Håkon Fruseth Christiansen

Generering av enlinjeskjema

Masteroppgave i Teknisk kybernetikk
Veileder: Sverre Hendseth
Juni 2020

Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU
Kunnskap for en bedre verden

Håkon Fruseth Christiansen

Generering av enlinjeskjema

Masteroppgave i Teknisk kybernetikk
Veileder: Sverre Hendseth
Juni 2020

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Problembeskrivelse

Enlinjeskjema er en type blokkdiagram/skjematikk for å representere trefasesystemer. Man kan ha enlinjeskjema over et helt kraftnett, eller det kan være begrenset til innholdet av en kraftstasjon. Skjemaet er gjerne integrert i et dataprogram, og benyttes av nettselskapet for å hjelpe til med operasjon og vedlikehold av et kraftnett, samt planlegging av utbygging. Skjemaene er for det meste tegnet manuelt, som er en tidkrevende prosess. Å generere slike skjema automatisk vil redusere kostnader og manuelt arbeid.

I denne oppgaven skal det lages et dataprogram som genererer et enlinjeskjema av komponentene i en kraftstasjon. Det inkluderer transformatorer, brytere, samleskinner, laster og generatorer. Enlinjeskjemaene skal kunne brukes for å visualisere innholdet av en stasjon i en nettapplikasjon.

Sammendrag

I masteroppgaven er det lagd et dataprogram i Python som genererer enlinjeskjema av stasjoner i kraftnettet. Programmet leser inn data fra en XML-fil på CIM-standarden innenfor kraftnett, ved hjelp av biblioteket PyCIM. Komponentene inne i hver stasjon gjøres om til en NetworkX graf. I den prosesses gjøres også flere forenklinger, ved å fjerne uinteressante elementer fra grafen.

Det er implementert en algoritme som gir komponenter posisjon. Algoritmen lager en layout for hvert spenningsnivå i stasjonen uavhengig av de andre spenningene, før de slås sammen på slutten. Algoritmen deler inn komponentene i grener av sammenhengende komponenter. En gren posisjoneres i en vertikal kolonne. Grener som er koblet sammen plasseres ved siden av hverandre.

Det er også generert layout ved hjelp av grafvisualiseringsalgoritmen *dot*. Enlinjeskjemaene fra de to ulike layoutalgoritmene sammenlignes og kvaliteten blir vurdert. Begge algoritmene har sterke og svake sider, som gjør at ingen av algoritmene får gode resultater for alle typer stasjoner, men alle stasjoner får en god layout med minst en av algoritmene.

I oppgaven er det også vist hvordan enlinjeskjemaet kan tegnes som et interaktivt SVG-bilde, basert på den genererte layouten.

Programmet er testet på et datasett med 1155 stasjoner. Programmet har fortsatt mangler, som gjør at 29 av disse ikke blir tegnet.

Abstract

I this thesis it is written a computer program in Python, which generates one-line diagrams of substations in a power grid. The program reads data from an XML file, by using the library PyCIM. The XML file follows the CIM standard for power grids. For each substation in the grid, the components and their connections are used to create a NetworkX graph. During that process the graph is simplified by removing uninteresting elements, which are part of the CIM standard.

It is implemented an algorithm to give posisions to the components. The algorithm makes a layout for each voltage level within the substation independent of the others, before they are merged together. The algorithm divides the components into branches, which are positioned in a vertical line. Connected branches are positioned next to each other.

The graph visualization algorithm *dot* is also used to generate a layout. The resulting one-line diagram from both methods are compared and their quality are considered. Both algorithms have strong suits and weak suits, which means that neither algorithm can draw all kinds of substations successfully. However, all substations get a good layout from at least one of the algorithms.

The layout is used to draw an one-line diagram as an interactive SVG image.

The program is tested on a dataset with 1155 substations. The program still has some flaws, which causes 29 of the substations to not be drawn.

Forord

Jeg vil rette en stor takk til veilederene mine Sverre Hendseth (Institutt for teknisk kybernetikk) og Andreas Brandsøy Våg (Kongsberg Digital Utilities), for hjelp med rapportskriving og interessante diskusjoner. Jeg vil også takke resten av kollegaene på KDI for et godt arbeidsmiljø og inspirasjon.

Håkon Fruseth Christiansen, 01.06.2020

Innholdsfortegnelse

1 Introduksjon

Oppgaven er gitt av Kongsberg Digital Utilities. Kongsberg Digital utvikler datasystemer som skal hjelpe nettselskaper å overvåke og styre kraftnettene sine. Det er flere utfordringer ved å sette opp datagrunnlaget som datasystemene skal benytte. Denne jobben må per i dag gjøres manuelt. Et program som kan generere en layout for enlinjeskjema hadde vært av stor nytte.

Det i hovedsak fire ulike bruksområder hvor layout og tegning av enlinjeskjema er av nytte:

1. Legge ut innholdet i en kraftstasjon i en geografisk fil.
2. Lage enlinjeskjema av stasjon i SVG-format.
3. Lage layout for et analyseprogram.
4. Generere en layout for data som ikke har geografiske posisjoner på kraftlinjer.

Et av datasystemene til Kongsberg Digital er en nettapplikasjon som er kartbasert. Det vil si at det benytter geografiske data for å vise fram de elektriske komponentene. Nettselskapene har geografiske koordinater på stasjoner og kraftlinjer, men de elektriske komponentene inne i en stasjon har ikke egne geografiske koordinater. Hvis komponentene inne i en stasjon skal få geografiske posisjoner, må endepunktene på kraftlinjene flyttes, og stasjonen må gjøres om til å ha en utstrekning som dekker alle komponentene. Derfor vil det å bruke de eksakte posisjonene direkte, likevel presentere en stor manuell jobb. Dette er ofte det som gjøres i dag når en geografisk representasjon av kraftnettet er ønsket.

Et alternativ til å legge ut en stasjon geografisk, er å få opp et enlinjeskjema over stasjonen når den trykkes på i nettapplikasjonen. Da vil altså stasjonen være et punkt i kartet. Denne løsningen gjør det mulig å bruke de eksakte posisjonene på stasjoner og kraftlinjer. Det er ikke lenger mulig å studere hele nettet i GIS-programvare som ArcGIS.

Kongsberg har et analyse/simuleringverktøy kalt K-Spice. K-Spice har mulighet for å lage grafikk av et kraftnett, men ingen måte å automatisk generere denne grafikken. For større nett vil det være en stor jobb å tegne grafikken. Det er noen muligheter for å generere en elektrisk modell av et kraftnett. Simuleringer kan gjøres uten grafikk, men grafikk gjør det betydelig enklere å utforske nettet. Grafikk i K-Spice er definert i et XML-format.

Det fjerde bruksområdet er relevant når man benytter anonymiserte data. Det vil si data som har fått fjernet alt som kan binde det til et sted og nettselskap.

I hovedsak er det navn på stasjoner og kraftlinjer, og posisjoner. Alt dette kan fjernes uten at selve kraftnettet påvirkes, det vil fortsatt være mulig å gjøre analyser på nettet.

I denne oppgaven er hovedfokuset på bruksområde nummer to, men layouten vil være nyttig for alle.

2 Kraftnett

Kraftnett transporterer elektrisitet fra produsent til forbruker. I Norge er kraftnettet delt opp i tre deler: transmisjonsnett, regionalnett og distribusjonsnett.

Transmisjonsnettet er motorveiene for strøm. Det kobler sammen regionalnettene. De aller største kraftverkene og kundene er koblet direkte på dette nettet. I tillegg har det forbindelser til utenlandske transmisjonsnett. Det har typisk spenning på mellom 320 kV og 400 kV, men noen steder er den 132 kV. Transmisjonsnettet er operert av Statnett SF.

Regionalnett er bindeleddet mellom transmisjonsnettet og distribusjonsnettet. Større kunder og kraftverk er koblet til dette nettet. Spenningen er mellom 33 kV og 132 kV.

Distribusjonsnett leverer elektrisitet til vanlige forbrukere. I nyere tid har også distribuert kraftproduksjon blitt mer populært. Det er plusskunder som har installert for eksempel solceller på taket og i perioder kan mate elektrisitet inn på nettet. Distribusjonsnettet benytter spenninger fra 230 V til 22 kV. Ofte er det delt inn i høyspent og lavspent distribusjonsnett, der grensen går på 1 kV. Regionalnett og distribusjonsnett eies av nettselskap. [1]

2.1 Oppbygning

Kraftnettet er bygd som et trefasesystem. Trefase er en metode for å overføre elektrisk energi i vekselstrømsystemer. Det benyttes tre eller fire ledere. Tre ledere vil lede strøm, der fasen på spenningene er 120° forskjøvet i forhold til hverande. Den siste lederen, nullederen, skal helst ikke lede strøm, men det er sjeldent tilfelle. Med tre faser kan nettet levere tre ganger så mye effekt som et enfasesystem, for enten en eller to flere ledere. Det blir altså billigere å bygge ut i forhold til kapasiteten man får. [2]

Kraftnettet er bygd opp av flere ulike komponenter.

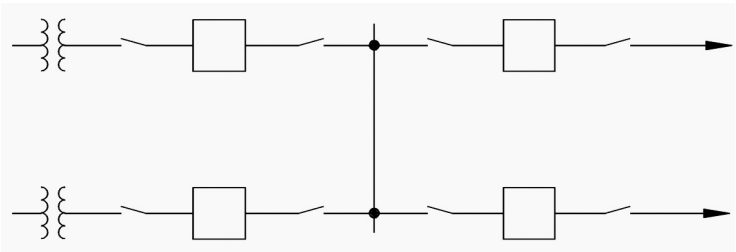
- Kabler og linjer: Kabler og linjer leder strøm mellom komponenter på ulike steder. Kabler går under bakken, mens linjer går i lufta.
- Transformatorer: Omformer elektrisitet fra en spenning til en eller flere andre spenninger. Det vanligste er to eller tre ulike spenninger.
- Laster og generatorer: Laster er alt som er koblet til nettet som forbruker energi, mens generatorer sender energi til nettet. Generatorer har tradisjonelt vært større kraftverk, men nå er det i økende grad også distribuert kraftproduksjon.

- Samleskinner (busser): Der flere ledere møtes, kobles kablene til en felles samleskinne, som leder strøm mellom alle linjene.
- Brytere og sikringer: Brytere og sikringer kan bryte strømmen mellom to ledere. En sikring er en sikringsmekanisme som slår ut ved for høy strøm, akkurat som i hjemmet[3]. En bryter kan slås av og på, og brukes for å styre det elektriske nettet. Ved å skru av og på brytere kan man få strømmen til å gå i andre linjer, og hindre at utsatte linjer blir overbelastet. Bryterene er derfor veldig viktig i den daglige driften av nettet. Det er mange ulike typer bryter, som fyller litt ulike funksjoner og har ulik konstruksjon. En spesiell type bryter er isolatorer. Disse er brukt for å isolere en del av en krets. Den skal ikke åpnes mens det går strøm gjennom den. [4]
- Shunt/Jumper: En leder som kobles inn for å lede strømmen utenom en annen komponent.

Alle komponenter utenom kabler, linjer og laster er som oftest samlet i en stasjon. Inne i en stasjonen er det flere vanlige måter å organisere komponentene. Måten å organisere det på kalles busskjema eller busskonfigurasjon. De har ulik redundans og mengde komponenter som brukes.

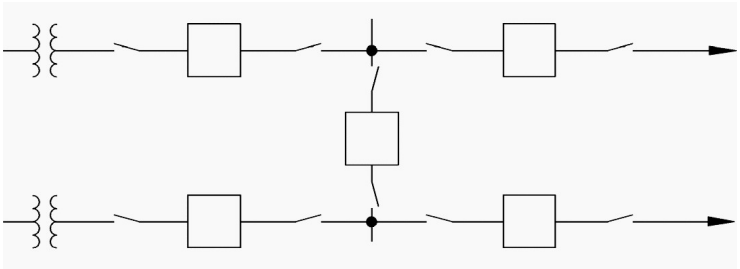
- Enkel buss
- Delt enkel buss
- "Main bus and transfer bus"
- Ringbuss
- "Breaker-and-a-half"
- Dobbel-buss dobbel-bryter ("Double breaker double bus")

I enkel buss konfigurasjonen er alle kretser koblet på samme buss. Det er den billigste konfigurasjonen, men gir ingen beskyttelse mot feil.[5]



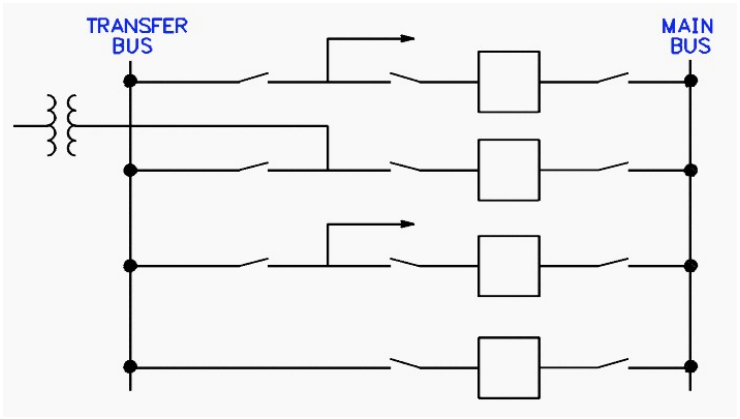
Figur 1: Typisk enlinjeskjema av enkel buss konfigurasjon. Hentet fra [5]

I delt enkel buss er bussen delt med en bryter. Hvis det er feil på den ene siden av bryteren kan den åpnes, så vil den andre siden fortsatt kunne fungere. [5]



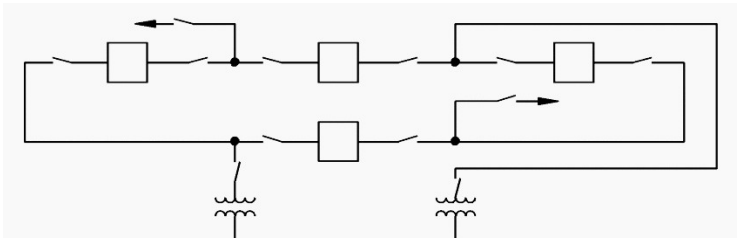
Figur 2: Typisk enlinjeskjema av delt enkelt buss konfigurasjon. Hentet fra [5]

"Main bus and transfer bus" benytter to busser. Alle kretser er koblet til begge bussene, men har kun bryter mot hovedbussen. Bussene er adskilt med en bryter og isolatorer. Normalt brukes kun hovedbussen, men ved vedlikehold av en bryter kan den ekstra bussen kobles inn. Det er altså mulig med vedlikehold av brytere uten å avbryte service.[5]



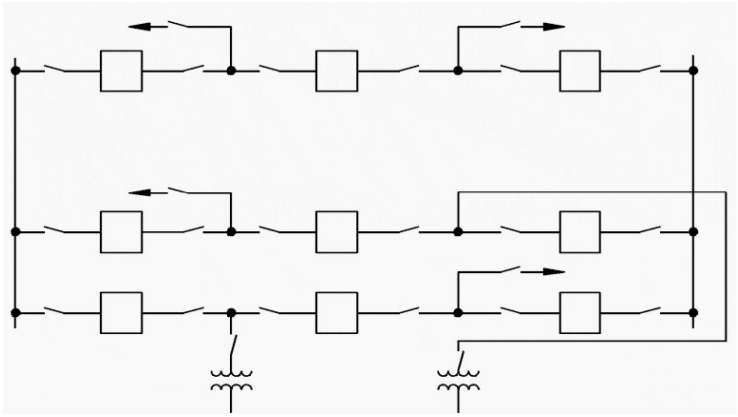
Figur 3: Typisk enlinjeskjema av "Main bus and transfer bus" konfigurasjon. Hentet fra [5]

I ringbuss konfigurasjon er kretsene koblet sammen i en ring. Det gjør det mulig å koble ut én bryter for å vedlikeholde denne. Jo flere kretser som er koblet sammen på denne måten, jo større er faren for feil som ikke kan takles. Derfor er det sjelden mer en seks kretser koblet sammen i en ringbuss. [5]



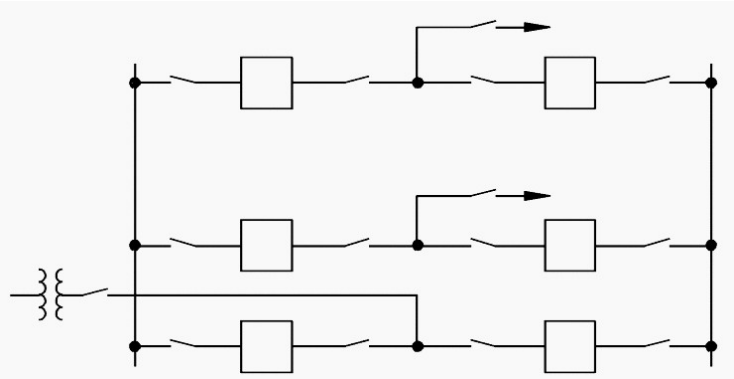
Figur 4: Typisk enlinjeskjema av ringbuss konfigurasjon. Hentet fra [5]

"Breaker-and-a-half" bruker to busser, som normalt begge er i bruk. Bussene kobles sammen med tre brytere i serie, og mellom bryterene er det koblet til en krets. Det gjør at kretsene har en og en halv bryter hver, derav navnet. Her kan en buss eller en bryter isoleres for vedlikehold uten at service må avbrytes. [5]



Figur 5: Typisk enlinjeskjema av "Breaker-and-a-half" konfigurasjon. Hentet fra [5]

"Double bus double breaker" er konfigurasjonen med mest redundans. Det er to busser, og alle kretser er koblet til begge bussene med en bryter mellom. Denne konfigurasjonen kan den også enkelt brukes for å endre hvilke kretser som er koblet sammen, ved å endre hvilken buss de er koblet på. [5]



Figur 6: Typisk enlinjeskjema av "Double bus double breaker" konfigurasjon. Hentet fra [5]

Det er vanlig å kategorisere en stasjon som enten transformatorstasjon, nettstasjon eller kabelskap. Transformatorstasjoner er de store stasjonene som kobler et regionalnett til et distribusjonsnett. De har mange tilkoblinger, flere transformatorer og kompliserte busskonfigurasjoner. En nettstasjon er en mindre stasjon som transformerer spenning fra høyspent distribusjonsnett til lavspent distribusjonsnett. De har ofte enklere busskonfigurasjoner. Kabelskap er små stasjoner som kobler sammen flere lavspente kretser. De har ingen transformatorer, og ofte kun en samleskinne.

