IFT-2008 – Algorithmes et structures de données

Travail pratique 1

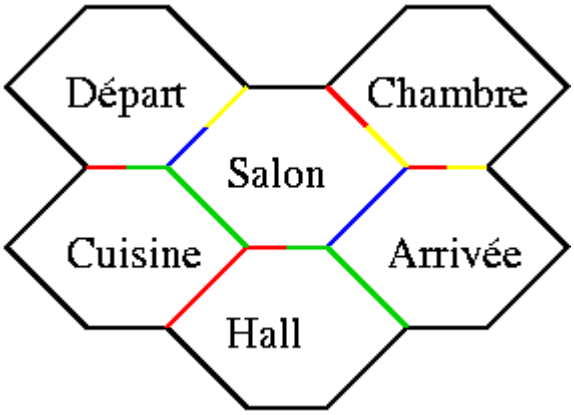
À remettre, sur le portail du cours, avant 17h00 le lundi 24 février 2020

1 Mise en contexte

Imaginons un labyrinthe composé de pièces hexagonales reliées entre elles par des portes de couleurs différentes. Quatre personnages sont placés dans une pièce déterminée du labyrinthe appelée la case départ, et doivent se rendre en un minimum de déplacements à une autre case désignée comme la case d'arrivée. Les quatre personnages sont représentés par les couleurs rouge, vert, bleu et jaune. Chaque porte est aussi colorée en rouge, en vert, en bleu ou en jaune. Un personnage ne peut traverser que les portes de sa couleur. Plusieurs portes peuvent permettre le passage entre deux mêmes pièces. Par exemple, la case de départ pourrait avoir une porte rouge vers la case du bas, une porte verte vers la case du bas, et une autre porte jaune vers la case en diagonale en bas à droite. Les portes ne sont pas à sens uniques. Elles peuvent donc être utilisées dans les deux sens. Entre deux pièces, il peut y avoir jusqu'à quatre portes, soit une de chaque couleur. De plus et étant donné que les pièces sont hexagonales, chaque pièce qui n'est pas sur la bordure du labyrinthe est adjacente à six autres pièces.

Pour un labyrinthe donné, les chances de chacun des joueurs d'arriver à la case d'arrivée le premier ne sont pas égales. Par exemple, des portes rouges pourraient être placées de façon à permettre au joueur rouge d'arriver à la case finale par un chemin en ligne droite, tandis que le joueur jaune pourrait être obligé de faire un détour coûteux.

Le but du présent travail est de déterminer, pour un labyrinthe donné en entrée au programme, lequel des quatre joueurs peut se rendre le plus rapidement, c'est-à-dire en le moins de déplacements de pièce en pièce, à la case d'arrivée. Par exemple, un labyrinthe peut ressembler au suivant :

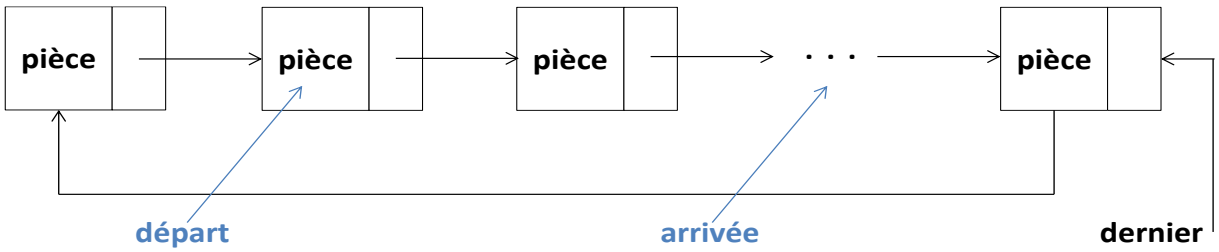


Dans ce cas, le joueur rouge peut se rendre du départ à l'arrivée en 5 déplacements, le joueur jaune peut le faire en 3 déplacements, le joueur vert peut y arriver en 4 déplacements, puis le joueur bleu peut y arriver en 2 déplacements. C'est donc le joueur bleu qui a le plus de chances de gagner.

