Université de Cergy-Pontoise

Maîtrise informatique - 2004/2005

Projet d'informatique théorique

Jean-Philippe Aumasson

Florent Kaisser

Jean-Baptiste Langlois

Sandrine Phcar

Réalisation d'un simulateur de machines de Turing et machines à registres

5 janvier 2005

Table des matières

Première partie

Introduction

Le présent document constitue le rapport de nos activités lors de la réalisation du projet d'informatique théorique. Le logiciel à concevoir est un simulateur de deux modèles de calculs équivalents, étudiés en cours : les machines de Turing et machines à registres. Les fonctions imposées sont détaillés dans le sujet, joint au programme.

Notre période de travail s'étend de mi-octobre à la première semaine de janvier, soit dès l'annonce verbale et la description grossière du sujet jusqu'à l'échéance. L'équipe est formée au début de cette période, et les rôles rapidement répartis d'un commun accord.

Une étape préliminaire est l'estimation de la masse de travail nécessaire à la réalisation d'un programme répondant à nos attentes, et l'identification des principales difficultés. Comme annoncé par le responsable du projet, et de façon évidente, l'implémentation et éventuellement création d'un algorithme de placement du graphe représentant les machines constituera certainement le point le plus complexe du travail, nous en faisons notre priorité dès le départ. Plus tard, la conception de variantes de machines de Turing risque d'être source de réflexions et choix déterminants. Nous tentons alors d'en déduire, sinon une hasardeuse approximation du temps nécessaire, une idée de l'intensité du travail à fournir, qui sera dès le départ quasi-quotidienne.

Une page fut immédiatement créée sur Internet, afin de recencer et partager des liens utiles, et surtout de fournir un espace hébergeant les sources et autres fichiers crées au cours du développement. Par prudence, l'accès à cette page est seulement réservé aux détenteurs d'un couple login/password.

Un point à ne pas négliger dès le départ est l'organisation du travail en équipe, dont le principal objectif est l'optimisation de la communication entre chaque membre, résidant d'une part dans les medias utilisés, d'autre part, plus qualitativement, en la capacité d'inter-compréhension et de s'accorder quant aux objectifs à remplir et méthodes à utiliser. De nombreux débats et discussions, lors de réunions ou improvisés dans un couloir, nous ont permis d'échanger nos vues et de parvenir à faire converger nos opinions. Un point crucial du travail d'équipe a été la communication non physique, via mails et messagerie instantanée, qui, même si parfois entâchée d'aléas techniques perturbant, a clairement été décisive dans la mise au point de points délicats impliquant les travaux de plusieurs personnes. De plus la divergence de nos méthodes de travail et de programmation a parfois pu être un frein à la rapidité de la conception, de part nos confrontation d'opinions, mais est certainement un apport positif finalement quand, de plusieurs options, on choisit la meilleure.

Dans ce document, on s'attachera à décrire le résultat de notre travail en faisant état quand nécessaire de la chronologie des réalisations et des difficultés

rencontrées. On utilisera les conventions typographiques usuelles, la qualité des images pourra varier selon le logiciel de visualisation utilisé. Ne prétendant pas à l'exhaustivité, et par souci d'éviction du superflu, nous nous sommes limités aux points essentiels du développement.

Utilisation Le programme est réalisé en langage Java, on recommande d'utiliser un JDK à partir de la version 1.4, avec le 1.5 on aura cependant des avertissements pour la non gestion des types génériques, non implémentés dans le JDK 1.4. Les instructions de compilation et exécution sont spécifiées dans le fichier *README*. Le compilateur Java d'IBM, jikes, est supporté et pourra être invoqué par *make jinstall*.

Nous avons nommé notre programme SIMS, acronyme de *SIMS Is a Machine Simulator*.

Deuxième partie

Machines de Turing.

par Jean-Philippe Aumasson <jpnossa@free.fr>.

1 Machines de Turing

1.1 Fonctionnalités

On livrera une énumération non exhaustive des fonctionnalités implémentées pour les machines de Turing. Cette liste restitue grossièrement la chronologie du développement :

- Simulation de machines mono-bande et multi-bandes.
- Simulation de machines déterministes, non-déterministes par simulation déterministe, probabilistes, à gravité.
- Accepte les commentaires dans le fichier selon les conventions Java. accepte les chaînes de caractères dans les directives si entourées de guillemets, ou sans aucun guillemet.
- Quatre types d'états : acceptants, rejetants, point d'arrêt, ou standard (aucun comportement remarquable).
- Simulation de bandes infinies à gauche et à droite.
- Teste la validité des directives, dont le mot d'entrée et son appartenance au langage A^* (A étant l'alphabet définit, et le caractère vide).
- Définition d'un état initial avec la directive *#init* .

- Spécification de la nature de la machine par les directives *#nd*, *#prob*, *#gravity*.
- Vérification de la validité de la nature des valeurs associées aux directives.
- Gestion du caractère générique ('*') selon son acceptation usuelle.
- Indication du déplacement insensible à la casse.
- Affichage la complexité en temps (nombre de transitions effectuées) et en espace (nombre total de cases visitées).
- Instructions portant sur l'ensemble des bandes ou bien un de ses sous-ensemble non vide.
- Si la machine comporte une seule bande on ne précisera pas le numéro de la bande dans les transitions.
- Les états sont symbolisés par des chaînes de caractères de taille quelconque non nulle.
- Indication des erreurs de syntaxe, accompagnée du numéro de ligne dans le fichier.
- Vitesse ajustable, cinq niveaux proposés, plus un mode *turbo*.
- Simulation exécutée dans un *thread* indépendant du *thread* principal du logiciel.
- Possibilité d'attribuer un alphabet différent pour chaque bande.
- Modification des bandes pendant la simulation pour les machines déterministes.
- Visualisation du graphe correspondant et de l'état des bandes en temps réel avec affichage de l'état courant.
- Visualisation des bandes avec décalage au cas où la tête de lecture dépasse d'un bord de l'écran.
- Compilation et enregistrement d'un code de machine dans l'interface, compilation automatique si *reset* la machine.

1.2 Introduction

Les principales difficultés rencontrées lors de l'implémentation ont été le traitement du fichier, la gestion du caractère générique et des instructions ne posant pas de restriction sur toutes les bandes (nommées instructions variables, ou instructions multiples dans les commentaires des sources), puis les cas de non-déterminisme et leur simulation. Je détaillerai surtout ce dernier point, sans doute la partie la plus intéressante du projet.

L'étude préliminaire qui a consisté à étudier les différents formalismes et présentations de machines, m'a permis de comparer notamment les notations et terminologies, conventions adoptées, formes des transitions, rigueurs, etc... Cette étude répétée des machines m'a donc permis de les "penser" plus facilement lors de l'implémentation, et d'avoir une vision un peu moins étroite et rigide de ce modèle de calcul. Peut-être plus anecdotique est la lecture, à mon sens indispensable, de l'article originel d'Alan M. Turing, *On computable numbers, with an application to the Entscheidungsproblem* [1], où j'ai notamment découvert l'origine de certaines notations, et les libertés prises depuis par les divers

auteurs. Après l'implémentation d'une machine déterministe simple, il a ensuite été nécessaire de recenser les différentes variantes de machines. Dans les ouvrages étudiés, on trouve peu de divergences concernant les machines non-déterministes simples, mais davantage pour les machines probabilistes, dont le détail est donné plus bas.

Programmant en Java, langage orienté objet, une étape précédant la programmation fut d'identifier des classes candidates. Intuitivement on créera des classes pour une bande et la machine. J'ai finalement une classe *TmParser* qui lit le fichier et stocke les données collectées dans les structures non finales, puis la classe *MachineRead*, sous-classe de *TuringMachine*, prend le relais pour traiter ces données et les insérer dans les structures finales appropriées, notamment les tables de hachage pour les instructions. Ces tables de hachage ont comme clés des configurations modélisées par des chaînes de caractères décrivant l'état courant et le symbole lu sur chaque bande, les valeurs associées étant les mouvements et écritures sur les bandes, puis le nouvel état courant.

1.3 Traitement des fichiers descriptifs des machines

Pour *parser* le fichier, des outils d'analyse lexicale et syntaxique peuvent être utilisés afin de générer automatiquement un compilateur en Java. Cependant, étant donné la syntaxe assez stricte et succinte du fichier, j'ai préféré réaliser le *parsing* "à la main", en utilisant les primitives de lecture Java, principalement la méthode *readLine()*. J'ai d'abord crée une syntaxe et le *parser* correspondant jusqu'à la mise en ligne du sujet qui imposait une syntaxe particulière. Tout a donc été re-programmé mais relativement rapidement étant donné que le principe était identique, et les méthodes de lecture désormais familières. Le principe est assez simple : on lit une ligne, on identifie sa nature (directive, commentaire ou instruction), puis on traite cette ligne selon le cas. Les erreurs de syntaxe sont signifiées à l'utilisateur accompagnées d'un message descriptif de l'erreur, et du numéro de la ligne dans le fichier.

Les directives sont directement interprétées dans la classe *TmParser*, c'est à dire que les variables ou structures finales sont immédiatement modifiées (par exemple, on attribuera le nom de la machine à la lecture de la directive *#name*). Les éléments composant les instructions (soit chaque terme dans les parenthèse, séparés par une virgule) sont stockés dans un vecteur de vecteurs, pour faciliter leur traitement dans la classe *MachineRead*.

1.4 Traitement des instructions

On aura plusieures tables de hachage de transitions, pour les instructions concernant toutes les bandes, pour les instructions variables, et une dernière pour stocker les collisions, soit les instructions alternatives lors d'une machine non-déterministe. Dans l'implémentation, ces tables correspondent respectivement, dans la classe *TuringMachine*, à *moveState*, *moveStateUnsized*, *ndMoveState*.

On crée de plus une table de hachage *stateStatus* qui associe aux états disposant d'un statut particulier une chaîne de caractères décrivant ce statut ("acc", "rej", "stop", respectivement pour des états acceptant l'entrée, la rejetant, et pour les points d'arrêts).

La gestion du caractère générique consiste à insérer dans la table adaptée les clés correspondant à toutes les substitutions du caractère par les symboles de l'alphabet dans la clé le contenant. Par récurrence on traitera la présence de plusieurs caractères génériques (pour plusieurs bandes). Les deux situations possibles sont le cas d'un caractère générique en tant que symbole lu et écrit, et comme symbole lu seulement. L'interprétation de ces situations est évidente, et précisée dans le sujet du projet.

