Notater - sammendrag

Comparison sorts

Sorting algorithms

Insertion sort bruker $\Theta(n^2)$ tid WC. Det er allikevel en kjapp *in-place* sorteringsalgoritme for små input.

Merge sort har bedre asymptotisk kjøretid, $\Theta(n \lg n)$,, men MERGE-prosedyren opererer ikke in place.

Heapsort har kjøretid $O(n \lg n)$ og bruker en viktig datastruktur, heap.

Quicksort har WC kjøretid $\Theta(n^2)$. Forventet kjøretid er imidlertid $\Theta(n \lg n)$. Vanligvis bedre enn heapsort - lav skjult faktor. Populær for store input.

> *In place betyr at kun et konstant antall inputelementer blir lagret utenfor området.*

Alle disse er comparison sorts. Det kan vises at $\Omega(n \lg n)$ er en absolutt nedre grense for WC running time av en valgfri comparison sort på n input. Dermed er heapsort og mergesort asymptotisk optimale comparison sorts.

Counting sort skal vi senere se at bryter denne nedre grensen, med WC kjøretid $\Theta(k+n)$.

Radix sort kan sortere n tall på $\Theta(d(n+k))$ tid, der d er antall siffer, og k er antall verdier hvert siffer kan ta. Når d er konstant og k er $O(n)$ sorterer denne i lineær tid.

Bucket sort krever informasjon om den probalilistiske distribusjonen til inputtallene. Den kan sortere n reelle tall som er uniformt distribuert i det halv-åpne intervallet $[0,1)$ i AC $O(n)$ tid.

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Insertion sort

INSERTION SORT

$\Theta(n^2)$ men meget effektiv på små input ($n \leq 5$). Inplace!

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Merge Sort

MERGE SORT

Asymptotisk optimal, n lg n, men MERGE er ikke inplace.

MERGE(A, p, q, r) sammenligner det øverste kortet i to ferdig sorterte bunker ($p..q$, $q+1..r$) og legger det laveste av de to bakerst i den ferdig sorterte bunken ($A[i]$).

```
MERGE-SORT(A, p, r):
if p < r:
    q = ⌊(p + r) / 2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

Algoritmen kaller seg selv rekursivt helt til $p \geq r$ og datasettet består av høyst ett element (basecase). Fra basecase kalles MERGE på stadig større sorterte delmengder til til slutt A er sortert.

Heapsort

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       exchange A[1] with A[i]
4       A.heap-size = A.heap-size − 1
5       MAX-HEAPIFY(A, 1)
```

Heapsort bruker $O(n \lg n)$ tid siden BUILD-MAX-HEAP tar $O(n)$ tid og hvert av de n - 1 kallene til MAX-HEAPIFY bruker $O(\lg n)$ tid.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Kjøretid: $O(\lg n)$.

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

Kjøretid: $O(\lg n)$.

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Kjøretid: $O(\lg n)$. Samlet kjøretid: $O(\lg n)$.

Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

For å sortere hele lista A, kalles QUICKSORT(A, 1, $A.length$).

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Kjøretida til PARTITION er $\Theta(n)$, der $n = r - p + 1$.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Radix sort

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

Bruker for eksempel COUNTING SORT som en stabil (*men ikke inplace!*) sorteringsalgoritme.

Kjøretiden til Radix Sort er $\Theta(d(n+k))$, som er $\Theta(n)$, forutsatt at n er det dominerende leddet (og ikke k..).

Bucket sort

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Average-case kjøretid for bucket sort er $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Selv om vi ikke henter input fra en uniform distribusjon, kan bucket sort framdeles sortere i lineær tid. Så lenge input har den egenskapen at summen av kvadratene til bøttestørrelsene er lineær mhp. totalt antall elementer, gir ligningen over at bucket sort vil kjøre i lineær tid.

Selection

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

,

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

RANDOMIZED-SELECT *returnerer det i'te minste elementet i lista* $A[p..r]$.

Worst-case kjøretid for RANDOMIZED-SELECT er $\Theta(n^2)$, selv for å finne minimum, dersom vi er så uheldige å alltid partisjonere rundt det største elementet. Forventet kjøretid er imidlertid lineær, og siden input blir stokket i tilfeldig rekkefølge har vi ingen type input som sannsynliggjør worst-case kjøretid.