2 Modélisation d'un labyrinthe hexagonale

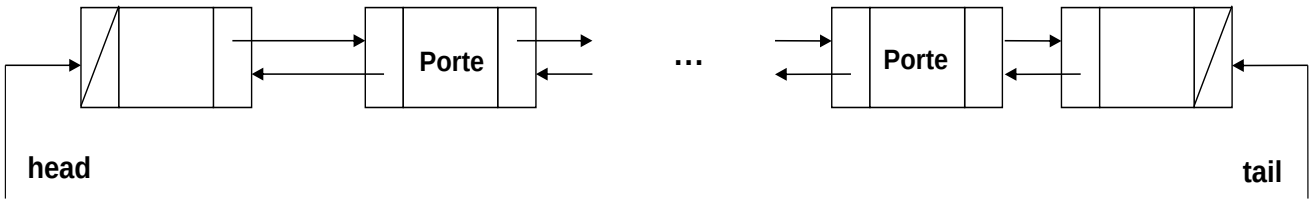
L'ensemble des classes que vous devez utiliser pour réaliser ce travail pratique est défini dans les fichiers fournis (Labyrinthe.h, Porte.h, Piece.h, etc). Vous pouvez ajouter d'autres méthodes à ces classes, qu'elles soient publiques ou privées. Vous pouvez même ajouter des données membres, cependant**, il est interdit de modifier les définitions qui y sont déjà présentes.** Vous pouvez par contre ajouter le préfixe m_ dans le nom des attributs membres des classes fournies ou le préfixe p_ dans le nom des paramètres des méthodes fournies, mais ce n'est pas obligatoire. Le TP utilise d'ailleurs les classes suivantes:

1. Un **labyrinthe** utilisant une liste chaînée circulaire de pièces (voir **Labyrinthe.h**):



   Elle contient un pointeur vers la dernière pièce de la liste, un pointeur vers la pièce de départ et un pointeur vers la pièce d'arrivée. Il ne faut pas confondre la première pièce de la liste avec la pièce de départ. L'ordre dans lequel les pièces sont placées dans la liste de pièces ne signifie rien. La première pièce de cette liste n'est pas nécessairement la pièce de départ, et la dernière pièce de cette liste n'est pas nécessairement la pièce d'arrivée du labyrinthe. Pour connaître la pièce de départ et la pièce d'arrivée, il faut utiliser les pointeurs *depart* et *arrivee*. Vous devez également utiliser une file (*queue* de la STL) afin d'implanter la méthode permettant de solutionner le labyrinthe. L'algorithme qu'il faut utiliser est bien détaillée dans le fichier Labyrinthe.h (voir la méthode **solutionner()**).

2. Une **pièce** contenant une liste de portes est modélisée dans un conteneur *list* de la STL implémenté dans une liste doublement chainée (voir fichier **Porte.h**).



   La pièce contient également un nom (chaîne de caractères) qui doit être unique pour chaque pièce ainsi qu'un booléen facilitant l'implémentation de l'algorithme qui résout le labyrinthe pour un joueur.

3. Une **porte** contenant une couleur (rouge, vert, bleu ou jaune) et un pointeur vers une pièce du labyrinthe (voir fichier **Porte.h**).

Notez bien. Dans le modèle d'implantation imposé, une pièce contient une liste de portes qui la relie à d'autres pièces. Par contre, ces portes relient les paires de pièces dans les deux sens, même si elles ne se trouvent que dans la liste de portes de l'une des deux portes reliées. Dans votre algorithme qui résout le labyrinthe, vous devez absolument veiller à vérifier la possibilité, à partir d'une pièce, d'utiliser une porte dans sa propre liste de porte, mais aussi d'utiliser une porte dans la liste de porte d'une autre pièce qui pourrait mener à la pièce.

3 Travail à faire

Il s'agit d'implémenter dans les fichiers respectifs .cpp toutes les méthodes décrites dans les fichiers d'interface .h correspondant. Vous allez aussi devoir ajouter ou enrichir les commentaires d'interface et d'implémentation en format *Doxygen* dans les divers fichiers .h et .cpp. Nous vous suggérons très fortement de créer des fichiers permettant de tester séparément les méthodes de chaque classe que vous devez implanter (les méthodes des classes Piece, Porte et Labyrinthe). Vous devez donc faire tous les tests nécessaires à chaque étape importante dans votre implémentation afin de vous assurer que cette implantation est correcte au fur et à mesure de sa conception.

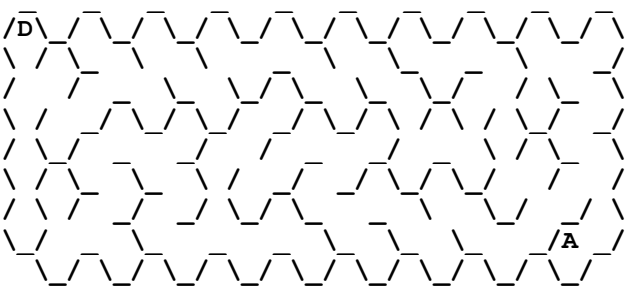
Pour vous aider à tester le bon fonctionnement de votre implémentation (classes impliquées dans ce travail), nous vous fournissons un programme contenant une fonction main(), la méthode qui charge le labyrinthe et celle qui ajoute une pièce. Ce programme devra être compilé avec votre fichier Labyrinthe.cpp complété et fera appel à toutes les méthodes des classes demandées, directement ou indirectement. Ce programme est écrit dans le fichier Principal.cpp que nous vous fournissons, et prend en entrée la description d'un labyrinthe en quatre fichiers, chacun décrivant la position des portes d'une couleur donnée.

