

Recherche de chemins optimaux dans les graphes
François-Régis Chaumartin

Fiche technique	2
1 - Introduction.....	2
1.1 Les graphes: quelques définitions.....	2
1.2 Un exemple	2
2 - Représentation des données.....	3
2.1 Graphe = matrice creuse.....	3
2.2 Les tableaux dynamiques en Pascal.....	3
2.3 Les arbres binaires de recherche.....	5
2.4 Représentation des matrices creuses	5
3 - Les algorithmes mis en jeu	6
3.1 Présentation générale	6
3.2 Algorithme de Bellman	7
3.3 Algorithme de Dijkstra	7
3.4 Algorithme de Faure-Bellman	8
4 - Le programme	8
4.1 Représentation des graphes	8
4.2 Implantation des algorithmes	9
4.3 Le programme de démonstration	9
4.4 Une session d'utilisation	9
4.5 Conclusion	10
5 - Applications possibles	10
5.1 Le Métro.....	10
5.2 Le réseau autoroutier	10
6 - La recherche opérationnelle	11

Fiche technique

Résumé: cet article présente les algorithmes de recherche des chemins de "coût" minimal dans un graphe. Un programme met en oeuvre ces algorithmes, en proposant notamment une représentation machine des graphes.

Mots clés: recherche opérationnelle - graphe - Bellman - Dijkstra - Faure-Bellman.

Techniques Pascal: représentation de matrices creuses - gestion de tableaux dynamiques - non contrôle des dépassements d'indice (directive de compilation R-).

Pascal utilisé: Turbo-Pascal version 3 à 6.

Application type: déterminer le meilleur parcours possible entre deux points (celui qui minimise, par exemple, la durée du voyage, ou la distance parcourue).

1 - Introduction

1.1 Les graphes: quelques définitions

Cf fig.1, fig.2, fig.3.

Un graphe est composé de sommets reliés entre eux par des arcs. On associe une valeur à chaque arc, qui correspond au coût pour passer d'un sommet à un autre.

On parle de graphe orienté quand le coût d'un arc dépend du sens du trajet, c'est-à-dire dans le cas où le coût $[i,j]$ peut différer du coût $[j,i]$.

On dit qu'un graphe présente des circuits si on peut y "tourner en rond", c'est-à-dire suivre des trajets de la forme A-B-C-A. On remarque que tout graphe non orienté présente des circuits; en effet, on peut toujours y effectuer A-B-A.

Un circuit absorbant est un circuit dont le coût cumulé des arcs est négatif.

On va chercher ici à appliquer des algorithmes de recherche de chemins optimaux pour trouver les meilleurs chemins, c'est-à-dire ceux qui minimisent le "coût" d'un trajet.

On ne peut donc considérer les graphes à circuits absorbants, car on ne peut y trouver de chemin minimal: en tournant en rond dans un circuit absorbant, on trouve un chemin de valeur toujours plus basse.

1.2 Un exemple

Un voyageur veut se rendre par métro de Porte de Vanves à Ledru-Rollin. Il voudrait déterminer quels sont les meilleurs changements à effectuer et combien de temps durera le voyage.

Le plan du métro est assimilable à un graphe dont les sommets sont les stations de métro; la valeur de l'arc reliant deux stations est le "coût" du voyage entre les deux stations: ce peut être, par exemple, la durée du trajet, ou la distance à parcourir.

La recherche des chemins minimaux fournit à notre voyageur la liste des stations par lesquelles il doit transiter.

2 - Représentation des données

2.1 Graphe = matrice creuse

Considérons un graphe à n sommets, numérotés de 1 à n . Ce qu'il importe de connaître, c'est le coût de tout arc reliant deux sommets. On peut donc représenter le graphe sous la forme d'une matrice carrée d'ordre n , c'est-à-dire un Array [1.. n , 1.. n]. L'élément [i,j] contient le coût du chemin menant du sommet i au sommet j . S'il n'existe pas d'arc [i,j], on convient que ce coût est infini.

Dans cette représentation tabulaire, un graphe à n sommets est donc assimilable à un tableau à n^2 éléments. Mais cela entraîne une occupation mémoire très importante. Examinons le métro parisien: il y a quelque chose comme 300 stations. Ce qui forcerait à gérer un tableau de $300 * 300 = 90.000$ éléments. Si on considère que le coût entre stations est une durée, on peut le coder par un REAL (soit 6 octets), pour arriver à un tableau de $90.000 * 6 = 540.000$ octets.

