TP C#9
PathFinding

Rendu

Archive

Vous devez rendre un zip suivant précisément l'architecture suivante:

```
rendu-tp-prenom.nom.zip
    |-- rendu-tp-prenom.nom/
        |-- AUTHORS
        |-- README
        |-- PathFinding/
            |-- PathFinding.sln
            |-- PathFinding/
                |-- Program.cs
                |-- Map.cs
                |-- Room.cs
                |-- tout sauf bin/ and obj/
```

- Vous devez évidemment remplacer *prenom.nom* par votre nom.
- Votre code doit compiler (Il ne doit y avoir aucun warning).

AUTHORS

Ce fichier doit contenir: une étoile (*), un espace, votre login et un retour à la ligne.
Voici un exemple:

```
* prenom.nom
```

Notez que le fichier AUTHORS n'a AUCUNE extension.

README

Ce fichier contient les informations par rapport à votre TP : qu'est ce que vous êtes arrivé à faire, qu'est ce qui vous a posé problème, **les bonus que vous avez réalisé**...
Notez que le fichier README n'a AUCUNE extension.

1 Introduction

1.1 Objectifs

Durant ce TP, vous allez apprendre de nouvelles notions comme:

- Un algorithme de pathfinding[1] (Dijkstra)
- Les piles génériques: `Stack<T>`
- Les Tuples: `Tuple<T, ...>`

Et revoir d'importantes notions comme:

- Les classes
- Les exceptions
- Les listes génériques: `List<T>`

2 Cours

2.1 Pathfinding

D'après Wikipedia, la recherche de chemin, ou pathfinding, est le calcul par l'ordinateur du plus court chemin entre deux points.

Vous l'avez compris, cette semaine nous allons parler de la recherche d'un chemin entre le point A et le point B. Pour les humains, c'est simple de trouver le chemin le plus court entre A et B, pensez par exemple lorsque vous prenez le métro dans Paris. Vous pouvez le trouver, mais pour un ordinateur c'est vachement plus compliqué. Vous devez lui faire comprendre ce qu'est une station de métro, ce qu'est une distance, quelles sont les connexions entre ces deux endroits. Avec deux points, c'est simple de trouver le plus court chemin (parce qu'il n'y a qu'un seul chemin) mais si on ajouter plusieurs stations, 3 par exemple, comment peut-il faire pour le trouver ? Vous avez des yeux, vous avez un cerveau (enfin certains d'entre vous en ont un) donc vous pouvez calculer quel chemin est meilleur mais comment faire penser un ordinateur ? Dans ce TP, vous n'allez pas faire un ordinateur possèdant une conscience mais vous allez le rendre un peu plus intelligent.

Le pathfinding c'est aussi un problème de complexité. Notre but c'est de construire un algorithme qui calcule rapidement le chemin entre 2 points dans une map. Pour ce TP, on pourrait mettre tous les chemins du point source jusqu'au point destination dans une matrice et calculer (très lentement) le plus court chemin. Mais cette technique n'est pas du tout optimisée et ne fonctionne qu'avec de très petites maps. C'est pour cela que nous allons voir une approche du pathfinding qui marchera avec toutes les maps, quelles que soient leurs tailles et dans un temps raisonnable.

[1] Recherche de chemin

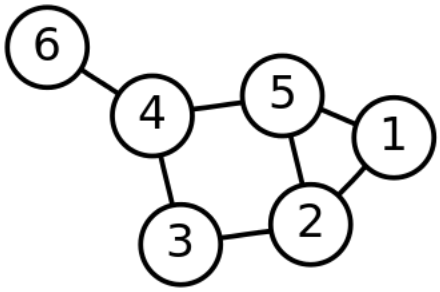

Figure 1: Un exemple de graphe

2.2 Graphes

Aujourd'hui nous allons voir une version simplifié des graphes. Un graphe est un ensemble de noeuds (aussi appelés sommets) qui sont liés avec des arcs. Vous avez déjà vu les arbres binaires dans vos cours d'algorithmique. Un arbre binaire est un type de graphe particulier. Dans ce TP, nous allons utiliser des graphes non-orientés avec des coûts positifs et négatifs. Traduction : on peut aller du sommet A au sommet B et du sommet B au sommet A avec un coût de 1 ou un coût de -5. Il y a d'autres types de graphes mais vous allez les voir l'année prochaine.

