Informasjon

Du måtte klare 8 av 15 oppgaver for å få øvingen godkjent.

Finner du feil, mangler eller forbedringer, ta gjerne kontakt!

Oppgave 2

«*Rangen u.rank for en node u er en øvre grense for høyden til u.*» er riktig.

Kommentar:
At rangen $u.rank$ for en node u er en øvre grense for høyden til u står på side 571 i læreboka.

At rangen $u.rank$ for en node u ikke er nøyaktig høyden til u kan man se ved at hvis man kjører FIND-SET på alle etterkommernodene til u vil høyden på u bli 1 (om u har etterkommernoder), men *rank*-attributtet endres ikke.

FIND-SET(x) finner riktig representant også uten stikomprimeringsheuristikken. Stikomprimeringsheuristikken gjør kun at man ikke må traversere samme sti opp til rotnoden flere ganger, siden den oppdaterer nodene på veien.

Etter FIND-SET(x) kan noder i treet som x tilhører ha en annen forelder enn $x.p$, siden det kun er nodene på stien opp til rotnoden som oppdateres, og ikke alle nodene i treet.

Oppgave 3

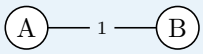
«*Alltid kun ett*» er riktig.

Kommentar:
Som bevis, anta det motsatte, nemlig at det finnes to ulike minimale spenntrær T_1 og T_2. La $\{e_1, e_2, \dots e_k\}$ og $\{e'_1, e'_2, \dots e'_k\}$ være kantene i henholdsvis T_1 og T_2 sortert i stigende rekkefølge etter kantvekt. La videre j være det laveste tallet der $e_j \neq e'_j$. Siden T_1 og T_2 er ulike, vil det finnes en slik j, og videre, siden alle kantvektene er unike må $w(e_j) \neq w(e'_j)$. Anta uten tap av generalitet at $w(e_j) < w(e'_j)$. Sett så inn e_j i T_2. Dette vil føre til en sykel. La e være kanten i sykelen med høyest vekt. Om $e = e_j$ vil dette bety at at sykelen består av e_j samt kanter e'_i der $i < j$, men siden vi har antatt at $e_i = e'_i$ for $i < j$, må denne sykelen også være i T_1, noe som ikke går siden T_1 er et tre. Derfor er $e \neq e_j$. Siden kantvektene er unike, betyr dette at $w(e) > w(e_j)$, og dermed kan vi få et spenntre med lavere vekt enn T_2 ved å bytte ut e med e_j, noe som bryter med at T_2 er et minimalt spenntre. Ergo kan det kun finnes ett minimalt spenntre i dette tilfellet.

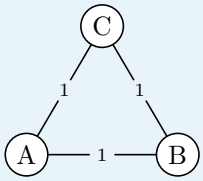
Oppgave 4

«*Noen ganger kun ett, andre ganger flere*» er riktig.

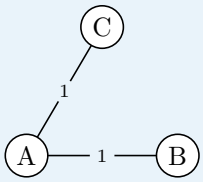
Kommentar:
Grafen

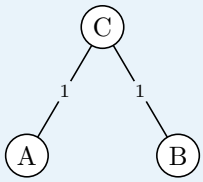


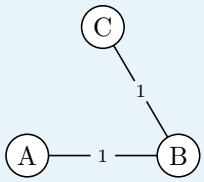
har kun ett minimalt spenntre, nemlig grafen selv. Denne grafen derimot



har de minimale spenntrærne







Oppgave 5

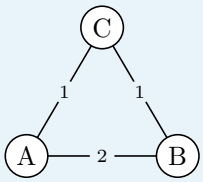
«*Ja, alltid*» er riktig.

Kommentar:
La T være et minimalt spenntre, der (u, v) ikke er med. Hvis vi setter inn (u, v) i T vil vi få en sykel. Siden alle de andre kantene i sykelen har en kantvekt høyere enn det kanten (u, v) har, kan vi bytte ut en av de kantene med (u, v) og få et tre med lavere kantvekt. Ergo er ikke T et minimalt spenntre, og (u, v) må være med i alle minimale spenntrær.

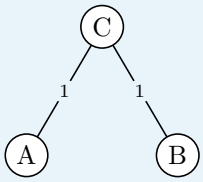
Oppgave 6

«*I noen tilfeller*» er riktig.

