Graphes et algorithmes (& modélisation)
---11/09/12---
16h de cours et 20h de TD

- plan du cours
  - Flots | connexité
  - Graphes planaires
    - un graphe peut il ce dessiner sans croiser les arrêtes ?
  - Postier chinoise
  - Voyageur de commerce
- objectifs
  - algos polynomiaux
    - aspect optimalité
  - problèmes difficiles
    - approximation, heuristiques,
  - problèmes de Coloration, stable maximum...
  - algorithmes probabilistes
  - algorithmes distribués (?)
  - algorithmes exponentiels

- Graphe G(V,E)
  - Orienté
    - Sommets : V
    - Arcs: E, chaque arc est un couple de sommet E € VxV
    - V = {s,a,b,c,d,p}
    - E = { (s,a) , (s,c) , (c,d) , (a,b) , (b,p) , (a,d) , (d,p) }
    - représentation mémoire
      - vecteur de sommets, puis pour chaque sommet, la liste des successeurs

S	A	C
A	B	D
B	P	
C	D	
D	P	
P		

        - Place en mémoire :
          - n = nb sommets
          - m = nb d'arcs
          - O(n+m)
      - Matrice d'adjacence O(n²).

	S	A	B	C	D	P
S	0	1	0	1	0	0
A	0	0	1	0	1	0
B	…					
C						
D						
P						

- <u>Problème du flot maximum dans un réseau</u>
  - Entrée : réseau R=G=(V,E),
    - un graphe orienté G=(V,E),
    - un sommet source S, et un sommet puits P,
    - une fonction de capacité associant E ==> N, je met sur chaque arc une valeur qui est un nombre naturel
  - Sortie :
    - Flot :
      - une fonction f : E ==>N associant a chaque arc (x,y) un flot(nb naturel) f(x,y).
      - contrainte de capacité :
        - pour tout arc(x,y) on a f(x,y) <= cap(x,y).
      - contrainte de conservation du flot
        - pour tout sommet X!= {s,p} : flot_rentrant(x) = flot_sortant(x)
        - somme de y prédécesseur de x f(y,x) = somme de z successeur de x f(x,z).
      - valeur d'un flot :
        - |f| = flot_sortant(s) - flot_rentrant(s)= pour tout z succ de s(somme(f(s,z)) – pour tout y pred s (somme(f(y,s))).
- Méthode de Ford-Fulkerson [1956]
  - Entrée : réseau
  - Sortie : flot de valeur max
    - F ← 0 ;// 0 partout.
    - Gf ← G
    - Tant qu'il existe un chemin améliorant M de s à p dans le graphe résiduel (Gf)
      - f ' ← flot de mu de val cap (mu).
      - f ← f + f ' // attention aux signes
      - Gf ← graphe résiduel (G, cap, f).
      - fin tant que ;
    - retourner f ;
  - GrapheResiduel(G,f)
  - Gf :
    - sommets : les mêmes que pour G
    - arcs : pour tout arc (x,y) de G //supposons qu'il n'y a pas (y,x) dans G, on met dans Gf

- si f(x,y) < cap(x,y) : on met l'arc (x,y) de cap'(x,y) = cap(x,y) – f(x,y)
- si f(x,y) >0 : on met l'arc (y,x) de cap'(y,x) = f(x,y).

- flot maximum et coupe minimum
  - coupe : partition (X,!X) des sommets telles que :
    - s € X et p €!X
    - capacité d'une coupe = ∑ des capacité des arcs qui vont de gauche à droite et que l'on coupe.
    - On remarque que la valeur du flot maximum, est égale à la capacité de la coupe minimum.
  - <u>Lemme 1 :</u>
    - Soit G= (V,E) , s, p, c : E → N ,un réseau
    - Pour tout flot f et toute coupe (Y,!Y) , on a que la valeur du flot est <= à la capacité de la coupe.
    - <u>Preuve :</u>
  - <u>Théorème 1 de Ford-Fulkerson(FF) :</u>
    - Soit f(Ford-F), un flot tel que dans G(f(FF)) il n'y a plus de chemin de s à p, soit X l'ensemble des sommets que l'on peut atteindre a partir de s dans G(f(FF))
    - on a alors :
      - |f(FF)| = cap(X,!X) // où (X,!X) est la coupe du graphe
    - <u>Preuve :</u>
      - 1.on cherche a montrer que pour tout arc (u,v) de G, avec u€ X et v €!X on a f(u,v) = c(u,v)
        - si f(u,v)!= c(u,v) avec u€X et v€!X alors dans le graphe résiduel de G , il existe un arc uv de capacité >0 et donc v est accessible depuis s et donc v €X, CONTRADICTOIRE, car v €!X , donc f(u,v) = c(u,v)
      - 2. pour tout arc (zt) de g avec z €!X et t €X on a f(zt) =0
        - si f(zt) > 0 alors dans le graphe résiduel de g, il existe un arc tz de cap >0, et donc z est accessible depuis s et z €X , CONTRADICTOIRE,car z €!X donc f(zt) =0 .
      - 3. f(FF)(X,!X) = ∑f(uv) – ∑f(zt) = ∑ c(uv) – 0 = par def c'est capde la coupe (X,!X)
      - 4. soit f(FF) un flot tel que il n'existe plus de chemin améliorant, soit (X,!X

