Autour du Manège Enchanté

Antonin Dudermel

Voici la version complète de mon rapport de TIPE de type ENS. Le résumé s'arrête à la page 5, vient ensuite l'ensemble de mes programmes, puis, page 35, l'ensemble des figures. Le code complet (avec un `Makefile`*) se trouve à l'adresse* `https://github.com/adud/info/tree/master/roundabouts/`

Résumé

Ce TIPE a pour but d'étudier le rond-point de Swindon, Angleterre, appelé The Magic Roundabout. En premier temps, une étude qualitative du rond-point a été faite à l'aide de la théorie des graphes, puis deux automates cellulaires modélisant le trafic routier ont été implantés, sans que je réussisse à les adapter à l'étude du rond-point.

1 Un Graphe pour le Manège

1.1 Le Manège

Le Manège enchanté est un rond-point assez particulier situé sur une intersection à 5 voies à Swindon en Angleterre : il est composé de 5 petits ronds-points tournant dans le sens horaire (comme les ronds-points anglais) disposés en périphérie d'un grand rond-point central tournant dans le sens inverse, la priorité étant aux voitures situés à l'intérieur des petits ronds-points, comme le montre la figure 3.

Une telle disposition permet une diversité des itinéraires possibles pour aller d'une entrée à une sortie (voir figure 1). Le manège serait grâce à cela une réponse plus efficace au problème des intersections routières : assurer un trafic le plus fluide possible, des distances plus courtes, des infrastructures plus sûres… L'objectif de ce TIPE est de montrer que le manège enchanté remplit bien de telles conditions. Nous avons pour cela mis en place deux modèles : un utilisé en pratique pour étudier des infrastructures routières, par automates cellulaires, mais face à la complexité de ce modèle, nous nous sommes rabattus sur une étude plus élémentaire *via* la théorie des graphes.

1.2 Modéliser par un graphe

On peut aisément représenter un ensemble de routes par un graphe orienté pondéré : il suffit de considérer chaque intersection comme un sommet et chaque route comme une arête reliant une intersection à une autre de poids la longueur de la route. En appliquant ce principe au manège enchanté, on aboutit au graphe représenté par la figure 2a. De même on peut construire un graphe représentant le rond-point formé par la périphérie du manège.

L'intérêt principal de l'étude étant plus la forme du manège que cet exemple particulier, par souci d'implantation, le graphe a encore été simplifié en considérant les sorties et les intersections internes disposées en pentagones réguliers.

1.3 Réduction des distances

Muni de ces deux graphes, on peut dès lors appliquer l'algorithme de Floyd-Warshall pour connaître les distances entre les entrées-sorties, et les comparer entre les deux graphes. Le tableau 4 montre le quotient des distances du manège sur celles du rond-point. Sans surprise, on remarque un gain énorme quand il s'agit de prendre la sortie située immédiatement à gauche de l'entrée, puisqu'il n'est pas nécessaire de faire tout le tour du rond-point. Le manège réduit donc bien les distances par rapport à un rond-point classique.

1.4 Résistant aux coupures de section

1.4.1 arc critique

Un point remarquable du manège est que, comme le montre la figure 1, le conducteur dispose de plusieurs itinéraires pour aller d'une entrée à une sortie. Ainsi, si suite à un évènement, certaines sections se trouvent impraticables, le manège reste fonctionnel. Un algorithme permet de trouver quelles sections ne sont pas nécessaires au bon fonctionnement du manège. Défini formellement :

Définition 1 *Soit* $G = (V,E)$ *un graphe orienté. L'arc* $(a,b) \in E$ *est dit critique si le graphe* $G' = (V, E \backslash \{(a,b)\})$ *n'a pas les mêmes composantes fortement connexes que* G.

Un théorème intéressant nous permet de déterminer de tels arcs :

Théorème 1 *Soit* $G = (V,E)$ *un graphe orienté fortement connexe,* $(a,b) \in E$*, alors :* $G' = (V, E \backslash \{(a,b)\})$ *est fortement connexe SSI il existe un chemin de a à b dans* G'

Pour savoir si (a,b) est un arc critique, il suffit donc de déterminer si b est atteignable depuis a dans G*ı*, à l'aide d'un simple parcours du graphe.

1.4.2 implantation et complexité

Considérons un graphe $G = (V,E)$. L'algorithme fonctionne donc sur le principe décrit ci-dessus : On dispose de la liste des arcs critiques, initialement vide. Pour chaque arête (a,b) du graphe, on supprime cette arête, si b n'est pas accessible depuis a dans G*ı*, alors on ajoute (a,b) à la liste des arcs critiques, puis on rajoute (a,b) à G*ı*.

Le besoin de supprimer des arêtes pousse à choisir la structure de matrice d'adjacence (voir 3.1) pour représenter le graphe, suppression effectuée pour cette structure en temps constant. Dans ce cas, on rappelle la complexité en temps d'un parcours de $G : O(|V|^2)$.

La complexité totale de l'algorithme est alors un $O(|V|^2 + |E||V|^2) = O(|E||V|^2)$

2 Un Modèle par automate cellulaire

2.1 Automate cellulaire

Le principe des automates cellulaires est de décrire le système en discrétisant le temps et l'espace. Dans ces modèles, les voitures évoluent sur des cases, et ont des vitesses entières. Empiriquement, on prend un pas temporel $\Delta t = 1,2$ s et un pas spatial $\Delta x = 7.5$ m.

Notre automate se base sur celui de Kai Nagel et Michael Schreckenberg [1], adapté à une section de route, dont nous rappelons brièvement le principe ici : on considère n voitures $c_1, \dots, c_n$ sur la section. On note pour $i \in [\![1; n]\!]$, $v_{i,t} \in [\![1; v_{\mathrm{M}}]\!]$ la vitesse du véhicule i, et $x_{i,t}$ sa position au temps t. La règle de transition est définie ainsi :

- Accélération : toute voiture n'ayant pas atteint la vitesse maximale voit sa vitesse incrémentée de 1 : $v_{i,t'} = \min(v_{i,t} + 1, v_{\mathrm{M}})$
- Décélération : chaque véhicule freine pour ne pas percuter la voiture de devant : $v_{i,t''} = \min(v_{i,t'}, x_{i+1,t} - x_{i,t} - 1)$
- Décélération aléatoire (bruit) : chaque véhicule a la probabilité $p \in [0; 1]$ de freiner : $v_{i,t'''} = \max(v_{i,t''}, 0)$ avec une probabilité p, $v_{i,t''}$ sinon
- Avancer : on a déterminé la vitesse de chaque voiture au temps $t+1$, on accède à la position de chaque voiture au temps $t+1$: $v_{i,t+1} = v_{i,t'''}$ et $x_{i,t+1} = x_{i,t} + v_{i,t+1}$

2.2 Le Problème des intersections

Cependant, ce modèle n'est pas adapté à l'étude d'un rond-point : il faut définir le comportement d'une voiture aux intersections. Par exemple, figure 5, quelle voiture faire passer ? J'ai donc tenté d'établir un modèle d'automate répondant à cette question, automate que j'ai ensuite comparé à celui proposé par Chen Rui-Xiong, Bai Ke-Zhao et Liu Mu-Ren[2] pour les ronds-points, décrivant les interactions entre une voie prioritaire et une voie non-prioritaire : si deux voitures, une sur la voie prioritaire, une autre sur la voie non-prioritaire (il s'agit nécessairement des voitures de tête sur leur voie) cherchent à atteindre l'intersection, que faire ?

