TIPE : Application de la théorie des graphes et des principes d'optimisation à la fluidification du traffic routier : Programme

Nodet Gaëtan (Candidat : 37757)

Definition

```
type arrete = {
   depart : int ;
   arrivee : int ;
   tempsMin : float ;
    acroissement : float
};;

type graphe = {
   nombre_noeuds : int ;
   nombre_arretes : int ;
   noeuds : int list array;
   arretes : arrete array;
};;

type parcours = (int list);;

type chemin = parcours * int ;;
```

Saisie du graphe

```
let n = [|
   [0;4];
   [1;2;6];
   [3;7;8];
   [9;11];
   [5;10];
|];;

let a = [|
   {depart = 0 ; arrivee = 1; tempsMin =  1000.; acroissement =3.};
   {depart = 1 ; arrivee = 2; tempsMin =   500.; acroissement =5.};
   {depart = 1 ; arrivee = 2; tempsMin =   600.; acroissement =7.};
   {depart = 2 ; arrivee = 3; tempsMin =   800.; acroissement =2.};
   {depart = 0 ; arrivee = 4; tempsMin =  1500.; acroissement =4.};
   {depart = 4 ; arrivee = 3; tempsMin =  2000.; acroissement =1.};

   {depart = 1 ; arrivee = 0; tempsMin =  1000.; acroissement =3.};
   {depart = 2 ; arrivee = 1; tempsMin =   500.; acroissement =5.};
   {depart = 2 ; arrivee = 1; tempsMin =   600.; acroissement =7.};
   {depart = 3 ; arrivee = 2; tempsMin =   800.; acroissement =2.};
   {depart = 4 ; arrivee = 0; tempsMin =  1500.; acroissement =4.};
   {depart = 3 ; arrivee = 4; tempsMin =  2000.; acroissement =1.};
|];;

let graphe = {nombre_noeuds = 5 ; nombre_arretes = 12 ; noeuds = n ; arretes =
   a};;
```

Algorithme de Djikstra

```
let rec ajoute chemin temps file = match file with
    | [] -> [(chemin,temps)]
    | (c,t)::f when t < temps -> (c,t)::(ajoute chemin temps f)
    | _ -> (chemin,temps)::file;;

let rec ajoute_chemins arretes (parcours,fin) temps file graphe = match arretes
    with
    | [] -> file
    | a::l -> let arrete = graphe.arretes.(a) in
       let f = ajoute (a::parcours,arrete.arrivee) (temps +. arrete.tempsMin)
           file in
          ajoute_chemins l (parcours,fin) temps f graphe;;

let rec djikstra_aux file non_visiter arrivee graphe= match file with
    | [] -> failwith "pas_de_chemin"
    | ((parcours,fin),temps)::f when fin = arrivee -> parcours
    | ((parcours,fin),temps)::f when non_visiter.(fin) -> (
       non_visiter.(fin) <- false ;
       djikstra_aux
          (ajoute_chemins graphe.noeuds.(fin) (parcours,fin) temps f graphe)
          non_visiter arrivee graphe)
    | ((parcours,fin),temps)::f -> djikstra_aux f non_visiter arrivee graphe;;

let djikstra depart arrivee graphe =
   let non_visiter = Array.make (graphe.nombre_noeuds) true in
      djikstra_aux [(([],depart),0.)] non_visiter arrivee graphe;;

djikstra 0 3 graphe;;
```

Algorithme de Djikstra modifié

```
let rec ajoute chemin temps non_visiter file = match file with
    | [] -> [(chemin,temps,non_visiter)]
    | (c,t,nv)::f when t < temps -> (c,t,nv)::(ajoute chemin temps non_visiter f)
    | _ -> (chemin,temps,non_visiter)::file;;

let rec ajoute_chemins arretes (parcours,fin) temps non_visiter file graphe =
   match arretes with
       | [] -> file
       | a::l -> let arrete = graphe.arretes.(a) in
          let f = ajoute
             (a::parcours,arrete.arrivee)
             (temps +. arrete.tempsMin)
              (Array.copy non_visiter)
             file in
          ajoute_chemins l (parcours,fin) temps non_visiter f graphe;;

let rec djikstra_aux file arrivee graphe = match file with
    | [] -> []
    | ((parcours,fin),temps,non_visiter)::f when fin = arrivee ->
        parcours::djikstra_aux f arrivee graphe
    | ((parcours,fin),temps,non_visiter)::f when non_visiter.(fin) -> (
       djikstra_aux
          (ajoute_chemins graphe.noeuds.(fin) (parcours,fin) temps non_visiter f
             graphe)
          arrivee graphe)
    | ((parcours,fin),temps,non_visiter)::f ->djikstra_aux f arrivee graphe;;

let djikstra depart arrivee graphe =
   let non_visiter = Array.make (graphe.nombre_noeuds) true in
      djikstra_aux [(([],depart),0.,non_visiter)] arrivee graphe;;

djikstra 0 3 graphe;;
```