2.3 Dijkstra

L'algorithme de Dijkstra est un algorithme de recherche de chemin très connu, nommé d'après son créateur Edsger W. Dijkstra un chercheur très connu. Il était une personne les plus influentes dans le monde de l'informatique et est connu pour ce que nous allons étudier cette semaine mais également pour une tonne d'autres choses qui ont définies l'informatique d'aujourd'hui. [2] Cet algorithme est généralement utilisé avec des graphes possèdant des coûts mais il fonctionne quand même avec le type de graphes que nous allons utiliser aujourd'hui.
Le principe de Dijkstra est simple : chaque sommet contient un booléen pour savoir s'il a été visité ou non, son meilleur parent et une distance représentant la distance entre le sommet de départ et ce sommet.
Voici les étapes principales de Dijkstra:

- Mettre tous les sommets à non-visité et leurs distances à infini.
- Mettre le sommet de départ en tant que sommet courant. Sa distance est 0, il n'a pas de meilleur parent et n'a pas encore été visité.
- Mettre à jour les distances avec ses fils: pour chaque fils, vous ajoutez la distance du sommet courant (dans ce cas 0) avec la distance du fils. Par exemple, si le fils est à 5 de distance alors sa distance depuis le sommet courant est $0 + 5 = 5$. Si la distance trouvée est inférieure à la distance du fils (dans notre cas, 5 c'est vachement moins que l'infini) alors vous mettez à jour la distance du fils avec la valeur trouvée et vous mettez le meilleur parent du fils comme étant le sommet courant. Lisez, re-lisez et re-re-lisez ce point tant que vous ne le comprennez pas.

[2] Vous pouvez en trouver plus sur Wikipedia, c'est très impressionnant.

- Maintenant que vous avez fait cela, vous mettez le booléen visité du sommet courant à Vrai. Et vous mettez le pointeur du sommet courant à rien.
- Maintenant vous regardez tous les sommets non-visités. Vous devez trouver le sommet avec la distance la plus petite **qui n'a pas été visité**. Ce sommet devient votre sommet courant.
- Vous mettez à jour la distance avec ses fils comme vous l'avez fait dans le troisième point de cette liste (n'oubliez pas de changer le meilleur parent si besoin !) et vous mettez le sommet courant comme étant visité.
- Vous faîtes les deux derniers points tant que vous n'avez pas fait tous les chemins possibles du graphe. Il n'est pas toujours utile de vérifier tous les chemins pour trouver le plus court mais nous n'allons pas l'expliquer dans ce cours. Si vous voulez le savoir, vous pouvez aller sur Google [3].
- Vous avez tout fait ? Super. Maintenant la partie la plus simple. Prenez une pile, mettez le sommet final dedans puis allez sur son "meilleur" parent. Empilez le. Puis allez sur son "meilleur" parent et empilez le. Puis... Faîtes le tant que vous n'êtes pas sur le sommet de départ. Empilez le et voilà, c'est terminé.
- Pour connaître le chemin le plus court du sommet de départ à celui d'arrivé, dépilez tous les sommets que vous avez mis dans la pile.

On sait que cela peut être compliqué à comprendre mais lisez cette partie plusieurs fois, faîtes des schémas avec des graphes et réalisez l'algorithme dessus, regardez sur Wikipedia si vous en avez besoin. Prenez votre temps pour comprendre comment Dijkstra fonctionne, c'est important pour ce TP[4].

Ne vous inquiétez pas, vous allez être guidé pendant tout le TP. C'est plus simple qu'il n'y paraît!

[3] https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra#Sp.C3.A9cialisation_de_l.27algorithme
[4] Slide avec des exemples : http://licence-math.univ-lyon1.fr/lib/exe/fetch.php?media=gla:dijkstra.pdf