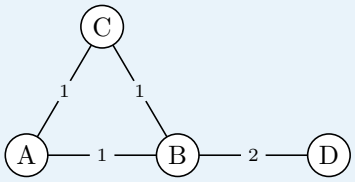
Kommentar:
I grafen



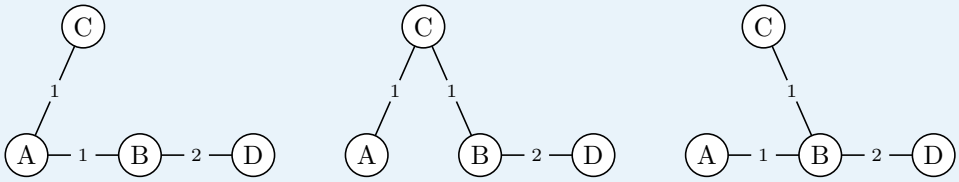
er ikke den tyngste kanten med i det minimale spenntreet, som er



I grafen



derimot er den tyngste kanten i alle de minimale spenntrærne



Oppgave 7

12 er riktig.

Kommentar:
Her kan man se at kantene med vekter 1, 2 og 3 danner et spenntre, og vil dermed bli valgt av Kruskals algoritme. Vekten blir da $1 \cdot 1 + 1 \cdot 2 + 3 \cdot 3 = 1 + 2 + 9 = 12$

Oppgave 8

```
class HigherEdSolver:
    # Disjoint-set forest
    class Set:
        def __init__(self, name):
            self.__p = self
            self.rank = 0
            self.name = name

        @property
        def p(self):
            if self.__p != self:
                self.__p = self.__p.p
```

```
            return self.__p

        @p.setter
        def p(self, value):
            self.__p = value.p

    def union(self, x, y):
        x = x.p
        y = y.p
        if x.rank > y.rank:
            y.p = x
        else:
            x.p = y
            y.rank += x.rank == y.rank

    def initialize(self, institutions):
        self.institutions = {i: HigherEdSolver.Set(i) for i in institutions}

    def parent_institution(self, institution):
        return self.institutions[institution].p.name

    def fuse(self, institution1, institution2, new_institution):
        institution1 = self.institutions[institution1]
        institution2 = self.institutions[institution2]
        self.union(institution1, institution2)
        institution1.p.name = new_institution
        self.institutions[new_institution] = institution1.p
```

Kommentar:
Poenget i denne oppgaven var å slå sammen institusjonene ved hjelp av skog-implementasjonen av disjunkte mengder.

Det var også mulig å implementere dette ved å bruke oppslagstabellene (*dictionary*) som allerede finnes i Python. Koden under er et eksempel på dette.

```
class HigherEdSolver:
    def initialize(self, institutions):
        self.parents = {i: i for i in institutions}

    def parent_institution(self, institution):
        parent = self.parents[institution]
        while parent != self.parents[parent]:
            self.parents[institution] = self.parents[parent]
            parent = self.parents[parent]
        return parent

    def fuse(self, institution1, institution2, new_institution):
        self.parents[new_institution] = new_institution
        self.parents[institution1] = new_institution
        self.parents[institution2] = new_institution
```

Oppgave 9

```
def hamming_distance(s1, s2):
    return sum(a != b for a, b in zip(s1, s2))


# En implementasjon av disjunkte mengder
class Set:
    def __init__(self):
        self.__p = self
        self.rank = 0

    @property
    def p(self):
        if self.__p != self:
            self.__p = self.__p.p
        return self.__p

    @p.setter
    def p(self, value):
        self.__p = value.p


    def union(x, y):
        x = x.p
        y = y.p
        if x.rank > y.rank:
            y.p = x
        else:
            x.p = y
            y.rank += x.rank == y.rank


def find_clusters(E, n, k):
    n_clusters = n
    # Lager en disjunkt mengde for hvert dyr
    sets = [Set() for _ in range(n)]

    # Sorterer kanten basert på vekt
    E.sort(key=lambda x: x[2])

    for n1, n2, w in E:
        # Stopper Kruskals når vi har k disjunkte mengder
        if n_clusters == k:
            break
        if sets[n1].p != sets[n2].p:
            union(sets[n1], sets[n2])
            n_clusters -= 1

    # Gjør om mengdene til lister med elementene i mengden
    clusters = {}
    for index, set in enumerate(sets):
        if set.p in clusters:
            clusters[set.p].append(index)
        else:
```

```
            clusters[set.p] = [index]

    return list(clusters.values())
```

