RECHERCHE DE PLUS COURT CHEMIN DANS UN GRAPHE EN APPRENTISSAGE PAR RENFORCEMENT

Gabriel St-Pierre
Maîtrise en informatique (I.A.)
Université Laval
111 178 727

Léo Backert
Maîtrise en informatique (I.A.)
Université Laval
536 916 167

Romain Chanteloup
Maîtrise en informatique (I.A.)
Université Laval
536 916 219

17 décembre 2021

ABSTRACT

La recherche du plus court chemin (*shortest path*) est au cœur de nombreux algorithmes et applications de graphes. Habituellement, les algorithmes Dijkstra et Bellman-Ford sont utilisés dans ce type de résolution de problème. Dans le présent article, nous tentons d'implémenter et d'entraîner un modèle d'apprentissage par renforcement qui serait capable de trouver le plus court chemin entre deux sommets dans un graphe dans le contexte d'optimisation de trajet en voiture. Notre contribution consiste en l'apprentissage sur un graphe à connexions aléatoires, c'est-à-dire pour lequel l'agent ne connaît pas d'avance les propriétés du graph (nombre de noeud, de connexions, valeurs, etc.). Cette propriété rend l'utilisation de méthodes à actions discrètes impossibles à utiliser. Nous tentons donc d'implémenter un agent asynchrone avec l'algorithme *n-step Q-Learning* afin d'accomplir la tâche.

1 Introduction

Trouver le plus court chemin entre deux nœuds dans un graphe est critique au sein de nombreux domaines tel que la télécommunication ou encore la planification de trajets. Étant donné un graphe et des coûts sur chaque arête (transition), le problème consiste à minimiser la somme des coûts du chemin emprunté pour se rendre d'un nœud A vers un nœud B. Il existe déjà des algorithmes de complexité polynomiales permettant de résoudre ce problème, dont particulièrement ceux de Bellman-Ford [1] et de Dijkstra [2]. Cependant, nous sommes tout de même en droit de se poser la question suivante : est-il possible de faire mieux avec un algorithme d'apprentissage par renforcement profond (Deep-RL)? Pour ce faire, nous allons tenter l'implémentation d'une version asynchrone de *n-step Q-Learning* et évaluer les résultats sur différentes tailles de trajectoires optimales.

1.1 Revue de la littérature

Vikram Waradpande et al. [3] proposent différentes approches pour la représentation d'un graph. Ces représentations permettent de condenser les informations des noeuds, connexions et trajets du graph en très peux de valeurs, ce qui pourrait être utilisé comme entrée pour un réseau de neuronnes. Par contre, les expériences montrées se résument à l'utilisation d'un environnement de déplacement en grille 2D, ce qui ne correspond pas au notre et serait donc difficile à intégrer.

Plusieurs membres de Google DeepMind et du MILA [4] ont proposé des approches d'entraînement asynchrones afin d'accélérer les temps d'entraînement. Ils proposent entre-autre une version asynchrone de *n-step Q-Learning*. Il a été démontré que, contrairement à *DQN* [5], un replay-buffer n'était pas nécéssaire puisque celui-ci est simulé par les mises à jour asynchrones provenant des différents *threads*. On y propose également différentes stratégies et paramètres afin de favoriser la convergence, qui seront fortement utilisées dans le présent projet.

Finalement, Adam Stooke et Pieter Abbeel [6] donnent différents conseils quand aux meilleures pratiques en entraînement asynchrone. On y discutte notamment de l'utilisation de copies locales temporaires des modèles versus une seule

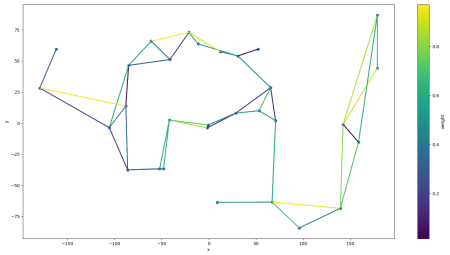

Figure 1: Visualisation d'un graphe généré par l'environnement.

