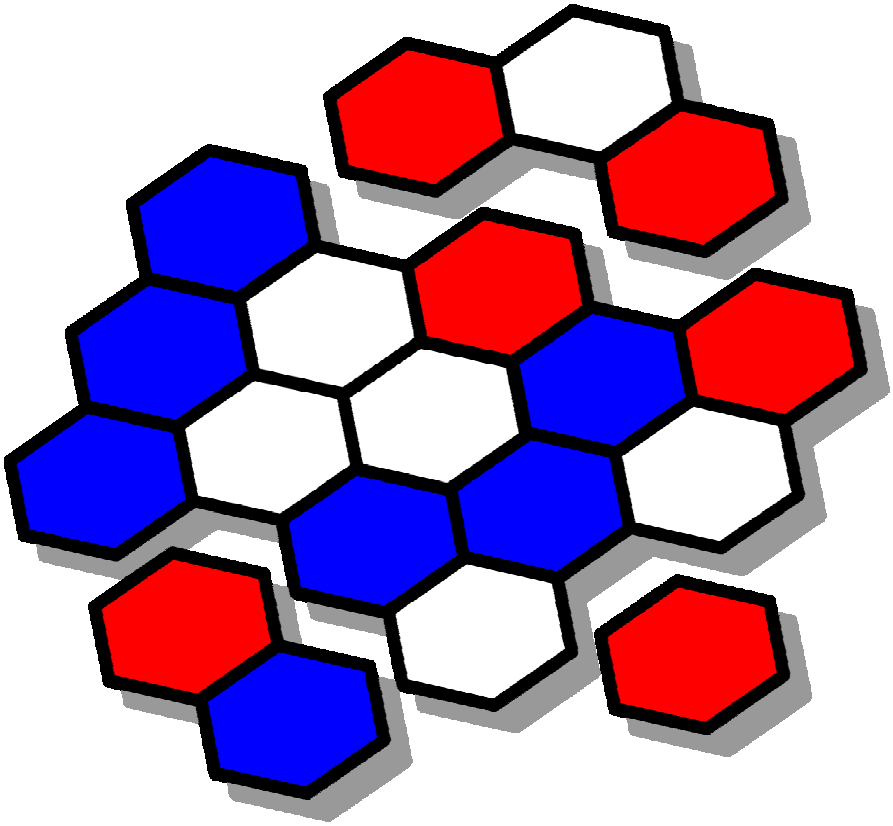
Den korteste vej til algoritmer og spil



af

Simon Larsen

&

Jens Frederiksen

Vejleder:
Anna Östlin Pagh

December 2004

IT – Universitet i København

0 Abstrakt

Denne rapport og den dertilhørende Java kildekode, er kulminationen på vores 16 uger programmeringsprojekt på IT – Universitetet i efteråret 2004.

Vi har undersøgt hvilke algoritmer og datastrukturer der skal anvendes for at lave en repræsentation af Hex spillet som Piet Hein og John Nash opfandt uafhængigt af hinanden for snart 50 år siden.

Vi har kigget på to forskellige *korteste vej* algoritmer; brede-først-søgning og Dijktras algoritme. For at finde den korteste vej er graf strukturen et naturligt valg, til at undersøge problematikken. Der er benyttet *disjunkte mængder datastrukturen* for at nemt at kunne undersøge om en spiller har skabt en forbindelse med spillerens to sider.

Hex spillet er blevet brugt som case for at udforske de algoritmer og datastrukturer, for at få spillet til at fungere. Vi har implementeret to versionen af spillet i Java, hvor den ene version er den som Hein og Nash opfandt og den anden er en version hvor alle felterne på spillebrættet har en vægt som har betydning for den korteste vej i fra den vindende spillerens to sider.

1 Rapportens overordnede inddeling

Hex Spillet

Kort om Hex spillet vi har brugt til at undersøge de forskellige aspekter ved at finde den korteste vej i et computer spil.

Applikationen

Hvordan er applikationen designet og hvilke overordnet metoder skal der bruges.
Applikationens grafik bliver også gennemgået.

Graf datastrukturer

Graf datastrukturen muliggør at benytte korteste vej algoritmerne. I afsnittet vil der være en gennemgang af den benyttede graf teori samt implementeringen af to forskellige graf typer; simple ikke-vægtede og vægtede grafer

Korteste vej

Korteste vej kan være vejen mellem to punkter som kræver færrest trin eller omkostning. Derfor er to korteste vej algoritmer beskrevet i teori og implementering.

Disjunkte mængder

Korteste vej algoritmerne kan bruges til at finde ud af om der er skabt forbindelse mellem en spillers to sider, men det er krævende algoritmer. Disjunkte mængder datastrukturen kan hurtigt afgøre om de to sider er forbundet. Derfor er disjunkte mængder strukturen benyttet. Strukturen er ligeledes beskrevet i teori og implementering.

Indholdsfortegnelse

Kildekoden til vores applikation er vedlagt som bilag.

2 Forord

Denne rapport og den dertilhørende kode skrevet i Java, er kulminationen på vores 16 ugers projekt som blev udarbejdet i perioden fra den 6. september til den 17. december 2004.

Der var flere formål med projektet, først og fremmest ville vi gerne blive bedre til at programmere ved ”learning-by-doing”. Sekundært ville vi gerne have en ”blid” indførelse i algoritmens verden, da ingen af os forinden havde beskæftiget sig med emnet, og da slet ikke hørt om begreber såsom Dijkstra, vægtede grafer og knuder.

Selvom vi forsøgte at begrænse projektets omfang så vidt muligt, opdagede vi hurtigt at algoritmernes store verden indeholder ufattelig mange emner som ved første øjekast virker overvældende. Projektet sendte os derfor på en sand ”odysse” i strukturering af data, hvilket vi ikke havde forstillet os ved projektets start. Ud over at vi har fået en forståelse for hvad f.eks. en hægtet liste er, har vi fået åbnet øjnene for vigtigheden af valget af datastrukturer ved implementeringen af algoritmer

Vi håber at denne rapport lever op til sin egen målsætning om at videreformidle viden om algoritmer og datastrukturer på en letforståelig måde så studerende efter os som ønsker at gå samme vej, kan have gavn af de opdagelser og erfaringer vi gjorde os undervejs.

Jens Frederiksen og Simon Larsen,
København, december 2004.

3 Indledning

Vores mål med projektet var at kigge nærmere på spillet Hex / Polygon Game som et klassisk graf problem inden for algoritmer og datastrukturer. Vi præsenterer vores implementering af disse ved hjælp af et simpelt userinterface udarbejdet i Java, hvor de underlæggende algoritmer og datastrukturer liggende bagved også programmet i Java.
Vi valgte programmeringssproget Java, da vi tidligere har erfaringer med dette og da et af projektets formål var at udforske objektorienteret programmering virkede dette som et oplagt valg.

Det overordnende mål i vores projekt er at opnå en grundlæggende forståelse af grafer, træer og datastrukturer, samt hvordan de benyttes og udarbejdes på en hensigtsmæssig måde i forbindelse med konstruktion af spil. Implementeringen af disse i Java er sekundært, og udarbejdelsen af et userinterfase tertiært.

Projektet er derfor todelt, i henholdsvis en teoretisk og en praktisk del. Den teoretiske del skal give os en indsigt i algoritmernes og datastrukturers overordnede opbygning. Den praktiske del vil bestå i en implementering af de teoretiske elementer i Java.

For både at få den teoretiske indlæring om datastrukturer og algoritmer og for at sikre en mere praktisk tilgang til emnet har vi anvendt spillet Hex som en case. Derigennem har vi direkte kunne afprøve og af- eller bekræfte noget af den læste teori.

3.1 Problemformulering

Hvordan sikre man at spillet Hex altid har styr på om der er en vinder og dernæst for at finde den korteste vej fra den vindende spillers to sider?

3.2 Afgrænsning

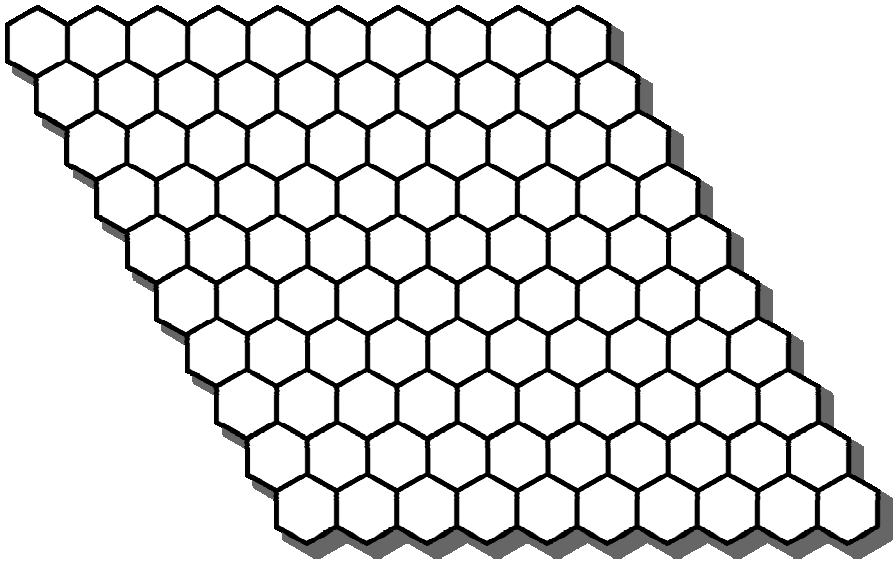
Hex spillet udspringer af matematikkens verden, vi vil dog ikke dække det matematiske aspekt i spillets udformning eller forskellige strategier. Vi bruger blot Hex spillet til at udforske datastrukturer og algoritmer.
Vi beskriver ikke alle benyttede datastrukturer i detaljer, f.eks. prioritetskøer. Ligeledes har vi ikke brugt de matematiske analysemodeller for datastrukturer og algoritmer, som f.eks. Stor O notation.

3.3 Målgruppe

Det er vores mål at give en beskrivelse af de teorier og emner, som vi gennemgår i rapporten på et abstraktionsniveau, hvor essensen er bibeholdt, men hvor de matematiske beviser for de forskellige begreber træder i baggrunden. Det er for at holde fast i vores mål om at skrive denne rapporten til vores valgte målgruppe; studerende som os, som ikke har haft undervisning i algoritmer eller data-strukturer og derfor ikke kan forventes at have nogen forkundskaber eller viden omkring emnet.

4 Hex spillets historie

Hex spillet opfundet af den danske matematiker Piet Hein i 1942 under navnet Polygon. Uafhængigt heraf opfandt den senere nobelpristager John Nash, spillet sidst i 1940'erne også spillet under navnet Nash. I 1952 lave spil firmaet Parker Brothers spillet til det kommercielle marked under navnet Hex, hvilket er blevet spillets navn i dag (Milnor, 2002, s. 31).



Figur 1: Et 10 x 10 spillebræt til Hex spillet.

Gennem tiderne har spillet dannet grundlag for utallige artikler og bøger, pga. af dens fremragende matematiske eksempler. Dog har spillet aldrig fundet samme indpas i vores samfund som f.eks. skak og andre logiske / strategiske spil. Dette skyldes muligvis at den startende spiller altid har en fordel og spillet derfor ikke kan betragtes som værende fuldstændig lige, som skak f.eks. er det[1].

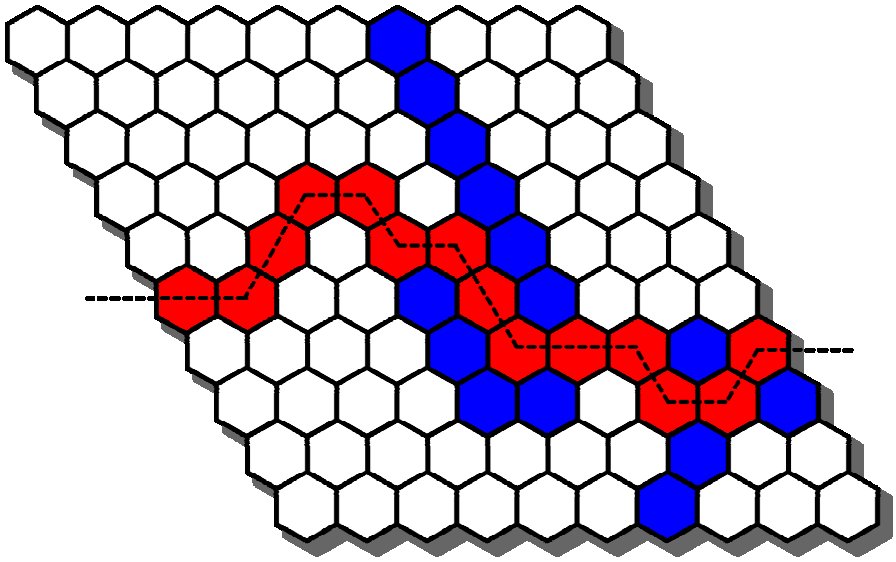
Det er vigtig at understrege at spillebrættet skal være lige bredt som det er højt. Ellers ville den ene spille (på den korte led) have en stor fordel, da det gælder om for begge spillere at forbinde henholdsvis øst og vest siderne eller nord og syd siderne. Dette anvendes dog flere steder for at skabe lidt balance i spillet, da den startende spiller som nævnt har en fordel. Andre tiltag for at gøre spillet mere balanceret indebærer at den ikke-startende spiller må vælge farve *efter* den første brik er blevet sat.

[1] Der findes mange forskellige strategier for at starte. John Nash menes at have fundet en vinderstrategi der indebar at den startende spiller *altid* ville vinde. Dog bygger denne strategi på matematiske modeller, som ikke er videnskabelig anerkendt som værende sikre. På Wolfram's MathWorld sås der også kraftig tvivl om dette bevis holder vand på spilbræt som er over 7x7 store (Eric W. Weisstein / MathWorld). Men sikkert er det at den startende spiller som regel altid har større chance for at vinde spillet. Det kan alle som har prøvet spillet skrive under på.

5 Regler for Hex spillet

Spillet Hex har nogle meget enkel regler. Der er to spillere, som skal hver især skal skabe forbindelse fra den ene side til den anden. Spillebrættet er udformet med heksagon formet felter, hvilket skaber et diamantformet bræt. Brættet er oftest i størrelsen 7x7 – 14x14 felter[2].

Når spillet starter er alle felter tomme. Hver spiller sætter derefter en brik efter tur og det glæder derefter for hver spiller at forbinde hans / hendes sider med hinanden med én lang ubrudt linie af sine brikker. Rød spiller skal forbinde den østlige og den vestlige side, og blå spiller skal forbinde den nordlige og sydlige side. Det er frit for begge spillere at sætte sine brikker hvor end man ønsker, blot man kan man ikke sætte en brik i et felt hvor der allerede ligger en brik. Spillet kan af den grund aldrig ende uafgjort, da spillet slutter når den ene spiller har forbundet sine sider og derfor har én ubrudt linie fra side til side. Det umuliggøre med andre ord at den anden spiller kan forbinde sine sider.



Figur 2: Rød spiller har nu dannet en forbindelse fra øst til vest og det er derfor ikke længere muligt for blå spiller at vinde.

[2] John Nash fandt frem til at det størrelsen 14 x 14 gav den bedste spil oplevelse (Milnor, 2002, s. 31).

6 Design af applikationen

Her følger en kort beskrivelse af hvordan vi har designet for applikation, og hvordan grafikken bliver tegnet på skærmen. Senere i rapporten følger en beskrivelse af hvordan de underliggende datastrukturer og algoritmer er opbygget og hvordan de anvendes.

6.1 Foranalyse

Disse overordnede ting skal indgå i designet af applikationen for at den skal fungere.

Brættet og brikkerne

- Et spillebræt af størrelse *n* x *n* med blanke felter
- Blå og røde brikker

Spillerne

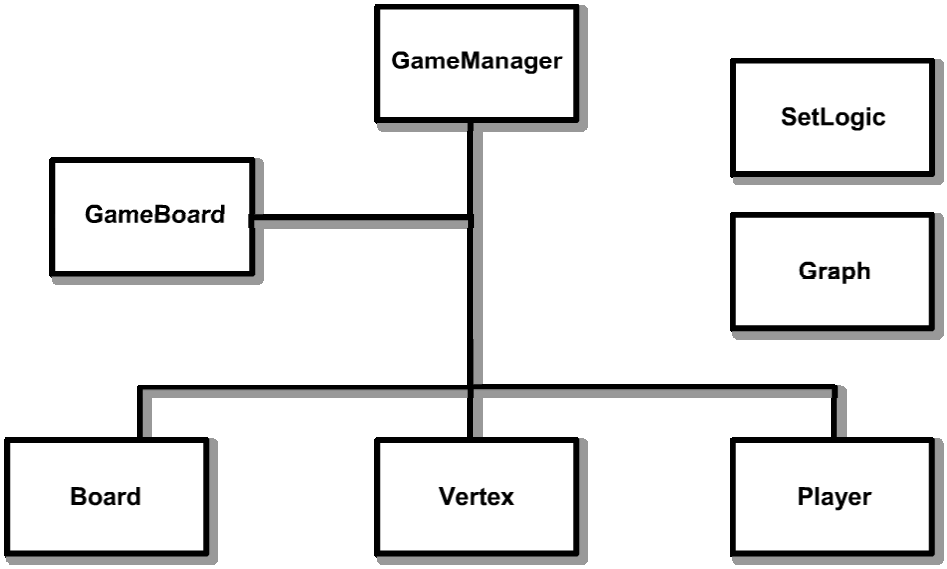
- En metode som skifter spillernes tur.
- En metode som registrerer når en spiller klikker på et felt i spillebrættet, sætter en af spillerens brikker og tildeler dette felt til denne spiller.

Spillets status

- En metode som forbinder en spillers brikker med hinanden.
- En metode som checker om en af spillerne har vundet
- Hvis en spiller har vundet skal der være en metode som finder den korteste vej fra den vindende spillers sider.
- En metode som afslutter spillet.

6.2 Klasserne og deres opdeling

I et forsøg på at gøre vores program så overskueligt som muligt har vi opdelt klasserne i små klasser som kun indeholder enkelte metoder eller data.



Figur 3: Simplificeret klasse diagram for Hex applikationen.