2.2.1 Le modèle de priorité dynamique

Ce modèle est plus complexe : il considère le temps nécessaire à chaque voiture pour arriver à l'intersection, puis fait avancer d'abord la voiture arrivant la première, puis la voiture arrivant ensuite, en tenant compte du déplacement de la première. En cas d'égalité, c'est la voiture la plus rapide qui passe, et encore en cas d'égalité, c'est la voiture sur la voie prioritaire qui passe. Ce modèle ne me semblait pas très réaliste : il impliquait en effet que les conducteurs anticipent des temps d'une durée de l'ordre de $\Delta t/6 = 0,2$ s. Ce qui m'a poussé à développer le modèle de priorité absolue, aussi plus simple à implanter (pour plus de détails sur le modèle de priorité dynamique, voir [2])

2.2.2 Le modèle de priorité absolue

Ce modèle correspond à un conducteur très prudent : il consiste à faire passer la voiture prioritaire, et à faire freiner la voiture non-prioritaire juste devant l'intersection (sur la dernière case de sa voie).

2.3 Étude locale

2.3.1 Grandeurs étudiées : le diagramme fondamental

Pour l'étude d'une infrastructure routière, on étudie principalement trois grandeurs : le flux J, la vitesse v, et la densité ρ. Le flux J représente le nombre de voitures passant par unité de temps à travers une surface donnée, c'est une grandeur qu'on cherchera à optimiser. ρ représente le nombre de voitures par unité d'espace, plus ρ est élevé, plus on risquera de se retrouver dans une situation d'embouteillage. On cherchera bien sûr aussi à optimiser v.

Une relation de mécanique des fluides (ou par analogie avec l'électromagnétisme) nous donne la relation pour une section $J = \rho * v_m$ où v_m est la vitesse moyenne des véhicules sur la section.

D'après cette relation, on construit le diagramme fondamental $J = f(\rho)$ de la figure 6.

2.3.2 Expérience

Le but de cette expérience est d'étudier l'influence de la densité de la voie prioritaire, sur la voie non-prioritaire. Pour cela, nous avons simulé grâce à l'automate une intersection à deux voies, avec sur la voie non-prioritaire un flot constant et peu dense de voiture et sur la voie prioritaire, un flot croissant de voitures, ce pour les deux modèles d'intersection. Le graphique obtenu est le diagramme fondamental de la voie non-prioritaire, en fonction de la densité de la voie prioritaire (voir figure 5, page 38 l'expérience en cours).

2.3.3 Premiers résultats

Le graphique de la figure 8 met déjà en évidence le problème des intersections à priorité : face à une circulation trop dense sur la voie prioritaire, les voitures de la voie non-prioritaire se retrouvent bloquées, et n'avancent plus du tout. Cependant, un résultat surprenant est qu'on observe peu de différence entre le modèle de priorité absolue et de priorité dynamique. Des observations de la simulation ont montré que le trafic sur la voie prioritaire était extrêmement fluide, même sous de très fortes densités, modèle aberrant ne correspondant pas du tout à la réalité, dû à la suppression de la décélération aléatoire dans le modèle de Nagel-Schreckenberg. Il nous fallait donc revenir sur l'automate modélisant les sections.

2.3.4 Ajout d'un temps de redémarrage

Une amélioration simple à ce modèle est de considérer un temps de redémarrage pour un véhicule à l'arrêt : on se fixe un temps d'arrêt $t_a \in \mathbb{N}$, on ajoute à chaque voiture une variable arrêt a_i, à qui on attribue la valeur t_a chaque fois que la vitesse de la voiture passe de 1 à 0, et ensuite, si une voiture est à l'arrêt, si $a_i = 0$, on passe sa vitesse à 1, sinon, on décrémente de 1 a_i.

Cela permet à la voie prioritaire d'être affectée par la voie non-prioritaire comme le montre la figure 9. Face à une densité élevée de voitures sur la voie prioritaire, ces dernières se retrouvent elles aussi embouteillées, ce qui permet à quelques voitures non-prioritaires de s'engager dans l'intersection (comme le montre l'itération de l'automate en figure 7). Ce modèle permet aussi d'établir des différences entre le modèle de priorité absolue et celui de priorité dynamique, les voitures prioritaires ayant une plus forte tendance à l'arrêt dans le second (voir figure 10)

2.4 Limites du modèle pour le manège

Le modèle en tant que tel ne nous a pas permis d'étudier le manège, et ce, pour plusieurs raisons. Il faut d'abord déterminer si le modèle est pertinent : les graphes obtenus avec un temps de réaction, pour des densités élevées traduisent-ils une réalité empirique où relèvent-ils des limites du graphe? Le manège aurait été aussi lourd à implanter (même si le code permet déjà de construire assez simplement des ronds-points). Enfin, les problèmes et les solutions étudiés pour une étude locale ne se retrouveront pas forcément sur un automate de plus grande taille.

Bibliographie

[1] Kai NAGEL et Michael SCHRECKENBERG : A cellular automaton model for freeway traffic. Journal de physique I, 2(12):2221–2229, 1992.

[2] Chen RUI-XIONG, Bai KE-ZHAO et Liu MU-REN : The CA model for traffic-flow at the grade roundabout crossing. Chinese Physics, 15, 2006.

3 Annexe

3.1 Graphe du manège

graphe.mli

```
type graphe
;;

(*crée un graphe pondere a n sommets sans arete sous
forme matricielle*)
val creer_graphe : int -> graphe;;

val taille : graphe -> int;;

(*cree un arc de poids p de a a b dans g*)
val lier : graphe -> int -> int -> float -> unit;;

(*supprime l'arc a -> b
ne fait rien si l'arc n'existe pas*)
val suppr : graphe -> int -> int;;
```

```
(*succs g s retourne les successeurs du sommet s dans le graphe g*)
val succs : graphe -> int -> int list;;

(*itere_succ f g s itere f sur les successeurs de s dans g*)
val itere_succ : (int -> unit) -> graphe -> int -> unit;;

(*floyd_warshall g applique l'algorithme de Floyd-Warshall au graphe g*)
val floyd_warshall : graphe -> float array array;;

(*acces g a b retourne true ssi il existe un chemin de a a b dans g*)
val acces g a b : graphe -> int -> int -> bool;;
```

