RAIRO. RECHERCHE OPÉRATIONNELLE

C. PRINS

Comparaison d'algorithmes de plus courts chemins sur des graphes routiers de grande taille

RAIRO. Recherche opérationnelle, tome 30, n° 4 (1996), p. 333-357

<http://www.numdam.org/item?id=RO_1996__30_4_333_0>

Recherche opérationnelle/Operations Research
(vol. 30, n° 4, 1996, pp. 333-357)

COMPARAISON D'ALGORITHMES DE PLUS COURTS CHEMINS SUR DES GRAPHES ROUTIERS DE GRANDE TAILLE (*)

par C. PRINS ([1])

Communiqué par C. TAPIERO

Résumé. – *Dans cet article, nous faisons le point sur les algorithmes de la littérature pour le calcul des plus courts chemins dans des grands graphes peu denses à valuations non négatives. Nous présentons ensuite les résultats d'une étude comparant des implémentations efficaces sur PC, appliquées à des graphes de grande taille modélisant des réseaux routiers. Sur des graphes à 15000 sommets, certains algorithmes sont jusqu'à 218 fois plus rapides que l'algorithme classique de Dijkstra.*

Mots clés : Plus court chemin, algorithme, graphe peu dense, micro-ordinateur, réseau routier.

Abstract. – *In this paper we review the algorithms available for computing shortest paths in large sparse graphs with non-negative weights. We then present the results of a study comparing efficient PC-based implementations, executed on large road-like networks. Some algorithms are up to 218 times faster than Dijkstra's classical algorithm when run on graphs with 15000 nodes.*

Keywords: Shortest path, algorithm, sparse graph, microcomputer, road network.

1. INTRODUCTION

Cet article résume une étude [18] comparant des algorithmes efficaces pour les plus courts chemins dans des graphes de grande taille, peu denses, et à valuations positives. Il vise trois buts principaux :

a) Faire connaître au praticien des algorithmes efficaces permettant de dépasser les éternels classiques des livres de Recherche Opérationnelle que sont les algorithmes de Bellman, Dijkstra, Floyd et Ford.

b) Montrer que les progrès en structures de données et la puissance croissante des PC permettent de traiter aujourd'hui des graphes de grande taille sur micro-ordinateur, en des durées raisonnables.

(*) Reçu en juin 1994.

([1]) Département de Productique, École des Mines de Nantes, 4, rue Alfred Kastler, 44070 Nantes Cedex 03, E-mail: prins@auto.emn.fr.

Recherche opérationnelle/Operations Research, 0399-0559/96/04/$ 7.00

c) Calculer rapidement un distancier pour des problèmes de tournées de véhicules. Cette tâche est souvent négligée or, si les distances sont modifiées souvent (par exemple à cause de travaux, d'inondations, ou de barrières de dégel), les logiciels de tournées consacrent en fait l'essentiel de leur temps au calcul du distancier.

Nous définissons d'abord les problèmes de plus courts chemins, et donnons une liste des algorithmes disponibles en indiquant ceux retenus pour l'évaluation. Nous présentons ensuite les algorithmes testés, le protocole d'évaluation sur des grands graphes de type routier, et enfin des tableaux de résultats comparatifs.

2. PROBLÈMES DE CHEMINS OPTIMAUX ET RÉSEAUX ROUTIERS

Considérons un *graphe orienté valué* $G = (X, A, W)$, où X est un ensemble de N nœuds, A un ensemble de M arcs, et $W(i, j)$ un coût entier associé à l'arc (i, j). Dans un contexte de réseau routier, nous supposons que les coûts sont non négatifs (distances, temps de parcours, coût de transport).

Cependant, certains des algorithmes que nous avons testés sont également valables pour des coûts quelconques, nous signalerons cette propriété le moment venu. On s'intéresse au calcul de *chemins de coût minimal* dans G (ou *plus courts chemins*), le *coût d'un chemin* étant la somme des coûts de ses arcs. La littérature distingue 3 types de problèmes :

Pb A : Étant donnés deux sommets s et t, trouver un plus court chemin de s à t.

Pb B : Étant donné un sommet s, trouver un plus court chemin de s vers tout autre sommet.

Pb C : Trouver un plus court chemin entre tout couple de sommets, c'est-à-dire calculer une matrice $N \times N$ appelée *distancier*.

Ces problèmes sont liés. Un algorithme pour A peut être appliqué plusieurs fois pour résoudre B ou C. Un algorithme pour B peut résoudre C si on l'applique à chaque sommet de départ possible. Enfin, des algorithmes pour B comme celui de Dijkstra traitent définitivement un sommet de destination x à chaque itération. Ils peuvent donc résoudre A en étant stoppés dès que $x = t$.

Notre but est de calculer le plus rapidement possible le distancier (problème C), pour des algorithmes d'optimisation de tournées de véhicules. Ce problème est crucial car le distancier doit être recalculé souvent si on modifie des coûts d'arcs. L'examen de bases de données géographiques vendues dans le commerce montre que les graphes routiers sous-jacents

ont les caractéristiques suivantes : une grande dimension (5000 à 30000 sommets), une faible densité (degré moyen des sommets d'environ 3), des valuations positives (distances ou temps de parcours) et occupant une plage de valeurs assez restreinte (1 à 50 km). Ils sont de plus quasi-planaires et quasi-symétriques à une échelle assez grande (régionale ou nationale).

3. ALGORITHMES DISPONIBLES. ALGORITHMES RETENUS

Le tableau I résume les principaux algorithmes connus, avec les types de problèmes et de graphes concernés, la complexité, et des références pour les lecteurs intéressés par plus de détails. Pour des graphes routiers, on peut simplifier les complexités en tenant compte du fait que $M = O(N)$. U désigne le maximum des coûts des arcs.

Pour notre problème C du distancier, nous avons écarté l'algorithme spécifique de Floyd car il exploite mal les faibles densités : par exemple,

TABLEAU I
Principaux algorithmes de chemins optimaux

Problème	Graphe	Auteurs ou nom	Complexité	Références
B, A	W = constante	Exploration en largeur	$O(M)$	2, 3
B, A	$W \geq 0$	Dijkstra, version sans tas	$O(N^2)$	2, 3, 11, 15
B, A	$W \geq 0$	Dijkstra, avec tas	$O(M \,.\, \log N)$	3, 15
B, A	$W \geq 0$	Dijkstra, tas de Fibonacci	$O(M + N \,.\, \log N)$	3, 21
B, A	$W \geq 0$	Algorithmes à buckets	$O(U + N^2)$*	4, 5, 6, 12
B, A	$W \geq 0$	Ahuja *et al.*	$O(M + N \,.\, \log U)$	1
B, A	$W \geq 0$, planaire	Frederickson	$O(N \,.\, (\log N)^{1/2})$**	7
A	$W \geq 0$, euclidien	Sedgewick, Vitter	$O(M \,.\, \log N)$	15, 19
B, A	sans circuit	Bellman+tri topologique	$O(M)$	11
B	quelconque	Bellman	$O(N \,.\, M)$	11
B	quelconque	Algorithme FIFO	$O(N \,.\, M)$	2
B	quelconque	D'Esopo, Pape	exponentiel	16, 20
B	quelconque	Glover *et al.*	$O(N \,.\, M)$	8, 9, 10
C	quelconque	Floyd, Warshall	$O(N^3)$	3, 11

* : Complexité au pire, mais complexité moyenne $O(M + U)$.

