Rapport de TIPE

Images et Géométrie

Mathieu GOUTELLE (goutelle@easynet.fr)
Année 1997-98

Texte mis en forme avec LaTeX 2ε

Table des matières

1 Introduction **2**
- 1 Présentation . . . 2
- 2 Interface graphique et informatique . . . 2
  - 2.1 La bureautique . . . 2
  - 2.2 La programmation . . . 3
- 3 Objectifs . . . 3

2 Les arbres **4**
- 1 Dessin . . . 4
- 2 Création par parcours dans l'arbre . . . 4
- 3 Création d'un arbre par les graphes . . . 5
  - 3.1 Equivalence des définitions d'arbres . . . 5
  - 3.2 Test de connexité . . . 6
  - 3.3 Définition de l'arbre à partir du graphe . . . 7

3 Les automates **9**
- 1 Le tracé des transitions . . . 9
  - 1.1 Introduction . . . 9
  - 1.2 Courbes de BEZIER . . . 10
  - 1.3 Programmation . . . 12
- 2 Création graphique d'automate . . . 13
  - 2.1 Généralités . . . 13
  - 2.2 Création d'un automate « simple » . . . 13
  - 2.3 Création d'un automate avec transitions instantanées . . . 14
- 3 Dessin d'automate . . . 14
  - 3.1 Généralités . . . 14
  - 3.2 Définition des « noyaux » d'un automate . . . 15
  - 3.3 Calcul de la position des états et des transitions . . . 15

4 Conclusion **20**
- 1 Généralités . . . 20
- 2 Améliorations possibles . . . 20

Liste des figures **21**

Liste des programmes **22**

Bibliographie **23**

Chapitre 1

Introduction

1 Présentation

Le thème qui nous a été proposé cette année pour l'épreuve de TIPE en informatique est « Images et Géométrie ». Vaste sujet s'il en est, nous allons nous intéresser plus particulièrement ici à l'utilisation du graphisme dans l'amélioration de l'interface homme/machine.

Depuis les débuts de l'informatique, les relations entre l'utilisateur et son outil ont beaucoup compté et se sont améliorées par la force des choses, la puissance des machines se développant. Les ingrates lignes de commande ont fait place aux interfaces graphiques, plus faciles d'emploi et d'accès parce que plus intuitives. Le développement de l'informatique a demandé aux programmeurs beaucoup d'efforts d'imagination et d'adaptation pour permettre à tous d'accéder à cette science nouvelle qui allait influencer la vie de tous les jours.

Plus récemment, on a voulu faciliter le travail des programmeurs eux-mêmes en introduisant le graphisme dans les interfaces de programmation. De façon plus modeste ici, nous allons essayer de permettre à un utilisateur de CAML de construire de façon graphique des arbres et des automates.

Remarque : seules seront présentées dans ce document les fonctions ayant une grande importance et une approche facile. Ainsi, tous les utilitaires, fonctions de base ou procédures trop longues en sont exclus. Pour les détails de programmation, il sera nécessaire de consulter les descriptions du langage CAML, [1] et [2].

2 Interface graphique et informatique

2.1 La bureautique

Pour que l'informatique soit utilisée par tous et de façon « productive », il fallait que l'utilisation des logiciels devienne intuitive. On a donc banni tous les logiciels non WYSIWYG [1], c'est-à-dire ceux qui utilisaient des interfaces en mode texte ou en lignes de commande.

Tous les traitements de texte et autres tableurs ont profité du développement des systèmes d'exploitation graphiques pour devenir les logiciels que nous connaissons aujourd'hui. Seuls certains résistent, parce que souvent réservés aux informaticiens chevronnés, comme le processeur de texte TEX qui reste plus performant que les logiciels possédant une interface.

2.2 La programmation

Là aussi, pour que tout le monde ait accès à l'informatique et donc à la programmation, l'*Assembleur* est devenu un langage ésotérique auquel on a préféré des langages comme *Turbo*

1. « What you see is what you get » : Ce que vous voyez est ce qui sera imprimé

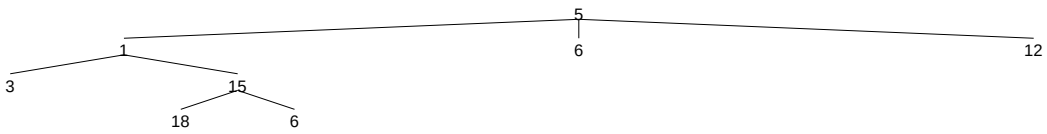
Pascal, *Visual Basic* ou *Delphi* qui bénéficient d'un véritable environnement.

Plus récemment, on a dépassé ce stade avec *Klik & Play*, *Click & Create* ou encore *The Games Factory* [3]. On aboutit alors à de la programmation visuelle puisqu'il suffit de définir des objets auxquels on associe des actions et des événements.

3 Objectifs

Lors des premiers pas avec le programme de l'option Informatique en Mathématiques Spéciales, on appréhende plus facilement les notions abordées avec un dessin. Il en va ainsi des arbres et des automates.

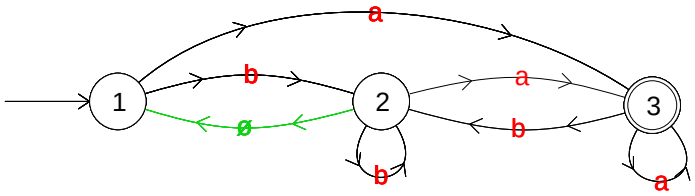
Un arbre se conçoit facilement lorsqu'on le dessine (cf. FIGURE 1.1). On peut alors élaborer des programmes à partir de ce dessin, essayer de tracer une fonction. Mais que faire lorsque l'on veut tester cette fonction ? Essayer de créer un arbre à la main, c'est s'exposer à toute sorte d'oubli de crochet, parenthèse, point virgule et autre virgule. Autrement dit, tout exemple complexe est à exclure. Il serait alors avantageux de pouvoir dessiner l'arbre et qu'un utilitaire confectionne l'arbre comme CAML pourrait le comprendre. Et inversement, lorsqu'une fonction donne comme résultat un arbre, on pourrait alors le dessiner.




```
noeud (5,
      [noeud(1,[feuille 3;noeud(15,[feuille 18;feuille 6])]);
       feuille 6;feuille 12]);;
```

FIGURE 1.1 : *Exemple d'arbre*

Quant aux automates (cf. FIGURE 1.2), ils s'élaborent souvent par un dessin, qui permet à l'explication d'être à la fois concise et compréhensible. D'ailleurs, l'automate reste souvent au stade du dessin, car de toute façon, sa programmation à partir d'un graphe est aisée et sa donnée n'est pas d'un grand secours. On pourrait soit définir un automate en le dessinant et inversement, dessiner un automate donné.




```
{ordre = 3; initial = [1]; final = [3]; instantanée = [2, 1];
 transition =
  [(3, 'a'), 3; (2, 'b'), 2; (3, 'b'), 2; (2, 'a'), 3; (1, 'b'), 2;
   (1, 'a'), 3]};;
```

FIGURE 1.2 : *Exemple d'automate*

Chapitre 2

Les arbres

1 Dessin

Le dessin d'un arbre ne pose guère de problème. La racine doit être tracée en haut au milieu de l'écran et ses fils sont dessinés plus bas dans le reste de l'écran divisé selon le nombre de fils. Il suffit alors d'appliquer cette méthode récursivement.

