V 1.03

C++ PoP – Sections Electricité et Microtechnique

Printemps 2020 : *Archipelago* © R. Boulic & collaborators

Un outil interactif pour structurer une ville flottante

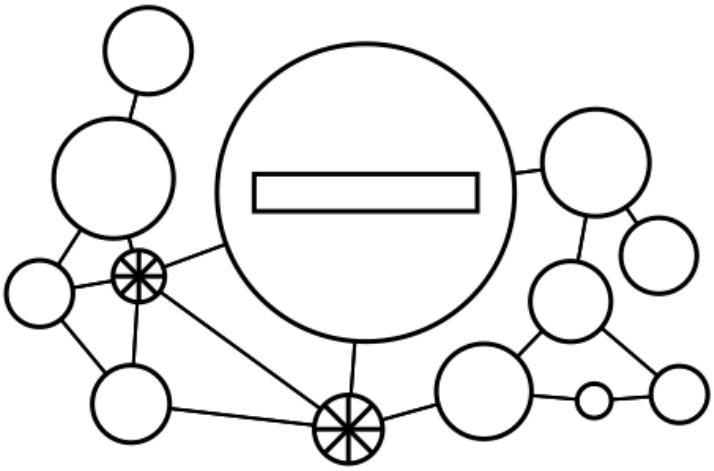

Fig 1 : exemple de ville comportant 10 nœuds logement, 2 nœuds de transport et un nœud de production

1. Introduction

Ce projet est indépendant de celui du semestre dernier. Le lien reste néanmoins la mise en œuvre des grands principes (*abstraction, ré-utilisation*), les conventions de présentation du code et les connaissances accumulées jusqu'à maintenant dans ce cours. Le but du projet est surtout de se familiariser avec deux autres grands principes, celui de *séparation des fonctionalités* (*separation of concerns*) et celui *d'encapsulation* qui deviennent nécessaires pour structurer un projet important en *modules* indépendants.

Nous mettrons l'accent sur le lien entre *module* et *structure de données*, et sur la robustesse des modules aux erreurs. Par ailleurs *l'ordre de complexité* des algorithmes sera testé avec des fichiers tests plus exigeants que les autres.

Vous pouvez faire plus que ce qui est demandé dans la donnée mais n'obtiendrez aucun bonus ; notre but est d'éviter que vous passiez plus de temps que nécessaire pour faire ce projet au détriment d'autres matières.

Dans tous les cas, vous êtes obligé de faire ce qui est demandé *selon les indications de la donnée et des documents des rendus*. Vos éventuelles touches personnelles ne doivent pas interférer avec les présentes instructions.

Le projet étant réalisé par groupes de deux personnes, il comporte un oral final individuel noté pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de l'ensemble du projet. Une performance faible à l'oral peut conduire à un second oral approfondi et une possible baisse individuelle des notes des rendus.

La suite de la donnée indique les variables en ***italique gras*** et les constantes globales en **gras** (la valeur des constantes est disponible dans le fichier **constantes.h** fourni en Annexe A. On utilisera toujours la *double précision* pour les données de position dans l'espace et les calculs en virgule flottante.

Pitch du sujet : à cause d'une élévation annoncée du niveau de la mer, une cité-état établie en bordure de l'océan décide de se transformer en ville flottante. Vous allez mettre au point un outil pour évaluer un plan de ville dans lequel chaque quartier est une île circulaire et les quartiers sont reliés par des ponts flottants. Certains critères d'efficacité seront évalués systématiquement.

2. Modélisation du problème

Le but de ce projet est de mettre au point un outil permettant d'éditer le plan d'une ville en plaçant différents types de quartiers dans l'espace 2D. On se concentre sur l'efficacité des déplacements tout en cherchant un certain équilibre entre les différents types de quartier et en estimant les coûts d'infrastructure.
On utilise deux abstractions pour modéliser et évaluer la ville flottante :

- **Géométrie : pour tester la répartition correcte des quartiers et des connexions**
  La ville est définie dans un système de coordonnée monde d'origine (0,0) ; X est l'axe horizontal, orienté positivement vers la droite, et Y est l'axe vertical, orienté positivement vers le haut. L'unité est le m. L'espace initial est délimitée par [-**dim_max, dim_max**] selon X et Y mais rien n'empêche ensuite de changer la taille de cet espace avec la souris et le clavier (section 6).
  L'abstraction géométrique est le moyen de vérifier les ***règles de non-superposition*** (section 2.1.3) entre deux quartiers et entre une connexion et un quartier. Pour cela on adopte cette approche :
  - Un quartier est représenté par un cercle dont le rayon dépend du nombre de personnes
  - Une connexion entre deux quartiers est représentée par un segment de droite.

- **Graphe : pour déterminer l'efficacité du plan proposé**
  - Un quartier est un nœud du graphe
  - Une connexion entre deux quartiers est un lien entre deux nœuds du graphe.
    Grâce à l'algorithme de ***parcours de graphe de Dijkstra*** on peut déterminer le temps de parcours le plus court d'un quartier à un autre. Un second avantage du parcours de graphe est un coût calcul plus faible que l'algorithme de Floyd vu en ICC sur une table de distances. C'est pourquoi nous exigeons de ne PAS utiliser l'algorithme de Floyd pour ce projet[1].

