Projet de recherche opérationnelle *[IS202]*

Rapport Final 11/05/2007

Équipe de développement :

segment impair { Maxime Bochon, Yannick Martin }

segment pair { Thibaut Carlier, Mohamed El Afrit, Matthieu Lefebvre }

Table des matières

Chapitre 1

Introduction

Dans un carrefour, les voitures se croisent. Nous devons donc réglementer le passage de celles-ci. Dans le cas de gros carrefours, on utilise des feux tricolores. L'objectif de ce projet est d'optimiser la circulation automobile. Nous devons en jouant sur les temps d'attentes et les temps de vert faire passer le plus de voitures à travers le carrefour.

Pour résoudre ce problème, nous disposons d'un certain nombre de paramètres (explicités plus bas). Nous utilisons lp_solve afin de résoudre le problème de recherche opérationnelle.

Pour cela, nous devons générer automatiquement les équations d'entrées correspondant aux contraintes et à la fonction économique. Ces équations sont stockés dans un fichier qui sera passé en argument à lp_solve pour produire les résultats. Nous devons ensuite interpréter les résultats pour produire une analyse cohérente.

De manière logique, nous avons décidé de séparer ces deux étapes. Nous avons un programme qui gère la génération des équations et un programme qui interprète les résultats.

Nous allons d'abord aborder le problème de recherche opérationnelle avant d'expliquer l'implémentation en C des programmes cités précédemment, avec notamment une justification du type abstrait de donnée ayant été retenu. Ensuite nous expliciterons les choix de programmation liées à la définition de la première partie.

Enfin, après avoir exposé la manière dont le post-traitement a été réalisé, nous parlerons en annexe de la procédure de tests et du déroulement global du programme.

Première partie

Recherche Opérationnelle

Chapitre 2

Point de vue théorique

2.1 Problématique

Dans un premier temps on s'interessera à chercher les équations théoriques dont la résolution nous permettra d'aboutir à un résultat optimal. On se basera par la suite sur ces équations pour construire notre programme.

2.2 Équations

Dans ce chapitre, nous allons énoncer les équations de contraintes qui régissent les variables du problème.

2.2.1 Débits et temps de vert

Nous pouvons générer deux équations de contrainte sur le temps de vert pour un feu donné. Tout d'abord, nous avons l'équation suivante :

$$Q \times TC \times \mu = Q_{sat} \times vert$$

Cette équation met en relation pour un feu i le temps de vert, le débit de saturation noté Q_{sat}, le débit réel Q, le temps de cycle TC ainsi qu'un coefficient noté μ.

Ce coefficient peut être interprété comme une coefficient sans dimension mesurant la réserve du feu étudié. L'objectif est d'avoir un coefficient μ le plus grand possible.

La deuxième équation permet de fixer un temps minimal fixé par le responsable de l'étude du carrefour :

$$Vmin \leq vert$$

$Vmin$ représente un temps de vert minimal pour un feu donné.

2.2.2 Dates d'allumage et temps de cycle

Il existe une contrainte qui lie pour chaque feu le temps de début avec le temps de cycle. Cette équation de contrainte est la suivante :

$$deb(i) \leq TC$$

où $deb(i)$ représente la date d'allumage du feu(i).

Comme l'étude est mené sur un cycle et que les résultats trouvés doivent être valable pour tous les cycles alors on interdit qu'un feu s'allume pour la première fois après la fin d'un cycle.

2.2.3 Fonction économique

Maintenant que nous avons introduit les contraintes sur les débits ainsi que le coefficient μ, nous pouvons parler de la fonction économique.

Nous avions dit dans la partie précédente que nous devions maximiser les coefficients μ.
Une première idée serait de dire que la fonction économique serait :

$$max(min(\mu_i))$$

en posant μ_i le coefficient μ du feu i.

Cependant la fonction ainsi posée n'est pas linéaire. On est donc obligé de générer des contraintes du type :

$$\mu \leq \mu_i$$

ainsi notre μ sera le plus petit des coefficients.

La fonction économique devient :

$$max(\mu)$$

2.2.4 Décalages

Il y a décalage entre le feu i et le feu j lorsque l'un doit s'allumer avant l'autre.
Nous avons considéré deux types de décalages.

Pour le décalage a l'ouverture entre le feu i et le feu j, nous avons l'équation suivante :

$$deb(i) + duree \leq deb(j)$$

C'est à dire que le feu j doit s'allumer après le feu i de $duree$ secondes.
Il faut prendre en compte le décalage a la fermeture. Cela nous donne une équation :

$$deb(i) + vert(i) + duree \leq deb(j) + vert(j)$$

C'est à dire que le feu j doit s'éteindre avant le feu i de $duree$ secondes.

Remarque : la contrainte de type inclusion, c'est à dire que le feu j est inclu dans le feu i peut être traduite par une contrainte de décalage à l'ouverture et une contrainte de décalage à la fermeture de 0 seconde.

2.2.5 Contraintes entre les feux

Dans cette partie, nous allons énoncer les équations qui traduisent le fait que deux feux qui sont incompatibles ne peuvent s'allumer en même temps.

Par exemple : soient un feu i et un feu j incompatibles, il faut que le feu j s'allume au moins après que le feu i soit passé au rouge plus un temps de écurité.

Ce qui donne :

$$deb(i) + vert(i) + sec(i,j) \leq deb(j)$$

en notant $sec(i,j)$ le temps de sécurité entre le feu i et le feu j.

Comme le phénomène est cyclique, il faut aussi veiller, si le temps de vert du feu j déborde sur le cycle suivant, à ce qu'au cycle suivant le feu i s'allume après le feu j, d'où :

$$deb(j) + vert(j) + sec(j,i) \leq deb(i) + TC$$

Dans le cas précédent, on a supposé que le feu i s'allume en premier cependant, il est possible qu'il s'allume apres le feu j.

On va donc utiliser les règles vues dans le cours de recherche opérationelle pour écrire des règles de disjonctions.

Ce qui nous donne, finalement :

$$deb(i) + vert(i) + sec(i,j) - deb(j) \leq M \times delt(i,j)$$
$$deb(j) + vert(j) + sec(j,i) - deb(i) - TC \leq M \times delt(i,j)$$

$$deb(j) + vert(j) + sec(j,i) - deb(i) \leq M \times (1 - delt(i,j))$$
$$deb(i) + vert(i) + sec(i,j) - deb(j) - TC \leq M \times (1 - delt(i,j))$$

Chapitre 3

Point de vue pratique

3.1 Résolution par lp_solve

3.1.1 Structure du format LP

Le format LP qui est un des formats gérés par lp_solve a des contraintes structurelles.

