

# Graph Teori

Johan Brinch

18. november 2009

# Indhold

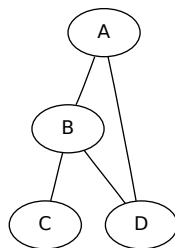
<b>1</b>	<b>Introduktion til grafer</b>	<b>3</b>
1.1	Knuder og kanter . . . . .	3
1.2	Orienterede grafer . . . . .	3
1.3	Veje . . . . .	5
1.4	Vægtede kanter . . . . .	5
1.5	Søgning i grafer . . . . .	6
1.5.1	Dybde først . . . . .	6
1.5.2	Bredde først . . . . .	6
1.6	Korteste vej . . . . .	6
<b>2</b>	<b>Eksempler på grafer (Hvad kan det bruges til?)</b>	<b>6</b>
2.1	Vennenetværk (ikke-orienteret graf) . . . . .	6
2.2	Telefonnumre (orienteret graf) . . . . .	6
2.3	Repræsentation af en sudoku som en graf . . . . .	6
<b>3</b>	<b>Implementation af en graf i Python</b>	<b>6</b>
3.1	Analyse . . . . .	6
3.2	Designovervejelser . . . . .	7
3.2.1	Implementation af metoder . . . . .	8
3.2.2	Fejlhåndtering . . . . .	10
3.3	Endelig Implementation . . . . .	12
3.4	Test og afprøvning . . . . .	15

# 1 Introduktion til grafer

Grafen er en datastruktur. Af andre datastrukturer findes blandt andet `list`, `tuple` og `dictionary`.

Lister og tupler lagrer elementer i en rækkefølge, f.eks. `[1,0,2,1]`. Dictionaries lagrer par af nøgler og værdier, så en værdi kan findes ved opslag på dens tilhørende nøgle, f.eks. `{a: 1, b: 0, c: 2, d: 1}`.

Grafer bruges til at beskrive forhold. Grafen indeholder elementer, kaldet knuder, og forhold mellem par af elementer, kaldet kanter. For eksempel kunne en graf lagre personer hvis forhold er givet ved hvorvidt to personer er venner eller ej. Hver knude ville symbolisere en person og hver kant et venskab.



Figur 1: Eksempel på graf

## 1.1 Knuder og kanter

En graf består af knuder og kanter. Alle kanter forbinder to knuder. En knude kan have nul eller flere kanter. Knuderne i grafen er „emnerne“, som grafen beskriver. Kanterne er deres relationer til hinanden.

### Eksempel 1.1.1 *Metronettet*

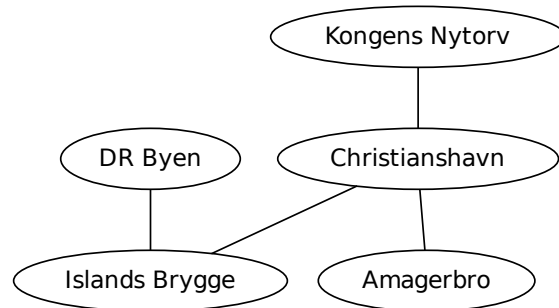
Et eksempel på en graf kunne være metronettet. Her kunne stationer repræsenteres som knuder og forbindelser imellem dem som kanter.

Figur 2 på den følgende side viser en graf over metronettet med 5 knuder og 4 kanter. Knuderne er stationer og kanterne forbindelser. Grafen bruges her til at vise hvilke stationer der er forbundet til hinanden.

Grafen kaldes retningsløs eller *undirected* fordi kanterne ikke angiver en retning. En forbindelse virker begge veje. Hvis man kan tage toget fra Nørreport til Kongens Nytorv er det også muligt at gå den anden vej. I en graf hvor kanterne angiver en retning (en *directed* graf) er det kun muligt at „følge“ kanten i den angivne retning. Forbindelsen virker kun den ene vej. Dens er *ensrettet*.