2.1 Représentation des quartiers et des connexions

2.1.1 Caractéristiques communes à tous les quartiers

Tous les quartiers sont identifiés par un code postal (entier positif) constituant leur identificateur unique[2] ***uid***. De même ils ont une capacité exprimée en nombre de personnes ***nbp*** dans l'intervalle [**min_capacity, max_capacity]**. Etant de forme circulaire, ils sont tous caractérisés par la position ***(x,y)*** de leur centre et leur ***rayon*** (en **m**) qui dépend de leur capacité comme suit : ***rayon = sqrt(nbp).***

2.1.2 Caractéristiques spécifiques aux quartiers et aux connexions

On considère trois types de quartier spécialisés (Fig2): **Logement** (au plus **max_link** connexions), **Transport** (connexion de vitesse **fast_speed** entre 2 quartiers de ce type, sinon vitesse **default_speed** pour toutes les autres connexions), **Production** (pas de transit possible).

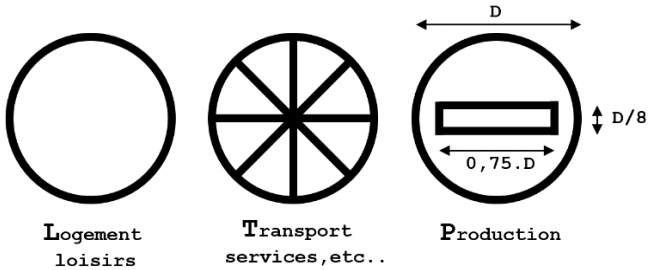

Figure 2 : représentation des 3 types de quartier.

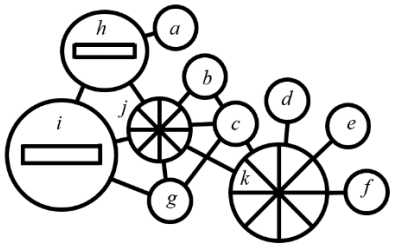

Figure 3 : exemple de ville avec les 3 différents types de quartiers et les connexions existantes.

Une connexion entre deux quartiers est le segment de ligne droite reliant leurs centres ; sa longueur est la distance entre les 2 quartiers. Il est dessinée dans l'espace en dehors des quartiers (Fig3, ex : entre a et h).

[1] La seconde raison est que l'algorithme de Floyd a été utilisé pour le projet de l'an dernier.
[2] La seule valeur interdite comme identificateur unique est **no_link** qui est utilisée pour la recherche de chemin.

2.1.3 Contraintes géométriques à respecter

Règle1 : aucune superposition entre les quartiers.

Ex : le nouveau quartier x ne peut pas être créé car il se superpose avec le quartier existant b

Règle2 : aucune superposition entre une connexion et un quartier

Ex : la connexion entre les quartiers a et i ne peut pas être créée car elle se superpose avec le quartier h

Note : deux connexions peuvent se croiser.

Ex : entre c et g. Cependant cela ne permet pas de tourner pour aller directement de g à k ; il faut d'abord passer par c ou par j.

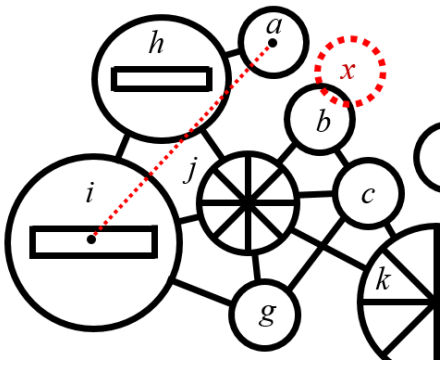

Figure 4 : deux cas interdits

Attention point technique concernant les tests de superposition:

Nous utilisons un fichier formaté pour mémoriser une ville (section 4). Cela peut introduire une petite perte de précision. Pour éviter les problèmes de fausse détection de superposition, il faut procéder comme suit :

Tests effectués à la lecture d'un fichier ;

- il y a collision entre 2 quartiers si :

  distance séparant 2 quartiers ≤ somme des rayons (1)

- il y a collision entre une connexion de U vers V et un quartier W si :

  distance minimale séparant le segment UV du centre du quartier W ≤ rayon de W (2)

Tests effectués pendant l'utilisation de l'application pour créer et/ou de déplacer un nouveau quartier ou pour créer une nouvelle connexion entre des quartiers existants (sections 5-6).

- Dans ce contexte il faudra indiquer introduire une marge supplémentaire de sécurité **dist_min** pour éviter les cas tangents (Fig 5); il faudra interdire la création et/ou le déplacement d'un quartier en cas de collisions définies par :

  distance séparant 2 quartiers ≤ somme des rayons **+ dist_min** (3)

- Il faut interdire la création de la connexion de U vers V s'il existe un quartier W tel que :

distance minimale séparant le segment UV du centre du quartier W ≤ rayon de W **+ dist_min** (4)

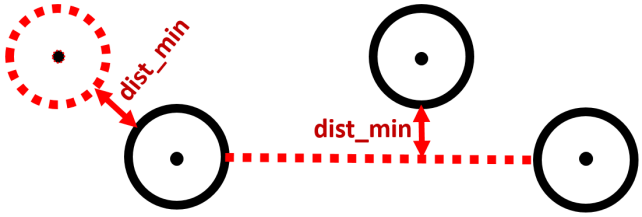

Fig 5 : prise en compte de la marge de sécurité **dist_min**

2.1.3 Contraintes sur la structure du graphe

Un nœud de type **Logement** peut avoir au maximum **max_link** connexions (section 2.1.2).
Les deux autres types de nœuds ne sont pas limités en nombre de connexions.

Un nœud de type **Production** ne peut pas être utilisé comme nœud intermédiaire pour relier deux autres nœuds (section 2.1.2). C'est pour cette raison que son dessin ressemble à un panneau de sens interdit…
Les deux autres types de nœuds n'ont pas cette limitation.