) la coupe définie dans THM1
        - on a |f(FF)| = cap(X,!X), f(FF) est un flot de valeur maximum et (X,!X) est une coupe de capacité minimum.
    - f un flot, (Y ,!Y) un coupe,
      - f(Y,!Y) = ∑f(u,v) - ∑f(z,t) // u,t €Y, et z,v €!Y
      - l'idée est de montrer que la valeur du flot dans une coupe est une valeur x, et que si l'on rajoute dans cette coupe un sommet de son !Y, alors la valeur du flot ne change pas, grâce à la propriété de conservation du flot dans les sommets.
- <u>Méthode de Ford-Fulkerson</u>
  - tout algo qui trouve un flot f(FF) tel qu'il n'y ai plus de chemin améliorant à trouvé un flot max.

- on trouve aussi la coupe de capacité minimum
- Algo FlotMaxCoupeMin
  - f(FF)AlgolFlotMax
  - Calcul Gf(FF)
  - on lance un parcours depuis s dans Gf(FF)
  - on prend X = ensemble des sommets accessible depuis s dans Gf(FF)
  - et la coupe min = (X,!X).
  - complexité : versions polynomiales $O(n^3)$.

28/09/12

Algorithme de reconnaissance de graphe planaire
E : graphe G=(V,E) , non-orienté, 2-connexe(connexe et enlever 2 sommet pour le déco).
S : booléen : vrai si g est planaire, faux sinon
F = ensemble des faces d'un dessin planaire de g // si g planaire

G:graph init
H : sous-graphes planaire déjà déssiné
les <u>ponts</u> de G par rapport H : morceau de G pas encore dessinés mais qui se raccrochent à H.
définition : ponts :

- pont dégénérés : arête XY de G qui n''est pas dans H mais dont les deux extrémités sont dans H.
- pont non dégénérés :
  - formés de C : une composante connexe de G-H
  - N(C) : ensemble des voisins de C dans G.
  - toutes les arêtes de G qui touchent un sommet de C

Soit B un pont,
<u>pieds</u> du pont les sommets de B qui se trouvent dans H
Rq : chaque pont à >= 2 pieds (par 2-connexité)
<u>faces compatibles :</u> Une faces fi de H est compatible avec B, si pieds(B) inclus dans fi .

E : graphe G=(V,E) , non-orienté, 2-connexe(connexe et enlever 2 sommet pour le déco).
S : booléen : vrai si g est planaire, faux sinon
F = ensemble des faces d'un dessin planaire de g // si g planaire

soit C un cycle de G // il existe car le graphe est 2-connexe
H ← C // sommets + arêtes de G.
F ← {f1,f2} où f1 et f2 sont formés des sommets du cycle.
Calculer l'ensemble β des ponts de G par rapport à H.
pour chaque ponts B€ β , calculer facescompatibles(B).
//boucle principale

tantQue G!= H
    si il existe un pont B sans face compatibles
        return faux ;//g non planaire
    choisir un pont B avec le moins de faces compatibles. //O(n)
    Choisir fi € facesCompatibles(B). // O(n)
    soit x,y € Pieds(B) et mu un chemin de x à y dans B // O(n+m)
    ajouter mu au graphe H //O(n)
    la face fi est remplacée par deux nouvelles faces. Mise a jour de l'ensemble des faces
    Mise à jour des ponts et des faces compatibles.
FinTantQue
return Vrai ;
#ponts : | β| <= m € O(n)
#faces : |F| € O(n)

5/10/12
théorème : Soit g un graphe planaire avec un dessin donné. Soit f un face de ce dessin. On peut redessiner G de sorte a ce que f soit la face extérieur du dessin.

