Disjunkte mengder ADT

Weiss kap. 8.1–8.5

- Løser ekvivalensproblemet
- Lett og rask implementasjon
- Vanskelig tidsforbrukanalyse

Binære relasjoner

$A \times A = \{(x, y) \mid x, y \in A\}$: mengden av **ordnede par** over A.

Hvis A har n elementer, er det n^2 ordnede par.

$\mathcal{R}$ er en **binær relasjon over** A hvis $\mathcal{R} \subseteq A \times A$.

Hvis $(a, b) \in \mathcal{R}$ skriver vi $a\mathcal{R}b$ og sier at a **er relatert til** b, eller at a **står i forhold til** b **via** $\mathcal{R}$.

Eksempel:

$\leq$-relasjonen på mengden av heltall.

$3 \leq 5$ er sann, så 3 er relatert til 5 (med hensyn på mindre-eller-lik-relasjonen). Derimot er $6 \leq 5$ usann, så 6 er ikke relatert til 5.

Representasjon av binære relasjoner

En binær relasjon er en rettet graf:

- A er mengden av noder
- det er en kant fra a til b hvis og bare hvis $a\mathcal{R}b$

En binær relasjon kan representeres ved en boolsk matrise:

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

- Gir konstant-tid oppslag
- Krever kvadratisk minneplass
- For mange anvendelser er n^2 plass ikke akseptabelt

Ekvivalensrelasjoner

En **ekvivalensrelasjon** $\sim$ på en mengde S er en relasjon $\sim$ med følgende egenskaper:

1. $a \sim a$ for alle elementer a i S (**refleksivitet**)
2. Hvis $a \sim b$, så er $b \sim a$ (**symmetri**)
3. Dersom $a \sim b$ og $b \sim c$, så har vi også $a \sim c$ (**transitivitet**)

Eksempel:

En velkjent ekvivalensrelasjon er '='–relasjonen på tall.

Et annet eksempel er **bilveirelasjonen**:

La S være mengden av tettsteder i Norge, og la a og b være to tilfeldige tettsteder. Da er $a \sim b$ hvis og bare hvis det finnes en måte å komme fra a til b ved å kjøre bil uten å bruke ferge.

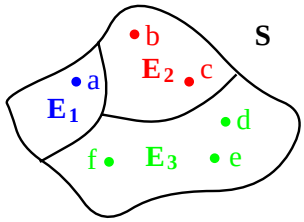
Under forutsetning av at ingen veier er enveiskjørte, er dette en ekvivalens-relasjon.

Et tredje eksempel:

Relasjonen **går i samme klasse som** definert over elevene ved en skole.

Ekvivalensklasser

- En ekvivalensrelasjon $\sim$ på en mengde S deler elementene i S inn i **ekvivalensklasser** slik at alle elementene i en ekvivalensklasse E_i er relaterte til hverandre, men ikke med noen elementer i andre ekvivalensklasser i S.
- Følgelig er ekvivalensklassene **disjunkte** delmengder av S.



- Grafen til en ekvivalensrelasjon består av frakoblede, totale subgrafer.

Eksempel:

Ekvivalensklasser som kan tenkes indusert av bilveirelasjonen definert over mengden av tettsteder i Norge:

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

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

$E_3 = \{$ alle tettstedene på fastlandet $\}$

Eksempel:

En labyrint over et rutenett definerer en ekvivalensrelasjon ved at to ruter er definert hviss det finnes en vei mellom dem.

Ekvivalensklassene består av maksimale mengder av ruter der alle kan nå alle

Det dynamiske ekvivalensproblemet

- **Ekvivalensproblemet** består i å avgjøre om to elementer a og b i 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 i matrisen for å avgjøre om $a \sim b$.
- Problemet er at vi ikke alltid får eksplisitt oppgitt alle verdiene i matrisen, dvs. hvilke elementer som er relatert til hverandre.
- Grunnen til det er at antall ordnede par (mulige relasjonsforhold) vokser kvadratisk (n^2) i forhold til antall elementer i mengden.

- I stedet er det vanlig å få oppgitt noen relasjonsforhold, f.eks. at $a \sim b$, $c \sim d$, $e \sim a$ og $d \sim b$.
  Så må vi raskt kunne svare på om det fra de tre ekvivalens-egenskapene følger at $a \sim d$.
- Vi får altså implisitt informasjon som vi skal "pakke ut"
- Eksempel: labyrint. Vi får kun angitt nabo-rutene!
- **Viktig observasjon:**

  For å bestemme om $a \sim d$, er det nok å sjekke om de er i samme ekvivalensklasse!
- Algoritmene vi skal se på implementerer er gode på å sjekkeom to elementer er i samme ekvivalensklasse.

Operasjonene til disjunkte mengder

Disjunkt mengde ADT tilbyr to operasjoner:

- **Find(a)** returnerer en representant for ekvivalensklassen til element a.

  Representanten er altid den samme, uavhengig av hvilken a i ekvivalensklassen man angir.
- **Union(a, b)** legger inn opplysningen om at $a \sim b$.

  Operasjonen heter 'Union' fordi effekten av å legge inn at $a \sim b$ er at ekvivalensklassene til a og b blir slått sammen (fordi de nå må tilhøre samme ekvivalensklasse).

