

Rapport de projet INFO-F308

Hugo CALLENS, Rayan CONTULIANO BRAVO, Ziyad HALTOUT RHOUNI, Manu MATHEY-PREVOT, Moussa TALLIH

Résumé—Supposons que vous êtes étudiant à l'Université, et que vous avez besoin de trouver votre salle de classe dans un bâtiment que vous ne connaissez pas. Comment faire pour la trouver rapidement et sans vous perdre ? L'objectif de ce projet est d'utiliser la théorie des graphes afin de trouver le plus court chemin entre votre position actuelle et votre salle de classe en utilisant les algorithmes de recherche de plus court chemin. Plus besoin de jouer au détective pour trouver votre salle de classe grâce à l'application de la théorie de graphes.

I. INTRODUCTION

Le problème du plus court chemin dans un graphe est un problème fondamental en informatique et en mathématiques appliquées. Il trouve des applications dans divers domaines tels que la planification des réseaux de transport, la conception de circuits électroniques, la modélisation des réseaux sociaux, etc. La résolution efficace de ce problème est cruciale dans de nombreux contextes pratiques, car elle permet de trouver le chemin le plus court entre deux points dans un réseau, ce qui peut conduire à une optimisation des ressources, une réduction des coûts ou une meilleure planification des trajets. Ainsi, l'étude et le développement d'algorithmes efficaces pour résoudre le problème du plus court chemin revêtent une grande importance tant sur le plan théorique que pratique.

II. ETAT DE L'ART

Cette section vise à présenter un aperçu de l'état actuel des solutions proposées pour le problème du plus court chemin dans un graphe. Les travaux antérieurs dans ce domaine ont produit diverses approches et algorithmes visant à résoudre ce problème de manière efficace. Parmi les méthodes les plus couramment utilisées, on trouve l'algorithme de Dijkstra [1], l'algorithme de Bellman-Ford [2], l'algorithme de Floyd-Warshall [3], et les algorithmes basés sur les tas binaires tels que l'algorithme A* [4]. Ces algorithmes ont été largement étudiés et utilisés dans de nombreuses applications pratiques en raison de leur efficacité et de leur polyvalence.

Notamment, l'algorithme de Dijkstra est largement utilisé pour résoudre le problème du plus court chemin dans les graphes pondérés non négatifs, tandis que l'algorithme de Bellman-Ford est plus adapté aux graphes avec des poids négatifs. L'algorithme de Floyd-Warshall, quant à lui, est efficace pour calculer les plus courts chemins entre toutes les paires de sommets dans un graphe, même avec des poids négatifs. Enfin, l'algorithme

Superviseur : Valérie GILCHRIST

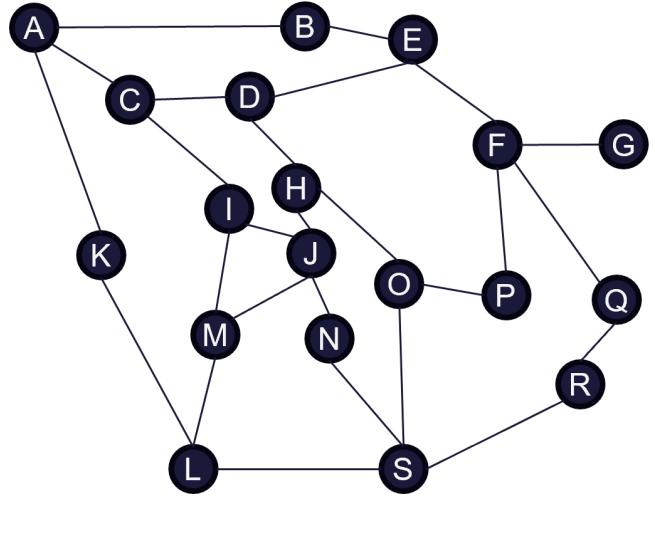
A* combine la recherche heuristique avec la recherche de coût minimal, ce qui en fait une méthode efficace pour trouver des chemins optimaux dans des graphes de grande taille.

Cependant, malgré les avancées réalisées dans ce domaine, des défis persistent, notamment dans le cas de graphes massifs ou dynamiques. De plus, l'amélioration continue des algorithmes existants et le développement de nouvelles approches restent des sujets de recherche actifs dans la résolution du problème du plus court chemin dans les graphes.