2.2 Lagring av informasjon

For å sikre et effektivt kraftmarked og tilfredsstillende leveringskvalitet er Statnett SF utpekt til systemansvarlig. De er ansvarlig for at det hele tiden er balanse mellom forbruk og produksjon av energi og at frekvensen er 50 Hz [6]. Da må Statnett vite en del om nettet til alle nettselskapene i Norge. Den informasjonen rapporteres inn av nettselskapene via webløsningen Fosweb [7]. Informasjonen de krever er geografiske koordinater på stasjoner og alle elektriske komponenter over 1000 volt. Avhengig av type komponent må elektriske komponenter rapporteres inn med verdier for blant annet driftspenning og strømgrense. Linjer og kabler er koblet til to stasjoner. De kan deles inn i flere segmenter, og kan inkludere geografisk informasjon. De andre elektriske komponentene er samlet under et *felt* som igjen er underlagt en stasjon. Feltet viser hvilke komponenter som er koblet sammen.

Nettselskapene må altså selv skaffe og lagre dokumentasjon som er påkrevd av Statnett. Nettselskap kan ha lagret mer informasjon enn det som er påkrevd av Statnett, for eksempel koordinater på lavspent distribusjonsnett. Geografisk informasjon lagres enten som lokale filer eller i databaser. De fleste filer og

databaser kan åpnes i GIS-programvare[1] som QGIS og ArcGIS. Det er flere ulike filtyper som kan brukes. Den norske standarden for utveksling av digitale kartdata er SOSI (Samordnet Opplegg for Stedfestet Informasjon). Et annet format som brukes av nettselskaper i Norge er Shape. Den består egentlig av flere filer, hvorav tre er obligatoriske. De obligatoriske filene er .shp, .shx og .dhb, som lagrer henholdsvis selve geometrien, et indekseringssystem og attributter for objektene [8]. GeoJSON er et format basert på JSON. Det er et åpent og lettvekt format, der alle objektene kan lagres i en fil med geometri og attributter [9]. Statnett aksepterer geografiske data som Shape eller SOSI [6]. Det finnes også mange ulike databaser å velge blant. PostgreSQL med utvidelsen PostGIS er ofte brukt.

Det er få felles standarder for å lagre informasjonen. Historisk sett har ikke informasjonen vært spesielt nyttig i driften av nettet, og det har ikke vært nødvendig å dele informasjonen. Det er i endring nå, siden nettene er mer sammenkoblet og i raskere endring enn før. Det kreves også bedre utnyttelse av nettet for å unngå dyre utbygginger mange steder i Norge. Ulike nettselskaper har gjerne lagret informasjonen på ulike måter, i forhold til hvilke elektriske komponenter som er lagret, hvilke attributter og egenskaper som er lagret for hvert objekt, og hvordan det er lagret (filformat eller database).

2.3 Datasystemer for kraftnett

For å hjelpe nettselskaper med å lagre data om kraftnett, samt drifte og vedlikeholde det, er det laget flere datasystemer. Disse kan deles inn i kategorier ettersom hvilket formål de tjener:

- SCADA står for Supervisory Control and Data Aqcuistion. Innenfor kraftnett dekker det datasystemer som brukes til å overvåke og styre høyspentnettet. Disse systemene er veletablerte. [10]
- DMS står for Distribution Management Systems. Det er datasystemer som brukes for å overvåke og styre distribusjonsnettet. De fungerer som en beslutningsstøtte for operatørene. I motsetning til SCADA er det ikke veletablerte systemer. [11]
- NIS står for Network Information Systems. Det er systemer for å lagre og administrere alle relevante data om et kraftnett. NIS bygges gjerne på geografiske informasjonssystemer (GIS). [12]

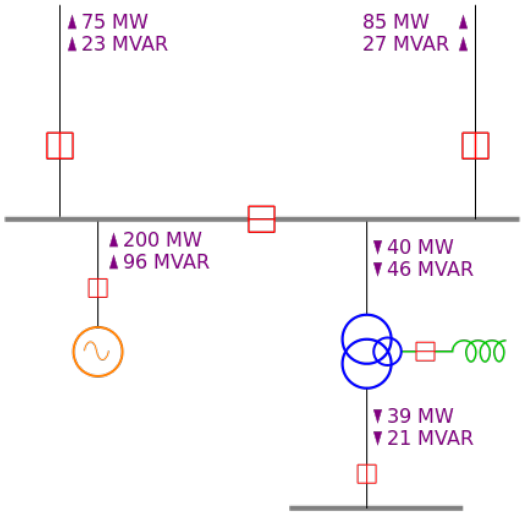
[10]

[1] GIS – geografiske informasjonssystemer

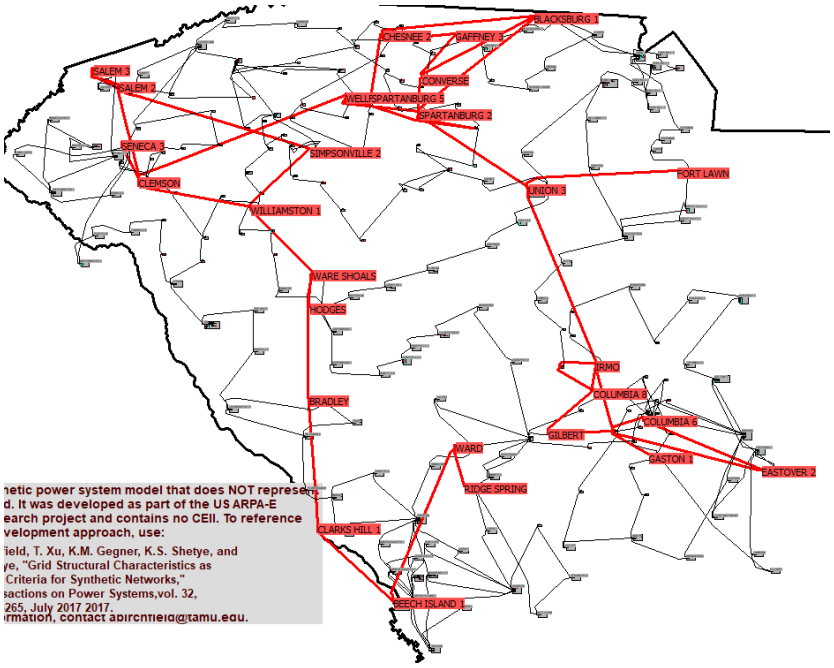
NETBAS er et dataprogram utviklet av Powel. Det er et NIS-system. Mange norske nettselskap bruker det i dag. Det tilbyr flere tjenester, blant annet dokumentering, visning av nett både skjematisk og geografisk, og analyser av nettet. [13]

3 Enlinjeskjema

Enlinjeskjema (engelsk: one-line diagram eller single line diagram) er en form for blokkdiagram ment å gi en forenklet visning av trefasesystemer. Enlinjeskjema er ingen entydig definert standard. Alle typer diagram hvor en trefaselinje er forenklet til å vise en linje, kan kalles enlinjeskjema. Et enlinjeskjema kan være av en kraftstasjon med komponentene inne i den og hvordan disse er koblet sammen, slik som det er vist i Figur 7. Den type diagram kan brukes når nettselskapet gjør vedlikehold av stasjonen. Det er også mulig å vise flere stasjoner og hvordan de er koblet sammen i et større enlinjeskjema (Figur 8). Da kan enlinjeskjemaet dekke hele kraftnettet til et nettselskap, eller en bestemt del av det.



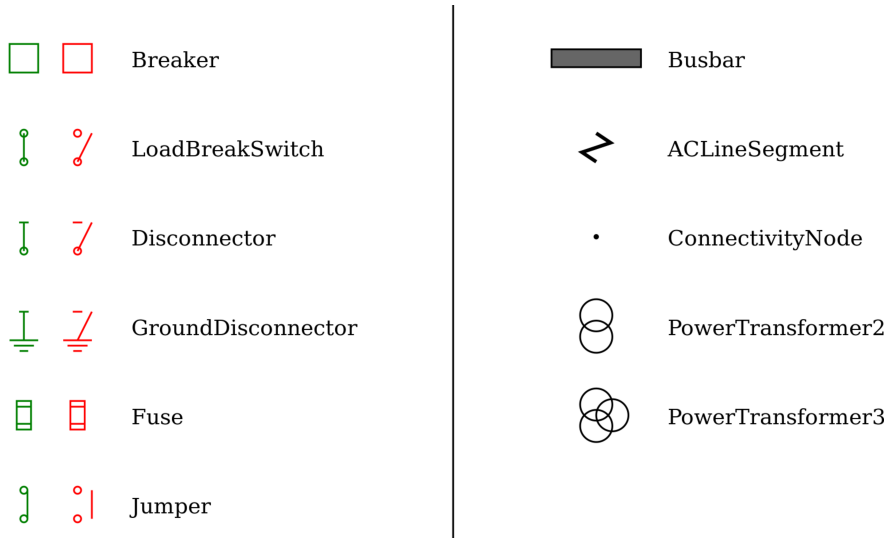
Figur 7: Enlinjeskjema av komponentene i en stasjon. By BillC - Own work, CC BY 3.0, `https://commons.wikimedia.org/w/index.php?curid=3450745`



Figur 8: Enlinjeskjema av et helt nett. Skjermbilde av et syntetisk nett vist i applikasjonen PowerWorld Viewer

Koblingsbildet (hvordan bryterene er konfigurert) bør vises, slik at det kan brukes av operatører på nettsentralen til å styre nettet. Brytere kan for eksempel ha ulike farger eller ulike symboler avhengig av posisjonen de er i.

Det er ingen standard for symbolene som brukes i et enlinjeskjema. Et forslag til symboler for flere av komponentene kan bli funnet i [14]. Symbolet for transformatorer er hentet derfra. I [15], [4] og [3] er det flere varianter av de ulike typen brytere. Symbolene brukt i denne oppgaven er vist i Figur 9. Til venstre i figuren er symbolene som kan være åpne eller lukket, og hver tilstand har sitt eget symbol. Siden ConnectivityNode ikke representerer en faktisk komponent, er den et lite symbol.



Figur 9: Symbolene som brukes i denne oppgaven

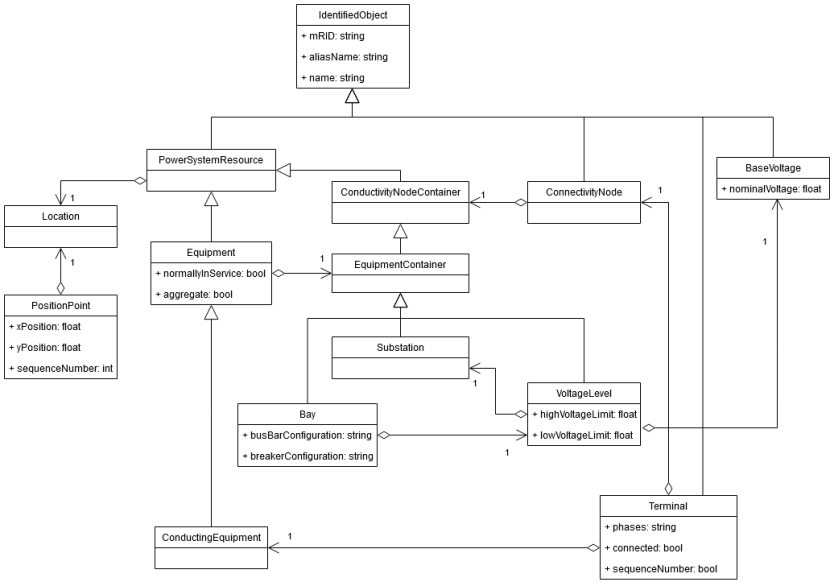
4 CIM - Common Information Model

4.1 Generelt om CIM

CIM er en standard for å lagre informasjon om elektriske nett. Den ble laget for å legge til rette for å dele informasjon mellom nettselskap og innad i selskapet. CIM består av IEC standardene IEC 61970-301 og IEC 61968-11. IEC 61970-301 har klasser for elektriske komponenter og hvordan de er koblet sammen. Den inneholder alt som trengs for å kunne bygge opp selve kraftnettet og kjøre analyser på det. IEC 61968-11 har klasser som beskriver andre sider av kraftnettet, slik som informasjon om kunder, operatører og geografisk plassering.

Den er gitt ut i flere versjoner, den nyeste er CIM16. CIM16 definerer en rekke nye klasser og variabler som ikke er i CIM15. CIM14 til CIM15 har flere endringer i hvordan noen klasser er bygd opp, spesielt PowerTransformer.

CIM er beskrevet som flere UML klassediagram. Klassene er bygd opp slik at de arver fra hverandre. For eksempel vil alle komponenter som fører elektrisk strøm arve fra klassen ConductingEquipment. De mest vanlige klassene fra CIM-standarden er vist i Figur 10. Av klassene vist her er det kun Location og PositionPoint som er i IEC 61968, resten er i IEC 61970.



Figur 10: Klassediagram av de mest brukte klassene i CIM

Elektriske komponenter er koblet sammen med terminaler og konnektivitetsnoder. En terminal er grensesnittet mellom én komponent og én konnektivitets-

node. Konnektivitetsnodene er koblet til minst en terminal. Det virker kanskje tungvint å ha så mange mellomledd for å beskrive hva som er koblet sammen, men det er nødvendig for å hindre mange-til-mange forhold som ikke lar seg implementere på en enkel måte i XML.

De ulike klassene fra CIM som brukes mye i denne oppgaven er forklart i den følgende listen.

- BaseVoltage er en klasse som lagrer en spenning som et tall. Mange klasser har en referanse til en instans av denne klassen.
- EquipmentContainer er et foreldreklasse som samler komponenter. Det er flere klasser som arver fra denne. Det er kun klassene som arver som er i bruk.
  - Substation lagrer innholdet i en stasjon. I tillegg til å ha komponentene som ikke er knyttet til en spenning, kan den inneholde flere VoltageLevel (en VoltageLevel vil ha referanse til en Substation).
  - VoltageLevel samler komponenter som er knyttet til en spenning. VoltageLevel kan inneholde flere Bay, og da vil ikke komponentene ha referanse til VoltageLevel men til Bay. VoltageLevel har en referanse til en BaseVoltage-instans, som sier hvilken spenning det er.
  - Bay er en del av et spenningsnivå med komponenter som er koblet sammen. En Bay har en referanse til en VoltageLevel.
  - Line kan brukes for å samle flere ACLineSegments som går i serie eller parallell mellom samme sted. Ikke brukt i dataen i denne oppgaven.
- ConductingEquipment er en foreldreklasse for alle klassene som leder strøm. Alle klasser som arver av denne har referanser fra terminaler. De kan også ha referanse til en BaseVoltage.
  - PowerTransformer er klassen for transformatorer. PowerTransformer har minst to PowerTransformerEnd, som lagrer endene i transformatoren. Terminalene i PowerTransformer matcher med sekvensnummeret til PowerTransformerEnd.
  - ACLineSegment representerer alle typer linjer og kabler. Den har attributter for lengde, resistans og kapasitans.
  - Switch er en foreldreklasse for alt utstyr som kan være koblet ut eller inn. Den har en variabel for om den er normalt koblet inn eller ikke. Klasser som arver fra denne er: Breaker, LoadBreakSwitch, Disconnector, GroundDisconnector, Fuse og Jumper. Det er forskjeller i hvordan disse klassene kan opereres. LoadBreakSwitch kan slås inn og ut under normale tilstander, men ikke hvis det er en kortslutning som gjør at det det går stor strøm gjennom den. Disconnector og GroundDisconnector skal kun opereres når det går neglisjerbar

strøm gjennom den. GroundDisconnector er en isolator mot jord. En Breaker kan slås ut og inn i normale situasjoner og under feil. Noen typer Breaker kan også fungere som isolator. [15]. Fuse er en sikring. I dataen som brukes i denne oppgaven er Jumper brukt for å representere en samleskinne som går mellom to komponenter.

  - BusbarSection representerer en samleskinne. Den har kun en terminal, som igjen er koblet til en konnektivitetsnode. Alt som er koblet til den tilhørende konnektivitetsnoden er koblet sammen på denne samleskinna.

- Terminal har referanse til en ConductingEquipment, og en referanse til en ConnectivityNode
- ConnectivityNode har ingen variabler lagret i seg, men blir referert til fra terminaler. Den kan ha et vilkårlig antall referanser til seg.

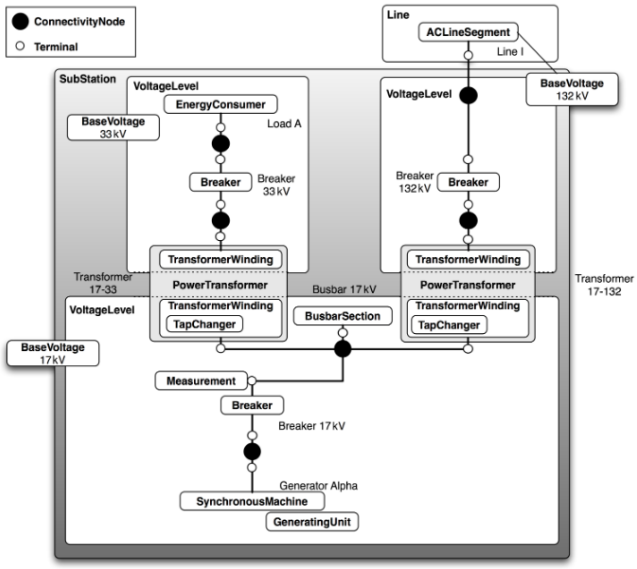
Et eksempel på hvordan en transformatorstasjon vil se ut på CIM standarden er vist i Figur 11. Denne figuren er fra en eldre CIM-versjon. Den eneste forskjellen til CIM16 er at transformatoren er bygd opp er annerledes. Man kan se hvordan EquipmentContainerene Substation og VoltageLevel inneholder komponenter. PowerTransformer ligger under Substation, alt annet ligger under en VoltageLevel. Legg spesielt merke til hvordan BusbarSection er lagt. I virkeligheten er BusbarSection, eller samleskinne på norsk, en komponent som kobler sammen flere andre komponenter. I dette eksempelet er begge 17kV endene på transformatorene og bryteren koblet til samleskinna. [16]

4.2 XML implementasjon av CIM

Det er definert en XML implementasjon av CIM standarden. I cimxml-filene er relasjoner lagret kun i et av objektene. Siden det er komplisert å lagre lister i xml-filer, blir relasjonen lagret i objektet som kun har en relasjon. For eksempel vil et PositionPoint objekt ha en referanse til et Location objekt, men Location objektet vil ikke ha en referanse til PositionPoint objektet. Det kan være flere PositionPoint objekter som har referanse til samme Location. Slik blir mange-til-en relasjoner lagret.