1.5 Variantes des machines de Turing

La variante triviale de machines à plusieures bandes n'est pas détaillée ici, mais évidemment implémentée. La variante courante de machine à demi-bande (seulement infinie d'un côté) n'est pas gérée, mais apporterait peu d'interêt. On s'intéressera donc seulement aux machines non-déterministes, mono-bandes ou multi-bandes.

Dans l'implémentation, la nature d'une machine est spécifiée par un ensemble de booléens vrais ou faux, selon la nature de la machine. Une machine probabiliste sera définie d'une part comme non-déterministe, d'autre part comme machine probabiliste. Pour réaliser les transitions de la machine, une méthode *nextConf()* est appellée, qui selon les cas appelle *nextDConf()* ou *nextNDConf()*, respectivement pour des machines déterministes et pour des machines non-déterministes (sans forcément faire face à une indétermination). La méthode *nextNDConf()* gère les trois types de machines non-déterministes présentées ci-dessous. Au niveau utilisateur on spécifiera la nature de la machine dans son fichier descriptif par les directives *#nd*, *#prob*, et *#gravity*, dont la valeur associée sera soit *oui* (ou *yes*, insensible à la casse), ou *non* (*no*). Selon la hiérarchie spécifiée, on n'aura pas nécessairement besoin de spécifier qu'une machine probabiliste est non-déterministe, ou qu'une machine à gravité est probabiliste.

1.5.1 Machines non-déterministes simples

Ces machines de Turing ont la spécificité de pouvoir atteindre une configuration donnant lieu à plusieurs choix possibles d'instructions. Un choix, arbitraire ou non, doit alors être fait, et selon les choix effectués, le calcul de la machine sera différent d'un ensemble de choix à un autre, et éventuellement le résultat, si la machine s'arrête, et si le temps de calcul est raisonnable.

Dans le programme, nous simulons classiquement de façon déterministe ces machines non-déterministes, et le premier état acceptant rencontré arrêtera la machine. On poursuivra le calcul pour tout autre type d'état, la machine peut donc

ne jamais s'arrêter, l'utilisateur aura la possibilité de remettre à zéro la machine (*reset*) lors d'une pause ou pendant le calcul.

L'algorithme de simulation est fortement inspiré de la preuve du théorème de simulation déterministe de toute machine non-déterministe de l'ouvrage de Hopcroft [2]. La principale divergence réside dans le stockage des configurations à explorer ; là où une bande annexe est utilisée dans la preuve du théorème, j'ai préféré stocker les configurations en tant qu'instances d'une classe *Configuration* dans une structure *FIFO* (classe *Fifo*). Une configuration correspondant au contenu des bandes, à la position des têtes de lecture et à l'état courant, une instance de *Configuration* stockera des instances de la classe *Tape*, l'état courant (chaîne de caractère, objet *String*).

Les valeurs de complexité affichées (complexité en temps et complexité en espace) correspondent à la complexité résultant d'un parcours déterministe jusqu'à la configuration courante, et non à la complexité de la simulation du non-déterminisme.

Un exemple joint au programme simule le comportement non-déterministe d'un robot parcourant la bande à la recherche de l'arrivée (détails dans le fichier, *robot_ND.tur*), cet exemple ludique ne reflète certainement pas la puissance du non-déterminisme, mais permet une visualisation évidente des choix à effectuer à travers la métaphore du robot.

1.5.2 Machines probabilistes

La simulation de ces machines donnent lieu à un "vrai" non-déterminisme, leur principe de fonctionnement étant le suivant : à l'occurence d'une indétermination, le choix est fait aléatoirement de façon équiprobable sur l'ensemble des transitions possibles.

Cela revient à attribuer une probabilité égale à l'inverse du degré d'indétermination local à chaque transition candidate. L'algorithme stochastique utilise comme générateur de nombres pseudo-aléatoires la méthode *random()* de la classe *java.lang.Math*, suffisante pour nos simulations.

Au niveau de l'implémentation on utilise les structures d'une machine déterministe, le comportement changeant à la rencontre d'une indétermination ; au lieu de stocker les transitions possibles et de toutes les effectuer, on choisit pseudo-aléatoirement une des transitions possibles.

L'exemple d'un générateur aléatoire d'anagrammes est fourni dans le fichier *anagram_P.tur*.

1.5.3 Machines à gravité

Ce modèle, à ma connaissance inédit, est un prototype de machine probabiliste à probabilités non-uniformes, et nommé ainsi pour son analogie avec le principe

naturel de gravité : chaque état possèdera une valeur correspondant à sa masse, et lors d'une indétermination on créera virtuellement un vecteur stochastique pour pondérer les probabilités d'accession aux états par leur masse. Virtuellement, car dans l'implémentation le vecteur *selectionner* aura comme taille la somme des poids des états, pour un état de masse N on créera N entrées dans le vecteur, chacune associée à l'indice de la transition dans le vecteur *ndQueue*, stockant les transitions possibles pour chaque indétermination. On génère ensuite aléatoirement un nombre entier compris entre zéro et la taille du vecteur *selectionner* pour déterminer la transition élue.

Ce modèle de machine sera équivalent aux machines de Turing déterministes, au même titre que les machines probabilistes simples. On admettra ce résultat.

Ces machines simulent un phénomène naturel, et peuvent aboutir à des comportements particuliers : on pourra par exemple avoir une machine dont la probabilité qu'elle s'arrête tende vers zéro, sans jamais être nulle. Une machine pourra aussi simuler des notions physiques d'équilibre et d'inertie, par exemple en aboutissant à un équilibre, autrement dit à un chemin type dont la probabilité qu'on le poursuive tend vers l'unité, ou en persistant dans un état bouclant sur lui-même, ayant alors de plus en plus de mal à "s'en sortir ".

Des variantes non-implémentées de ces machines pourraient avoir pour principe de fonctionnement une sélection systématique de l'état de plus forte masse, avec recours au "hasard" si concurrence, ce qui permettrait de simuler un semi-déterminisme, ou encore un modèle avec des transitions en six-uplets, le terme ajouté étant l'état global : cet état correspondant à l'état du système de plus forte masse. L'avantage de ce dernier modèle est la définition d'une contrainte supplémentaire qui pourrait permettre de minimiser la complexité dans certains cas. Cependant, la difficulté pour le concepteur d'établir au préalable des transitions selon l'état global, constitue un frein au développement de ce type de machines.

Un exemple de marche aléatoire est livré avec le programme (*marchealeat_ G.tur*), ainsi qu'une machine décrivant une chaîne d'états dont le i-ème état est lié aux i+1 et i+2-ième de façon circulaire, la chaîne tend à se stabiliser vers un état d'équilibre (*chained_ G.tur*). Ces deux machines ne s'arrêtent pas, étant donné qu'elles servent à montrer le comportement du système pour une durée longue.

L'exemple *randomized_ G.tur* illustre le cas d'une machine pouvant toujours s'arrêter mais dont la probabilité peut tendre vers zéro à l'infini. Bien entendu, le hasard fait que le phénomène ne se produit pas à chaque calcul de la machine. Elle génère une suite de bits aléatoires de taille aléatoire finie (car on a toujours $P(arrt) > 0$), soit un nombre aléatoire. La "quantité" d'aléatoire de ce nombre, si on considère la période du début à la fin du calcul, est déterminée par celle du générateur semi-aléatoire utilisé. On pourra s'intéresser à la probabilité d'arrêt avant un instant donné, ce qui revient à étudier la chaîne de Markov non-homogène correspondant à cette machine, mais celà apporte peu d'interêt dans la mesure où on occulte alors la notion de conditionnement par rapport au contenu de la bande et aux déplacements effectués sur celle-ci.

2 Visualisation graphique et interface utilisateur

Dès l'implémentation des machines déterministes, j'ai crée une visualisation graphique des bandes et les commandes de contrôle associées afin de pouvoir tester plus confortablement le programme, et en vue de l'interface utilisateur finale, qui devra en plus comporter un éditeur de texte, ainsi que le graphe.

La visualisation des bandes utilise de simples fonctions graphiques de Java (classe *Graphics*). Les bandes sont présentées horizontalement, la position de la tête de lecture est formalisée par une coloration en rouge de la cellule courante. Initialement cette visualisation était placée au centre de l'écran, la barre de commandes au-dessous. Plus tard allait se poser le problème du placement des composants du programme : la fenêtre de visualisation des bandes, celle du graphe (réalisée par Jean-Baptiste), et la fenêtre principale de la machine comportant l'éditeur de texte, les commandes et les boutons de fonctions, ces deux derniers étant amovibles, et "dockables" à la fenêtre (réalisation par Florent, que j'ai ensuite adaptée à mes machines en créant la classe *TmLauncher*).

Le choix d'adapter la taille des fenêtres en fonction des dimensions de l'écran se serait avéré trop fastidieux, étant donné que l'on devrait aussi adapter les icônes de boutons, le graphe, les bandes. Nous avons choisi de réaliser une application utilisable pour les dimensions standard d'écrans de 1280*1024, 1024*768, et éventuellement 800*600, alors beaucoup moins ergonomique.

La fenêtre principale d'une machine est placée à son lancement au sommet haut gauche de l'écran, afin d'être dans le champ de vision direct de l'utilisateur, cette fenêtre étant généralement la première qu'il regardera (le fichier contient les spécifications et commentaires sur la machine), et la seule avec laquelle il pourra interagir, soit avec la souris, soit par les raccourcis clavier.

Le graphe est situé au sommet haut-droit de l'écran, constituant hiérarchiquement le second composant de la machine, étant donné que c'est la représentation usuelle, et permet une meilleure compréhension que les lignes d'instructions brutes, mais ne gère aucune interaction avec l'utilisateur, hormis les fonctions de redimensionnement et de fermeture de la fenêtre.

Pour optimiser le confort visuel, j'ai jugé que la partie haut gauche (vue de l'utilisateur) était la zone où la vision de l'utilisateur était la plus concentrée, et où il serait naturellement amené à porter son regard (notamment par habitude des icônes sur le bureau, du menu "Démarrer" de *Windows* ou *KDE*, et surtout de la position du visage par rapport à l'écran, les yeux généralement à près de deux tiers du bas vers le haut).