Denne arkitektur indebærer at nogle af klasserne bliver såkaldte data klasser. Deres funktioner er kun at være data bærende (indeholdende programmets status). Strukturen for dataklasserne er opbygget således at de fungerer som en form for database for spillet. Det er her at alle informationer om henholdsvis brættet, knuderne og spillere ligger.

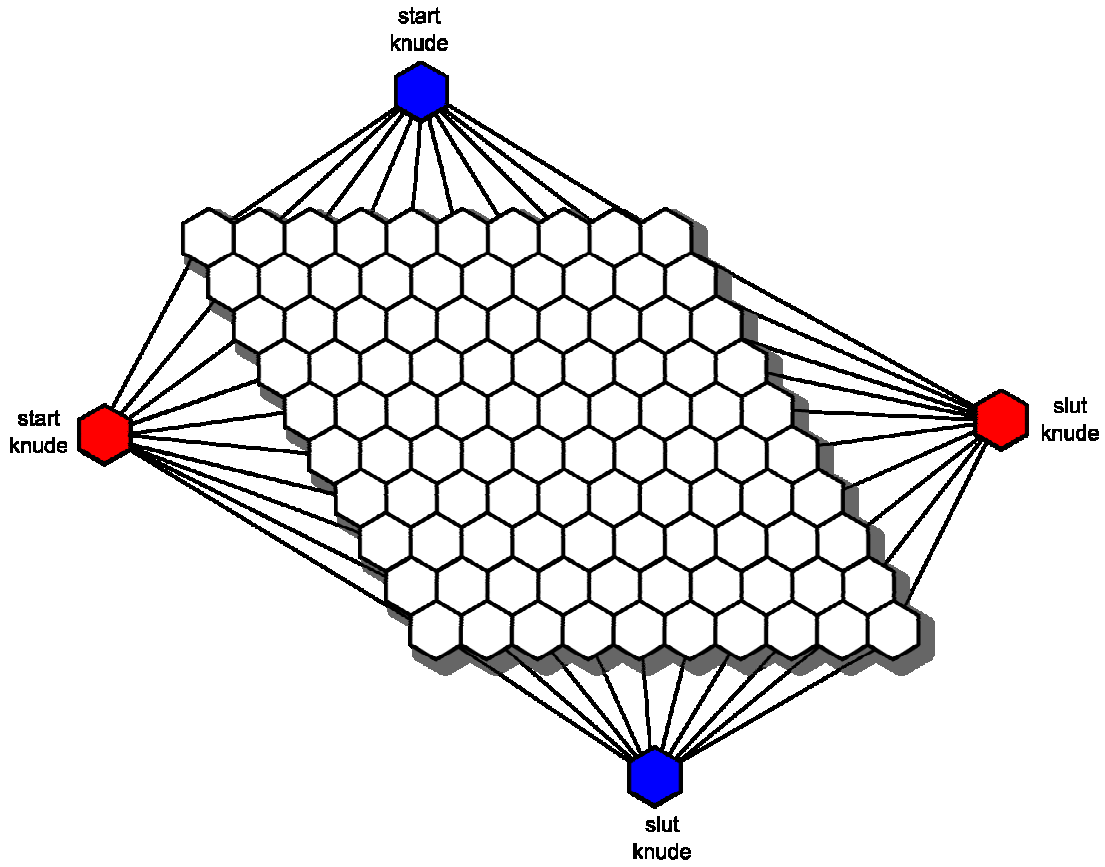
Alle metoder som omhandler beregninger ligger i kontrol klasserne, såkaldte logik klasser, som er statiske[3]. Det er her at alle beregninger på brugerinput bliver fortaget, og resultatet af disse beregninger bliver sendt til data klasserne via getters og setters. På denne måde opnår man at have et ”mellemlag” mellem brugeren og programmets data som hele tiden kontrollere data som sendes frem og tilbage, deraf navnet kontrol klasser. Brugeren kan altså ikke tilgå data i programmet direkte.
Selve tegningen af spillebrættet bliver tegnet er også placeret i sin egen klasse. På den måde er det helt afkoblet fra kontrol klasserne, og på denne måde kan man udskifte hele det grafiske element i applikationen i fremtidige programmer uden at skulle ændre på de underlignende kontrol- og data klasser.

Alle klasserne bliver styret af GameManager klassen, som er den klasse som hele tiden holder styr på spillets status og søger for at de underliggende datastrukturer opdateres. Det er GameManager klassen som er selve hjertet i applikationen. Al interaktion til og fra brugerne går gennem denne klasse.

[3] Grunden til at vi har valgt at lave kontrolklasserne statiske er at man på den måde ikke behøver at oprette instanser af dem, men blot kan tilgå dem direkte.

6.3 Konstruktion af spilbrættet

I vores implementering af Hex spiller har vi lavet et 14 x 14 bræt hvorpå spille foregår. For hele tiden at check om en af spillerne har vundet har vi indsat en yderligere virtuel brik som er forbundet til alle felterne i deres pågældende side.
Hvis f.eks. blå spiller sætter en brik i en af de sider som han/hun skal forbinde bliver den virtuelle brik som ligger uden for brættet også forbundet til denne nye brik. Algoritmerne i programmet checker derfor for hver satte brik om de virtuelle brikker nu er forbundet af én ubrudt forbindelse.

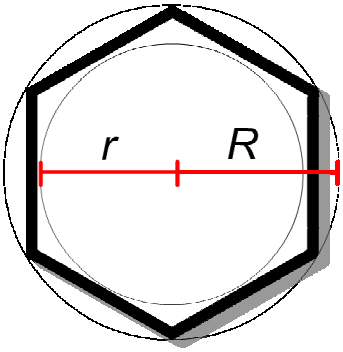


Figur 4: Spillebrættet for Hex og de fire virtuelle brikker.

6.4 Tegning af grafikken

For at tegne en heksagon i Java skal man benyttes sig af drawPolygon metoden fra Graphics klasse fra AWT biblioteket (java.AWT.graphics). Den skal have to heltal array og et antal punkter. Det ene heltal array indeholder x positionerne for punkterne og den anden y positionerne. Antallet af punkter er selvsagt 6, som antallet af punkter i en heksagon. Men for at finde punkternes *position* er det først nødvendigt at udregne deres indbyrdes afstand.

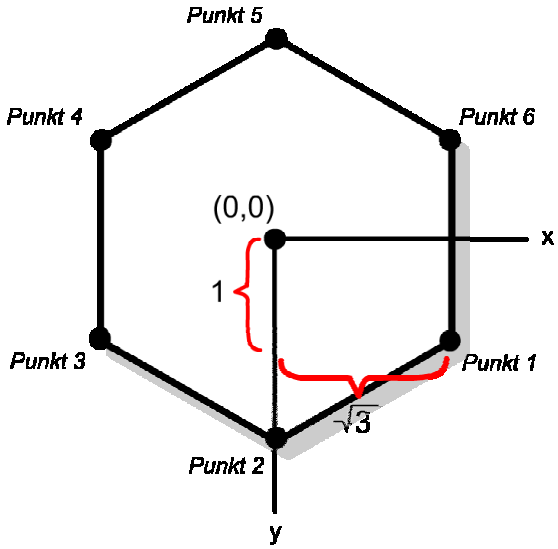
Ifølge Eric W. Weisstein på Wolfram's MathWorld vil en heksagon altid[4] have en indre radius $\frac{1}{2}\sqrt{3}$ og en ydre radius på 1. Den indre radius betegnes med *r* og den ydre radius med *R*.



Figur 5: En heksagons indre og ydre radius.

Ved at gange både den indre og den ydre radius med 2 for at få deres respektive diameter. Kender man diameteren kan punkterne plottes ind efter den nedenstående tabel. Heksagonen man tegner bliver da dobbelt så stor som enhedsheksagonen.

	X	Y
Punkt 1	$\sqrt{3}$	1
Punkt 2	0	2
Punkt 3	$-\sqrt{3}$	1
Punkt 4	$-\sqrt{3}$	-1
Punkt 5	0	-2
Punkt 6	$\sqrt{3}$	-1



Figur 6: Udregning til placering af punkterne.

For at disse tal skal kunne anvendes i Java (der som nævnt skal have heltal array til at tegne en polygon) skal disse tal ganges med et stort tal, ellers bliver de forkortet for meget i forbindelse med at de bliver type-castet fra decimal tal til heltal.

[4] Det matematiske bevis for hvor den indre og de ydre radius altid er disse værdier, ligger uden for denne rapports område og vil derfor ikke blive gennemgået heri.

Næste trin er at udregne hvor disse punkter skal placeres. For at få dem til at blive tegnet ordentligt i f.eks. et 14 x 14 spil er det også nødvendigt at forskylde heksagoner lidt når de tegnes. Ellers ville mellemrummet mellem brikker blive for stort og den ønskede grafiske fremstillingen ville ikke opnås. Dette kan gøres i forbindelse med at man tegne brættet. I vores tilfælde tegnes det ved at lave en dobbelt *for-løkke* på 14 x 14, som tegner en enkelte heksagon ved hvert gennemløb. Ved hvert gennemløb bliver der lagt et tal til x og y positionerne for på den måde at forskyde dem.

Denne måde til at tegne heksagoner i Java er kraftigt inspireret af Arthur Vauses metode som han anvender i sin egen version af spillet Hex. Vauses' version kan findes på denne hjemmeside: http://web.ukonline.co.uk/arthur.vause/Hex.html

6.5 To spil i et

I vores applikation er det både muligt for at spille et normalt Hex spil og så en modificeret udgave hvor alle felterne har fået en vægt. I den oprindelige version af Hex har alle felter i spillebrættet samme værdi / vægt.

Vi har konstrueret applikationen på denne måde for også at kunne arbejde med vægtede grafer og de søge algoritmer som hører dertil. I de efterfølgende afsnit vil de to version af spillet indgå som eksempler hvorpå vi har anvendt den beskrevne teori i praksis.

7 Grafer

Bagved alt grafikken i spillet ligger nogle datastrukturer og nogle algoritmer som holder styr på spillets status. For at finde en vinder i spillet og den korteste vej, er graf strukturen et naturligt valg, eftersom denne datastruktur er en direkte analogi til vores opfattelse af veje mellem to eller flere punkter. Dernæst har matematikken allerede skabt effektive strategier for at beregne om to punkter er forbundet eller for at finde den korteste vej ud fra en graf data struktur. Derfor er graf data strukturen god til at finde den korteste vej i Hex spillet.

Grafer kan have to betydninger indenfor matematikkens verden, grafer for funktioner og grafer som beskriver en mængde punkter og deres forbindelser[5]. Det er den sidste beskrivelse af grafer, der er udgangspunktet for denne rapport.

7.1 Generelt om grafer

Den overordnede graf teorien blev introduceret i det 17. århundrede af den schweiziske matematiker Leonhard Euler for hans løsning af Königsberg bro problemet[6] (Rosen, 1999, s. 475). I dag bruges grafer indenfor mange fagområder for at forstå og modellere problematikker f.eks. computer netværk, transportveje, flyruter m.m.

Helt formelt kan grafer defineres på følgende måde[7]:

En graf G = (V, E), består af en mængde knuder (eng.: vertices) V = {v_1, v_2, … v_n} og en mængde kanter (eng.: egdes) E = {e_1, e_2, … e_m}. En kant e forbinder to knuder u og v: e = (u, v).

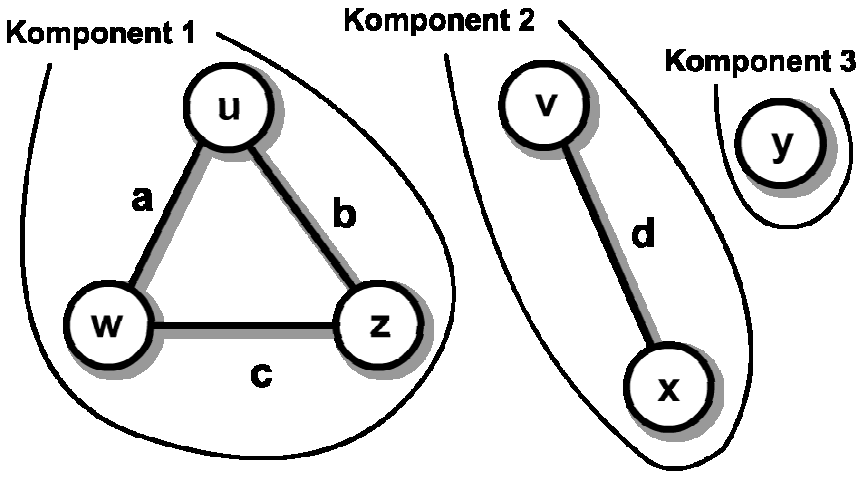
Dette er en meget komprimeret måde at beskrive en graf på. Det som man skal lægge mærke til en den ovenstående definition, er ordene *mængde*, *knuder* og *kanter*.
Mængde henviser til det faktum at en graf *kan* bestå af ingen eller flere knuder og kanter. *Knuder* henviser til de punkter eller felter om man vil, som er i den graf man arbejder med. Og *kanterne* henviser til de forbindelse der er mellem disse knuder.

[5] Kilde: Wolfram's MathWorld (http://mathworld.wolfram.com/Graph.html)
[6] Leonhard Fuler blev kendt for at komme med en løsning på det som senere skulle komme til at bære hans navn: Euler Paths. Problemet gik i sin enkelthed ud på, at han ville finde en rute gennem byen Königsberg i Kalininggrad (som dengang hed Prussia) hvor man kunne gå én lang tur gennem byen, og krydse alle byens broer én gang og komme tilbage til sit udgangspunkt. I 1736 publicerede Fuler en løsning på dette problem. (Rosen, 1999, s. 475-75).
[7] Denne definition er taget ordret fra Flemming Koch Jensens website DocJava.dk (http://www.docjava.dk/datastrukturer/grafer/grafer.htm)

En graf er en mængde med punkter (V) og en mængde forbindelse (E) mellem disse punkter. Disse mængde kan være tomme, dog er mængden med kanter altid tom (E = Ø[8]), hvis mængden med knuder er tom (V= Ø).

7.1.1 Komponenter

I en graf kan der være knuder, der ikke er forbundet til andre knuder. Disse knuder kaldes *isolerede knuder*, f.eks. vil alle knuderne i en graf være *isolerede*, hvis E = Ø og V ≠ Ø. Isolationen kan også forekomme mellem knuder, som er forbundene med andre knuder.



Figur 7: En graf med flere komponenter.

De knuder der er forbundet med hinanden kaldes komponenter. I Figur 7 findes der tre komponenter. Grafen G består derfor af V={u,w,z,v,x,y} og E={a,b,c,d}, hvor komponent 1 består af V={u,w,z} og E={a,b,c}, komponent 2 består af V={v,x} og E={d} og komponent 3 består af V={y} og E{Ø}.

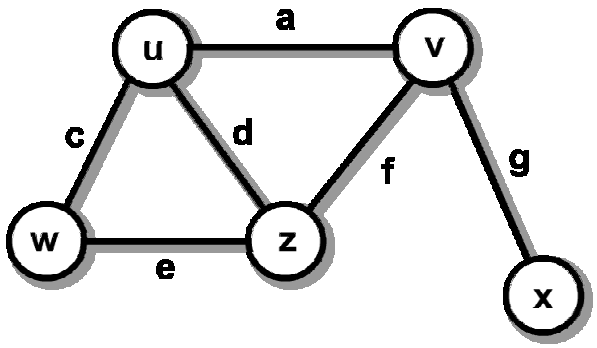
Denne anskuelse af grafstrukturen, som bestående af komponenter bruges ved disjunkte mængder datastrukturen. Disjunkte mængder datastrukturen er implementeret i Hex spillet og bliver beskrevet senere i afsnittet om disjunkte mængder på side 36.

[8] Ø betyder tom mængde

7.2 Forskellige typer grafer

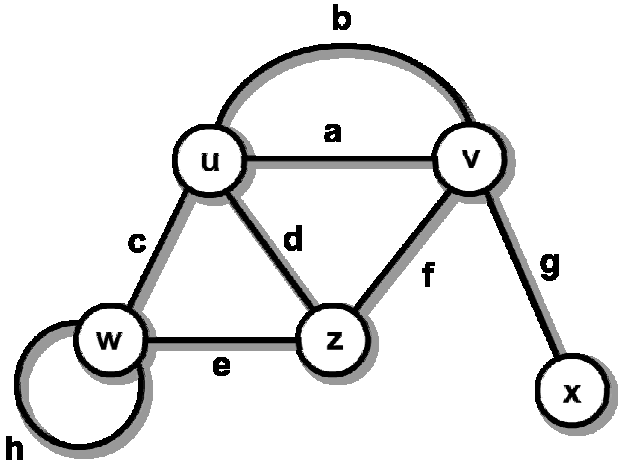
Grafer kan indeholde forskellige informationer om strukturen. Her er en kort præsentation af 6 forskellige grafer med hver deres fokus af informationer indenfor 3 kategorier.

Simpel graf: Afspejlinger forbindelse mellem grafens knuder, med en kant mellem knuderne, selvom der muligvis er flere forbindelser, eller som oftest en forbindelse som går begge veje, f.eks. kant c = (u, w) og (w, u).



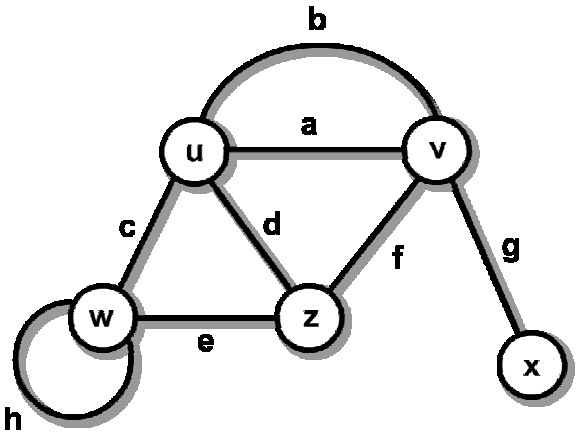
Figur 8: Simpel graf.

Multigraf: Afspejlinger antallet af forbindelse mellem grafens knuder, med en kant per forbindelse. Multigraf kan også indeholde graf-loops, hvor en knude har en kant til den selv[9].