CIM standarden er veldig generell, noe som byr på visse problemer når den skal brukes i praksis. Mange av klassene som er definert blir ikke nødvendigvis brukt. Det er også sjeldent at alle verdiene for hver klasse blir brukt. Det å ta høyde for alle mulige implementasjoner av CIM er en alt for stor oppgave. Det kreves derfor en felles implementasjon av CIM-eksport og -import.

Data som nettselskap har er ikke lagret som CIM. Derfor kreves det en eksportjobb fra de eksisterende datalagrene til en cimxml fil. De norske nettsel-



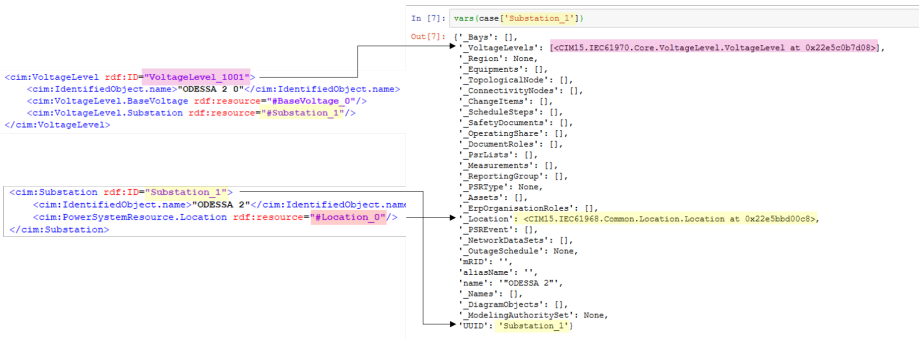
Figur 11: CIM representasjon av en stasjon. Hentet fra [16]

skapene som Kongsberg Digital jobber med har lagd en slik eksport, men den er fortsatt under utvikling.

4.3 PyCIM

PyCIM er et Python bibliotek for å lese og skrive cimxml filer. Det har støtte for versjonene CIM14 og CIM15, men ikke den nyeste versjonen CIM16. PyCIM kan fortsatt lese filer av CIM16 versjonen, men noen verdier som ble introdusert i CIM16 blir ignorert. PyCIM definerer klasser for alt som er definert i CIM-standarden. Når den leser en cimxml-fil, lagres alt som objekter i en Python dictionary. Figur 12 viser hvordan en instans av Substation ser ut. Den har et spenningsnivå og en lokasjon som referanser til andre objekt. [17]

PyCIM biblioteket har dårlig ytelse når det kommer til større cimxml-filer. Det er først og fremst fordi PyCIM er skrevet i ren Python, og Python er et tregt språk. I tillegg blir alle referansene som er lagret kun på et av objektene i xml-filen, blir lagret til begge instansene i Python. Det er løst ved å iterere gjennom cimxml-filen to ganger. Først blir alle instansene initialisert til tomme objekter av sin klasse, og lagret til en dictionary med sin unike id som nøkkel. Når PyCIM tar sin andre iterasjon gjennom filen vil alle referanser og attributter legges til instansene.



Figur 12: Hvordan objektene i cimxml-filen korresponderer med instanser i PyCIM

5 SVG - Scaleable Vector Graphics

5.1 Introduksjon om SVG

SVG er et bildeformat basert på xml. Som navnet tilsier er det vektorgrafikk, der grafikken er definert som vektorer av geometriske figurer. Dette er i motsetning til punktgrafikk, som er et rutenett av piksler. SVG holder seg skarpe selv når det blir zoomet mye inn. Oftest bruker vektorgrafikk mindre lagringsplass også [18].

De fleste nettlesere kan vise fram SVG. Et problem med å vise SVG i en nettleser er mangel på pan og zoom. Spesielt på større bilder er det en ulempe, siden mye av bildet kan havne utenfor skjermen.

Det er mulig å farge geometriske figurer, både med én farge, men også mer avanserte metoder som lineære og radielle gradienter. Det er også mulig å legge til filter på figurer, slik som blur og ulike farge-konverteringer. SVG kan også bli gitt stiler av CSS, akkurat som andre elementer i et HTML-dokument. Det støtter til og med interaksjoner som `:hover`, som gjør det mulig å endre farger eller vise tekst hvis man holder over et objekt.

5.2 svgwrite

Python biblioteket *svgwrite* kan brukes for å skrive SVG-filer. *svgwrite* benytter seg av biblioteket xml.etree.ElementTree for å skrive SVG-filen. Hvis man lærer seg de ulike objektene en SVG-fil består av, kan man lage SVG-filen selv via for eksempel xml.etree.ElementTree eller det oftest raskere lxml.etree biblioteket. [19]

svgwrite har en overordnet klasse `Drawing` som lagrer hele bildet. `Drawing` har en metode `add()` som legger til andre elementer. For å lagre til fil kalles `Drawing.save()`. Når den lagrer til fil lagres alle elementene som har blitt lagt til i `Drawing` i filen.

Det er mange elementer som kan legges til `Drawing`. De som brukes i denne oppgaven er `Style`, `Script`, `Group`, tekst og de ulike geometriske formene. Ellers kan man legge til alt som er definert i SVG-standarden, slik som gradienter og filtere. `Style` tar en tekststreng og lager et CSS-element som kan legges til i SVG-filen. Hvis man vil legge til en ekstern CSS-fil må man bruke funksjonen `Drawing.add_Stylesheet(href, title)`. `Script` legger enten til en tekststreng som et skript innebygd i SVG-filen, eller er referanse til et eksternt skript. `Group` lager en gruppe som man kan legge nye SVG-elementer i. Det er en måte å samle og strukturere elementer, som er mye brukt i dette programmet.

6 Grafer og grafvisualisering

En graf er en abstrakt datastruktur bestående av *noder* og *relasjoner*. Nodene er punkter, og en relasjon binder sammen to noder. En relasjon kalles ofte en kant. [20]

Siden elektriske nett har en grafstruktur kan metodene som er utviklet for å tegne grafer også tegne elektriske nett.

Blant de vanligste metodene finner man:

- Kraft-baserte layout: Minimerer krefter på nodene. Det er krefter som dytter nodene fra hverandre som ligner på elektrisk frastøting, og krefter som trekker de sammen som ligner på fjærkraft slik som definert av Hookes lov. Algoritmen er iterativ. For hver iterasjon beregnes kreftene på alle nodene. Noden blir flyttet basert på summen av kreftene på noden. [21]
- Spektral layout: Benytter egenvektorer for å plassere ut nodene.
- Sirkulær layout: Plasserer nodene sirkulært, gjerne med like mellomrom. Den egner seg for å vise nettverk med stjerne- eller ringtopologi.
- Hierarkisk layout: Kalles også lagvis layout. Grafen blir lagt ut i en trestruktur med nodene i horisontale lag kalt rang. Nodene har et hierarki, slik at pilene i diagrammet går i en retning. Det krever en rettet graf for å lage hierarkiet. I tillegg til å gi nodene et horisontalt lag, blir nodene på hvert lag organisert slik at det blir minst mulig kryssing av kanter. [22]

[23]

Innholdet i de fleste stasjoner har et naturlig hierarki, for eksempel fra høyeste spenning til laveste spenning. Derfor er hierarkisk layout den mest lovende for å visualisere komponentene i en stasjon.

Kraft-basert layout kan derimot være nyttig for å visualisere nettet som en helhet, ved å benytte stasjonene som noder og ikke tegne selve komponentene.

6.1 Dot-algoritmen

En av de mest kjente og brukte algoritmene for hierarkiske layout er utviklet av Graphviz. Den kalles dot-algoritmen. Algoritmen krever en rettet graf og fungerer i fire trinn som bruker heuristikk for å gi brukbare resultater kjapt.

Grafen er spesifisert i et språk som også kalles *dot*. Det er et tekstformat som kan endres på manuelt via en teksteditor. Det er lagd mange biblioteker

for ulike programmeringsspråk som fungerer som et grensesnitt mot Graphviz-programvaren. I dot-filen er alle nodene og kantene i grafen skrevet inn. Hver node og hver kant kan ha en rekke med attributter som er definert i `https://www.graphviz.org/doc/info/attrs.html`. Mange av attributtene er knyttet til stil og form på noder og kanter. Man kan blant annet endre farger, pilen på slutten av en linje og formen på en node. Noen av attributtene skrives på starten av filen og gjelder for hele grafen. Noen eksempler på det er retningen grafen skal tegnes i (ovenfra og ned, venstre til høyre) og minste avstand mellom noder, enten på ulik rang eller innenfor samme rang. [24]

Selve algoritmen er lagd med hensyn på fire astetiske kriterier:

1. Vise fram hierarkisk struktur i grafen.
2. Unngå visuelle anomalier som gjør grafen mindre forståelig, slik som kryssende linjer og skarpe kanter.
3. Hold kantene korte. Da blir det enklere å finne noder som henger sammen.
4. Symmetri og balanse.

Algoritmen er delt opp i fire steg som kjøres sekvensielt.

1. Gi nodene en optimal rank som holder kantene korte.
2. Organiser noden innenfor en rank for å redusere krysninger av kanter.
3. Gi nodene posisjon.
4. Lage kanter mellom nodene.

For å finne rank til nodene løses et optimaliseringsproblem som minimerer lengden på kantene, samtidig som kantene ikke er kortere enn den minimale lengden som er tillatt.

$$\begin{aligned} &min \sum_{(v,w) \text{ in } E} \omega(v,w)(\lambda(w) - \lambda(v)) \\ &\text{subject to } \lambda(w) - \lambda(v) \geq \delta(v,w) \end{aligned}$$

$\omega(v,w)$ er vekten av kanten mellom nodene v og w. $\lambda(v)$ er ranken til node v, så $\lambda(w) - \lambda(v)$ er vertikal avstand mellom v og w. $\delta(v,w)$ er kravet til forskjell i rank mellom de to nodene. Den er normalt 1, men kan være alle heltall større enn eller lik 0. Problemet løses med en nettverk simplex algoritme.

Før andre steget kjøres lages det virtuelle noder slik at ingen kanter "hopper over" en rank. Andre steget løses ved å gi en startorder til nodene, deretter

iterere et bestemt antall ganger gjennom to funksjoner. Den første funksjonen er en vektet median funksjon. Den itererer gjennom alle rankene, annenhver gang ovenfra og nedenfra. For hver node på hver rank kalkuleres medianen av nodene den har kanter til, på forrige rank. Nodene sorteres etter denne medianen. Den andre funksjonen itererer gjennom nodene på hver rank i par av nabonoder. For hvert par sjekker den om det blir færre krysninger av kanter hvis disse to nodene bytter plass. Denne prosessen gjentas til det ikke lenger fører til forbedringer.

Det tredje steget er å gi nodene faktiske posisjoner. Først gis alle nodene X-koordinat, deretter Y-koordinat. Y-koordinatet er enkelt, det er bare å sørge for at ingen noder er nærmere hverandre enn en bestemt minsteverdi `ranksep(G)`. Alle noder på samme rank får samme Y-koordinat.

$$min \sum_{e=(v,w)} \Omega(e)\omega(e)|x_w - x_v|$$
$$\text{subject to } x_b - x_a \geq \frac{xsize(a) - xsize(b)}{2} + nodesep(G)$$

$\Omega(e)$ er en intern parameter som er avhengig av om kanten kobler sammen ingen, en eller to virtuelle noder. Virtuelle noder indikerer en lengre kant, og disse blir prioritert å holde korte. $\Omega(e)$ vil altså være større for en kant mellom to virtuelle noder enn for en kant mellom to ekte noder.

Problemet kan løses som et lineært program ved å bruke kunstige variabler og ulikheter for å erstatte absoluttverdien. Å løse problemet slik tar mye tid for større grafer. Derfor bruker dot-algoritmen en heuristikk som stort sett gir en god layout (men ikke alltid).

X-koordinatene initialiseres basert på orderen fra forrige steg. Nodene blir pakket mot venstre så tett det lar seg gjøre uten å bryte den minimale avstanden mellom noder. Deretter gjøres 8 iterasjoner av en rekke heuristikker. De er vanskelige å programmere, og vanskelig å tilpasse for å gi bedre resultater.

Til slutt skal kantene mellom noder tegnes. Dot algoritmen tegner kantene som kurver i stedet for linjesegmenter. Det fjerde steget finner den glatteste kurven mellom tilkoblede noder som unngår andre noder og kurver.

[25]

7 Tidligere forskning på generering av enlinjeskjema

I artikkelen [14] presenteres en metode for å generere enlinjeskjema av en stasjon. Den gir kun en oversikt, og ikke detaljerte forklaringer av hvordan algoritmen fungerer. Artikkelen tar utgangspukt i data lagret på CIM/E format i en graf-database. CIM/E er en forenkling av cimxml som fjerner terminalene mellom komponenter og konnektivitetsnoder. Prosessen med å lage et enlinjeskjema er delt i to. Først genereres en layout, som lagres på et JSON-format. Så tolkes informasjonen fra JSON-fila, symbolene tegnes og vises i en nettside.

Generering av layout er delt i fire steg:

1. Finn spenningsnivåer og gi de et område.
2. Layout av busser innenfor hvert spenningsnivå.
3. Gi posisjon til transformatorer.
4. Tegn grener. En gren er en serie med komponenter koblet til en buss.

I første steget lages alle spenningsområder som et rektangel. Spenningsområdene blir sortert fra venstre mot høyre og ovenfra og ned, avhengig av antallet ulike spenninger.

Steg to finner busskonfigurasjonen innenfor hvert spenningsnivå. Deretter regner man ut bredden på bussene ved å gå gjennom en lignende logikk som for å tegne grener.

Steg tre plasserer transformatorer. Transformatorer er viktige siden de bestemmer plasseringen til flere grener.

Steg fire tegner grener. Først bestemmes retningen på grena. Normalt skal grena tegnes oppover. Hvis den er koblet til en generator rettes den nedover. Hvis den er koblet til en transformator, blir retningen bestemt av spenningen i forhold til de andre spenningene. Grenene med retning oppover sorteres etter antall noder, i tillegg til å sjekke om de er koblet til en annen buss. Grenene under sorteres etter posisjonene til transformatoren den er koblet til. Hver gren kan ha flere undergrener koblet til seg, så bredden på hver gren estimeres før alle grenene får posisjoner relativt til bussen de tegnes fra. Til slutt tegnes alle grenene, med en rekursiv metode for å tegne alle undergrenene.

8 Overordnet design av eget program

I dette kapitlet presenteres det overordnede designet og strukturen på program som er implementert i oppgaven. Det øverste nivået av programmet er skrevet som celler i Jupyter Notebook. Funksjonene som kalles er definert i Python-moduler skrevet i egne .py-filer. Prosessen for å lage et enlinjeskjema er delt i flere steg:

1. Lese inn cimxml-filer, til en Python struktur som kan brukes videre
2. Lage en graf, som forenkler sammenkoblingene mellom elektriske komponenter
3. Generere en layout
4. Lage skjema med symboler basert på layouten

Datagrunnlaget er i form av cimxml-filer. Dataene som brukes i denne oppgaven kommer delt i flere filer. PyCIM har støtte for å lese inn flere filer. Med PyCIM kan all informasjonen leses inn til Python klasser samlet i en dictionary. Det gjør det enkelt å finne informasjonen man trenger videre i programmet. Den eneste ulempen med PyCIM er hastigheten, større nett vil ta tid å lese inn. Det er likevel ikke et stort problem, alle cimxml-filene som brukes her blir lest inn på litt over et minutt. Derfor blir PyCIM brukt for å lese inn cimxml-filene.

For å lage en graf er Python biblioteket NetworkX valgt. Det er enkelt å bruke i forhold til å konstruere grafen. Det har også grensesnitt mot Graphviz, via bibliotekene Pydot eller Pygraphviz. Pydot og Pygraphviz kan også lage grafer ved å legge til noder og kanter, men NetworkX har i tillegg mange graf-algoritmer som kan brukes på NetworkX grafer. Det gir mange muligheter når det skal genereres en layout.

Hvordan grafen er lagd har mye å si senere når layouten skal lages og når enlinjeskjemaet skal tegnes. Kort sagt må all informasjonen som kreves for begge delene, lagres til grafen i dette steget.

Layout er blitt generert med to ulike metoder. Først ble det prøvd å bruke dot-algoritmen. Deretter ble det skrevet en egen algoritme. For dot-algoritmen kreves det en rettet graf, som betyr at det må lages en rettet graf før layouten blir generert.

Enlinjekjemaet blir tegnet som SVG, og for å tegne det fra Python benyttes biblioteket *svgwrite*. Det har alle egenskapene som trengs for å tegne selve symbolene, gruppere elementer, legge til CSS-stil og JavaScript funksjoner. Det er også lett å bruke og har god dokumentasjon. For å gjøre systemet mest mulig oversiktlig deles denne i tre filer. En fil som spesifiserer symbolene, i form av et

SVG-element, samt lagre andre verdier som trengs for å koble symboler sammen. En fil lager selve SVG-filen. Symbolene og SVG-filen er ganske uavhengige hverandre, så det er mulig å definere nye symboler for å erstatte de som er lagd her. Den siste filen bruker de to andre filene for å lage bilde fra en layout.

Kode 1 og Kode 2 viser hvordan stegene kombineres for henholdsvis layout med dot-algoritmen og layout med den selvlagde algoritmen. Innlesingen av cimxml-filer er allerede gjort, og lagret til variabelen `cim`.

```
1  def make_substation_diagram_dot(substation_id, filename):
2      print(f'Creating substation diagram of {substation_id}')
3      G = make_internal_graph(cim[substation_id])
4      D = bfs_direct_graph(G)
5
6      pos = dot_layout(D) # This fails if the graph has attributes
                             connected to the nodes
7
8      nx_util.copy_node_attributes(G, D) # Which is why the
                             attributes are added later
9
10     draw_substation_symbols(filename, D, pos)
```

Kode 1: Funksjon for å generere et enlinjeskjema av en stasjon med dot-algoritmen

```
1  def make_substation_diagram_custom(substation_id, filename):
2      print(f'Creating substation diagram of {substation_id}')
3      G = make_internal_graph(cim[substation_id])
4
5      pos = custom_layout(G)
6
7      D = custom_direct_graph(G, pos)
8      nx_util.copy_node_attributes(G, D)
9
10     draw_substation_symbols(filename, D, pos)
```

Kode 2: Funksjon for å generere et enlinjeskjema av en stasjon med den selvlagde algoritmen

9 Utvikling og programmeringsmiljø

Utviklingen har foregått i Jupyter Lab, som er en videreutvikling av Jupyter Notebook. Jupyter er en nettbasert IDE (Integrated Development Environment) som lar deg dele koden i flere celler som kan kjøre uavhengig av hverandre. Alle variabler på et globalt nivå blir lagret slik at andre celler kan bruke de. Det er veldig nyttig under utvikling, så man slipper å kjøre kode for å lage den samme variabelen om og om igjen når man tester noe som skjer senere. For eksempel kan resultatet fra å lese inn cimxml-filene lagres, så slipper man å kjøre innlesingen på nytt.