** : Cette complexité exploite le fait que $M \leq 3 \,.\, N - 6 = O(N)$ pour un graphe simple planaire.

l'algorithme de Dijkstra avec tas peut construire un distancier ligne par ligne en seulement $O(N.M.log\,N)$. Nous avons ignoré le cas W = constante (peu réaliste) ainsi que l'algorithme insuffisamment performant de Bellman, mais pas l'algorithme de Dijkstra sans tas, pour l'utiliser comme référence et voir les temps prohibitifs qu'obtiendrait un naïf s'avisant de l'utiliser. L'algorithme FIFO a été testé en tant qu'alternative moins connue à l'algorithme de Bellman, celui de d'Esopo et Pape et celui de Glover *et al.* à cause de leur réputation de rapidité en moyenne. Ces trois derniers algorithmes sont encore valides pour des coûts quelconques. Nous présentons maintenant ces algorithmes, après une section précisant les notations et primitives qu'ils utilisent.

4. STRUCTURES DE DONNÉES ET PRIMITIVES

Les sommets de G sont codés par des entiers de 1 à N. G est stocké avec des listes des successeurs, utilisant deux tableaux Tete et TSuc. TSuc concatène les listes de successeurs des sommets 1 à N. Tete[x] désigne l'indice dans TSuc où commence la liste des successeurs de x. Ces successeurs sont donc rangés de TSuc[Tete[x]] à TSuc[Tete[$x+1$]-1]. Cette règle est aussi valable pour le sommet N en « fermant » sa liste en posant Tete[$N+1$]$= M+1$.

Les *valuations* sont rangées dans un tableau W en regard de TSuc. Par exemple, si un successeur y de x est rangé dans TSuc[k], le coût de l'arc (x, y) est dans W[k]. Le *sommet de départ* est noté s (problèmes A et B) et celui d'arrivée t (problème A). Un tableau V donne les *étiquettes* de chaque sommet (valeurs des plus courts chemins de s vers chaque sommet). Pour alléger l'écriture, nous donnons les algorithmes sans stockage des chemins. L'arborescence des chemins trouvés peut être obtenue avec un tableau P donnant le père de chaque sommet. Dans les algorithmes pour les problèmes A et B, il suffit d'ajouter l'instruction $P[y] = x$, quand l'étiquette de y est améliorée en venant d'un sommet x. Pour le problème C, P est une matrice $N \times N$: on stocke dans $P[s, y]$ le père de y dans l'arborescence des plus courts chemins de s vers les autres sommets.

Nous utilisons des piles et des files pour stocker des ensembles de sommets (pour les structures de données de cet article, voir l'excellent livre de Cormen *et al.* [3]). Une *pile* stocke les sommets de manière LIFO (dernier entré-premier sorti) et une *file*, en mode FIFO (premier entré-premier sorti). Nous utilisons une structure mixte pile-file, codée sous forme de liste circulaire dans un tableau de taille N, avec les seules primitives suivantes. Toutes sauf

Clear sont implémentables en temps constant $O(1)$. Clear est en $O(N)$ car elle doit initialiser le tableau pour que Inside fonctionne correctement.

Clear (S)	initialise une pile-file S avec l'ensemble vide;
Push (S, x)	ajoute (empile) le sommet x en tête de S;
EnQueue (S, x)	ajoute (enfile) le sommet x en queue de S;
DeQueue (S, x)	enlève le sommet en tête de S et le renvoie dans x (appelé aussi Pop);
Delete (S, x)	enlève le sommet x de S, même s'il n'est pas en tête ou en queue;
Empty (S)	fonction boolénne vraie si et seulement si la pile-file S est vide;
Inside (S, x)	fonction booléenne vraie si et seulement si x est actuellement dans S.

Nous utilisons aussi des structures de *tas* (voir annexe) pour avoir rapidement le sommet d'étiquette minimale. Un tas permet d'enlever le sommet d'étiquette minimale, de modifier l'étiquette d'un sommet interne, et d'ajouter un nouveau sommet. Voici les primitives qui nous sont utiles :

Clear (H)	initialise le tas H à vide
Insert (H, V, x)	insère le sommet x dans le tas H basé sur les étiquettes du tableau V
MoveUp (h, V, x)	reforme le tas après diminution de la valeur $V[x]$ d'un élément interne x
TakeMin (H,V, x)	enlève le sommet d'étiquette minimale, le renvoie dans x, puis reforme le tas
Inside (H, x)	fonction vraie si et seulement si x est actuellement dans H.
Empty (H)	fonction vraie si et seulement si H est vide.

Inside et Empty sont réalisables en O (1), Clear en O (N) et les autres primitives, en O (log N). Les tas permettent d'implémenter les algorithmes de Sedgewick-Vitter et de Dijkstra en O (M.log N) au lieu de O (N^2). Le *tas de Fibonacci* est une structure plus compliquée [3, 21] qui améliore MoveUp et Insert : le coût au pire est toujours O (log N), mais la complexité moyennée sur la pire suite d'appels (*complexité amortie*) est seulement O (1) grâce à des effets de compensation. Il permet une version de l'algorithme de Dijkstra en $O(M + N.\log N)$.

On peut partitionner une plage de valeurs d'étiquettes entières en B intervalles de taille L, numérotés de 0 à $B-1$, et associer à chacun d'eux un

ensemble de sommets appelé *bucket*. Ce système est souvent codé comme un tableau Buck de B listes. Le bucket Buck $[k]$ stocke des sommets d'étiquettes entre $k \,.\, L$ et $(k+1) \,.\, L - 1$. Au pire, les N sommets vont dans le même bucket, et on serait tenté de déclarer B tableaux de N éléments pour cette structure de données. En fait, il suffit d'un seul tableau Next de N éléments, partagé par les buckets. Next[i] indique le suivant du sommet i dans le même bucket et vaut 0 si i est dernier de son bucket. Le tableau Buck sert alors à indiquer le 1er sommet de chaque bucket.

Le bucket d'un sommet x, accessible en $O(1)$, est Buck[V[x] div L] (div étant la division euclidienne). Un nouveau sommet est insérable en $O(1)$ en tête de bucket. La recherche d'un sommet se fait en $O(N)$ par balayage de son bucket. L'intérêt est la complexité moyenne en $O(N/B)$ si les sommets sont bien répartis. Le système le plus simple a des buckets de largeur 1. Celui de x est alors tout simplement Buck[V[x]]. L'inconvénient est d'avoir autant de buckets que la largeur de l'intervalle des étiquettes.