En fait, on se rend compte que la plupart des stations ne sont reliées qu'à deux autres (celle qui suit et celle qui la précède sur la ligne). Les correspondances entre deux lignes sont reliées à quatre stations. Très peu de stations ont plus de voisines. Considérons que la densité moyenne est de 3 arcs par station. Le métro parisien est donc caractérisé par $3 * 300 = 900$ coûts, soit $900 * 6 = 5.400$ octets.

On manipule donc ici des matrices creuses, c'est-à-dire un tableau carré de nombres, où finalement très peu de cases sont significatives.

Une matrice sera un tableau dynamique de pointeurs vers chaque ligne. Une ligne sera codée par un arbre binaire de recherche, qui permet de retrouver très rapidement l'intersection de la ligne avec une colonne donnée.

Deux types reviennent très souvent dans le programme: TypeIndice est le type des indices de la matrice. C'est donc aussi le type du nom des sommets du graphe. Ainsi, pour un graphe contenant moins de 256 sommets que l'on décide de numéroté, BYTE conviendra parfaitement (à partir de Turbo-Pascal version 4). Si on voulait rentrer les lignes de métro en gardant tels quels les noms des stations, il faudrait le déclarer en tant que String[20], mais au prix d'un gaspillage de la mémoire.

TypeValeur est le type du coût des arcs du graphe. Ce sera généralement un nombre réel. Le type qui convient le mieux est donc un SINGLE (toujours à partir de Turbo-Pascal version 4) qui n'occupe que 4 octets au lieu de 6 pour un REAL.

2.2 Les tableaux dynamiques en Pascal

Créer dynamiquement (pendant l'exécution du programme et non à la compilation) un tableau à une dimension en Turbo-Pascal n'est pas plus compliqué que de déclarer une variable Array. En fait, il suffit de connaître à la compilation quel sera le premier indice du tableau.

Mettons qu'on veuille écrire l'équivalent dynamique de:

VAR MonTableau : ARRAY [1..NombreElements] of TypeElement;

Nous allons procéder en deux temps. D'abord définir deux types: un type tableau abstrait pour indiquer au compilateur l'indice de départ du tableau et le type de base de ses éléments; un type pointeur vers ce type de tableau. Ce qui donne:

**TYPE Tableau = ARRAY[1..1] of TypeElement;
TableauPtr = ^Tableau;**

Ces déclarations de types permettront à Pascal de calculer correctement l'adresse en mémoire de chaque élément du tableau.

Le tableau sera manipulé via un pointeur, qu'on doit déclarer:

VAR MonTableau : TableauPtr;

Deux paramètres suffisent à caractériser complètement toute variable: l'adresse en mémoire et la longueur. Le reste n'est que l'interprétation d'une suite d'octets à travers un typage. Dans le corps du programme, on réserve alors l'espace mémoire nécessaire au tableau (c'est ici que le caractère dynamique apparaît):

GetMem(MonTableau, NombreElements * SIZEOF(TypeElement));

SIZEOF est une précaution utile pour être sûr de réserver juste la taille mémoire requise. Le tableau est alors créé dans le tas. On remarque au passage qu'un autre avantage de cette méthode est de permettre de dépasser la barrière des 64 Ko réservés aux données. Toutefois, une limitation subsiste: chaque tableau ne pourra dépasser individuellement 65.520 octets (limite d'un bloc affecté par GetMem ou New).

Ensuite on manipule MonTableau (presque) comme s'il s'agissait d'un banal Array. MonTableau est un pointeur vers un ARRAY[1..1] of TypeElement. Donc MonTableau^ (notez le ^) est un ARRAY[1..1] of TypeElement. On peut donc légitimement écrire **MonTableau^[1] := 123**. Maintenant, qu'en est-il de **MonTableau^[2]** ? Pascal devrait signaler un débordement d'indice. Et bien, il suffit de précéder chaque accès au tableau par la directive de compilation {\$R-}, ce qui permet de passer outre ces contrôles (et d'accélérer sensiblement l'exécution vu que le programme ne passe plus son temps à faire des contrôles d'éventuels débordements). L'assignation:

{\$R-} MonTableau^[2] := Valeur; {\$R+}

s'exécutera correctement. Pour finir, il faut récupérer en fin de traitement l'espace précédemment alloué avec:

FreeMem(MonTableau, NombreElements * SIZEOF(TypeElement));

Le programme faisant une utilisation intensive de tableaux dynamiques, le meilleur choix était de mettre une directive globale {\$R-} en début de programme plutôt que de multiplier les directives locales.

```

PROGRAM TableauDynamique;
{ prog. utilisant un tableau dynamique }
TYPE
  TabReal=ARRAY[1..1] OF REAL;
  TabRealPtr=^TabReal;
VAR
  MonTab:TabRealPtr;
  Dimension,Indice:INTEGER;
BEGIN
  Write('Dimension:'); ReadLn(Dimension);
  GetMem(MonTab,Dimension*SizeOf(REAL));
  FOR Indice:=1 TO 10 DO
    {$R-} MonTab^[Indice]:=Indice; {$R+}
  FreeMem(MonTab,Dimension*SizeOf(REAL));
END.