Docker er et Platform as a Service produkt som lar deg sette opp programvare i konteinere. Her brukes Docker for å lage programmeringsmiljøet som Jupyter kjører i. Den store fordelen med det er at man vet at miljøet er likt hver gang man starter det opp. Oppdateringer på pcen, overføring til en ny pc, til og med nytt operativsystem vil ikke endre miljøet.

Docker-konteineren startes med kommandoen `docker-compose up -d --build`. Ved oppstart vil Graphviz og alle Python bibliotekene som trengs lastes ned. Jupyter Lab blir startet, og kan aksesseres i en nettleser på `localhost:8889`.

```
FROM jupyter/scipy-notebook
USER root
RUN apt-get update && apt-get install -y gdal-bin libgdal-dev
USER jovyan
RUN pip install coconut
RUN pip install matplotlib networkx PyCIM svgwrite

USER root
RUN apt-get -y update && apt-get install -y graphviz
RUN pip install pydot

CMD ["coconut", "--jupyter", "lab"]
```

Kode 3: Dockerfile som lager miljøet som er brukt

```
version: "3"
services:
  jupyter:
    build:
      context: ./
      dockerfile: Dockerfile
    ports:
      - 8889:8888
    volumes:
      - ./:/home/jovyan/master:rw
```

```
11      environment:
12        - JUPYTER_ENABLE_LAB=yes
13      command:
14        [
15          "start-notebook.sh",
16          "--ip",
17          "0.0.0.0",
18          "--no-browser",
19          "--NotebookApp.password='sha1:7f0c851a2860:
                                          e61efd0689de608e95b384fcc918bad51dff53ce
                                          '",
20          "--NotebookApp.allow_origin='*'",
21        ]
```

Kode 4: docker-compose.yaml som lager miljøet som er brukt

Forklaring av Kode 4

`4-6:` Når docker-compose kommandoen kalles blir konteineren bygd basert på innholdet i dockerfilen i samme mappe som docker-compose.yaml filen.

`7-8:` Gjør porten 8888 inne i Docker tilgjengelig for utsiden på port 8889. Hvis man går på localhost:8889 på pcen som Docker kjører i, vil man få vist innholdet på port 8888. Port 8888 er porten som Jupyter Lab kjøres på.

`9-10:` Gjør mappa som docker-compose.yaml ligger i tilgjengelig for Docker konteineren.

`11-12:` Kjører Jupyter Lab i stedet for Jupyter Notebook.

`13-22:` Kommandoen og ekstra argumenter for å starte Jupyter. Setter et passord for å komme inn på Jupyter Lab.

10 Innlesing av cimxml-filer

Cimxml-filer leses ved hjelp av PyCIM modulen. Det lagrer all informasjonen i filen som instanser av klasser definert i PyCIM. Alt er samlet i en Python dictionary der den unike id-en er nøkkelen og instansen er verdien. Det er mulig å lese inn flere filer til samme dictionary, ved å oppgi en *start_dict*. PyCIM vil da fortsette å legge til nye instanser i den oppgitte dictionary-en.

Funksjonen som brukes for å lese cimxml-filer er gitt under. Den sjekker om *path* er en fil eller en mappe, før den enten leser den ene filen ved hjelp av PyCIM, eller iterer gjennom alle filene i mappen som er gitt og leser de til samme dictionary.

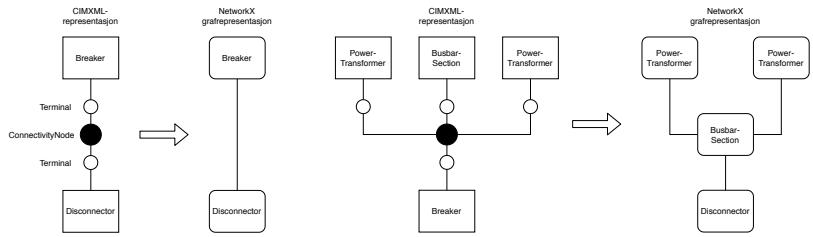
```
1  def cimread(path: str, start_dict=None): cim = start_dict if
                                    start_dict is not None else {}
2      if os.path.isfile(path):
3          cim = PyCIM.cimread(path, start_dict=cim)
4      elif os.path.isdir(path):
5          for file in glob.glob(f'{path}/*.xml'):
6              cim = PyCIM.cimread(file, start_dict=cim)
7      return cim
8
9  def cimread_list(paths: list):
10     cim = {}
11     for path in paths:
12         print(path)
13         cim = cimread(path, start_dict = cim)
14     return cim
```

Kode 5: Kode som leser cimxml til en dictionary

11 Lage graf fra PyCIM dictionary

Funksjonen som lager en graf over en stasjon skal ta inn en instans av PyCIM Substation og returnere en graf. Grafen skal ha med den første tilkoblede komponenten som er utenfor stasjonen. Det er for å vise grensesnittet mot omverdenen, selv når diagrammet vises på egenhånd. Det gjør også layout-algoritmen enklere å lage. Grafen er ikke en en-til-en representasjon av innholdet i cimxml-filene. CIM inneholder mange elementer som ikke er interessante å vise fram, deriblant Terminal, de fleste ConnectivityNodes, og BusbarSection. Disse blir tatt bort fra grafen. For BusbarSection blir konnektivitetsnoden den er koblet til gjort om til en node med type BusbarSection. Disse forenklingene er demonstrert i Figur 13.

ConnectivityNodes viser kun koblinger mellom komponenter, i seg selv er den ikke viktig. Der den kun kobler sammen to komponenter kan den fjernes uten tap av informasjon. Det gjør skjemaet mindre og lettere å lese. ConnectivityNodes brukes også som et grensesnitt mot utsiden av stasjonen, der det enten ikke er informasjon om hva som er utenfor, eller der det er en blindvei (en serie med komponenter som ikke er koblet til noe).



Figur 13: Hvordan terminaler, konnektivitetsnoder og BusbarSection forenkles bort under generering av graf

Nodene i grafen må inneholde all informasjon som kan brukes når layouten skal lages. For alle komponenter blir navnet på komponenten og typen det er lagret. Typen trengs for å avgjøre hvilket symbol som skal lages. Navnet er for å kunne lage en navnelapp som vises når man holder musa over symbolet, men det trengs ikke for selve diagrammet. For klassen PowerTransformer gjøres typen om til "PowerTransformer2" eller "PowerTransformer3" avhengig av antallet terminaler. Spenningsnivåene på de ulike endene i transformatoren lagres også. I alle andre komponenter, også ConnectivityNode, lagres spenningen. Den trengs for å kunne koble til riktig ende i transformatoren. For de ulike typene bryter, som enten kan være koblet inn eller ut må også den tilstanden lagres.

En siste viktig detalj er at grensesnittet mot utsiden markeres slik at de er lette å finne senere. Det gjelder både første komponenten som er utenfor stasjonen

(bør normalt være ACLineSegment), og ConnectivityNodes som kun er koblet til en annen komponent (og som forklart over, kan være en grense mot utsiden). I tillegg blir koblinger mellom to komponenter som er utenfor stasjonen fjernet. To slike komponenter pleier ofte å skulle tegnes på samme rank, men hvis det er en kobling mellom de vil dot-algoritmen plassere de på ulike nivåer, siden den ikke tegner kanter horisontalt.

Funksjonen som lager en graf av komponentene i en stasjon kan deles inn i tre trinn:

1. Legg til ConductingEquipment som noder i grafen
2. Legg til ConnectivityNode og lag kanter mellom ConnectivityNode og ConductingEquipment
3. Fjern ConnectivityNode som kobler sammen kun to ConductingEquipment, og lag kanten mellom ConductingEquipment i stedet.

Det første trinnet er vist i Kode 6. Det utnytter at Substation instansen har referanser til alt innholdet i stasjonen, enten direkte eller indirekte via VoltageLevel eller Bay. En Substation kan inneholde et vilkårlig antall VoltageLevel, som igjen kan inneholde et vilkårlig antall Bay. Transformatorer ligger direkte under Substation, siden de ikke er knyttet til et bestemt spenningsnivå. Alle andre komponenter ligger enten under et VoltageLevel eller en Bay, avhengig av hvordan cimxml-filen er bygd opp. Når noden legges til G, finner funksjonen også attributtene som må lagres til noden. Det gjøres ved å sjekke typen på komponenten, og deretter legge til egenskapene man vet den har. Når en ny komponent legges til, sjekkes også komponentene den er koblet til. Hvis det er noen komponenter som er utenfor stasjonen, blir disse lagt til grafen. Dette må gjøres slik, fordi komponentene utenfor stasjonen ikke vil ligge i noen av EquipmentContainerene under stasjonen.

```
1  def make_internal_graph(substation) -> nx.Graph:
2      G = nx.Graph()
3      equipments = []
4      for equip in substation.getEquipments():
5          # Add equipment to G and equipments
6          ...
7
8      for u in substation.getVoltageLevels():
9          for equip in u.getEquipments():
10             # Add equipment to G and equipments
11             ...
12
13         for bay in u.getBays():
14             for equip in bay.getEquipments():
15                 # Add equipment to G and equipments
16                 ...
```

```
17      ...
```

Kode 6: Pseudokode for funksjonen som lager en graf. Steg 1: Legg til ConductingEquipment

Forklaring av Kode 6

`4-6:` Lager noder av alle komponenter som ligger direkte under stasjonen. Det bør kun være PowerTransformer. I tillegg til navn og type, hentes spenningene til de ulike endene på transformatoren. Instansen av Python klassen legges til `equipment`, og det legges til en node i grafen med all viktig informasjon.

`9-11:` Lager noder av alle komponentene som har en VoltageLevel som EquipmentContainer. Legger navn, type og spenning til noden som attributter. I tillegg sjekkes alle elektriske komponenter som er koblet sammen med denne. Hvis noen av komponentene er utenfor stasjonen legges de også til som en node i grafen. De får også en ekstra attributt som markerer at de er utenfor stasjonen.

`14-16:` Lager noder av alle komponentene som har en Bay som EquipmentContainer. Komponentene lages på samme måte som for komponentene under spenningsnivå.

Kode 7 viser det andre steget som legger til konnektivitetsnoder og kanter i grafen. Konnektivitetsnodene finnes ved å iterere over alle Terminals som er lagret i alle komponentene som ble funnet i forrige steg. Hvis en konnektivitetsnode er koblet til en BusbarSection, så gjøres noden i grafen om til en node av type Busbar, mens den opprinnelige BusbarSection-noden som ble lag i forrige steg, blir slettet.

```
1       ...
2       for equip in equipments:
3           for t in equip.getTerminals():
4               node = t.getConnectivityNode()
5               G.add_node(node.UUID, type=get_type_string(node))
6               if isinstance(equip, CIM15.IEC61970.Wires.BusbarSection
                                                            ):
7                   nx.set_node_attributes(G, { node.UUID: { 'busbar':
                                                    equip.UUID, '
                                                    voltage': equip.
                                                    getBaseVoltage().
                                                    nominalVoltage }
                                                    })
8                   G.remove_node(equip.UUID)
9               else:
```

```
                G.add_edge(node.UUID, equip.UUID)
    ...
```

Kode 7: Funksjonen som lager en graf. Steg 2: Legg til ConnectivityNode og kanter

Kode 8 viser det siste steget i å lage en graf. Den gjør de fleste forenklingene av grafen som ble diskutert tidligere i kapittelet.

```
    ...
    changes = []
    for n, data in G.nodes.items():
        if 'busbar' in data:
            G.nodes[n]['type'] = 'Busbar'
        if data['type'] == 'ConnectivityNode':
            neighbors = []
            for nbr in G[n]:
                neighbors.append(nbr)
                if 'voltage' in G.nodes[nbr]:
                    G.nodes[n]['voltage'] = G.nodes[nbr]['voltage']
            if len(neighbors) == 1:
                if 'outside' in G.nodes[neighbors[0]]:
                    changes.append({
                        'remove_node': n
                    })
                else:
                    G.nodes[n]['outside'] = True

            if len(neighbors) == 2:
                if 'outside' in G.nodes[neighbors[0]] and 'outside'
                                                    in G.nodes[
                                                    neighbors[1]]:
                    changes.append({
                        'remove_node': n
                    })
                else:
                    changes.append({
                        'remove_node': n,
                        'add_edge': neighbors
                    })

    for change in changes:
        if 'remove_node' in change:
            G.remove_node(change['remove_node'])
        if 'add_edge' in change:
            G.add_edge(*change['add_edge'], ConnectivityNode=change
                                            ['remove_node'])

    return G
```

Kode 8: Funksjonen som lager en graf. Steg 3: Fjern ConnectivityNode som kobler sammen kun to ConductingEquipment, og lag kanten mellom ConductingEquipment.

Forklaring av Kode 8

`4-5:` Fullfører endringen av konnektivitetsnode til samleskinne, som var påbegynt i Kode 7.

`12-18:` Hvis konnektivitetsnoden har en nabo, betyr det at den er koblet til kun en komponent. Er denne komponenten utenfor stasjonen, så fjernes konnektivitetsnoden. Er den ikke det, så markeres konnektivitetsnoden som en komponent utenfor stasjonen.

`21-24:` Hvis to komponenter som er koblet sammen er utenfor stasjonen, så blir denne koblingen fjernet helt, inkludert konnektivitetsnoden.

`25-29:` Alle andre koblinger der konnektivitetsnoden er koblingen mellom to komponenter, blir konnektivitetsnoden fjernet og en ny kant lagt til grafen mellom de to komponentene.

12 Layout med dot-algoritmen

Dot-algoritmen krever en rettet graf for å fungere. Den urettete grafen blir gjort om til en rettet graf ved å gjøre et bredde først søk fra inngangene til stasjonen. For å finne inngangene finner funksjonen først den høyeste spenningen. Deretter finner den alle nodene med attributten `outside=True` som har den høyeste spenningen. Koden som utfører bredde-først-søket og lager den rettete grafen er vist i Kode 9. Koden som finner inngangene er vist i Kode 10.

```
1  def bfs_direct_graph(G: nx.Graph) -> nx.DiGraph:
2      roots = get_roots(G)
3      D = nx.DiGraph()
4      queue = deque(roots)
5      visited = set()
6      while queue:
7          node = queue.popleft()
8          if node not in visited:
9              visited.add(node)
10             for neighbor in G.neighbors(node):
11                 if neighbor not in visited:
12                     D.add_edge(node, neighbor)
13                     queue.append(neighbor)
14     return D
```

Kode 9: Funksjonen som retter grafen

```
1  def get_roots(G: nx.Graph):
2      max_voltage = 0
3      for id, data in G.nodes.items():
4          max_voltage = max(max_voltage, data.get('voltage', 0))
5      roots = []
6      for id, data in G.nodes.items():
7          if data.get('voltage', 0) == max_voltage and data.get('
                                                outside', False):
8              roots.append(id)
9      return roots
```

Kode 10: Funksjonen som finner inngangene til stasjonen

Pydot eller Graphviz krasjer hvis den rettete grafen som det skal lages layout av har attributter på nodene. Rettingen av grafen vil derfor returnere en rettet graf uten attributter på nodene. Disse blir kopiert over etter at layouten er lagd.

Koden for å benytte dot-algoritmen til å lage layout er vist i Kode 11. Den tar inn en rettet graf, bruker grensesnittet mot Graphviz som er gitt av NetworkX

og Pydot, og får ut en dictionary med posisjonene til nodene. Posisjonen for en node er gitt som en tuple med x og y koordinat. For å komprimere resultatet blir x-posisjonene halvert i etterkant. Y-koordinatene speiles for å tilpasse koordinatene til SVG-koordinatsystemet.

```
1 def dot_layout(D: nx.DiGraph):
2     pos = nx.drawing.nx_pydot.graphviz_layout(D, prog='dot')
3     max_y = max(pos.values(), key = lambda p: p[1])[1]
4     for id, p in pos.items():
5         pos[id] = (p[0]/2, -p[1] + max_y)
6     return pos
```

Kode 11: Hvordan bruke dot-algoritmen for å generere en layout

13 Layout med egen algoritme

13.1 Mål og forutsetning

Den nye layout-algoritmen som presenteres her er hovedsaklig tilpasset å tegne dobbel-buss konfigurasjoner. Den kan tegne stasjoner med andre typer buss konfigurasjoner, men ikke alle blir vellykket. Den skal tegne dobbel-buss konfigurasjoner omtrent som de pleier å tegnes manuelt, der de to bussene er plassert ovenfor hverandre i stedet for på samme høyde. Komponentene som kobler bussene sammen skal tegnes som rette linjer mellom bussene.

Det er gjort en rekke antagelser for å forenkle problemet. Disse antagelsene er gjort med bakgrunn i datasettet som er brukt i denne oppgaven. I andre kraftnett er det ikke sikkert at de samme antagelsene holder.

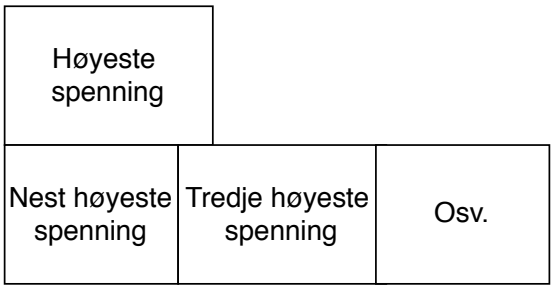
1. Grener med parallelle komponenter har kun en rank med parallelle komponenter. Hvis det er flere kan linjene fort bli tegnet på kryss og tvers. Alle komponentene og undergrener vil bli tegnet, men sannsynligvis ikke pent.
2. Antar at grenene som kobler sammen to busser på samme spenningsnivå ikke har parallelle komponenter i grenen.
3. Antar maksimalt tre ulike spenninger, der alle spenningene er koblet til transformatorer med tre ender hvis det er tre spenninger. Dette er viktig siden det er kun den høyeste spenningen som skal ha transformatorene under seg.
4. Maks to samleskinner på hvert spenningsnivå. I dataen som brukes i oppgaven her er BusbarSection kun brukt på de større samleskinnene som faktisk kobler sammen mange grener. I andre datasett kan det tenkes at det brukes flere steder der det kobles sammen flere komponententer. Dette vil skape problemer, for da vet ikke algoritmen hvilke som er "hovedsamleskinner" som skal brukes for å lage layout, og hvilke som er mindre samleskinner som burde plasseres på samme måte som vanlige komponenter.