9.3 Selection in worst-case linear time

Algoritmen SELECT finner et gitt element med $O(n)$ som worst-case kjøretid. Den partisjonerer i likhet med RANDOMIZED-SELECT input-arrayet rekursivt, men *garanterer* en god split mellom partisjonene. SELECT bruker den samme partisjoneringsalgoritmen PARTITION, men modifiserer denne til å ta pivotelementet som en inputparameter.

SELECT-algoritmen og dens kjøretid er definert på side 243 i Cormen og fungerer som følger:

1. Deler elementene inn i $n/5$ grupper på 5 elementer hver.
2. Finner medianen i hver av gruppene ved å insertion-sorte elementene i hver gruppe og velge medianen.
3. Bruker SELECT rekursivt til å finne medianen av medianene funnet i 2.
4. Partisjonere input-arrayet rundt medianen av medianer ved å bruke modifisert PARTITION.
5. Dersom $i = k$, returner medianen av medianer. Hvis ikke, bruk SELECT rekursivt.

SEARCH(S, k)
: En query som returnerer en peker x til et element i S slik at $x.key = k$, eller NIL dersom k ikke finnes i S.

INSERT(S, x)
: En modifying operation som setter inn elementet pekt til av x.

DELETE(S, x)
: En modifying operation som fjerner elementet pekt til av x fra S.

MINIMUM(S)
: En query på en fullstendig ordnet mengde S som returnerer en peker til det elementet i S som har den minste nøkkelen.

MAXIMUM(S)
: En query på en fullstendig ordnet mengde S som returnerer en peker til det elementet i S som har den største nøkkelen.

SUCCESSOR(S, x)
: En query som returnerer en peker til det neste større elementet i S, eller NIL dersom x er det største elementet.

PREDECESSOR(S, x)
: En query som returnerer en peker til det neste mindre elementet i S, eller NIL dersom x er det minste elementet.

Et kall til MINIMUM etterfulgt av n - 1 kall til SUCCESSOR sorterer elementene i stigende rekkefølge.

Chapter 10 Elementary Data Structures

Stacks

INSERT-operasjonen på en stack kalles ofte PUSH. DELETE kalles POP. Illustreres ved en stabel med tallerkener på en restaurant.

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Operasjonene tar $O(1)$ tid.

Queues

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

10.2 Linked lists

En ***lenket liste*** er en datastruktur der objektene er sortert i en linær rekkefølge. Denne rekkefølgen er imidlertid ikke bestemt av indeksen, men av pekere som ligger i hvert objekt. Alle operasjonene nevnt over støttes av lenkede lister.

Chapter 11 Hash Tables

En hash table er en effektiv datastruktur for å implementere dictionaries. Selv om worst case kjøretid er like dårlig som en lenket liste ($\Theta(n)$), er forventet kjøretid ekstremt bra ($\mathrm{O}(1)$).

Hash tables er å foretrekke når antall nøkler *faktisk* lagret er få sammenlignet med antall mulige nøkler. Tabellindeksen blir *beregnet* ut fra nøkkelen vha. hashfunksjoner, i stedet for at nøkkelverdien blir brukt direkte.

11.1 Direct-address tables

Direkte adressering er en enkel teknikk som fungerer bra når antall mulige nøkler er rimelig lavt. Vi antar unike nøkler.

Vi bruker en ***direct-address table*** for å representere det dynamiske settet $T[0..m-1]$, der hver posisjon, eller ***slot***, korresponderer med en nøkkel i U.

11.2 Hash tables

Dersom universet U er stort kan det å lagre en tabell T av størrelse $|U|$ være upraktisk eller umulig. Settet av nøkler faktisk lagret, K, kan være så lite relativt til U at mye plass brukes til ingen nytte. Når K er mye mindre enn U bruker en hash table mye mindre lagringsplass enn en direct-address table. Vi kan redusere plassbehovet til $\Theta(|K|)$ uten å miste muligheten til å søke etter et element i $\mathrm{O}(1)$ tid. Imidlertid gjelder denne øvre grensa kun for *average case*, og ikke for *worst case kjøretid*, som for direkte adressering.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == k
5            return j
6       i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Chapter 12 Binary Search Trees

Et søketre støtter mange dynamiske operasjoner, inkludert SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, og DELETE. Altså kan vi se på set søketre både som en dictionary og en prioritetskø.

