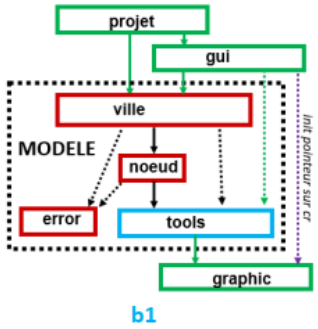
Rapport Final

Architecture



b1

1. Répartition des tâches des modules du modèle

Au niveau le plus bas, nous avons le module de visualisation, *graphic*, qui utilise la bibliothèque Gtkmm pour dessiner des formes simples telles qu'un segment ou un cercle.

Ensuite, nous passons au modèle. Outre les modules de bas niveau error et constantes, fournis par la donnée, nous avons au niveau le plus bas du modèle le module *tools*. Celui-ci prend en charge le calcul de collision entre les formes misent en jeu (segment, cercle), possède quelques fonctions utilitaires pour le reste du modèle, comme un calcul de norme ou la vérification si un point est dans un cercle par exemple, et relaie la tâche du dessin des formes à *graphic*, en prenant tout de même en charge le changement de couleur.

Au-dessus du module tools nous avons le module *noeud*. Ce module est au cœur de notre projet, et a la charge de la plupart des aspects calculatoires. Il gère notamment l'ensemble des noeuds et des liens, structurés avec des vectors.

Ce module est celui qui crée ou détruit des noeuds, utilisant la batterie de tests dont il dispose également, soit répondant à la lecture soit à une interaction avec l'utilisateur. Étant donné qu'il a la charge de l'ensemble des noeuds et des liens, il s'occupe de la sauvegarde de l'état actuel de la ville, et donne les instructions nécessaires à tools pour réaliser le dessin des noeuds et des liens.

Noeud a aussi la charge de la partie calculatoire des critères de la ville, ENJ, CI, et MTA. Il peut répondre aux appels des niveaux supérieur pour les interactions avec l'utilisateur, car c'est lui qui s'occupe des tâches d'éditions, comme bouger/agrandir un noeud, créer/détruire des liens, ou encore déterminer les appels à faire à tools pour l'affichage de *shortest path*.

Enfin, au niveau le plus élevé du modèle, nous avons le module *ville*. Ce module agit comme un relai entre l'interface utilisateur (graphique ou ligne de commande) et le module calculatoire *noeud*. Ceci veut dire qu'il a la charge de la lecture des fichiers fournis par la ligne de commande. Ce module a aussi la charge de la gestion du noeud courant, et notamment de vérifier la validité de certains appels (pour éviter des segmentation faults en essayant de faire des changements à un noeud qui n'existe pas par exemple).

Ce module d'assez haut niveau donne également une vue d'ensemble sur le modèle, avant de déléguer le gros des calculs à des modules inférieurs.

Nous passons enfin au système de contrôle. Celui-ci comporte deux modules : projet, et gui.

Le module gui prend en charge le dialogue avec l'utilisateur. C'est lui qui contient la *window* de Gtk, et qui lui fournit tous les outils nécessaire (*DrawingArea*, *Buttons*, etc.) pour que l'utilisateur puisse utiliser les fonctionnalités offertes par le modèle. Celui-ci lui donne la responsabilité des boutons, et de faire apparaître les dessins créés par *graphic* et *tools*.

Le module final, *projet*, est modeste, car il ne contient que la fonction main. Celle-ci déclare l'instance de la fenêtre, et délègue la lecture à *ville* au besoin.

2. Structuration des données pour le dessin de la ville

Nous avons réparti la gestion des données sur deux modules : *noeud*, qui gère une classe *Noeud* ainsi qu'un struct *Lien*, et *ville* qui gère une classe *Ville*.

Comme mentionné plus haut nous utilisons l'outil vector pour stocker l'ensemble des noeuds, et l'ensemble des liens dans *noeud*. Un Noeud possède plusieurs attributs, dont son type, car nous avons jugé la solution avec héritage peu logique, la plupart des attributs étant les mêmes quel que soit son type (nbp, uid, le vector de liens de chaque noeud (uid de chaque voisin), etc.). Chaque lien possède le uid et la position de ses deux extrémités, afin d'alléger la recherche de ces éléments souvent utilisés lors de calculs. Ces vectors sont placés dans un namespace, où ils peuvent être accédés à tout moment dans le module. Nous avons également fait le choix d'avoir des tableaux séparés de logement, production et transports, pour le calcul des critères ainsi que quelques tâches d'éditions.

Le module *ville* a moins de responsabilités avec les données. Il garde néanmoins une trace des critères ENJ, CI, et MTA de la Ville, ainsi que son nom (le nom du fichier).