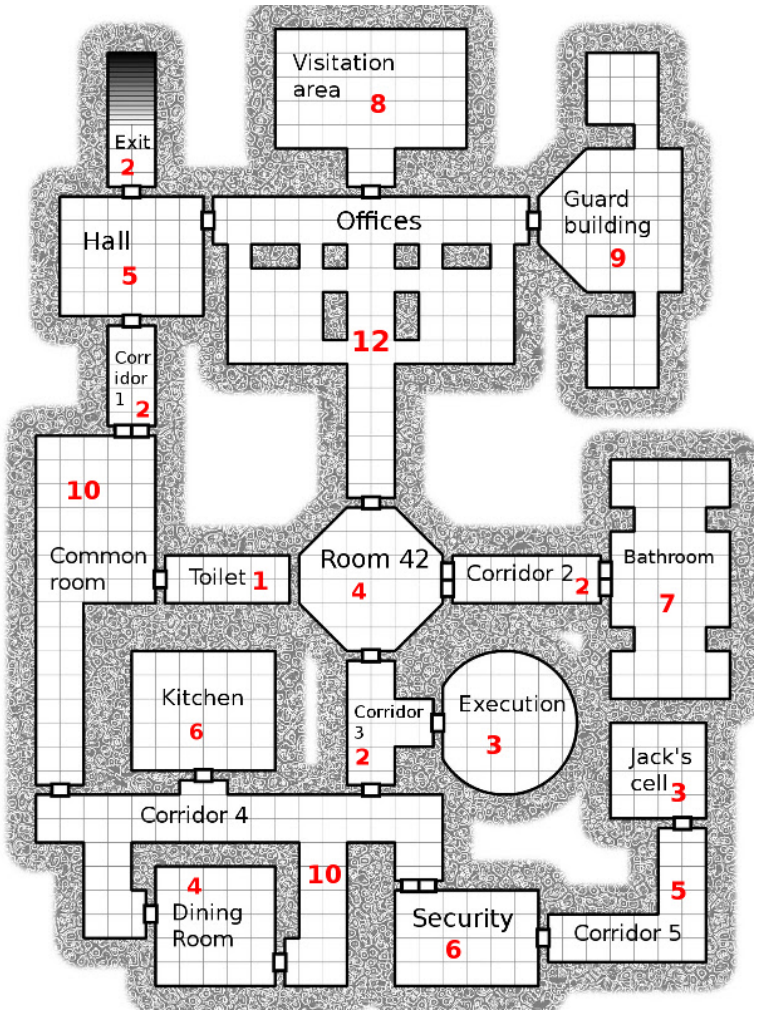

Figure 2: Le plan de la prison

3 Exercice: Get Jack the fuck outta here!

Le Capitaine Barbossa a capturé Jack Sparrow et l'a enfermé dans la plus grande prison des Caraïbes pour qu'il ne trouve aucun moyen de s'échapper et qu'il y reste enfermé pour l'éternité. D'après le code des pirates, nous n'avons pas le choix, nous devons délivrer Jack le plus tôt possible.

3.1 Avant de commencer

Dans cet exercice, votre but est de coder un algorithme de pathfinding pour trouver le plus court chemin entre la cellule de Jack et la sortie de la prison.
Vous allez coder dans 3 fichiers: **Room.cs**, **Map.cs** et **Program.cs**. Room.cs et Map.cs contiennent respectivement les classes `Room` et `Map`. Le fichier Program.cs contient uniquement la méthode `Main`.
Comme nous allons utiliser des collections génériques telles que `List<T>` et `Stack<T>`, n'oubliez pas d'ajouter `using System.Collections.Generic;` au début de chaque fichier.

En terme de design objet, notre graphe va représenter la prison et chaque pièce sera un noeud du graphe.

3.2 Room.cs

Chaque noeud de notre graphe peut être vu comme une pièce de la prison. Donc nous avons besoin d'une classe `Room`!

La classe Room

La première classe que vous avez à coder est la classe `Room`. Une `Room` a un `name`, une `size` et des `neighbours`.
Les `neighbours` d'une `Room` X sont toutes les autres `Room`s connectées à X avec une porte.
Les 3 attributs sont `private`.
Notez que l'attribut `size` permettra à votre Dijkstra de trouver un plus court chemin en considérant la taille de chaque pièce. Les différentes tailles de chaque pièces, ainsi que leurs noms, sont affichés sur la Figure 2: Le plan de la prison.

```
1  public class Room
2  {
3          // Attributes
4          private string name;
5          private int size;
6          private List<Room> neighbours;
7
8          // Methods
9          // FIXME
10 }
```

Le constructeur de Room

Le constructeur de `Room` initialise le `name` et la `size` avec les arguments et les `neighbours` avec une `List<Room>` vide.

```
public Room(string name, int size);
```

Toutes les méthodes suivantes sont bien évidemment à coder dans la classe `Room`.

Name getter

Comme l'attribut `name` est `private`, nous avons besoin d'un getter pour y accéder depuis l'extérieur de la classe.

```
public string GetName();
```

Size getter

Comme l'attribut `size` est `private`, nous avons besoin d'un getter pour y accéder depuis l'extérieur de la classe.

```
public int GetSize();
```

Ajouter une porte avec une autre pièce

Cette méthode ajoute à la liste `neighbours` la pièce en argument. N'oubliez pas que vous devez aussi ajouter la pièce actuelle dans les `neighbours` de la pièce en argument! (Si il y a une porte pour passer d'une pièce A à une pièce B, alors on peut aussi passer de B à A...)

```
public void AddDoorWith(Room room);
```

Checker si une pièce est connectée à une autre

Cette méthode check si la pièce actuelle à la pièce en argument dans ses `neighbours`. Elle retourne `true` si la pièce en argument est dans ses `neighbours` et `false` sinon.

```
public bool HasDoorWith(Room room);
```

3.3 Map.cs

Si les noeuds du graphe sont les pièces, vous avez deviné que la map est le graphe. Construisons la classe `Map`!