## 1.2 Orienterede grafer

En orienteret (*directed*) graf er en graf hvori alle kanter har en tilknyttet retning. Kanten kan nu kun følges i den angivne retning. Det er dog muligt at tillade

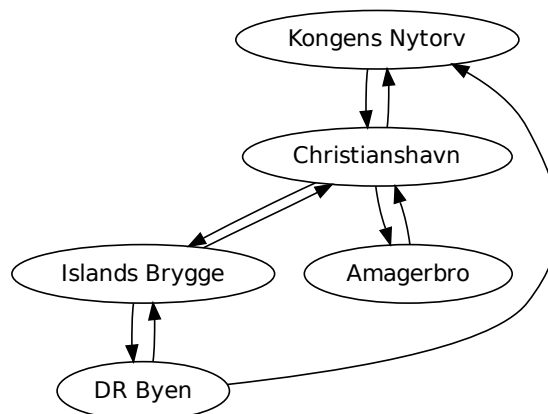


Figur 2: Graf over et udsnit af metronettet

„bevægelse“ mellem to knuder i begge retninger, ved at tilføje to kanter: en i hver retning.

#### Eksempel 1.2.1 Metronettet udbygges med ensretning

Sæt nu, at man vælger at bygge en ensrettet forbindelse mellem DR Byen og Kongens Nytorv. Det er nu muligt at tage toget fra DR Byen til Kongens Nytorv, men ikke den modsatte vej. Grafen over metronettet må dermed laves med en orienteret graf, hvor alle kanter angiver en gyldig retning. Ved de forbindelser, der tillader bevægelse i begge retninger bruger vi to kanter. Figur 3 viser grafen fra figur 2 udvidet med orientering og den nye kant.



Figur 3: Orienteret graf over metronet

### 1.3 Veje

En vej i en graf er et sæt af knuder, der er forbundne af kanter.

#### Eksempel 1.3.1 Til DR Byen og tilbage igen

Figur 2 på foregående side viser en uorienteret graf over metronettet. Betragt vejen:

DR Byen  $\rightarrow$  Islands Brygge  $\rightarrow$  Christianshavn  $\rightarrow$  Kongens Nytorv

Det er den korteste vej (3 brugte kanter) mellem DR Byen og Kongens Nytorv. Men det er ikke den eneste vej mellem de to stationer. Man kunne have taget forbi Amagerbro på vejen, blot for at vende om og fortsætte mod Kongens Nytorv. Denne vej ville have brugt 5 kanter i stedet for 3.

Hvis vi i stedet vender blikket mod den orienterede graf fra figur 3 på forrige side, ser vi at ovenstående ikke længere er den korteste vej fra DR Byen til Kongens Nytorv. Vi kan nu gøre brug af den nye direkte forbindelse og få vejen:

DR Byen  $\rightarrow$  Kongens Nytorv

Vejen indeholder nu kun 1 kant (2 mindre). Desværre kan den direkte forbindelse kun bruges i en retning, så tilbagevejen bliver noget længere:

Kongens Nytorv  $\rightarrow$  Christianshavn  $\rightarrow$  Islands Brygge  $\rightarrow$  DR Byen

Tilbagevejen bruger igen 3 kanter.

#### Eksempel 1.3.2 Cykler

En vej kaldes en cykel (*cycle*) hvis start- og slutknuden er den samme. Vi kan sammensætte de to veje fra eksempel 1.3.1 og få cyklen:

DR Byen  $\rightarrow$  Kongens Nytorv  $\rightarrow$  Christianshavn  $\rightarrow$  Islands Brygge  
 $\rightarrow$  DR Byen

En vej kaldes en cykel, hvis den starter og slutter med samme knude.

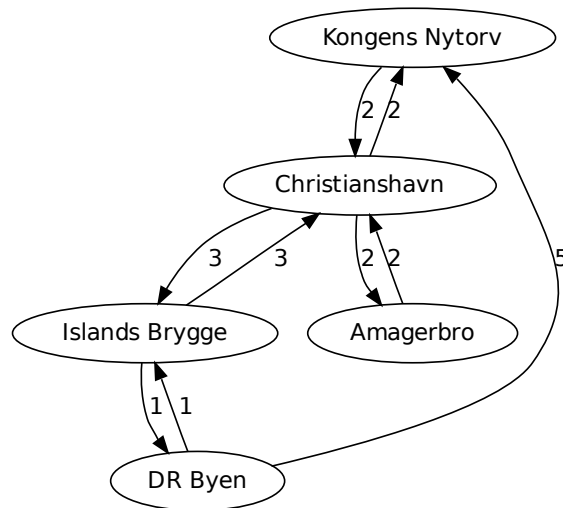
### 1.4 Vægtede kanter

Indtil nu har vi kun beskæftiget os med uvægtede grafer. I korteste vej eksemplet fra eksempel 1.3.1 var længden af vejen antallet af kanter der indgik i vejen. Man tænke på det, som at hver kant kostede 1 at bruge.