```

2.3 Les arbres binaires de recherche

Un abr est un arbre binaire dont chaque noeud contient une clef. Tous les éléments du sous-arbre gauche (resp. droit) ont une clef de valeur strictement inférieure (resp. supérieure). Cette règle s'applique de façon récursive à tous les noeuds.

On peut donc retrouver un noeud particulier avec un temps moyen de recherche proportionnel au logarithme du nombre total de noeuds (sous réserve que l'arbre ne soit pas trop déséquilibré).

```

TYPE
  AbrPtr=^Abr;
  Abr=RECORD
    Clef:TypeClef;
    Valeur:TypeValeur;
    FilsGauche,FilsDroite:AbrPtr;
  END;

```

Le lecteur est invité à se reporter au listing pour découvrir les détails des procédures de création de la racine (AbrInit), d'insertion d'un nouveau noeud (AbrInsere), de recherche d'un noeud existant (AbrCherche) et de destruction d'un arbre (AbrFini).

2.4 Représentation des matrices creuses

C'est finalement le record suivant:

```

TYPE
  Tableau=ARRAY[1..1] OF AbrPtr; { tableau dynamique d'abr }
  TableauPtr=^Tableau;
  Matrice=RECORD
    Dimension:TypeIndice; { nombre de lignes de la matrice }
    Tab:TableauPtr; { tableau de pointeurs vers les lignes }
    Symetrique:BOOLEAN; { True si matrice symétrique }
  END;

```

Ici, pour chaque abr, la clef d'un noeud représentera un numéro de colonne. Trouver l'élément [i,j] d'une matrice se décompose alors en deux temps:

- prendre dans le tableau dynamique le $i^{\text{ème}}$ élément (correspondant à la $i^{\text{ème}}$ ligne) qui pointe vers un abr;
- dans cet abr, effectuer une recherche récursive du noeud de clef j. Si ce noeud existe, renvoyer le coût correspondant, sinon renvoyer +infini pour indiquer que l'élément [i,j] n'a jamais été défini.

Le type matrice contient un drapeau indiquant si elle est symétrique ou pas (élément[i,j]=élément[j,i]). Pour une matrice symétrique, on peut ne stocker que la partie supérieure (les éléments [i,j] tels que $i \leq j$). Ce genre de matrice correspond à un graphe non-orienté, par exemple le réseau auto-routier où chaque segment est à deux voies: la distance entre Paris et Lille est égale à la distance entre Lille et Paris. La circulation dans Paris n'est en revanche pas symétrique, à cause des sens interdits. Cette symétrie éventuelle est mise à profit par les routines MatriceMetElement et MatriceDonneElement.

La fonction MatricePositive indique si tous ses éléments sont positifs ou non. Cela sera utile par la suite, car les graphes à valeurs positives font l'objet d'un traitement spécifique.

3 - Les algorithmes mis en jeu

3.1 Présentation générale

Les trois algorithmes que nous allons présenter ont tous la même structure. Ils nécessitent comme données en entrée:

- un graphe dont chaque arc contient le coût pour aller d'un sommet à un autre;
- le sommet du graphe dont on part, noté S dans la suite;

Ils donnent comme résultat en sortie pour chaque sommet X:

- le coût du chemin minimal qui mène de S à X;
- le sommet qui précède X dans ce chemin: on peut ainsi reconstituer totalement le chemin minimal, en passant par les antécédents successifs.

Chaque type de problème nécessite l'application d'un algorithme bien déterminé. Ainsi, si n est le nombre de sommets du graphe:

Pour un graphe	On applique l'algo de	De	complexité
- sans circuit	Bellman		$O(n^2)$
- à valeurs positives	Dijkstra	$O(n^2)$	$O(n^2)$
- sans circuit absorbant	Faure-Bellman		$O(n^3)$
- avec circuit absorbant	pas de solution!		

