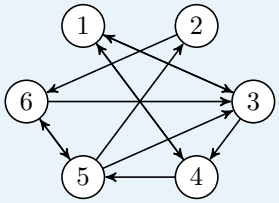
Informasjon

Du måtte klare 11 av 19 oppgaver for å få øvingen godkjent.

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

Oppgave 2



Kommentar:
Her må man sjekke om alle kantene i nabolistene er i grafen, og at det ikke er noen kanter som ikke skal være der.

Oppgave 3

	1	2	3	4	5	6
1	0	0	1	0	0	1
2	0	0	1	1	0	0
3	1	1	0	1	1	0
4	0	1	1	0	1	1
5	0	0	1	1	0	0
6	1	0	0	1	0	0

Kommentar:
Her må man sjekke om alle kantene i nabomatrisen er i grafen, og at det ikke er noen kanter som ikke skal være der.

Oppgave 4

Riktige alternativer:

1. $\langle 3, 4, 5, 6, 1, 2 \rangle$
2. $\langle 4, 3, 5, 1, 6, 2 \rangle$
3. $\langle 3, 4, 5, 1, 2, 6 \rangle$

Kommentar:
Her må vi sjekke at det ikke går noen kanter ut fra en node til en annen node tidligere i sorteringen.

Oppgave 5

Riktige alternativer:

1. $\langle 1, 2, 5, 3, 4, 6 \rangle$
2. $\langle 1, 3, 2, 5, 6, 4 \rangle$
3. $\langle 1, 5, 3, 2, 4, 6 \rangle$
4. $\langle 1, 2, 5, 3, 6, 4 \rangle$

Kommentar:
Her finnes det flere måter å tenke på. En måte er å se at node 1 besøkes først, siden det er startnoden. Når node 1 besøkes, så legges nodene $\{2, 3, 5\}$ i køen, men vi vet ikke i hvilken rekkefølge, siden vi ikke vet hvilken rekkefølge det er på nodene i nabolistene. Uansett betyr dette at et riktig alternativ må bestå av 1 og deretter en permutasjon av tallene 2, 3 og 5.

Hva er rekkefølgen på 4 og 6 til slutt? Hvis node 2 besøkes først av nodene $\{2, 3, 5\}$, legges både node 4 og 6 inn i køen når node 2 besøkes, og dermed kan vi ikke si noe om hvilken av nodene som legges inn først. Hvis 3 besøkes først, så legges node 6 inn i køen under besøket, mens node 4 ikke gjør det, så da må 6 besøkes før 4. Hvis 5 besøkes først, så må tilsvarende node 4 besøkes før node 6.

Oppgave 6

```
def compatibility_graph(donors, recipients, k):
    graph = [[] for donor in donors]
    for donor, adj_list in zip(donors, graph):
        for ind, recipient in enumerate(recipients):
            if sum([attr1 == attr2 for attr1, attr2 in zip(donor, recipient)
   ]) >= k:
                adj_list.append(ind)
    return graph
```

Oppgave 7

«BFS *bruker en kø og* DFS *bruker en stakk.*» er riktig.

Kommentar:
Hvilken prioritetskø som brukes kan man finne ved å se på hver av algoritmene, men det er logiske

grunner bak valget om å bruke en kø for BFS og en stakk for DFS. Disse grunnene er det viktig å være klar over og forstå.

I BFS skal vi utføre et bredde-først-søk. Det vil si at vi ønsker å besøke nodene i samme rekkefølge som vi ser dem. Dette passer veldig bra med en kø, siden vi kan sette inn nodene i køen etterhvert som vi ser dem og dermed hente ut nodene i samme rekkefølge.

I DFS skal vi heller utføre et dybde-først-søk. Det vil si at vi ønsker å besøke de nodene vi har sett sist først. Til dette passer en stakk veldig bra, siden vi kan legge til nodene i rekkefølgen vi har sett dem i, og hente ut nodene i motsatt rekkefølge av den vi har sett dem i.

Oppgave 8

«*Et dybde-først-søk av nodene som kan nås fra s.*» er riktig.