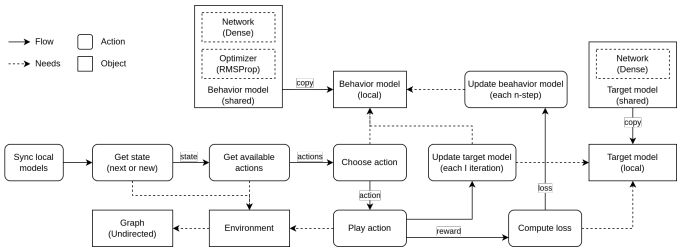

Figure 2: Étapes principales de l'agorithme d'apprentissage.

instance globale, ou encore de l'utilisation des GPUs pour accélérer certaines opérations. Dans le cadre du présent projet, ces techniques ont été notées mais non implémentées en raison de leurs complexitées d'implémentation.

2 Approche proposée

2.1 Environnement

En raison de la complexité de la tâche à accomplir, un gestionnaire d'environnement sur mesure a dû être implémenté. Tout d'abord, l'environnement est représenté par un graphe, contenant une disposition de noeuds et d'arêtes aléatoires. Les noeuds sont représenté par des coordonnées en 2 dimensions. Les arêtes sont non-dirigées et possèdent des poids allant de 0 à 1. Dans le contexte de planification routière, les noeuds corresponderaient aux coordonnées géographiques des intersections, et les arêtes aux routes les liant, avec les poids représentant un estimé du temps de parcours (sans unité) du tronçon de route. Avec une collecte de données GPS des utilisateurs de la route, il serait facile de déterminer les valeurs de ces poids et de les ajuster en temps-réel. Le graphe peut être généré aléatoirement selon une configuration donnée en minimisant les croisements d'arêtes, ou encore selon une liste d'arêtes prédéterminées. La figure 1 montre un exemple de graphe généré par l'environnement.

Ainsi, à chaque épisode, un nouveau trajet optimal est déterminé aléatoirement à partir du graphe en utilisant un algorithme déterministe comme Dikjstra [2]. L'état est représenté par le noeud courant et le noeud de destination, et l'action est le prochain noeud à choisir. Afin d'uniformiser les entraînement, les valeurs des coordonnées des noeuds sont normalisées selon l'approche *Z-Score normalization* [7]. De plus, afin d'aider à l'entraînement, il sera impossible pour l'agent de revenir sur son dernier pas.

2.2 Algorithme

Une variante de l'algorithme *S2* de l'article A2C [4], basée sur une version asynchrone de *n-step Q-Learning*, a été développée afin de se conformer à l'environnement utilisé. À chaque *n-step*, chaque *thread* synchronise une copie des modèles *behavior* et *target* globaux, puis parcours le graphe et accumulant des erreurs selon les valeurs retournées par les modèles. Les poids des modèles et les valeurs d'epsilon partagés sont ensuite mises à jour. L'algorithme 1 (annexe) montre la description complète des étapes, et la figure 2 montre une illustration de ces étapes et des dépendances entre les différents modules requis pour l'entraînement.

3 Expérimentation

Les paramètres d'entraînement *behavior* sont basés sur ceux de l'article A2C [4], c'est-à-dire des réseaux de neuronnes pleinement connectés (3 couches de 64 neuronnes), un otpimiseur RMSProp ($lr = 0.01, \alpha = 0.99$), $\gamma = 0.99$, $\tau = 0.5$, $n = 5$ et $T_{target} = 5$. Le modèle a été entraîné sur un graphe de 20 noeuds, avec des coordonnées $x \in [-180, 180]$ et $y \in [-90, 90]$ pendant 3000 itérations sur 8 *threads*. Les valeurs de ϵ_1, ϵ_2 et ϵ_3 commencent à 1, puis diminuent d'un facteur 0.05, 0.01 et 0.0015 jusqu'aux valeurs 0.1, 0.01 et 0.5, avec des probabilité de sélection de 0.4, 0.3 et 0.3, le tout respectivement. Cela permet de conserver des degrés d'exploration prédéfinis et diminue l'importance des facteurs de diminution puisqu'une exploration importante ($\epsilon = 0.5$) reste toujours possible.

Plusieurs fonctions de récompense ont été testées, et ont menées à des résultats intéressants qui sont discuttés ci-bas.

