INF1008 – Analyse d'algorithmes

Session HIV-21

TRAVAIL PRATIQUE #1

Indications principales	
Date de présentation de l'énoncé	18 février 2021
Date de remise du travail	1er avril 2021
Politique sur remises tardives	-25% par jour de retard
Éléments et format de remise	Rapport et fichiers complet de la solution .NET ou du projet remis sous forme électronique sur le portail de cours dans un seul fichier compressé (.zip)
Pondération	15%
Nb d'étudiants par équipe	3 à 5, sans exception

INTRODUCTION

Nous avons vu en classe l'algorithme de Prim, qui permet de trouver l'arbre sous-tendant minimal à un arbre, comme le fait également l'algorithme de Kruskal.

Une des applications intéressantes de l'algorithme de Prim est de s'en servir pour générer un labyrinthe aléatoire.

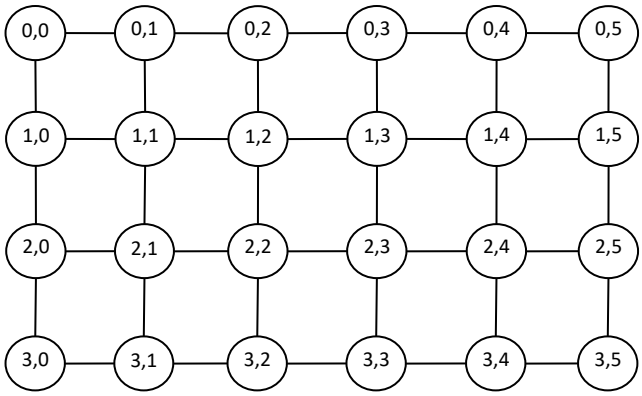
Dans ce travail pratique, vous serez invités à implémenter une application qui permet de générer aléatoirement un labyrinthe de grandeur m x n, avec une entrée et une sortie, puis à afficher le labyrinthe à la console ou dans un output de votre choix.

Vous devrez programmer le tout dans un langage .NET de votre choix (C++, C#, VB.NET, F#), ou Java. Tout autre langage doit être approuvé par l'enseignant.

PRINCIPES DE FONCTIONNEMENT

Comment peut-on arriver à générer un labyrinthe à partir de l'algorithme de Prim?

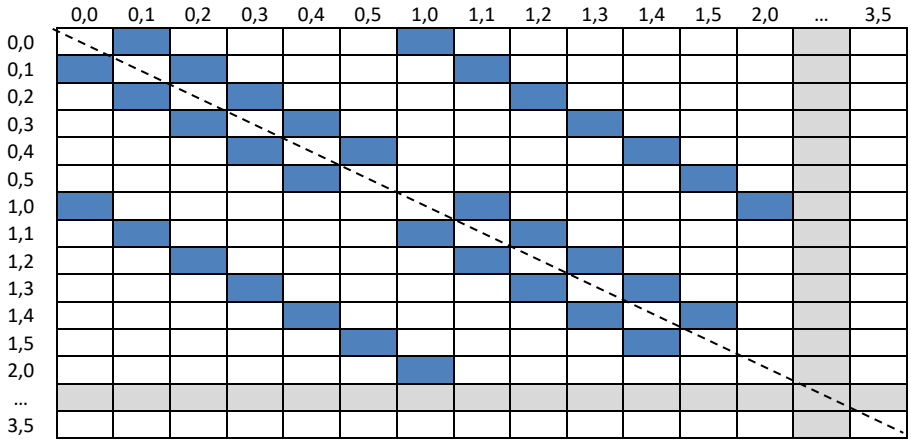
- Pensez à un graphe de la forme suivante, où chaque nœud, de « 0,0 » à « 3,5 », représente également une grille de taille 6 x 4 :



