Compte rendu intermédiaire du projet de recherche

Ivan, Pierre, Damien

Sommaire

Contents

1 Introduction

L'objectif de ce projet est de mesurer la robustesse d'une ville et l'impact d'attaques sur ses liens. Pour ce faire, nous représentons le graphe sous la forme d'un Linkstream. Des liens peuvent disparaitre pour un certain temps car bloqués par une attaque.

Le graphe est implémenté sous la forme d'une matrice creuse format CSR (Compressed sparse row) où la matrice est représenté par trois tableaux :

- ROW_INDEX qui contient les bornes des intervalles de valeurs liés aux liens partant d'un sommet s_i
- COL_INDEX et V qui contiennent les sommets adjacents des sommets s_i et le poids des liens, dont les valeurs se situent dans $[\![ROW\_INDEX[i], ROW\_INDEX[i+1]]\!]$ si s_i a au moins un lien sortant $(i < i+1)$, sinon $i = i+1$

2 Génération du graphe

Le graphe est récupéré via la librairie Python OSMNX[1], sous la forme d'un multi-graphe orienté. La gestion du graphe est facilité par l'utilisation d'une classe dont le constructeur prends comme paramètre le nom d'une ville et propose différentes méthodes le manipuler ou récupérer des informations.

Transformation du graphe en Matrice creuse

Le graphe est transformé en une matrice creuse au moment de l'écriture des fichiers pour le programme C. Certaines données nécessaires aux calculs peuvent êtres et sont remplacées par des valeurs par défaut en constantes qui peuvent êtres modifiées

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**
      $weigth \leftarrow []$
      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

[1] `osmnx.readthedocs.io/en/stable/`

3 Génération des attaques

Les attaques:

Pour l'attaque, nous possédons un budget de liens à supprimer, qui peut être, au choix, une constante, ou une fonction qui renvoie la largeur de la rue attaquée, déterminée avec le nombre de voies.

Le problème se transforme donc en problème d'optimisation: trouver les liens à supprimer pour minimiser la robustesse du graphe tout en respectant le budget. Calculer la solution parfait comme un vrai problème d'optimisation est impossible, nous allons donc utiliser différentes stratégies d'attaques, et les comparer.

Nous avons la classe Attaque permettant de gérer ceci, avec 2 primitives:

- Attaquer 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 gérer le reste.

Les stratégies peuvent se classer en 2 catégories :

- Statiques, qui suppriment les liens une fois pour toute
- Dynamiques, où un lien peut être supprimé puis réapparaître plus tard

Attaques Dynamiques

L'attaque aléatoire:

La première stratégie, est de supprimer des liens aléatoirement à chaque pas de temps. C'est simple à implémenter, mais ne donne pas de résultats très intéressants.

for $t \leftarrow 0$ to TempsMax **do**
  $LiensRestants \leftarrow$ Mélanger(Graphe.liens())
  $BudgetRestant \leftarrow$ Budget
  while $BudgetRestant > 0$ **do** and $LiensRestants \neq \emptyset$
    $lienSuppr \leftarrow Liens$.pop()
    $Attaque$.ajouter($lienSuppr$, t)
    $Prix \leftarrow lienSuppr$.prix()
    $BudgetRestant \leftarrow BudgetRestant - Prix$
  end while
end for

L'attaque mouvante:

La deuxième stratégie est de commencer par une attaque aléatoire, mais à chaque pas de temps, les obstacles bloquant un lien vont se déplacer vers un lien

voisin. Elle est plus réaliste, dans le contexte de manifestations par exemple, mais est plus compliquée à implémenter.

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

Attaques Statiques

L'attaque par centralité intermédiaire:

La troisième stratégie est de supprimer les liens les plus importants pour relier les noeuds entre eux. La centralité intermédiaire est une bonne mesure de l'importance d'un lien dans un graphe, en calculant le nombre de plus courts chemins passant par ce lien. Cette stratégie est la plus intelligente, mais aussi la plus coûteuse en temps de calcul.

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

Note : Cet algorithme est juste un algorithme de séléction des maximums en triant selon un certain critère, et est donc générique, nous utilisons l'exemple de la centralité intermédiaire, étant le plus pertinent.

4 Le programme de Simulation

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'execution du programme se déroule alors en 3 étapes :

1. Génération d'un LinkStream 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 LinkStream 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.

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 LinkStream, elle est composée de deux "sous-structures" :

1. Une matrice creuse représentant le graphe associé au LinkStream (qui contient tous les noeuds et les liens qui apparaitront dans le LinkStream)
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. pour indexer une DMA, on utilise la notation $dma.matrix[t][i][j]$ où t est le temps, i et j sont les sommets du graphe.

L'algorithme de "Floyd Warhsall Temporel" :

L'algorithme de "Floyd Warshall Temporel" est une adaptation de l'algorithme de "Floyd Warshall" pour un LinkStream. Il permet de calculer les plus courts chemins, à chaque instant, entre chaque paire de sommets du LinkStream. La complexité de cet algorithme est en $O(t * n^3)$ où n est le nombre de sommets du LinkStream et t le nombre de "pas de temps" du LinkStream.
TODO : EXPLIQUER POURQUOI FLOYD WARSHALL ET PAS AUTRE CHOSE L'algorithme de Floyd Warshall temporel calcule les plus courts chemins pour tout les sommets du LinkStream 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 tout les chemins possibles entre deux sommets en passant par un troisième sommet.
ie : $dist(t,i,j) = min(dist(t,i,j), dist(t,i,k) + dist(t + dist(t,i,j), k, j))$

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
$tmax * nbNodes * (nbNodes - 1)$ où tmax est le temps maximal du LinkStream et nbNodes le nombre de sommets du LinkStream. On appelle sommets atteignables à un temps t les sommets pour lesquels il existe un chemin minimal à une distance
$dist(t,i,j) < t + tmax$ où tmax est le temps maximal du LinkStream. Le nombre obtenu est ensuite renvoyé dans la sortie standard ou écrit dans un fichier.