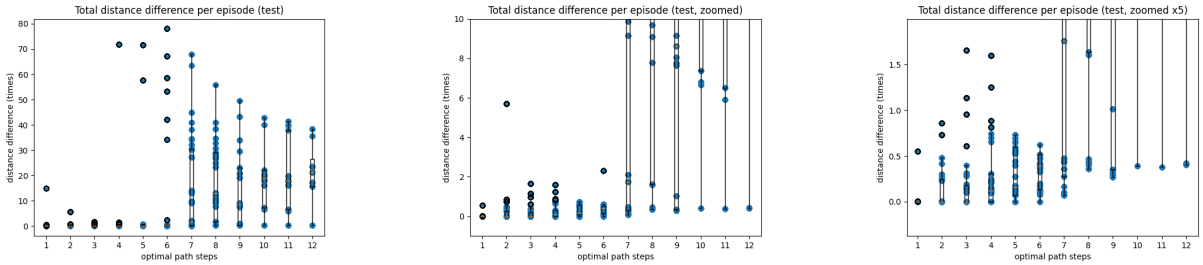

Figure 3: Résultats de test. L'axe vertical montre le % de différence entre la distance trouvée et celle optimale. Différentes échelles sont montrées afin de voir la vue d'ensemble ainsi que les détails fins. Une valeur de 0 indique que le chemin optimal a été trouvé, alors qu'une valeur de 1 indique que le chemin trouvé est 2 fois plus long (1x de plus que le chemin optimal).

4 Résultats

Les résultats d'entraînement présentés à la figure 4 (annexe) montrent que l'agent semble converger lentement, avec une variance importante au niveau de la *loss*.

Les résultats de test présentés à la figure 3 montrent que bien que l'algorithme est loin d'être optimal, il est tout de même capable d'apprendre certaines trajectoires. La grande majorité des trajets comportant 1 à 3 transitions sont optimalement trouvés, alors que les plus courts chemins pour les trajets de 4 à 5 transitions sont relativement souvent trouvés (ou presque). à partir de 6 noeuds, l'agent trouve seulement quelques-uns des trajets optimaux alors qu'à plus de 7 noeuds, l'algorithme se comporte aléatoirement dans le graphe. Ce manque de convergence rend malheureusement la poursuite de tests additionels (résistance aux mutations d'arêtes et de noeuds du graphe, comparaison de performance avec Dijkstra, portabilité du modèle entrainé, etc.) impossibles pour le moment.

5 Discussion

5.1 Difficultés techniques

En raison d'une difficulté d'implémentation avec la librarie `PyTorch` [8], la perte est calculée avec le modèle partagé θ au lieu du modèle synchronisé local θ'. Ainsi, les prédictions et la perte peuvent différer de quelques mises à jour de θ entre temps, ce qui n'est pas idéal afin de converger optimalement. Cette problématique vient également ralentir les temps d'entraînement en raison des attentes dûes aux verrous de synchronisation inter-processus. Afin de résoudre le problème, il faudrait que chaque *thread* fige les poids locaux du modèle et calcule les pertes en gardant l'optimiseur global en référence, ce qui semble être difficilement réalisable avec la librairie utilisée.

De plus, les ressources limitées ne permettant pas d'effectuer des expériences à grande échelle, les résultats ont été basés seulement sur de petits graphes.

5.2 Fonction de récompense

Le choix de la fonction de récompense donne lieu à de nombreuses variations au niveau des résultats. Tout d'abord, puisque le graphe est pondéré (c'est-à-dire que les connexions ont des poids qui diffèrent), l'utilisation d'une reward fixe à chaque pas de temps n'est pas envisageable. En effet, il s'agirait d'une technique intéressante pour les environnements en grille, ou chaque transition serait considéré comme une perte. Dans le présent cas, un trajet peu être considéré plus court même s'il demande plus de transitions qu'un autre. D'après nos essais, l'utilisation d'une reward négative fixe à chaque pas de temps (-0.01, -0.1, -0.5 ou -1) empirait les résultats.

Ensuite, nous avons tenté de miniser l'utilisation du trajet de vérité (trouvé avec un algorithme comme Dijkstra) en fournissant une reward fixe positive lors de l'atteinte du noeud final. Par contre, une telle technique engageait plutôt l'agent à trouver *un* chemin viable le plus vite possible, mais qui n'était probablement pas le meilleur. Ainsi, l'agent convergeait vers des solutions sous-optimales.

Finalement, la fonction de récompense utilisée est de retourner le quotient entre le trajet optimal et celui trouvé ($\in [0, 1]$), et 0 sinon. Nous avons décidé de ne pas utiliser de récompenses négatives lorsque le trajet n'était pas trouvé après un

certain temps car celles-ci pouvaient mener à une contre-indication des valeurs Q réelles (ex: lorsque les derniers pas de temps sont sur le trajet optimal, à quelques transitions du noeud cible).

