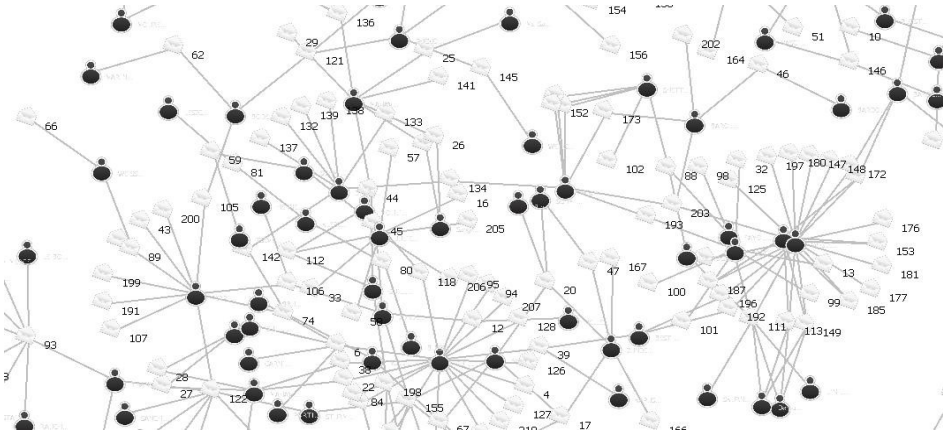
PLUS COURTS CHEMINS



Implémentation de DIJKSTRA et comparaisons selon les différentes PQ.

BOLTENHAGEN Mathilde
HUBER Laurine
SDA 2 – 2014

Sommaire :

1-Dijkstra

L'algorithme de Dijkstra permet de calculer le chemin minimal d'un point à un autre avec le calcul des distances. Différentes structures conviennent à cette algorithme mais certaines sont plus optimisés selon le graphe de base. C'est cela que nous essayons de montrer dans ce projet. Nous savons que l'algorithme de DIJKSTRA peut être amélioré d'une complexité de $o(n^2)$ à $o((n+m)log(n))$ selon la priority queue. On entend par « priority queue » la structure de donnée utilisée.

Complexité au pire pour DIJSKTRA (formule générale) :

N*(extract_min + insert_key) + M* deacrease_key

avec N le nombre de nœuds et M le nombre d'arêtes.

L'algorithme de Dijkstra ne change pas selon la priority queue, seule les fonctions internes sont propres à chacune. Nous proposons donc la spécification suivante de l'algorithme

spécification de l'algorithme de Dijkstra :

```
inititialisation( PQ ) // qui remplit la PQ avec des priorités = infini pour tous sauf le départ
tant que non vide(Graphe)
        extraire_min(PQ)
        tant que successeur(min)!=NULL
                recherche(PQ,successeur)
                si priorité(successeur)+poids(min, successeur)<priorité(successeur)
                        alors modification(PQ,successeur)
                fsi
                successeur=successeur suivant ;
        ftq
        supression(Graphe, min)
ftq
```

On constate donc qu'il nous faudra pour chaque structure de donnée étudiée quatre principales fonctions :
- initialisation
- extraire le minimum
- recherche par rapport à une valeur (et non une priorité)
- modification d'un élément

Durant la suite de ce projet, nous allons nous intéresser à cinq structures dont quatre que nous avons étudiés et trois qui sont correctement implémentées en C.

2-Spécification et compléxité selon la PQ

2-a. Le tableau:

t:tableau , g: graphe

initialisation:
ttq non-vide(g)
si sommet(g)=depart
alors init_depart(t) //initialisation avec priority=0
sinon init(t) //initialisation avec priority=INF
fsi
fttq

extrait le minimum et le supprime:
pre extract_min = nonvide(t)
extract_min(t)= init min =x //initialisation du min a x
pour i allant de 0 à size(t)-1 //parcours du tab
si t[i] < min
alors min=t[i]
fsi
supval(t,min) //suppression de la valeur
// où supval(t,x) supprime dans t
// l'élément x

recherche d'une valeur:
pre rech(t,x)=nonvide(t)
rech(adjt(t,x),y)= pour i allant de 0 à size(t)-1 //parcours du tableau
si x==y alors x //si x=y on renvoi x
sinon rech(t,y) //sinon recursivité sur t
fsi

extract_min: o(n)
recherche: o(n)
insertion:o(1)

Cet PQ est optimale pour une petite quantité de noeuds. Lorsque la quantité augmente, elle devient de plus en plus lente.