Toute ligne (déclaration ou équation) doit se terminer par un ";".
La fonction économique se trouve en tête et est précédée d'un mot clé du type "max" ou "min" pour expliciter à lp_solve comment optimiser le problème au regard de la fonction économique.

Dans un second temps, on peut mettre les équations de contrainte.

À la fin du fichier, et uniquement à cet endroit, on peut écrire toutes les déclarations des variables.

Nous allons voir ensuite les règles syntaxiques de lp_solve.

3.1.2 Aberrations syntaxiques à considérer

Voici une liste non exhaustive des écueils à éviter lors de l'écriture d'un fichier au format LP, au risque de voir les résultats varier du tout au tout.

- il ne faut pas oublier le ";" en fin de ligne.
- il faut respecter l'ordre : définition, déclaration pour les équations et les variables.
- il n'existe pas de signe de multiplication, l'espace entre un scalaire et une variable matérialise le produit.
- le produit dans lp_solve n'est pas commutatif, il faut veiller à respecter l'ordre scalaire-variable.
- il est interdit de faire : *scalaire scalaire variable* en effet, *LP_Solve* considère que 2 produits ne peuvent se suivre. L'exemple précédent est interprété comme : *scalaire scalaire* + *variable*
- lp_solve ne fait pas la différence entre les inégalités larges et les inégalités strictes.

Il est fortement conseillé de vérifier le fichier que l'on veut passer à lp_solve pour détecter les erreurs syntaxiques énumérées précédemment car il arrive que lp_solve

ne les signale pas ce qui conduit à des résultats parfois absurdes.

3.1.3 Résultats fournis par lp_solve

Le solveur lp_solve retourne les résultats sur la sortie standard. Il faudra dans notre cas faire une redirection vers un fichier.

Il y a d'abord une phrase qui nous donne la valeur maximale de la fonction économique. Ensuite, les retours sont de la forme : *variable espaces valeur*.

Chapitre 4

Encodage des données d'un carrefour

4.1 Considération du format en entrée

Nous considèrerons que dans notre programme toutes les variables exceptées les μ_i sont de type entier.

Voici le formatage des données en entrée :

- Temps de vert minimaux :
  `feu durée`

- Conflits :
  `feu_père feu_fils temps1 temps2 temps3 temps4`
  feu père / feu fils : temps1 + temps2
  feu fils / feu père : temps3 + temps4

- Decalages a l'ouverture :
  `feu_père feu_fils durée`

- Decalages a la fermeture :
  `feu_père feu_fils durée`

- Temps de cycle :
  `durée`

- Debits de saturation :
  `feu débit`

- Debits reels :
  `feu débit`

Nous disposons de paramètres pour pouvoir modéliser le carrefour.

4.2 Paramètres en entrée

Nous avons tout d'abord une matrice de temps. Nous la nommerons par la suite matrice de sécurité.

Elle nous informe des feux qui peuvent être allumés en même temps : lorsque deux feux sont en conflits, c'est à dire qu'ils ne peuvent pas être verts en même temps.

Nous avons au moyens de la matrice la donnée d'une valeur (entière) correspondant au temps qu'il faut attendre après le passage au rouge du feux i pour faire passer le feux j au vert.

Ce temps d'attente correspond à un temps (règlementé) de dégagement du carrefour.

Nous avons aussi la donnée des débits de saturation. Ces valeurs correspondent au nombre de véhicules maximal par heure que le carrefour peut écouler par le feu i.

Remarquons que nous incluons dans le carrefour les feux piétons mais que nous n'allons pas optimiser leur temps d'allumage. Il est donc inutile de fournir une valeur représentant le débit de piéton.

Nous avons aussi en paramètres le temps de vert minimum pour chaque feu ainsi que des temps correspondant à des décalages au niveau du passage au vert ou du passage au rouge.

Nous pouvons à tout moment décider qu'un feu i puisse s'allumer en retard par rapport au feu j et passer au rouge avant ce dernier.

La dernière donnée est celle du temps de cycle. Le passage au vert des feux se fait de manière périodique. Ce temps de cycle représente la période du phénomène.

Deuxième partie

Programmation

Chapitre 5

Analyse de haut niveau

5.1 Objectifs de programmation

L'objectif principal de ce projet consiste à réaliser une aide à la resolution et à l'interprétation du problème de recherche opérationnelle présentée précédemment, étant entendu que la résolution calculatoire du simplex est assurée par le programme lp_solve.

Cette aide se décompose en deux problèmes distincts modélisés par les couples entrée/sortie suivants :

Pré-traitement
Entrée : données sur le carrefour
Sortie : équations associées, exploitable par lp_solve

Post-traitement
Entrée : résultat brut de la résolution des équations par lp_solve
Sortie : mise en forme optimale de ce résultat

5.2 Pré-traitement

Arguments attendus :
- nom du fichier d'entrée
- nom du fichier de sortie
- taille du carrefour (nombre de feux)

Reprenons le couple entrée/sortie du problème, en le détaillant :

Entrée : données sur le carrefour

Ces données sont contenues dans un fichier selon un format à définir. Chaque ligne est soit :
- un en-tête de début de section, selon la syntaxe : `Nom de la section :` (intitulé fixe, mais insensibilité à la casse)
- une ligne blanche marquant la fin d'une section
- une suite de nombres séparés par des espaces simples, dont la syntaxe dépend de la section :

L'ordre des sections n'a pas d'importance. Ce format doit absolument être respecté car la robustesse n'est pas un objectif prioritaire, en tout cas dans un premier temps.

Ce fichier ne comprend pas de section `Taille du carrefour :` qui indiquerait le nombre de feux. Cela provient du fait que l'allocation de mémoire initiale, proportionnelle à la taille du feu, doit se faire avant la phase de lecture.

Sortie : équations associées, exploitables par lp_solve
Là encore des contraintes de format existent. Se référer à la partie Recherche opérationnelle pour en savoir plus.