La fenêtre de visualisation des bandes (classe *TuringView*) est seulement intéressante lorsque la simulation a été lancée, ou bien pour vérifier que le mot en entrée est bien écrit sur la première bande. Elle est donc située au bas gauche de l'écran, car c'est sur ce côté que l'on trouve le plus d'espace libre. La capacité infinie des bandes est simulée par des décalages de l'affichage, et des barres de scrolling sont ajoutées à la fenêtre, horizontalement pour pouvoir lire le contenu

de la bande le plus à droite, et verticalement pour visualiser toutes les bandes (défilement nécessaire à partir de quatre bandes).

L'affichage des complexités et de l'état courant se trouve en en-tête sur cette fenêtre. Les icônes de la fenêtre principale ne sont pas des créations originales, mais appartienent au kit d'icônes *Noia*, libre d'utilisation dans des logiciels non commerciaux.

Les fonctions réalisables par interaction avec l'utilisateur sont les suivantes :

- Contrôle de la simulation d'une machine, avec affichage des messages correspondant dans la fenêtre principale : démarrage, pause, redémarrage, *reset* (recompile, remet à zéro, et calcul démarré d'office si il était déjà lancé, à l'arrêt sinon), augmentation et diminution de la vitesse. Le mode turbo correspond à la vitesse zéro : la visualisation graphique est désactivée pour accélérer le calcul. C'est notamment utile pour les *busy beavers*, et pour évaluer le comportement de machines pour un grand nombre de transitions.
- Contrôle du chargement de la machine : enregistrement du fichier, compilation (enregistre automatiquement).
- Contrôle de l'interface : déplacement, redimensionnement et masquage des fenêtres, "dédockage/dockage" des boutons de contrôle, fermeture de la machine (ferme toutes les fenêtres liées à cette machine, termine le *thread*, et isole les objets en mémoire pour ramassage par le garbage collector).

3 Conclusion personnelle

L'implémentation des machines de Turing décrite ci-dessus, bien que fonctionnelle, est loin d'être complète et ne présente pas l'allure d'un "produit fini". L'idée de machine à gravité aurait certainement mérité plus de développement, et les algorithmes mis en oeuvre ne sont pas non plus des modèles d'optimisation. La robustesse et la rapidité du programme en est affectée, une plus grande rigueur de conception ayant fait défaut. Cependant le programme fonctionne correctement dans la grande majorité des cas, la quantité de bugs est réduite, et semble dépendre, après tests, de la plateforme et de la machine. Les divers exemples joints illustrent les fonctionnalités implémentées, et certains permettent de tester les limites du programme, comme les *busy beavers* ou les machines à gravité infinies.

Pour une éventuelle poursuite du développement on pourra d'abord s'attacher à optimiser les algorithmes, ou en employer d'autres de complexité réduite. L'implémentation d'autres variantes, telles les machines à oracles, pourrait être envisagée, ainsi que la détection de non-arrêt, par l'identification de configurations récurrentes. On devrait alors le choix entre se munir d'un large espace mémoire, et stocker de façon compressée les configurations calculées.

Troisième partie

Machines à registres, IHM.

par Florent Kaisser <florent.kaisser@free.fr>.

4 Machines à registres

4.1 Modélisation de la machine

Le langage de programmation utilisé étant de type objet, il semble intéressant d'utiliser une modélisation UML. Nous avons alors réfléchi sur la définition d'un diagramme de classe qui modélise une machine à registre. Les différentes classes retenues sont dans un premier temps :

- Machine à registres : simulation d'une machine à registres
- Programme : graphe correspondant à la machine à registre
- Graphe : ensemble de sommet et d'arc
- Sommet : sommet du graphe
- Arête : arc du graphe
- Instruction : sommet correspondant à un état de la machine
- Registre, début, fin : Instructions
- RegistreMoins : instruction R- (contenant sa valeur)
- RegistrePlus : instruction $R+$ (contenant sa valeur)

Après une première implémentation de ces classes (comprenant le premier analyseur syntaxique), on s'est aperçu que la modélisation ne convenait pas : un registre ne peut être utilisé qu'une seule fois dans la machine, car pour chaque instruction de type registre on crée un registre. On ajoute alors la notion de commande en l'ajoutant à notre panoplie de classes. Une commande est associée à chaque instruction. Cette commande peut être un état *Fin*, *Début* ou bien un registre (on a renommé la classe *Instruction* en *Commande*, et ajouté une classe *Instruction*, qui correspond à un état ou sommet). Donc il suffit de créer une commande pour chaque registre, une pour *Début* et une pour *Fin*. Chaque instruction correspond à un état (sommet) de la machine.

Une fois les quelques tests réalisés avec le nouvel analyseur syntaxique, une deuxième erreur apparaît dans la conception : les valeurs des registres sont stockées dans les objets *RegistreMoins* et *RegistrePlus* ce qui qui pose beaucoup de problèmes, car il devrait y avoir une valeur commune. L'erreur est de dire que "les valeurs sont stockées dans les etats", ce qui est erroné. Il serait plus judicieux de stockér toutes les valeurs des registres dans le programme (classe *Machine*). On associe alors les registres à la classe *Programme* (ce qui est plus évident, les registres appartenant à la machine). Et à chaque commande de type $R+$ ou R-

sera associée un registre. Un registre à donc une valeur, un indice, et peut être incrémenté ou décrementé. Notre modélisation finale est donc finalement :

- Machine à registre : simulation d'une machine à registres.
- Programme : graphe correspondant à la machine à registres.
- Graphe : ensemble de sommets et d'arcs.
- Sommet : sommet du graphe.
- Arête : arc du graphe. Peut être de deux types : *0* (la transition effecteur âpres un R- dont le registre est à zéro) ou *1* (par défaut).
- Instruction : sommet correspondant à un état de la machine
- Registre : objet stockant la valeur d'un registre.
- Début : commande associée à l'état *Début*.
- Fin : commande associée à l'etat *Fin*.
- RegistreMoins : commande associée à un état R-.
- RegistrePlus : commande associée à un état $R+$.
- Commande : associée à chaque instruction. Correspond à l'action à effectuer à chaque fois qu'on arrive dans l'état.
- CommandeRegistre : Commande associée à un état $R+$ ou R-.

Le diagramme de classe est livré en annexe (fichier UML_reg.gif).

Chaque instruction (état ou sommet) étant associé à une commande, pour faire tourner la machine il suffit de parcourir le graphe, et exécuter chaque instruction (état de la machine). L'instruction exécute à son tour la commande associée. Cette commande pouvant être *Début*, *Fin*, $R+$ ou R-. A l'issue de l'exécution de cette commande, on peux savoir la transition à effectuer (le choix ne se limite qu'après l'exécution d'un R-). On passe à l'état suivant, jusqu'à qu'il n'y ai plus de successeur (état *Fin*).Pour optimiser la recherche de successeurs, la classe *Programme* crée lors de son initialisation une matrice de successeurs de dimension *N*2* (N étant le nombre de sommet), qui permet d'associer à chaque sommet ses successeurs. Un graphe d'une machine à registre ayant au maximum deux successeurs, deux colonnes suffisent : une pour les successeurs de type *0* (transition après un *dec*, si le registre est à zéro), et une pour les successeurs de type *1* (transition par défaut). Pour connaître la prochaine instruction à exécuter il suffit alors d'aller la chercher dans la matrice en fonction de la transition souhaitée.

5 Analyse syntaxique

C'est le point délicat, et indispensable, de la simulation d'une machine. La machine doit être programmée dans un simple fichier texte. Ce fichier doit être codé selon une syntaxe particulière pour décrire la machine. Étant donné qu'on a commencé à réfléchir sur le projet deux semaines avant la publication de l'énoncé, on a d'abord créé notre propre syntaxe, qu'on nommera MAR (l'extension des fichiers étant "mar"). Cette syntaxe repose sur la description du graphe, chaque ligne correspondant à un sommet. La syntaxe étant par la suite imposée, on

décide d'abandonner les fichiers de type MAR. Accessoirement, la syntaxe s'est révélée incorrecte (à cause de l'erreur de modélisation des registres, voir plus haut).

Le travail à effectuer est donc de créer un analyseur syntaxique (parseur) de fichier REG (syntaxe imposée par l'énoncé).

Après réflexion, l'analyse se fera sans outil d'analyse particulier (JFlex, ANTLR...), au vu de sa simplicité. On a considéré qu'on passerait plus de temps à apprendre à se servir d'un outil, que "réinventer la roue" pour ce cas particulier. L'analyse du fichier se fait caractère par caractère, en utilisant une variante d'un automate à pile implicite (appel récursif de fonctions). C'est une variante car des libertés ont été prises, pour simplifier le problème (optimisation, gestion des commentaires, etc...).

L'analyse se fait en deux grandes étapes indépendantes : lecture des directives, puis lecture des instructions (code de la machine).

Une directive est écrit sous cette forme : #<*directive*>=<*valeur*>. L'analyseur cherche d'abord le caractère dièse ('#') , puis lit le nom de la directive, jusqu'à ce qu'on tombe sur '=', et lise la valeur.

Ensuite l'étape la plus compliquée : la lectures des instructions. L'analyseur lit des instructions jusqu'à ce qu'il n'y ait plus de caractère à lire (fin du fichier).

On définit un label comme étant soit une commande (*dec* ou *inc*), soit un descripteur qui définit un sommet du graphe (définition différente de l'énoncé).

Lors de la lecture d'une instruction trois cas peuvent se présenter :

1. C'est la défintion d'une instruction, par exemple :
   saut1 :
   inc 2 fin
2. C'est un saut dans une commande, par exemple :
   `inc 2` *saut1*
3. C'est la définition d'une instruction sans descripteur, par exemple :
   `dec fin`
   inc 2 saut1

Les caractères en italique correspondent à ceux lu lors de la lecture de l'instruction. La lecture d'une instruction renvoie toujours une instruction, correspondant soit à une existante (cas 2), soit à une nouvelle (cas 1 et 3).

Lorsqu'on crée une nouvelle instruction on doit lui définir la commande à exécuter (*dec*, *inc* sur un registre, *début*, ou *fin*) et son ou ses successeurs (transition). Un successeur correspondant à une instruction, on fait donc un appel récursif de la lecture d'une instruction. L'instruction renvoyée étant l'un des successeurs.

Lorsque l'instruction a été créée (cas 2), on renvoie tout simplement l'instruction correspondante.

Il se peut qu'un saut soit placé avant sa définition, dans ce cas on crée quand même l'instruction mais on ne lui affecte aucune commande, ni successeur. Ces affectations se feront lors de la définition de cette instruction.

