Föreläsning 4: Giriga algoritmer

Giriga algoritmer

Denna typ av algoritmer arbetar efter följande princip: Gör i varje situation det som är lokalt optimalt, d.v.s. bäst för stunden.
Några exempel vi redan sett är Dijkstras algoritm. Där utvidgar vi en mängd S med det hörn som är märkt med lägst värde.
Till många problem finns en naturlig girig algoritm, men ofta ger den inte en optimal lösning.
Giriga algoritmer är ofta snabba så de kan användas för att snabbt få fram en approximativ lösning.
Vi skall nu studera en följd problem som är kopplade till *aktivitetsplanering*

Att klara jobbet med så få personer som möjligt

Vi har n st aktiviteter med givna starttider s_i och sluttider f_i. Det innebär att aktivitet i måste utföras under tiden $[s_i, f_i]$ (hela tiden behövs). Hur många personer krävs för att genomföra samtliga aktiviteter om en person inte kan vara sysselsatt med mer än en aktivitet åt gången?
Iaktagelse: Låt oss anta att det finns en tid x så att x ligger i d olika aktiviteters intervall och att d är maximalt i detta avseende. Då krävs det minst d personer för att utföra jobbet.
Påstående: Aktiviteterna går att genomföra med d personer.
Vi använder en enkel girig algoritm för att lösa problemet. Vi tar d personer $p_1, p_2, \ldots, p_d$. Vi sorterar intervallen så att $s_1 \leq s_2 \leq \cdots \leq s_n$. Tag första aktiviteten och låt p_1 utföra aktiviteten. I fortsättningen gör vi följande: När tiden s_i för en ny aktivitet dyker upp låter vi den första person p_j som inte är upptagen göra den.
Det går enkelt att visa att de d personerna tillsammans klarar av att utföra aktiviteterna. Det *giriga* inslaget i algoritmen är man bara går igenom aktiviteterna i tidsordning och tilldelar dem till personer utan att göra någon mer övergripande planering (som inte kan löna sig).

Aktivitetsplanering för ensam person

Vi har n st aktiviteter $a_1, a_2, ..., a_n$ som äger rum under tidsintervall på formen $[s_i, f_i)$. Inga intervall får överlappa varandra. (Intervallen är definierade som *halvöppna*. Det innebär t.ex. att $[2, 4)$ och $[4, 5)$ inte överlappar varandra.) Vi vill välja ut ett maximalt antal aktiviteter att utföra. Hur väljer man ut ett maximalt antal aktiviteter som inte överlappar varandra?

Girig algoritm för aktivitetsplanering

Lite överraskande visar det sig att det är sluttiderna för aktiviterna som är det viktigaste. Sortera efter **sluttid** så att $f_1 \leq f_2 \leq ... \leq f_n$. vi vill välja en mängd A av aktiviteter som vi skall utföra.

```
(1)   A ← {a_1}
(2)   i ← 1
(3)   for j ← 2 to n
(4)       if s_j ≥ f_i
(5)           A ← A ∪ {a_j}
(6)           i ← j
(7)   return A
```

Det giriga i denna algoritm består i att vi hela tiden väljer det lediga intervall som slutar först. Det går ganska lätt att visa att algoritmen ger optimalt resultat.

Jobb med deadlines

Vi har nu n st jobb som skall utföras av en person. Jobb i kräver en viss tid t_i att utföra. Jobb i måste också vara klart vid en viss tidpunkt d_i. I övrigt kan jobben utföras när man vill. Vi skall göra en planering av jobben så att $s(i)$ är starttiden för jobb i och $f(i)$ är sluttiden. Vi har följande krav:

- $f(i) = s(i) + t_i$ för alla i.
- Inga intervall $[s(i), f(i)]$, $[s(j), f(j)]$ får överlappa varandra.
- $f(i) \leq d_i$ för alla i.