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

```
InitTFW(LinkStream lks, DistanceMatrixes dma)
for matrix in dma : do
    for element in matrix : do
        element ← ∞
    end for
end for
for tlink in lks : do
    dma.matrix[tlink.time][tlink.link] ← tlink.weight
end for
for i in {0..lks.nbnodes} do
    for matrix in dma do
        matrix[i][i] ← 1
    end for
end for
```

```
TemporalFloydWarshall(LinkStream lks, DistanceMatrixes dma)
InitTFW(lks, dma)
for time in {dma.NumMatrixes .. 0} do
    for k in {0 .. lks.NumNodes } do
        for i in {0 .. lks.NumNodes } do
            for j in {0 .. lks.NumNodes } do
                distIJ ← dma.matrix[t][ij]
                distIK ← dma.matrix[t][ik]
                distKJ ← dma.matrix[t + distIK][kj]
                if distIJ > distIK + distKJ then
                    dma.matrix[t][ij] ← distIK + distKJ
                end if
            end for
        end for
    end for
end for
```

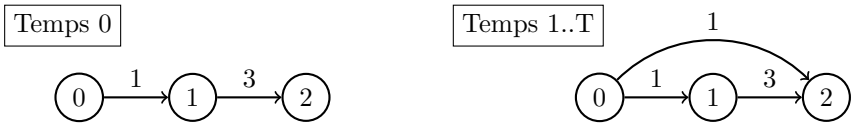
5 Progrès prévus

- Parallélisation : Le but du projet est de calculer l'évolution de la robustesse du graphe en fonction du budget, et chaque simulation est indépendante des autres. On peut donc paralléliser les instances sur plusieurs coeurs, et les faire tourner en même temps. Mais cela ne résoud pas le problème de la compléxité et le gain dépend du nombre de coeurs disponibles. Sinon, l'algorithme de Floyd-Warshall est parallélisable, il existe des papiers sur un Floyd-Warshall multi-threadé, et même un sur GPU. De plus, nous avons écrit un wrapper sur l'API de Vulkan permettant de faire des calculs sur le GPU facilement depuis le programme C.

- Mise en place d'un cache de chemins : Pour des graphes peu attaqués, il est possible que les plus courts chemins ne changent pas beaucoup, on peut donc calculer des plus courts chemins sur le graphe de base, et les garder en mémoire, puis recalculer seulement les chemins possédant un lien attaqué. Cependant, cela nécessite de modifier l'algorithme pour calculer des chemins plûtot que des distances.

- Propagation des résultats pour les attaques statiques : Comme les attaques statiques ne changent pas, on peut calculer les distances minimales une fois pour toute, plutôt que de les recalculer à chaque pas de temps.

- Changement de la structure de données : Les linkstreams sont assez difficiles à manipuler, notamment à cause de liens qui peuvent disparaître et réapparaître. La plupart des algorithmes de graphes classiques sont donc rendus inutilisables, car ils supposent que le graphe est statique. 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 G:



  Et considérons que nous voulons aller de 0 à 2 au temps 0. Pour Dijkstra, au temps 0, le seul noeud visitable est le noeud 1 et va donc passer par le lien 0 –1–> 1, une fois au noeud 1, le seul lien accessible à parcourir est le lien 1 –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 –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. On va 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, et rendre beaucoup plus de parcours de graphes adaptables temporellement, notamment Dijkstra, Bellman-Ford, A*, Johnson, etc.

Construction : Pour chaque lien (A –p–> B) supprimé d'un temps t à t', Pour chaque pas de temps de tx de t à t', on ajoute un lien (A –p+(t'-tx)–> B, tx) qui sera disponible au temps tx, seulement.

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 sur des graphes plus grands.

Tout en considérant que ces idées ne sont pas mutuellement exclusives, et peuvent être combinées pour un gain de performance plus important.

Attaque par coupe de graphe

D'autres équipes du lip6 ont travaillé sur des coupes de graphe, c'est-à-dire le diviser en sous-composantes connexes, et le but est de minimiser le nombre de liens à supprimer pour obtenir cette division. Nous allons récupérer certaines de leurs coupes calculées, et les utiliser pour attaquer le graphe, afin de vérifier si elles sont pertinentes.

6 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 de robustesse sur des linkstream. Cependant, la partie du programme écrite en C , chargée de calculer la robustesse, n'est pas assez efficace pour des linkstream de grande taille, c'est pour cela que nous allons explorer différentes pistes d'optimisations afin de pouvoir efficacement produire des données.

References

[1] Simart Fréderic (2021), Evaluating metrics in link streams, Social Network Analysis and Mining

[2] Anu Pradhan, G. (Kumar) Mahinthakumar, (2012) 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

[3] D.Kulkarni, N.Sharma, V.Varma, P.Shinde (2015) Parallelization of Shortest Path Finder on GPU: Floyd-Warshall, International Journal of Computer Applications

[4] atapy, M., Viard, T. , Magnien, C. (2018) Stream graphs and link streams for the modeling of interactions over time. Soc. Netw. Anal. Min. 8, 61

[5] M.H.Xu , Y.Q.Liu, Q.L.Huang, Y.X.Zhang, G.F.Luan (2006) An improved Dijkstra's shortest path algorithm for sparse network, Procedia Computer Science

[6] W.Peng , X.HU, F.Zhao, J.Su (2012) A Fast Algorithm to Find All-Pairs Shortest Paths in Complex Network, Procedia Computer Science