La classe Map

Nous arrivons enfin dans le vif du sujet!
La classe `Map` a un attribut `private` qui est la liste des pièces (`List<Room>`) appelée `rooms`.

```
1  public class Map
2  {
3          // Attributes
4          private List<Room> rooms;
5
6          // Methods
7          // FIXME
8  }
```

Le constructeur de Map

Le constructeur de `Map` initialise l'attribut `rooms` avec un tableau de `Tuple<string, int>`. Rappellez vous que le constructeur de `Room` prend une `string` et un `int`.
Hint1: Pour accéder aux premier et second éléments d'un `Tuple` utilisez respectivement les champs `Item1` et `Item2`.

Hint2: Pour ajouter un élément dans une liste, utilisez la méthode `Add` de `List<T>`. Quelque chose comme `this.rooms.Add(new Room(...))` semble utile dans notre cas.

```
public Map(Tuple<string, int>[] rooms);
```

Les méthodes suivantes sont bien évidemment à coder dans la classe `Map`!

Ajouter une porte entre 2 pièces

La méthode suivante ajoute une porte entre 2 pièces de notre `Map`.
Vous voulez trouver une pièce spécifique dans une `List<Room>` avec uniquement son nom?
`this.rooms.Find(room => room.GetName() == room1)` retourne la pièce nommée `room1` si elle existe dans `this.rooms` et `null` sinon. Allez voir MSDN pour plus d'infos.

```
public void AddDoorBetween(string room1, string room2);
```

Il faut sauver le soldat Jack

Vous devez coder la méthode `FindShortestPath` qui trouve le plus court chemin de la pièce nommée `src` à la pièce nommée `dest`.
Cette méthode retourne une `Stack<Room>`[5] parce que quand vous aurez rempli le vecteur de père, vous allez empiler toutes vos pièces, de la pièce destination à la pièce source, dans la pile. Ensuite, pour obtenir le plus court chemin, de la source à la destination, vous n'aurez qu'à dépiler les pièces une par une.
Cette méthode lance une exception, avec un message approprié, si la pièce `src` ou la pièce `dest` n'existe pas dans la map.

```
public Stack<Room> FindShortestPath(string src, string dest);
```

N'hésitez pas à relire encore et encore la partie cours sur Dijkstra ou à demander de l'aide à vos ACDCs si vous êtes complètement perdu. **Google reste votre meilleur ami.**

Vous pouvez bien sûr vous aider mais, faites attention, comme l'algorithme de Dijkstra est assez difficile à implémenter en Sup, nous ne tolérerons aucune triche! Si 3 étudiants ou plus ont exactement la même implémentation de l'algorithme, nous le verrons et votre note sera divisée par le nombre de tricheurs.

Souvenez-vous du chemin

Il vous reste une chose à faire pour aider le Capitaine : lui donner le chemin d'une manière compréhensible par tous, et surtout par lui. Il va oublier le chemin dès qu'il va passer par la porte donc écrivez le lui dans un fichier. Utilisez la méthode `WritePath` pour l'aider. Retournez false si quelque chose de mal s'est passé, true si tout va bien.

```
public bool WritePath(string filename, Stack<Room> path);
```

[5] Documentation des Stack<T>: https://msdn.microsoft.com/fr-fr/library/3278tedw(v=vs.110).aspx

Le fichier devra être comme ceci :

```
--- Path to follow ---
Room 1
Room 2
...
Room n
--------------------
Total cost: 24

// If an error occurs, it should display on the console
Error during the opening of [FILENAME]
[ERROR MESSAGE]
```

