

INF2220 - Algoritmer og datastrukturer

HØSTEN 2008

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 10:
Disjunkte Mengder

Bjarne Holen (Ifi, UiO)

INF2220

H2008, forelesning 10

1 / 1

Eksempel

\leq relasjonen på $\mathbb{N} \times \mathbb{N}$, med vanlig tolkning av \leq

- ▶ $(3, 5) \in \leq$
- ▶ $(5, 3) \notin \leq$

Binære Relasjoner

- ▶ $A \times A = \{(x, y) \mid x, y \in A\}$: ordnede par over A
- ▶ $|A| = n$ så har vi n^2 ordnede par
- ▶ \mathcal{R} er en binær relasjon over A hvis: $\mathcal{R} \subseteq A \times A$
- ▶ Hvis $(a, b) \in \mathcal{R}$ sier vi at
 - ▶ a er relatert til b
 - ▶ a står i forhold til b via \mathcal{R}

Bjarne Holen (Ifi, UiO)

INF2220

H2008, forelesning 10

3 / 1

Bjarne Holen (Ifi, UiO)

INF2220

H2008, forelesning 10

2 / 1

Representasjon av Binære Relasjoner

En binær relasjon kan representeres ved en **rettet graf**

- ▶ Det er en kant fra a til b hviss $a \mathcal{R} b$

En binær relasjon kan representeres ved en **boolsk matrise**

	a	b	c	d	e
a	0	0	0	1	1
b	1	1	1	1	0
c	0	1	1	0	0
d	0	1	1	1	0
e	1	0	0	1	1

Konstant tidsoppslag

Kvadratisk minnepllass n^2

4 / 1

Ekvivalensrelasjoner

En **ekvivalensrelasjon** er en binær relasjon med følgende egenskaper:

- ▶ $a \sim a$ for alle elementer, (**refleksivitet**)
- ▶ $a \sim b \rightarrow b \sim a$ (**symmetri**)
- ▶ $a \sim b \wedge b \sim c \rightarrow a \sim c$ (**transitivitet**)
- ▶ **Eksempel**
 - ▶ Likhet (=) for tall
 - ▶ Har bursdag på samme dag for mengden av mennesker
 - ▶ Byer der en kan kjøre bil fra en til en annen uten å bruke ferge
 - ▶ Elever som går i samme skoleklasse

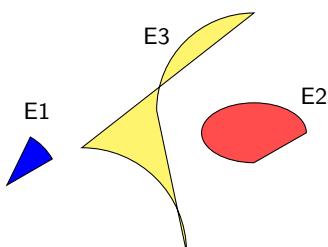
Eksempel

Ekvivalensklasser vi får dersom vi bruker bilveirelasjonen der vi tilhører samme ekvivalensklasse hvis det er mulig å kjøre bil fra a til b uten å bruke ferje.

$$E_1 = \{\text{Longyearbyen}\}$$

$$E_2 = \{\text{Svolvær, Kabelvåg, Stamsund, Leknes, Reine, Sørvågen}\}$$

$$E_3 = \{\text{alle andre tettsteder i Norge}\}$$

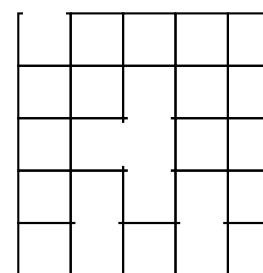


Ekvivalensklasser

- ▶ En ekvivalensrelasjon på en mengde S , deler mengden inn i forskjellige **ekvivalensklasser**, slik at alle elementene i en ekvivalensklasse er relatert til alle andre elementene i denne klassen, men ikke til noen elementer i andre ekvivalensklasser.
- ▶ Denne egenskapen fører til at elementene i ekvivalensklassene er **disjunkte** delmengder av S .
- ▶ Grafen til en ekvivalensrelasjon består av frakoblede subgrafer.

