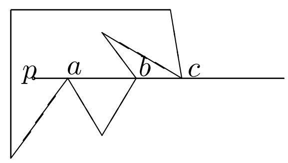
UPMC — Master d'Informatique —
Examen Algorithmique avancée du 17 décembre 2008
Eléments de corrigé – durée 2h

1 Intérieur d'un polygone simple



En a, il faut compter 0 intersection, en c aussi. En b, il faut compter 1 intersection.

1.
```
Fonction estSurContour(p,P)
  q <- s0
  Repeter
    Si appartientSegment(p,q,succ(q)) Alors Retourne(Vrai) Fin Si
    q <- succ(q)
  Jusque q=s0
  Retourne(Faux)
```
   On suppose que l'instruction de retour, `Retourne`, fait sortir de la fonction.

   Attention : ne pas oublier de tester le côté $[s_n, s_0]$.

   On remarque que l'algorithme est en $O(n)$ (il y a au plus n itérations et chacune coûte un temps constant).

2. (a)
```
Fonction estInterieur(p,P)
  Si estSurContour(p,P) Alors Retourne(Faux) Fin Si
  q <- s0 ; cpt <- 0
  Repeter
    Si intersecte(p,u,q,succ(q)) ou
        (appartientDemiDroite(p,u,succ(q)) et dePartEtDautre(p,u,q,succ(succ(q)))
    Alors cpt <- cpt+1
    Fin Si
    q <- succ(q)
  Jusque q=s0
  Retourne(impair(cpt))
```
   (`impair(x)` est une fonction qui rend vrai ssi x est impair, facile à définir).

   (b) L'algorithme est en $O(n)$ car `estSurContour` est en $O(n)$ et le reste de l'algorithme est aussi en $O(n)$: il y a n itérations, chacune coûte un temps constant et la fonction `impair` est en $O(1)$.

3.
```
Fonction dePartEtDautre(p,u,a,b)
  Si det(u,pa).det(u,pb) < 0 Alors Retourne(Vrai) Sinon Retourne(Faux) Fin Si
```
   On pourrait aussi tester si $ccw(p, p+u, a).ccw(p, p+u, b) = -1$.

4. Dans le cas général il faut aussi tester, pour tout sommet q du contour, si q et le successeur de q appartiennent tous deux à la demi-droite Δ. Si c'est le cas il faut alors tester si le prédécesseur de q et le successeur du successeur de q sont de part et d'autre de Δ : si oui compter 1 intersection, si non compter 0 intersection.

2 Arbres équilibrés

1. Arbres binaires qui vérifient $\mathcal{P}_0$: arbres complets. Arbres binaires qui vérifient $\mathcal{P}_1$: arbres H-équilibrés (AVL si ce sont des arbres de recherche).
   Exemples : facile.
2. Soit $c \in \mathbb{N}^*$, la fonction $x(1+x)^c$ vaut 0 en $x = 0$ et 2^c en $x = 1$. Comme elle est continue, elle prend toutes les valeurs entre 0 et 2^c. Par conséquent il existe $\alpha \in]0,1]$, tel que $\alpha(1+\alpha)^c = 1$.
3. Soit $\alpha \in]0,1]$, tel que $\alpha(1+\alpha)^c \leq 1$. On montre que $n \geq (1+\alpha)^h - 1$ pour tout arbre binaire de taille n et de hauteur h qui vérifie $\mathcal{P}_c$.
   Si l'arbre est vide c'est vrai ($n = 0$ et $(1+\alpha)^h - 1 = (1+\alpha)^{(-1)} - 1 \leq 0$).
   Soit $h \geq -1$, supposons que la propriété soit vraie pour tout arbre de hauteur inférieure ou égale à h. Soit T un arbre binaire vérifiant $\mathcal{P}_c$, de taille n et de hauteur $h+1$. Les deux sous-arbres T_1 et T_2 de T sont l'un de hauteur h et l'autre de hauteur supérieure ou égale à $h-c$. Ils vérifient tous deux $\mathcal{P}_c$. Soient n_1 et n_2 les tailles de T_1 et T_2.

$$\begin{array}{rcl} n & = & n_1 + n_2 + 1 \\ n & \geq & (1+\alpha)^h - 1 + (1+\alpha)^{h-c} - 1 + 1 = (1+\alpha)^h + (1+\alpha)^h . \frac{1}{(1+\alpha)^c} - 1 \\ n & \geq & (1+\alpha)^h + (1+\alpha)^h . \alpha - 1 \quad (\text{puisque } \alpha(1+\alpha)^c \leq 1) \\ n & \geq & (1+\alpha)^{h+1} - 1 \end{array}$$

   La propriété est donc vraie pour tout arbre binaire vérifiant $\mathcal{P}_c$.
4. Si T vérifie la propriété $\mathcal{P}_c$ alors $n \geq (1+\alpha)^h - 1$ donc

$$h \leq \frac{\log(n+1)}{\log(1+\alpha)}$$

   La hauteur de T est donc en $O(\log n)$.