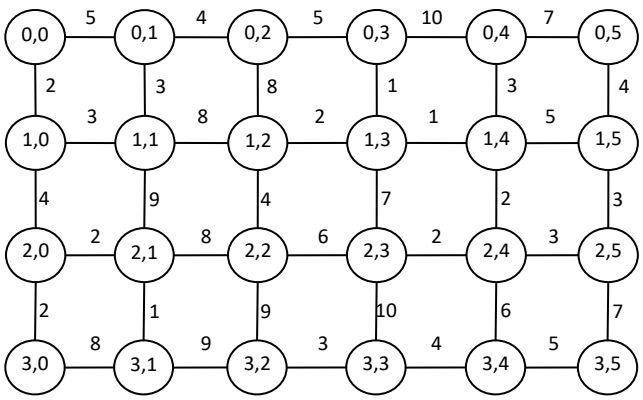
On se retrouve donc avec 24 nœuds reliés, lorsque c'est possible, aux nœuds :

- *« i, j + 1 »*
- *« i, j – 1 »*
- *« i + 1, j »*
- *« i – 1, j »*

- On peut par ailleurs imaginer représenter/implémenter le graphe sous la forme d'une matrice d'adjacence *n x n* nœuds, où les cellules de couleur représentent les arrêtes entre les nœuds pour lesquels il y aura des valeurs (longueur ou poids) positives (tous les autres étant à 0):



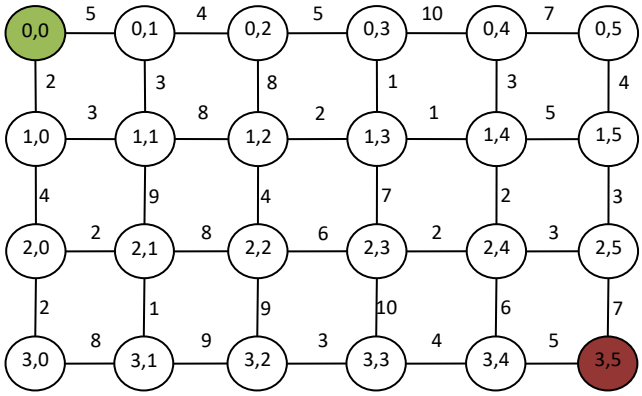
- Imaginons ensuite que pour toutes ces arrêtes, on génère une valeur aléatoire entre 1 et 10.



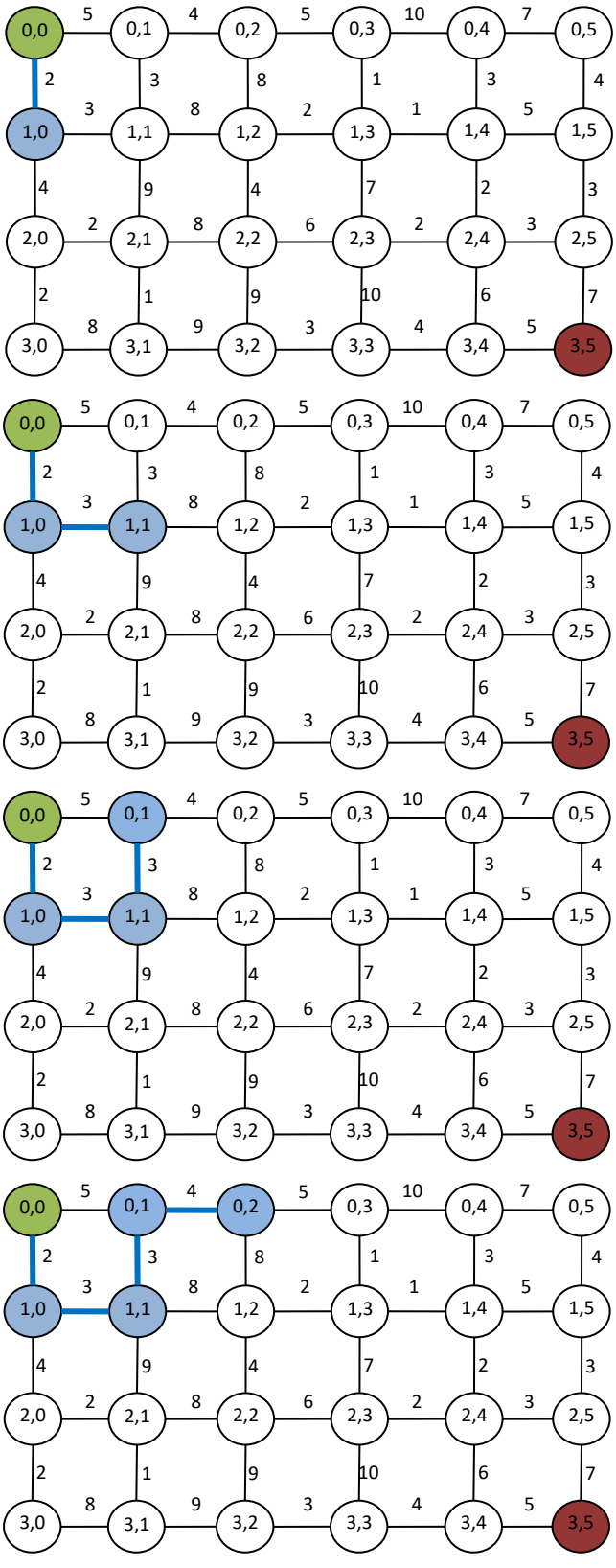
Et la matrice d'adjacence qui le reflète tel que :