Basisoperasjoner på et binært søketre tar en mengde tid proposjonalt med høyden på treet, altså $\Theta(\lg n)$ worst-case tid.

Chapter 15 Dynamic Programming

Dynamisk programmering løser problemer ved å kombinere løsningene til delproblemene, likt som med divide-and-conquer-løsninger. Dynamisk programmering tar over for divide-and-conquer-fremgangsmåten når delproblemene overlapper, altså når delproblemene deler delproblemer. En dynamisk algoritme løser hvert del-delproblem kun én gang, og lagrer svaret i en tabell. Slik sparer den seg selv for arbeid.

Vi bruker dynamisk programmering i *optimeringsproblemer*. Slike problemer kan ha mange mulige løsninger, og vi ønsker å finne en optimal løsning. Dynamisk probrammering følger fire steg:

1. Karakteriser strukturen til en optimal løsning.
2. Definer verdien til en optimal løsning rekursivt.
3. Beregn verdien til en optimal løsning, vanligvis bottom-up.
4. Konstruer en optimal løsning fra den beregnede informasjonen.

Steg 1-3 utgjør grunnlaget for en dynamisk-programmering-løsning til et problem.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = −∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

Denne metoden definerer at et problem av størrelse *i* er *mindre* enn et problem av størrelse *j* dersom $i < j$.

Begge framgangsmåtene gir kjøretid $\Theta(n^2)$. Bottom-up på grunn av sin doble **for**-løkke, den memoiserte har samme kjøretid, men av grunner det kan være vanskeligere å se (**for**-løkken kjører n ganger og gir en aritmetisk rekke med $\Theta(n^2)$ iterasjoner).

15.2 Matrix-chain multiplication

Nok et eksempel på dynamisk programmering. For å matrisemultiplisere en kjede med fire matriser $\langle A_1, A_2, A_3, A_4 \rangle$ kan vi sette parantesene på fem forskjellige måter:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

Hvilken vi velger kan ha stor innvirkning på kostnaden ved multiplikasjonen.

