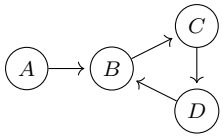
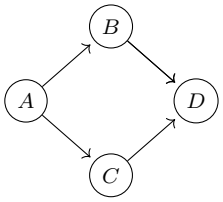
Dependency hell 12 poeng

I moderne programvareutvikling er det vanlig at en applikasjon er avhengig av mange *biblioteker*, som igjen kan avhenge av andre biblioteker. Et bibliotek kan ses på som en samling med kode som er designet for gjenbruk. Bibliotekene som brukes for å bygge en applikasjon kalles *avhengighetene* til applikasjonen. Avhengighetene mellom biblioteker er gitt som en rettet graf $G = (V, E)$, der en node representerer et bibliotek, og hvor det går en kant fra u til v dersom u bruker v som et bibliotek.

Eksemplene under er situasjonener ofte omtalt som *dependency hell*.



Sirkulær avhengighet



Diamantavhengighet

Sykliske avhengigheter er som regel uønsket fordi en oppdatering av et av bibliotekene i en sykel vil kunne påvirke alle de andre bibliotekene i sykelen. *Diamantavhengighet* er også som regel uønsket fordi en oppdatering kan føre til at man blir avhengig av forskjellige versjoner av det samme biblioteket samtidig.

Du skal utvikle algoritmer som kan hjelpe med å identifisere slike uønskede situasjoner.

(a) Mer presist sier vi at grafen inneholder en *syklisk avhengighet* dersom det finnes en sti fra en node v til den samme noden v som består av minst to noder.

Skriv en prosedyre som, gitt en rettet graf $G = (V, E)$, returner **`true`** dersom grafen inneholder *sykliske avhengigheter*, og **`false`** ellers. Dersom prosedyren din benytter seg av algoritmer kjent fra pensum må du redgjøre kort for hvordan de fungerer.

Dette bør løses på lineær tid. Mest naturlig er å bruke topologisk sortering (som oppdager sykler). Algoritmen for topologisk sortering plasserer først alle noder med inngrad 0 på en stack. Noder plukkes av stacken så lenge den ikke er tom, og for hver node som plukkes av fjernes alle dens utgående kanter (som fører til at dens naboer får lavere inngrad). Hvis en node får inngrad 0 plasseres den på stacken. Grafen er asyklisk hvis og bare hvis alle nodene blir prosessert.

```
1  Procedure isCyclic(G)
2      stack ← empty stack
3      processed ← 0
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          processed ← processed + 1
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     return processed < |V|
```

En annen mulighet er å finne de sterkt sammenhengende komponentene av grafen, og sjekke om det finnes komponenter av størrelse større enn én.

Det er også fullt mulig å skrive et modifisert dybde-først søk som oppdager sykler. I en slik løsning vil man typisk skille mellom en node er *uoppdaget*, *under prosessering* eller *ferdig prosessert*. Dersom man under et søk finner en kant til en node som er *under prosessering* kan man konkludere med at grafen inneholder en sykel. En annen lignende (og nærliggende) løsning er å sjekke om en node er på *stacken* (fordi da er den under prosessering); med en vanlig stack vil dette gi $\mathcal{O}(|V| \cdot (|V| + |E|))$.

- Opptil 5 poeng for en godt beskrevet algoritme som løser problemet i $\mathcal{O}(|V| + |E|)$.
- Opptil 3 poeng dersom løsningen er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

(b) I denne deloppgaven kan du anta at grafen ikke inneholder sykliske avhengigheter.

Mer presist sier vi at grafen inneholder en *diamantavhengighet* dersom det finnes to distinkte stier fra u til v. Husk at to stier er distinkte dersom de ikke deler noen noder eller kanter utenom u og v.

Skriv en prosedyre som, gitt en rettet asyklisk graf $G = (V, E)$, returnerer **`true`** dersom grafen inneholder *diamantavhengigheter*, og **`false`** ellers.

Det er tilstrekkelig å gjøre en traversering for hver node med inngrad null. Hvis vi sier at V_S er nodene i V som har inngrad null vil algoritmen være i $\mathcal{O}(|V_S| \cdot (|V| + |E|))$. Hvis vi antar at det kun er én node med inngrad null, så er algoritmen i $\mathcal{O}(|V| + |E|)$. Hvis vi antar at alle nodene har inngrad null, så algoritmen i $\mathcal{O}(|V|)$ (fordi alle søk vil starte og stoppe i samme node). Verste tilfelle må ligge et sted mellom disse ytterkantene, og være begrenset til $\mathcal{O}(|V| \cdot (|V| + |E|))$. Det er allikevel en betydelig forbedring fra å søke fra alle nodene.

```
1  Procedure isDiamond(G)
2      V_S ← the empty set
3
4      for v ∈ V do
5          if indegree of v is 0 then
6              add v to V_S
7
8      for s ∈ V_S do
9          stack ← stack containing s
10         visited ← empty set
11
12         while stack is not empty do
13             u ← stack.pop()
14             if u ∈ visited then
15                 return true
16             add u to visited
17             for (u, v) ∈ E do
18                 add v to stack
19
20     return false
```

En lignende løsning er å ta utgangspunkt i alle noder som har inngrad større eller lik to, og søke i den reverserte grafen.

Uten disse forbedringene vil en løsning typisk gjøre én traversering for hver node, som er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

- Opptil 7 poeng for en god løsning som er omtrent like effektiv som forslaget ovenfor.
- Opptil 5 poeng for en god løsning som gjør en traversering per node i grafen.
- Opptil 4 poeng for en god løsning som ikke er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

Generelt bør det gis noe uttelling for de som viser god forståelse for problemet, og skisserer en løsning som kunne fungert.