I en vægtet graf tildeles hver kant en vægt (*weight*), der angiver hvor meget kanten koster. Man kan tænke på det som en afstand mellem de to knuder kanten forbinder. Vægten angiver dermed kantens længde.

Lad vende tilbage til metronettet. Figur 4 på næste side viser figur 3 på forrige side med angivene vægte. Hver vægt viser antallet af minutter det tager at bevæge sig over den tilhørende kant.

Ifølge grafen tager den direkte vej fra DR Byen til Kongens Nytorv 5 minutter, mens den „gamle“ vej, over Christianshavn, tager 6 minutter. Cyklen fra DR Byen til Kongens Nytorv og tilbage igen ville altså tage 11 minutter at gennemføre.



Figur 4: Vægtet graf over metronet

## 1.5 Søgning i grafer

### 1.5.1 Dybde først

### 1.5.2 Bredde først

## 1.6 Korteste vej

## 2 Eksempler på grafer (Hvad kan det bruges til?!)

### 2.1 Vennetværk (ikke-orienteret graf)

### 2.2 Telefonnumre (orienteret graf)

### 2.3 Repræsentation af en sudoku som en graf

## 3 Implementation af en graf i Python

Nu hvor det ligger fast hvad en graf er og hvad den kan bruges det ville det være rart med en implementation, som vi kan bruge i Python-programmer.

I denne sektion vil vi igennem analyse (hvad skal det kunne?) designovervejelser (hvordan skal den kunne det?) og implementation (hvordan koder man det?) få opbygget et Python grafmodul.

### 3.1 Analyse

I dette afsnit diskuteres funktionaliteten af en graf. Hvad skal den kunne?

Kogt helt ned, skal et grafmodul besidde funktionalitet:

1. Oprettelse af en tom graf
2. Indsættelse af knuder i en graf
3. Sletning af knuder i en graf
4. Tilføjelse af kant mellem to knuder i en graf
5. Sletning af kant mellem to knuder i en graf

Med disse funktioner kan vi opbygge og nedrive grafer. Men før de er anvendelige er det nødvendigt også at kunne undersøge grafer. Følgende funktionalitet omhandler netop dette:

1. Findes knude  $k$  i grafen?
2. Er knude  $k_1$  og knude  $k_2$  forbundet af en kant?
3. Hvor mange knuder indeholder grafen?
4. Hvor mange kanter indeholder grafen?

### 3.2 Designovervejelser

I analysen afsnit 3.1 på foregående side gennemgik jeg funktionaliteten af en graf. Nu vil jeg diskutere hvordan denne funktionalitet kan implementeres i Python.

Eftersom mine funktionaliteter beskæftiger sig med en specifik instans af en graf, vælger jeg at lave en klasse `Graf`, der repræsenterer en graf med de beskrevne muligheder. Hver funktionalitet implementeres som en metode, der arbejder på grafen.

Inden jeg kan gå i gang med at beskrive, hvordan en graf kan ændres, skal jeg først vælge en repræsentation af en graf. Hvordan kan skal grafobjektet holde styr på knuder og kanter?

Der er følgende traditionelle måder at gøre dette på:

1. Hver knude indeholder en liste af andre knuder, som den er forbundet til.
2. Hver kant består af et par knuder, som denne forbinder.
3. En matrix af (knuder  $\times$  knuder) angiver om et given par er forbundet<sup>1</sup>.

Den første løsning kan implementeres med en `Knude`-klasse, som beskriver en knude. Det eneste knuden skal indeholde er sin egen værdi, samt en liste over andre knuder, som den er forbundet til. At undersøge om to knuder er forbundet kan gøres ved at løbe deres lister igennem og se om de har hinanden som element. Køretiden for operationen afhænger af hvor mange elementer de to knuder er forbundet til.