2.2 Critères d'évaluation d'une ville

Ici nous supposons que les contraintes géométriques et les contraintes sur la structure du graphe sont satisfaites ; nous nous concentrons sur trois mesures permettant d'évaluation la qualité d'un plan proposé.

2.2.1 Critère *ENJ* d'Equilibre Nuit-Jour

Le critère ***ENJ*** évalue la différence de capacité entre d'une part l'ensemble des quartiers dédiés au repos (**L**ogement) et d'autre part l'ensemble des quartiers dédiés à une activité diurne (**T**ransport/service et **P**roduction). Pour que cet indicateur ait un sens, la différence est normalisée par le total de la *capacité* de tous les nœuds. S'il n'y a aucun quartier, le critère ***ENJ*** vaut zéro.

$$Critere\ ENJ = (\Sigma(nbp(L)) - \Sigma(nbp(T) + nbp(P))) / \Sigma(nbp(L) + nbp(T) + nbp(P)) \quad (5)$$

2.2.2 Critère *CI* du Coût de l'Infrastructure

Le critère ***CI*** évalue le coût de l'infrastructure représentée par l'ensemble des connexions. On considère que ce coût est proportionnel à la *distance* de la connexion, à la *vitesse moyenne* et à la *capacité* de la connexion.

La vitesse moyenne est **default_speed** sauf pour une connexion entre deux nœuds de **T**ransport ; elle vaut alors **fast_speed**.

$$Critere\ CI = \Sigma (dist(connexion) * capacité(connexion) * vitesse(connexion)) \quad (6)$$

La *capacité* d'une connexion est la plus petite capacité des 2 nœuds reliés par cette connexion.

2.2.3 Critère MTA du Meilleur Temps d'Accès

Le critère ***MTA*** est une moyenne calculée pour tous les nœuds **Logement**. Pour chaque nœud Logement, on évalue la somme du *meilleur* temps d'accès à un nœud **Transport** et du *meilleur* temps d'accès à un nœud **Production**.

Le temps est évalué en faisant la somme des temps individuels passés sur la suite des connexions pour atteindre le nœud de Transport le plus proche et le nœud de Production le plus proche.

Le temps passé sur une connexion individuelle est donné par ~~le produit~~ la division de la distance de cette connexion par la vitesse moyenne de cette connexion. La vitesse moyenne est **default_speed** sauf pour une connexion entre deux nœuds de transport ; elle vaut alors **fast_speed**. Si, pour une raison ou une autre il n'existe pas de chemin vers un nœud Transport ou un nœud Production alors on utilisera la constante **infinite_time**. Rappel : on ne peut pas traverser un nœud **Production**. S'il n'y a aucun quartier, le critère ***MTA*** vaut zéro.

$$Critere\ MTA = Moyenne (MTA(Logement \rightarrow Transport) + MTA(Logement \rightarrow Production)) \quad (7)$$

Pour un nœud Logement de départ **d**, la recherche du meilleurs temps d'accès est faite à l'aide d'une variante de l'algorithme de **Dijkstra** qui part de **d** et progresse de nœud en nœud dans le graphe en avançant en priorité selon le parcours ayant *le plus faible temps d'accès*.

2.3 Recherche du meilleur chemin avec l'algorithme de Dijkstra

Cette section indique comment modéliser un nœud du graphe et fournit le pseudocode permettant de déterminer le meilleurs chemin d'un nœud de départ de type Logement vers un nœud de type Production (vous devrez l'adapter pour aussi trouver le meilleur chemin vers un nœud de type Transport).

2.3.1 Structure de donnée pour représenter un nœud du graphe.

D'un point de vue algorithmique, une seule structure de donnée de type Nœud suffit pour représenter un graphe en mémorisant :

- un identificateur unique ***uid***
- son ***type*** (Logement / Transport / Production), ce qui est aussi réalisable par héritage. L'algorithme de Dijsktra présenté ci-dessous utilise seulement le fait qu'on cherche un nœud de type Production pour déterminer la fin de l'algorithme avec succès.
- l'**ensemble *V* des liens vers les nœuds voisins auxquels il est directement connecté**. Notre plan de ville est un réseau tout comme le réseau d'amis vu au premier semestre. Pour le pseudocode, on représente l'ensemble ***V*** avec un tableau de liens[3] vers les nœuds voisins.

2.3.2 Principe de l'algorithme de Dijkstra pour le meilleur chemin vers un nœud Production.

Pour que l'algorithme de recherche du meilleur chemin fonctionne correctement on doit mémoriser trois informations supplémentaires dans chaque Nœud :

- Un booléen ***in*** indiquant si le Nœud fait partie des noeuds à traiter par l'algorithme (init à **true**)
- Une valeur ***access*** du temps d'accès cumulé depuis le nœud logement de départ (init à **infinite_time**).
- Un lien vers un Nœud ***parent*** qui permet de mémoriser le nœud précédent dans le chemin (initialement **no_link**). Cette information permet de remonter du nœud trouvé jusqu'au départ.

