Congestion en milieu urbain
fluidifier le trafic grâce au routage dynamique

Martin Pupat - 26609 - MPI

1. Introduction

L'utilisation intensive des voitures par les voyageurs pendulaires dans la société actuelle, en dépit d'une infrastructure routière urbaine sous-dimensionnée, s'est manifestée par la croissance des temps de transport quotidiens en raison des embouteillages. Pour résoudre ce problème et réduire au maximum les temps de transport, la représentation du problème comme un problème de routage statique (dont des solutions sont données par l'algorithme A^* ou de Dijkstra) n'est pas suffisante puisque les routes sont susceptibles de se retrouver bloquées à tout moment.
Dans la littérature [1, 2, 3, 4, 5, 6], il existe de nombreuses méthodes pour router une voiture individuelle dans un graphe routier en évolution, mais aucune généralisation n'a été faite à ma connaissance pour étudier l'ensemble des voitures. Dans ce TIPE, on s'intéresse à un cas de ce problème dans lequel les voitures ont une connaissance partielle de la position des autres voitures sur le complexe routier.

2. Définition du problème

Définition : Un **problème de routage dynamique** est un problème d'optimisation formalisable de la manière suivante.

- Données : un graphe routier $G(t) = (S, A, \omega(\alpha \in A, t))$ ($|S| = n$, $|A| = m$) dont la pondération ω, représentant le temps pour parcourir une arête, évolue suivant t ; v agents $V_1, \dots, V_v$; des nœuds de départ $a_1, \dots, a_v$ pour ceux-ci, des nœuds $b_1^1, \dots, b_1^{p_1}, \dots, b_v^{p_v}$ à parcourir pour ceux-ci (suivant un ordre précis ou non).
- Optimisation : Le temps minimum pour que tous les agents soient passés par tous leurs nœuds à parcourir.

Exemples : Problème du voyageur de commerce dynamique ($v = 1$, $p_1 > 1$), problème de collecte de déchets ($n > 1$, les b_i sont collectifs à tous les agents)...

Dans le cadre d'étude de ce TIPE, on effectue les considérations suivantes :

- Chaque agent (voiture) n'a qu'un point de départ et d'arrivée, donc $\forall i \in [\![1, v]\!]$, $p_i = 1$.
- La pondération $\omega(\alpha)$ d'une arête donnée $\alpha \in A$ dépend du nombre d'agents μ engagés dessus. À chaque arête on associe une capacité $q(\alpha) \in \mathbb{N}$ (avec Q la capacité totale du graphe) et une durée $\tau \in \mathbb{R}_+$. On a alors $\omega(\alpha) = \tau$ si $\mu(\alpha) \leqslant q(\alpha)$ et $\omega(a) = +\infty$ sinon (dans ce cas, la route est dite **saturée**).
- On restreint l'étude à $v < Q$ agents ; on appelle le rapport v/Q la **saturation** du graphe.
- À tout instant, chaque agent a une connaissance partielle du graphe (soit uniquement l'arête sur laquelle il s'engage, soit toutes les arêtes en aval).
- Tous les calculs se font via l'intermédiaire d'un « calculateur » centralisé, qui a une connaissance totale du graphe.

De plus, le graphe est supposé connexe et planaire (en réalité, il est quasi-planaire puisque les ponts et les tunnels sont rares en mileu urbain, mais existent). On a $|S| - 1 \leqslant |A| \leqslant 3|S| - 6$, donc $m = \mathcal{O}(n)$ et $n = \mathcal{O}(m)$ (preuve en partie 1 de l'annexe).

3. Algorithmes

Dans un graphe statique, pour calculer le chemin le plus court dans G entre deux points a et b, on utilise l'algorithme A^* avec comme heuristique la distance géodésique entre les deux points, qui fournit un chemin de poids minimal et a une complexité temporelle en $\mathcal{O}(n \log n)$ (preuve : partie 2 de l'annexe).

Dans les graphes sur lesquels j'ai conduit mon étude expérimentale, une approche en brute-force montre que le diamètre des graphes (distance maximale entre 2 sommets) est inférieur à $\sqrt{n}$ (avec n le nombre de sommets). Le calculateur renvoie donc des chemins de longueur $L \in \mathcal{O}(\sqrt{n})$.