graphe.ml

```
(*etude du manege : theorie des graphes*)

type graphe = float array array
;;

let creer_graphe n = Array.make_matrix n n infinity
;;

let taille g = Array.length g
;;

let lier g a b p =
  g.(a).(b) <- p
;;


let suppr g a b = lier g a b infinity
;;

let succs g s =
  let li = ref [] in
  for i=0 to taille g - 1 do
    if g.(s).(i) < infinity then
      li := i:: !li
  done;
  !li
;;

let itere_succ f g s =
  List.iter f (succs g s)
;;

(*implantation de l'algorithme de Floyd-Warshall*)

(*relache graphe -> int -> int -> int -> unit
effectue une operation de relaxation elementaire sur w*)
```

```
let relache w a b k =
  let it = w.(a).(b) in
  let nit = w.(a).(k) +. w.(k).(b) in
  if nit < it then
    w.(a).(b) <- nit
;;

let floyd_warshall g =
  let n = taille g in
  let w = creer_graphe n in
  (*initialiser w*)

  for i=0 to (n-1) do
    for j=0 to (n-1) do
      w.(i).(j) <- g.(i).(j)
    done
  done;

  (*for i=0 to (n-1) do
    w.(i).(i) <- 0.
  done;*)

  for k = 0 to (n-1) do
    for i = 0 to (n-1) do
      for j = 0 to (n-1) do
        relache w i j k
      done
    done
  done;
  w
;;

(*constantes pour les poids des aretes*)
let pi = 4. *. atan 1.
;;

(* d'apres google maps : 60px = 10m *)

let r1,r2,r3 = 100.,40.,200.

let lcer = 2.*.pi*.r2/.3.
;;
let lint = 2.*.pi*.r1/.5.
;;
let lext = 2.*.pi*.r3/.5. -. lcer /. 2.
;;


(*creer_manege : float -> float -> float -> graphe*graphe
creer_manege lext lcer lint
```

```
(*cree le manege et le rond-point associe pour les tailles de section donnees*)

let creer_manege lext lcer lint h =

  let mag = creer_graphe 20 in
  let rop = creer_graphe 20 in

  for i = 0 to 4 do
    let sor = 4*i in
    let pst = 4*i + 1 in
    let pre = 4*i + 3 in
    let ent = 4*i + 2 in
    let nxt = (4*i + 7) mod 20 in
    let ant = (4*i + 6) mod 20 in

    lier rop sor pst (lcer +. h);
    lier rop pre sor lcer;
    lier rop pre pst lcer;
    lier rop pst nxt lext;

    lier mag sor pst (lcer +. h);
    lier mag pre sor lcer;
    lier mag pst ent lcer;
    lier mag ent pre lcer;
    lier mag pre pst lcer;
    lier mag pst nxt (lext +. h);
    lier mag ant ent (lint +. h);

  done;
  mag,rop
;;

(*info_circ: graphe -> graphe
extrait d'un graphe de manege les lignes et les colonnes correspondant
a des entrees-sorties*)

let info_circ w =
  Array.init 5 (fun i -> Array.init 5 (fun j -> w.(i*4).(j*4)))

;;

let print_matrix mx =
  let u,v = Array.length mx,Array.length mx.(0) in
  for i = 0 to u-1 do
    print_int (i*4) ; print_string " & ";
    for j = 0 to v-2 do
      print_int (truncate (mx.(i).(j) *. 100.));
      print_string " & "
    done;
    print_int (truncate (mx.(i).(v-1) *. 100.));
```

```
    print_string " \\\\\n";
  done
;;


(*(++) float array array -> float array array -> float array array
somme de deux matrices de flottants
precondition : les deux matrices sont de memes dimensions*)



let ( ++ ) =
  Array.map2 (fun t1 t2 -> Array.map2 (fun x y -> x +. y) t1 t2)
;;

let ( //.) =
  Array.map2 (fun t1 t2 -> Array.map2 (fun x y -> if abs_float y > 1.e-10 then x /. y
      else 1.) t1 t2)
;;


let opp  =
  Array.map (fun t -> Array.map (fun x -> -.x) t)
;;

let alp =  [|
    [|2.;3.|];
    [|4.;5.|];
  |]
;;

(*h est un poids handicap pour les give-way*)

let diff h =
  let mag,rop = creer_manege lext lcer lint h in
  opp (info_circ (floyd_warshall mag)) ++ (info_circ (floyd_warshall rop))
;;

let rprt h=
  let mag,rop = creer_manege lext lcer lint h in
  (info_circ (floyd_warshall mag)) //. (info_circ (floyd_warshall rop))

(*somme des elements pour une matrice de flottants*)

let total m =
  Array.fold_left
    (fun x t -> x +. Array.fold_left (fun x y -> x +. y) 0. t) 0. m
;;

let dep_h i j =
```

```
  for h=i to j do
    print_int h; print_char '\t';
    print_float (total (diff (float_of_int h)));
    print_newline ();
  done
;;


let dep_h_rprt i j =
  for h=i to j do
    print_int h; print_char '\t';
    print_float (total (rprt (float_of_int h))/.25.);
    print_newline ();
  done
;;

(*fin de la partie FW, maintenant notion de section critique*)
(*transforme le graphe pondere represente par une matrice
en graphe oriente non-pondere represente par listes d'adjacences*)

let adj_of_mat g =
  let n = taille g in
  let s = Array.make n [] in
  for i=0 to n-1 do
    for j=n-1 downto 0 do
      if g.(i).(j) < infinity then
        s.(i) <- j::s.(i)
    done
  done;
  s
;;

(*pas une tres bonne idee : on a besoin de supprimer des arcs*)

let acces g a b =
  let n = taille g in
  let vus = Array.make n false in
  let rec loop v =
    let snv = List.filter (fun v -> not vus.(v))  (succs g v) in
    vus.(v) <- true;
    v = b || List.exists loop snv
  in
  loop a
;;

(*pres_conn graphe -> int -> int
pres_conn g a b revoie true ssi la suppression de l'arete (a->b) dans g
preserve les composantes connexes fortes de g
PREC : l'arete a->b existe
cf mail Judi*)
```

```
let pres_conn g a b =
  let arc = g.(a).(b) in
  suppr g a b;
  let s = acces g a b in
  lier g a b arc;
  s
;;

(*arcs_critiques : graphe -> int*int list
retourne la liste des arcs dont la supression entrainerait la
modification de la connexite du graphe *)

let arcs_critiques g =
  let n = taille g in
  let s = ref [] in
  for i = n-1 downto 0 do
    List.iter (fun j -> if not (pres_conn g i j)
                        then s := (i,j)::!s)
              (succs g i)
  done;
  !s
;;
```

3.2 Automate cellulaire

3.2.1 objets

Module définissant les objets de base (section, intersection, voiture...) et implantant le modèle de Nagel-Schreckenberg sur des sections

objets.mli

```
val p : float ref(*la probabilite de ralentissement de la voiture*);;

val redm : int ref (*temps de redemarrage apres un arret*);;

type distr (*une entree, une sortie ou une intersection*);;

type intersection (*une intersection entre deux sections d'autoroute*);;

type voiture (*une voiture*);;

type section (*un fragment de route*);;

(*creer les objets*)

val dums : section;;

val creer_section : int -> int -> section;;
```

```
(*creer_section sz ms cree une section vide de taille sz et de vitesse max ms*)

val creer_inter : int -> ( (voiture * int * section * section) option array -> section
    array -> section list array -> int -> unit) -> distr;;
(*creer_inter n f cree une intersection a n entrees
dont le comportement est decrit par la fonction de comportement f*)

val creer_spawn : unit -> distr;;
(*cree une distribution Spawn*)

val creer_sortie : string -> distr;;
(*creer_sortie st cree une sortie de nom st*)

val creer_voiture : int -> section list -> string -> voiture;;
(*creer_voiture s d ncree une voiture de vitesse s et d'itineraire d nommee n*)

val itere_voitures : (voiture -> unit) -> section -> unit;;
(*itere_voitures f s : itere f sur l'ensemble des voitures de s*)

(*questionner les objets*)

val who : voiture -> string;;

val panneau : section -> int (*un panneau indicateur de vitesse pour la section*);;

val tsec : section -> int (*la taille de la section*);;

val observer : section -> voiture option array (*etudie une section de route*);;

val survoler : section -> bool array;;
(*un tableau representant la section : true s'il y a une voiture*)

val radar : voiture -> int (*simule un radar donne la vitesse de la voiture*);;

val firstcar : section -> int ;;
(*regarde ou est la premiere voiture d'une section*)

val patients : distr -> (voiture * int * section * section) option array ;;
(*retourne vitesse et distance de la voiture qui patiente dans l'intersection
en entree (souleve une erreur sinon)*)

val nombre_voitures : section -> int ;;
(*retourne le nombre de voitures presentes dans la section en entree*)

val densite : section list -> float;;
(*retourne la densite de l'ensemble des sections en entree*)

val vitesse_moy : section list -> float;;
(*retourne la vitesse moyenne des voitures de l'ensemble des sections en entree*)
```

```
val flot_moy : section list -> float;;
(*retourne le flot moyen de voitures dans l'ensemble des sections en entree*)

(*manipuler les objets*)

val ajouter_sortie : distr -> int -> section -> unit;;
(*ajouter sortie d p sec si d est une intersection i : autorise les voitures venant de i
    .ent.(p) a sortir par sec*)

val lier : distr -> int -> distr -> int -> section -> unit;;
(*lier d1 p1 d2 p2 sec fait le lien de la distribution d1 a la distribution d2 par la
    route sec en positionant sec comme parmi les sorties possibles de d1 et comme entree
     no p2 de d2*)

val ajcar : section -> voiture -> int -> unit;;
(*ajcar s c p ajoute la voiture c dans s a la position p s'il n'y en a pas
deja une*)
val ajcar_sil : section -> voiture -> int -> unit;;
(*une version moins verbeuse de ajcar : ne souleve pas d'echec si
on met une voiture sur une autre, ne fait rien a la place*)

(*iterations de l'automate*)

val brake : voiture -> unit;;
(*brake c : fait freiner la voiture*)

val accel : voiture -> int -> unit;;
(*accel c vmax fait accelerer la voiture, dans la limite de vmax*)

val desc : voiture -> int -> unit;;
(*desc c dsec fait ralentir la voiture c en-dessous de dsec
eviter les collisions*)

val descrand : voiture -> float -> unit;;
(*descrand c p fait ralentir la voiture c avec une proba p*)

val move : int -> section -> unit;;
(*si une voiture se trouve dans la section s a la position k,
fait avancer dans s la voiture s.(k)*)

val ajdir : voiture -> section -> unit;;
(*ajdir v s ajoute la direction s a la voiture v *)

val increment : section -> unit;;
(*increment s itere s selon Nagel-Schreckenberg*)

val traverser : distr -> int -> unit;;
  (*traverser d t itere d selon sa fonction de comportement au temps t*)
```