3. Pseudocode de l'algorithme de Dijkstra

Input : tableau **nœuds** de Nœuds, tableaux **production**, **logement**, et **transport** (depuis namespace),
indice **d** de départ, **type** du nœud d'arrivée
Output : **indice** du nœud d'arrivée

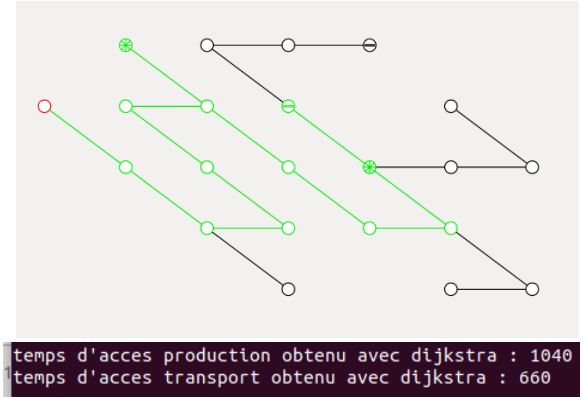
Pour tous les éléments de **nœuds**
Initialisation : **in** ← true, access ← infinite_time, parent ← no_link, remplissage de TA avec les indices
Nœuds[d].access ← 0
TA trié dans l'ordre croissant des **access**

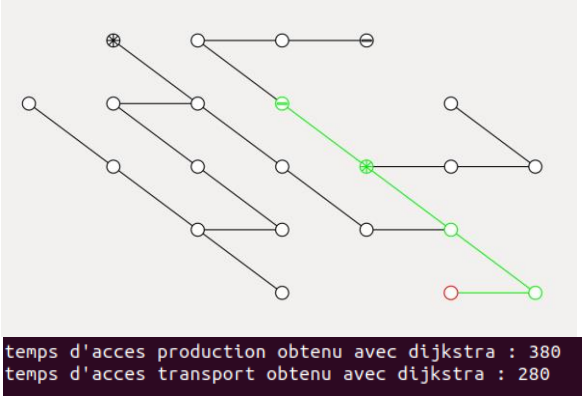
Si type = production **et** production est vide
Sortir no_link
Si type = transport **et** transport est vide
Sortir no_link

Tant que TA non-vide
n ← find_min_access (TA)
Si n = no_link
Sortir no_link
Si n = type
Sortir n
Nœuds[n].**in** ← false
Pour tous les liens de nœuds[n]
Pour tous les nœuds
Si le $i^{ème}$ uid = $k^{ème}$ lien de nœuds[n]
Rank ← i
Si rank n'est pas no_link **et** nœuds[rank].in = true
new_access_time = nœuds[n].access + access de **n** à **rank**
Si nœuds[rank].access plus grand que new_access_time
Nœuds[rank].**access ← new_access_time**
Nœuds[rank].**parent ← n**
Trier TA par rapport à **access**

Vider TA
Sortir no_link

4. Plus court chemin





5. Méthodologie

Notre groupe était en contact pendant la plupart du projet, ce qui nous a permis de diviser le travail assez localement, plutôt par rapport à des tâches que par module. Nous avons utilisé un google drive pour mettre en commun les fichiers, indépendamment de la plateforme, et discord pour communiquer avec la période de confinement.

Cependant, la plupart des tâches qui touchent au dessin ont été faites par Gabriel (module graphic, gestion de DrawingArea, etc.), alors que la plupart des tâches calculatoires ou de dialogue ont été faites par Joshua (calcul de critère, boutons, lecture/sauvegarde, etc.)

Pour le rendu 1, la priorité était de poser les bases avec *tools* avant d'attaquer *noeud* ou *ville*. Une fois les bases posées, nous avons procédé dans des directions opposées. Gabriel a pris en charge les modules de bas niveau comme *graphic* et *tools*, et Joshua les modules de haut niveau *gui* et *ville*. *Noeud* a été réparti par tâche, comme mentionné auparavant. Nous avons trouvé cette organisation a été assez efficace et nous nous sommes bien organisés pour pouvoir rester productif séparément tout en avançant.

Nos tests ont souvent utilisé les fichiers fournis sur Moodle, que nous modifions dépendant de notre besoin et pour voir de différentes situations et tester les fonctionnalités (nous nous sommes aussi aidés de la démo pour la création de fichiers tests).

Le problème qui nous a pris le plus de temps à résoudre était d'obtenir la bonne valeur pour MTA, et de mettre en œuvre une version opérationnelle de Dijkstra. Avoir une valeur en un temps raisonnable pour des gros fichiers était également un défi. Nous avons résolu ce problème en gérant le tableau de données TA plus localement.

Le bug le plus fréquent (autre que des erreurs de syntaxe/d'utilisation de widgets/fonctions Gtk) est sûrement le segmentation fault, souvent dû à un test manquant à un niveau plus haut (par exemple en essayant d'accéder à des valeurs du tableau de nœuds qui n'existent pas).

6. Conclusion

Les points forts de notre travail ont sûrement été notre organisation qui nous ont permis d'être efficace sans être face-à-face, ainsi qu'une direction assez claire qui nous a permis de ne pas perdre trop de temps sur la conception (architecture, encapsulation, etc.). L'environnement mis à notre disposition a été très utile, le forum répondant souvent à nos questions et les données étant claires.

Cependant, des points d'améliorations possible sont une meilleure abstraction de certaines fonctions (par exemple une fonction qui renverrait l'indice d'un nœud donné son uid, ou certaines fonctions de tests que nous avons dû modifier pour le rendu 3) ainsi qu'une meilleure organisation du code, nos modules devenant très longs et difficile à naviguer. C'est un point à retenir qu'une bonne division de chaque module est à faire (par exemple *nœud* prend en charge les actions d'édition, calculs de critères, tests pour la création, etc.) pour améliorer la lisibilité et l'organisation du code lui-même.