Programme 2.1 : Le type d'arbre

```
type ('a,'b) arbre =
    |nil
    |feuille of 'a
    |noeud of 'b*(('a,'b) arbre list)
;;
```

On utilise le type d'arbre décrit par le Programme 2.1. La fonction de dessin (cf. Programme 2.2) comporte deux fonctions récursives imbriqués. Les fonctions `affichea` et `afficheb` servent à dessiner les valeurs des feuilles, respectivement des noeuds, de l'arbre. Les paramètres `x, y, w` correspondent aux coordonnées du coin supérieur gauche et à la largeur de l'écran virtuel où l'on trace le noeud courant. L'appel récursif divise l'écran virtuel entre les fils du noeud courant.

2 Création par parcours dans l'arbre

Pour créer ou modifier un arbre, on peut essayer de se « promener » dans l'arbre pour y faire toutes les modifications possibles :

- modifier la valeur d'un noeud ou d'une feuille
- supprimer ou ajouter une branche
- faire du Copier/Coller entre les différentes parties de l'arbre

Au bout du compte, l'utilisateur peut modifier au fur et à mesure son oeuvre et la voir évoluer. Cependant, cette méthode présente deux inconvénients. D'une part, il faut à tout moment connaître le parcours dans l'arbre et ceci est problématique lors des déplacements latéraux ou vers le haut par exemple. De plus, cela nécessite à chaque mouvement de repartir de la racine de l'arbre pour calculer par exemple les coordonnées du point courant.

D'autre part, le type d'arbre (cf. Programme 2.1) facilite certes l'ajout et le retrait d'une branche de l'arbre par le choix des listes, mais il ne respecte pas totalement la philosophie des arbres. En effet, un des intérêts des arbres est le parcours très rapide dans l'arbre alors que les listes ont un accès séquentiel. Ces deux inconvénients incitent à trouver une autre méthode, développée dans le paragraphe suivant.

Programme 2.2 : Le dessin d'arbre

```
let pasy=30;;

let rec dessineliste liste_ab x y w affichea afficheb =
     match liste_ab with
          |[]    -> ()
          |t::q -> (rdessine t x y w affichea afficheb;
                    dessineliste q (x+w) y w affichea afficheb)
and rdessine arbre x y w affichea afficheb  = match arbre with
     |nil -> ()
     |(feuille(val)) -> affichea (x+w/2) y val
     |(noeud(_,[]))   -> raise (Erreur "Arbre malforme")
     |(noeud(val,l)) -> let longl=list_length l in let new_w=w/longl in
          begin
          afficheb (x+w/2) y val;
          for t=0 to (longl-1) do
               moveto (x+w/2) (y-14);
               lineto (x+new_w/2+t*new_w) (y-pasy)
          done;
          dessineliste l x (y-pasy) (new_w) affichea afficheb
          end
;;

let dessine arbre affichea afficheb couleur =
     set_color couleur;
     rdessine arbre 0 (size_y()) (size_x()) affichea afficheb
;;
```

3 Création d'un arbre par les graphes

3.1 Equivalence des définitions d'arbres

Les arbres tels qu'ils sont décrits au programme de l'option Informatique sont ce qui est nommé dans *Introduction à l'algorithmique* [4], à la page 88, « des arbres enracinés ». La définition de départ est **G**raphe **N**on **O**rienté **C**onnexe **A**cyclique (ou GNOCA). Un graphe est un couple $(\mathcal{E},\mathcal{R})$ où $\mathcal{E}$ est un ensemble et $\mathcal{R}$ une relation sur $\mathcal{E}$. Les éléments de $\mathcal{E}$ sont dits sommets, les couples en relation sont joints par un arc. Le graphe est non orienté si on suppose la relation symétrique. Il est acyclique s'il n'y a pas de cycle, c'est-à-dire qu'il n'existe pas de liste $s_1,\ldots,s_n$ $(n \geq 3)$ de sommets distincts telle que s_k soit relié à s_{k+1} pour tout k et s_n relié à s_1.

Théorème On a alors l'équivalence des définitions suivantes :

(i) $\mathcal{A}$ est un arbre (GNOCA).

(ii) $\mathcal{A}$ est un GNO et deux quelconques sommets sont toujours reliés par un unique chemin injectif.

(iii) $\mathcal{A}$ est un GNOC et toute suppression d'arêtes le « déconnecte ».

(iv) $\mathcal{A}$ est un GNOC et $\text{card}(\textit{Arcs}) = \text{card}(\textit{Sommets}) - 1$.

(v) $\mathcal{A}$ est un GNOA et $\text{card}(\textit{Arcs}) = \text{card}(\textit{Sommets}) - 1$.

(vi) $\mathcal{A}$ est un GNOA et tout ajout d'arête le « désacyclise ».

Preuve

(vi) $\Rightarrow$ (i) : Si $\mathcal{A}$ est non connexe, x et y sont non reliés. Alors, l'ajout de l'arête (x,y) ne le rendrait pas non acyclique, d'où une contradiction.

(i) $\Rightarrow$ (ii) : L'existence d'un chemin injectif résulte de connexe (il existe un chemin) et de la

finitude (on prend un chemin de longueur minimum, il est « donc » injectif). Si on a deux chemins injectifs $a_1,\dots,a_m$ et $\alpha_1,\dots,\alpha_n$ joignant (a,b) qui divergent au $p+1$-ième sommet, on peut définir : $q = \min\{k \,/\, k \geq p+1 \text{ et } a_k \in \{\alpha_{p+1},\dots,\alpha_n\}\}$ (q existe car $a_m = \alpha_n = b$). On a un unique i tel que $\alpha_i = a_q$ et on a ainsi constitué un cycle $a_p, a_{p+1},\dots,a_q,\alpha_{i-1},\dots,\alpha_{p+1}$, d'où une contradiction.

