Les graphes
Plus courts chemins
Shortest Paths
Correction

1 La star des algos

Solution 1.1 **(Dijkstra)**

4. (b) *Application de ce principe au graphe de la figure 1 pour trouver le plus court chemin de 1 à 8 :*
Les sommets sont traités dans l'ordre suivant :
1, 2, 4, 3, 7, 5, 6, 8, 9

	1	2	3	4	5	6	7	8	9	10	11	12	13
dist	0	4	7	1	9	4	5	11	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
pere	1	6	2	1	7	4	2	5					

Le plus court chemin entre les sommets 1 et 8 est donc : 1, 4, 6, 2, 7, 5, 8 de coût 11.

Si seul le vecteur des pères est donné en résultat (suffisant), alors il faut initialiser celui-ci (par exemple à -1), afin de repérer les sommets non atteints depuis la source.

(c) **Spécifications :**
La procedure `Dijkstra (t_graph_dyn` G, `entier` s, `t_vect_entiers` *dist*, *pere*) recherche les plus courts chemins du sommet s vers tous les autres sommets du graphe G à coûts positifs : le vecteur *dist* contient les coûts des chemins représentés par le vecteur *pere*.

```
algorithme procedure Dijkstra
    parametres locaux
        t_graph_dyn    G
        entier         source
    parametres globaux
        t_vect_entiers    dist, pere

    variables
        entier         i, imin, x, y
        t_liste        L           /* contient des pointeurs sur sommets */
        t_listsom      ps, ps2
        t_listadj      pa
debut
                                                  /* initialisations */
    pour i ← 1 jusqu'a G.ordre faire
        dist[i] ← +∞
        pere[i] ← -1
    fin pour
    dist[source] ← 0
    pere[source] ← source      /* pas nécessaire... */

    L.elts[1] ← recherche (source, G)
    L.longueur ← 1
```

```
faire
   ps ← L.elts[1]              /* recherche du minimum de la liste */
   imin ← 1
   pour i ← 2 jusqu'a L.longueur faire
       ps2 ← L.elts[i]
       si dist[ps2↑.som] < dist[ps↑.som] alors
           ps ← ps2
           imin ← i
       fin si
   fin pour
                         /* suppression du minimum */
   L.elts[imin] ← L.elts[L.longueur]
   L.longueur ← L.longueur - 1

   x ← ps↑.som               /* parcours des successeurs de x */
   pa ← ps↑.succ
   tant que pa <> NUL faire
       y ← pa↑.vsom↑.som
       si dist[x] + pa↑.cout < dist [y] alors        /* relâchement arc (x, y) */
           si dist[y] = +∞ alors
               L.longueur ← L.longueur + 1
               L.elts[L.longueur] ← pa↑.vsom
           fin si
           dist[y] ← dist[x] + pa↑.cout
           pere[y] ← x
       fin si
       pa ← pa↑.suiv
   fin tant que
tant que L.longueur <> 0
fin algorithme procedure Dijkstra
```

Solution 1.2 **(L'aller, puis le retour ...)**

3. **Spécifications :**
   La procédure pccAR (G, scr, dst, $pereA$, $pereB$) trouve un chemin aller-retour de src à dst, les deux chemins sont donnés dans $pereA$ et $pereR$. Le graphe G est tel qu'il existe forcément un chemin aller-retour de src à dst.

```
algorithme procedure pccAR
    parametres locaux
        t_graph_dyn         G
        entier              src, dst
    parametres globaux
        t_vect_entiers      pereA, pereR
    variables
        entier              i
        t_vect_booleens     M
debut
    pour i ← 1 jusqu'a G.ordre faire
        M[i] ← faux
    fin pour
    dijkstra (G, src, dst, M, pereA)
    i ← dst
    tant que i <> src faire
        M[i] ← vrai
        i ← pereA[i]
    fin tant que
    dijkstra (G, dst, src, M, pereR)
fin algorithme procedure pccAR
```

Spécifications :