La lecture d'une instruction se déroule selon l'algorithme suivant :

```
LireInstruction(I) : Instruction
Debut
 Lire label (L)1
 Si le L n'est pas une commande
 Alors
  // le label est un descripteur de sommet
  Si L est un descripteur d'une instruction deja définie
  Alors
   // retourne l'instruction correspondant au descripteur
   I<-DescripteurIntruction(L)
  Sinon
   // crée une instruction avec comme descripteur le label
   I<-Créer une instruction(L)
  FinSi
  // définition d'une instruction, dans ce cas, suivi du caractère ' :'
  Si le caractère courant est ' :'
  Alors
   ProchainCaractere() // lit le caractère suivant
   // c'est la définition d'une nouvelle instruction
   // le nom du descripteur doit être suivi par une commande
   // on lit la nom de la commande
   Lire label(L)
   // la commande sera lue plus bas
  Sinon
  // c'est un saut vers l'instruction correspondant au label
  // on renvoie l'instruction correspondante (et on quitte la fonction)
  Retourner I
  FinSi
 Sinon
  // c'est la définition d'une instruction sans descripteur
  I<-Créer une instruction()
  AjouterSommet(I)
 FinSi
 // lit la commande
 Lire entier (R)
 Si L = = 'Dec'
 Alors
  Lire Instruction(IZ) // correspondant à la transition 0
  Lire Instruction(IN) // correspondant à la transition par défaut
  A<-CreerArrete(I,IZ,0)
  AjouterArrete(A)
```

```
  A<-CreerArrete(I,IN,1)
  AjouterArrete(A)
  I.commande.type='Dec'
  I.commande.idRegistre=R
 Sinon
  Si L = = 'Inc'
  Alors
Lire Instruction(IN) // correspondant à la transition par défaut
   A<-CreerArrete(I,IN,1)
   AjouterArrete(A)
   I.commande.type='Inc'
   I.commande.idRegistre=R
  FinSi
 FinSi
Fin
```

On identifie un espace comme toute suite de caractères suivants :

- Espace.
- Tabulation.
- Retour à la ligne (\r et \n).
- La virgule.

Deux labels d'instruction dans une commande *dec* (ex : `dec 2 saut1,saut2`) peuvent donc être séparés par un espace quelconque (virgule non obligatoire).

Un espace sépare chaque mot du langage de la syntaxe.

Si l'analyseur, à la suite de la lecture d'un caractère, rencontre un commentaire, il lit alors (sans les interpréter) tous les caractères du commentaire, puis il lit le premier caractère suivant le commentaire. Donc pour la fonction qui lit les instruction et les directive, les commentaires n'existent pas.

La définition de l'instruction *fin* n'est pas nécessaire, car elle n'a pas de successeur, ni de commande à exécuter. Si l'instruction *fin* est quand même écrite dans le fichier source (*fin :* à la fin du fichier), l'analyseur l'ignore.

Pour garder une correspondance entre le nom de l'état (instruction) et son objet (instance d'*Instruction)* correspondant, on utilise une table de hachage, avec comme clé le nom. A chaque fois qu'on souhaite obtenir une instruction à partir de son nom (descripteur), on invoque la table de hachage. Si cette instruction n'est pas encore rentrée (elle devrait l'être par la suite), on la crée, et on l'ajoute à la table.

Au moment de parser le fichier, si l'analyseur tombe sur un caractère non reconnu par la syntaxe (par exemple un caractère de l'alphabet, alors qu'un entier est attendu), il stoppe son analyse et déclenche une erreur, tenant compte de la ligne d'occurence de l'erreur dans le fichier. Cette erreur est communiquée au programme à l'aide d'une exception Java. D'autres cas peuvent provoquer une erreur de syntaxe, comme une instruction non définie (figurant dans un saut, mais non défini par la suite), on alors la fin du fichier non attendue.

6 La simulation

La simulation de la machine à registres consiste à exécuter l'ensemble des instructions de la machine. Entre l'exécution de deux instructions on peut soit ne rien faire, soit attendre (selon la vitesse de simulation) ou soit s'arrêter (point d'arrêt, ou mise en mode pas à pas).

La simulation s'exécute dans un *thread*, pour éviter de bloquer l'IHM.

Pendant la simulation, l'IHM peut alors demander au simulateur les actions suivantes :

- Charger un programme : charge un programme à partir d'une fichier.
- Démarrer la simulation : exécute toutes les instructions restantes jusqu'à la fin du programme. Permet de quitter le mode pas à pas (mise au point).
- Arrêter la simulation : rentre en mode pas à pas.
- Réinitialiser la simulation : revient au début du programme, en réinitialisant les registres à leurs valeurs initiales.
- Prochain pas : exécute l'instruction suivante et s'arrête.
- Définir la vitesse : définit la vitesse d'exécution (temps d'attente entre deux exécutions d'une instruction).

Le simulateur commence toujours en mode pas à pas.

Lors de la simulation, l'utilisateur doit être en mesure d'observer les changements d'état de la machine, d'identifier les instants de début, fin, et point d'arrêt. Pour celà le simulateur alerte l'IHM par l'intermédiaire d'actions. Chaque évènement correspond à une méthode de la classe de l'IHM. Le simulateur appelle donc ces méthodes quand nécessaire.

7 L'IHM (Interface Homme-Machine)

L'interface sera quasiment analogue pour les deux types de machines, les principales fonctions étant communes, notamment pour le contrôle de la simulation. Au niveau de l'implémentation ceci est formalisé par les interfaces *InterfaceSimulation* et *GrapheInterface*, implémentée par les deux machines. Les divergences seront principalement visuelles (placement, icônes).

On peut diviser l'IHM en plusieurs composants, chacun identifié à une fenêtre de l'environnement graphique :

- La fenêtre principale du programme, ouverte au lancement, permettant d'ouvrir ou créer une machine.
- La fenêtre principale de la machine, apparaissant à l'ouverture ou création d'une machine, composée de l'éditeur de texte et des boutons de contrôle de la simulation (démarrage, arrêt), de la visualisation (masquage des fenêtre), et de la machine (compilation, enregistrement).
- La fenêtre propre au graphe.

– La fenêtre d'affichage des registres, ou celles des bandes, respectivement pour les machines à registres et pour les machines de Turing.
– Une fenêtre d'affichage de messages textuels (informations, avertissement, erreurs).

7.1 L'éditeur

Le logiciel permet d'éditer directement le code de la machine. Pour cela on a créé un composant d'édition de texte, héritant de la classe *JTextPane*. Ce composant offre donc toutes les fonctionnalités d'un *JTextPane*, mais avec la possibilité de surligner une ligne de code. Après avoir lu la documentation de la classe *JTextPane*, on s'est aperçu qu'*a priori*, toute opération qu'on peux faire sur le texte , est indexé par la position d'un caractère et non d'une ligne comme on le souhaite.
L'idée serait alors de faire un correspondance entre numéro de ligne et numéro de caractère. Un simple tableau indexé par le numéro de ligne et contenant la position du caractère de retour à la ligne permet d'effectuer cette correspondance. On peux donc retrouver la position du premier et du dernier caractère d'une ligne. Ce tableau est mis à jour lors d'un chargement ou d'un enregistrement, ce qui est suffisant car on ne peux pas exécuter le programme si le code à été modifié, et une erreur de compilation a toujours été précédée d'une recompilation et donc d'un enregistrement. Avec cette information on peut demander à *JTextPane* de sélectionner cette ligne. Mais le résultat n'est pas celui attendu, pour deux raisons : la ligne est surlignée seulement là où il y a des caractères (pas de surlignage entre le dernier caractère de la ligne et l'extrémité droite de la fenêtre), et si l'utilisateur souhaite sélectionner une zone avec son curseur, il enlève le surlignage.
La seconde idée serait de demander un surlignage des caractères de toute la ligne à *JTextPane*, ce qui est possible, mais ne résoud pas le premier problème.
La troisième idée (la bonne!) est d'utiliser le système d' *Highlight* de la classe *JTextPane*. Avec ceci on peut dessiner sous le texte pour caracteriser le surlignage. Donc à chaque rafraîchissement de la fenêtre (soit à chaque pas du calcul), *JTextPane* appelle une méthode calculant la position du premier caractère de la ligne à mettre en évidence, puis dessine un rectangle, jaune ou rouge selon le cas, sur toute la largeur de la fenêtre. Un problème fut rapidement rencontré lors des premiers essais de surlignage : au raffraîchissement de la fenêtre la ligne précedemment surlignée restait plus ou moins partiellement colorée. Ce fut corrigé en demandant à *JTextPane* de repeindre la zone du précédent surlignage.

7.2 La barre d'outils

Pour les différentes actions possibles sur l'éditeur, le simulateur, ou bien le compilateur (analyseur syntaxique), on les associe à des boutons accessibles de la fenêtre principale. Ces boutons seront placés dans des barres d'outils (classe

JToolBar). Pour contrôler le simulateur on utilise une barre d'outils commune aux deux machines (voir ci-dessous). En utilisant ces composants on a remarqué qu'ils pouvaient être séparés de la fenêtre principale pour devenir une fenêtre indépendante, pour ensuite être rétablis à leur emplacement initial (*dockage/dedockage*).
Comme on peux mettre n'importe quel composant sur un *JToolBar*, on s'est dit qu'il serait intéressant d'ajouter certaines fenêtres en tant que barres d'outils (pour que l'utilisateur puisse choisir entre fenêtres intégrées à la principale, ou fenêtres indépendantes). Malheureusement la classe *JToolBar* est assez basique, et on peut rencontrer les problèmes suivants : fenêtres non redimensionnables, quand elle sont indépendantes, placement sur la fenêtre principale peu aisé, et redockage pas forcément systématique, impossibilité de cacher une fenêtre indépendante. Les seules fenêtres imbriquées dans un *JToolBar* sont alors la fenêtre des messages et l'affichage des composants de la machines (registres ou rubans).