III. MÉTHODOLOGIE

A. Crédit des graphes

Un graphe est une structure mathématique et informatique composée d'un ensemble de points, appelés noeuds ou sommets, et d'un ensemble de lignes, appelées arêtes ou arcs, reliant ces points. Cette représentation permet d'agencer et de visualiser efficacement des données complexes.



● = noeud

— = arête

FIGURE 1. Exemple de graphe

La présence d'une arête entre deux noeuds d'un graphe indique l'existence d'une relation ou d'un lien entre ces

nœuds. Cette relation peut être de nature diverse, telle qu'une route, une connexion internet, une relation entre deux personnes, etc., et dépend du contexte dans lequel le graphe est utilisé.

Il est possible d'associer un poids à chaque arête d'un graphe. Ce poids est généralement une valeur numérique qui représente de manière abstraite la force, la distance, le coût ou la durée de la relation entre les nœuds reliés par l'arête. Par exemple, dans un graphe représentant une infrastructure réseau, le poids sur les arêtes peut représenter le temps de latence entre deux routeurs. Dans un graphe représentant une carte routière, le poids sur les arêtes peut représenter la distance à parcourir entre deux points.

Trouver le chemin le plus court entre deux nœuds d'un graphe est un problème fondamental en théorie des graphes. Dans ce contexte, le chemin le plus court est défini comme étant le chemin qui a le plus petit poids total. Le poids total d'un chemin est calculé comme la somme des poids de toutes les arêtes du chemin, comme indiqué dans l'équation suivante :

$$\text{Poids total} = \sum_{i \in n} \text{Poids de l'arête } i \quad (1)$$

où n est l'ensemble des arêtes du chemin entre deux nœuds.

Dans ce projet, la théorie des graphes est utilisée pour trouver le plus court chemin entre deux points d'intérêt dans un campus universitaire. Les nœuds du graphe représentent des points clés à l'intérieur ou à l'extérieur des bâtiments, et les arêtes représentent les chemins qui les relient, leur modélisation sera détaillée dans la section suivante. Les poids sur les arêtes quand à eux représentent la distance à parcourir entre ces points clés.

1) Modélisation d'un bâtiment: Dans le contexte de ce projet, un ensemble de données composé de plans en vue aérienne de l'ensemble des bâtiments situés sur le campus du Solbosch de l'Université Libre de Bruxelles a été utilisé.

Pour modéliser un bâtiment, chaque pièce a été représentée par un nœud du graphe. Les arêtes du graphe correspondent aux portes et chemins reliant les pièces entre elles. En raison de la qualité insuffisante de l'ensemble de données, l'échelle du plan n'était pas bien définie ou même inexistante. Par conséquent, il a été décidé de considérer que la distance entre chaque pièce était égale à 1. Ainsi, un chemin entre deux pièces correspond simplement au nombre de nœuds à traverser pour passer de l'une à l'autre.

Cependant, cette modélisation présente une limite : la distance réelle entre deux pièces n'est pas prise en compte. Toutefois, cette modélisation simplifiée est suffisante pour les besoins du projet, car il suffit de placer des nœuds à des distances approximativement égales (voir Figure 2).

Afin de modéliser le graphe d'un bâtiment le plus clairement possible, nous avons décidé d'associer un *type* à chaque nœud. Les types possibles sont les suivants :

- *Class* : une salle de classe, auditoire, etc. Généralement l'endroit recherché par les étudiants.

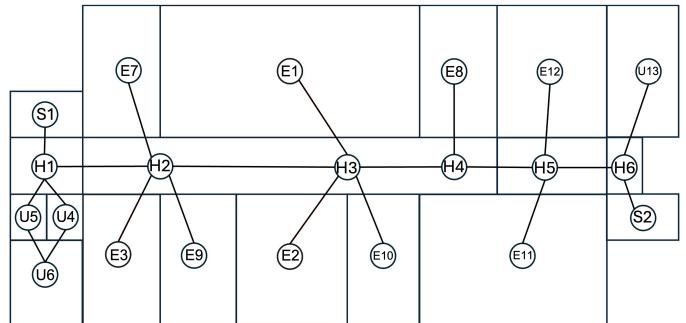


FIGURE 2. Exemple de modélisation d'un bâtiment en graphe

```

1   {
2     "N° étage": [
3       {
4         "id": "",
5         "name": "",
6         "neighbors": [
7           {
8             "id": "",
9             "direction": {
10               "id_pred": ""
11             }
12           }
13         ]
14       },
15     ]
16   }
17 }
```