2-b. Le tas binaire:

Le tas binaire étant représenté comme un tableau avec des propriétés particulières, les spécifs de recherche et initialisation sont les mêmes.
Le minimum cependant est récupéré à la première place du tableau, nous n'avons pas besoin de parcourir le tableau.
En plus de cela, on a eu besoin de fonction qui renvoient le père, fils gauche, fils droit (c'est à dire connaître les indices de ceux-ci dans le tableau), ainsi que de fonctions percolate_up et tamiser qui ont permis le rangement du tas.

pos: une position courante dans le tas, h: heap (tas binaire)

indices nécessaires dans le tableau, ne pas oublier que le tas est à la base représenté par un arbre binaire et qu'on construit ici un tableau

```
indicefg(pos)= 2*pos+1
indicefd(pos)= 2*pos+2
indicepere(pos)= si pos%2==0 && pos!= 0 //si on est à droite et pas a la
                                        //racine
                 alors (pos-2)/2
                 sinon pos%2!=0 && pos!=0 //si on est a gauche et pas
                                          //a la racine
                 alors (pos-1)/2
                 sinon -1 //cas ou on est a la racine, pas de père, erreur
                 fsi
```

percolate up pour la dernière valeur ajoutée dans le tas, on la range à sa place

```
percolate_up(h)= init last = size(h)-1 //initialisation du dernier elem à last
                 ttq indicepere(last) >=0
                 si pere(last) > last //ou père donne la valeur du père
                 echange(pere(last),last) //ou echange échange les
                                          //valeurs
                 fsi
                 last=last-indicepere(last)-1 //on coupe le tableau pour
                                              //recommencer
                 fttq
```

tamiser le tas :

```
tamiser(h)= pour i allant de 0 à dernieretage(h)
            //dernieretage teste si, étant un arbre binaire, on serait au
        //dernier étage
            minf=minfils(h,h(i)) // renvoie le minimum des fils de la valeur
                    // courante
            //si le minimum des fils est plus petit, je replace la valeur à sa //
place (percolate_up_val fait la même chose que percolate up // mais pour une
certaine valeur et non pour la dernière)
```

si minf < cour
alors percolate_up_val(h,minf)
fsi

init:o(n)
insert: o(log n)
extract_min: o(n)
percolate_up: o(h(x)) ou h est la hauteur du tas binaire

Il existe un moyen plus optimal de construire le tas binaire. Pour cela, on remplit d'abbord tout le tas puis on trie ensuite. J'ai quant à moi trié mon tas au fur et à mesure de la modification des valeurs, et je pense que le problème du résultat final vient de là.

2-c. Liste chainée

La liste simplement chainée non triée est semblable au tableau. On se déplace à partir de la tête et on parcourt par le chainage les différents éléments.

initialisation: o(1)

```
tant que non vide(g)
      si sommet de g= depart
            alors adjonction en tete du depart (c'est à dire avec priority =0)
       sinon adjonction en tete normale (priority = INF)
       fsi
 ftq
```

extrait et supprime le minimum :o(n)

```
extract_min_liste(adjt(nouv(),x)) = x;
extract_min_liste(adjt(l,x))= min
      avec (l,min) = init (supt(l),x)
                        ttq non vide(l)
                        rep si tete(l)<min alors (supt(l), tete(l))
                              sinon (supt(l),min)
                        frep
                  fttq
```

recherche d'une valeur dans la liste: o(n)

```
recherche_liste(nouv(),x)=null;
recherche_liste(adjt(l,y),x)= si y==x alors y
                                          sinon recherche_liste(l,x)
                                          fsi
```

modification : o(n)

condition : valeur appartient à la liste

```
priority(recherche_Liste(l,valeur))=nouv_priorit ;
antecedant(recherche_liste(l,valeur))=nouv_antecedant ;
```

2-d. L'AVL

L'AVL est un arbre de recherche binaire équilibré. On trouve à chaque nœud, les poids plus faibles à gauche et plus élevés à droite. L'arbre est dit équilibré car la différence entre la hauteur de son sous arbre gauche et droit est compris entre -1 et 1.

