Laboratoire 3
Exploration de Graphes et Problèmes de Plus Court Chemin

ELE440 - Algorithmes
Automne 2015

Pondération : 12 points

I. Objectifs

Ce laboratoire porte sur l'exploration de graphes. Il porte aussi sur les concepts d'algorithmes gloutons et de programmation dynamique. Son but est de :

1. Se familiariser avec la notion de graphes pondéré et non-pondéré, ainsi que la notion de graphes orienté et non-orienté,
2. Se familiariser avec l'exploration de graphes en largeur et en profondeur,
3. Se familiariser avec le concept d'algorithme glouton et de programmation dynamique,
4. Expérimenter la notion d'algorithme optimal, sous-optimal et non-optimal, et
5. Utiliser les techniques d'exploration de graphes, d'algorithmes gloutons et de programmation dynamique pour résoudre le problème de recherche du plus court chemin dans un graphe.

II. Implémentation (70 %)

Écrire un programme qui permet de résoudre les problèmes suivants de recherche du plus court chemin :

1. Problèmes de la tour de Londres et du labyrinthe[1].
2. Problème du plus court chemin entre des paires de villes sur une carte routière.

Chacun de ces problèmes se présente sous la forme d'un graphe non-pondéré (problème 1) ou pondéré (problème 2), orienté ou non-orienté. Le travail à faire est décrit dans les sous-sections suivantes :

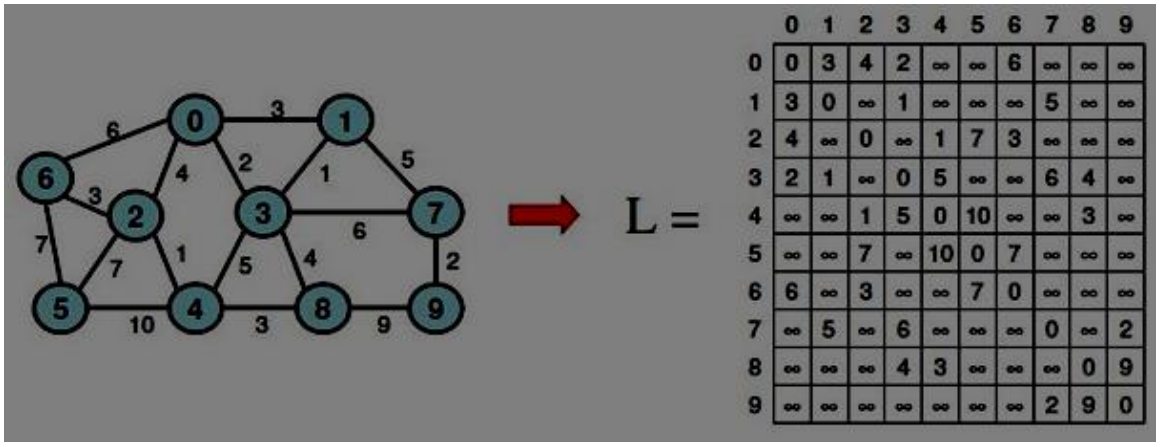
II.1. Création du graphe

Le graphe se présente sous la forme d'une matrice d'adjacence **L**, qui sera lue à partir d'un fichier ou générée aléatoirement par le programme, à la demande de l'usager. Les éléments de **L** sont définis par l'équation (E.1) :

[1] Ces deux problèmes ont une structure très semblable.

$$L(i,j) = \begin{cases} 0 & \text{si } i = j \\ p_{ij} & \text{s'il existe un lien entre } i \text{ et } j \\ \infty & \text{s'il n'existe pas un lien entre } i \text{ et } j \end{cases} \quad \text{(E.1)}$$

Pour le graphe ci-dessous, la matrice d'adjacence sera tel que illustré dans la Figure 1 :



	0	1	2	3	4	5	6	7	8	9
0	0	3	4	2	∞	∞	6	∞	∞	∞
1	3	0	∞	1	∞	∞	∞	5	∞	∞
2	4	∞	0	∞	1	7	3	∞	∞	∞
3	2	1	∞	0	5	∞	∞	6	4	∞
4	∞	∞	1	5	0	10	∞	∞	3	∞
5	∞	∞	7	∞	10	0	7	∞	∞	∞
6	6	∞	3	∞	∞	7	0	∞	∞	∞
7	∞	5	∞	6	∞	∞	∞	0	∞	2
8	∞	∞	∞	4	3	∞	∞	∞	0	9
9	∞	∞	∞	∞	∞	∞	∞	2	9	0

Figure 1. Exemple de graphe pondéré et sa matrice d'adjacence.