Calcul des fontions liants le temps au débit pour chaque chemin

```
let rec ajoute_arretes_indirect i fonctions liste arrete = match liste with
   | [] -> ()
   | j::l -> (fonctions.(j).(i) <- fonctions.(j).(i) +. arrete.acroissement;
      fonctions.(i).(j) <- fonctions.(i).(j) +. arrete.acroissement;
      ajoute_arretes_indirect i fonctions l arrete);;

let rec ajoute_arretes_direct parcours i fonctions usages graphe n= match
   parcours with
   | [] -> ()
   | a::p -> let arrete = graphe.arretes.(a) in (
      fonctions.(i).(n) <- fonctions.(i).(n) +. arrete.tempsMin;
      fonctions.(i).(i) <- fonctions.(i).(i) +. arrete.acroissement;
      ajoute_arretes_indirect i fonctions usages.(a) arrete;
      usages.(a) <- i::(usages.(a));
      ajoute_arretes_direct p i fonctions usages graphe n);;

let parcours_vers_fonctions parcours graphe n =
   let fonctions = Array.make_matrix n (n+1) 0.0 in
   let usages = Array.make graphe.nombre_arretes [] in
   for i = 0 to n-1 do
      ajoute_arretes_direct parcours.(i) i fonctions usages graphe n;
   done;fonctions;;

parcours_vers_fonctions par graphe 3;;
```

Calcul du gradient

```
let calcule fonction repartition n =
   let temps = ref fonction.(n) in
   for i = 0 to n-1 do
      temps := !temps +. fonction.(i) *. repartition.(i);
   done; !temps;;

let gradient fonction n =
   let grad = Array.make n 0. in
   for i = 0 to n-1 do
      grad.(i) <- fonction.(i);
   done;grad;;

let gradient_max fonctions repartition n =
   let max = ref 0. in
   let grad = ref (Array.make n 0.) in
   for i = 0  to n-1 do
      let valeur = (calcule fonctions.(i) repartition n) in
         if valeur > !max
            then (max := valeur; grad := gradient fonctions.(i) n);
   done;!grad;;
```

Algorithme de déscente du gradient

```
let distance a b n =
   let distance = ref 0.0 in
   for i = 0 to n-1 do
      distance := !distance +. (a.(i) -. b.(i)) *. (a.(i) -. b.(i));
   done; !distance;;

let nouvelle_reparition fonctions repartition pas n =
   let grad = gradient_max fonctions repartition n in
   let nouvelle = Array.copy repartition in
      for i = 1 to n-1 do
         nouvelle.(i) <- nouvelle.(i) -. (grad.(i) -. grad.(0))*. pas;
         nouvelle.(0) <- nouvelle.(0) +. (grad.(i) -. grad.(0))*. pas;
      done;
      print_newline();
      print_repartition grad n;
      print_repartition nouvelle n;
      nouvelle;;

let descente fonctions debit n =
   let pas = 0.1 in
   let repartion = ref (Array.make n 0.) in
      (!repartion).(0) <- debit ;
      let nouvelle = ref (nouvelle_reparition fonctions !repartion pas n) in
      while ((distance  !repartion !nouvelle n) > 0.5) do
         print_float (distance  !repartion !nouvelle n);
         repartion := !nouvelle;
         nouvelle := (nouvelle_reparition fonctions !repartion pas n);
      done;!nouvelle;;

descente fonctions 400. 3;;
```

Création du système

```
let fonctions_vers_system fonctions debit n =
   let system = Array.make_matrix n (n+1) 1. in
      system.(0).(n) <- debit ;
      for i = 1 to n-1 do
         for j = 0 to n-1 do
            system.(i).(j) <- fonctions.(0).(j) -. fonctions.(i).(j);
         done;
         system.(i).(n) <- fonctions.(i).(n) -. fonctions.(0).(n);
      done; system;;

fonctions_vers_system fonctions 400. 3;;
```