En gren (eng: branch) er en serie med komponenter. Den kan bestå av komponentene mellom to busser, komponentene fra utsiden av stasjonen til en buss eller komponentene fra en transformator til en buss. Det finnes også undergrener (eng: subbranch) som går fra en konnektivitetsnode i en annen gren til en buss, komponent på utsiden, transformator eller en endekomponent (GroundDisconnector).

13.2 Sammenkobling av spenningsnivåer

Det første algoritmen gjør er å lage en graf for hvert spenningsnivå. Grafen vil inneholde alle komponentene som har den spenningen, inkludert eventuelle transformatorer. Når det er gjort kan det lages en layout for hver spenning uavhengig av de andre spenningene. Dette er en forenkling som ikke alltid vil gi et godt resultat. Hvordan layouten for hver spenning gjøres gjennomgås i detalj senere.

Til slutt kobles layoutene til de ulike spenningene sammen. Hver layout er en dictionary der nøkkelen er id-en til noden og verdien er posisjonen som en tuple. Hver layout har en rektangulær grense. De ulike layoutene forskyves slik at de ikke overlapper, illustrert i Figur 14. Denne enkle måten å forskyve layoutene før de slås sammen egner seg for opptil tre spenningsnivåer. Er det flere spenningsnivåer vil det gå koblinger fra nederst i enlinjeskjemaet til toppen av det neste spenningsnivået. Da blir mange kryssende linjer.



Figur 14: Hvordan layouten fra spenningsnivåene legges ut ved siden av hverandre

Transformatorene har fått posisjoner fra alle spenningsnivåene. Ved overlappende nøkler vil den som legges til siste bli stående. For trafostasjoner blir det best å la den høyeste spenningen bestemme posisjonen på trafoene. Derfor reverseres rekkefølgene på layoutene før de slås sammen.

Innenfor hvert spenningsnivå brukes et rutenett av heltall som koordinatsystem, der forskjellen mellom to ranks ved siden av hverandre er 1. Først etter at layoutene er slått sammen konverteres koordinatene til et pikselbasert koordinatsystem. Det gjøres ved å skalere x-koordinatene med en passende avstand mellom grener, og å skalere y-koordinatene med en passende avstand mellom ranks.

```
1 def custom_layout(G: nx.Graph):
2     levels = get_voltages(G)
3
4     top = True
```

```
5
6       layout = {}
7       level_layouts = []
8       level_bounds = []
9
10      for level in levels:
11          level_layout = layout_voltage_level(level, top)
12          level_layouts.append(level_layout)
13
14          bbox = get_bounds(level_layout)
15          level_bounds.append(bbox)
16
17          top = False
18
19      dy = level_bounds[0][3]
20      dx = 0
21
22      for level_layout, level_bound in zip(level_layouts[1:],
                                             level_bounds[1:]):
23          translate_layout(level_layout, dx, dy)
24          dx += level_bound[2] + 1
25
26      layout = merge_dicts(level_layouts[::-1])
27
28      pos = give_positions(G, layout, ranksep=100, nodesep=150)
29
30      return pos
```

Kode 12: Funksjonen som lager en layout for en stasjon

Forklaring av Kode 12

`2:` Lager graf av hvert spenningsnivå for seg selv

`4-17:` Lager en layout for alle spenningsnivåene og finner størrelsen til hver layout.

`19-24:` Forskyver layoutene til spenningsnivåene, slik at det øverste spenningsnivået er for seg selv over de andre, og resten ligger under på rekke.

`26:` Slår sammen dictionaries som lagrer layouten for de ulike spenningene. Sammenslåingen skjer i reversert ordre, slik at posisjonen til transformatorer blir bestemt av det høyeste spenningsnivået.

For å lage en graf av hvert spenningsnivå itereres det gjennom alle nodene i den opprinnelige grafen. Der samles de etter spenningsnivået. Den første iterasjonen fanger opp alle komponenter utenom transformatorer. Den andre iterasjonen legger til transformatorene, ved å finne den nærmeste spenningen som allerede er lagt til. Dette må gjøres fordi spenningene på transformatorene ikke alltid er helt lik som spenningene til resten av nivået.

```
def get_voltages(G: nx.Graph):
    voltages = defaultdict(list)
    for n, data in G.nodes.items():
        if 'voltage' in data:
            voltages[data['voltage']].append(n)
    for n, data in G.nodes.items():
        if data['type'].startswith('PowerTransformer'):
            vs = [v for k, v in data.items() if k in ['t0', 't1', '
                                                t2']]
            for v in vs:
                voltages[min(voltages.keys(), key=lambda k: abs(k-v
                                                    ))].append(n)

    subgraphs = []
    for voltage in sorted(voltages, reverse=True):
        subgraphs.append(G.subgraph(voltages[voltage]))

    return subgraphs
```

Kode 13: Lager en graf for hvert av spenningsnivåene

13.3 Layout av et spenningsnivå

Layouten for et spenningsnivå gjøres i fem steg:

1. Finn busskonfigurasjon
2. Finn komponentene som er innganger og utganger til spenningsnivået
3. Gi rank til nodene
4. Finn alle grenene
5. Bruk grenene og ranks som allerede er funnet til å gi posisjoner til nodene.

Hvordan layouten på et spenningsnivå bør lages er avhengig av typen busskonfigurasjon. Det første som gjøres er å lage en liste med bussene. Den lista blir brukt av flere av funksjonene senere.

```
def layout_voltage_level(level: nx.Graph, top: bool, debug=False):

    busbars = []
    for n, data in level.nodes.items():
        if data['type'] == 'Busbar':
            busbars.append(n)

    scheme, paths = get_bus_schemes(level, busbars)
```

```
9
10      source, sink = get_source_sink(level, top)
11
12      if scheme == 'double_bus' or scheme == 'sectionalized_bus':
13          ranks = give_rank_double_bus(level, paths, source, sink,
                                             busbars)
14      else:
15          ranks = give_rank_single_bus(level, source, sink, busbars)
16
17      node_ranks = {}
18      for i, ranka in enumerate(ranks):
19          for node in ranka:
20              node_ranks[node] = i
21
22      branches = find_all_branches(level, paths, source, sink,
                                         busbars)
23
24      layout = layout_branches(level, branches, node_ranks)
25
26      return layout
```

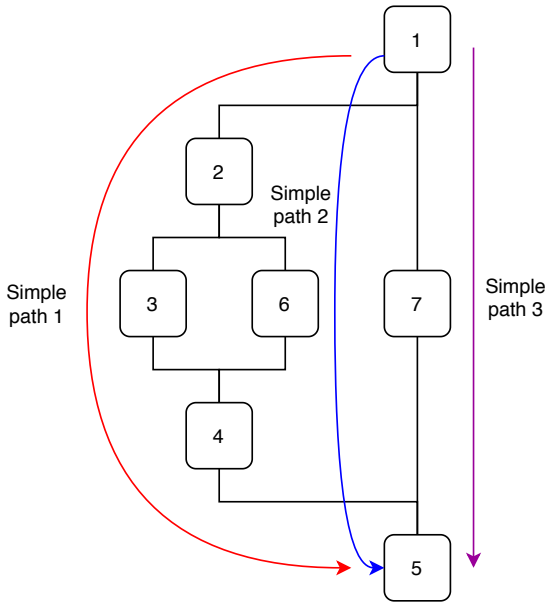
Kode 14: Hvordan det lages layout for et spenningsnivå

13.4 Steg 1: Finn busskonfigurasjon

Så finner man typen busskonfigurasjon ved å telle antall busser og telle "simple paths" mellom bussene. En "Simple path" er en serie med noder som kobler to komponenter sammen uten sykler, slik som det er vist i Figur 15. Koden er vist i Kode 15.

```
1   def get_bus_schemes(G: nx.Graph, busbars):
2       if len(busbars) == 0:
3           return 'no_bus', []
4       elif len(busbars) == 1:
5           return 'single_bus', []
6       elif len(busbars) == 2:
7           paths = list(nx.all_simple_paths(G, busbars[1],busbars[0]))
8           if len(paths) == 0:
9               return 'single_bus', []
10          elif len(paths) == 1:
11              return 'sectionalized_bus', paths
12          return 'double_bus', paths
13      return 'multi_bus', []
```

Kode 15: Funksjonen som finner typen busskonfigurasjon på et spenningsnivå



Figur 15: De fargede pilene viser de tre ulike "simple paths" mellom node 1 og node 5. Simple path 1 inneholder nodene [1, 2, 3, 4, 5]. Simple path 2 inneholder nodene [1, 2, 6, 4, 5]. Simple path 3 inneholder nodene [1, 7, 5].

13.5 Steg 2: Finn innganger og utganger

Kode 16 viser hvordan man finner innganger og utganger til et spenningsnivå. Den følger antagelse nummer 3, slik at det er kun det høyeste spenningsnivået som skal ha transformatorer under seg og utsiden av stasjonen over seg. For alle andre spenningsnivåer er det motsatt. Funksjonen kan ikke vite selv om det er det høyeste nivået eller ikke, så det må den få som et argument fra lenger opp i algoritmen.

```
1  def get_source_sink(G: nx.Graph, top: bool):
2      sources = []
3      sinks = []
4      if top:
5          for id, data in G.nodes.items():
6              if data.get('outside', False):
7                  sources.append(id)
8              elif data.get('type', '').startswith('PowerTransformer'
                                                    ):
9                  sinks.append(id)
10     else:
11         for id, data in G.nodes.items():
12             if data.get('outside', False):
13                 sinks.append(id)
```

```
14              elif data.get('type', '').startswith('PowerTransformer'
                                                     ):
15                  sources.append(id)
16      return sources, sinks
```

Kode 16: Funksjonen som finner komponentene som er innganger og utganger

13.6 Steg 3: Gi rank til nodene

Metoden for å gi rank er basert på avstand fra komponentene som er koblet til andre spenningsnivåer og utsiden av stasjonen. Denne er avhengig av typen busskjema. Koden for å finne ranks i en dobbel-buss-konfigurasjon er vist i Kode 18. Rankene er representert som en liste av en liste, slik som vist i Kode 17. Denne strukturen gjør det mulig å legge til en ny rank mellom to eksisterende.

```
[[1, 2], # rank 0
[3],     # rank 1
[4, 5]]  # rank 2
```

Kode 17: Eksempel på hvordan ranks lagres i funksjonen

Høyeste ranken består kun av inngangen, og siste rank består kun av utgangene. Det gjøres et bredde-først-søk fra inngangene, der hver nye dybde i søket lagres til en ny rank. Søket terminerer når den finner noder som er i en av grenene mellom de to bussene. Det gjør at alle komponenter mellom inngangene og hovedgrenene mellom bussene får rank over den øverste bussen. Det gjøres et tilsvarende søk fra utgangen, der alle komponentene får rank under den nederste bussen.

For å finne ut hvilken buss som skal ligge øverst regnes det på den gjennomsnittlige lengden fra innganger til den øverste bussen og utganger til den nederste bussen. Lengden fra en inngang til en buss er antallet noder i den korteste veien mellom inngangen og bussen. Verdien gir en score for de to alternativene. Rekkefølgen med lavest score returnes. Koden for dette er vist i Kode 19.

Til slutt legges alle komponentene mellom bussene til. Også dette gjøres med et bredde-først-søk likt som tidligere, men nå termineres søket når den kommer til noder som allerede har fått rank.

For enkelt-buss konfigurasjoner vil denne funksjonen krasje. Det kan være en buss, eller flere busser som ikke er koblet sammen. Det er lagd en egen funksjon for å gi rank til det tilfelle. Da trengs ikke bussene å rangeres, så den/de legges

på samme rank. Det siste søket som legger til komponentene mellom bussene er fjernet.

```
def give_rank_double_bus(G, paths, sources, sinks, busbars):
    all_path_nodes = [node for path in paths for node in path]

    ranks = [sources.copy(), sinks.copy()]

    visited = set()

    i = 1

    # Add Source branches to ranks above
    prev_rank = sources
    while True:
        next_rank = []
        for node in prev_rank:
            if not node in visited:
                visited.add(node)
                for neighbor in G.neighbors(node):
                    if not neighbor in visited and not neighbor in
                                                        all_path_nodes
                                                        :
                        next_rank.append(neighbor)
        if len(next_rank) == 0:
            break
        ranks.insert(i, next_rank)
        prev_rank = next_rank
        i += 1

    # Add Sink branches to ranks below
    prev_rank = sinks
    while True:
        next_rank = []
        for node in prev_rank:
            if not node in visited:
                visited.add(node)
                for neighbor in G.neighbors(node):
                    if not neighbor in visited and not neighbor in
                                                        all_path_nodes
                                                        :
                        next_rank.append(neighbor)
        if len(next_rank) == 0:
            break
        ranks.insert(i, next_rank)
        prev_rank = next_rank

    # Insert busbars to rank
    top_busbar, bottom_busbar = sort_busbars(G, sources, sinks,
                                        busbars)
    ranks.insert(i, [top_busbar])
    i += 1
    ranks.insert(i, [bottom_busbar])

    # Give rank to the components between buses
```

```
    prev_rank = [top_busbar]
    while True:
        next_rank = []
        for node in prev_rank:
            if not node in visited:
                visited.add(node)
                for neighbor in G.neighbors(node):
                    if not neighbor in visited:
                        next_rank.append(neighbor)
        if len(next_rank) == 0:
            break
        ranks.insert(i, next_rank)
        prev_rank = next_rank
        i += 1

    return ranks
```

Kode 18: Funksjonen som gir rank til nodene hvis det er en dobbel-buss konfigurasjon

```
def sort_busbars(G, sources, sinks, busbars):
    a = 0
    for source in sources:
        a += nx.shortest_path_length(G, source, busbars[0])
    b = 0
    for sink in sinks:
        b += nx.shortest_path_length(G, sink, busbars[1])
    config1_score = a/len(sources) + b/len(sinks)

    a = 0
    for source in sources:
        a += nx.shortest_path_length(G, source, busbars[1])
    b = 0
    for sink in sinks:
        b += nx.shortest_path_length(G, sink, busbars[0])
    config2_score = a/len(sources) + b/len(sinks)

    if config1_score < config2_score :
        return busbars[0], busbars[1]
    else:
        return busbars[1], busbars[0]
```

Kode 19: Funksjon som finner hvilken av de to bussene som bør være øverst

13.7 Steg 4: Finn alle grener

Det neste steget er å finne alle grenene. En gren lagres som en instans av en klasse kalt `Branch`. Klassen lagrer en liste med nodene i grenen, bredden på den

(i tilfelle den inneholder parallelle komponenter), en liste med undergrener som skal tegnes oppover og en liste med undergrener som skal tegnes nedover. Når grenene skal tegnes i neste steg blir de tegnet i samme rekkefølge som de legges til i her.

Grenene mellom to busser er allerede funnet. Først gjør man et rekursivt kall for å finne alle undergrenene fra disse hovedgrenene. Dette vil kun finne alle noder som er koblet til hovedgrenene. I enkelt-buss konfigurasjoner er det ingen hovedgrener, så man har en tom liste etter det. I dobbel-buss konfigurasjoner er det ofte en gren som kun er koblet til en av bussene.

Det trengs mer logikk som finner resten av grenene. Algoritmen finner alle endenoder som ikke er lagt til allerede. Deretter iterer den over disse. Hvis endenoden er en inngang eller utgang lages det en ny gren ved å finne den korteste veien fra endenoden til den nærmeste bussen. Deretter brukes en rekursiv funksjon for å finne alle undergrener til den nye grenen. Denne er vist i Kode 21 og forklares senere. Hele tiden holder den styr på hvilke endenoder som har blitt funnet i løpet av søket, siden det kan være endenoder som er i en undergren av en annen gren.

Alle endenoder som ikke er en inngang eller utgang er en undergren av en annen gren. Derfor trengs ikke de å finnes for seg selv. Det er flere fordeler ved kun å lage nye grener til endenoder som er enten en inngang eller utgang. Strukturen på grenen blir bedre, siden hovedgrenen er den som skal lengst. Retningen på grenen stemmer i forhold til ranken nodene i den har fått. Man unngår også å tegne en linje til en GroundDisconnector fra undersiden.

```
1  def find_all_branches(G, paths, sources, sinks, busbars):
2      branches = []
3      for path in paths:
4          branch = Branch(nodes=path)
5          find_subbranches(G, branch, sources, sinks)
6          branches.append(branch)
7
8      found_nodes = []
9      for branch in branches:
10         found_nodes.extend(get_branch_nodes(branch))
11
12     unfound_dead_ends = [n for n, d in G.degree() if d == 1 and n
                                    not in found_nodes]
13
14     found_ends = []
15     new_branches = []
16     for dead_end in unfound_dead_ends:
17         if dead_end in sources or dead_end in sinks and dead_end
                                        not in found_ends:
18             try:
19                 path = nx.shortest_path(G, source=busbars[0],
                                                target=dead_end)
20             except nx.NetworkXNoPath:
```

```
21                  path = None
22
23              for busbar in busbars[1:]:
24                  try:
25                      new_path = nx.shortest_path(G, source=busbar,
                                                    target=
                                                    dead_end)
26                  except nx.NetworkXNoPath:
27                      new_path = None
28
29                  if not new_path is None and (path is None or len(
                                                 new_path) < len(
                                                 path)):
30                      path = new_path
31
32              branch = Branch(nodes=path)
33              subbranch_new_ends = find_subbranches(G, branch,
                                                      sources, sinks)
34              new_branches.append(branch)
35              found_ends.extend([dead_end, *subbranch_new_ends])
36
37      new_branches.sort(key=lambda x: x.nodes[0])
38      branches.extend(new_branches)
39
40      return branches
```

Kode 20: Funksjonen som finner alle grenene på et spenningsnivå

Den rekursive funksjonen for å finne alle undergrener til en gren utnytter at undergrener vil være koblet til en konnektivitetsnode. Alle naboene som ikke er i grenen er starten på en ny undergren. Funksjonen passer på å ikke lage en gren til samme endenode flere ganger. Første steget i å lage en ny undergren er å lage en graf med kun nodene som kan være de nye undergrenene. Det gjøres med et bredde-først-søk der konnektivitetsnoden i hovedgrenen allerede er i listen over besøkte noder.

