attaques dynamiques sont des attaques où un lien peut être bloqué puis débloqué plus tard, notamment, car les obstacles se déplacent au cours du temps.

3.1.1 Attaque aléatoire

La première stratégie que nous allons étudier est la plus bête et méchante d'entre toutes : l'attaque aléatoire. Elle consiste à attaquer des liens aléatoirement, sans se soucier de leur importance. Cette attaque nous servira de base pour comparer les autres stratégies.

Algorithm 2 Attaque aléatoire

```
for t ← 0 to TempsMax do
    LiensRestants ← Mélanger(Graphe.liens())
    BudgetRestant ← Budget
    while BudgetRestant > 0 ∧ LiensRestants ≠ ∅ do
        lienABloquer ← Liens.pop()
        Attaque.bloquer(lienABloquer, t)
        Prix ← lienABloquer.prix()
        BudgetRestant ← BudgetRestant − Prix
    end while
end for
```

3.1.2 Attaque mouvante

Pour cette stratégie, nous commençons par une attaque aléatoire pour le premier pas de temps t_0, mais pour les suivants, les obstacles bloquant un lien vont se déplacer vers un lien voisin non bloqué s'ils le peuvent. Dans notre contexte de manifestations et de marches, cette stratégie est plus réaliste, car les manifestants vont se déplacer dans la ville.

Algorithm 3 Attaque mouvante

```
LiensOriginaux ← AttaqueAléatoire(Graphe, t_max=1, Budget)
for t ← 1 to t_max do
    NewLiens ← ∅
    for lien ← LiensOriginaux do
        Voisins ← lien.voisins()
        for v ← Voisins do
            if v ∉ NewLiens then
                NewLiens.bloquer(v)
                continue to next lien
            end if
        end for
        NewLiens.bloquer(lien)
    end for
    LiensOriginaux ← NewLiens
end for
```

3.2 Attaques statiques

Au contraire des attaques dynamiques, les attaques statiques sont des attaques où un lien bloqué est bloqué de t_0 à $t_{\max}$.

3.2.1 Attaque par centralité intermédiaire

Dans l'état-de-l'art, l'attaque par centralité intermédiaire est considérée comme la stratégie la plus efficace. La centralité intermédiaire est une mesure représentant l'importance d'un lien dans un graphe. Elle est calculée en comptant le nombre de plus courts chemins passant par ce lien.

Cette stratégie consiste donc à bloquer les liens les plus importants en les triant par centralité intermédiaire. Nous notons d'ailleurs qu'OSMnx contient des options pour récupérer les centralités intermédiaires des liens d'un graphe directement.

Algorithm 4 Attaque par centralité intermédiaire

```
Liens ← Graphe.liens()
LiensTries ← TrierParCentralité(Liens)
for t ← 0 to TempsMax do
    BudgetRestant ← Budget
    for lien ← LiensTries do
        Attaque.bloquer(lien, t)
        Prix ← lien.prix()
        BudgetRestant ← BudgetRestant − Prix
        if BudgetRestant ≤ 0 then
            break
        end if
    end for
end for
```

Remarque : cette stratégie est en fait une stratégie générique, qui peut être utilisée pour trier les liens par n'importe quel critère, comme le nombre de voisins, le nombre de voies composant la rue, etc... Nous l'utilisons avec la centralité intermédiaire, car c'est le critère le plus couramment utilisé dans la littérature.

4 Simulation

4.1 Structure Générale du programme C

Le programme de Simulation, écrit en C a un fonctionnement assez simple. Il prend en entrée un fichier de graphe généré par le script Python CityGraph.py et un fichier d'attaque généré par le script Python Stategies.py.

L'exécution du programme se déroule alors en 3 étapes :

1. Génération d'un graphe temporel représenté par une matrice creuse et un tableau associatif ayant pour clefs un lien et en valeur la liste des temps où il est supprimé.
2. Calcul de l'évolution des plus courts chemins sur le graphe temporel en utilisant l'algorithme de "Floyd Warshall Temporel".
3. Calcul de la mesure de robustesse du réseau en utilisant les plus courts chemins calculés précédemment et affichage/écriture de la robustesse.