La structure de chacun de ces fichiers est la suivante :

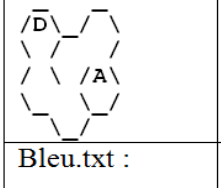
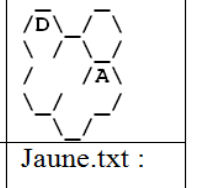
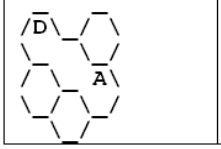
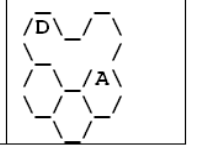
- Une première ligne spécifiant le nombre de colonnes du labyrinthe et le nombre de lignes.
- Un dessin du labyrinthe hexagonal tel que vu par l'un des joueurs, à l'aide des caractères /, \, et _. Dans ce dessin, le caractère D désigne la case départ et le caractère A (ou F) désigne la case d'arrivée.

Par exemple, voici un fichier qui décrit les portes rouges d'un labyrinthe :

10 8



Remarquez que vous n'avez pas à vous attarder à la description formelle de ces fichiers de labyrinthe, puisque nous vous fournissons la méthode de la classe Labyrinthe qui les lit. Les quatre fichiers qui seront lus par le programme de test que nous vous fournissons correspondent aux portes rouges, aux portes vertes, aux portes bleues, puis aux portes jaunes d'un labyrinthe de même taille. Par exemple, si on considère le labyrinthe à la fin de la page 1 de cet énoncé, les quatre fichiers le décrivant contiendraient le texte suivant:

Rouge.txt :	Vert.txt :
3 4 	3 4 
Bleu.txt :	**Jaune.txt :**
3 4 	3 4 

Il faudrait alors exécuter le programme en s'assurant que vous fournissiez ces quatre fichiers. Le programme de test nous répondrait donc que c'est le joueur bleu qui devrait gagner, puisqu'il peut solutionner le labyrinthe en seulement deux déplacements. Vous pouvez, et c'est même nécessaire de le faire, modifier le programme Principal.cpp pour y **ajouter des tests supplémentaires sur les méthodes demandées**. Nous vous fournissons comme exemple 2 jeux de 4 labyrinthes déjà générés que vous pouvez utiliser dans vos essais (voir le répertoire « *data* »).

4 Ce que vous devez remettre

Vous devez rendre un fichier .zip comportant **uniquement** les fichiers suivants :

- L'ensemble des fichiers .h contenant les commentaires d'interface enrichis en format Doxygen.
  - Labyrinthe.h, Piece.h, Porte.h
- Les fichiers .cpp complétés avec les commentaires d'implémentation en format Doxygen.
  - Labyrinthe.cpp, Piece.cpp, Porte.cpp et Principal.cpp
- Le répertoire « *data* » contenant les exemples de labyrinthes.
- NoteTp1-IFT2008.xls (un fichier Excel contenant le barème de correction. Il faut juste ajouter votre nom et matricule sur la première ligne).

PS. Vous n'avez pas à générer la documentation Doxygen en format html pour l'ensemble du projet (pour ne pas envoyer un gros fichier zip). Le correcteur pourra le faire lors de la correction.

Le nom du .zip doit respecter le format suivant : TP1-Matricule.zip. Nous vous rappelons qu'il est important de faire la remise par voie électronique uniquement en vous connectant à: http://monportail.ulaval.ca/ (aucun travail envoyé par courriel n'est accepté). Il est toujours possible de faire plusieurs remises pour le même travail. Vous pouvez donc remettre votre travail plus tôt afin d'être sûr qu'il a été remis en temps, et remettre par la suite une version corrigée avant l'heure limite de remise. De plus, il est de votre responsabilité de vérifier après la remise que vous nous avez envoyé les bons fichiers (**non vides et non corrompus**), sinon vous pouvez avoir zéro comme note pour ce travail.

Attention ! Tout travail remis en retard, un retard débute dès la limite de remise dépassée (dès la première minute), se verra attribué la note zéro.

5 Remarques importantes

Normes de programmation

Vous devez respecter les normes de programmation du cours surtout à ce qui a trait à l'indentation du code. Vous devez également utiliser les balises Doxygen pour commenter votre code.

Tolérance zéro vis-à-vis des erreurs de compilation et portabilité des programmes

Vos programmes doivent absolument pouvoir être compilés et exécutés sans erreurs dans n'importe quel environnement. Un programme qui ne compile pas, peu importe la raison, se verra attribuer la note zéro. Testez intensivement votre programme. **Pour cela, nous recommandons chaudement l'utilisation de Google Test**. **Par contre, la remise des testeurs est à votre discrétion.** Si vous ne testez pas suffisamment votre travail, il risque de provoquer des erreurs à l'exécution lors de la correction. La moindre erreur d'exécution rend la correction extrêmement difficile et vous en serez donc fortement pénalisés.

Bon travail.