Spanneröar och spannervägar

Mikael Nilsson

Examensarbete för 30 hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in Computer Science, 30 ECTS Credits
Department of Computer Science, Faculty of Science, Lund University

Spanneröar och spannervägar

Sammanfattning

I det här magisterarbetet undersöks om det är möjligt att på ett effektivt sätt dela upp en graf i spanneröar, dvs. öar som uppfyller spanneregenskapen som består i att avståndet mellan två noder via grafens bågar inte får vara för stort i förhållande till det euklidiska avståndet mellan noderna. Att hitta en uppdelning som skapar så få kontaktpunkter mellan öarna som möjligt eftersöks. Ett antal heuristiker testas och utvärderas med resultatet att en heuristik som använder sig av MAX-FLOW för att dela upp noder som bryter mot spannervillkoret presterar bäst för täta grafer medan en heuristik av typ *single-link clustering* presterar bäst för glesa grafer.

I arbetet visas att problemet att finna en spannerväg, en väg där noderna som passeras uppfyller spannervillkoret, mellan två noder i en graf av storlek n är *NP*-komplett om spannerkonstanten är större än $n^{1+1/k}\sqrt{k}$ för något heltal k. En algoritm för att hitta spannervägar med komplexiteten $O(2^{0.822n})$ presenteras.

Ett specialproblem där grafen ligger längs tallinjen och bara har noder på heltalspunkter studeras slutligen och här konstrueras en algoritm med komplexiteten $O(2^{c(\log n)^2})$ där c är en konstant som beror på spannerkonstanten. Till exempel nås $O(2^{5.32(\log n)^2})$ för stretch 1.5.

Spanner Islands and Spanner Paths

Abstract

In this Master Thesis the possibility to efficiently divide a graph into spanner islands is examined. Spanner islands are islands of the graph that fulfill the spanner condition, that the distance between two nodes via the edges in the graph cannot be too far, regulated by the stretch constant, compared to the Euclidian distance between them. In the resulting division the least number of nodes connecting to other islands is sought-after. Different heuristics are evaluated with the conclusion that for dense graphs a heuristic using MAX-FLOW to divide problematic nodes gives the best result whereas for sparse graphs a heuristic using the single-link clustering method performs best.

The problem of finding a spanner path, a path fulfilling the spanner condition, between two nodes is also investigated. The problem is proven to be *NP*-complete for a graph of size n if the spanner constant is greater than $n^{1+1/k}\sqrt{k}$ for some integer k. An algorithm with complexity $O(2^{0.822n})$ is given.

A special type of graph where all the nodes are located on integer locations along the real line is investigated. An algorithm to solve this problem is presented with a complexity of $O(2^{c(\log n)^2})$, where c is a constant depending only on the spanner constant. For instance, the complexity $O(2^{5.32(\log n)^2})$ can be reached for stretch 1.5.

Förord

Från början var jag inriktad på att göra ett arbete inom området beräkningskomplexitet. Det fanns dock ingen handledning inom detta område varför jag valde område efter handledare istället. Christos Levcopoulos har genom åren varit ett utmärkt val. Han kom med idén till arbetet och har kontinuerligt ställt upp och diskuterat alla problem jag ställts inför trots att jag ändrat fokus under arbetets gång. Från början fokuserade jag på beräkningskomplexitet då det verkade alldeles för svårt att hitta en effektiv lösning till problemet. Denna del av arbetet var väldigt teoretisk. Mot arbetets slut bytte jag fokus och fokuserade mer på att undersöka olika implementationer av approximeringar. Detta gav arbetet en praktisk motpart till den teoretiska delen och mig lärdomen att man inte förstår ett problem helt förrän man testat det i både praktik och teori. Det finns i princip tre delar som är helt fristående vilket ger arbetet en ganska omfattande storlek. Det känns dock som att de alla måste ingå för att arbetet skall vara komplett. Delarna är beräkningskomplexiteten, heltalsgrafen och heuristikerna.
Jag vill rikta ett tack till Rolf Karlsson som jag även haft långa givande diskussioner med och till Christian Grosse som korrekturläst innehållet.
Slutligen ett stort tack till min sambo Linda Jansson som varit med hela resan som bollplank och motivator när det känts jobbigt. Alla behöver en coach ibland.

Mikael Nilsson
Lund, augusti 2009

Innehåll

1 Inledning **6**

1.1 Bakgrund . . . 6
1.2 Syfte . . . 7
1.3 Metod . . . 7
1.4 Tidigare resultat . . . 7
1.5 Översikt . . . 7

2 Definitioner och grundläggande Teori **10**

2.1 Spanners . . . 10
2.2 Flödesgrafer . . . 11
2.3 Asymptotisk notation . . . 12
2.4 Klasserna P och NP . . . 12
2.4.1 Klassen P . . . 14
2.4.2 Klassen NP . . . 14
2.4.3 Reduktioner . . . 15
2.4.4 NP-kompletta problem . . . 16
2.5 Algoritmer . . . 17
2.5.1 Dijkstras algoritm . . . 17
2.5.2 Floyds algoritm . . . 17
2.5.3 Edmonds-Karps algoritm . . . 18

3 Spannervägar **19**

3.1 Algoritmer . . . 19
3.2 Komplexitetsbestämning . . . 20
3.2.1 Reduktion från 3SAT . . . 21
3.2.2 Reduktion från Väg med förbjudna par . . . 27
3.2.3 Jämförelse mellan reduktionerna . . . 34
3.3 Små spannerkonstanter . . . 37
3.4 Ett par observationer . . . 37
3.5 Slutsats och vidare arbete . . . 39

4 Heltalsgrafen **41**
4.1 Introduktion och definition 41
4.2 Kända egenskaper 42
4.3 Inkrementell metod 48
4.4 Alternativ inkrementell metod 53
4.5 Slutsats och vidare arbete 62

5 Uppdelning av en graf i spanneröar **64**
5.1 Inledning 64
5.2 Observation 65
5.3 Metod 66
5.3.1 Slumpgrafer 66
5.3.2 Heuristiker 69
5.3.3 Testgrafer 92
5.4 Resultat 93
5.4.1 14 noder, 10000 grafer 94
5.4.2 16 noder, 1000 grafer 95
5.4.3 20 noder, 10000 grafer 96
5.4.4 40 noder, 1000 grafer 97
5.4.5 Exekveringstider 98
5.5 Jämförelser 98
5.5.1 Jämförelse mellan Diametermetoderna 99
5.5.2 Jämförelse mellan Multiklustermetoderna 100
5.5.3 Jämförelse mellan M1 och M3 100
5.5.4 Jämförelse mellan M2b och M4b 100
5.5.5 Jämförelse mellan flödesheuristikerna 101
5.5.6 Jämförelse mellan de olika metodtyperna 104
5.6 Analys 104
5.6.1 Analys av Diametermetoderna 104
5.6.2 Analys av Multiklustermetoderna 105
5.6.3 Analys av lokala och globala heuristiker 105
5.6.4 Analys av flödesheuristikerna 107
5.6.5 Analys av Sveplinjemetoden 109
5.6.6 Analys av exekveringstider 110
5.7 Slutsats och vidare arbete 110
5.7.1 Vidare arbete 111

A Redovisning av beräkningar **114**
A.1 Antalet vägar mellan s och t 114
A.2 Tidskomplexitet för delgrafalgoritmen 115
A.3 Tidskomplexitet för splittringsalgoritmen 116
A.4 Jämförande exempel 122
A.5 Beräkningar för heuristikerna 125
A.6 Verktyget 125

Kapitel 1

Inledning

1.1 Bakgrund

Betrakta problemet med att bestämma avståndet mellan två städer. Att lösa detta problem för ett lands alla städer utmynnar ofta i en tvådimensionell tabell, med städerna längs vänsterkolumnen och över första raden samt de inbördes avstånden i tabellen. Man inser att tabellen troligen innehåller redundant data eftersom avståndet från A till B troligen är lika med avståndet från B till A. På så vis kan man få ner tabellen till halva storleken, men den är fortfarande $O(n^2)$ stor givet n städer.

Om man inte längre kräver att avstånden är exakta är det möjligt att minska det utrymme som används men ändå leverera svaret i konstant tid. Andersson et al. [1] visar att givet en graf $H_1 = (V, E_1)$, där V är noderna och E_1 är bågarna, som består av N olika disjunkta komponenter som uppfyller spanneregenskaper (se kapitel 2.1), samt en graf $H_2 = (V, E_2)$ med bågar mellan M olika noder, så är det är möjligt att konstruera en datastruktur med storlek $O(M^2 + n \log n)$ som svarar på $(1+\varepsilon)$-approximativa frågor om kortaste vägar i grafen $G = (V, E_1 \cup E_2)$. Här är det tänkt att de M noderna fungerar som kopplingar mellan de N disjunkta komponenterna med spanneregenskaper, men det är också möjligt att flera av noderna som sammanbinds i H_2 finns i samma komponent. Man kan se de disjunkta komponenterna som öar och de M bågarna som flygförbindelser dem emellan.

Andersson et al. [1] ger ingen fingervisning om hur man transformerar en godtycklig graf till det format som krävs för att kunna utnyttja den algoritm de presenterar. Det går alltid att transformera en graf till deras format, men i värsta fall kommer den inte att ge en mindre tabell än den triviala. För att via algoritmen nå ett så bra resultat som möjligt vill man transformera den givna grafen på ett sätt som ger M^2 så nära $n \log n$ som möjligt. Enligt tidigare resultat (se kapitel 1.4) kommer det att vara omöjligt att samtidigt hålla tabellen liten och nå en bra approximation.

Ett underliggande problem när en graf skall delas upp i öar är huruvida två noder kan ligga på samma ö. Detta leder till frågan om det finns en väg mellan dessa noder som uppfyller spannervillkoret för varje par av noder längs vägen.

Algoritmen som ges i Andersson et al. [1] kommer hädanefter att benämnas *AGL*-algoritmen då den diskuteras.

1.2 Syfte

Mot den nyss presenterade bakgrunden försöker examensarbetet att svara på följande frågor: Är det möjligt att hitta ett hitta optimalt M-värde givet en fix stretch t? Finns det en effektiv algoritm som ger det optimala värdet? Om inte, vilka effektiva algoritmer kan tänkas ge bra approximationer? Går det att hitta en effektiv algoritm för det underliggande problemet, att finna en spannerväg mellan två noder? Om inte, finns det någon delmängd av grafer som tillåter en effektiv algoritm?

1.3 Metod

Frågorna behandlas en efter en och det underliggande problemet angrips först. Olika algoritmer som kan lösa detta presenteras och analyseras. Sedan komplexitetsbestäms problemet och slutligen undersöks en begränsad variant av det. Därefter angrips huvudproblemet och olika heuristiker testas och analyseras.

1.4 Tidigare resultat

En översikt över tidigare resultat när det gäller huvudproblemet ges i Thorup och Zwick [12]. Deras tabell återges i tabell 1.1 där n anger antalet noder, m antalet bågar, $\omega < 2.376$ är exponenten för matrismultiplikation och k är en heltalskonstant. Här är det framförallt resultaten med svarstid $O(1)$ som är av intresse då AGL algoritmen har konstant svarstid. Vidare levererar AGL algoritmen en $(1+\varepsilon)$-approximation så för att den skall vara effektivare än motsvarande metod i tabellen måste $M < n$.

1.5 Översikt

I kapitel 2 redovisas grundläggande definitioner och teori som sedan används framöver i arbetet. Här finns definitioner som rör spanners, flödesgrafer, asymptotisk notation, klasserna P och NP samt en redovisning av hur de välkända algoritmer som används i arbetet fungerar. Resten av kapitlen fokuserar antingen på huvudproblemet, att dela upp en graf i spanneröar, eller det underliggande problemet att

Stretch	Svarstid	Utrymme	Byggtid
1	$O(1)$	$O(n^2)$	$O(mn)$
	$O(m)$	$O(m)$	0
$1+\varepsilon$	$O(1)$	$O(n^2)$	$O(n^\omega)$
2	$O(1)$	$O(n^2)$	$O(m^{1/2}n^{3/2})$
7/3	$O(1)$	$O(n^2)$	$O(n^{7/3})$
3	$O(1)$	$O(n^2)$	$O(n^2)$
	$O(1)$	$O(m^{1/3}n + n^2/m^{1/3})$	$O(m^{2/3}n)$
	$O(1)$	$O(n^{3/2})$	$O(mn^{1/2})$
$2k-1$	$O(n^{1+1/k})$	$O(n^{1+1/k})$	$O(mn^{1+1/k})$
	$O(kn^{1/k})$	$O(kn^{1+1/k})$	$O(mn^{1/k})$
	$O(k)$	$O(kn^{1+1/k})$	$O(kmn^{1/k})$
$2k+\varepsilon$	$O(kn^{1/k})$	$O(kn^{1+1/k})$	$O(kmn^{1/k})$
$64k$	$O(kn^{1/k})$	$O(kn^{1+1/k})$	$O(kmn^{1/k})$

Tabell 1.1: Tidigare resultat hämtade från Thorup och Zwick [12].

hitta en spannerväg i grafen. Kapitel 3 ägnas helt åt det underliggande problemet. Här formuleras problemet SPANNERVÄG (se definition 3.1.1). I kapitel 3.1 redovisas tre olika algoritmer som löser det. Den bästa av dem når en komplexitet av storlek $O(2^{0.822n})$. Algoritmernas analys finns i appendix A.

I kapitel 3.2.2 presenteras kapitlets huvudsakliga resultat, att problemet SPANNERVÄG är *NP*-komplett för stretch $\sqrt{k}n^{1+1/k}$ (se sats 3.2.1). Detta bevisas genom en reduktion från ett annat *NP*-komplett problem, *Path with Forbidden Pairs*. Detta resultat föregås i kapitel 3.2.1 av en reduktion direkt från $3SAT$ som ger ett något svagare resultat, *NP*-kompletthet för stretch $\theta(n^2)$. Ett resultat som behövs för reduktionen från *Path with Forbidden Pairs* är att detta problem visas *NP*-komplett även för oriktade grafer och disjunkta nodpar. Detta görs i kapitel 3.2.2.

Kapitel 4 ägnas åt att studera spannervägar i heltalsgrafen som är en speciell typ av graf där alla noder ligger på heltalspositioner längs tallinjen. I kapitel 4.2 visas att SPANNERVÄG i heltalsgrafen är polynomiellt lösbart för stretch 1 (sats 4.2.1) och stretch n^2 (sats 4.2.2). Kapitel 4.3 redovisar ett första försök att lösa problemet via en algoritm som är specialanpassad för heltalsgrafen. Algoritmen presterar bättre än den generella med en komplexitet av storlek $O(2^{0.2n})$ för stretch 1.1. Detta kan jämföras med den generella algoritmens $O(2^{0.822n})$. Kapitlets huvudresultat finns i kapitel 4.4 och utgörs av en bättre algoritm som får komplexiteten $O(2^{c(\log n)^2})$ där c är en konstant som beror på stretchkonstanten.

Slutligen så undersöks huvudproblemet i kapitel 5. Här testas tolv olika heuristiker och deras prestationer jämförs. Testningen sker på ett stort antal randomi-

serade grafer indelade i olika testserier efter antalet noder och bågar. Heuristikerna är indelade i olika grupper. Det finns en girig heuristik, två som försöker optimera spanneröarnas diameter. En heuristik utnyttjar *single-link clustering* och det finns två som utnyttjar *complete-link clustering*. Majoriteten, fem, av heuristikerna använder sig av maxflödet i grafen för att dela upp den i spanneröar. Slutligen finns det en heuristik som använder en sveplinje för att separera problemnoder. De randomiserade graferna och heuristikerna presenteras i kapitel 5.3. Resultatet utgörs av ett antal tabeller som hittas i kapitel 5.4. Heuristikerna jämförs mot varandra i kapitel 5.5 och dessa jämförelser analyseras i kapitel 5.6.

I appendix A återfinns detaljerade redovisningar för flertalet komplexitetsbestämningar. Här finns också en bild på det verktyg som utvecklats för att undersöka huvudproblemet.

Kapitel 2

Definitioner och grundläggande Teori

Här redovisas begrepp och teori som används senare i arbetet.

2.1 Spanners

Begreppet spanner användes först i Peleg och Ullman [11]. I deras version är grafen en hyperkub och alla bågar har längden 1, så avståndet får högst vara t för att uppfylla kravet. Den definition som ges här är anpassad till grafer med godtyckliga båglängder. Graferna anses också vara oriktade, dvs. alla bågar kan passeras i båda riktningarna, om inget annat nämns.

Definition 2.1.1 **Avståndet** *mellan noderna u och v i grafen G betecknas $d_G(u, v)$. Detta är längden på den kortaste vägen i G som binder samman noderna u och v.*

Definition 2.1.2 *Givet en graf $G = (V, E)$, där V är mängden av noder i grafen och E är mängden av bågar. Låt $G' = (V, E')$ beteckna en delgraf till G där $E' \subseteq E$. G' är en* **generell t-spanner** *för G om det för varje par $(u, v) \in E'$ gäller att $d_{G'}(u, v) \leq t \cdot d_G(u, v)$.*

Definition 2.1.3 *Det* **euklidiska avståndet** *mellan två punkter i ett n-dimensionellt rum ges av $d_E(u, v) = \sqrt[n]{(v_1 - u_1)^2 + \cdots + (v_n - u_n)^2}$. I två dimensioner är alltså $d_E(u, v) = \sqrt{(v_1 - u_1)^2 + (v_2 - u_2)^2}$.*

I fortsättningen kommer det förenklade uttrycket $d(u, v)$ att användas för alla typer av avstånd. Det framgår då av sammanhanget om det avsedda avståndet är inom grafen eller det euklidiska.

Definition 2.1.4 *I en* **euklidisk graf** *är alla avstånd längs bågarna lika med det euklidiska avståndet mellan nodernas positioner i rummet.*

Definition 2.1.5 *En graf $G = (V, E)$ är* **fullständig** *om det för varje par av noder $(u, v) \in V$ finns en båge $(u, v) \in E$.*

Definition 2.1.6 *En graf $G = (V, E)$ är en* **euklidisk t-spanner** *för noderna i V om det för varje par av noder $(u, v) \in V$ där det finns en väg från u till v gäller att $d_G(u, v) \leq t \cdot d_E(u, v)$.*

Kopplingen mellan euklidiska och generella t-spanners görs genom att man för den euklidiska grafen låter G i definition 2.1.2 vara en fullständig euklidisk graf. Då sammanfaller G' ur definition 2.1.2 med G i definition 2.1.6.

Definition 2.1.7 *Konstanten t i t-spanner kallas* **stretchkonstant** *för grafen.*

Definition 2.1.8 *En nod som har minst en båge till en annan spannerö kallas en* **flygplats**.

I arbetet kommer uttrycket "uppfyller spannervillkoret" ibland att vara en egenskap för ett par noder. Det innebär att avståndet mellan dessa via bågar i den graf de ligger i inte får vara för stort i förhållande till det euklidiska avståndet emellan dem. Vilken stretch som krävs kommer att framgå av sammanhanget.

2.2 Flödesgrafer

I många sammanhang kan en graf användas för att representera ett flöde av något. Det kan vara vatten genom rör, eller bildelar som fraktas från en underleverantör till fabriken. Gemensamt är att vissa noder måste passeras på vägen och mellan dessa noder finns det en begränsande kapacitet som bestämmer hur mycket som kan flyttas. Det kanske går att frakta bildelar mellan två städer via tåg, medan andra rutter trafikeras av lastbilar eller båtar. Frakten har en distinkt startnod och en målnod. Det ger upphov till följande definitioner:

Definition 2.2.1 *En* **flödesgraf** *är en riktad graf (varje båge har en startnod och en målnod där ordningen påverkar riktningen) där varje båge har en positiv kapacitet. Det finns en speciell nod s, kallad* **källa**, *som flödet har sitt ursprung i och en speciell nod t, kallad* **avlopp**, *där flödet lämnar grafen. Vidare gäller för varje nod v i grafen att det finns en väg från s till t via v.*

Definition 2.2.2 *Ett* **flöde** *i flödesgrafen är en tilldelning av ett flöde, dvs. en använd del av en båges kapacitet, till varje båge i grafen. Flödet längs en båge får inte vara större än bågens kapacitet. Flödet in i en nod måste vara lika med flödet ut ur den, med undantag för s och t.*

En flödesgraf ger upphov till ett naturligt optimeringsproblem.

Definition 2.2.3 MAX-FLOW *problemet består i att givet en flödesgraf finna det största maximala flödet från källan till avloppet.*

2.3 Asymptotisk notation

I syfte att förenkla jämförelser mellan olika funktioner införs asymptotisk notation. När funktionerna beskriver exekveringstid för olika algoritmer saknar små detaljer ofta betydelse och man är intresserad av hur algoritmen klarar av stora probleminstanser. Detta betyder inte att det är ointressant med detaljerna i alla lägen och en del resultat kommer att presenteras i detalj. Det är i huvudsak då det finns asymptotiskt lika effektiva algoritmer som det kan behövas detaljerade studier för att komma fram till vilken som är bäst. Ibland finns det en brytpunkt i probleminstansens storlek där det börjar löna sig att byta algoritm.

Asymptotiskt betyder i det här sammanhanget "växer ungefär lika fort för stora värden". Det finns tre olika asymptotiska begrepp. Ett för att uttrycka att en funktion växer högst lika fort som en annan grupp av funktioner, ett för att uttrycka att en funktion växer minst lika snabbt som en annan grupp av funktioner, och slutligen ett för att uttrycka att en funktion växer lika snabbt som en annan grupp av funktioner.

Definition 2.3.1 *En funktion $t(n)$ är* **stort ordo** *$f(n)$ om det finns konstanter c och n_0 så att för alla $n \geq n_0$ gäller det att $t(n) \leq cf(n)$. Skrivsättet $t(n) = O(f(n))$ används. $O(f(n))$ ses som en övre gräns för t:s tillväxthastighet.*

Definition 2.3.2 *En funktion $t(n)$ är* **stort omega** *$f(n)$ om det finns konstanter d och n_0 så att för alla $n \geq n_0$ gäller det att $t(n) \geq df(n)$. Skrivsättet $t(n) = \Omega(f(n))$ används. $\Omega(f(n))$ ses som en nedre gräns för t:s tillväxthastighet.*

Definition 2.3.3 *En funktion $t(n)$ är* **stora theta** *$f(n)$ om det finns konstanter c, d och n_0 så att för alla $n \geq n_0$ gäller det att $df(n) \leq t(n) \leq cf(n)$. Skrivsättet $t(n) = \Theta(f(n))$ används. $\Theta(f(n))$ ses som ett mått på hur t växer med stora n.*

Egentligen gäller inte likhet i skrivsätten ovan då det är en hel klass av funktioner som står i höger led. I stället borde man skriva t.ex. $t(n) \in \Omega(f(n))$ men det är allmänt accepterat att man använder likhet och det bör inte ge några problem om man är uppmärksam. Det existerar notationer med prefixet "lilla" för ordo och omega, men de används inte i det här arbetet. Därför kommer "stora" att utelämnas och i arbetet kommer fortsättningsvis bara att stå ordo i de fall då en funktion t.ex. är $O(f(n))$.

För djupare teori och exempel samt ett antal räknelagar, se Kingston [9] eller Brassard och Bratley [3].

2.4 Klasserna *P* och *NP*

I det här arbetet används två komplexitetsklasser, P och NP. Det finns olika sätt att gå tillväga för att definiera dessa. Här återges en kort version av det som

står i Brassard och Bratley [3]. Det finns en annan mer komplicerad definition som använder sig av Turingmaskiner för att uppnå samma resultat. Den återges i Papadimitriou [10] och en del av resultaten ur denna bok används senare i arbetet.

Definition 2.4.1 *Ett* **beslutsproblem** *är ett problem som kan besvaras med sant/falskt eller ja/nej beroende på frågans typ. En korrekt algoritm för ett problem X accepterar de probleminstanser för vilka det korrekta svaret är ja och avvisar de vilkas svar är nej. X betecknar både problemet och mängde av de probleminstanser som ger sant eller ja, dvs. alla instanser som accepteras.*

Alla problem är inte av beslutstyp, men de flesta kan omformas till den här typen.

Definition 2.4.2 *Låt $p(n)$ vara ett polynom. En algoritm till ett problem X är* **effektiv** *om den tar en tid i storleksordningen $O(p(n))$ för att lösa ett problem av storleken n. Algoritmen kallas även polynomiell.*

Definition 2.4.3 HAMILTON CYKEL *problemet består i att hitta en väg (Hamiltonsk cykel) i en graf som startar i en nod s och sedan besöker alla andra noder en gång innan den återvänder till s igen.*

Exempel Problemet kan omvandlas till ett beslutsproblem HAMILTON CYKEL(B) som nu består i att avgöra om en graf innehåller en Hamilton cykel. De här två problemen är lika svåra i den mening att om det finns en effektiv algoritm för den ena så kan man finna en för den andra och tvärt om. Hittar man en cykel så vet man direkt att det finns en, så ett svar på beslutsversionen fås direkt av algoritmen för det första problemet. Har man en algoritm för beslutsproblemet kan man genom att köra denna upprepade gånger få fram en cykel på följande sätt:

1. Kör algoritmen på hela grafen. Om ingen cykel finns (algoritmen avvisar) så svara att ingen cykel finns. Om en cykel finns fortsätt till 2.

2. Ta bort en godtycklig båge och testa igen om en cykel finns. Finns det fortfarande en cykel så behövdes inte den bågen för att hitta cykeln och vi låter den vara borttagen. Om cykeln upphörde när bågen togs bort, sätt tillbaka bågen igen. Upprepa detta förfarande tills alla bågar testats.

3. Kvar finns nu de bågar som genom att tas bort orsakade att cykeln försvann. Dessa är $n + 1$ stycken och genom att starta i en nod och gå runt i grafen tills startnoden nås igen kan svaret presenteras.

Eftersom algoritmen tittar på alla bågar en gång tar den med asymptotiskt uttryck $O(|E|) = O(|V|^2)$ gånger den tid det tar att lösa beslutsproblemet. Här betecknar $|E|$ antalet bågar i grafen och $|V|$ antalet noder. Finns det en effektiv algoritm för beslutsproblemet så kommer det att finnas en för originalproblemet eftersom ett polynom gånger ett polynom fortfarande är ett polynom. Därför anses problemen ekvivalenta ur komplexitetssynpunkt, de tillhör samma komplexitetsklass.

2.4.1 Klassen *P*

Klassen P innehåller de problem som vi anser vara lätta. Här återfinns alla problem med effektiva algoritmer.

Definition 2.4.4 **P** *är klassen av alla beslutsproblem som kan lösas med en polynomiell algoritm.*

Exempel Reduktionen ovan som omformar svaret mellan HAMILTON CYKEL(B) och HAMILTON CYKEL tar $O(n^2)$ tid om grafen har n noder. Detta är ett exempel på en polynomiell algoritm.

Exempel Dijkstras algoritm som presenteras i kapitel 2.5 är ett annat exempel på en polynomiell algoritm. Även den tar en tid i storleksordningen $O(n^2)$ i värsta fall.

2.4.2 Klassen *NP*

Klassen *NP* är något svårare att beskriva. Den består informellt av alla problem som har polynomiellt verifierbara egenskaper. Formellt måste några nya begrepp införas.

Låt X vara ett godtyckligt problem och samtidigt beteckna mängden av alla probleminstanser som accepteras av en algoritm som löser X. Låt Q beteckna en mängd som vi kallar bevisrummet för X. Ett bevissystem för X är en mängd F av par $\langle x, q\rangle$. För varje $x \in X$ måste finnas ett $q \in Q$ så att $\langle x, q\rangle \in F$. Å andra sidan får inget sådant q finnas om $x \notin X$. Om ett q finns så att $\langle x, q\rangle \in F$ är alltså $x \in X$. Ett sådant q kallas ett bevis eller certifikat för att $x \in X$.

Exempel Om X är mängden av alla grafer med minst en Hamilton cykel så kan Q vara mängden av alla sekvenser av noder ur graferna och F bestå av alla par $\langle G, \sigma\rangle$ där σ är en Hamilton cykel i grafen G. Om en graf $g \in X$ finns alltså ett certifikat σ som bevisar detta.

Definition 2.4.5 *Klassen* **NP** *består av alla beslutsproblem X som har bevissystem $F \subseteq X \times Q$ sådana att det finns ett polynom $P(n)$ och en polynomiell algoritm A med följande egenskaper*

- *För alla $x \in X$ finns det ett $q \in Q$ så att $\langle x, q\rangle \in F$, och storleken på q är högst $p(n)$. Här är n storleken på x.*
- *För alla par $\langle x, q\rangle$, finns en effektiv algoritm A som kan verifiera om $\langle x, q\rangle \in F$ eller inte. Med andra ord, $F \in P$.*

Ett problem i *NP* har egenskapen att det finns certifikat för dess lösningar som är verifierbara i polynomiell tid. Om någon säger sig ha en lösning går det lätt att verifiera om det är sant. Däremot sägs ingenting om hur svårt det är att ta fram en lösning. Det finns kryptografiska metoder som utnyttjar detta faktum där meddelandet kan verifieras komma från rätt avsändare i polynomiell tid, medan en falsk avsändare måste lösa ett *NP*-svårt problem för att kunna imitera den verkliga avsändaren om han vill förfalska innehållet.

Exempel Eftersom ett problem i P kan lösas i polynomiell tid och det får ta polynomiell tid att verifiera ett certifikat i *NP* kan man verifiera om ett certifikat är sant genom att lösa problemet. Bevisrummet blir då trivialt.
I detalj:
Låt X vara ett godtyckligt beslutsproblem i P och $Q = \{0\}$ bevisrummet. Då kan bevissystemet $F = \{\langle x, 0\rangle | x \in X\}$ användas. 0 är då certifikatet för alla $x \in X$. Givet ett x måste alltså bedömas i polynomiell tid om $\langle x, 0\rangle \in F$, men detta sker då $x \in X$ och eftersom $X \in P$ går det att verifiera om så är fallet i polynomiell tid och alltså uppfylls punkt två i definitionen av *NP* (som algoritm A kan den algoritm användas som gör att $X \in P$). Punkt 1 uppfylls genom att storleken av q är 1. Det innebär att $P \subseteq NP$. Det förblir fortfarande datavetenskapens största gåta huruvida $P = NP$.

2.4.3 Reduktioner

Det finns många sätt att reducera ett problem till ett annat. I arbetet kommer Turingreducerbarhet att användas. Till de här reduktionerna används så kallade orakel.

Definition 2.4.6 *Ett* **orakel** *för problemet X är en algoritm som kan lösa problemet i konstant tid. Användning av orakel är en metod som kan användas vid t.ex. reduktioner.*

Definition 2.4.7 *Låt A och B vara två problem. A är* **polynomiellt turingreducerbart** *till B om det finns en algoritm för att lösa A i polynomiell tid givet ett orakel för B. Detta kan skrivas $A \leq B$. Om även det motsatta förhållandet gäller, $A \geq B$ kallar man problemen Turingekvivalenta och skriver $A \equiv B$.*

Exempel Låt A och B vara två problem. Om $A \leq B$ och B kan lösas i polynomiell tid så kan även A det. Eftersom $A \leq B$ finns en algoritm som givet en instans till A löser denna i en polynomiell tid givet att ett orakel för B finns. Men det finns bara en polynomiell algoritm för B givet. Hur påverkar detta möjligheten att lösa A? På grund av att reduktionen existerar finns en algoritm som tar tiden $p(n)$ för att lösa A givet ett orakel för B. Detta orakel kan högst frågas $p(n)$ gånger. Dessutom kan storleken på de instanser som oraklet frågas på inte vara större än tiden som spenderas i algoritmen dvs. $p(n)$. Om det finns en algoritm för att lösa B som tar

$O(t(n))$ tid så kommer en algoritm för A att ta en tid i $O(p(n) \cdot t(p(n)))$. Detta är ett polynom gånger en sammansättning av två polynom och är alltså ett polynom. Därmed finns det även en polynomiell algoritm för A.

2.4.4 *NP*-kompletta problem

Det finns många *NP*-kompletta problem. I det här arbetet används två olika sådana för att visa att ett tredje även det är *NP*-komplett.

Definition 2.4.8 *Ett beslutsproblem $X \in NP$ är* **NP-komplett** *om det för varje problem $Y \in NP$ gäller att $Y \leq X$.*

Definition 2.4.9 *En disjunktion av literaler kallas för en* **klausul**.

Definition 2.4.10 *Ett booleskt uttryck ϕ är i* **konjunktiv normalform**, **CNF** *om det kan skrivas $\phi = (\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$, där alla ϕ_i är klausuler.*

Det bevisas av Papadimitriou [10] att alla booleska uttryck kan skrivas om till ett ekvivalent uttryck i CNF form.

Definition 2.4.11 Satisfierbarhetsproblemet, **SAT** *definieras som följande beslutsproblem: Givet ett booleskt uttryck i konjunktiv normalform (CNF). Finns det en tilldelning till variablerna så att uttrycket blir sant? En vanligt förekommande variant av SAT är* **3SAT** *där det krävs att alla klausuler i CNF formen består av tre literaler (det är tillåtet att samma literal förekommer flera gånger i en klausul). En sista variant som förekommer är här döpt till* **3SAT3V2L**. *Detta är en variant av 3SAT där varje variabel bara får förekomma tre gånger i uttrycket och varje literal högst två gånger.*

Det kan vara på sin plats att förklara skillnaden på variabel och literal för att 3SAT3V2L skall få en tydlig mening. En variabel är ett variabelt värde, t.ex. en inparameter x_i, medan en literal är ett uttryck av formen x_i eller $\neg x_i$ där $\neg$ står för logisk negation. Att variabeln får förekomma tre gånger men varje literal bara två gånger innebär att varken x_i eller $\neg x_i$ får finnas med i uttrycket tre gånger.

Beviset för att SAT är *NP*-komplett är känt som Cooks sats. Det återges i Papadimitriou [10] och ligger utanför ramen för det här arbetet. I kort så visas att problemet CIRCUIT SAT är *NP*-komplett och sedan reduceras det till SAT. I reduktionen används bara uttryck av 3SAT typ vilket direkt ger att även 3SAT är *NP*-komplett. För att bevisa att 3SAT3V2L är *NP*-komplett krävs lite mer trixande och detaljerna finns i Papadimitriou [10].

För att visa att ett problem X är *NP*-komplett väljs ett passande redan känt *NP*-komplett problem och sedan reduceras detta till X. I det här arbetet görs bland annat reduktioner från 3SAT och 3SAT3V2L till problem som skall visas vara *NP*-kompletta.