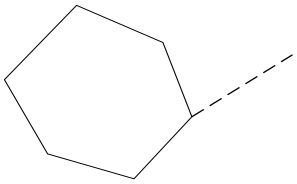
(ii) $\Rightarrow$ (iii) : $\mathcal{A}$ est connexe par l'existence d'un chemin et si l'on supprime l'arête (a,b), alors on ne peut plus joindre a à b à cause de l'unicité du chemin entre les sommets.

(iii) $\Rightarrow$ (iv) : Une relation symétrique est un ensemble de cases cochées dans un tableau (une partie de $\mathcal{E} \times \mathcal{E}$).

Soit s le nombre de sommets et a le nombre d'arcs.

Si on avait $a \leq s-2$, une colonne serait vide et le graphe non connexe. On a donc $a \geq s-1$. Prouvons $\{a \leq s-1\}$ par récurrence sur le nombre de sommets n : la relation est vraie pour $n = 1$. Soit $\mathcal{G}$ un graphe à $n+1$ sommets. En ôtant une arête, on le déconnexifie. On obtient k composantes connexes ($k \geq 2$) qui vérifient toutes les hypothèses de (iii) et ont au maximum n sommets. Pour chacune, $a_i \leq s_i - 1$. On somme ces relations et en ajoutant l'arête enlevée au début, on obtient la relation pour $\mathcal{G}$.

(iv) $\Rightarrow$ (v) : On a $\mathcal{A}$ connexe et $|\text{arcs}| = |\text{sommets}| - 1$. On veut $\mathcal{A}$ acyclique.



S'il existe un cycle $\mathcal{C}$, il a p arêtes et p sommets donc $\mathcal{C} \neq \mathcal{A}$. Il existe donc un sommet qui n'est pas dans $\mathcal{C}$ et comme $\mathcal{A}$ est connexe, il existe un sommet adjacent. Mais ce nouveau $\mathcal{A}'$ vérifie les mêmes hypothèses : on recommence jusqu'à avoir $\mathcal{A}$ en totalité et $|\text{arcs}| = |\text{sommets}|$, d'où une contradiction.

(v) $\Rightarrow$ (vi) : $\mathcal{A}$ a k composantes connexes et chacune vérifie donc $|\text{arcs}| = |\text{sommets}| - 1$ car ce sont des arbres. On ajoute et on a donc $k = 1$: $\mathcal{A}$ est donc connexe.

On a donc (i), donc (ii), donc si on ajoute une arête, on crée un cycle.

3.2 Test de connexité

Après avoir dessiné un graphe, il est nécessaire de savoir s'il s'agit vraiment d'un arbre. Pour cela, il suffit (cf. 3.1) de savoir si $\text{card}(\text{Arcs}) = \text{card}(\text{Sommets}) - 1$ et si le graphe est connexe. Le premier test est assez facile. Quant au deuxième, il est un peu plus délicat mais reste quand même plus aisé que le test d'acyclicité.

Pour savoir si `(S1,...,Sn)` est connexe (`Sk` sommets du graphe), on définit la liste `[V1;...;Vn]` des voisins de `Sk`. On initialise un tableau de n booléens et on met le premier à `true`, car `S1` touche `S1`. Puis, on met à true tous ceux que `Sk` touche en enregistrant ceux que l'on n'a jamais vus dans une liste. On recommence avec les nouveaux tant que la liste n'est pas vide. Quand elle est vide, on regarde si tous les éléments du tableau ont la valeur `true`.

Cette méthode donne le test décrit par le Programme 2.3. La fonction `traite` permet de savoir s'il existe dans `tab` un élément qui n'a pas la valeur `true`. La fonction `cree_voisins` rend la liste des voisins de `t`. Les fonctions `union` et `subtract` sont des fonctions définis dans CAML qui réalisent l'union et la différence de deux listes au sens ensembliste.

Programme 2.3 : Test de connexité

```
let est_connexe depart =
     let temp=make_vect (long_pt+1) false
     and traite tab =
          let result=ref true and c=ref 0
          and long=(vect_length tab -1) in
               while (!result & !c<=long) do
                    result:=tab.(!c); incr c
               done;
               !result
in     let rec test a_traiter deja_vu = match a_traiter with
          |[] -> traite temp
          |t::q -> let voisins=cree_voisins t in
            do_list (fun k -> (temp.(k)<-true)) voisins;
            test (union q (subtract voisins deja_vu))
                 (union (t::voisins) deja_vu);
in     (temp.(depart)<-true; test [depart] []);;
```

3.3 Définition de l'arbre à partir du graphe

Pour passer du graphe à l'arbre, il faut avoir d'abord écrit une fonction de dessin qui permettra de définir le graphe et qui rendra les sommets, les liaisons et la racine du « futur » arbre. Il faut aussi définir un autre type d'arbre, qui, contrairement au précédent, permettra un accès direct aux fils d'un noeud (cf. Programme 2.4).

Programme 2.4 : Un autre type d'arbre

```
type GNOCA =
     |Nil
     |Feuille of string
     |Noeud of string*(GNOCA vect)
;;
```

Pour créer l'arbre, on définit la matrice de voisinage du graphe : un élément `v.(i).(j)` est initialisé à `1` si les sommets `i` et `j` sont reliés, `0` sinon. On part alors de la racine du futur arbre et on définit ses fils comme étant ses voisins. On « coche » les voisins dans la matrice en transformant, sur la ligne qui leur correspond, les `1` en `2`. Puis on recommence avec les fils de la racine. On obtient alors la fonction de transformation d'un graphe en arbre (cf. Programme 2.5).