Dans le cas des graphes non-pondérés, le poids $\mathbf{p_{ij}}$ de tous les liens est égal à **1**. De plus, dans le cas des graphes non-orientés, la matrice **L** est symétrique. Finalement, nous utiliserons le code **-1** pour représenter le symbole ∞, lorsqu'il n'y a pas de lien entre un nœud **i** et un nœud **j**.

Une liste de nœuds est obtenue à partir de la matrice d'adjacence et sera utilisée par les différents algorithmes. La liste des nœuds aura la forme illustrée dans la Figure 2 :

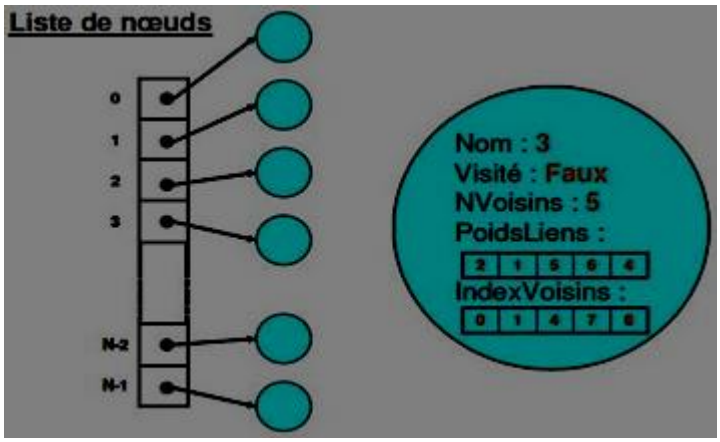


Figure 2. Exemple de liste de nœuds pour un graphe non-pondéré.

L'usager peut choisir la provenance de la matrice d'adjacence **L** du graphe (lue à partir d'un fichier ou générée automatiquement). La matrice **L** servira ensuite à bâtir la liste des nœuds décrite précédemment.

1. *Matrice lue à partir d'un fichier :*

- L'usager doit fournir le nom du fichier.
- La première ligne du fichier fournit **N**, le nombre de nœuds du graphe.
- Les lignes suivantes fournissent les rangées de la matrice **L**. Chaque rangée débute sur une nouvelle ligne dans le fichier.
- Le caractère de séparation entre deux éléments de la même ligne est la tabulation « \t » (voir l'ANNEXE 3 pour un exemple du fichier).

2. *Matrice générée automatiquement :*

- Les graphes générés automatiquement seront toujours non-orientés, donc la matrice d'adjacence **L** sera toujours symétrique.
- Le programme demande à l'usager les informations suivantes :
    - Nombre de nœuds **N**
    - Nombre minimum et maximum de liens pour chaque nœud, sachant que le minimum absolu est **2** et le maximum absolu est **N-1**
    - Le poids minimum et maximum des liens, sachant que le minimum absolu est **1**. Aussi, lorsque le minimum et le maximum sont égaux, le graphe est considéré comme non-pondéré.

Avec ces informations, le programme génère aléatoirement la matrice d'adjacence **L** qui représente le graphe. L'algorithme donne généralement des résultats corrects, mais il peut arriver (rarement) qu'un nœud ait moins que le minimum de liens demandé. Donc, sentez-vous libres de l'améliorer.

```
CreerGraphe (N, minL, maxL, minP, maxP) : L[0..N-1,0..N-1]
   L[0..N-1,o..N-1], numL[0..N-1], tpL[0..N-1]
      Pour n = 0 à N-1
        Pour m = 0 à N-1
           Si (n = m)
              L[n,m] = 0
           Sinon
              L[n,m] = -1
           numL[n] = minL + Random() mod  (maxL-minL)
      Pour k = 1 à minL
        Pour n = 0 à N-1
           Si numL[n] > 0
           K = 0
            Pour p = 0 à N-1
                  Si (L[n,p] < 0) et (numL[p] > 0)
                  TpL[K] = p
                  K = K + 1
            Si (K > 0)
                  v = Random() mod K
                  a = tpL[v]
                  p = minP + Random() mod  (maxP-minP+1)
                  L[n,a] = p
                  L[a,n] = p
                  numL[n] = numL[n] - 1
                  numL[a] = numL[a] - 1
```

II.2. Problèmes de la tour de Londres et du labyrinthe

Les problèmes de la tour de Londres (TL) et du labyrinthe sont apparentés dans le sens où tous les deux se présentent sous la forme d'un problème de recherche de plus court chemin dans un **graphe non-orienté et non-pondéré**.

Le **problème de la tour de Londres** se présente sous la forme de trois tiges de hauteurs différentes et trois anneaux de couleurs différentes. Il s'agit de trouver la plus courte séquence de déplacements d'un anneau à la fois qui permet de partir d'une configuration de départ et d'atteindre une configuration désirée, tel que montré dans l'exemple de la Figure 3.