La partie de pré-traitement comprend deux phases : la lecture des données et l'écrire des équations associées. La phase de lecture consiste à extraire l'ensemble des données du problème à partir d'un fichier d'entrée codé en ASCII. Il s'agit de stocker ces valeurs de manière ordonnée en mémoire, avant de passer la main à la seconde phase de pré-traitement, l'écriture.

Lecture
Entrée : suite de caractères ASCII encodant les données sur le carrefour (dans un fichier)
Sortie : représentation numérique structurée des données sur le carrefour (en mémoire)

Écriture
Entrée : représentation numérique structurée des données sur le carrefour (en mémoire)
Sortie : mise en forme des données dans le format reconnu par lp_solve

5.3 Post-traitement

5.3.1 Affichage des résultats

Affichage en caractères ASCII

Arguments attendus :
- nom du fichier contenant les résultats générées par lp_solve
- nom du fichier d'entrée contenant les données initiales

Affichage avec GNUPLOT

Arguments attendus :
- nom du fichier contenant les résultats générées par lp_solve
- nom du fichier d'entrée contenant les données initiales
- le fichier `temps.dat` pour stocker les données à tracer avec GNUPLOT

Le script `script.plt` reste inchangé à condition de nommer le fichier (troixième argument) `temps.dat`. En effet ce script que GNUPLOT utilise pour afficher les résultats fait appel à `temps.dat`.

En se basant sur le tutorial de GNUPLOT, les données dans le fichiers temps.dat doivent être stockées sous forme de matrice à au moins[1] *Nombre_feux* lignes et

[1] Lorsque la durée du vert dépasse le temps de cycle il faut créer une nouvelle ligne contenant les informations nécessaires pour représenter le fragment de segment qui est en dehors du cycle ; ce fragment de segment doit alors être placé au début du cycle

cinq colones : pour chaque feux la première colone représente le numéro du feux, la deuxième colone est le temps de cycle qui est constant pour tout les feux, la troixième colonne doit être à 0, la quatrième colonne représente le début du vert et la cinquième colonne représente la fin du vert.

5.4 Choix d'un TAD

Nous avions pour traiter le sujet deux possibilités quant à l'organisation du TAD.

Nous pouvions exploiter la matrice telle qu'elle (chargée en mémoire) et charger en mémoire si besoin est les différents paramètres tels que les décalages ou les débits.

Nous avons pour de raison d'ordre décidé de structurer ces données. Nous disposons de trois classes :

- carrefour : on stocke les données liées au carrefour, c'est à dire l'ensemble des feux, le nombre de feux ainsi que le temps d'un cycle.

- feu : on y stocke le temps de vert minimal, le débit de saturation. Nous avons un autre champ correspondant au débit réel, nombre de voitures par heure qui passent devant le feu. Nous avons ensuite la liste des conflits.

- conflit : nous avons implémenté le type élément de la liste des conflits comme une structure donnant l'indice du feu en conflit ainsi que le temps de sécurité ou de décalage. Nous avons aussi un pointeur sur l'élément suivant.

L'idée quand nous avions implémenté ce type abstrait de donnée c'était d'avoir un TAD évolutif, l'ajout d'une nouvelle contraintes est possible. On pourrait ensuite rajouter les fonctions pour exploiter les contraintes.

Lorsque dans un second temps, nous allons penser à non plus étudier les feux individuellement mais étudier les feux par classes d'équivalences, nous pourrions ajouter à la structure feu un champs qui indiquera à quelle classe appartient un feu donné.

De plus, nous souhaitions essayer d'optimiser la complexité en temps et en espace. Tout d'abord, au lieu de déclarer une matrice de taille $N * N$, N représentant le nombre de feux, nous utilisons une liste de conflit par feu.

Quelque soit le carrefour, l'utilisation de la matrice entraine que la complexité en espace est de n^2 alors que dans le cas des listes, le n^2 n'arrivera pas.

Comme nous devons analyser tous les conflits, nous diminuons par la même occasion la complexité temporelle du programme générant les é quations.

5.5 Manipulation du TAD

5.5.1 Carrefour

Pour modifier le type Carrefour, nous avons implémenté deux fonctions permettant d'initialiser et de détruire un carrefour.
Elles se nomment :

- Carrefour_initialiser
- Carrefour_detruire

Nous expliciterons dans le détails leur implémentation dans la suite de ce rapport. Cependant si le lecteur s'intéresse déjà à ce fait, l'implémentantion de ces fonctions se trouve dans le fichier `types.c`.

Carrefour_initialiser

Cette fonction prend en argument un pointeur sur un carrefour et un entier correspondant au nombre de feux. Elle renvoie un entier permettant de savoir si le carrefour a été créé avec succès.
Une allocation dynamique de mémoire est d'abord faite, puis chaque champ est rempli.

Carrefour_detruire

Carrefour_detruire prend en argument un pointeur sur un carrefour et ne renvoie rien.
Si le carrefour existe, c'est à dire que le pointeur d'entrée ne pointe pas sur NULL, la fonction free est appelée sur le tableau de structures Feux, composant le carrefour.

5.5.2 Feu

Pour modifier le type Feu, nous avons implémenté trois fonctions qui permettent de lire, d'ajouter et de supprimer des éléments.
Elles se nomment :

- Feu_lireConflit
- Feu_ajouterConflit
- Feu_supprimerConflit

Nous expliciterons dans le détails leur implémentation dans la suite de ce rapport. Cependant si le lecteur s'interesse déjà à ce fait, l'implémentantion de ces fonctions se trouve dans le fichier `types.c`.

Lecture d'un conflit

La fonction Feu_lireConflit prend en argument un feu (pointeur sur la structure du feu) et le type de conflit. Dans la structure du feu passé en argument elle retourne l'adresse du conflit.

Avec cette adresse (adresse sur une structure), on peut y lire le feu en conflit ainsi que le temps de sécurité.

Remarque : cette fonction ne modifie pas le TAD.

Suppression d'un conflit

La fonction Feu_supprimerConflit prend en argument un feu (pointeur sur la structure du feu) et le type du conflit à supprimer.

Dans la liste des conflits, elle supprime le premier élément en libérant l'espace alloué pour stocker le conflit et reconstruisant le chaînage.
Remarque : cette fonction ne retourne rien, beaucoup de tests ont été implémentés pour supprimer un conflit qui existe.

Ajout d'un conflit

Cette fonction doit initialiser l'espace mémoire pour stocker le nouveau conflit et initialiser les champs de la structure Conflit.

