Algorithme de Dijkstra

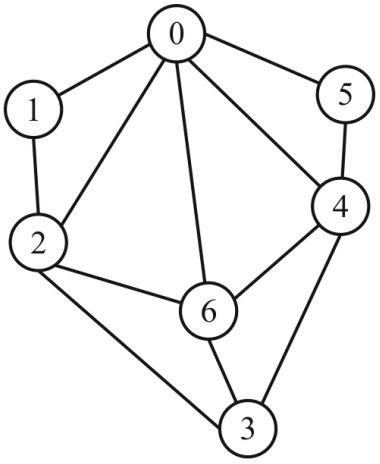
Cf. cours : "Graphes".

Coloration d'un graphe par la méthode de Welsh et Powell

La **coloration** d'un graphe consiste à colorier les sommets d'un graphe de sorte que deux sommets joints par une arête n'aient jamais la même couleur ; le but étant évidemment d'utiliser au total un minimum de couleurs. On a démontré qu'un graphe planaire (que l'on peut représenter sans que les arêtes ne se croisent) peut se colorer en utilisant au plus 4 couleurs.
La méthode décrite ici permet de déterminer un moyen de colorer un graphe avec assez peu de couleurs (mais ce n'est pas la coloration optimale). Pour cela, on introduit le **degré** d'un sommet du graphe : c'est le nombre d'arêtes qui partent de ce sommet. On procède alors de la façon suivante :

- On classe les sommets dans l'ordre décroissant de leurs degrés.
- En parcourant cette liste dans l'ordre, on attribue une nouvelle couleur au premier sommet non encore coloré ainsi qu'à tous les autres sommets non encore colorés et qui ne sont pas voisins d'un sommet de cette couleur.
- On recommence jusqu'à ce que tous les sommets soient colorés.

1. Appliquer cet algorithme pour colorer le graphe suivant :



2. Ecrire une fonction `degre(sommet)` qui prend en argument le numéro d'un sommet et qui renvoie son degré ; on supposera que le graphe est représenté par sa matrice d'adjacence `G`.

3. Ecrire une fonction `trier()` qui renvoie la liste des sommets du graphe, triés dans l'ordre des degrés décroissants.

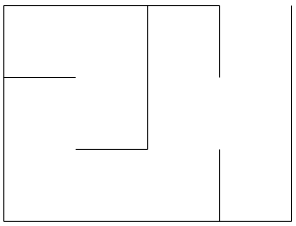
4. Ecrire une fonction `colorer()` qui renvoie une liste de la forme `[(sommet,couleur),...]` : on représentera les couleurs par des entiers naturels. Pour cela, on pourra utiliser la méthode `remove(elt)` qui permet de supprimer `elt` d'une liste.

Sortir d'un labyrinthe

On peut modéliser un labyrinthe, de taille $n \times m$, à l'aide d'un graphe (puis de sa matrice d'adjacence par exemple) de la façon suivante :

- Toutes les cases du labyrinthe sont numérotées de 0 à $nm-1$ et constituent les sommets du graphe.
- Si on peut passer d'une case à une autre (donc si elles sont voisines et ne sont pas séparées par un mur), on place une arête entre les sommets correspondants ; il y a donc au maximum 4 arêtes par sommet.

1. Déterminer le graphe correspondant au labyrinthe suivant puis sa matrice d'adjacence.



2. Ecrire une fonction `numero(i,j)` qui prend en argument les coordonnées `i,j` d'une case du labyrinthe, $0 \leqslant i \leqslant n-1$ et $0 \leqslant j \leqslant m-1$, et qui renvoie le numéro du sommet correspondant dans le graphe. Ecrire de même une fonction `case(k)` qui prend en argument le numéro `k` d'un des sommets du graphe, $0 \leqslant k \leqslant nm-1$, et qui renvoie les coordonnées de la case du labyrinthe correspondant.

3. En adaptant la méthode de parcours en profondeur d'un graphe vue en cours, écrire une fonction `sortir(i,j)` qui prend en argument les coordonnées d'une case du labyrinthe et qui renvoie un chemin permettant de sortir du labyrinthe (la sortie est la case `(0,m-1)` et on supposera qu'il existe bien un moyen de sortir). Comme le parcours en profondeur revient en arrière lorsqu'il ne trouve pas de nouveau voisin à un sommet, on utilisera une liste supplémentaire `chemin` pour mémoriser le chemin de sortie ; il suffit, lorsque le parcours en profondeur remonte à la bifurcation précédente (ie lorsqu'on dépile) d'effacer les dernières cases du `chemin`.

4. Expliquer pourquoi le parcours en profondeur correspond plus à une tentative concrète que le parcours en largeur.