Figur 9: Multigraf.

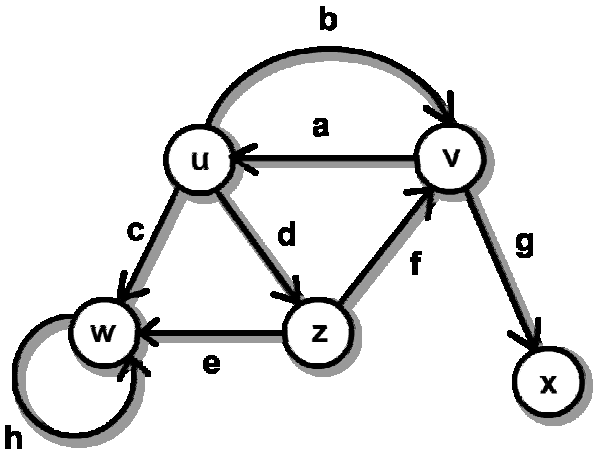
Uorienteret graf: Afspejlinger forbindelse mellem grafens knuder, med en kant mellem knuderne, er kanten underforstået som pegende begge retninger, f.eks. kant c = (u, w) og (w, u). Uorienteret graf kan bruges til at modellere f.eks. telefonlinier.



Figur 10: Ikke-orienteret graf.

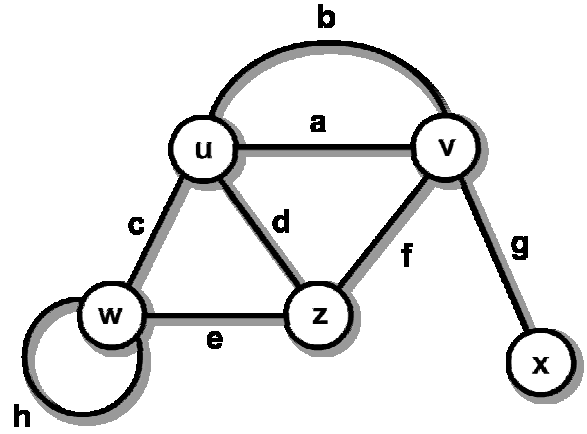
[9] Der findes ingen enighed om der må forekomme graf loops i multigrafer, nogle tillader det, mens andre operer med pseudo grafer der er multigrafer, som kan indeholde graf loops. Se endvidere Wolfram's MathWorld (http://mathworld.wolfram.com/Multigraph.html)

Orienteret graf: Afspejler hvilken retning kanten har, f.eks. kant c = (u, w), man kan kun bevæge sig fra u til w og ikke omvendt $c \neq (w, u)$.
Orienteret graf kan bruges til at modellere f.eks. flyveruter.



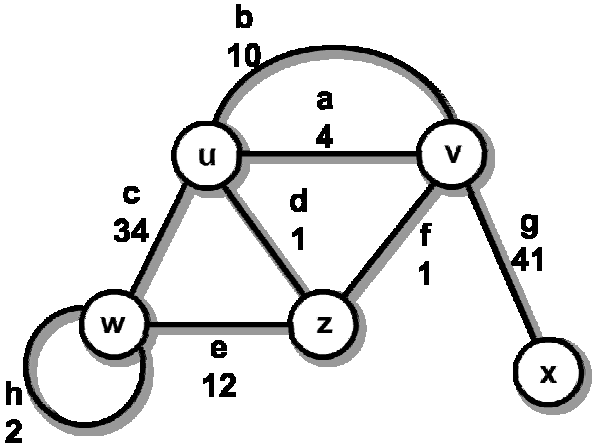
Figur 11: Orienteret graf.

Uvægtede grafer: Afspejler kun at der er en kant mellem to knuder. Grafen kan være ikke-orienteret eller orienteret.
Uvægtede grafer kan bruges til at modellere den korteste rute.



Figur 12: Uvægtet graf.

Vægtede grafer: Afspejler omkostningerne ved at benytte en kant, f.eks. c er 34, det koster altså 34 at bevæge sig fra u til w. Grafen kan være ikke-orienteret eller orienteret.
Vægtede grafer kan bruges til at finde den mindst omkostningsfulde vej i f.eks. i et strategispil hvor forbindelserne mellem de forskellige felter på et kort ikke har samme værdi, f.eks. at nogle felter er bjerge, søer eller flade sletter.

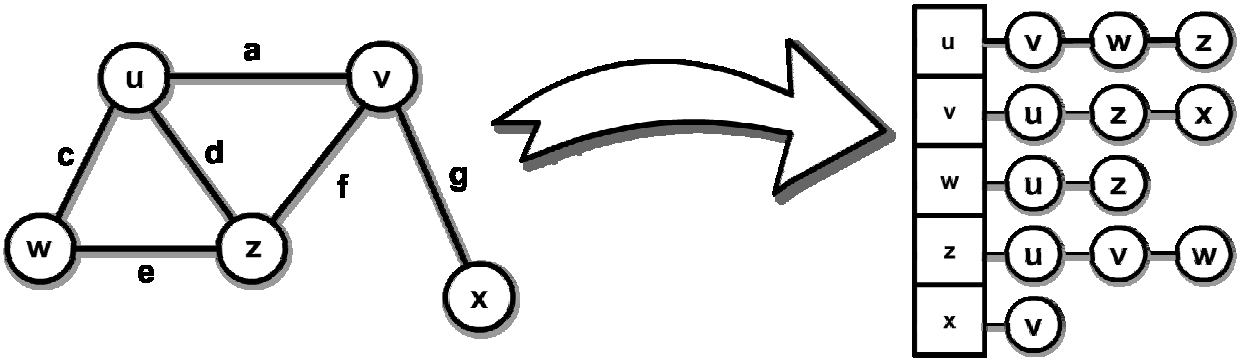


Figur 13: Vægtet graf.

I implementeringen af Hex spillet benyttes kun simple uorienterede grafer med og uden vægte.

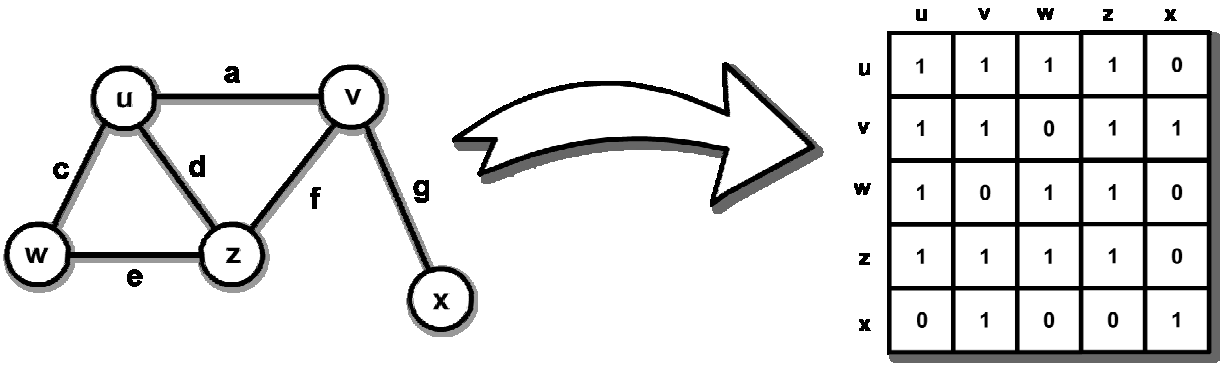
7.3 Repræsentation af grafer

For at kunne bruge grafer, kræver det et værktøj til at repræsenter grafer på. Grafer kan repræsenteres på to måder; Nabo list og Nabo matrice, som hver har forcer og ulemper. En Nabo liste består af grafens knuder, hver har en liste med pointers til alle de knuder den er forbundet til. Derfor er et naturligt valg at implementere en knudes liste med hægtede lister.



Figur 14: Et eksempel på en nabo liste.

En Nabo matrice er en vektor, hvor 0 betyder ingen forbindelse og 1 er forbindelse. Hvis det er en multigraf kan en Nabo matrice angive antallet af kanter mellem to knuder.



Figur 15: En multigraf med en nabo matrix.

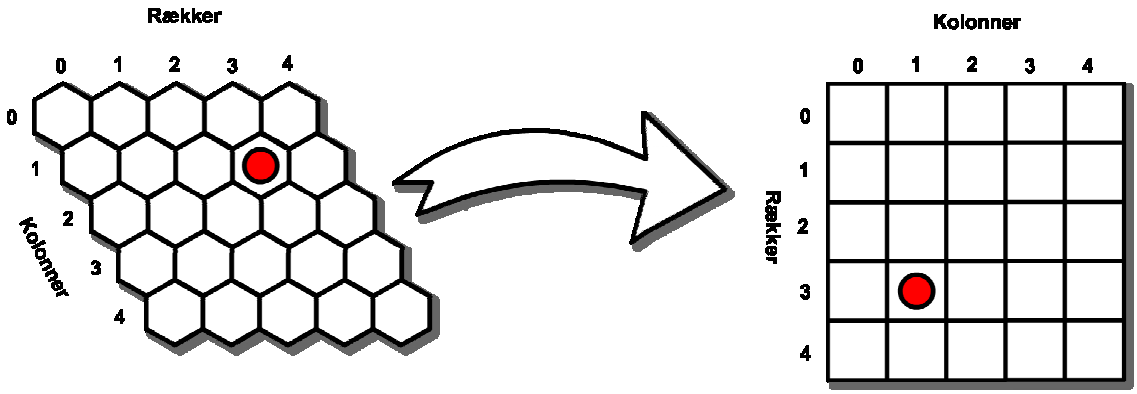
7.3.1 Valget af lister eller matrix.

En nabo list er god til grafer med få kanter[10], da hver knudes liste bliver kort og dermed både plads besparende og hurtig at søge i. Nabo matrix er derimod gode til grafer med mange kanter[10], fordi matricen er hurtig og nem at inspicere. I tilfældet af en *tæt graf* er en nabo liste meget langsom at søge i, når der skal findes en knude i en liste. Dernæst optager en nabo liste plads for mængden af knuder (V) samt mængden af kanter (E). Det vil betyde at en multigraf vil fylde meget i en nabo liste, men mindre i en nabo matrix. Nabo matrix fylder nemlig mængden af knuder opløftet i anden (V^2) pga. knuderne repræsenteres i kolonner og i rækker.

[10] Graf med få kanter kaldes tynd (sparse) og en graf med mange kanter kaldes tæt (dense) (Cormen, et al, 1999, s. 465).

7.4 Implementering af graf struktur

For at holde styr på spillets tilstand har bruges et dobbeltarray hvor alle brikkernes status noteres. Dette array er kvadratisk og der kræves derfor en speciel algoritme for at finde nabobrikker i et sådan array når de enkelte brikker *ikke* er kvadratiske.



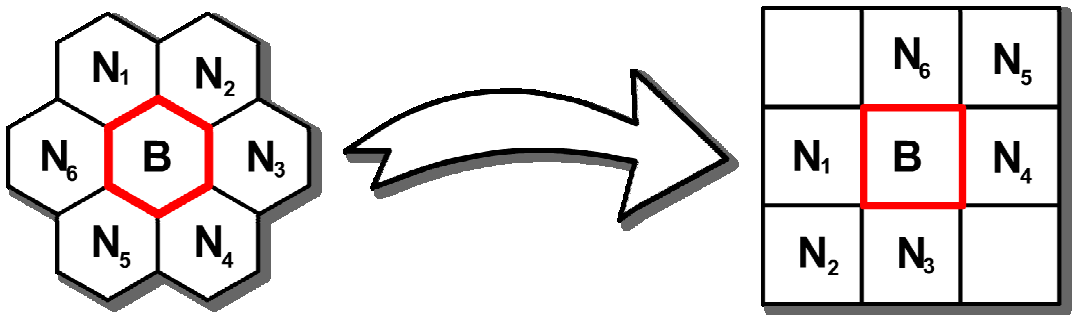
Figur 16: Afspejling af det heksagonformede spilbræt i et 2D array.

For at afbillede et heksagonformet bræt i et 2D array skal man have for øje at kolonnerne og rækkerne bliver bytte om. Det skyldes måden som heksagonerne bliver tegnet på som beskrevet i afsnit 6.4 på side 13.

7.4.1 At findes sine naboer

Nå spilet er i gang er det altafgørende for spillets logik at der hele tiden holdes styr på hvilke brikker der er forbundet, for på den måde at afslutte spillet, når en spiller har én ubrudt linie som forbinder spillernes respektive sider.

Når der sættes en brik skal programmet derfor finde ud af om der i denne briks omkreds er andre brikker fra samme spiller. Dette gøres ved at spørge til de seks sider som heksagonen har. Disse seks siders placering kan da findes i det føromtalte 2D array således og der kan det ses om der rent faktisk er andre brikker fra samme spiller.



Figur 17: Naboer (N_n) til den nye satte brik B.

Koordinaterne for disse seks sider er, relative til B:

Nabo 1	x + 0	y + (-1)
Nabo 2	x + 1	y + (-1)
Nabo 3	x + 1	y + 0
Nabo 4	x + 0	y + 1
Nabo 5	x + (-1)	y + 1
Nabo 6	x + (-1)	y + 0

7.4.2 Kanterne

Graf strukturen er implementeret i Hex spillet, ved at der internt i alle knuderne ligger en nabo liste med knudens naboer. En knudes nabo list vedligeholdes af den ovenstående metode, kaldes hver gang en brik bliver sat, og en knude bliver tildelt en spiller.

I metoden forbindes knuden med dens naboer ved at opdatere knudens og dens naboers nabo list. En knudes naboer med samme spillerfarve bliver lagt ind i knudes nabo liste og nabo listen for hver nabo med samme spillerfarve bliver opdateret med den nye knude.

```
Input: Knude, naboer
1   Så længe der er naboer, gør følgende for hver nabo
2       Lav en reference fra knudens nabo list til naboen
3       Lav en reference fra naboens nabo list til knuden
```

Pseudokode 1: Kanterne.

Derved er kanterne i Hex spillet repræsenteret ved pointers fra knude til knude. Denne implementering bevirker at kanternes vægt beskrives ved det indbyrdes forhold mellem knuderne. Vægten/omkostningerne for en kant mellem to knuder udregnes ved at addere de to knuders vægte.

$Kant_{i,j}$ vægt = $knude_i$ vægt + $knude_j$ vægt.

Altså hvis en knude der har vægten 45 og en nabo knude som har vægten 60, bliver vægten på kanten mellem disse 45 + 60 = 105, og så fremdeles. Kanterne kunne også have været implementeret med en separat kant klasse som havde indeholdt disse værdier.

8 Korteste vej

Korteste vej algoritmerne anvendes til at finde den korteste afstand mellem to knuder i en graf. Den korteste vej kan i graf sammenhæng have forskellig betydning, alt efter hvad formålet med den korteste vej er. Det kan betyde antal knuder/*trin* man skal igennem for at når fra A til B, eller omkostningerne for at komme fra A til B. Denne anskuelse kendes fra f.eks. online rutebeskrivelser som Map24.com, hvor man kan vælge den korteste længde eller den hurtigste vej.

Hex spillet er implementeret med to forskellige spilbræt typer, som henholdsvis repræsentere spillet i en simpel ikke-vægtede graf (se Figur 8 på side 18) og en simpel vægtede graf (se Figur 13 på side 19).
Efter et generelt afsnit om korteste vej, er der et afsnit om korteste vej med antal *trin* vha. simple ikke-vægtede graf datastruktur. Herefter kommer afsnittet om den mindst omkostningsfulde vej med den simple positiv vægtede graf datastruktur.

8.1 Generelt om kortest vej algoritmer

Der findes mange forskellige algoritmer til at finde den korteste vej mellem to knuder i en graf struktur. I denne rapport undersøges der to algoritmer, som benytter følgende fællestræk:

- Algoritmerne tilhører familien af *grådige algoritmer* (eng. *Greedy algorithm*).
- Algoritmerne skal have en start knude, som den korteste vej findes ud fra, *enkelt-kilde algoritme* (eng. *Single-source algorithm*).
- Algoritmerne skaber en træstruktur for de besøgte knuder.

8.1.1 Grådige algoritmer

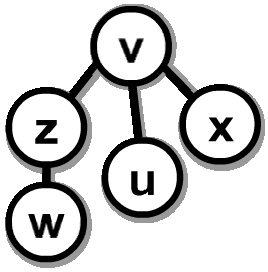
Grådige algoritmer vælger altid den bedste mulige løsning set fra et umiddelbart/lokalt velkendt punkt og regner med det føre til den globale overordnet bedste løsning.
Man kan anskue det som at en *grådig algoritme* har en sekvens af valgmuligheder ud fra et start punkt. Disse valg muligheder sammenlignes og den bedste vælges. Ud fra dette valg udsøges de nye valgmuligheder og muligheder opvejes igen og den bedste vælges.

8.1.2 Enkelt-kilde algoritme

Som nævnt søger *grådige algoritmer* ud fra et velkendt punkt, hvilket betyder at algoritmen skal have givet et velkendt startpunkt at beregne den korteste vej ud fra. Fra dette punkt besøger algoritmen alle grafens kanter. Derfor kaldes algoritmen en *enkelt-kilde algoritme* (eng. *Single-source algorithm*). Hex spillet er dette startpunkt altid en defineret *start knude* for hver farve, som er forbundet til en af den respektive farves sider, dvs. udenfor 2D arrayet (se Figur 4 på side 13).
Hex spillet har, som figuren viser en slut knude, der er implementeret ligesom start knuden. Derved er både start og slut punktet for den korteste vej velkendt fra spillet påbegyndes. Derfor er selve implementeringerne af algoritmerne tilpasset problemet, men algoritmerne er først beskrevet i deres originale form.

8.1.3 Træstruktur

Når algoritmerne bevæger sig fra start punktet ud til alle kanterne, besøges alle komponents knuder mindst én gang. I eksemplet med Figur 7 på side 17 kan startknuden være knude *u*, hvorved knude *z* og *w* besøges mindst én gang, hvorimod *v*, *x* og *y* ikke bliver besøgt.
Når en knude besøges, får denne knude en pointer til knuden, den er blevet besøgt af. F.eks. besøger knude *v* knude *z*, derved får knude *z* en pointer til knude *v*. Efterfølgende besøger kunde *z* knude *w* som så får en pointer til knude *z*. Derved dannes en træstruktur, hvor en besøgt knude har én og kun en pointer til knuden, som den blev besøgt af. Dette forhold kaldes i træstrukturen for *forældre/barn* (eng. *Parent/child*) relation.