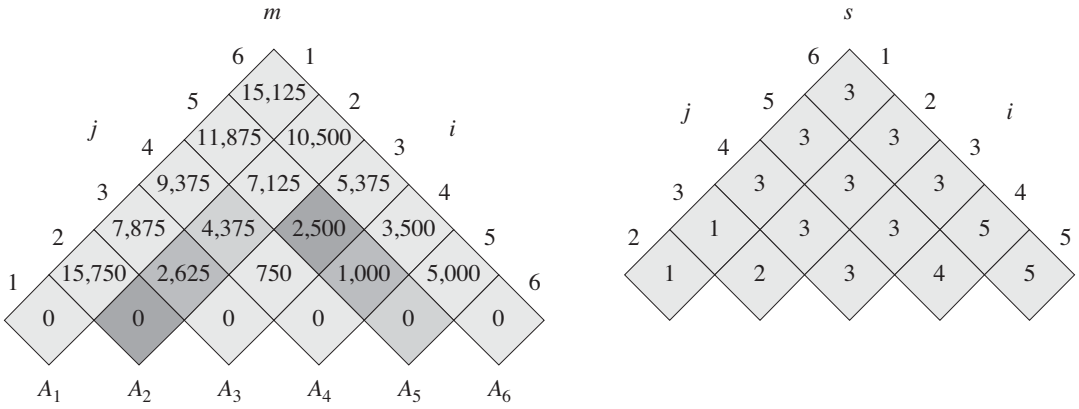
```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

Applying dynamic programming

For å bruke dynamisk programmering for å bestemme den optimale parantesstrukturen følger vi disse fire stegene:

1. Karakteriser strukturen til en optimal løsning.
2. Definer verdien til en optimal løsning rekursivt.
3. Beregn verdien til en optimal løsning.
4. Konstruer en optimal løsning fra den beregnede informasjonen.

Disse fire stegene for dette eksemplet er gjennomgått i Cormen side 373-377.



Tabellene beregnet av MATRIX-CHAIN-ORDER *for* $n = 6$.

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

For å konstruere den endelige optimale løsningen kaller vi følgende algoritme:

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

I eksemplet over ville et slikt kall printet $((A_1(A_2A_3))((A_4A_5)A_6))$.

15.3 Elements of dynamic programming

Når skal vi bruke dynamisk programmering? Vi skal se på to nøkkelingredienser som et optimeringsproblem må ha for at det skal være noen vits: Optimal substruktur og overlappende delproblemer.

Optimal substructure

Et problem har ***optimal substruktur*** dersom en optimal løsning til problemet inneholder optimale løsninger til delproblemene. Følgende mønster går igjen når vi leter etter en optimal substruktur:

1. Vi viser at løsningen til problemet består av å ta et valg, for eksempel det første kuttet i et stålrør. Ved å ta dette valget får vi flere delproblemer som må løses.
2. Vi antar at for et gitt problem blir vi gitt valget som gir optimal løsning.
3. Gitt dette valget finner vi ut hvilke delproblemer som melder seg og hvordan vi best kan karakterisere denne mengden av delproblemer.
4. Vi viser at løsningene til delproblemene som må brukes i en optimal løsning på hovedproblemet i seg selv er optimale ved bruk av "klipp-og-lim"-teknikken. Vi antar altså at løsningen av hvert delproblem ikke er optimal og viser at dette gir en kontradiksjon. Ved å "klippe ut" den ikke-optimale løsningen og "lime inn" den optimale viser vi at dette gir en bedre løsning til hovedproblemet. Altså viser vi ved kontradiksjon at vi hadde en optimal løsning i utgangspunktet.

For å beskrive mengden delproblemer er det en god regel å prøve å holde dette så enkelt som mulig, og heller utvide det dersom det blir nødvendig.

Optimal substruktur varierer på to måter:

1. hvor mange delproblemer en optimal løsning av hovedproblemet bruker, og
2. hvor mange valg vi har når vi skal bestemme hvilke delproblemer vi bruker i en optimal løsning.

Kjøretida til en algoritme i dynamisk programmering avhenger av et produkt av to faktorer: Antall delproblemer og hvor mange valg vi har i hvert delproblem. I rørkuttingen hadde vi $\Theta(n)$ delproblemer, og max n valg i hvert, altså fikk vi kjøretid $O(n^2)$.

Overlappende delproblemer

For at det skal være snakk om dynamisk programmering krever vi at en rekursivt algoritme for problemet løser de samme delproblemene om og om og om og om igjen.

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
                + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

Chapter 16 Greedy Algorithms

Man må være uhyre forsiktig hvis man vil konstruere en grådig algoritme som løsning på et problem. Det kan være relativt lett å finne en slik en, men uhyre vanskelig å bevise at den faktisk gir optimal løsning for alle input. I noen tilfeller kan vi også finne grådige algoritmer som ikke fungerer for alle input, men som vi tror gjør det, fordi de fungerer for visse input (de vi trenger)