L'ordre de la procédure est la suivante :

- initialisation avec un malloc
- initialisation du conflit avec les valeurs passées en paramètres
- construction du chaînage pour insérer le conflit dans la liste des conflits

Chapitre 6

Analyse de moyen et bas niveaux

6.1 Opérations sur les fichiers

6.1.1 Choix des fonctions de haut niveau

État des lieux

En langage C, on dispose de deux niveaux d'accès aux fichiers, dont les principales caractéristiques sont réunies dans ce tableau comparatif :

niveau d'accès	bas niveau	haut niveau
tampon	absent	présent par défaut
nature du manipulateur	descripteur de fichier	pointeur de fichier
type du manipulateur	int	FILE *
entrée standard	0	stdin
sortie standard	1	stdout
sortie d'erreur	2	stderr
connexion	open	fopen
déconnexion	close	fclose
lecture	read	fread, fscanf, fgets
écriture	write	fwrite, fprintf

Dans les faits, les fonctions de haut niveau constituent une surcouche logicielle standard qui s'appuie sur les fonctions de bas niveau. Les possibilités étant très grandes, le tableau ne présente pas une liste exhaustive de ces dernières.

La principale différence entre haut et bas niveaux réside dans l'utilisation d'un tampon. Cela a pour avantage de limiter le nombre d'appels système, ce qui est bénéfique du point de vue des performances. En contre partie, toute écriture peut subir un retard par rapport au code source appelant, qui peut conduire à une perte ou à un entrelacement de tout ou partie de la sortie.

On peut cependant empêcher ces désagréments en forçant la vidange du tampon (fonction fflush) ou, dans le cas d'un affichage, en choisissant la sortie d'erreur (non *bufferisée*) plutôt que la sortie standard.

Justification de nos choix

Certaines opérations du programme font appel à des fonctions avancées n'existant que parmi les fonctions de haut niveau, dont fgets et fprintf. Dans ce cas, la

connexion à un fichier donne lieu à la délivrance d'un pointeur de fichier, incompatible avec les fonctions de bas niveau.

Bénéficiant par ailleurs du gain de performance, nous avons choisi de n'utiliser que des fonctions de haut niveau pour ne pas mélanger les types de manipulateurs. Nous prendrons cependant en considération la possibilité d'utiliser la sortie d'erreur lors de certains affichages.

6.1.2 Accès

L'accès à un fichier commence par une étape de connexion offrant un moyen de manipuler ce fichier pendant tout la durée de la connexion. Nous parlerons plus simplement de l'*ouverture* d'un fichier, réalisée selon un mode donné, et ce jusqu'à sa *fermeture*.

Dans notre cas, le manipulateur est un pointeur de fichier, qu'il convient de déclarer comme suit :

```
FILE * pointeurFichier;
```

N'étant jamais à l'abris d'une erreur d'accès au fichier, il convient de prendre la précaution suivante :

```
if (NULL == (pointeurFichier = fopen(chemin, mode))) {
  fprintf(stderr, "L'ouverture du fichier d'entree a echoue.\n");
  return (1);
}
```

On veillera également à fermer un fichier dès que possible :

```
fclose(pointeurFichier);
```

6.1.3 Lecture

La fonction fgets permet d'extraire la ligne courante d'un fichier et de la stocker dans une chaîne de caractères. Son utilisation est très pratique mais présente l'inconvénient de conserver la butée `\n`, ce qui n'est pas forcément gênant.

Il suffit ensuite d'extraire des valeurs selon un *format* grâce à la fonction sscanf, qui est le pendant de fscanf pour les chaînes de caractères.

Le fichier d'entrée est lu ligne après ligne grâce à fgets et s'arrête lorsque la fin du fichier est atteinte et que fgets renvoie NULL. A chaque fois la ligne est décortiquée avec sscanf, et s'il y a lieu les informations utiles sont stockées à l'endroit adéquat.

6.1.4 Écriture

Pour écrire dans un fichier, nous avons le choix entre 3 fonctions.

- write
- fwrite
- fprintf

La fonction choisie est fprintf, car en cas de tronquage de la chaîne de caractère inscrite la valeur renvoyée est de -1.