5. ALGORITHME DE DIJKSTRA ET VERSIONS AVEC TAS

A chaque itération, l'algorithme de Dijkstra fixe définitivement l'étiquette d'un sommet x. On montre qu'un sommet d'étiquette minimale convient et que la complexité est en $O(N^2)$ [2, 3, 11, 15]. Voici en pseudo-code cet algorithme auquel nous donnons le nom de code DIJKSTRA. Un tableau de booléens Done indique les sommets fixés. Si on veut calculer un plus court chemin de s vers un sommet t (Problème A), il suffit d'arrêter dès que t est traité définitivement, par exemple en modifiant le « Jusqu'à » en « Jusqu'à $(\text{VMin} = +\infty)$ ou $(i = t)$ ».

```
Initialiser le tableau V à +∞
Initialiser le tableau Done à Faux
V[s]:=0
Répéter
   {Cherche sommet i de V minimal}
   VMin:= +∞
   Pour j:= 1 à N tel que (non Done(j) et (V[j]<Vmin)
      i    :=j
      VMin :=V[j]
   FP
   Si VMin< +∞ alors                    {Si i existe}
       Done[i] :=Vrai                   {On le fixe}
       Pour k :=TETE[i] à TETE[i+1] -1 {On met à jour ses successeurs}
```

```
        j :=TSUC[k]
        Si V[i]+W[k]<V[j] alors V[j] :=V[i]+W[k]
      FP
   FS
Jusqu'à VMin = +∞.
```

Le Répéter effectue au plus N itérations si tous les sommets sont accessibles au départ de s, la première boucle Pour interne est en $O(N)$, et la deuxième en $O(d^+(i))$. Le travail cumulé du premier Pour sur les itérations du Répéter est donc $O(N^2)$, tandis que celui du deuxième est égal à la somme des demi-degrés extérieurs des sommets de G, soit M. L'algorithme de Dijkstra est donc en $O(N^2)$.

Dans la version avec tas en $O(M \cdot \log N)$, de nom de code DIJTAS, un tas H stocke les sommets non fixés d'étiquette non infinie. La boucle de recherche de i et le tableau Done sont remplacés par un TakeMin, qui extrait i et reforme le tas. Puis, pour tout successeur j amélioré, on insère j dans le tas s'il n'y est pas déjà, sinon on fait un MoveUp car son étiquette a diminué.

Comme dans la version sans tas, le nombre cumulé d'itérations du Pour est en $O(M)$, mais ici chaque itération effectue un TakeMin ou un MoveUp coûtant chacun $O(\log N)$. Ceci explique la complexité en $O(M \cdot \log N)$. Voici l'algorithme pour les problèmes A et B (pour B, il faut appeler l'algorithme avec $t = 0$). La version à tas de Fibonacci, de nom de code DIJFIB, a le même texte mais il faut bien sûr modifier le contenu des primitives pour gérer un tas de Fibonacci (*voir* [3, 21] pour ces primitives).

```
Initialiser le tableau V à +∞
V[s] :=0
Clear (H)                          {Initialise le tas à vide}
Insert (H,V,s)                     {Et insère s}
Répéter
   TakeMin (H,V,i)                 {Enlève sommet de V minimal, i}
   Pour k :=TETE[i] à TETE[i+1] -1 {Développe les successeurs de i}
      j :=TSUC[k]
      Si V[i] + W[k] < V[j] alors {Amélioration de V[j] en passant
                                  par i}
         V[j] := V[i]+W[k]
         Si non Inside (H,j) alors {j n'est pas dans le tas, on
                                   l'ajoute}
            Insert (H,V,j)
         Sinon                     {j est dans le tas, on le fait
                                   monter}
```

```
            MoveUp  (H,v,j)
          FS
       FS
    FP
Jusqu'à Empty (H) ou (i = t).
```

6. DÉRIVÉS A BUCKETS DE L'ALGORITHME DE DIJKSTRA

Il s'agit d'algorithmes de type Dijkstra utilisant un système de buckets. Nous avons d'abord testé un algorithme à buckets de largeur 1 proposé à l'origine par Dial [5] et Denardo et Fox [4], auquel nous donnons le nom de code BUCK1. Comme les sommets d'un bucket ont même étiquette, on peut prendre le premier pour développement. On peut montrer qu'à tout instant les étiquettes des sommets non fixés ont une plage de valeurs de largeur $1 + U$, U étant le maximum des coûts des arcs. Un tableau Buck de $B = 1 + U$ buckets suffit s'il est réutilisé circulairement. La complexité au pire est $O(U + N^2)$, mais elle est en pratique en $O(U + M)$ si les étiquettes sont bien dispersées.

L'indice du bucket actuel est CB (*current bucket*), il varie de 0 à U puis repasse à 0. La variable LB (*last bucket*) mémorise le dernier numéro de bucket consulté. La fin de l'algorithme est détectée quand CB fait un tour complet et repasse par LB sans trouver de bucket non vide. Les buckets sont des piles-files comme décrites en section 4, avec leurs primitives EnQueue et Delete. Dans le texte suivant de l'algorithme, l'opération a mod b désigne le reste de la division entière de a par b.

```
Initialise les B buckets à vide
Initialise le tableau V à +∞
V[s] := 0
EnQueue (Buck[0],s)                          {Met s dans le bucket 0}
CB :=0                                       {n° bucket actuel}
Fin :=Faux                                   {Indicateur de fin}
Répéter
  Si Buck[CB] = 0 alors                      {Le bucket actuel est vide}
     LB := CB                                {Garde n° bucket actuel}
     Répéter                                 {Cherche 1er bucket non
                                              vide}
      CB := (CB + 1) mod (U+1)
    Jusqu'à (Buck[CB] > 0) ou (CB = LB)
    Si CB = LB alors Fin := Vrai             {Tous les buckets sont
                                              vides: fin}
```

```
  FS
  Si non Fin alors
     DeQueue (Buck[CB],i)                    {Enlève 1er noeud bucket CB}
     Pour k := TETE[i] à TETE[i+1] -1        {Développement des succes-
                                              seurs de i}
       j := TSUC[k]
       Si V[i] +W[k] < V[j] alors             {Améliore V[j] en passant
                                              par i}
         Si V[j] < +∞ alors                   {j est déjà dans un bucket,
                                              OldB}
            OldB := (CB + V[j] - V[i]) mod (U+1)
            Delete (Buck[OldB],j)             {Enlève j du bucket OldB}
         FS
         V[j] := V[i] + W[k]
         NewB := (CB+W[k]) mod (U+1)          {Met j dans son nouveau
                                              bucket}
         EnQueue (Buck[NewB],j)
       FS
    FP
  FS
Jusqu'à Fin ou (i = t)                  {Mettre t à 0 pour le problème B}
```

Nous avons aussi implémenté sous le nom de BUCKL une version due à Hung et Divoky [6, 12], qui utilise un nombre fixe de buckets B et qui calcule L à l'initialisation de façon que $B \cdot L \geq 1+U$. D'après leurs résultats, l'algorithme a une durée d'exécution stable entre $B = 100$ et $B = 500$. Pour une valeur de B inférieure à 100, l'algorithme perd du temps à balayer le contenu des buckets pour trouver le sommet d'étiquette minimale. Pour une valeur supérieure à 500, il perd du temps à consulter des buckets vides. Son avantage est un besoin en mémoire indépendant de U, contrairement à BUCK1. Nous ne donnons pas son texte, qui se déduit aisément de BUCK1. La principale différence est que des sommets d'étiquettes différentes peuvent coexister dans le même bucket. Il faut donc balayer le bucket CB pour trouver le sommet i d'étiquette minimale, et l'enlever avec un Delete.

Nous avons enfin codé sous le nom de AHUJA un algorithme plus récent, dû à Ahuja *et al.* [1], que nous ne pouvons détailler ici par manque de place. Il utilise une structure de données tenant à la fois du tas et du bucket, appelée tas redistributif (r-heap). Contrairement aux deux algorithmes précédents, il y a $O(\log U)$ buckets, et leur largeur double quand on passe d'un bucket à celui d'indice immédiatement supérieur. Le bucket 0 contient les sommets de valeur minimale. Quand il est vide, on change dynamiquement les intervalles des buckets et on redistribue les sommets du premier bucket non vide i dans

les buckets 0 à $i-1$. Le système s'apparente à un tas car ses buckets « tamisent » de plus en plus finement les valeurs quand on descend vers le bucket 0.