Den anden løsning kan implementeres med en `Kant`-klasse der indeholder et par af knuder (de to kender, som kanten forbinder). `Graf`-klassen skal nu indeholde en liste af knuder sammen med en liste af kanter. At undersøge om

---

<sup>1</sup>tænk på matricen som en liste af lister, der hver har et element for hver knude.

to knuder er forbundet af en kant kan gøres ved at løbe listen af kanter igennem og for hver kant undersøge om de to knuder passer med parret. Køretiden er afhængig nu af det samlede antal kanter i grafen.

Den tredje løsning kan også implementeres med lister, selvom den er tænkt til matricer. Her behøves hverken en `Kant`- eller `Knude`-klasse. `Graf`-klassen indeholder en liste af knuder, der hver har en liste af knuder. Hvis der findes en kant fra knude nummer 1 til knude nummer 2 vil det andet element i den første liste være sand (f.eks.: `kanter[1][2] == True`). Hvis kanten ikke findes vil værdien være falsk (altså: `kanter[1][2] == False`). Køretiden afhænger (grundet valget af lister) af antallet af knuder og bliver dermed  $O(k)$ .<sup>2</sup>

Jeg vælger at implementere model nummer et på grund af dens simplicitet. I det følgende beskriver jeg hvordan jeg implementere hver enkelte funktionalitet.

### 3.2.1 Implementation af metoder

**Opret tom graf** Den tomme graf er et `Graph`-objekt uden nogen tilhørende knuder. Listen af knuder er sat til den tomme liste. En tom graf kan oprettes ved at instansiere klassen `Graph`. Dennes `__init__` metode sætter listen af knuder til den tomme liste:

```
class Graph:
    ...
    def __init__(self):
        self.nodes = []
```

**Indsættelse af knude** En knude kan indsættes med `Graph`-metoden `add_node(node)`. Metoden virker ved at tilføje den nye knude til grafens liste over knuder:

```
class Graph:
    ...
    def add_node(self, node):
        self.nodes.append(node)
```

**Sletning af knude** En knude kan fjernes fra en graf med `Graph`-metoden `del_node`. Metoden virker ved at fjerne knuden fra grafens list af knuder:

```
class Graph:
    ...
    def del_node(self, node):
        self.nodes.remove(node)
```

**Tilføjelse af kant** En kant fra knude  $a$  til knude  $b$  kan tilføjes, ved at tilføje knude  $b$  til listen over kanter i knude  $a$ .

```
class Graph:
    ...
    def add_edge(self, node_a, node_b):
        node_a.add_edge(node_b)
```

I `Node`-klassen tilføjer `add_edge` metoden knuden til dens liste over kanter:

---

<sup>2</sup>med matricer kan køretiden reduceres til  $O(1)$ .



```
class Node:
    ...
    def add_edge(self, node):
        self.edges.append(node)
```

**Sletning af kant** En kan fra knude  $a$  til knude  $b$  kan slettes, ved at fjerne knude  $b$  fra listen over kanter i knude  $a$ .

```
class Graph:
    ...
    def del_edge(self, node_a, node_b):
        node_a.del_edge(node_b)
```

I Node-klassen fjernes knuden fra listen over kanter:

```
class Node:
    ...
    def del_edge(self, node):
        self.edges.remove(node)
```

**Findes knude  $k$  i grafen?** Hvis knude  $k$  ligger i grafen, ligger den i listen over knuder:

```
class Graph:
    ...
    def has_node(self, node):
        return (node in self.nodes)
```

**Findes en kant fra knude  $k_1$  til knude  $k_2$ ?** Hvis en kan findes fra knude  $k_1$  til knude  $k_2$  må knude  $k_2$  ligge i listen over  $k_1$ 's kanter:

```
class Graph:
    ...
    def has_edge(self, node_a, node_b):
        return node_a.has_edge(node_b)
```

I Node-klassen testes om  $node_b$  ligger i listen over kanter:

```
class Node:
    ...
    def has_edge(self, node):
        return (node in self.edges)
```

**Hvor mange knuder indeholder grafen?** Antallet af knuder i grafen er det samme som længden af listen over knuder:

```
class Graph:
    ...
    def count_nodes(self):
        return len(self.nodes)
```

**Hvor mange kanter indeholder grafen?** Antallet af kanter i grafen er lig med antallet af fra de enkelte knuder:

```
class Graph:
    ...
    def count_edges(self):
        count = 0
        for node in self.nodes:
            count += node.count_edges()
        return count
```

I Node-klassen tælles antallet af kanter ved at tælle antallet af elementer i listen over kanter:

```
class Node:
    ...
    def count_edges(self):
        return len(self.edges)
```

### 3.2.2 Fejlhåndtering

Flere af ovenstående punkter kan give en fejl. Man kan forsøge at fjerne en knude der ikke findes, eller tilføje en kant der findes i forvejen. I dette afsnit vil jeg belyse de metoder der kan fejle, og udvide dem med fejlhåndtering.