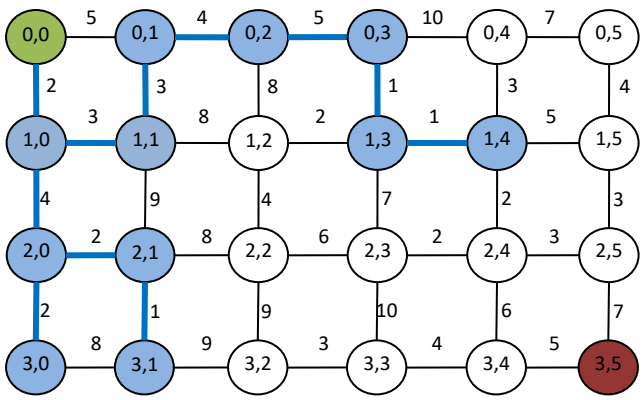
	0,0	0,1	0,2	0,3	0,4	0,5	1,0	1,1	1,2	1,3	1,4	1,5	2,0	…	3,5
0,0		5					2								
0,1	5		4					3							
0,2		4		5					8						
0,3			5		10					1					
0,4				10		7					3				
0,5					7							4			
1,0	2							3					4		
1,1		3					3		8						
1,2			8					8		2					
1,3				1					2		1				
1,4					3					1		5			
1,5						4					5				
2,0							4								
…															
3,5															

- Déterminons maintenant un point d'entrée dans le labyrinthe, et un point de sortie : utilisons les nœuds « 0,0 » et « 3,5 », aux coins opposés :

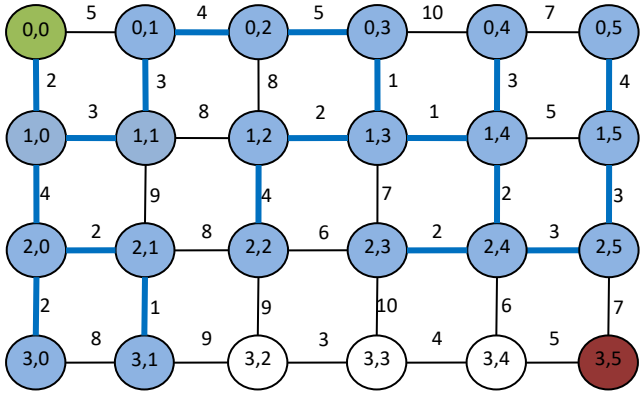


- Que se passe-t-il alors lorsqu'on applique l'algorithme de Prim sur ce graphe à partir du nœud arbitraire « *0,0* »? ***Rappel*** : *à chaque étape/itération, l'arête avec la valeur la plus courte qui relie un nœud déjà considéré et un nœud parmi tous les nœuds non considérés ayant une arête en commun avec un des nœuds considérés est retenue* :