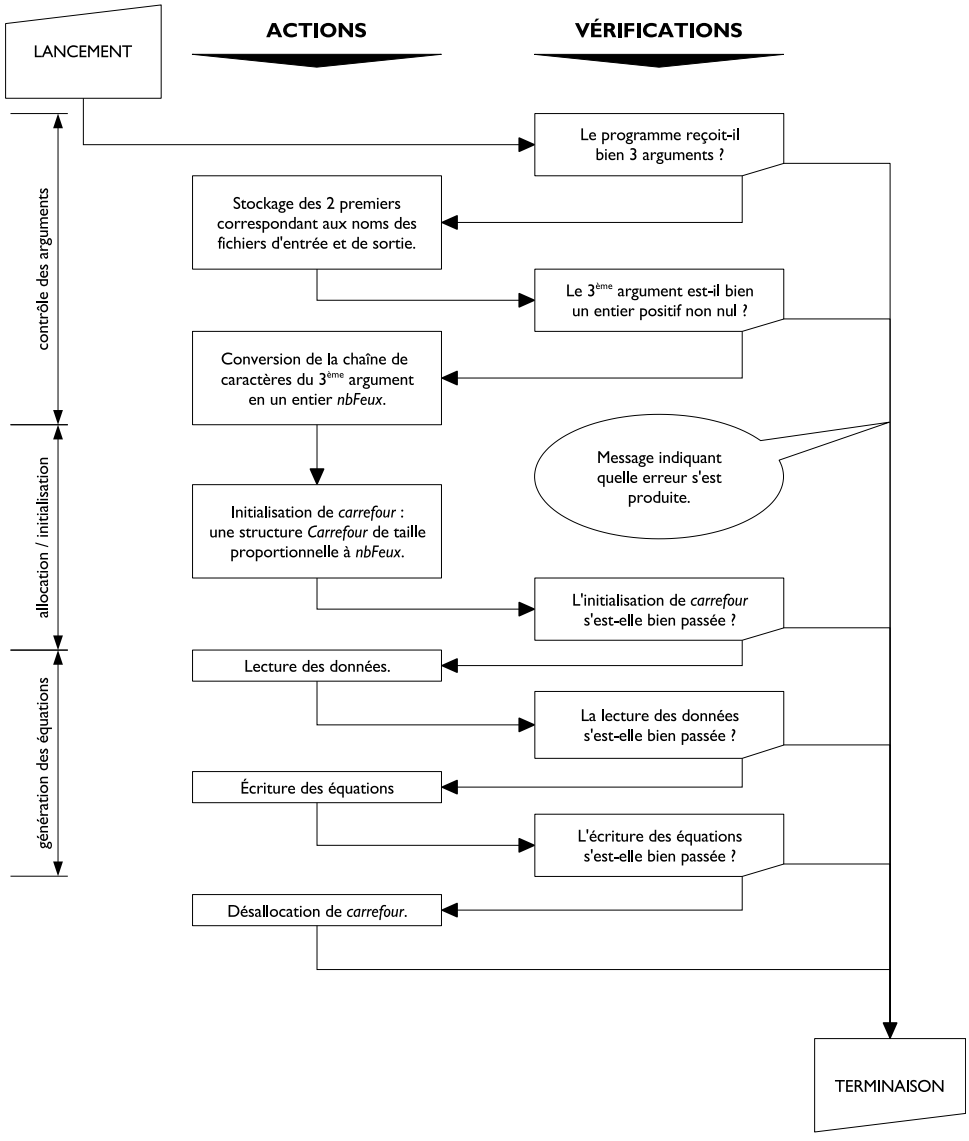
6.2 Pré-traitement

6.2.1 Fonction mère

La fonction mère est celle qui recueille les arguments, appelle les sous-programmes, en l'occurence lecture et ecriture, et informe l'utilisateur du bon déroulement du pré-traitement. Elle correspond bien évidement au main du programme C.

Son déroulement linéaire est une suite d'*actions* entrecoupée de *vérifications*. Chaque étape donne lieu à un avertissement de réussite sur la sortie standard, et à l'arrêt du programme en cas d'échec.

Le diagramme suivant est une synthèse du déroulement du main :



Comme le montre ce diagramme, le main peut se decomposer en trois parties.

Faisons quelques remarques techniques sur les deux premières.

Contrôle des arguments

Afin de récuprer les arguments fournis au main, son prototype doit être le suivant :

```
int main(int argc, char * argv[])
```

De cette manière, argc fournit le nombre d'arguments et argv les arguments en eux même. Par la suite, l'unique subtilité réside dans le fait que l'argument indicé 0 existe toujours et qu'il s'agit du nom de l'exécutable.

Une pré-condition importante pour la partie suivante est d'avoir une variable nbFeux valide, c'est à dire un entier positif non nul. Le troisième argument utile, de type *chaîne de caractères*, est donc testé par la fonction estEntierPositifNonNul avant d'être converti en entier par atoi. Cette dernière fonction requière l'inclusion de <stdlib.h>.

Enfin, toute erreur sur les arguments entraine un appel de la fonction usage qui informe l'utilisateur de la syntaxe à adopter.

Allocation et initialisation

L'espace mémoire destiné aux données sur le carrefour étant utilisé par lecture et par ecriture, la déclaration, l'allocation et l'initialisation d'une structure Carrefour a lieu avant ces dernières. Faisant appel à Carrefour_initialiser, puis plus tard à Carrefour_detruire, le fichier main.c a besoin du fichier d'en-tête types.h.

6.2.2 Lecture

La ligne constitue l'unité de traitement du fichier d'entrée. Une boucle de premier niveau parcourt chaque section et s'arrête a la fin du fichier. Elle s'appuie sur `fgets` car cette fonction permet directement d'extraire une ligne. On notera que le retour à la ligne (`\n`) fait partie des caractères transmis, mais cela n'est pas gênant dans notre cas. On met en place les éléments suivants nécessaires à l'extraction des données :

`Section` : énumération
- `CONFLIT_ENTRE_FEUX` → 0
- `CONFLIT_DECAL_OUV` → 1
- `CONFLIT_DECAL_FERM` → 2
- `DEBIT_SATUR` → 3
- `DEBIT_REEL` → 4
- `VERT_MIN` → 5
- `TEMPS_CYCLE` → 6
- `NB_DE_SECTIONS` → 7

`labels` : tableau de `NB_DE_SECTIONS` chaînes
- `CONFLIT_ENTRE_FEUX` → `"Conflits :\n"`
- `CONFLIT_DECAL_OUV` → `"Decalages a l'ouverture :\n"`
- `CONFLIT_DECAL_FERM` → `"Decalages a la fermeture :\n"`
- `DEBIT_SATUR` → `"Debits de saturation :\n"`
- `DEBIT_REEL` → `"Debits reels :\n"`
- `VERT_MIN` → `"Temps de vert minimaux :\n"`
- `TEMPS_CYCLE` → `"Temps de cycle :\n"`

Voici l'algorithme de lecture du fichier de données :

```
cas : Section
i : entier
tant que ⌐ finDeFichier (ligne ← extraireLigne (entree))
    cas ← indéfini ()
    pour i de 0 à (NB_DE_SECTIONS − 1) faire
        si égalitéChaînes (ligne, labels [i]) alors
            cas ← i
            terminer pour
    si estIndéfini (cas) alors
        court-circuiter tant que
    tant que ⌐ finDeFichier (ligne ← extraireLigne (entree))
        si égalitéChaînes (ligne, "\n") alors
            terminer tant que
        traiter (ligne, cas)
```

Ce qui est appelé ici traiter correspond à la fonction qui, pour une section donnée (par la variable cas), extrait les valeurs et les ajoute à la structure en mémoire.

En C, "**court-circuiter**" correspond à l'instruction continue et "**terminer**" à break. La routine qui teste l'égalité de chaînes correspond à la fonction strncmp de la bibliothèque standard string. Celle servant à extraire une ligne correspond à fgets, qui renvoie NULL en cas de fin de fichier.

6.2.3 Écriture

Dans le cadre de l'optimisation du traffic routier au sein d'un carrefour, nous devons générer des équations :

- équation economique
- contraintes entre les feux
- contraintes sur les temps de vert

Au vue des tâches à faire, nous avons pu découper le problème de génération des équations en plusieurs sous problèmes traités par des fonctions particulières.

Fonction principale

Nous avons implémenté une fonction qui fait office d'interface entre le programme et les fonctions de générations d'équations.

La fonction se nomme ecriture et est définie dans le fichier : `ecriture/ecire.c`.