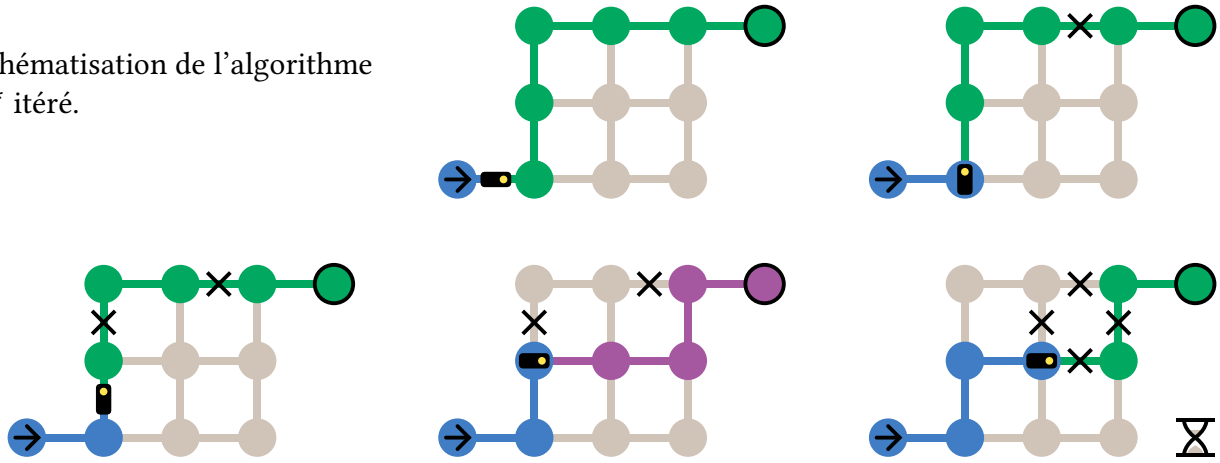
Deux algorithmes se présentent à nous pour résoudre des problèmes de routage dynamique.

3.1. Algorithme A^* itéré

L'algorithme A^* itéré consiste en les étapes suivantes pour un agent donné :

- Le calculateur calcule le chemin le plus court entre le départ et l'arrivée.
- L'agent suit ce chemin.
- Si l'agent est sur le point de s'engager sur une arête saturée, il s'arrête et la machine cherche un chemin entre sa position actuelle et l'arrivée.
- Si ce chemin existe, l'agent poursuit sa route.
- Sinon, il attend un temps δt et le calculateur réessaie.

Schématisation de l'algorithme A^* itéré.



Explication : (i) L'agent s'engage sur un chemin pour rejoindre le point en haut à droite ; (ii) Une arête de celui-ci devient saturée, mais pas celle immédiatement devant l'agent, donc il continue son chemin ; (iii) L'agent va s'engager sur une arête saturée ; (iv) Le calculateur trouve un nouveau chemin sans arête saturée, en violet ; (v) Lorsqu'il n'existe pas de telle alternative, l'agent attend.

3.2. Algorithme BallString simplifié

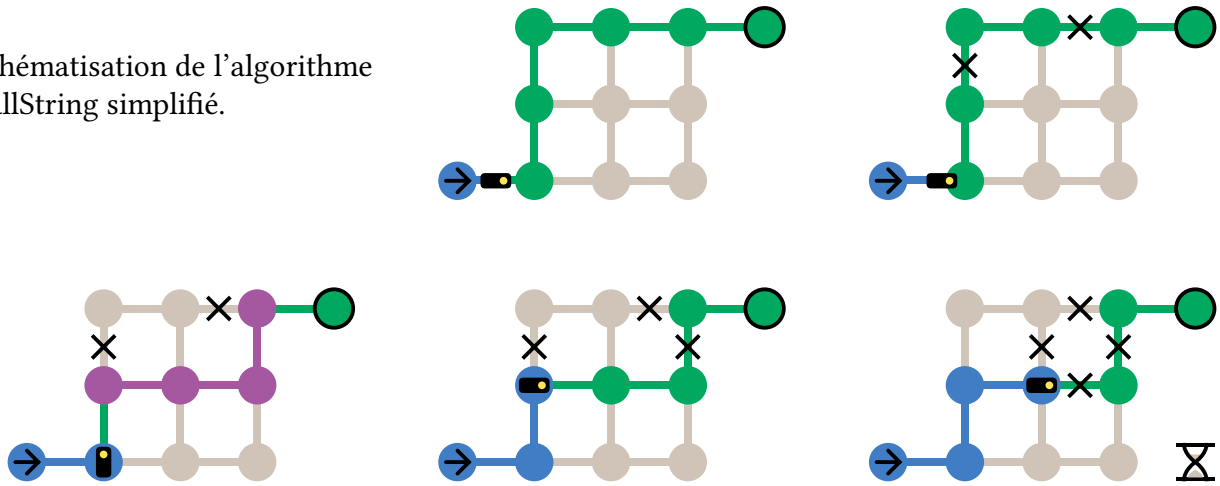
L'algorithme BallString simplifié (que l'on appellera aussi BallString par la suite) consiste en les étapes suivantes pour un agent donné :