Note : on utilise l'opérateur . pour accéder à ou modifier une information d'un nœud

Hypothèse de travail : un tableau **TN** contient les N nœuds de la ville ; leur valeur ***access*** est initialisée à **infinite_time** sauf pour le nœud de départ qui a une valeur nulle. Le nombre d'éléments de cet ensemble **TN** n'est pas modifié par l'algorithme ; tous les nœuds restent présents et conservent un indice constant dans **TN**. En effet, ce sont ces indices qui servent pour représenter les liens vers les nœuds voisins ; il est donc important de garantir cette propriété d'invariance.

```
Input    : Tableau TN de l'ensemble des Nœuds, indice d du nœud de départ de type Logement
Output : un lien vers le Nœud Production le plus proche du nœud de départ (ou no_link s'il n'y en a pas)

Pour tous les éléments de TN
    Initialisation de in à true, access à infinite_time, et parent à no_link
TN(d).access ← 0.

Tableau TA des indices de nœuds dans TN, trié selon l'ordre croissant de TN(TA(i)).access

Tant que TN est non-vide
    n  ← find_min_access(TA, TN)    // trouve l'indice n du nœud avec le plus faible temps d'accès et in à true
    Si TN(n).type = Production
        Sortir  n                   // fin avec succès

    TN(n).in ← false                // sort symboliquement le nœud d'indice n de TN
    Pour chaque lien lv de  TN(n).V
        Si TN(lv).in = true         // le nœud voisin fait encore partie de TN ; il doit être traité
            alt ← TN(n).access + compute_access(TN,n,lv)
            Si (TN(lv).access  > alt )   // on vient de trouver un chemin plus court pour le noeud lv
                TN(lv).access  ← alt
                TN(lv).parent ← n
                sort (TA, lv)       // mise à jour de la position du lien lv dans TA

Sortir no_link      // fin sans trouver un seul nœud Production
```

[3] La notion de lien utilisée pour ce pseudocode correspond à un indice dans un tableau

Idée au cœur de l'algorithme :
Le second ensemble **TA** de taille N contient seulement les indices des éléments de **TN**. L'ensemble **TA** est ce qu'on appelle une **file d'attente de priorité** car il doit être trié selon la valeur ***access*** de chaque élément **TN(TA(i))**. Nous supposons dans ce pseudocode que le tri est fait dans l'ordre croissant.

Dans une boucle sur **TA** on cherche l'indice **n** du nœud de **TN** dont la distance ***access*** est la plus faible ; on *enlève symboliquement* le nœud correspondant de **TN** en faisant passer son booléen ***in*** de true à false. Si le nœud est un nœud Production, l'algo est terminé.

Sinon, on parcourt l'ensemble des liens vers les nœuds voisin. Pour chaque lien on examine si le nœud voisin est encore dans **TN** (i.e. si son booléen ***in*** est true). Si la valeur ***access*** du voisin est plus grande que celle du nœud d'indice **n** à laquelle on ajoute la valeur du temps d'accès entre les 2 noeuds alors on a trouvé qu'il est plus rapide de passer par le nœud d'indice **n** pour atteindre le nœud voisin !

Dans ce cas, on remplace la valeur ***access*** du voisin par cette valeur plus courte et **n** devient le ***parent*** du voisin. Enfin, très important, **TA** est re-trié du fait que la valeur ***access*** de ce nœud voisin est maintenant plus courte. Ce tri est limité à trouver le nouvel emplacement de l'indice du voisin dans **TA** (se souvenir de l'algorithme par insertion vu en ICC).

3 Actions à réaliser par l'éditeur de ville

Le but du programme est de pouvoir réaliser les actions suivantes :

- **Lecture** d'un fichier pour initialiser l'état de la ville (section 4).
- **Ecriture** d'un fichier décrivant l'état actuel de la ville (section 4)
- **Editer** une ville = **ajouter**, **enlever**, **modifier** interactivement un nœud ou un lien (section 5)

Les actions de **lecture** et **d'édition** doivent mettre en œuvre des tests pour garantir que les contraintes indiquées en section 2.1 sont respectées. En cas d'erreur à la lecture, il faut afficher dans le terminal les messages d'erreurs que nous vous fournissons (section 8). En cas d'erreur pendant la définition ou la modification d'un nœud ou d'un lien dans la fenêtre graphique, celui-ci n'est pas créé ou modifié (section 6).

De plus, les critères d'évaluation (section 2.2) doivent être évalués et affichés dans l'interface graphique du programme à la lecture d'un nouveau fichier, ou à chaque modification une ville (sections 5-6).

4. Sauvegarde et lecture de fichiers tests : format du fichier

Votre programme doit être capable d'initialiser l'état de la ville à partir d'un fichier texte. Il doit aussi pouvoir mémoriser la configuration actuelle de la ville dans un fichier texte également. Cela vous permettra de pouvoir créer vos propres scénarios de tests avec un éditeur de texte comme geany.

Caractéristiques de fichiers tests :
Le nombre maximum de caractères par ligne est de **max_line**.
Les lignes vides commençant par **\n** ou **\r**, les commentaires commençant par **#** précédé éventuellement d'espaces, et les espaces avant ou après les données doivent être ignorés ; *il peut y en avoir un nombre différent d'un fichier à un autre*. Les fins de lignes peuvent contenir **\n** et/ou **\r** à cause du système d'exploitation sur lequel le fichier a été créé ; votre programme doit pouvoir traiter ces cas (cf série fichier).

Le format de fichier est décrit dans le tableau suivant. Il contient d'abord les données de chaque type de nœud (Logement, Transport, Production) puis l'ensemble des données des liens. Pour chaque type de nœud, on indique le nombre de nœuds sur une ligne de fichier. Ensuite il doit y avoir une ligne de fichier par nœud. Le format indique le nombre et la nature des données. Si plus de données sont fournies sur la ligne de fichier elles sont simplement ignorées. Un commentaire peut aussi suivre les données.

Un lien est représenté par la paire des identificateurs uniques des deux nœuds qu'il connecte. Seule la valeur **no_link** n'est pas autorisée comme valeur d'identificateur unique.