Figur 18: Grafen fra Figur 8 er blevet modificeret til en træstruktur vha. en algoritme.

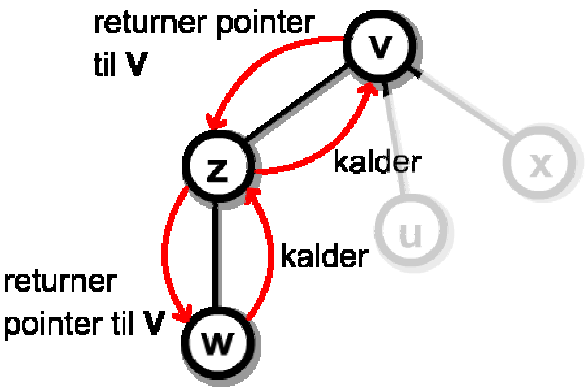
Et træ kan også opfattes som en *forbundet simpel orienteret graf* (se Figur 8 på side 18) uden cirkulære forbindelser og hver knude har ingen eller 1 pointers til en anden knude. Hvis knuden ikke har pointers til en anden knude er det træets *rod* (eng. *Root*), i *enkelt-kilde korteste vej* algoritmer er start knuden *roden*. Hvis der derimod ikke er en pointer, der peger på knuden kaldes den knude i træstrukturen for et *blad* (eng. *Leaf*). Et *blad* er en *ende* knude, hvor korteste vej findes ved at følge

bladets stamtræ tilbage til *start* knuden. Det er altså denne træstruktur, som algoritmerne fremkalder, der muliggør at finde den korteste vej.

8.2 Rekursion

Begrebet rekursion er i forbindelse med programmering en systematik, hvor man får metoder til at kalde sig selv.

Rekusionen bliver i vores Hex spil anvendt til at finde repræsentanten i en given komponent. Metoden hedder findSet(). Knuden som kalder metoden, spørger sig selv om den har en forælder. Hvis ja, udføres samme kald på knudens forælder og hvis nej, skal metoden svare tilbage med en pointer til sig selv. Pointeren til denne forældreløse knude fortælles ned gennem hele det rekursive kald. På den måde får alle knuderne som har adspurgt om hvem roden var, svaret tilbage.



Figur 19: Som det ses giver denne rekusive metode roden på træet.

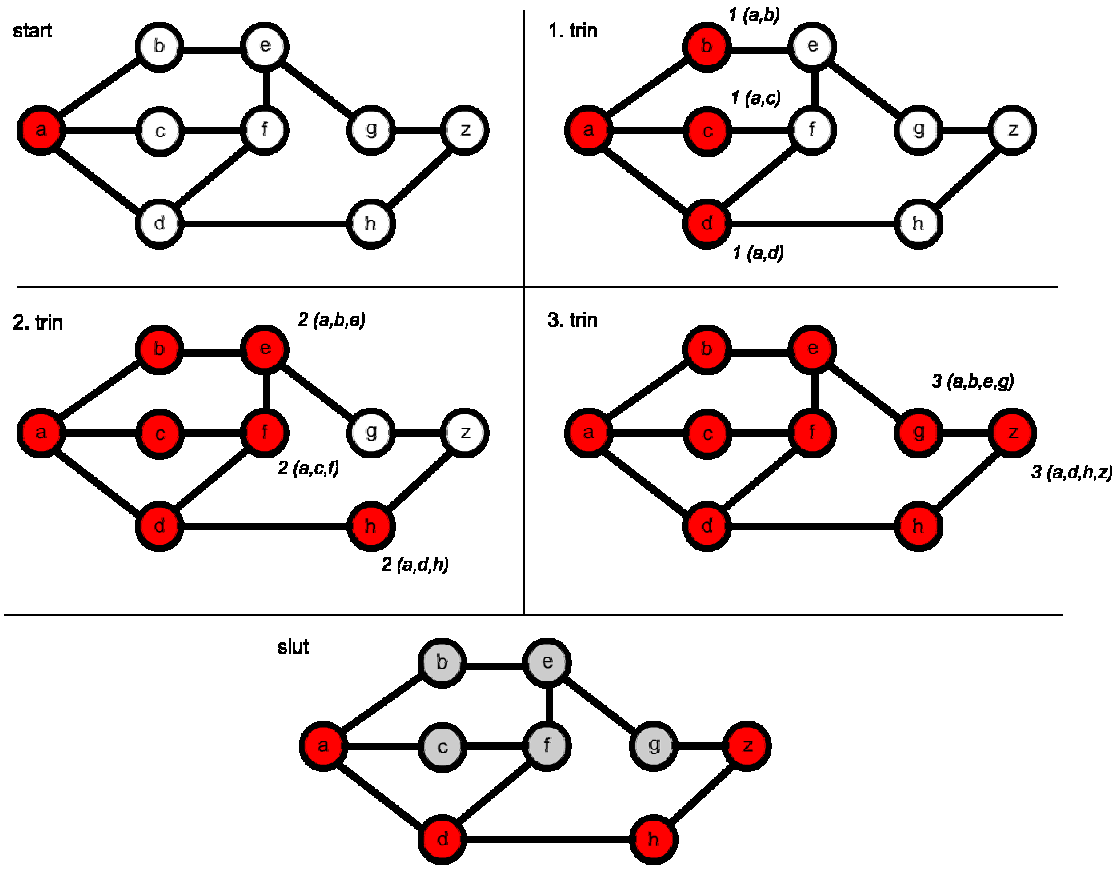
Rekursion bruges også når den korteste vej skal optegnes ved at gå fra slut knuden til start knuden. Her spørges hver knude, som er en del af en korteste vej, om hvem der er dens forældre. På den måde optegnes den korteste vej fra slut knuden til start knuden.

8.3 Korteste vej i simple ikke-vægtede grafer

Der findes forskellige algoritmer til at finde den korteste vej i en simpel ikke-vægtet graf. Her er kigges på algoritmen *Brede-først-søgning.*

8.3.1 Brede-først-søgning

Brede-først-søgning (eng. *Breadth-first-search*) BFS, algoritmens søge-strategi går ud på at inspicere alle de knuder, som er forbundet startpunktet og noterer, derefter hvor mange trin den anvendte for at komme dertil (altså 1 trin til at begynde med).

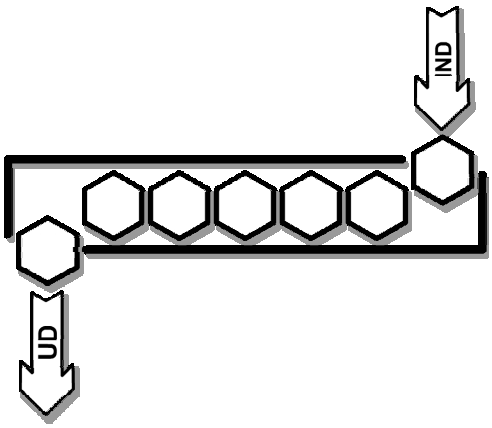


Figur 20: Brede-først-søgning.

Proceduren gentages igen for næste led, og igen noter BFS hvor mange trin den skulle bruge for at komme til den pågældende knude samt markere pågældende knude som *besøgt* fra forrige leds knude (*forældre* knuden i BFS træet).

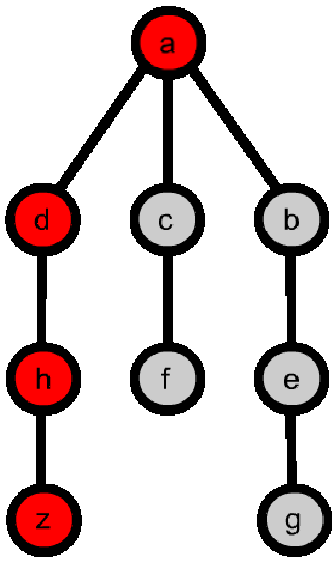
Hvis algoritmen støder på en knude, som allerede er blevet besøgt tidligere, må konklusionen være at dette *ikke* er den korteste vej til denne knude, da den alt andet lige, måtte anvende flere trin for at komme til samme knude. Den pågældende rute bliver da stoppet og algoritmen fortsætter med at undersøge resten af grafen ud fra samme devise til alle knuderne i grafer har været inspiceret en gang. Det betyder at alle grafens kanter har besøg af BFS, dog kun én gang. Det kan derfor argumentere for at BFS søge hastighed afhænger af antallet af kanter i grafen.

For at implementer BFS bliver algoritmen nød til at huske hvilke knuder, der skal inspiceres efterfølgende. Det kræver derfor en kø[11], som lister en knudes kanter op. Køen bevirker at grafen bliver undersøgt led for led ud fra start knuden.



Figur 21: Kø; Først-ind, Først-ud.

Det er når BFS notere fra hvilken knude BFS ankom, at der opstår den omtalte træstruktur med rod/forældre/barn egenskaberne. Når alle knuderne er besøgt, noteres antallet af *trin* for at nå til et specifikt blad i den træstruktur BFS har skabt den korteste vej. Træstrukturen informationer om *forældre/barn* relationerne kan ruten tilbage spores. Denne tilbage sporingen af en rute kan implementeres med en rekursiv operation (se afsnit 8.2 på side 26), der kalder en knudes forælder indtil en knude ikke har nogen forælder. Denne forældreløse knude må derved være roden i BFS træstrukturen. Nedenstående Figur 22 er træstrukturen BFS har skabt ud fra grafen i Figur 20 på side 27.



Figur 22: Brede-først-søgnings opbygning af en træstruktur.

[11] Datastruktur hvor det først ankommende element vil komme først ud af elementer i køen, hvilket kaldes First-in, First-out (FIFO) semantik. Køer kan nemt implementeres i Java med LinkedList klassen. Java 5.0 API har fået en kø klasse implementeret (`java.util.Queue` class), som er et interface for hægtede lister.

8.3.2 Implementering af brede-først-søgning

I implementeringen af BFS bruges start knudens naboliste for at søge led for led. Start knuden puttes i køen, hvorefter et loop køre i gennem køen til der ikke er flere knuder i den. Inde i loopet hentes knuder ud af køens forende, derved undersøges den først ankommende knude.

Når en knude inspiceres sker der følgende:

```
Input: Start knude

1   Sæt start knudens dybde = 0
2   Sikre at start knude ikke har nogen forælder
3   Put start knuden i køen
4   Farv startknuden GRÅ, da vi vil inspicere den

5   Så længe køen ikke er tom, gør følgende:
6       Den først knude i køen hentes og derved fjernes fra køen.
7           For hver nabo til den aktuelle knude tjekkes:
8               Har naboen ikke været i køen før, dvs. knuden er HVID, gør følgende
9                   Sæt afstanden til roden, dybden, til forælders dybde + 1
10                  Sæt denne knudes forælder til at være den knude der er hentet fra køen
11                  Put denne knuden i køen.
12                  Marker at denne knude vil blive undersøgt. Dette gøres ved at farve den GRÅ
13      Når knuden fra køen har fået tjekket sin naboliste markers den SORT
```

Pseudokode 2: Brede-først-søgning.

For at optegne den korteste vej spørges ende knuden om at finde sin rod i BFS træet. Til dette bruges en rekursiv metode(se Figur 19 side 26), som stopper når den kommer til roden. Metoden sætter farven for hver knude metoden besøger til GRØN, derved kan den korteste vej optegnes.

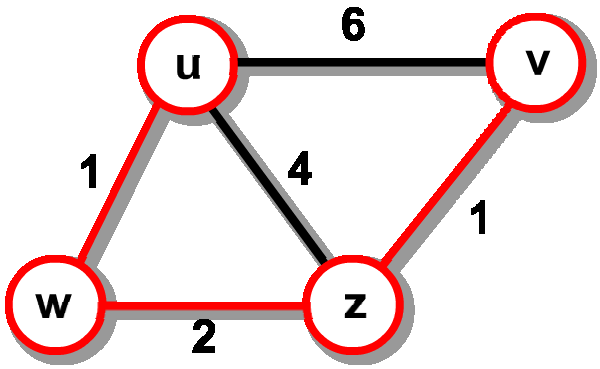
```
Input: Ende knuden

1   Marker knuden GRØN når den er med i sekvensen, som danner den korteste vej.
2   Hvis input knuden ingen forælder har
3       Er knuden er roden
```

```
4       Stop.
5   Ellers
6       Kald denne metode på knudens forælder.
```

Pseudokode 3: Sporing af den korteste vej.

8.4 Korteste vej i vægtede grafer

Når man snakker om den korteste vej i vægtede grafer er det vigtigt at bemærke at de ikke er nok blot at kigge på antal knuder man går i gennem fra sit udgangspunkt til sit slutpunkt. Hvis vi i Figur 23 vil gå fra knude *u* til *v* er den korteste vej (altså den med mindst omkostning) faktisk gennem knude *w* og *z* for så til sidst at havne i knude *v*.



Figur 23: Den korteste vej fra knude *u* til *v* i en vægtet graf.

Omkostningen fra at gå denne tilsyneladende lange vej er kun 4 (1 + 2 + 1), hvorimod den lige vej fra *u* til *v* ville være 6 og 5 hvis man gik gennem *z*. Man skal altså ikke længere kigge på antallet af knuder som traverseres da dette i nogle tilfælde kan blive meget højt.

Et eksempel på korteste vej gennem vægtede grafer, kunne være en AI i et computerspil som skulle passere et bjergpas. Bjergpasset bestod af 2 felter der hvert ville tage 5 timer at passere, hvorimod en rute helt uden om bjerget ville indebære at flytte sig gennem 8 felter som dog kun ville tage en enkelt time hver. Det er altså ikke nødvendigvis den lige vej som er den korteste i vægtede grafer.

8.5 Dijkstra

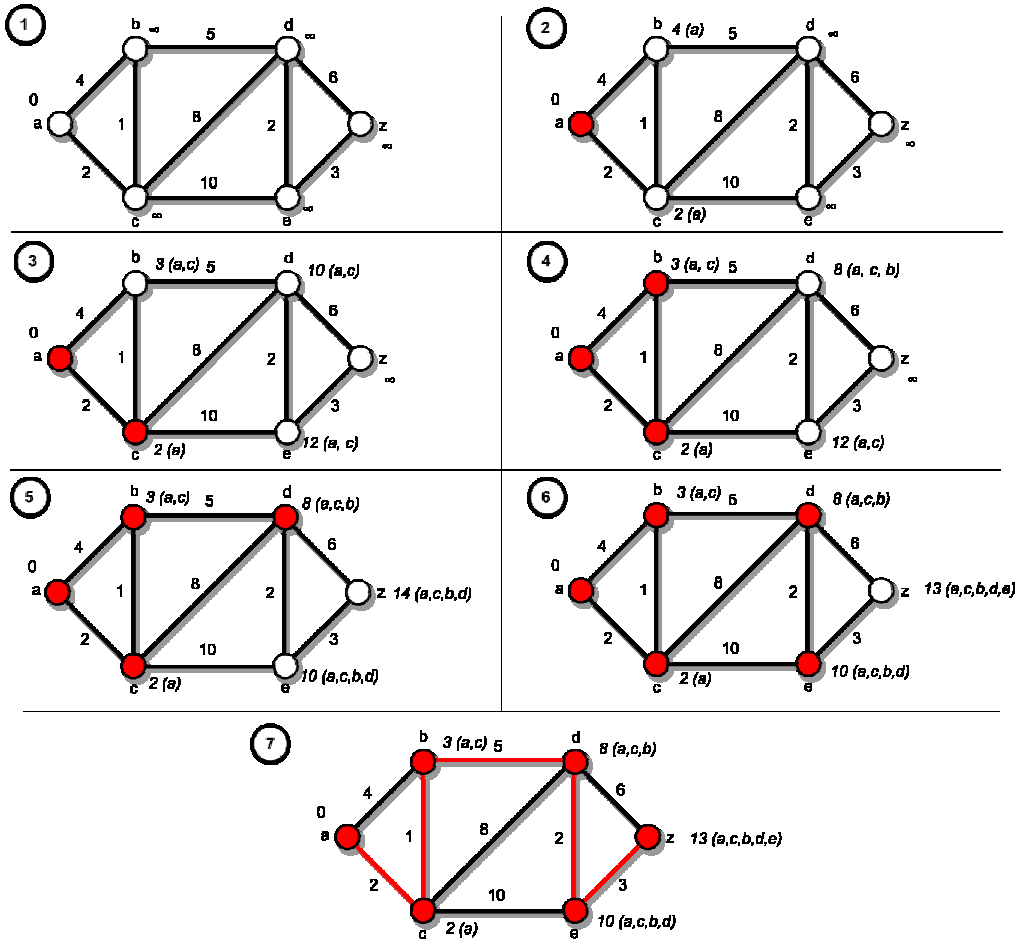
En kendt og afprøvet algoritme for at finde korteste vej i en vægtet graf er den såkaldte Dijkstra algoritme opkaldt efter den hollandske computer videnskabsmand ved navn Edsger Dijkstra[12].

8.5.1 Kort om Dijkstra

Dijkstra algoritmen har overordnet set samme søge-strategi som BFS, hvor en start knudes nabo liste inspiceres og derved skaber en træstruktur. I Dijkstra algoritmen er køen prioriteret[13] efter omkostninger, så den af de inspicerende knuder med mindst omkostning kommer først ud til inspektion.
Dijkstra kan i princippet kun anvendes i *positiv vægtede orienteret grafer*. Som nævnt i indledningen har vi lavet to versioner af Hex spillet, en med en *simpel uvægtet graf* og en anden hvor der er implementeret med en *simpel positiv vægtede graf* struktur samt en defineret slut punkt(ende knude). Af den grund er Dijkstra udvidet i implementeringen af Hex spillet, hvilket vil blive beskrevet i implementeringsafsnittet..

I Figur 24 på side 32 illustreres Dijkstra algoritmens vej gennem en graf.