3.4 Program.cs

Dans ce fichier nous allons juste mettre notre méthode `Main` afin de tester le pathfinding. Pour rappel, ne tentez pas de nous tromper avec un path hard codé. De toute façon, nous n'essayerons pas que votre Dijkstra avec la map de la prison mais avec plein d'autres dont des beaucoup plus grandes pour voir si votre algorithme fonctionne vraiment.

```
1  public static void Main(string[] args)
2  {
3          // Create a new map with pairs name/size for each room
4          Map map = new Map(new[] {
5                  new Tuple<string, int>("Corridor 1", 2),
6                  new Tuple<string, int>("Corridor 2", 2),
7                  new Tuple<string, int>("Corridor 3", 2),
8                  new Tuple<string, int>("Corridor 4", 10),
9                  new Tuple<string, int>("Corridor 5", 5),
10                 new Tuple<string, int>("Exit", 2),
11                 new Tuple<string, int>("Hall", 5),
12                 new Tuple<string, int>("Visitation Area", 8),
13                 new Tuple<string, int>("Offices", 12),
14                 new Tuple<string, int>("Guard Building", 9),
15                 new Tuple<string, int>("Room 42", 4),
16                 new Tuple<string, int>("Common Room", 10),
17                 new Tuple<string, int>("Toilet", 1),
18                 new Tuple<string, int>("Kitchen", 6),
19                 new Tuple<string, int>("Bathroom", 7),
20                 new Tuple<string, int>("Dining Room", 4),
21                 new Tuple<string, int>("Security", 6),
22                 new Tuple<string, int>("Jack's Cell", 3),
```

```
                new Tuple<string, int>("Execution", 3)
        });

        // Add doors between rooms to get a concrete map
        map.AddDoorBetween ("Jack's Cell", "Corridor 5");
        map.AddDoorBetween ("Security", "Corridor 5");
        map.AddDoorBetween ("Security", "Corridor 4");
        map.AddDoorBetween ("Dining Room", "Corridor 4");
        map.AddDoorBetween ("Kitchen", "Corridor 4");
        map.AddDoorBetween ("Common Room", "Corridor 4");
        map.AddDoorBetween ("Common Room", "Toilet");
        map.AddDoorBetween ("Common Room", "Corridor 1");
        map.AddDoorBetween ("Hall", "Corridor 1");
        map.AddDoorBetween ("Hall", "Exit");
        map.AddDoorBetween ("Hall", "Offices");
        map.AddDoorBetween ("Visitation Area", "Offices");
        map.AddDoorBetween ("Guard Building", "Offices");
        map.AddDoorBetween ("Room 42", "Offices");
        map.AddDoorBetween ("Corridor 3", "Corridor 4");
        map.AddDoorBetween ("Corridor 3", "Execution");
        map.AddDoorBetween ("Corridor 3", "Room 42");
        map.AddDoorBetween ("Corridor 2", "Room 42");
        map.AddDoorBetween ("Corridor 2", "Bathroom");

         try
        {
                // Get the path using our Dijkstra algorithm
                Stack<Room> path = map.FindShortestPath("Jack's Cell", "Exit");

                // FIXME: Display a formatted message on stdout
                // with the path from the Jack's Cell to the Exit
        }
        catch (Exception e)
        {
                // FIXME: Print the exception
        }
}
```

Tu as trouvé un chemin reliant la cellule de Jack à la sortie? Alors lève toi pirate et crie...

Souvenez-vous de ce jour comme le jour
où vous avez FAILLI capturer le capitaine Jack Sparrow!

Eh attendez !

Vous pensiez que c'était tout ? Non vous n'êtes pas aussi stupide que ça ! On parle du Capitaine Jack Sparrow! Vous savez comment il est, il aime jouer donc voici un jeu: il va essayer de trouver le plus court chemin avant que vous le lui donniez. Mais à cause de son rang, c'est vous qui allez écrire ses prédictions et les comparer avec le bon résultat. Pour cela, vous allez coder les méthodes suivantes:

```
publicc List<string> GetPredictions();
public void ComparePredictions(List<string> predictions, Stack<Room> path);
```

Le rendu devra être comme ceci :

```
// The predictions are wrong (the number of rooms are not the same)
Ohoh, you got it wrong !

// Predictions are wrong too
Ok, you're wrong buddy. You got [NUMBER] on [NUMBER] good answers

// Predictions are right
You are a wizard buddy, you were right!
```

4 Bonii

Voici une liste de bonii pour votre TP. Il est conseillé de les réaliser dans l'ordre donné. Comme ils sont entièrement basés sur le TP, **vous devez finir la partie obligatoire avant de commencer les bonii**.

- Vous pouvez afficher le plus court chemin dans la console avant (ou après) l'avoir mis dans un fichier. Mais si vous le faîtes, n'oubliez pas d'ajouter quelques couleurs.
- La prison est souvent sale donc il y a parfois des salles fermées pour qu'on puisse les nettoyer. Ajouter une option pour que l'utilisateur choisisse les salles fermées
- Ajouter les bonii que vous souhaitez, seulement si vous avez fait les précédents. Soyez créatifs et amusez-vous

N'oubliez pas de nous dire quels bonus vous avez réalisé dans votre README.

The code is the law.