objets.ml

```
(*les divers objets qui seront necessaires pour le programme*)


let p = ref 0.
;;
let redm = ref 1
;;

type distr =
        |Spawn
        |Quit of string
        |Int of intersection

 and intersection = {mutable ent:section array;mutable sor:section list array;
                     mutable att:(voiture*int*section*section) option array; transf: (
                         voiture*int*section*section) option array-> section array ->
                         section list array -> int -> unit}

(*si att.(x) contient (v,d,e,s) c'est qu'une voiture v se trouvant a d (d>0
de l'intersection, venant de e, allant vers s ou x est la case de att reservee a e*)

 and voiture = {mutable spd:int;
                mutable dir:section list;
                nom:string;
                mutable arret: int}

 and section = {mutable pre:distr;data:voiture option array;
                           maxspd:int;mutable post:distr}
;;


(*creer les objets*)

let creer_section sz ms =
        let d = Array.make sz None in
        {pre=Spawn;post=Quit("");data=d;maxspd=ms}
;;

let dums = creer_section 0 0
;;


let creer_inter n f = Int({ent=Array.make n dums;sor=Array.make n [];att=Array.make n
    None;transf=f})
;;

let creer_spawn () = Spawn
;;
```

```
let creer_sortie s = Quit(s)
;;

let creer_voiture s d n = {spd=s;dir=d;nom=n;arret=0}
;;

let itere_voitures f sec =
  let g sc =
    match
      sc
    with
    |Some(c) -> f c;
    |None -> ();
  in
  Array.iter g sec.data
;;

(*questionner les objets*)
let who c = c.nom
;;

let rec somme f ls =
  match ls with
  |[] -> 0
  |x::r -> (f x) + somme f r
;;

let panneau sec = sec.maxspd
;;

let tsec sec = Array.length sec.data
;;

let observer sec = sec.data
;;
let survoler sec =
  let f x =
    match
      x
    with
    |Some(_) -> true
    |None -> false
  in
  Array.map f sec.data
;;


let radar c = c.spd
;;
let firstcar sec =
```

```
        let n = Array.length sec.data in
        let act = ref 0 in
        while !act < n && sec.data.(!act) = None do
                incr act
        done;
        !act
;;

let patients d =
  match
    d
  with
  |Int(isec) -> isec.att
  |_ -> failwith "patients : personne ne patiente pour entrer/sortir"
;;
(*etudier les objets : outils de mesure*)

let nombre_voitures sec =
  let co = ref 0 in
  let f sc =
    match
      sc
    with
    |Some(c) -> incr co
    |None -> ()
  in
  Array.iter f sec.data;
  !co
;;

let nombre_voitures sec =
  let c = ref 0 in
  let f x = incr c in
  itere_voitures f sec;
  !c
;;

let somme_vitesses sec =
  let c = ref 0 in
  let f x = c:= radar x + !c in
  itere_voitures f sec;
  !c

let densite lsec =
  let nb = somme nombre_voitures lsec in
  let taille = somme tsec lsec in
  if taille = 0
  then 0.
  else
    (float_of_int nb) /. (float_of_int taille)
```

```
;;

let vitesse_moy lsec =
  let nb = somme nombre_voitures lsec in
  let vts = somme somme_vitesses lsec in
  if nb = 0
  then
    0.
  else
    (float_of_int vts) /. (float_of_int nb)
;;

let flot_moy lsec =
  (densite lsec) *. (vitesse_moy lsec)
;;


(*manipuler les objets*)

let ajouter_sortie d p sec =
  match
    d
  with
  |Spawn -> ()
  |Quit(_) -> failwith "ajouter_sortie : entrer par une sortie"
  |Int(i1) -> i1.sor.(p) <- sec::i1.sor.(p)
;;

let lier d1 p1 d2 p2 sec =
        sec.pre <- d1;
        sec.post <- d2;
        ajouter_sortie d1 p1 sec;
        begin
          match d2 with
          |Spawn -> failwith "lier : sortir par une entree"
          |Quit(_) -> ()
          |Int(i2) -> i2.ent.(p2) <- sec
        end
;;


let ajcar sec c pos =
        match sec.data.(pos) with
        |Some(_) -> failwith "ajcar : apparition d'une voiture sur une autre"
        |None -> sec.data.(pos) <- Some(c)

;;

let ajcar_sil sec c pos =
  try
```

```
    ajcar sec c pos
  with
  |Failure "ajcar : apparition d'une voiture sur une autre"
   -> ()
(*iterations de l'automate*)

let brake c =
  c.spd <- 0;
  c.arret <- !redm;
;;

let accel c vmax =
  if c.arret > 0
  then c.arret <- c.arret - 1
  else c.spd <- min (c.spd + 1) vmax
;;


let desc c dsec =
  let vo = c.spd in
  c.spd <- min c.spd (dsec - 1);
  if c.spd = 0 && vo > 0
  then c.arret <- !redm
;;

let descrand c p =
  let vo = c.spd in
  if Random.float 1. <= p
  then c.spd <- max 0 (c.spd - 1);
  if c.spd = 0 && vo > 0
  then c.arret <- !redm
;;

let move p sec =
  match sec.data.(p) with
  |None -> failwith "move :trying to move no car"
  |Some(c) ->
    if
      c.spd > 0
    then
      begin
        let d = p + c.spd in
        match sec.data.(d) with
        |Some(_) -> failwith "move : collision"
        |None ->
          begin
            sec.data.(d) <- Some(c);
            sec.data.(p) <- None;
          end
      end
```

```
    else ()
;;

let ajdir c s =
  c.dir <- s::c.dir
;;

let indexq x ar =
  let n = Array.length ar in
  let r = ref 0 in
  while !r < n && ar.(!r) != x
  do incr r
  done;
  if !r = n
  then raise Not_found
  else !r
;;



(*increment: section -> unit
met a jour la section avec les regles de l'automate cellulaire*)

let increment sec =
  let next = ref 0 in
  let act = ref 0 in
  let n = Array.length sec.data in
  act := firstcar sec;
  if !act < n then (*la section n'est pas vide*)
    begin
      next := !act+1;
      while !next < n do
        match sec.data.(!next) with
        |None-> incr next
        |Some(vdev) ->
          match sec.data.(!act) with
          |None -> failwith "increment : empty car"
          |Some(vder) ->
            (*on vient de determiner la voiture actuelle et la voiture de devant*)
            accel vder sec.maxspd;
            desc vder (!next - !act);
            descrand vder !p;
            move !act sec;
            (*la voiture suivante devient voiture actuelle*)
            act := !next;
            next := !act + 1;

      done;
    (*sec.data.(!act) contient la derniere voiture de la section qui peut
     quitter cette derinere lors de l'iteration*)
```

```
      match sec.data.(!act) with
      |None -> failwith "increment : empty lastcar"
      |Some(c) ->
        begin
          accel c sec.maxspd;
          if !act + c.spd < n
          then move !act sec
          else
            match c.dir,sec.post
            with
            |_,Spawn -> failwith "increment : arriver sur un depart"
            |_,Quit(s) ->
              begin
                (*print_string s;
                print_newline ();*)
                sec.data.(!act) <- None;
              end
            |[],_ -> failwith "increment : objectiveless car"
            |q::r,Int(inter) ->
              begin
                let pat = c,n-(!act),sec,q in
                let v = indexq sec inter.ent in
                inter.att.(v) <- Some(pat);
                c.dir <- r;
                sec.data.(!act) <- None;
              end
        end
    end
;;

let traverser dst t =
        match dst with
        |Int(isec) ->
          isec.transf isec.att isec.ent isec.sor t;
          Array.fill isec.att 0 (Array.length isec.att) None;
        |_-> failwith "traverser : passer a travers d'un debut ou d'une fin"
;;
```