[12] Kilde: Wikipedia.org (http://en.wikipedia.org/wiki/Dijkstra)
[13] Køen i Dijkstra er ikke en almindelig kø og har ikke FIFO semantik, som i BFS. Denne slags kø kaldes en prioritets kø. Køen prioriter elementerne efter et bestemt parameter, hvor oftest er elementet med den mindste værdi er forrest i køen. Her er parameteret omkostningen, sådan at den knude det er billigst at komme til er forrets i køen.



Figur 24: Dijkstras algoritme som finder den korteste vej fra knude a til z (frit efter Rosen 1999, s. 495).

Dijkstra bevæger sig her fra start knude a til slut knude z. Grafens knuder V={a,b,c,d,e,z} og grafens kanter er vægtet og har derfor et tal for dens vægt f.eks. kant (a,b) = 4. Udfor knuderne står deres omkostninger for at komme fra start knuden a. Med udtagelse af start knuden, har alle knuderne omkostning til $+\infty$ ved begyndelsen, da Dijkstra ikke kender omkostningerne for at kommer dertil. I det knuden bliver inspiceret noteres den omkostning for at komme til knuden ad den aktuelle vej. Når en har undersøgt alle mulighed for næste ryk, markers den rød, hvilket er ens betydende med knuden har forladt køen.

Før Dijkstra køre opdateres de implicerede knuder, for at algoritmen kan undersøge grafen ud fra en enkelt-kilde. Disse trin indtil algoritmens løkke kaldes *initialisering af enkelt-kilde* algoritme. Først sikres at start knuden ingen forældre har, samt at omkostningerne for start knuden er 0. Dernæst sættes omkostningerne for grafens andre knuder til at være uendelig høje (omkostninger = $+\infty$), da disse knuder ikke er undersøgte og derfor opfattes som uendelig langt væk. Start knude bliver lagt i køen af knuder der skal inspiceres.

Algoritmen starter nu en løkke, som kører indtil prioritets køen er tom. I løkken hentes den forreste knude fra kø ud og inspicere den. Herefter beregnes omkostningerne for at gå til hver nabo fra knuden. Er omkostningen den nuværende laveste for at komme til denne nabo, sættes en pointer i naboen til start knuden. Operationen kaldes *afspænding* (eng. *Relaxation*)[14]. *Afspændings* operationen sammenligner om omkostningen fra den aktuelle vej har den mindste omkostning for at komme til netop denne nabo. Hvis det er tilfældet vil naboen få opdateret sin pointer til knuden, der inspiceres fra, samt naboens omkostninger bliver opdateret. Efter som alle knudernes omkostning = $+\infty$ vil først besøg hos en nabo resultere i naboen får en pointer til den inspicerende knude.

Efter *afspændingen* bliver den inspicerede nabo puttet i kø. Herved ender alle knudens naboer i køen, derfor besøger Dijkstra alle grafens kanter.

8.6 Implementering af Dijkstra

Grafen i Hex spillet er en uorienteret vægtede graf, derfor er Dijkstra modificeret for at passe til Hex spillet. Knuderne markeres for *inspiceret(*markers sorte*)*, for herved at undgå mulige loops, som følge af grafen i Hex spillet er uorienteret. Det betyder kanter peger begge veje, altså at hver knude har en pointer til hinanden. Det indebærer at når Dijkstra inspirer en af start knudens naboer, vil den lægge start knuden i forrest i køen, da den har omkostningerne = 0. Den vil så vælge den nabo der netop har været undersøgt og så fremdeles.
Korteste vej problematikken i Hex spillet er som nævnt ikke et *Enkelt-kilde korteste vej problem*, da ende knuden kendes i forvejen. Derfor er Dijkstra implementeret med et statement der afslutter algoritmen, når slut knuden er inspiceret.

Input: Alle knuderne, Start knude og ende knude

1 Opret en prioritets kø.

2 Sæt start knudens forælder til ingen (parent = NIL)
3 Sæt start knudens omkostninger = 0
4 Sæt alle andre knuders omkostninger = $+\infty$

[14] Relaxation er en proces, hvor der testes om omkostningerne for at komme til en knude kan forbedres. Hvis relaxations processen kan for bedre omkostningerne, opdateres omkostningerne for knuden og dens forældre pointer bliver sat til den ny forældre.

```
5   Put start knuden i prioritets køen.


6   Så længe prioritets køen ikke er tom, gør følgende:
7       Hent den første knude (n) fra prioritets køen og fjern den fra køen
8       Marker at n nu bliver inspiceret ved at farve den sort
9       For hver nabo til n, gør følgende:
10          Afspænd naboen
11          Læg naboen i prioritets køen, hvis den ikke tidligere er blevet undersøgt.


12      Hvis n er ende knuden, afslut Dijkstra
```

Pseudokode 4: Prioritets kø.

Prioritets køen er implementeret med en hægtede liste, hvilket kan kræve at hele listen bliver kørt igennem for at kunne placer et nyt element.
Omkostningerne for listens knuder bliver sammenlignet med knuden, som skal lægges i køen. Hvis den nye knude er mindre end en af knuderne fra køen overtager den nye knude denne placering. Køens knuder får derved et indeks der er 1 højere end før. Hvis den nye knudes omkostninger ikke er mindre end nogle af knuderne fra køen, lægges den nye knude bagerst i køen.

```
Input: kø og ny knude


1   For hver knude i køen
2       Hvis omkostningen og ny knude mindre end kø knude_n-1
3           Indsæt ny knude på knude_n-1 plads
4   Ellers indsæt ny knude bagerst i køen
```

Pseudokode 5: Relaxtion.

Hvis prioritets køen er meget lang, kan implementeringen med hægtede lister være langsommelig, hvilket dog ikke vil ske i Hex spillet. Der findes også andre datastrukturer til at implementere en prioritetskø med, som vil gøre prioritetskøen mere anvendelig ved håndtering af store mængder data.

For at vide om en af spillerne i Hex spillet har vundet, skal start knuden og ende knuden være forbundet. Dette kan undersøges af korteste vej algoritmerne, men den inspicere alle eller mange knuder, blot for at konstatere af der ikke er forbindelse mellem start og ende knuderne. Det vil derfor være godt med en mere effektiv metode til at tjekke, om der er forbindelse mellem start og slut knude.

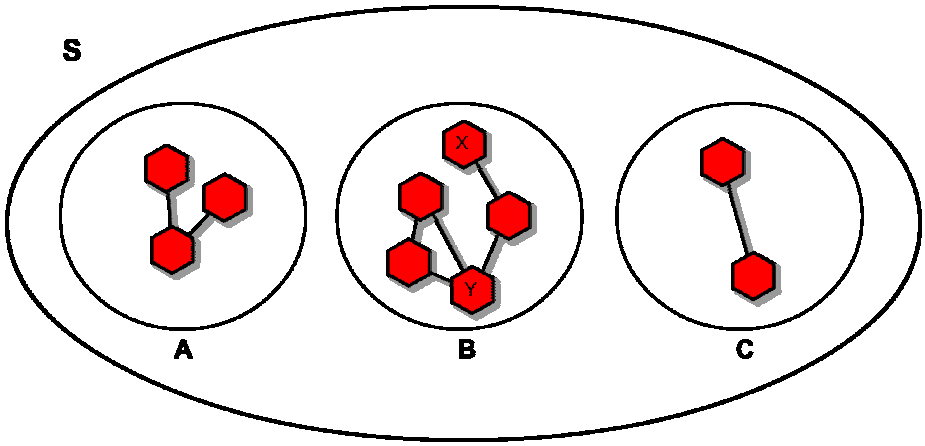
9 Disjunkte mængder datastruktur

I stedet for en tung algoritme, kan vi undersøge om start og ende knuderne blot er i samme komponent, for at tjekke om der er forbindelse mellem start og ende knuden eller ej. Dette er grundlaget for at benytte disjunkte mængder datastrukturen, da strukturen giver mulighed for hurtigt og effektivt at undersøge om to knuder er i samme komponent.

9.1.1 Mængdelærer analogi

For at forstå dette, kan graf komponenterne opfattes som delmængder i en mængde, hvor ingen af delmængderne har elementer med forbindelse til hinanden.
Den overordnede betragtning er at vi betragter hver spillers brikker, som en stor mængde i sig selv. Det er grafens mængde af knuder, V.

Når vi kigger så på disse brikker, kan nogle være forbundene komponenter, disse lader vi være mængder for sig. Hvis en af disse mængder både indeholder start knuden og slut knuden for en spiller, er der fundet en vinder.
Grafen (S) består af 3 delmængder A, B og C. Disse har ingen elementer med nogen relation.
$S \supseteq A, B, C.$
$A \cap B = Ø, A \cap C = Ø, B \cap C = Ø.$



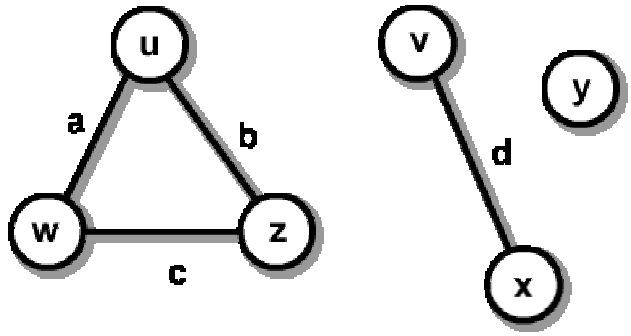
Figur 25: Delmængder i mængden af røde brikker.

I Figur 25 kan der nemt undersøges om brik X og Y er forbundet blot ved at undersøg den mængde de er i. Det er altså kun delmængde B, der bliver undersøgt, mens delmængde A og C bliver ikke undersøgt.

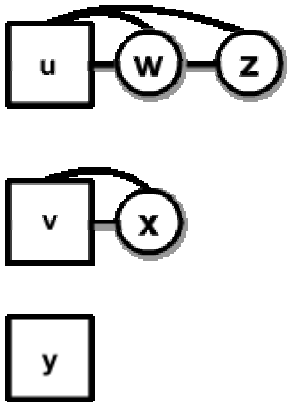
En datastruktur for denne anskuelse af problematikken, om der er forbindelse mellem to knuder, er disjunkte mængder datastruktur.
Hver delmængde har et element, som repræsenter delmængden. Alt efter strukturens opgave, kan denne repræsentant for en mængde være tilfældig eller specielt valgt. Repræsentanten skal være et element fra mængden, da ethvert element er entydigt for mængden i samlingen pga. $A \cap B = \emptyset$, $A \cap C = \emptyset$, $B \cap C = \emptyset$. Valget af en delmængde repræsentant bliver senere diskuteret i dette afsnit om optimeringsmetode for disjunkte mængder datastrukturen.

For at benytte denne *mængdelærer* analogi, kan du tænke på en mængde når der skrives graf og delmængde når der skrives komponent.

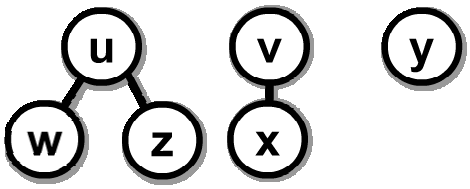
Disjunkte mængder datastrukturen kan implementeres med *Hægtede lister*, hvor hver komponent er en dobbelt hægtede liste, se Figur 27. Den hægtede liste skal være en dobbelt hægtede, så alle knuder har en pointer i listens *Hoved*[15], som er komponentens repræsentant. En anden implementering er at hver komponent er en træstruktur, hvor træets rod er repræsentanten for komponenten. Denne implementering af disjunkte mængder kaldes en disjunkte mængder *skov* (eng. *Forest*) pga. strukturen består af flere træer, se Figur 28.



Figur 26: Graf med 3 komponenter.



Figur 27: Disjunkte mængder med hægtede lister.



Figur 28: Disjunkte mængde *skov*.

[15] En hægtet listes *Hoved* er det første element i listen.

9.2 Operationer for disjunkte mængder datastruktur

Disjunkte mængder datastrukturen har tre overordnet operationer for at vedligeholde strukturen.

- *Lav-komponent(knude)* (eng. *Make-Set(Vertex)*),
- *Forene(knude$_i$, knude$_j$)* (eng. *Union(Vertex$_i$,Vertex$_j$)*)
- *FindKomponent(knude)* (eng. *Find-Set(Vertex)*)

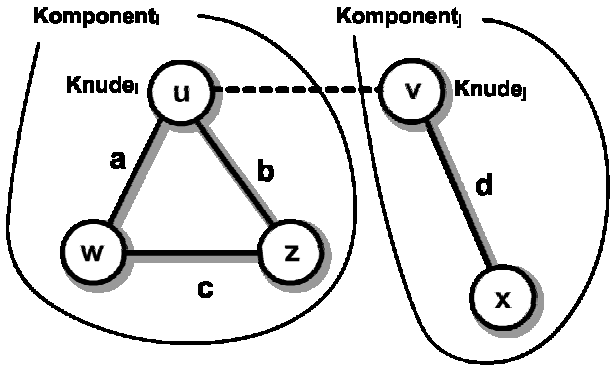
9.2.1 Lav komponent operation

Lav-Komponent(knude) opretter et ny komponent, hvor input knuden er komponentens eneste knude. Derfor er knuden også komponentens repræsentant, med andre ord komponentens identifikation. Operationen kræver at enten en hægtet liste bliver initialiseret med knuden som *hoved* eller knuden er roden i dette komponents træstruktur. Denne operations hastighed er ikke berørt af om strukturen er implementeret med Hægtede lister eller træstruktur.

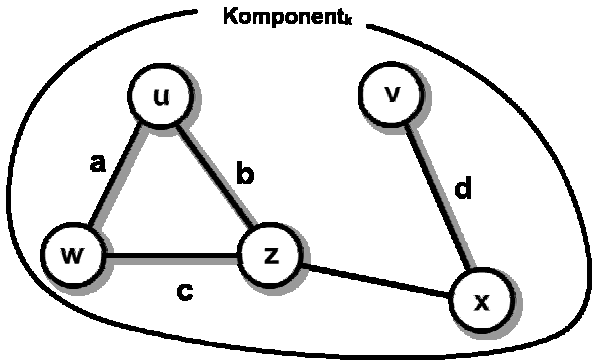
9.2.2 Forene operation

Forene(knude$_i$, knude$_j$) sammenlægger *komponenten*$_i$ med *knude$_i$* og *komponenten*$_j$ med *knude$_j$*, til et nyt *komponent*$_k$ med alle knuderne fra de to komponenter. Operationen foretages når to knuder, som bliver naboer og ikke tilhører samme komponent, altså de to knuder har ikke samme repræsentations knude(*hoved / rod*). En komponent kan ikke indeholde knuder med forbindelse til knuder fra andre komponenter.

Når en ny komponent oprettes, kan komponentens repræsentations knude være nabo til en knude fra en anden komponent. Derved bliver en forening af de to respektive komponenter nødvendig.



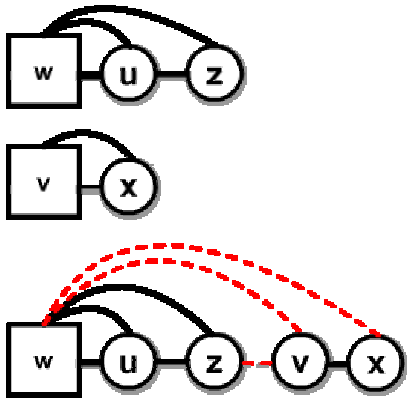
Figur 29: To komponenter der skal forenes.



Figur 30: Ny komponent med alle knuderne fra de to komponenter.

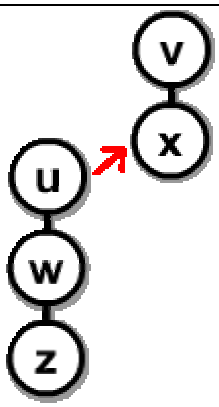
Princippet er at en ny komponent med alle knuderne oprettes og de to komponenter, som knuderne var i bliver fjernes. I realiteten genbruges det ene komponent, som det *nye* komponent og det andet komponent fjernes. Praksis er at det ene komponents repræsentant bliver den nye repræsentant for alle knuderne. Her vil strukturens implementering have betydning for hastigheden af *forene* operationen.

Hægtede lister kræver mange opdateringer, fordi alle pointers i listen skal opdateres ved en ny repræsentant



Figur 31: Forening af to komponenter med hægtede lister.

Træstrukturen kræver derimod kun en opdatering for at udfører denne operation, nemlig at den ene komponents repræsentant får en pointers til den anden komponent



Figur 32: To komponent træer forenes.

9.2.3 Find komponent operation

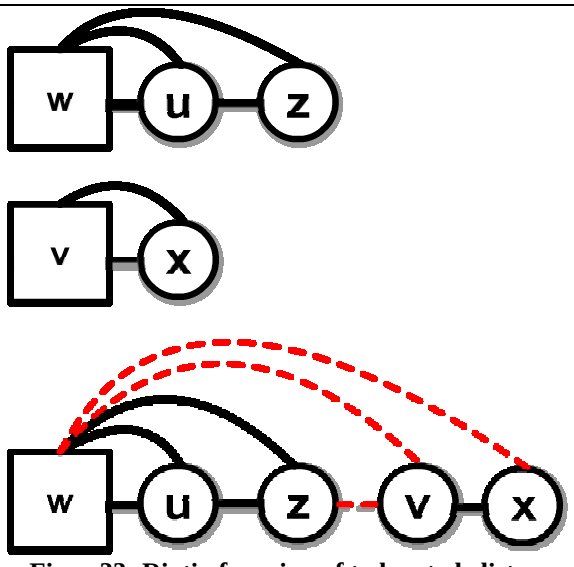
Operationen returnerer en pointer til repræsentanten for en komponent, som indeholder input knuden(Cormen, 1999, side 441). Her spiller implementeringen også ind på operationens hastighed. Med en hægtede liste kræver operationen blot et kald, da hver knude i komponenten har en pointer til listens *hoved*, som er repræsentanten for komponenten.

Træstrukturen skal derimod finde sin komponent repræsentant ved at kalde op igennem træstrukturen, hvilket kan gøres ved en rekursiv operation. Det kan i værste tilfælde betyde at den rekursive operation skal kaldes ligeså mange gange som der er knuder i træet, se Figur 19 side 9.