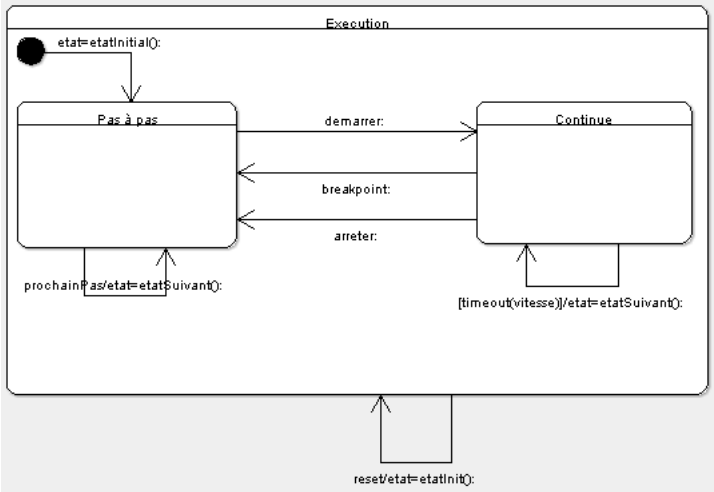
7.3 Barre d'outils de commande commune

Le contrôle de la simulation d'une machine (démarrage et arrêt du calcul, modification de la vitesse) est opéré via une tableau de boutons, dont l'analogie avec les commandes de contrôle usuelles (magnétoscope, chaîne hi-fi, etc...) facilitera la prise en main, bien que certaines divergences de principes existent. Initialement réalisé pour la simulation des machines de Turing, cet ensemble de commandes de contrôles a subi plusieurs modifications avant d'être adopté par les deux interfaces, pour les deux types de machines.

La version définitive comporte cinq boutons :

1. Démarrer/Arrêter : démarre le calcul à la vitesse courante si la machine est à l'arrêt, interrompt le calcul sinon. Le pictogramme sera modifié pour chaque cas d'utilisation.
2. Pas suivan t : effectue le prochain pas de calcul de la machine, si elle n'est pas dans un état terminal. Evidemment, le calcul de la machine sera identique si on procède par succession de pas ou par calcul à une vitesse spécifiée.
3. Reset : *reset*, selon sens commun, la machine chargée. Cependant le mode de fonctionnement de la machine n'est pas altéré lors d'une remise à zéro ; si la machine est en cours de simulation à une vitesse donnée, une pression sur le bouton *reset* reviendre à poursuivre le calcul à la dernière vitesse atteinte à partir de l'état initiale de la machine.
4. Diminuer vitesse : décrémente la vitesse d'une unité si celle-ci est supérieure à un. Permet d'entrer en mode *turbo* (voir plus bas).
5. Augmenter vitesse : incrémente la valeur de la vitesse d'une unité si celle-ci est inférieure à cinq.

L'utilisation d'un code commun permet, sinon de certifier un fonctionnement identique des deux machines, de simplifier l'implémentation, l'intégration étant transparente d'une machine à l'autre grâce à l'utilisation d'une interface implémentée par les deux machines. Nous nous sommes mis d'accord pour l'action à associée à chaque bouton, afin de garantir l'analogie des actions d'une machine à l'autre. Ceci consiste en la définition de méthodes communes, dans la classe implémentant l'interface *InterfaceSimulation* pour chaque type de machine, et respectant le *statechart* suivant :



Du point de vue de l'IHM, on a d'abord isolé le tableau des commandes dans une fenêtre, avant de l'intégrer à la fenêtre principale comportant l'éditeur sous forme d'une barre d'outils amovible (classe *JToolBar*).

Mode *turbo* On peut faire varier la vitesse de simulation de la machine de un à cinq, la rapiditié d'exécution d'une simulation dépendra alors d'une part de la valeur de vitesse sélectionnée, et de l'intervalle de latence lui étant associé, et d'autre part de la machine sur laquelle le programme est lancé (matériel, occupation de la mémoire, du processeur, etc..). La dépendance matérielle sera particulièrement flagrante à la vitesse cinq, celle-ci correspondant à une exécution ininterrompue des instructions. Il est évident que les instructions les plus gourmandes en temps sont les entrées/sorties graphiques, soit la mise à jour des information des registres ou bandes, du graphe, et le surlignage de la ligne courante dans l'éditeur. Le mode dit *turbo* réalise la simulation sans altérer les composants graphiques de visualisation pour réaliser une énorme économie de temps. Ce sera particulièrement utile pour des calculs longs ou infinis, par exemple pour un *busy beaver* à cinq états s'achevant en près de deux millions de transitions. Le mode *turbo* correspond à la vitesse nulle dans les commandes de contrôle.

7.4 Fenêtre des messages

La fenêtre des message permet à l'utilisateur de connaître les différentes informations sur l'état du simulateur : erreur à la compilation, vitesse actuelle, début et fin de calcul. Cette fenêtre possède une simple liste (classe *JList*), le message le plus récent étant le premier de la liste (ajout en tête).

Quatrième partie

Algorithme de placement de sommets dans un graphe orienté.

par Jean-Baptiste Langlois <jean-baptiste.langlois@wanadoo.fr>.

8 Définition du problème

8.1 Présentation

Le problème à résoudre est notoirement NP-complet, au même titre que les problèmes de cliques ou d'autres problèmes d'optimisation de graphes. Etant donc impossible d'obtenir une solution optimale en un temps raisonnable, qui de plus répondrait à des critères arbitraires, on se limitera à rechercher un algorithme d'approximation correct, en faisant reposer le jugement de qualité d'un positionnement sur des critères esthetiques évidents, impliquant les éléments visuels d'un graphe orienté, ses sommets (*vertices*) et arcs (*edges*).

8.2 Exigences

En plus d'avoir à résoudre un problème difficile de placement, il y avait de nombreuses exigences à rajouter au travail initial. En effet, le travail à effectuer pouvait très bien amener l'affichage de plus d'une dizaine de sommets dans le cas de machines complexes. Pour les graphes les plus simples (au plus quatre sommets) la solution est triviale, mais le calcul d'un placement, sinon optimal (la notion de "qualité" étant non définie a priori, le sens d'optimal est assez flou, on parlera de "bon placement" pour un positionnement des états esthétiquement acceptable), du moins d'un bon placement, pour un grand nombre de sommets implique nécessairement une réflexion profonde sur l'algorithme qui repondrait à nos attentes. Autre problème rapidement soulevé : le nombre d'arcs. Au départ, le nombre maximal d'arcs avait été supposé être le double du nombre d'états ce qui déjà rendait la lecture particulièrement difficile dans le cas d'un graphe

non optimisé (croisements d'arcs, chevauchement avec des sommets). Si cette constation pouvait suffire pour la représentation des machines à registres, on était bien loin de ce nombre lors de la représentation de machines de Turing pouvant avoir des sommets avec $n-1$ arcs sortants, avec n le nombre d'états du graphe. Dans un cas similaire, les arcs pouvaient se recouper en un paquet informe dans lequel chaque arc était indiscernable d'un autre, dans le cas d'un placement non optimisé. Dans le sujet du projet à réaliser, deux visions s'opposaient : "la taille du graphe doit être minimal" et "la lecture doit rester facile". Il en a été conclu que le graphe devait avoir ses sommets positionnés de telle façon que la variance de la longueur des arcs devaient être faible devant la moyenne de leur longueur. Il fallait donc faire en sorte que les longueurs des arcs varient le moins possible. Ultime constatation, il fallait choisir quelle type de courbes seraient employés une fois les états bien positionnés. Fallait-il des arcs elliptiques, des lignes courbes (type courbe de Bézier) ou des droites. Bien que comparés aux deux premiers problèmes, celui-ci semble assez facile, c'est celui qui posa le plus de problèmes bien que sa solution s'imposa d'elle même quand un algorithme intéressant de placements des sommets fut trouvé.

Le problème semblait loin d'être résolu. Les différentes conditions additionnées au fait qu'il s'agit là d'un problème NP-complet ne facilitait pas la chose. Après quelques essais pratiques sans aucun fondements théoriques, il apparaissait qu'au delà de quatre sommets, le problème ne pouvait pas se résoudre sans une méthode d'approche rigoureuse, et que les conjectures naïves sur le conditionnement nécessaire à une bonne configuration n'étaient pas forcément justes. Mais après quelques tatonnements, une idée s'imposa.

9 Première idée : la méthode heuristique

9.1 Le principe

Afin d'obtenir un placement optimal d'un graphe, on définit en premier lieu une mesure de la qualité d'une configuration. Cette grandeur que nous avons nommée *graphitance* sera simplement une combinaison linéaire du nombre de croisements d'arcs, de superpositions d'états, de sommets coupés par des arcs, et de la variance de la taille des arcs. Ces critères étant donc ceux retenus pour l'évaluation - subjective - de la qualité d'une configuration. Les coefficients seront attribués d'abord arbitrairement, puis modifiés après les tests. Cet indicateur sera calculé à chaque itération d'une boucle qui s'arrêtera après un nombre d'itérations donnés, à chaque fois générera aléatoirement un placement, et retournera la meilleure configuration obtenue.

Cette heuristique simpliste, après quelques optimisations, livrera des résultats satisfaisants pour des graphes simples, mais pour des graphes plus complexes le caractère purement aléatoire ne permet pas d'obtenir un placement correct systématiquement. Pour cela il serait nécessaire de rendre l'heuristique intelligente,

par exemple en procédant par étapes ; placer les états successivement, conserver de bons placements de sous-ensembles d'états et rechercher le placements des autres sommets, etc... On cours alors le risque d'atteindre un minimum local qui ne correspondra pas à une configuration satisfaisante. Ce problème peut facilement être réglé, mais nous avons trouvé une meilleure solution. Durant nos recherches nous trouvons un article de chercheurs d'ATT ([6]) ayant conçu un algorithme de placement en quatre étapes, dont nous allons fortement nous inspirer pour l'algorithme final.