Definition 2.4.12 *Ett beslutsproblem X är* **NP-svårt** *om det för varje problem $Y \in NP$ gäller att $Y \leq X$.*

Observera att det inte krävs att $X \in NP$. Begreppet NP-svårt är lite svagare eftersom det inte säger något närmre om X:s komplexitet. Det används då man inte är intresserad av precis hur svårt X är utan är nöjd med att konstatera att det minst är lika svårt som NP-kompletta problem.

2.5 Algoritmer

Tre av de algoritmer som används i arbetet förklaras här. Dijkstras algoritm hittar givet en graf och en startnod kortaste vägen från startnoden till alla andra noder i grafen. Floyds algoritm bestämmer kortaste vägen mellan alla noder i en graf. Edmonds-Karps algoritm beräknar givet en flödesgraf det maximala flödet mellan källa och avlopp.

2.5.1 Dijkstras algoritm

I Dijkstras algoritm används en prioritetskö Q för att hålla koll på alla noder som för tillfället går att besöka. De ligger i kön i ordning så att den nod som har kortast avstånd till startnoden (s) via redan besökta noder ligger först och kommer att tas ut först. Algoritmen arbetar så här:

1. s läggs in i Q.
2. Ta ut minsta elementet, e, ur Q.
3. Alla noder som går att nå från e läggs in i Q om de inte redan ligger där. Deras nycklar blir avståndet till e plus avståndet från e till de respektive noderna. Om de redan ligger där jämförs deras avstånd med det som nu går att få via e. Om det nya avståndet är mindre uppdateras det i kön.
4. Upprepa 2-3 tills Q är tom. Alla noder har som avstånd till s det värde de hade då de togs ut ur kön.

Ett korrekthetsbevis finns i Kingston [9]. Tidskomplexiteten bestäms också där till $O(n \log n + m)$ där m är antalet bågar i grafen. Detta ger i värsta fall komplexiteten $O(n^2)$. Det bör nämnas att algoritmen fungerar lika bra på oriktade som riktade grafer.

2.5.2 Floyds algoritm

Floyds algoritm arbetar hela tiden på en matris av storlek n^2 där varje cell $m_{i,j}$ är ett avstånd mellan noderna i och j. Algoritmen itererar över matrisen n gånger vilket ger en tidskomplexitet på $O(n^3)$. Matrisen D initieras till D_0 där alla celler

innehåller det direkta avstånden mellan två noder. Nodpar som inte har en båge mellan får värdet ∞ i sin cell.

I varje steg ökas index på D. Indexet symboliserar de noder som får användas i en väg mellan två noder. Cell $m_{i,j}$ i matris D_k innehåller kortaste avståndet mellan noderna i och j om noderna $1, 2, \ldots, k$ får användas. Slutresultatet finns i D_n.

I varje iteration kontrolleras för varje nodpar (i, j) om det värde som kan fås via k är mindre än det tidigare bästa värdet ($m_{i,k} + m_{k,j}$ jämförs mot $m_{i,j}$ som är det tidigare värdet) i så fall uppdateras cellen.

Implementationen av Floyds algoritm består av en for loop med tre nivåer där endast jämförelsen utförs innerst. Därför har Floyds algoritm troligen en jämförelsevis liten konstant i ordo-uttrycket. Mer information finns i Cormen et al. [5].

2.5.3 Edmonds-Karps algoritm

Givet en flödesgraf $G = (V, E)$, där V är mängden noder och E mängden bågar, beräknar Edmonds-Karps algoritm maxflödet inom en tid av storleksordningen $O(VE^2)$.

Algoritmen arbetar i sin helhet på det som kallas residualgrafen, R, till G. Detta är en graf som hela tiden visar möjliga ändringar i flödet. Till att börja med ser den ut precis som G. Men så fort ett flöde f sker längs en båge i G kommer R att påverkas på två sätt. Dels kommer bågens kapacitet i R att minskas med f, blir den 0 tas bågen bort. Dels kommer en båge åt andra hållet att skapas om den inte redan fanns och dess kapacitet ökas med f. Vad residualgrafen visar i det här läget är att antingen kan man trycka igenom mer flöde längs bågen, om den finns kvar, eller så kan man ångra sig och reversera flödet då det nu finns en båge i motsatt riktning med kapacitet minst f.

Edmonds-Karps algoritm söker i varje iteration en utökning av flödet, från att ha börjat på noll. Dijkstras algoritm används för att hitta en väg genom residualgrafen. Den hittar i varje iteration den kortaste vägen från källan till avloppet. Med kortaste menas här minst antal bågar längs vägen. Längs den här vägen utökas flödet med kapaciteten av den båge som har lägst kapacitet. Detta ger en utökning i varje steg och till slut nås maxflödet. Bevis för korrekthet och komplexitetsbestämning finns i Cormen et al. [5].

Kapitel 3

Spannervägar

Ett problem som är relaterat till att dela upp en graf i spanneröar är huruvida två noder kan ingå i en t-spanner med varandra. Detta kan också formuleras så här: Givet en graf U och två noder, a och b, finns det en delgraf till U som innehåller a och b samt är en t-spanner? Den minsta sådana grafen kommer att bestå endast av en väg mellan a och b. Denna väg kallas för en spannerväg. Längs en spannerväg gäller att varje nod uppfyller spannervillkoret med avseende på alla andra noder.

Intresset bakom spannervägar beror på att det är väldigt svårt att dela upp en graf i spanneröar. Det är då naturligt att undersöka de underliggande egenskaper som inverkar på detta problem. Ett enklare problem torde vara att bedöma om två noder ens kan ligga på samma ö. Detta ger ingen lösning till huvudproblemet men komplexiteten bakom kan ge en insikt i dess svårighet.

3.1 Algoritmer

Nu definieras problemet SPANNERVÄG som kommer att undersökas i kapitel 3 och 4.

Definition 3.1.1 SPANNERVÄG *problemet består i att givet en graf G, två speciella noder a och b och en spannerkonstant t, avgöra om det finns en spannerväg mellan a och b med stretch högst t.*

Det enklaste sättet att bedöma om två noder, a och b, ligger längs en spannerväg är att testa alla möjliga vägar mellan dem och se om någon fungerar. En väg består som mest av n noder som måste testas mot alla andra vilket ger $O(n^2)$ test. Antalet vägar är $O((n-2)!)$, se appendix A.1. Sammantaget får en sådan här algoritm tidskomplexiteten $O(n!)$ och använder $O(n)$ utrymme.

Detta är en extremt dålig tidskomplexitet. Det går att få en lite bättre med följande nästan lika simpla algoritm, delgrafalgoritmen.

Välj ut ett antal av de $n-2$ noderna förutom a och b. Bilda en delgraf av dessa med a och b samt de bågar som gick mellan noderna i ursprungsgrafen. Applicera

sedan Dijkstras algoritm 2.5.1 på denna graf och bestäm kortaste vägen mellan a och b. Är den här vägen en t-spanner så svara ja, annars fortsätt leta. Om det finns en spannerväg mellan a och b så finns det en kortaste spannerväg mellan dem. De noder som ingår i vägen kommer att väljas av algoritmen i något steg och den kortaste spannervägen upptäcks av Dijkstras algoritm. Antalet nodmängder som kan väljas är 2^{n-2} eftersom varje nod antingen kan vara vald eller inte. Dijkstras algoritm är i värsta fall $O(n^2)$. Detta ger komplexiteten $O(n^2 2^n)$, se appendix A.2 för detaljerna (detta är för övrigt den tid som en uttömmande sökningsalgoritm för SAT använder, se Papadimitriou [10]). Delgrafalgoritmen använder $O(n)$ utrymme. En nackdel med den här algoritmen är att om det visar sig att två noder inte kan ligga längs kortaste vägen mellan a och b så kommer ändå alla 2^{n-4} övriga kombinationer med dessa två noder att genomsökas. Med hjälp av en sista algoritm, splittringsalgoritmen kan detta undvikas.

Kör först Dijkstras algoritm och hitta kortaste vägen mellan a och b. Om detta är en spannerväg avslutas algoritmen med positivt svar. Om det finns två noder längs vägen som inte är kompatibla så kan ingen spannerväg mellan a och b passera båda dessa. Detta beror på att Dijkstras algoritm hittar den kortaste vägen och noderna har för lång väg mellan sig längs denna. Det kan då inte finnas en annan väg mellan dem som är kortare och uppfyller spannervillkoret. Alltså kan algoritmen ta bort den ena noden och fortsätta rekursivt på den kvarvarande grafen. Sedan när det spåret är testat sätter den tillbaka noden och tar bort den andra noden. Den resterande grafen testas igen rekursivt. Detta leder till att algoritmen till slut hittar en lösning om det finns en. Det visar sig att algoritmen som kan implementeras i två varianter får tidskomplexiteterna $O(2^n)$ resp. $O(2^{0.822n})$. Beräkningarna redovisas i appendix A.3.

Den första varianten använder djupet-först sökning och tar upp n gånger grafens utrymme då det blir n grafer att hålla i minnet under beräkningen. Totalt blir det $O(n^3)$ utrymme.

Den andra varianten, tabellvarianten, använder $\binom{n-2}{n/5.618}$, se appendix A.3. Detta kan uppskattas med $O(2^{0.644n})$ vilket är för mycket för att i praktiken använda den. Att algoritmerna blev så dåliga får sin förklaring i nästa avsnitt.

Som jämförelse kan nämnas att den bästa algoritmen för 3SAT är av komplexitet $O(2^{0.559n})$, se Brueggemann och Kern [4].

3.2 Komplexitetsbestämning

Komplexitetsbestämningen sker på två sätt med olika resultat på stretchkrav. Först visas en direkt reduktion från 3SAT varianten i kapitel 2.4.4, 3SAT3V2L. Det går dock att få ett skarpare resultat varför en andra reduktion utförs via mellanproblemet *Path with Forbidden Pairs*.

Sats 3.2.1 *Problemet SPANNERVÄG är* NP*-komplett för stretch* $\sqrt{k}n^{1+1/k}$*, där* k *är ett heltal och* n *storleken på problemet angett i antalet noder.*

Bevis Beviset utgörs av innehållet i kapitel 3.2.1 och 3.2.2.

3.2.1 Reduktion från 3SAT

Observera att stretchkonstanten i sats 3.2.1 inte är konstant utan beror på problemets storlek. Referenser till 3SAT i det här underkapitlet är till varianten 3SAT3V2L. Ett exempel illustrerar hur reduktionen går till.

Exempel Givet följande instans till 3SAT varianten:

$$(a \vee b \vee \bar{c}) \wedge (\bar{d} \vee e \vee f) \wedge (c \vee b \vee d) \wedge (a \vee g \vee h) \wedge (\bar{e} \vee \bar{b} \vee \bar{f}) \wedge (\bar{a} \vee \bar{c} \vee d)$$

Skriv om uttrycket på höjden med en klausul per rad och inför radnummer. Tabell 3.1 visar resultatet.

$$\begin{array}{l} 1\ a\ b\ \bar{c} \\ 2\ \bar{d}\ e\ f \\ 3\ c\ b\ d \\ 4\ a\ g\ h \\ 5\ \bar{e}\ \bar{b}\ \bar{f} \\ 6\ \bar{a}\ \bar{c}\ d \end{array}$$

Tabell 3.1: Exempel på en 3SAT probleminstans.

Idén är att bygga en graf där en väg igenom innebär att det finns en tilldelning som satisfierar uttrycket medan ingen väg innebär att ingen tilldelning satisfierar uttrycket. Literaler som tillhör samma variabel måste passeras så att inte a och $\bar{a}$ båda ingår i vägen. Detta hjälper spanneregenskapen till med. En konflikt i spanneregenskapen kommer att likställas med en konflikt i sanningsvärden. Varje variabel har bara ett sanningsvärde och detta åstadkoms genom att upprätthålla spanneregenskapen.

De literaler som tillhör samma variabel paras ihop två och två: $1a$,$6\bar{a}$; $1b$,$5\bar{b}$; $1\bar{c}$,$3c$; $2\bar{d}$,$3d$; $2e$,$5\bar{e}$; $2f$,$5\bar{f}$. Genom att placera dessa noder tillräckligt nära varandra garanterar man att båda inte kan ingå i en spannerväg.

Följande variabler saknar par och är därför för närvarande fria: $3b$; $4a$; $4g$; $4h$; $6\bar{c}$; $6d$

Literalerna 4g och 4h, som bara förekommer i en form (här ickekonjugerade) kan ignoreras då grafen byggs eftersom de inte kan hamna i konflikt med andra noder/tilldelningar. Egentligen skulle dessa rader kunna tas bort helt eftersom de alltid kan passeras via de obundna noderna, men i exemplet har de behållits.

I syfte att balansera då en variabel förekommer 3 gånger i ett uttryck dupliceras vissa rader. Detta ger nya noder som kan paras ihop med de tidigare men förändrar inte 3SAT uttryckets värde. Tabell 3.2 visar hur nya rader lagts till. De nya raderna

$1\ a\ b\ \bar{c}$
$2\ \bar{d}\ e\ f$
$\mathbf{1}\ \bar{d}\ e\ f$
$3\ c\ b\ d$
$\mathbf{2}\ c\ b\ d$
$4\ a\ g\ h$
$5\ \bar{e}\ \bar{b}\ \bar{f}$
$\mathbf{3}\ \bar{e}\ \bar{b}\ \bar{f}$
$6\ \bar{a}\ \bar{c}\ d$
$\mathbf{4}\ \bar{a}\ \bar{c}\ d$

Tabell 3.2: Nya rader har lagts till.

$3b$,$\mathbf{3}\bar{b}$
$4a$,$\mathbf{4}\bar{a}$
$\mathbf{6}\bar{c}$,$2c$
$6d$,$\mathbf{1}\bar{d}$

Tabell 3.3: Nybildade par.

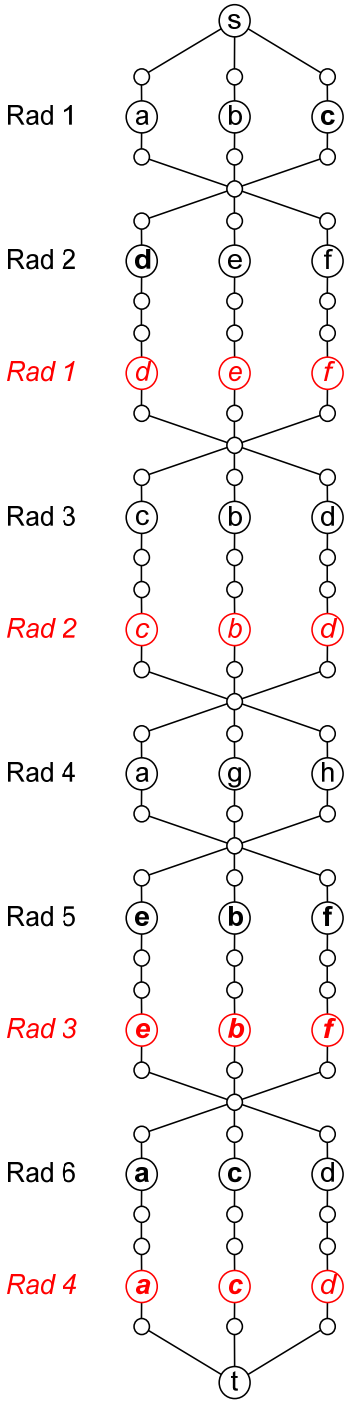
har lagts till under den rad de duplicerar och har fått radnummer från ett men med fetstil. De nya par som bildas syns i tabell 3.3

De nyintroducerade literaler som inte paras ihop med andra literaler kommer bara att passeras rakt förbi så att om t.ex. $6d$ passeras kommer även $\mathbf{4}d$ att göra det, men det är $6d$ som garanterar att inte $\mathbf{1}\bar{d}$ eller $2\bar{d}$ passerats.

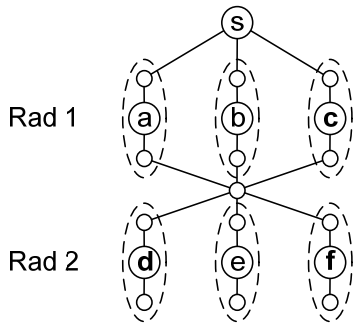
Figur 3.1 visar hur grafen ser ut i första konstruktionsfasen. I figuren har de nya raderna kursiverats och negerade literaler återges med fetstil. Ett antal "småNoder" har infogats. De fungerar som kopplingsnoder och är till för att underlätta visualiseringen av tillbakabågar som snart kommer att införas.

De underlättar också genom att rikta vägen mellan raderna. Kopplingspunkter mellan en övre rad och en undre kursiv rad är kopplade rakt. Detta leder till att om t.ex. $5\bar{e}$ genomlöps av vägen kommer även $\mathbf{3}\bar{e}$ att passeras. Mellan övriga rader finns en central kopplingspunkt som förhindrar att en väg går tillbaka utan att den innehåller en cykel. På så vis fås en riktning från s till t. Man kan se kopplingspunkten före en nod, noden och kopplingspunkten efter som en stornod. Figur 3.2 visar hur dessa tänkta stornoder ser ut.

Nu introduceras bakåtbågar. Detta innebär att vissa noder flyttas bakåt. Det är bara bokstavsnoderna som flyttas, så stornoderna behåller sin plats. Då båda sorters literaler till en variabel ingår i 3SAT uttrycket kan de förekomma på fyra olika sätt:



Figur 3.1: Tidigt skede i reduktionen.



Figur 3.2: Exempel på stornoder.

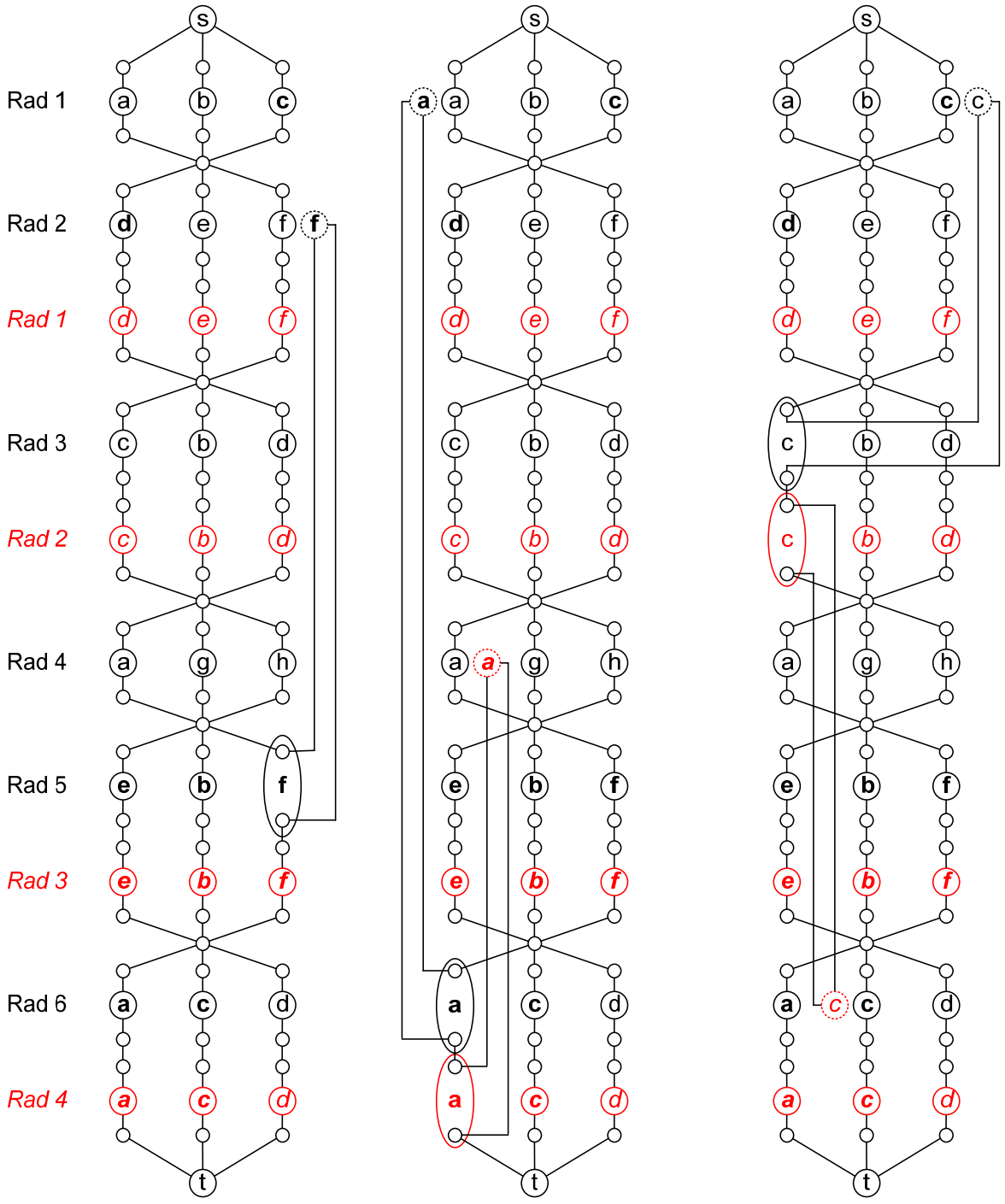
- båda literaler förekommer en gång, t.ex. $2f$,$5\bar{f}$
- ena literalen finns under två av andra typen, t.ex. $1a$,$4a$,$6\bar{a}$
- ena literalen finns över två av andra typen, t.ex. $2\bar{d}$,$3d$,$6d$
- ena literalen finns mellan två literaler av andra typen, t.ex. $1\bar{c}$,$3c$,$6\bar{c}$

Andra förekomster kan omvandlas till dessa genom att negera alla literaler från samma variabel. Detta påverkar inte uttryckets värde ur 3SAT synpunkt. En lösning med a sann blir då en lösning med $\bar{a}$ sann. Fall tre hanteras precis som fall två fast hamnar omvänt i grafen, de kan därför betraktas som ett fall. Figur 3.3 visar hur de tre olika fallen hanteras. Bågarna går i konstruktionen kortaste vägen mellan noderna, men i syfte att åskådliggöra kopplingarna i figuren bättre har bågarna ritats på ett enklare sätt.

Genom att placera noderna vid bakåtbågarna tillräckligt nära sina negationer (som de hamnar intill) kan man förhindra att båda ingår i samma spannerväg. Noden $2f$ kan alltså inte ligga på samma spannerväg som stornoden $5\bar{f}$. Detta leder till att variabeln f tilldelas ett sanningsvärde beroende på vilken av noderna som passeras. $2f$ leder till att f sätts till sant. $5\bar{f}$ leder till att f sätts till falsk. På samma sätt finns det ingen tillåten väg i mittengrafen som kan innehålla ett a och ett $\bar{a}$. Detta avslutar exemplet.

För att visa att ett problem är *NP*-komplett krävs att det ligger i *NP*. Här följer nu den definition som används i Papadimitriou [10]. Ett problem ligger i *NP* om en icke-deterministisk turingmaskin kan lösa det i polynomiell tid. Detta stämmer för problemet SPANNERVÄG. En ickedeterministisk turingmaskin kan generera alla vägar och testa dem i polynomiell tid. Finns en väg svarar den ja, annars nej. Det är alltså konstaterat att SPANNERVÄG $\in$ *NP*.

Det tar polynomiell tid att konstruera grafen ovan givet ett uttryck i 3SAT. Det behövs en ny rad i värsta fall för var tredje literal. Om 3SAT har ett uttryck med m klausuler och $n = 3m$ literaler så kommer den genererade grafen att ha



Figur 3.3: Olika typer av bakåtbågar.

högst m extra rader. Detta beror på att $n/3 = m$ literaler kan behöva balanseras och i värsta fall balanserar varje ny rad bara en literal. Den konstruerade grafen kommer totalt ha mindre än $7(2m-1)+6m+8$ noder. Första termen är antalet kopplingsnoder (högst 7 i varje internt lager), den andra är antalet riktiga noder (3 per rad och högst 2m rader) och den sista är noderna kring s och t. Sammanlagt finns $O(m)$ noder.

En reduktion från 3SAT till SPANNERVÄG är alltså att konstruera grafen enligt tidigare instruktioner. Nu verifieras att en SPANNERVÄG finns genom grafen om och endast om 3SAT uttrycket är satisfierbart.

Om det finns en spannerväg från s till t så finns det ett sätt att passera varje rad utan att hamna i konflikter. Att passera en rad innebär en tilldelning till en variabel så det finns alltså en tilldelning utan konflikter som gör att minst en literal i varje klausul blir sann och alltså finns en tilldelning till 3SAT uttrycket som gör det sant.

Å andra sidan, om det finns en tilldelning som satisfierar 3SAT uttrycket så gör den alla klausuler sanna. Det innebär att det finns minst en sann literal i varje klausul. Om man plockar bort alla noder i grafen som motsvarar literaler som är falska i tilldelningen fås en mindre graf. I den här grafen finns minst en nod kvar på varje rad, nämligen den eller de som gjorde klausulen sann. Vidare finns inga konflikter kvar, en nod ur varje par som konstruerats i grafen har försvunnit då variabeln har ett distinkt värde. Eftersom alla rader kan passeras utan konflikter finns en spannerväg igenom denna mindre graf. Men om det finns en väg i en mindre graf försvinner inte den om man återställer grafen till ursprungsgrafen som alltså också har en spannerväg och SPANNERVÄG returnerar sant.

Det finns alltså en spannerväg precis då motsvarande 3SAT uttryck är lösbart. Detta innebär att SPANNERVÄG är *NP*-komplett då det ligger i NP och på polynomiell tid reducerar 3SAT. Detta avslutar beviset.

Kvar återstår att undersöka vilken stretch som behövs för att tillåta ovanstående konstruktion. Det är förhållandet mellan kortaste avståndet mellan två tillåtna noder och det längsta mellan dessa som sätter gränsen för stretchkonstanten.

Avståndet mellan noder och bakåtflyttade noder får inte vara för stort, samtidigt skall alla andra vägar vara tillåtna. Om kopplingsnoderna placeras så att längsta vägen mellan två rader är ε och den kortaste vägen är η kan en längsta båge uppskattas. Det finns $2m$ rader som bågar kan gå fram och tillbaka mellan. Bågar till s och t kan utelämnas eftersom de är riktade och aldrig passeras igen. För att även ha ett mått på kopplingsnoderna vid s och t kan de också sättas så längsta avståndet blir ε. Den längsta båge som kan förekomma är en båge mellan sista raden och första raden. Denna blir $\leq 2m\varepsilon$. I en spannerväg genom grafen kommer bara en nod per rad att ingå. Om denna nod är en bakåt nod så kommer bakåtbågens längd att ingå två gånger då nodens rad passeras. Det innebär att maxlängden på vägen genom en stornod blir $2 \cdot 2\varepsilon m = 4\varepsilon m$. Den längsta vägen genom grafen fås med så många bakåtbågar som möjligt. För att få en grov uppskattning antas

att varje rad nås baklänges. Vägen kan uppskattas grovt med $2m \cdot 4\varepsilon m = 8\varepsilon m^2$, eftersom det är $2m$ rader som maximalt behöver sträckan $4\varepsilon m$ för att passeras. Detta är ett ganska grovt mått då sats 4.2.3 gör det troligt att avståndet i snitt är hälften av det maximala. Ur vägen har här termen $2\varepsilon m$ strukits. Detta är det faktiska avståndet rakt igenom grafen mellan alla kopplingsnoderna, men är så litet i förhållande att det äts upp av den grova uppskattningen. Att den utelämnas gör framställningen lite lättare att läsa och påverkar inte ordo-notationen.

Om avstånden mellan noderna inom en rad sätts till 1 så kan spannerkonstanten som krävs för att få lov att ta en bakåtbåge beräknas. Placeringen av bakåtnodens avstånd från sin negation kan också beräknas.

Avståndet från bakåtnoden till den närmsta granne som kan passeras av vägen är ungefär 1. Spannerkonstanten måste därför vara större än $8\varepsilon m^2/1 = 8\varepsilon m^2$ för att tillåta denna passage. Positioneringen av bakåtnoden måste vara så nära att passage av dess negation inte fungerar. Avståndet via grafen från negationen till bakåtnoden är minst 2η eftersom vägen efter att ha passerat grannen måste ner i bakåtnodens stornod och vända. Denna väg ger en stretch på $2\eta/d$ där d är det euklidiska avståndet mellan noderna. Om denna kvot skall bryta mot spannervillkoret måste $2\eta/d > 8\varepsilon m^2$ vilket ger $d < \eta/(4\varepsilon m^2)$. Observera att ε inte kan väljas för litet eftersom avståndet mellan raderna är minst 2 på grund av det interna avståndet mellan noderna på samma rad. Förhållandet mellan ε och η är på grund av geometrin $\varepsilon = \sqrt{\eta^2 + 4}$.

Resultatet av reduktionen visar att problemet SPANNERVÄG är *NP*-komplett för spannerkonstanter i storleksordningen $\Theta(n^2)$ där n anger antalet noder i grafen. En skärpning av spannerkonstanten görs strax i kapitel 3.2.2.

En kommentar angående spannervägen är på sin plats. Eftersom grafen är riktad finns det inget som hindrar en väg från att innehålla samma noder flera gånger. Det går dock att skärpa kravet på spannervägen så att inga cykler tillåts utan att det får någon påverkan på tidigare diskussion. På så vis kan cykler uteslutas.

3.2.2 Reduktion från Väg med förbjudna par

En bättre spannerkonstant kan nås om hjälp hämtas från ett annat redan känt *NP*-komplett problem. Problemet "väg med förbjudna par" (på engelska Path with Forbidden Pairs) erbjuder den hjälp som behövs.

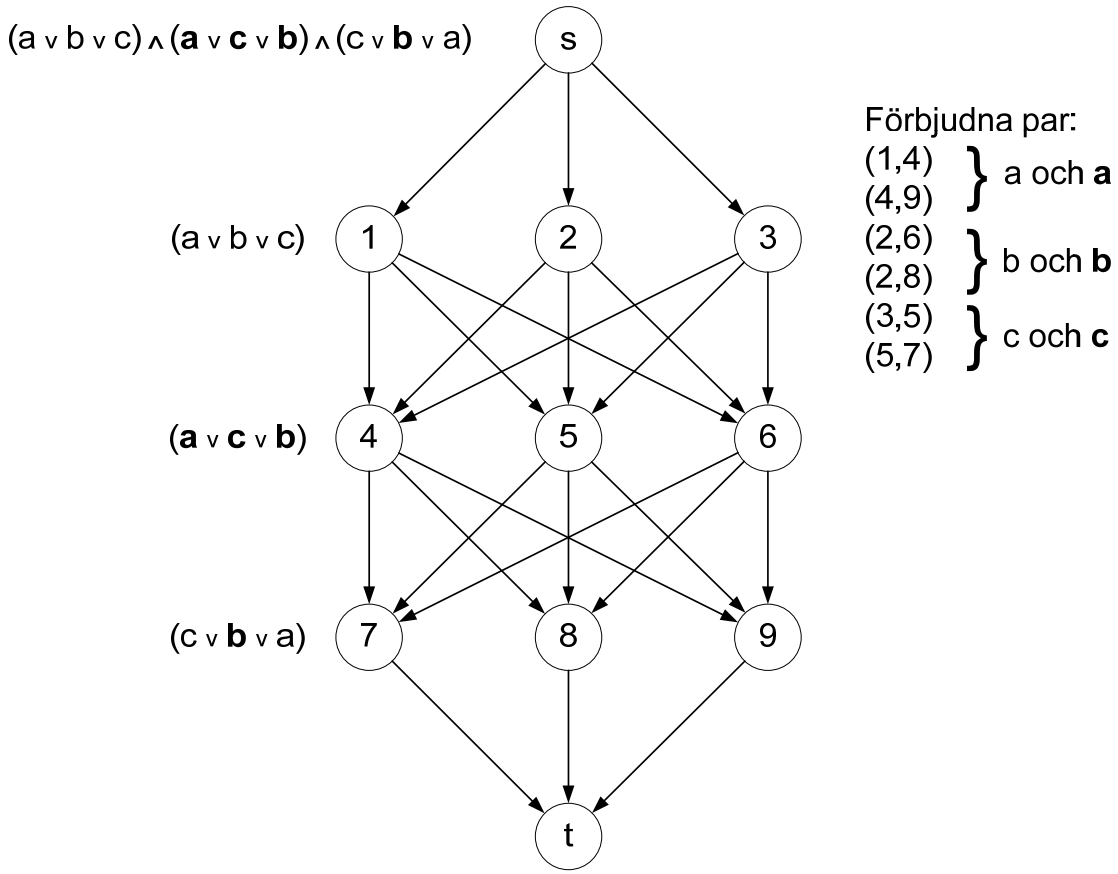
Definition 3.2.1 *Givet en riktad graf $G = (V, A)$ med n noder och två speciella noder, s och t, samt en lista $C = \{(a_1, b_1), \ldots, (a_m, b_m)\}$ med nodpar. Problemet* **PwFP** *består i att avgöra om det finns en väg i G från s till t sådan att den högst innehåller en av de två noderna ur paren i C.*

Ett sätt att testa mjukvara är att dela in den i block som genomlöps sekventiellt. Dessa block är sedan kopplade till varandra genom hopp i programflödet. Vissa block kan bara exekveras om andra block inte exekveras under körningen, de är parvis uteslutande. If-satser är ett exempel där bara det ena blocket exekveras.

Det är dock så att resultatet av ett test i en if-sats kan påverka andra if-satser på andra ställen i programmet. Givet en graf över blocken och par som visar vilka som inte kan exekveras i samma testrunda så söks ett antal testvägar genom grafen så att alla block testas någon gång. Målet är att hitta en väg från en startnod s till en målnod t som innehåller högst en av noderna i varje par. Problemet studerades och bevisades *NP*-komplett av Gabow et al. [6]. Det togs upp i en lista över kända *NP*-kompletta problem i Garey och Johnson [7] där det utvidgades till att även gälla oriktade grafer och till att gälla då alla nodpar är disjunkta. Om detta gällde samtidigt framgick inte och jag utreder här hur det ligger till.

Beviset i Gabow et al. [6] reducerar allmänna 3SAT till PwFP genom att konstruera en graf och ett antal förbjudna par. Om en väg hittas genom grafen finns en lösning till 3SAT problemet. Referenser till 3SAT i det här underkapitlet är till den allmänna versionen, se definition 2.4.11.

Exempel Så här går reduktionen till för en liten exempelgraf.



Figur 3.4: Gabows reduktion.