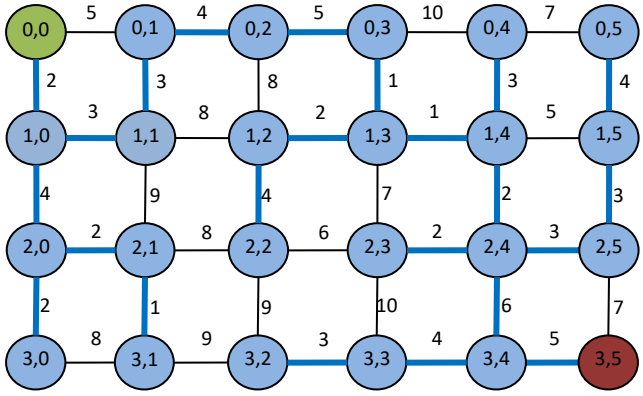

0,0 0,1 0,2 0,3 0,4 0,5
1,0 1,1 1,2 1,3 1,4 1,5
2,0 2,1 2,2 2,3 2,4 2,5
3,0 3,1 3,2 3,3 3,4 3,5
...


0,0
0,1
0,2
0,3
0,4
0,5
1,0
1,1
1,2
1,3
1,4
1,5
2,0
2,1
2,2
2,3
2,4
2,5
3,0
3,1
3,2
3,3
3,4
3,5

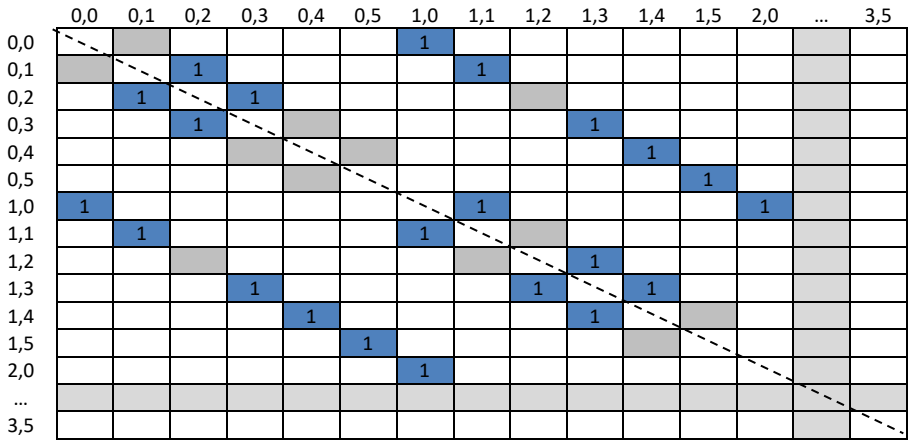
...


0,0
0,1
0,2
0,3
0,4
0,5
1,0
1,1
1,2
1,3
1,4
1,5
2,0
2,1
2,2
2,3
2,4
2,5
3,0
3,1
3,2
3,3
3,4
3,5

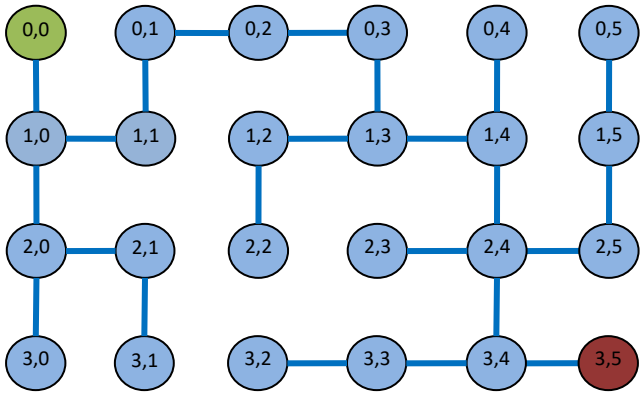
...