Nouveau Chapitre : Cycle eulérien / Cycle Hamiltonien

exemple : postier chinois, voyageur de commerce

Soit g un multi-graphe,

- cycle eulérien : cycle passant <u>exactement</u> un fois par chaque arête
- graphe eulérien : graphe possédant un cycle eulérien .

Problème du postier chinois
Entrée : multigraphe G , longueur(pos) sur les arêtes.
Sortie : Cycle passant au moins une fois par chaque arête , de longueur minimum.

Cycle hamiltonien : cycle passant exactement une fois par chaque sommet

Problème du voyageur de commerce
Entrée : multigraphe G , longueur(pos) sur les arêtes.
Sortie : Cycle passant au moins une fois par chaque sommet , de longueur minimum.

12/10/12

graphe eulérien , algo polynomial

Graphe Hamiltonien
    problème d'hamiltonicité
Entrée : un graphe G non-orienté
Sortie : un booléen, vrai si G est hamiltonien, Faux sinon

// problème NP-Difficile

- on ne connaît pas d'algo polynomial pour les résoudre
- il est conjecturé qu 'il n'y a pas d'algo polynomial pour les résoudre.

Feuille de td1 approximation _ grands classiques

<u>parcours de lifo par un agent de sécurité :</u>

- 1 cas, structure d'arbre :
  - ParcoursProfondeur(Graphe G){
    - //initialisation
    - pour chaque sommet x de G {
      - etat[x] ← nonAtteint ;
    - }
    - ParcoursProfondeurRec(Sommet entree) ;
  - }
  - ParcoursProfondeurRec(Sommet x){
    - etat[x] ← atteint ;
    - pour chaque y voisin de x{
      - si etat[y] == nonAtteint {
        - print(x) ;
        - ParcoursProfondeurRec(y) ;
      - }
    - }
    - etat[x] ← traité ;
    - print(- x)
- 2 montrons que l'agent parcours chaque arête exactement deux fois .
  - Chaque sommet est vu >= 1fois.
  - Chaque arête est vue >= 2 fois.

- Cas général : on considère maintenant un graphe quelconque, et supposons de plus que l'on associe a chaque arête un longueur positive(la longueur du couloir respectif). Nous souhaitons trouver le chemin le plus court. Nous cherchons une solution convenable, même si celle ci n'est pas optimale.
  - Tarpm (ArbreRecouvrantPoidsMinimum).
  - Quelque soit un Tour Poids(Tarpm) < Poids(Tour)
  - preuve : le tour Tour Contient Strictement un arbre recouvrant T, il suffit de supprimer des arêtes du Tour , sans détruire la propriété de connexité, poids(Tour) > poids(T)≥poids(Tarpm).

- Nous souhaitons un algorithme pour le voyageur de commerce
- VoyageurDeCommerce2Approx
- Entrée : un graphe non-orienté G, poids sur les arêtes

- Sortie : Tour « pas trop mal »
  - Tarpm ← ArbreRecPoidsMin(G,poids) // algorithme de Kruskal,prim etc..
  - Tourt ← ParcourProfondeur(Tarmp) ;
  - return Tourt ;
- Théorème : le tour calculé par l' algorithme est au pire deux fois plus long que le tour optimum. Poids(Tour) = 2poids(Tarmp)// a cause du parcours profondeur
  - poids(Tarpm) < poids(TourOPT). // Q2.1

16/10/12

Problème voyageur commerce

E : graphe G=(V,E) non-orienté, poids strictement positifs sur les arêtes
S : Tour : cycle passant au moins une fois par chaque sommet.
Objectif : minimiser le poids du Tour.
Ce que l'on a : VoyageurDeCommerce2Approx

- polynomial : O(m.log(n))
- calcul un tour
- poids(Tour) < 2*Poids(TourOPTIMAL)
- on a donc un algorithme de 2-approximation pour un problème d'Optimisation, 2-approximation car la solution est au pire 2 fois l'optimum.