Givet ett uttryck i 3SAT. Konstruera en graf enligt figur 3.4 där varje nod är

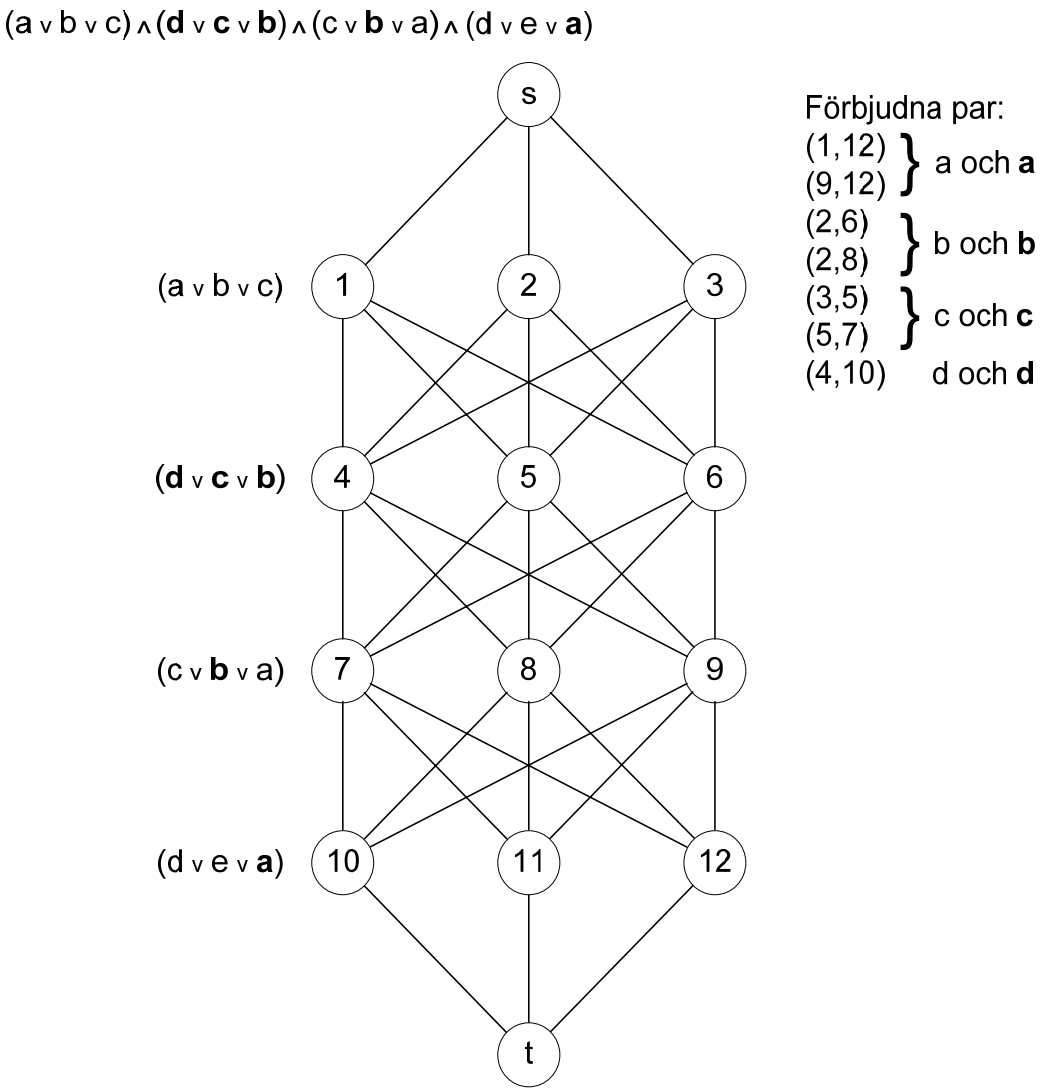
kopplad till de tre noderna i lagret under. I figuren är negerade literaler markerade med fetstil. De speciella noderna s och t placeras som i bilden. Varje nod symboliserar då en literal och de noder vars literaler är komplement förbjuds att båda ingå i vägen genom grafen. Detta säkerställer att inte både literalen och dess negation ingår samtidigt.

Om en väg mellan s och t hittas kan den översättas till en tilldelning av sanningsvärden åt literalerna som satisfierar 3SAT uttrycket. En av noderna i varje lager kommer att passeras vilket innebär att en av literalerna i varje klausul är sann. På grund av de förbjudna paren är det en giltig tilldelning som löser 3SAT instansen.

Om det finns en 3SAT tilldelning som satisfierar 3SAT instansen, så finns det en tilldelning som gör minst en literal sann i varje klausul. Den här tilldelningen använder sig aldrig av båda noderna i ett förbjudet par. Alltså finns den väg mellan s och t som söks av PwFP. PwFP ligger i NP eftersom det går att gissa en lösning och verifiera den i polynomiell tid. Sammantaget är alltså PwFP NP-fullständigt.

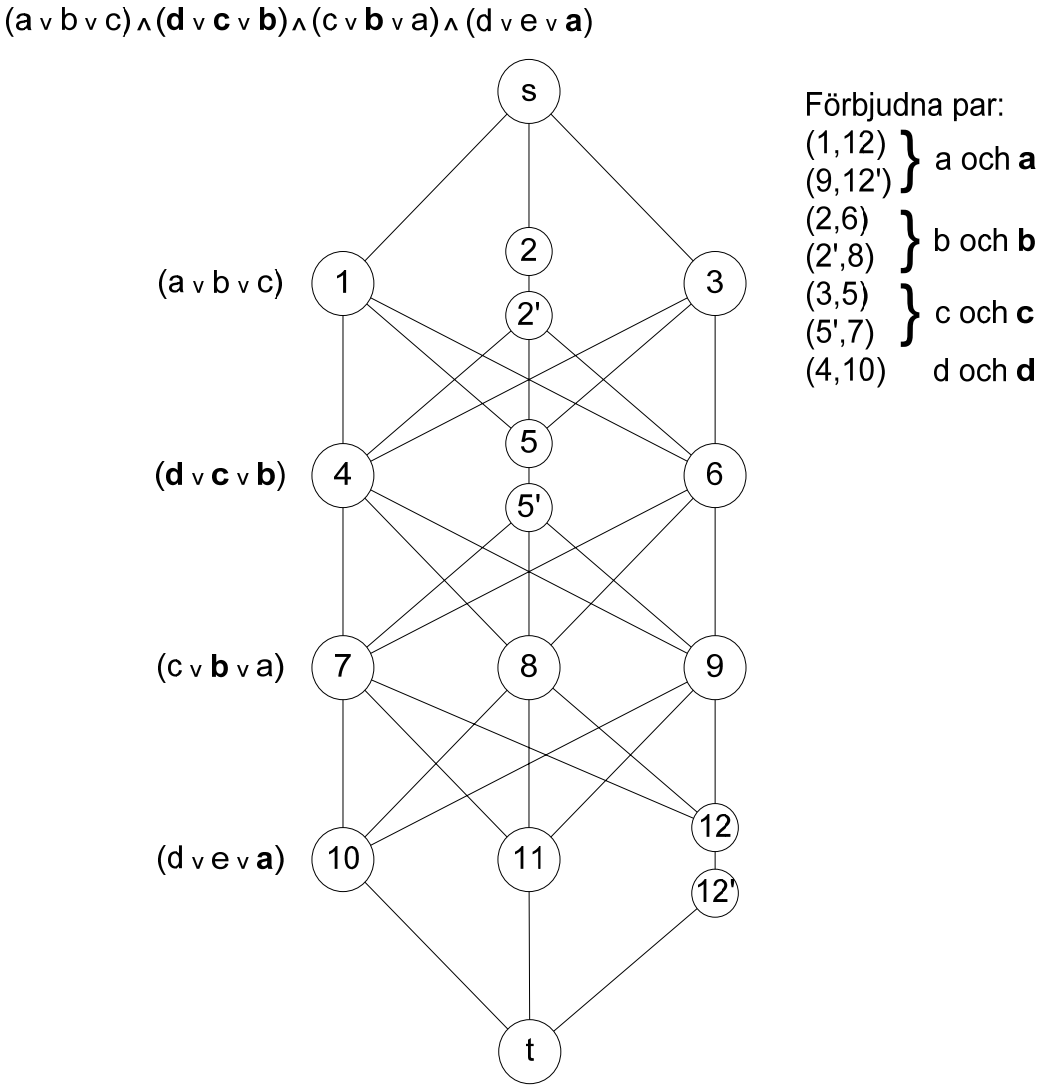
Nu tillverkas en konstruktion som reducerar 3SAT3V2L varianten från kapitel 2.4.4. Det går givetvis att reducera från allmänna 3SAT, men det blir enklare och lika kraftfullt på det här viset. Dessutom kan idén med fördubbling av raderna från kapitel 3.2.1 återanvändas. Som förut löses problemet med tre literaler genom att fördubbla den som är ensamt förekommande. Nästa sida visar ett exempel på hur det går till med en liten probleminstans.

Exempel Figur 3.5 visar en oriktad exempelgraf utan disjunkta par, t.ex. ingår 12 i två par.



Figur 3.5: Oriktad version av grafen.

Nu skapas de disjunkta paren. Eftersom 12 ingår i två par splittras den i två noder, 12 och 12'. Detsamma gäller för 2 och 5. Figur 3.6 visar slutresultatet.



Figur 3.6: Disjunkta par införs.

Exemplet visar hur en graf konstrueras med högst m extra noder från en instans i 3SAT3V2L med m klausuler. Det framgår att en väg från s till t som innehåller högst en nod från varje par ger en tilldelning som satisfierar 3SAT3V2L och tvärt om, en lösning till 3SAT3V2L ger en väg genom grafen. Att grafen är oriktad påverkar inte resultatet eftersom det inte gör något om vägen passerar en rad två gånger. Det kritiska är om den kan passera över huvud taget.

Exempel Det går att hitta en väg igenom PwFP instansen i det föregående exemplet som tilldelar en variabel två olika värden. T.ex. kan vägen $s \to 1 \to 5 \to 2' \to 4 \to 7 \to 11 \to t$ hittas. Den tilldelar variabeln c först värdet sant då 5 passeras och sedan falskt då 7 passeras. Det finns inget krav i PwFP instansen som förhindrar detta. Däremot är det så att om en väg studsar tillbaka på sin väg från s till t så passeras det lager som studsas mot senare via en annan nod. Det går då att förkorta vägen genom att klippa bort den del av vägen som studsar och ersätta den med en direkt väg igenom lagret. I det här exemplet kan vägen istället gå $s \to 1 \to 4 \to 7 \to 11 \to t$ och på så sätt slippa en otillåten tilldelning till 3SAT uttrycket. Detta sista steg i reduktionen utförs i polynomiell tid och tillåts inom ramen för reduktioner.

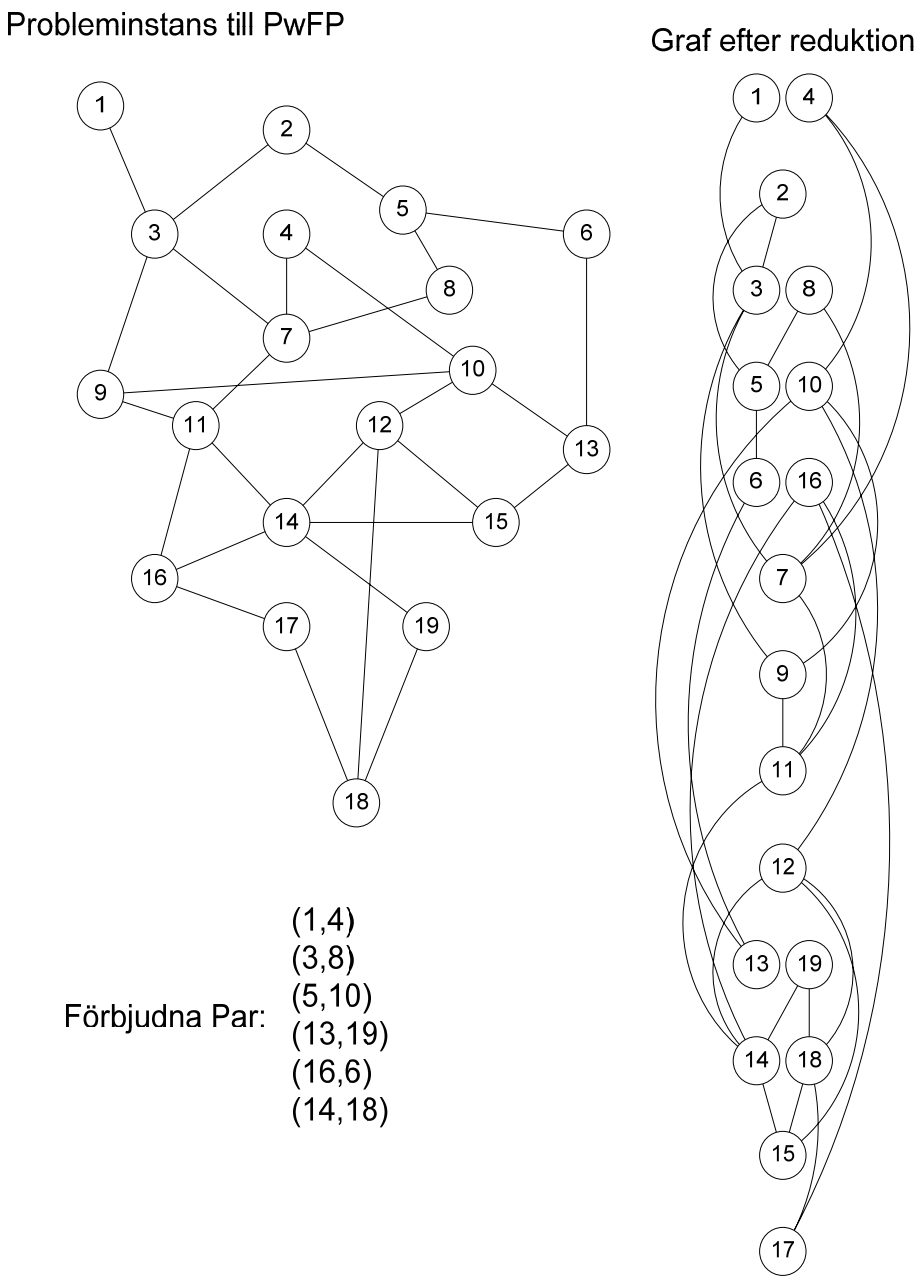
Sammanfattningsvis är alltså PwFP *NP*-komplett för oriktade grafer där alla par är disjunkta. Detta kan nu användas för att göra en reduktion från PwFP till SPANNERVÄG som har lägre stretchkonstant än den tidigare reduktionen. Beviset kommer ett steg längre ifrån 3SAT3V2L och behöver nu inte längre låta noder vara knutna till literaler utan kan betrakta dem bara som noder. För instansen till PwFP gäller att om det finns bågar mellan noder som är i ett förbjudet par kan dessa tas bort eftersom de aldrig kan användas.

Exempel Figur 3.7 visar ett exempel på en reduktion från PwFP till SPANNERVÄG.

Givet den vänstra grafen och de förbjudna paren som probleminstans till PwFP består reduktionen i att konstruera den högra grafen. Detta kan göras i polynomiell tid. Noderna placeras vertikalt efter sitt nummer. Detta nummer är godtyckligt och påverkar inte resultatet. De noder som ingår i ett förbjudet par placeras på samma rad som noden med lägst nummer skulle placeras på.

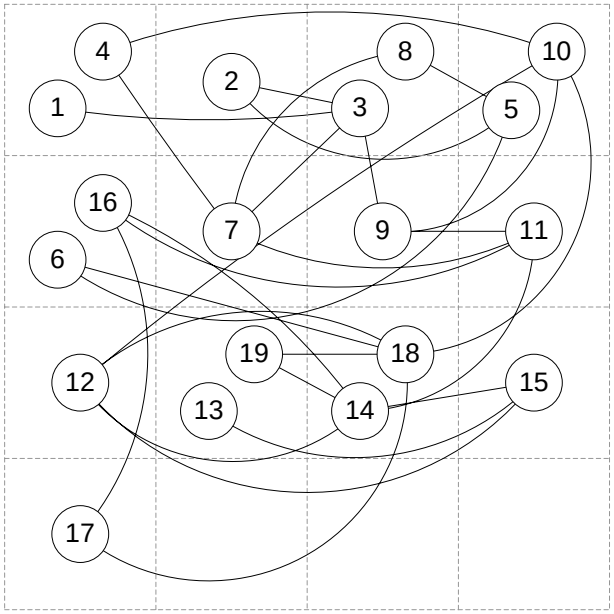
I den vänstra grafen saknas avstånd på bågarna. Den högra grafen är geometrisk och avståndet i vertikal led mellan noderna är en enhet och avståndet mellan nodparen ε enheter. Bågarna går egentligen raka vägen men för att se att grafen är samma som den vänstra är de ritade tydligare. På samma sätt som förut är det förhållandet mellan kortaste avståndet mellan två tillåtna noder och det längsta mellan dessa som sätter gränsen för stretchkonstanten. Den längsta bågen är i storleksordningen n och en väg mellan två tillåtna noder är då högst $n(n-1)$. Kortaste vägen från en nod i ett förbjudet par till den andra noden i paret är ungefär 2. För att detta inte skall tillåtas i en väg måste $2/\varepsilon$ vara större än n^2 om vi tänker oss SPANNERVÄG med stretchkonstant n^2. Alltså blir $\varepsilon = 2/n^2$.

Anledningen till att kravet på disjunkta par uppstår är att det inte skulle fungera att ha fler än två noder per rad. Det skulle kunna innebära problem om två eller fler av dessa måste besökas.



Figur 3.7: Exempelgraf före och efter reduktion.

Exempel Genom att öka dimensionen på den av reduktionen producerade grafen får vi en lägre stretchkonstant. Idén illustreras genom att fortsätta på föregående exempel. Figur 3.8 visar hur reduktionen kan se ut i två dimensioner.



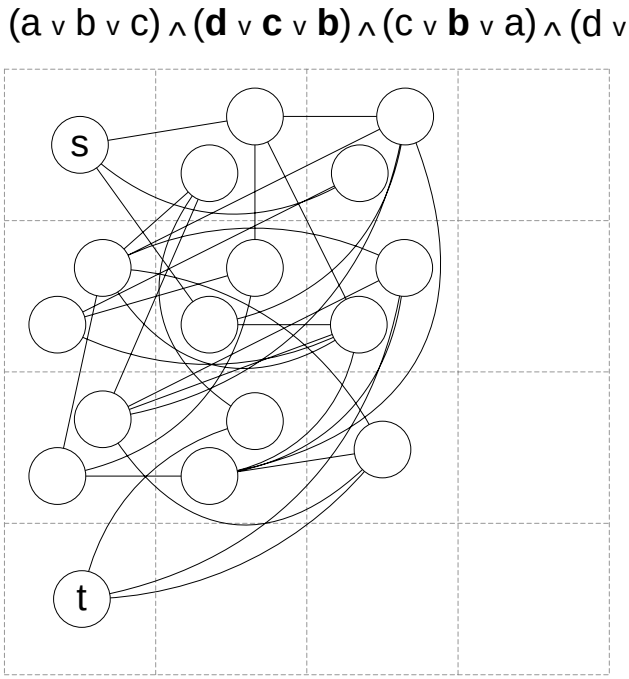
Figur 3.8: Tvådimensionell reduktion.

Genom att placera varje nod eller nodpar i en kvadrat fås en ny reduktion. Om n noder placeras i en kvadrat kommer dimensionerna att vara ungefär $\sqrt{n} \cdot \sqrt{n}$ rutor. Den längsta bågen är av längd $\sqrt{2n}$. Analogt med ovan fås då längsta vägen mellan två noder till $(n-1)\sqrt{2n}$. Det går alltså att få *NP*-komplettheten att hålla för en stretchkonstant av storlek $\Theta(n^{3/2})$. Genom att vidare öka dimensionen till k fås det generella uttrycket $\Theta(n^{1+1/k}\sqrt{k})$. Det framgår av exemplen ovan att en lösning till SPANNERVÄG innebär en lösning till PwFP och tvärtom. För alla dessa stretchkonstanter är alltså SPANNERVÄG *NP*-komplett. Detta är en klar förbättring av det tidigare resultatet från kapitel 3.2.1.

3.2.3 Jämförelse mellan reduktionerna

För att återknyta till den direkta reduktionen från 3SAT3V2L visas nu de två olika typerna av reduktioner från samma startuttryck. Den senare reduktionen återges i två dimensioner. 3SAT3V2L uttrycket som reduceras är det som används i figur 3.5 och figur 3.6.

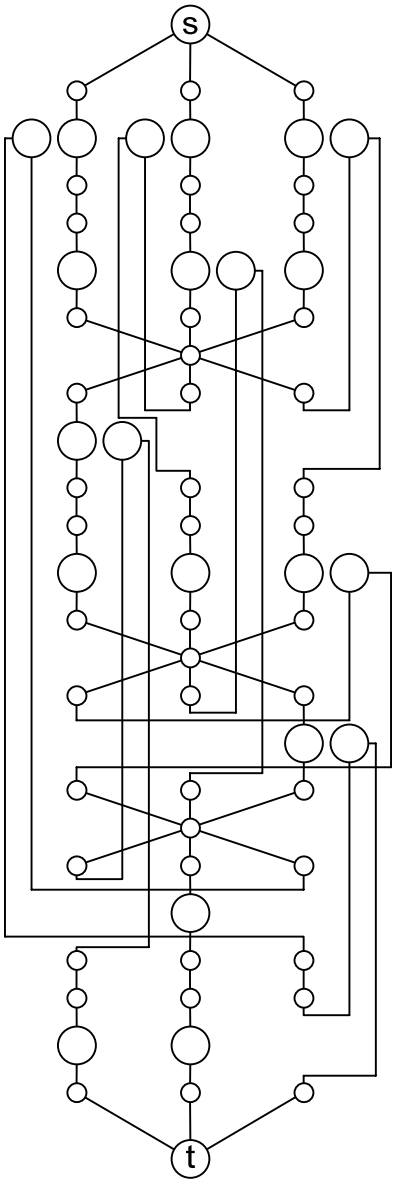
I bilden är bågarna som förut böjda, men har som vikter det geometriska avståndet mellan noderna. Vissa av noderna inom en kvadrat har placerats för att underlätta dragning av bågarna. Om längden av en ruta är 1 så kommer 17-



Figur 3.9: Reduktion av 3SAT med hjälp av PwFP.

spannervägar att vara ekvivalent med en tilldelning till 3SAT uttrycket. Detta är en uppskattning som detaljerat beskrivs i appendix A.4. Där visas också att noderna bör placeras inom $2/17$ från varandra inom en ruta. Om Θ-uttrycket hade använts istället hade spannerkonstanten som krävts blivit 99 i stället vilket är värsta fallet.

Genom att placera även de noder som hamnar ensamma två och två i samma ruta så långt ifrån varandra som möjligt fås antalet rutor ner till $n/2$. Då måste även avståndet ändras mellan par som inte kan vara tillsammans då det nu blir lite kortare vägar mellan dessa.



Figur 3.10: Direktreduktion från 3SAT uttrycket.

Figur 3.10 visar hur motsvarande reduktion direkt från 3SAT ser ut. Det visar sig finnas två längsta vägar genom grafen med identisk längd. I appendix A.4 beräknas att det krävs en stretch på 72 för att möjliggöra reduktionen. Bakåtnoder bör placeras högst 1/36 från de ordinarie noderna. Om Θ-uttrycket använts istället skulle det krävas en spannerkonstant på 529.

Detta exempel visar att de värstafallsbedömningar som gjorts i texten kanske är i värsta laget. Det kanske inte ens finns grafer där dessa inträffar. Det kan också

vara så att detta exempel var speciellt lätt, men det troliga är att uppskattningarna är en stor faktor på den säkra sidan. Exemplet visar också att direktreduktionen får ett sämre värde vilket motiverar reduktionen från PwFP.

3.3 Små spannerkonstanter

I huvudsak har det diskuterats stora stretchkonstanter. Det har inte ens varit konstanter utan värden som beror på antalet noder i grafen. Vad gäller då för mindre konstanter?

Sats 3.3.1 *För spannervägar med stretchkonstant 1 går det att avgöra i polynomiell tid om en väg finns.*

Bevis På grund av den extrema konstanten får inga avvikelser göras på vägen mellan de två noderna. Så alla noder som inte ligger på linjen mellan s och t kan ignoreras. Vidare kommer bara noder på linjen som sedan leder framåt att kunna användas. En simpel algoritm går igenom de noder som ligger på linjen en efter en och ser efter om den kan ligga på vägen. Den här föreslagna algoritmen använder *backtracking*. Steg för steg arbetar algoritmen som följer:

1. Ta bort noder utanför linjen.
2. Lägg till s i den aktuella vägen.
3. Kontrollera vilka noder som kan nås från den sista noden på den aktuella vägen. Om någon går i riktning mot t så lägg till den.
4. Om ingen nod hittas i steg 3, ta bort den nod vi befinner oss i eftersom den inte kan ligga på vägen. Om det var s som togs bort avsluta med ett negativt svar, annars börja om från 3.

Varje nod läggs till högst en gång och tas bort högst en gång dvs. algoritmen tar $O(n)$ tid.

Sats 3.3.2 *Varje graf har ett ε-värde för vilket SPANNERVÄG är polynomiellt för $(1+\varepsilon)$-stretch.*

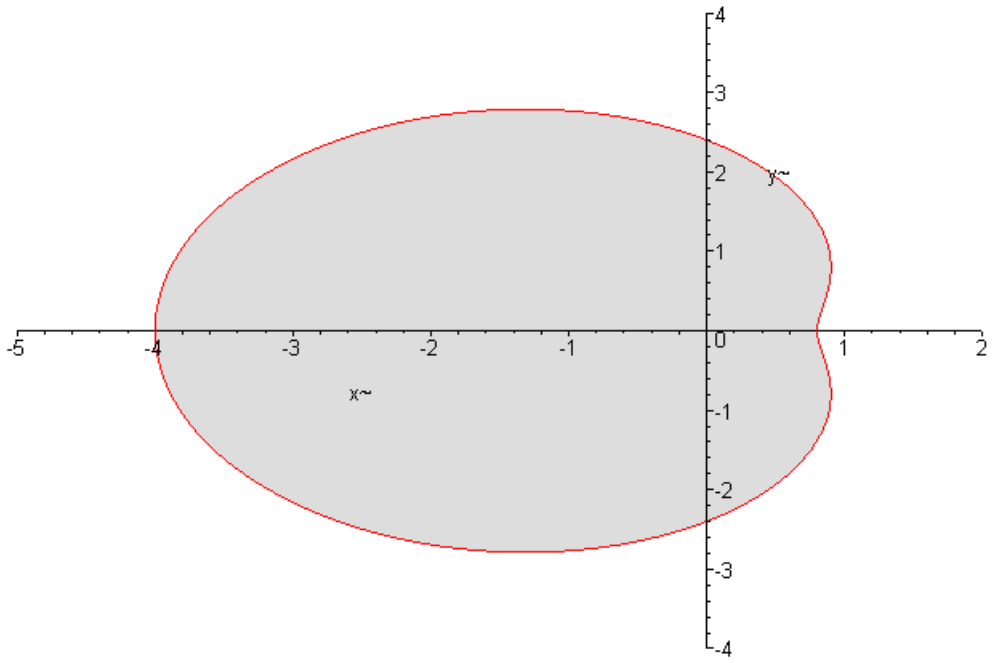
Bevis Om ε blir tillräckligt litet kan alla noder utanför linjen mellan s och t räknas bort. Därför fungerar algoritmen från sats 3.3.1 oförändrat.

3.4 Ett par observationer

Det finns ett par intressanta observationer som ger lite mer förståelse av problemet. Den första gäller grafer där avstånden mellan noderna skiljer sig exponentiellt åt. Det innebär att nodavstånden följer olika skalor. Ett exempel är en graf där noderna ligger på punkter i planet med y-koordinat 0 och x-koordinater 1,2,4, ..., 2^n.

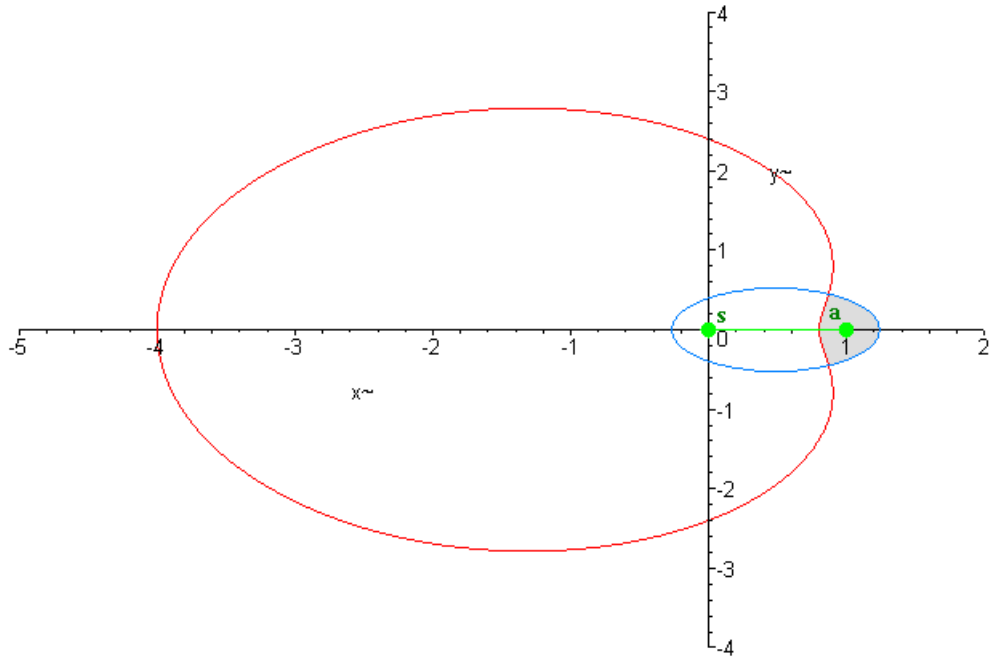
Om stretchkonstanten fixeras vid 1.1 så tillåts inte långa omvägar. Om grafen ligger i planet kommer varje båge som används att göra en stor del av detta onåbart.

Figur 3.11 visar hur mycket av planet som blir onåbart efter det att första noden besökts om den ligger på position $(1, 0)$ och s ligger på $(0, 0)$. Om resterande noder ligger utanför detta område påverkas de inte av om $(1, 0)$ besökts eller inte. På liknande sätt kan noderna placeras längre och längre bort ifrån varandra så att alla noder som ligger längre från s än den nyss besökta blir nåbara utan att ett brott mot spanneregenskapen sker. Detta ger en exponentiell graf och de möjliga spannervägarna måste gå mot noder längre och längre bort från s. Detta gör att en algoritm liknande den för stretch 1 i föregående kapitel kan användas då det bara finns en riktning, mot större avstånd från s.



Figur 3.11: Område som blockeras av första bågen $(0, 0) \rightarrow (1, 0)$.

En annan intressant situation är den som visas i figur 3.12 med stretch 1.1. Här visas ett område som är tillåtet att besöka efter noden $a = (1, 0)$. Dessutom är det så att a kan besökas efter noder som ligger i det skuggade området. Det är en förhållandevis liten del av planet som innehåller noder med dessa egenskaper.



Figur 3.12: Det skuggade området innehåller noder som kan komma före eller efter a längs en spannerväg.

3.5 Slutsats och vidare arbete

Huvudresultatet i kapitel 3 är sats 3.2.1 som formuleras i kapitel 3.2. Den säger att problemet SPANNERVÄG är *NP*-komplett för stretch $\Theta(n^{1+1/k}\sqrt{k})$. Satsen bevisas i kapitel 3.2.2 genom att reducera det *NP*-kompletta problemet *Path with Forbidden Pairs* till SPANNERVÄG. I kapitel 3.2.1 görs en reduktion direkt från 3SAT vilket ger *NP*-kompletthet för stretch $\Theta(n^2)$.

Mot bakgrund av dessa resultat är det inte konstigt att den bästa algoritm som presenteras i kapitlet har en exekveringstid av storleksordningen $O(2^{0.822n})$. Den återfinns i kapitel 3.1 och delar upp grafen i två delar för varje upptäckt par som bryter mot spannervillkoret. Sedan löser den dessa delar rekursivt.

Då jag först började söka efter spannervägar genom grafer trodde jag att det skulle vara möjligt att hitta dessa på ett effektivt sätt. Jag testade giriga algoritmer,

divide-and-conquer algoritmer, algoritmer som använde dynamisk programmering och slutligen en variant på *backtracking* men alla försök blev icke-polynomiella. Till slut valde jag analysera några algoritmer vilket gav kapitel 3.1.

Det kändes naturligt efter dessa försök att bevisa problemet *NP*-komplett. Detta visade sig svårare än väntat, i alla fall för konstant stretch. Det var ganska rakt fram att reducera 3SAT3V2L varianten i kapitel 3.2.1 och jag gjorde detta innan jag kände till PwFP problemet. Men trots att den andra reduktionen skärper stretchen från $\Theta(n^2)$ till $\Theta(n^{1+1/k}\sqrt{k})$ är denna fortfarande inte konstant. Dessutom är det troligast att grafer som används t.ex. för avståndsbestämning med hjälp av AGL-algoritmen är tvådimensionella. Då blir den bästa bevisade stretchen $\Theta(n^{3/2})$.

Efter att ha försökt hitta effektiva algoritmer för konstant stretch och inte lyckats kändes det som om problemet verkligen var *NP*-komplett även här. Men efter att ha försökt visa detta är jag tillbaka på ruta ett. Jag kan inget säkert säga angående komplexiteten. Erfarenheter från nästa kapitel får mig dock att tro att det kan gå att lösa problemet effektivt om vissa krav ställs på grafen. T.ex. skulle krav som minsta och största båglängder hjälpa. Detta skulle utesluta konstiga grafer av exponentiell typ och dessutom omöjliggöra de grafer som används i bevisen för *NP*-kompletthet.

Vidare arbete inom området skulle kunna bestå i att ta fram heuristiker för att hitta spannervägar. Dessa kan arbeta antingen positivt eller negativt. Med det menar jag att de antingen kan arbeta efter att hitta vägar eller att blockera så många som möjligt för att snabbt minska sökmängden. Vid hög stretch tillåts fler vägar och det borde leda till att en väg hittas snabbare. Vid låg stretch blockeras fler vägar och det kan löna sig att satsa på att minska sökmängden.