Programme 2.5 : Transformation d'un graphe en arbre

```
let traitement (tab_pt,tab_liaison,pt_racine) =
     let long_pt=(vect_length tab_pt)-1
     and long_liaison=vect_length tab_liaison

in     let voisinage=make_matrix (long_pt+1) (long_pt+1) 0
     and indice x tab = let c=ref 0 and fini=ref false in
          while (not !fini) do (fini:=(x=tab.(!c));incr c) done;
          !c -1
     and met_a_jour k =
          for c=0 to long_pt do
               if voisinage.(c).(k)=1 then voisinage.(c).(k) <- 2
          done
     and cree_voisins k = let voisins=ref [] in
          for m=0 to long_pt do
               if voisinage.(k).(m)=1 then voisins:=m::(!voisins)
          done;
          !voisins

in     let est_connexe depart =
          let temp=make_vect (long_pt+1) false
          and traite tab =
          let result=ref true and c=ref 0 and long=(vect_length tab -1)
               in     while (!result & !c<=long) do
                         result:=tab.(!c); incr c
                    done;
                    !result
     in let rec test a_traiter deja_vu = match a_traiter with
          |[] -> traite temp
          |t::q -> let voisins=cree_voisins t in
            do_list (fun k -> (temp.(k)<-true)) voisins;
            test (union q (subtract voisins deja_vu))
                 (union (t::voisins) deja_vu);
     in (temp.(depart)<-true; test [depart] [])

in     let rec cree_arbre k =
     let courant=tab_pt.(k) and voisins=vect_of_list (cree_voisins k) in
          do_vect met_a_jour voisins;
          match (vect_length voisins) with
               |0 -> Feuille !(courant.val)
               |_ -> Noeud(!(courant.val), map_vect cree_arbre voisins);

in     if long_pt= -1 then Nil
     else let ind_rac=indice pt_racine tab_pt in
          begin
          for c=0 to (long_liaison-1) do
               let (x,y)=tab_liaison.(c) in
                    begin
                    voisinage.(indice x tab_pt).(indice y tab_pt) <- 1;
                    voisinage.(indice y tab_pt).(indice x tab_pt) <- 1;
                    end
          done;
          if (est_connexe ind_rac & long_liaison=long_pt)
          then (met_a_jour ind_rac; cree_arbre ind_rac)
          else raise (Erreur "Graphe cyclique ou non connexe")
          end
;;
```

Chapitre 3

Les automates

1 Le tracé des transitions

Pour dessiner des automates « à la main », on utilise des courbes pour tracer les liaisons. Pour les faire dessiner par un programme, la solution peut être d'utiliser des courbes de BEZIER. Elles serviront uniquement en tant qu'outil ici et ne font donc pas partie intégrante de ce TIPE. Néanmoins, un peu de théorie et quelques explications semblaient indispensables.

Remarque : Tous les développements théoriques suivants sont tirés du *Cours d'Infographie* [5] de Patrick TRAU [1].

On a souvent besoin de représenter avec l'ordinateur des courbes ou surfaces dont le type n'est soit non classifiable (ni parabole, ni cercle, ...) soit non connu à l'avance. On les regroupe généralement sous le nom de « courbes et surfaces non mathématiques », car l'équation mathématique n'est pas connue par le programmeur. Ces courbes ou surfaces sont en général définies par un ensemble de points, plus ou moins nombreux.

1.1 Introduction

Le problème qui se posait à la régie RENAULT consistait à représenter les surfaces de carrosserie dessinées par les designers, pour réaliser les matrices d'emboutissage. La pièce était définie par des sections planes successives. A l'arrivée des machines à commande numérique, la méthode manuelle a du être abandonnée. Après création d'une maquette 3D (en bois, plâtre, ... modifiée au fur et à mesure), la surface était déterminée par une matrice de points obtenus à l'aide d'une machine à mesurer en 3D. Pour obtenir une bonne précision, le nombre de points nécessaire est très important. Le nombre de maquettes réalisées en cours de conception étant important (d'où pertes de temps), on a demandé aux stylistes de créer sur ordinateur, les maquettes pouvant alors être facilement et rapidement réalisées en commande numérique.

P. BEZIER [2] a du mettre au point une méthode permettant de définir la surface par un nombre minimal de points caractéristiques, permettre de modifier facilement la surface par déplacement d'un minimum de points, pouvoir représenter toute surface (y compris plane), sans « cassure » (continûment dérivable).

1. Patrick TRAU, professeur d'informatique et d'automatisme à l'IPST (Institut Professionnel des Sciences et Technologies) de l'Université Louis PASTEUR de Strasbourg

2. Cette représentation des courbes a été découverte indépendamment par Pierre BEZIER, ingénieur pour RENAULT et Paul DE CASTELJAU, pour CITROËN. Le secret sur ses travaux a empêché P. DE CASTELJAU de publier ses travaux en 1959, bien qu'il soit antérieur à ceux de P. BEZIER (1962). C'est pourquoi l'on a retenu le nom de BEZIER. Mais l'algorithme fondamental, à la base de la construction de ces courbes, est aujourd'hui crédité à P. DE CASTELJAU.

1.2 Courbes de BEZIER

Soit $\vec{P}(u) = \overrightarrow{OA} + f_1(u)\,\vec{a}_1 + \cdots + f_n(u)\,\vec{a}_n$ avec $u \in [0;1]$.

Cette définition de la courbe est possible aussi bien en 2D qu'en 3D. Les vecteurs $a_1,\ldots,a_n$ forment le « polygone caractéristique ».

Conditions

On impose à l'origine ($u = 0$) :
- passage en A ($f_i(0) = 0$ pour $i > 0$)
- la tangente ne dépend que de $\vec{a}_1$ ($f_i'(0) = 0$ pour $i > 0$)
- la dérivée seconde ne dépend que de $\vec{a}_1$ et $\vec{a}_2$ ($f"_i(0) = 0$ pour $i > 0$)
- de façon générale, la dérivée k-ième ne dépend que des k vecteurs $\vec{a}_1$ à $\vec{a}_k$

De plus, le dernier sommet du polygone caractéristique impose le point final de la courbe ($u = 1$), la tangente en ce point ne dépend que du dernier vecteur $\vec{a}_n$, etc.

Exemple : pour $n = 3$, on impose les conditions suivantes :
- passage de la courbe aux deux extrémités du polygone caractéristique :

$$f_1(0) = f_2(0) = f_3(0) = 0 \quad \text{et} \quad f_1(1) = f_2(1) = f_3(1) = 1$$

- la tangente en A ne dépend que de $\vec{a}_1$ (idem à l'autre extrémité) :

$$\frac{d}{du} f_2(0) = \frac{d}{du} f_3(0) = 0 \quad \text{et} \quad \frac{d}{du} f_1(1) = \frac{d}{du} f_2(1) = 0$$

- la dérivée seconde en A ne dépend que de $\vec{a}_1$ et $\vec{a}_2$:

$$\frac{d^2}{du^2} f_3(0) = 0 \quad \text{et} \quad \frac{d^2}{du^2} f_1(1) = 0$$

Détermination des polynômes de BEZIER

Plaçons-nous encore dans le cas de trois points : on a 12 conditions. On peut donc prendre trois fonctions du troisième degré.

Soit $f_i(u) = a\,u^3 + b\,u^2 + c\,u + d$. La dérivée sera donc $f_i'(u) = 3a\,u^2 + 2b\,u + c$, et la dérivée seconde $f"_i(u) = 6a\,u + 2b$.