La procédure `dijkstra` (*G*, *scr*, *dst*, *M*, *pere*) trouve un plus court chemin de *src* à *dst* ne passant pas par les sommets marqués par le vecteur de booléens *M*, **le chemin existe forcément** et est donné dans le vecteur d'entiers *pere*.

```
algorithme procedure dijkstra
    parametres locaux
        t_graph_dyn      G
        entier           src,  dst
        t_vect_booleens  M
    parametres globaux
        t_vect_entiers   pere

    variables
        entier        i
        t_vect_reels  dist
        t_tas         T     /* contient des couples (pointeur, valeur de tri) */
        t_listsom     ps
        t_listadj     pa
  debut
      pour i ← 1 jusqu'a G.ordre faire
          pere[i] ← -1
          dist[i] ← +∞
      fin pour
      pere[src] ← src
      dist[src] ← 0
      ps ← recherche (src, G)
      init_tas (T)
      tant que src <> dst faire
          pa ← ps↑.succ
          tant que (pa <> NUL) faire
              i ← pa↑.vsom↑.som
              si non M[i] et (dist[i] > dist[src] + pa↑.cout) alors
                  pere[i] ← src
                  dist[i] ← dist[src] + pa↑.cout
                  maj_tas (pa↑.vsom, dist[i], T)
              fin si
              pa ← pa↑.suiv
          fin tant que
          ps ← supp_min (T)
          src ← ps↑.som
      fin tant que
  fin algorithme procedure dijkstra
```

Cet algorithme suppose que le chemin cherché existe : Il est donc inutile ici de tester si le tas est vide avant de choisir un sommet, on arrête dès que l'on tombe sur la destination.

Attention : le choix du sommet ne peut pas être placé en fin de boucle dans une version où l'existence du chemin n'est pas garantie. En effet, le tas peut être temporairement vide après en avoir supprimé le minimum alors qu'il reste des sommets à traiter.

Solution 1.4 **(Et l'Astar dans tout ça ?)**

1. *Le but :* On cherche à trouver le plus court chemin dans un 1−graphe orienté valué à coûts positifs en traitant un minimum de sommets inintéressants.

   Principe de l'algorithme A^* :

   La différence avec les algorithmes de plus courts chemins classiques est que l'on utilise l'estimation donnée par une heuristique pour le choix des sommets à traiter : à chaque itération on choisit le sommet dont la **somme** du chemin parcouru avec l'estimation du devin est la **plus petite**. Il faut de plus éviter les circuits, donc on ne traitera pas à nouveau les sommets déjà traités : ces sommets sont dits "fermés", ceux non encore traités sont "ouverts".

   L'heuristique : estime la distance à parcourir depuis le sommet vers la destination. L'estimation ne tient pas compte du chemin parcouru pour atteindre le sommet.

2. **Résultat avec l'heuristique donnée**

s	1	2	3	4	5	6	7	8	9	10
$H(s)$	3	2	3	1	3	6	4	3	2	0

   Les sommets traités (dans l'ordre) : 1 3 2 4 10

   Le résultat :

	1	2	3	4	5	6	7	8	9	10
dist	0	4	1	6	5	2	∞	∞	∞	7
pere	-1	1	1	2	2	3	-1	-1	-1	5

   Résultat avec Dijkstra

   Les sommets traités (dans l'ordre) : 1 3 6 7 2 8 5 9 4 10

   Le résultat :

	1	2	3	4	5	6	7	8	9	10
dist	0	4	1	6	5	2	3	4	5	6
pere	-1	1	1	2	2	3	6	7	8	5

À la $3^{ème}$ itération, A^* ne choisit pas le sommet 6 (trop loin de la destination). De plus après 4, 10 est choisi et donc l'algo s'arrête : le chemin donné n'est pas le plus court (distance 7 au lieu de 6), mais est trouvé plus rapidement.

En plus l'algorithme :

Spécifications :

La procédure `Asterix (t_graph_dyn` *G*, `entier` *source*, *destination*, `t_vect_entiers` *pere*) remplit le vecteur *pere* qui permettra de retrouver le chemin entre le sommet *source* et le sommet *destination*. On supposera que le chemin existe toujours.