Kapitel 4

Heltalsgrafen

4.1 Introduktion och definition

Då det är svårt att visa eller motbevisa att ett problem har en viss svårighetsgrad är det naturligt att se efter om liknande problem kan bidra med användbar information. Det kan också vara så att generaliseringar eller specialiseringar av problemet undersöks. Denna information kan kanske sedan användas på ursprungsproblemet för att visa eller motbevisa svårighetsgraden. I syfte att komma till djupare insikter i huruvida det är möjligt att lösa problemet SPANNERVÄG med konstant stretch kommer nu en speciell typ av graf att undersökas - heltalsgrafen.

Definition 4.1.1 *En* **heltalsgraf** *är en graf där alla noder ligger på heltalspositioner längs tallinjen. I en heltalsgraf med n noder ligger noderna på heltalspunkterna* $0, 1, 2, \ldots, n-1$.

Denna typ av graf är lättare än en allmän graf på två viktiga punkter:

- Den är endimensionell
- Den har en minsta längd på bågar

Definition 4.1.2 Spannerproblemet i heltalsgrafen *består i att hitta en spannerväg från nod 0 till nod* $n-1$ *som uppfyller spannervillkoret för önskad stretch. Nod 0 kommer att kallas för s och nod* $n-1$ *för t i enlighet med tidigare benämningar. En båge som traverseras från en nod med lägre koordinat till en med högre kommer att kallas en* **högerbåge** *och analogt för* **vänsterbågar** *som också kallas* **bakåtbågar** *då riktningen från s går framåt mot t.*

Båda de egenskaper som skiljer heltalsgrafen från allmänna grafer omöjliggör *NP*-fullständighetsbevisen i 3.2. Är då spannerproblemet givet ett antal bågar och noder placerade så här fortfarande *NP*-fullständigt? Till att börja med kan sägas att *NP*-kompletthetsbevisen för generella grafer i första skedet endast gäller

då stretchen är kvadratisk. Som en jämförelse kommer att visas att motsvarande problem i heltalsgrafen är polynomiellt lösbart.

Detta ger en intressant betraktelse. För generella grafer gäller att 1-spanners går att lösa i polynomiell tid medan n^2-spanners troligen inte går (*NP*-komplett). Antingen kan det vara så att 1-spanners är den enda typen av spanners som är polynomiellt lösbara, eller så måste det finnas en gräns där algoritmerna går från att vara polynomiella till att inte vara det. Det är högst otroligt att det i intervallet $[1, n^2]$ finns flera byten mellan de två typerna av komplexiteter.

Om detta skulle gälla även för heltalsgrafen där det på båda sidor om det okända intervallet finns polynomiella algoritmer för att lösa problemet så skulle det innebära att det för alla stretchkonstanter i intervallet går att lösa problemet i polynomiell tid. Det borde alltså gå att hitta en polynomiell algoritm för att t.ex. bedöma om det finns en väg från s till t som uppfyller kravet att vara en 1.5-spannerväg. Det återstår nu att undersöka om en sådan algoritm kan konstrueras.

4.2 Kända egenskaper

I det här avsnittet samlas ett antal egenskaper som gäller för heltalsgrafen.

Sats 4.2.1 *Problemet SPANNERVÄG med stretch 1 är lösligt i polynomiell tid för heltalsgrafen.*

Bevis Satsen följer eftersom den är ett specialfall av sats 3.3.1.

Sats 4.2.2 *Problemet SPANNERVÄG med stretch n^2 är lösligt i polynomiell tid för heltalsgrafen.*

Bevis Satsen följer av sats 4.2.3 som säger att längsta vägen mellan två noder i grafen är mindre än n^2. Då avståndet mellan två noder har en minsta gräns som är 1 är kvoten mellan avståndet i grafen och avståndet geometriskt aldrig större än n^2. Alltså är alla vägar tillåtna och en väg hittas då i $O(n)$ tid med t.ex. en djupet-först sökmetod. En algoritm för problemet REACHABILITY som innebär att hitta en väg mellan två noder finns i Papadimitriou [10].

Sats 4.2.3 *För längsta möjliga vägen i en heltalsgraf med n noder, S_n, gäller att $S_n \leq \frac{n^2}{2}$.*

Bevis Beviset följer direkt av lemma 4.2.1 och lemma 4.2.2 som bevisas nedan. Eftersom en väg i en graf inte blir kortare om grafen har fler bågar används en fullständig heltalsgraf i bevisen. Det som gäller för denna blir då också en övre gräns för alla heltalsgrafer.

Lemma 4.2.1 *Den längsta vägen i en fullständig heltalsgraf antas då varje båge kopplar samman en nod från vänstra halvan med en nod från högra halvan.*

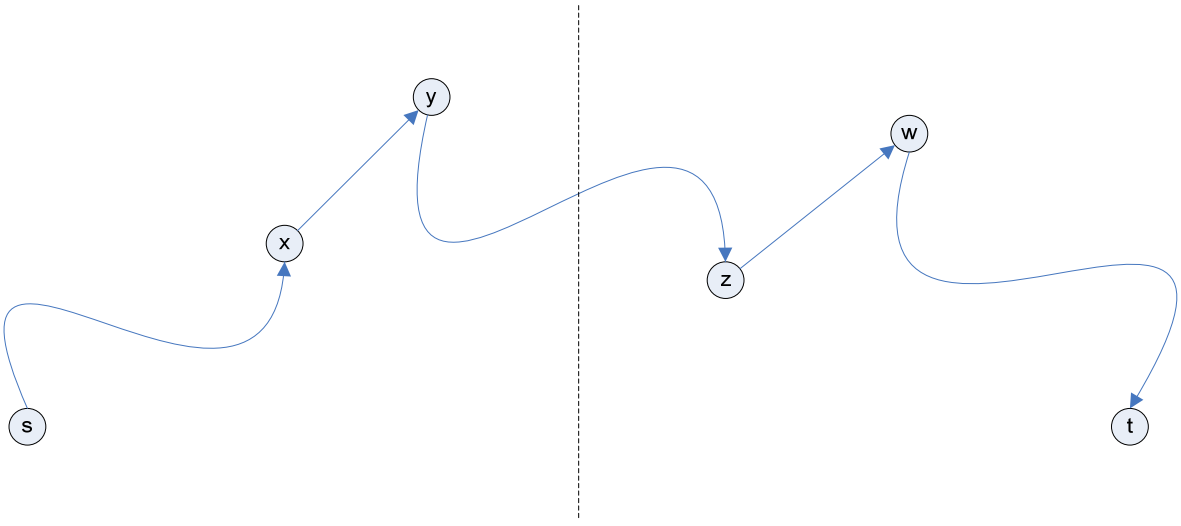
Bevis Antag till att börja med att grafen har ett jämnt antal noder. Detta innebär att gränsen mellan vänster och höger halva går mellan de mittersta noderna. Problemet med udda antal noder löses senare.

Det bevisas att för varje väg mellan s och t där det finns en båge inom ena halvan finns det en väg där denna båge tagits bort och ersatts av bågar mellan halvorna. Denna nya väg är minst lika lång som den ursprungliga vägen och har färre bågar som inte korsar mitten. Genom att upprepa detta tills alla bågar korsar mitten fås en graf där alla bågar korsar mitten. Denna slutliga väg är minst lika lång som den första vägen.

Det kan antas att alla noder i grafen passeras längs en väg mellan s och t av maximal längd. Om inte så blir vägen inte kortare för att den mellanlandar på de noder som eventuellt inte passerades.

Givet en väg från s till t där inte alla bågar går mellan de två halvorna. Då finns två noder, x och y på vänster halva som är förbundna med varandra. Det finns då även två noder, z och w på höger halva som är förbundna med varandra. Detta måste gälla då det är lika många noder på båda sidorna och två noder på vänster sida inte är kopplade till noder på höger sida. Eftersom alla noder antogs besökas har alla en båge in och en båge ut i riktning längs vägen.

Det finns två möjligheter för s att ansluta till paren xy och zw. Antingen kommer s att nå xy först eller zw först. Figur 4.1 visar hur det kan se ut när s når xy och mer bestämt x först.



Figur 4.1: Graf med noder kopplade inom samma sida.

Bilden visar inte heltalsgrafen i ett plan då korsande vägar medför problem att visualisera det som sker. De kurviga pilarna visar vägar som kan gå hur som helst men förbinder de två noderna. De kan t.ex. korsa mittlinjen en eller flera gånger.

De raka pilarna visar en direkt förbindelse mellan två noder. Det finns inte heller ett krav på att x ligger till vänster om y utan detta är bara ett exempel. Följande beteckningar används nu:

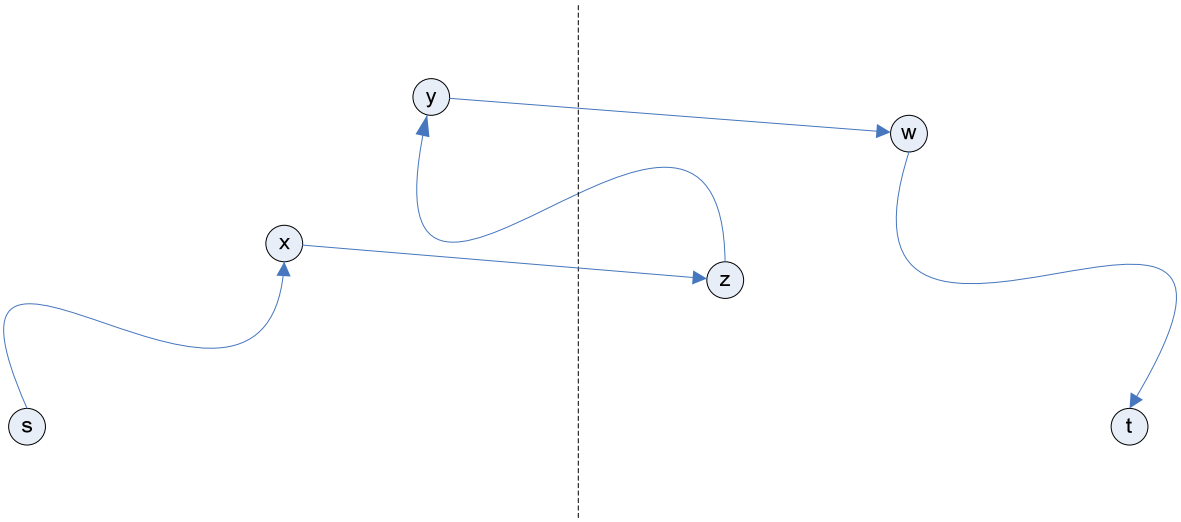
s^*x - Längden av vägen från s till x. Vägen går antingen direkt eller via andra noder.

xy - Längden av bågen mellan x och y.

Då kan enligt bilden skrivas

$$s^*t = s^*x + xy + y^*z + zw + w^*t \tag{4.1}$$

I syfte att hitta en längre väg kopplas noderna om i grafen vilket ger en ny väg enligt figur 4.2.



Figur 4.2: Grafen efter omkastning av nodordningen.

Vägen ges nu av

$$s^*t = s^*x + xz + z^*y + yw + w^*t \tag{4.2}$$

Här gäller att $z^*y = y^*z$ då det är samma väg i olika riktningar. Differensen kan då skrivas

$$(4.2)-(4.1) = (s^*x+xz+z^*y+yw+w^*t)-(s^*x+xy+y^*z+zw+w^*t) = xz+yw-xy-zw \tag{4.3}$$

I exemplet är ordningen på noderna x-y-z-w. Detta ger (i den 1-dimensionella heltalsgrafen) $xz = xy + yz$ och $yw = yz + zw$. Med hjälp av dessa substitutioner

Ordning	Substitutioner	Förenklat
$x-y-w-z$	$xz = xy + yw + wz$	$(4.3) = 2yw > 0$
$y-x-z-w$	$yw = yx + xz + zw$	$(4.3) = 2xz > 0$
$y-x-w-z$	$xz = xw + wz$ $yw = yx + xw$	$(4.3) = 2xw > 0$

Tabell 4.1: Beräkningarna för de tre permutationerna i tabellform.

blir följande förenkling $(4.3) = 2yz > 0$ möjlig. Alltså är vägen i figur 4.2 längre än den ursprungliga. Tabell 4.1 visar de tre andra möjliga permutationerna av noderna. I samtliga fall ser vi att en förändring ger en längre väg.

Även då x och y ligger på högra sidan och z samt w ligger på vänstra sidan håller argumenten ovan. Dessa fall uppstår då första bågen som inte korsar mitten ligger på högra sidan. Det ger på samma sätt som förut upphov till en tabell som visar hur differensen beräknas, se tabell 4.2.

Ordning	Substitutioner	Förenklat
$z-w-x-y$	$xz = xw + wz$ $yw = yx + xw$	$(4.3) = 2xw > 0$
$z-w-y-x$	$xz = xy + yw + wz$	$(4.3) = 2yw > 0$
$w-z-x-y$	$yw = yx + xz + zw$	$(4.3) = 2xz > 0$
$w-z-y-x$	$xz = xy + yz$ $yw = yz + zw$	$(4.3) = 2yz > 0$

Tabell 4.2: Beräkningarna för de andra fyra permutationerna i tabellform.

Även här syns att vägen i samtliga fall blir längre. Slutsatsen är att oavsett om första paret ligger i vänstra eller högra halvan, samt att oavsett inbördes ordning så kommer vägen att bli längre om bågarna går mellan halvorna varvid lemmat är bevisat, för ett jämnt antal noder.

Observera att resonemanget ovan innebär även att första och sista bågen bör gå mellan halvorna. Om första bågen går inom vänstra halvan är sista bågen tvingad att gå inom högra halvan och genom att byta dessa och sedan följa den resterande vägen baklänges fås en längre väg.

Om grafen har ett udda antal noder finns det två möjligheter för vägen att passera den mellersta noden. Antingen passeras den som en del av en väg mellan sidorna, eller också kommer vägen att via noden studsa tillbaka till samma sida den kom från. Det första alternativet leder till att vägen går från sida till sida på samma sätt som tidigare visat sig ge minst lika långa vägar som andra varianter. Det

andra alternativet leder till att 2 noder på ena halvan samt mitten noden används upp. Då måste det finnas två noder på den andra halvan som är sammankopplade och detta leder som visats ovan till att vägen kan vara kortare än det första alternativet. Sammanfattningsvis gäller alltså det som bevisats ovan också för grafer med udda antal noder, i resonemanget räknas mittnoden tillhöra samma sida som dess föregångare längs vägen. Detta avslutar beviset.

Lemma 4.2.2 *Låt S_p beteckna längden av en väg mellan s och t i en heltalsgraf med p noder, där alla noder ingår i vägen och varje båge korsar mittlinjen. Då gäller att $S_p \leq \frac{p^2}{2}$.*

Bevis Först beskrivs längden i en graf med jämnt antal noder. Låt p beteckna antalet noder i grafen. Då gäller $p = 2n$ för något n. Avståndet mellan nod n_i och n_j skrivs $|n_i - n_j|$. Längden av vägen kan då skrivas $|s-n_1|+|n_1-n_2|+\cdots+|n_{p-2}-t|$ där noderna indexerats i den ordning de passeras. Här gäller att varje nod utom s och t är med i två termer. Eftersom bågarna korsar mitten gäller att varje term innehåller en nod från vardera sidan och att alla heltal från 0 till $p-1$ förekommer. I denna summa kommer t.ex. att finnas $|n_i - 1|$ och $|1 - n_j|$ då noden på position 1 har ingrad ett och utgrad ett längs vägen. Eftersom 1 ligger i vänstra halvan kan absolutbeloppet tas bort vilket ger $n_i - 1$ och $n_j - 1$. Genom att konsekvent göra detta för alla termer fås genom att även byta indexering i höger led

$$\begin{aligned} &|s - n_1| + |n_1 - n_2| + \cdots + |n_{p-2} - t| = \\ &\quad = n_1 + (n_2 - 1) + (n_3 - 1) + (n_4 - 2) + \cdots + (n_{p/2} - (p/2 - 1)) \end{aligned} \tag{4.4}$$

Här används med den nya indexeringen index 1 för den nod som kopplas till 0, index 2 och 3 för de som kopplas till 1 osv. Noderna som anges med index ligger på höger sida medan de som anges med siffror ligger till vänster. Eftersom varje term förekommer två gånger (utom s och t) kan detta sammanfattas till

$$\begin{aligned} S_{p=2n} &= |s - n_1| + |n_1 - n_2| + \cdots + |n_{p-2} - t| = \\ &= (p-1) + 2\sum_{k=p/2}^{p-2} k - 2\sum_{k=1}^{p/2-1} k = \frac{p^2}{2} - p + 1 \leq \frac{p^2}{2} \end{aligned} \tag{4.5}$$

Om grafen har udda antal noder, $p = 2n+1$, gäller enligt tidigare att mittnoden passeras som en del i en väg mellan sidorna. Den kommer då att ingå i en term med en nod från vänster sida och en term med en nod från höger sida. Detta innebär att den i summan förekommer två gånger med motsatt tecken och därmed försvinner. Därför kommer summan att se ut precis likadan som för $2n$ noder med

den skillnaden att varje båge får en ökad längd med 1. Då vägen består av $2n$ bågar kommer alltså summan att bli

$$S_{p=2n+1} = |s - n_1| + |n_1 - n_2| + \cdots + |n_{p-2} - t| = S_{2n} + 2n = \\ = \frac{(2n)^2}{2} - 2n + 1 + 2n = \frac{p^2}{2} - p + \frac{3}{2} \leq \frac{p^2}{2} \tag{4.6}$$

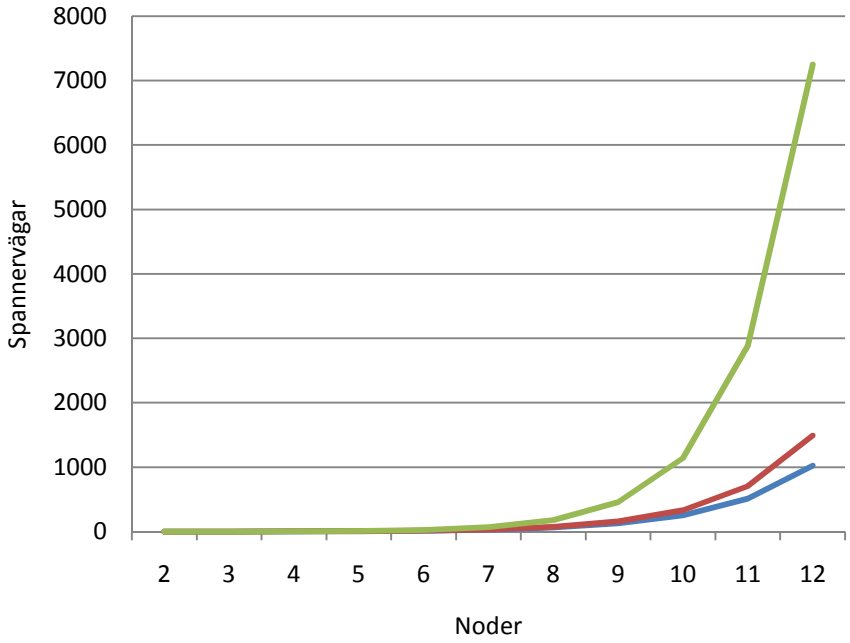
Detta avslutar beviset.

I syfte att få en känsla för hur många spannervägar det kan finnas i en heltalsgraf redovisas nu lite information. Tabell 4.3 visar antalet vägar och antalet spannervägar vid olika stretch i heltalsgrafer med upp till 16 noder. Figur 4.3 visualiserar några av dessa värden.

Noder	Vägar	Stretch 1	Stretch 2	Stretch 4	Stretch 8	Stretch 14
2	1	1	1	1	1	1
3	2	2	2	2	2	2
4	5	4	4	5	5	5
5	16	8	8	12	16	16
6	65	16	17	28	62	65
7	326	32	36	70	198	326
8	1957	64	76	180	568	1641
9	13700	128	160	456	1728	7582
10	109601	256	336	1141	5514	30129
11	986410	512	708	2880	18102	104496
12	9864101	1024	1492	7246	58751	375952
13	108505112	2048	3144	18136	187052	1418996
14	1302061345	4096	6624	45566	585904	5622771
15	16926797486	8192	13952	114628	1850740	22983694
16	236975164805	16384	29394	288280	5895322	94487897

Tabell 4.3: Antalet vägar och spannervägar vid olika stretch i fullständiga heltalsgrafer av olika storlek.

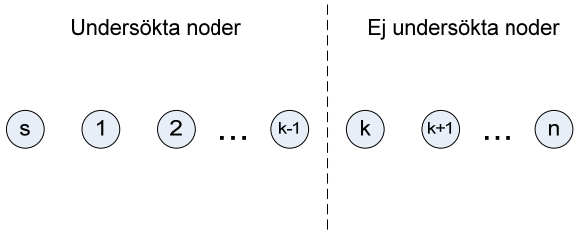
I figuren ser ökningen ut att vara exponentiell. Detaljerna i tabellen visar att antalet spannervägar för stretch 1 är $(n-2)!$. Detta beror på att alla vägar går åt höger så en nod är antingen med eller inte med längs vägen. För stretch n^2 blir antalet vägar $\Theta(n!)$. Detta beror på att alla vägar är tillåtna och då blir antalet vägar lika med antalet permutationer av noderna mellan s och t. Övriga stretchvärden hamnar mellan dessa båda gränser. En intressant fråga är för vilken stretch antalet vägar övergår från att vara exponentiellt till fakultet.



Figur 4.3: Antal spannervägar för stretch 1, 2 och 4.

4.3 Inkrementell metod

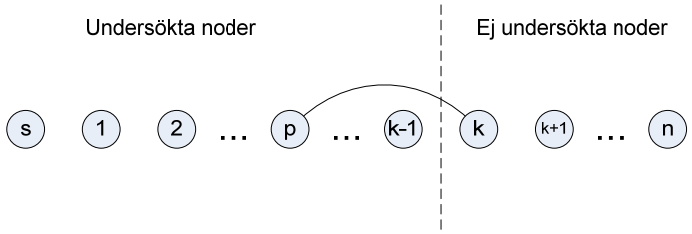
Det går att nå ett resultat för heltalsgrafen som är bättre än det för allmänna grafer genom att bygga inkrementellt. Algoritmen går igenom grafen från vänster till höger, en nod i taget. I varje iteration upprätthålls invarianten att alla spannervägar mellan noder som finns med än så länge har upptäckts. Det kommer att krävas n iterationer att gå igenom hela grafen. Figur 4.4 visar hur det ser ut i iteration k.



Figur 4.4: Steg k i den inkrementella algoritmen.

Enligt tidigare har alla spannervägar som innehåller noderna s till och med $k-1$ blivit beräknade. Dessa är undansparade och används nu för att nå k. Noderna gås igenom från vänster till höger och undersöks efter bågar till k. Om en båge hittas från p till k, se figur 4.5, kommer nya vägar att skapas. Utgående från alla de spannervägar från s till p som finns lagrade läggs sista bågen (p, k) till och det

kontrolleras om detta ger en spannerväg. Om så är fallet sparas vägen undan. Detta innebär att alla spannervägar till k, bestående endast av noder till vänster om k, som har p som näst sista nod nu är sparade.



Figur 4.5: En båge funnen från nod p till nod k.

När detta har upprepats för noderna $s, 1, 2, \ldots, k-1$ så är alla spannervägar till k innehållande noder $\leq k$ upptäckta. Nu kontrolleras vilka av noderna $1, 2, \ldots, k-1$ som kan nås från k. Om det finns en båge från k till q med $1 \leq q \leq k-1$ undersöks om någon av spannervägarna till k kan utökas med en båge till q. Om så är fallet läggs vägen till listan över spannervägar till q. När en väg till q lagts till undersöks rekursivt vilka nya vägar via q som kan hittas. På så sätt upprätthålls invarianten att alla spannervägar innehållande noder $\leq k$ är upptäckta.

Vad är tidskomplexiteten för den här algoritmen? Och hur stort utrymme behöver den? Både utrymmeskraven och tidskomplexiteten är väldigt stora. Eftersom alla spannervägar lagras fås en undre gräns i både utrymme och tid genom att betrakta antalet spannervägar. Enligt tabell 4.3 och resonemangen som följer denna är komplexiteten någonstans mellan $O(2^n)$ och $O(n!)$.

Det går att skärpa algoritmen lite om det ställs krav på stretchkonstanten. Ett krav är därför fortsättningsvis att stretchkonstanten t är mindre än 2. Väldigt liten stretch kommer att betraktas även om resultaten kommer att gälla även för större stretch. Inledningsvis betraktas $t = 1.1$ som ett exempel. Här accepteras alltså endast 10% avvikning mellan avstånden via spannervägen och det avståndet längs tallinjen. Detta ger en inskränkning i vilka vägar som är intressanta i en viss nod.

Invarianten ändras till "i iteration k är det fastställt vilka noder $\leq k$ som går att nå via spannervägar innehållande endast noder $\leq k$. Vidare finns i iteration k all information sparad som senare kan behövas för att ta reda på vilka noder som går att nå via spannervägar". De vägar till nod k som behövs sparas sägs sparas i nod k. På så vis vet varje nod vilka vägar till den som behövs sparas.

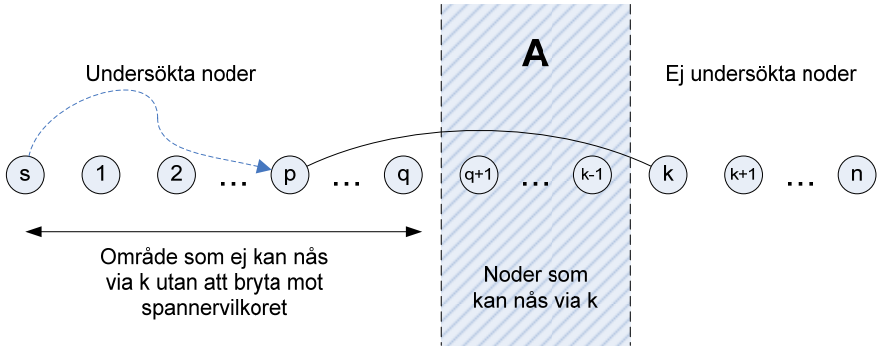
Detta är en invariant som behöver en ingående förklaring. Behovet att spara alla vägar släpps och nu sparas endast resultatet av vilka noder som går att nå samt information som behövs av senare iterationer. Den här informationen består som snart kommer att visas av avslutningen på vissa vägar till k. En observation är att om den eftersökta spannervägen från s till t hittas så finns bara möjlighet att säga att den finns, inte återge vilka noder som ligger längs den.

Om det är känt att det existerar en spannerväg mellan s och t går det dock att

köra algoritmen n gånger för att få reda på vilka noder som ingår i vägen. En nod tas bort och algoritmen exekveras, hittas ingen väg längre ingick den borttagna noden i vägen från s till t. Finns det fortfarande en väg kan noden tas bort under resten av körningarna. Till slut kommer endast de noder som krävs för att det skall finnas en väg att vara kvar. Det kan finnas flera vägar med olika kombinationer av noder. Det förfarande som nyss beskrevs ger endast noderna längs en väg. När det är känt vilka noder som ingår i vägen kan vägen beräknas genom att alla andra noder tas bort och Dijkstras algoritm, se kapitel 2.5.1, exekveras. Eftersom det finns en spannerväg finns det enligt tidigare resonemang, se kapitel 3.1, en kortaste sådan vilket är vad Dijkstras algoritm hittar.

Vilken information behöver sparas i iteration k? Eftersom alla vägar av intresse som bara berör noder $\leq k-1$ redan finns sparade behöver ny information endast sparas om vägar via k. Dels skall alltså undersökas om k kan nås och om så är fallet den information som kan behövas senare angående de olika vägar till k som hittats. I iterationen måste dessutom alla nya vägar till noder $\leq k-1$ undersökas då dessa kan ge nya vägar och ny information som behövs för att upprätthålla invarianten.

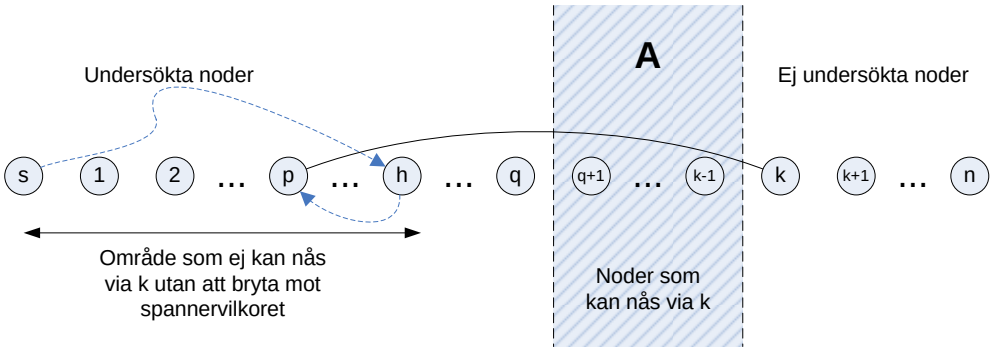
Antag att en väg hittas till k i iteration k. Det finns då en nod till vänster om k, säg p med bågen (p, k). Denna väg måste dessutom uppfylla spannervillkoret för att vara av intresse.



Figur 4.6: En väg hittas från p till k.

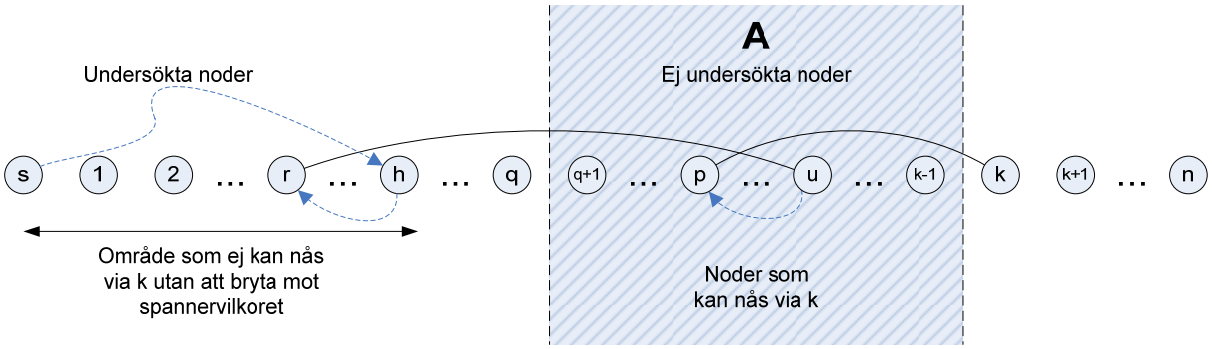
Figur 4.6 visar att noder utanför den vänstra gränsen av området **A** inte kan nås via k. Detta beror på att spannervillkoret bryts då avståndet från s till noder $\leq q$ via grafen blir för stort i förhållande till det euklidiska avståndet. Följande information angående vägen till k är nu av intresse. Vilken var den högraste noden, h, till vänster om **A** som passerades på vägen till p? Hur lång var vägen mellan denna nod och k? Figur 4.7 visar situationen.

Om h är nära $q+1$, eller om delvägen från h till k var lång, kommer detta att begränsa möjligheten för spannervägar via k att nå noder nära den vänstra gränsen av **A**. Det kommer även att innebära motsvarande begränsningar för noder inom **A** att senare nå noder nära den vänstra gränsen. Alltså måste den här informationen



Figur 4.7: Exempel på information som måste sparas.

sparas. Om det finns många vägar till p måste de alla granskas ur denna synpunkt och den bästa, dvs. den som tillåter störst område till höger om q att besökas, sparas i k. Det spelar alltså ingen roll för senare vägar exakt vad som händer till vänster om gränsen eftersom detta område aldrig kan besökas.



Figur 4.8: k nås via en nod inom **A**.

Figur 4.8 visar hur det kan se ut om p ligger i **A**, eller om **A** besökts innan p. De noder som besökts i intervallet kommer att inskränka möjligheten för vägen via k att senare nå noder till vänster om k. Alltså måste dessa sparas för varje väg till k.

Sammanfattningsvis skall i iteration k för varje väg till k följande lagras:

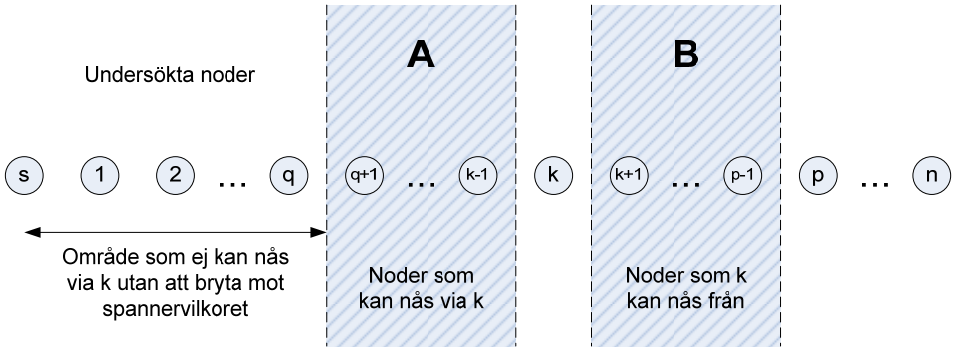
1. Den del av vägen till k som startar med r vilket är sista noden innan vägen för första gången passerar gränsen mellan q och $q+1$ dvs. in i **A**.

2. Den högraste noden, h, till vänster om **A** samt längden av vägen från h till k.

Om vägen i det streckade området är identisk för två vägar skall den lagras som har kortast avstånd mellan h och k. Här kan det hända att h sammanfaller med r eller p med u.

Då en väg hittats till k fortsätter algoritmen med att se efter vilka noder $\leq k-1$ som kan besökas. Det kan innebära att tidigare obesökta noder nu kan besökas, men det kan också leda till alternativa vägar till noder som redan besökts.

Hur mycket kommer i värsta fall att lagras i k? Till att börja med en väg per kombination av noder som kan besökas i området. Det kommer också att lagras vägar som når k i senare iterationer. Figure 4.9 visar två intressanta intervall. I syfte att få en begränsning av antalet vägar som lagras i k uppskattas områdenas storlekar med värstafallet. **A** är som störst då $p = n$. Detta ger även en övre uppskattning för **B** då de båda beräknas på samma sätt. Båda områdena kommer då att vara mindre än $0.1n$, eftersom $t = 1.1$. Så sammantaget kommer de att ha en bredd mindre än $0.2n$ i varje steg. Det finns då högst $2^{0.2n}$ olika vägar till k som behöver sparas. Det innebär att inte all information om en väg lagras utan endast vilka noder som ingår. Då vägen behövs beräknas den genom Dijkstras algoritm precis som förklarats tidigare i kapitlet.



