Correction du TD 13

Julien Reichert

Exercice 1

Le graphe a sept sommets et dix arcs.

Représentations en OCaml (versions avec des entiers pour simplifier) :

```
type graphe1 = { mutable sommets : int list;
mutable arcs : (int * int) list; };;

let mon_graphe1 = {sommets = [0; 1; 2; 3; 4; 5; 6];
arcs = [(0, 1); (0, 2); (0, 3); (1, 3); (1, 5); (2, 3); (3, 4); (3, 5); (4, 6); (6, 5)];;

type graphe2 = int list array;;

let mon_graphe2 = [| [1; 2; 3]; [3; 5]; [3]; [4; 5]; [6]; []; [5] |];;

type graphe3 = bool array array;;

let mon_graphe3 =
[|
[|false; true; true; true; false; false; false|];
[|false; false; false; true; false; true; false|];
[|false; false; false; true; false; false; false|];
[|false; false; false; false; true; true; false|];
[|false; false; false; false; false; false; true|];
[|false; false; false; false; false; false; false|];
[|false; false; false; false; false; true; false|];
|];;
```

Représentations en C (tableaux avec sentinelles pour les listes d'adjacence) :

```
struct grapho { int nbsommets; char* sommets; int nbarcs; char* arcs; };
typedef struct grapho graphe_oriente;

graphe_oriente g1 = { .nbsommets = 7 , .sommets = "0 1 2 3 4 5 6" ,
.nb_arcs = 10 , .arcs = "0-1 0-2 0-3 1-3 1-5 2-3 3-4 3-5 4-6 6-5" };
```

```
struct graphe_oriente_tableau_adjacence { int nb_sommets; int** tableau_adjacence; };
typedef struct graphe_oriente_tableau_adjacence gota;

gota init2()
{
  int** ta = malloc(7 * sizeof(int*);
  for (int i = 0 ; i < 7 ; i += 1)
  {
    ta[i] = malloc(4 * sizeof(int));
  }
  ta[0][0] = 1; ta[0][1] = 2; ta[0][2] = 3; ta[0][3] = -1;
  ta[1][0] = 3; ta[1][1] = 5; ta[1][2] = -1;
  ta[2][0] = 3; ta[2][1] = -1;
  ta[3][0] = 4; ta[3][1] = 5; ta[3][2] = -1;
  ta[4][0] = 6; ta[4][1] = -1;
  ta[5][0] = -1;
  ta[6][0] = 5; ta[6][1] = -1;
  return { .nb_sommets = 7 , .tableau_adjacence = ta };
}

gota g2 = init2();

struct graphe_oriente_matrice_adjacence { int nb_sommets; bool** matrice_adjacence; };
typedef struct graphe_oriente_matrice_adjacence goma;


gota init3()
{
  bool** ma = malloc(7 * sizeof(int*);
  for (int i = 0 ; i < 7 ; i += 1)
  {
    ma[i] = malloc(7 * sizeof(bool));
    for (int j = 0 ; j < 7 ; j += 1) ma[i][j] = false;
  }
  ma[0][1] = true; ma[0][2] = true; ma[0][3] = true;
  ma[1][3] = true; ma[1][5] = true;
  ma[2][3] = true;
  ma[3][4] = true; ma[3][5] = true;
  ma[4][6] = true;
  ma[6][5] = true;
  return { .nb_sommets = 7 , .matrice_adjacence = ma };
}

goma g3 = init3();
```

Exercice 2

Un chemin possible de 0 à 6 est $0, 3, 4, 6$. C'est aussi le plus court. Le graphe n'est pas fortement connexe car il n'existe aucun chemin commençant ailleurs qu'en 0 et finissant en 0.

Exercice 3

Le graphe n'admet pas de chemin hamiltonien car 1 n'est pas accessible depuis 2 ni vice-versa. Pour la même raison, il n'existe pas de chemin eulérien (autre raison : 0 est de degré sortant 3 et de degré entrant 0).

Exercice 4

Pour la fermeture transitive du graphe, on ajoute les arcs suivants : $(0, 4), (0, 5), (0, 6), (1, 4), (1, 6), (2, 4), (2, 5), (2, 6), (3, 6)$ et $(4, 5)$.

