# TP2 - Graphes et *Pathfinding*

## PARTIE B

### Mise en contexte

Cette partie attaque l'application de votre bibliothèque de graphes et d'algorithmes de recherche de chemins. Le contexte est un jeu dans Unity. Il y aura aussi des ajouts à la bibliothèque, mais ceux-ci ne seront pas à propos de représentations et algorithmes de graphes.

### 1) Héritage entre interfaces

C# permet à une interface d'hériter d'une autre. Une interface qui hérite d'une autre peut réduire le nombre de lignes de code et permet du polymorphisme d'interfaces . Cette technique est même souvent utilisée dans des interfaces des bibliothèques de .NET. Par exemple, IReadOnlyList<T> implémente IEnumerable<T>, IReadOnlyCollection<T>, et IEnumerable : https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlylist-1?view=net-7.0

**Modifiez** votre IWeightedGraphRepresentation pour exploiter l'héritage entre interfaces : héritez de IGraphRepresentation. Ce qui spécialise un IWeightedGraphRepresentation est sa possibilité de prendre en paramètre un coût pour l'ajout d'un lien entre deux nœuds. Vous devez ajouter à AdjacencyMatrixCosts et AdjacencyListCosts une version d'AddEdge à deux paramètres, pour satisfaire IGraphRepresentation. Faites en sorte que cette surcharge de la méthode assigne un coût par défaut, si jamais un client décide de l'utiliser au lieu de la surcharge à trois paramètres. Vous pouvez décider du coût par défaut, mais il est recommandé d'utiliser un coût de 1.

### 2) Setup du projet Unity

Créez un nouveau projet Unity. Dans ce projet, importez votre bibliothèque de la partie A. Votre bibliothèque se trouve dans bin/Debug/net6.0 et a le nom PathfindingLib.dll. **À chaque fois que vous faites une modification à votre bibliothèque, vous devez la recompiler et remplacer le .dll dans le projet Unity.**

## 2) Création de graphes dans Unity

Créez une composante `GameGraphs`. Ajoutez `using PathfindingLib` au code pour utiliser votre bibliothèque.
Dans `Awake`, créez 4 graphes. Vous devez avoir un graphe pour chaque sorte dans votre bibliothèque (matrice d'adjacence avec et sans coûts, liste d'adjacence avec et sans coûts). Décidez du nombre de nœuds par graphe via un `SerializedField`. Décidez aussi les liens via les méthodes `AddEdge(...)`. Pour les graphes avec coûts, indiquez également des coûts pour les liens.

Pour simplifier le travail, nous tenons pour acquis que le nombre de nœuds d'un graphe dans le jeu ne changera pas. Nous tenons aussi pour acquis qu'un graphe ne sera pas détruit pendant le jeu. Par contre, la connectivité d'un graphe peut changer pendant l'exécution.

## 3) Notion de positions pour les noeuds

Pour le jeu, chaque nœud d'un graphe est associé à une position dans l'espace du monde (world space). Les positions seront utilisées pour visualiser les nœuds et les liens. Elles seront également utiles pour A*.

Nous tenons pour acquis que toutes les positions peuvent être initialisées dans le `Awake` et que les positions ne changeront pas pendant l'exécution.

Trouvez des structures de données appropriées pour tenir en mémoire les valeurs de positions pour chaque nœud. Inventez des positions et insérez-les dans les structures de données.

## 4) Visualisation des graphes dans Unity

Pour chaque graphe, créez une visualisation du graphe. Utilisez `public void OnDrawGizmos()` et des méthodes telles que `Gizmos.DrawSphere` et `Gizmos.DrawLine`. Vous devez afficher chaque nœud et chaque lien entre les nœuds. Utilisez des couleurs avec une interpolation linéaire pour distinguer vos nœuds et liens. Voici un exemple.

```
float nNodesF = nNodes;

for (int i = 0; i < nNodes; ++i)

{
   Gizmos.color = Color.Lerp(Color.red, Color.blue, i/nNodesF);

   ...
```

## 6) Algorithmes

Pour chaque graphe, exécuter un algorithme de pathfinding de votre choix. Vous devez avoir au moins un exemple de chaque algorithme. Pour A*, utilisez la distance entre deux nœuds connectés comme heuristique. Par exemple, les voisins plus loins seront moins privilégiés par A*. Dans la console Unity, affichez le chemin trouvé.

## 7) Déplacement

Créez des GameObjects qui se déplacent d'un nœud à l'autre à l'aide de chemins trouvés par les algorithmes. Ces GameObjects peuvent être donnés à la composante via des `SerializedField`. Lorsqu'un objet atteint son but, trouvez-lui un nouveau but et un nouveau chemin optimal.

Vous pouvez implémenter le tout dans `GameGraph` pour la simplicité, malgré qu'il est recommandé de découpler les graphes et le code de déplacement.

**Hint:** Les `Queue` pourront vous aider à garder en mémoire les nœuds à visiter pendant le déplacement. Par exemple, `Dequeue` un nœud une fois que l'objet s'y rend et commencer le déplacement vers le nouveau nœud. Il existe un constructeur pour `Queue` qui prend en paramètre une `List`, afin de construire la file.

**Hint:** N'oubliez pas que la distance entre deux points sera sans doute un float qui sera rarement == 0. Utilisez un *threshold* de distance. Par exemple, si la distance entre l'objet et son prochain nœud à visiter est plus petite que 0.1f…

## Évaluation Partie B

| Travail | % |
|---|---|
| Création des graphes | 5 |
| Positions des noeuds | 5 |
| Visualisation | 15 |
| Algorithmes | 10 |
| Déplacement | 15 |
| **TOTAL** | **50** |

# Remise

Un membre de l’équipe doit soumettre le TP dans la boite de remise dédiée sur ColNet. Assurez-vous de bien identifier les noms des membres dans le nom du fichier .zip.

La date de remise est le 6 octobre avant minuit.