L'intérêt de cet algorithme est d'avoir un temps de calcul au pire très amélioré par rapport à BUCK1 et BUCKL : il est en $O(M+N.\log U)$ dans le cas général, et donc en $O(N.\log U)$ dans le cas des graphes routiers où $M = O(N)$. Le r-tas que nous avons vu est dit à un niveau. En utilisant un r-tas dit à 2 niveaux où les buckets se composent de sous-buckets, on peut définir un algorithme plus compliqué en $O(M+N.\log U/(\log\log U))$, que nous n'avons pas testé.

7. ALGORITHME DE SEDGEWICK-VITTER

Cet algorithme [15, 19] de type Dijkstra concerne le problème A dans des graphes euclidiens, c'est-à-dire des graphes non orientés dont les sommets sont des points d'un espace euclidien et les arêtes, des segments entre points valués par leur longueur. Pour chaque sommet non fixé i, il maintient un minorant $\mathrm{B}[i]$ du coût d'un plus court chemin de s à t passant par i : $\mathrm{B}[i] = \mathrm{V}[i] + \mathrm{D}(i, t)$, $\mathrm{D}(i, t)$ désignant la distance euclidienne de i à t. Les coordonnées nécessaires pour ces distances sont fournies avec les bases de données routières du commerce.

Voici une version en $O(M.\log N)$ de l'algorithme de Sedgewick-Vitter basée sur un tas, à laquelle nous donnons le nom de code SEDVIT. Par rapport à Dijkstra, on fixe à chaque itération le sommet de minorant minimal, et le tas *H* est donc classé selon les valeurs de *B* au lieu de *V*. L'algorithme est aussi bien plus rapide en moyenne : les sommets fixés avant t forment un fuseau plus ou moins étroit de s à t et ne représentent qu'une petite partie des sommets de *G*.

```
Initialiser le tableau V à +∞
V[s] :=0
B[s] := D(s,t)
Clear (H)                          {Initialise le tas à vide}
Insert (H,B,s)                     {Et y insère s}
Répéter
  TakeMin (H,B,i)                  {Enlève sommet de B minimal, i}
  Pour k := TETE[i] à TETE[i+1]-1  {Développement des successeurs de
                                   i}
    j := TSUC[k]
```

```
    Si V[i] + W[k] < V[j] alors    {Amélioration de V[j] en passant
                                    par i}
       V[j] := V[i]+W[k]
       B[j] := V[j]+D(j,t)         {Rafraichit l'évaluation}
       Si non Inside (H,j) alors   {j n'est pas dans le tas, on
l'ajoute}
        Insert (H,B,j)
      Sinon                        {j est dans le tas, on le fait
                                    monter}
       MoveUp (H,B,j)
      FS
    FS
  FP
Jusqu'à Empty (H) ou (i = t).
```

Une preuve de cet algorithme est donnée dans [17] et [19], notons qu'il s'agit en fait d'une forme spécialisée de l'algorithme A* [14]. Nous avons montré dans [18] que l'algorithme est encore valide pour tout graphe à valuations positives, pour lequel on dispose d'un minorant $D(i, j)$ du coût $L(i, j)$ d'un plus court chemin de i à j, vérifiant l'inégalité triangulaire. C'est le cas des graphes routiers valués par des kilomètres : bien que ces valuations ne vérifient pas en général l'inégalité triangulaire, le minorant représentée par la distance euclidienne la vérifie.

8. ALGORITHME FIFO

Cet algorithme très simple, dérivé de celui de Bellman [2], stocke dans une file $\mathcal{Q}$ les sommets *ouverts* (qui viennent d'être améliorés). Contrairement à l'algorithme de Dijkstra, FIFO accepte des valuations négatives et peut développer plusieurs fois un sommet i. $V[i]$ peut donc varier jusqu'à la fin, et FIFO n'est pas accélérable pour résoudre le problème A. La complexité est en $O(N.M)$ et est donc sensible à la densité du graphe. Nous avons testé cet algorithme sous le nom de FIFO. A noter : une stratégie LIFO au lieu de FIFO (remplacement de $\mathcal{Q}$ par une pile) donne un algorithme non polynômial sur certains graphes, bien qu'étant aussi rapide en moyenne.

```
Initialise le tableau V à +∞
V[s] := 0
Clear (Q)
EnQueue (Q,s)
Répéter
```

```
   DeQueue (Q,i)
   Pour k := TETE[i] à TETE[i+1]-1
      j := TSUC[k];
      Si V[i] + W[k] < V[j] alors
         V[j] := V[i]+W[k]
         EnQueue (Q,j)
      FS
   FP
Jusqu'à Empty (Q).
```

9. ALGORITHME DE D'ESOPO ET PAPE

Cet algorithme [16, 20] utilise une pile-file NEXT comme décrit en 4, c'est-à-dire une file où on peut faire l'ajout en queue (EnQueue) ou en tête (Push). Un sommet atteint la 1^{re} fois est mis en fin de file. Si on l'atteint encore par la suite, on l'insère en tête de file, à condition qu'il ne soit pas déjà dans la file. Ce critère de gestion est basé sur l'idée intuitive suivante : quand on atteint pour la première fois un sommet par un chemin provisoire, il n'est pas urgent de développer ses successeurs car les chemins obtenus risquent d'être mauvais dans un premier temps. Comme l'algorithme LIFO, cet algorithme n'est pas polynômial : il a un comportement exponentiel sur certains graphes, heureusement improbables en pratique. Mais il est réputé pour sa très bonne performance moyenne sur les graphes peu denses. Nous l'avons testé sous le nom de code ESOPO.

```
Initialiser le tableau V à +∞
V[s] := 0
Clear (NEXT)
EnQueue (NEXT,s)
Répéter
  DeQueue (NEXT,x)                {prélève le sommet en tête de file}
  Pour k :=TETE[x] à TETE[x+1]-1
    y := TSUC[k]
    Si V[x] + W[k] < V[y] alors
       V[y] := V[x] + W[k]
       Si V[y] = +∞ alors         {y n'est jamais allé dans NEXT}
        EnQueue (NEXT,y)          {ajoute y en fin de file}
       Sinon Si non Inside(NEXT,y) alors
         {y a été dans NEXT, mais n'y est plus, ajout en tête de NEXT}
         Push (NEXT,y)
       FS
     FS
```

```
   FP
Jusqu'à Empty (NEXT).
```

10. ALGORITHME À SEUIL

Glover *et al.* [9] ont conçu en 1984 un algorithme général en $O(N.M)$ appelé PSP (*Partionned Shortest Path*), contenant en cas particulier les algorithmes FIFO et de Dijkstra. Il partitionne les sommets en deux ensembles Now (sommets « urgents ») et Next (sommets « non urgents »). Au début, Now contient seulement s et Next est vide. Les sommets de Now sont d'abord traités dans un ordre quelconque, et tout successeur amélioré est placé dans Next. Quand Now est vide, on transfère le contenu de Next dans Now et on recommence, jusqu'à ce que Next soit vide.