3.2.2 comportements

Module pour les comportements aux intersections

comportements.mli

```
open Objets;;

(*un comportement est une fonction gerant la traversee d'une intersection *)

val internasch : voiture -> int -> section -> section -> unit;;
(*Applique NaSch a l'intersection*)
```

```
val corresp : section -> section -> section array -> section list array -> int -> bool;;
(*corresp e s ent sor i renvoie true si la voiture se trouvant dans l'intersection Inter
a la position  Inter.att.(i) a pu arriver a cette position, et peut en repartir, la ou
elle veut aller*)

val passif : (voiture * int * section * section) option array -> section array ->
    section list array -> int -> unit;;
(*le comportement vide, en quelque sorte*)

val onemany : (voiture * int * section * section) option array -> section array ->
    section list array -> int -> unit;;

(*les comportements faciles : une voiture en entree, une ou plusieurs en sortie*)

val prioabs : (voiture * int * section * section) option array -> section array ->
    section list array -> int -> unit;;

(*comportement twomany : deux entrees plusieurs sorties, l'entree 0 a la priorite
    absolue sur l'entree 0 :
s'il y a une voiture de 0 qui veut passer alors la voiture de 1 ne bouge pas et la
    laisse passer*)

val priodyn : (voiture * int * section * section) option array -> section array ->
    section list array -> int -> unit;;
(*priorite 2 entrees du type decrit par Rui-Xiong*)

val feux : int -> int -> int ->  (voiture * int * section * section) option array ->
    section array -> section list array -> int -> unit;;
(*feux dur1 dur2 ph retourne un comportement de feux : att.(0) passe pendant dur1, puis
    att.(1) pendant dur2 en commencant par ph*)


(*comportements d'apparition : fonctions gerant la frequence d'apparition des voitures*)

val pente : float -> float -> int -> int -> float;;
  (*pente b e s t retourne pour un automate pendant s etapes, une apparition de voitures
      croissante en fonction de t allant de b a e*)
```

comportements.ml

```
(*divers comportements aux intersections*)
open Objets
;;
let amemq x t = Array.fold_left (fun b a -> (a == x)||b) false t
;;

let corresp e s ent sor i =
  List.memq s sor.(i) && ent.(i) == e
```

```
(*modelise Nagel-Schreckenberg pour un carrefour*)

let refuse c e s =
  brake c;
  ajdir c s;
  ajcar e c (tsec e - 1);

;;

let internasch c d e s =
  accel c ( panneau s);
  desc c ( firstcar s + d);
  descrand c !p;

  let pos = radar c - d in
  if
    pos < 0
  then
    refuse c e s
  else
    ajcar s c pos
;;

let onemany att ent sor t =
  match
    att.(0)
  with
  |Some(c,d,e,s) ->
    if
      corresp e s ent sor 0
    then
      internasch c d e s
    else
      failwith "onemany_error : sorties non correspondantes"
  |None -> ()

;;

let twomany cpm att ent sor t =
  match
    att.(0), att.(1)
  with
  |None,None -> ()
  |Some(c,d,e,s),None ->
    if corresp e s ent sor 0
    then internasch c d e s
    else failwith "twomany_error 1: I/O non correspondantes"
  |None,Some(c,d,e,s) ->
    if corresp e s ent sor 1
    then internasch c d e s
```

```
    else failwith "twomany_error 2: I/O"
(*le cas interessant : deux voitures cherchent a traverser en meme temps*)
  |Some(cp,dp,ep,sp),Some(cl,dl,el,sl) ->
    if  corresp ep sp ent sor 0 && corresp el sl ent sor 1
    then
      cpm cp dp ep sp cl dl el sl
    else failwith "prioabs_error 3 :I/O"


let absolu cp dp ep sp cl dl el sl =
  internasch cp dp ep sp;
  refuse cl el sl;
;;

let dynamique cp dp ep sp cl dl el sl =
  let temps c d s = (*le temps necessaire pour arriver a l'intersection*)
    (*denominateur : estimation du resultat Nasch sans alea*)
    float_of_int d /. float_of_int (min (panneau s) (min (d + firstcar s) (radar c +1)))
  in
  let priop cp dp sp cl dl sl =
    let tp,tl = (temps cp dp sp),(temps cl dl sl) in
    tp < tl || (tp = tl && dp <= dl)
  in
  if priop cp dp sp cl dl sl
  then
    begin
      internasch cp dp ep sp;
      internasch cl dl el sl;
    end
  else
    begin
      internasch cl dl el sl;
      internasch cp dp ep sp
    end
;;
let prioabs att ent sor t = twomany absolu att ent sor t
;;

let priodyn att ent sor t = twomany dynamique att ent sor t
;;

let feux dur1 dur2 ph att ent sor t =
  let t' = (t - ph) mod (dur1 + dur2) in
  let passer vert i =
    match
      att.(i)
    with
    |None -> ()
    |Some(c,d,e,s) ->
      if corresp e s ent sor i
```

```
      then
        if
          vert
        then
          internasch c d e s
        else
          refuse c e s
      else
        failwith "feux error : I/O"
  in
  passer (t' < dur1) 0;
  passer (t' >= dur1) 1;

;;

let passif att ent sor t= ()
;;


  (*comportements d'apparition de voitures*)

let pente b e s t =
  b +. float_of_int t *. (e -. b) /. float_of_int s
```