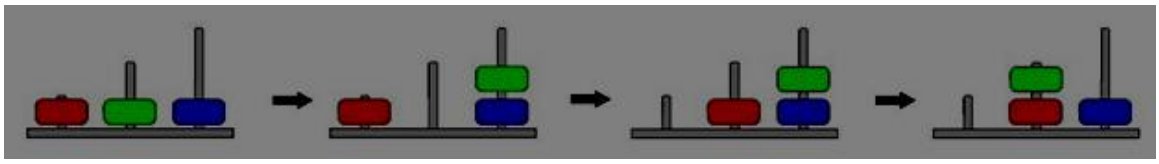


Figure 3. Exemple de séquence de déplacements d'anneaux.

L'ensemble des 36 configurations possibles est donné dans la Figure A1.1 de l'ANNEXE 1. Chaque configuration est représentée par un nœud dans un graphe et chaque transition permise d'une configuration à une autre est représentée par un lien entre les deux nœuds correspondants. La matrice d'adjacence **L** du graphe de la tour de Londres est donnée dans le fichier « *Londres.txt* » fourni avec l'énoncé.

Le **problème du labyrinthe** se présente sous la forme d'une grille contenant des cases blanches et des cases noires. On se déplace dans le labyrinthe horizontalement ou verticalement uniquement sur les cases blanches et le jeu consiste à trouver le plus court chemin d'un point du labyrinthe à un autre. Le problème du labyrinthe est converti en un problème de graphe en associant un nœud du graphe à chaque case blanche du labyrinthe. Les liens du graphe représentent les transitions permises (à gauche, à droite, en haut ou en bas) d'une case blanche vers une autre case blanche adjacente. Un exemple de labyrinthe est donné dans la Figure A1.2 de l'ANNEXE 1 et la matrice d'adjacence **L** correspondante est donnée dans le fichier « *Labyrinthe.txt* ».

Pour résoudre cette catégorie de problème, nous utiliserons deux approches :

1. Exploration en profondeur du graphe.
2. Exploration en largeur du graphe.

L'exploration en profondeur consiste à suivre le premier lien d'un nœud qui donne accès à un nœud non-visité jusqu'au but ou jusqu'à un cul-de-sac. Si un cul-de-sac est atteint, alors on rebrousse chemin et on prend le prochain lien qui donne accès à un nœud non-visité. L'algorithme suggéré pour l'exploration en profondeur est donné dans l'ANNEXE 2 et peut nécessiter des modifications pour accommoder vos besoins. Notez que cet algorithme n'est pas optimal pour la recherche du plus court chemin. En fait, il n'est même pas sous-optimal et même s'il donne un chemin, ce chemin est souvent loin de l'optimum.

L'exploration en largeur, quant à elle, consiste à visiter chacun des liens d'un nœud et ensuite chacun des liens de nœuds adjacents au premier et ainsi de suite. La recherche en largeur peut produire un tableau contenant la plus petite distance entre le nœud de départ et chacun des nœuds du graphe. L'exploration en largeur peut aussi produire un tableau de chemins qui donne, pour chaque nœud, le nœud voisin faisant partie du parcours le plus court entre ce nœud et le nœud de départ. Pour le type de problème étudié dans cette section, cet algorithme est optimal et fournit donc le ou l'un des chemins optimaux.

- **TRAVAIL À FAIRE :**

Vous devez adapter les deux algorithmes à vos besoins et vous en servir pour résoudre les problèmes de la tour de Londres et du labyrinthe. À partir d'un graphe obtenu selon la méthode décrite dans la section II.1, votre programme demande à l'usager le nœud de départ et le nœud d'arrivée désirés ainsi que l'algorithme à utiliser. Le programme affiche ensuite le parcours du nœud de départ au nœud d'arrivée ainsi que la distance parcourue (en nombre de nœuds visités sur le parcours). Votre programme affiche aussi le temps de calcul.

Entrée :

- Fichier ou matrice d'adjacence générée aléatoirement.

Sortie :

- Nombre de nœuds visités
- Liste des nœuds visités
- Temps de calcul

II.3. Problème de carte routière

Le problème de recherche du plus court chemin entre deux villes à partir d'une carte routière, s'apparente à de nombreux problèmes d'optimisation. Il se présente sous la forme d'un graphe *pondéré*, orienté ou non. Chaque sommet représente une ville sur une carte. Les arcs entre les sommets une route menant d'une ville à une autre et la distance entre les deux villes selon cette route.