- Le calculateur calcule le chemin l le plus court entre le départ et l'arrivée.
- L'agent suit ce chemin.
- Chaque fois que l'agent est sur le point de s'engager sur une certaine arête α :
  - Le calculateur calcule le plus petit chemin f inclus dans l qui contient toutes les arêtes saturées de l.
  - En notant f_- et f_+ les sommets initiaux et finaux de f, le calculateur calcule f' le chemin le plus court entre f_- et f_+ ne contenant pas de sommets saturés (donc l'alternative à f).
  - Si $f_- = f_+$, l'agent continue son chemin selon l.

— Sinon si f' existe, le calculateur remplace f par f' dans l et élimine les cycles (s'il en existe), et l'agent suit le nouveau chemin c'.
— Sinon si α n'est pas saturée, l'agent s'engage quand même sur α et demandera une alternative au compilateur à l'intersection suivante.
— Sinon, l'agent attend un temps δt et le calculateur réessaie de trouver une alternative.

L'algorithme BallString initial, décrit dans 5, a pour but de trouver rapidement une alternative à un chemin dans un graphe dynamique dont les arêtes ne vont pas seulement de τ à $+\infty$ mais peuvent avoir plusieurs valeurs en fonction du nombre d'agents les parcourant. Celui-ci consiste en un recalcul du chemin uniquement autour des arêtes dont la pondération change, en cherchant à passer par les arêtes avoisinantes dont la pondération a le plus décru (ou le moins cru). Puisque dans cette étude, la pondération est simplifiée, cette dernière information n'est pas disponible ; une simple recherche par A^* est donc faite sur les arêtes non saturées.

Schématisation de l'algorithme BallString simplifié.



Explication : (i) L'agent s'engage sur une route pour rejoindre le point en haut à droite ; (ii) Deux arêtes de celui-ci deviennent saturées ; (iii) Le calculateur trouve un nouveau sous-chemin sans arête saturée, en violet ; (iv) Une arête est saturée en aval, mais aucune alternative n'a été trouvée par le calculateur, donc l'agent poursuit son chemin ; (v) L'agent ne peut plus avancer, donc attend.

4. Étude de complexité

On remarque que pour chacun de ces algorithmes, le nombre d'agents ne peut excéder la capacité totale du graphe. Par souci de simplification, la saturation (ou non) d'une arête n'affecte que les agents cherchant à s'engager sur celle-ci, pas ceux déjà dessus.

Sur les graphes que j'ai étudiés, j'ai trouvé en calculant pour de nombreuses paires de sommets choisies aléatoirement ($5n$ fois, pour n le nombre de sommets d'un graphe) un chemin les reliant par A^*, puis en saturant une arête aléatoire sur celui-ci et en recalculant, le coût de contournement (la distance supplémentaire de la plus courte route alternative entre deux points par rapport au chemin le plus court) vaut environ $e \approx 2,7$ arêtes.

4.1. Algorithme A^* itéré

Dans cette section, on s'intéresse à la complexité temporelle asymptotique en fonction de n de l'algorithme A^* itéré.

Pour simplifier (et ce qui est cohérent avec mon étude pratique), on considère que les agents sont équirépartis sur les arêtes, et que celles-ci ont toutes les mêmes capacités et longueurs. La probabilité qu'une arête soit saturée est donc $p = \frac{v}{Q}$.

On s'intéresse d'abord à un agent donné.

On remarque d'abord que la probabilité que la variable aléatoire Σ représentant le fait que la $i$ème arête sur laquelle l'agent s'engage soit saturée suit une loi géométrique de paramètre p. De ce fait, $\mathbb{E}(\Sigma) = 1/p$. (Soit dit en passant, pour que l'algorithme termine, on doit avoir $\mathbb{E}(\Sigma) > e$, donc $pe < 1$.)