4.2 Structures de données internes du programme C

Le programme C utilise deux structures de données principales afin d'effectuer ces calculs. La première est celle représentant le graphe temporel, elle est composée de deux "sous-structures" :

1. Une matrice creuse représentant le graphe associé au graphe temporel (qui contient tous les noeuds et les liens qui apparaitront dans le graphe temporel)
2. Un tableau associatif ayant pour clefs un lien et en valeur la liste des temps où il est supprimé.

La seconde structure de données est celle représentant les plus courts chemins calculés par l'algorithme de "Floyd Warshall Temporel". Elle est composée d'une liste de matrices, chaque matrice représentant les plus courts chemins entre chaque liens. Cette structure est appelée "DistanceMatrixes" (ou DMA) dans le programme, où $dma.matrix[t_i][u][v]$ contient la distance minimale trouvée entre les noeuds u et v à l'instant t_i. [5].

4.3 L'algorithme de "Floyd Warhsall Temporel"

L'algorithme de "Floyd Warshall Temporel" est une adaptation de l'algorithme de "Floyd Warshall"[1] pour un graphe temporel. Il permet de calculer les plus courts chemins, à chaque instant, entre chaque paire de sommets du graphe temporel.

L'algorithme de Floyd Warshall est un algorithme permettant de calculer les plus courts chemins entre tous les noeuds d'un graphe même si celui-ci contient des noeuds de poids négatifs, donc facilement adaptable à notre problème, en plus d'être simple à implémenter. C'est pour cela que nous nous sommes basé dessus pour notre algorithme de calculs de plus courts chemins.

L'algorithme de Floyd Warshall temporel calcule les plus courts chemins pour tous les sommets du graphe temporel pour chaque pas de temps en commençant par le dernier pas de temps jusqu'au premier pas de temps. En effet, la distance d'un sommet à un autre dépend de la distance entre ces deux sommets lors des temps suivants. Pour calculer les distances à chaque pas de temps, l'algorithme compare tous les chemins possibles entre deux sommets en passant par un troisième sommet k. Cet algorithme utilise donc 3 boucles imbriquées pour calculer les plus courts chemins entre chaque paire de sommets à chaque pas de temps, et possède en conséquent une complexité temporelle de $O(|V|^3 * t_{\max})$.

$$dist(t_i, u, v) = \min(dist(t_i, u, v), dist(t_i, u, k) + dist(t + dist(t_i, u, v), k, v)) \quad (1)$$

4.4 Calcul de la mesure de robustesse

La mesure de robustesse est calulée à la fin du programme en sommant toutes les distances minimales entre les sommets atteignables à chaque pas de temps. Puis en divisant ce nombre par $t_{\max} * |V| * (|V| - 1)$ On appelle sommets atteignables à un temps t les sommets pour lesquels il existe un chemin minimal à une distance $dist(t_i, u, v) < t_i + t_{\max}$ Le nombre obtenu est ensuite renvoyé dans la sortie standard ou écrit dans un fichier.

4.5 Pseudocode de l'algorithme de Floyd Warshall Temporel et de l'initialisation des matrices de distances :

Algorithm 5 *InitTFW*(*graphetemporel GT*, *DistanceMatrixes dma*)

```
for matrix in dma : do
    for element in matrix : do
        element ← ∞
    end for
end for
for tlink in GT : do
    dma.matrix[tlink.time][tlink.link] ← tlink.weight
end for
for u in {0..|GT.V|} do
    for matrix in dma do
        matrix[u][u] ← 1
    end for
end for
```

5. Nous indexons la DMA directement avec des noeuds et des pas de temps, car chaque noeud est un nombre unique de 0 à |V| dans notre implémentation, de même pour les pas de temps.

Algorithm 6 *TemporalFloydWarshall(graphetemporel GT, DistanceMatrixes dma)*

```
InitTFW(GT, dma)
for time in {dma.NumMatrixes .. 0} do
    for k in {0 .. |GT.V| } do
        for u in {0 .. |GT.V| } do
            for v in {0 .. |GT.V| } do
                distUV ← dma.matrix[t_i][u][v]
                distUK ← dma.matrix[t_i][u][k]
                distKV ← dma.matrix[t_i + distUK][k][v]
                if distUV > distUK + distKV then
                    dma.matrix[t_i][u][v] ← distUK + distKV
                end if
            end for
        end for
    end for
end for
```

5 Travail restant

5.1 Lancement des simulations

Nous avons écrit le programme de calcul de l'efficacité, le programme de génération des attaques, ainsi qu'un script pour lier les deux. Nous avons testé le projet sur des villes assez petites, et les résultats obtenus sont cohérents, il ne reste plus qu'à les lancer sur des villes comme Paris sur les supercalculateurs du lip6.