La fonction `heuristix (`*G*, *src*, *dst*`)` calcule une estimation du coup d'un chemin entre *src* et *dst* dans le graphe *G*.

```
algorithme procedure Astar
    parametres locaux
        t_graph_dyn     G
        entier          src, dest        /* on suppose src <> dest ! */
    parametres globaux
        t_vect_entiers     pere

     variables
        t_listsom     ps
        t_listadj     pa
        entier        s, sadj
        t_vect_reels     dist
        t_vect_booleens fermes
        t_tas     T

debut
                                                        /* initialisations */
    pour s ← 1 jusqu'a G.ordre faire
        dist[s] ← ∞
        pere[s] ← -1
        ferme[s] ← faux
    fin pour
    dist[src] ← 0

    ps ← recherche (src, G)
    init_tas (T)

    faire
        s ← ps↑.som
        ferme[s] ← vrai

        pa ← ps↑.succ              /* parcours des successeurs de s */
        tant que pa <> NUL faire
            sadj ← pa↑.vsom↑.som                        /* relâchement arc (s,sadj) */
            si non ferme[sadj] et (dist[s] + pa↑.cout < dist [sadj]) alors
                dist[sadj] ← dist[s] + pa↑.cout
                pere[sadj] ← s
                maj (T, pa↑.vsom, dist[sadj] + heuristique (G, sadj, dest))
            fin si
            pa ← pa↑.suiv
        fin tant que

        ps ← supp_min (T)

    tant que ps^.som <> dest

fin algorithme procedure Astar
```

2 C'est négatif

Solution 2.2 **(Plus court chemin et parcours profondeur)**

1. **Tri topologique et élimination des circuits :**

   (c) **Spécifications :** La procédure `prof_pour_Bellman` (G, *src*, *tri*, *suff*) effectue un parcours profondeur de G à partir du sommet *src*, uniquement sur les sommets atteignables. Elle empile les sommets rencontrés en suffixe dans la pile *tri* et donne dans *suff* l'ordre de rencontre des sommets en suffixe.

/ algo récursif */*

```
algorithme procedure pprof_rec
    parametres locaux
        t_listsom        ps
    parametres globaux
        t_vect_booleens  M
        entier           cpt
        t_vect_entiers   os
        t_pile           p

    variables
        t_listadj        pa
        entier           s, sa
debut
    s ← ps↑.som
    M[s] ← vrai
    pa ← ps↑.succ
    tant que pa <> NUL faire
        sa ← pa↑.vsom↑.som
        si non M[sa] alors
            pprof_rec (pa↑.vsom, M, cpt, os, p)
        fin si
        pa ← pa↑.suiv
    fin tant que
    cpt ← cpt + 1
    os[s] ← cpt
    p ← empiler(ps, p)
fin algorithme procedure pprof_rec
```

/ algo d'appel */*

```
algorithme procedure prof_pour_Bellman
    parametres locaux
        t_graph_dyn G
        entier      src
    parametres globaux
        t_pile   tri  /* contient des t_listsom */
        t_vect_entiers  suff

    variables
        entier  i
        t_vect_booleens M
debut
    pour i ← 1 jusqu'a G.ordre faire
        M[i] ← faux
    fin pour
    tri ← pile_vide ()
    i ← 0
    pprof_rec (recherche (src,G), M, i, suff, tri)
fin algorithme procedure prof_pour_Bellman
```

2. **Plus courts chemins :**

   (b) **Spécifications :**

   La procedure `Bellman_prof` (*G*, *src*, *pere*, *dist*) calcule les plus courts chemins depuis le sommet *src* dans *G*. L'algorithme remplit le vecteur de pères (*pere*) et le vecteur de distances (*dist*) avec les chemins calculés.

```
algorithme procedure bellman_prof
    parametres locaux
        t_graph_dyn        G
        entier             src
    parametres globaux
        t_vect_entiers     pere
        t_vect_reels       dist

    variables
        t_vect_entiers     os
        t_pile             tri
        t_listsom          ps
        t_listadj          pa
        entier             s, sa
debut
    prof_pour_Bellman (G, src, tri, os)
    pour s ← 1 jusqu'a G.ordre faire
        pere[s] ← 0
        dist[s] ← +∞
    fin pour
    pere[src] ← src
    dist[src] ← 0
    ps ← depiler(tri)           /* forcément la source */
    tant que non est_vide (tri) faire
        s ← ps↑.som
        pa ← ps↑.succ
        tant que pa <> NUL faire
            sa ← pa↑.vsom↑.som
            si non os[s] < os[sa] alors
                si dist[sa] > dist[s] + pa↑.cout alors
                    dist[sa] ← dist[s] + pa↑.cout
                    pere[sa] ← s
                fin si
            fin si
            pa ← pa↑.suiv
        fin tant que
        ps ← depiler(tri)       /* inutile de voir les successeurs du dernier ! */
    fin tant que
fin algorithme procedure bellman_prof
```

***Solution 2.3* (Construction)**

1. *Modélisation du projet sous forme de graphe.*

   Les sommets sont les tâches. Il existe un arc (t_i, t_j) si la tâche t_i doit être terminée avant de pouvoir commencer la tâche t_j. Le coût de l'arc (t_i, t_j) est la durée de la tâche t_i.