D'où : $f_1(0) = 0 \Rightarrow d = 0$
$f"_1(1) = 0 \Rightarrow 6a + 2b = 0$ d'où $b = -3a$
$f_1'(1) = 0 \Rightarrow 3a - 6a + c = 0$ d'où $c = 3a$
$f_1(1) = 1 \Rightarrow a + b + c = a + 3a - 3a = 1$ d'où $a = 1, b = -3, c = 3$
Donc $f_1(u) = u^3 - 3u^2 + 3u$

De même : $f_2(0) = 0 \Rightarrow d = 0$
$f_2'(0) = 0 \Rightarrow c = 0$
$f_2(1) = 0 \Rightarrow a + b = 1$ d'où $b = 1 - a$
$f_2'(1) = 0 \Rightarrow 3a + 2b = 3a + 2 - 2a = 0$ d'où $a = -2, b = 3$
Donc $f_2(u) = -2u^3 + 3u^2$

Et : $f_3(0) = 0 \Rightarrow d = 0$
$f_3'(0) = 0 \Rightarrow c = 0$
$f"_3(1) = 0 \Rightarrow b = 0$
$f_3(1) = 1 \Rightarrow a = 1$
Donc $f_3(u) = u^3$

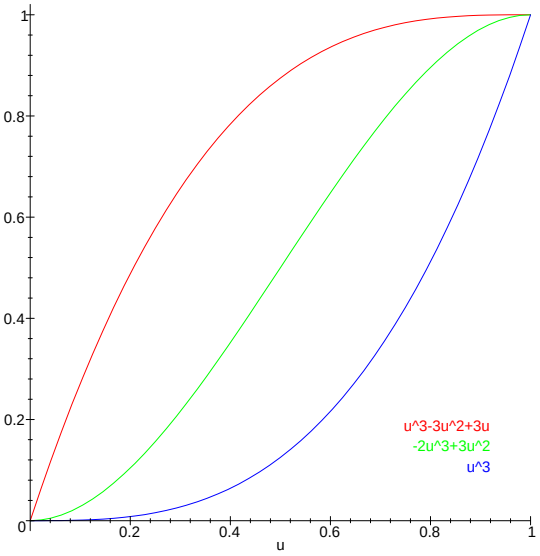


FIGURE 3.1 : *Tracé des fonctions* $f_{im}(u)$ *dans le cas* $m = 3$

Pierre BEZIER a déterminé le cas général (à l'ordre m) :

$$f_{im}(u) = \frac{(-u)^i}{(i-1)!}\frac{d^{i-1}}{du^{i-1}}\left(\frac{(1-u)^m - 1}{u}\right) \quad \text{ou} \quad f_{im}(u) = \sum_{k=i}^{m}\binom{m}{k}\binom{k-1}{i-1}(-1)^{i+k}u^k$$

$$\text{avec} \quad \binom{a}{b} = \frac{a!}{b!(a-b)!}$$

Cette seconde écriture se traduit facilement de manière informatique, bien que comportant beaucoup de calculs.

Forme polynomiale

Plutôt que d'utiliser la formulation vectorielle, on définit le polygone caractéristique par les coordonnées x_i, y_i, z_i de ses sommets. On obtient les coordonnées de tout point de la courbe :

$$\begin{aligned} X(u) &= \sum_{i=0}^{n} X_i \binom{n}{i} u^i (1-u)^{n-i} \\ Y(u) &= \sum_{i=0}^{n} Y_i \binom{n}{i} u^i (1-u)^{n-i} \\ Z(u) &= \sum_{i=0}^{n} Z_i \binom{n}{i} u^i (1-u)^{n-i} \end{aligned}$$

Pour accélérer les calculs, on cherchera à stocker dans des tableaux les valeurs intermédiaires réutilisées plusieurs fois pour un tracé. D'autres formulations ont également été développées, on peut les trouver dans divers ouvrages, par exemple dans la collection *Maths et CAO*.

Intérêt

On peut définir une courbe complexe avec assez peu de points caractéristiques. Certes la courbe ne passe pas par ces points, mais avec un peu d'expérience, on « sent » facilement comment déplacer quelques points caractéristiques pour modifier une courbe. Ceci est encore plus intéressant pour des surfaces tridimensionnelles, où les modifications elles aussi se limiteront aux déplacements de quelques points.

Exemples

Ces exemples (cf. FIGURE 3.2) nous montrent la diversité des courbes possibles avec trois ou quatre points.

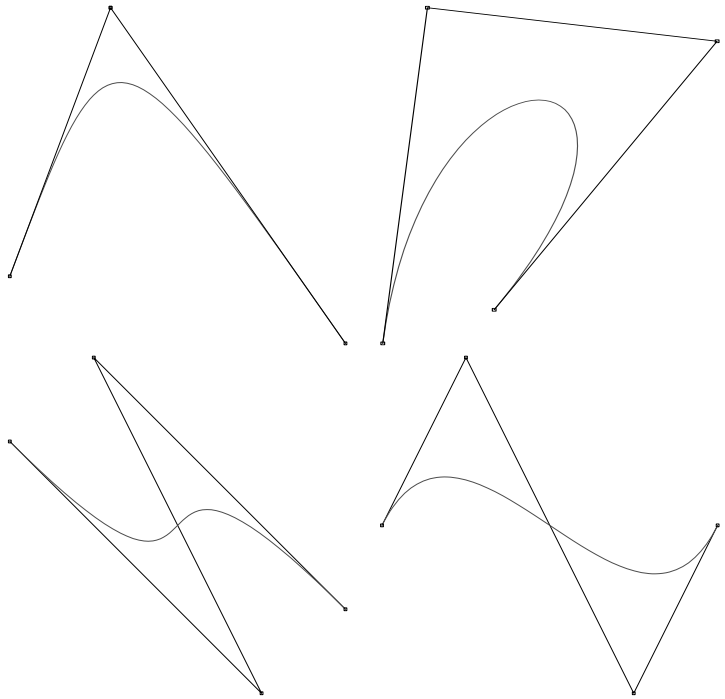


FIGURE 3.2 : *Exemples de courbes de BEZIER*

1.3 Programmation

Pour tracer une courbe de BEZIER, il faut définir le type des points et de la courbe (cf. Programme 3.1). Ensuite et pour suivre les conseils précédents, les valeurs des paramètres, c'est-à-dire les coefficients des polygones caractéristiques, sont stockées dans une variable globale pour éviter de les recalculer à chaque tracé. C'est la fonction `Cvalues` (cf. Programme 3.2) qui est chargée de ce travail, sachant que l'on utilisera 100 points pour tracer la courbe. Il ne reste plus alors qu'à tracer la courbe de BEZIER (cf. Programme 3.3), en ayant auparavant crée la liste `tValues` qui contient les valeurs des paramètres.