- Définitions :
  - problème d'optimisation :
    - Entrée. (Instance)
    - Solution admissible : dépend de l'entrée et satisfait certaine contraintes quelque chose d'acceptable faute de mieux.
    - Objectif : soit maximiser un bénéfice, soit minimiser un coût.
    - Exemple : TSP (Travelling Salesman Problem)
      - Entrée : graphe+ poids
      - solution admissible : Tour (cycle passant au moins un fois par chaque sommet)
      - objectif : minimiser le poids du tour .
    - Exemple 2 : plus court chemin
      - entrée : graphe , poids, sommet source, sommet destination.
      - Sortie : chemin de la source a destination
      - objectif : minimiser le poids du chemin
    - Idéalement , pour un problème d'optimisation, on veut un algorithme
      - efficace , au moins polynomial
      - calculant la solution optimale
      - On sais faire pour beaucoup de problème(plus court chemin, arpm, prog linéaire)
    - Constat : beaucoup de problème d'optimisation sont « difficiles »
      - définition formelle : S2
      - en clair : on n'a pas d'algorithme efficace qui trouve la solution

optimale.

- Que faire des problèmes difficiles ?
  - Algorithmes qui calculent une solution optimale, quitte à mettre un temps exponentiel. (branch_and_bound , en programmation linéaire)
  - Heuristiques : algorithme souvent efficace qui trouve des solutions admissibles mais sans garantie sur l'objectif
    - méta-heuristique : algorithmes génétiques, recherche tabou, recuit simulé
  - Algorithmes de C-Approximation pour le problème P (C : constante >1).
    - algorithme polynomial
    - trouve une solution admissible.
    - Garantie sur l'objectif
    - C est le facteur de garantie
    - Si P est un problème de minimisation,
      - je suis C-content si le cout de la solution de l'algo est ≤ C * cout(Optimum)
    - si P est un problème de maximisation
      - je suis C-content si le bénéfice de l'algorithme est ≥ 1/C * bénef(Optimum
- Lemme : Soit H un graphe , le nombre de sommets de degré impair de H est pair.
  - VoyageurDeCommerce1.5Approx
    - 1 Tarpm ← ArbreRecPoidsMin(G,poids) ;
    - 2 Vimp : l'ensemble de sommets de degré impair DANS l'arbre Tarpm.
    - 3 Calculer, pour tout x,y € Vimp, un plus court chemin µ(x,y) de x à y dans G.
    - 4 Soit Kimp(Vimp,Eimp) : Clique sur Vimp, poids : pour tout x,y €Vimp , w(x,y) = poids(µ(x,y)).
    - 5 M ← CouplageParfaitPoidsMin(Kimp,w) //
      - couplage : ensemble d'arêtes sans extrémité commune.
      - Parfait : touche tous les sommets.
    - 6 H ← Tarpm
      - pour chaque arête x,y de M
        - ajouter à H les arêtes de µ(x,y) // H sera après un multigraphe.
    - 7 Tour ← CycleEulérien(H).
    - return Tour.
- Algo
  - polynomial
  - calcul un tour
  - Poids(TourAlgo) = Poids(Tarpm) + w(M) // poids M
  - rappel : poids(Tarpm) < poids(TourOPT).

