Autour du Manège Enchanté

Antonin Dudermel

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 anti-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 1.

Une telle disposition permet une diversité des itinéraires possibles pour aller d'une entrée à une sortie (voir figure 2). 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 cellulaire, 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

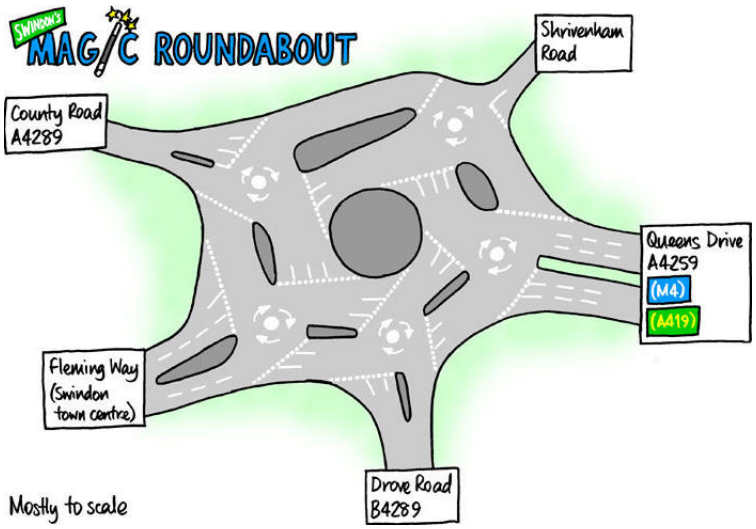
On peut aisément représenter un ensemble de routes par un graphe orienté pondéré : il suffit de considérer chaque intersection comme un sommet et chaque route comme une arête reliant une intersection à une autre de poids la longueur de la route. En appliquant ce principe au manège enchanté, on aboutit au graphe représenté par la figure 3a. 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 la distance entre les entrées-sorties, et les comparer entre les deux graphes. Le tableau 4 montre le rapport des distances du manège sur celles

FIGURE 1 – le manège enchanté



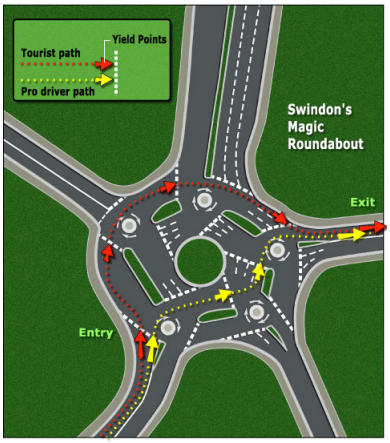
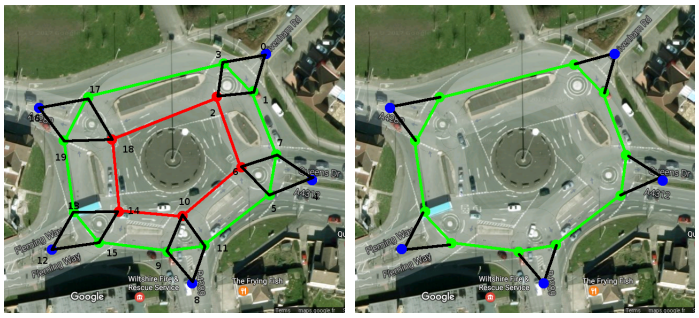


FIGURE 2 – itinéraires possibles



(a) le manège (b) le rond-point

FIGURE 3 – un graphe adapté au manège

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

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

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 2, 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* $G' = (V, E \setminus \{(a,b)\})$ *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 $G\prime$, 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 (on a $G\prime$), si b n'est pas accessible depuis a dans $G\prime$, alors on ajoute (a,b) à la liste des arcs critiques, puis on rajoute (a,b) à $G\prime$.

Le besoin de supprimer des arêtes pousse à choisir la structure de matrice d'adjacence (voir 2.5) 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 : $O(|V|^2)$.

Lors de l'exécution de l'algorithme, on effectue donc :

— Des opérations en temps constant
— Un parcours des arêtes de G, en un temps $O(|V|^2)$
— |E| fois :
  — des opérations en temps constant (suppression, ajout...)
  — Un parcours d'un graphe à (|V|) sommets, en temps $O(|V|^2$

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

2 Un Modèle par automate cellulaire

2.1 Automate cellulaire

2.2 Le Problème des intersections

2.3 Étude locale

2.4 Limites du modèle pour le manège

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