9.2 L'algorithme

La méthode dite de la graphitance réside d'abord en placement aléatoire, soit à la génération pseudo-aléatoire de coordonnées pour chaque sommet à placer sur la surface donnée. Une fois une configuration obtenue, on calcule sa graphitance, qu'on note comme étant la meilleure. Par la suite, on itère un certain nombre de fois avec un nouveau calcul de graphitance sur des positions aléatoires. A chaque pas, si la nouvelle graphitance est plus faible que la meilleure, la nouvelle graphitance serait marqué comme étant la meilleure (plus la graphitance est faible, meilleur, selon les critères fixés, sera le placement). On a alors l'algorithme suivant :

```
fonction Placement()
meilleure, graphi : entiers
début
 sommets <- placement_aleatoire(Nb_Sommets)
 meilleure <- graphitance(sommets)
 pour i allant de 1 à Max_Iterations
  faire
  nouveau <- placement_aleatoire(Nb_Sommets)
  graphi <- graphitance(nouveau)
  si (graphi<meilleure)
  alors
   meilleure <- graphi
   sommets <- nouveau
  fin si
 fin pour
 retourne sommets
fin.
```

Où *sommets* est un tableau de coordonnées pour chaque état et *Max_Iterations* est un entier arbitrairement initialisé à 1000. La fonction *placement_aleatoire* place aléatoirement les sommets sur le graphe puis la fonction *graphitance* calcule la graphitance de ce placement et renvoie sa valeur. Ensuite on détermine si ce nouveau placement est meilleur ou pas en comparaison à la meilleure graphitance. La graphitance calculée dans la fonction *graphitance* est calculé de la façon suivante :

$$\zeta = \alpha(n_{AxS}) + \beta(|\bar{d} - d_m| + |\bar{d} - d_M|) + \gamma(n_{AxA})$$

Où ζ est la valeur de la graphitance, n_{AxA} le nombre d'arcs croisant des arcs, n_{AxS} le nombre d'arcs croisant des sommets, $\bar{d}$la moyenne de la longueur des arcs, d_m la plus petite longueur d'arc et d_M la plus grande longueur d'arc. α, β et γ représentent des fonctions linéraires respectivement de coefficients a, b et c, qu'on nommera coefficients graphitiques de ζ. Comme on peut s'en douter, plus la priorité donnée à un critère est importante, plus son coefficient graphitique sera important ; dans notre cas, il était de 2 pour b, de *3* pour c et de 20 pour a. Pour la priorité concernant la superposition des sommets, cette considération fit parti du coup de la graphitance durant un temps, mais nous estimâmes bien vite qu'il fallait faire tendre la valeur de la graphitance vers l'infini si une telle chose arrivait. Par la suite, il fut décidé que tout comme la superposition des sommets, un arc croisant un sommet n'était pas une possibilité acceptable et si cela devait se produire, la graphitance tendra alors encore vers l'infini. Dès lors, ce calcul se limita à :

$$\zeta = \beta(|\bar{d} - d_m| + |\bar{d} - d_M|) + \gamma(n_{AxA})$$

On notera l'exemple de trois sommets qui, si positionné en triangle équilatéral donne une graphitance de nulle.

9.3 Application et conclusion

Au départ, le bien-fondé de cette théorie nous semblait évident. D'ailleurs, le problème soulevé semblait tellement complexe que seule l'approche heuristique nous semblait faisable. Après quelques modifications pour lier cette méthode à notre programme, nous étions fin prêt à tenter une application pratique de notre théorie. Malheureusement, le résultat fut loin d'être le résultat escompté. En effet, le premier problème (bien que facilement resolvable) fut que le calcul se faisant aléatoirement, à chaque fois qu'on redimensionnait, le calcul se relançait et on arrivait à un nouveau graphe. Plus gênante était l'apparence du graphe. Très peu d'entre eux étaient rééllement esthétiquement valables. Cela s'explique principalement par le fait que nous étions plus axé sur la formule de la graphitance que sur la partie heuristique de la fonction. Pour que l'application stricte et pure de la graphitance, il faudrait deux modifications majeures :

- Tout d'abord, quand à notre nombre d'itérations (arbitrairement 1000), il est bien trop faible. Notre fenêtre d'affichage du graphe fait 600 pixels sur 600 pixels, soit 360 000 positions possibles pour chaque sommets. Dans le cas d'un graphe à n sommets, on doit donc avoir (360 000)n itérations. Dans le cas d'un graphe à 5 sommets, on arrive à la bagatelle de 6,04 x 1047 itérations ; on est là bien loin de nos négligeables 1000 itérations.
- Supposons que durant 1000 itérations, les sommets sont affectés à la même place (probabilité faible mais existante) ; ces itérations auront été inutiles. Il nous faudrait donc noter, pour chaque itération, les emplacements de chaque sommet et à chaque nouveau placement, comparer pour savoir si ce placement

n'a jamais été essayé. Pour des raisons évidentes de temps de calcul et de place en mémoire, aucune de ces modifications n'étaient envisageables. Nous avons décidé d'abandonner cette façon de faire et de trouver un autre moyen d'accéder à un algorithme valable de placement des sommets. La vérité est donc ailleurs.

L'erreur qui a été faite avec cette méthode, a été d'itérer. Ainsi, la méthode la plus "logique" pour ce type de problèmes n'est pas la méthode cette itération "bête". Nous avons tenté une méthode algorithmique dont l'idée première était de détecter des sous-ensembles de V (pour un graphe $G = (V, E)$), pour former des sous-graphes, à partir de composantes fortement connexes du graphe, mais cela n'a que peu fonctionné. C'est alors que je mis la main sur un document émis par les laboratoires d'AT&T [6] qui offre une approche totalement inédite par rapport à nos premières expériences .

10 Deuxième idée : la méthode algorithmique

10.1 Principe et limites

Le document auquel nous nous référons, provenant des laboratoires d'AT&T, propose un algorithme pour obtenir un placement des sommets relativement esthétique. Il propose notamment une approche radicalement différente de celle que nous avons pu avoir. Il est basé sur deux calculs, l'ordre et le rang, chacun censé déterminer une coordonnée. L'ordre pour les abscisses et le rang pour l'ordonnée. Puis, on affecte les coordonnées données à partir du rang et de l'ordre. Le rang est déterminé par une initialisation des sommets en les plaçant tous dans une file, les extracant un par un selon leur degré entrant en leur affectant un rang donné. Après ceci, on minore la différence entre chaque rang, puis on ajoute une valeur correspondant à leur position dans le graphe, donnée par un parcours en largeur du graphe. Puis on étudie les sommets qui n'ont pas de prédécesseurs. Pour l'ordre, il est initialisé en effectuant un parcours en profondeur, puis par une méthode dite de la médiane, on itère afin de trouver l'ordre avec le croissement d'arcs minimal. On peut éventuellement échanger deux arcs entre eux afin d'obtenir un résultat optimal. L'ultime action de cette méthode consiste à prendre le rang pour ordonnée et à travailler sur l'ordre (les futures abscisses) de façon à obtenir une différence de longueur des arcs faible devant leur moyenne. Dès lors, il ne reste plus qu'à tracer. L'intérêt primordial de cette méthode se situe dans le fait qu'après avoir initialisé rang et ordre, on est certain d'avoir un placement déjà optimal.

De plus, le calcul ne faisant pas intervenir de mesures arbitraires, on est certain d'obtenir le même placement à chaque fois que l'on charge ce graphe. Toutefois, l'application stricte de cette méthode a rendu des résultats pour le moins étonnant qui nécessitaient une retouche. En effet, de nombreuses différences subsistent entre leur cas et le nôtre pour que leur méthode fut appliqué telle quelle.

Par exemple, on étudie les sommets qui n'ont pas de prédécesseurs si on suit bien leur méthode, mais cela ne se peut pas dans notre cas de représentations graphiques de machines de Turing ou machines à registre car le seul sommet (si la machine est bien formée) pouvant ne pas avoir de successeur est l'état initial. De même, j'ai beaucoup de mal à me figurer comment on pourrait obtenir le nombre de croisements du graphe juste en connaissant l'ordre. C'est pour ces raisons que l'algorithme donné ne semblait pas correspondre à notre cas, à savoir l'affichage d'un graphe optimal pour nos deux types de machines. Toutefois, certaines idées étaient suffisamment intéressantes pour que l'on décide de s'inspirer de ce document pour la conception de notre propre algorithme.

10.2 L'algorithme

Comme cela a été dit ci-dessus, le principal intérêt de l'algorithme des laboratoires de AT&T étaient l'assurance d'avoir un graphe quasi optimal dès l'affectation du rang et de l'ordre ; par quasi optimal, il faut entendre "avec un nombre minimal de croisements d'arcs" (bien entendu, le croisement d'un état par un arc n'est même pas imaginable). Il fut donc décidé d'utiliser les notions de rang et d'ordre pour trouver un algorithme crédible. Toutefois, mes priorités étant différentes de celles du document d'AT&T (relativement formel et rigoureux), il était prévu que l'algorithme final n'aurait qu'une vague inspiration de la part du document.

Mes principales considérations quant à l'écriture d'un algorithme étaient d'ordre esthétique. En cela, après de nombreux tests, on détermina que le nombre de croisements d'arcs influaient plus que la longueur des arcs dans la lecture du graphe. Ainsi, bien que la taille des arcs fut un des critères originels, elle fut laissée de coté pour l'instant pour concentrer l'algorithme sur une minimalisation du nombre de croissements des arcs entre eux.

Ensuite, alors que l'algorithme initial se concentrait principalement sur l'ordre pour déterminer les positions des sommets privilégiant les abscisses aux ordonnées, devant selon eux, trouver leur place aisément une fois l'abscisse des différents sommets définis, mon idée portait principalement sur la considération que le rang était la notion était la plus importante.

Enfin, alors que les valeurs initiales du rang et de l'ordre étaient soumises à de grands changements par la suite chez AT&T, les résultats que j'obtins pour les graphes étudiés montraient que les résultats étaient assez convaincant pour être conservés. Ainsi, durant une bonne partie de l'écriture de cet algorithme, ces modifications ne furent pas prises en compte, considérant que le peu de modifications en résultant coûterait bien trop cher en complexité. Pour le premier jet, il fut décidé de réécrire deux fonctions, chacune établissant une valeur de rang et d'ordre pour chaque sommet, et de n'utiliser que ces fonctions pour le placement des sommets. La théorie suivant laquelle on a établi ces deux notions sont décrites ci-dessous :

– Pour le rang, il semblait évident qu'il soit défini grâce aux degrés entrants et sortants de chaque sommet. En fait, seuls les degrés permettent de déterminer l'importance des différents sommets. Afin d'éviter que tous les sommets de mêmes degrés ait le même rang, il fallait ajouter une modification à l'expression qui fut trouvée à travers le rappel du rang du prédécesseur. De même, les successeurs d'un sommet ne devaient pas avoir le même rang pour des raisons évidentes. De même, pour rester cohérents dans le placement des états, un parcours en largeur fut choisi. Dès lors, l'algorithme suivant devient une évidence :

```
procedure init_rangs()
F : File d'entiers
début
 rang(etat_initial) <- degree_sortant(etat_initial)
 visite(etat_initial) <- vrai
 enfiler(F, etat_initial)
 tant que ¬(tous les états ont été visités)
  sommet <- défiler(F)
  visite(sommet) <- vrai
  pour i de 0 à degree_sortant(sommet)
   rang(Successeur(sommet,i))
              <- rang(sommet)*(i+1)/degree_sortant(sommet)
   enfiler(F,Successeur(sommet,i))
  fin pour
 fin tant que
fin.
```

– L'ordre, quant à lui, est simplement défini par un parcours préfixe permettant juste d'avoir un minimum d'arcs se croisant vu que cette action ne sert qu'à définir les abscisses des sommets. Il faut donc juste que les valeurs des abscisses soient différentes selon leur hauteur par rapport à l'état initial.

```
procedure init_ordre(sommet, valeur : entier)
début
 si ordre(sommet) = 0
 alors (* ordre non défini *)
  ordre(sommet) <- valeur
  valeur <- valeur + 1
  pour chaque successeur j de sommet
   init_ordre(j,valeur)
  fin pour
 fin si
fin.
```

En initialisant *init_ ordre* de telle façon que sommet soit l'état initial avec valeur à 0 (mais on peut bien sûr choisir une autre valeur de départ, car celle-ci peut très bien être arbitraire).