5.2 Analyse des résultats

Une fois les simulations terminées, nous pourrons analyser les résultats, et voir comment les différentes attaques impactent l'efficacité des villes. Nous pourrions notamment trouver des corrélations entre les caractéristiques des villes et leur résilience face aux attaques, des points de rupture, ou des stratégies d'attaques plus efficaces que d'autres.

6 Perspectives

Dans cette section, nous allons discuter de diverses propositions pour améliorer le projet. Elles ne sont pas obligatoires, mais elles pourraient être intéressantes à explorer.

6.1 Optimisation du programme

Le programme actuel est très lent, dû à la complexité des algorithmes utilisés, dont l'algorithme de Floyd-Warshall temporel, qui a une complexité en $O(|V|^3 * t_{\max})$4.3. Pour améliorer la vitesse du programme, nous allons discuter des 2 extrémités du spectre de l'optimisation : paralléliser les caluls ou faire moins de calculs. Sachant que ces deux solutions ne sont pas mutuellement exclusives, et qu'il est possible de les combiner.

6.1.1 Parallélisation des calculs

Parallélisation de l'algorithme de Floyd-Warshall

Avec autant de calculs sur des parties différentes du graphe, on peut se douter que l'algorithme de Floyd-Warshall est parallélisable. Nous avons donc cherché sur internet et il existe des papiers sur un Floyd-Warshall multi-threadé[2][3], et même une implémentation GPU[4], qui promettent des gains de performance considérables, jusqu'à 50 fois plus rapide. De plus, nous avons déjà écrit un wrapper sur l'API de Vulkan permettant de faire des calculs sur le GPU facilement depuis le programme C.

Parallélisation des simulations

Le but du projet est de calculer l'évolution de l'efficacité du graphe en fonction du budget, et chaque instance de la simulation est entièrement indépendante des autres. Nous pouvons donc facilement paralléliser les instances sur plusieurs coeurs, voire plusieurs machines, et les faire tourner en même temps, pour maximiser l'utilisation du CPU, et attendre moins longtemps pour obtenir les résultats.

Mais cela ne résout pas le problème de la complexité et le gain dépend du nombre de coeurs disponibles, sans parler du coût énergétique de l'opération.

6.1.2 Moins de calculs

Utilisation d'un cache

Pour les attaques avec peu de budget, nous pouvons nous imaginer que la plupart des plus courts chemins ne changent pas, car ils ne passent pas par les liens bloqués. Nous pouvons donc mettre en place un système de stockage pour les plus courts chemins sur le graphe non-attaqué, et ne recalculer que les plus courts chemins qui ont changé. Cela nécessite de changer notre algorithme pour calculer les plus courts chemins plutôt que leurs distances, ainsi qu'un système efficace pour invalider les plus courts chemins impactés par les attaques.

Propriétés des attaques statiques

Pour optimiser, il est important de prendre en compte toutes les propriétés de nos données, notamment les attaques statiques. Comme les attaques statiques ne changent pas, on peut calculer les distances minimales sur le graphe attaqué une fois pour toutes, et les propager dans le temps, plutôt que de les recalculer à chaque pas de temps.

Structure de données plus efficace

Les graphes temporels sont assez difficiles à manipuler, notamment à cause de liens qui peuvent être bloqués à certains moments, mais pas à d'autres. La plupart des algorithmes de graphes classiques sont donc rendus inutilisables, car ils supposent que le graphe est statique, et ne voient pas dans le futur. Prenons notamment l'exemple de l'algorithme de Dijkstra, qui est très utile pour trouver les plus courts chemins, mais qui ne peut pas être utilisé dans notre implémentation actuelle. Soit le graphe temporel GT ci-dessous 1, où chaque lien est étiqueté par sa distance, et considérons que nous voulons aller de 0 à 2 au temps 0.