Listing 1: JSON d'un Bâtiment

- *Hallway* : un couloir, il permet de se déplacer d'un nœud à un autre.
- *Stairs* : un escalier, permet de monter ou descendre d'un étage.
- *Elevator* : un ascenseur, permet de monter ou descendre d'un étage. Similaire à un escalier sauf que certaines personnes sont obligées de l'utiliser.
- *Entrance* : une entrée/sortie d'un bâtiment.
- *Unknown* : un nœud dont le type est inconnu.

Ils nous permettent de donner des spécificités à des nœuds, comme par exemple le fait que les escaliers et ascenseurs sont à tous les étages, qu'il puisse y avoir plusieurs entrées pour un auditoire, etc ...

a) Modélisation en JSON: Maintenant que la logique de modélisation des bâtiments est claire, nous pouvons réellement le modéliser en utilisant la syntaxe **JSON** suivante :

Chaque étages aura une liste de nœuds, ces nœuds auront tous comme attributs :

- *id* : un identifiant unique pour chaque nœud. Exemple : E123
- *name* : le nom de la salle ou du couloir. Exemple :

Auditoire Janson

- *neighbors* : une liste de voisins directs du noeud identifiés par leurs nœuds.

Cette liste de voisins permet de déterminer les chemins possibles, ainsi que la direction à prendre à partir d'un nœud donné.

- *id* : l'identifiant du voisin. Exemple : E124
- *direction* : est un objet qui va contenir chaque prédécesseur possible du noeud dans l'***id_pred*** et la direction à prendre à partir de celui-ci jusqu'au voisin en passant par le nœud actuel.

La direction est très importante. Étant donné qu'il est difficile d'obtenir la position de l'utilisateur dans le bâtiment à un étage *e* et de le diriger en conséquence, nous optons pour cette méthode qui est plus simple à mettre en œuvre.

2) *Modélisation du campus*: Pour la modélisation d'un campus universitaire, une sélection judicieuse de points est effectuée, correspondant aux nœuds d'un graphe. Ces points peuvent représenter des bâtiments, des points d'intérêt, des intersections, etc. Les arêtes du graphe représentent les chemins reliant ces points. Les poids sur les arêtes correspondent à la distance en mètres entre ces points.

Pour construire le graphe du campus, des points ont été placés sur une carte du campus à l'aide de l'outil en ligne Map Marker [5]. Une fois tous les points placés, les données de latitude et de longitude ont été extraites et les voisins directs de chaque point ont été déterminés. Ces informations ont ensuite été traitées pour construire le graphe du campus.



FIGURE 3. Exemple de modélisation d'un campus sur le site Map Marker avec les entrées des bâtiments en rouge et les autres en vert

a) *Modélisation en JSON*: Similairement à la modélisation d'un bâtiment, un campus aura une liste de nœuds, ces nœuds auront tous comme attributs

- *id* : un identifiant unique pour chaque nœud. Exemple : *c9* si nœud ordinaire et *eJ_1* si c'est l'entrée d'un bâtiment.

```

1   "Campus": [
2     {
3       "id": "",
4       "latitude": "",
5       "longitude": "",
6       "neighbors": [
7         {
8           "id": "",
9           "weight": ""
10        }
11      ]
12    }
13  ],
14 ]
15 ]
16 }
```

Listing 2: JSON du campus

- *latitude* : Coordonnée de latitude du nœud sur la terre,
- *longitude* : Coordonnée de longitude du nœud sur la terre,
- *neighbors* : une liste de voisins directs du nœud identifiés par leur nœuds.

Cette liste de voisins nous permet de déterminer quels nœuds sont accessibles par d'autres ainsi que la distance entre ceux-ci.