3.2 Algorithme de Bellman

Comme le graphe ne contient pas de circuit, on l'explore sommet après sommet, en calculant le meilleur antécédent et en affectant à chaque fois la valeur du chemin minimal.

```

SommetExploré := {S}
ValeurChemin[S] := 0
Antecedent[S] := (pas d'antécédent)

TANT QU'il existe un sommet X non exploré
dont tous les antécédents aient été explorés FAIRE

    ValeurChemin[X] := Min( ValeurChemin[Y] + Coût(Y,X) )
    pour Y appartenant à l'ensemble des antécédents de X

    Soit Y0 qui réalise le minimum, c'est-à-dire tel que:
    ValeurChemin[X] = ValeurChemin[Y0] + Coût(Y0,X)

    SommetExploré := SommetExploré U {X}
    Antecedent[X] := Y0

FIN TANT QUE
  
```

3.3 Algorithme de Dijkstra

On tient à jour l'ensemble des sommets dont la valeur du chemin minimal a été définitivement attribuée. On recalcule périodiquement les valeurs affectées aux autres sommets.

```

ValeurChemin[S] := 0
Antecedent[S] := (pas d'antécédent)

pour tout sommet X du graphe différent de S
    Antecedent[X] := S
    ValeurChemin[X] := Coût(S,X) (+infini si arc inexistant)

Définitif := {S}

TANT QUE il existe des sommets non définitifs

    soit X0 qui réalise le Min(ValeurChemin[X]) pour
    X appartenant à l'ensemble des sommets non définitifs

    Définitif := Définitif U {X0}

    pour tout sommet X non définitif, FAIRE
    SI ValeurChemin[X] > ValeurChemin[X0] + Coût(X0,X)
    ALORS
        ValeurChemin[X] := ValeurChemin[X0] + Coût(X0,X)
        Antecedent[X] := X0
    FIN SI

FIN SI
  
```

FIN TANT QUE

3.4 Algorithme de Faure-Bellman

Pour les graphes avec circuits et valeurs quelconques, on utilise cet algorithme qui a l'inconvénient d'être beaucoup plus long que les autres (complexité $O(n^3)$ au lieu de $O(n^2)$).

A chaque étape m , on calcule $ValeurChemin^{(m)}$ en fonction de $ValeurChemin^{(m-1)}$. On désigne par n le nombre de sommets du graphe.

```
ValeurChemin(1)[S] := 0
Antecedent[S] := (pas d'antécédent)

pour tout sommet X du graphe différent de S
  ValeurChemin(1)[X] := Coût(S,X) (infini si pas d'arc)
  Antecedent[X] := S

POUR m := 1 JUSQUE n-2 FAIRE
  POUR tout sommet X du graphe FAIRE
    ValeurChemin(m+1)[X] := Min( ValeurChemin(m)[X],
      Min( ValeurChemin(m)[Y] + Coût(Y,X) ) )
    pour Y sommet du graphe différent de X
```

4 - Le programme

4.1 Représentation des graphes

Comme nous avons choisi de stocker non seulement la description du graphe, mais aussi le résultat de l'application de l'un des algorithmes, notre type graphe sera finalement:

```
TYPE
  TabInd=ARRAY[1..1] OF TypeIndice;
  TabIndPtr=^TabInd; { tableau dynamique d'indices }

  TabVal=ARRAY[1..1] OF TypeValeur;
  TabValPtr=^TabVal; { tableau dynamique de valeurs }

  Graphe=RECORD
    Mat:Matrice; { la matrice des coûts entre sommets }
    ValeurChemin:TabValPtr; { le coût du chemin optimal }
    Antecedent:TabIndPtr; { le sommet précédent }
  END;
```

4.2 Implantation des algorithmes

Les procédures GrapheBellman, GrapheDijkstra et GrapheFaureBellman reprennent les algorithmes exposés plus haut. De façon générale, ils commencent toujours par déclarer les tableaux locaux dont ils ont besoin.

Un mot sur Faure-Bellman: cet algorithme force à gérer des tableaux des valeurs successives des chemins calculés. Comme à chaque étape m on calcule $ValeurChemin^{(m)}$ en fonction de $ValeurChemin^{(m-1)}$, on peut dans l'implantation de l'algorithme se borner à manipuler deux tableaux seulement. C'est le rôle de `ValChemin:Array[False..True]` et `TabValPtr` associé au booléen `UnSurDeux`, qui permet d'alterner entre les deux tableaux, qui deviennent successivement le prédécesseur l'un de l'autre.

4.3 Le programme de démonstration

Le programme principal lit un graphe orienté préalablement stocké sur disque. Il détermine le type de graphe (avec ou sans circuit, valeurs positives ou quelconque). Puis il demande à l'utilisateur le sommet de départ, et applique l'algorithme idoine.