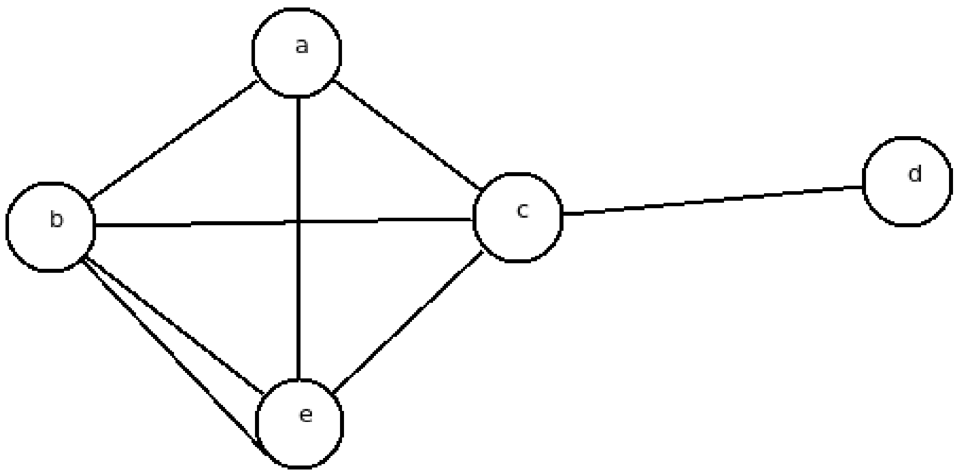
26/10/12

- algorithmes polynomiaux

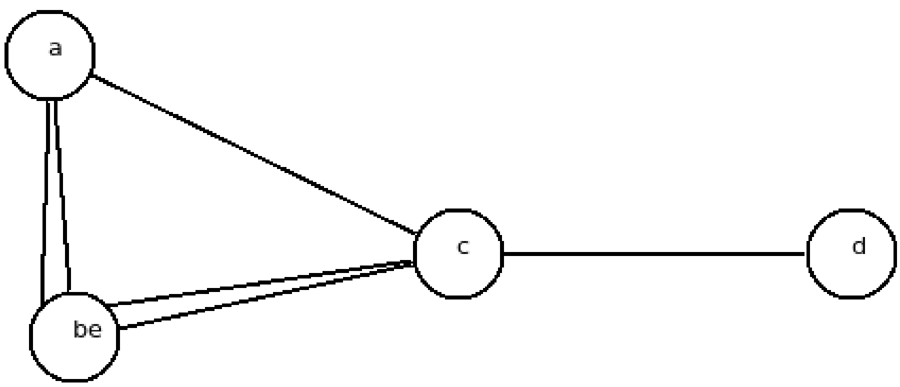
- algorithmes d'approximation
- algorithmes probabilistes
- algorithmes probabilistes d'approximation
- …

- Multigraphe G
- contraction de l'arête xy :
  - les deux sommets x et y sont contractés en un seul sommet appelé xy.
  - Les anciennes arêtes entre x et y sont supprimés(sinon boucles)
    - chaque arête de type xz(ou yz) devient une arête entre le nouveau sommet xy et z

G



G/be // contraction de be



- minCut – Karger – '95
  - Karger & Stein '04
  - coupe minimum
  - Entrée : multi-graphe G non-orienté, sans boucle
  - Sortie : un ensemble d'arête F telles que G-F non connexe
  - Objectif ; minimiser |F|.
  - Observation , toute coupe F de G/xy(G avec les sommets x et y contractés) est aussi une coupe de G.

- DevineCoupeMin(G)
  - pour i de n à 2
    - choisir une arête xy uniformément au hasard
    - G ← G/xy
  - finpour
  - retourner les arêtes restantes
  - fin
- Quelle est la probabilité qu'il est retourné la bonne coupe (optimale)
- Pok(n) : probabilité que l'algorithme retourne une coupe optimale (n = nombre de sommets)
- Ppasok(n) : probabilité que l'algorithme retourne une coupe qui n'est pas optimale.
- Pok(n) = P(nepasecraseruneaaretedelacoupelorsdupremierchoix) × Pok(n-1) // car événement indépendants.
  - ne pas écraser une arête de la coupe lors du premier choix
  - que le graphe avec n-1 sommets renvoi toujours la coupe que je cherche.
- Bonne nouvelle Pok(2) = 1.
- P(nepasecraseruneaaretedelacoupelorsdupremierchoix) = 1- |F|/m ≥ 1-2/n
- lemme : m ≥ |F|.n /2
- pour tout sommet x , degré(x) ≤ |F|
- Σ degré(x) = 2m ≥ n.|F|
- Pok(n) ≥ (n-2)/n Pok(n-1). & Pok(2)=1 .
- Pok(n) ≥ 2/(n.(n-1))
- KARGERCOUPEMIN(G)
  - N←
  - mintrouv = infini
  - meilleure coupe ← null
  - répéter N fois
    - coupe ← DevineCoupeMin(G) ;
    - if(|coupe| < mintrouve)
      - mintrouv ←|coupe|
      - meilleure coupe←coupe
  - return meilleure coupe
- théorème : Soit N = c.n.(n-1)/2.ln(n)
  - la probabilité que KARGERCOUPEMIN trouve la coupe optimale est ≥ 1-1/N^c*
- Pkargerpasok // probabilité qu'il se goure, mais pour cela il doit ce gourer à chaque itération
  - Pkargerpasok(n) = [Ppasok(n)]^N ≤ (1- 2/(n*(n-1)))^N
- lemme : 1-x ≤ 1/e^x
- donc Pkargerpasok(n) ≤ 1/(e^(2N/(n*(n-1)))) ≤ 1/e^(ln(N).c) 1/(n^c) * f
- ContractJusquaM(G,m){
  - pour i de n à m{
    - choisir une arête uniformément au hasard et la contracter
  - }
- retourner G.