For å designe grådige algoritmer går vi gjennom følgende tre steg:

1. Gjør om optimeringsproblemet til et der vi må gjøre et valg, og står igjen med kun ett delproblem.
2. Bevis at det alltid finnes en optimal løsning til hovedproblemet som tar det grådige valget, slik at det alltid er trygt å velge det grådige valget.
3. Demonstrer den optimale delstrukturen ved å vise at etter vi har tatt det grådige valget står vi igjen med et delproblem som innehar den egenskapen at dersom vi kombinerer en optimal løsning til delproblemet med det grådige valget vi tok, får vi en optimal løsning på det originale problemet.

Vi trenger altså en ***greedy-choice property*** og en ***optimal delstruktur***.

22.2 Breadth-first search

Prims og Dijkstras algoritmer for MST og shortest path bygger på ***BFS***.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Kjøretid: $\Theta(V + E)$.

Shortest paths

BFS kan brukes til å finne korteste veier. Vi definerer en ***korteste vei*** $\delta(s,v)$ fra s til v som det minste antall kanter i enhver vei fra s til v. Dersom ingen veier finnes er $\delta(s,v) = \infty$.

22.3 Depth-first search

Dybde-først-søk (DFS) utforsker kanter ut fra den mest nylig oppdagede noden v som fortsatt har uoppdagede kanter ut fra seg. Deretter backtracker søket for å utforske kantene som går ut fra noden som oppdaget v.

Kjøretid: $\Theta(V + E)$.

Classification of edges

Vi har fire typer kanter i en DFS-skog G_π produsert av DFS kjørt på G:

1. ***Tree edges*** er kanter i DFS-skogen G_π.
2. ***Back edges*** er de kantene som fobinder en node u til en forgjenger v i et DFS-tre. Self-loops ses på som back edges.
3. ***Forward edges*** er kanter som ikke er tree edges og som forbinder en node u til en etterkommer v i et DFS-tre.

4. ***Cross edges*** er alle andre kanter. Disse kan gå mellom kanter i samme DFS-tre, så lenge den ene noden ikke er etterkommer av den andre, eller de kan gå mellom noder i forskjellige DFS-trær.

Når vi først oppdager en kant fra *u* til *v*, forteller fargen til node *v* oss noe om kanten:

- WHITE indikerer en tree edge,
- GRAY indikerer en back edge, og
- BLACK indikerer en forward *eller* cross edge.

22.4 Topological sort

Vi kan bruke DFS til topologisk sortering av en directed acyclic graph (dag). En ***topologisk sortering*** av en dag $G = (V, E)$ er en lineær ordning av alle nodene slik at dersom det finnes en kant fra *u* til *v*, forekommer *u* før *v* i ordningen. Dette er ikke oppnåelig dersom grafen har sykler.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

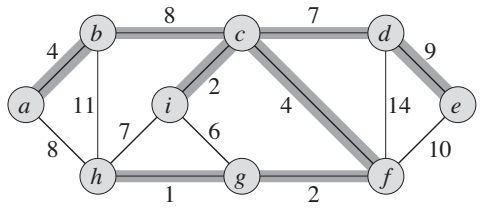
Kjøretid: $\Theta(V + E)$.

Chapter 23 Minimum Spanning Trees

For å koble sammen *n* noder på et kretskort kan vi bruke et oppsett med *n* - 1 kabler, der hver kabel kobler sammen to noder. Det optimale er å bruke så lite kabel som mulig. Vi kan se på dette problemet som en sammenkoblet, urettet graf $G = (V, E)$, der V er mengden noder, E er mengden mulige sammenkoblinger og $w(u, v)$ er mengden kabel som trengs for å sammenkoble to noder. Vi ønsker å finne et asyklisk subsett $T \subseteq E$ som sammenkobler alle nodene og hvis totale vekt

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