Les algorithmes à seuil (*threshold algorithms*) sont des versions de PSP où les sommets transférés de NEXT à NOW sont tous ceux d'étiquettes inférieure ou égale à un seuil T fixé [8, 9, 10]. Ce seuil est ajusté dynamiquement en cours d'algorithme. Nous avons choisi d'implémenter la version THRESH-X décrite en [8] comme la plus rapide en moyenne sur des graphes pas trop denses. Nous ne pouvons détailler dans cet article cet algorithme un peu long et complexe.

11. PROTOCOLE D'ÉVALUATION

11.1. Langage et machine

Les algorithmes ont été programmés en Borland Pascal 7, mode protégé, et exécutés sur un compatible PC équipé d'un 80486-DX à 33 MHz. Toujours sur PC, on pourrait diviser les durées fournies plus loin par 2 avec un 80486 à 66 MHz, et encore par 2 avec un langage disposant d'un compilateur 32 bits fortement optimiseur (Fortran 77 de Watcom, ADA d'Alsys par exemple).

11.2. Modèle non euclidien

Nous avons défini un modèle dit « non euclidien » pour simuler des graphes routiers à l'échelle régionale ou nationale pour tous les algorithmes sauf celui de Sedgewick-Vitter. A cette échelle, les graphes routiers du commerce sont planaires, symétriques, et ont un degré des sommets valant seulement 3 en moyenne. Si on tente de produire aléatoirement des graphes de ce type avec des degrés variables, on obtient le plus souvent des graphes peu réalistes car non connexes et non planaires.

La génération de graphes planaires de degré moyen imposé étant délicate, nous avons utilisé des réseaux à mailles hexagonales, planaires par définition, où tout sommet non périphérique a un degré 3. Des comparaisons avec des sous-graphes extraits de bases de données routières montrent la pertinence de ce modèle pour les durées d'exécution des algorithmes. Nous avons construit de tels réseaux avec le même nombre d'hexagones horizontalement et verticalement. La *figure* 1 montre un réseau avec $N = 18$ sommets et 2×2 hexagones.

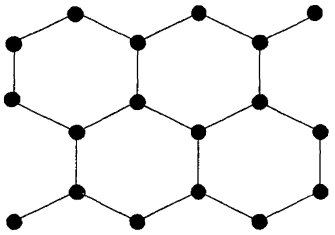

Figure 1. – Exemple de réseau avec N=18 et 2×2 hexagones.

Nos tests utilisent des coûts d'arcs tirés au sort selon une loi uniforme entre 1 et 1000 inclus pour éviter des étiquettes énormes et partent d'un sommet le plus central possible, ce qui s'est révélé le pire cas pour les algorithmes. Des intervalles de coûts différents affectent très peu les temps de calcul, sauf celles des trois algorithmes BUCK1, BUCKL et AHUJA conçus pour exploiter les faibles plages de variation des coûts.

11.3. Modèle euclidien

Comme indiqué en 7, l'algorithme de Sedgewick-Vitter pour le problème A se généralise aux coûts non euclidiens, mais utilise toujours la distance euclidienne comme minorant pour se diriger vers le sommet de destination. On considère un plan maillé par des carrés comme dans la *figure* 2. Les carrés hachurés sont interdits pour le placement des points.

Notre modèle euclidien considère des carrés de côté 1000 et tire au sort un point par carré permis. On relie ensuite ces points pour obtenir un graphe planaire formé d'hexagones déformés. La distance euclidienne $D(i, j)$ est utilisée comme minorant. La valeur $W(i, j)$ de tout arc (i, j) est tirée au sort dans un intervalle $[D(i, j), K \cdot D(i, j)]$, $K \geq 1$. Si $K = 1$, on a un vrai graphe euclidien. Dans les réseaux réels, le coût d'un arc (i, j) est en

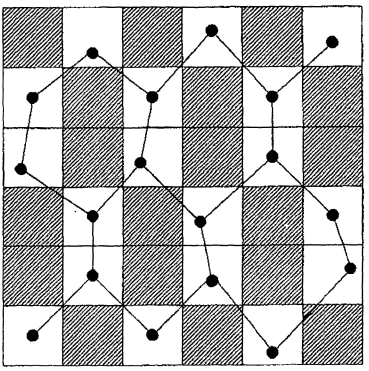

Figure 2. – Réseau hexagonal déformé du modèle euclidien.

moyenne 10 % supérieur à la distance euclidienne entre i et j. On a donc pris $K = 1,2$ pour avoir *en moyenne* $W(i, j) = 1.1 \times D(i, j)$.

11.4. Détail des tests

Le modèle non euclidien a été utilisé pour tester tous les algorithmes sauf SEDVIT sur des graphes à 1000, 5000, 10000 et 15000 sommets. Les algorithmes ont été codés pour calculer les plus courts chemins à partir d'un sommet s le plus au centre des réseaux hexagonaux (problème B). Le tableau II donne les temps de calcul moyen sur 20 graphes générés aléatoirement. Les algorithmes se sont montrés stables, et un nombre supérieur d'exécutions n'est donc pas nécessaire. Le tableau III donne les facteurs de vitesse correspondants par rapport à l'algorithme de Dijkstra avec tas binaire (DIJTAS), choisi comme référence (coefficient 1). Pour avoir la durée de calcul d'un distancier (problème C), il faut multiplier les durées par le nombre N de sommets.

Pour mesurer l'influence de la densité, on a comparé les algorithmes sur des graphes à 5000 sommets de densité moyenne 10, 15, 20 et 30 %, pour lesquels on a abandonné le modèle hexagonal. Les durées moyennes sur 20 exécutions sont consignées dans le tableau IV. Les algorithmes à buckets, sensibles au coût maximal U des arcs, ont été évalués sur des graphes à 5000 et 10000 sommets du modèle euclidien, mais avec $U = 1000$, 4000, 8000, et 16000. Les durées moyennes sur 20 exécutions sont données dans le tableau V.

Nous avons enfin comparé l'algorithme de Sedgewick-Vitter (SEDVIT) pour le problème A avec l'algorithme de Dijkstra avec tas (DIJTAS), sur le

modèle euclidien avec 10000 sommets. Pour chaque algorithme et $K = 1$, 1,2, 1,5 et 2, on a tiré au sort 100 paires de sommets distincts s et t et on a enregistré le temps moyen de calcul par paire ainsi que le nombre de sommets ayant reçu leur étiquette définitive. Les résultats sont donnés par le tableau VI.

12. RÉSULTATS ET COMMENTAIRES

12.1. Modèle non euclidien. Tableaux II et III

L'algorithme classique DIJKSTRA qu'un profane choisirait se révèle très lent : pour 15000 sommets, il est 115 fois plus lent que l'algorithme de référence DIJTAS et 218 fois plus lent que l'algorithme le plus rapide, BUCK1. La version avec tas DIJTAS, choisie comme référence, est très rapide : sa durée d'exécution est quasi linéaire en N pour les graphes étudiés. Les algorithmes à complexité théorique séduisante (AHUJA avec tas redistributif et DIJFIB avec tas de Fibonacci) sont finalement moins bons que DIJTAS en moyenne.

ESOPO est remarquablement rapide, plus même que DIJTAS. De plus, il a une grande simplicité de programmation. L'algorithme FIFO est jusqu'à 3 fois plus lent que DIJTAS. THRESH-X est un peu meilleur qu'ESOPO.

TABLEAU II

Modèle non euclidien : réseau hexagonal, sommets internes de degré 3, coûts dans [1,1000]. Problème B : plus cours chemins du sommet le plus central vers tous les autres. Durées d'exécution en secondes, moyennes sur 20 exécutions.