Programme 3.1 : Les types pour les courbes de BEZIER

```
type PointOrVector= PoV of (float*float);;
type BezierCurve=
     Bezier of PointOrVector*PointOrVector*PointOrVector*PointOrVector;;
```

2 Création graphique d'automate

2.1 Généralités

Les états de l'automate seront représentés par un entier. Au cours de la création, il sera nécessaire de connaître leurs coordonnées à l'écran. Quant aux transitions, elles seront codées par un état de départ et de fin, un caractère ainsi qu'une courbe de BEZIER (cf. Programme 3.1). On utilisera donc les types décrits par le Programme 3.4.

Programme 3.2 : Précalcul des paramètres des courbes de BEZIER

```
let CValues nbPts= let Step=1./.nbPts in
     let rec rValues t l=
        let t2=t*.t in
        let t3=t2*.t in
        let ct=1.-.t and ct2=1.-.2.*.t+.t2 and ct3=1.-.3.*.t+.3.*.t2-.t3
     in    if t>.1. then l
           else rValues (t+.Step) ((t3,3.*.t2*.ct,3.*.t*.ct2,ct3)::l)
in     rValues 0. [];;
```

Programme 3.3 : Tracé des courbes de BEZIER

```
let DrawBezierCurve
  (Bezier((PoV(x1,y1)),(PoV(x2,y2)),(PoV(x3,y3)),(PoV(x4,y4)))) couleur=
     let PutPt (x,y,z,t)=
          lineto (trunc(x1*.x+.x2*.y+.x3*.z+.x4*.t))
                 (trunc(y1*.x+.y2*.y+.y3*.z+.y4*.t))
in     begin
     set_color couleur;
     moveto (trunc x1) (trunc y1);
     do_list PutPt tValues
     end
;;
```

Programme 3.4 : Le type des états et transitions

```
type etat= {numero : int; pt : PointOrVector };;
type transition= {debut : etat; fin : etat; car : char;
                  courbe : BezierCurve};;
```

La fonction de dessin permet à l'utilisateur de créer les états, les transitions, de les supprimer et de définir les états initial et final. L'état initial est repéré par une flèche horizontale ou verticale pointant sur lui et l'état final est doublement cerclé. Les transitions sont orientées avec deux flèches placées au quart et au trois quart de la courbe, l'étiquette étant tracée en leur « milieu » (point obtenu pour la valeur 0.5 du paramètre). Toutes ces actions seront effectuées à la souris. Le résultat de la fonction de dessin est :

- une liste de transition
- l'état initial
- l'état final

2.2 Création d'un automate « simple »

A partir du résultat de la fonction de dessin, il faut définir l'automate dont le type est décrit par le Programme 3.5, avec `ordre` le nombre d'états de l'automate.

Programme 3.5 : Le type d'automate

```
type automate1 = {ordre : int;
                  initial : int; final : int -> bool;
                  transition : (int*char) -> int};;
```

Cela nécessite néanmoins quelques transformations. Il faut en effet supprimer toutes les données nécessaires au dessin, c'est-à-dire toutes les coordonnées et autres courbes de BEZIER. Ce travail est effectué par la fonction finale (cf. Programme 3.6).

Programme 3.6 : La fonction finale de création de l'automate

```
let boulot () = let (transit, init, fina)= (open_graph ""; dessin())
in     let rec liste_a_fonction liste =
          let intermediaire (e,a) =
               try (assoc (e,a) liste)
               with Not_found -> failwith "Transition non prevue"
          in intermediaire
     and transforme = function
          |[]   -> []
          |t::q -> ((t.debut.numero,t.car),t.fin.numero)::transforme q
in     {initial=init.numero;
      final=(function n -> n=fina.numero);
      transition=liste_a_fonction (transforme transit)}
;;
```

La fonction `liste_a_fonction` permet de transformer notre liste de transitions en une véritable fonction. Elle utilise la fonction `assoc` de CAML qui, à un élément `a`, va associer `b` d'une liste de couples, `(a,b)` étant la première occurrence dans la liste. L'automate est automatiquement rendu complet car les transitions absentes tombent dans un puits, c'est-à-dire génère une erreur.

2.3 Création d'un automate avec transitions instantanées

Les modifications à apporter sont minimes. Il faut d'une part définir un nouveau type (cf. Programme 3.7). Ceci permet déjà une amélioration : on peut en effet créer un automate non déterministe puisque les transitions sont stockées en tant que liste et non plus comme une fonction.

La fonction de dessin se chargera de définir, au gré de l'utilisateur, des transitions instantanées, dessinées en vert à l'écran. Une autre modification a été aussi apportée à cette fonction : elle permet maintenant de définir plusieurs états finaux et plusieurs états initiaux. Les mêmes transformations (cf. Programme 3.6) sont nécessaires pour la création de l'automate.

Programme 3.7 : Le type d'automate avec transitions instantanées

```
type automate2 = {ordre : int;
                  initial : int list; final : int list;
                  instantanee : (int*int) list;
                  transition : ((int*char)*int) list};;
```

3 Dessin d'automate

3.1 Généralités

Un des objectifs de départ était de pouvoir dessiner un automate donné. Mais la tâche, en apparence assez simple, s'est révélée beaucoup plus ardue. En effet, un problème s'est très vite posé. Il concernait la minimisation des croisements de transitions. Il fallait choisir judicieusement d'une part, la position des états à l'écran et d'autre part les courbes définissant les transitions afin de minimiser ces croisements.

Une solution peut être d'analyser très profondément l'automate et ses transitions pour définir des « noyaux », c'est-à-dire rassembler les états possédant beaucoup de transitions entre eux. On peut alors s'occuper de ses « noyaux » constitués de 2 ou 3 états, les traiter de façon très poussée pour minimiser les croisements. Il suffit ensuite de les relier avec les transitions qu'ils possèdent en commun.

3.2 Définition des « noyaux » d'un automate

Ceci peut se faire encore une fois grâce à une matrice de voisinage : l'élément $v_{i,j}$ contient le nombre de liaisons qu'ils existent entre les états i et j. Pour définir une classe, il suffit de partir d'un élément et de lui adjoindre les états ayant au moins deux transitions en communs avec lui. La matrice de voisinage étant préalablement définie, la fonction (cf. Programme 3.8) va construire la liste des classes d'un automate donné. La fonction `range` calcule la liste `[1;...;n]` et la fonction `copains` la liste des états possédant au moins deux transitions avec `t`.