Ett första problem är att avgöra om detta är möjligt och hur planeringen då ser ut.
Om det inte går att genomföra planeringen måste vi ha $f(i) > d_i$ för minst ett i. Vi är då intresserade av att "minimera misslyckandet". Det finns olika idéer om vad vi skall försöka minimera. En naturlig idé är följande:
Sätt $L = \max_i f_i - d_i$.
Försök sedan få L så liten som möjligt. Vi minimerar maximal försening.
Algoritmen för planering är mycket enkel. Sortera först jobben så att $d_1 \leq d_2 \leq \cdots \leq d_n$. Vi antar att alla $d_i > 0$ och att vi börjar arbeta vid tiden 0.

```
(1)   s(1) ← 0, f(1) ← t_1
(2)   for i ← 2 to n
(3)       s(i) ← f(i − 1), f(i) ← s(i) + t_i
(4)   return s, f
```

Det giriga består i att vi först väljer det jobb som måste vara klart först o.s.v. Det går att visa att denna algoritm ger optimalt resultat även om beviset är lite mer invecklat än bevisen för de andra algoritmerna.
Det finns andra förslag på mått på misslyckandet. Vi vill kanske att vi skall missa så få deadlines som möjligt. Den typen av krav är inte lika lätta att klara av med giriga algoritmer.

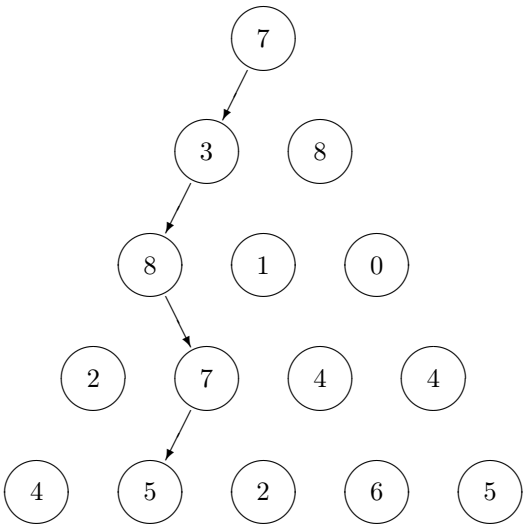
Passande problem

En girig algoritm passar i situationer där man kan genomföra följande resonemang:
1. Problemet kan delas upp i ett första girigt val som lämnar kvar ett delproblem.
2. Det går att visa att om det finns en optimal lösning L på problemet så går det alltid att modifiera L till en ny optimal lösning L' som har samma första val som vårt val. ("Första valet kan inte vara fel")
3. Vi kan genom induktion eller motsvarande visa att vår algoritm ger rätt resultat. Den optimala lösningen på delproblemet går alltid att kombinera med det giriga valet och ger då en optimal lösning.

Ibland kan en girig algoritm naturligtvis misslyckas:

Exempel: Ett triangelproblem

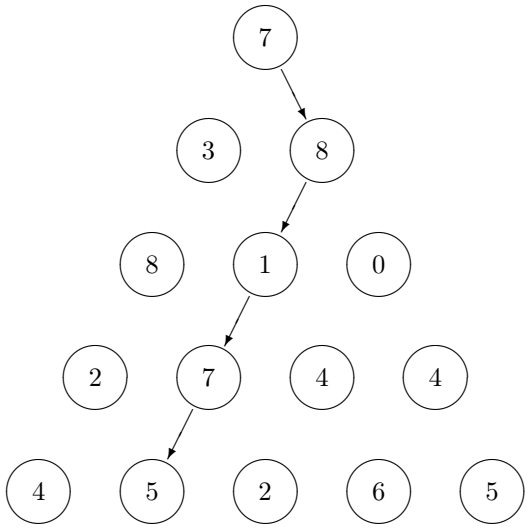
Problem: Hitta stigen från toppen till botten som maximerar summan av de ingående talen.



Låt n vara antalet rader. Det finns 2^{n-1} olika stigar att prova.

Girig algoritm för triangelproblemet

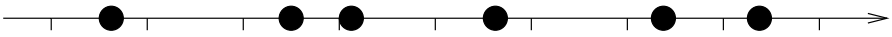
Gå i varje steg till det största av de två talen på raden nedanför.



Den giriga algoritmen ger här summan 28 medan den bästa stigen ger 30 ($7 + 3 + 8 + 7 + 5$).