4.4 Une session d'utilisation

Il faut d'abord créer un graphe sous DOS. Pour ce faire, on peut utiliser son éditeur préféré (celui de Turbo convient parfaitement) ou bien recourir à un:

COPY CON GRAPHE1.DAT

On met d'abord le nombre de sommets du graphe, puis on décrit chaque arc sous la forme d'un triplet: sommet de départ, sommet d'arrivée, coût de l'arc. L'exemple de la fig. 4 donnera:

```
6          { 6 sommets }
1          { arc 1 -> 2 vaut 3 }
2
3
2          { arc 2 -> 3 vaut 3 }
3
3
1          { arc 1 -> 3 vaut 6 }
3
6
(...)
5          { arc 5 -> 6 vaut 5 }
6
5
```

Si on utilise un COPY CON, il ne faudra pas oublier le CTRL+Z rituel final.

Ensuite, l'exécution du programme donne:

Recherche de chemins minimaux dans les graphes
le graphe ne doit pas contenir de circuits absorbants

Fichier contenant le graphe : **graphe1.dat**

Graphe à 6 sommets

Arc (1 , 2) = 3.00

Arc (2 , 3) = 3.00

Arc (1 , 3) = 6.00

Arc (1 , 5) = 3.00

Arc (5 , 3) = 2.00

Arc (2 , 4) = 6.00

Arc (3 , 4) = 1.00

Arc (4 , 5) = 6.00

Arc (4 , 6) = 1.00

Arc (5 , 6) = 5.00

Graphe à valeurs positives, avec circuits,
donc algorithme de Dijkstra

Sommet de départ : 1

De 1 à 2 Coût = 3.00 2 - 1

De 1 à 3 Coût = 5.00 3 - 5 - 1

De 1 à 4 Coût = 6.00 4 - 3 - 5 - 1

De 1 à 5 Coût = 3.00 5 - 1

De 1 à 6 Coût = 7.00 6 - 4 - 3 - 5 - 1

4.5 Conclusion

Le programme a donc affiché le coût des chemins minimaux du sommet 1 à tous les autres sommets, ainsi que ces chemins. Par exemple, pour aller de 1 à 6, il faut suivre 1-5-3-4-6.

5 - Applications possibles

5.1 Le Métro

Le principal problème est de rentrer à la main les lignes station par station et de chronométrer les temps entre les rames. Il faut établir une bijection entre les noms des stations et les sommets du graphe, par exemple en numérotant toutes les stations.

Attention, c'est un graphe orienté! En effet, on ne peut parcourir les stations Michel Ange - Molitor - Chardon - Mirabeau que dans ce sens. Il faut donc penser à doubler chaque ligne en l'entrant dans le sens A-B-C-D... puis ...D-C-B-A sauf pour les quelques exceptions.

5.2 Le réseau autoroutier

Il faut rentrer les distances entre les principales villes reliées par autoroute. Ici, on a (a priori) affaire à un graphe non-orienté, on pourra donc utiliser l'attribut symétrique de la matrice avec `Graphelnit(G, NombreSommets, True)`.

6 - La recherche opérationnelle

Cette branche des mathématiques traite une vaste gamme de problèmes, avec à chaque fois l'objectif d'optimiser une certaine quantité:

- recherche de chemins optimaux d'un point d'un graphe à tous les autres.
- problème du voyageur de commerce: passer chez tous ses clients en parcourant la plus petite distance.
- maximiser le flot de pétrole circulant dans un réseau de pipe-lines.
- gestion optimale des files d'attente, des stocks...
- problèmes d'ordonnancement. Soit un projet décomposé en tâches liées par des contraintes:
 - a) temporelles: la construction ne peut commencer qu'après la livraison du matériel;
 - b) disjonctives: l'utilisation d'une machine ne peut avoir lieu pendant les périodes de révision;
 - c) cumulatives, liées à l'utilisation des ressources disponibles: une équipe de 240 ouvriers ne peut construire simultanément trois bâtiments nécessitant chacun 100 personnes.

Le problème est ici de déterminer la durée minimale de réalisation du projet en déterminant le "chemin critique" de succession des tâches.

L'auteur est élève-ingénieur en deuxième année à l'Institut d'Informatique d'Entreprise. Son temps libre se partage entre la Junior-Entreprise de l'I.I.E. (DIESE) qu'il vice-préside, et l'enseignement de l'informatique en maths sup au lycée Marcelin Berthelot de Saint-Maur des Fossés.