3.2.3 plateau

Module construisant et itérant les automates

plateau.mli

```
open Objets

type plateau;;
(*un plateau de jeu contenant sections et distributions
i.e. un automate*)

type rond_point;;

val rp_ent : rond_point -> section array;;

val rp_sor : rond_point -> section array;;

val rp_ron : rond_point -> section array;;

val rp_dis : rond_point -> distr array;;
(*acceder aux composantes de l'enregistrement*)

val creer_rond_point: int -> int -> int -> int -> int -> ( (voiture * int * section *
    section) option array -> section array -> section list array -> int -> unit)
                      -> rond_point ;;
```

```
(*creer_rond_point n l p vmi vme cmp cree un rond-point a n entrees/sorties
de taille l, un anneau interne de taille n*p, des intersection de comportement
de comportement cmp, avec une vitesse max interne(resp externe) de vmi (resp vma)*)

val faire_itin: rond_point -> int -> int -> section list -> section list;;
(*faire_itin rp ne ns fin
complete l'itineraire fin en lui ajoutant devant l'itineraire de l'entree ne a la sortie
    ns*)

val pl_add_rp : plateau -> rond_point -> unit;;
(*pl_add_rp pl rp ajoute le rond-point rp au plateau pl*)

val construire: section list -> distr list -> (int -> unit) list -> plateau;;
(*construire s d sp construit le plateau p avec
une liste de sections s
une liste de distributions d
une liste d'evenements (pour l'instant spawn) dependant du temps t*)

val iterer : plateau -> int -> unit;;
  (*gere les evenements
   itere l'ensemble des sections
   puis des intersections du plateau
   si l'horloge vaut t*)

val afficher : plateau -> unit;;
  (*affiche toutes les sections du plateau*)

val imager : plateau -> (section*((int*int)*float*Graphics.color)) list -> unit;;

val spawn_car : int -> int -> int -> int -> section -> section list ->int -> unit
(*spawn_car per ph v pos sec itin t
fait apparaitre une voiture dans sec a la vitesse v position pos
d'itineraire itin si c'est le moment de faire apparaitre une voiture
i.e. t == ph mod per*)

val rnd_spawn_car : float -> int -> int -> section -> section list ->int -> unit;;
(*rnd_spawn_car p v pos sec itin t version stochastique de spawn_car : fait apparaitre
    une voiture
avec une probabilite de p *)

val spawn_prog : (int -> float) -> int -> int -> section -> section list -> int -> unit
    ;;
  (*spawn_prog f ... effectue un rnd_spawn_car avec une proba de (f t)
permet de faire une apparition croissante des éléments*)

val faire : plateau -> int -> int -> (plateau -> int -> unit) -> (plateau -> int -> unit
    ) -> (plateau -> int -> unit) -> (plateau -> int -> unit) -> unit;;
(*faire p i f dbt bent bsor fin : trivial par induction sur les termes du lambda calcul
*)
```

```
val silence : plateau -> int -> int -> unit;;
  (*joue des iterations de l'automate sans rien afficher*)

val jouer : plateau -> int -> int -> unit;;
  (*jouer p i f joue (f-i) iterations de l'automate
   en les affichant a chaque tour, l'horloge
commencant a i inclus et finissant a f exclus*)

val animer : plateau -> int -> int -> (section*((int*int)*float*Graphics.color)) list ->
     unit;;
  (*animer p i f gr anime a l'aide du module Graphics en representant le plateau decrit
      par gr*)


val modeliser : plateau -> int -> int -> (plateau -> float list) -> unit;;
  (*modeliser p i f info joue (f-i) iterations de l'automate, et affiche les
      informations collectees par info a chaque tour, espacees d'un retour a la ligne*)

val sauvegarder : plateau -> int -> int -> (plateau -> float list) -> string -> unit;;
  (*idem modeliser, mais au lieu de les afficher, les sauve dans le fichier donne en
      dernier argument*)
```

plateau.ml

```
open Objets
;;
open Affichage
;;
open Comportements
;;

type plateau =
  {mutable secte:section list;
   mutable distrs:distr list;
   mutable events:
             (int -> unit) list
  }
;;

type rond_point =
  {ents: section array; sors : section array; rond: section array;
   diss: distr array}

let rp_ent rp = rp.ents
;;

let rp_sor rp = rp.sors
;;

let rp_ron rp = rp.rond
;;
```

```
let rp_dis rp = rp.diss
;;


let creer_rond_point n l p vmi vme cmp =

  let sp = creer_spawn () in
  let dumi = creer_inter 0 passif in
  let isec = Array.make n dumi in

  let dums = creer_section 0 0 in
  let ent = Array.make n dums in
  let sor = Array.make n dums in
  let ron = Array.make n dums in

  (*creation des sections et des intersections*)
  for i=0 to (n-1) do
    ent.(i) <- creer_section l vme;
    sor.(i) <- creer_section l vme;
    ron.(i) <- creer_section p vmi;
    isec.(i) <- creer_inter 2 cmp;
  done;

(*liaisons internes*)
  for i=0 to n-2 do
    lier isec.(i) 0 isec.(i+1) 0 ron.(i)
  done;
  lier isec.(n-1) 0 isec.(0) 0 ron.(n-1);

(*liaisons externes*)
  for i=0 to n-1 do
    lier isec.(i) 0 (creer_sortie "") 1 sor.(i) ;
    lier sp 1 isec.(i) 1 ent.(i);
    ajouter_sortie isec.(i) 1 ron.(i)
  done;

    {ents=ent;
     sors=sor;
     rond=ron;
     diss=isec}
;;

let faire_itin rp ne ns fin =
  let inner = rp.rond in
  let n = Array.length inner in
  let rec boucle e s li =
    if e == s
    then li
    else if e == n
```

```
    then boucle 0 s li
    else
      inner.(e)::boucle (e+1) s li
  in
  rp.ents.(ne)::boucle ne ns (rp.sors.(ns)::fin)

let pl_add_rp pl rp =
  pl.sects <- List.flatten (List.map Array.to_list [rp.ents;rp.sors;rp.rond]) @ pl.sects
    ;
  pl.distrs <- Array.to_list rp.diss @ pl.distrs;
;;

let construire s d ev = {sects=s;distrs=d;events=ev};;

let iterer p t =
  List.iter (fun f -> f t) p.events;
  List.iter increment p.sects;
  List.iter (fun x -> traverser x t) p.distrs;

;;
let afficher p =
  List.iter print_section p.sects;
;;

let imager p grcr =
  let f sec =
    try
      let (x,y),th,col = List.assoc sec grcr in
      draw_section sec (x,y) th col
    with
    |Not_found -> ()
  in
  List.iter f p.sects

let spawn_car per ph v pos sec itin t =
  if
    (t-ph) mod per = 0
  then
    ajcar_sil sec (creer_voiture v itin "truc") pos
  else
    ()
;;

let rnd_spawn_car prob v pos sec itin t =
  let r = Random.float 1. in
  if
    r < prob
  then
    ajcar_sil sec (creer_voiture v itin "truc") pos
  ;
```

```
    ignore t
;;

let spawn_prog f v pos sec itin t =
  rnd_spawn_car (f t) v pos sec itin t
;;


let faire p i f dbt bent bsor fin =
  Random.self_init ();
  dbt p;
  for t = i to (f-1) do
    bent p t;
    iterer p t;
    bsor p t;
  done;
  fin p f;
;;

let rien (p:plateau) t = ();;

let silence p i f =
  faire p i f rien rien rien rien
;;

let jouer p i f =
  let aff p t =
    ignore t;
    afficher p;
    print_newline ()
  in
  faire p i f rien aff rien aff
;;

let patience d =
  let tf = (Unix.gettimeofday () )+. d in
  let rec loop () =
    let t = (Unix.gettimeofday ()) in
    if t < tf then loop ()
      (*begin
        print_string "dodo";
        print_newline ();
        Unix.sleep (max 1 (int_of_float (tf -. t)));
        print_string "reveil";
        print_newline ();
        loop ();
      end*)
  in loop ()
;;
```

```
let animer p i f grcr =
  let bent p t =
    ignore t;
    Graphics.clear_graph ();
    imager p grcr;
    Graphics.synchronize ();
    patience 0.05;
    (*ignore (Graphics.wait_next_event [Graphics.Key_pressed]);  *)
  in
  faire p i f rien bent rien rien
;;


let modeliser p i f info =
  let bent p t=
    print_info (info p);
    print_newline ();
  in
  faire p i f rien bent rien rien
;;

let sauvegarder p i f info fout =
  let out = open_out fout in
  let bent p t =
    save_info (info p) out;
    output_string out "\n";
  in
  faire p i f rien bent rien (fun p f -> close_out out)
```