Le graphe étant sans circuit, ses composantes connexes (maximales) sont réduites aux sommets isolés.

Exercice 5

Étape 1 : 0 est déjà visité, appel sur 0.

Étape 2 : 0 et 1 sont déjà visités, appel sur 0 mis en attente, appel sur 1.

Étape 3 : $0, 1$ et 3 sont déjà visités, appel sur 0 mis en attente, appel sur 1 mis en attente, appel sur 3.

Étape 4 : $0, 1, 3$ et 4 sont déjà visités, appel sur 0 mis en attente, appel sur 1 mis en attente, appel sur 3 mis en attente, appel sur 4.

Étape 5 : $0, 1, 3, 4$ et 6 sont déjà visités, appel sur 0 mis en attente, appel sur 1 mis en attente, appel sur 3 mis en attente, appel sur 4 mis en attente, appel sur 6.

Étape 6 : $0, 1, 3, 4, 6$ et 5 sont déjà visités, appel sur 0 mis en attente, appel sur 1 mis en attente, appel sur 3 mis en attente, appel sur 4 mis en attente, appel sur 6 mis en attente, appel sur 5.

(5 n'a pas de voisins, appel dépilé.)

(6 n'a plus de voisins, appel dépilé.)

(4 n'a plus de voisins, appel dépilé.)

Étape 7 : $0, 1, 3, 4, 6$ et 5 sont déjà visités,appel sur 0 mis en attente, appel sur 1 mis en attente, appel sur 3 poursuivi.

(5 est déjà visité.)

(3 n'a plus de voisins, appel dépilé.)

Étape 8 : $0, 1, 3, 4, 6$ et 5 sont déjà visités, appel sur 0 mis en attente, appel sur 1 poursuivi.

(5 est déjà visité.)

(1 n'a plus de voisins, appel dépilé.)

Étape 9 : $0, 1, 3, 4, 6$ et 5 sont déjà visités, appel sur 0 poursuivi.

Étape 10 : $0, 1, 3, 4, 6, 5$ et 2 sont déjà visités, appel sur 0 remis en attente, appel sur 2.

(3 est déjà visité.)

(2 n'a plus de voisins, appel dépilé.)

Étape 10 : $0, 1, 3, 4, 6$ et 5 sont déjà visités, appel sur 0 poursuivi une fois de plus.

(3 est déjà visité.)

(0 n'a plus de voisins, appel dépilé.)

La pile des appels est vide. L'algorithme se termine et la liste des sommets déjà visités est retournée.

Exercice 6

Début : 0 déjà visité, la file contient 0.

Étape 1 : $0, 1, 2$ et 3 déjà visités, suite au traitement de 0, la file contient $1, 2$ et 3.

Étape 2 : $0, 1, 2, 3$ et 5 déjà visités, suite au traitement de 1 (3 déjà visité, non enfilé), la file contient $2, 3$ et 5.

Étape 3 : $0, 1, 2, 3$ et 5 déjà visités, suite au traitement de 2 (3 déjà visité, non enfilé), la file contient 3 et 5.

Étape 4 : $0, 1, 2, 3, 5$ et 4 déjà visités, suite au traitement de 3 (5 déjà visité, non enfilé), la file contient 5 et 4.

Étape 5 : $0, 1, 2, 3, 5$ et 4 déjà visités, suite au traitement de 5 (pas de voisin), la file contient 4.

Étape 6 : $0, 1, 2, 3, 5, 4$ et 6 déjà visités, suite au traitement de 4, la file contient 6.

Étape 7 : $0, 1, 2, 3, 5, 4$ et 6 déjà visités, suite au traitement de 6 (5 déjà visité, non enfilé), la file est vide.

L'algorithme se termine et la liste des sommets déjà visités est retournée.

Exercice 7

Comme aucun arc n'est de poids négatif, on peut utiliser l'algorithme de Dijkstra.