Eksempel

En labryint over et rutenett der vi sier at to ruter er i samme ekvivalensklasse dersom det finnes en vei mellom dem.



Det Dynamiske Ekvivalensproblem

- ▶ Det dynamiske ekvivalensproblem består i å avgjøre om to elementer a og $b \in S$ står i relasjon til hverandre, dvs. om $a \sim b$ for en gitt ekvivalensrelasjon \sim .
- ▶ Hvis alle relasjonene mellom elementene er gitt eksplisitt ved en 2-dimensjonal boolsk matrise, behøver vi bare et enkelt oppslag for å avgjøre om $a \sim b$.
- ▶ Problemets er at vi ofte ikke får oppgitt en slik matrise, pga. plassproblemer med en slik datastruktur for store mengder.

Disjunkt Mengde ADT

Den abstrakte datatypen vi skal se på implementerer to operasjoner

- ▶ `find(a)` returnerer en representant for ekvivalensklassen til a . Representanten er alltid den samme uavhengig av hvilken a i ekvivalensklassen man oppgir.
 - ▶ `union(a, b)` legger inn opplysning om at $a \sim b$
- Operasjonen heter 'union' fordi effekten av å legge inn opplysning om at $a \sim b$, blir at ekvivalensklassene til a og b blir slått sammen (fordi de nå må tilhøre samme ekvivalensklasse).

Det Dynamiske Ekvivalensproblem

- ▶ Det er vanlig at en får oppgitt noen relasjonsforhold, så må en beregne hvilke elementer som er i de forskjellige ekvivalensklassene. Dvs. en kan få oppgitt at $a \sim b$, $c \sim d$ og $e \sim b$ osv. . . Så må en ut ifra den informasjonen kunne bestemme om to elementer er i samme ekvivalensklasse.
- ▶ Eksempel: labrinten, der vi bare får vite hvilke veggene som er revet
- ▶ **Viktig observasjon:** for å bestemme om $a \sim e$ så er det nok å sjekke at de tilhører samme ekvivalensklasse.
- ▶ Algoritmene vi skal se på implementerer gode måter å sjekke om to elementer er i samme ekvivalensklasse.

Disjunkt Mengde ADT

Vi skal represesertre ekvivalensklassene effektivt. En tom relasjon svarer til at hvert element er i sin egen ekvivalensklasse.

Vi kan løse ekvivalensproblemet på en mengde S ved følgende algoritme:

- ▶ Ved starten av algoritmen er relasjonen tom (hvert element utgjør en distinkt ekvivalensklasse).
- ▶ Når vi får vite at $a \sim b$, bruker vi operasjonen `union(a, b)`.
- ▶ Når vi skal avgjøre om $c \sim d$, sjekker vi om `find(c) == find(d)`.

Algoritmen er dynamisk ved at ekvivalensklassene forandrer seg etter hvert som vi utfører union-operasjonene.

Implementasjon av Disjunkt Mengde ADT

Observasjon 1:

- ▶ Vi sammenligner ikke navnene (verdiene) til elementene direkte.
Alt vi er interessert i er hvilken (ekvivalens)klasse de er i.
- ▶ Dermed kan vi for enkelhets skyld anta at at vi jobber med elementer fra 1 til N og at $E_i = \{i\}$ når algoritmen starter.

Observasjon 2:

- ▶ Vi bryr oss egentlig ikke så mye om navnet på klassen som `find` returnerer.

Det som er viktig er at `find(a)==find(b)`
hvis og bare hvis $a \sim b$
(dvs. at a og b er i samme klasse).

Vi skal vise to hovedstrategier for å implementere disjunkt sett ADT:

- ▶ Den første fokuserer på rask `find`
Den gjør `find` $\mathcal{O}(1)$, mens `union` blir relativt treg, $\mathcal{O}(\log N)$
- ▶ Den andre fokuserer på rask `union`
Den gjør `union` $\mathcal{O}(1)$, mens `find` blir relativt treg, $\mathcal{O}(\log N)$

Det er bevist at man ikke kan få både `find` og `union` $\mathcal{O}(1)$ samtidig.