Pour calculer les poids entre les nœuds, un script a été utilisé, faisant appel à la bibliothèque geopy [6] pour calculer la distance en mètres entre deux points géographiques.

3) *Recherche du plus court chemin*: Dorénavant, il est possible de créer ces graphes en utilisant les fichiers **JSON** ainsi que la librairie NetworkX [7]. Des algorithmes de recherche de plus court chemin peuvent maintenant être utilisés pour trouver le chemin le plus court entre deux points de ces graphes.

Pour ce faire, l'algorithme de Dijkstra a été implémenté. Cet algorithme utilise le concept de *file à priorité* afin de s'exécuter efficacement.

De manière simplifiée, l'algorithme de Dijkstra fonctionne de la manière suivante : Étant donné une **pile** de nœuds non visités du graphe où la priorité de chaque nœud est initialisé à $+\infty$. Un nœud de départ est sélectionné et sa priorité est fixée à 0 dans la pile. Tous les voisins de ce nœud sont ensuite visités, car il possède la priorité la plus basse $0 \leq +\infty$, et pour chacun d'eux, leur priorité dans la pile est mise à jour en fonction de la distance entre le nœud actuel et le voisin **si et seulement si** la priorité du nœud est plus grande que celle qui vient d'être calculée. Une fois tous les voisins visités, le nœud actuel (de départ) est retiré de la pile. Le prochain nœud est sélectionné en prenant celui qui a la priorité la plus basse. Le processus est répété en visitant tous les voisins de ce nœud et en mettant à jour leurs priorités. Ce processus est répété jusqu'à ce que

tous les nœuds du graphe aient été visités ou que le noeud d'arrivée ait été atteint.

Algorithm 1 Algorithme de Dijkstra

```

1: pile ← graphe.noeuds
2: for all noeud  $n$  dans graphe do
3:    $n.distance \leftarrow +\infty$ 
4: end for
5: départ ← pile[0]
6: départ.distance ← 0
7: while pile n'est pas vide do
8:   courant ← noeud avec la plus petite priorité dans
      la pile
9:   retire courant de la pile
10:  if courant = destination then
11:    return chemin trouvé
12:  end if
13:  for all voisin  $v$  de courant do
14:    nouvelle_distance ← courant.distance + dis-
       tance de courant à  $v$ 
15:    if nouvelle_distance <  $v.distance$  then
16:       $v.distance \leftarrow$  nouvelle_distance
17:    end if
18:  end for
19: end while

```

4) Implémentation d'une application: Afin d'appliquer le raisonnement des sections précédentes, une application web a été créée afin de faciliter l'utilisation du projet par n'importe qui.

a) Implémentation d'une API: Étant donné que le code est déjà écrit en Python, le développement d'une application moderne a été entrepris, cette application est conforme aux standards actuels des applications dynamiques. Elle se compose d'une interface de programmation applicative (API) renfermant la logique métier de l'application, conjointement à une application cliente qui communique avec cette API par le biais de requêtes HTTP. L'objectif est d'assurer la gestion des interactions avec l'utilisateur via un affichage optimal et faciliter l'utilisation de notre application par nos utilisateurs. Cette api contient deux points d'entrée qui sont les urls :

- */api/ask* : Ce point d'entrée requiert en paramètres une position géographique et un lieu spécifique au sein de l'université reconnu par l'application. En retour, il fournit le chemin le plus court entre les deux points, à condition que la position géographique se trouve à l'intérieur du campus universitaire.
- */api/ask_from_inside* : Ce point d'entrée requiert deux lieux spécifiques au sein de l'université reconnus par l'application en paramètres. Il fournit en retour le chemin le plus court entre ces deux points.

Il convient de noter que tout chemin renvoyé par l'API comprend les nœuds, les instructions détaillées pour la transition d'un nœud à un autre, ainsi que des représentations en images 3D pour illustrer ces instructions.

b) Implémentation de l'interface web: Par la suite, la création d'une interface graphique s'est avérée nécessaire. Il a donc été utile de développer une page principale permettant à l'utilisateur de choisir entre utiliser sa localisation, récupérée par son navigateur avec son consentement, ou spécifier directement le lieu de départ ainsi que celui de destination. Pour guider l'utilisateur, les instructions nécessaires sont affichées, accompagnées des images correspondantes, de manière séquentielle. L'utilisateur peut parcourir ces instructions en utilisant deux boutons : "Précédent" et "Suivant", jusqu'à atteindre la destination souhaitée.