10.3 Application

Une fois, l'ordre déterminé, il ne restait plus qu'à admirer. Avec la seule application du seul rang et ordre, il apparaissait que nous pouvions obtenir un graphe qui, bien qu'il ne soit pas optimal, n'en restait pas moins cohérent. Pas de superposition de sommets et le nombre de croisements d'arcs était faible pour ne pas dire inexistant. Ainsi durant une longue partie de mon travail, je ne modifiai rien d'autre que l'algorithme suivant :

```
procedure placement()
début
 init_rangs()
 init_ordre(0,1)
 normalisation()
 defCordonnees()
fin
```

Ici, la procédure *normalisation()* est une fonction destinée à centrer mon placement. Ainsi, si les rangs ont pour valeur 3, 4, et 6, le graphe associé sera décalé vers le haut ou le bas de l'écran (rappelons que le rang défini l'ordonnée). Il serait dès lors plus intéressant de les transcrire vers 0,1, et 3, ce qui centrerait le graphe, tout en conservant les proportions. La procédure *defCordonnees()* est, quant à elle, relativement simple. Elle convertit juste les rangs et ordres des différents sommets en valeurs de pixels utilisables pour l'affichage d'un graphe à l'écran. Elle se définit (par exemple, pour l'ordonnée) par :

$y_k = TailleEcran \times (\frac{Rank_k}{max(rank_0,rank_1...,rank_j}) \, , \, avec 0 \leq k \leq j$

Le résultat est impressionnant. Le seul défaut subsistant se situe avec les graphes à beaucoup d'états (plus de dix). En effet, l'occupation de la fenêtre de visualisation commence à être importante et il arrive que des sommets aient le même rang et le même ordre : ils sont donc virtuellement superposés. Pour celà un déplacement des sommets concernés est nécessaires ce qui appelle à une "correction" du graphe. Malheureusement, je vis rapidement que cela pouvait créer plus de croisements d'arcs qu'ils n'y en avait. C'est là, où la transposition vient faire son apparition ; derrière ce nom se cache un principe somme toute élémentaire. Il s'agit de déplacer les sommets afin de voir quel placement amène le moins de croisements d'arcs.

```
procedure transposition()
début
 ∀i ∈ S
  ∀j ∈ S \ j > i
   si croisement(i,j)>croisement(j,i)
   alors
    echanger(i,j)
   finsi
 fin
```

L'affichage du graphe en est donc amélioré. La grande question qui demeurait concernait la forme des arcs de transitions à savoir, je le rappelle, choisir entre des courbes et des droites. Or, comme cela avait été évoqué, la réponse vient à présent d'elle-même ; la théorie que j'utilise et qui semble correspondre à nos attentes, minimise le nombre de croisements en supposant l'accès aux autres sommets de façon rectilignes. En effet, dans le cas où on utiliserait des courbes, comment pourrait-on dire ou non, qu'aucun sommet n'est coupé par un arc. Les tests confirment cette idée, où des sommets sont "contournés" par une opération du St-Esprit (ou assimilé) dans le graphe. La réponse est donc évidente ; l'utilisation de lignes courbes seraient trop chaotiques dans l'optique que l'on a suivi depuis la recherche d'un algorithme de placement optimal.

En conclusion, après avoir tenté une méthode naïve, qui bien qu'intéressante sur le papier ne correspondait pas à une application réelle dans notre cas présent, une réflexion plus intense et une lecture plus professionnelle nous permit de dégager un procédé plus profond et reposant sur des bases bien plus théorisés. Bien que les priorités des deux méthodes ne soient pas du tout les mêmes, on ne peut constaté que la méthode algorithmique renvoie des résultats bien meilleur au niveau de l'aspect que la méthode heuristique. On peut toutefois déplore que cet façon de procéder n'est pas parfaite. En effet, la superposition des sommets est monnaie courante et la difficulté de rendre cet algorithme applicable aux graphes cycliques fut un de mes principaux problèmes. Certes, le résultat en aurait été bien différent si l'on avait suivi les spécifications exactes d'AT&T, mais cela suffit largement dans notre cas estudiantin. Pour finir, on peut dire que bien que la méthode heuristique n'est pas respecté ses promesses, tout n'est pas perdue car l'idée de la graphitance tient la route et après avoir trouvé une méthode de calcul valable via l'algorithmique, je continue d'utiliser la graphitance pour tester l'optimisation de mon graphe final.

11 Visualisation

La façon dont a été trouvé l'algorithme de placement du graphe fut expliqué ci-dessus. Dès lors qu'une simulation acceptable avait été trouvé en comparant les coordonnées des points qui furent reportés sur GIMP pour tester l'esthétisme du graphe, on a pu dire que, pour le placement du graphe, le plus gros avait été fait. En effet, trouver la position la meilleure de sommets dans un graphe est un problème NP-Complet, étant à ce jour indécidable et donc sans réélle réponse. Seules quelques approximations sont disponibles et parmi elles, l'algorithme exposé ici. Malheureusement, bien que le plus dur soit passé, il fallait reporter le placement des sommets graphiquement et y adjoindre des arcs. Si le principe d'exposer la façon de procéder pour un problème aussi simple en apparence peut prêter à sourire, on verra qu'il n'est pas aussi simple et que certains problèmes ont été soulevés et qui méritent d'être explicités ici. Sera présenté ici, la façon de procéder pour afficher tout d'abord les sommets puis les arcs et enfin les transitions.

11.1 Placement des sommets

Le plus gros problème soulevé par les sommets est la façon de les placer. Après tout, une fois placé, ce n'est rien qu'un ensemble de cercles relié entre eux par des droites (les arcs). En effet, le problème se révela trivial en créant simplement des cercles dont le centre correspond aux coordonnées données par l'algorithme de placement. Il suffit par des tests à raffiner pour déterminer la taille du rayon ainsi que de la police affichant le nom des sommets (0 ou q0 ou encore R_1^+). L'élégance nous poussa même à modifier deux choses pour des raisons purement esthétique ; Préférer l'affichage de R_1^+ à $R1+$ et permettre la modification du rayon des sommets ainsi que de la police des états selon la taille de la fenêtre de visualisation. En réalité, le seul petit problème qui se posa se situe au niveau des machines à registres ; les états sont la plupart du temps constitués d'une seule lettre à l'exception de Début et Fin. Pour corriger ces cas-là, l'utilisation d'une ellipse s'imposa. De même rayon horizontal et vertical dans le cas d'un état "classique", d'un rayon horizontal augmenté de la moitié de la largeur du mot à afficher dans le cas d'un état initial ou final d'une machine à registres.

Comme on peut le voir, le placement des sommets fut somme toute assez simple, comme il avait été prévu. Après avoir déterminé les coordonnées des différents sommets selon la taille de la fenêtre, il ne restait plus qu'à afin des cercles aux emplacements désirés. Ce placement n'était toutefois qu'une simple vision par rapport au travail à effectuer pour le placement des arcs qui allaient me renvoyer dans mes cours de trigonométrie longtemps négligés.

11.2 Placement des arcs

Un arc est la flèche représentant un déplacement d'un état vers un autre. En cela, elle est constitué de deux parties : une ligne (droite ou courbe) et un triangle à une extrémité représentant la direction prise par la flèche. La première idée utilisée pour représenter un arc entre deux sommets fut d'utiliser une droite reliant une extrémité du sommet à une autre. Le principal avantage de cette méthode était la facilité évidente qu'il en résultait pour positionner le triangle à l'extrémité de la flèche. Facile : il suffisait de prendre l'extrémité de la droite pour avoir les coordonnées du triangle. Quant au choix d'une droite pour relier deux sommets, l'explication tient au caractère "aléatoire" que peut prendre une ligne courbe, ne sachant pas si la ligne passe à travers certains sommets. Le problème de la méthode exposée ci-dessus se situe dans le fait de bien situer les flèches par rapport à leur sommet. Bien que les arcs soient dessinés à partir de leur sommet d'origine, la lourdeur des calculs complique la relecture de l'algorithme. En effet, pour calculer à quelle position la flèche débutait, il fallait déterminer l'angle créé entre les deux sommets et donc recourrir à des fonctions avancées de trigonométrie telle l'arctangente. De plus, en supposant que tout se passe bien, que se passeraient-il si, du sommet *1* on a un arc allant au sommet *2* et vice-versa ? On aurait alors une seule droite, ayant un triangle à chaque extrémité :

les deux flèches seraient superposées Pour éviter cela, la technique d'approche a été modifiée ; au lieu de tracer les flèches à partir du bord des sommets, il a été décidé que le centre du cercle constituant le sommet, serait une extrémité de l'arc. Le principal serait alors de dessiner le sommet après les arcs de façon à ne pas voir les arcs à l'intérieur du sommet. Bien entendu, cela simplifie grandement le calcul des extrémités des arcs mais, *a contrario*, complique le calcul des coordonnées des triangles qui ne se trouvent alors plus à l'extrémité de la droite mais sur le bord du sommet. L'équation suivie pour le calcul de l'angle fait par le triangle est le suivant :

$angle = tan^{-1}(\frac{y_2 - y_1}{x_2 - x_1})$

où les couples (x_1, y_1) et (x_2, y_2) représentent les coordonnées des sommets alors que la position initiale du triangle en découle :

$x_{triangle} = R_{sommet} \times cos(tan^{-1}(\frac{y_2 - y_1}{x_2 - x_1}))$

$y_{triangle} = R_{sommet} \times sin(tan^{-1}(\frac{y_2 - y_1}{x_2 - x_1}))$

Par contre, par une simple modification des coordonnées dans un sens (arbitrairement choisi), on obtient un visuel de bien meilleure facture pour l'affichage.