De plus, pour un graphe de taille assez grande, la probabilité de ne pas trouver de chemin alternatif lors d'un recalcul est effectivement celle que toutes les arêtes sortantes du nœud sur lequel se trouve l'agent soient bloquées. On peut donc schématiser par une variable aléatoire $\mathrm{B} \rightsquigarrow \mathcal{G}(1-p^x)$ le nombre de recalculs que l'agent doit faire suite à un bloquage, d'espérance $\mathcal{E}(\mathrm{B}) = 1/(1-p^x)$.

À chaque recalcul réussi (donc en moyenne pour $\mathbb{E}(\mathrm{B})$ recalculs), la distance réduit donc de $\delta = e - \mathbb{E}(\Sigma)$. Pour une distance initiale de $L \in \mathcal{O}(\sqrt{n})$, on aboutit à la propriété suivante :

Propriété (coût de l'algorithme A^* itéré) : Lorsque $pe < 1$, le coût temporel asymptotique moyen de l'algorithme A^* itéré, à saturation fixée, est

$$\hat{c}_n \underset{n\to+\infty}{\sim} A \times \frac{L}{\delta} \times \mathbb{E}(\mathrm{B}) \underset{n\to+\infty}{\sim} \frac{AL}{(1-pe)(1-p^x)} = \mathcal{O}(n\sqrt{n}\log n)$$

soit pour v agents (avec donc $v = \mathcal{O}(n)$ puisque la saturation est fixée)

$$\widehat{C}_n \underset{n\to+\infty}{=} \mathcal{O}(n^2\sqrt{n}\log n)$$

De manière similaire, en fixant n et en faisant varier la saturation (donc p), pour un agent et pour $e > 1$, on a

$$\hat{c}_p \underset{p\to(1/e)_-}{=} \mathcal{O}(1/(1-pe))$$

donc

$$\widehat{C}_p \underset{p\to(1/e)_-}{=} \mathcal{O}(1/(1-pe))$$

Le fait que la saturation ne peut excéder $1/e$ fois la capacité du graphe se manifeste sur les résultats expérimentaux (voir figure 2), puisque dès 80% de saturation lorsque l'heure de départ des agents suit une courbe gaussienne (soit 32% de saturation maximale à un instant donné), l'exécution de l'algorithme ne termine pas en un temps raisonnable.

4.2. Algorithme BallString simplifié

Nous pouvons essayer d'effectuer une analyse similaire pour l'algorithme BallString simplifié.

Puisque l'on recalcule un chemin chaque fois que l'une des arêtes devient saturée, en notant $\sigma_i(j)$ l'évènement « la $j$ème arête est saturée lorsque l'agent cherche à s'engager sur la $i$ème arête » (et en supposant ceux-ci indépendants suivant i et j) , on a

— D'une part, la probabilité qu'au moins une arête dans le chemin prévu soit saturée lorsque l'agent cherche à s'engager sur la $i$ème arête est $\mathbb{P}(\bigcup_{k=i}^{n} \sigma_i(j)) = 1 - (1-p)^{n-i}$.
— D'autre part, la probabilité qu'aucune arête dans le chemin prévu (moins le chemin déjà fait) ne soit saturée lorsque celui-ci cherche à s'engager sur la $i$ème arête est
$\mathbb{P}(\bigcap_{k=0}^{i-1} \bigcap_{l=k}^{n} \sigma_k(l)) = \prod_{k=0}^{i-1}(1-p)^{n-k+1} = (1-p)^{-\frac{i^2}{2}+i(n+\frac{3}{2})}$.

On a donc, pour $i \in \{0; \dots; n-1\}$: $\mathbb{P}(\Sigma = i) = \left[1-(1-p)^{n-i}\right]\left[(1-p)^{-\frac{i^2}{2}+i(n+\frac{3}{2})}\right]$, et
$\mathbb{P}(\Sigma = n) = 1 - \sum_{i=1}^{n-1} \mathbb{P}(\Sigma = i)$. On remarque que ces probabilités sont bien définies et entre 0 et 1 ; cependant, mes études de la variable aléatoire Σ ont été infructueuses.

Quitte à perdre l'information de l'influence de p sur la complexité temporelle à n fixé, on sait tout de même que par [5], la complexité temporelle asymptotique de l'algorithme BallString est en pratique $\hat{c}_n \underset{n\to+\infty}{=} \mathcal{O}(n \log n)$ pour un agent. On peut donc conclure.