Programme 3.8 : Définition des « noyaux » d'un automate

```
let rec def_classe () =
     let rec rdef result a_voir = match a_voir with
          |[] -> result
          |t::q ->  let voisins=copains t
                    in let new=it_list union [t] (rdef [] voisins)
                    in rdef (new::result) (subtract a_voir new)
in rdef [] (range nbre);;
```

3.3 Calcul de la position des états et des transitions

A partir des classes définies précédemment, on définit la position des états sur l'écran : on répartit les classes verticalement, puis chaque classe est représentée en plaçant régulièrement les états sur une horizontale.

Ensuite, pour chaque classe, les transitions sont définies en faisant varier un angle et la longueur de la tangente à la courbe de BEZIER (cf. FIGURE 3.3). La variation de l'angle est choisie pour augmenter plus rapidement au début qu'à la fin et ainsi tendre lentement vers $\pi/2$. Quand ces transitions ont été définies, il suffit de définir celles existant entre les classes.

Le résultat de la fonction sera le même que celui de la fonction de dessin d'automate, c'est-à-dire la liste des états, des transitions, des transitions instantanées, des états initiaux et des états finaux. Ceci peut permettre de pouvoir modifier ces données justement grâce à cette fonction de

dessin (déplacement d'états, de transitions, ...) et donc d'améliorer le résultat, qui est souvent imparfait.

Les exemples présentés montrent ce que ce programme parvient à faire. On voit d'une part que même si certains croisements existent, le résultat est proche de ce que l'on peut faire à la main (cf. FIGURE 1.2 et FIGURE 3.4). D'autre part, ce résultat dépend beaucoup de l'automate : l'automate des MINES (cf. FIGURE 3.5) est correctement représenté alors que celui de CENTRALE (cf. FIGURE 3.6) est complètement illisible car il nécessite une représentation par arbre.