Pour résoudre cette catégorie de problème, nous allons utiliser trois approches :

1. Exploration en profondeur de type glouton,
2. Algorithme de Dijkstra, et
3. Algorithme de Floyd-Warshall.

Les deux premières approches sont de type *glouton* et la troisième est de type *programmation dynamique*. L'exploration de graphes en profondeur est mieux adaptée à ce problème que dans le cas du problème de la section II.2. En effet, le coût associé aux liens du graphe permet de diriger la recherche selon un principe *glouton* et, sans garantir de solution optimale, l'algorithme fournit une bonne solution. C'est donc une approche *sous-optimale.* L'algorithme de Dijkstra, bien que de type glouton et bien que les algorithmes gloutons ne garantissent habituellement pas l'obtention de la solution optimale, est quant à lui *optimal*. Finalement, l'algorithme de Floyd-Warshall, de type

programmation dynamique, garantit une solution optimale.

L'exploration en profondeur de type glouton est identique à l'exploration en profondeur utilisée dans la section II.2 à ceci près : plutôt que choisir le premier lien disponible vers un nœud non-visité, on choisit le lien le moins coûteux vers un nœud non-visité. Ainsi l'algorithme de la section II.2 peut aisément être adapté à ce problème. Le plus simple est de trier la liste des liens contenue dans chaque nœud selon le poids des liens (voir champs **PoidsLiens** et **IndexVoisins**). Ainsi, il suffit de sélectionner les liens dans l'ordre pour balayer du lien à plus faible coût au lien à coût plus élevé.

Un pseudocode pour l'algorithme de Dijkstra est fourni dans l'ANNEXE 2 (d'autres pseudocodes ainsi que des explications plus détaillées peuvent être dénichés dans des livres et sur Internet). Il consiste à extraire de la liste des nœuds celui qui est le plus près du nœud de départ, jusqu'à l'épuisement de la liste. Appelons ***S*** l'ensemble des nœuds qui ont été sélectionnés (extraits) et ***C*** l'ensemble des nœuds restants. Alors, le chemin le plus court entre les nœuds de ***S*** et le nœud de départ est connu. Le chemin le plus court entre un nœud de ***C*** et le nœud de départ est constitué de (1) le lien le plus court entre ce nœud et le plus proche nœud de ***S*** et (2) du chemin le plus court de ce dernier nœud jusqu'au nœud de départ. Le résultat est un tableau de chemins qui donne, pour chaque nœud, le nœud voisin faisant partie du chemin le plus court jusqu'au nœud de départ.

L'algorithme de Floyd-Warshall, dont un pseudocode est fourni dans l'ANNEXE 2 (d'autres pseudocodes ainsi que des explications plus détaillées peuvent être dénichés dans des livres et sur Internet), est une méthode tabulaire pour résoudre le problème du plus court chemin. Cet algorithme s'appuie sur le principe d'optimalité, qui stipule ici que si un nœud **k** *fait partie du chemin le plus court* (optimal) entre un nœud **i** et un nœud **j**, alors le chemin du nœud **i** jusqu'au nœud **k** et le chemin du nœud **k** jusqu'au nœud **j** doivent tous deux être optimaux. Ainsi, l'algorithme trouve le plus court chemin entre chaque paire de nœuds du graphe. Le résultat est un tableau de chemins entre des paires de nœuds. Ainsi, selon le tableau de chemins ci-dessous (Tableau 1), le chemin le plus court entre le nœud 0 et le nœud 2 passe par le nœud 3. Ensuite, le chemin le plus court entre le nœud 0 et le nœud 3 passe par le nœud 1, tandis que le chemin le plus court entre le nœud 3 et le nœud 2 est direct (puisque *Chemin[3,2] = -1*).

Tableau 1. Exemple de chemins les plus courts entre trois nœuds.

		Nœud de Départ			
		0	**1**	**2**	**3**
Nœud d'arrivée	**0**	-1	-1	3	1
	1	3	-1	3	-1
	2	-1	0	-1	-1
	3	-1	0	-1	-1

- **TRAVAIL À FAIRE :**

Vous devez adapter les trois algorithmes à vos besoins. À partir d'un graphe obtenu selon la méthode décrite dans la section II.1, votre programme demande à l'usager le nœud de départ et le nœud d'arrivée désirés ainsi que l'algorithme à utiliser. Le programme affiche ensuite le chemin du nœud de départ jusqu'au nœud d'arrivée ainsi que la distance parcourue (le coût du parcours). Votre programme doit afficher aussi le temps de calcul.

Entrée :

- Fichier ou matrice d'adjacence générée aléatoirement.