Voici le format général	remarques

`# Nom du scenario de test`
`#`
`nbNodeL  # seulement 2 3 pour cet exemple`
`uid0 x0 y0 nbp0`
`uid1 x1 y1 nbp1`
`uid2 x2 y2 nbp2`
 	
`nbNodeT  # seulement 1 pour cet exemple`
`uid3 x3 y3 nbp3`
 	
`nbNodeP  # seulement 1 pour cet exemple`
`uid4 x4 y4 nbp4`
 	
`nbLink  # seulement 6 pour cet exemple`
`uid0 uid3`
`uid1 uid3`
`uid2 uid3`
`uid0 uid1`
`uid1 uid2`
`uid3 uid4`
	Il y a 3 zones pour les nœuds, dans cet ordre : logement, puis Transport, puis Production.
	
	Chaque zone commence par le nombre de nœud suivi par la liste des nœuds.
	
	Pour chaque noeud (L ou T ou P) on indique son indentificateur unique (unsigned int), sa position (double): x et y, sa capacité (unsigned int).
	
	Il y a un noeud par ligne.
	
	Une dernière zone indique le nombre de liens suivi par la liste des liens, un par ligne.
	
	Chaque lien est défini par la paire d'identificateurs uniques des 2 nœuds reliés par ce lien.

5. Description de l'interface utilisateur (GUI)

Nous utilisons une seule fenêtre graphique divisée en deux parties:

- l'interface graphique utilisateur = GUI (Fig 7)
- le dessin de l'état courant de la ville dans un *canvas* (Fig 1).

L'interface utilisateur doit contenir (Fig 7):

Commandes générales :

- **Exit :** quitte le programme de jeu
- **New :** efface complètement la ville courante, on obtient un écran blanc.
- **Open** : remplace la ville par le contenu du fichier dont le nom est fourni. En cas d'erreur détectée à la lecture, les structures de données sont supprimées, ce qui produit un écran blanc.
- **Save** : mémorise l'état actuel de la ville dans le fichier dont le nom est fourni.

Commandes/Options de visualisation :

- **Shortest path** : option indiquant si on veut voir le plus court chemin
- **Zoom in / Zoom out / zoom reset** : ajustent le facteur de zoom (cf 6.1.7)
- Affichage du facteur de zoom courant (section 6.1.7)

Edition de la ville

- **Edit link** : option indiquant si on veut créer/supprimer un lien
- Indication du prochain type du nœud qui sera créé avec la souris

Affichage des trois critères ENJ, CI et MTA pour l'état actuel de la ville.

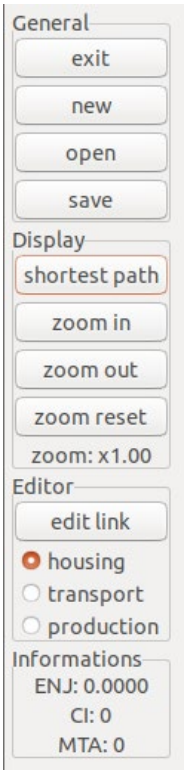

Fig 7 : Interface graphique (GUI)

L'activation des options **shortest_path** et **edit_link** utilise un **toggleButton** qui mémorise que l'option est active :

- Si le bouton **shortest_path** est actif et qu'un nœud Logement est sélectionné en tant que nœud courant alors il faut dessiner les plus courts chemins entre le nœud courant et 1) le nœud Transport le plus proche et 2) le nœud Production le plus proche. La couleur du dessin est indiquée en section 6.
- Si le bouton **edit_link** est actif alors le fait de cliquer sur le bouton gauche de la souris sert à la création d'un lien entre le nœud courant le nœud cliqué (section 6.1.5).

Le choix du type de prochain nœud créé est fait à l'aide d'un **radioButton** avec 3 options mutuellement exclusives (Logement, Transport, Production).

6. Affichage et interaction dans la fenêtre graphique

A partir des rendus 2 et 3, l'exécution du programme ouvre une fenêtre GTKmm contenant l'interface graphique utilisateur (Fig 7) le dessin de la ville dans un *canvas,* qui couvre l'espace [-**dim_max, dim_max**] selon X et Y.

Taille de la fenêtre d'affichage en pixels : La taille initiale du *canvas* dédié au dessin de la ville est de **default_drawing_size** en largeur et en hauteur. Elle peut changer durant l'exécution du programme. Un changement de taille de fenêtre ne doit pas introduire de distorsion dans le dessin (un cercle reste un cercle, quelle que soit la taille et la proportion de la fenêtre). La Fig1 montre un exemple d'affichage du jeu. La Fig2 montre la convention de représentation ; il faut respecter les tailles des éléments.

Les couleurs : Le dessin est en **noir** sur fond **blanc** par défaut. Le **nœud courant** est dessiné en **rouge**. Si le bouton **shortest_path** est actif et que le nœud courant est un Logement, alors ce nœud courant reste dessiné en **rouge** tandis qu'on dessine les plus courts chemins en **vert**, c'est-à-dire l'ensemble des nœuds et des liens entre le nœud courant et 1) le nœud Transport le plus proche et 2) le nœud Production le plus proche.

6.1 Interaction avec la souris et le clavier

Les manipulations décrites ci-dessous doivent respecter les conditions détaillées en section 2.1.3.
Le concept de **noeud courant** est nécessaire pour effectuer certaines opérations comme la création de liens.

6.1.1 Création d'un nœud et définition de ses paramètres:

S'il n'y a pas de nœud courant et qu'on clique dans la fenêtre de dessin avec le bouton **gauche** de la souris en dehors *de tout noeud*, alors cette position indique le centre d'un nouveau nœud dont le type est indiqué par le GUI (Fig 7). Ce nouveau nœud a la capacité **min_capacity** et le rayon correspondant par défaut. Il n'est relié à aucun autre nœud.