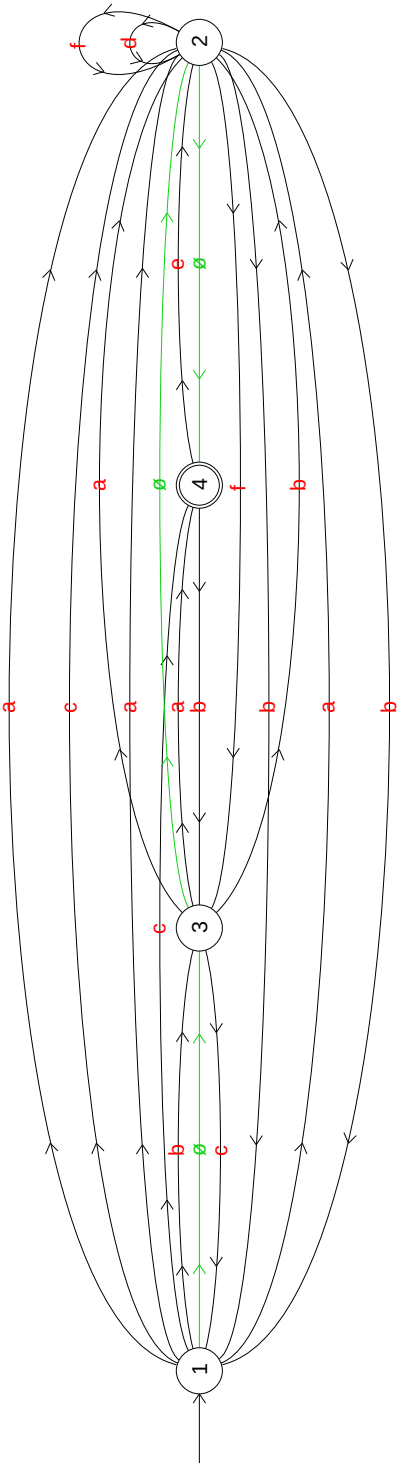


FIGURE 3.3 : *Pour détailler la méthode de calcul des transitions*

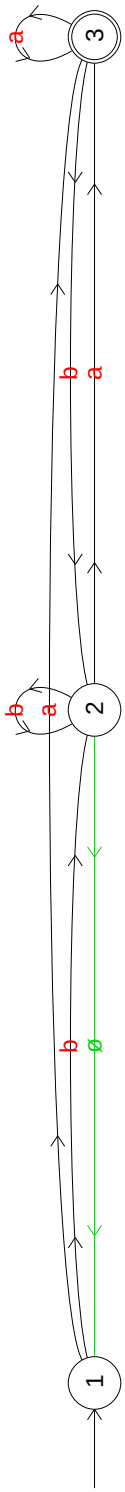


FIGURE 3.4 : *L'automate de la* FIGURE *1.2*

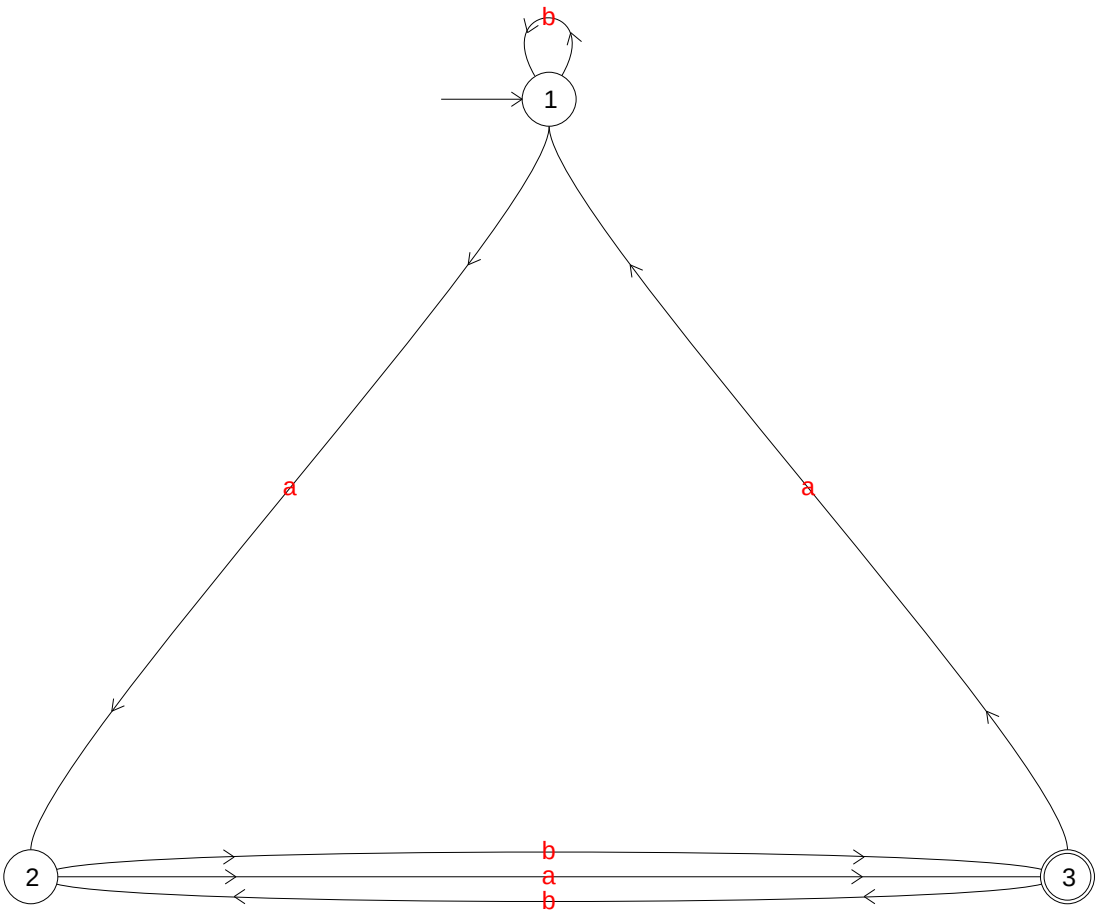


FIGURE 3.5 : *L'automate du sujet* MINES *'98*

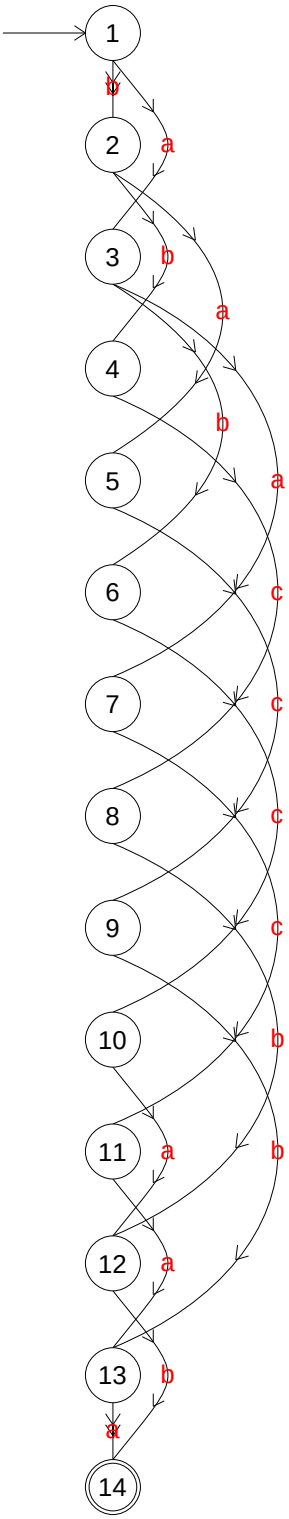


FIGURE 3.6 : *L'automate du sujet* CENTRALE *'98*

Chapitre 4

Conclusion

1 Généralités

Deux difficultés majeures se sont présentées au cours de cette année. D'une part, j'ai du travailler de façon individuelle, ce qui n'a pas facilité une recherche bibliographique sérieuse et un travail rapide. D'autre part, les rares documents que j'ai pu trouver étaient souvent inabordables avec mon niveau (en Théorie des graphes planaires par exemple) et j'ai du souvent improvisé particulièrement avec le dessin d'automate.

Tous les utilitaires présentés ici sont fonctionnels mais ils possèdent de nombreux défauts pour un utilisateur :

- leur lenteur, dans le rafraîchissement de l'écran par exemple
- leur manque d'ergonomie
- peut être, leur manque d'utilité ...

En effet, l'aide à la programmation n'est pas quelque chose de facile. On s'aperçoit très vite de ses limites, car il faut pouvoir penser « un niveau au-dessus », prévoir tous les besoins d'un utilisateur qui devra pouvoir se servir de notre programme sans avoir besoin de consulter un fichier d'aide rébarbatif.

De plus, l'existence de deux projets similaires, mais nettement plus avancés et plus étendus, en matière de visualisation graphique d'arbres et d'automates, réalisés par Messieurs J. ODOUX et M. QUERCIA [1], ne facilite pas la comparaison.

2 Améliorations possibles

Le manque de temps a empêché le développement de certaines idées. Il aurait été, par exemple, intéressant de pouvoir, à partir du dessin d'un arbre, faire des changements dynamiques de la racine, c'est-à-dire de façon visuelle avec toutes les étapes, ou encore d'ajouter toutes les fonctions de déterminisation et de minimalisation d'un automate.

Il aurait aussi fallu pousser un peu plus loin le dessin des automates, notamment en y ajoutant une interface permettant une intervention de l'utilisateur dans le résultat final, le programme faisant uniquement une partie du travail.

1. `OdouxGraph` : J. ODOUX, professeur de l'option Informatique de Grenoble (`odoux@ac-grenoble.fr`)
`AutomatX` : M. QUERCIA, professeur de l'option Informatique de Dijon (`quercia@cal.enst.fr`)

Table des figures

1.1 Exemple d'arbre 3
1.2 Exemple d'automate 3

3.1 Tracé des fonctions $f_{im}(u)$ dans le cas $m = 3$ 11
3.2 Exemples de courbes de BEZIER 12
3.3 Pour détailler la méthode de calcul des transitions 16
3.4 L'automate de la FIGURE 1.2 17
3.5 L'automate du sujet MINES '98 18
3.6 L'automate du sujet CENTRALE '98 19

Liste des programmes

2.1 Le type d'arbre 4
2.2 Le dessin d'arbre 5
2.3 Test de connexité 7
2.4 Un autre type d'arbre 7

2.5 Transformation d'un graphe en arbre 8
3.1 Les types pour les courbes de BEZIER 12
3.2 Précalcul des paramètres des courbes de BEZIER 13
3.3 Tracé des courbes de BEZIER 13
3.4 Le type des états et transitions 13
3.5 Le type d'automate 14
3.6 La fonction finale de création de l'automate 14
3.7 Le type d'automate avec transitions instantanées 14
3.8 Définition des « noyaux » d'un automate 15

Bibliographie

[1] Pierre WEIS; Xavier LEROY, *Manuel de référence du langage CAML* - InterEditions

[2] Pierre WEIS; Xavier LEROY, *Le langage CAML* - InterEditions

[3] *Le site non officiel de Klik & Play*, `http://www.tne.net/silky`

[4] Thomas CORMEN; Charles LEISERSON; Ronald RIVEST, *Introduction à l'algorithmique* - Dunod

[5] Patrick TRAU, *Cours d'Infographie* - accessible depuis le site de l'Université Louis PASTEUR de Strasbourg : `http://l3mi.u-strasbg.fr/pat/program/infogr.htm`