Sortie :

- Nombre de nœuds visités
- Liste des nœuds visités
- Coût total du parcours
- Temps de calcul

III. Rapport (30 %)

Le rapport est écrit **dans vos mots** et doit contenir les items suivants (les longueurs des sections sont proposées uniquement à titre indicatif)

1. *Introduction* (maximum 1 page)

Cette première section contient une description des buts du laboratoire et des objectifs visés ainsi que sert à introduire les sections qui suivent.

2. *Les algorithmes de recherche du plus court chemin* (environ 1 à 2 pages/algorithme)

La deuxième section explique le principe de fonctionnement de chaque algorithme et donne son pseudocode tel qu'il a été implémenté, **sans oublier de mentionner sa provenance** et en mettant l'accent sur les ajustements qui ont dû être apportés à la source originale pour les besoins particuliers de l'équipe. Elle donne aussi un court rapport sur les difficultés et autres curiosités rencontrées lors de l'implémentation.

3. *L'analyse théorique* (maximum 1 page par algorithme)

Cette section fournit le détail de l'analyse théorique de chaque algorithme et justifie les conclusions de l'analyse.

4. *Conclusion* (environ 1 à 2 pages)

Cette section établit des conclusions quant à l'atteinte des objectifs de départ. Il faut se focaliser particulièrement sur les avantages et inconvénients de chaque algorithme et leur utilité dans différents cas de figure.

5. *Références*

Cette dernière section doit énumérer toutes les références de vos sources. Dans le corps du rapport, vous devez également mettre un renvoi après chaque élément emprunté, ex. [1], [2], etc.

REMARQUE : Si une information provient de l'énoncé de laboratoire ou du matériel de cours, il n'est pas nécessaire de citer cette référence.

Si le document cité est un volume :

1. De Garmo, E.P., Sullivan, W.G. & Bontadelli, J.A. (1989). Engineering Economy (8e ed.). New York : MacMillan.

Si le document cité provient d'un site internet :

2. École de technologie supérieure. Politique d'éthique de la recherche avec des êtres humains, [En ligne]. http://www.etsmtl.ca/SG/Politique/polethsh.pdf (Consulté le 14 novembre 2000).

Si le document cité est un article de périodique :

3. Gargour, C.S., Ramachandran, V., Bogdadi, G. (1991). Design of Active RC and Switched Capacitor Filters Having Variable Magnitude Characteristics Using a Unified Approach. J. of Computers and Electrical Engineering, 17(1), 11-12.

De plus, vous devez fournir au chargé de laboratoire le code source de votre programme ainsi que vos fichiers de résultats de l'analyse expérimentale en version électronique.

NOTE IMPORTANTE : Un soin particulier doit être accordé au français, dans la rédaction de ce rapport. Les rapports mal écrits seront pénalisés jusqu'à concurrence de 10 %.

ÉCHÉANCIER[2] :

Semaine 1 :

- Lecture d'une matrice d'adjacence à partir d'un fichier.
- Génération automatique d'une matrice d'adjacence.
- Transformation de la matrice d'adjacence en une liste de nœuds.

Semaine 2 :

- Implémentation et validation des algorithmes de recherche en largeur et en profondeur ainsi que des algorithmes de plus court chemin (sections II.2 et II.3)

Semaine 3 :

- Poursuite de la validation et début de l'analyse.
- **Le fonctionnement du programme complet est démontré au chargé de laboratoire.**

Semaine 4 :

- Remise du rapport de laboratoire au début de la séance de laboratoire.

[2] L'échéancier est fourni à titre indicatif sauf pour la démonstration du code prévue à la troisième semaine du laboratoire.

Note importante concernant la démonstration de la troisième semaine :

Pour que la démonstration du code source de chaque groupe se passe dans les meilleures conditions possibles, voici quelques informations à propos de cette activité obligatoire :

- Une démonstration est requise par **groupe**.
- Chaque démonstration est **notée**. La note attribuée vaut 70% de la note globale du troisième laboratoire.
- Au moins une personne doit être présente durant la séance laboratoire de la troisième semaine pour faire la démonstration de son groupe. L'absence de tous les membres d'un groupe pourrait valoir la note zéro.
- Lors de la démonstration, le fonctionnement du programme tel que décrit dans le protocole d'implémentation (voir la section II « *Implémentation* », pp. 1-7) est validé. Une attention particulière sera accordée à l'implémentation des algorithmes d'exploration de graphes.
- Le chargé du laboratoire vous fournira **un ou plusieurs fichiers de type texte** pour la validation de votre programme au début de la troisième semaine du laboratoire. Les fichiers fournis respectent le format détaillé dans l'ANNEXE 3 de la page 13.