Exercice 8

Avec l'algorithme de Dijkstra, étape par étape :

Sommet	0	1	2	3	4	5	6	File de priorité
Distance	0							0
Prédécesseur	∅							(départ)
Distance	0	4	2	10				2 1 3
Prédécesseur	∅	0	0	0				(étape 1)
Distance	0	4	2	9				1 3
Prédécesseur	∅	0	0	2				(étape 2)
Distance	0	4	2	7		11		3 5
Prédécesseur	∅	0	0	1		1		(étape 3)
Distance	0	4	2	7	8	10		4 5
Prédécesseur	∅	0	0	1	3	3		(étape 4)
Distance	0	4	2	7	8	10	8	6 5
Prédécesseur	∅	0	0	1	3	3	4	(étape 5)
Distance	0	4	2	7	8	9	8	5
Prédécesseur	∅	0	0	1	3	6	4	(étape 6)
Distance	0	4	2	7	8	9	8	(vide)
Prédécesseur	∅	0	0	1	3	6	4	(étape 7)

L'arbre réalisant les distances minimales s'obtient en gardant 6 arcs, de destination tous les sommets sauf 0 et d'origine le sommet donné par l'information du prédécesseur.

Exercice 9

Algorithme de Dijkstra en OCaml :

```
type graphe_dijkstra = (int * int) list array;;

let graphe_test =
[|
  [(2, 2); (1, 4); (3, 10)];
  [(3, 3); (5, 7)];
  [(3, 7)];
  [(4, 1); (5, 3)];
  [(6, 0)];
  [];
  [(5, 1)]
|];;
```

```
type 'a tas = Vide | Noeud of 'a tas * 'a * 'a tas;;

let creer_fp () = Vide;;

let est_fp_vide fp = fp = creer_fp ();;

(* Attention, la structure est persistante ! *)
let rec ajouter_fp dist fp s = match fp with
| Vide -> Noeud(Vide, s, Vide)
| Noeud(g, t, d) ->
    let (nouvrac, nouvajout) = (if dist.(s) < dist.(t) then (s, t) else (t, s)) in
    Noeud(d, nouvrac, ajouter_fp dist g nouvajout);;

(* Renvoie la racine et la file sans celle-ci. *)
let rec extraire_fp dist fp = match fp with
| Vide -> failwith "File de priorité vide"
| Noeud(g, s, Vide) -> s, g
| Noeud(Vide, s, d) -> s, d
| Noeud((Noeud(gg, sg, gd) as g), s, (Noeud(dg, sd, dd) as d)) ->
(* Utilisation exceptionnelle de la syntaxe as qui simplifie l'écriture *)
    s, (if dist.(sg) < dist.(sd)
    then Noeud(snd (extraire_fp dist g), sg, d)
    else Noeud(g, sd, snd (extraire_fp dist d)));;

(* Autre possibilité : ajouter systématiquement, quitte à avoir des doublons,
et tester au moment de retirer un sommet. *)
let rec remonter_fp dist fp t = match fp with
(* attention, coût linéaire ici, une optimisation reviendrait à localiser l'élément *)
| Vide -> Vide (* Tous les cas où le sommet n'est pas trouvé dans la branche... *)
| Noeud(g, s, d) when s = t -> fp
| Noeud(Noeud(gg, sg, gd), s, d) when sg = t ->
    if dist.(t) < dist.(s) then Noeud(Noeud(gg, s, gd), t, d) else fp
| Noeud(g, s, Noeud(dg, sd, dd)) when sd = t ->
    if dist.(t) < dist.(s) then Noeud(g, t, Noeud(dg, s, dd)) else fp
| Noeud(g, s, d) ->
    let gbis = remonter_fp dist g t and dbis = remonter_fp dist d t in
    begin
      match gbis, dbis with
      | Vide, Vide -> fp
      | Noeud(gg, sg, gd), _ when sg = t && dist.(t) < dist.(s) ->
        Noeud(Noeud(gg, s, gd), t, d)
(* pas besoin de dbis, d ne peut pas être modifié alors *)
      | _, Noeud(dg, sd, dd) when sd = t && dist.(t) < dist.(s) ->
        Noeud(g, t, Noeud(dg, s, dd))
      | _, _ -> Noeud(gbis, s, dbis)
    end;;
```

```
let dijkstra graphe origine =
  let n = Array.length graphe in
    let dist = Array.make n max_int in
      dist.(origine) <- 0;
      let rec whilerecursif ouverts = if not (est_fp_vide ouverts) then begin
        let s, ouverts2 = extraire_fp dist ouverts in
          let voisins = graphe.(s) in
            let fonction_a_folder fp (t, poids) =
              if dist.(s) + poids < dist.(t) then
                begin
                  let il_faut_ajouter = dist.(t) = max_int in
                  dist.(t) <- dist.(s) + poids;
                  if il_faut_ajouter then ajouter_fp dist fp t
                  else remonter_fp dist fp t
                end
              else fp
            in whilerecursif (List.fold_left fonction_a_folder ouverts2 voisins)
       end
       in whilerecursif (ajouter_fp dist (creer_fp ()) origine);
       dist;;
```