Kommentar:
Koden over inneholder ikke funksjonen *find_animal_groups* da denne er uendret fra den utdelte koden. Den bruker Kruskals algoritme til å lage et minimalt spenntre og stopper når vi har utført UNION $n - k$ ganger. Dette vil alltid produsere k klynger, da UNION setter sammen to disjunkte mengder og dermed reduserer antall gjenværende disjunkte mengder med en. Siden vi starter med n disjunkte menger vil vi ende med $n - (n - k) = k$ disjunkte mengder.

Her er det ikke nødvendig å anvende Kruskals algoritme. Man kunne like så godt anvendt andre algoritmer for konstruksjon av minimale spenntrær.

Oppgave 10

«*Alltid et maksimalt spenntre*» er riktig.

Kommentar:
Første observasjon vi kan gjøre er at om vi skifter fortegn på alle vektene i grafen vil det som tidligere var et minimalt spenntre nå bli et maksimalt spenntre. Dette er enkelt å vise ved å anta at T er et minimalt spenntre. Hvis vi så snur fortegnet på vektene $w'(u, v) = -w(u, v)$ og så finner at $w'(\mathrm{T}') > w'(\mathrm{T})$ for et tre T′, vil dette bety at $-w(\mathrm{T}') > -w(\mathrm{T}) \implies w(\mathrm{T}') < w(\mathrm{T})$, noe som betyr at T ikke kan være et minimalt spenntre med de opprinnelige kantvektene.

Om vi i stedet for å bytte ut MST-PRIM med MST-PRIM′ hadde byttet ut $w(u, v)$ med $w'(u, v) = -w(u, v)$ ville Prims algoritme funnet et minimalt spenntre med de nye vektene, som ville være det samme som et maksimalt spenntre med de gamle kantvektene. Vi må nå vise at denne algoritmen gir samme resultat som MST-PRIM′ .

Her er MST-PRIM med $w(u, v)$ byttet ut med $-w(u, v)$:

MST-PRIM(G, w, r)
1 **for** each $u \in$ G.V
2 $\quad u.key = \infty$
3 $\quad u.\pi =$ NIL
4 $r.key = 0$
5 Q = G.V
6 **while** Q $\neq \emptyset$
7 $\quad u =$ EXTRACT-MIN(Q)
8 $\quad$ **for** each $v \in$ G.$adj[u]$
9 $\quad\quad$ **if** $v \in$ Q and $w(u, v) > -v.key$
10 $\quad\quad\quad v.\pi = u$
11 $\quad\quad\quad v.key = -w(u, v)$

Sammenlign dette med MST-PRIM′:

```
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-MAX(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)
```

Her kan vi se at den eneste forskjellen mellom de to algoritmen er at nøklene i prioritetskøen har motsatt fortegn og det at Q er en min-prioritetskø i MST-PRIM og en maks-prioritetskø i MST-PRIM′. Siden det å endre fortegn på nøklene i en kø er ekvivalent med å endre om køen er en maks- eller min-kø, så gir disse algoritmene samme resultat. Etter diskusjonen over gir MST-PRIM et maksimalt spenntre når vi har byttet ut $w(u, v)$ med $-w(u, v)$ og derfor må MST-PRIM′ også gi ut et maksimalt spenntre.

Oppgave 11

«*Alltid et maksimalt spenntre*» er riktig.

Kommentar:
Dette kommer også av at dersom vi skifter fortegn på alle vektene i grafen vil det som tidligere var et minimalt spenntre nå bli et maksimalt spenntre. Hvis vi skifter fortegn på vektene etter de er sortert synkende i den nye Kruskal-algoritmen, vil kantene være sortert i stigende rekkefølge. Da er vi garantert at Kruskal gir et minimalt spenntre. Hvis vi så snur fortegnet på vektene igjen vil vi dermed få et maksimalt spenntre. Siden det å snu fortegnet på vektene ikke endrer hvilke kanter Kruskal velger, siden algoritmen ikke ser på vektene etter kantene er sortert, vil algoritmen alltid gi et maksimalt spenntre også om vi ikke snur fortegnet på vektene.