Kommentar:
Husk at i et dybde-først-søk ønsker vi å besøke nodene vi har sett sist først, mens i et bredde-først-søk ønsker vi å besøke nodene i samme rekkefølge som vi så dem. Når vi traverserer grafen i BFS bruker vi en kø til å holde styr på hvilke noder vi skal besøke videre. Om vi bytter ut denne køen med en stakk ender vi opp med at vi besøker noden vi så sist først, da disse ligger på toppen av stakken og dermed vil bli hentet ut først. Det vil si at vi får et dybde-først-søk. Siden BFS kun starter i en oppgitt startnode er det ikke sikkert at vi besøker alle nodene i grafen. Vi kan kun besøke de nodene som kan nås fra startnoden.

Oppgave 9

```
def resolve_and_install(package):
    if package.is_installed:
        return
    for dep in package.dependencies:
        resolve_and_install(dep)
    install(package)
```

Kommentar:
Her er det selvfølgelig også mulig å bruke «vanlig» topologisk sortering først, og deretter sende pakkene inn i *install* i riktig rekkefølge.

Oppgave 10

«$\Theta(V + E)$ *med nabolister og* $\Theta(V^2)$ *med nabomatrise.*» er riktig.

Kommentar:
Dette står på side 591 i læreboka.

Oppgave 11

«O(E) *med nabolister og* O(1) *med nabomatrise.*» er riktig.

Kommentar:
Hvis man skal sjekke om en kant (u, v) er i grafen, så må man i en naboliste-representasjon potensielt gå gjennom hele $Adj[u]$ for å sjekke om v ligger der. Siden lengden på $Adj[u]$ kan være alt fra $\Theta(1)$ til $\Theta(E)$ vil dette ta O(E) tid. I en nabomatrise-representasjon der nabomatrisen er A med elementer a_{ij} for $i, j \in G.V$, trenger man kun å sjekke om a_{uv} er 1. Dette kan gjøres i O(1) tid.

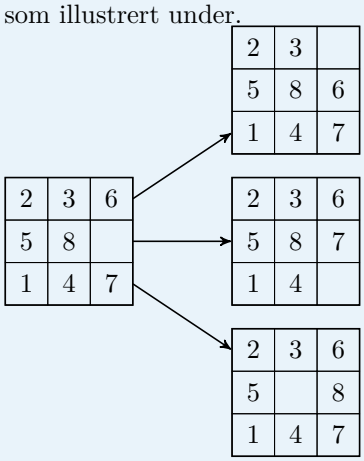
Oppgave 12

«*BFS*» er riktig.

Kommentar:
Her kan vi lage en graf, der hver tilstand brettet kan være i er en node. Med en tilstand menes en bestemt posisjonering av brikkene i spillet. For eksempel er

2	3	6
5	8	
1	4	7

en tilstand. Hver tilstand har en kant til alle tilstander man kan komme til ved å flytte én brikke, som illustrert under.



Målet er å ende opp i tilstanden:

1	2	3
4	5	6
7	8	

Siden det er én brikke som flyttes for hver kant, og vi ønsker å finne en løsning der vi må flytte færrest mulig brikker, må vi finne korteste vei fra starttilstanden til måltilstanden målt i antall kanter. Dette kan vi gjøre med BFS.

Oppgave 13

«*Topologisk sortering*» er riktig.

Kommentar:
Her kan man lage en graf, hvor hvert i listen fag er en node. Videre går det en kant fra et fag a til et fag b hvis a er anbefalt forkunnskap til b. En gyldig topologisk sortering vil dermed være en gyldig rekkefølge å ta fagene i.

Oppgave 14

«*DFS*» er riktig.

Kommentar:
Vi kan se på denne grafen som et rotfestet tre. Rotnoden er topplederen og alle de ansatte (bortsett fra topplederen) har en kant mellom seg selv og sjefen sin. Man kan her se at en ansatt A er underordnet en annen ansatt B dersom A sin node er en ekte etterkommernode (*proper descendant*) av B sin node. Vi kjører DFS fra rotnoden. Parentesteoremet sier at dersom ansatt A sin node a oppdages og avsluttes innenfor intervallet mellom oppdagelse og avslutning av ansatt B sin node b, så er a en ekte etterkommer av b. Dermed kan vi sjekke om en node er en ekte etterkommer av en annen node i $\mathrm{O}(1)$ tid, eller med andre ord sjekke at en ansatt er underordnet en annen, ved å sammenlikne $b.d < a.d$ og $a.f < b.f$ (Korollar 22.8 i boka (s. 608)).

