Algoritmer og Datastrukturer 2

Aflevering 4

Kristian Gausel[1], Lasse Alm[2], and Steffan Sølvsten[3]

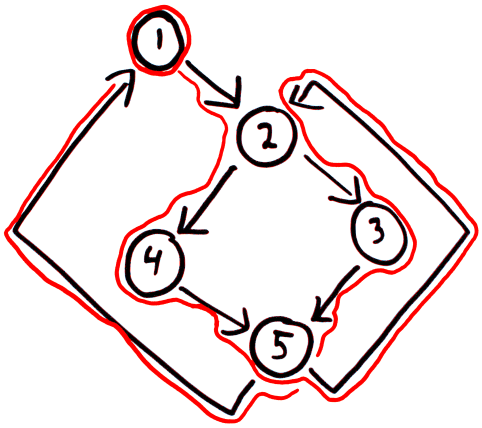
[1]201509079@post.au.dk
[2]201507435@post.au.dk
[3]201505832@post.au.dk

22. december 2016

1 [CLRS] Problem 22.3

En *Euler tour* i en orienteret graf $G = (V, E)$, der også er et stærkt sammenhængskomponent, er en cykel som besøger alle kanter i G præcis en gang, og kan besøge en knude mere end en gang.

I figur 1 er en graf bestående af 5 knuder, hvori en Euler tour startende i knuden v_1 er optegnet. Denne Euler tour kan skrives som: $1, 2, 4, 5, 2, 3, 1$.



Figur 1: En Euler tour (rød) startende i knude 1

2 a - InDegree(V) = OutDegree(V) hviss en Euler tour eksisterer

Der skal vises, at **hviss** $in - degree(v) = out - degree(v)$, så er der en Euler Tour i grafen G, hvor $v \in V$. I det følgende vil der blive argumenteret for at implikationen er gensidig.

2.1 Euler tour eksisterer $\implies$ InDegree(V) = OutDegree(V)

Idet det er givet, at G består af et stærkt sammenhængskomponent og at der eksisterer en Euler tour, så må enhver knude være del af Euler touren. Alle kanter for en arbitrær knude $v \in V$ er inkluderet i Euler touren. Da Euler touren er en cykel, hvor hver kant kun besøges en gang, så må for hver inadgående kant til v være en korresponderende udadgående kant. Modsat må der for hver udadgående kant fra v også være en korresponderende inadgående kant.

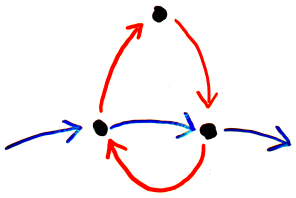
Figur 2: For $v \in V$ må $in - degree(v) = out - degree(v)$, hvis der er en Euler Tour

2.2 InDegree(V) = OutDegree(V) $\implies$ Euler tour eksisterer

Idet at G består af et stærkt sammenhængskomponent og at for alle $v \in V$ er $in - degree(v) = out - degree(v)$, så må der kunne findes en sti fra v til sig selv, hvor hver kant kun besøges en gang. Særskilt for både de besøgte og også de ikke besøgte kanter er $in - degree(v) = out - degree(v)$ en invariant.

Det er dog ikke givet, at denne cykel er en Euler tour, idet dele af grafen muligvis ikke er inkluderet. Flere cykler kan dog dannes, indtil at der ikke er flere ubesøgte kanter, hvormed for ubesøgte kanter på knuden v er $in - degree(v) = 0 = out - degree(v)$.

Da hele grafen er et stærkt sammenhængskomponent, så må der være en eller flere overlappende knuder i to cykler. Følges den første cykel fra start til slut, men hvor den anden cykel følges fra start til slut efter den første fælles knude, så vil cyklerne *svejses* sammen.



Figur 3: For to cykler, så må der være overlappende knuder, hvorom cyklerne kan *svejses* sammen

Sammensættes alle cykler på denne måde, så må der dannes en stor cykel. Idet grafen er et stærkt sammenhængskomponent, så er alle kanter inkluderet i alle subcyklerne og dermed i den endelige store cykel. Denne cykel må være en Euler tour.

3 b - Algoritme for bestemmelse af *Euler tour*

En algoritme til at finde en Euler tour af G, der har en tidskompleksitet på $O(E)$ skal bestemmes.

3.1 Algoritme

Algoritmen er todelt, hvor cykler dannes i *CreateCycle* (1) og disse samles sammen til en EulerTour i *FindEulerTour* (2)

Til at bestemme en cykel, hvor en kant kun besøges en gang, benyttes farvning af kanterne. En kant, som endnu ikke er besøgt er *hvid*, imens en allerede besøgt kant farves *sort*. Startende i en knude v, som er givet med som et argument, følges tilfældige hvide kanter, som farves sort løbende. Hver knude, som mødes på stien, tilføjes til en dobbeltkædede liste. Dette slutter, efter at en kant følges tilbage igen til knuden v. Listen af stien fulgt returneres.