Figur 4.9: Områdena **A** och **B** som kan innehålla intressant information.

Hur mycket tid kommer att spenderas för att lagra vägar i nod k? För att få en övre gräns betraktas värsta fallet, iteration $n-1$. Det finns färre än n noder som kan nå k. Var och en av dessa har färre än $2^{0.2n}$ vägar till sig. Alltså måste högst $n \cdot 2^{0.2n}$ vägar undersökas. För att undersöka en väg måste den beräknas via Dijkstras algoritm eftersom bara noderna sparades och inte information om vilken ordning de kom i. Detta tar $O(n^2)$ tid. Det är bara spannervägar som lagras så det enda som kan få vägen till k att inte vara en spannerväg är om k orsakar ett problem. Det tar en tid i storleksordningen $O(n)$ att jämföra avstånden från k till alla noder i vägen och på så vis fastställa om det är en spannerväg. Detta ger algoritmen en körtid som är i storleksordningen $O(n^5 2^{0.2n})$. Om stretchkonstanten, t, skrivs om till $t = 1+p$ fås $O(n^5 2^{2pn})$ eftersom $0.1n$ beräknades genom att se hur

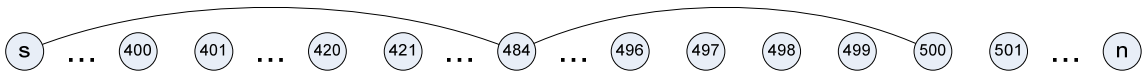
långt stretchen tillät bågar att gå bakåt. Här syns att för $p = 0$ dvs. $t = 1$ blir det en polynomiell lösning.

Eftersom $0.2n$ var en uppskattning uppåt så gäller att komplexiteten kan skrivas $O(n^5 2^{2pn})$ med $p = 0.1 - \varepsilon$ för $\varepsilon > 0$. Det innebär att uttrycket kan skrivas om via $n^5 2^{2pn} = 2^{\log n^5 + 2pn} = 2^{0.2n + (\log n^5 - 2\varepsilon n)}$ som asymptotiskt är av storleken $O(2^{0.2n})$. Det går alltså för stretch 1.1 att hitta en bättre algoritm än den allmänna som var $O(2^{0.822n})$.

4.4 Alternativ inkrementell metod

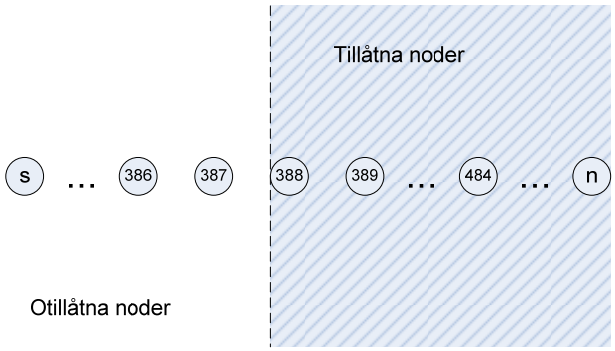
Det finns ett annat sätt att bygga upp den inkrementella lösningen. Den här metoden fokuserar på vilka områden som är tillåtna efter att en nod besökts. Den alternativa metoden åskådliggörs bäst med ett exempel.

Exempel I det här exemplet används stretch 1.5. Figur 4.10 visar hur det kan se ut i ett typiskt steg av algoritmen.



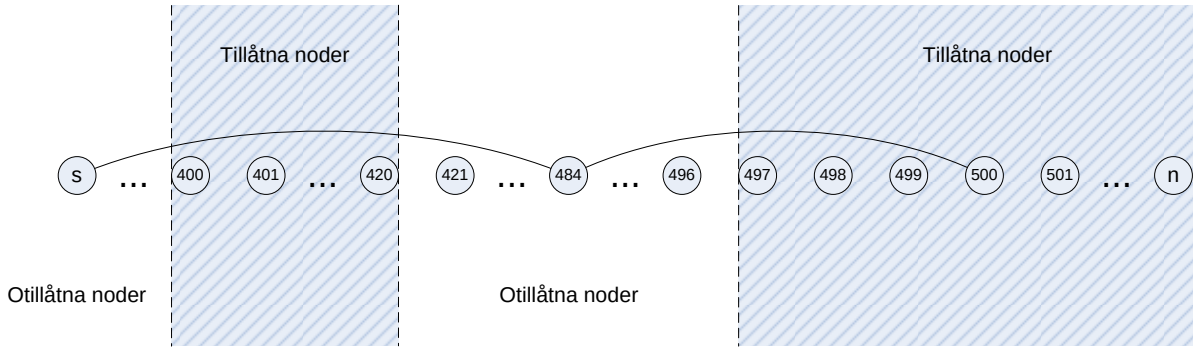
Figur 4.10: En väg till nod 500.

Här upptäcks i iteration 500 att det finns en väg till nod 500 via nod 484. Figuren visar hela vägen från s till 500. Den kompletta vägen kommer aldrig att behandlas av algoritmen som bara tittar på om bågen (484,500) får användas. I nod 484 lagras en bild per spannerväg som når noden. Bilderna visar vilka noder som är tillåtna att besöka efter nod 484. Bilden för nod 484 i exemplet syns i figur 4.11.



Figur 4.11: Bild av tillåtna noder i 484.

Alla noder i det streckade intervallet kan besökas efter nod 484. Här är det viktigt att komma ihåg att bilden bara gäller för en väg till nod 484. Det kan finnas andra vägar som ger upphov till andra bilder. En liknande bild skapas för nod 500. Den utgår från den tidigare bilden, för 484, men kring varje besökt nod kommer det tillåtna intervallen att eventuellt påverkas. Figur 4.12 visar hur ett otillåtet intervall växer fram runt 484 efter att 500 besökts.



Figur 4.12: Bild av tillåtna noder i 500 då exempelvägen följts.

Algoritmen följer exemplet, för varje nod och väg sparar den en bild av grafen med de områden som är tillåtna att gå till markerade. Om den i steg k hittar en väg från p till k kommer den direkt utifrån bilderna i p att kunna avgöra dels om k är tillåten och om så är fallet konstruera en ny bild för k. Detta betyder att den i varje steg hittar alla bilder för den aktuella noden. Istället för att titta på kombinationer av noder i vägen tittar den istället på kombinationer av tillåtna områden i bilden. På liknande sätt som förut så kommer den efter att en väg till k hittats att rekursivt undersöka om nya vägar till noder till vänster om k kan konstrueras. Invarianten för algoritmen blir "i iteration k har alla bilder för noder $\leq k$ vars vägar bara använder noder $\leq k$ hittats".

Det är uppenbart att algoritmen hittar en väg om den finns och på samma sätt som förut går det att få fram vägen genom att köra algoritmen upprepade gånger och varje gång ta bort en nod samt alla bågar till den. Om det fortfarande finns en väg behövs inte noden, annars sätts den tillbaka och en annan nod testas. Till slut finns bara precis de noder som ingår i en väg kvar och Dijkstras metod kan användas för att få fram den kortaste vägen.

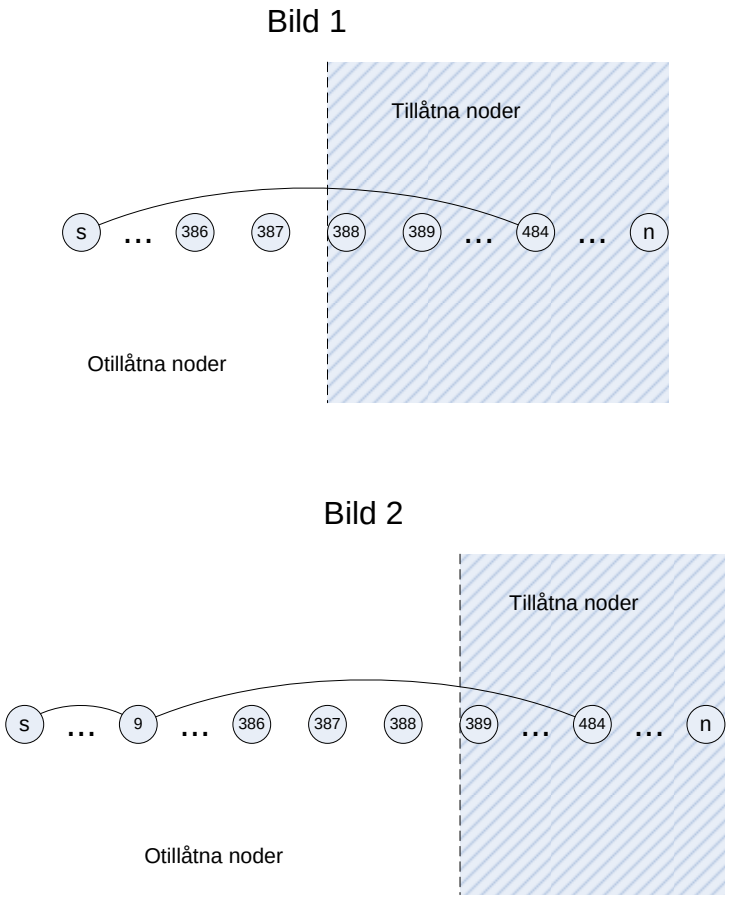
En undersökning av tidskomplexiteten är något mer avancerad än för tidigare algoritmer. Den centrala delen är antalet bilder som kan förekomma i en nod. Detta undersöks efter att ett antal begrepp retts ut.

Det kan hända att flera vägar till en nod ger upphov till liknande bilder. Det är då av intresse att bara spara den bästa under förutsättning att bilderna är jämförbara.

Definition 4.4.1 *En* **bild** *är en mängd intervall som anger vilka noder som är tillåtna att besöka.*

Definition 4.4.2 *Begreppet* **bildkvalité** *används för att jämföra olika bilder. En bild är av bättre kvalité än en annan om dess tillåtna intervall täcker samtliga intervall i den andra bilden.*

Exempel Figur 4.13 visar två jämförbara bilder. Här syns att på grund av att vägen i bild 2 går via nod 9 kommer det att ske en inskränkning av de tillåtna noderna. Alltså är bild 1 bättre än bild 2 och endast bild 1 kommer att lagras i iteration 484.

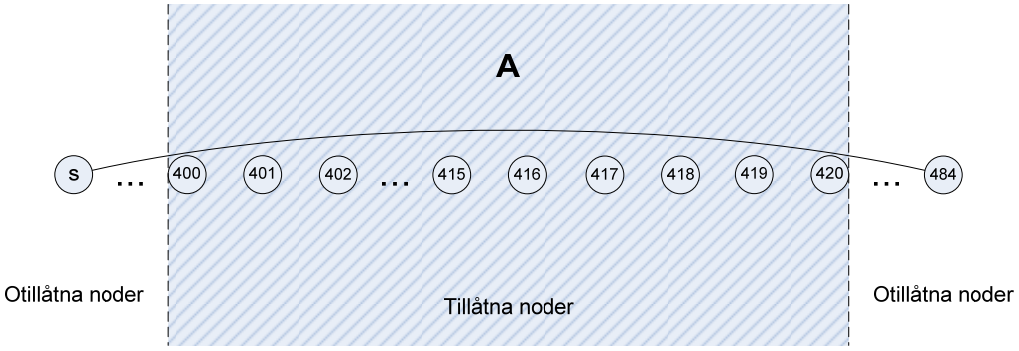


Figur 4.13: Kvalitetsjämförelse mellan två bilder.

Att jämföra två bilder för att bestämma vilken som är bäst tar för en naiv algoritm en tid i storleksordningen $O(n)$ eftersom intervallen kan spänna över denna

bredd. Men det är möjligt att få ner denna tid till $O(\log n)$ genom att bara lagra intervallens ändpunkter. Det finns enligt vad som bevisas senare nämligen högst $O(\log n)$ intervall som behöver jämföras.

En undersökning av hur intervallen som bygger upp bilderna ser ut görs nu? Det tidigare exemplet undersöks med fokus på det vänstra intervallet. Figur 4.14 visar det vänstra intervallet i uppförstoring.

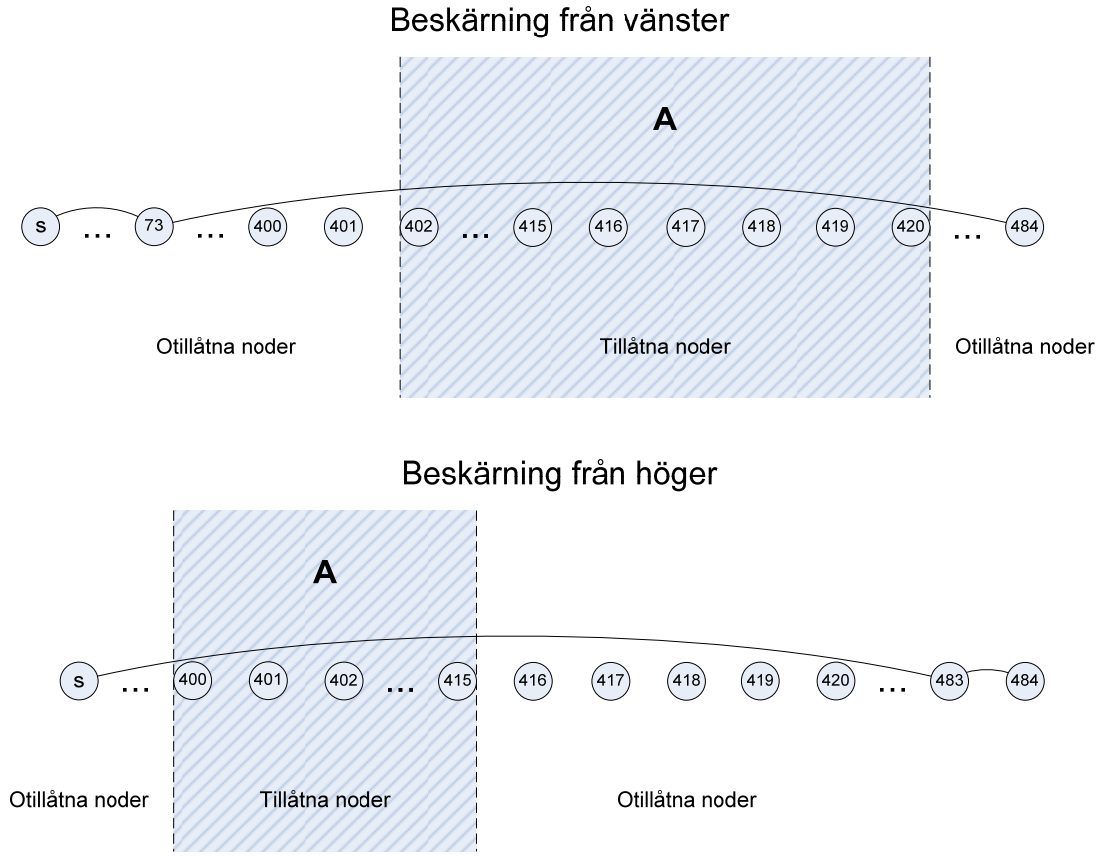


Figur 4.14: Intervallet **A** är en uppförstoring av det vänstra intervallet i figur 4.12.

Figur 4.15 visar hur intervallets gränser kan påverkas. Noder som besöks till vänster om **A** kommer att påverka den vänstra gränsen av **A** medan noder som besöks till höger om **A** påverkar främst den högra gränsen. De påverkar den högra gränsen med sin närhet och den vänstra genom att de förlänger vägen. Figur 4.15 visar strax att det är omöjligt att få 416-419 att utgöra gränsen. Den granulariteten går inte att få. Då det är en värstafallsberäkning som görs kommer en förenkling att göras, nämligen att gränserna för intervallet kan vara vilka noder som helst. Detta innebär en förenkling men kommer att underlätta då en övre gräns eftersöks.

På grund av att bättre bilder ersätter sämre kommer det i värsta fall för ett intervall av längd n att finnas n olika delintervall att spara. Största mängden olika intervall som inte överlappar är nämligen mängden av alla intervall av längd 1.

Figur 4.11 visar ett exempel på ett maximalt tillåtet intervall. Detta kan sedan delas upp i delintervall om vägen till 484 går vi andra noder. Av figur 4.15 framgår hur intervallet kan skäras till från vänster genom att en nod till vänster om intervallet besökts. Intervallet kan också skäras av till vänster på grund av att vägen till 484 har varit till höger om 484 innan den nås. Figur 4.12 visar vad som händer om en nod inuti intervallet besöks innan den nod som sparar bilden nås. Här syns att om andra noder i intervallet besöks längs vägen till 500, som t.ex. figur 4.12 visar, kommer det att leda till en uppdelning i 2 intervall. Det finns en brytgräns för att på det här sättet få två intervall, om en mellannod ligger för långt till vänster kommer inget vänsterintervall, **A**, att uppstå. Återstående intervall som ligger på höger sida kan betraktas som ett helt nytt intervall som på nytt kan delas upp i 2



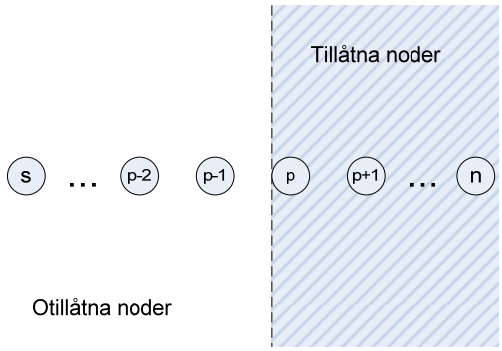
Figur 4.15: Beskärning av intervallet **A**.

delar om dess brytgräns passeras innan nästa mellannod besöks. På det här sättet kommer den bild som ryms i slutnoden, t.ex. 500 att kunna byggas upp. Antalet intervall kommer att asymptotiskt bli logaritmen av intervallets längd medan antalet kombinationer per intervall kommer att bli intervallets längd.

Nu görs en kraftig förenkling för att få en uppskattning av antalet bilder i en nod. Nod n betraktas då flest bilder till vänster om noden finns för den. (Egentligen nod $n-2$ då numreringen går från 0 till $n-1$ som är målnoden, men att använda n ger lättare beräkningar och det är en övre gräns som söks.) Längden av första intervallet för ett $(1+t)$-spannervillkor med $(0 < t < 1)$ uppskattas nu. Antag att situationen är den som visas i figur 4.16. Spannervillkoret innebär att vägen $s \to n \to p$ inte får vara längre än $(1+t)$ gånger vägen $s \to p$. Det ger ekvation 4.7.

$$n + (n - p) \leq (1 + t)p \Leftrightarrow p \geq 2n/(2 + t) \tag{4.7}$$

Alltså är maxbredden på intervallet $n - 2n/(2+t) = nt/(2+t)$.



Figur 4.16: Noder som ingår i beräkningen.

När det gäller längden av delintervallen så kommer det direkt att bero på antalet delintervall. Det ger en komplicerad optimering och i syfte att få en övre gräns kommer här enklast möjliga uppskattning att göras. Längden av alla intervall sätts till maxlängden. Antalet intervall sätts till maximalt antal vilket uppnås då en nod precis på brytgränsen besöks för varje delintervall.

Antalet delintervall är lika med antalet gånger det går att finna en brytgräns. Detta är antalet gånger det går att substituera intervall längden in i rekursionsekvationen 4.8 innan värdet blir mindre än 1, då det inte går att dela upp igen.

$$\left.\begin{array}{l} n_i = \dfrac{tn_{i-1}}{2+t} \\ n_0 = n \end{array}\right\} \quad \Rightarrow \quad n_i = \left(\frac{t}{2+t}\right)^i n \tag{4.8}$$

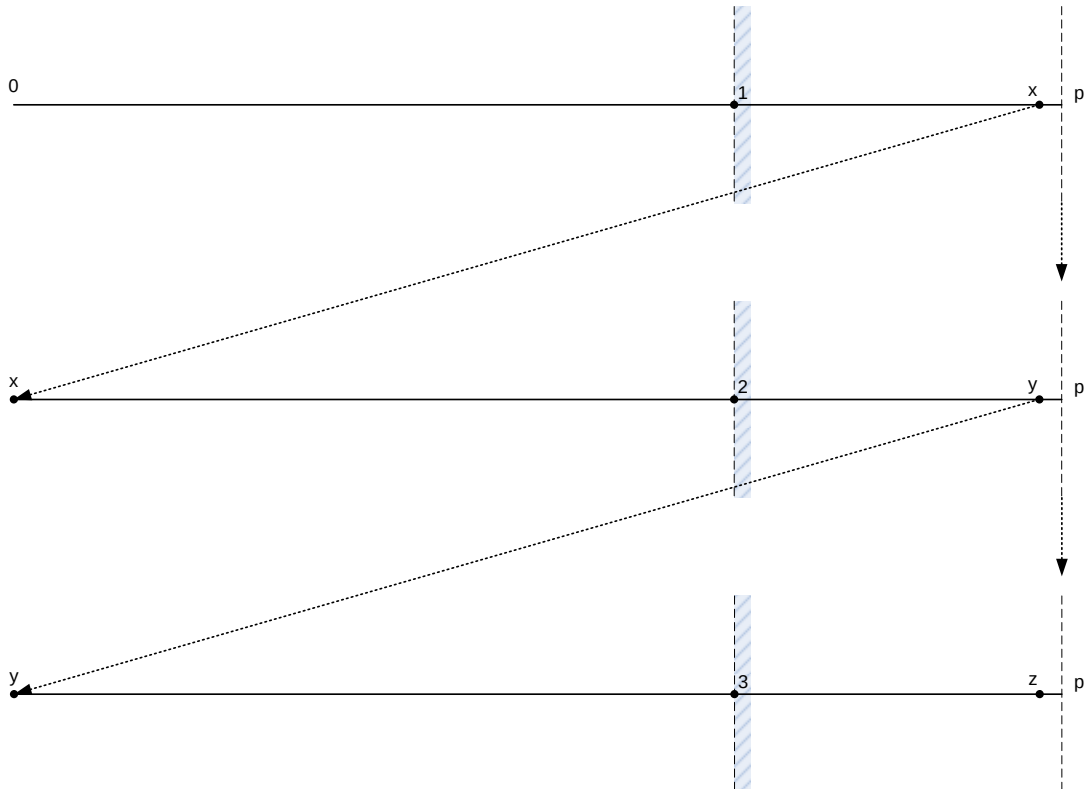
Vad som eftersöks är alltså för vilket i detta blir < 1. Lösningen ges av $i = \frac{t}{2+t}\log\frac{1}{n}$. En övre uppskattning är att vart och ett av dessa i intervall ger upphov till $nt/(2+t)$ olika delintervall. Det totala antalet olika bilder i nod n ges då av $(nt/(2+t))^i$. Låt $a = (2+t)/t$, då kan uttrycket förenklas enligt 4.9.

$$\left(\frac{nt}{2+t}\right)^i = \left(\frac{n}{a}\right)^{\frac{1}{a}\log\frac{1}{n}} = \frac{1}{n}\cdot n^{\frac{1}{a}\log\frac{1}{n}} = \frac{1}{n}\cdot n^{\log n/\log a} \tag{4.9}$$

Så sammantaget gäller att antalet bilder i en nod alltid understiger $\frac{1}{n}\cdot n^{\log n/\log a} = O(2^{\frac{(\log n)^2}{\log a}})$. Ett exempel åskådliggör hur de många kombinationerna kan uppstå.

Exempel Låt t vara 0.5, dvs. stretchen är 1.5. Nu skapas en graf där 5% av det maximala intervallet som ligger till vänster om p tillåts vara besökbart i varje iteration. Detta område är streckat i figur 4.17. I första iterationen visar x var en punkt bör ligga för att skapa det streckade intervallet. För att få maximalt antal

kombinationer bör en eller flera noder i närheten av x vara besökta längs vägen till p. Om de ligger till höger om x och besöks i riktning mot p kommer de inte att inskränka något av intervallen som skapas. De får dock inte ligga för nära de intervall som skapas i senare iterationer eftersom de då blockerar dessa. x hamnar 96% av vägen mot p. I nästa iteration hamnar y som är nästa punkt på nytt 96% av den resterande vägen mot p. I figuren har detta nya intervall zoomats in till samma storlek som det första. Detta kan fortgå tills intervallet är så litet att den itererade punkten sammanfaller med p, dvs. tills de 4% är mindre än 1. Intervallets längd är då ungefär 25.



Figur 4.17: Exempel på kombinatorisk tillväxt av antalet vägar.

Antalet iterationer beräknas enligt 4.10.

$$(0.04^i) \cdot p < 1 \Leftrightarrow 0.04^i < \frac{1}{p} \Leftrightarrow i < 0.215 \log(p) \leq \frac{\log(1/p)}{\log(0.04)} \tag{4.10}$$

Intervallängden är 5% av det maximala intervallet som i sin tur är 25% av totala intervallet. Alltså är längden $0.0125p$ i första iterationen. Sedan minskar det i varje iteration och i iteration k är det streckade intervallet $0.0125p \cdot (0.04^k)$.

Sammantaget fås att antalet kombinationer är minst så många som uttryck 4.11 visar.

$$\begin{aligned}\prod_{k=0}^{i-1}(0.0125p \cdot 0.04^k) = (0.0125p)^i \cdot 0.04^{\frac{i(i-1)}{2}} = \\ = 2^{i\log 0.0125 + i\log p + \frac{i(i-1)}{2}\log 0.04} \geq \\ \geq 2^{-1.40\log p + 0.215(\log p)^2 - 2.34i(i-1)} \geq \\ \geq 2^{0.215(\log p)^2 - 1.40\log p - 2.34i^2} \geq \\ \geq 2^{0.215(\log p)^2 - 1.40\log p - 0.12(\log p)^2} \geq \\ \geq 2^{0.201(\log p)^2 - 1.40\log p} \geq 2^{0.2\log p} = \Omega(2^{0.2(\log p)^2})\end{aligned} \tag{4.11}$$

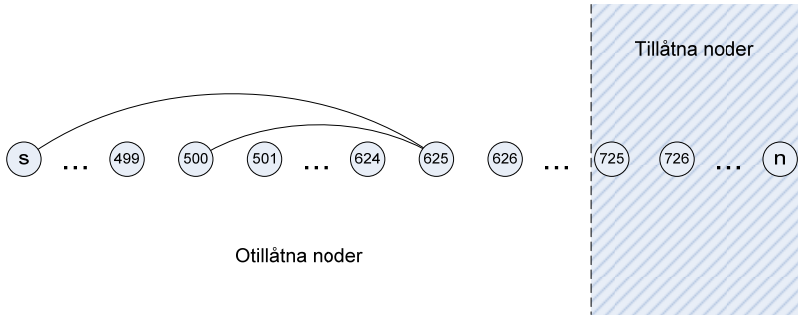
Här uppskattas i uppåt med $0.22\log p$ och nedåt med $0.215\log p$ och uppskattningarna gäller för stora p.

Det är kanske inte troligt att det finns en graf som tillåter alla dessa möjligheter. Men om bara en procent av dessa finns innebär det att den asymptotiska komplexiteten för att lösa problemet med kända metoder blir $\Omega(2^{0.2(\log n)^2})$.

Det gav en uppskattning om antalet bilder, men bara då bilderna innehöll intervall till vänster om p. Även intervall till höger om p kommer att ingå i de bilder som finns i p. Att vägen någon gång innan den når p har varit till höger om p ger inga nya tillåtna intervall till vänster om p. Dessa intervall kan bara påverkas så att de blir färre eller krymper. Alltså gäller att ovanstående beräkning fortfarande ger en över gräns för antalet bilder i p med avseende på den vänstra sidan. Hur många bilder tillkommer då vägen tillåts att även besöka noder till höger om p?

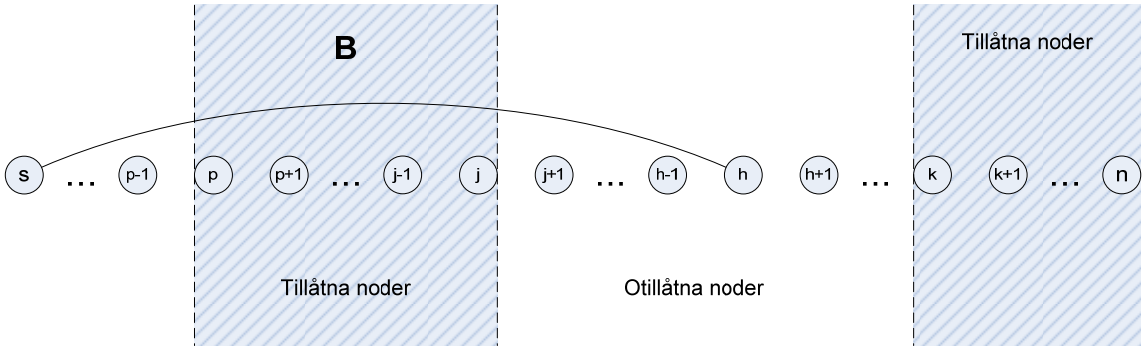
Till att börja med finns det en nod som är så långt till höger som det är möjligt för vägen att gå, då p sedan skall besökas. Denna nod, h, finns på position $(1+t/2)p$, som visas av ekvation 4.12. Figur 4.18 visar ett exempel runt nod 500 med $t = 1.5$. Då blir h nod 625 och efter att nod 500 besökts måste vägen färdas ett långt stycke innan det igen går att besöka noder utan att bryta mot spannervillkoret.

$$h + (h - p) \leq (1 + t)p \quad \Leftrightarrow \quad h \leq (1 + t/2)p \tag{4.12}$$



Figur 4.18: Exempel på väg till höger om 500.

Störst antal intervall till höger om p fås då h besöks först. Efter att h besökts blir situationen densamma som tidigare med den enda skillnaden att riktningen är omvänd. Intervallet (p,h) kommer att delas upp i högst två delar för varje nod vägen besöker. Det är givetvis möjligt att vägen passerar p igen, men det kommer inte att leda till nya intervall eller fler kombinationer så detta kan bortses ifrån då en övre uppskattning söks. Figur 4.19 visar den allmänna situationen. Antalet intervall som **B** kan delas upp i påminner om antalet intervall **A** kan delas upp i, den enda skillnaden är att **B** är något mindre från början. En total övre uppskattning på antalet bilder i en nod p fås av att multiplicera antalet möjliga bilder för de båda halvorna. Detta gäller då en väg genom **A** inte behöver inverka på hur **B** kommer att se ut.



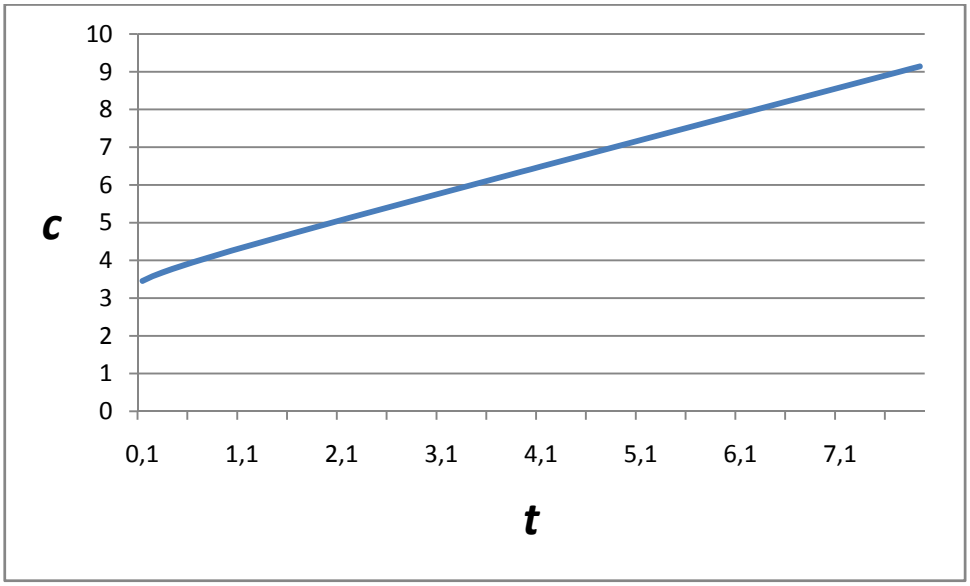
Figur 4.19: Intervallet **B**.

Slutligen gäller att antalet bilder i en nod alltid understiger $(\frac{1}{n}n^{\log n/\log a})^2 = O(2^{\frac{2(\log n)^2}{\log a}})$.

När antalet bilder nu är begränsat upptill kan en övre gräns för algoritmen konstrueras. Om varje nod antas ha maximalt antal bilder så kommer i varje steg $\leq n$ noder att ha vägar till den aktuella noden. Dessa kommer att ha $(\frac{1}{n}n^{\log n/\log a})^2$

bilder var som behöver undersökas och sedan skall en ny bild konstrueras om det går. Att konstruera en ny bild tar $O(\log n)$. Detta beror på att varje tidigare intervall påverkas av den nya bågen till p och därför måste beräknas om samt att det enligt tidigare visat finns ett logaritmiskt antal intervall. Utöver detta införs ett nytt intervall runt den senast besökta noden då ingen begränsning runt denna tidigare fanns. I syfte att förenkla beräkningarna används att $O(\log n)$ även är $O(n)$. När en ny bild beräknas måste avståndet från alla noder till p beräknas. Detta görs genom att köra Dijkstras algoritm på bilden. Detta tar en tid i storleksordningen $O(n^2)$, så bildbyggandet tar $O(n^3)$ totalt.

Totalt används alltså av algoritmen per nod $\leq n \cdot O(n^3) \cdot (\frac{1}{n} n^{\log n/\log a})^2$. I den här tiden ingår bilder som kommer att konstrueras i senare steg, det är en totaluppskattning av hur mycket tid som kan spenderas i en nod. Då det finns n noder blir slutligen algoritmens komplexitet $O(n \cdot n \cdot O(n^3) \cdot ((\frac{1}{n} n^{\log n/\log a})^2 = O(n^5 (\frac{1}{n} n^{\log n/\log a})^2) = O(2^{(3+\frac{2}{\log a})(\log n)^2}) = O(2^{c(\log n)^2})$ där c blir en konstant som bara beror på spannerkonstanten. Figur 4.20 visar hur c beror av t. För små t-värden är detta beroende linjärt. Med stretchkonstant 1.1 blir komplexiteten $O(2^{3.46(\log n)^2})$ vilket är klart bättre än den tidigare algoritmens $O(2^{0.2n})$.