Algorithme de Dijkstra en C (on suppose les en-têtes inclus) :

```
struct arc_pondere { int sommet; int poids; };
typedef struct arc_pondere arcp;

struct t_p { int nb_successeurs; arcp* successeurs; };
typedef struct t_p tp;

struct graphe_oriente_tableau_poids { int nb_sommets; tp* tableau_poids; };
typedef struct graphe_oriente_tableau_poids gotp;

const int dpt = 1073741824;

int infini_t(gotp g)
{
  int n = g.nb_sommets;
  int rep = 1;
  for (int i = 0 ; i < n ; i += 1)
  {
    for (int j = 0 ; j < g.tableau_poids[i].nb_successeurs ; j += 1)
    {
      rep += g.tableau_poids[i].successeurs[j].poids;
    }
  }
  return rep;
}
```

```
// Sans se fatiguer, avec une simple liste pour la file de priorité.
// De même, pas de prédécesseurs calculés.
int* dijkstra(gotp g, int sommet)
{
  int n = g.nb_sommets;
  int infi = infini_t(g);
  int* rep = malloc(n * sizeof(int));
  for (int i = 0 ; i < n ; i += 1) rep[i] = infi;
  rep[sommet] = 0;
  int* fp = malloc(n * sizeof(int));
  fp[0] = sommet;
  int taille_fp = 1;
  while (taille_fp > 0)
  {
    int posmin = 0;
    for (int i = 0 ; i < taille_fp ; i += 1)
    {
      if (rep[fp[i]] < rep[fp[posmin]]) posmin = i;
    }
    int s = fp[posmin];
    fp[posmin] = fp[taille_fp-1];
    taille_fp -= 1;
    tp successeurs = g.tableau_poids[s];
    for (int i = 0 ; i < successeurs.nb_successeurs ; i += 1)
    {
      int t = successeurs.successeurs[i].sommet;
      int poids = successeurs.successeurs[i].poids;
      if (rep[s] + poids < rep[t])
      {
        rep[t] = rep[s] + poids;
        bool ajouter = true;
        for (int j = 0 ; j < taille_fp ; j += 1)
        {
          if (fp[j] == t) ajouter = false;
        }
        if (ajouter)
        {
          fp[taille_fp] = t; taille_fp += 1;
        }
      }
    }
  }
  free(fp);
  return rep;
}
```

```
int main()
{
  arcp a01 = { .sommet = 1, .poids = 4 };
  arcp a02 = { .sommet = 2, .poids = 2 };
  arcp a03 = { .sommet = 3, .poids = 10 };
  arcp a13 = { .sommet = 3, .poids = 3 };
  arcp a15 = { .sommet = 5, .poids = 7 };
  arcp a23 = { .sommet = 3, .poids = 7 };
  arcp a34 = { .sommet = 4, .poids = 1 };
  arcp a35 = { .sommet = 5, .poids = 3 };
  arcp a46 = { .sommet = 6, .poids = 0 };
  arcp a65 = { .sommet = 5, .poids = 1 };
  tp* tableaux_poids = malloc(7 * sizeof(tp));
  tableaux_poids[0].nb_successeurs = 3;
  tableaux_poids[0].successeurs = malloc(3 * sizeof(arcp));
  tableaux_poids[0].successeurs[0] = a01;
  tableaux_poids[0].successeurs[1] = a02;
  tableaux_poids[0].successeurs[2] = a03;
  tableaux_poids[1].nb_successeurs = 2;
  tableaux_poids[1].successeurs = malloc(2 * sizeof(arcp));
  tableaux_poids[1].successeurs[0] = a13;
  tableaux_poids[1].successeurs[1] = a15;
  tableaux_poids[2].nb_successeurs = 1; tableaux_poids[2].successeurs = &a23;
  tableaux_poids[3].nb_successeurs = 2;
  tableaux_poids[3].successeurs = malloc(2 * sizeof(arcp));
  tableaux_poids[3].successeurs[0] = a34;
  tableaux_poids[3].successeurs[1] = a35;
  tableaux_poids[4].nb_successeurs = 1 ; tableaux_poids[4].successeurs = &a46;
  tableaux_poids[5].nb_successeurs = 0 ; tableaux_poids[5].successeurs = NULL;
  tableaux_poids[6].nb_successeurs = 1 ; tableaux_poids[6].successeurs = &a65;
  gotp g = { .nb_sommets = 7 , .tableau_poids = tableaux_poids };

  int* rep = dijkstra(g, 0);
  printf("Distances :");
  for (int i = 0 ; i < 7 ; i += 1)
    printf(" %d", rep[i]);
  printf("\n");

  free(tableaux_poids[0].successeurs);
  free(tableaux_poids[1].successeurs);
  free(tableaux_poids[3].successeurs);
  free(tableaux_poids);
  free(rep);

  return 0;
}
```