3.2.4 affichage

Module donnant des informations sur les sections `affichage.mli`

```
(* les affichages sont suivis d'un print_newline*)
val scale : float;;
val print_section : Objets.section -> unit (*affiche une section*);;

(*affiche les voitures en attente dans la section*)
val preview_section : Objets.section -> unit;;
(*affiche grossierement une section*)
val info_fdmal : Objets.section list -> float list;;
(*donne les informations du graphe fondamental de la liste de sections :
densite <vitesse> <flux>*)

val print_info : float list -> unit;;
(*print_info data affiche les infos donnees par data *)

val save_info : float list -> out_channel -> unit;;
(*sauve ces infos dans un fichier*)
```

```
(*petite interface graphique des familles*)
val pi : float;;

val init : int -> int -> unit;;
(*initialise l'interface*)

val draw_section : Objets.section -> int*int -> float -> Graphics.color -> unit;;
(*dessine une section*)
```

affichage.ml

```
open Objets

;;
let print_car c =
  print_int (radar c)

let print_section sec =
  let d =  observer sec in
  for i = 0 to (Array.length d) - 1 do
    match
      d.(i)
    with
    |None -> print_string "."
    |Some(c) -> print_int ( radar c)
  done;
  print_newline ();
;;

let preview_section sec =
  let d = observer sec in
  for i=0 to (Array.length d) - 1 do
    match
      d.(i)
    with
    |None -> print_string " "
    |Some(c) -> print_string "."
  done;
  print_newline ();
;;

let sep = "\t"
;;

let info_fdmal lsec =
  let d = densite lsec in
  let v = vitesse_moy lsec in
  [d;v;d*.v(*J*)]
;;
```

```
let print_info data =
  List.iter (fun x -> print_float x; print_string sep) data

;;

let save_info data out =
  List.iter
    (fun x -> output_string out (string_of_float x); output_string out sep)
    data
;;

open Graphics
;;
let foi = float_of_int
;;
let trc = truncate
;;
let pi = 2. *. asin 1.
;;
let scale = 10. (*pix/m sachant dx = 7m *)

let init m n =
  open_graph (" "^string_of_int m^"x"^string_of_int n);
  auto_synchronize false;
;;
(*on travaille uniquement avec des flottants,
la traduction en entiers se fait juste avant
l'appel des fonctions du module Graphics
les valeurs tronquees seront precedees d'un p (comme pixel)
Au fait, on compte aussi en radians*)

let rotation (cx,cy) the (x,y) =
  let th = (-.the) in
  let nx = (x-.cx)*.(cos th) +. (y-.cy)*.(sin th) +. cx in
  let ny = (x-.cx)*.(-1.)*.(sin th) +. (y-.cy)*.(cos th) +. cy in
  (nx,ny)
;;


let case_droite (x,y) =
  let d = (scale /. 2.) in
  [|x-.d,y-.d;x+.d,y-.d;x+.d,y+.d;x-.d,y+.d|]
;;

let case (x,y) th =
  Array.map (rotation (x,y) th) (case_droite (x,y))
;;

let sec_contour n (x,y) th =
  let fn = foi n in
```

```
  let lex,bex = (x -. scale/.2.),(y -. scale/.2.) in
  let rex,tex = (lex +. fn *. scale),(bex +. scale) in
  let cont = [|lex,bex;
                 lex,tex;
                 rex,tex;
                 rex,bex;
               |] in
  let barre i =
    let absc = lex +. (foi i +.1.) *. scale in
    (absc,bex,absc,tex)
  in
  let dec = Array.init (n-1) barre in

  let contour = Array.map (rotation (x,y) th) cont in
  let f (xa,ya,x',y') =
    let ((xr,yr),(xr',yr')) = rotation (x,y) th (xa,ya),
                              rotation (x,y) th (x',y')
    in
    (xr,yr,xr',yr')
  in
  let decoupe = Array.map f dec in
  contour, decoupe
;;

let sec_voit arr (x,y) th =
  let r = ref [] in
  let f i ex =
    if
      ex
    then
      r := rotation (x,y) th ((x +. (scale *. (foi i))),y)::!r
    else ()
  in
  Array.iteri f arr;
  !r
;;

let draw_voits vsec =
  let f (x,y) =
    fill_circle (trc x) (trc y) (trc (scale *. 0.4))
  in
  List.iter f vsec
;;

let draw_contour (cont,dec) =
  let tr (x,y) = (trc x, trc y) in
  let tr2 (x,y,x',y') = (trc x, trc y, trc x', trc y') in
  draw_poly (Array.map tr cont);
  draw_segments (Array.map tr2 dec);
;;
```

```
let draw_section sec (px,py) th col =
  let x,y = foi px,foi py in
  let n = tsec sec in
  set_color col;
  draw_contour (sec_contour n (x,y) th);
  draw_voits (sec_voit (survoler sec) (x,y) th);
;;
```

3.2.5 local

Module utilisé pour les expériences `local.ml`

```
(*etude locale de l'interet d'une intersection dynamique*)
open Objets
;;
open Affichage
;;
open Comportements
;;

(*infos :

d_p | v_p | J_p | d_np | v_np | J_np

faire dynamique et absolu
graphes interessants :
pente 0. 1.
pente .2 .5
*)


(*test two_one*)
let nit = 10000
;;

let modele fout tint =
  let ent0 = creer_section 50 5 in
  let ent1 = creer_section 50 5 in
  let sor = creer_section 10 5 in
  let a = creer_spawn () in
  let s = creer_sortie "s" in

  let bottle = creer_inter 2 tint in

  let grcr = [ent0,((20,80),0.,Graphics.red);
              ent1,((30,20),0.08,Graphics.blue);
              sor, ((550,80),0.,Graphics.black)] in
```

```
  let itin = [sor;sor] in

  lier a 0 bottle 0 ent0;
  lier a 0 bottle 1 ent1;
  lier bottle 0 s 0 sor;
  lier bottle 1 s 0 sor;

  let s = [ent0;ent1;sor] in

  (*let stable p t = p in*)
  let sp0 = Plateau.spawn_prog (pente 0. 1. nit) (panneau ent0) 0 ent0 itin in
  let sp1 = Plateau.spawn_car 5 0 (panneau ent1) 0 ent1 itin in
  let p = Plateau.construire s [bottle] [sp0;sp1] in
  let info t = ignore t;info_fdmal [ent0] @ info_fdmal [ent1] in


  let deb = false in
  if
    deb
  then
    begin
      init 800 100;
      Plateau.animer p 0 nit grcr;
    end
  else
    Plateau.sauvegarder p 0 nit info fout;
;;

let () =
  redm := 2;
  modele "localabs.dat" prioabs;
  redm := 2;
  modele "localdyn.dat" priodyn;
;;
```

3.3 figures

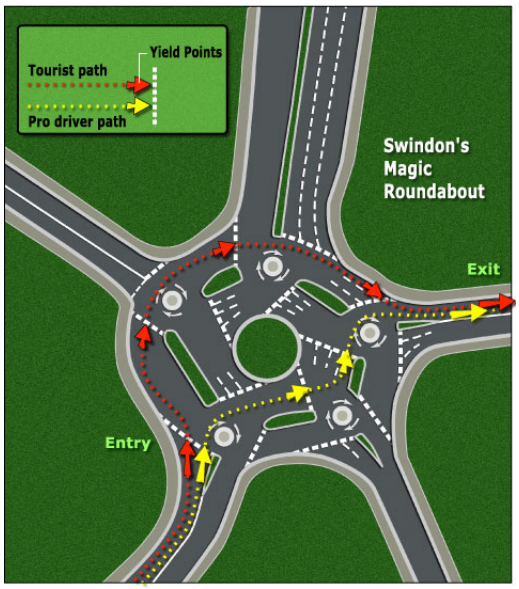
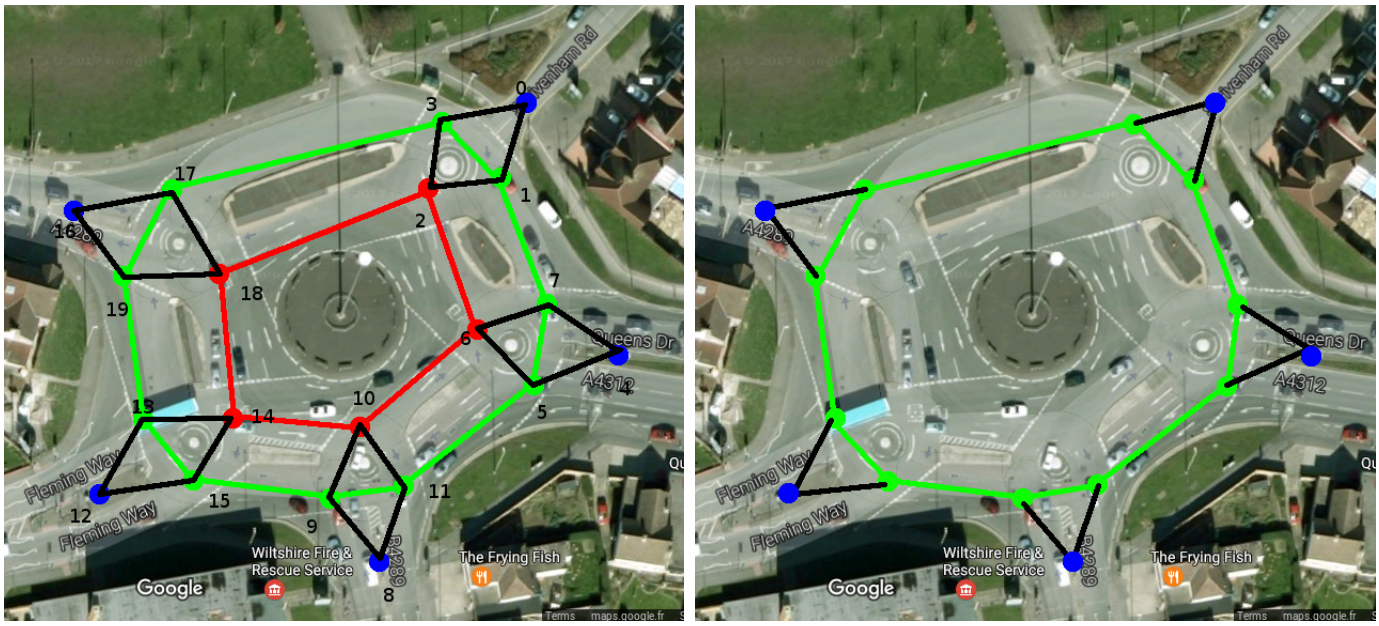


