Föreläsning 7 i ADK

Algoritmkonstruktion: giriga algoritmer, totalsökning

Stefan Nilsson

KTH

Metod 1: Giriga algoritmer (greedy algorithms)

Algoritmer som löser, iterativt, en bit av problemet i taget. I varje steg görs det som ger bäst utdelning/kostar minst

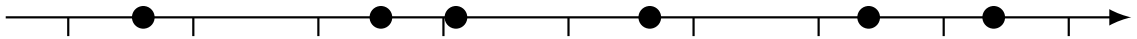
- Giriga algoritmer är speciellt användvara för approximationsalgoritmer och heuristiker
- Ibland kan dock giriga algoritmer ge optimala lösningar. Då måste man bevisa att algoritmen gör det

Exempel på problem som kan lösas optimalt med en girig algoritm:

- Hitta kortaste vägen mellan två punkter i en graf med kantvikter
- Hitta minimala trädet som spänner upp en graf
- Givet en mängd aktiviteter med start- och sluttider, hitta största mängden aktiviteter som inte överlappar

Girig algoritm för intervalltäckning

Givet är n punkter på tallinjen. Hitta minimalt antal intervall av längd 1 vars union innehåller alla punkterna!



Girig algoritm:

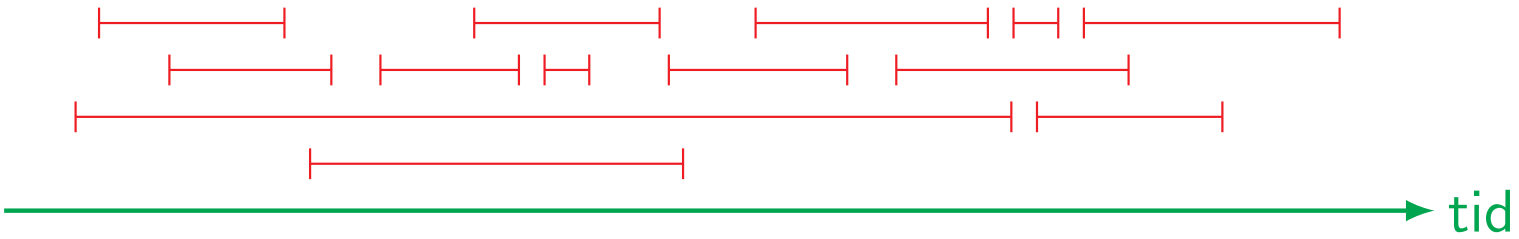
Sortera punkterna i växande ordning
while någon punkt finns kvar **do**
    Placera ett intervall med vänstra ändpunkten
    i punkten med lägst koordinat.
    Ta bort alla punkter som täcks av detta intervall.

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

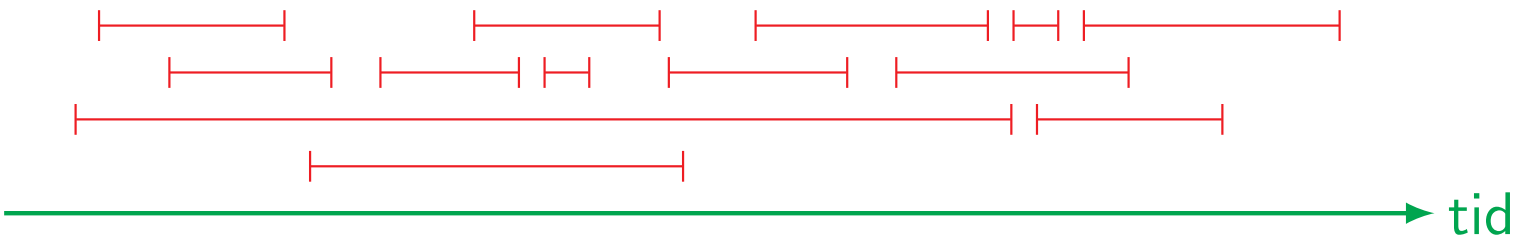
Aktivitetsproblemet

Indata: En mängd med n aktiviteter beskrivna av starttid s_i och sluttid f_i
Utdata: En maximal delmängd av aktiviteter som inte överlappar, dvs så många aktiviteter som möjligt som en person kan delta i utan att det blir någon krock.

I exemplet nedan representeras varje aktivitet av ett rött intervall.



Girig algoritm för aktivitetsproblemet



if $\mathtt{n} = 0$ **then return** $\emptyset$
Sortera aktiviteterna efter sluttid f_i så att $f_1 \leqslant f_2 \leqslant f_3 \leqslant \cdots \leqslant f_n$
$\mathtt{M} \leftarrow \{1\}$ // *Lägg in första aktiviteten i lösningsmängden*
$\mathtt{current} \leftarrow f_1$
for $\mathtt{i} \leftarrow 2$ to n **do**
    if $s_i \geqslant \mathtt{current}$ **then**
        $\mathtt{M} \leftarrow \mathtt{M} \cup \{i\}$ // *Aktivitet i behöver läggas till i lösningsmängden*
        $\mathtt{current} \leftarrow f_i$
return M

Girig algoritm för aktivitetsproblemet

Bevis:

- Låt D vara en optimal delmängdsaktiviteter sorterade efter stigande sluttid
- Om D är tom eller består av ett element för algoritmen rätt
- Låt k vara första aktiviteten i D
  - Om $k = 1$ väljer algoritmen samma aktivitet (1)
  - Om $k \neq 1$ väljer algoritmen en aktivitet med sluttid $f_1 \leqslant f_k$
  - Vi kan alltså byta k mot 1 i D och fortfarande ha en lösning!
- Ingen aktivitet med starttid $\leqslant f_1$ kan vara med i D, så vi plockar bort dessa.
- Nu har vi samma problem fast med en mindre mängd aktiviteter (som inte överlappar med 1).
- Induktion!