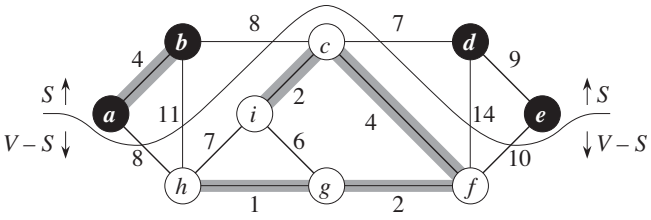
er minimert. Siden T er asyklisk og kobler sammen alle nodene, må det være et tre, som vi kaller et ***spenntre***, siden det spenner ut grafen *G*. Vi kaller problemet over et ***minimalt spenntre-problem***. Figuren på neste side viser et eksempel på en sammenkoblet graf og et minimalt spenntre.



Et minimalt spenntre for en sammenkoblet graf.

Theorem 23.1

La $G = (V, E)$ være en sammenkoblet, urettet, vektet graf og la A være en mengde kanter inkludert i et eller annet MST for G. La $(S, V - S)$ være et kutt som respekterer A, og la (u, v) være en lett kant som kutter $(S, V - S)$. Da er kanten (u, v) en trygg kant for A.



En måte å se et kutt av grafen på forrige figur på. De svarte nodene er i settet S, og de hvite er i settet V - S. (d, c) er den unike lette kanten.

Proof Let T be a minimum spanning tree that includes A, and assume that T does not contain the light edge (u, v), since if it does, we are done. We shall construct another minimum spanning tree T' that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that (u, v) is a safe edge for A.

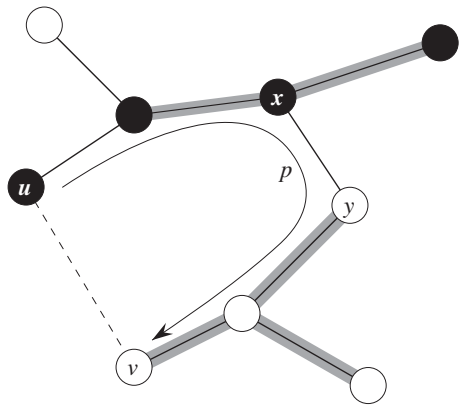
The edge (u, v) forms a cycle with the edges on the simple path p from u to v in T, as Figure 23.3 illustrates. Since u and v are on opposite sides of the cut $(S, V - S)$, at least one edge in T lies on the simple path p and also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique simple path from u to v in T, removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

We next show that T' is a minimum spanning tree. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \leq w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) . \end{aligned}$$

But T is a minimum spanning tree, so that $w(T) \leq w(T')$; thus, T' must be a minimum spanning tree also.

It remains to show that (u, v) is actually a safe edge for A. We have $A \subseteq T'$, since $A \subseteq T$ and $(x, y) \notin A$; thus, $A \cup \{(u, v)\} \subseteq T'$. Consequently, since T' is a minimum spanning tree, (u, v) is safe for A. ■



Illustrasjon av beviset på teoremet over.

Corollary 23.2

Den lette kanten som kobler sammen et MST for G med en annen samenkoblet komponent er trygg for A. Dette er fordi kuttet mellom MST og resten blir respektert av $(V_C, V - V_C)$. Da kanten det er snakk om er den lette kanten for dette kuttet er denne trygg.

23.2 The (MST-)algorithms of Kruskal and Prim

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Kjøretid: $\mathsf{O}(E \lg V)$.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Kjøretid: $\mathsf{O}(E + E \lg V)$.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Det å ***relaxe*** en kant består i teste hvorvidt vi kan forbedre den korteste veien til v ved å gå gjennom u, og i tilfelle oppdatere $v.d$ og $v.\pi$.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Alle korteste vei-algoritmene vi ser på her bruker denne teknikken, dog på foskjellige kanter til forskjellige tider.

23.1 The Bellman-Ford algorithm

Bellman-Ford returnerer korteste vei og dens vekter, eller FALSE, dersom grafen inneholder negative sykler.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Bellman-Ford-algoritmen kjører på $O(VE)$ tid. I Cormen side 651-653 finnes en del fine lemmaer og korollarer, for den som er sugen på mer av det.