IV. RÉSULTATS

En résultat, nous avons une application entièrement fonctionnelle qui répond parfaitement à la problématique. Nous allons donc dans cette section vous la présenter.

A. Expérience utilisateur

Nous allons exposer les étapes distinctes de notre application par lesquelles l'utilisateur procède pour localiser un local. Initialement, l'utilisateur est confronté à une décision : utiliser la fonction de localisation GPS ou spécifier manuellement sa position au sein du campus.



FIGURE 4. Lorsque l'utilisateur arrive sur l'application.

a) À partir de ma position: Si l'utilisateur décide de se localiser grâce à la fonction de localisation, l'application affiche le chemin le plus court sur une carte, reliant sa position actuelle au bâtiment où se trouve le lieu recherché, comme illustré dans la figure 5. Une fois arrivé dans le bâtiment demandé, il est guidé à l'intérieur du bâtiment à l'aide d'instructions, comme le montrent les figures 6 et 7.

b) À partir d'une classe: Si l'utilisateur préfère utiliser la localisation du local le plus proche, l'application le guidera dans le bâtiment jusqu'à la sortie ou jusqu'au lieu demandé s'il se trouve dans le même bâtiment, comme indiqué dans les figures 6 et 7. Lorsque l'utilisateur doit passer d'un bâtiment à un autre, l'application agira de la même manière que dans le cas où l'utilisateur sélectionne "À partir de ma position" dès qu'il quitte le bâtiment initial.

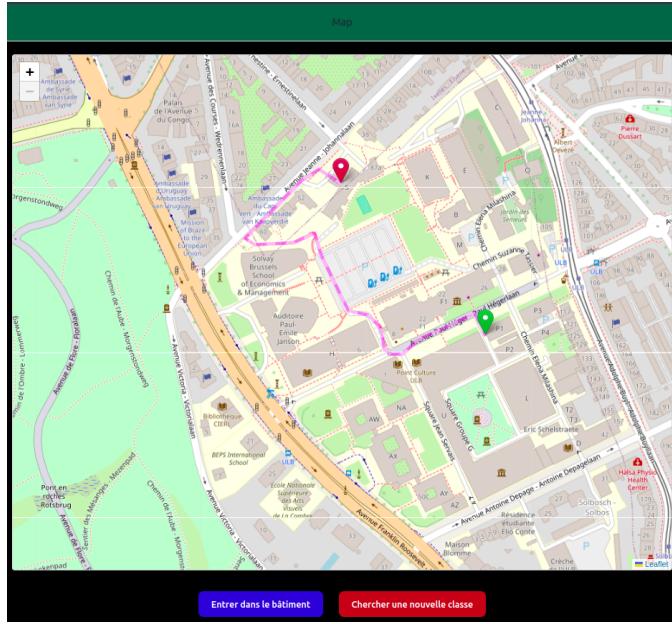


FIGURE 5. L'affichage d'une instruction.



FIGURE 6. L'affichage d'une instruction.



FIGURE 7. L'affichage de l'arrivée à l'endroit demandé.

V. CONCLUSION

Dans ce projet, la théorie des graphes a été appliquée dans le but de trouver le plus court chemin entre 2 points d'intérêt d'un campus universitaire. Pour ce faire, nous avons modélisé les graphes représentants le campus lui-même et les bâtiments du campus. Ce projet a permis de montrer une application de la théorie des graphes dans la résolution de problèmes du monde réel. Cependant, il existe encore des défis à relever, notamment dans le cas de graphes massifs ou dynamiques.

RÉFÉRENCES

- [1] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.
- [2] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1) :87–90, 1958.
- [3] Robert W. Floyd. Algorithm 97 : Shortest path. *Communications of the ACM*, 5(6) :345, 1962.
- [4] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2) :100–107, 1968.
- [5] Map Marker dev team. Map marker website.
- [6] GeoPy's dev team. Welcome to geopy's documentation !
- [7] NetworkX dev team. Networkx tutorial.