initialisation: o(log n)

```
arbre=anouv()
tant que non vide(g)
        si le sommet de g == depart
                alors elemt->priority = 0
                sinon elemt->priority =INF
        fsi
        insertion(elemt, arbre)
 ftq
```

extrait et supprime le minimum :o(log(n))

```
tant que non vide (g(arbre))
        arbre=g(arbre)
 ftq
supprimer(arbre)
```

recherche d'une valeur : o(log(n))

```
 si vide(a) alors null
 sinon
        si val = etiquette(a) alors val
                sinon si (recherche_avl(g(a),val)==NULL
                                alors recherche_avl(d(a),val)
                                fsi
                fsi
        fsi
fsi
```

modification d'une valeur dans l'AVL :

supression(arbre,valeur) , inverstion(arbre,nouvelle) ;

3- Implémentation

3-a. Graphes aléatoires

La structure graphe que nous utilisons est la liste d'adjacente vue en TP et en cours. Cette structure se compose de deux structures : une liste des successeurs d'un sommet et une liste de sommets chacun ayant sa propre liste de successeurs.

Nos graphes aléatoires sont crées à partir de la fonction 'rand()' qui renvoie un chiffre entre deux bornes. C'est la fonction remplissage qui construit notre graphe. Nous avons choisit de créer deux fois plus d'arete qu'il n'existe de sommets.

3-b. Implémentation de nos différentes PQ

Chaque PQ contient une structure appelée elmtpq (élément de priorité) qui est composé de son antécédant (par exemple la ville qui lui a permis d'avoir une si faible distance) , sa priorité et sa valeur (nommé courant).

```
typedef struct strPq{
        S antecedant;
        int priority;
        S courant;
} elmtpq;
```

La **liste** est la structure de donnée la plus simple à comprendre comme on peut le voir ci dessous :

```
typedef struct strliste{
elmtpq *a;
struct strliste *suiv;
}liste, Liste;
```

Pour le **tableau et le tas binaire**, on note la capacité possible déjà allouée ainsi que la taille que l'on utilise afin de réallouée si la taille atteint la capacité :

```
typedef struct table{
int capacity;
int size;
elmtpq *tab;
} table;
```

Pour l'**AVL**, la structure est un peu plus grande car on note le fils gauche, le droit et le père :

```
typedef struct sAvl {
elmtpq *elemt; //
struct sAvl* ad;//arbre droit
struct sAvl* ag;//arbre gauche
struct sAvl* pere;
} strAvl, * Avl;
```

3-c. Lancement de notre programme

Nous avons utilisés les macros, et décidé de permettre à l'utilisateur de comparer la

liste (dans tous les cas, car c'est le plus représentatif), avec soit le tas, soit le tableau. Pour cela, il faut décider à la compilation, taper **make TAB=1** pour une comparaison avec le tableau, ou **make BH=1** pour une comparaison avec le tas. Vous pourrez ensuite choisir le nombre de noeuds du graphe que vous voulez tester, et observer les temps d'éxecution.

4-d. Nos difficultés rencontrés

Au départ, nous n'avions pas compris qu'il fallait supprimer le minimum de la PQ. Ainsi nos algorithme étaient très complexe. Nous avions un booléen 'fait' qui indiquait si on pouvait ou non prendre ce minimum ou s'il fallait prendre le précédant. Pour l'AVL, par exemple, nous avions implémenter la fonction infixe qui parcours l'arbre de la gauche à la racine à la droite.

La deuxième difficulté que nous avons rencontré est lors de la comparaison de nos fonctions pour l'implémentation de Dijkstra. Laurine n'utilise pas de pointeur sur les éléments de priorité tandis que moi si. Nous n'avons pas réussi à résoudre ce problème à temps et c'est pour cela que nous avons plusieurs algorithme de Dijkstra.

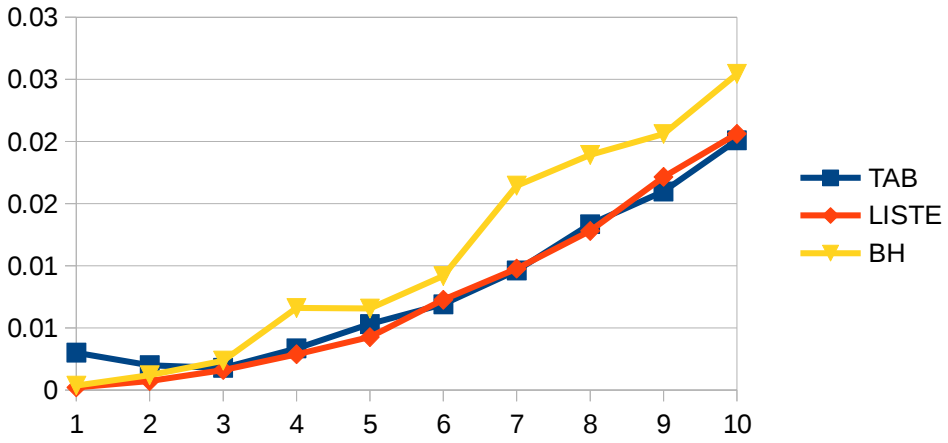
Enfin, nous avons eu un problème lors du «vidage» de nos SQ. // je sais pas si on doit le dire ça.