5.3 Approches futures

La première approche à essayer serait l'utilisation d'un réseau LSTM [9] afin d'encoder la trajectoire parcourue. Il a été démontré que l'utilisation d'un tel encodage aide à la découverte de relations de longues durées, où un événement lointain du passé (ex: premières transitions dans le graphe) pourrait être fortement corrélé aux décisions qui doivent être prises dans le futur (ex: arriver à la cible en minimisant la distance totale) [10]. Par contre, une telle approche demande un entraînement beaucoup plus assidu, ce qui n'était pas possible selon les resources allouées au présent projet.

De plus, les approches *Actor-critic* [11] ont été vivement recherchées et sont souvent appliquées de nos jours dans les problèmes de trajectoires. Par contre, le fait que nos actions ne sont pas globalement fixées rend l'intégration difficile. Nous avons tenté une adaptation à une méthode basée sur les plus proches voisins [12], en choisissant l'action permise (prochain noeud) la plus près de celle prédite par le modèle. Cette méthode n'a malheureusement pas été explorée davantage, mais nous désirons la considérer pour le futur. Le même article propose une approche stochastique basée sur le même principe, mais où la sortie du modèle consiste en une moyenne et écart-type qui sont ensuite utilisés pour la génération aléatoire de l'action choisie.

Finalement, l'intégration à une librarire de gestion d'environnement comme [13] ou encore d'entraînement avec des modèles standards comme [14] serait fortement envisageable. Encore une fois, le fait que les actions ne soient pas fixes rend la tâche complèxe, mais tout de même probablement possible. L'utilisation de libraries existantes permettrait de limiter les erreurs d'implémentation.

6 Conclusion

Nous avons tenté une implémentation asynchrone de l'algorithme *n-step Q-Learning* pour la recherche de plus court chemin sur des graphes non-orientés. Notre approche fonctionen bien sur des petits graphes pour de courts trajets, mais peine à *scaler* sur des trajets de tailles réalistes. Or, plusieurs approches futures ont été mentionnées afin de possiblement améliorer l'agent développé. Pour le moment, nous jugeons qu'une approche algorithmique tel que Dijkstra fonctionne mieux.

References

[1] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] V. Waradpande, D. Kudenko, and M. Khosla, "Deep reinforcement learning with graph-based state representations," *CoRR*, vol. abs/2004.13965, 2020.

[4] V. Mnih, A. P. Badia, and M. e. a. Mirza, "Asynchronous methods for deep reinforcement learning," *arXiv preprint arXiv:1602.01783v2*, 2016.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[6] A. Stooke and P. Abbeel, "Accelerated methods for deep reinforcement learning," *CoRR*, vol. abs/1803.02811, 2018.

[7] Google, "Data preparation and feature engineering for machine learning : Normalization." `https://developers.google.com/machine-learning/data-prep/transform/normalization#z-score`, 2021. Accessed : 2021-11-12.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[9] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[10] B. Bakker, "Reinforcement learning with long short-term memory," in *Advances in Neural Information Processing Systems* (T. Dietterich, S. Becker, and Z. Ghahramani, eds.), vol. 14, MIT Press, 2002.

[11] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 2000.

[12] G. Dulac-Arnold, R. Evans, P. Sunehag, and B. Coppin, "Reinforcement learning in large discrete action spaces," *CoRR*, vol. abs/1512.07679, 2015.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[14] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