Propriété (coût de l'algorithme BallString simplifié) : Le coût temporel asymptotique moyen de l'algorithme Ballstring simplifié est

$$\widehat{C}_n \underset{n\to+\infty}{=} \mathcal{O}(n^2 \log n)$$

5. Analyse expérimentale

Chacun des points des courbes de ce rapport est la moyenne de 25 expériences avec les mêmes paramètres. Le graphe à 3391 arêtes est celui du premier arrondissement de Paris, et les autres graphes sont ceux de villages alpins et de villes de banlieue parisienne. Les variations sont assez négligeables pour ne pas figurer sur les graphes.

5.1. Influence de la taille du graphe

On rappelle qu'on a vu précédemment que l'algorithme A^* a une complexité asymptotique en $n^{5/2} \log n$, alors que l'algorithme Ballstring a une complexité asymptotique en $n^2 \log n$. L'analyse expérimentale effectuée en figure 1 semble confirmer ces complexités, puisque les courbes ont le profil attendu.

Les graphes routiers étudiés (de 231 à 18701 arêtes) correspondent à des graphes de communes entre 120 et 65000 habitants. En raison du comportement exponentiel de l'algorithme A^* itéré, il semble difficile à généraliser sur un graphe routier de l'envergure d'un pays (pour la France, `openstreetmap.org` recense 466 millions de nœuds) sans une capacité de calcul importante.

5.2. Influence de la saturation

On peut ensuite s'intéresser à l'influence de la saturation sur le graphe routier et les agents (figures 2 à 4, étude faite sur le graphe routier du premier arrondissement de Paris). On remarque tout d'abord qu'au-delà de 70% de saturation (donc 70$ de voirutes par rapport à la capacité totale du graphe) pour l'algorithme A^* itéré et 80% pour l'algorithme BallString, le programme ne semble pas terminer (figure 4, programmes arrêtés à 5 minutes d'exécution, contre un résultat attendu de l'ordre de la seconde), ce qui est cohérent (du moins pour A^*) avec l'étude faite plus haut. Les temps d'exécution sont aussi meilleurs pour BallString à haute saturation. Par ailleurs, A^* itéré renvoie des chemins légèrement plus courts que BallString, (ce qui est cohérent avec [5]), mais ce dernier laisse les voitures moins longtemps à l'arrêt, voire pas du tout sur les faibles saturations.

Figure 1 : Pour un graphe saturé à 50% :

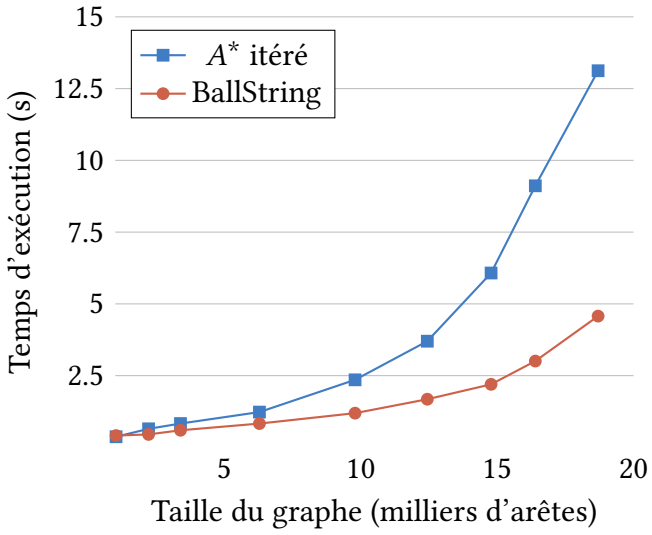


Figure 2 : Pour un graphe à 3391 arêtes :

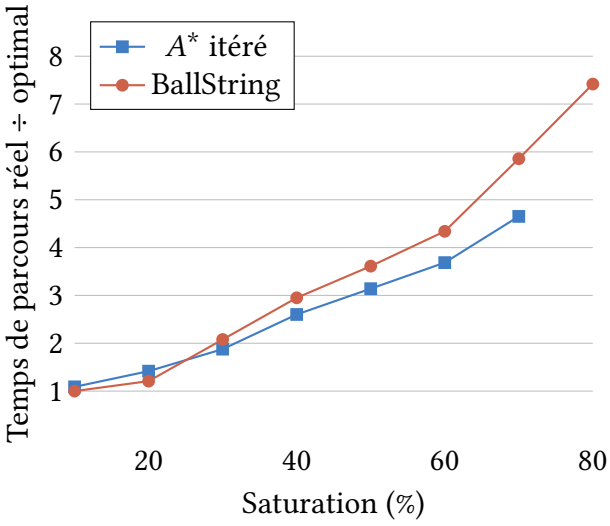


Figure 3 : Pour un graphe à 3391 arêtes :