Vi skal representere 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.

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).

Implementasjon av disjunkte mengder ADT

Vi skal vise to hovedstrategier for å implementerer 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**

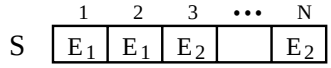
  Den gjør **Union** $\mathcal{O}(1)$, mens **Find** blir relativt treg, $\mathcal{O}(\log n)$

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 finn

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

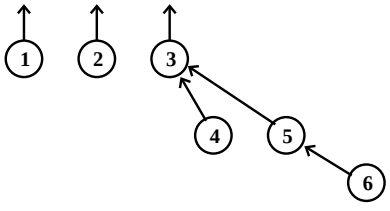


- Da blir **Find** bare et enkelt oppslag i arrayet.
- Men **Union** er kostbar fordi vi må gå gjennom hele arrayet og bytte klassenavnet til alle elementer i klassene som skal slås sammen. Det tar $\mathcal{O}(n)$ tid.

- 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 roten, altså ingen barnepekere.



- Legg merke til at trærne kan representeres ved et enkelt 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.

	1	2	3	4	5	6
S	-1	-1	-1	3	3	5

- **Union** gjøres ved å sette den ene rotpekeren til å peke på den andre roten. Det tar konstant tid hvis vi allerede kjenner røttene til klassene som skal slås sammen.
- **Find(a)** tilsvarer å traversere forelderpekeren opp til roten. 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.
- **Eksempel og programkode:** Figur 8.6–8.9 i Weiss.

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.

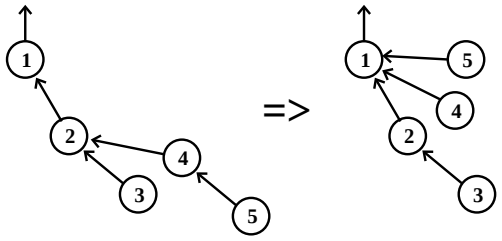
- Dermed fordobles (minst) antall noder i treet hver gang dybden til a øker med 1.
  Det kan bare skje $\log N$ ganger.
- Finn-operasjonen blir altså $\mathcal{O}(\log n)$.
- Vi må lagre størrelsen til hvert tre. Det kan typisk gjøres ved å lagre den **negative** størrelsen i tabellcellen til roten.
- **Eksempel:** Se figurene 8.10–8.12 i Weiss.

Union-by-height

- En annen union-strategi er å lagre høyden til hvert tre (den lengste veien fra roten 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**.
- **Eksempel og programkode:** Se figur 8.13 i Weiss.

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 roten.



Før **Find(5)** Etter **Find(5)**

- Anta at a har b som forelder. Da kan denne strategien enkelt implementeres rekursivt ved å sette $S[a] = Find(b)$, dvs. returverdien av det rekursive kallet til forelderen.
- Denne strategien kalles **stikomprimering**.
- **Eksempel og programkode:** Se figur 8.14 og 8.15 i Weiss.
- I avsnitt 8.6 i Weiss står det et bevis for at dersom man kombinerer stikomprimering med union-by-size eller union-by-height, så vil tidsforbruket til M union/finn-operasjoner bli *nesten* $\mathcal{O}(M)$.

  Beviset er interessant, men komplisert, og *ikke* er pensum i INF2220.

Eksempel: Labyrint

- Vi skal generere en labyrint ved å bruke disjunkt mengde ADTen.
- I utgangspunktet plasseres alle rutene i hver sin ekvivalensklasse.
- En 5 X 6 labyrint vil da se slik ut:

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

- Vi har markert inn- og utgangen fra labyrinten ved at det er åpning i ytterveggen fra rute 0 og rute 29.
- Dette er de eneste åpningene i ytterveggene vi skal tillate.

- Vi trekker så tilfeldig indre vegger i labyrinten og fjerner disse.
- Anta at de fire første veggene vi fjerner, er veggene mellom rute 8 og 14, mellom 13 og 19, mellom 7 og 1, og mellom 20 og 19.
- Da ser labyrinten slik ut:

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

- Vi har nå fått ekvivalensklassene [1,7], [8,14] og [13,19,20], mens alle de andre rutene er sin egen ekvivalensklasse.

- Etter å ha fjernet noen flere vegger, kan labyrinten se slik ut:

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

- Anta at vi her trekker veggen mellom rute 10 og 11.
- Siden **find(10)==find(11)**, er 10 og 11 allerede i samme ekvivalens-klasse, så vi lar veggen stå.
- Anta så at vi trekker veggen mellom 20 og 14.
- Siden **find(20)!=find(14)**, skal vi rive veggen.
- Det gjør vi med **union(20,14)**.

- Fortsetter vi å fjerne vegger, vil vi før eller siden få en labyrint der start- og sluttruten er i samme ekvivalensklasse, f.eks.:

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

- Da har vi en vei gjennom labyrinten og kan avslutte.
- Så er det opp til den enkelte om man vil fortsette til alle rutene er i samme ekvivalensklasse, det vil si at man kan nå alle rutene fra inngangen.