Det itereres gjennom alle endenodene to ganger. Første iterasjonen finner alle grener som peker oppover, altså grenene der endenoden er i inngangene til spenningsnivået. Andre gangen finner den alle de andre grenene som ikke allerede er funnet. Disse skal tegnes nedover. Endenodene her er enten i utgangene fra spenningsnivået eller GroundDisconnectors.

Rekkefølgen på iterasjonene gjør at hvis det er en gren som er koblet til både en inngang og utgang, blir den først tegnet mot inngangen, mens grenen mot utgangen blir en undergren til den første. Utenom denne prioriteringen er hvilke grener som blir undergrener avhengig av rekkefølgen på endenodene. Rekkefølgen på endenodene er tilfeldig, men den er konsistent for samme graf.

I linje 20-25 lages den nye undergrenen. Det tas hensyn til at det kan være flere simple_paths i samme gren. Det vil si at det er parallelle komponenter i

grenen. Nodene i grenen legges kun til en gang, og de sorteres ikke på noen måte. Antagelsen om at det kun er en rank med parallelle komponenter gjør at dette går fint når grenen skal tegnes senere. Å legge til alle parallelle komponenter i grenen er viktig for at funksjonen ikke skal prøve å lage undergrener fra de andre parallelle komponentene, da det hadde funnet nesten alle endenodene på spenningsnivået.

Etter at den nye grenen er lagd kalles `find_subbranches()` på nytt for å finne undergrenene som hører til den nye grenen.

```
1  def find_subbranches(G: nx.Graph, branch, sources: list, sinks:
                                    list):
2    ends = []
3    for main_node in branch.nodes:
4      if G.nodes[main_node].get('type') == 'ConnectivityNode':
5        for neighbor in G.neighbors(main_node):
6
7          if not neighbor in branch.nodes:
8
9            all_nodes = bfs(G, [neighbor], sources, sinks, visited =
                                          set([main_node]))
10
11           subgraph = G.subgraph(all_nodes)
12
13           dead_ends = [n for n, d in subgraph.degree() if d == 1
                                          and n != neighbor]
14
15           for end in dead_ends:
16             if end in sources and not end in ends:
17
18               paths = list(nx.all_simple_paths(G, source = neighbor
                                            , target = end))
19
20               new_branch_nodes = set(paths[0])
21               if len(paths) > 1:
22                 for path in paths[1:]:
23                   new_branch_nodes |= set(path)
24               new_branch_nodes = list(new_branch_nodes)
25               new_branch = Branch(new_branch_nodes, width=len(paths
                                            ))
26               ends.append(end)
27
28               new_ends = find_subbranches(G, new_branch, sources,
                                            sinks)
29               ends.extend(new_ends)
30               branch.up_subbranches.append(new_branch)
31
32           for end in dead_ends:
33             if not end in ends:
34               ... # Same as in above loop
35               branch.down_subbranches.append(new_branch)
36
37     return ends
```

Kode 21: Funksjonen som finner og legger til undergrener i en gren

13.8 Steg 5: Gi nodene posisjoner

Siste steget tar grenene som ble funnet i forrige steg og rankene for å gi ferdige posisjoner til alle nodene. Y-koordinatene til nodene er lik som ranken som ble funnet tidligere. X-koordinatene gis ut fra venstre mot høyre. For hver gren som tegnes økes variabelen `shift` slik at neste gren blir tegnet lenger til høyre enn den forrige.

Undergrener som peker i retning oppover fra der de er koblet til tegnes på venstre side av hovedgrenen, mens undergrener som peker nedover tegnes til høyre for hovedgrenen. Alle undergrener får layout ved å kalle `layout_branch()`. Det tillater at undergrener har egne undergrener.

Rekkefølgen som grener tegnes i, og dermed posisjonen langs x-aksen, er avhengig av rekkefølgen grenene ble lagt til i listen.

```
1  def layout_branches(branches, node_ranks):
2      layout = {}
3      shift = 0
4      for i, branch in enumerate(branches):
5          local_layout, shift = layout_branch(branch, node_ranks,
                                                shift)
6          layout.update(local_layout)
7      return layout
```

Kode 22: Funksjonen som gir posisjoner til alle nodene

```
1  def layout_branch(branch, node_ranks, shift):
2      layout = {}
3
4      for subbranch in branch.up_subbranches:
5          local_layout, shift = layout_branch(subbranch, node_ranks,
                                                shift)
6          layout.update(local_layout)
7
8      if branch.width == 1:
9          for node in branch.nodes:
10             layout[node] = (shift, node_ranks[node])
11     else:
12         ranks = defaultdict(list)
```

```
13          for node in branch.nodes:
14              ranks[node_ranks[node]].append(node)
15
16          for rank, nodes in ranks.items():
17              for i, node in enumerate(nodes):
18                  layout[node] = (shift + i, node_ranks[node])
19
20      shift += branch.width
21
22      for subbranch in branch.down_subbranches:
23          local_layout, shift = layout_branch(subbranch, node_ranks,
                                                shift)
24          layout.update(local_layout)
25
26      return layout, shift
```

Kode 23: Rekursiv funksjon som gir posisjoner til en gren inkludert alle undergrener

14 Tegning av SVG-bilde med symboler

For å gi muligheten til pan og zoom av SVG-bildene brukes et eksternt javascript skript. Skriptet som benyttes er skrevet av Andrea Leofreddi, og kan finnes her [26]. Skriptet fanger opp aktiviteten til musa, og legger på en transformasjon på SVG-gruppen med id-en "viewport". Det vil endre posisjonen på alle elementene i gruppa. Transformasjonen er i form av en transformasjonsmatrise slik som den vises i Ligning 1. Bruken av dette skriptet krever at alle SVG-elementene samles under en felles gruppe.

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & d & e \\ b & c & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{prev} \\ y_{prev} \\ 1 \end{bmatrix} \tag{1}$$

SVG modulen er delt i tre filer, en som definerer symboler ('symbols.py'), en som lagrer alle symbolene til en fil ('svg_image.py'), og en som har funksjonen som lager symbolene fra NetworkX grafen og layouten ('drawing.py').

14.1 Lage SVG-bilde

I filen 'svg_image.py' ligger klassen `SvgImage`, som brukes for å lage et nytt bilde. Klassen er vist i Kode 24. Klassen fungerer som en wrapper for Drawing-klassen fra svgwrite. Ved initialisering får instansen et navn, som vil være filnavnet når den lagres. Den legger til en stil, som er definert tidligere i samme fil som en tekststreng. CSS-stilen er vist i Kode 25.

'svgpan.js' er javascript-filen som gjør det mulig å dra rundt og zoome i SVG-bildet. Filen leses inn til en tekststreng og legges inn som et lokalt skript i SVG-filen, på samme måte som CSS stilen. Da bli SVG-filen helt uavhengig eksterne filer, noe som er veldig beleilig når man skal flytte på filen eller dele den med andre. Til slutt lages en gruppe med id "viewport", som vil bli brukt av 'svgpan.js' når den anvender transformasjonen.

```
1  class SvgImage():
2      def __init__(self, name):
3          self.name = name
4          self.drawing = Drawing(name + '.svg')
5          self.drawing.add(Style(DEFAULT_STYLE))
6          with open('svgpan.js', 'r') as file:
7              script_string = file.read()
8          self.drawing.add(Script(content = script_string))
9          self.group = Group(id='viewport')
10         self.drawing.add(self.group)
11
```

```
12      def save(self):
13          self.drawing.save(pretty=True)
14
15      def draw(self, svg_element):
16          self.group.add(svg_element)
```

Kode 24: Klassen for SVG-bilde

CSS stilen vist i Kode 25 er ikke kun for å gi statiske stiler til de ulike symbolene. Den brukes også for å synliggjøre hva man holder over ved hjelp av `:hover` selektoren. Siden tekststrengen med klassen label er samlet i gruppe med symbolet, kan den bli vist kun når man holder musa over symbolet den tilhører.

```
1  svg {
2      background-color:white;
3  }
4  .label {
5      visibility: hidden;
6      text-anchor: middle;
7      font-size: 9px;
8      stroke: black;
9  }
10 .symbol {
11     fill: rgba(0,0,0,0);
12     stroke-width: 1;
13     stroke: black;
14 }
15 .aclinesegment {
16     stroke-width: 2;
17 }
18 .busbar {
19     fill: #666666;
20 }
21 .busbar:hover {
22     stroke: red;
23 }
24 .connectivitynode {
25     fill: black;
26 }
27 .open {
28     stroke: red;
29 }
30 .closed {
31     stroke: green;
32 }
33 .line {
34     stroke: black;
35 }
36 .line:hover {
37     stroke: red;
38 }
```

```
39 .symbol:hover > .label {
40     visibility: visible;
41 }
```

Kode 25: CSS stilen som lagres i SVG-filene

Når symboler for de ulike komponentene i enlinjeskjemaet skal legges til i SVG-filen kalles funksjonen `draw_symbol(symbol)`. Parameteren `symbol` er en instans av en klasse som definerer symbolet. Alle de ulike symbolene har sin egen klasse. Felles for alle klassene har metoden `draw()`, som vil returnere en gruppe fra *svgwrite*. Gruppa inneholder den/de geometriske figurene som definerer selve symbolet, samt et tekstfelt. Tekstfeltet har informasjon om symbolet, i dette tilfelle er det typen symbol og den unike id'en til komponenten. For transformatorer vises også de ulike spenningsnivåene. Ved å bruke CSS stilen vist i Kode 25, er tekstfeltet usynlig til man holder over symbolet. Slik unngår man at bildet blir fullt av tekst, samtidig som informasjonen er der hvis det er noe spesielt man vil se på. Gruppa som returneres fra symbolet legges til i tegningens 'viewport'-gruppe.

14.2 Symboler

Symbolene som brukes er definert i filen 'symbols.py'. Når symbolene blir tegnet ser de ut slik som vist i Figur 9. Alle arver fra en baseklasse, `Symbol`, som har attributtene som er felles for alle. `Symbol` klassen er vist i Kode 26. `Symbol` arver fra klassen `ABC`, som kommer fra biblioteket *Abstract Base Classes*. Metodene som må implementeres i alle klasser som arver fra `Symbol` får dekoratoren `@abstractmethod`. Det vil gi en feilmelding og krasje programmet dersom man prøver å initialisere en klasse som ikke har gjort det. Strengt tatt er ikke dette nødvendig for at programmet skal fungere, men det hjelper for å se strukturen i klassehierarkiet. Hvis man på et senere tidpunkt kommer over et nytt symbol som må legges til programmet, vil det fungere som en påminnelse om hva som må implementeres. På denne måten kan alle symbolene brukes likt.

```
1  class Symbol(ABC):
2      def __init__(self, type: str, id: str, x: float, y: float,
                                                width: float = 16, height:
                                                float = 16):
3          self.type = type
4          self.id = id
5          self.x = x
6          self.y = y
7          self.width = width
8          self.height = height
9
10     def __str__(self):
```

```
11          return f' Symbol: {self.type}:  ( X: {self.x},  Y: {self.y}
                                              )'
12
13      def get_posision(self):
14          return (self.x, self.y)
15
16      def get_size(self):
17          return (self.width, self.height)
18
19      @abstractmethod
20      def get_parent_connection(self, *extra, **kwargs):
21          pass
22
23      @abstractmethod
24      def get_child_connection(self, *extra, **kwargs):
25          pass
26
27      @abstractmethod
28      def draw(self):
29          pass
```

Kode 26: Klasse for en linje

Selve symbolene kan utvides ved å legge til nye egenskaper. For eksempel; alle typer som kan være lukket eller åpen har attributten `closed`, som indikerer posisjonen den normalt er i. Det inkluderer typene *Breaker*, *Disconnector*, *Jumper* og de ulike variantene av disse. Transformatorer (*PowerTransformer2* og *PowerTransformer3*) lagrer også en liste over de ulike spenningene den har.

For de fleste symbolene vil `get_parent_connection()` og `get_child_connection()` returnere et fast punkt som sier hvor en linje skal koble seg til. `get_parent_connection()` gir punktet som skal brukes for koblinger fra komponenter over i diagrammet, og `get_child_connection()` gir punktet som skal brukes for koblinger til komponenter under i diagrammet. Unntakene for denne regelen er *Busbar* og *PowerTransformer3*. Det er også derfor funksjonene kan ta inn et ubestemt antall posisjonelle og navngitte argumenter via `*args` og `**kwargs`. Funksjonen som lager alle symbolene (vist i Kode 32) gir posisjonen som posisjonelle argumenterer og egenskapene til noden som skal kobles til som navngitte argumenter. For symbolene som gir et fast punkt uansett blir argumentene ignorert,

Et typisk symbol er *Breaker*, som er vist i Kode 27. Den initialiseres med `__init__()` funksjonen, som kaller initialiseringsfunksjonen fra `Symbol` først. x- og y-koordinatene blir justert til å være det øverste hjørne til venstre. Utenom det som allerede arves fra `Symbol` blir normalposisjonen lagret til attributten `closed`.

```
1  class Breaker(Symbol):
2      def __init__(self, x: float, y: float, size: float = 16,
```

```
3                 closed: bool = True, id='breaker', **kwargs):
4         super().__init__('Breaker', id, x - size/2, y - size/2,
                                                size, size)
5         self.closed = closed
6
7     def get_parent_connection(self, *args, **kwargs):
8         return (self.x + self.width/2, self.y)
9
10    def get_child_connection(self, *args, **kwargs):
11        return (self.x + self.width/2, self.y + self.height)
12
13    def draw(self):
14        class_ = f'symbol breaker {"closed" if self.closed else "
                                                open"}'
15        g = Group(class_=class_, id=self.id)
16        g.add(Text(f'{self.id} {self.type}', insert=self.
                                                get_posision(),
17                class_='label'))
18        g.add(shapes.Rect(insert=self.get_posision(), size=self.
                                                get_size()))
19        return g
```

Kode 27: Klasse for en bryter

For 'Busbar' vil det å legge til en ny kobling potensielt medføre at x-posisjonen og/eller bredden endres, slik at tilkoblingen blir en vertikal linje. Av argumentene som kommer inn til `get_parent_connection()` og `get_child_connection()` funksjonene brukes kun x-posisjonen. Dette er vist i Kode 28

```
1  class Busbar(Symbol):
2      ...
3      def get_parent_connection(self, x: float, *extra, **kwargs):
4          self.adjust_width(x)
5          return (x, self.y)
6
7      def get_child_connection(self, x: float, *extra, **kwargs):
8          self.adjust_width(x)
9          return (x, self.y + self.height)
10
11     def adjust_width(self, x: float):
12         MARGIN = 5
13         if x < self.x - MARGIN:
14             self.width = self.width + self.x - x + MARGIN
15             self.x = x - MARGIN
16         if x > self.x + self.width - MARGIN:
17             self.width = x - self.x + MARGIN
18     ...
```

Kode 28: Klasse for en samleskinne

Det andre symbolet som skiller seg ut er *PowerTransformer3*, en transformator med tre spenninger. Den har tre ulike punkter som kan kobles til, avhengig av spenningen. Den lagrer en liste med de ulike spenningene, samt en dictionary som kobler de ulike spenningene til et punkt. I `get_parent_connection()` og `get_child_connection()` funksjonene brukes det navngitte argumentet `voltage`, som kommer fra egenskapene til noden. Det er ikke sikkert at spenningen som er gitt for transformatoren er helt lik spenningen til komponentene rundt. Derfor må den koblingen med likest spenning brukes hvis den ikke er helt lik.

```
1  class PowerTransformer3(Symbol):
2      def __init__(self, x: float, y: float, height: float = 30, id='
                                        powertransformer3', **kwargs)
                                        :
3          self.radius = 3/10 * height
4          super().__init__('PowerTransformer3', id, x, y, self.radius
                                            * 3, height)
5          self.voltages = [v for k, v in kwargs.items() if k in ['t0'
                                            , 't1', 't2']]
6          self.connections = {
7              self.voltages[0]: (self.x, self.y - self.height/2),
8              self.voltages[1]: (self.x, self.y + self.height/2),
9              self.voltages[2]: (self.x + self.radius*2, self.y)
10         }
11
12     def get_parent_connection(self, *args, voltage=0, **kwargs):
13         if voltage in self.voltages:
14             return self.connections[voltage]
15         # else find the closest voltage
16         return self.connections[min(self.connections.keys(), key=
                                            lambda k: abs(k-voltage))
                                            ]
17
18     def get_child_connection(self, *args, voltage=0, **kwargs):
19         if voltage in self.voltages:
20             return self.connections[voltage]
21         # else find the closest voltage
22         return self.connections[min(self.connections.keys(), key=
                                            lambda k: abs(k-voltage))
                                            ]
23     ...
```

Kode 29: Klasse for en transformator med tre ulike spenninger

Det er lagd en egen klasse for linjer også. Den lagrer en liste med punkter (tupler med 2 verdier). I likhet med symbolene har denne klassen også en `draw()` funksjon som returnerer et objekt fra *svgwrite*. Her er det riktignok kun en linje som returneres og ikke en gruppe, men den kan legges til 'viewport'-gruppa på samme måte.

```
class Line():
    def __init__(self, points: list, from_symbol: str = '',
                                        to_symbol: str = ''):
        self.points = points
        self.from_symbol = from_symbol
        self.to_symbol = to_symbol

    def draw(self):
        return shapes.Polyline(self.points, class_='line')
```

Kode 30: Klasse for en linje

I 'symbols.py' er det også en funksjon som gjør det enkelt å lage nye symboler, basert på hvilken type komponent det er.

```
1 def create_symbol(type_: str, x: float, y: float, **kwargs):
2     if type_ == 'PowerTransformer2':
3         return PowerTransformer2(x, y, **kwargs)
4     elif type_ == 'Breaker':
5         return Breaker(x, y, **kwargs)
6     ...
7     else:
8         return Unkown(x, y, **kwargs)
```

Kode 31: Funksjonen som lager et symbol av basert på typen

14.3 Tegning av SVG basert på layouten

Funksjonen som tar inn en graf og posisjonen til nodene, og lager en SVG-fil, ligger i 'drawing.py'. Det itereres over alle nodene i grafen, der symbolene lages ved hjelp av `create_symbol()` funksjonen. Symbolene legges til en dictionary. For å lage linjene itereres det over alle etterfølgerene til noden. Det sjekkes at etterfølgeren eksisterer, hvis ikke lages den. Når symbolet for både noden og etterfølgeren er lagd og lagret, finner man koblingene mellom de og lager en linje. Her ser man fordelen ved at alle symbolene har samme grensesnitt, det trengs ingen if-setninger for å sjekke typen på symbolene. Til både `get_parent_connection()` og `get_parent_connection()` gis posisjonen som posisjonelle argumenterer og egenskapene til noden som skal kobles til, så er det opp til implementasjonen av symbolet om argumentene skal brukes.

```
1 def draw_substation_symbols(filename: str, D: nx.DiGraph, pos: dict
                                    ):
2     symbols = {}
```

```
3       lines = []
4
5       for node, data in D.nodes.items():
6           if not node in symbols:
7               symbols[node] = create_symbol(data['type'],
8                       *pos[node], id=node, **D.nodes[node])
9
10          for child in D.successors(node):
11              if not child in symbols:
12                  symbols[child] = create_symbol(
13                          D.nodes[child]['type'], *pos[child],
14                          id=child, **D.nodes[child])
15
16              top_connection = symbols[node]
17                      .get_child_connection(*pos[child], **D.nodes[
                                                            child])
18
19              bottom_connection = symbols[child]
20                      .get_parent_connection(*pos[node], **D.nodes[
                                                            node])
21
22              lines.append(Line(
23                      (top_connection, bottom_connection),
24                      node, child))
25
26      img = SvgImage(filename)
27
28      for id, symbol in symbols.items():
29          img.draw(symbol.draw())
30
31      for line in lines:
32          img.draw(line.draw())
33
34      img.save()
```

Kode 32: Funksjonen som tar inn en graf og posisjonene til nodene, og lager en SVG-fil med symboler

15 Resultater og diskusjon

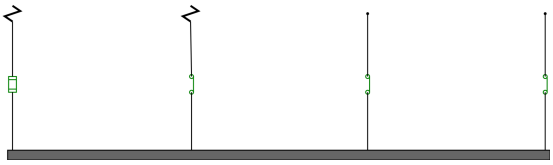
Hele datasettet som har vært tilgjengelig i løpet av arbeidet med denne oppgaven er delt i 18 filer på totalt 105 MB. Kraftnettet inneholder 1155 stasjoner av ulike størrelser. To av filene som inneholder 25 stasjoner er studert nøye og brukt aktivt i utviklingen.