- }
- devinerMieuxCoupeMin(G){
  - si n == 2 {
    - retourner arêtes
  - }
  - Coupe1 ← devinerMieuxCoupeMin(contractJusquam(G,n/root(2))
  - Coupe2 ← devinerMieuxCoupeMin(contractJusquam(G,n/root(2)).
  - return min(Coupe1, Coupe 2).

7/11/12

- Techniques algorithmiques avancées
- algorithme d'approximation
- algorithme probabiliste
  - algorithme qui à un moment donné, fais un choix aléatoire
  - Objectifs :
    - Algorithmes Las Vegas
      - donnent toujours la bonne réponse
      - on utilise l'aspect probabiliste pour gagner du temps en moyenne
      - ex : quicksort
        - trie correctement
        - complexité en moyenne est de l'ordre de O(nlog(n)).
    - Algorithmes Monte-Carlo
      - ne donnent pas toujours la bonne réponse
      - la probabilité que la réponse de l'algorithme soit égale à la bonne réponse tend vers 1 lorsque la taille de l'entrée est grande.
        - P[algodonnebonneréponse] ≥ 1/2+ e où e > 0.
      - exemple : algorithme kargercoupemin
      - Objectif des Monte-Carlo
        - complexité meilleure que les algorithmes déterministes pour le même problème
- algorithme probabiliste de C-approximation
  - pour des problèmes d'optimisation
  - temps polynomial
  - solution rendue admissible
  - pb de minimisation
    - E[cout(sol algo)] ≤ c.cout(solOpt).
    - En moyenne
  - pb de maximisation
    - E[bénef(sol algo)] ≥ bénef(solOpt) / C
  - objectifs des algorithmes probabilistes d'approximation
    - aller plus vite que l'algorithme déterministe
    - avoir une meilleure approximation que ce que l'on sais faire avec les algorithmes déterministes d'approximation
- problème vertex cover = transversal minimum
  - entrée : G(V,E), non orienté,

- sortie : un transversal, un ensemble de sommets tel que chaque arête touche un sommet de l'ensemble.
- Objectif, minimiser|T|
- bad news : transversal-minimum est Np-difficile
- T ← Ø
- pour chaque arête uv
  - if (u ¢T et v ¢ T)
  - choisir x parmi {u,v}
  - T ← T U {x}
- fin tant que
- return T ;
- O(n+m)
- théorème
  - algorithme VCproba est une 2-approximation probabiliste, c'est à dire que le facteur 2 de l'approximation est garantie en moyenne.
  - Variable aléatoire cout : X = |T|
  - E[X] ≤ 2 ×|Topt|
  - Soit Ti le transversal après i itérations
  - montrons que pour tout i
    - E[|Ti ∩ Topt|] ≥ E[|Ti -Topt|]
    - bons sommets ≥ mauvais sommets
  - Xibon = 1 si le sommet choisi a l'étape i est dans topt, 0 sinon
  - E[X] = E[somme Xj] = somme E[Xj]
  - Xj = 1× P[sommet choisi a l'étape i est bon]
  - E[X] ≥ ½ .|Ti|
- Vcproba (graphe G(E,V), w : V→N(retourne le poids du sommet))
  - T←Ø
  - pour chaque arêtes uv
    - si u ¢ T et v ¢ T
      - choisir x parmi u et uv aléatoirement selon
        - x←u avec probabilité w(v)/(w(v)+w(u))
        - x←v avec probabilité w(u)/(w(v)+w(u))
      - T ← T U {x}
  - return T
- théorème : Vcproba est une 2-approximation probabiliste pour VertexCoverPondéré
  - Tbon = T inter Topt
- Objectif 1 : algorithme déterministe de 2-approximation pour vertexCovernonpondéré
- objectif 2 : idem mais pondéré
- algorithme vertexcoverGlouton
  - T←Ø
  - tant que G à des arêtes
    - x ← sommets de degré maximum
    - T←T U{x}
    - G ← G – x
  - return T ;
- Algorithme VCCouplageMinimal

- T ← Ø // ensemble des sommets
- M ← Ø
- pour chaque arête uv
  - si U¢T et v ¢T
    - M ← M U (uv)
    - T← T U {u,v}
- return T ;