   On ajoute deux tâches supplémentaires :
   - la tâche *début* est liée à toutes les tâches n'ayant aucun prédécesseur par des arcs de coût nul ;
   - chaque tâche sans successeur contient un arc sortant vers la tâche de *fin* avec pour coût sa durée.

   Voir la figure 1

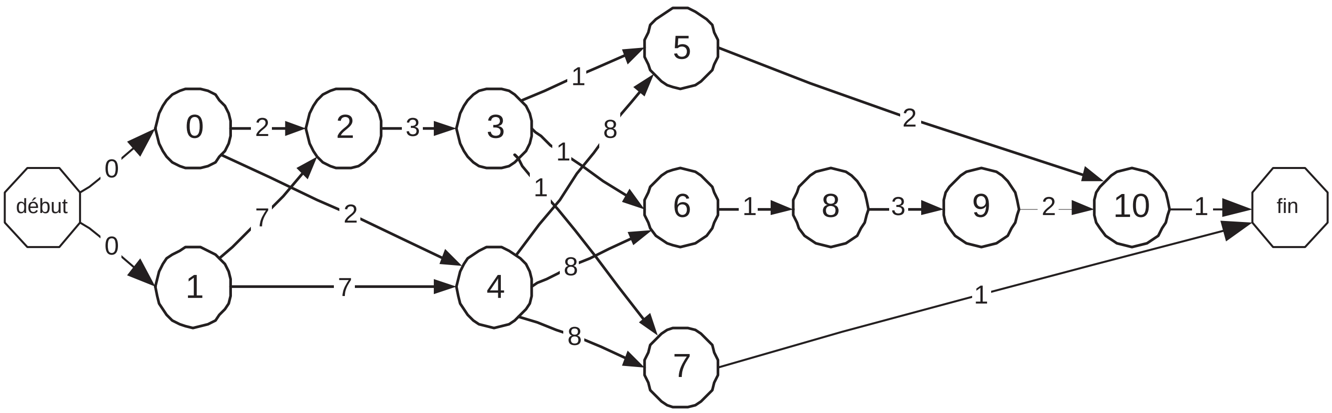

Figure 1 – Le graphe du projet

2. *Durée minimale du projet :* 22 semaines.

   Pour obtenir cette durée, il faut calculer la longueur du plus long chemin de la tâche de *début* jusqu'à celle de *fin*.

3. On se ramène à un problème de plus court chemin en inversant les coûts du graphes. Ceux-ci étant maintenant négatifs, mais le graphe sans circuit, on peut utiliser une solution de tri topologique comme ordre de sommets dont on relâche les arcs sortants.

   L'application de cette méthode sur le graphe auquel on a inversé les coûts, en prenant pour solution de tri topologique les sommets en ordre croissant (*début*, et *fin*, à leurs places respectives !), donne :

	début	0	1	2	3	4	5	6	7	8	9	10	*fin*
dist	0	0	0	-7	-10	-7	-15	-15	-15	-16	-19	-21	-22
pere		*début*	*début*	1	2	1	4	4	4	6	8	9	10

   Remarque : Le vecteur *pere* n'est pas nécessaire pour obtenir la durée minimale du projet.

4. On modifie l'algorithme de Bellman : les coûts sont positifs, on utilise donc les opposés (à noter qu'on aurait pu directement maximiser le chemin). Le vecteur *pere* n'est pas utile ici.

   C'est la version avec mise à jour des demi-degrés intérieurs qui est utilisée ici : on utilise la procédure `calcul_ddi` $(G,\ ddi)$ qui remplit le vecteur *ddi* avec les degrés entrants de tous les sommets du graphe (à noter que dans le cas de notre projet, tous les sommets sont atteignables depuis la source).

   Optimisation : plutôt que de rechercher à chaque fois dans tout le tableau ddi, une valeur nulle, on conserve (dans une file par exemple) les sommets dès que leurs degrés entrants passent à 0.