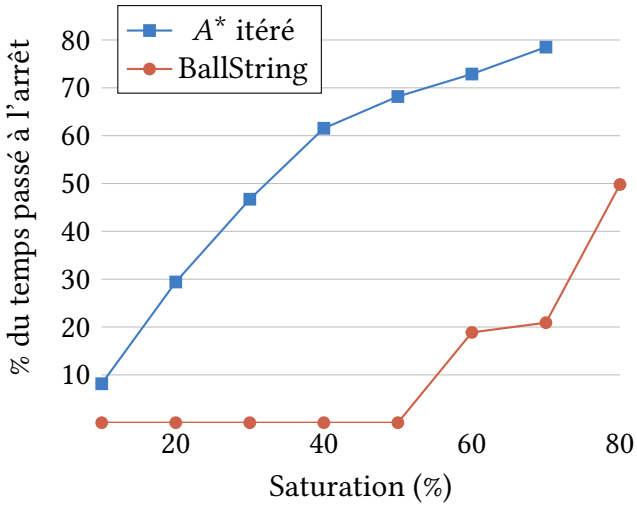
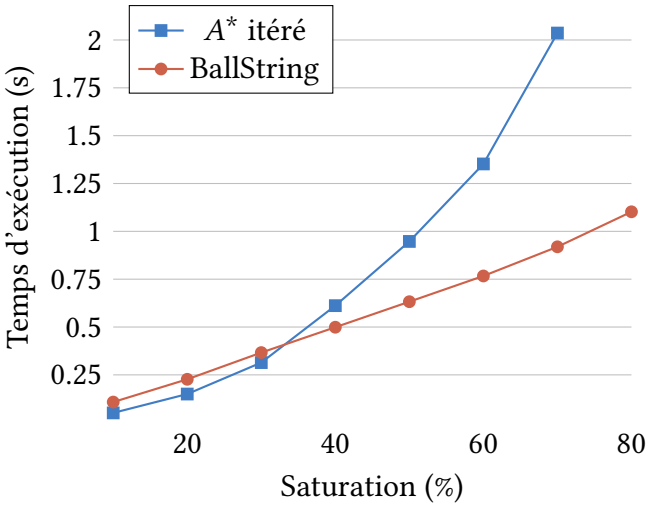


Figure 4 : Pour un graphe à 3391 arêtes :



Références

[1] Department of Engineering Science. *Fast Shortest Path Algorithms for Large Road Networks.* URL : `orsnz.org.nz/conf36/papers/Engineer.pdf`.

[2] Dimitris J. Bertsimas et Garrett Van Ryzin. *Stochastic and Dynamic Vehicle Routing with General Demand and Interarrival Time Distributions.* DOI : `doi.org/10.2307/1427801`.

[3] David Larsen. *The Dynamic Vehicle Routing Problem.* URL : `portal.issn.org/resource/ISSN/0909-3192`.

[4] Edward P.F. Chan et Yaya Yang. *Shortest Path Tree Computation in Dynamic Graphs.* DOI : `doi.org/10.1109/TC.2008.198`.

[5] Giacomo Nannicini et Leo Liberti. *Shortest Paths on Dynamic Graphs.* DOI : `doi.org/10.1111/j.1475-3995.2008.00649.x`.

[6] Melanie Smith et Roger Mailler. *The Effect of Congestion Frequency and Saturation on Coordinated Trac Routing.* DOI : `doi.org/10.1007/978-3-642-25044-6`.

Décompte de caractères (hors annexe) : 10002

6. Annexes

6.1. Lien entre $|A|$ et $|S|$ dans un graphe planaire connexe

Par définition, un graphe planaire connexe peut se représenter sur un plan infini sans qu'aucune arête n'en croise une autre, et deux sommets quelconques de ce graphe sont joignables par un chemin.