ALGORITHME	N =1000	N =5000	N =10000	N =15000
DIJKSTRA	0,94	23,45	94,20	227,22
DIJTAS	0,11	0,60	1,26	1,98
DIJFIB	0,33	1,68	3,47	5,16
FIFO	0,11	1,35	3,27	6,26
ESOPO	0,11	0,50	0,94	1,42
THRESH-X	0,11	0,49	0,92	1,38
BUCK1	0,05	0,33	0,71	1,04
BUCKL (B=256)	0,06	0,38	0,77	1,16
AHUJA	0,20	0,92	1,84	2,75

TABLEAU III

Modèle non euclidien : réseau hexagonal, sommets internes de degré 3, coûts dans [1,1000]. Problème B : plus cours chemins du sommet le plus central vers tous les autres. Coefficients de vitesse par rapport à DIJTAS.

ALGORITHME	N =1000	N =5000	N =10 000	N =15 000
DIJKSTRA	8,55	39,08	74,76	114,76
DIJTAS	1,00	1,00	1,00	1,00
DIJFIB	3,00	2,80	2,75	2,60
FIFO	1,00	2,25	2,60	3,16
ESOPO	1,00	0,83	0,75	0,72
THRESH-X	1,00	0,82	0,73	0,70
BUCK1	0,45	0,55	0,56	0,53
BUCKL (B=256)	0,55	0,63	0,61	0,59
AHUJA	1,82	1,53	1,46	1,39

Ces performances sont cependant remarquables pour des algorithmes conçus pour le cas général de valuations quelconques.

La bonne surprise vient des algorithmes à buckets. BUCK1 est le plus rapide des algorithmes testés, presque 2 fois plus rapide que DIJTAS, mais il pourrait nécessiter trop de mémoire pour de grandes valeurs de U. BUCKL, à peine plus lent, serait alors plus intéressant grâce à son nombre fixe de buckets, indépendant de U. Notre version de BUCKL utilise 256 buckets, mais nous avons pu vérifier que le temps de calcul varie très peu entre 100 et 500 buckets. Notons que plusieurs algorithmes ne sont pas départageables pour 1000 sommets, la raison étant une précision insuffisante du *timer* MS-DOS pour d'aussi courtes durées d'exécution.

12.2. Influence de la densité. Tableau IV

Le tableau IV résume des tests à 5 000 sommets ayant pour but de savoir si les bons algorithmes sur les réseaux routiers ont un intérêt plus général. La colonne $d = 3$ concerne les graphes de type routier, elle est recopiée des résultats du tableau II. Les autres graphes sont générés aléatoirement et ont des degrés variables selon les sommets, mais égaux en moyenne à d. On constate que l'algorithme DIJKSTRA reste très lent, mais est peu affecté par les variations de densité. Les autres algorithmes se dégradent plus ou

TABLEAU IV

Influence du degré d des sommets pour $N = 5000$ et des coûts des arcs dans [1, 1000] $d = 3$: réseaux à mailles hexagonales, $d > 3$: graphes aléatoires de degré moyen d. Problème B : plus cours chemins du sommet le plus central vers tous les autres durées en secondes, moyennes sur 20 exécutions.

ALGORITHME	$d = 3$	$d = 10$	$d = 15$	$d = 20$	$d = 30$
DIJKSTRA	23,45	25,27	25,76	26,25	28,34
DIJTAS	0,60	1,43	1,87	2,31	3,13
DIJFIB	1,65	3,19	3,68	4,12	4,96
FIFO	1,48	2,58	3,95	4,89	7,39
ESOPO	0,50	2,59	4,23	5,66	9,90
THRESH-X	0,49	2,14	3,73	4,83	7,14
BUCK1	0,33	0,99	1,37	1,76	2,59
BUCKL	0,39	1,32	1,86	2,42	3,55
AHUJA	0,92	1,48	1,88	2,30	3,04

moins rapidement. En particulier, ESOPO et THRESH-X ne sont plus du tout intéressants pour $d = 30$. BUCK1 reste plus rapide que DIJTAS, mais pas BUCKL. AHUJA avec son tas redistributif commence à battre DIJTAS. La version à tas de Fibonacci DIJFIB devient peu à peu compétitive, elle bat même DIJTAS à partir de $d \approx 70$ mais on sort alors des graphes de type routier.

12.3. Influence des coûts. Tableau V

Le tableau V montre l'influence de U sur les algorithmes sensibles à ce paramètre. L'algorithme de référence DIJTAS est par construction insensible. BUCK1, le meilleur pour $U = 1000$, devient moins bon que DIJTAS pour $U = 16\,000$. AHUJA se dégrade également. La bonne surprise vient de BUCKL. Cette version à nombre fixe de buckets (256) est remarquablement stable, ce qui confirme les conclusions de Hung et Divoky [6, 12].

12.4. Modèle euclidien. Tableau VI

Le modèle décrit en 11.3 sert à comparer DIJTAS et l'algorithme de Sedgewick-Vitter SEDVIT. Les graphes considérés ont 10 000 sommets, et on résout une série de problèmes A en tirant au sort des paires de sommets

TABLEAU V

Influence du coût maximal des arcs U
Modèle non euclidien : réseaux à mailles hexagonales, sommets internes de degré 3.
Problème B : plus courts chemins du sommet le plus central vers tous les autres.
Durées en secondes, moyennes sur 20 exécutions.

N	5000				10000			
U	1000	4000	8000	16 000	1000	4000	8000	16 000
DIJTAS	0,60	0,60	0,61	0,61	1,26	1,25	1,23	1,22
BUCK1	0,33	0,49	0,60	0,88	0,71	0,78	0,94	1,32
BUCKL	0,38	0,38	0,39	0,39	0,66	0,65	0,66	0,66
AHUJA	0,92	1,04	1,08	1,12	1,84	2,04	2,17	2,25

TABLEAU VI

Modèle euclidien, hexagones déformés dans le plan, $N = 10000$, sommets internes de degré 3.
Problème A : calcul d'un plus court chemin entre 2 sommets tirés au sort s et t.
Durées en secondes, moyennes sur 100 requêtes.

K	Algorithme	Durée	Étiquettes fixées
1,0	DIJTAS	0,84	6704
	SEDVIT	0,24	1479
1,2	DIJTAS	0,82	6551
	SEDVIT	0,28	1792
1,5	DIJTAS	0,78	6254
	SEDVIT	0,31	2034
2,0	DIJTAS	0,74	5937
	SEDVIT	0,35	2331

(s, t). Par rapport au problème B (plus courts chemins de s vers tout autre sommet), DIJTAS effectue 59 à 67 % du travail : il obtient en effet en sous-produit des chemins optimaux vers de nombreux sommets. Un essai de visualisation graphique de la propagation de l'algorithme nous a montré qu'il fixe *grosso-modo* les sommets contenus dans un cercle de rayon $[s, t]$. Quand K augmente, le travail effectué en moyenne par l'algorithme diminue légèrement, pour une raison inconnue.