Men vi skal senere i dagens forelesning vise en strategi som *nesten* oppnår dette.

Implementasjon som gir rask `find`

- ▶ Hvis vi ønsker $\mathcal{O}(1)$ tid for `find`, kan vi bruke et array der vi for hvert element lagrer navnet på ekvivalensklassen:

1	2	3	4	5	6	7
E_1	E_2	E_2	E_3	E_1	E_3	E_3

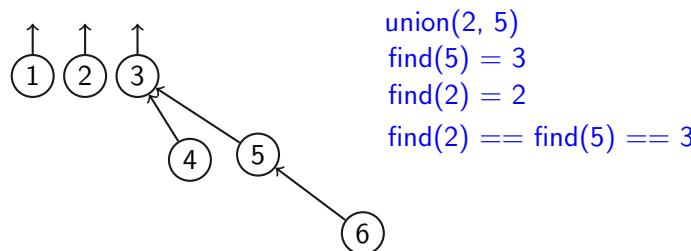
- ▶ Da blir `find` bare et enkelt oppslag i arrayet.
- ▶ Men `union` er kostbar fordi vi må gå gjennom hele arrayet og bytte klassenavnet til alle elementer i klassene som skal slås sammen. Det tar $\mathcal{O}(n)$ tid.

Implementasjon som gir rask `find`

- ▶ Ved å ta vare på størrelsen til hver ekvivalensklasse og alltid la klassen med færrest elementer bytte navn til klassen med flest elementer, kan man garantere at $N - 1$ unioner (som er det meste man kan ha før alle N elementene er i samme klasse) ikke tar mer enn $\mathcal{O}(N \log N)$ tid.
- ▶ Det kommer av at et element a bare kan skifte klassetilhørighet log N ganger fordi antall elementer i klassen til a vil bli minst fordoblet ved hver eneste union hvor a skifter klasse.

Implementasjon som gir hurtig union

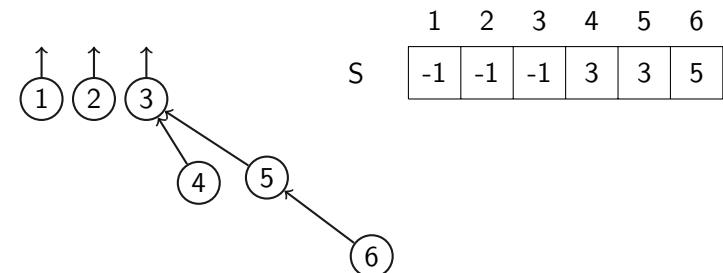
- ▶ Vi implementerer operasjonene til disjunkte sett ved hjelp av en skog, dvs. en mengde trær.
- ▶ Ideen er å plassere alle elementer i en ekvivalensklasse i samme tre, og la roten i treet identifisere ekvivalensklassen.
- ▶ Trærne er ikke binære, men allikevel veldig enkle fordi vi bare behøver å lagre forelderpekeren for å finne rotten, altså ingen barnepekere.



Implementasjon som gir hurtig union

- ▶ Trærne kan representeres ved en enkel array S fra 1 til N :

- ▶ $S[i] == y$ betyr at node nr. i har y som forelder.
- ▶ $S[i] == -1$ betyr at noden er en rotnode.



Implementasjon som gir hurtig union

- ▶ `union` gjøres ved å sette den ene rotpekeren til å peke på den andre rotten. Det tar konstant tid hvis vi allerede kjenner røttene til klassene som skal slås sammen.
- ▶ `find(a)` tilsvarer å traversere forelderpekeren opp til rotten. Tidsforbruket er proporsjonalt med dybden til node a , og den er i verste fall $\mathcal{O}(N)$ (hvis alle nodene er i samme ekvivalensklasse).
- ▶ Gjennomsnittsanalysen til operasjonene er vanskelig.