Figur 4.20: c som funktion av t.

4.5 Slutsats och vidare arbete

I kapitel 4.4 visar jag att det går att lösa problemet SPANNERVÄG för heltalsgrafer inom en tid av storleksorningen $O(2^{c(\log n)^2})$ där c är en konstant som endast beror

på stretchkonstanten. Detta är det kraftfullaste resultatet i kapitlet som också presenterar en något sämre algoritm i kapitel 4.3. Denna har en exekveringstid av storleksordningen $O(2^{0.2n})$ då stretchkonstanten är 1.1. I kapitel 4.2 visar jag dessutom att problemet SPANNERVÄG är lösligt för heltalsgrafer i polynomiell tid för spannerkonstanterna 1 och n^2 (se sats 4.2.1 och 4.2.2). Beviset för sats 4.2.2 bygger på sats 4.2.3 som säger att längsta möjliga vägen i en heltalsgraf är $\leq n^2/2$.

Det kändes naturligt att välja den specialisering som heltalsgrafen utgör för att kunna hitta en effektiv algoritm för problemet SPANNERVÄG. Dels införs en fast minsta längd på bågarna och dels känns det som om restriktionen till en dimension kan ta udden av *NP*-komplettheten. Det är ofta en skillnad i frihetsgrader som ligger bakom att ett problem tar steget från att vara i P till att bli *NP*-komplett. T.ex. är 2SAT i P medan 3SAT som bekant är *NP*-komplett. Denna ökning från två variabler till tre i klausulerna innebär en höjning av komplexiteten.

Resultaten i kapitel 4.2 som säger att problemet är i P för stretch 1 och n^2 gör det troligt att det även ligger där för stretchar mellan dessa värden, t.ex. för stretch 1.5. Detta beror på att det inte är troligt att komplexiteten byter ordning flera gånger i intervallet.

Tyvärr lyckades ingen av algoritmerna som presenterades att visa detta. Med stretch 1.1 fick den första en exekveringstid i $O(2^{0.2n})$ och den andra en exekveringstid i $O(2^{3.46(\log n)^2})$.

Dels kan detta bero på att algoritmerna var för långsamma, men det kan också vara så att de faktiskt kan exekveras i polynomiell tid och det är de hårda uppskattningarna som leder till ett icke-effektivt resultat. Det är i så fall den senare algoritmen som kommer att göra bäst ifrån sig eftersom uppskattningen för denna är bättre än för den förra.

När det gäller algoritmerna så är de ganska lika, den ena kommer ihåg vilka noder som ingår i vägen och den andra vilka som kan ingå framöver. Den avgörande punkten ur komplexitetssynpunkt är att antalet intervall i bilderna är logaritmiskt jämfört med antalet noder i vägarna som är linjärt. Det är detta som leder till att den senare algoritmen lyckas bäst.

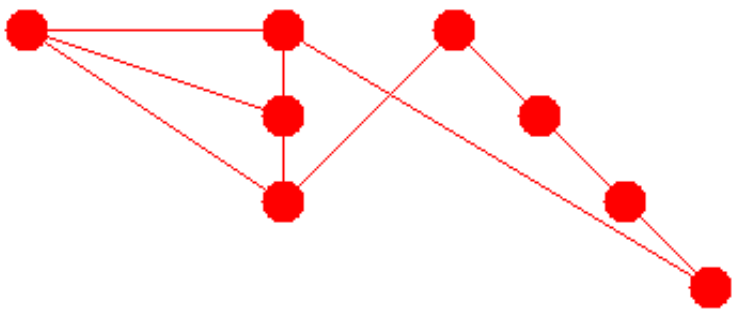
Vidare arbete på det här området skulle kunna bestå i att faktiskt testa algoritmerna på riktiga grafer och se hur de presterar. Det kanske finns andra sätt att försöka lösa problemet på som skulle kunna leda till nya algoritmer. Det skulle också vara intressant att se hur pass resultaten kan utvidgas till fler dimensioner, t.ex. ett rutnät i två dimensioner, eller kanske till att klara endimensionella grafer med fria nodpositioner.

Kapitel 5

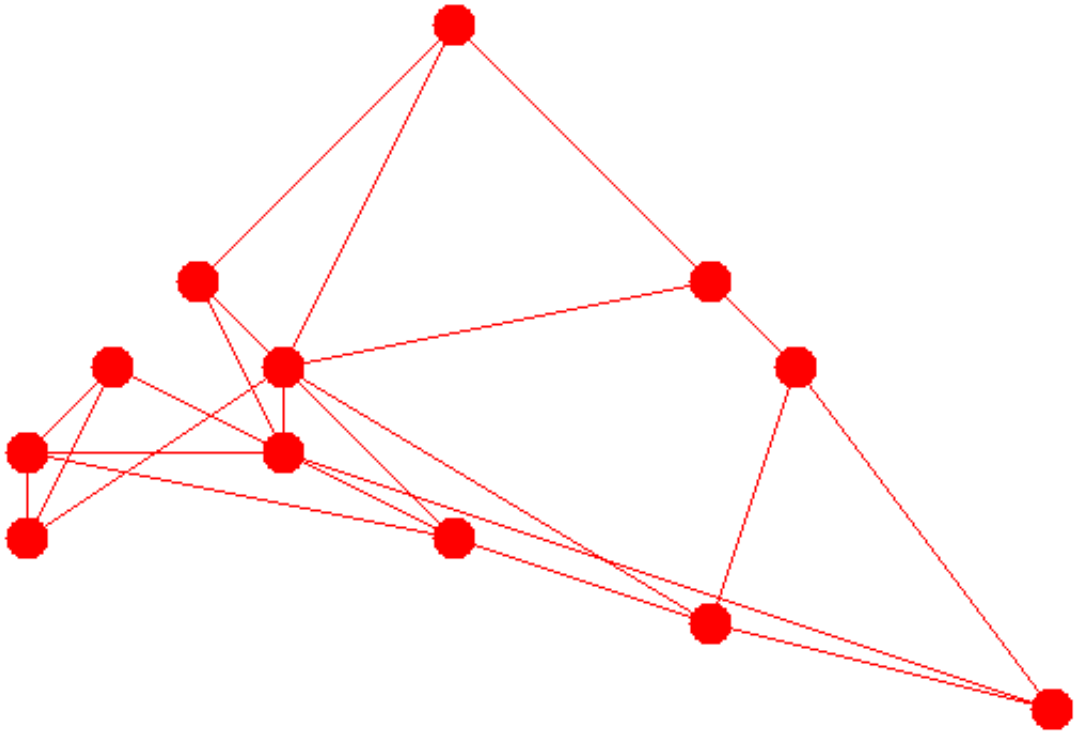
Uppdelning av en graf i spanneröar

5.1 Inledning

Det här kapitlet fokuserar på huvudproblemet, att givet ett t-värde finna minsta möjliga m-värde för vilket grafen är en t-spanner. Det syftar till att ge en djupare förståelse av problemet samt till att producera en heuristikkandidat som kan användas för att lösa problemet så bra som möjligt. Samtliga heuristiker som presenteras är effektiva algoritmer, dvs. de har en polynomiell körtid. Ett verktyg för att generera grafer och köra testheuristikerna utvecklades (se appendix A.6). Bilder som återges i det här kapitlet kommer ifrån verktyget och alla exempel visas utifrån exempelgraferna i figur 5.1 och 5.2. I samtliga exempel antas t-värdet vara 2.0 eftersom detta valts för testserierna som kommer att presenteras längre fram. Det finns inga dolda bågar i exempelgraferna, det innebär att en båge som går in i en nod slutar där, den fortsätter inte igenom.



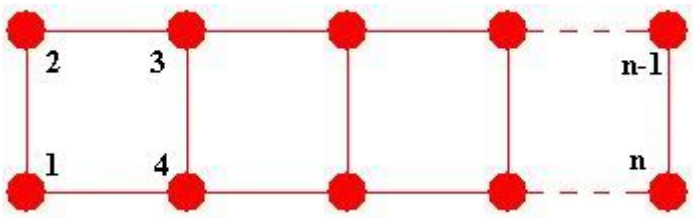
Figur 5.1: Liten exempelgraf.



Figur 5.2: Stor exempelgraf.

5.2 Observation

Att dela upp en graf i öar som är förbundna via så få flygplatser som möjligt är ett svårt problem. Om t-värdet är för lågt kommer antalet flygplatser att bli väldigt stort. Det räcker t.ex. med att t-värdet är $\sqrt{2}$ för att grafen i figur 5.3 skall få många flygplatser, faktiskt n stycken. Å andra sidan så är ett stort t-värde inte troligt om metoderna skall fungera för vägnät där avstånden oftast inte är alltför långa längs vägarna jämfört med fågelvägen. Därför är 2.0 ett bra val för de undersökningar som följer.



Figur 5.3: Värstafall i spanneruppdelning.

Om grafen i figur 5.3 skall delas upp i $\sqrt{2}$-spanners så kommer nod 1 att antingen vara i samma spanner som nod 2 eller nod 4. Nod 1 kan inte vara i samma spanner som nod 3 för $t < \sqrt{2}$ då det geometriska avståndet är $\sqrt{2}$ medan det inom grafen är 2. Så inga noder längs diagonaler kan vara i samma spanners. Möjliga spanneröar blir då endast att göra två öar som täcker övre och undre raden eller $n/2$ öar där varje ö består av ett kolumnpar. I båda fallen kommer varje nod att vara kopplad till en nod i en annan spanner vilket innebär att den är en flygplats. En slutsats av detta är för små t-värden, under $\sqrt{2}$ går det inte att komma undan stora värstafall.

5.3 Metod

Ett antal heuristiker presenteras. Dessa angriper problemet på olika sätt. Därefter exekveras de med en testmängd av slumpvis genererade grafer som indata. Resultatet kommer efter statistisk analys att fastställa vilken heuristik som är bäst lämpad att lösa problemet.

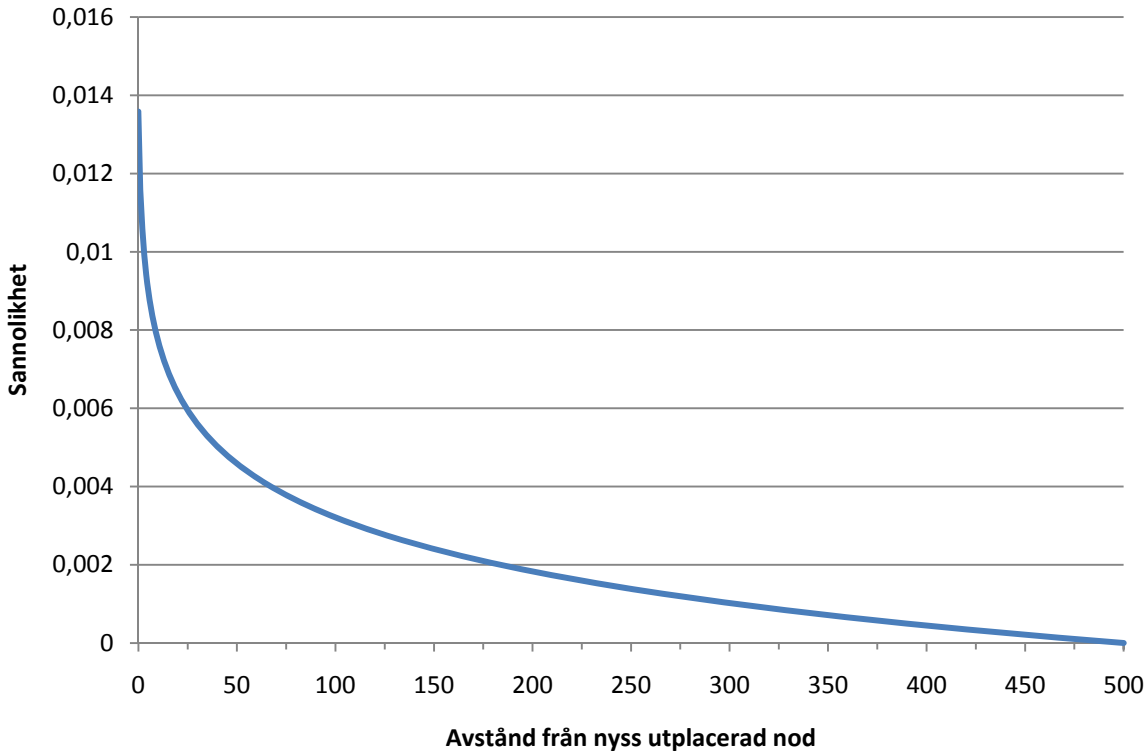
5.3.1 Slumpgrafer

Verktyget, se appendix A.6, konstruerades med två inställningar för att generera noder och två för att generera bågar. Det finns vidare två olika storlekar på det tvådimensionella plan där graferna ritas i verktyget. Den lilla ytan upptar 600x600 pixlar och den stora 1000x1000 pixlar. I syfte att göra grafen åskådlig har ett rutnät skapats av 40x40 stora rutor. Noder skapas endast i dessa skärningspunkter. Detta underlättar när grafen granskas då noder annars delvis kunnat hamna ovanpå varandra.

Noder kan skapas antingen helt slumpvis, eller klustrat. Klustrat innebär att den första noden slumpas ut och därefter slumpas nästa nod ut via en vinkel och ett avstånd från den föregående. Om noden hamnar på en annan nod eller utanför rutnätet görs randomiseringen om. Avståndet från den tidigare noden slumpas först som ett värde från 0-500 och sedan från 0 till det först erhållna värdet. Det ger den fördelning på slumpvärden som figur 5.4 visar. Fördelningen medför att en ny nod med en högre sannolikhet kommer att ligga nära den föregående. Eftersom alla noder sätts ut i rutnätets skärningar kommer de 500 värdena att delas upp i 12 stycken rutor med bredd 40 pixlar. Figur 5.5 visar hur fördelningen blir över rutor. Avrundningen sker nedåt vilket leder till att intervallet 0-39 räknas som ruta 0 vilken inte är tillåten. Hamnar slumpvärdet här tas ett nytt fram. Därför är figur 5.5 normaliserad över de möjliga tillåtna resultaten.

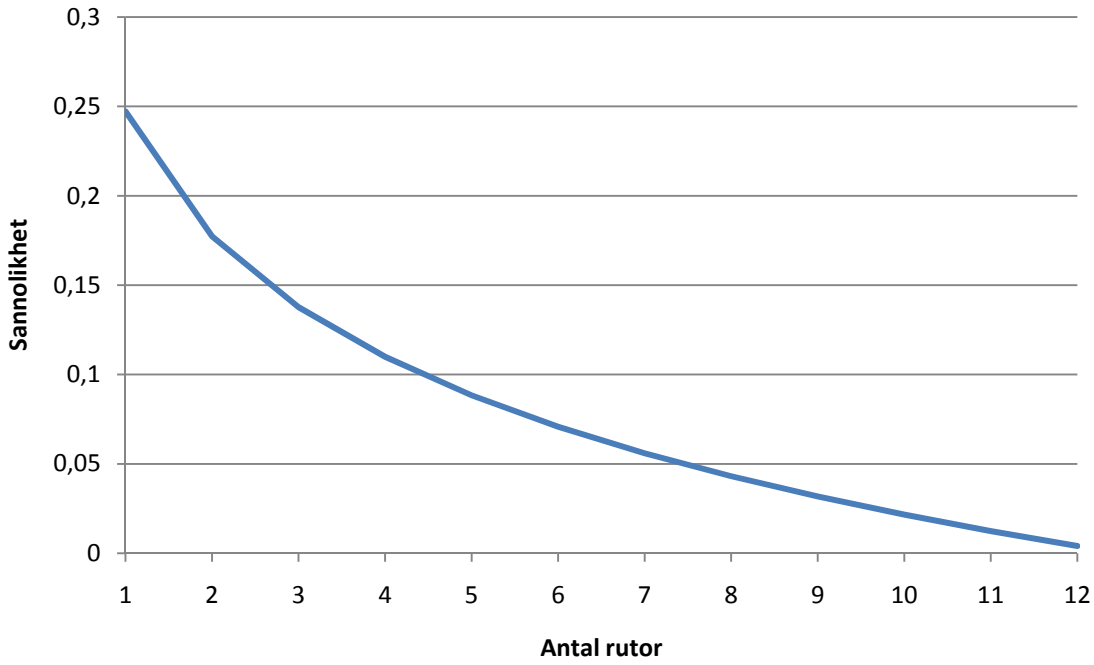
Slumpvisa bågar konstrueras så att först dras en båge från varje nod till en slumpvis annan. Detta minskar risken att få en icke sammanhängande graf. Sedan väljs slumpvis ut två noder som sammankopplas med en båge.

Klustrade bågar genereras till en början precis som de slumpvisa, varje nod länkas till en annan. Sedan väljs en slumpvis nod att dra en båge ifrån. Slutdes-



Figur 5.4: Sannolikhetsfördelning för avståndet till nästa nod.

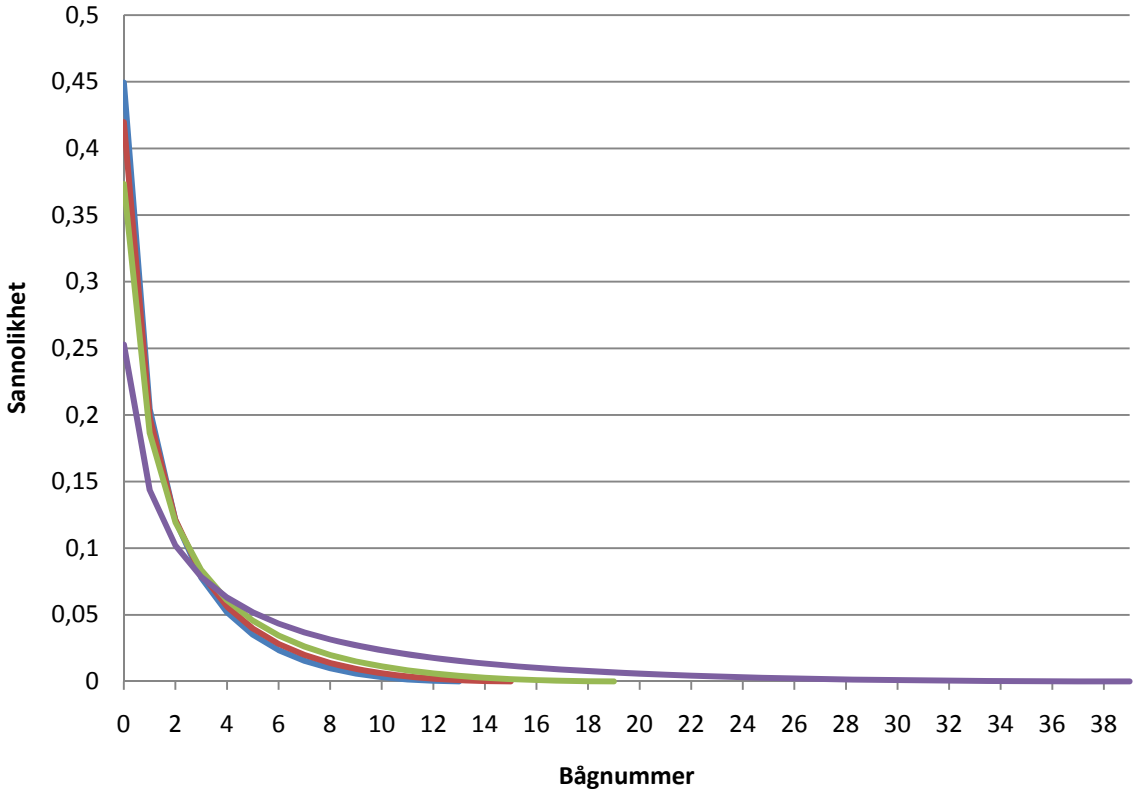
tinationen för bågen beräknas genom att först sortera alla noder i ordning med avseende på avståndet till startnoden. Därefter slumpas tre gånger i följd ut en nod där maxvärdet är föregående resultat. Detta ger en mycket högre sannolikhet att bågen går till en närliggande nod. Figur 5.6 visar hur sannolikheten att välja en båge till en nod längre bort avtar med det relativa avståndet. Figuren visar 4 kurvor för antalet noder 14, 16, 20 och 40 som valts till de olika testmängderna längre fram. De underliggande slumptalen som används är likafördelade.



Figur 5.5: Normaliserat avstånd i rutor (om 40 pixlar).

Figur 5.7 på sidan 70 visar exempel på de fyra typer av slumpgrafer som genereras av programmet. Här följer en numrerad lista på typerna med nodrandomisering följt av bågrandomisering. Numreringen matchar den som finns i figur 5.7.

1. Slumpvis - slumpvis.
2. Klustrat - slumpvis.
3. Slumpvis - klustrat.
4. Klustrat - klustrat.



Figur 5.6: Sannolikhet att välja bågar.

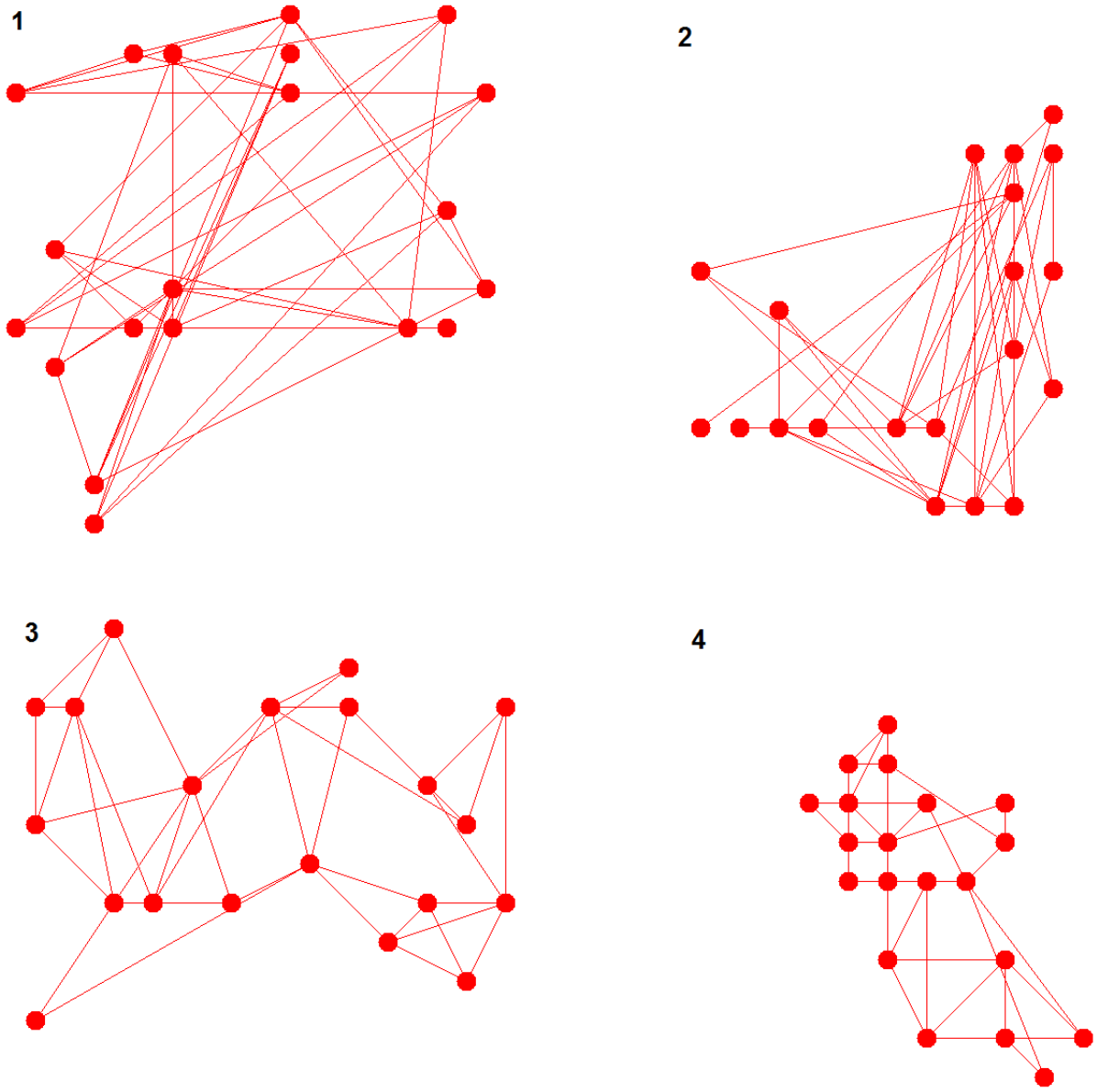
5.3.2 Heuristiker

Begreppet kluster som nämns i vissa heuristiker är en mängd noder som anses höra ihop. De utgör ofta en ö när heuristiken är färdig. Följande definition behövs för att underlätta beskrivningen av heuristikerna.

Definition 5.3.1 *En båge anses vara* **skuggad** *om dess båda ändpunkter är flygplatser (se definition 2.1.8).*

Att skugga en båge innebär att sätta ändpunkterna till flygplatser. Det är det sätt heuristikerna använder då de inte får modifiera grafen genom att ta bort bågar som orsakar problem. Genom att skugga bågen finns det en chans att noderna hamnar på olika öar och ur spanneregenskapssynpunkt är detta detsamma som om bågen var borttagen. Det kan dock vara så att det finns en annan väg mellan noderna som då fortfarande anses ligga på samma ö.

Skuggade bågar avbildas streckade i exempelfigurerna.



Figur 5.7: De olika typer av grafer som kan genereras av grafmjukvaran.

Girigmetoden (M1)

Den här metoden är en typisk girig metod. I varje steg försöker den lägga till den närmaste noden till den ö som håller på att byggas. Steg för steg går det till så här:

1. Välj en startnod för den ö som konstrueras. Den nod som ligger längst ifrån

grafens centrum och inte tidigare valts till någon ö väljs. (Centrum beräknas som medelvärdet av nodernas x- respektive y-koordinater)

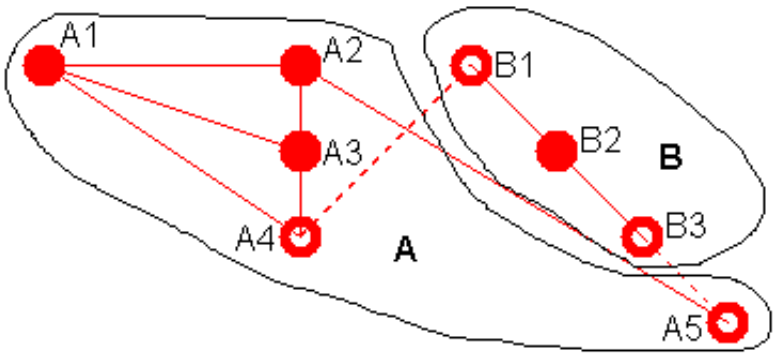
2. Lägg till den nod som har kortast avstånd till en nod på den ö som för tillfället byggs. Noden får inte orsaka att ön bryter mot spannervillkoret.

3. Upprepa 2 så länge det går. När det inte längre går att lägga till noder till ön är den färdig.

4. Låt alla noder som har bågar som lämnar ön bli flygplatser.

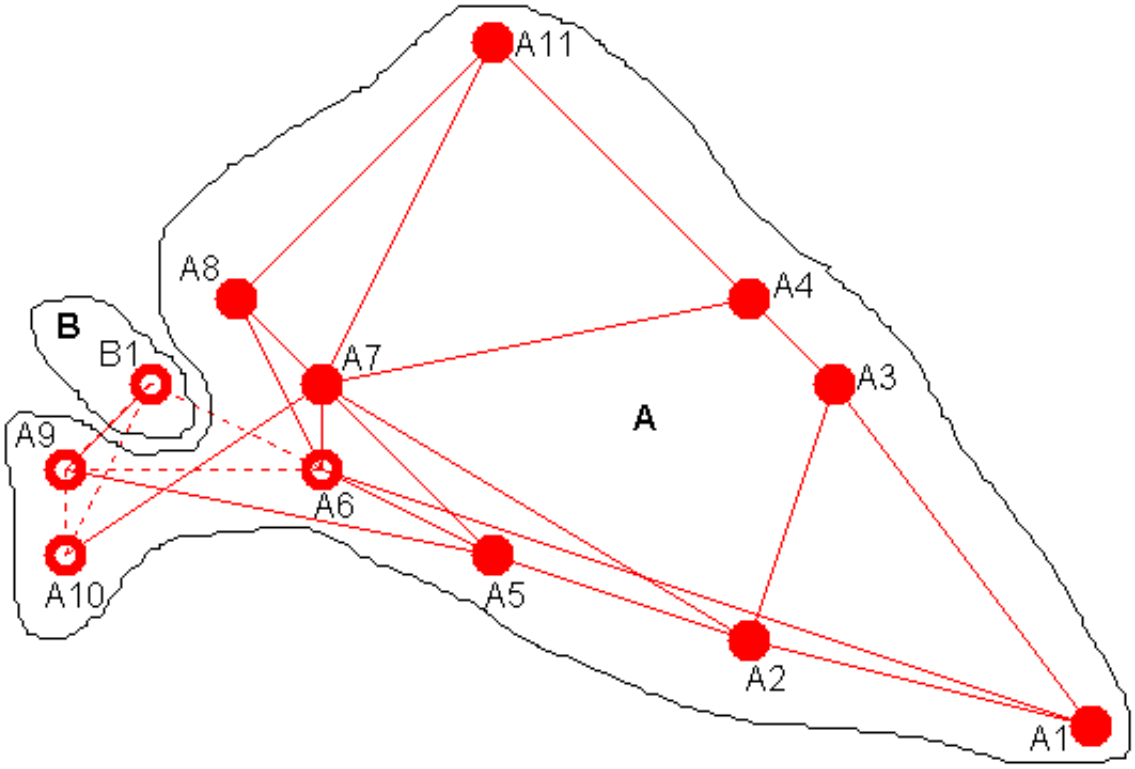
5. Så länge det återstår noder som inte blivit tilldelade en ö, upprepa från 1.

Figur 5.8 visar hur heuristiken arbetar. Den startar med noden längst från centrum, A1. Sedan växer ön **A** via närmsta noden, A2. Därefter läggs A3 och A4 till eftersom det är närmsta noderna. När heuristiken försöker lägga till B1 som nu är närmast går inte detta eftersom det skulle bryta mot spannervillkoret. Istället upptäcks att A5 kan läggas till. Det tar sedan stopp då inte heller noden B3 som gränsar till ön **A** går att lägga till. Observera att heuristiken i varje steg försöker med alla noder som går att nå, så B1 förkastas även i detta steg. Ön är färdig och alla noder som har bågar som lämnar ön, dvs. A4 och A5 blir flygplatser. Sedan fortsätter den med att välja B1 som ny startnod för ön **B** och adderar i tur och ordning B2 och B3 innan alla noder är använda.



Figur 5.8: Exempel på girigmetodens arbetsgång.

Figur 5.9 visar hur heuristiken arbetar med den större exempelgrafen. Den här grafen byggs upp på samma vis från A1 till A11 och sedan skapas en ny ö **B** eftersom B1 inte kan ingå i ön **A**, spannervillkoret bryts till exempel av A8-B1.



Figur 5.9: Exempel på en något större graf.

Diametermetoden x2 (M2a, M2b)

Den här heuristiken bygger en ö åt gången och medan den bygger försöker den hålla nere diametern på ön. Det finns två olika heuristiker som mäter diametern på olika sätt. Den ena beräknar diametern som det euklidiska avståndet mellan de två noder som ligger längst ifrån varandra (M2a). Den andra beräknar istället diametern till den längsta vägen inom ön mellan två noder (M2b). Det är väldigt liten skillnad i praktiken på dessa sätt att mäta och heuristikerna räknas därför som samma.

Steg för steg går det till så här:

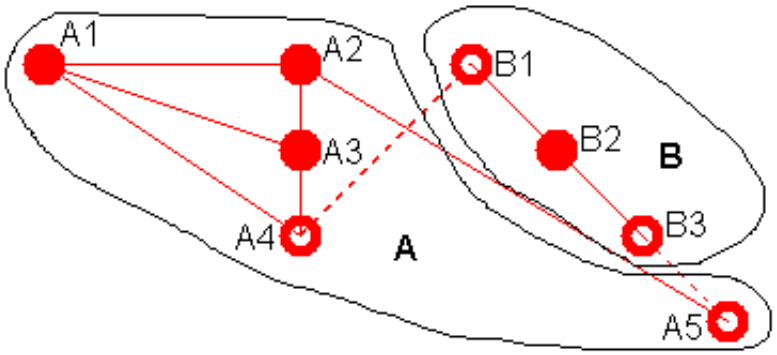
1. Välj en startnod för den ö som konstrueras. Den nod som ligger längst ifrån grafens centrum och inte tidigare valts till någon ö väljs. (Centrum beräknas som medelvärdet av nodernas x- respektive y-koordinater)

2. Lägg till den nod som ger minst diameter på den växande ön. Noden får inte orsaka att ön bryter mot spannervillkoret.

3. Upprepa 2 så länge det går. När det inte längre går att lägga till noder till ön är den färdig.

4. Låt alla noder som har bågar som lämnar ön bli flygplatser.

5. Så länge det återstår noder som inte blivit tilldelade en ö, upprepa från 1.

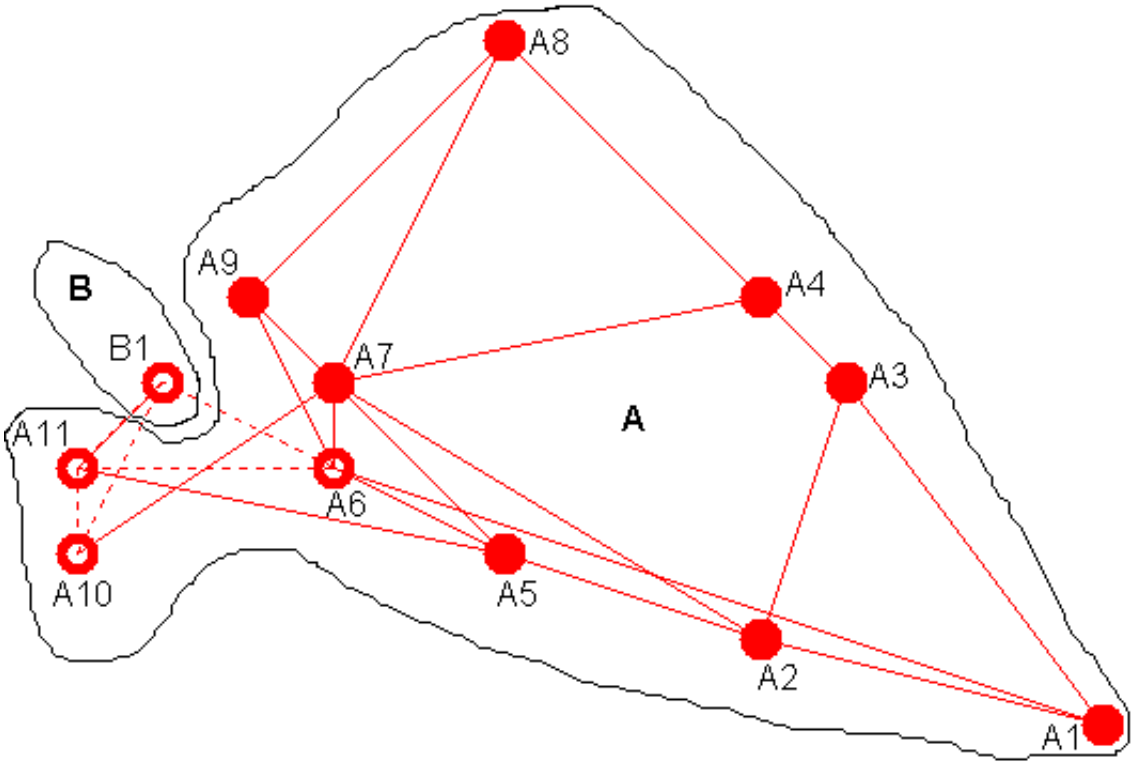


Figur 5.10: Exempel på arbetsgången för diametermetoden.