Elle gère l'ouverture et la fermeture des flots vers les fichiers qui contiendront les équations de contraintes et les déclarations des variables.
Elle gère aussi les appels vers les fonctions de génération d'équations.

Elle prend en argument la structure du carrefour ainsi que le nom du fichier cible à générer. La valeur qu'elle retourne permet de déterminer si la génération des contraintes s'est passée comme prévu.

Débit et temps de vert

Nous allons stockér les équations dans un fichier. Une fonction de niveau supérieur a ouvert un flot sur un fichier.

L'implémentation de la fonction qui génère les équations de contraintes sur les temps de vert se trouve dans : `ecriture/eqDebit-v1.c`

Cette fonction nommée genEqDebit prend en argument la structure du carrefour, le fichier où seront écrites les équations et un fichier où seront stockés les déclarations des variables. Elle ne retourne aucune valeur à la fonction appelante. Elle ne fait que modifier un fichier.

L'écriture des équations pour tous les feux se fait au moyens d'une boucle *while*. Dans cette boucle, nous écrivons de manière séquentielle les équations rapportées à un feu.

Comme on traite, dans ce carrefour, deux types de feux (feux piétions et feux pour les voitures), on fait un test pour savoir quel type de feu on va traiter.

En effet, n'ayant pas de valeur de débit pour les feux piétons, nous leur avons assigné une valeur sentinelle fixée à -1.

On peut donc seulement générer les équations utiles.

Remarque : Suite aux modifications du sujet par les encadrants de R.O., nous avons implémenté une seconde version de la fonction genEqDebit. Son implémentation se trouve dans : `ecriture/eqDebit-v2.c`.

Seule l'équation change.

Cycle et date d'allumage

La fonction qui génère les équations est implémentée dans `ecriture/eqDebut.c`.

Elle prend en argument le fichier de sortie où sont stockées les équations, un fichier où seront stockées les déclarations des variables ainsi que la structure du carrefour.

Elle ne retourne aucune valeur (ne procède à aucun test de sécurité).

La définition de la fonction est très simple. À l'aide d'une boucle *while*, on génère les équations pour tous les feux.

Equation économique

Pour cela nous avons implémenté la fonction genEqEconomique dont la définition peut se trouver dans : `ecriture/eqEconomique-v1.c`.

Cette fonction prend en argument le fichier où seront écrites toutes les équations ainsi que la structure carrefour.

Nous devons générer une somme.

Il y a une première boucle qui génere dans la fonction économique les premiers éléments de la somme. On utilise un test sur le débit réel pour ne prendre en compte que les feux piétons.

Une seconde boucle génère les contraintes pour le μ.

Aucune valeur de retour vers la fonction appelante n'a été implémentée. On pourrait considérer une valeur qui permettrait de savoir si toutes les contraintes ont

été écrites.

Remarque : Suite aux modifications du sujet par les encadrants de R.O., nous avons implémenté une seconde version de la fonction genEqEconomique. Son implémentation se trouve dans : `ecriture/eqEconomique-v2.c`.
Seul l'équation économique change.

Contraintes de décalage

L'implémentation est très simple. Cela se fait avec une boucle *while* qui à chaque tour de boucle génère une équation.

Les deux fonctions qui génèrent les contraintes sont :

- genEqDecalOuv
- genEqDecalFerm

Elles prennent toutes les deux en arguments le fichier où sont écrites les équations et le carrefour.

Contraintes entre les feux

Dans le fichier se trouvant dans : `ecriture/eqDisjonctive.c`, nous avons implémenté la fonction genEaDisjonctives. Cette fonction va pour chaque feu lire la liste des conflits entre feux.

Explication de la procédure :

- lecture d'un conflit (si existence)
- génération des équations et des déclarations des variables.
- suppression du conflit de la liste
- lecture du conflit suivant

On réitère cette boucle jusqu'à ce qu'il n'y ait plus de conflit dans la liste.

Nous avons une boucle de niveau supérieur qui réitère ce processus pour tous les feux.

Concaténation de fichier

Dans les équations disjonctives, nous avons utilisés des booléens noté *delta*. Comme lp_solve ne gère pas un type booléen, il faut donc les déclarer comme étant des variables entières inférieure ou égale à 1.

Nous avons été confrontés à un problème avec le solveur lp_solve. Il faut obligatoirement que les déclarations de variables soient en fin de fichiers.

Comme nous voulions déclarer dans le fichier des équations juste le nombre de variables, c'est à dire ne pas générer tous les booléens possibles, nous avons décidé de stocker la déclaration de ces variables dans un fichier temporaire. C'est uniquement à la fin de la génération des contraintes que nous pouvons fusionner le fichier de déclaration avec le fichier des contraintes.

Une alternative aurait été de stocker les noms de variables en mémoire plutôt que dans un fichier. Préférant rester dans l'optique de l'économie au niveau espace

et temps, nous avons opter pour la solution avec fichier temporaire.

Nous avons alors implémenté une fonction qui s'apelle fileConcat. La définition de cette fonction se trouve ici : `ecriture/fileConcat.c`.

Elle prend en argument les deux fichiers à fusionner. La méthode utilisée est de créer en mémoire une zone tampon. On copie une ligne dans le fichier source que l'on copie dans la zone tampon. Ensuite, on copie le contenu du tampon dans le fichier cible.
À la fin, on libère la zone tampon.

La valeur de retour permet de savoir si la fonction a pu allouer sa zone mémoire tampon afin de faire le travail de recopie.

6.3 Post-traitement

Le but de ce programme est, dans un premier temps, de récupérer les données fournies pas lp_solve, puis de les stocker en mémoire. Dans un second temps, il sera question de les afficher à l'écran.

6.3.1 Récupération des données

La récupération des données se fait au moyen de la fonction récupération :

Entrée : représentation numérique structurée des données fournies par lp_solve
Sortie : Les temps de vert et les debut de vert stockés dans deux tableaux d'entiers.

Dans un premier temps la fonction récupération ouvre un flux de fichier associé au fichier fourni en entrée : ce fichier contient (entre autre) les temps de vert, et les dates de début de vert de chaque feux.

Ce fichier est ensuite lu ligne par ligne selon la commande suivante :

```
sscanf(ligne, "%[a-z]%*[[]%u%*[]]%*[ ]%u",nom, &numero, &temps);
```

La fonction renvoie en sortie deux pointeurs sur les tableaux d'entiers indexés par les numéros des feux :
- debut : date de debut du temps de vert.
- duree : durée pendant laquelle les feux restent allumés.