Oppgave 12

```
MST-KRUSKAL'(G, w)
 1  A = ∅
 2  for each vertex v ∈ G.V
 3       MAKE-SET(v)
 4  cycle = FALSE
 5  sort the edges of G.E into nondecreasing order by weight w
 6  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
 7       if FIND-SET(u) ≠ FIND-SET(v)
 8            A = A ∪ {(u, v)}
 9            UNION(u, v)
10       else if not cycle
11            A = A ∪ {(u, v)}
12            cycle = TRUE
```

Kommentar:
Bevis for korrekthet:

La A' være en optimal løsning. A' må være sammenhengede. Hvis A' har mer enn én sykel kan vi fjerne en kant fra en av syklene og fortsatt ha både en sammenhengde graf og minst én sykel. Siden kantvektene er positive vil denne nye grafen ha lavere total kantvekt, noe som strider med at A' er en optimal løsning. Derfor må A' være en sammenhengende graf med nøyaktig én sykel.

La e være kanten i sykelen til A' som har høyest kantvekt (velg vilkårlig dersom flere har lik kantvekt). $A' - \{e\}$ må da være et minimalt spenntre. Anta det motsatte, at det finnes et minimalt spenntre T der $w(T) < w(A' - \{e\})$. La e' være en av kantene i sykelen i A' som ikke er i T. $T \cup \{e'\}$ blir da en sammenhengende graf med minst én sykel. Videre, siden $w(e') \leqslant w(e)$ og $w(T) < w(A' - \{e\})$, må $w(T \cup \{e'\}) = w(T) + w(e') < w(A') = w(A' - \{e\}) + w(e)$. Dette bryter med at A' er en optimal løsning.
La $\hat{A}$ være en graf laget av å ta et minimalt spenntre, kalt $\hat{T}$, og deretter satt inn kanten med lavest vekt som gjør grafen syklisk. La $\hat{e}$ være kanten som ble satt inn. Da kan vi vise at $w(\hat{A}) = w(A')$. Fjern den kanten med høyest vekt i sykelen i A' og kall de resulterende treet T'. La kanten som ble fjernet være e'. Etter diskusjonen ovenfor er T' et minimalt spenntre. I tillegg er $\hat{T}$ er et minimalt spenntre slik som vi har definert det, noe som betyr at $w(\hat{T}) = w(T')$.

Av kantene i sykelen i A' må det være minst én av dem som ikke er i $\hat{T}$, siden $\hat{T}$ er et tre. La e være den av disse kantene med lavest kantvekt. $w(\hat{e}) \leqslant w(e)$, siden $\hat{A}$ ble laget ved å legge inn kanten med lavest vekt som gjorde $\hat{T}$ syklisk. Videre må $w(e) \leqslant w(e')$ siden e' var kanten med høyest vekt i sykelen. Dette gir $w(\hat{e}) \leqslant w(e) \leqslant w(e') \implies w(\hat{e}) \leqslant w(e')$. Dette sammen med at $w(\hat{T}) = w(T')$ fører til at $w(\hat{A}) = w(\hat{T}) + w(\hat{e}) \leqslant w(A') = w(T') + w(e')$. Videre er $w(A') \leqslant w(\hat{A})$ siden A' er optimal, noe som betyr at $w(A') = w(\hat{A})$.

Det eneste som mangler nå er å vise at A som algorimen kommer med er dannet på samme måte som A'. Dette kan vi se ved at *cycle* delen av algoritmen ikke endrer hvilke kanter som legges til på linje 8, som er der alle kanter i det minimale spenntreet legges til. Den kanten med lavest vekt som lager en sykel vil bli lagt til på linje 11.