**Definition af fejl:** Til at starte med definerer jeg fire fejltyper (exceptions), som grafen kan *kaste* i tilfælde af fejl:

```
class NodeAlreadyExistsError(Exception):
    """ Knuden findes allerede """
    pass

class NoSuchNodeError(Exception):
    """ Knuden findes ikke """
    pass

class EdgeAlreadyExistsException(Exception):
    """ Kanten findes allerede """
    pass

class NoSuchEdgeError(Exception):
    """ Kanten findes ikke """
    pass
```

**Implementation af fejlhåndtering:** Jeg ændrer de problematiske metoder, så de nu kaster en af ovenstående fejl, når en ulovlig handling forsøges.

Argumentet til `add_node` skal være en knude, som ikke ligger i grafen:

```
def add_node(node):
    # Fejlhåndtering
    if node in self.nodes:
        raise NodeAlreadyExistsError(node)
    # Tilføj knuden
    self.nodes.append(node)
```

Argumentet til `del_node` skal være en knude, som findes i grafen:

```
def def_node(node):
    # Fejlhåndtering
    if not node in self.nodes:
        raise NoSuchNodeError(node)
    # Slet knude
    self.nodes.remove(node)
```

**Smartere fejlhåndtering:** Jeg ser, at fejlhåndteringen tager følgende form:

1. Undersøg om en knude eller kant findes eller ej
2. Kast fejl hvis det ikke passer med forventningen

Ved at definere en generel metode for dette, kan jeg reducere antallet af linjer brugt på fejlhåndtering betydeligt. Jeg definerer en funktion for knude og en for kanter. Jeg vælger at kalde dem `ensure_nodes` og `ensure_edges`:

```
class Graph:
    ...

    def ensure_nodes(self, nodes, must_exist):
        """ Kast en fejl hvis en af knuderne findes/ikke findes """
        for node in nodes:
            exists = self.has_node(node)
            if must_exist and not exists:
                raise NoSuchNodeError(node)
            if not must_exist and exists:
                raise NodeAlreadyExistsError(node)

    def ensure_edges(self, edges, must_exist):
        """ Kast en fejl hvis en af kanterne findes/ikke findes """
        for node_a, node_b in edges:
            exists = self.has_edge(node_a, node_b)
            if must_exist and not exists:
                raise NoSuchEdgeError((node_a, node_b))
            if not must_exist and exists:
                raise EdgeAlreadyExistsError((node_a, node_b))
```

Jeg kan nu skrive fejlhåndteringen for `add_node` og `del_node` således:

```
def add_node(node):
    # Fejlhåndtering
    self.ensure_nodes([node], True)
    # Tilføj knuden
    self.nodes.append(node)

def def_node(node):
    # Fejlhåndtering
    self.ensure_nodes([node], False)
    # Slet knude
    self.nodes.remove(node)
```

Bemærk, at der kun bruges en enkelt linje til fejlhåndtering i den nye kode. Jeg fortsætter med at tilføje fejlhåndtering til de andre kritiske metoder:

Argumenterne til `add_edge` skal være to knuder, der findes i grafen (der må gerne findes en kant i forvejen):

```
class Graph:
    ...
    def add_edge(self, node_a, node_b):
        # Fejlhåndtering
        self.ensure_nodes( [node_a, node_b] , True )
        # Tilføj kant
        node_a.add_edge(node_b)
```

Argumenterne til `del_edge` skal være to knuder, der deler en kant:

```
class Graph:
    ...
    def del_edge(self, node_a, node_b):
        # Fejlhåndtering
        self.ensure_edges( [(node_a, node_b)], True )
        # Slet kant
        node_a.del_edge(node_b)
```