Pour finir, Mathilde n'a pas réussi à implémenter correctement l'AVL mais un bon début est même capable de marcher sur des arbres de 10 sommets. La suppression marche. L'erreur se trouve sûrement lors de l'équilibrage dans la suppression...

4-Résultats

4-a Résultats obtenus

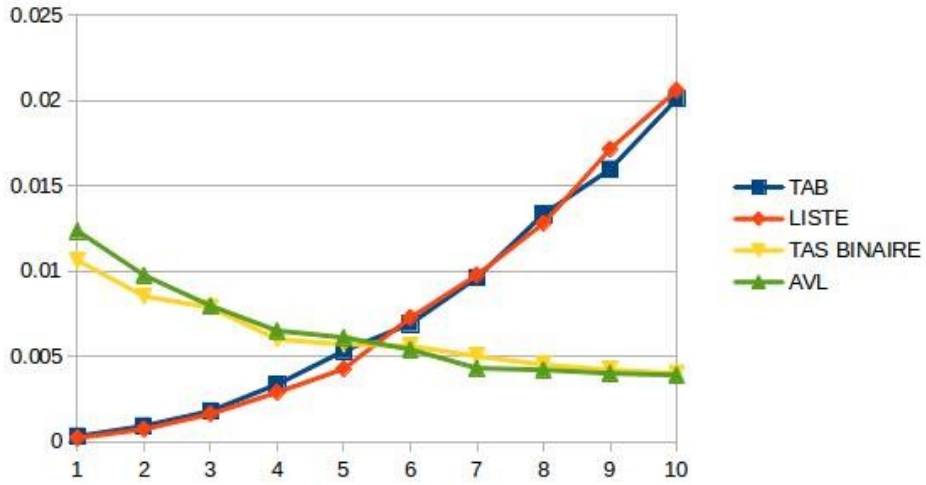
Les résultats que l'on à obtenu ne se rapprochent pas de ce que l'on attendait. Les temps d'éxécution des 2 priority queue simple croient quasiement de la même façon. Faute de temps, nous n'avont pas réussi à déterminer d'où provient le problème. On pense cependant qu'il s'agit du fait que l'on a pas implémenté dijkstra de la même façon que celle proposé dans le tp. On s'est servit des algorithmes expliqués dans les cours (graphes et SDA), qui font exactement la même chose, mais pas exactement de la même façon (bien que la différence ne soit pas énorme). La courbe du tas binaire s'éloigne plus encore de nos attentes puisqu'elle aurait du passer sous la courbe à partir de 6 (600sommets) or son temps d'execution augmente encore et s'éloigne même des deux autres courbes.



4-b Résultats attendus

En évaluant les complexité des différentes PQ, on en conclue la chose suivante: normalement, pour des petits graphes (<300 noeuds environ), c'est le tableau qui est le plus rapide, puis la liste chaînée , l'avl, et le tas qui est le moins. Arrivée à un nombre de noeuds suffisant, on est censé observer un point ou les courbes s'inversent, le tas devient le plus rapide, puis l'avl, la liste chaînée, et le tableau en dernier lieu.

On obtiendrait un graphe de ce genre (graphique fictif pour illustrer nos propos) :



Conclusion: en comparant les différentes compléxités, nous nous sommes rendu compte que plus le nombre de noeuds augmente, plus le tas binaire sera efficace. Cependant, nos résultats ne sont pas ceux escomptés. La structure de donnée choisie pour un programme change donc la complexité puisque les fonctions sont plus ou moins complexes selon la PQ. Il est donc important de prendre soin de bien choisir une structure de donnée. Pour l'algorithme de Dijkstra par exemple, nous savons que c'est le tas de Fibonacci le plus avantageux.