Définition : Lorsqu'un graphe planaire connexe est mis sous la forme précédente, il forme une partition du plan infini en $f \geqslant 1$ **faces**.

Cette définition va nous permettre de prouver le lemme suivant :

Lemme (formule d'Euler) : Dans un graphe planaire connexe à n sommets et m arêtes, on a $n - m + f = 2$.

Preuve par induction forte sur le nombre d'arêtes :

Il existe un unique graphe connexe à 0 arêtes, qui a 1 sommet et 1 face. On a bien $1 - 0 + 1 = 2$.

Supposons maintenant le résultat vrai pour tous les graphes à $m - 1$ arêtes, et soit G un graphe à m arêtes. Deux cas se présentent :

- G n'a pas de cycle. C'est donc un arbre, donc il n'a qu'une seule face et $m + 1$ sommets. On a donc bien la relation $m + 1 - m + 1 = 2$.
- G a (au moins) un cycle C. Notons $f \geqslant 2$ le nombre de faces de G. En supprimant une arête a de C, on fusionne les deux faces autour de a, et le graphe $\widetilde{G}$ a $m - 1$ arêtes et $f - 1$ faces (et reste connexe). Donc $\widetilde{G}$ vérifie la relation $n-(m-1)+(f-1) = 2$, donc on a bien $n-m+f = 2$ pour G.

Propriété : Tout graphe planaire connexe à $n \geqslant 3$ sommets et m arêtes vérifie $n - 1 \leqslant m \leqslant 3n - 6$. On a donc $m \in \mathcal{O}(n)$ et $n \in \mathcal{O}(m)$.

Preuve : La relation $n - 1 \leqslant m$ est en fait prouvable pour tout $n \geqslant 0$ et pour tout graphe connexe. Montrons-la par induction.

- Le seul graphe à 1 sommet a 0 arête et donc vérifie la relation.
- Si les graphes à $n - 1$ sommets vérifient la relation, soit un G graphe à n sommets et s un sommet quelconque de G tel que $G\backslash s$ est connexe (qui existe bien car soit il existe un sommet de degré 1 donc on prend celui-ci, soit tout sommet fait partie d'un cycle donc on peut en prendre un) est un graphe à $n-1$ sommets, et par connexité, s est relié au reste du graphe par au moins une arête α (qui n'appartient pas à $G\backslash s$), donc $G\backslash s$ a au moins une arête de moins que G. Donc G a au moins $(n - 1) - 1$ (hypothèse de récurrence appliquée à $G\backslash s$) +1 (l'arête α) arêtes, et la relation est vérifiée.

Pour la relation $m \leqslant 3n - 6$, utilisons la formule d'Euler, en ne considérant tous d'abord que les graphes planaires avec un nombre d'arêtes m maximal. Soit G un tel graphe pour $n \in \mathbb{N}$.

Tout d'abord, toutes ses faces intérieures sont triangulaires. En effet, si l'une d'entre elles avait 4 arêtes ou plus, le graphe formé par ajout d'une arête entre deux sommets non adjacents de cette face reste planaire, et donc G ne vérifiait pas initialement m maximal.

De plus, tous les sommets de G sont de degré au moins 2. En effet, si un sommet de degré 1 est à l'intérieur d'une face quelconque, on peut le relier par une arête à un autre sommet non déjà voisin de la face et le graphe reste planaire.

Ainsi, tous les sommets font partie d'un cycle, sinon il existerait un sommet de degré 1. Comme vu précédemment, un cycle partitionne le plan infini en 2, donc toute arête sépare 2 faces ; puisque

ces cycles sont des triangles, $2m = 3f$. On a donc, par la formule d'Euler, $n - m + 2m/3 = 2$ donc $m = 3n - 6$. Cela étant pour le cas maximal, on a bien, de manière générale, $n \leqslant 3m - 6$.