- Morale
  - algorithmes probabilistes d'approximation
    - souvent simples
    - preuves non triviales : non intuitives
    - E(cout(sol algorithme)) ≤ c.cout(solopt)
    - souvent on est proche de la moyenne
  - algorithme déterministes d'approximation
    - si , pour un problème , on a un algorithme déterministe d'approximation, aussi bon que l'algorithme probabiliste , alors il est préférable d'utiliser le déterministe
  - comment prouver qu'un algorithme déterministe est une C-approximation.
    - Problèmes de minimisation
    - difficultés : on ne connaît pas la solution optimale
      - pour la contourner borne inférieure , que l'on sait calculer

---

23/11/12

- Programmation linéaire
- Autres techniques pour les problèmes d'optimisation difficiles
  - Algorithmes paramétrés
  - Algorithmes exacts // la complexité sera typiquement exponentielle.
    } non traités
  - Heuristiques }
- Rappels : programmation linéaire
  - variables x1,...,xn
  - constantes aij, bij, cj
  - Trouver l'affectation de xi qui minimise(maximise) ∑contrainte * xi
    - les contraintes sont de la forme ax1+zx2+...+ ≤ un truc...
- passer de TransversalMinPondéré à un programme linéaire en nombres entiers
- Variables :
- pour tout sommet i , une variable xj
  - signification :
    - xj = 0 si j ∉ Topt
    - xj = 1 si j ∈ Topt
- Objectif
  - minimisation de ∑ w[j].xj
- Contraintes
  - ∀j , 0≤ xj ≤ 1

- xj est entiers.
- ∀arete kl
  - Xk +Xl ≥ 1.
- algorithme TransversalMinPondéré (2-approx déterministes)
  - construire le programme linéaire en nombre entier associé au problème (celui du dessus) appelé PLNE.
  - Soit le programme relaché(sans contrainte de type xj entier)
    - (x1*, x2*,...,xn*) solution du simplexe en PL
    - //solution du Pl relaché
  - //construction du transversal.
    - T ← Ø
    - pour j de 1 à n
      - si xj* ≤ ½
        - xj ← 1.
        - ajouter j à T
      - sinon
        - xj* ← 0
    - return T ;
  - théorème : cet algorithme est une 2-approximation pour TransversalMinimum
    - temps polynomial /!\ toute fois attention au simplexe
    - montrez que T est un transversal.
      - Pour tout k,l, si kl non gardée:alors x'k et x'l = 0
      - donc x*k et x*l < ½, or x*k+x*l ≥ 1 dans résolution du simplexe.
    - Le poids du transversal est au plus deux fois le poids de Topt
      - w(T) ≤ 2*(Topt)
      - w(T) =∑ w[j].x'j ≤ 2* ∑ w[j] x*j // x'j ≤ 2* x*j pour tout j
      - { = 2×Z* // solopt du PL relaché}
      - // Z * ≤ Zopt(PLNE) = w(Topt)
      - donc ≤ 2*w(Topt) CQFD
- Algorithmes paramétrés
  - VertexCoverParam
    - E : G=(V,e), paramètre k « petit »
    - S : existe il un transversal de taille ≤ k ?
  - Algorithme en temps O(2^k × n²).
- Algorithme TransversalAuPlusK(G,k)
  - si k =0
    - si G n'as plus d'arête retourner VRAI
    - sinon return FAUX ;
  - si G n'as pas d'arête
    - retourner VRAI
  - // G a des arêtes
  - soit xy une arête //toutes les arêtes sont non-gardée car on retire les arêtes incidentes à x à chaque fois
    - return TransversalAuPlusK(G\{x},k-1) OU TransversalAuPlusK(G\{y},k-1) ;
- Compléxité
  - /!\ nb Appels récursifs au plus 2^(k+2) -1 (nombre de nœuds dans un

arbre de hauteur k+1)

- un appel $O(n^2)$ car il faut supprimer le sommet et parfois recopier le graphe .
- Meilleur algorithme $O((1.3^k) \times n^2)$.

Conclusion

[Modélisation], graphes et algorithmes

- flots maximums et coupes minimums } problèmes
- graphes planaires } O(poly(n)
  - reconnaissance et coloration } on sais faire

  } polynomial
- Cycles eulériens } algorithmes
  - voyageur de commerce } non triviaux
- Techniques avancées
  - Algorithmes d'approximation
  - algorithmes probabilistes
  - algorithmes paramétrés, exacts, heuristiques
  - dans cette partie les problèmes sont difficiles on a souvent a faire a une approximation, et souvent cette solution approximative est très acceptable