Oppgave 13

```
# Disjoint-set forest
class Set:
    def __init__(self, i):
        self.__p = self
        self.rank = 0
        self.i = i

    @property
    def p(self):
        if self.__p != self:
            self.__p = self.__p.p
        return self.__p

    @p.setter
    def p(self, value):
        self.__p = value.p


def union(x, y):
    x = x.p
    y = y.p
    if x.rank > y.rank:
        y.p = x
    else:
        x.p = y
        y.rank += x.rank == y.rank


# Returnerer True om den treffer på en sykel
def DFS(nodes, edges):
    closed_nodes = set()
    active_nodes = set()
    for node in nodes:
        if DFS_visit(node, edges, active_nodes, closed_nodes):
            return True
    return False


# Returnerer True om den treffer på en sykel
def DFS_visit(node, edges, active_nodes, closed_nodes):
    if node in closed_nodes:
        return False
    if node in active_nodes:
        return True
    active_nodes.add(node)
    for n in edges[node]:
        if DFS_visit(n, edges, active_nodes, closed_nodes):
            return True
    active_nodes.remove(node)
    closed_nodes.add(node)


def check(variables, constraints):
```

```
    variables_set = {i:Set(i) for i in variables}
    for a, comp, b in constraints:
        if comp == "=" and variables_set[a].p != variables_set[b].p:
            union(variables_set[a], variables_set[b])
    nodes = set([var.p.i for var in variables_set.values()])
    edges = {i: set() for i in nodes}
    for a, comp, b in constraints:
        if comp == "<":
            edges[variables_set[a].p.i].add(variables_set[b].p.i)
        elif comp == ">":
            edges[variables_set[b].p.i].add(variables_set[a].p.i)
    return not DFS(nodes, edges)
```

Kommentar:
Denne oppgaven var inspirert av oppgave 15 på eksamen 2019S.

Poenget her var å lage en graf. Først bruker vi skog-implementasjonen av disjunkte mengder til å slå sammen variabler som skal være like til samme komponent (man kan også gjøre dette med for eksempel DFS). Hvert slikt komponent er en node i grafen. Her setter vi inn en kant fra en komponent A til en komponent B (potensielt med A = B) hvis en av variablene i B skal være større enn en av variablene i A. Nå kan man se at man kan gi tallverdier til variablene som overholder begrensningene hvis og bare hvis grafen er asyklisk.

Bevis: Hvis det finnes tallverdier til variablene som overholder begrensingene, vil en sortering av komponenter etter tallverdier være en gyldig topologisk sortering. En kant (u, v) vil bety at u er før v i den topologiske sorteringen, siden kanten tilsvarer en restriksjon om at tallverdien til komponentet u må være lavere enn tallverdien til komponentet v. Tilsvarende, hvis grafen er asyklisk kan man topologisk sortere grafen og gi tallverdier til komponentene stigende i topologisk sortert rekkefølge.

Vi kan finne ut om en rettet graf er asyklisk eller ikke ved å kjøre DFS og se om vi treffer på noen bakoverkanter (Lemma 22.11 side 614 i læreboka).

Oppgave 14

```
from collections import deque


class Node:
    def __init__(self):
        self.dist = float('inf')
        self.set = None


# Disjoint-set forest
class Set:
    def __init__(self):
        self.__p = self
        self.rank = 0
```

```
    @property
    def p(self):
        if self.__p != self:
            self.__p = self.__p.p
        return self.__p

    @p.setter
    def p(self, value):
        self.__p = value.p


def union(x, y):
    x = x.p
    y = y.p
    if x.rank > y.rank:
        y.p = x
    else:
        x.p = y
        y.rank += x.rank == y.rank


def power_grid(m, n, substations):
    grid = [[Node() for i in range(n)] for j in range(m)]
    queue = deque()
    for station in substations:
        node = grid[station[0]][station[1]]
        node.dist = 0
        node.set = Set()
        queue.append(station)
    edges = []
    while len(queue):
        pos = queue.popleft()
        node = grid[pos[0]][pos[1]]
        for i, j in [(pos[0] + 1, pos[1]), (pos[0] - 1, pos[1]), (pos[0],
    pos[1] + 1), (pos[0], pos[1] - 1)]:
            if 0 <= i < m and 0 <= j < n:
                new_node = grid[i][j]
                if new_node.set is None:
                    new_node.set = node.set
                    new_node.dist = node.dist + 1
                    queue.append((i, j))
                elif new_node.set.p != node.set.p:
                    edges.append((node.set, new_node.set, node.dist +
    new_node.dist + 1))
    edges = sorted(edges, key=lambda x: x[2])
    tot_cost = 0
    for s1, s2, cost in edges:
        if s1.p != s2.p:
            union(s1, s2)
            tot_cost += cost
    return tot_cost
```