Résolution du système : Pivot de Gauss

```
let echanger system i j n =
   for k = 0 to n do
      let temp = system.(i).(k) in
      system.(i).(k) <- system.(j).(k) ;
      system.(j).(k) <- temp;
   done;;

let reduit system i c n =
   for k = 0 to n do
      system.(i).(k) <- system.(i).(k) /. c;
   done;;

let combine system i j c n =
   for k = 0 to n do
      system.(i).(k) <- system.(i).(k) +. system.(j).(k) *. c
   done;;

let pivot system k n =
   let indice = ref k in
   while (system.(!indice).(k) = 0.) do
      indice := !indice +1;
      if !indice = n
         then failwith "matrice non inversible";
   done; !indice;;

let resoud system n =
   for k = 0 to n-1 do
      let p = pivot system k n in
         echanger system k p n ;
         reduit system k (system.(k).(k)) n;
         for i = 0 to n-1 do
            if i != k
               then combine system i k (-.system.(i).(k)) n;
         done;
   done;
   let solution = Array.make n 0.0 in
   for k = 0 to n-1 do
      solution.(k) <- system.(k).(n);
   done; solution;;

resoud system 3;;
```

Programme de répartition

```
let repartit graphe debit =
   let parcours = djikstra 0 3 graphe in
   let n = List.length parcours in
   let fonctions = parcours_vers_fonctions (Array.of_list parcours) graphe n in
   let system = fonctions_vers_system fonctions debit n in
      resoud system n;;

repartit graphe 400.;;
```

Algorithme de Djikstra en générateur

```
let rec ajoute chemin temps non_visiter file = match file with
   | [] -> [(chemin,temps,non_visiter)]
   | (c,t,nv)::f when t < temps -> (c,t,nv)::(ajoute chemin temps non_visiter f)
   | _ -> (chemin,temps,non_visiter)::file;;

let rec ajoute_chemins arretes (parcours,fin) temps non_visiter file graphe =
   match arretes with
      | [] -> ()
      | a::l -> let arrete = graphe.arretes.(a) in
         file := ajoute
            (a::parcours,arrete.arrivee)
            (temps +. arrete.tempsMin)
             (Array.copy non_visiter)
             !file;
         ajoute_chemins l (parcours,fin) temps non_visiter file graphe;;

let rec djikstra file arrivee graphe= match !file with
   | [] -> []
   | ((parcours,fin),temps,non_visiter)::f when fin = arrivee ->
      file := f;
      parcours;
   | ((parcours,fin),temps,non_visiter)::f when non_visiter.(fin) ->
      file := f;
      ajoute_chemins graphe.noeuds.(fin) (parcours,fin) temps non_visiter file
          graphe;
      djikstra file arrivee graphe;
   | ((parcours,fin),temps,non_visiter)::f ->
      file := f;
      djikstra file arrivee graphe;;

let creer_file depart graphe =
   let non_visiter = Array.make (graphe.nombre_noeuds) true in
   ref [(([],depart),0.,non_visiter)];;

let file = creer_file 0 graphe;;

djikstra file 3 graphe;;
djikstra file 3 graphe;;
djikstra file 3 graphe;;
djikstra file 3 graphe;;
```

Quelques fonctions utiles

```
let rec calcul_temps_min parcours graphe = match parcours with
    | [] -> 0.
    | a::p -> let arrete = graphe.arretes.(a) in
       arrete.tempsMin +. calcul_temps_min p graphe;;

let calcul_temps repartition fonctions n =
    let temps = Array.make n 0. in
    for i = 0 to n-1 do
       for j = 0 to n-1 do
          temps.(i) <- temps.(i) +. fonctions.(i).(j) *. repartition.(j);
       done;
       temps.(i) <- temps.(i) +. fonctions.(i).(n);
    done;temps;;

let max_temps repartition fonctions n =
    let temps = calcul_temps repartition fonctions n in
    let max = ref (temps.(0)) in
    for i = 1 to n-1 do
       if temps.(i) > !max
          then max := temps.(i);
    done;!max;;
```

Programme de répartition amélioré

```
let repartit graphe depart arrivee debit =
    let file = creer_file depart graphe in
    let parcours = ref [djikstra file arrivee graphe] in
    let n = ref 1 in
       if !parcours = [[]]
          then failwith "pas_de_chemin"
          else
       let fonctions =
          ref (parcours_vers_fonctions (Array.of_list !parcours) graphe !n) in
       let repartition = ref [|debit|] in
       let parcours_suivant = ref (djikstra file arrivee graphe) in
       while (calcul_temps_min !parcours_suivant graphe) <= (max_temps
          !repartition !fonctions !n) && (!parcours_suivant != []) do
          parcours := (!parcours_suivant)::(!parcours);
          n := !n + 1;
          fonctions := parcours_vers_fonctions (Array.of_list !parcours) graphe
             !n;
          let system = fonctions_vers_system !fonctions debit !n in
             repartition := resoud system !n;
          parcours_suivant := (djikstra file arrivee graphe);
       done;!repartition;;
```