Exercice 10

Algorithme de Floyd-Warshall en OCaml (avec une matrice de poids contenant `max_int` aux positions où il n'existe pas d'arc, en admettant qu'il n'y ait pas de risque d'approcher une telle valeur avec des arcs) :

```
let graphe_fw = Array.make_matrix 7 7 max_int;;
graphe_fw.(0).(1) <- 4;;
graphe_fw.(0).(2) <- 2;;
graphe_fw.(0).(3) <- 10;;
graphe_fw.(1).(3) <- 3;;
graphe_fw.(1).(5) <- 7;;
graphe_fw.(2).(3) <- 7;;
graphe_fw.(3).(4) <- 1;;
graphe_fw.(3).(5) <- 3;;
graphe_fw.(4).(6) <- 0;;
graphe_fw.(6).(5) <- 1;;

let floyd_warshall graphe =
  let n = Array.length graphe in
    let dist = Array.make_matrix n n max_int in
      for i = 0 to n-1 do
        for j = 0 to n-1 do
          dist.(i).(j) <- graphe.(i).(j)
        done
      done;
      for i = 0 to n-1 do
        if dist.(i).(i) > 0
          then dist.(i).(i) <- 0
      done;
      for passage = 1 to 2 do
        for k = 0 to n-1 do
          for i = 0 to n-1 do
            for j = 0 to n-1 do
              if min dist.(i).(k) dist.(k).(j) = min_int
                && max dist.(i).(k) dist.(k).(j) < max_int
                then dist.(i).(j) <- min_int
              else if max dist.(i).(k) dist.(k).(j) < max_int
                then dist.(i).(j) <- min dist.(i).(j) (dist.(i).(k) + dist.(k).(j))
            done;
            if dist.(i).(i) < 0
              then dist.(i).(i) <- min_int
          done
        done
      done;
      dist;;
```