ANNEXE 1
ILLUSTRATIONS DES PROBLÈMES DE LA SECTION II.2

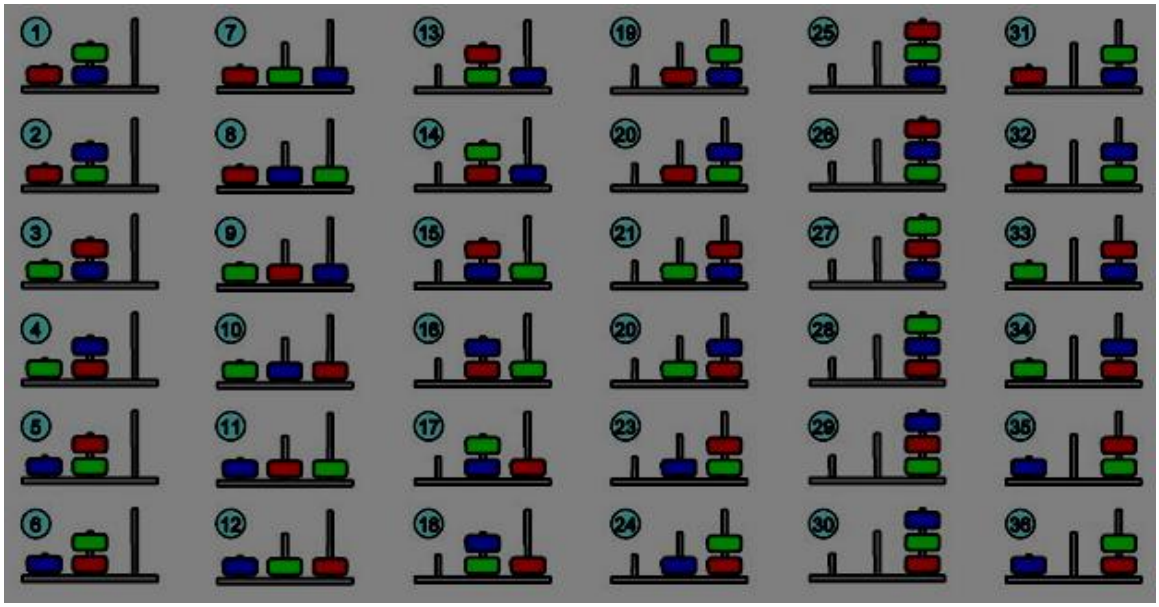


Figure A1.1. Configurations possibles pour les déplacements d'anneaux (problème TL).