Kode 1: Kode for at udprinte indeks for delsekvenser af z, som er lig x

```
1  CreateCycle (V_init)
2      Tour = New Circular Doublylinked List
3      V = V_init
4      //Follow a random edge, until we reach the start
5      do
6          Tour.append(V)
7          //Find a white outgoing edge from V
8          edge = None
9          for e in V.getEdges():
10             if e.color == "white":
11                 edge = e
12                 break
13
14         //Follow the edge and color it black
15         if edge ≠ NIL
16             edge.color = "black"
17             V = edge.getTo()
18         else
19             error "Found a dead end"
20
21     while V ≠ V_init
22
23     //Append the start vertice again
24     Tour.append(V)
25
26     return Tour
```

Listerne dannet i *CreateCycle* sammensættes i *FindEulerTour*, der tager en graf G som argument. Hertil farves først alle kanter hvide, hvorefter en tilfældig knude v vælges. En cykel startende i v dannes med *CreateCycle*. Denne cykel gennemgås herefter for at finde stadig hvide kanter. Hvis en sådan hvid kant findes, så kaldes igen *CreateCycle* på denne knude, hvormed en cykel herfra dannes. Denne cykel indsættes i den første cykel på knudens egen plads. Dette gentages, indtil hele den første cykel, inklusive dens indsatte cykler er gennemløbet.

Kode 2: Kode for at samle alle cycler i én, så en Euler tour dannes

```
1  FindEulerTour(G)
2      //Take a random vertice in the graph
3      //Easiest to just take the first one in list of vertices
4      V_start = G.getVertices[0]
5
6      //Color all edges white
7      for every edge in G.getEdges
8          edge.color = "white"
9
10     //Find an initial cycle
11     EulerTour = CreateCycle(V_start)
12     i = 0
13     V = EulerTour[i]
14
15     do
16         //Find a white outgoing edge from V
17         edge = NIL
18         for e in V.getEdges():
19             if e.color == "white":
20                 edge = e
21                 break
22
23         //Is there an unresolved cycle?
24         if edge ≠ NIL
25             Cycle = CreateCycle(V)
26             Insert Cycle in EulerTour replacing V
27
28         //Follow to the next
29         V = next element in EulerTour
30
31     while V ≠ last in EulerTour
32
33     if all edges in G.getEdges are black
34         return EulerTour
35     else
36         error "Not a strongly connected graph"
```

3.2 Korrekthed

Såfremt at en Euler tour findes i G, så bør denne kunne blive bestemmes på en lignende metode som beskrevet i afsnit 2.2. Da for en Euler tours eksistens bør grafen være et stærkt sammenhængskomplement, så må alle knuder være inkluderet, hvorfor den første knude i listen V kan vælges.

Cyklerne dannet af *CreateCycle* besøger aldrig en kant, som allerede er tidligere besøgt af selv eller af et andet kald til *CreateCycle*, da alle besøgte kanter farves sort. For at en Euler tour kan findes, så må, som argumenteret i afsnit 2, $in - degree(v) = out - degree(v)$, hvorfor stien må ende op som en lukket cykel. Løkken terminerer dog ved ankomsten til den startende knude, hvorfor denne skal tilføjes særskilt for korrekt dannelse af cyklen.

I gennemløbes af den første cykel for at finde stadig hvide kanter fastsat til knuder v, så tilføjes nye cykler hertil startende i v. Da *CreateCycle* danner en ny korrekt lukket cykel, der ikke genbruger tidligere kanter, så kan den indsættes som en subcykel i stedet for v. Da subcyklen er lukket, så starter og slutter den i v, hvormed den første cykel ikke er ugyldig. Gennemløbet går videre gennem den nyindsatte cykel, så hvis der er flere hvide kanter på samme knude v, så kan alle disse cykler dannes, når v undersøges efter gennemløbet af subcyklen.

Da $in - degree(v) = out - degree(v)$, så må antallet indgående og udgående hvide kanter altid være konstant, da en cykel altid farver en indgående og en udgående kant sort ved dannelsen af cykler i *CreateCycle*. For hele grafen må der derfor gælde, såfremt der er en Euler tour

$$|\text{Hvide inadgående kanter} \in G| = |\text{Hvide udadgående kanter} \in G| \tag{1}$$

Da G bør være et stærkt sammenhængskomplement, så bør alle kanter kunne nås fra enhver knude v, hvormed de alle bør være på cyklen, efter tilføjelse af alle subcykler er indsat. Er der derimod stadig hvide kanter, så kan disse ikke nås fra v, hvormed der ikke er nogen Euler tour. Algoritmen vil derfor kun returnere den dannede cykel, hvis alle kanter er sorte og bør give en fejl, hvis invarianten i ligning 1 er brudt.

3.3 Tidskompleksitet

Ved kørsel af algoritmen besøges og udføres beregninger på en kant kun to gange. Dette er under dannelsen af en cykel med *CreateCycle* og derefter i gennemløbet af den endelige cykel i *FindEulerTour*. Idet listen af cykler er dannet som en cirkulær dobbeltkædede liste, så tager det konstant tid at finde det første og sidste element og det tager konstant tid at udskifte en knude v i en liste med en ny cykel startende i v. Idet, at der bruges konstant arbejde per kant, så vil tidskompleksiteten af algoritmen være $O(E)$.

Litteratur

[1] Cormen, Thomas H. mfl.: *Introduction to Algorithms*, 3rd ed.