Compte Rendu de Mi-Projet de Recherche

Pierre, Ivan, Damien

13 mars 2024

Je crois que je mélange robustesse et efficacité des fois je sais plus ptdr

Je crois aussi que j'utilise la voix passive des fois, à vérifier

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

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 l'ensemble des liens e = (u,v,w) où u et v sont les sommets reliés par le lien e et w est le poids de ce lien. 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. Nous regarderons cette évolution de manière discrète avec des pas de temps t_i allant de 0 à $t_{\max}$, $t_{\max}$ étant calculé en fonction du diamètre de la ville d=<plus long plus court chemin>. . Nous appelerons le blocage d'un lien (u,v,w) par un obstacle le fait de rendre inaccessible le lien (u,v) du graphe pour un instant t donné. GT = (G, A), avec G le graphe d'une ville G = (V,E), avec V qui représente les carrefours et intersections, et E qui représente les rues. A l'attaque du graphe, qui est l'ensemble des blocages de liens. 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 ϵ=<Insérer big formule ici>

insérer la formule du diamètre

insérer la formule d'efficacité

Les villes attaquées seront importées depuis une bibliothèque de données de rues nommées OpenStreetMap, puis le paquet OSMnx 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

Explique comment on génère le graphe

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 1 Attaque aléatoire

for $t \leftarrow 0$ to TempsMax **do**
  $LiensRestants \leftarrow$ Mélanger(Graphe.liens())
  $BudgetRestant \leftarrow$ Budget
  while $BudgetRestant > 0 \wedge LiensRestants \neq \emptyset$ **do**
    $lienABloquer \leftarrow Liens$.pop()
    $Attaque$.bloquer($lienABloquer$, t)
    $Prix \leftarrow lienABloquer$.prix()
    $BudgetRestant \leftarrow BudgetRestant - Prix$
  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 2 Attaque mouvante

$LiensOriginaux \leftarrow$ AttaqueAléatoire(Graphe, $t_{\max}$=1, Budget)
for $t \leftarrow 1$ to $t_{\max}$ **do**
  $NewLiens \leftarrow \emptyset$
  for $lien \leftarrow LiensOriginaux$ **do**
    $Voisins \leftarrow lien$.voisins()
    for $v \leftarrow Voisins$ **do**
      if $v \notin NewLiens$ **then**
        $NewLiens$.bloquer(v)
        continue to next $lien$
      end if
    end for
    $NewLiens$.bloquer($lien$)
  end for
  $LiensOriginaux \leftarrow NewLiens$
end for

Peut etre réunir l'algorithme à la meme page que le titre

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 que OSMnx contient des options pour récupérer les centralités intermédiaires des liens d'un graphe directement.

Algorithm 3 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

Explique comment on simule

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 tester 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 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 $O(|V|^3 * t_{\max})$. 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.

Reference vers la section correspondante

6.1.1 Parallélisation des calculs

6.1.1.1 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[1], et même une implémentation GPU[2], 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.

6.1.1.2 Parallélisation des simulations

Le but du projet est de calculer l'évolution de la robustesse 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 plusieures 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 résoud pas le problème de la compléxité et le gain dépend du nombre de coeurs disponibles, sans parler du coût énergétique de l'opération.

hack dégueu pour mettre un espace

6.1.2 Moins de calculs

6.1.2.1 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 algortihme pour calculer les plus courts chemins plûtot que leurs distances, ainsi qu'un système efficace pour invalider les plus courts chemins impactés par les attaques.

6.1.2.2 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 toute, et les propager dans le temps, plutôt que de les recalculer à chaque pas de temps.

6.1.2.3 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 G ci-dessous1, où chaque lien est étiqueté par sa distance, et considérons que nous voulons aller de 0 à 2 au temps 0.

Beurk ce code

FIGURE 1 – Graphe temporel G

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, tout en calculant quand même la vraie distance minimale. Cette nouvelle structure de données va rendre beaucoup plus de parcours de graphes adaptables temporellement, notamment Dijkstra, Bellman-Ford, A*, Johnson, etc...

Algorithm 4 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

D'autres équipes du lip6 ont travaillé sur des coupes de graphe, c'est-à-dire le diviser en sous-composantes connexes, et leur 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 voir si elles sont pertinentes.

citer les gens qui ont fait ça

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 plûtot que l'efficacité elle-même.

7 Conclusion

Conclusi

Références

[1] 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,"

[2] 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.