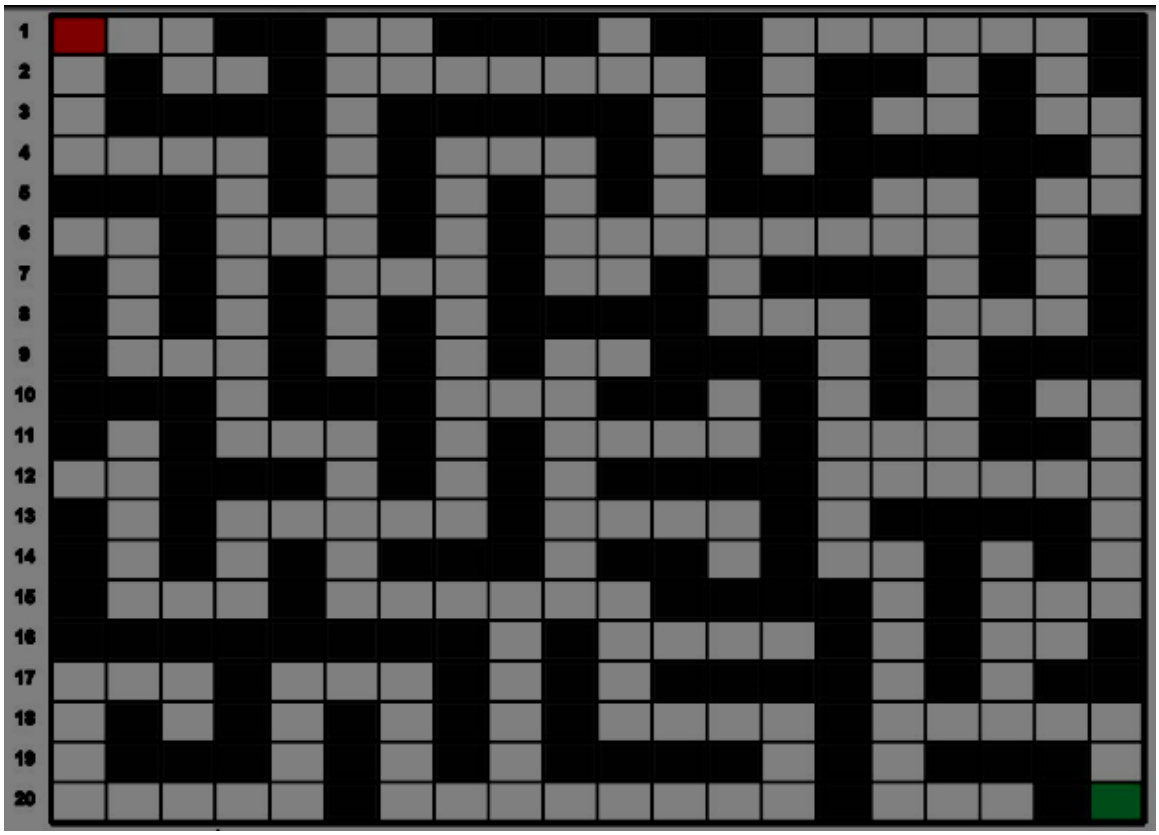


Figure A1.2. Exemple de labyrinthe (problème du labyrinthe).

ANNEXE 2
PSEUDOCODE DES ALGORITHMES

1. Pseudocode de l'algorithme de Parcours en Profondeur :

```
Parcours_Profondeur(A[0..N-1], Source, Destination, Chemin[0..N-1], p) : Booléen
       Si A[Source].visité
           Retourner Faux
       A[Source].visité = Vrai
       Trouvé = Faux
       k = 0
       Tant que (k ≤ A[Source].NVoisins) et (Pas Trouvé)
           v = A[Source].lien[k]
           Si (A[v].nom == Destination)
              Trouvé == Vrai
              Chemin[p+1] = A[v].nom
           Sinon
              Trouvé = Parcours_Profondeur(A, v, Destination, Chemin, p+1)
           Si Pas Trouvé
              k = k + 1
       Si Trouvé
           Chemin[p] =A[Source].nom
       A[Source].visité = Trouvé
```

2. Pseudocode de l'algorithme de Dijkstra : adapté à partir de :

- Brassard G., Bratley P., *Fundamentals of Algorithmics*, Prentice Hall, New Jersey, USA, 1996, p. 199 et
- Wikipédia : Algorithme de Dijkstra, en ligne, http://fr.wikipedia.org/wiki/algorithme_de_dijkstra, consulté le 29 octobre 2012.

```
Dijkstra(L[0..N-1,0..N-1], Source, Destination) : Chemin[0..N-1] D[0..N-1], Visité[0..N-1], Chemin[0..N-1]

       Pour n = 0 à N-1
           D[n] = L[Source,n]
           Chemin[n] = Source
           Visité[n] = FAUX
       Visité[Source] = VRAI

       nombreVisité = 1
       Tant que nombreVisité < N
           plusProche = sommet non visité avec le plus petit D
           Visité[plusProche] = VRAI
           nombreVisité = nombreVisité + 1
       Pour tous les prochainSommet dans la liste des voisins de plusProche
           Si D[prochainSommet] > D[plusProche] + L[plusProche, prochainSommet]
                 D[prochainSommet]    =    D[plusProche]    +    L[plusProche, prochainSommet]
      Chemin[prochainSommet] = plusProche
```

Dans ce pseudocode, **L** représente la matrice d'adjacence. **Visité** indique pour chaque sommet s'il a été visité ou non. Le tableau **D** contient la plus petite distance connue (à une

étape donnée de l'algorithme) de chaque sommet à la source. Le tableau **Chemin** est construit pour contenir le prédécesseur de chaque sommet dans le plus court chemin allant de la source à ce sommet, ce qui vous permet d'écrire vous-même la fonction **RetrouverChemin**.

3. Pseudocode de l'algorithme de Floyd-Warshall : adapté à partir de :

– Brassard G. et Bratley P., Fundamentals of Algorithmics, Prentice Hall, New Jersey, USA, 1996, pages 269-270)

```
Floyd-Warshall(L[0..N-1,0..N-1], Source, Destination) : Chemin[0..N-
1,0..N-1]
      D[0..N-1,0..N-1], Chemin[0..N-1,0..N-1]
      Pour i = 0 à N-1
          Pour j = 0 à N-1
             D[i,j] = L[i,j]
             Chemin[i,j] = -1

      Pour k = 0 à N-1
          Pour i = 0 à N-1
             Pour j = 0 à N-1
                Si (D[i,j] > D[i,k]+D[k,j])
                   D[i,j] = D[i,k]+D[k,j]
                   Chemin[i,j] = k
```

ANNEXE 3
FORMAT DU FICHIER SOURCE

Le format du fichier contenant la matrice d'adjacence relative à un graphe donné respecte les règles suivantes :

- La première ligne du fichier fournit **N**, le nombre de nœuds du graphe.
- Les lignes suivantes fournissent les rangées de la matrice **L**. Chaque rangée débute sur une nouvelle ligne dans le fichier.
- Le caractère de séparation entre deux éléments de la même ligne est la tabulation (i.e., « \t »).
- Le fichier est enregistré avec l'extension « **.txt* » (fichier texte standard).