Dans l'optique de ses auteurs Sedgewick et Vitter, SEDVIT a été conçu uniquement pour les graphes véritablement euclidiens ($K = 1,0$). Dans ce cas, l'algorithme ne fixe que 14 % des sommets en moyenne, soit 4,5 fois moins que DIJTAS. Le rapport des temps d'exécution n'est que de 3,5, à cause des calculs de distances nécessitant des nombres réels. Sur un 486-SX, sans coprocesseur arithmétique, l'algorithme est même à peine plus rapide que DIJTAS. Graphiquement, l'algorithme fixe les sommets dans un losange plus ou moins étroit ayant s et t pour sommets opposés.

Comme nous l'avons démontré, SEDVIT est en fait valide pour tout graphe dont les coûts posittifs disposent d'un minorant vérifiant l'inégalité triangulaire. Les coûts eux-mêmes n'ont pas besoin de vérifier cette inégalité triangulaire. C'est le cas des réseaux routiers réels, où la distance à vol d'oiseau est inférieure ou égale à celle parcourue sur le terrain. Le tableau montre que SEDVIT se dégrade quand K augmente, mais il reste encore 2 fois plus rapide que DIJTAS pour $K = 2$. Pour des réseaux routiers réels en plaine, K vaut environ 1,2 et l'algorithme est 3 fois plus rapide.

13. CONCLUSION

En conclusion, il existe pour des graphes peu denses comme les réseaux routiers des algorithmes jusqu'à 218 fois plus rapides que l'algorithme classique de Dijkstra. L'algorithme le plus rapide est celui à buckets de largeur 1, BUCK1 : il calcule en 1 seconde sur PC les plus courts chemins d'origine s dans des réseaux à 15 000 sommets. Nous avons vérifié sur quelques tests que ce temps est divisible par 4 avec un PC à 66 MHz et un langage disposant d'un compilateur 32 bits.

BUCK1 est de plus facile à programmer. Des alternatives un peu plus lentes, mais tout aussi faciles à coder sont BUCKL et ESOPO, THRESH-X est également rapide, mais d'implémentation nettement plus compliquée. On constate que les algorithmes de meilleure complexité théorique (AHUJA et DIJFIB) sont battus en moyenne par des algorithmes exponentiels (ESOPO) ou en $O(N^2)$ (BUCK1 et BUCKL)!

Quant à l'algorithme de Sedgewick-Vitter, c'est le plus rapide pour répondre à des requêtes ponctuelles de plus court chemin entre 2 points. Il n'est cependant pas assez rapide pour calculer les distances au coup par coup dans un algorithme d'optimisation de tournées, et ainsi se passer de distancier. Une utilisation possible reste le calcul de distances entre deux villes, en mode interactif.

RÉFÉRENCES

1. R. K. AHUJA, K. MELHORN, J. B. ORLIN et R. E. TARJAN, Faster algorithms for the shortest path problem, *Technical Report* 193, MIT Operations Research Center, 1988.
2. D. BEAUQUIER, J. BERSTEL et Ph. CHRÉTIENNE, *Éléments d'algorithmique,* Masson, 1992.
3. T. H. CORMEN, C. L. LEISERSON, R. L. RIVEST, Introduction to algorithms, *The MIT Press,* 1992.
4. E. DENARDO et B. FOX, Shortest-route methods. 1. Reaching, pruning and buckets, *Operations Research,* 1979, *27*, p. 161-186.
5. R. DIAL, Algorithm 360: Shortest path forest with topological ordering, *Communications of the ACM*, 1969, *12*, p. 632-633.
6. J. J. DIVOKY, M. S. HUNG, Performance of shortest path algorithms in network flow problems, *Management Science*, 1990, *36*, (6), p. 661-673.
7. G. N. FREDERICKSON, Fast algorithms for shortest paths in planar graphs with applications, *SIAM Journal on Computing*, 1987, *16*, (6), p. 1004-1022.
8. F. GLOVER, R. GLOVER et D. KLINGMAN, Computational study of an improved shortest path algorithm, *Networks*, 1984, *14*, p. 25-36.
9. F. GLOVER, D. KLINGMAN et N. V. PHILLIPS, A new polynomially bounded shortest path algorithm, *Operations Research*, 1985, *33*, (1), p. 65-73.
10. F. GLOVER, D. KLINGMAN, N. V. PHILLIPS et R. F. SCHNEIDER, New polynomial shortest path algorithms and their computational attributes, *Management Science*, 1985, *31*, (9), p. 1106-1129.
11. M. GONDRAN et M. MINOUX, *Graphes et Algorithmes*, Eyrolles, 1979.
12. M. S. HUNG et J. J. DIVOKY, A computational study of efficient shortest path algorithms, *Computers and Operations Research,* 1988, *15*, (6), p. 567-576.
13. D. KLINGMAN, R. F. SCHNEIDER, Microcomputer-based algorithms for large scale shortest path problems, *Discrete Applied Mathematics*, 1986, *13*, p. 183-206.
14. M. MINOUX, *Programmation mathématique,* Masson, 1983 (2 volumes).
15. J. A. MCHUGH, *Algorithmic graph theory*, Prentice Hall, 1990.
16. U. PAPE, Algorith 562: shortest path lengths, *ACM Trans. Math. Software*, 1980, *5*, p. 450-455.
17. J. PEARL, *Heuristics*, Addison-Wesley, 1984.
18. C. PRINS, Calcul sur micro-ordinateur de plus courts chemins dans les grands graphes peu denses, *Rapport DAP-93-11*, Département d'Automatique et Productique, École des Mines de Nantes.
19. R. SEDGEWICK et J. S. VITTER, Shortest paths in euclidean graphs, *Algorithmica,* 1986, *1*, p. 31-48.
20. M. SYSLO, N. DEO et J. S. KOWALIK, *Discrete optimization algorithms with Pascal programs,* Prentice Hall, 1993.
21. R. E. TARJAN, Data structures and network algorithms, *SIAM*, 1983.

ANNEXE : Structure de tas

Considérons un ensemble H de n éléments dont tout élément x possède une valeur numérique $W[x]$, et dans lequel les prélèvements ne concernent que le plus petit élément. Le tas (*heap*) est une structure de donnée puissante qui permet de réaliser ces prélèvements de minimums et l'ajout d'éléments

quelconques en seulement $O(\log n)$. Il s'agit d'un *arbre binaire* particulier. Dans un arbre binaire, tout élément x pointe au plus sur deux autres éléments, les *fils* de x. Tout élément x est pointé par un seul autre élément, le *père* de x, sauf un élément distinct r, la *racine*, qui n'a pas de père. La racine constitue le *niveau* 0 de l'arbre. Elle peut avoir au plus 2 fils au niveau 1, pouvant avoir à leur tour 4 fils dans le niveau 3, etc. Il y a donc jusqu'à 2^p éléments au niveau p, et un arbre binaire complet de p niveaux contient $2^p - 1$ éléments.

Pour coder un arbre binaire limité à p niveaux, il suffit d'un tableau Body de $n = 2^p - 1$ éléments. Les indices de Body correspondent à un numérotage des places dans l'arbre complet, niveau par niveau, et de gauche à droite dans chaque niveau. Body[1] contient la racine. Pour $i > 1$, Body[i] a pour père Body[i div 2] (div désigne la division euclidienne entière : 7 div 3 = 2 par exemple). Body[i] a pour fils Body[$2.i$] et Body[$2.i+1$]. Ainsi, on passe d'un élément à ses fils ou à son père par de simples multiplications ou divisions par 2 !