0,0
0,1
0,2
0,3
0,4
0,5
1,0
1,1
1,2
1,3
1,4
1,5
2,0
2,1
2,2
2,3
2,4
2,5
3,0
3,1
3,2
3,3
3,4
3,5

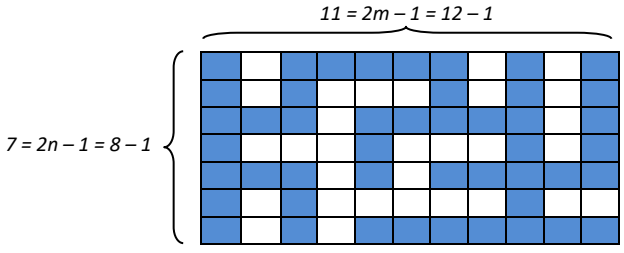
- Si on ignore les poids sur les arêtes *(qui ne sont plus importants une fois l'arbre sous-tendant minimal généré)* et les arêtes qui ne font pas partie de la solution (du labyrinthe), on aurait alors la matrice d'adjacence suivante *(en gris : les arêtes qui du graphe original qui ne font pas partie des chemins du labyrinthe)*:



On se retrouve alors avec une représentation de notre graphe/labyrinthe final :



- Enfin, on peut aussi afficher ce labyrinthe à la console ou dans d'autres modes d'affichage. Il y a alors une certaine difficulté car il faut penser à considérer et les nœuds et les arêtes comme devant être affichés.



Ainsi, pour générer l'affichage, on pourrait choisir, par exemple :

- Un caractère pour signifier un nœud ou une arête en tant que « chemin » : « **O** »;
- Un caractère pour signifier une arête non incluse (un mur, en quelque sorte) : « # »;
- Un caractère pour l'entrée du labyrinthe : « E »;
- Un caractère pour la sortie du labyrinthe : « S »;

De cette façon, on pourrait à l'affichage avoir cette représentation :

```
E  #  O  O  O  O  O  #  O  #  O
O  #  O  #  #  #  O  #  O  #  O
O  O  O  #  O  O  O  O  O  #  O
O  #  #  #  O  #  #  #  O  #  O
O  O  O  #  O  #  O  O  O  O  O
O  #  O  #  #  #  #  #  O  #  #
O  #  O  #  O  O  O  O  O  O  S
```

Il faut alors penser à une stratégie à savoir comment parcourir notre matrice d'adjacence de façon à pouvoir générer un tel affichage.

- <u>Des questions qu'on peut se poser :</u>
  - A-t-on besoin d'exploiter/fouiller toute la matrice d'incidence, ou seulement une des parties au-dessus ou sous la diagonale?
  - Peut-on en effet confirmer la complexité algorithmique pour l'algorithme de Prim : $\boldsymbol{\Theta}(n^2)$?
  - Quelle est la complexité des autres opérations nécessaires pour les différentes fonctionnalités décrites?
    - Initialisation du labyrinthe;
    - Génération aléatoire des poids pour les arêtes;
    - Affichage/lecture du labyrinthe.

SPÉCIFICATIONS DU TRAVAIL DEMANDÉ

Dans le cadre de ce travail, vous devez implémenter une application de génération aléatoire de labyrinthe basée sur l'algorithme de Prim.

Application (11.5%)

Votre application console ou avec UI doit implémenter la solution à la suite décrite précédemment, soit :

- Une classe *Labyrinthe* qui doit ***(8.5%)***:
  - Être d'une taille spécifiée lors de l'instanciation d'un de ses objets ***(0.5%)***;
  - Permettre de générer/regénérer aléatoirement le labyrinthe sur appel en faisant usage de l'algorithme de Prim ***(4.5%)***;
  - Contenir une structure de donnée adéquate pour stocker le labyrinthe généré ***(0.5%)***;
  - Permettre un affichage adéquat du labyrinthe tel que décrit dans la section précédente lors de l'appel de la méthode *ToString()* ***(3%)***.