3 Plus courts chemins

1. (a) Montrons que pour tout i, à l'itération i, pour tous les sommets u tels que $dist[u] < \infty$, alors $dist[u]$ est bien la distance de r à u, et $pere[u]$ est le père de u dans l'arborescence de parcours en largeur. Ainsi à la fin de l'algorithme on aura obtenu un parcours en largeur du graphe à partir de r.
   La propriété est vraie pour $i = 1$: à la première itération on atteint tous les successeurs

de r. Supposons la propriété vraie pour i et montrons qu'elle est aussi vraie pour $i+1$. Soit u le sommet traité à l'itération $i+1$, et soient $v_1, ..., v_l$ ses successeurs ; pour chaque v_j, si $dist[v_j] < \infty$, c'est que $dist[v_j]$ a été affectée avant l'itération $i+1$, donc par hypothèse de récurrence, c'est la distance de r à v_j et $pere[v_j]$ est le père de v_j dans l'arborescence de parcours en largeur. Dans le cas où $dist[v_j] = \infty$, on remplace par $dist[v_j] = dist[u] + 1$, et on a bien $dist[v_j]$ distance de r à v_j et u père de v_j dans l'arborescence de parcours en largeur, en effet si ce n'était pas le cas v_j aurait un père u' avec une distance à r plus petite, et on aurait déjà atteint v_j par l'intermédiaire de u'. Donc le plus court chemin de r à v_j passe par u et $dist[v_j] = dist[u] + 1$ est la distance de v_j.

(b) L'algorithme effectue n extractions du minimum et m mises à jour de `dist`.
Son temps de calcul dépend de la représentation de Q.
- si Q est un tableau, l'extraction du minimum est en $O(n)$ et la mise à jour en O(1), d'où complexité en $O(n^2 + m) = O(n^2)$,
- si Q est une liste triée, l'extraction du minimum est en $O(1)$ et la mise à jour en O(n), d'où complexité en $O(n + mn)$,
- si Q est un tas, l'extraction du minimum est en $O(\log n)$ et la mise à jour en $O(\log n)$, d'où complexité en $O((n + m) \log n)$,
- si Q est une file binômiale, même complexité que pour un tas
- si Q est une file Fibonacci, l'extraction du minimum est en $O(\log n)$ et la mise à jour en coût amorti $O(1)$, d'où complexité en $O(n \log n + m)$.

(c) En gérant Q comme une file FIFO, que l'on initialise à r, et dans laquelle on insère au fur et à mesure tous les successeurs des sommets traités, on obtient l'algorithme

Procedure PL(r)
  dist[r]=0 ; Q :=r
  Repeter Tant que non estVide(Q) **Faire**
    u= supprimerFile(Q)
    Pour chaque sommet v successeur de u **Faire**
      Si `dist`= ∞ **Alors**
        pere[v]=u ; dist[v]=dist[u]+ 1 ; insererFile(v,Q)
      Fin Si
    Fin Pour
  Fin Repeter
Fin Procedure

Cet algorithme effectue un parcours en largeur du graphe à partir de r, en donnant pour chaque sommet sa distance à r et son père dans l'arborescence. La preuve résulte de celle de Dijkstra, en remarquant que l'extraction du minimum se réduit ici à la suppression de la tête de file, et que les valeurs de `dist` sont modifiées au plus une fois (lorsqu'un sommet v est atteint pour la première fois, $dist[v]$ prend sa valeur définitive).

L'algorithme fait n insertions et suppressions dans la file, accompagnée des mises à jour de `dist`, ainsi que m tests sur la valeur de `dist` ; sa complexité est donc en $O(n+m)$, ce qui est meilleur que tout ce que l'on peut faire en utilisant l'algorithme de Dijkstra avec une file de priorité.

2. (a) Toute arborescence avec n sommets ayant $n-1$ arcs, étant donné un sommet quelconque x, le nombre d'arcs qui le séparent de la racine est donc $\leq n-1$; il s'en suit que $dist[x] \leq k(n-1)$ si la valuation de chaque arc est bornée par k.

   (b) Dans le cas qui nous intéresse, on a vu que la file de priorité (`dist`) contient des valeurs bornées par $K = k(n-1)$. De plus les minimums successifs forment une suite croissante (il est facile de montrer que si le sommet u_1 est extrait de Q avant le sommet u_2, alors en fin d'execution, $dist[u_1] \leq dist[u_2]$). Par ailleurs le nombre total d'opérations de l'algorithme est majoré par m diminutions clés $+n$ extractions ; donc finalement on peut calculer les plus courts chemins en temps $O(kn+m)$.

   (c) Représentation du graphe : table T de 0 à K (borne des valuations), où chaque case j pointe vers la liste des sommets pour lesquels la distance estimée (`dist`) vaut j (et la case $T(0)$ pointe sur la liste des sommets pour lesquels `dist` est infini). La table `dist` (de taille n), qui s'accompagne d'une table PT (de taille n) de pointeurs dans laquelle chaque sommet est relié à son occurrence dans les listes de T.

   Pour la mise à jour : étant donné un sommet u on regarde tous ses successeurs, et s'il y a une mise à jour de `dist` à faire sur v (dont la valeur dans `dist` passe de $d1$ à $d2 < d1$), on enlève v de $T(d1)$ –accès direct par $PT(v)$–, on rajoute v dans la liste de $T(d2)$, et on modifie $PT(v)$ pour pointer sur la nouvelle place de v dans les listes de T : toutes ces opérations se font en $O(1)$. L'accès au min se fait aussi en $O(1)$: au départ min vaut 1, puis à chaque étape, si la liste $T(min)$ n'est pas vide on renvoie l'un quelconque de ses éléments (pour (`extractmin`) et sinon on incrémente min (on utilise ici l'hypothèse que la suite des min est en ordre croissant).