Man kan se at det er $n = \mathrm{O}(n)$ noder i treet siden det er en node for hver ansatt. Det er $n-1 = \mathrm{O}(n)$ kanter i treet siden hver node, bortsett fra noden til topplederen, har én kant til sjefen. Kjøringen av DFS tar $\Theta(\mathrm{V}+\mathrm{E}) = \mathrm{O}(n)+\mathrm{O}(n) = \mathrm{O}(n)$ tid, som dermed blir tidskompleksiteten på algoritmen.

(Denne oppgaven er basert på oppgave 3b), 2011H)

Oppgave 15

«*Kan brukes til å finne korteste vei fra en node til en annen dersom alle kantvektene er like og ikke-negative*» er riktig.

Kommentar:
BFS kan brukes til å finne korteste vei fra en node til en annen dersom alle kantvektene er like og ikke-negative. Etter teorem 22.5 i læreboka (s. 599) vil BFS finne korteste vei fra en node til en annen målt i antallet kanter. Hvis kantvektene er alle $w \geqslant 0$ vil det aldri lønne seg å gå på flere kanter enn nødvendig. Om antallet kanter på en sti er k_1 og på en annen sti er k_2, så vil lengden på stiene være gitt av henholdsvis $k_1 w$ og $k_2 w$. Videre har vi at

$$w \geqslant 0 \implies [k_1 w < k_2 w \implies k_1 < k_2]$$

og dermed vil den korteste veien målt i antall kanter være optimal, selv om kantvektene ikke er 1.

Man er ikke garantert at BFS besøker alle nodene i grafen. BFS starter kun å søke fra en node,

noe som betyr at om en node ikke er mulig å nå fra startnoden, vil den heller ikke bli besøkt.

BFS har ingen problemer om grafen har sykler. Den besøker fortsatt alle nodene den kan og finner fortsatt korteste vei fra startnoden.

Oppgave 16

Riktige alternativer:

- Man er garantert at alle nodene blir besøkt i traverseringen.
- Kan brukes til kantklassifisering.

Kommentar:
Man er garantert at alle nodene blir besøkt i traverseringen, siden man har en ytre løkke som går gjennom alle nodene og prøver å traversere fra dem om de ikke er besøkt.

DFS kan brukes til kantklassifisering i følge læreboka (s. 609).

DFS kan brukes med sykler. Det er eksempler der DFS er brukt på grafer med sykler i læreboka.

DFS kan ikke brukes til å finne korteste vei mellom to noder, siden DFS kan traversere til en node på en vei som ikke er kortest, og kan derfor ikke propagere korteste vei slik som BFS kan.

Oppgave 17

«*Tre-kanter*» er riktig.

Kommentar:
Siden det kun finnes en sti mellom to noder i et tre, må en kant vi traverserer gå til en node vi ikke har besøkt før, og dermed blir kanten en tre-kant.

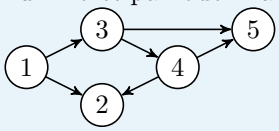
Oppgave 18

Riktige alternativer:

- Tre-kanter
- Foroverkanter
- Krysskanter

Kommentar:
En DAG vil ikke ha noen bakoverkanter (lemma 22.11 på side 614 i læreboka).

Vi ser nå på følgende DAG, som vi kjører DFS på. Kantene i nabolistene er sortert stigende etter nummeret på noden kanten går til.