24.2 Single-source shortest paths in directed acyclic graphs

Dersom vi relaxer alle kantene i en vektet dag i topologisk rekkefølge finner vi korteste vei på $\Theta(V + E)$ tid.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

24.3 Dijkstra's algorithm

Dijsktras algoritme kjører raskere enn Bellman-Ford.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Kjøretida avhenger av så mye rart, men det virker som vi ender på rundt en $O(E \lg V)$ for *sparse* grafer.

Chapter 25 All-Pairs Shortest Paths

25.2 The Floyd-Warshall algorithm

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Kjøretiden til Floyd-Warshall bestemmes av den tre-doble **for**-løkka fra linje 3-7 som gir kjøretid $\Theta(n^3)$. Koden er tight uten fancy datastrukturer, og har dermed kun en liten konstant. Dermed er Floyd-Warshall-algoritmen ganske praktisk selv for moderat store inputs.

Transitiv lukking av Floyd-Warshall:

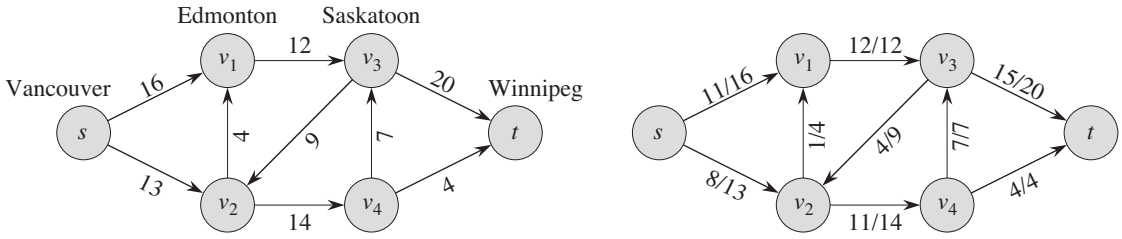
```
TRANSITIVE-CLOSURE(G)
 1  n = |G.V|
 2  let T^(0) = (t_ij^(0)) be a new n × n matrix
 3  for i = 1 to n
 4      for j = 1 to n
 5          if i == j or (i, j) ∈ G.E
 6              t_ij^(0) = 1
 7          else t_ij^(0) = 0
 8  for k = 1 to n
 9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10      for i = 1 to n
11          for j = 1 to n
12              t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
13  return T^(n)
```

Chapter 26 Maximum Flow

26.1 Flow networks

Flow networks and flows

Et ***flytnettverk*** $G = (V, E)$ er en rettet graf der hver kant har en ikkenegativ ***kapasitet*** $c(u,v) \geq 0$. Vi har to særegne noder i nettverket, en ***kilde*** s og et ***sluk*** t.



Et flytnettverk for et trucking problem. Grafen til venstre viser kun kapasitet, mens grafen til høyre viser faktisk flyt for hver kant.

I et ***max-flyt-problem*** er vi gitt et flytnettverk G med s og t og ønsker å finne den største mulige flyten.

26.2 The Ford-Fulkerson method

En metode for å løse max-flyt-problemet. Metoden øker verdien av flyten iterativt.

FORD-FULKERSON-METHOD(G, s, t)

1 initialize flow f to 0
2 **while** there exists an augmenting path p in the residual network G_f
3 augment flow f along p
4 **return** f

Residual networks

Gitt et nettverk G og en flyt f består residualnettverket G_f av kanter med kapasiteter som kan representere hvordan vi kan endre flyten gjennom kanter i G. Gitt en s og t og et nodepar u, v i V har vi ***residualkapasiteten*** $c_f(u,v)$ gitt ved

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v & \text{hvis } (u,v) \in E, \\ f(v,u) & \text{hvis } (v,u) \in E, \\ 0 & \text{ellers.} \end{cases}$$

Edmonds-Karp algorithm

Forbedring av FORD-FULKERSON som kjører i $O(VE^2)$ tid. Trikset med Edmonds-Karp er å finne stien p vha. BFS. Altså finner vi den *korteste* veien fra s til t i residual-nettverket, der vi ser bort fra kantvekter. Vi har at denne korteste-vei avstanden vil være monotont økende for hver flytendring:

Det kan vises at Edmonds-Karp får en øvre grense for kjøretiden:

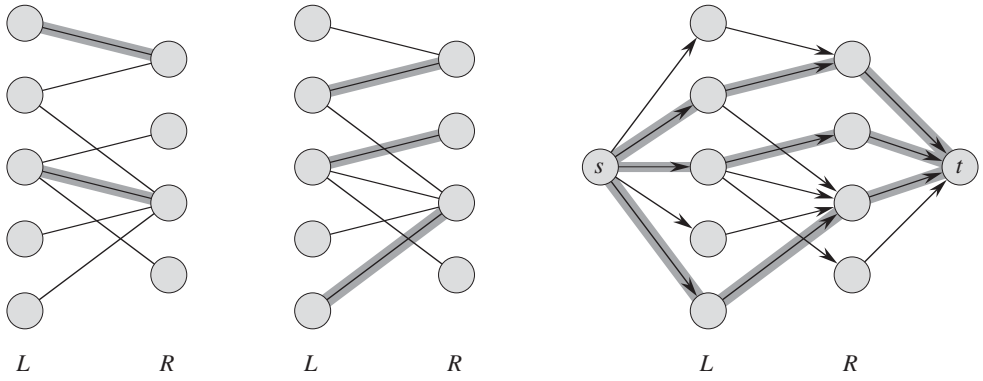
> ***Theorem 26.8***
> If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

26.3 Maximum pipartite matching

Vi skal se at problemer som tilsynelatende ikke har noe særlig til felles med flytnettverk kan reduseres til max-flyt-problemer. Et slikt problem er å finne maksimal matching in en bipartitt graf. For å løse dette problemet skal vi bruke Ford-Fulkerson.