Union-by-size

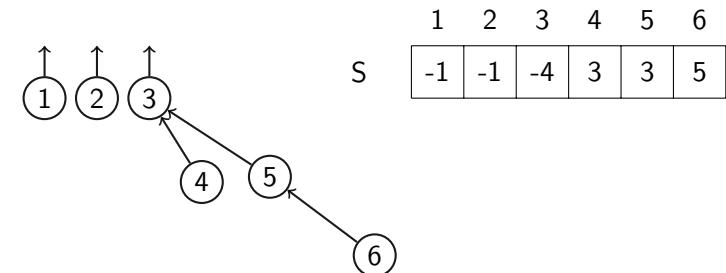
- ▶ Vi kan redusere tidsforbruket til finn-operasjonen ved å bruke en smartere union-strategi:
- ▶ Vi lar alltid det minste treet (færrest elementer) bli et subtre i det største (flest elementer).
- ▶ Med denne strategien, kalt `union-by-size`, blir dybden til et tre maks $\log N$.
- ▶ Det kommer av at når dybden til en gitt node a øker (med 1), så skjer det ved at treet det er med i blir slått sammen med et tre som er større enn seg selv.

Union-by-size

- ▶ Dermed fordobles (minst) antall noder i treet hver gang dybden til øker med 1. Det kan bare skje $\log N$ ganger.
- ▶ Finn-operasjonen blir altså $\mathcal{O}(\log n)$.
- ▶ Vi må lagre størrelsen til hvert tre. Det kan typisk gjøres ved å lagre den **negative** størrelsen i tabellcellen til rotten.

Union-by-size

- ▶ Vi lagrer negativ størrelse for røtter.

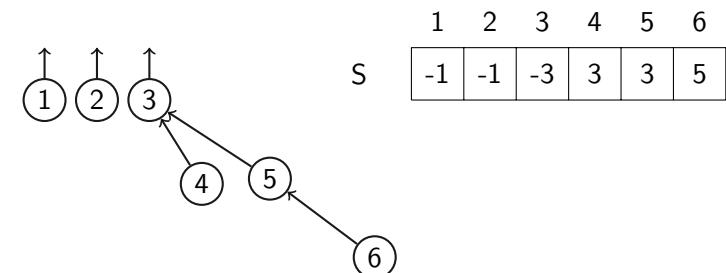


Union-by-height

- ▶ En annen union-strategi er å lagre høyden til hvert tre (den lengste veien fra rotten til en bladnode), og alltid la treet med minst høyde bli subtre av treet med størst høyde.
- ▶ Høyden til det nye treet vil bare øke (med 1) når trærne som slås sammen har samme høyde!
- ▶ Også denne strategien gir $\mathcal{O}(\log N)$ tidsforbruk **worst case**.

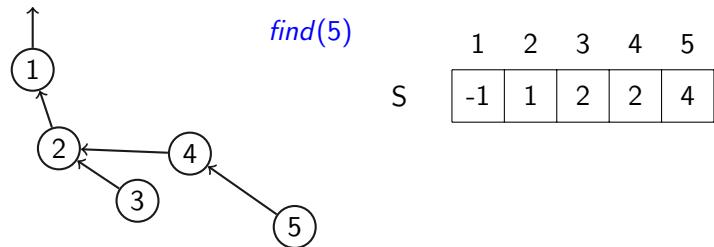
Union-by-height

- ▶ Vi lagrer negativ høyde for røtter.



Stikomprimering

- ▶ Det er sannsynligvis ikke mulig å gjøre union på en smartere måte, så vi kan i stedet prøve en lurere finn-strategi:
- ▶ Når vi skal svare på `find(a)`, kan vi minske dybden til nodene i den grenen som a ligger i ved å forandre på forelderpekerne slik at de peker direkte på rotens.



Labyrint