Algorithme de Floyd-Warshall en C (on suppose les en-têtes inclus) :

```
struct graphe_oriente_matrice_poids
{ int nb_sommets; int** matrice_poids; };
typedef struct graphe_oriente_matrice_poids gomp;

const int dpt = 1073741824;

int** floyd_warshall(gomp g)
{
  int n = g.nb_sommets;
  int** rep = malloc(n * sizeof(int*));
  for (int i = 0 ; i < n ; i += 1)
  {
    rep[i] = malloc(n * sizeof(int));
    for (int j = 0 ; j < n ; j += 1)
      rep[i][j] = g.matrice_poids[i][j];
    if (rep[i][i] < 0)
      rep[i][i] = -dpt;
    else
      rep[i][i] = 0;
  }
  for (int k = 0 ; k < 2 * n ; k += 1)
// Astuce pour les deux répétitions : calculer k modulo n
  {
    k %= n;
    for (int i = 0 ; i < n ; i += 1)
    {
      for (int j = 0 ; j < n ; j += 1)
      {
        if (rep[i][k] != dpt && rep[k][j] != dpt)
        {
          int test = rep[i][k] + rep[k][j];
          if (rep[i][k] == -dpt || rep[k][j] == -dpt)
            test = -dpt;
          if (rep[i][j] > test)
            rep[i][j] = test;
        }
      }
      if (rep[i][i] < 0)
        rep[i][i] = -dpt;
    }
  }
  return rep;
}
```

```
int main()
{
  int** mp = malloc(7 * sizeof(int*));
  for (int i = 0 ; i < 7 ; i += 1)
  {
    mp[i] = malloc(7 * sizeof(int));
    for (int j = 0 ; j < 7 ; j += 1)
    {
      mp[i][j] = dpt;
    }
  }
  mp[0][1] = 4;
  mp[0][2] = 2;
  mp[0][3] = 10;
  mp[1][3] = 3;
  mp[1][5] = 7;
  mp[2][3] = 7;
  mp[3][4] = 1;
  mp[3][5] = 3;
  mp[4][6] = 0;
  mp[6][5] = 1;
  gomp g = { .nb_sommets = 7 , .matrice_poids = mp };
  int** rep = floyd_warshall(g);
  for (int i = 0 ; i < 7 ; i += 1)
  {
    for (int j = 0 ; j < 7 ; j += 1)
    {
      printf("%d ", rep[i][j]);
    }
    printf("\n");
  }
  for (int i = 0 ; i < 7 ; i += 1)
  {
    free(mp[i]);
    free(rep[i]);
  }
  free(mp);
  free(rep);
  return 0;
}
```

Exercice 11

Comme on l'a observé, l'algorithme de Floyd-Warshall était similaire dans les deux langages. Pour l'algorithme de Bellman-Ford, on se contentera d'une version en OCaml.

Pour la lisibilité (et au mépris de l'optimalité), on utilisera une matrice de poids pour le graphe en entrée.

```
let bellman_ford graphe origine =
  let n = Array.length graphe in
  let dist = Array.make n max_int in
  dist.(origine) <- 0;
  for k = 0 to 2 * n - 1 do
    let nouv_dist = Array.copy dist in
    for sommet = 0 to n-1 do
      for pred = 0 to n-1 do
        if graphe.(pred).(sommet) < max_int && dist.(pred) < max_int then
          let essai = dist.(pred) + graphe.(pred).(sommet) in
          nouv_dist.(sommet) <- min nouv_dist.(sommet) essai
      done
    done;
    if k < n then
      for recopiage = 0 to n-1 do
        dist.(recopiage) <- nouv_dist.(recopiage)
      done
    else
      for verif = 0 to n-1 do
        if nouv_dist.(verif) < dist.(verif) then dist.(verif) <- min_int
        (* Il y a un circuit de poids strictement négatif. *)
      done
  done; dist;;
```

Une version avec des listes d'adjacence issue du DS 5 de 2022/2023 (rejet des circuits de poids strictement négatif, et une optimisation était demandée dans le DS, consulter la correction pour la solution) :

```
let bellman_ford graphe origine =
  let n = Array.length graphe in
  let dist = Array.make n max_int in dist.(origine) <- 0;
  for k = 1 to n do
    let nouv_dist = Array.copy dist in
    for sommet = 0 to n-1 do
      if dist.(sommet) < max_int then
        for isucc = 0 to Array.length graphe.(sommet)-1 do
          let succ, poids = graphe.(sommet).(isucc) in
          nouv_dist.(succ) <- min nouv_dist.(succ) (dist.(sommet) + poids)
        done
    done;
    if k < n then
      for recopiage = 0 to n-1 do
        dist.(recopiage) <- nouv_dist.(recopiage)
      done
    else
      for verif = 0 to n-1 do
        if nouv_dist.(verif) < dist.(verif)
        then failwith "Circuit de poids strictement négatif !"
      done
  done; dist;;
```