Ces deux informations sont nécessaires afin d'établir le plan du carrefour. Dans la partie suivante, nous expliquerons comment nous avons mis en oeuvre l'exploitation de ces données afin d'afficher à l'écran un plan complet du carrefour.

6.3.2 Exploitation des données et affichage du plan de carrefour

Afin de rendre l'interface plus confortable pour l'utilisateur nous avons implémenté deux solutions quant à l'affichage du plan de carrefour.

Affichage ASCII

L'obectif étant de tracer le plan du carrefour, nous avons décidé de représenter les temps de fonctionnement de chaque feux par des lignes horizontales constituées de tirets : lorsqu'un tiret est présent le feux est vert, lorsque il est absent le feux est rouge.

Le tracé ainsi décrit est effectué par la fonction tracerFacile :
Entrée : les temps de vert et les début de vert stockés dans deux tableaux d'entiers.
Sortie : la valeur témoin 0 (la fonction ne s'occupe que de l'affichage)

Bien que le tracé soit assez élémentaire, il présente néanmoins une petite difficulté :
Les temps d'allumage des différents feux ne peuvent excéder le temps de cycle, cependant ils peuvent commencer à s'allumer à n'importe quelle date pouvu qu'elle soit comprise entre le temps 0 et le temps de cycle. De ce fait il arrive que les temps de vert des feux dépassent le temps de cycle. Il faut donc replacer les tirets au début du cycle :

Pour un feux i dont le temps de vert est de 80 et qui débute à la date 50 on ne peut afficher :

```
feux i |                ------------------|--------
       0                                  TPS_CYCLE
```

Il faut afficher :

```
feux i |--------        ------------------|
       0                                  TPS_CYCLE
```

La solution consiste à tester le débordement éventuel sur le cycle suivant puis d'ajuster les indices afin d'afficher les tirets correspondant en priorité.

Affichage GNUPLOT

Maintenant notre objectif est de tracer les resultats avec GNUPLOT. Pour cela il faudrait utiliser un script que nous avons écrit dans le fichier `script.plt` ce script reste inchangé à condition que le fichier de sortie s'appelle `temps.dat`
Le script que nous avons utilisé est le suivant :

```
set title 'GESTION DE CARREFOURS'

set autoscale
unset log
unset label

set xtic auto
set ytic auto

set border 3
set xtics nomirror
set ytics nomirror

set xlabel "numéro feu"  #place le label entre apostrophes contre l'axe des x
set ylabel "durée feu"   #place le label entre apostrophes contre l'axe des y
```

```
#pour rediriger la sortie vers un fichier eps
#set terminal postscript #portrait color
#set output 'image.eps'


#pour afficher les feux rouge mais il faut remplacer le plot de la
ligne après par replot

#plot[0:*][0:*] "temps.dat" using 1:3:3:2 with yerrorbars title 'feu rouge'

#pour afficher les feux vert
plot "temps.dat" using 1:3:4:5 with yerrorbars lt 2 title 'feu vert'


# lt pour la couleur
#0 -> noir
#1 -> rouge
#2 -> vert
#3 -> bleu
#4 -> magenta
#5 -> cyan
#6 -> jaune
#7 -> orange
```

Les données dans le fichier `temps.dat` sont organisées (par le programme) de la manière suivante :

les données dans le fichiers `temps.dat` doivent être stockées sous forme de matrice à cinq colonnes. Pour chaque feux la première colonne représente le numéro du feu, la deuxième colonne est le temps de cycle qui est constant pour tout les feux, la troixième colonne doit être à 0, la quatrième colonne représente le début du vert et la cinquième colonne représente la fin du vert.

Chapitre 7

Conclusion

Le projet a été mener à son terme, dans la mesure où la chaîne de résolution du problème mise en place en language C autour du solveur fonctionne. Il a permis d'appliquer l'enseignement de recherche opérationnelle au domaine de la circulation automobile, sans chercher à développement le moteur de résolution du simplex, tâche dévolue à un programme tiers.

L'intérêt technique principal de ce projet réside d'une part dans l'implémentation de types abstraits de données utilisant l'allocation dynamique de mémoire, et d'autre part dans le maniement et le traitement des fichiers à l'aide de la bibliothèque standard du C.

Un intérêt secondaire apparaît en ce concerne la nécessité de se plier aux exigences des programmes tiers utilisés, à savoir lp_solve (très capricieux) et GNUPLOT (plus accessible). Cela requiert une lecture approfondie des documentations fournies avec ces derniers.

D'un point de vue des compétences humaines, ce projet nous a permis d'être à nouveau confrontés au travail d'équipe et aux difficultés qui en découlent. En effet, la communication est coeur des décisions : il faut faire comprendre ses idées aux autres et trouver le meilleur compromis qui satisfasse à la fois les membres du projet et le problème posé. Ainsi ce qui a été avant tout mis en avant dans le projet est la démarche de l'ingénieur dans une entreprise : savoir convaincre et communiquer au sein d'une équipe, se motiver et répartir le travail afin d'avancer au mieux.

Troisième partie

Annexes

Annexe A

Procédure de tests

Pour compiler les fonctions de tests, il faut taper la commande suivante :

```
make test
```

Pour activer, le mode de deboggage c'est à dire l'instrumentation du code pour les deboggeurs, il faut taper la commande suivante :

```
make DEBUG=yes
```

A.1 Tests sur le TAD

Nous avons implémenté des fonctions qui testent les primitives des types abstraits de données.

On peut les trouver dans le dossier `test`.

A.1.1 Carrefour

Les fonctions relatives au TAD sont définie : `test/carrefourTest.c`.

On teste dans la fonction nommée testCarrefour_initialiser, nous faisons dans un premier temps une initialisation du carrefour en y mettant des valeurs définies.
Nous véfifions par la suite la validité de l'initialisation en vérifiant la valeur stockée dans le TAD avec la valeur de référence.

Nous supposons que la fonction Carrefour_detruire qui désalloue l'espace memoire réservé fonctionne correctement (simplicite du codage). Nous l'appelons au travers de testCarrefour_detruire avant de terminer la procédure de test.

En analysant les valeurs de retour des fonctions de test nous pouvons savoir si la procédure s'est passée comme prévu.