6.1.2 Sélection/Destruction/Dé-sélection du nœud courant:

Lorsqu'on clique dans la fenêtre de dessin avec le bouton **gauche** de la souris, et que le point cliqué est à l'intérieur d'un nœud existant alors ce nœud devient le nœud **courant**. Si on clique avec le bouton **gauche** dans le nœud courant, celui-ci est *détruit*. S'il y a un nœud courant et qu'on clique avec le bouton **gauche** en dehors *de tout noeud*, il n'y a plus de nœud courant.

6.1.3 Modification de la capacité du nœud courant:

Lorsqu'on appuie sur le bouton **gauche** en dehors de tout noeud, on peut modifier la capacité du nœud courant. Pour cela il faut *continuer d'appuyer sur le bouton gauche* et déplacer la souris. Quand on relâche le bouton, la variation de rayon est traduite en une modification de la capacité du nœud compatible avec l'intervalle [**min_capacity, max_capacity**] du nœud courant (Fig 8). L'échelle est donnée par la relation entre capacité et rayon d'un nœud (section 2.1.1) selon la formule :

nbp = (rayon_courant +(rayon_fin - rayon_debut))²

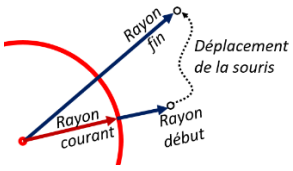

Fig 8 : les variables impliquées dans la mise à jour de la capacité d'un noeud

6.1.4 Déplacement du nœud courant:
Si on clique avec le bouton **droit**, le centre du nœud courant se déplace à l'endroit cliqué (comme vous préférez, soit au moment où presse ou alors au moment où on relâche).

6.1.5 Création d'un lien:
Il faut qu'un nœud courant soit sélectionné et que l'option **edit_link** soit aussi active. Si on clique avec le bouton **gauche** sur un nœud différent du nœud courant, alors un lien est créé entre le nœud courant et le nœud cliqué (pas plus d'un lien entre deux nœuds). Si l'un des deux nœuds est un nœud Logement il faut vérifier le nombre maximum de liens autorisé (2.1).

6.1.6 Destruction d'un lien:
Il faut qu'un noeud courant soit sélectionné et que l'option **edit_link** soit aussi active. Si on clique avec le bouton **gauche** dans l'un des *nœuds voisin* du nœud courant, alors le lien entre ces deux nœuds est supprimé.

6.1.7 Gestion du zoom in/out/reset :
A l'initialisation la fenêtre d'affichage couvre l'espace [-**dim_max, dim_max**] selon X et Y pour une fenêtre de rapport largeur/hauteur (*aspect ratio*) valant 1. Les calculs de zoom-in/out s'effectuent en conservant le centre **C** de la fenêtre immobile. Le *facteur de zoom* vaut initialement 1. ; il divise la largeur de l'espace visualisé dans le canvas :

Zoom-in : la frappe de '**i**' incrémente le *facteur de zoom* de **delta_zoom** mais pas plus que **max_zoom**.
Zoom-out : la frappe de '**o**' décrémente le *facteur de zoom* de **delta_zoom** mais pas moins que **min_zoom**.
Zoom reset : Si on veut revenir au *facteur de zoom* unitaire, il suffit de frapper sur la touche '**r**'.

Les trois boutons de même nom du GUI permettent de réaliser les mêmes commandes (Fig 7).

7. Syntaxe d'appel et répartition du travail en 3 rendus notés
Votre exécutable doit s'appeler **projet**. On doit pouvoir lui transmettre un argument optionnel sur la ligne de commande : un nom de fichier de test. Ex : `./projet test1.txt`

Rendu1 : Ce rendu SANS GTKmm sera toujours testé en indiquant un nom de fichier de test sur la ligne de commande. Il aura un poids inférieur aux deux suivants. Son architecture est précisée par la Fig 11a.
Ce rendu vérifie les points suivants à la lecture du fichier de test :
- Absence de collision entre nœuds ou entre nœud et lien =>
- Cohérence du graphe, dont nombre max de liens « Logement »

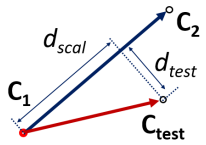

Fig 9 : éléments pour tester une collision le nœud de centre C_{test} et le lien reliant C_1 à C_2

Rendu2 : Ce rendu et le suivant utilisent toujours GTKmm (avec l'architecture de la Fig 11b). Si un nom de fichier est indiqué sur la ligne de commande il doit être ouvert pour initialiser l'interface graphique et le dessin, incluant :
- Affichage de la valeur initiale des 3 critères ENJ, CI et MTA.
- l'algorithme de Dijkstra est inclus dans ce rendu pour le calcul du critère MTA

Ce rendu sera testé en effectuant plusieurs lecture/écriture/relecture avec le GUI pour vérifier que l'affichage est bien correct. Avant de commencer la lecture d'un fichier, il faut ré-initialiser les structures de données et libérer la mémoire. On utilisera des *stubs* dans le module **ville** pour simuler les boutons du rendu3.

Rendu3 : Edition, performances, rapport.
- Action d'ajout et de retrait de nœud ou de lien
- Déplacement du nœud courant
- Gestion de changement de taille de fenêtre, de facteur de zoom.

Ce rendu sera testé en lançant des scénarios de test de complexité croissante pour estimer les performances de votre approche.