Fem av de 25 stasjonene er plukket ut for rapporten, hvorav to er kabelskap, to er nettstasjoner og én er en trafostasjon. Disse er plukket for å demonstrere noen av de ulike situasjonene algoritmene kan bli brukt på. I de følgende underkapitlene blir resultatene vist fram og diskutert. Det er de samme stasjonene som er vist for begge algoritmene, det vil si Figur 16 og Figur 22 viser samme kabelskap, Figur 18 og Figur 24 viser samme nettstasjon og så videre.

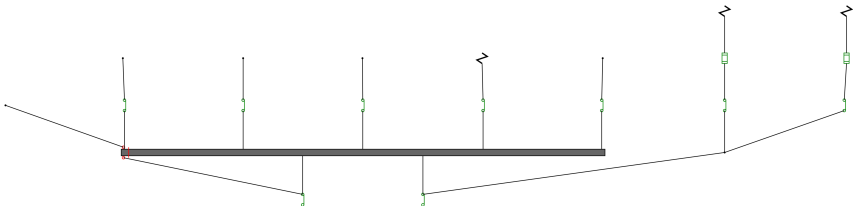
Alle figurene vist i rapporten er også med som SVG-filer i vedlegget.

15.1 Dot-algoritmen

Figur 16 og Figur 17 viser enlinjeskjema av to kabelskap som har fått layout fra dot-algoritmen. De får stort sett god layout. Unntaket er når det er ulik lengde fra inngangene til bussen, slik Figur 17 viser. Bredde-først-søket som lager en rettet graf besøker bussen før den andre komponenten som peker på den som er feilplassert. Hver gang det er ulik lengde på grenene inn til bussen vil denne situasjonen oppstå.

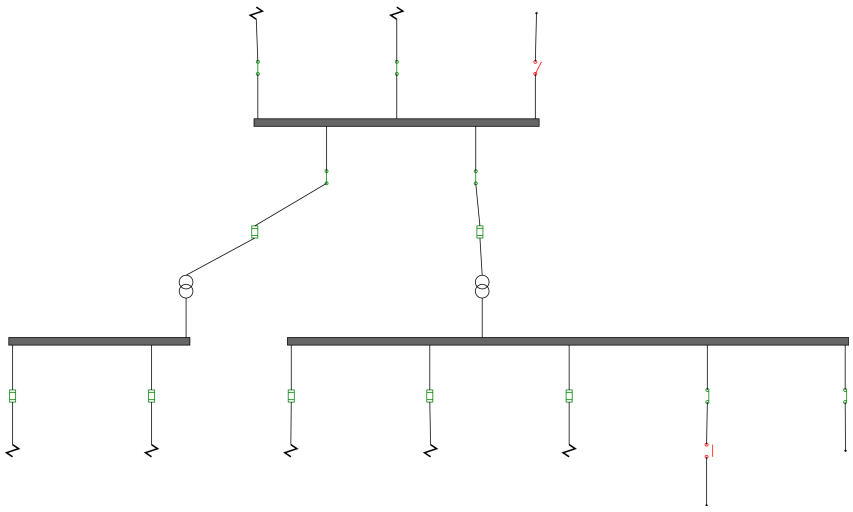


Figur 16: Et kabelskap med en buss, tegnet med dot

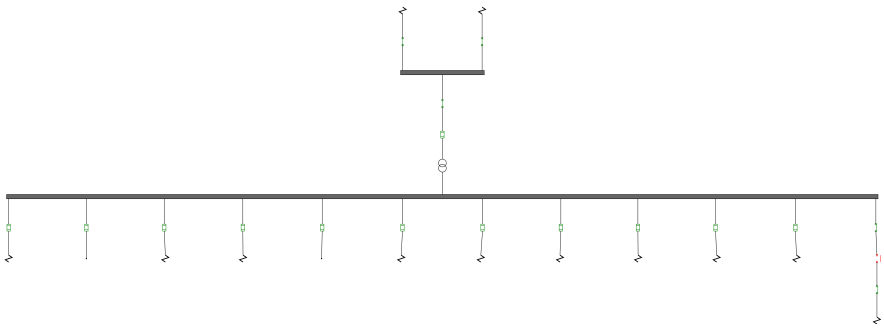


Figur 17: Et litt større kabelskap med en buss, tegnet med dot

Figur 18 og Figur 19 viser enlinjeskjema av to nettstasjoner som har fått layout fra dot-algoritmen. Begge får en god layout. Ingen komponenter overlapper, det er lett å se hva som er koblet sammen, det er ingen kryssende linjer. De to adskilte samleskinnene på det lavere spenningsnivået er til og med på samme rank. Det er kun fordi det er like mange komponenter mellom den øverste samleskinnen og de to nedre. Så lenge stasjonen har en slik symmetrisk oppbygning, blir resultatet fra dot-algoritmen veldig bra. Hvis oppbygningen ikke er symmetrisk, så har ikke dot-algoritmen noe som kompenserer for det, og busser og transformatorer vil havne på ulike høyder. Det er ikke et stort problem, siden oppbygningen og flyten i stasjonen kommer godt frem likevel.



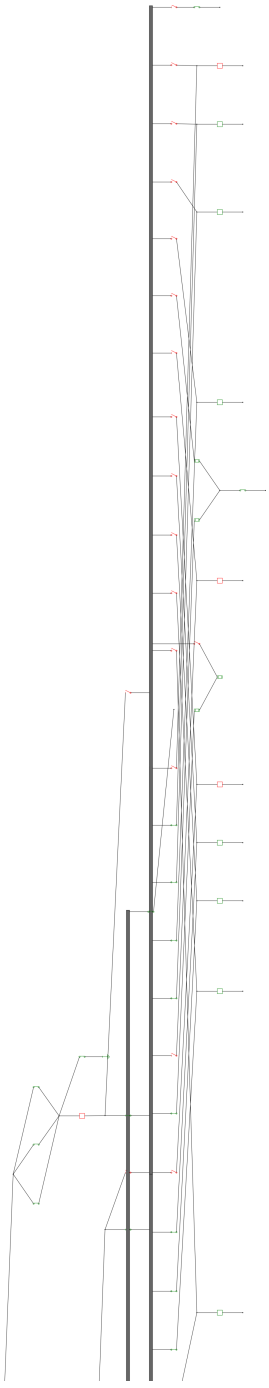
Figur 18: En nettstasjon med tre busser og to spenningsnivåer, tegnet med dot



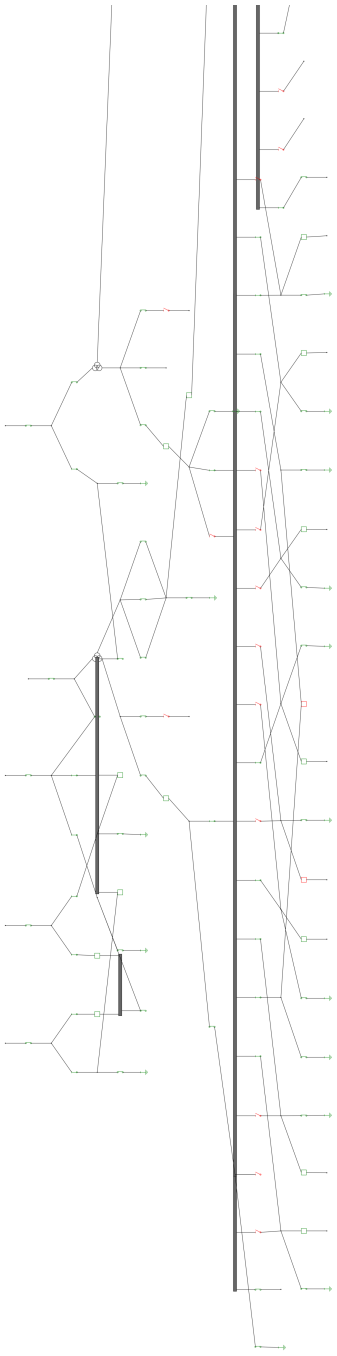
Figur 19: En nettstasjon med to busser og to spenningsnivåer, tegnet med dot

Layouten av den store trafostasjonen vist i Figur 20 og Figur 21 er ikke bra.

På begge de lavere spenningsnivåene overlapper de to bussene hverandre. Det kommer av at i dot-algoritmen er bussene kun et punkt, det er først når bussene erstattes med symbolet sitt at de får en større utstrekning. I de fleste mindre stasjoner går det fint, siden det ikke er plassert noe annet på samme rank som samleskinna. Bussene får samme rank siden det er like langt fra transformatoren til bussene. Ikke bare får det bussene til å overlappe, men det gir også mange kryssende linjer for komponentene som ligger mellom bussene. Grenene mellom de to bussene burde se ut som en vanlig dobbel-buss dobbel-bryter konfigurasjon, slik det er vist i Figur 6.



Figur 20: En trafostasjon med tre ulike spenningsnivåer, og dobbel buss, dobbel bryter konfigurasjon på alle spenningene, høyre side. Figuren er rotert 90 grader



Figur 21: En trafostasjon med tre ulike spenningsnivåer, og dobbel buss, dobbel bryter konfigurasjon på alle spenningene, venstre side. Figuren er rotert 90 grader

Hva som er dårlig med layouten fra dot-algoritmen kan oppsummeres i tre punkter:

- Mer kompliserte busskjema blir ikke pene.
- Symboler overlapper hverandre, spesielt rundt samleskinner.
- Transformatorer og busbar som burde ha samme rank er ikke det.

Korte linjer er ikke nødvendigvis det beste for et enlinjeskjema. Det skal være tydelig hva som er hvilke spenningsnivåer og hva som er innganger og utganger. Det er litt av grunnen til at dot-algoritmen ikke egner seg for mer kompliserte busskonfigurasjoner.

Ved å gjøre endringer i grafen som brukes i dot-algoritmen kan man kanskje få finere resultater. Noen forslag til punkter som kunne vært eksperimentert med for å få dot til å lage bedre layout:

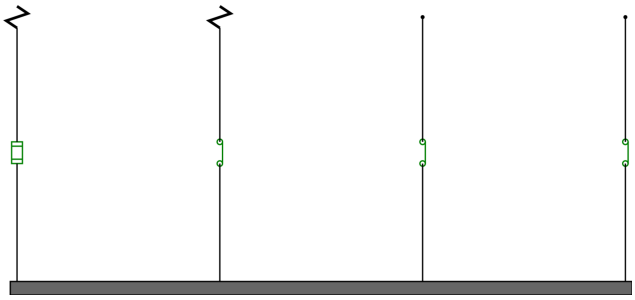
- Tilpassing av hvilken retning noen av kantene går, slik at den rettede grafen representerer hierarkiet og rekkefølge komponenter skal tegnes i bedre.
- Sette vekt på kanter, slik at noen kan tegnes lenger enn andre. Endring av vekt vil påvirke hvilken rank de ulike nodene får.
- Bruke subgrafer med `rank = "same"` for å spesifisere noder som skal ha samme rank.

Uansett er det vanskelig å "lure" dot-algoritmen til å tegne de mer kompliserte busskjemaene på samme måte som de gjerne tegnes manuelt.

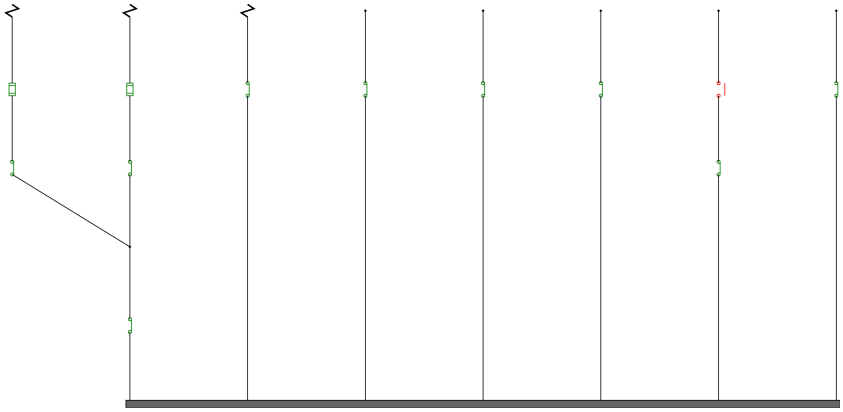
Siden de fleste stasjonene i et nett er små og enkle nettstasjoner og kabelskap, vil dot-algoritmen gi en god visualisering av en stor del av nettet. Dette er også de stasjonene det er minst sannsynlig at nettselskapet har tegnet skjematikk av manuelt.

15.2 Egen algoritme

Figur 16 og Figur 17 viser enlinjeskjema av to kabelskap som har fått layout fra den selvlagde algoritmen. De får en god layout.



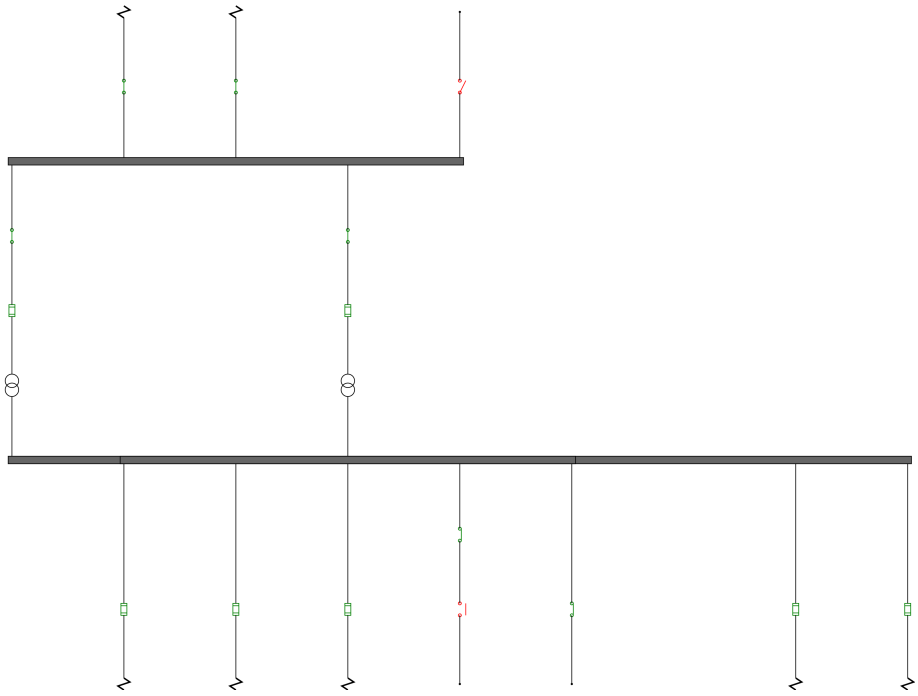
Figur 22: Et kabelskap med en buss tegnet med den nye algoritmen



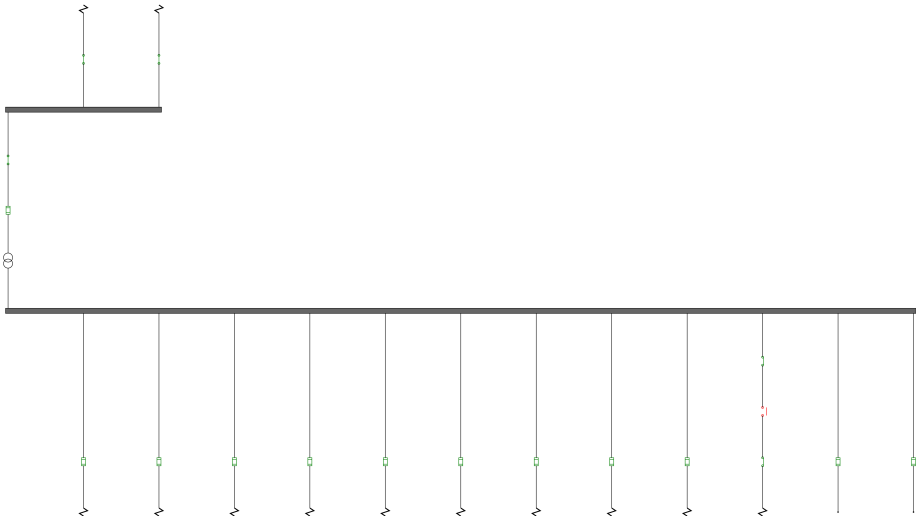
Figur 23: Et litt større kabelskap med en buss tegnet med den nye algoritmen

Figur 24 og Figur 25 viser to nettstasjoner. Layouten lagd med den egne algoritmen er dårlig. Hvis det er to separate busser, er det stor sjanse for at disse vil overlappe. Det kommer av at transformatorene får plasseringen fra layouten til det øverste spenningsnivået. Når bussene skal tegnes får de utstrekning fra x-koordinaten til transformatoren til x-koordinatene til resten av grenene, og siden plasseringen av transformatoren er avhengig av plassering på høyeste spenningsnivå kan det bli krasj. Det er ingen symmetri mellom de ulike spenningsnivåene.