### 3.3 Endelig Implementation

```
# coding: utf-8

class NodeAlreadyExistsError(Exception):
    """ Knuden findes allerede """
    pass

class NoSuchNodeError(Exception):
    """ Knuden findes ikke """
    pass

class EdgeAlreadyExistsException(Exception):
    """ Kanten findes allerede """
    pass

class NoSuchEdgeError(Exception):
    """ Kanten findes ikke """
    pass

class Graph:
    def __init__(self):
        """ Tom graf, ingen knuder """
        self.nodes = []

    def add_node(self, node):
        """ Tilføj en knude """
```

```

        # Fejlhåndtering
        self.ensure_nodes( [node], False )
        # Tilføj knuden
        self.nodes.append(node)

def del_node(self, node):
    """ Slet en knude """
    # Fejlhåndtering
    self.ensure_nodes( [node], True )
    # Slet knuden
    self.nodes.remove(node)

def get_nodes(self):
    """ Hent en kopi af grafens knuder """
    return list(self.nodes)

def get_edges(self):
    """ Hent en kopi af kant-par """
    edges = []
    for node_a in self.get_nodes():
        for node_b in node_a.get_edges():
            # tilføj kant fra a til b
            edges.append( (node_a, node_b) )
    return edges

def add_edge(self, node_a, node_b):
    """ Tilføjer en kant fra a til b """
    # Fejlhåndtering
    self.ensure_nodes( [node_a, node_b] , True )
    self.ensure_edges( [(node_a, node_b)], False )
    # Tilføj kant
    node_a.add_edge(node_b)

def del_edge(self, node_a, node_b):
    """ Sletter kanten fra a til b """
    # Fejlhåndtering
    self.ensure_edges( [(node_a, node_b)], True )
    # Slet kant
    node_a.del_edge(node_b)

def has_node(self, node):
    """ Undersøger om node ligger i grafen """
    return (node in self.nodes)

def has_edge(self, node_a, node_b):
    """ Undersøger om en kant fra a til b findes """
    return node_a.has_edge(node_b)

def count_nodes(self):

```

```

        """ Tæller antallet af knuder """
        return len(self.nodes)

    def count_edges(self):
        """ Tæller antallet af kanter """
        count = 0
        for node_a in self.nodes:
            # Læg a's kanter til
            count += node_a.count_edges()
        return count

    def ensure_nodes(self, nodes, must_exist):
        """ Kast en fejl hvis en af knuderne findes/ikke findes """
        for node in nodes:
            # Findes knuden?
            exists = self.has_node(node)
            # Overholder den begrænsningen?
            if must_exist and not exists:
                raise NoSuchNodeError(node)
            if not must_exist and exists:
                raise NodeAlreadyExistsError(node)

    def ensure_edges(self, edges, must_exist):
        """ Kast en fejl hvis en af kanterne findes/ikke findes """
        for node_a, node_b in edges:
            # Findes knuden?
            exists = self.has_edge(node_a, node_b)
            # Overholder den begrænsningen?
            if must_exist and not exists:
                raise NoSuchEdgeError((node_a, node_b))
            if not must_exist and exists:
                raise EdgeAlreadyExistsError((node_a, node_b))

class DirectedGraph(Graph):
    """ Implementation af en orienteret graf """
    pass

class UndirectedGraph(Graph):
    """ Implementation af en ikke-orienteret graf """

    def add_edge(self, node0, node1):
        """ Tilføjer en kant til grafen """
        Graph.add_edge(self, node0, node1)
        Graph.add_edge(self, node1, node0)

    def del_edge(self, node0, node1):

```

```

        """ Fjerner en kant fra grafen """
        Graph.del_edge(self, node0, node1)
        Graph.del_edge(self, node1, node0)

    def count_edges(self):
        """ Tæller antallet af kanter i grafen """
        # Divider med to, da alle kanter ligger dobbelt
        # (en for hver retning)
        return Graph.count_edges(self) / 2

class Node:
    def __init__(self):
        """ Ny knude, ingen kanter """
        self.edges = []

    def add_edge(self, node):
        """ Tilføj kant """
        self.edges.append(node)

    def del_edge(self, node):
        """ Fjern kant """
        self.edges.remove(node)

    def has_edge(self, node):
        """ Test om en kant findes """
        return (node in self.edges)

    def count_edges(self):
        """ Tæl antallet af kanter """
        return len(self.edges)

```

### 3.4 Test og afprøvning