- Une classe *Main* ou un formulaire qui doit ***(1.5%)*** :
  - Permettre à un utilisateur (par un menu qui boucle si vous faites cela en application console) de créer/recréer des labyrinthes avec des tailles différentes ***(0.5%)***;
  - Afficher à la console ou dans des contrôles adéquats le labyrinthe *(1%)*.

- Tout le nécessaire pour comptabiliser le nombre d'opérations élémentaires exécutées (vous devrez donc injecter des instructions aux endroits stratégiques dans votre code en conséquence) par ***(1.5%)***:
  - L'initialisation de la matrice d'adjacence ***(0.5%)***;
  - La génération du labyrinthe ***(0.5%)***;
  - La lecture/l'affichage du labyrinthe ***(0.5%)***.

  On doit pouvoir accéder à ces nombres d'opérations élémentaires par des propriétés ou des méthodes, et ils doivent être affichés à la console ou dans des contrôles suite à l'utilisation des fonctionnalités auxquels ils sont associés.

CONSEILS :
Ne négligez pas d'ajouter tout commentaire pertinent à votre code. Ils ne seront pas évalués comme tel, mais ils pourront tout de même aider lors de la correction à comprendre ce que vous vouliez faire en cas d'erreur.

Rapport (3.5%)

Vous devez également remettre un **rapport Word ou PDF** qui doit contenir les éléments suivant :

- Page de présentation ***(0.5% les 3 premières exigences)***;

- Problèmes et difficultés rencontrés (s'il y a lieu);

- Instructions spéciales d'exécution du programme (s'il y a lieu);

- Des réponses aux questions suivantes, avec **justification détaillée et exemples** (appuyés d'imprimes-écran tirés de l'exécution de votre générateur de labyrinthe avec différentes tailles, par exemple) ***(3%)***:
  - Démontrez que votre implémentation pour la partie de l'application qui exécute l'algorithme de Prim est bien de complexité algorithmique $\boldsymbol{\Theta}(n^2)$ ***(1%)***;

  - Quelle est votre complexité algorithmique pour la lecture/le balayage de votre labyrinthe ***(1%)***?

  - Quelle est la complexité générale de tout le processus de génération du labyrinthe (incluant l'initialisation, la génération aléatoire des poids pour les arêtes, l'exécution de la matrice de Prim et toute autre instruction reliée) ***(1%)***?

  Vous pouvez utiliser la notation Grand O ou Grand Thêta, à votre guise.

EXIGENCES ET INSTRUCTIONS SUPPLÉMENTAIRES

Langage de programmation

Au niveau du langage de programmation, vous devez utiliser soit l'une ou l'autre des options suivantes, **sans aucune exception sans approbation** :

- C#/VB/C++/F# avec Microsoft Visual Studio .NET 2019.
- Java avec Eclipse, NetBeans, IntelliJ, …

Éléments à remettre

Vous devrez remettre l'ensemble des extrants exigés qui seront énumérés ci-dessous dans un seul fichier compressé (en format .zip) dans la section de dépôt des travaux sur le portail du cours **avant le début du cours de la remise.**

Dans le cas où vous remettiez le travail en retard, vous ne pourrez alors le déposer sur le portail et vous devrez me le retourner via courriel. Une pénalité de 25% par jour de retard, à compter de l'heure de remise, serait alors appliquée.

Les éléments à remettre sont les suivants :

- Tous les fichiers relatifs au projet de l'application :
    - Par exemple, en Visual Studio .NET, vous devez remettre le répertoire contenant la solution, le ou les projets associés, les fichiers contenant le code source, les exécutables, etc.

  Dans tous les cas, assurez-vous que le projet soit prêt à être exécuté directement à l'intérieur de la plateforme de développement, c'est-à-dire sans erreur de compilation et sans configuration spéciale non explicitement donnée. Aucun débogage ne sera fait lors de la correction. C'est TRÈS important. Si je dois reconstruire un nouveau projet et importer les classes, les fichiers de configuration et configurer des paramètres pour vous, il y aura d'importantes pénalités appliquées.
- Le rapport PDF.