   Au départ, la file ne contient que la source. L'algorithme s'arrête dès que la file est vide !

   Spécifications :

   La fonction `duree_minimale` $(G,\ source,\ fin)$ calcule la longueur du plus long chemin (la durée minimale du projet) dans le graphe G, entre les tâches *source* et fin.
   Le graphe G est sans circuit. Le sommet fin est atteignable depuis la source !

```
algorithme fonction duree_minimale : reel
    parametres locaux
        t_graph_dyn    G
        entier         s, f

    variables
        t_list_som      ps
        t_list_adj      pa
        t_vect_entiers  ddi
        t_vect_reels    dist
        t_file          prochain   /* contient les pointeurs vers les sommets de ddi nul */
        entier          sadj
debut
    pour a ← 1 jusqu'a G.ordre faire                    /* init */
        dist[a] ← +∞
    fin tant que
    dist[s] ← 0
    calcul_ddi (G, ddi)

    prochain ← file-vide ()
    prochain ← enfiler (recherche (s,g), prochain)
    faire
        ps ← defiler(prochain)
        s ← ps↑.som
        pa ← ps↑.succ
        tant que pa <> NUL faire
            sadj ← pa↑.vsom↑.som
            si dist[s] - [1] pa↑.cout < dist [sadj] alors  /* relâchement arc sortant (s,a)*/
                dist[sadj] ← dist[s] - pa↑.cout
            fin si
            ddi[sadj] ← ddi[sadj]-1                          /* et màj ddi[sadj] */
            si ddi[sadj] = 0 alors
                prochain ← enfiler(pa↑.vsom, prochain)
            fin si
            pa ← pa↑.suiv
        fin tant que
    tant que non est-vide (prochain)
    retourne (-dist[f])
fin algorithme fonction duree_minimale
```

Remarque : on peut aussi directement maximiser la somme des coûts.

5. Le tableau de distances obtenu lors de la recherche du plus long chemin contient les **dates au plus tôt** (en négatif ici) : c'est la durée minimale avant de pouvoir commencer chaque tâche.
   Pour obtenir les **dates au plus tard**, il faut considérer le graphe inverse (où l'on inverse le sens des arcs !) et rechercher les plus longs chemins depuis la tâche de fin (Dans l'algorithme ci-dessus, il suffit de parcourir les listes de prédécesseurs, à la place des successeurs !) : la date au plus tôt d'une tâche est la différence entre la durée minimale du projet et la longueur du chemin obtenu.
   Les tâches constituant le plus long chemin obtenu lors du calcul de la durée minimale du projet sont des tâches critiques (voir le tableau *pere*). Mais cela ne suffit pas pour les obtenir toutes. En effet, il peut y avoir plusieurs chemins de même longueur !

6. Les dates au plus tôt, au plus tard pour chaque tâche :

	début	0	1	2	3	4	5	6	7	8	9	10	fin
Dates au plus tôt	0	0	**0**	7	10	**7**	15	**15**	15	**16**	**19**	**21**	22
Dates au plus tard	0	5	**0**	11	14	**7**	19	**15**	21	**16**	**19**	**21**	22

Les tâches critiques : 1, 4, 6, 8, 9, 10

1. Les coûts sont considérés positifs.

Solution 2.4 **(Floyd revisité –** ***Partiel 2013*****)**

1. La modification est simple. Un circuit absorbant va renvoyer un coût négatif sur la distance calculée d'un sommet x à ce même sommet x (circuit et absorbant). Il suffit donc de tester, lorsque le sommet $x = y$, si la valeur de distance calculée est négative. Si c'est le cas, on provoque le débranchement de la procédure.

2. Là encore l'utilisation n'est pas très compliquée. On utilise la matrice renvoyant les plus petites distances pour chaque couple de sommets (x, y) du graphe. Pour chaque sommet x de 1 à n, on calcule sa valeur d'excentricité en conservant sa plus grande valeur de distance avec les autres sommets (le max de ses plus petites distances). Il ne reste plus alors qu'à comparer les n excentricités calculées (une pour chaque sommet) et de déterminer la plus petite. Le sommet auquel est appartient est le centre du graphe.