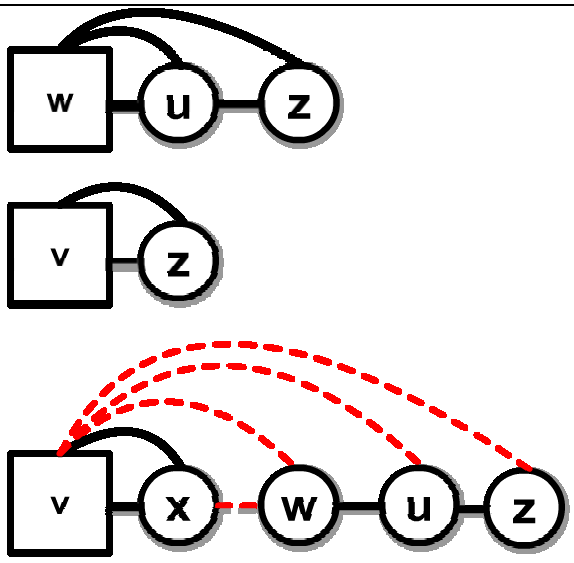
9.2.4 Optimering disjunkte mængder strukturen

En ny komponents repræsentant kan godt skabe forbindelse mellem flere komponenter. Der med forenes flere komponenter som allerede består af mange knuder. I hægtede lister skal mange pointers opdateres og i træstrukturen kan der dannes meget dybde træer. Dybe træer betyder at *find-komponent* operationen bliver langsommere jo dybere træet bliver for komponenten.

Forene operationen kan tilføjes en optimeringsmetode, for at håndtere denne problematik og forbedrer strukturens operations tider. Optimeringsmetoden er, at den mindste komponent forenes med den største komponent. Derved sikres at det er færrest mulige pointers, der skal opdateres i en hægtede liste komponent ved en *forene* operation. Denne optimeringsmetode for hægtede lister kaldes for *vægtet forene operation* (Cormen, 1999, side 445). Figur 33 og Figur 34 på side 40, illustrer at optimeringsmetoden sikre at der kun er 3 opdateringer af pointers, mod risikoen for 4 opdateringer. I forbindelse med meget lange hægtede lister giver denne optimeringsmetode store besparelser.



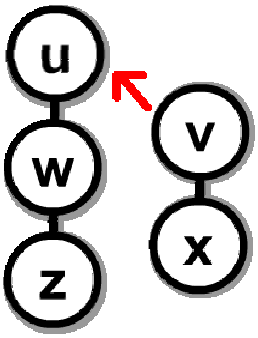
Figur 33: Rigtig forening af to hægtede lister.



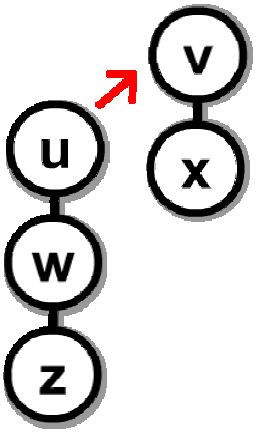
Figur 34: Forkert forening af to hægtede lister.

For træstruktur implementeringen gør denne optimeringsmetode *find-komponent* operationen hurtigere. Repræsentanten for træet med mindst dybe bliver forenet med repræsentanten for det dybeste

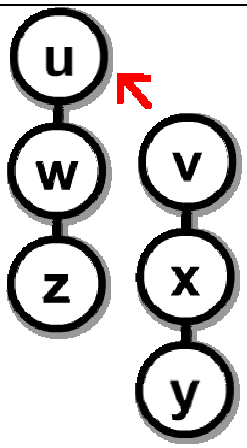
træ. Derved bliver det nye træ ikke dybere. Hvis to lige dybe træer forenes, vælges et af træerne til den nye repræsentant, hvilket medfører at det nye træs dybde forøges med 1. Træets dybde kan dermed ikke vokse med mere end 1 ved en *forene* operation. Når optimeringsmetodeen benyttes med træer kaldes det *rang forene operation*, da dybde for et træ også kaldes træets rang (Cormen, 1999, side 447). Nedenstående Figur 35 viser forening af to træer, hvor optimeringsmetodeen bruges. Derfor forbliver det dybeste træs dybde på 2. Figur 36 forøges det dybeste træs dybde med en, så træens dybde efter foreningen er 3. Figur 37 illustrer at ved en foreningen af to træer med ens dybde, bliver dybden ”kun” forøget med 1, så det dybeste træ er 3.



Figur 35: Rigtig forening af to træer.



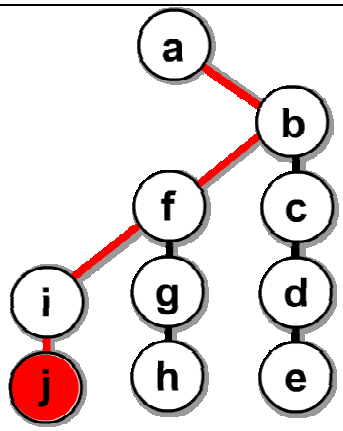
Figur 36: Forkert forening af to træer.



Figur 37: Forening af to træ med ens dybde.

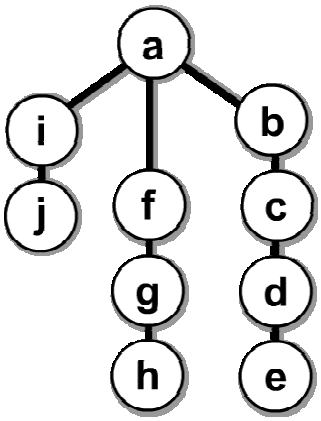
Træstrukturen kan optimeres yderligere ved at benytte *sti kompressions* (eng. *Path compression*) optimeringsmetoden, som implementeres i *find komponent* operationen. Optimeringsmetoden er at når *find komponent operationen* alligevel er i gang med at finde repræsentanten, kan operationen på vej op gennem træet lige så godt sætte knudernes pointers direkte til komponentens repræsentant med det samme. Træets dybde forbliver uændret, så *rang forening* optimeringsmetoden stadig fungere efter hensigten.

Find komponent operationens vej op gennem træet fra knude j



Figur 38: Find komponent operation kaldt fra knude j.

Efter sti kompressionen er knuder operationen har besøgt, fået opdateret deres pointers direkte til komponentens repræsentant



Figur 39: Træet efter sti kompression.

9.2.5 Vedligeholdelse af disjunkte mængder datastrukturen

De 3 operationer kan tilsammen vedligehold strukturen ved proceduren *Forbundene-komponenter(graf)* (eng. *Connected-components(Graph)*).

Input: Graf

Forbundene komponenter

```
1  For alle knuder som er i input grafen
2      Lav en ny komponent
3  For hver forbindelse mellem to knuder fra grafen
4      Hvis repræsentanten for de to knuder ikke er den samme.
5          Forene knudernes komponenter
```

Pseudokode 6: Forbundene komponenter (Cormen, et al, 1999).

Her ved bliver strukturen hele tiden vedligeholdt, så strukturen består af komponenter med knuder uden forbindelser/relationer.
Vigtigst for brugen af disjunkte mængder strukturen er at afgøre om to knuder er i samme komponent. Til dette formål er der proceduren *Samme-komponent(knude*$_i$, *knude*$_j$*)* (eng. *Same-component(vertex*$_i$, *vertex*$_j$*)*) for at teste om *knude*$_i$ og *knude*$_j$ er forbundet. Samme komponent proceduren returnerer sandt eller falsk, alt efter om de to input knuder er i samme komponent eller ej.

Input: to knuder
Output: sand eller falsk

```
Samme komponent(element 1, element 2)
1   Hvis findSet(element 1) = findSet(element 2) // finder hver elements mængde repræsentant
2       Det er sandt at element 1 og element 2 er i samme komponent
3   Ellers Det er falsk at element 1 og element 2 er i samme komponent
```

Pseudokode 7: ”Introduction to Algorithms, 1.edition” kap. 22.

Forbundene-komponenter proceduren skal have været kaldt på grafen inden der testes med *Samme-komponent* proceduren for at sikre det rigtige resultat.

9.3 Implementering af disjunkte mængder datastrukturen

Hex spillet er implementeret med træstrukturen. Disjunkte mængder datastrukturen er implementeret for at kunne finde ud af om to knuder er forbundet.
Til det formål bruges *samme komponent* (implementeringsnavn: *sameComponent*) for at undersøge om start knuden og ende knude er i samme komponent, hvilket vil betyder at der er en vinder.

```
Samme komponent(knudei, knudej)
1   Hvis find komponent (knudei) = find komponent (knudej)
2       returner sand
3   Ellers returner falsk
```

Pseudokode 8: Samme komponent.

Samme komponent benytter *Find komponent*(implementeringsnavn: *findSet*) to gange for at sammenligne repræsentanterne for hver knude. Find komponent benytter rekursion for at finde kompo-

nentens repræsentant. Dernæst er der implementeret *sti kompression* ved at det rekursive kald, sætter hver besøgte knudes forældre til komponents repræsentant, hvorved træet bliver kortere.

```
Find komponent(knude)
1  Hvis knuden ingen forælder har og er derfor repræsentanten for komponenten
2      returner pointer til knude
3  Ellers
4      kald Find Komponent(forælder), derved startes det rekursive kald og sæt denne knudes for
       ældre til operations resultat.
```

Pseudokode 9: Find komponent.

Nå en brik i Hex spilet sættes kaldes metoden updateStructures i GameManager klasse, herfra kaldes metoden Forbundene komponenter (implementeringsnavn: connectedComponent).
Denne nye brik er en ny knude i grafen. Grafen ændrer sig altså hver gang en ny brik sættes.
Derfor skal datastrukturen opdateres for at være sikker på at sammenligningen er korrekt.
Den nye knude kan maksimalt have 6 naboer, hvilket betyder at det kun er knuden og knudens mulige 6 naboer, som skal opdateres med *Forbundene-komponent* proceduren for at sikre at grafen er opdateret.

Operationen i Hex spillet er derfor modificeret, så den passer bedre til Hex spillet. Metoden i Hex spillet gør derfor følgende:

```
Forbundene komponenter(knude_i, naboer_n (Sæt med knudes_i naboer))
1  Lav komponent(knude_i)
2  Hvis sættet med naboer ikke er tom, gør følgende
3      Tag nabo n-1 ud af sættet
4      Hvis ikke samme komponent(knude_i, nabo_n-1)
5          Forene(knude_i , nabo_n-1), så knude_i og nabo_n-1 er i samme komponent
```

Pseudokode 10: Forbundene komponenter.

Den første operation på linie 1, der kaldes er *Lav komponent*(implementeringsnavn: *makeSet*), der er implementeret på følgende måde:

```
Lav komponent(knude)
1  Sæt knude til at være rod, så brikken tegnes med i prik
2  Sæt knudes dybde til 0
```

Pseudokode 11: Lav komponent.

Linie 4 i Pseudokode 10 sammenlignes knuden komponent med naboens komponent. Hvis de ikke er i samme komponent skal de to komponenter forenes vha. *Forene*(implementeringsnavn: *union*). For at optimere datastrukturen er *rang forene* optimeringsmetoden implementeret, dvs. komponent repræsentanten med mindst dybde, får en pointer til komponent repræsentanten med størst dybde. Hvis komponent repræsentanterne har lige stor dybde vælges en af repræsentanterne til ny repræsentant og dennes dybde forøges med 1.

```
Forene(knude_i, knude_j)
1  repræsentant_i for knude_i = find komponent(knude_i)
2  repræsentant_j for knude_j = find komponent(knudej)
3  Hvis dybden for repræsentant_i er større end dybde for repræsentant_j
4       sæt repræsentant_i som forælder for repræsentant_j
5       sæt repræsentant_j til ikke at være rod.
6  Ellers
7       sæt repræsentant_j som forælder for repræsentant_i
8       sæt repræsentant_i til ikke at være rod.
9       Hvis dybden for repræsentant_i er = dybde for repræsentant_j
10           forøg dybden for repræsentant_j med 1
```

Pseudokode 12: Forene.

10 Den endelig version

Foranalysens krav, fra side 11, kan nu opfyldes ved følgende klasser og metoder

Brættet og brikkerne

Kravet var at programmet skulle kunne tegne et spillebræt af størrelse *n* x *n* med blanke felter, og samtidig være i stand til at tegne rød og blå brikker.
Klassen Board og GameBoard tegner brættet. Klassen Vertex bliver farvet efter hvilket spiller objekt (klasse: Player) der knyttes til knuden.

Spillerne

Kravet var at der skulle være metoder som kunne skifte spillernes tur, og at programmet skulle kunne registrerer når en spiller klikkede på spillebrættet.

Den overordnede procedure når en brik sættes med et tryk på musen, er:

```
1   Hvis der ingen vinder er fundet gør følgende
2       Gennemløb felterne indtil det aktiverede felt er fundet.
3           Hvis feltet er indenfor brættets rammer og der ikke er tilknyttet en spiller
4               Tilknyt spiller til feltets knude.
5               Opdater spillets datastruktur

6               Hvis spillerens start og slut knude er forbundet gør
7                   Hvis vægtet bræt
8                       Kald Dijkstra
9                   Ellers
10                      Kald brede-først søgning

11                  Stop spillet ved at sætte winner til true
12                  Optegn den korteste vej

13              Hvis spilleren ikke har forbundet start og slut knude
14                  Skift spiller tur
15              Optegn brættet med den nye status.
```

Pseudokode 13: Registring af klik på spillebrættet.

Spillets status

Kravet var at programmet kunne forbinde en spillers brikker med hinanden, og checke om en spiller havde vundet. Hvis en spiller havde vundet, skulle der være en metode som fandt den korteste vej fra den vindende spillers sider, og en metode som derefter afslutter spillet.

Metoderne for status er kun beregningsmetoder, derfor ligger de i pakken *logic*. Klassen *Graph* omhandler alt om graf datastrukturen og klassen *SetLogic* bruges til modulering af disjunkte mængde datastrukturen.
I klassen Graph er metoderne findNeighbors, som finder nabo brikker med en identisk spiller tilknyttet. Herved opbygges grafen for spillet. SetLogic klassen har metoden sameComponent, som tjekker om start og slut knuderne er forbundet, hver gang en brik sættes.
Alt efter hvilket bræt der er anvendt, benyttes algoritmerne fra Graph klassen, breathFirstSearch (til brættet uden vægte) og Dijkstra (til brættet med vægte). Efter algoritmen kaldes shortestPath fra Graph klassen, så knuderne i den korteste vej optegnes med farven grøn.

I klassen GameManager er en boolean, *winner*, som bliver aktivere når en spiller forbinder sine to sider og dermed vinder. Denne boolean deaktivere hele *mouse pressed event* for brættet, og derved kan der ikke spilles videre før et nyt spil er initialiseret.

Til at holde styr på alt dette der er implementeret en metode til at opdatere datastrukturen. Den forløber på følgende måde:

Input knuden array position, spilleren, adgang til knude arrayet

1 Hent knuden fra knude arrayet
2 Hent naboerne til knuden ind i en hægtet liste
3 Put naboerne i knudens nabo liste
4 Tilknyt spilleren til knuden
5 Opret en ny komponent med den nye knude
6 Forbind naboernes komponenter med den nye komponent

Pseudokode 14: Opdatering af datastrukturen.

11 Konklusion

For at sikre Hex spillet altid har styr på om der er en vinder i spillet, er det vigtigt at benytte velovervejede datastrukturer. Vi valgte at implementer en graf datastruktur, fordi denne rummer mange muligheder for at finde den korteste vej i et computerspil.

Beregningen af den korteste vej kan anskues på to måder, hvor det enten er færreste antal trin eller den laveste omkostning, for at komme fra start til slut punktet. Disse to anskuelser kræver hver deres korteste vej algoritme. Vi har implementeret brede-først-søgning for at finde den trinmæssige korteste vej og Dijkstras algoritme for at beregne vejen med den laveste omkostning. Efter modificering af begge algoritmerne, virker de efter hensigten i vores applikation.

For at undersøge om en spiller har forbundet start og slut knuderne kan man bruge korteste vej algoritmen. Dog vil algoritmen skabe en træstruktur som ikke indeholder slut knuden, hvis der ikke er forbindelse mellem start og slut knuden. Af den grund vil algoritmen aldrig nå frem til slut knuden og kan ikke optegne den korteste vej.

I stedet for kan man lave en supplerende datastruktur, som strukturerer grafen i uforbundene komponenter. Ved at kigge på om nogle af disse komponenter indeholder både start og slut knuderne, kan det afgøres hvornår korteste vej algoritmerne skal eksekveres. Det betyder at når start og slut knuderne er i samme komponent er de dermed også forbundet. På denne måde køres algoritmerne kun én gang. Det sker først når applikationen får besked om at der er findes en vej mellem start og slut knuderne som kan undersøges.
At spørge om en komponent indeholder start og slut knuden kræver langt færre operationer end korteste vej algoritmerne. Struktureringen af data betyder hermed at man effektiviserer applikationen.

12 Litteraturliste

12.1 Bøger

Rosen, Kenneth: *"Discrete Mathematics and Its Applications"*. 1999, McGraw-Hill Education. ISBN: 0072899050.

Goodrich, Michael T. & Tamassia, Roberto: *"Data Structures and Algorithms in Java"*. 2004, John Wiley and Sons. ISBN 0471469831.

Cormen, Thomas H., et al: *"Introduction to Algorithms"*. 1999, MIT Press. ISBN: 0262530910.

12.2 Artikler

Milnor, John: *"The Game of Hex"* i Kuhn, Harold W. & Naser, Sylvia: *"The Essential John Nash"*. 2002, Princeton University Press. ISBN: 0691095272.

12.3 Websites

Wikipedia.org (www.wikipedia.org):

- Hex (board game) - http://en.wikipedia.org/wiki/Hex_%28board_game%29.
- Edsger Dijkstra - http://en.wikipedia.org/wiki/Dijkstra.

DocJava.dk (www.docjava.dk):

- Grafer - http://www.docjava.dk/datastrukturer/grafer/grafer.htm.