8. Architecture logicielle

8.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 10 décrit l'organisation minimum du projet en sous-systèmes avec leur responsabilité (Principe de Séparation des Fonctionnalités) :

- **Sous-système de Contrôle : son but est de gérer le dialogue avec l'utilisateur.** Si une action de l'utilisateur impose un changement de l'état de la ville, ce sous-système doit appeler une fonction du **sous-système du Modèle** qui est le seul responsable de gérer les structures de données de la ville (voir point suivant).
  Le sous-système de contrôle est mis en œuvre avec deux modules :
  - Le module **projet** qui contient la fonction **main** : son rôle est modeste car il est responsable de traiter l'éventuels argument fourni sur la ligne de commande au lancement du programme. Pour le rendu1, le sous-système de **Contrôle** ne contient que le module **projet**.
  - le module **gui** est créé à partir du rendu2 pour rassembler pour gérer le dialogue utilisateur à l'aide de l'interface graphique mise en œuvre avec GTKmm.

- **Sous-système du Modèle**: est responsable de gérer les structures de données de la ville et d'effectuer les calculs associés (critères d'évaluation, recherche de plus court chemin). Il est mis en œuvre sur plusieurs niveaux d'abstractions selon les Principes d'Abstraction et de ré-utilisation (section 8.2).

- **Sous-Système de Visualisation : son but est de dessiner l'état courant de la ville à l'aide de formes élémentaires. Ce sous-système rassemble les dépendances vis-à-vis de GTKmm vis-à-vis du **dessin**.

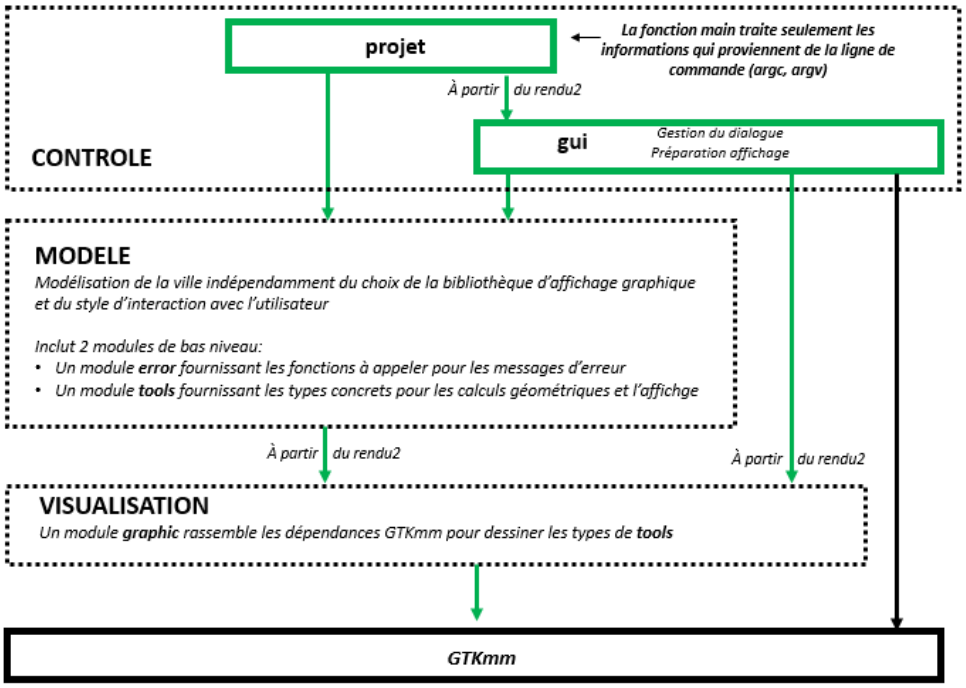

Fig 10 : Architecture logicielle minimale à respecter (option correspondant à Fig11 b1)

8.2 Décomposition du sous-système MODELE en plusieurs modules

Le Modèle doit être organisé en plusieurs modules, pour maîtriser la complexité du problème et faciliter sa mise au point. La Figure 11 présente l'organisation minimale à adopter en termes d'organisation des modules :

- Au plus haut niveau, le module **ville** gère les structures de données des ensembles de quartiers et calcule les critères d'évaluation ENJ, CI et MTA. Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi,

en vertu du principe d'abstraction le module **ville** *est le seul module* dont on peut appeler des fonctions en dehors du MODELE (Fig 11)[4].

- *Niveau intermédiaire*: **noeud.** Selon ses préférences, ce module peut gérer un type paramétré ou être conçu comme une hiérarchie de classes pour intégrer les différentes sortes de nœuds (logement, transport, producton). Il est autorisé de commencer par un type paramétré (rendu1) puis passer à une hiérarchie de classe pour les rendus suivants. Ce module doit effectuer les opérations de bases pour le rendu1 (création, lecture, écriture, accesseurs, etc…) et être capable de mettre en œuvre l'algorithme de Dijkstra à partir du rendu2.

- **Module de bas niveau tools** (Principe de ré-utilisation): il est demandé de créer un module **tools** de bas niveau mettant à disposition des **types concrets** pour réaliser des opérations dont vous avez besoin sur des éléments **2D** tels que des **points,** des **vecteurs**, des **segments** de droite ou des **cercles**. L'expression **type concret** veut dire que vous pouvez définir ces nouveaux types à l'aide de structures ***dont les modèles sont visibles dans l'interface tools.h***. Il y a certes une perte de contrôle en exposant les modèles de structure à l'extérieur du module mais il s'agit d'entités de bas-niveau dont on suppose qu'elles peuvent et doivent être validées et stabilisées rapidement.