Her vil vi ha:

- Tre-kanter: $\{(1,2),(1,3),(3,4),(4,5)\}$
- Foroverkanter: $\{(3,5)\}$
- Krysskanter: $\{(4,2)\}$

Oppgave 19

```
from collections import deque


def buildable(i, j, build_map):
    if 0 <= i < len(build_map) and 0 <= j < len(build_map[0]):
        return build_map[i][j]
    return False


def shortest_road(build_map, start, end):

    # Forgjengere
    predecessors = [[None for _ in build_map[0]] for _ in build_map]
    predecessors[start[0]][start[1]] = (-1, -1)

    # Bruker en kø for å ta vare på nodene som skal besøkes
    queue = deque()
    queue.append(start)
    while len(queue):
        pos = queue.popleft()
        i, j = pos
        if (i, j) == end:
            positions = [(i, j)]
            while predecessors[i][j] != (-1, -1):
                i, j = predecessors[i][j]
                positions.append((i, j))
            return positions

        for i, j in [(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)]:
            if buildable(i, j, build_map) and predecessors[i][j] is None:
                queue.append((i, j))
                predecessors[i][j] = pos
    return None
```

Kommentar:
Dette var en oppgave der målet var å finne korteste vei mellom landsbyene, i en implisitt graf gitt av kartet. Siden hver rute i kartet har vekt 1, vil BFS gi riktig svar.

For de som ønsket highscore (utenfor pensum):
Et problem med å søke med BFS i slike grafer, er at man søker i alle retninger samtidig. Hvis bakken er slik at det går an å grave de fleste steder, gir det mening å rette inn søket slik at man først og fremst søker i retningen mot jordet. For å komme høyt på highscore-listen måtte du nok implementert et mer rettet søk som for eksempel A*:

```
from heapq import heappush, heappop


def manhattan_distance(pos1, pos2):
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])


# A* med manhattanavstand som heuristikk
def shortest_road(build_map, start, end):

    # Oppslagstabell med korteste avstand or forgjenger.
    nodes = {}
    nodes[start] = (0, (-1, -1))

    # Besøkte noder
    visited = set()

    # Prioritetskø med (søkeverdi = avstand + heuristikk, avstand, posisjon)
    heap = []
    man_dist = manhattan_distance(start, end)
    heappush(heap, (man_dist, 0, start))
    shortest_dist = float("inf")
    while len(heap):
        heuristic, dist, pos = heappop(heap)
        if heuristic >= shortest_dist:
            break
        if pos in visited:
            continue
        visited.add(pos)
        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 < len(build_map)
                and 0 <= j < len(build_map[0])
                and build_map[i][j]
            ):
                if (i, j) not in nodes or nodes[(i, j)][0] > dist + 1:
                    new_dist = dist + 1
                    new_heuristic = new_dist + manhattan_distance(
                        (i, j), end
                    )
```

```
                nodes[(i, j)] = (new_dist, pos)
                heappush(heap, (new_heuristic, new_dist, (i, j)))
                if (i, j) == end:
                    shortest_dist = new_dist

    # Henter ut korteste vei, hvis funnet
    if shortest_dist != float("inf"):
        i, j = end
        positions = [(i, j)]
        while nodes[(i, j)][1] != (-1, -1):
            i, j = nodes[(i, j)][1]
            positions.append((i, j))
        return positions

    return None
```

Oppgave 20

Problemet her er at når vi setter inn nodene fremst i den lenkete listen, tar dette O(1) tid, men det å sette noe fremst i en dynamisk tabell tar $\Theta(n)$ tid. Dynamiske tabeller er laget slik at det å sette inn et element bakerst i tabellen tar amortisert konstant tid, men hvis man skal sette inn noe foran i tabellen må man både øke størrelsen på tabellen (som tar amortisert konstant tid) og flytte alle elementene et hakk bakover (som tar lineær tid). Siden vi må flytte alle nodene et hakk bakover hver gang vi skal sette inn en ny node i listen, vil dette gi en tidskompleksitet på $\Theta(V^2 + E)$ i stedet for $\Theta(V + E)$. Det finnes enkle løsninger på dette (ikke en utfyllende liste):

1. Lage en ny type amortisert tabell, der man kan sette inn foran i amortisert konstant tid.
2. Sette inn nye noder i tabellen bak, som tar amortisert konstant tid, og reversere tabellen til slutt i lineær tid.
3. Initialisere tabellen til å være av størrelse $|V|$ i utgangspunktet og stadig sette inn nodene på bakerste ledige plass i tabellen.

Oppgave 21

Dette er en versjon av Dijkstras algoritme, og den finner korteste vei dersom kantvektene i grafen er ikke-negative. Bevis for korrekthet kan du finne på side 660 i læreboka.

Oppgave 22

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.