Pour être un tas, un arbre binaire doit d'abord être complet, au moins jusqu'à l'avant-dernier niveau. Si le dernier niveau n'est pas complet, ses éléments doivent occuper des places consécutives à partir de la gauche. De plus, tout élément doit avoir une valeur supérieure ou égale à celle de son père. Ainsi, en descendant le long d'une branche, on trouve des éléments de valeurs non décroissantes, et l'élément de valeur minimale se trouve à la racine.

La figure 3 est un exemple de tas dont les 9 éléments 1, 2, 3, 5, 7, 8, 9, 11 et 12 ont pour valeurs respectives 42, 57, 41, 67, 30, 36, 49, 35 et 63. Dans chaque nœud de l'arbre se trouve un élément suivi de sa valeur. Le numéro à gauche de chaque nœud est l'indice de rangement dans le tableau Body équivalent, donné après la figure. Notez que les 4 niveaux pourraient stocker jusqu'à 15 éléments, et que les éléments du dernier niveau occupent des places consécutives à partir de la gauche, conformément à la définition d'un tas.

Pour réaliser les différentes primitives sur des tas de sommets, nous codons un tas H d'au plus n sommets par un entier Num et deux tableaux de n entiers Body et Where (l'ensemble étant codable par un *record* en Pascal). Num indique le nombre de sommets contenus dans le tas. Ces Num éléments sont rangés dans Body, entre les indices 1 à Num. Where sert à localiser les sommets dans le tas. Where $[x] = 0$ signifie que x n'est pas actuellement dans H. Sinon, Where $[x]$ désigne l'indice où est rangé x dans Body, et

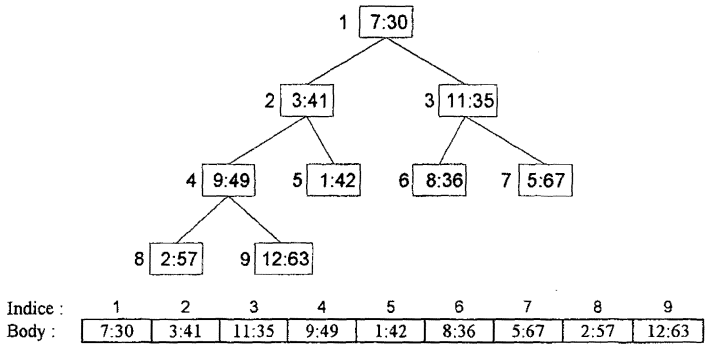

Figure 3. – Exemple de tas.

on a Body [Where $[x]$] $= x$. Les valeurs des sommets sont rangées dans un tableau W, en dehors de la structure de tas. La valeur d'un sommet x s'obtient donc simplement en consultant $W[x]$.

Les primitives de tas s'expriment simplement si on les ramène à deux opérations de base MoveUp et MoveDown. La procédure MoveUp (H, W, x) reforme un tas H correct après diminution de la valeur $W[x]$ d'un sommet x donné. Elle fait monter x en le permutant avec son père, tant qu'on n'est pas à la racine et que x est inférieur à son père. Sur la figure, si l'élément 9 voit sa valeur 49 passer à 29, on n'a plus un tas puisque que $W[9]$ est inférieur à la valeur du père. En permutant 9 avec 3, puis 7, l'élément 9 devient la racine du tas (ce qui est normal puisqu'il est devenu le minimum).

La procédure MoveDown (H, W, x) reforme un tas correct après l'augmentation de la valeur $W[x]$ d'un sommet x. Elle fait descendre x en le permutant avec son *plus petit fils*, tant qu'on n'est pas sur une feuille et que x est inférieur à ses fils. Supposons que l'élément 7 de la figure ait sa valeur augmentée de 30 à 43. Notez que si on le permute avec son plus grand fils (3, de valeur 41) on altère le tas car l'élément désormais à la racine n'est plus le minimum. En permutant 7 avec 11, puis 8, on récupère un tas correct. Rappelons qu'un tas de p niveaux a $n = 2^p - 1$ éléments au maximum. Le nombre de niveaux est donc égal à la partie entière de $\log_2(n) + 1$. Les opérations MoveUp et MoveDown déplacent dans le pire des cas l'élément modifié le long d'une branche complète de p niveaux. Ces deux opérations sont donc en $O(p) = O(\log n)$, tous les logarithmes étant du même ordre.

Par exemple, un tas de 10 niveaux peut stocker 1023 éléments. Si un élément du dernier niveau devient le minimum, MoveUp le monte à la

racine en seulement 9 permutations ! Outre MoveUp et MoveDown, les primitives de tas utiles ont pour noms Clear, Insert, Empty, TakeMin et Inside. Clear (H) initialise un tas H à vide en $O(n)$. La fonction Empty (H), en $O(1)$, est vraie si et seulement si un tas H est vide. Inside (H, x) est vraie si et seulement si x est dans H.

Insert (H, W, x) insère un nouvel élément x dans un tas H. Pour cela, on ajoute un nouveau nœud en bas du tas pour recevoir l'élément x, puis on fait monter x à sa position définitive avec MoveUp. TakeMin (H, W, x) prélève l'élément de valeur minimale et le renvoie dans x. L'accès au minimum est immédiat, c'est l'élément situé à la racine du tas. Mais il faut ensuite reformer le tas. Il suffit de prendre le dernier élément, de le déplacer à la racine, puis de le faire descendre à sa position définitive avec MoveDown. Ces deux primitives héritent de la complexité de MoveUp et MoveDown : $O(\log \text{Num})$.

Voici ces procédures en pseudo-langage. Dans MoveUp et MoveDOwn, on évite les permutations de la façon suivante : x est temporairement enlevé et laisse un trou qu'on déplace par décalage d'éléments. Il est réinséré à l'emplacement final de ce trou. Dans MoveDown, j désigne l'indice du fil du nœud d'indice i en cours de traitement. Il est initialisé sur le fils gauche de i. Le test « Si (j<Num) » corrige éventuellement i pour qu'il pointe sur le plus petit fils.

```
Clear (H)          : Num := 0
                     Pour k := 1 à N
                      Where[k] := 0
                     FP
Empty  (H)         : Empty := (Num = 0)
Inside (H,x)       : Inside := (Where[x] > 0)
MoveDown (H,W,x)   : i := Where[x]
                     j := 2 * i
                     Si (j < Num) et (W[Body[j+1]] < W[Body[j]]) alors
                        j := j+1
                     FS
                     Tant que (j <= N) et (W[Body[j]] < W[x])
                       Body[i]         := Body[j]
                       Where[Body[j]] := i
                       i               :=j
                       j               := 2 * i
                       Si (j < N) and (W[Body[j+1]] < W[body[j]] alors
                          j := j+1
                       FS
```

```
                    FT
                    Body[i] := x
                    Where[x] := i
Insert(H,W,x)     : Num := Num + 1
                    Body [Num] := x
                    MoveUp (H,W,X)
TakeMin (H,W,x)   : x := Body[1]
                    Where[x] := 0
                    Num := Num - 1
                    Si Num > 0 alors
                       Where[Body[Num+1]] := 1
                       MoveDown (H,W,Body[Num+1])
                    FS
MoveUp (H,W,x)    : i := Where[x]
                    j := i div 2
                    Tant que (j <= N) and (W[Body[j]] > W[x])
                       Body[i]         := Body[j]
                       Where[Body[j]] := i
                       i               := j
                       j               := i div 2
                    FT
                    Body[i] := x
                    Where[x] := i
```