The maximum-bipartite-matching problem

I en urettet graf G er en ***matching*** et subsett av kantene M som er slik at alle nodene v i G er representert i M med maksimalt én av sine kanter. En ***maximum matching*** er en matching av maksimal kardinalitet, altså at for en hver matching M' har vi $|M| \geq |M'|$.



En bipartitt graf G og matching med kardinalitet 2, 3 og et flytnettverk med max flyt.

Vi ser en sammenheng mellom matching i G og et flytnettverk G'. Dette er formulert i følgende lemma:

> ***Lemma 26.9***
> Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$.

Følgende teorem og korollar stadfester sammenhengen mellom problemet og finne maksimal matching i en bipartitt graf og maxflytproblemet:

> ***Theorem 26.10 (Integrality theorem)***
> If the capacity function c takes on only integral values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

> ***Corollary 26.11***
> The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in its corresponding flow network G'.

Altså kan vi finne maksimal matching M i en bipartitt graf ved å generere flytnettverket G', kjøre Ford-Fulkerson, og få en maksimal matching M fra heltallsverdien an maxflyten f.

Vi kan finne M i en bipartitt graf i $O(VE') = O(VE)$ tid (siden $|E'| = \Theta(E)$.

VII Selected Topics

Chapter 27 Multithreaded Algorithms

27.1 The basics of dynamic multithreasing

Performance measures

Vi har ***work law***:

$$T_P \geq \frac{T_1}{P}$$

Vi har ***span law***:

$$T_P \geq T_\infty$$

Vi definerer ***speedup*** til å være T_1/T_P, og denne sier noe om hvor mange ganger raskere beregningen kjører på P prosessorer enn på 1 prosessor. Speedup'en er begrenset av *work law* til å være i høyden P fordi $T_1/T_P \leq P$.

Vi støter ofte på ***lineær speedup***, $T_1/T_P = \Theta(P)$. Når $T_1/T_P = P$ har vi ***perfekt lineær speedup***.

Vi sier at T_1/T_∞ betegner ***parallelliteten*** til den multitrådede beregningen. Denne betegner maksimalt oppnåelig speedup for et gitt antall prosessorer, men den gir oss også en grense for sannsynligheten for å oppnå perfekt lineær speedup. Altså, dersom vi har flere prosessorer enn parallelliteten, er det ikke mulig å oppnå perfekt lineær speedup for beregningen.

Fra *work law* har vi at den beste oppnåelige kjøretida er $T_P = T_1/P$, og fra *span law* har vi at den beste oppnåelige kjøretida er $T_P = T_\infty$.

34.5 NP-complete problems

34.5.1 The clique problem

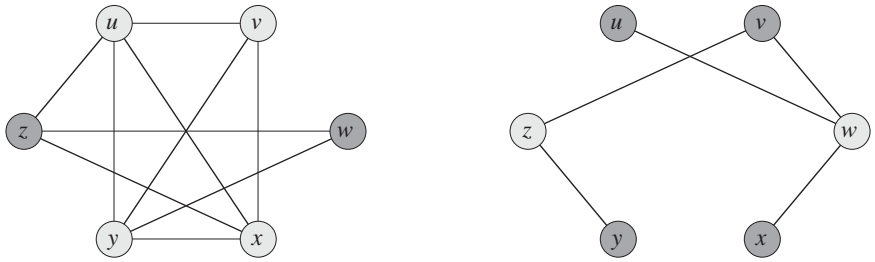
En ***clique*** i en urettet graf G er en komplett subgraf av G. Det vi kaller ***clique problem*** kan vi gjøre om til et decision problem ved å spørre om hvorvidt det finnes en cilque av størrelse k i grafen. Clique-problemet er NP-komplett:

> ***Theorem 34.11***
> The clique problem is NP-complete.

34.5.2 The vertex-cover problem

Et ***vertex cover*** til en urettet graf G er et subsett av nodene V' som er slik at dersom en kant går fra u til v i G så er enten u i V', u i V' eller begge er det.

Det kan vises at CLIQUE kan reduseres til VERTEX-COVER. Med andre ord kan vi vise at vertex-cover-problemet også er NP-komplett:



Reduksjon av CLIQUE *til* VERTEX-COVER.

Flere NP-komplette problemer inkluderer, men er ikke begrenset til:

- The hamiltonian-cycle problem (34.5.3)
- The traveling-salesman problem (34.5.4)
- The subset-sum problem (34.5.5)