  Contrainte sémantique importante : Ce module **tools** doit être indépendant des entités définies dans **ville** et dans **noeud** afin d'être ré-utilisable ultérieurement dans d'autres applications. Cela veut dire que le module **tool** n'a aucune idée de ce qu'est un *quartier* ou une *connexion* entre deux *quartiers ;* de même ce module n'a pas de notion de *Nœud* d'un graphe ou de *Lien* connectant deux noeuds : CES MOTS/CONCEPTS NE DOIVENT PAS APPARAITRE dans le module **tools**. C'est seulement au niveau des appels des fonctions dans les modules de plus haut niveau que les données fournies en argument seront des cercles représentant des quartiers ou des segments de droite représentant des connexions.

- **Module de bas niveau error** (spécifique à notre organisation de projet): nous mettons à disposition un ensemble de fonctions dans **error.h** qu'il faut appeler pour faire afficher les messages d'erreurs détectés à la lecture d'un fichier. Une fonction est fournie pour afficher un message de lecture avec succès. Il n'est pas autorisé de modifier le code source de ce module car il sera utilisé par notre programme de notation automatique du rendu1.

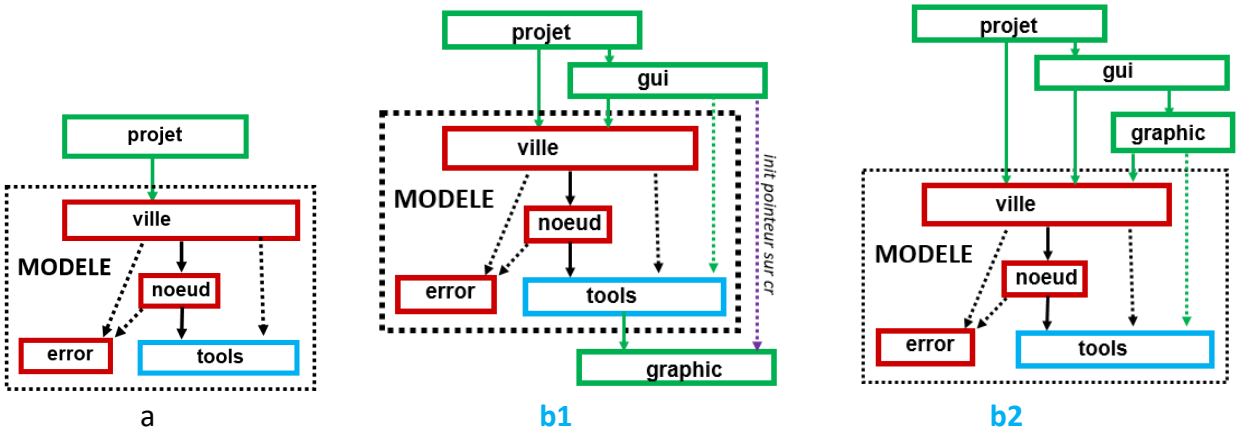

Figure11 : (a) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (rendu1); (b) modules et dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendus 2 et 3)

Deux choix possibles : (b1) **graphic** sous le Modèle (cf Topic5), (b2) **graphic** au-dessus du Modèle

8.3 Module graphique de bas-niveau graphic (à partir du rendu2)

Cette année nous adoptons l'approche suivante pour gérer le dessin de l'état de la ville. La représentation de la ville définit explicitement des entités géométriques (point, vecteur2D, cercle) fournies par le module **tools** non seulement pour lui déléguer certains calculs MAIS AUSSI pour lui déléguer la tâche du dessin.

[4] si le sous-système de Contrôle veut modifier l'état de la ville cela doit se faire par un appel d'une fonction de **ville.h**.

Cependant le module **tools** doit rester indépendant d'une libraire graphique particulière (principe de regroupement des dépendances). C'est pourquoi les dépendances vis-à-vis de la bibliothèque GTKmm doivent être rassemblées dans le module **graphic**. C'est dans ce module qu'on écrira les fonctions pour définir des couleurs de dessin, épaisseur de trait, et le tracé des formes géométriques de cercle ou de droites.

ANNEXE A : constantes globales définies dans constantes.h

Ces constantes sont appelées « globales » car elles pourraient être nécessaires dans plus d'un module. L'utilisation de **constexpr** crée automatiquement une instance *locale* dans chaque fichier où constantes.h est inclus ; il n'y a donc pas de problème de définition multiple de ces entités.

Si vous désirez mettre en œuvre vos propres constantes, les bonnes pratiques sont les suivantes :
- utilisez constexpr pour les définir
- définissez-les *le plus localement possible* ; inutile de les mettre dans l'interface d'un module (.h) si elles ne sont utilisées que dans son implémentation (.cc)

Selon nos conventions de programmation, un nom de constante définie avec constexpr suit la même règle qu'un nom de variable (E12). Le texte de la donnée les fait apparaître en **gras** dans le texte.

constexpr double **dim_max**(1000.) ;	m
constexpr double **dist_min**(10.) ;	m
constexpr double **default_speed**(5.) ;	m/s
constexpr double **fast_speed**(20.) ;	m/s
constexpr unsigned **min_capacity**(1e3);	nombre de personnes
constexpr unsigned **max_capacity**(1e6);	nombre de personnes
constexpr unsigned **max_line**(80);	nombre de caractères par ligne de fichier
constexpr unsigned **max_link**(3);	nb de connexions max d'un Logement
constexpr unsigned **no_link**(static_cast<unsigned> -1);	exprime qu'un lien n'existe pas
constexpr double **infinite_time**(1e100);	approx d'un temps infini en secondes
constexpr unsigned **default_drawing_size**(800);	en pixels
constexpr double **delta_zoom**(0.2);	variation du facteur de zoom
constexpr double **max_zoom**(3.);	facteur de zoom max
constexpr double **min_zoom**(0.2);	facteur de zoom min