Un autre problème soulevé fut la mise en application des arcs provenant d'un sommet de allant vers le même sommet, nommés *self-vertex edges*. En effet, il était là impossible de tracer juste une droite et il fut tenté d'appliquer le tracé d'une ligne courbe mais après de multiples tentatives, on retint finalement l'utilisation d'une ellipse, plus simple à mettre en application et finalement bénéficiant d'un meilleur rendu. Par contre, la création de l'interface en fut bouleversé. En effet, l'utilisation d'ellipses ne permettait pas l'utilisation des méthodes déjà créés qui passaient par le centre du sommet concernés. Il fallu donc créer des méthodes spécialement étudiés pour les self-vertex edges et ainsi, comme ils ne pouvaient pas être invoqués de façon similaire aux autres, modifier la façon dont l'interface concevait ces arcs. Ainsi, une instance de type booléenne devait indiquer si l'arc était un self-vertex ; si ce n'était pas le cas, on devait traiter l'arc comme à l'accoutumée. Cela nous permit de faire cohabiter des droites et des ellipses pour le tracé des arcs.

Après avoir créé un algorithme crédible qui permet de tracer les sommets ainsi que les arcs tout en évitant toute superposition, la majeure partie du travail était effectué ; bien que les tracés des arcs ne soient pas toujours particulièrement esthétiques, l'ensemble restait acceptable et il ne restait plus qu'à travailler sur l'affichage des transitions entre deux états (sommets).

11.3 Placement des transitions

Comme évoqué plus haut, le tracé des transitions, sans être particulièrement compliqué, est pourtant fondamental. La visualisation de ces entrées permet d'afficher les transitions entre deux états (sommets). Pour ce faire, j'ai réfléchi

à la façon selon laquelle un homme écrirait ces transitions, mais ce ne fut pas la meilleure idée que j'ai eue. En effet, la plupart des gens place les transitions là où il y a de la place ce qui n'est pas facile à transcrire par un algorithme. Je songeais alors au plus simple : au milieu entre les deux sommets. De façon plus formelle, il suffisait de déterminer l'équation de la droite d'un sommet à un autre, puis en calculant l'abscisse centrale, en déterminer l'ordonnée du point médian.

Doté de cette équation, des tests s'en suivirent mais une importante correction se fit dans le fait où avec une telle abscisse, l'étiquette des transitions "transperçait" l'arc. Pour éviter cela, une correction de l'abscisse fut entreprise, ce qui permit de déplacer l'abscisse vers la gauche d'un nombre calculé à partir du nombre de lettres du test de transitions multipliés par la largeur des lettres. Malheureusement, bien que ce calcul autorisait une visualisation intéressante, cela devenait vite illisible dès que le nombre de sommets augmentait. Pour éviter cela, j'ai modifié la classe gérant la transition pour permettre le déplacement du texte à droite de l'arc, si on a déjà un arc allant dans le sens contraire entre les deux sommets. Cela permet une visualisation de meilleure facture pour un grand nombre de sommets. Toutefois, ce résultat est loin d'être optimal ; en effet, quand on a un grand nombre d'états (à partir d'une vingtaine) et plus particulièrement avec des machines de Turing multi-bandes, les textes de transition finissent toutefois par se superposer. Cela renvoie une impression gênante au niveau de l'esthétisme du graphe mais ne change rien à la lecture car ces textes de transition sont accessibles via la fenêtre d'édition du graphe.

Après avoir écrit l'algorithme de placement des sommets sur une fenêtre, le plus dur était fait. Il restait toutefois quelques ajouts à faire au niveau de l'affichage du rendu des données. Parfois trivial, comme ce fut le cas avec le tracé des sommets, qui n'étaient rien de plus que des ellipses, parfois réellement compliqué, comme le cas des *self-vertex edges*, où le successeur d'un sommet est lui-même, le résultat final, étant au-delà de mes espérances, valu bien quelques nuits blanches à régler les bugs ; d'un point de vue général, le résultat est largement probant, avec un affichage convaincant des sommets et des arcs. Seul le tracé des transitions n'était pas parfait au delà d'un certain nombre de sommets. En effet, le principal problème de ce type de projet est le temps et il aurait fallu un temps considérable pour trouver un moyen de ne pas faire se chevaucher les transitions ; or, dans un système où le temps est le principal ennemi, on doit parer au plus urgent. Ici, c'est un pari gagné, car l'algorithme de placements des sommets selon leurs successeurs est impressionnant alors que l'affichage des transitions, somme toute accessoire, n'est pas le plus important étant donné que le texte affiché apparaît également dans les fenêtres d'édition des machines de Turing et machine à registres. Ne nous y trompons pas ! L'esthétisme, valeur primordiale dans le tracé d'un graphe valable, n'est affecté en rien par l'affichage ou non des transitions.

Cinquième partie

Conclusion générale

A ce jour nous livrons un programme fonctionnel et dans sa version quasi-définitive, fruit de près de trois mois de travail régulier, le "cahier des charges" et le délai imparti étant respectés. Nos principales réussites ont été de commencer le travail relativement tôt, d'identifier les difficultés chronophages, tout en anticipant d'éventuels imprévus. Cependant, dans l'absolu, et *a fortiori* dans un cadre professionel, beaucoup de lacunes subsitent quant à l'organisation et au génie logiciel, à notre décharge on pourra remarquer que ce projet constituait notre première expérience sérieuse en Java, langage auquels nous étions certes familiers, mais la maîtrise ne s'acquiert que par l'expérience.

Nous avons découvert les joies du travail en équipe, notamment réunions et *brainstormings* ; nous exprimions nos points de vues et tentions d'en limiter les divergences, des états des lieux étaient régulièrement effectués. Les questionnemenst particulièrement problèmatiques, ou concernant directement plusieurs développeurs, étaient soumis à l'ensemble de l'équipe. Un problème récurrent, pas forcément trivial, fut de fixer les dates et lieux de réunions convenant à tout le monde.

De possibles améliorations pourraient concerner l'interface graphique, notamment au niveau des localisation et dimensions des fenêtres, pour éventuellement les adaptés à tout type de résolution d'écran, le minimum recommandé étant pour l'instant une taille 1024*768 pixels. Nous avons songé à adapter la taille des composants graphiques en fonction des dimensions de l'écran, mais le fait de devoir retailler toutes les icônes, et fontes nous a vite fait abandonner. Le placement du graphe n'est pas non plus parfait, étant donné qu'il répond à des critères arbitraires et qu'une étude sérieuse serait longue, mais est suffisant pour des graphes de taille moyenne.
Pour de futur développeurs, ou stagiaires, la création d'une interface spécifique pour la création d'une machine serait appréciable, par exemple en créant graphiquement la machine par interactions entre la souris et une fenêtre graphique.

Ce projet fut l'occasion de s'interresser plus en détail au notions étudiées, et d'acquérir une expérience sérieuse de développement Java avec interface graphique en utilisant les librairires *awt* et *Swing*, dont nous avons pu apprécier la relative simplicité et l'étendue des possibilités offertes. L'utilisation de Java, par sa souplesse, la convivialité de sa documentation en ligne, et surtout sa portabilité a été très appréciable. Nous travaillions en effet sur plusieurs distributions Linux, MacOSX, et même Windows XP, les différences se limitant au *look and feel* des fenêtres.

Enfin, nous espèrons que ce programme ne tombera pas totalement aux oubliettes, et qu'il puisse être utilisé par d'éventuels étudiants ou enseignants, pour une visualisation moins austère et une simulation plus rapide que sur papier.

Le programme est en ligne à l'URL suivante : *http ://jpnossa.free.fr*. L'accès restreint sera désactivé dès la fin des soutenances.

Crédits

Florent a utilisé Eclipse 2.1.2, RCS, ArgoUML pour les modélisation UML, CygWin, le JDK 1.3, et après nos menaces de mort répétées a enfin adopté le JDK 1.4.2, JP remercie IBM pour le compilateur jikes 1.22, qui lui aura fait gagné un temps précieux, Duron 700 oblige. Jalopy a été utilisé pour la mise en forme des sources, JEdit 4.2 pour leur édition, et accessoirement GNU Emacs 21.3.2. Pour le rapport OpenOffice.org 1.1.0 et Lyx 1.3.3 ont été utilisés, ce-dernier adopté sans hésitation pour le rapport final, la conversion en pdf est réalisé par pdflatex 3.14159-1.10b, et on visualisera le fichier avec KGhostview ou Adobe AcrobatReader de préférence, xpdf et gnomeview sont à éviter. Nous remercions également : Carlitus pour son set d'icônes Noia (http ://www.carlitus.net/), Mr Rey pour ses réponses à nos mails, Le Gimp, l'inventeur des messageries instantanées, l'hébergeur Free et son serveur ftp capricieux, nos clés USB pour leurs fidèles et loyaux services, Steve Jobs et Linus Torvalds, Kraftwerk et les thés Twinings pour la survie en sommeil réduit, la doc Java de Sun, Google, et Colette !

Références

[1] A.M. Turing, *On computable numbers, with an application to the Entscheidungsproblem* (1936)

[2] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to automata theory, languages, and computation*

[3] H. Rogers, *Theory of recursive functions and effective computability* (1987)

[4] M.D.Davis, R.Sigal, E.J. Weyuker, *Computability, complexity and languages* (second edition, 1983)

[5] J.F. Rey, *Calculabilité, complexité et approximation*, Vuibert (2003)

[6] E.R.Gansner, E.Koutsofios, S.C.North, K.P.Vo (AT&T Bell Laboratories), *A Technique for Drawing Directed Graphs,* http ://www.graphviz.org/Documentation/TSE93.pdf

[7] D.Flanagan, *Java in a nutshell*, O'Reilly

[8] P-Y.Saumont, A.Mirecourt, *Le guide du développeur Java 2*, Eyrolles

[9] C.S.Hortsmann, G.Cornell, *Au coeur de Java 2,* Campus Press