La Figure A3.1 illustre le format du fichier tel qu'indiqué ci-dessus :

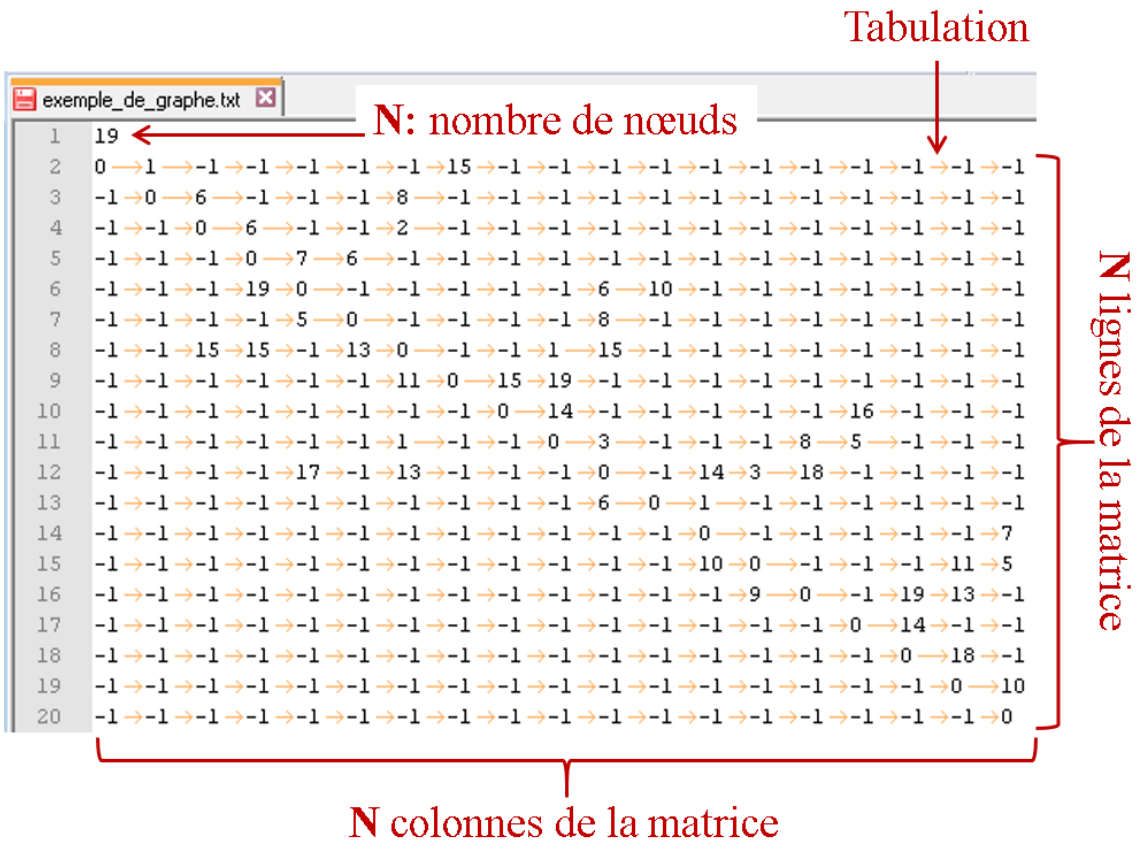


Figure A3.1. Illustration du format de fichier source.

ANNEXE 4
GRILLE DE CORRECTION DU LABORATOIRE

Rapport	
1. Introduction **3**	
2. Algorithmes **10**	
3. Analyse de performance **10**	
4. Conclusion **7**	
Annexes	
Références	
Total partiel **/ 30**	
Points Négatifs	
Rapport incomplet – Une section manquante -50% – Plus d'une section manquante : -10% additionnels par section.	
Références (max. -10%)	
Orthographe et grammaire (max. -10%)	
Présentation (max. -10%)	
Respect du gabarit (-10% si non respecté)	
Non-respect du protocole de livraison (jusqu'à -10%)	
Retard (-10% par jour)	
Note rapport **/30**	
Programmes	
Implémentation **/70**	
Points Négatifs	
Non-respect du protocole de livraison (jusqu'à -10%)	
Retard (-10% par jour)	
Note programmes **/ 70**	
Note finale laboratoire **/100**	