6.2. Algorithme A^* et caractéristiques

Voici le fonctionnement de l'algorithme A^* :

Entrée : un graphe pondéré $G = (S, A, \omega)$ avec d la distance entre deux points, une heuristique h, deux sommets *init* et *term*.
Sortie : un chemin de *init* à *term* (s'il en existe un).

```
fonction auxiliaire INSERTION_DIMINUTION(file, v, p) :
    Si v ∈ file alors file.priorité(v) = p
    Sinon, file.insertion(v, p)

fonction auxiliaire CHEMIN(parents, init, term) :
    res ← [term]
    actif ← term
    tant que actif ≠ init faire
        actif ← parents[actif]
        res ← [sommet; res]
    renvoyer res

fonction A* (G, init, term, h) :
    dist ← [+∞] * |S|
    dist[source] ← 0
    parents ← [−1] * |S|
    traitement ← file_priorité.création()
    traitement.insertion(init, h(init)
    tant que non(traitement.est_vide()) faire
        (actif, _) ← traitement.extraction()
        Si actif = term alors
            renvoyer CHEMIN(parents, init, term)
        sinon pour succ ∈ actif.voisins() faire
            d ← dist[actif] + d(actif, succ)
            Si d < dist[actif] alors
                parents[succ] ← actif
                dist[succ] ← d
                INSERTION_DIMINUTION(traitement, succ, d + h(succ))
    renvoyer []
```

Dans toute cette partie, pour un sommet *init*, on note $d(init, y)$ la distance minimale de *init* à y et $d^*(y)$ celle trouvée par l'algorithme A^*.

Définition : Pour un sommet *term* donné qu'on cherche à rejoindre depuis *init*, une **heuristique** de parcours $h(s)$ est une fonction approximant la distance de $s \in S$ à *term* dans le graphe. Elle est dite

- **admissible** si $\forall s \in S, h(s) \leqslant d(s, term)$
- **monotone** si pour tout arc $(s \to s')$, $h(s) \leqslant d(s, s') + h(s')$.

Ici, on a utilisé la distance géodésique comme heuristique pour A^*.

Propriété : L'algorithme A^* renvoie un chemin de poids minimal si h est admissible.

Preuve : Soient *init* et *term* deux sommets de G. Pour un sommet, on note $p^*(y) = d^*(y) + h(y)$.

Par l'absurde, soit l un chemin minimal et l^* le chemin (non minimal) renvoyé par A^*. On a donc $d^*(term) > d(init, term)$.
De plus, comme h est admissible, $h(term) \leqslant d(term, term) = 0$, donc $p^*(term) = d^*(term)$. Donc tout sommet y tel que $p^*(y) \leqslant p^*(term)$ a été extrait avant la fin de l'exécution.
En notant $l : init = y_0, \dots, y_k = term$, montrons par récurrence sur i que y_i a été extrait avant *term*, et que $d(init, y_i) \geqslant d^*(y_i)$.

- C'est vrai pour *init* puisque $d^*(init) = d(init, init) = 0$.
- Si la propriété est vraie pour y_i, alors par l'arête $y_i y_{i+1}$ et l'heuristique, on fixe $d^*(y_{i+1}) \leqslant d^*(y_i) + d(v_i, v_{i+1}) = d(v_0, v_{i+1})$ par hypothèse de récurrence. Donc $p^*(v_{i+1}) \leqslant d(v_0, v_{i+1}) + d(v_{i+1}, term) \leqslant d(init, term) \leqslant d^*(term)$ (car h est admissible). Donc y_{i+1} est bien extrait avant *term*.

Cela est vrai pout tout i, y compris $i = n$, donc *term* est strictement extrait avant *term*, ce qui est absurde.

Propriété : L'algorithme A^* a une complexité en $\mathcal{O}(n \log n)$ si l'heuristique est monotone.

Preuve : en d'autres termes, puisque la file de priorité est une structure avec une complexité d'insertion / suppression en $\mathcal{O}(n \log n)$, cela revient à prouver qu'un nœud donné n'est extrait qu'une fois de cette file.

Lorsqu'on extrait un sommet de la file, on a $p^*(y) = d^*(y) + h(y)$. Les sommets parmi ses voisins qui sont insérés vérifient $p^*(y') = d^*(y) + d(y, y') + h(y')$.
Mais comme h est cohérente, $d(y, y') + h(y') \geqslant h(y)$, donc $p^*(y') \geqslant p^*(v)$. Donc p^* est croissante au fil du temps en ne considérant que les sommets extraits de la file. Donc pour y'' un prédécesseur de y extrait ultérieurement, on aura $p^*(y'') = d^*(y'') + h(y'') \geqslant p^*(y) = d^*(y) + h(y)$ donc comme par monotonie, $h(y'') \leqslant d(y'', y) + h(y)$, on a $d^*(y'') + d(y'', y) \geqslant d^*(y)$, donc y n'est pas réinséré. Ainsi, A^* a une complexité en $\mathcal{O}(n \log n)$.