A Algorithme d'entraînement

Algorithm 1 Q-Learning n-step asynchrone adapté aux graphes

Sois les modèles θ et θ^- partagés entre les threads
Sois les paramètres partagés ϵ_1, ϵ_2 et ϵ_3, utilisés pour la politique ϵ-greedy
Soit T et e, le nombre total de trajectoires et d'épisodes pour tous les threads
Soit n le nombre de steps
Synchroniser le modèle $\theta' \leftarrow \theta$
while $e < e_{max}$ **do**
  Initialiser sous-compteur local $t \leftarrow 0$
  Initialiser le nouvel état s_t
  $t_{start} \leftarrow T$
  while s_t non terminal **do**
    Trouver les actions permises A selon l'état s_t
    Choisir l'action a_t basée sur la politique ϵ-greedy $Q(s_t, a; \theta') \quad \forall a \in A$
    Observer la récompense r_t et le prochain état s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
    if s_t terminal ou $t == n$ **then**
      $$R = \begin{cases} 0 & \text{si } s_t \text{ terminal} \\ \max_a Q(s_t, a; \theta^-) & \text{si } s_t \text{ non terminal} \end{cases}$$
      for **do**$i \in \{T ... t_{start}\}$
        $R \leftarrow \gamma R + r_i$
        $loss \leftarrow loss + (R - Q(s_t, a_i; \theta'))^2$
      end for
      $\theta \leftarrow \frac{d\ loss}{d\theta}$ (Update du modèle *behavior* θ)
      $\theta' \leftarrow \theta$ (Synchronizer le modèle *behavior* local)
      $t = 0$
    end if
    if $T \mod T_{target} == 0$ **then**
      $\theta' \leftarrow \tau\theta$ (Soft-update du modèle *target* θ^-)
    end if
    $s_t \leftarrow s_{t+1}$
  end while
end while

B Résultats d'entraînement

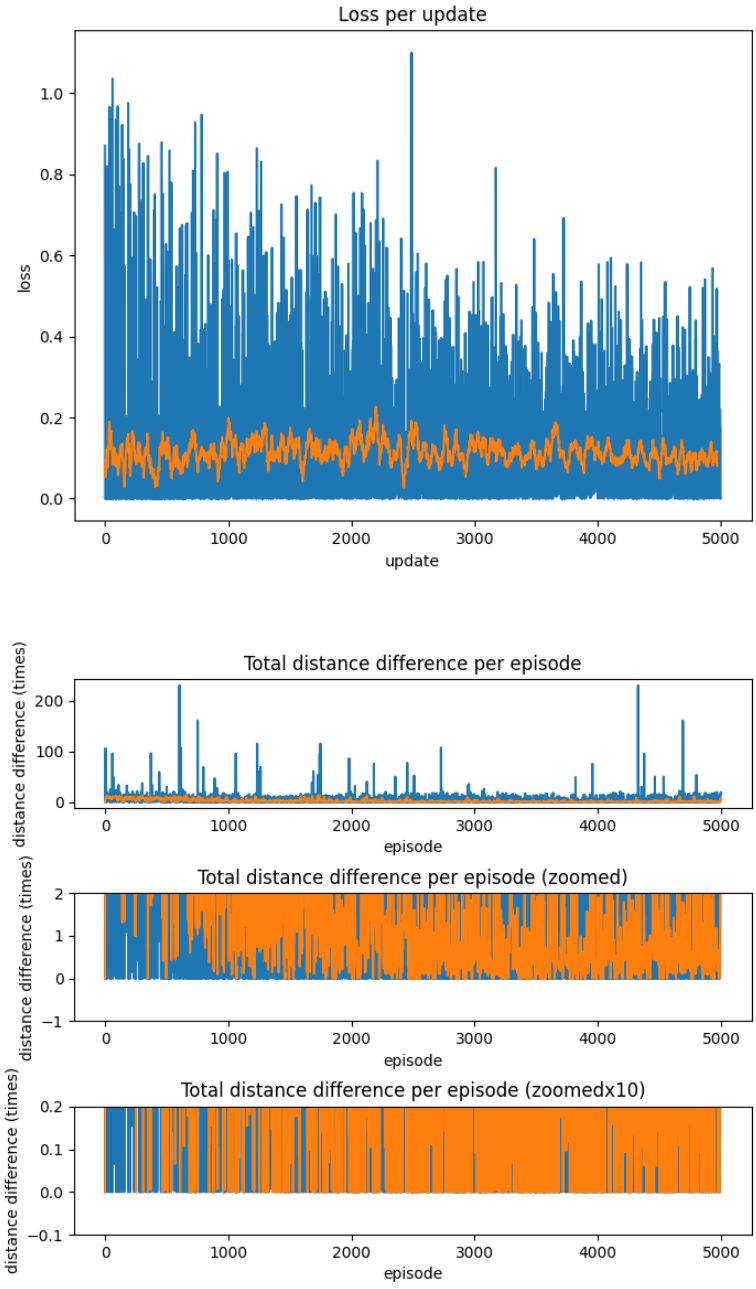

Figure 4: Résultats d'entraînement. La courbe orange est le résultat de l'application d'un filtre médian sur la courbe bleue originale.