Kommentar:
Dette er en blanding av BFS og Kruskal, der vi kjører BFS fra alle nettstasjonene samtidig for å

finne korteste vei fra hvert kryss til nærmeste nettstasjon. Hver gang vi treffer på et kryss hvor nærmeste nettstasjon er en annen en den nærmeste i krysset vi kom fra, så legger vi til en kant i listen *edges* mellom de to nettstasjonene. Kanten går langs korteste vei fra nettstasjonene til hvert av kryssene pluss veistrekningen mellom kryssene. Når vi har alle kantene, så kjører vi vanlig Kruskal.

Det eneste som mangler å bevise for korrekthet er at det ikke er noen kanter mellom to nettstasjoner som Kruskal-algoritmen skulle valgt, men ikke gjorde siden kantene ikke er i *edges*. La $d(u, v)$ være distansen mellom u og v i grafen.

La (u, v) være en kant (av potensielt flere) som skulle ha vært i *edges* for at Kruskal-delen av algoritmen skulle vært korrekt. Siden (u, v) ikke ligger i listen, må det være andre nettstasjoner som var nærmere kryss på korteste vei mellom u og v. La a_1 være første slik nettstasjon u møter på på veien, og la x_1 være krysset der de møttes. Vi har $d(a_1, x_1) \leqslant d(v, x_1)$, siden søket fra a_1 kom til x_1 først, som fører til $w(u, a_1) \leqslant d(u, x_1) + d(a_1, x_1) \leqslant w(u, v) = d(u, x_1) + d(v, x_1)$ Siden søkene fra u og a_1 møtes i x_1, legges (u, a_1) i *edges*. Siden søket fra a_1 kom til x_1 først, er det også dette søket som søker videre fra x_1 langs korteste vei mellom u og v mot v. Når vi søker videre langs denne veien kommer vi til et kryss vi kaller x_2 som a_2 kom til først. Da legges (a_1, a_2) inn i *edges*. Vi har $d(a_2, x_2) \leqslant d(v, x_2)$ siden søket fra a_2 kom til x_2 først. Dette gir $w(a_1, a_2) \leqslant d(a_1, x_1) + d(x_1, x_2) + d(a_2, x_2) \leqslant w(u, v) = d(u, x_1) + d(x_1, x_2) + d(v, x_2)$ Slik fortsetter vi til vi kommer til en a_k som treffer på v. Kantene som legges inn i *edges* er dermed $\{(u, a_1), (a_1, a_2) \dots (a_{k-1}, a_k), (a_k, v)\}$. Vektene til disse kantene er lik eller lavere vekten til (u, v), noe som betyr at en Kruskal-algoritme kan sortere disse kantene foran kanten (u, v), som igjen betyr at vi uansett ville være garantert at u og v var i samme tre før vi hadde kommet til kanten når vi itererte gjennom *edges*. Dermed ville ikke Kruskal-algoritmen lagt til kanten (u, v) uansett. Ergo er algoritmen korrekt.

Hva blir kjøretiden? Å generere nodene tar $\Theta(mn)$ tid, siden det er mn noder. I søket besøkes hver node nøyaktig én gang, og det er maksimalt fire kanter ut fra en node. Å finne forelder i «Disjoint-Set»-datastrukturen tar her konstant tid, siden alle er sin egen forelder. Dette gir at BFS delen av algoritmen bruker $\Theta(mn)$ tid. Siden det legges til maksimalt én kant i *edges* for hver gang vi traverserer en kant i grafen vil det være $O(mn)$ kanter i *edges*. Å sortere kantene tar dermed $O(mn \lg(mn))$ tid. Å gå gjennom kantene i Kruskal-delen av algoritmen tar $O(mn\,\alpha(mn))$ tid. Dette gir en kjøretid på $\Theta(mn) + \Theta(mn) + O(mn \lg(mn)) + O(mn\,\alpha(mn)) = O(mn \lg(mn))$. Siden kantvektene er heltall fra 1 til mn er det også mulig å bruke Counting-Sort for å få en kjøretid på $O(mn\,\alpha(mn))$.