- 0 correspond à : test réussi
- 1 correspond à : test échoué

A.1.2 Feux

Les fonctions relatives au TAD sont définies dans : `test/feuTest.c`.

À ce niveau, on suppose que les primitives du TAD Carrefour sont correctes.

Dans un premier temps, nous allons tester la fonction Feu_ajouterConflit. Nous réutilisons la même méthode que pour tester les primitives du TAD Carrefour.
Nous initialisons un conflit puis nous vérifions les valeurs contenues dans le TAD feux.

Nous testons aussi le chaînage des conflits. Nous ajoutons donc un second conflit. Nous vérifions ensuite l'initialisation de tous les conflits crées.

Nous testons ensuite la fonction Feu_lireConflit, en initialisant explicitement le TAD puis en comparant les valeurs que l'on lit avec la fonction avec celles de références.

Á ce stade, nous considérons que les fonctions testées précédemment fonctionnent correctement.

Nous allons tester la fonction supprimerConflit en utilisant la fonction ajouter-Conflit. On apelle la fonction supprimerConflit puis on vérifie que l'espace a bien été libéré.

Grâce aux valeurs de retour des fonctions de tests, on peut décider du bon fonctionnement des primitives du TAD Feu. Les valeurs de retour suivent la règle suivante :

- 0 correspond à : test réussi
- 1 correspond à : test échoué

A.1.3 Test global

Nous avons soumis le programme à Valgrind, en le testant sur deux exemples différents.

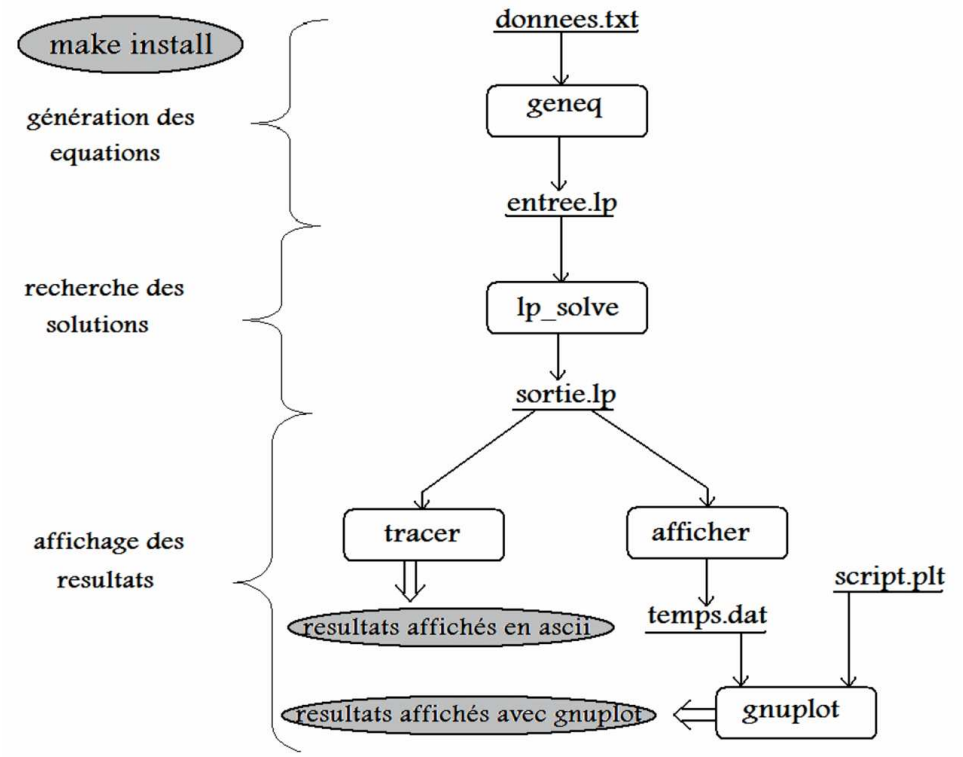
Le programme ne contient pas d'erreur (initalisation) et aucune fuite de mémoire.

Annexe B

Mode d'emploi

B.1 Vue d'ensemble du déroulement du programme

Ce diagramme reprend les différentes étapes du déroulement du programme en détaillant les fichiers en entrée et en sortie de chaque sous-programme :



Expliquons à présent les points importants concernant la compilation et l'utilisation des différents programmes.

B.2 Choix d'une fonction économique

Le génáteur de la fonction économique et des contraintes sur le débit a été écrits en deux versions (modification des encadrants de R.O. sur les équations).

On peut choisir la version en tapant la commande suivante :

```
make VERSION=x
```

avec x respectant la règle suivante :

- x=1 les équations sont celles du sujet.
- x=2 les équations sont celles proposées par les encadrants de R.O..

Remarque : Par defaut si l'option `VERSION` n'est pas spécifiée alors le programme est compilé avec les équations du sujet.

B.3 Compilation

Il faut disposer de GNU Make sans quoi le programme ne pourra se compiler. Il faut aussi avoir GNU gcc (recommandé) ou un compilateur C équivalent.

Il suffit de taper dans un shell, la commande suivante :

```
make
```

Remarque : Sur les machine Solaris de l'ENSEIRB, il faut au préalable taper la commande :

```
source ./setenv
```

À la fin de la compilation, nous disposant d'un programme se nommant `geneq`. La syntaxe de `geneq` est :

```
./geneq <fichier donnee> <fichier cible> <nombre de feux>
```

Si l'on dispose de donnée exploitable, on peut exécuter le programme. On peut ensuite faire traiter le fichier par le solveur lp_solve :

```
lp_solve <fichier cible>
```

Si on souhaite disposer d'un fichier contenant les résultats en sortie de lp_solve, la commande précédente devient :

```
lp_solve <fichier cible> >> <fichier resultat>
```

On dispose alors d'un fichier contenant tous les résultats.

B.4 Affichage

Pour afficher les resultats, nous proposons deux méthodes :
- avec des caractères ASCII, en utilisant la commande :
  `./tracer <fichier de sortie lp_solve> <fichier de données>`

– avec GNUPLOT, en générant le fichier `temps.dat` avec la commande :
`./affiche <fichier de sortie lp\_solve> <fichier donnee> temps.dat`
puis en faisant appel à GNUPLOT avec la commande :
`gnuplot -persist script.plt`

Le fichier `script.plt` fait appel de son tour au fichier `temps.dat` qui est déjà généré.