Hadde det vært en komponent mellom transformatoren og bussen den er koblet til hadde resultatet blitt litt bedre, siden bussene ville ikke lenger overlappet. Layouten hadde fortsatt vært usymmetrisk og det er en sjanse for flere kryssende linjer mellom spenningsnivåene.

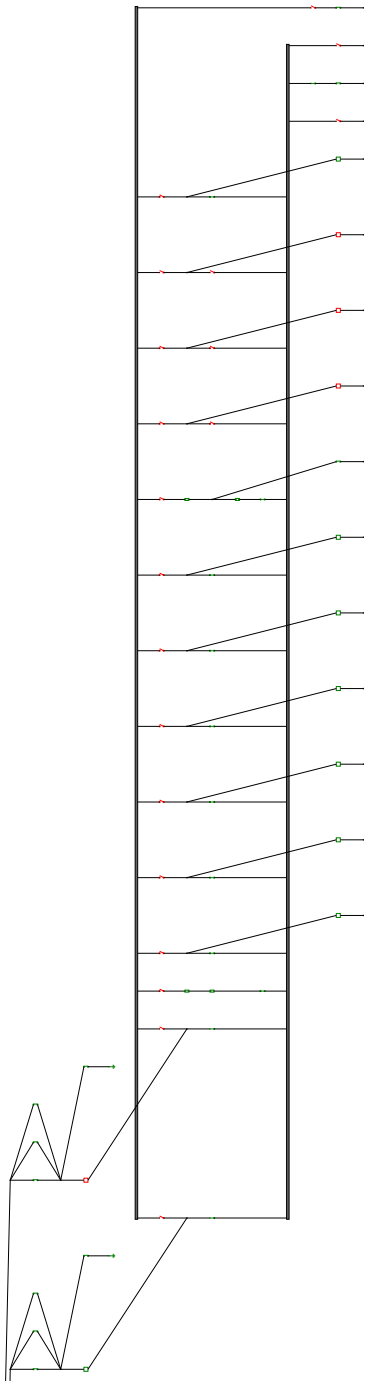


Figur 24: En nettstasjon med tre busser og to spenningsnivåer, tegnet med den nye algoritmen

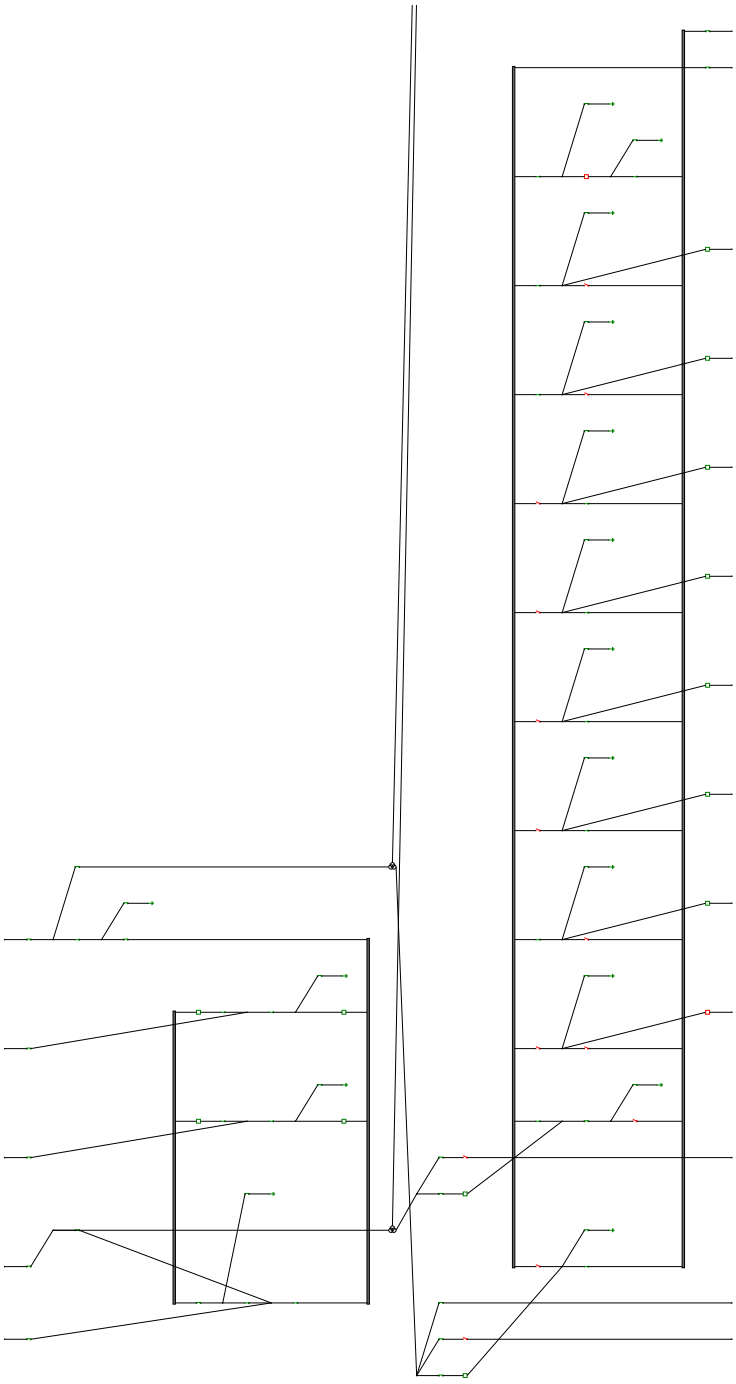


Figur 25: En nettstasjon med to busser og to spenningsnivåer, tegnet med den nye algoritmen

Den nye algoritmen gir god layouten av den store trafostasjonen vist i Figur 26 og Figur 27 . Det er enkelt å følge linjer for å se hva som er koblet sammen. Ingen komponenter overlapper. Det er tydelig hva som tilhører hvilket spenningsnivå, og hva som er innganger og utganger fra stasjonen. Layouten blir også bra for litt uvanlige konfigurasjoner, slik som det høyeste spenningsnivået. Det ligner på en dobbel-buss dobbel-bryter konfigurasjon, men har grener som kun er koblet til en buss.



Figur 26: En trafostasjon med tre ulike spenningsnivåer, og dobbel buss, dobbel bryter konfigurasjon på alle spenningene, høyre side. Figuren er rotert 90 grader



Figur 27: En trafostasjon med tre ulike spenningsnivåer, og dobbel buss, dobbel bryter konfigurasjon på alle spenningene, venstre side. Figuren er rotert 90 grader

Alt i alt er den egne algoritmen god på å tegne enkle stasjoner med kun en spenning, eller store stasjoner med dobbel-buss konfigurasjoner. De mellomstore stasjonene med to spenninger og flere separate busser får en dårlig layout.

Det er flere endringer man kan gjøre for å forbedre layouten:

- Finne ut hvor grener kan tegnes i samme kolonne. Nå får hver enkelt gren sin egen x-koordinat, og ingen andre grener får samme x-koordinat. I flere tilfeller gir dette mer tomrom enn nødvendig.
- Smartere plassering av transformatorene og grenene som er koblet til transformatorene. Dette vil spesielt hjelpe på nettstasjoner, der en god layout er mer avhengig av at bussene og transformatorene er på linje.
- Tegne rette linjer i stedet for de som går på skrå. Må også legge linjene smart slik at de ikke overlapper og går oppå hverandre.

I kapittel 13.1 ble det gjort fire antagelser for å forenkle generering av layout. Antagelse gjør at vi kan se bort i fra tilfeller som sjeldent forekommer. Basert på datasettet som har blitt brukt i denne oppgaven er disse antagelsene rimelige. I andre datasett kan det være anderledes. Da kan det være nødvendig å skrive om algoritmene for å ta høyde for slike tilfeller.

15.3 Sammenligning av layout-algoritmene

Enkle kabelskap der det er en spenning og en buss, tegnes stort sett likt av begge algoritmene. Det er bare forskjell i avstand mellom symbolene. Til og med rekkefølgen på komponentene er lik. For dot-layout vil det en sjelden gang være retning på noen kanter fra bussen i stedet for til, som gjør at layouten ikke blir så ren som den burde være. Den selvlagde algoritmen takler derimot dette bra, siden den rettete grafen der lages basert på ranks.

Den selvlagde algoritmen er dårlig på å tegne typiske nettstasjoner med to spenningsnivåer. Selv når det kun er en buss på alle spenningsnivåene gir dot-algoritmen enlinjeskjema som er mye mer symmetriske, og dermed også finere å se på.

Layouten av trafostasjonen blir mye bedre med den egne algoritmen enn med dot. Ved å ta inspirasjon fra manuelt tegnede enlinjeskjema blir utdelingen av ranks gjort på en måte som sprer komponentene mer i vertikal retning enn det dot-algoritmen gjør. Ved å behandle komponentene som er koblet sammen som grener og undergrener, blir komponentene i større grad plassert i samme kolonne.

Det vil alltid kunne være busskonfigurasjoner og oppbygninger av en stasjon som ikke følger de vanlige busskonfigurasjonene. I slike tilfeller kan det være vanskelig å lage en god layout, siden det kan være så varierende hvordan det burde tegnes. Det kan være nødvendig med en manuell jobb for å tilpasse enlinjeskjemaet i etterkant.

En mulighet er å kombinere eller veksle mellom dot-algoritmen og den egne algoritmen ettersom hva som gir best resultat. Dot-algoritmen gir gode resultater for nettstasjoner, mens den egne algoritmen gir gode resultater for kabelskap og trafostasjoner. Ved å gjenkjenne typen stasjon før layouten lages kan programmet bestemme hvilken algoritme som skal brukes for å generere layouten.

Algoritmene er ikke veldig robuste. Hvis et spenningsnivå ikke har noen busser vil den krasje. I testingen var det 10 av 1155 stasjoner som krasjet og ikke ble tegnet enlinjeskjema av.

Hvis tre eller flere komponenter utenfor stasjonen er koblet til samme konnektivitetsnode utenfor stasjonen, så vil ikke konnektivitetsnoden og kantene i grafen fjernes, slik de burde. Det gir spesielt problemer for den selvlagde algoritmen, som ikke ga de involverte grenene noen layout. Denne situasjonen oppstod i 19 av 1155 stasjoner. Dette må regnes som en bug, og bør fikses.

15.4 Andre diskusjonspunkter

Innlesingen av alle 18 cimxml-filene tok omtrent 20 sekunder. Det er ganske bra i forhold til størrelsen på nettet. Programmet tegner enlinjeskjema med begge layoutalgoritmene for alle stasjonene i løpet av fem minutter. Det inkluderer å lage den urettete grafen og tegning av SVG-fil. Det vil si at hver stasjon i snitt blir tegnet på under 0,3 sekunder. Det er raskt nok med tanke på at dette er en engangsjobb som ikke skal kjøres med jevne mellomrom. Hvis man ønsker/trenger at programmet er raskere bør programmeringsspråket byttes til et språk som er raskere enn Python.

En viktig faktor for programmet er at resultatet er deterministisk. Gitt samme graf som input, skal enlinjeskjemaet bli likt hver gang programmet kjøres. Både dot-algoritmen og den nye algoritmen gir det samme resultatet hver gang. Hvis inputen endres litt, for eksempel legge til en komponent, er det ingen garanti for at det nye skjemaet har samme rekkefølge på komponenter og koblinger til utsiden.

Programmet lager SVG-bilder som er godt egnet for å vise i en nettleser. Med mindre tilpasninger av programmet er det mulig å integrere SVG-bildene i en større nettapplikasjon. For de andre bruksområdene må det gjøres mer arbeid for å kombinere layouten av flere stasjoner til et enlinjeskjema.

I et enlinjeskjema av et overordnet nett burde layouten av komponentene inne i en stasjon være avhengig av koblingene med utsiden. Dette kan hindre mange linjer i å overlappe. Samtidig gjør det at intern og ekstern layout blir avhengig av hverandre, noe som gjør begge deler mer komplisert og tidskrevende.

16 Konklusjon

I denne oppgaven er det presentert to metoder for å tegne enlinjeskjema. Den ene benytter graflayout-algoritmen *dot*. Den gir gode resultater raskt for mindre og enkle stasjoner. Så snart dot skal tegne stasjoner med mer kompliserte busskjema får man overlappende busser og mange kryssende linjer.

Den andre layout-algoritmen er lagd hovedsaklig for å tegne stasjoner der bussene burde tegnes ovenfor hverandre i stedet for på samme rank, altså de som dot-algoritmen er dårlig på. Det gjelder for eksempel dobbel-buss dobbel-bryter konfigurasjon og "Breaker-and-a-half" konfigurasjon.

Layouten som genereres gir kun posisjoner for komponentene. I oppgaven er det også vist hvordan enlinjeskjemaet kan tegnes som et interaktivt SVG-bilde, basert på denne layouten. Det interaktive SVG-bildet burde være enkelt å integrere i en nettapplikasjon.

Begge algoritmene er deterministiske, så lenge de får samme input (oppbygningen av stasjonen) vil de tegne det samme enlinjeskjemaet.

Metoden presentert i denne oppgaven er rask nok til å kjøre på enkelte stasjoner i ikke veldig store nett. Hvis det skal tegnes enlinjeskjema av alle stasjonene i et større nett samtidig, vil denne metoden ta mange minutter.

I videre arbeid bør det ses på:

- Ytelsesforbedring: Bruke andre programmeringsspråk enn Python bør gi betydelig raskere kjøretider.
- Tegne grenene mer kompakt i den egne algoritmen, ved å la grener over en buss tegnes på samme x-koordinat som grener under bussen.
- Bedre sortering av grenene før de tegnes. Spesielt burde grenene koblet til transformatorer sorteres slik at de er sortert likt på alle spenningsnivå. Dette og det forrige punktet vil gi mer symmetriske enlinjeskjema.
- Tilpasse hvor utganger kan tegnes, slik at de ikke nødvendigvis må tegnes på nederste eller øverste rad. Dette gir noen ganger mange kryssende linjer som kunne vært unngått ved å gi plass for eksempel mellom to samleskinner.
- Finne en metode for å inkrementelt legge til nye komponenter til enlinjeskjema. Det skal da være gjenkjennbart det som allerede er lagt til, slik at det er lett for tidligere brukere å benytte det nye skjemaet.
- Gjøre algoritmen mer robust, og tilpasse den flere busskonfigurasjoner.

Kildehenvisning

[1] *Strømnettet*. Jan. 2019. URL: https://energifaktanorge.no/norsk-energiforsyning/kraftnett/. (Hentet: 07.11.2019).

[2] *Trefase*. URL: https://no.wikipedia.org/wiki/Trefase. (Hentet: 09.11.2019).

[3] *Difference between Circuit Breaker and Fuse*. URL: https://www.electricaltechnology.org/2019/07/difference-between-fuse-circuit-breaker.html. (Hentet: 12.12.2019).

[4] *Differences between Circuit Breaker and Isolator/Disconnector*. URL: https://www.electricaltechnology.org/2019/08/difference-between-circuit-breaker-isolator-disconnector.html. (Hentet: 12.12.2019).

[5] *6 common bus configurations*. URL: https://electrical-engineering-portal.com/bus-configurations-substations-345-kv. (Hentet: 12.12.2019).

[6] *Systemansvar - NVE*. URL: https://www.nve.no/reguleringsmyndigheten/systemansvar/?ref=mainmenu. (Hentet: 06.12.2019).

[7] *Fosweb Kraftsystemdata*. URL: https://www.statnett.no/Fosweb-Kraftsystemdata. (Hentet: 06.12.2019).

[8] *Shapefile*. URL: https://en.wikipedia.org/wiki/Shapefile. (Hentet: 14.12.2019).

[9] *GIS file formats*. URL: https://en.wikipedia.org/wiki/GIS_file_formats. (Hentet: 14.12.2019).

[10] Jostein Andreassen. *Scada /DMS / AMS / NIS: Begraper som har gått ut på dato?* Sept. 2017. URL: https://blogs.esmartsystems.com/no/scada-dms-ams-nis-begreper-som-har-gatt-ut-pa-dato. (Hentet: 04.12.2019).

[11] *Distibution Management System*. URL: https://en.wikipedia.org/wiki/Distribution_management_system. (Hentet: 07.12.2019).

[12] *Network Information System*. URL: https://en.wikipedia.org/wiki/Network_information_system. (Hentet: 07.12.2019).

[13] *Powel Netbas*. URL: https://www.powel.no/smarte-nettselskap/nettinformasjonssystem/Netbas-copy. (Hentet: 21.04.2020).

[14] Jing Honga et al. *Substation One-Line Diagram Automatic Generation and Visualization*.

[15] *Differences between disconnectors, load switches, switch disconnectors and circuit breakers*. URL: https://electrical-engineering-portal.com/disconnectors-load-switches-switch-disconnectors-cbs. (Hentet: 13.05.2020).

[16] Alan W. McMorran. *An Introduction to IEC 61970-301 & 61968-11: The Common Information Model*. University of Strathclyde, Jan. 2007.

[17] *PyCIM Github*. URL: https://github.com/rwl/PyCIM. (Hentet: 10.08.2019).

[18] *What is the difference between bitmap and vector images*. URL: https://etc.usf.edu/techease/win/images/what-is-the-difference-between-bitmap-and-vector-images/. (Hentet: 02.05.2020).

[19] *svgwrite Documentation*. URL: https://svgwrite.readthedocs.io/en/latest/. (Hentet: 02.05.2020).

[20] *Graph (abstarct data type)*. URL: https://en.wikipedia.org/wiki/Graph_(abstract_data_type). (Hentet: 08.11.2019).

[21] *Force directed graph drawing*. URL: https://en.wikipedia.org/wiki/Force-directed_graph_drawing. (Hentet: 08.11.2019).

[22] *Layered graph drawing*. URL: https://en.wikipedia.org/wiki/Layered_graph_drawing. (Hentet: 18.04.2020).

[23] *Graph drawing*. URL: https://en.wikipedia.org/wiki/Graph_drawing. (Hentet: 08.11.2019).

[24] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. Jan. 2015.

[25] Emden R. Gansner et al. *A Technique for Drawing Directed Graphs*.

[26] Andrea Leofreddi. *SVGPan library*. URL: https://code.google.com/archive/p/svgpan/. (Hentet: 06.12.2019).

NTNU

Kunnskap for en bedre verden