Solution 2.5 **(Et le sourire. . .)**

1. Représentation du graphe sous forme de matrice d'adjacence :

	PB	B	A	S	I	F	E
PB		-13	-1				
B	17		11			24	
A	9	-7		8		19	
S			-2		-13	10	
I				17		14	
F		-8	-11	0	-2		1
E						11	

Table 1 – Coûts de passage = T + C - A

La recherche du plus court chemin entre deux sommets donnera le meilleur coût d'exportation.

2. A première vue, le chemin P-B, B, A, S, I, F, E de coût 8 semble le meilleur.
En réalité, le chemin P-B, B, A, S, I, F, A, S, I, F, E est encore meilleur (de coût 6), et on peut tourner sur le circuit A, S, I, F, A, de coût -2, indéfiniment !

3. **Principe de recherche des plus courts chemins d'un sommet s donné vers tous les autres sommets dans un graphe à coûts négatifs avec circuits :**

   L'algorithme de Bellman-Ford relâche chaque arc $N-1$ fois (un plus court chemin ne peut contenir plus de $N-1$ arcs avec N l'ordre du graphe). Il est inutile de tenter de relâcher les arcs sortants d'un sommet dont la distance est encore égale à $+\infty$.

4. *Comment repérer les circuits absorbants ?*
Après avoir relâché tous les arcs $N-1$ fois, on effectue un nouveau test de relâchement pour chaque arc : si une amélioration est encore possible, alors le graphe contient un circuit absorbant.

   Et s'il n'y a plus de circuit absorbant ?
Si lors d'une itération, le relâchement de tous les arcs ne provoque aucune amélioration des distances déjà trouvées, l'algorithme peut s'interrompre.

5. La fonction `Bellman_Ford (t_graph_dyn` G`, entier` s`, t_vect_entiers` $dist$`,` $pere$`) : booleen` calcule les plus courts chemins de s à tous les autres sommets du graphe G. Elle retourne *vrai* si elle détecte un circuit absorbant, *faux* sinon.

```
algorithme fonction Bellman_Ford : booleen
    parametres locaux
        t_graph_dyn      G
        entier           s
    parametres globaux
        t_vect_entiers   dist, pere

    variables
        t_listsom    ps
        t_list_adj   pa
        entier       i, j, cout
        booleen      modif
debut
   pour i ← 1 jusqu'a G.ordre
      pere[i] ← -1
      dist[i] ← +∞
   fin pour
   modif ← vrai
   dist[s] ← 0
   i ← 1
   tant que (i <= G.ordre) et modif faire
      modif ← faux
      ps ← G.lsom
      tant que ps <> NUL faire
          cout ← dist[ps↑.som]
          si cout <> +∞ alors
              pa ← ps↑.succ
              tant que pa <> NUL faire
                  j ← pa↑.vsom↑.som
                  si cout + pa↑.cout < dist[j] alors
                      dist[j] ← cout + pa↑.cout
                      pere[j] ← ps↑.som
                      modif ← vrai
                  fin si
                  pa ← pa↑.suiv
              fin tant que
          fin si
          ps ← ps↑.suiv
      fin tant que
      i ← i+1
   fin tant que
   retourne modif
fin algorithme fonction Bellman_Ford
```

Optimisation ?
On pourrait optimiser un peu cet algorithme en ne traitant, à chaque itération, que les sommets dont la distance a été modifiée à l'itération précédente. On pourrait, par exemple, enfiler les sommets dès que leur distance est modifiée (en faisant attention de ne pas enfiler un sommet déjà présent dans la file !). Si le graphe ne contient pas de circuit absorbant, l'algorithme s'arrêtera lorsque la file sera vide. Dans le cas contraire, il faut conserver le compteur d'itérations (en insérant une "marque de changement de niveau" comme dans le parcours largeur) : si en sortant de la boucle la file n'est pas vide, c'est qu'il y a au moins un circuit absorbant.

Chemin de longueur défini : Une modification de cet algorithme permet d'obtenir des chemins de longueur (en nombre d'arcs) définie. Il suffit pour cela de prendre en compte les distances obtenues à l'itération précédente (et pas celles en cours de calcul, voir l'exemple ici si on prend les sommets dans l'ordre), en utilisant par exemple une matrice : chaque ligne i contient les plus courts chemins de i arcs, calculés à la i^{e}itération. Les distances sont à chaque itération calculées à partir des distances de la précédente (la ligne du dessus !).