FIGURE 1 – Graphe temporel GT

Pour Dijkstra, au temps 0, le seul noeud visitable est le noeud 1 et va donc passer par le lien $0 \xrightarrow{1} 1$, une fois au noeud 1, le seul lien accessible à parcourir est le lien $1 \xrightarrow{3} 2$, et il va donc prendre ce chemin, pour un coût total de 4. Alors que nous pouvons facilement voir que le plus court chemin est en fait d'attendre sur 0, puis de prendre le lien $0 \xrightarrow{1} 2$, pour un coût total de 2.

Cependant, nous avons récemment pensé à une nouvelle structure de données. Désormais, plusieurs liens peuvent exister entre 2 noeuds, mais ils ne sont pas tous actifs en même temps. Nous allons donc avoir des liens fantômes qui remplacent les liens supprimés, équivalents à la distance réelle plus l'attente nécessaire.

Cela va permettre de ne pas avoir besoin de regarder dans le futur pour calculer les chemins, tout en calculant quand même la vraie distance minimale, et ainsi rendre beaucoup plus de parcours de graphes adaptables temporellement, notamment Dijkstra, Bellman-Ford, A*, Johnson, etc...

Algorithm 7 Construction de la structure de données

```
NouveauGraphe ← ∅
for lien ← Attaque.liensBloqués() do
    t_BlocageInitial ← lien.tempsBlocageInitial()
    t_BlocageFinal ← lien.tempsBlocageFinal()
    for t ← t_BlocageInitial to t_BlocageFinal do
        LienFantome ← new LienFantome(lien.origine, lien.destination)
        LienFantome.poids ← lien.poids + (t_BlocageFinal − t)
        NouveauGraphe.ajouterLienDisponibleA(LienFantome, t)
    end for
end for
```

Cette solution est la plus compliquée à implémenter, mais aussi très prometteuse. Notre implémentation prototype montre déjà des résultats encourageants, mais il faut encore vérifier la validité des résultats et tester la performance sur des graphes plus grands.

6.2 Nouvelles attaques

L'équipe Complex Networks du Lip6 [6] travaille sur des coupes de graphes, c'est-à-dire découper le graphe en sous-composantes connexes, optimales sur des réseaux urbains tel que Paris. Nous pourrions donc utiliser ces coupes en tant qu'attaques, et comparer leur efficacité à d'autres attaques plus classiques.

6.3 Modification de la mesure

L'efficacité est une mesure intéressante, mais elle ne prend pas en compte la variation comparée à un graphe non-attaqué. Nous pourrions donc modifier la mesure pour prendre en compte cette variation, afin de comparer le changement d'efficacité plutôt que l'efficacité elle-même.

7 Conclusion

Durant cette première partie de projet, nous avons donc posé les bases du travail à effectuer, en implémentant des parties modulaires d'un programme de mesure d'efficacité sur des graphes temporels. Le plus gros point à traiter reste la partie du programme écrite en C, chargée de calculer la robustesse, qui n'est pas assez efficace pour des graphes de grande taille. C'est pour cela que nous allons explorer différentes pistes d'optimisations afin de pouvoir efficacement produire des données, en parallèle de la suite du projet.

6. https://www.lip6.fr/recherche/team.php?acronyme=ComplexNetworks

Références

[1] S. Hougardy, "The floyd–warshall algorithm on graphs with negative cycles," *Information Processing Letters*, vol. 110, no. 8-9, pp. 279–281, 2010.

[2] Students of the Parallel Processing Systems course, School of Electrical & Computer Engineering, National Technical University of Athens, "Parallelizing the floyd-warshall algorithm on modern multicore platforms : Lessons learned,"

[3] A. Pradhan and G. Mahinthakumar, "Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms," *Journal of computing in civil engineering*, vol. 27, no. 3, pp. 263–273, 2013.

[4] Dhananjay Kulkarni, Neha Sharma, Prithviraj Shinde, Vaishali Varma, "Parallelization of shortest path finder on gpu : Floyd-warshall," *International Journal of Computer Applications (0975 8887)*, 2015.