Wolfram's MathWorld (mathworld.wolfram.com):

- Weisstein, Eric W.: *"Game of Hex."* - http://mathworld.wolfram.com/GameofHex.html.
- Weisstein, Eric W.: *"Regular Polygon."* - http://mathworld.wolfram.com/RegularPolygon.html.
- Weisstein, Eric W.: *"Multigraph."* - http://mathworld.wolfram.com/Multigraph.html.

```java
package theHexGame;

/**
 *
 * @author Simon Larsen & Jens Fredersiksen
 */
public class Board {

    private int boardSize;
    private boolean weigthed;

    private Vertex endRed;
    private Vertex startRed;
    private Vertex endBlue;
    private Vertex startBlue;

    protected Vertex[][] vertexArray;

    /**
     * The positions for a vertex's  6 neighbors [0, -1][1, -1][1, 0][0, 1][-1, 1][-1, 0]
     */
    private int[] neigborPos = {0, -1, 1, -1, 1, 0, 0, 1, -1, 1, -1, 0};

    /**
     * @param boardSize int - the size of the current gameboard
     * @param p1 Player - player 1
     * @param p2 Player - player 2
     * @param weigthed boolean - if the game is on a weigthed gameboard or not
     */
    public Board(int boardSize, Player p1, Player p2, boolean weigthed){
        this.boardSize = boardSize;
        this.weigthed = weigthed;

        /**
         * The for virtual start and end vertex.
         * They are used to determine wether a player has won the game or not
         */
        endRed = new Vertex(p1, "END for RED");
        startRed = new Vertex(p1, "START for RED");
        endBlue = new Vertex(p2, "END for BLUE");
        startBlue = new Vertex(p2, "START for BLUE");

        vertexArray = new Vertex[boardSize][boardSize];

        /**
         * Initializes all the vertex in the gameboard
         * Sets their position.
         */
```

```
49          for(int i = 0; i < boardSize; i++){
50              for(int j = 0; j < boardSize; j++){
51
52                  vertexArray[i][j] = new Vertex(i, j);
53
54                  /**
55                   * if the gameboard is weigthed different weigths are assigned
56                   * to the different vertexs
57                   */
58                  if(weigthed){
59                      if((i > 0 && j > 0) && (i < boardSize-1 && j < boardSize-1)){
60                          vertexArray[i][j].setWeight(15);
61                          if((i > 1 && j > 1) && (i < boardSize-2 && j < boardSize-2)){
62                              vertexArray[i][j].setWeight(30);
63                              if((i > 3 && j > 3) && (i < boardSize-4 && j < boardSize-4)){
64                                  vertexArray[i][j].setWeight(45);
65                                  if((i > 5 && j > 5) && (i < boardSize-6 && j < boardSize-6)){
66                                      vertexArray[i][j].setWeight(60);
67                                  }
68                              }
69                          }
70                      }
71                  }
72              }
73          }
74      }
75
76      /**
77       * @return Returns the vertexArray.
78       */
79      public Vertex[][] getVertexArray() {
80          return vertexArray;
81      }
82
83      /**
84       * @return Returns the endBlue.
85       */
86      public Vertex getEndBlue() {
87          return endBlue;
88      }
89
90      /**
91       * @return Returns the endRed.
92       */
93      public Vertex getEndRed() {
94          return endRed;
95      }
96
```

```
 97      /**
 98       * @return Returns the startBlue.
 99       */
100      public Vertex getStartBlue() {
101          return startBlue;
102      }
103
104      /**
105       * @return Returns the startRed.
106       */
107      public Vertex getStartRed() {
108          return startRed;
109      }
110
111      /**
112       * @return Returns the boardSize.
113       */
114      public int getBoardSize() {
115          return boardSize;
116      }
117  }
118
119  //---------------------------------------------------------------------------
120  //                          E N D   O F   F I L E
121  //---------------------------------------------------------------------------
122
```

```java
/*
 * Created on 08-11-2004
 */
package theHexGame;

import java.awt.*;

/**
 * @author Simon Larsen & Jens Frederiksen
 */
public class GameBoard{

    private int weight;
    private String cost;

    private int boardSize;

    private Polygon[][] polygonArray;

    private final int UNIT = 10;
    private final double SQRT3 = Math.sqrt(3);

    private static final int XOFF = 50;
    private static final int YOFF = 50;

    public GameBoard(int size) {
        boardSize = size;
        polygonArray = new Polygon[boardSize][boardSize];
    }

    /**
     * Draws the hexagon at the specified x and y postion and with the specified
     * color.
     *
     * @param xPos  the x position of the origin of the hexagon.
     * @param yPos  the y position of the origin of the hexagon.
     * @param g     the graphics object of the hexagon.
     * @param c the color of the hexagon.
     * @see java.awt.Polygon#Polygon(int[], int[], int)
     */
    public void drawHexagon(int xPos, int yPos, Graphics g , Color c, Vertex v, boolean weigthed, boolean root)
    {
        if(weigthed){
            weight = v.getWeight();
            if(v.getCost() >= Integer.MAX_VALUE){
                char uendlig = 8734;
                cost = "" + uendlig;
            }
```

```
48              else{
49                  cost = "" + v.getCost();
50              }
51          }
52
53          /**
54           * These double arrays are use for calculation the position of where the
55           * 6 points in the hexagon is located. The x and y positions.
56           */
57          double xr[] = {SQRT3, 0, -SQRT3, -SQRT3, 0, SQRT3};
58          double yr[] = {    1, 2,      1,     -1,-2,    -1};
59
60          /**
61           * These int arrays are use of storing the 6 locations fo the points.
62           * It has to be int arrays because the fillPolygon method only takes
63           * in int arrays.
64           */
65          int y2[] = new int[6];
66          int x2[] = new int[6];
67
68          /**
69           * A for loop that casts the 6 points from the double arrays into the
70           * int array
71           */
72          for(int i = 0; i < 6; i++){
73
74              /**
75               * Moves the x position and moves the hexagon a ½ hexgon in the x
76               * direction to that they fall into place.
77               */
78              x2[i] = (int)((xr[i] * UNIT) + (2 * SQRT3 * UNIT * xPos) + (SQRT3 * yPos * UNIT) + XOFF);
79
80              /**
81               * Moves the x position and moves the hexagon a ½ hexgon down in
82               * the y direction to that they fall into place.
83               */
84              y2[i] = (int)((yr[i] * UNIT) + (3 * UNIT * yPos) + YOFF);
85          }
86
87          /**
88           * Draws the polygon in the polygon array at the specified x and y
89           * position using the two int arrays.
90           * Sets the color to the specified player color (RED or BLUE)
91           * Sets the border color to black.
92           */
93          polygonArray[xPos][yPos] = new Polygon(x2, y2, 6);
94          g.setColor(c);
95
```

```
 96          g.fillPolygon (x2, y2, 6);
 97          g.setColor(Color.black);
 98
 99          /**
100           * If the gameboard is weigthed a string of the weigth is also drawn
101           * inside the hexagon
102           */
103          if(weigthed){
104              g.drawString("" + weight, x2[1] - 5, y2[1] - 20);
105              g.drawString(cost, x2[1] - 5, y2[1] - 10);
106          }
107
108          /**
109           * if the current hexagon is the root of a given set a black circel
110           * is drawn inside the hexagon
111           */
112          if(root){
113              g.fillOval(x2[1] - 8, y2[1] - 27, 17, 17);
114          }
115
116          /**
117           * Draws the hexagon (as a polygon) at the location given by the x2 and
118           * the y2 array the 6 points.
119           */
120          g.drawPolygon (x2, y2, 6);
121      }
122
123      /**
124       * @return Returns the polygonArray.
125       */
126      public Polygon[][] getPolygonArray() {
127          return polygonArray;
128      }
129
130      /**
131       * @param polygonArray The polygonArray to set.
132       */
133      public void setPolygonArray(Polygon[][] polygonArray) {
134          this.polygonArray = polygonArray;
135      }
136  }
137
138  //---------------------------------------------------------------------------
139  //                          E N D   O F   F I L E
140  //---------------------------------------------------------------------------
141
```

```java
package theHexGame;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.GridLayout;
import java.awt.Point;
import java.awt.event.*;

import java.util.LinkedList;

import javax.swing.JApplet;
import javax.swing.JButton;

/**
 * @author Simon Larsen & Jens Frederiksen
 */
public class GameManager extends JApplet implements MouseListener, ActionListener{

    /**
     * The height and width of the applet
     */
    private final int WIDTH = 800;
    private final int HEIGHT = 500;

    /**
     * The size of the board
     */
    private int boardSize = 14;

    /**
     * The player variables.
     * The winner boolean that stops the game if a winner is found.
     * A hotseat variable that constantly switches between the two players.
     * And the player array with the two players.
     */
    private boolean winner = false;
    private int hotseat = 0;
    private Player[] PLAYERS = {
            new Player("RED"),
            new Player("BLUE")
    };

    /**
     * boolean used throughout the application to determine if the game is
     * taking place on a weighted gameboard or not.
     */
```

```
49      private boolean weigthedGame = false;
50
51      /**
52       * Initializing the gameboard and the board. The gameboard draws the graphics
53       * for the game and the board maintains the status of the game.
54       */
55      private GameBoard gameBoard = new GameBoard(boardSize);
56      private Board board = new Board(boardSize, PLAYERS[0], PLAYERS[1], weigthedGame);
57
58      /**
59       * The two buttons for either a normal or a weighted game.
60       */
61      private JButton newGameNormalButton = new JButton ("New normal game");
62      private JButton newGameWeightedButton = new JButton ("New weigthed game");
63
64      /**
65       * A container for the two buttons. By using this you can ensure that the
66       * buttons are located at the bottom of the application. See the layout.
67       */
68      private Container buttons = new Container();
69
70      /**
71       * The pane on which all the elements for the application is drawn
72       */
73      private Container pane = getContentPane();
74
75
76      /* (non-Javadoc)
77       * @see java.applet.Applet#init()
78       */
79      public void init() {
80
81          /**
82           * Initialize the applet with the given width and height.
83           */
84          setSize(WIDTH, HEIGHT);
85
86          /**
87           * Add the mouselistener for the gameboard and the actionlistener for
88           * buttons
89           */
90          addMouseListener(this);
91          newGameNormalButton.addActionListener(this);
92          newGameWeightedButton.addActionListener(this);
93
94          /**
95           * Creates a gridlayout for the button, each added element is allocated
96           * space in the gridlayout. And the background is set to white.
```

```
 97           */
 98          buttons.setLayout(new GridLayout());
 99          buttons.setBackground(Color.WHITE);
100          buttons.add(newGameNormalButton);
101          buttons.add(newGameWeightedButton);
102
103          pane.setLayout(new BorderLayout());
104          pane.add(buttons, BorderLayout.SOUTH);
105
106          setVisible(true);
107      }
108
109      /**
110       * The paint method.
111       * Changes the color of each hexagon acording to:
112       *  - no player
113       *  - Or one of the two players (RED or BLUE)
114       * There is a use of float variables to paint the gameboard in different
115       * grey values if the gameboard is weigthed.
116       *
117       * @see java.awt.Component#paint(java.awt.Graphics)
118       */
119      public void paint(Graphics g){
120
121          Color c = Color.WHITE;
122          float h = 0.0f;
123          float s = 0.0f;
124          float b = 1.0f;
125
126          for(int i = 0; i < boardSize; i++){
127              for(int j = 0; j < boardSize; j++){
128                  if(board.getVertexArray()[i][j].getPlayer() != null){
129                      if(board.getVertexArray()[i][j].getPlayer() == PLAYERS[0]){
130                          if(weigthedGame){
131                              b = 1.0f;
132                              s = 1.0f;
133                              h = 0f;
134                          }
135                          else {
136                              c = Color.RED;
137                          }
138                      }
139
140                      else if(board.getVertexArray()[i][j].getPlayer() == PLAYERS[1]){
141                          if(weigthedGame){
142                              b = 1.0f;
143                              s = 1.0f;
144                              h = 0.66f;
```

```
145                         }
146                         else {
147                             c = Color.BLUE;
148                         }
149                     }
150                 }
151 
152                 else{
153                     h = 0;
154                     s = 0;
155                     b = 1f - ((float)board.getVertexArray()[i][j].getWeight())/100;
156                     c = Color.WHITE;
157                 }
158 
159                 if(weigthedGame) {
160                     c = Color.getHSBColor(h, s, b);
161                 }
162 
163                 if(winner){
164                     if(board.getVertexArray()[i][j].getColor() == "BLACK"){
165                         c = Color.DARK_GRAY;
166                     }
167                     else if(board.getVertexArray()[i][j].getColor() == "short"){
168                         c = Color.GREEN;
169                     }
170                 }
171                 gameBoard.drawHexagon(i, j, g , c, board.getVertexArray()[i][j], weigthedGame, board.
                    getVertexArray()[i][j].isRoot());
172             }
173         }
174     }
175 
176     /**
177      * The mousePressed mouselistneer.
178      * Looks for where the user have clicked on the gameboard and delegates
179      * information to the board and checks wether the current player have won
180      * by putting down that playing piece.
181      *
182      * @see java.awt.event.MouseListener#mousePressed(java.awt.event.MouseEvent)
183      */
184     public void mousePressed(MouseEvent e){
185         if(!winner){
186             boolean fund = false;
187             /**
188              * Gets the point where the user clicked
189              */
190             Point p = e.getPoint();
191 
```

```
192                 /**
193                  * Runs through the entire gameboard
194                  */
195                 for(int i = 0; i < boardSize && !fund; i++) {
196                     for(int j = 0; j < boardSize && !fund; j++) {
197                         if(gameBoard.getPolygonArray()[i][j].contains(p) && board.getVertexArray()[i][j].getPlayer
                            () == null) {
198                             board.getVertexArray()[i][j].setPlayer(PLAYERS[hotseat]);
199                             updateStructures(i, j, PLAYERS[hotseat], board);
200 
201                             /**
202                              * If player blue's start and end vertex is in the same
203                              * component the player has won the game
204                              */
205                             if(logic.SetLogic.sameComponent(board.getEndBlue(), board.getStartBlue())) {
206                                 System.out.println(PLAYERS[hotseat].getColor() + " Won");
207 
208                                 /**
209                                  * The shorteste path is calulated, either by using
210                                  * Dijkstra (if it is a weigthed game, or breath-
211                                  * first-search if it is not.
212                                  */
213                                 if(weigthedGame) {
214                                     logic.Graph.dijkstra(board, board.getStartBlue(), board.getEndBlue());
215                                 }
216                                 else {
217                                     logic.Graph.breathFirstSearch(board.getStartBlue());
218                                 }
219                                 logic.Graph.shortestPath(board.getStartBlue(), board.getEndBlue());
220                                 winner = true;
221 
222                             }
223 
224                             /**
225                              * If player red's start and end vertex is in the same
226                              * component the player has won the game
227                              */
228                             if(logic.SetLogic.sameComponent(board.getEndRed(), board.getStartRed())){
229                                 System.out.println(PLAYERS[hotseat].getColor() + " Won");
230 
231                                 /**
232                                  * The shorteste path is calulated, either by using
233                                  * Dijkstra (if it is a weigthed game, or breath-
234                                  * first-search if it is not.
235                                  */
236                                 if(weigthedGame) {
237                                     logic.Graph.dijkstra(board, board.getStartRed(), board.getEndRed());
238                                 }
```

```
239                                 else {
240                                     logic.Graph.breathFirstSearch(board.getStartRed());
241                                 }
242                                 logic.Graph.shortestPath(board.getStartRed(), board.getEndRed());
243                                 winner = true;
244                             }
245                             hotseat ^= 1;
246                             repaint();
247                             fund = true;
248                         }
249                     }
250                 }
251             }
252         }
253
254         /**
255          * All thses methods must be implemented when the class in implementing
256          * the mouselistner. Even though they are not used.
257          */
258         public void mouseExited(MouseEvent e) {}
259         public void mouseReleased(MouseEvent e) {}
260         public void mouseEntered(MouseEvent e) {}
261         public void mouseClicked(MouseEvent e) {}
262
263         /**
264          * The actionlistener for the buttons.
265          *
266          * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
267          */
268         public void actionPerformed(ActionEvent e) {
269             Object button = e.getSource();
270
271             if(button.equals(newGameNormalButton)) {
272                 weigthedGame = false;
273                 newGame(weigthedGame);
274             }
275
276             if(button.equals(newGameWeightedButton)) {
277                 weigthedGame = true;
278                 newGame(weigthedGame);
279             }
280         }
281
282         /**
283          * New game method. Creates a new game that is either weigthed or not,
284          * accordingly to the boolean weigthedGame
285          *
286          * @param weigthedGame boolean - if the game is being played on a weigthed
```

```
287       * gameboard or not.
288       */
289      private void newGame(boolean weigthedGame){
290          this.weigthedGame = weigthedGame;
291          gameBoard = new GameBoard(boardSize);
292          board = new Board(boardSize, PLAYERS[0], PLAYERS[1], weigthedGame);
293          winner = false;
294          hotseat = 0;
295          repaint();
296      }
297
298      /**
299       * @param xPos int - The x position of the vertex
300       * @param yPos int - the y position of the vertex
301       * @param p Player - the current player
302       * @param board Board - The current board
303       */
304      public static void updateStructures(int xPos, int yPos, Player p, Board board){
305          Vertex v = board.getVertexArray()[xPos][yPos];
306          LinkedList neighborsList = logic.Graph.findNeighbors(xPos, yPos, p, board);
307          logic.Graph.attachNeighborsToEachOther(v, neighborsList);
308          v.setPlayer(p);
309          logic.SetLogic.makeSet(v);
310
311          logic.SetLogic.connectedComponent(v, neighborsList);
312      }
313  }
314
315  //---------------------------------------------------------------------------
316  //                        E N D   O F   F I L E
317  //---------------------------------------------------------------------------
318
```

```java
package logic;

import java.util.LinkedList;
import java.util.Iterator;

import theHexGame.Board;
import theHexGame.Player;
import theHexGame.Vertex;

/**
 * @author Simon Larsen & Jens Frederiksen
 */
public class Graph{

    /**
     * The positions for a vertex's  6 neighbors [0, -1][1, -1][1, 0][0, 1][-1, 1][-1, 0]
     */
    private static final int[] neigborPos = {0, -1, 1, -1, 1, 0, 0, 1, -1, 1, -1, 0};

    public static LinkedList adjancencyList = new LinkedList();

    /**
     * Find the neighbors for a vertex with a specific player
     * @param xPos  an int with the position for the vertex in the vertexArray
     * @param yPos  an int with the position for the vertex in the vertexArray
     * @param p The player with the hotseat
     *
     * @return  the neighbors in a linkedList
     */
    public static LinkedList findNeighbors(int xPos, int yPos, Player p, Board b){
        LinkedList neighbors = new LinkedList();

        if(xPos == 0 && b.getStartRed().getPlayer().equals(p)){
            neighbors.add(b.getStartRed());
        }
        else if(xPos == b.getBoardSize()-1 && b.getStartRed().getPlayer().equals(p)){
            neighbors.add(b.getEndRed());
        }
        else if(yPos == 0 && b.getStartBlue().getPlayer().equals(p)){
            neighbors.add(b.getStartBlue());
        }
        else if(yPos == b.getBoardSize()-1 && b.getStartBlue().getPlayer().equals(p)){
            neighbors.add(b.getEndBlue());
        }

        /**
         * Runs through the board six times to find the six possible neighbors.
         * 6 x and y positions (6 x 2 = 12).
```

```
49           */
50          for(int i = 0; i<12; i += 2){
51              if(xPos + neigborPos[i] >= 0
52                      && xPos + neigborPos[i] < b.getBoardSize()
53                      && yPos + neigborPos[i+1] >= 0
54                      && yPos + neigborPos[i+1] < b.getBoardSize()){
55                  Vertex actualNeighbor = b.getVertexArray()[xPos + neigborPos[i]][yPos + neigborPos[i+1]];
56
57                  if(actualNeighbor.getPlayer() != null){
58                      if(actualNeighbor.getPlayer().equals(p)){
59                          neighbors.add(actualNeighbor);
60                      }
61                  }
62              }
63          }
64          return neighbors;
65      }
66
67      /**
68       * Attach neighbors to the vertex's neighbors linkedList and vise versa
69       *
70       * @param xPos  an int with the position for the vertex in the vertexArray
71       * @param yPos  an int with the position for the vertex in the vertexArray
72       * @param neighbors a LinkedList with a Vertex neighbors, finded by the method findNeighbors
73       */
74      public static void attachNeighborsToEachOther(Vertex v, LinkedList neighbors){
75          Iterator itr = neighbors.iterator();
76
77          while (itr.hasNext()) {
78              Vertex element = (Vertex) itr.next();
79              v.setNeighbor(element);
80              element.setNeighbor(v);
81          }
82      }
83      /**
84       * @param s
85       * @return
86       */
87      public static void breathFirstSearch(Vertex s){
88          LinkedList queue = new LinkedList();
89          Vertex n;
90
91          /**
92           * The distance for this is 0
93           */
94          s.setDistanceFromRoot(0);
95
96          /**
```

```
 97          * s has no graph parent
 98          */
 99         s.setParent(null);
100
101         /**
102          * Put s in the queue of vertexs that is going to be searched
103          */
104         queue.add(s);
105
106         /**
107          * This vertex is being searched.
108          */
109         s.setColor("GREY");
110
111         /**
112          * Keep searching as long as the queue is not empty
113          */
114         while(!queue.isEmpty()){
115             n = (Vertex)queue.removeFirst();
116
117             for (int i = 0; i < n.getNeighbors().size(); i++) {
118                 Vertex v = (Vertex)n.getNeighbors().get(i);
119                 if(v.getColor().equals("WHITE")){
120                     v.setDistanceFromRoot(n.getDistanceFromRoot() + 1);
121                     v.setGraphParent(n);
122                     queue.add(v);
123                     v.setColor("GREY");
124                 }
125             }
126             n.setColor("BLACK");
127         }
128     }
129
130     /**
131      * The dijkstra algoritm inspired by http://www.gamasutra.com/features/19970801/pathfinding.htm
132      * and Introduction to Algorithms.
133      * @param b Board - The board the is going to be searched.
134      * @param start Vertex - The start vertex for the search.
135      * @param end Vertex - The end vertex fof the search
136      */
137     public static void dijkstra(Board b, Vertex start, Vertex end){
138         LinkedList queue = new LinkedList();
139         Iterator iter;
140         Vertex n;
141         int newCost;
142
143         initSingleSource(b, start);
144
```

```
145          /**
146           * Adds the start vertex to the queue
147           */
148          queue.add(start);
149
150          /**
151           * Keep searching the vertexs in the queue as long as it is not empty
152           */
153          while(!queue.isEmpty()){
154              n = (Vertex)queue.removeFirst();
155              n.setColor("BLACK");
156
157              /**
158               * Iterats through all the neighbors
159               */
160              for (iter = n.getNeighbors().iterator(); iter.hasNext();) {
161                  Vertex element = (Vertex) iter.next();
162
163                  /**
164                   * Check if the cost of the Vertex element is higher than the
165                   * Vertex n in the queue.
166                   */
167                  edgeRelaxation(n, element);
168
169                  if(!element.getColor().equals("BLACK")){
170                      addToPriorityQueue(queue, element);
171                  }
172              }
173
174              /**
175               * Stops the search if the vertex it encounters is the end vertex.
176               * Implemented because we already know the end vertex.
177               */
178              if(n.equals(end)){
179                  break;
180              }
181          }
182      }
183
184      /**
185       * Inspired from Introduction to Algoritmhs (page 520)
186       * @param b Board - The board if the game
187       * @param start Vertex - The start vertex in the search
188       */
189      private static void initSingleSource(Board b, Vertex start){
190          for (int i = 0; i < b.getVertexArray().length; i++) {
191              for (int j = 0; j < b.getVertexArray()[i].length; j++) {
192                  b.getVertexArray()[i][j].setCost(Integer.MAX_VALUE);
```

```
193                     b.getVertexArray()[i][j].setGraphParent(null);
194                 }
195             }
196             start.setCost(0);
197             start.setGraphParent(null);
198         }
199 
200         /**
201          * Checks the cost of Vertex v compared to the cost of Vertex u
202          * @param u Vertex
203          * @param v Vertex
204          */
205         private static void edgeRelaxation(Vertex u, Vertex v){
206             int newCost = u.getCost() + (u.getWeight() + v.getWeight());
207 
208             if(v.getCost() > newCost){
209                 v.setCost(newCost);
210                 v.setGraphParent(u);
211             }
212         }
213 
214         /**
215          * Adds the specified vertex to the priority queue
216          *
217          * @param queue LinkedList - The queue
218          * @param v Vertex
219          */
220         private static void addToPriorityQueue(LinkedList queue, Vertex v){
221             LinkedList queueCopy = new LinkedList(queue);
222             Iterator iter = queueCopy.iterator();
223             while (iter.hasNext()) {
224                 Vertex element = (Vertex) iter.next();
225 
226                 /**
227                  * If the cost is lower than the others in the queue the vertex
228                  * is inserted at the front of the more expensive vertexs.
229                  */
230                 if(v.getCost() < element.getCost()){
231                     /**
232                      * The vertex is placed on the place where the vertex with the
233                      * higher cost was placed. The more expensive vertexs are all
234                      * moved one place down.
235                      */
236                     queue.add(queue.indexOf(element), v);
237 
238                     /**
239                      * The vertex is placed in the queue so the method is stoped.
240                      */
```

```
241                     return;
242                 }
243             }
244
245             /**
246              * The input vertex is the most expensive in the queue, so it is put
247              * in the end of the queue.
248              */
249             queue.addLast(v);
250         }
251
252         /**
253          * @param Vertex start
254          * @param Vertex end
255          */
256         public static void shortestPath(Vertex start, Vertex end){
257             if(end.getGraphParent() != null)
258                 findGraphRoot(end);
259             else
260                 findGraphRoot(start);
261         }
262
263         /**
264          * @param v Vertex
265          * @return <code>true</code> if this vertex is the root of the graph
266          */
267         private static Vertex findGraphRoot(Vertex v){
268             v.setColor("short");
269             if(v.getGraphParent() == null){
270                 return v;
271             }
272             return findGraphRoot(v.getGraphParent());
273         }
274
275         public static void printAdjancencyList(Player p){
276             Iterator iter = adjancencyList.iterator();
277             while (iter.hasNext()) {
278                 Vertex element = (Vertex) iter.next();
279                 if(element.getParent() == null){
280                     System.out.println("ID: " + element.getPosition() + " " + element.isRoot());
281                 }
282                 else{
283                     System.out.print(element.getPosition() + " -> " );
284                     element.parentString();
285                 }
286             }
287             System.out.println("-----------------");
288         }
```

```
289  }
290
291  //---------------------------------------------------------------------------
292  //                          E N D   O F   F I L E
293  //---------------------------------------------------------------------------
294
```

```java
package theHexGame;

/**
 * @author Simon Larsen & Jens Frederiksen
 */
public class Player {

    /**
     * The color for the player. Being either RED og BLUE
     */
    private String color;

    /**
     * The Player constructor.
     * @param color String - the color for the player (either RED or BLUE)
     */
    public Player(String color){
        this.color = color;
    }

    /**
     * @return Returns the color as a String.
     */
    public String getColor() {
        return color;
    }


    /* (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    public String toString() {
        return color.toString();
    }
}

//------------------------------------------------------------------------------
//                        E N D   O F   F I L E
//------------------------------------------------------------------------------