Metod 2: Totalsökning (exhaustive search)

- Gå igenom alla tänkbara lösningar och kolla om det är den sökta lösningen
- Detta görs med fördel rekursivt
- Ofta är antalet tänkbara lösningar exponentiellt många (t.ex. 2^n) och då går det bara att använda för små n. Totalsökning är en metod man tar till i sista hand.
- Det svåraste med totalsökning är normalt att se till att man går igenom varje lösning en (och helst inte mer än en) gång. Att sedan kolla om det är den sökta (eller optimala) lösningen brukar vara lätt.
- Ibland kan man redan innan en tänkbar lösning är färdigkonstruerad se att den inte är möjlig som lösning. Då kan man strunta i att gå vidare med den och istället gå tillbaka och konstruera nästa möjliga lösning. Detta kallas **backtracking**.
- Exempel: Lösning av det generella handelsresandeproblemet.

Lösning av generell TSP med totalsökning

Datastrukturer:

- `perm[1..n]`: Här konstruerar vi varje tänkbar lösning, dvs permutation av städerna
- `visited[1..n]`: Anger om en stad besökts hittills i den permutationen som konstrueras

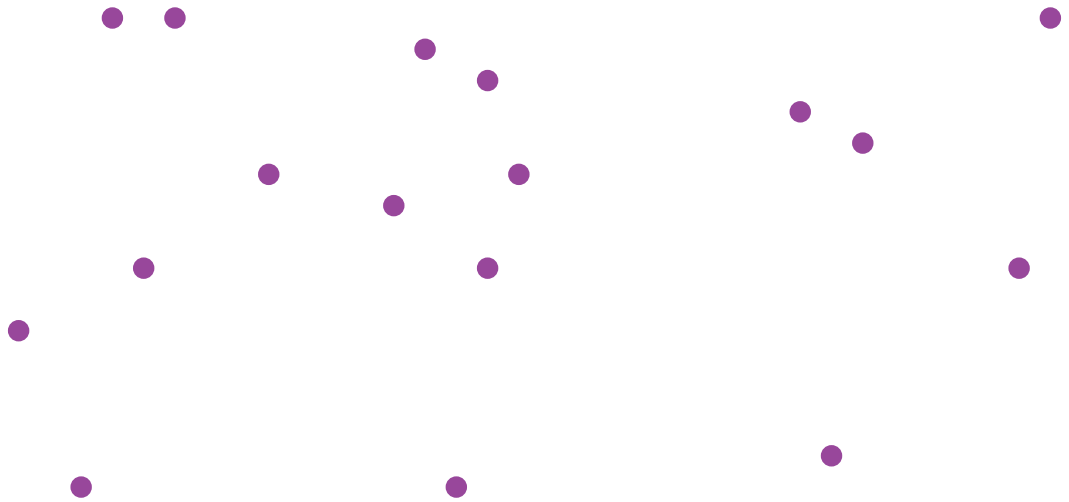
```
function TSP(n,d[1..n, 1..n])
    minlen ← ∞
    for i ← 1 to n do visited[i] ← false
    for i ← 1 to n do
        perm[1] ← i
        visited[i] ← true
        CheckPerm(2, 0)
        visited[i] ← false
    return minlen
```

Lösning av generell TSP med totalsökning

```
function CheckPerm(k,length)
    if k > n then
        totalLength ← length + d[perm[n], perm[1]]
        if totalLength < minlen then minlen ← totalLength
    else
                                    ▷ Backtracking kan införas här med satsen
                                    ▷ if length < minlen then
        for i ← 1 to n do
            if not visited[i] then
                perm[k] ← i
                visited[i] ← true
                CheckPerm(k+1, length+d[perm[k-1], i])
                visited[i] ← false
```

Tidskomplexitet: $\mathcal{O}(n^2 n!)$ vilket enkelt kan minskas till $\mathcal{O}(n!)$ om man håller reda på vilka städer som ännu inte besökts effektivare (t.ex. med en kö)

TSP - Handelsredandeproblemet (traveling salesperson problem)



Hitta kortaste turen som passerar alla städer en gång.

Olika varianter av problemet:

- Generell TSP
  Probleminstans: Graf med kantvikter
- Euklidisk TSP i dimension d
  Probleminstans: Städerna givna som koordinater i $\mathbb{R}^d$

8-damsproblemet

Placera ut 8 damer på ett schackbräde utan att någon pjäs hotar någon annan!

Totalsökningsalgoritm:

Placering av en dam på rad `row`:

- Pröva att placera damen i varje position på raden i tur och ordning
- Kan damen stå ohotad i den positionen så placeras nästa dam på rad `row+1` ut på samma sätt, om inte `row` $= 8$ för då har man hittat en lösning.

```
function TestRow(row)
    for i ← 1 to 8 do
        queenpos[row] ← i // Vi testar en dam i ruta (row,i)
        if PosOK(i) then
            if row = 8 then WriteSolution
            else TestRow(row+1)
```

Starta med TestRow(1)