Det er flere måter å forbedre algoritmen med tanke på highscore. Man kan blant annet kjøre Kruskal-delen under BFS-delen, der man kjører Kruskal på kantene hver gang *node.dist* øker i nodene man tar ut av køen. Da kan man stoppe søket når alle nettstasjonene er i samme tre, og dermed spare en del søking. Dette problemet er ellers en spesialisering av «Rectilinear minimum spanning tree»-problemet hvor Google kan gi dere flere raske algoritmer.

Oppgave 15

«*Ja*» er riktig.

Kommentar:
Anta i det minimale spenntreet for G, kalt T, at kantene som spenner V_u er T_u, men at et minimalt

spenntre for V_u er T'_u med $w(T'_u) < w(T_u)$. Da ville $(T - T_u) \cup T'_u$ ha lavere vekt enn T, noe som bryter med at T er et minimalt spenntre. Samme argument med T_v.

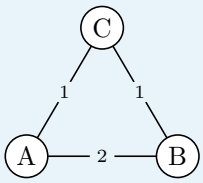
Oppgave 16

Se oppgaven over.

Oppgave 17

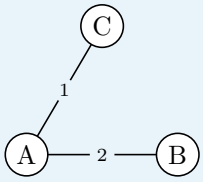
«*Nei*» er riktig.

Kommentar:
Et enkelt moteksempel er denne grafen:

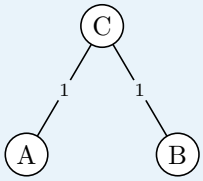


Hvis algoritmen lar $V_1 = \{A, B\}$ og $V_2 = \{C\}$, vil den ende opp med å måtte sette sammen disse to trærne:

noe den gjør ved å lage for eksempel dette treet:



noe som igjen ikke er det minimale spenntreet, nemlig:



Oppgave 18

For bevis for oppgavens ukorrekthet, se over.

Hva blir kjøretiden? Jeg antar her at nodene nummereres fra 1 til $|V|$ og at nodemengdene implementeres med balanserte binære søketrær. Siden nodemengden deles i to for hver rekursjon, vil rekursjonstreet ha en høyde på $\Theta(\lg V)$. Vi kan splitte en nodemengde A i to ved å traversere søketreet (som tar $\Theta(|A|)$ tid) og for hver ny node vi kommer til setter vi dem annenhver gang i binærtrærne B_1 og B_2. Hver innsetting tar $O(\lg |A|)$ tid og det er $|A|$ innsettinger, som gir en kjøretid på $O(|A| \lg |A|)$. Ved å se at summen av nodene i hvert nivå i rekursjonstreet er $O(V)$, siden en node kun sendes inn i én av de to undermengdene i rekursjonen, kan man se at splitting av nodemengder tar totalt $O(V \lg V)$ tid for hvert nivå i rekursjonstreet. Vi ser også at hver kant forekommer maksimalt to ganger på hvert nivå i rekursjonstreet, siden hver node opptrer maksimalt en gang i hvert nivå, og vi ser på en kant når vi itererer gjennom den i nabolistene. En kant er kun i nabolistene til de to nodene kanten binder sammen. Hver gang vi ser på en kant, sjekker vi om den ligger i den andre nodemengden, noe vi kan sjekke i $O(\lg V)$ tid ved å søke i søketreet. Dette gir et en kjøretid på $O(E \lg V)$ på å sjekke kanter for hvert nivå i rekursjonstreet. Totalt får vi dermed $O((V \lg V + E \lg V) \cdot \lg V) = O(E(\lg V)^2)$ om vi antar at $|V| = O(|E|)$.

Oppgave 19

For eksamensoppgavene, se løsningsforslaget til den gitte eksamenen.

For oppgaver i CLRS, så finnes det mange ressurser på nettet som har fullstendige eller nesten fullstendige løsningsforslag på alle oppgavene i boken. Det er ikke garantert at disse er 100% korrekte, men de kan gi en god indikasjon på om du har fått til oppgaven. Det er selvfølgelig også mulig å spørre studassene om hjelp med disse oppgavene.

Et eksempel på et ganske greit løsningsforslag på CLRS, laget av en tidligere doktorgradsstudent ved Rutgers, finnes her.