Figur 5.10 visar hur heuristiken arbetar med den lilla exempelgrafen. Den startar med noden längst från centrum, A1. Sedan växer ön **A** via närmsta noden, A2 som ger minst diameter på den växande ön. Därefter läggs A3 och A4 till då de ger minst diameterökning för ön. När heuristiken försöker lägga till B1 som nu hade givit minst diameter går inte detta eftersom det skulle bryta mot spannervillkoret. Istället upptäcks att A5 kan läggas till. Det tar sedan stopp då inte heller noden B3 som gränsar till ön **A** går att lägga till. Observera att heuristiken i varje steg försöker med alla noder som går att nå, så B1 förkastas även i detta steg. Ön är färdig och alla noder som har bågar som lämnar ön, dvs. A4 och A5 blir flygplatser. Sedan fortsätter den med att välja B1 som ny startnod för ön **B** och adderar i tur och ordning B2 och B3 innan alla noder är använda.

Med den här exempelgrafen skiljer sig inte de två olika sätten att mäta diameter. Heuristiken väljer dessutom exakt samma noder i samma ordning som girigheuristiken.

Figur 5.11 visar ett större exempel. Här syns en skillnad jämfört med girigheuristiken. A8 väljs framför A9 då A8 ger en mindre diameter på den växande ön. I det här steget valde girigheuristiken tvärt om då A9 ligger närmare de tidigare noderna än A8.



Figur 5.11: Exempel på arbetsgången för diametermetoden på en större graf.

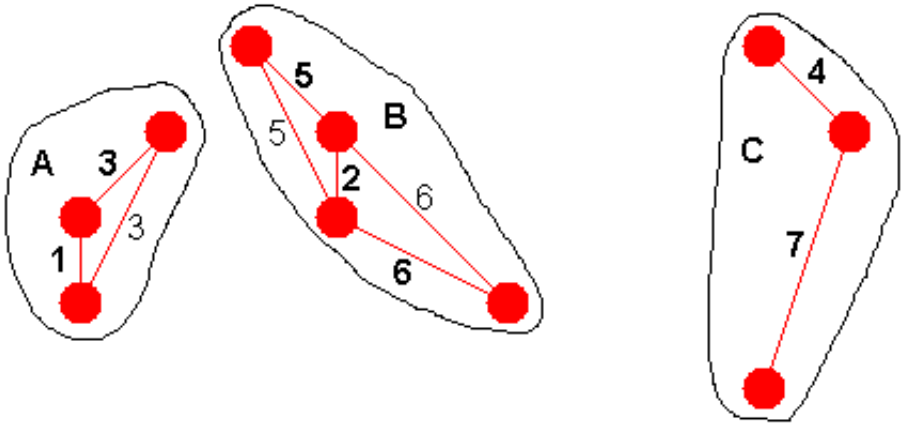
Enkelklustermetoden (M3)

Den här metoden bygger på *single-link clustering* idén. Det innebär att den i varje steg slår ihop de öar som ligger närmst till en större ö. Skillnaden mot girigmetoden är att den arbetar globalt, dvs. med alla öar samtidigt.

Steg för steg går det till så här:

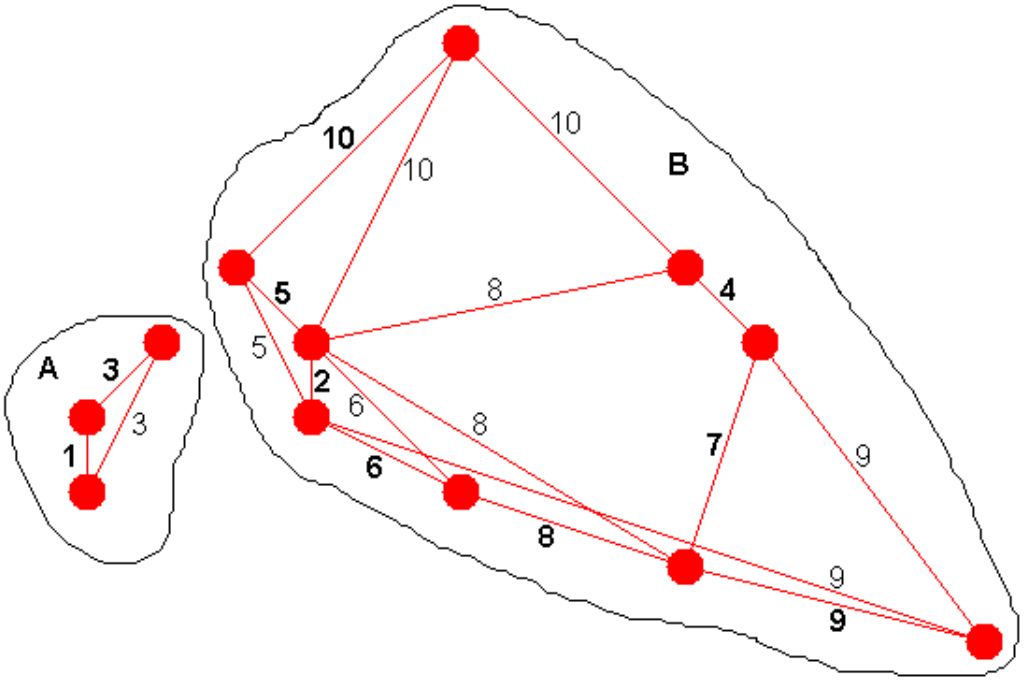
1. Alla noder betraktas som kluster bestående av endast en nod.
2. Mät avstånden mellan alla par av kluster. Gå igenom dessa avstånd i ordning där de närmsta behandlas först. Kontrollera om det går att slå ihop två kluster med bibehållen spanneregenskap. I så fall gå till 3, annars fortsätt leta. Om inget klusterpar går att slå samman, gå till 4.
3. Upprepa 2 så länge det går att slå ihop kluster.
4. Inga fler kluster kan slås ihop. De kvarvarande betraktas därför som färdiga öar. Sätt alla noder som har en båge som lämnar ön till flygplatser.

Figur 5.12 visar hur lösningen växer fram. Bågarna har numrerats i den ordning heuristiken väljer dem. Siffran med fetstil anger den valda bågen medan samma siffror i vanlig stil anger bågar som lagts till i samma skede eftersom de förbinder noder

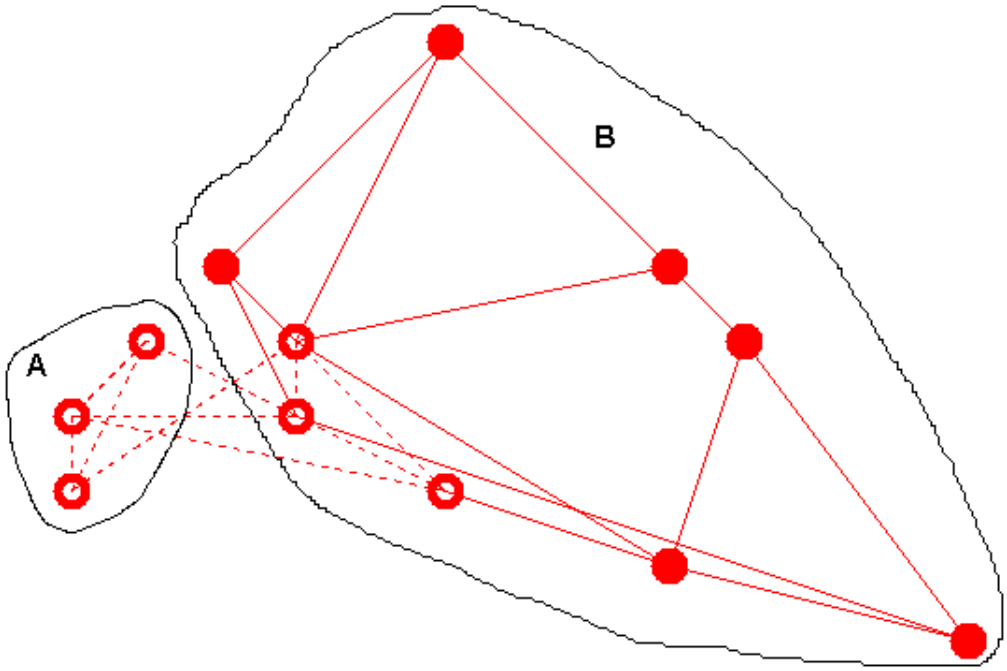


Figur 5.12: Tidigt skede i enkelklustermetoden.

på samma ö. I ett tidigt skede finns det flera öar. Detta är typiskt för heuristiken eftersom den arbetar globalt. Efterhand som fler bågar läggs till binds öarna ihop som figur 5.13 visar. Det slutgiltiga resultatet syns i figur 5.14. I praktiken kan alla noder på ön **A** anses vara egna öar då de endast är förbundna till andra noder via flygplatser. Tolkningen påverkar dock inte resultatet då endast antalet flygplatser mäts.



Figur 5.13: Inga fler kluster kan slås ihop.



Figur 5.14: Slutgiltigt resultat med flygplatser.

Multiklustermetoden x2 (M4a, M4b)

Den här metoden bygger på *complete-link clustering* idén. Det innebär att den i varje steg slår ihop de öar som ger den minsta diametern på den sammanslagna ön. Skillnaden mot diametermetoden är att den arbetar globalt, dvs. med alla öar samtidigt. Även här mäts diametern på två olika sätt. Det ena anger diametern till det euklidiska avståndet mellan de två noder som ligger längst ifrån varandra (M4a). Det andra anger istället diametern till den längsta vägen inom ön mellan två noder (M4b). Det är väldigt liten skillnad i praktiken på dessa sätt att mäta och heuristikerna räknas därför som samma.

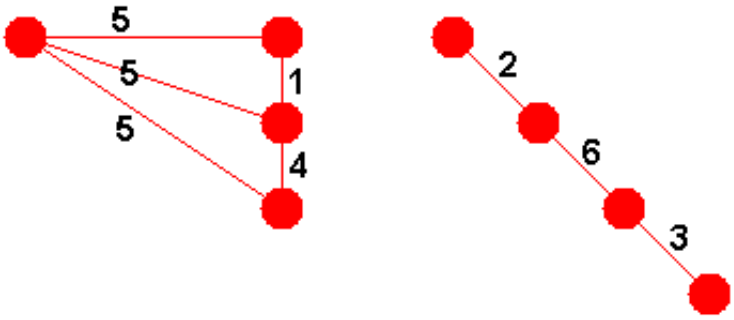
Steg för steg går det till så här:

1. Alla noder betraktas som kluster bestående av endast en nod.

2. Testa att slå ihop alla par av kluster. Förkasta de nya kluster som bildas som bryter mot spannervillkoret. Bland alla nya kluster som bildats och uppfyller spannervillkoret, välj i denna iteration den sammanslagning som har minst diameter. Återställ övriga kluster till den status de hade innan denna iteration påbörjades.

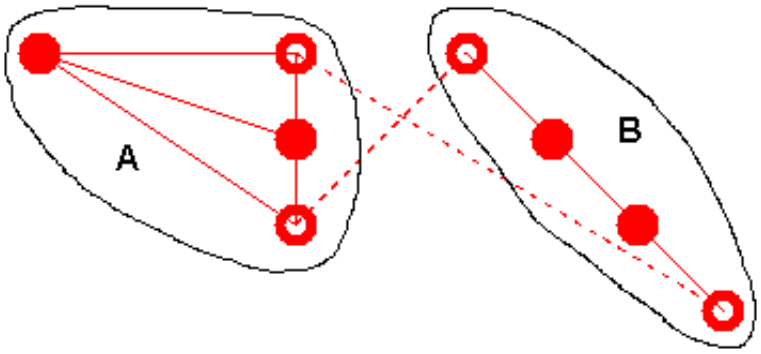
3. Upprepa 2 så länge det går att slå ihop kluster.

4. Inga fler kluster kan slås ihop. De kvarvarande betraktas därför som färdiga öar. Sätt alla noder som har en båge som lämnar ön till flygplatser.



Figur 5.15: Heuristiken har använt alla noder.

Nu följer ett exempel på hur heuristiken arbetar med den lilla exempelgrafen. Figur 5.15 visar hur den väljer kluster att slå samman. Båge 1 är den kortaste bågen som finns och den ger därför minst diameter på det resulterande klustret. På samma sätt väljs båge 2 och 3 och i det här läget består grafen av tre öar med två noder vardera och resten av noderna ligger ensamma på egna öar. Heuristiken hittar sedan båge 4 och 5 som ger lägre diameter än sista bågen, 6. Figur 5.16 visar resultatet med flygplatserna utsatta.



Figur 5.16: Resultatet med flygplatser utsatta.

Flödesmetoden (M5)

Den här heuristiken tar vara på insikten att ett par noder, s och t, som bryter mot spannervillkoret måste hamna på olika öar när heuristiken är färdig. Den försöker därför dela den ö som paret ligger på i två mindre öar. Till sin hjälp använder den en känd algoritm, MAX-FLOW. Maxflödet mellan noderna som skall separeras beräknas. Detta i en graf där varje båge tillåter flödet 1 i båda riktningarna. Resultatet blir en ny graf G där bara bågar som används i flödesberäkningen finns kvar. Att stänga ner flödet till 0 motsvarar att separera noderna till två olika öar.

Minskningen av flödet går till så att heuristiken delar upp G i vägar mellan s och t. Den initierar dessutom sin arbetsgraf till att se ut som ursprungsgrafen. Om flödet är n finns det n disjunkta vägar mellan s och t (samma nod kan ingå i flera vägar, men inte samma båge). Heuristiken angriper dessa en och en och försöker hitta en båge längs vägen som genom att skuggas sänker flödet i arbetsgrafen med ett (skuggning definieras i definition 5.3.1 på sidan 69).

Om ingen båge hittas längs en väg försöker heuristiken att ta bort bågarna i arbetsgrafen en åt gången på riktigt, dvs. genom att modifiera grafen. Detta är ingen tillåten operation, men kan ändå leda till ett bra val då bågen på ett något svagare sätt har möjlighet att påverka flödet. Det finns alltid bågar i flödet som uppfyller detta villkor (om inte så finns det för varje par av noder längs en flödesväg, en väg till vilket hade givit ett större flöde). När bågen väl identifierats återställs arbetsgrafen och bågen skuggas endast då det är förbjudet att modifiera grafen.

Steg för steg går det till så här:

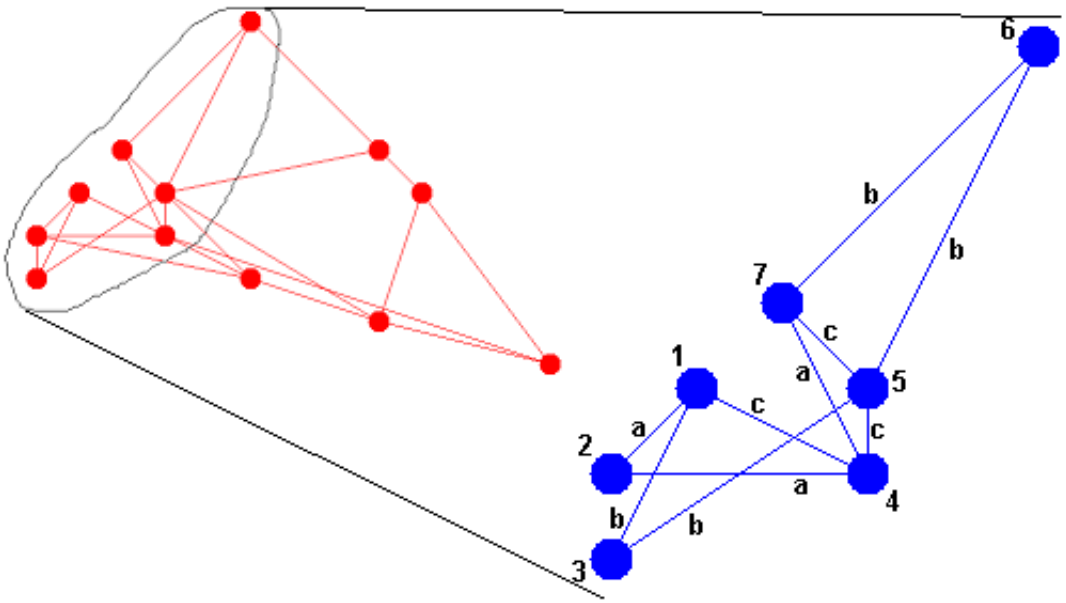
1. Identifiera det par som bryter mest mot spannervillkoret.
2. Sätt upp en flödesgraf där varje båge tillåter flödet 1 i båda riktningarna. Beräkna maxflödet mellan s och t.
3. Använd flödesgrafen för att hitta flödesvägarna mellan s och t. Villkor kontrolleras mot arbetsgrafen, men endast bågarna från flödesvägarna används framöver.

4. Välj en flödesväg. Testa att skugga varje båge, en åt gången. Fortsätt tills en båge hittas vars skuggning minskar flödet i arbetsgrafen. Om ingen båge hittats försök ta bort på riktigt tills en båge hittas. Uppdatera arbetsgrafen genom att skugga denna båge.

5. Upprepa 4 så länge det finns fler flödesvägar.

6. Nu är flödet stängt.

7. Upprepa från 1 så länge grafen inte är en spanner.

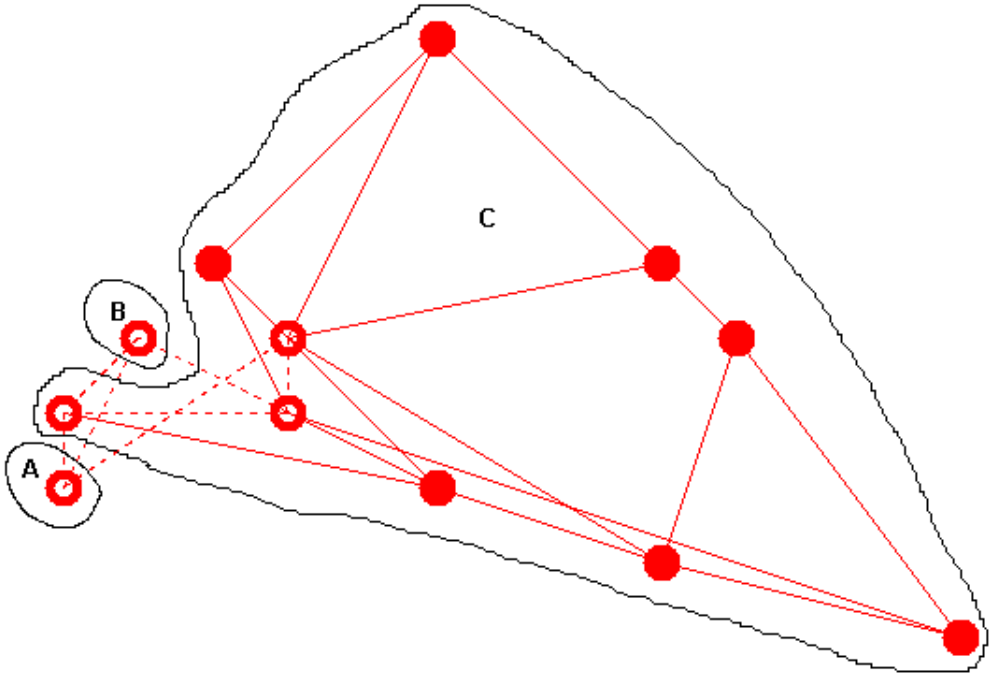


Figur 5.17: Nodparet som bryter mest mot spannervillkoret och flödet där emellan.

Figur 5.17 visar ett exempel på hur heuristiken arbetar med den stora exempelgrafen. Till vänster i figuren syns exempelgrafen och till höger syns en förstoring på det område som innehåller flödet mellan det nodpar, 1-7, som bryter mest mot spannervillkoret.

Mellan noderna 1 och 7 går flödena **a**, **b** och **c**. Flödet **a** går mellan noderna 1-2-4-7, flödet **b** mellan 1-3-5-6-7 samt flödet **c** mellan 1-4-5-7.

Heuristiken bryter flödet **a** genom att ta bort båge 1-2, flödet **b** genom att ta bort 3-5 och slutligen flödet **c** genom att ta bort 1-4. I detalj beror valen på följande: Ingen av bågarna längs flödet **a** kommer vid skuggningar att resultera i att flödet mellan 1 och 7 minskar i arbetsgrafen. Därför faller heuristiken tillbaka på andrahandsvalet och ser att bågen 1-2 minskar flödet om den plockas bort helt. När flödesväg **b** granskas har heuristiken redan skuggat bågen 1-2 i arbetsgrafen. Detta leder till att skuggning av bågen 3-5 minskar flödet i grafen. I och med skuggningen ligger inte 1 och 3 på samma ö längre så flödet kan inte ta den vägen. Detta kan jämföras med när 1-2 skuggas, då finns det en väg 1-4-2 som inte går via två flygplatser så 1 och 2 anses fortfarande ligga på samma ö. Här syns att heuristiken blir bättre ju längre den får köra eftersom den kan utnyttja tidigare val, men när det första valet görs finns ingen möjlighet att samordna och därför kommer tidiga val att bli sämre. Även bågen 1-4 minskar flödet när den skuggas vilket stänger flödet **c**. Figur 5.18 visar slutresultatet.



Figur 5.18: Slutresultat med flygplatser utsatta.

Flödesmetoden med lokaloptimering (M6)

Den här heuristiken fungerar precis som den tidigare, men inför en ny optimeringspunkt mellan punkterna 6 och 7 i heuristikbeskrivningen för M5.

Optimeringen utgår från givna flödesvägar och tidigare givna val som bryter dessa. Den försöker sedan hitta bättre val längs varje väg. I föregående heuristik var det svårt att hitta en bra första båge att bryta eftersom inga andra vägar var stängda. Det fanns troligen ett flöde runt den valda bågen. När nu heuristiken har kapat av alla flödesvägar kanske det går att hitta lokala optimeringar. Så genom att byta en vald båge mot en annan kanske flödet fortfarande är spärrat, men med färre antal flygplatser som resultat.

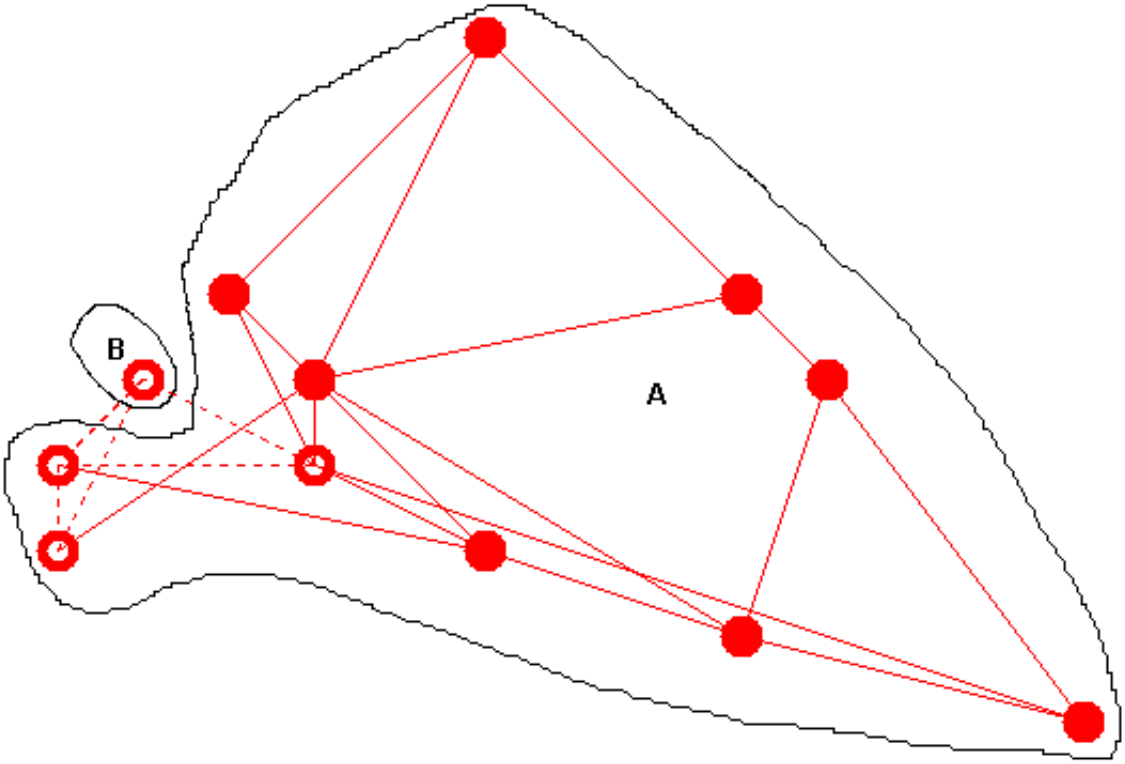
Steg för steg går det till så här:

1. Välj första flödesvägen.
2. Återställ den skuggade bågen (steg 6 för M5, eller steg 4 nedanför). Om två bågar som når samma nod skuggats kommer återställandet inte att ta bort dess flygplatsstatus.
3. Testa att skugga alla bågar längs vägen, en och en och se om någon ger bättre resultat.
4. Om ett bättre resultat hittas, byt det bågval som gjorts tidigare och starta om från 1.
5. Om inte ett bättre resultat hittas för den här flödesvägen, välj nästa väg och fortsätt från 2.
6. När alla vägar testats och ingen förbättring hittats är heuristiken färdig.

Observera att exekveringstiden för den här algoritmen fortfarande är polynomiell eftersom inte alla kombinationer testas utan endast ett polynomiellt antal bågar och detta sker högst ett polynomiellt antal gånger.

Nu följer en beskrivning på hur heuristiken arbetar med den stora exempelgrafen. Den börjar på precis samma sätt som den tidigare och optimeringen tar vid där den tidigare slutade. Detta beror på att det inte fanns fler par som behövde separeras. Om fler funnits hade den lokala optimeringen utförts direkt efter det att varje par separerats.

Först försöker heuristiken att optimera flödesväg **a** som syns i figur 5.17. Bågen 1-2 sätts tillbaka och sedan provar den att ta bort 2-4 och 4-7 utan att det blir bättre. När den fortsätter med flödet **b** ser den att 1-3 är ett bättre val än 3-5 eftersom flödet inte ökar men antalet flygplatser minskar. Anledningen till att den tidigare heuristiken inte kunde göra detta val var att den inte hade tillgång till det avbrutna **c** flödet när den försökte hitta ett sätt att bryta **b** flödet. Ingen mer optimering hittas sedan. Det slutliga resultatet ses i figur 5.19 där antalet flygplatser minskat med 1.



Figur 5.19: Slutresultat med flygplatser utsatta.

Flödesmetoden med globaloptimering x2 (M7a, M7b)

Den här heuristiken tar optimeringen ett steg längre. Den fungerar som den tidigare men med en skillnad i optimeringen.

Heuristiken börjar med att identifiera alla par av noder som bryter mot spannervillkoret. För varje par tas sedan fram en flödesgraf och bågarna i denna ges värden beroende på om de kan påverka flödet.

Enbart skuggning av en båge i ursprungsgrafen kommer inte att påverka flödet i denna om det finns andra vägar runt bågen. Därför används istället ett annat villkor för att se vilka bågar som är viktiga för att minska flödet. Om bågen genom att tas bort ur ursprungsgrafen sänker flödet i denna med ett så tilldelas bågen i flödesgrafen värdet ett. Annars får den värdet noll. Sedan slås alla värden på bågarna ihop för alla flödesgrafer. Det ger en graf där bågarna har en rang mellan noll och antalet problempar.

I den här heuristiken kommer information att delas mellan de lokala lösningarna så att när lösningen byggs på samma sätt som tidigare kan möjligen separationen av ett par dessutom separera eller delvis separera ett eller flera andra problempar.

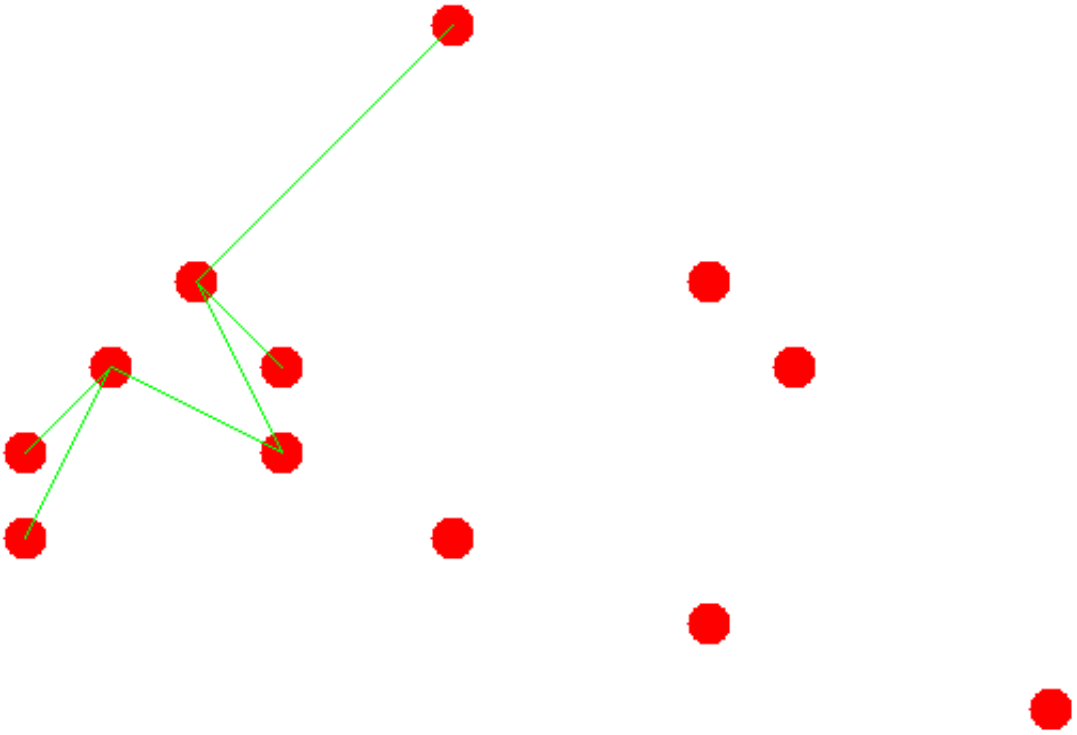
Det finns två varianter på den här heuristiken. När alla problem listats och den viktade grafen tagits fram börjar den ena varianten med att lösa de svåraste

problemen (M7b) medan den andra börjar med att lösa de lättaste (M7a).

Steg för steg går det till så här:

1. Ta fram alla par som bryter mot spannervillkoret.
2. Beroende på variant, sortera så att par med lägst/högst värde hamnar först.
3. Beräkna för varje par en flödesgraf där bågarna rangordnas efter hur de påverkar flödet enligt texten ovan.
4. Addera dessa värden till en graf med ursprungsutseendet men med värden på alla bågar.
5. Gör samma som i heuristiken för Flödesmetoden med lokaloptimering, men med skillnaden att "ett bättre resultat" här har en nivå till. Nytt för den här heuristiken är att om det finns flera bågar som kan sänka flödet med färre flygplatser så väljs den med högst rang.
6. Om det fortfarande finns problem, trots att alla från början kända par är separerade, lägg dessa till listan som togs fram i 1 och fortsätt från 2 men med en ren graf. De problem som finns kvar är nya problem som uppkommit genom lokaloptimeringen. Exemplet som följer med den lilla grafen visar hur det går till när nya problem skapas.

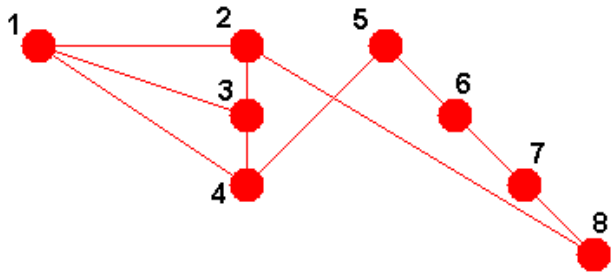
Nu följer ett exempel på hur den arbetar med den stora exempelgrafen. Det finns bara ett nodpar att ta hand om här så endast bågarna från deras flödesgraf kommer att ha rang större än 0. Figur 5.20 visar bågarna med rang 1. De bågar som syns i den här grafen har alltså förtur när flödena skall skäras av eftersom de själva kan minska flödet. Kravet är enligt ovan att de minskar flödet om de tas bort helt ur grafen.



Figur 5.20: Den ackumulerade grafen.