Girig algoritm för intervalltäckning

Givet är n punkter på tallinjen. Vi vill hitta ett minimalt antal intervall av längd 1 vars union innehåller alla punkterna.



Girig algoritm:

(1) Sortera punkterna i växande ordning.
(2) **while** någon punkt finns kvar
(3) Placera ett intervall med vänstra ändpunkten i punkten med lägst koordinat.
(4) Ta bort alla punkterna som täcks av detta intervall.

Algoritmen ger den minimala övertäckningen eftersom något intervall måste täcka punkten längst till vänster; den giriga placeringen av intervallet täcker alla punkter som skulle täckas av någon annan placering.

Kruskals och Prims algoritmer

Spännande träd: Om G är en sammanhängande graf så är ett spännande träd ett träd som innehåller alla noder i $V(G)$.

Vi antar att vi har kantvikter $w(e)$ på alla kanter i G.

Hur hittar man spännande träd med minimal vikt? (Kallas Minimala Spännande Träd , MST.)

Kruskals algoritm för MST

Idén bakom denna algoritm är upprätthålla en spännande skog och förena delträden utan att skapa någon cykel tills ett enda stort träd bildats.

```
KRUSKAL(V, E, w)
(1)   A ← ∅
(2)   foreach v ∈ V
(3)       MakeSet(v)
(4)   Sortera E efter stigande vikt
(5)   foreach (u, v) ∈ E (ordnat efter stigande vikt w)
(6)       if FindSet(u) ≠ FindSet(v)
(7)           A ← A ∪ {(u, v)}
(8)           MakeUnion(u, v)
(9)   return A
```

Tidskomplexitet: $O(|E| \log |E|)$ (p.g.a. sorteringen); FindSet och MakeUnion tar $O(\log |V|)$ tid.

Prims algoritm för MST

Prims algoritm bygger på samma principer som Dijkstras algoritm. I varje steg läggs det hörn till trädet som ligger närmast (d.v.s. har lägst märkning *key*).

```
PRIM(V, E, w, s)
(1)   key[v] ← ∞ för varje v ∈ V
(2)   key[s] ← 0
(3)   Q ← MakeHeap(V, key)
(4)   π[s] ← Null
(5)   while Q ≠ ∅
(6)       u ← HeapExtractMin(Q)
(7)       foreach granne v till u
(8)           if v ∈ Q och w(u, v) < key[v]
(9)               π[v] ← u
(10)              key[v] ← w(u, v)
(11)              Justera heapen vid v
```

Tidskomplexitet: $O(|V| \log |V|) + O(|E| \log |V|) = O(|E| \log |V|)$

Korrekthetsbevis: Bygger på följande hjälpsats. Vi antar att inga kanter har samma vikt.
Ett *snitt* S i grafen G är en uppdelning $V = S \cup (V - S)$.
Sats.
Låt S, $V - S$ vara ett snitt (så att ingen av S eller $V - S$ är tom). Låt $e = (u, v)$ vara sådan att den går från S till $V - S$ d.v.s. $u \in S$ och $v \notin S$ och så att e:s vikt är minimal. Då måste e finnas med i varje minimalt spännande träd.
Bevis: Låt T vara ett MST som inte innehåller (u, v). Betrakta stigen från u till v längs T. Den innehåller någon kant (x, y) som går från S till $V - S$. Eftersom (u, v) hade minst vikt av alla såna kanter är $T + \{(u, v)\} - \{(x, y)\}$ ett MST med lägre vikt, vilket är omöjligt.
Bevis för Kruskal: Låt (u, v) vara en kant som väljs i något steg i Kruskals algoritm. Låt S vara alla noder som kan nås från u via stigar över de kanter *som hittills valts*. Då gäller $u \in S$ och $v \notin S$. Det går också att se att (u, v) måste vara en minimal kant över snittet. Enligt satsen kan det inte vara fel att välja (u, v)
Bevis för Prim: Här låter vi S vara de noder som valts i algoritmen fram till ett visst steg. Kanten (u, v) är precis en sådan kant som anges i hjälpsatsen. Valet av den kan inte vara fel.