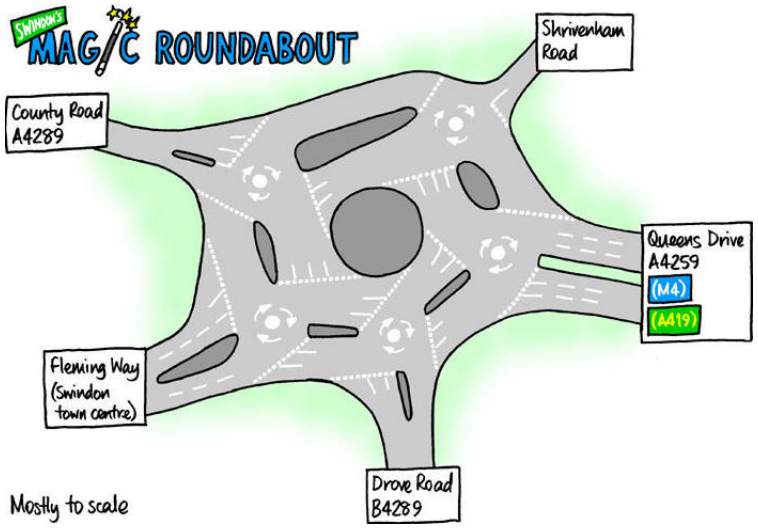
FIGURE 1 – itinéraires possibles



(a) le manège (b) le rond-point

FIGURE 2 – un graphe adapté au manège

FIGURE 3 – le manège enchanté



entrée	0	4	8	12	16
0	100	100	100	60	36
4	36	100	100	100	60
8	60	36	100	100	100
12	100	60	36	100	100
16	100	100	60	36	100

FIGURE 4 – Rapport entre la distance entrée-entrée pour le rond-point et le manège (en %)

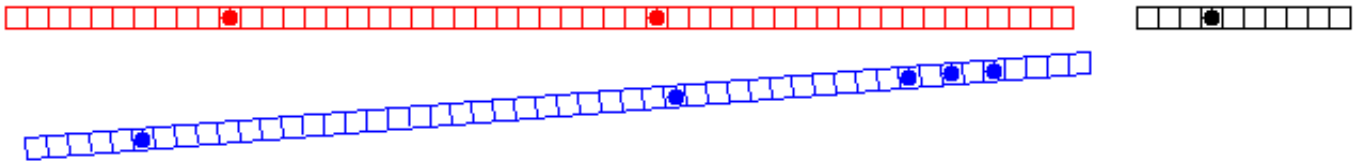


FIGURE 5 – Un exemple d'intersection 2-1

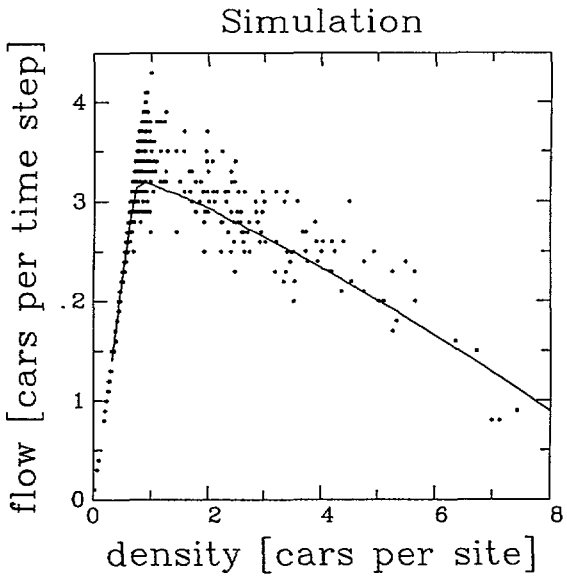


FIGURE 6 – Un exemple de diagramme fondamental, tiré de [1]

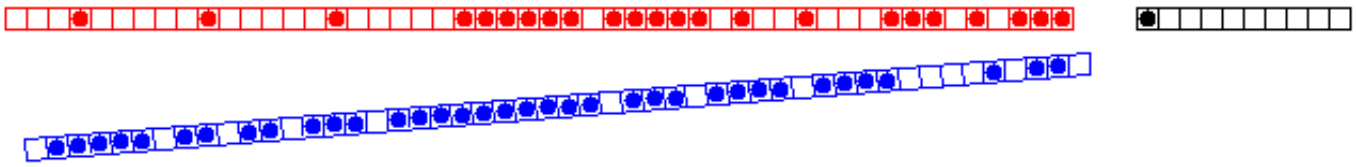


FIGURE 7 – début de saturation

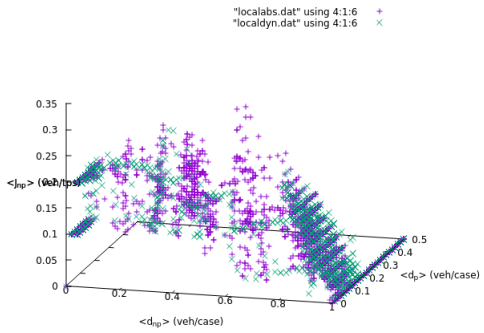


FIGURE 8 – Comparaison absolu (en violet) / dynamique (en vert)

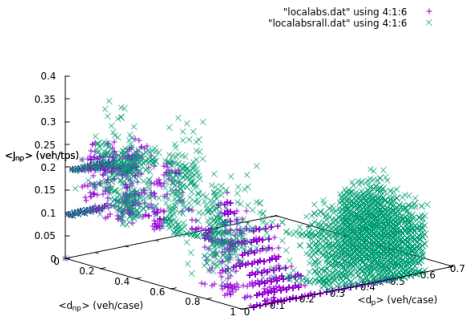


FIGURE 9 – Étude locale avec temps de démarrage
on observe une différence notable entre le modèle de priorité absolue sans temps de redémarrage (violet) et avec (vert)

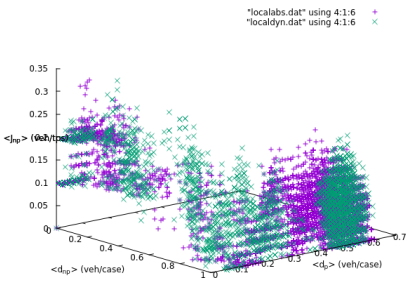


FIGURE 10 – Comparaison absolu-dynamique pour un temps de réaction de 1