```

```java
package logic;

import java.util.LinkedList;
import java.util.Iterator;

import theHexGame.Vertex;


/**
 * @author Simon Larsen & Jens Frederiksen
 */
public class SetLogic {

    /**
     * @param v Vertex
     */
    public static void makeSet(Vertex v){
        v.setRoot(true);
        v.setDepth(0);
    }

    /**
     * @param v1 Vertex
     * @param v2 Vertex
     */
    public static void union(Vertex v1, Vertex v2){
        Vertex root1 = findSet(v1);
        Vertex root2 = findSet(v2);
        if(root1.getDepth() > root2.getDepth()){
            root2.setParent(root1);
            root2.setRoot(false);
        }
        else{
            root1.setParent(root2);
            root1.setRoot(false);
            if(root1.getDepth() == root2.getDepth()){
                root2.setDepth(root1.getDepth() + 1);
            }
        }
    }

    /**
     * @param v Vertex
     * @return the vertex that is the root of the set.
     */
    public static Vertex findSet(Vertex v){
        if(v.getParent() == null){
            return v;
```

```
49          }
50          v.setParent(findSet(v.getParent()));
51          return v.getParent();
52      }
53
54      /**
55       * @param v Vertex
56       * @param neighbors
57       */
58      public static void connectedComponent(Vertex v, LinkedList neighbors){
59          makeSet(v);
60          Iterator iter = neighbors.iterator();
61          Vertex temp;
62          while(iter.hasNext()){
63              temp = (Vertex)iter.next();
64              if(!sameComponent(v, temp)){
65                  union(v, temp);
66              }
67          }
68      }
69
70      public static boolean sameComponent(Vertex v1, Vertex v2){
71          if(findSet(v1).equals(findSet(v2))){
72              return true;
73          }
74          return false;
75      }
76  }
77
78  //---------------------------------------------------------------------------
79  //                        E N D   O F   F I L E
80  //---------------------------------------------------------------------------
81
```

```java
package theHexGame;

import java.util.*;

/**
 * @author Simon Larsen & Jens Frederiksen
 */
public class Vertex{

    /**
     * The pole for the vertex. Being either start or the end vertex for blue
     * or red player
     */
    private String pole;

    /**
     * The x and the y position for the vertex in the board array
     */
    private int xPos;
    private int yPos;

    /**
     * The player that "owns" the vertex
     */
    private Player player;

    /**
     * The parent for the vertex. Can be null, if the vertex have no parent and
     * therefor is the root of the set.
     */
    private Vertex parent;

    /**
     * How deep the set that the vertex currently is in, is.
     */
    private int depth;

    /**
     * A linkedlist consisting of all the vertex's neighbors.
     */
    private LinkedList neighbors;

    /**
     * Wether the vertex is the root of the set or not.
     */
    private boolean root;

    /**
```

```
49       * Graph attributs. The color of the vertex, the distance from the root of
50       * the graph and the current vertex's graph-parent.
51       */
52      private String color;
53      private int distanceFromRoot;
54      private Vertex graphParent;
55
56      /**
57       * Weigthed graph attributes. The weight of the vertex and its cost
58       */
59      private int weight;
60      private int cost;
61
62      /**
63       *  Vertex Contructor
64       *
65       *  Precondition: The vertex x and y posistion in the array
66       *  Postcondition:
67       *
68       *  @param int x position, int y position
69       * */
70      public Vertex(int xPos, int yPos){
71          this.xPos = xPos;
72          this.yPos = yPos;
73
74          pole = null;
75          player = null;
76          parent = null;
77          neighbors = new LinkedList();
78          root = false;
79
80          color = "WHITE";
81          distanceFromRoot = 0;
82          graphParent = null;
83
84          weight = 1;
85          cost = 0;
86      }
87
88      /**
89       * The overloaded Vertex Constructor
90       *
91       * Precondition: The player p and a string with the pole describtion.
92       * @param p Player - The player that "owns" the vertex.
93       * @param pole String
94       */
95      public Vertex(Player p, String pole){
96          this.player = p;
```

```
 97          this.pole = pole;
 98          parent = null;
 99          neighbors = new LinkedList();
100          root = false;
101
102          color = "WHITE";
103          distanceFromRoot = 0;
104          graphParent = null;
105
106          weight = 1;
107          cost = Integer.MAX_VALUE;
108      }
109
110      /**
111       * Sets a vertex v to belong in the linkedlist of neighbors if it is not
112       * already present herein.
113       *
114       * @param v Vertex - The vertex that is going to be added to the neighbor
115       * list.
116       */
117      public void setNeighbor(Vertex v){
118          if(!neighbors.contains(v)) neighbors.add(v);
119      }
120
121      /**
122       * @return Returns a linkedlist with the neighbors.
123       */
124      public LinkedList getNeighbors() {
125          return neighbors;
126      }
127
128      /**
129       * @param parent Vertex - The parent to set.
130       */
131      public void setParent(Vertex parent) {
132          this.parent = parent;
133      }
134
135      /**
136       * @return Returns the parent of the vertex.
137       */
138      public Vertex getParent() {
139          return parent;
140      }
141
142      /**
143       * @return Returns the player of the vertex
144       */
```

```
145     public Player getPlayer() {
146         return player;
147     }
148 
149     /**
150      * @param player Player - The player to set.
151      */
152     public void setPlayer(Player player) {
153         this.player = player;
154     }
155 
156     /**
157      * @return Returns the depth of the set that the vertex is part of.
158      */
159     public int getDepth() {
160         return depth;
161     }
162 
163     /**
164      * @param depth int - The depth to set.
165      */
166     public void setDepth(int depth) {
167         this.depth = depth;
168     }
169 
170     /**
171      * The compareTo method
172      *
173      * @param o Object - The vertex that the current vertex is going to be
174      * compared to
175      * @return int - Returns 0 if the vertexs compare are equal. If not an
176      * interger is returned.
177      */
178     public int compareTo(Object o){
179         Vertex other = (Vertex) o;
180         if(this.compareTo(other) != 0){
181             return this.compareTo(other);
182         }
183         return 0;
184     }
185 
186     /**
187      * @return Returns the color of the vertex.
188      */
189     public String getColor() {
190         return color;
191     }
192 
```

```
193      /**
194       * @param color String - The color to set.
195       */
196      public void setColor(String color) {
197          this.color = color;
198      }
199
200      /**
201       * @return Returns the distance from root.
202       */
203      public int getDistanceFromRoot() {
204          return distanceFromRoot;
205      }
206
207      /**
208       * @param distanceFromRoot int - The distanceFromRoot to set.
209       */
210      public void setDistanceFromRoot(int distanceFromRoot) {
211          this.distanceFromRoot = distanceFromRoot;
212      }
213
214      /**
215       * @return Returns a boolean that is true or false if the vertex is the
216       * the root or not.
217       */
218      public boolean isRoot() {
219          return root;
220      }
221
222      /**
223       * @param root boolean - The root to set.
224       */
225      public void setRoot(boolean root) {
226          this.root = root;
227      }
228
229      /**
230       * @return Returns the graphParent.
231       */
232      public Vertex getGraphParent(){
233          return graphParent;
234      }
235
236      /**
237       * @param graphParent Vertex - The graphParent to set.
238       */
239      public void setGraphParent(Vertex graphParent) {
240          this.graphParent = graphParent;
```

```
241     }
242
243     /**
244      * @param v
245      * @return
246      */
247     public boolean findGraphRoot(Vertex v){
248         this.setColor("short");
249         if(this.graphParent == null){
250             return true;
251         }
252         else if(this.graphParent.equals(v)){
253             return false;
254         }
255         return graphParent.findGraphRoot(v);
256      }
257
258     /**
259      * @return Returns the cost.
260      */
261     public int getCost() {
262         return cost;
263     }
264
265     /**
266      * @param cost The cost to set.
267      */
268     public void setCost(int cost) {
269         this.cost = cost;
270     }
271
272     /**
273      * @return Returns the weight.
274      */
275     public int getWeight() {
276         return weight;
277     }
278
279     /**
280      * @param weight The weight to set.
281      */
282     public void setWeight(int weight) {
283         this.weight = weight;
284     }
285
286     /**
287      * @return String with the array position of the vertex (x,y)
288      */
```

```
289      public String getPosition(){
290          return "x:" + this.xPos + "/y:" + this.yPos + "; ";
291      }
292
293      /* (non-Javadoc)
294       * @see java.lang.Object#toString()
295       */
296      public String toString(){
297          String temp;
298          if(this.parent != null)temp = " parent: " + this.parent.getPosition();
299          else temp = " no parent";
300          return this.getPosition() +  temp;
301      }
302
303      /**
304       * Prints out "<- parent!" at the vertex that is the parent.
305       */
306      public void parentString(){
307          System.out.print(this.getPosition());
308          if(this.parent != null){
309              this.parent.parentString();
310          }
311          else{
312              System.out.println("<- parent!");
313          }
314      }
315  }
316
317  //---------------------------------------------------------------------------
318  //                            E N D   O F   F I L E
319  //---------------------------------------------------------------------------
320
```