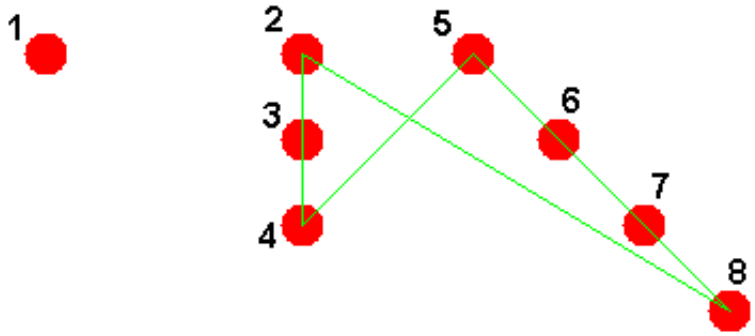
Den här heuristiken börjar på samma sätt som den grundläggande flödesmetoden och når ett mellanresultat som ser ut som figur 5.18. Den ser sedan på samma sätt som lokaloptimeringsheuristiken att 1-3 är ett bättre val än 3-5 för att minska flödet längs flödesväg b. Heuristiken tycker extra bra om denna båge då den förutom att minska antalet flygplatser med bibehållet nollflöde även har högre rang än den tidigare valda bågen. Slutresultatet blir samma som det i figur 5.19 och vi ser här att alla bågar som slutligen valdes har rang 1. Detta är en indikation på att rangen har betydelse för att hitta en optimal lösning.

Nu följer ett exempel på hur den arbetar på den lilla exempelgrafen. Den lilla grafen återges med numrerade noder i figur 5.21.



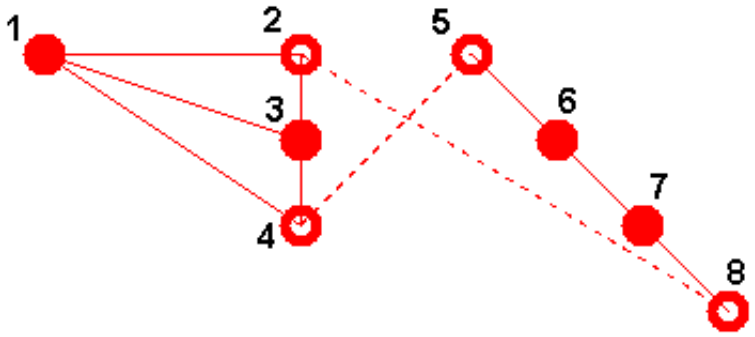
Figur 5.21: Den lilla exempelgrafen.

Problemparet 2-5 detekteras. Då det bara finns ett par sker ingen sortering. En flödesgraf beräknas enligt figur 5.22.



Figur 5.22: Flödesgrafen för paret 2-5.

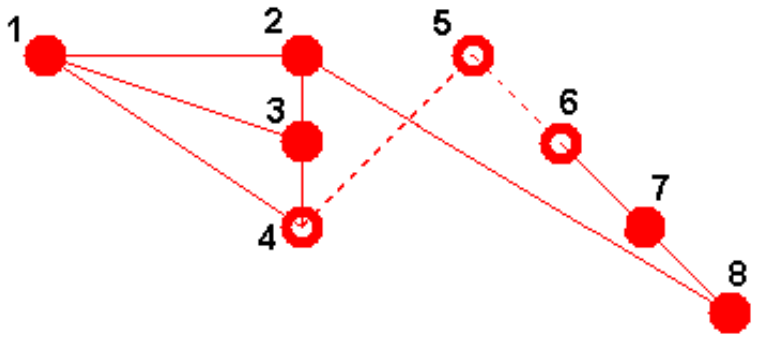
Flödesvägarna är 2-3-4-5 samt 2-8-7-6-5. Endast skuggning av bågen 4-5 längs första flödesvägen ger en minskning av flödet i arbetsgrafen som i det här läget ser ut som ursprungsgrafen. Därför väljs denna. Första bågen, 2-8, längs den andra flödesvägen ger direkt en minskning och det icke-optimerade resultatet blir enligt figur 5.23.



Figur 5.23: Före lokaloptimeringen.

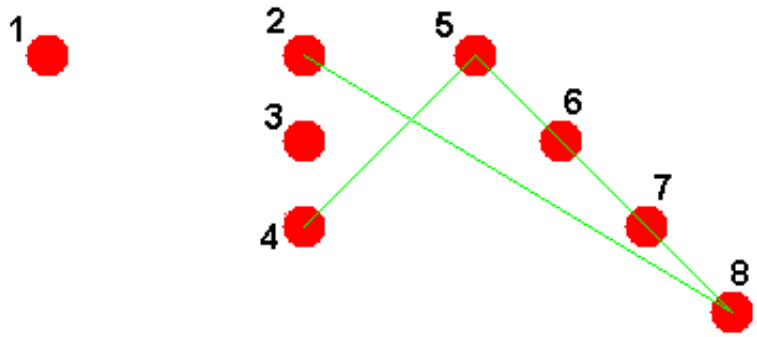
I det här läget startar lokaloptimeringen som testar att byta ut bågar längs flödesvägen. Ingen båge längs vägen 2-3-4-5 kan ersätta 4-5 eftersom de samtliga kan passeras via 1 och därför skulle öka flödet till nod 5. Däremot håller alla bågarna längs vägen 2-8-7-6-5 samma låga flöde. Därför väljer lokaloptimeringen att byta båge till 6-5 som håller samma låga flöde som 2-8 men samtidigt sänker antalet flygplatser med ett. Det ger grafen i figur 5.24.

Detta är ett exempel på när lokaloptimeringsheuristiken väljer dåligt. Visserligen har den lyckats sänka antalet flygplatser, men samtidigt skapa nya problem. De nya problemen är mellan noderna 4-7, 2-6, 3-3 samt 4-6. Globaloptimeringen försöker nu ta hand om detta genom att ta med denna information och backa ett



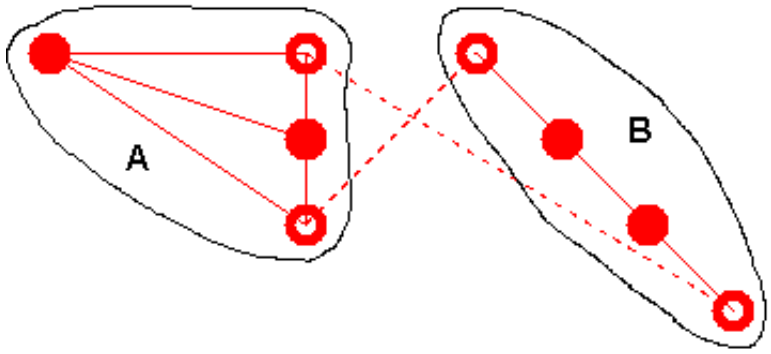
Figur 5.24: Efter lokaloptimeringen.

steg. En ny flödesgraf beräknas enligt figur 5.25 där varje båges rang visas. Eftersom alla bågarna ligger i en cykel kommer alla bågar med lika många gånger och ingen inbördes rangordning uppstår. Bågarna 2-3 och 3-4 får inte någon rang då de inte kan sänka flödet genom att tas bort ur ursprungsgrafen (se första stycket i heuristikens beskrivning).



Figur 5.25: Bågar med rang > 0.

Nu skiljer det sig mellan de två varianterna av heuristiken. Den ena startar med paret 4-7 och den andra med 4-6 (som har högst t-värde, 3.37). Båda två kommer till samma resultat. När bågarna längs vägen 4-3-2-8-7 och 4-5-6-7 testas väljs de första som fungerar (då de har samma rang alla), nämligen 2-8 resp. 4-5. Lokaloptimeringen hittar sedan inte någon förbättring. I tidigare steg hittades optimeringen tack vare att det fanns utsatta flygplatser som förband noderna som skulle separeras. Nu finns ingen sådan hjälp vilket lyckosamt gagnar heuristiken. När heuristiken återvänder efter att ha löst första problemet ser den att alla andra också lösts och avslutar som figur 5.26 visar.



Figur 5.26: Resultatet med flygplatser utsatta.

Översiktsmetoden (M8)

Den här heuristiken är ganska lik den globaloptimerande, men skiljer sig på en punkt. Den backar inte helt och försöker lösa alla problem igen. Den globaloptimerande heuristiken kommer att starta med en opåverkad graf i varje läge där den beräknar värden för olika bågar i flödesgrafen. Översiktsmetoden gör inte detta. Istället försöker den lösa alla problem som är synliga vid starten, kallade yttre problem. Om detta sedan ger upphov till nya problem, kallade inre problem, så använder den kännedomen om dessa för att försöka lysa de yttre problemen på ett sätt så att de inre inte uppkommer. Därför låter den de inre problemens ranger ingå i grafen som används för att rangordna bågarna när de yttre problemen löses. Om det trots detta inte går att förhindra att inre problem uppstår fortsätter den med den graf den fått fram och börjar inte om. Det innebär att de inre paren blir de yttre i nästa iteration.

Steg för steg går det till så här:

1. Ta fram alla par som bryter mot spannervillkoret. (Yttre problem)
2. Försök lösa dessa med globaloptimeringsmetodens båda varianter.
3. Uppstod det inre problem? I så fall, beräkna och poängsätt grafen där de yttre problemen upptäcktes med poäng från både yttre och inre problem.
4. Försök att lösa de yttre paren med den poängsatta grafen som fås i steg 3.
5. Upptäcks nya par så upprepa från 3.
6. Om upprepning av 3 inte gör att heuristiken kan lösa även de inre paren när den löser de yttre paren så fortsätt från steg 1, men med den graf där problemet har försökts lösas. Detta innebär att de inre par som inte kunde lösas blir yttre par i nästa iteration.

När denna heuristik exekveras på exempelgraferna returnerar globalheuristiken redan färdiglösta grafer.

Sveplinjemetoden (M9)

Den här heuristiken väljer en annan väg en de tidigare för att placera problemnoderna på olika öar. När problemnoderna är identifierade så använder den en linje för att dela upp grafen i två delar. Alla bågar som skärs av denna linje omvandlas till flygrutter och de noder som de sammanbinder blir flygplatser. För att avgöra var mellan problemnoderna det är bäst att skära av grafen används en sveplinjealgoritm.

Låt problemnoderna kallas s och t. Där s är den problemnod med lägst x-koordinat (lägst y-koordinat om de ligger på samma x-koordinat).

Algoritmen börjar med att skapa en sveplinje som är vinkelrät mot linjen mellan problemnoderna. Sveplinjen representeras av en normaliserad vektor och ges positiv y-koordinat (eller (-1,0) om s och t har samma x-koordinat). Denna linje läggs sedan genom noderna s och t. Noder som ligger till höger om linjen genom s och till vänster om linjen genom t kommer att innebära händelser när sveplinjen passerar mellan s och t. Dessa lagras i en prioritetskö med avståndet till sveplinjen genom s (start sveplinjen) som nyckel. Om två noder har samma avstånd lagras bara den ena eftersom det räcker med en händelse för att hantera båda. I varje steg flyttas sveplinjen längs linjen mellan s och t. Den flyttas så att den varje gång hamnar på ett avstånd från ursprunget som är lika med medelvärdet av de två senast utplockade noderna ur kön.

Steg för steg går det till så här:

1. Identifiera det nodpar som bryter mot spannervillkoret mest.
2. Tilldela s och t rätt noder enligt ordningen som nämnts ovan. Beräkna vektorn st och en normal till denna som har positiv y-koordinat. Normalisera denna vektor. Denna vektor symboliserar sveplinjens riktning. Om värdena urartar löses det som beskrivits ovan.
3. Utgå ifrån en full graf och ta bort alla noder till vänster om sveplinjen genom s. (Riktning beräknas med vektorprodukten.)
4. Ta från föregående graf bort alla noder till höger om sveplinjen genom t.
5. För alla återstående noder, beräkna avståndet till sveplinjen genom s. (Avståndet beräknas med hjälp av skalärprodukten och Pythagoras sats.) Stoppa in alla noder i en prioritetskö med avståndet som nyckel. Dubbletter tas bort. Även s och t stoppas in i kön.
6. Tag ut s ur kön.
7. Tag ut nästa nod ur kön och beräkna medelvärdet av denna och tidigare nods avstånd.
8. Placera sveplinjen på det nyss beräknade avståndet längs vektorn st. Sveplinjen placeras ortogonalt mot st.

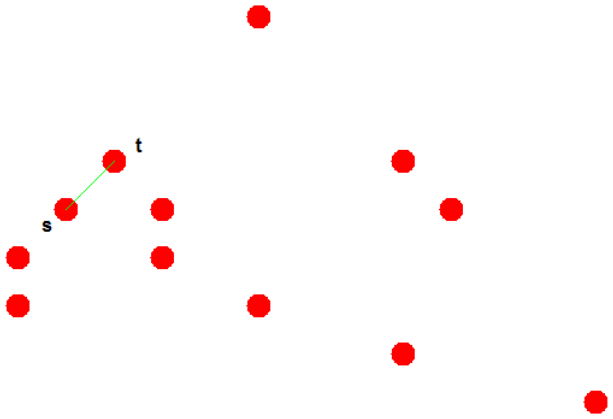
9. Beräkna för varje båge om den skärs. Detta sker då ena noden ligger till vänster om sveplinjen och den andra till höger. (Orienteringen beräknas med hjälp av tecknet på kryssprodukten.) Om en båge skärs, sätt ändpunkterna till flygplatser. Kom ihåg lägsta möjliga antal flygplatser.

10. Upprepa från punkt 7 tills alla händelser tagits om hand.

11. Utför uppdelning i öar enligt den sveplinje som gavs minst antal flygplatser.

12. Om grafen inte är en spanner, fortsätt vid punkt 1.

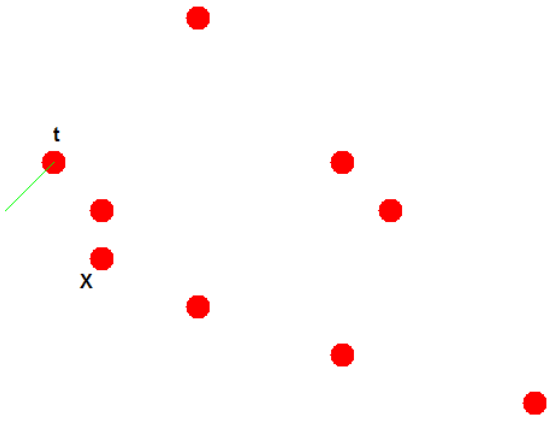
Nu följer ett exempel på hur den arbetar på den lilla exempelgrafen. Paret 2-5 i figur 5.21 identifieras som det värsta problemparet och grafen kommer därför att delas mellan dessa två noder. Området mellan dessa noder innehåller inga händelser utan det blir bara en linje som testas. Linjen hamnar mitt emellan noderna och resulterar i att bågarna 2-8 och 4-5 har noder på olika sidor. Dessa noder sätts till flygplatser och ger samma lösning som i figur 5.26.



Figur 5.27: Alla noder samt vektorn *st*.

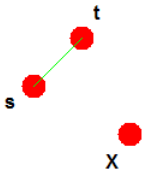
När heuristiken tar sig an den stora exempelgrafen inträffar händelser. Paret s-t upptäcks bryta mot spannervillkoret. Därefter beräknas en vektor mellan dessa och på så vis normalen till vektorn. Sveplinjen går från s till t i rät vinkel mot vektorn st. Figur 5.27 visar alla noder samt vektorn st.

I det här läget beräknar heuristiken alla noder som ligger till höger om sveplinjens start, detta beror på att sveplinjen rör sig åt höger. Figur 5.28 visar dessa.



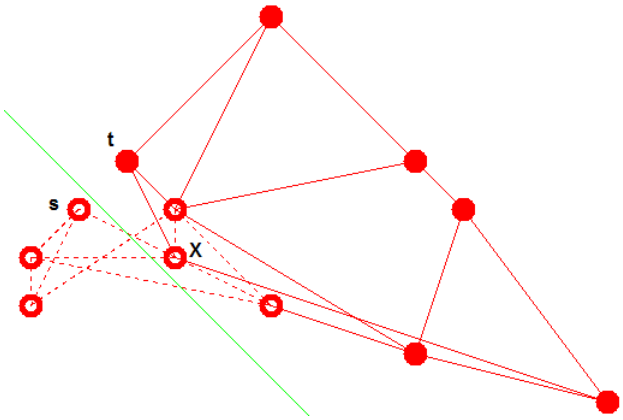
Figur 5.28: Noder till höger om sveplinjen.

Noden X är den enda som befinner sig inom det område som sveps. Den utgör en händelse längs linjen. Linjen kommer att svepa från s till X till t. Figur 5.29 visar de noder som blir händelser för sveplinjen.

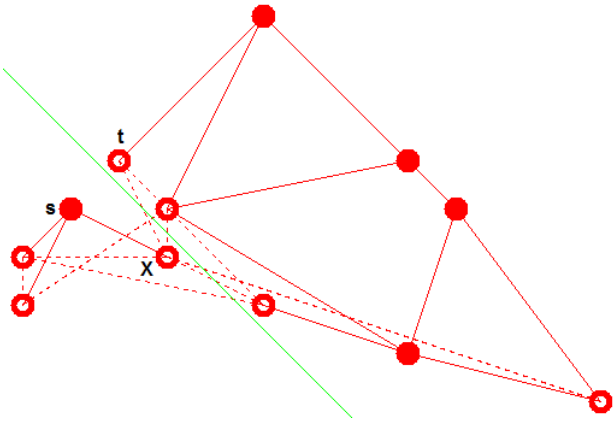


Figur 5.29: Noder som ger upphov till händelser för sveplinjen.

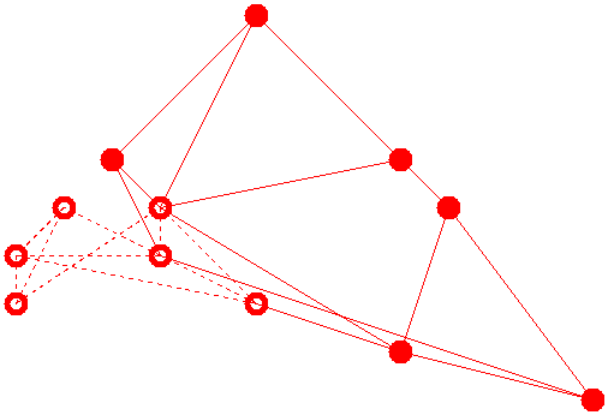
Här finns det två lägen för sveplinjen att befinna sig, halvvägs mellan s och X samt halvvägs mellan X och t. Inom dessa intervall skärs samma linjer överallt. Figur 5.30 och 5.31 visar hur sveplinjen skär samt vilka noder som på grund av uppdelningen blir flygplatser. Slutligen väljer heuristiken den lösning med minst antal flygplatser som syns i figur 5.32.



Figur 5.30: Sveplinjen i första intervallet.



Figur 5.31: Sveplinjen i andra intervallet.



Figur 5.32: Slutlig lösning.

5.3.3 Testgrafer

Det finns ett stort antal möjliga konstellationer att testa heuristikerna på. Fördelen med att testa på små grafer är att det optimala värdet kan beräknas genom t.ex. en *brute force* metod. Därför valdes ett antal små grafer ut till testmängden. För större grafer finns inte möjligheten att beräkna det optimala värdet, men det är intressant ändå att se hur heuristikerna lyckas. Därför valdes en mängd större grafer också ut till testmängden. De små graferna placerades på verktygets lilla yta (600x600) medan de stora placerades på den stora ytan (1000x1000). Slutligen så anpassades antalet grafer inom varje svit till ett rimligt antal. Rimligt här ansågs det vara om hela testsviten med en modern dator tog cirka en månad att beräkna. Tabell 5.1 visar de valda testsviterna.

Antal noder	Antal bågar	Antal grafer i sviten
14	24	10000
14	28	10000
14	32	10000
16	27	1000
16	32	1000
16	37	1000
20	30	10000
20	40	10000
20	50	10000
40	70	1000
40	80	1000
40	90	1000

Tabell 5.1: Testsviter.

Grafer med 14 noder går snabbt att beräkna jämfört med grafer med 16 noder och därför har de fått fler grafer per svit. För grafer med 20 resp. 40 noder beräknas inte det optimala värdet och dessa kan därför beräknas snabbt. Trots det innebär 40 noder en så tung belastning att färre grafer har valts här. När det gäller förhållandet mellan noder och bågar känns 1:2 förhållandet ganska lagom. Det är inte för glesa grafer men samtidigt inte för trångt. Problemet med slumpgrafer är att om för många bågar tillåts kommer alla noder att bli flygplatser. Det är inte heller rimligt, om man t.ex. tänker sig att grafen efterliknar ett vägnät, att det finns alltför många bågar som korsar varandra. Detta blir för dyrt och opraktiskt att underhålla och ofta blir dylika korsningar nya städer med tiden.

Av de fyra olika typer av slumpgrafer som programmet genererar valdes att samtliga testgrafer skall vara av typen slumpvis-klustrat. Figur 5.7 visar varför. Slumpvis-slumpvis grafer ser inte realistiska ut. Med tanke på att det inte finns någon struktur i dessa grafer är resultatet inte lika intressant då slumpen för mycket

kommer att påverka vilken heuristik som lyckas bäst (alla heuristiker har grafer som de kan lyckas bäst med). Klustrat-slumpvis är bättre, men utnyttjar inte hela grafytan då alla noder hamnar i en klump. Slutligen är klustrat-klustrat dåligt av anledningen som nyss nämnts men även därför att det blir alldeles för många bågar som går igenom noder (detta känns orimligt om grafen efterliknar ett vägnät). I figur 5.7 finns det 20 noder och 40 bågar i alla graferna. I graf 4 som är klustrat-klustrat syns inte alla dessa bågar på grund av att många går ovanpå andra bågar men slutar i andra noder. Detta är en egenskap som dels kommer från att klustrade grafer gärna lägger noderna i rad, dels kommer det från att klustrade bågar hamnar väldigt nära. Sammantaget leder denna kombination till många orealistiska bågar.

Även om fokus låg på grafer med förhållandet 1:2 mellan noder och bågar så valdes även grafer med andra förhållanden ut till testmängden. Testmängden varierar på båda sidor om förhållandet 1:2 för att se hur heuristikerna klarar dessa variationer.

5.4 Resultat

Resultaten presenteras med grafer som har lika antal noder inom samma underkapitel. Tabell 5.2 visar en sammanställning över samtliga heuristiker med förkortningar.

Förkortning	Beskrivning
M1	Girigmetoden
M2a	Diametermetoden - euklidisk
M2b	Diametermetoden - längs grafens bågar
M3	Enkelklustermetoden
M4a	Multiklustermetoden - euklidisk
M4b	Multiklustermetoden - längs grafens bågar
M5	Flödesmetoden
M6	Flödesmetoden med lokaloptimering
M7a	Flödesmetoden med globaloptimering - Lätta problem först
M7b	Flödesmetoden med globaloptimering - Svåra problem först
M8	Översiktsmetoden
M9	Sveplinjemetoden

Tabell 5.2: Översikt över heuristiker samt deras förkortningar.

5.4.1 14 noder, 10000 grafer

Resultatet presenteras som antalet flygplatser totalt summerat över hela testsviten. Tabell 5.3 visar heuristikerna med en representant för varje typ av heuristik. Tabell 5.4, 5.5 och 5.6 visar hur de olika heuristikerna av samma typ står sig mot varandra.

Bågar	M1	M2b	M3	M4b	M8	M9
24	71952	72011	71947	75374	63711	84532
28	68104	67478	73399	78530	58750	82841
32	65501	64754	77093	84380	55773	84675

Tabell 5.3: Summerat antal flygplatser per heuristik.

Bågar	M2a	M2b
24	73826	72011
28	69517	67478
32	66533	64754

Tabell 5.4: Jämförelse M2a - M2b.

Bågar	M4a	M4b
24	77048	75374
28	81004	78530
32	87016	84380

Tabell 5.5: Jämförelse M4a - M4b.

Bågar	M5	M6	M7a	M7b	M8
24	71952	66902	65347	65321	63711
28	66057	60800	59912	59687	58750
32	62811	57378	56675	56455	55773

Tabell 5.6: Jämförelse mellan flödesheuristikerna.

5.4.2 16 noder, 1000 grafer

Resultatet presenteras som antalet flygplatser totalt summerat över hela testsviten. Tabell 5.7 visar heuristikerna med en representant för varje typ av heuristik. Tabell 5.8, 5.9 och 5.10 visar hur de olika heuristikerna av samma typ står sig mot varandra.

Bågar	M1	M2b	M3	M4b	M8	M9
27	8962	9007	8965	9314	8379	10776
32	8327	8510	8530	9214	7288	10254
37	7730	7679	8854	9701	6648	10026

Tabell 5.7: Summerat antal flygplatser per heuristik.

Bågar	M2a	M2b
27	9194	9007
32	8510	8510
37	7812	7679

Tabell 5.8: Jämförelse M2a - M2b.

Bågar	M4a	M4b
27	9454	9314
32	9486	9214
37	10025	9701

Tabell 5.9: Jämförelse M4a - M4b.

Bågar	M5	M6	M7a	M7b	M8
27	9414	8791	8614	8590	8379
32	8157	7532	7429	7415	7288
37	7417	6786	6745	6710	6648

Tabell 5.10: Jämförelse mellan flödesheuristikerna.

5.4.3 20 noder, 10000 grafer

Resultatet presenteras som antalet flygplatser totalt summerat över hela testsviten. Tabell 5.11 visar heuristikerna med en representant för varje typ av heuristik. Tabell 5.12, 5.13 och 5.14 visar hur de olika heuristikerna av samma typ står sig mot varandra.

Bågar	M1	M2b	M3	M4b	M8	M9
30	125050	125319	122435	124794	124148	149816
40	123275	123277	123885	129999	114578	147796
50	103494	102936	117626	127638	90189	138181

Tabell 5.11: Summerat antal flygplatser per heuristik.

Bågar	M2a	M2b
30	126642	125319
40	126139	123277
50	106136	102936

Tabell 5.12: Jämförelse M2a - M2b.

Bågar	M4a	M4b
30	126452	124794
40	132492	129999
50	131251	127638

Tabell 5.13: Jämförelse M4a - M4b.

Bågar	M5	M6	M7a	M7b	M8
30	136244	130119	127286	126894	124148
40	125911	117381	117308	116382	114578
50	99864	92069	92137	91501	90189

Tabell 5.14: Jämförelse mellan flödesheuristikerna.

5.4.4 40 noder, 1000 grafer

Resultatet presenteras som antalet flygplatser totalt summerat över hela testsviten. Tabell 5.15 visar heuristikerna med en representant för varje typ av heuristik. Tabell 5.16, 5.17 och 5.18 visar hur de olika heuristikerna av samma typ står sig mot varandra.

Bågar	M1	M2b	M3	M4b	M8	M9
70	31363	31367	31048	31255	32846	35407
80	32992	32971	32717	33099	34029	36208
90	33123	33100	33269	33782	34061	36602

Tabell 5.15: Summerat antal flygplatser per heuristik.

Bågar	M2a	M2b
70	31614	31367
80	33202	32971
90	33463	33100

Tabell 5.16: Jämförelse M2a - M2b.

Bågar	M4a	M4b
70	31572	31255
80	33407	33099
90	34148	33782

Tabell 5.17: Jämförelse M4a - M4b.

Bågar	M5	M6	M7a	M7b	M8
70	34158	33441	33334	33171	32846
80	35037	34146	34611	34150	34029
90	35224	34139	34576	34176	34061

Tabell 5.18: Jämförelse mellan flödesheuristikerna.

5.4.5 Exekveringstider

Tabell 5.19 visar exekveringstiderna för samtliga heuristiker när de löser de 1000 första graferna i serien *20 noder 40 bågar*.

Heuristik	Exekveringstid (sekunder)
M1	22
M2a	37
M2b	375
M3	30
M4a	419
M4b	753
M5	1051
M6	2093
M7a	7141
M7b	7257
M8	9196
M9	52

Tabell 5.19: Exekveringstider för 1000 grafer.

5.5 Jämförelser

Här presenteras jämförelser mellan de olika metoderna samt övriga resultat som kan vara av intresse för analysen. Ett sådant resultat är det optimala antalet flygplatser per graf som visas i tabell 5.20.

Antal noder	Antal bågar	Flygplatser	Graf medel
14	24	55293	5,5293
14	28	52559	5,2559
14	32	51781	5,1781
16	27	7013	7,013
16	32	6335	6,335
16	37	5970	5,97

Tabell 5.20: Det optimala antalet flygplatser per testserie.

Tabellen visar att antalet flygplatser sjunker då antalet bågar ökar. En nod som ligger på en ö och är en flygplats kommer att skapa fler flygplatser på andra öar om

den har fler bågar som lämnar den. Å andra sidan innebär fler bågar att avstånden mellan noderna inom grafen sjunker. Tydligen är detta senare fenomenet starkare än det tidigare.

I det följande kommer tabeller över skillnaderna mellan två metoder att presenteras. Observera att det är den nedre gränsen för konfidensintervallen när skillnaden skattas som presenteras.

5.5.1 Jämförelse mellan Diametermetoderna

I samtliga tabeller syns att M2a ger ett något högre resultat än M2b. Här undersöks hur stor fördel M2b kan förväntas ha jämfört med M2a. Med hjälp av de statistiska metoder som beskrivs i appendix A.5 erhålls tabell 5.21 som visar skillnaden mellan M2a och M2b till M2bs fördel.

Antal noder	Antal bågar	Δ_{min}
14	24	0,1346
14	28	0,1556
14	32	0,1321
16	27	0,1380
16	32	0,1276
16	37	0,07980
20	30	0,08656
20	40	0,2218
20	50	0,2420
40	70	0,1812
40	80	0,1657
40	90	0,2907

Tabell 5.21: Skillnad mellan M2a och M2b.

Det framgår ingen tydlig trend mellan skillnaden för olika typer av grafer. Största skillnaden för 20 och 40 noders grafer inträffar med flest antal bågar. Möjligen är det så att för större grafer presterar M2b bättre ju fler bågar som finns tillgängliga.

5.5.2 Jämförelse mellan Multiklustermetoderna

I samtliga tabeller syns att M4a ger ett något högre resultat än M4b. Här undersöks hur stor fördel M4b kan förväntas ha jämfört med M4a. Med hjälp av de statistiska metoder som beskrivs i appendix A.5 erhålls tabell 5.22 som visar skillnaden mellan M4a och M4b till M4bs fördel.

Antal noder	Antal bågar	Δ_{min}
14	24	0,1249
14	28	0,1986
14	32	0,2129
16	27	0,09321
16	32	0,2189
16	37	0,2663
20	30	0,1241
20	40	0,1928
20	50	0,2880
40	70	0,2608
40	80	0,2529
40	90	0,3028

Tabell 5.22: Skillnad mellan M4a och M4b.

Här syns en tydlig trend då skillnaden i samtliga fall ökar med antalet bågar om antalet noder hålls konstant.

5.5.3 Jämförelse mellan M1 och M3

M1 och M3 arbetar på liknande sätt. De lägger hela tiden till den kortaste båge som går. Medan M1 arbetar lokalt och bygger en ö åt gången arbetar M3 globalt i grafen och bygger multipla öar samtidigt. M1 verkar ge bättre resultat och därför får Δ beteckna avståndet till M3.

Här syns tydligt att inom varje testserie så ökar Δ-värdet då antalet bågar ökar.

5.5.4 Jämförelse mellan M2b och M4b

M2b och M4b arbetar också på liknande sätt, de mäter diametern på samma sätt och lägger hela tiden till bågar som ger så liten diameterökning som möjligt. Medan M2b arbetar lokalt och bygger en ö åt gången arbetar M4b globalt i grafen och bygger multipla öar samtidigt. M2b verkar ge bättre resultat och därför får Δ beteckna avståndet till M4b. Tabell 5.24 visar värdena.

Även här syns tydligt att Δ-värdet ökar då antalet bågar ökar inom varje testserie.

Antal noder	Antal bågar	Δ_{min}
14	24	-0,07386
14	28	0,4475
14	32	1,072
16	27	-0,06987
16	32	0,1157
16	37	1,029
20	30	-0,3335
20	40	-0,03647
20	50	1,295
40	70	-0,40246
40	80	-0,37045
40	90	0,048668

Tabell 5.23: Skillnad mellan M1 och M3.

Antal noder	Antal bågar	Δ_{min}
14	24	0,2553
14	28	1,015
14	32	1,867
16	27	0,2247
16	32	0,7899
16	37	1,914
20	30	-0,1291
20	40	0,5661
20	50	2,340
40	70	-0,2009
40	80	0,3737
40	90	0,3416

Tabell 5.24: Skillnad mellan M2b och M4b.

5.5.5 Jämförelse mellan flödesheuristikerna

Det finns fyra intressanta jämförelser att göra här. Dels är M6 en direkt optimerande version av M5. Dels är det intressant att jämföra M7a och M7b eftersom de angriper delproblemen i olika ordning, men annars är lika. Slutligen är M8 den metod som verkar ha lyckats bäst så den måste jämföras med M6 och M7b för att se vilken som är bäst bland flödesheuristikerna. Tabell 5.25 visar skillnaden mellan M6 och M5 där M6 är den heuristik som presenterar lägre resultat. Tabell 5.26,

5.27 och 5.28 visar skillnaderna M7a-M7b, M6-M8 och M7b-M8 där den senare i paret presterar lägst. Tabell 5.26 visar hela konfidensintervallet med nedre och övre gränser enligt signifikans ***.

Antal noder	Antal bågar	Δ_{min}
14	24	0,4752
14	28	0,4974
14	32	0,5161
16	27	0,5884
16	32	0,5928
16	37	0,5995
20	30	0,5723
20	40	0,8099
20	50	0,7415
40	70	0,6569
40	80	0,8293
40	90	1,017

Tabell 5.25: Skillnad mellan M6 och M5.

Antal noder	Antal bågar	Δ_{min}	Δ	Δ_{max}
14	24	-0,03718	0,002600	0,04238
14	28	-0,01375	0,02250	0,05875
14	32	-0,009482	0,02200	0,05348
16	27	-0,03059	0,02400	0,07859
16	32	-0,03147	0,01400	0,05947
16	37	-0,003446	0,03500	0,07345
20	30	-0,02399	0,03920	0,1024
20	40	0,02779	0,09260	0,1574
20	50	0,008594	0,06360	0,1186
40	70	0,07638	0,1630	0,2496
40	80	0,3762	0,4610	0,5458
40	90	0,3095	0,4000	0,4905

Tabell 5.26: Skillnad mellan M7a och M7b.

Antal noder	Antal bågar	Δ_{min}
14	24	0,2758
14	28	0,1643
14	32	0,1241
16	27	0,3571
16	32	0,1961
16	37	0,09690
20	30	0,5343
20	40	0,2168
20	50	0,1315
40	70	0,5140
40	80	0,03330
40	90	-0,01047

Tabell 5.27: Skillnad mellan M6 och M8.

Antal noder	Antal bågar	Δ_{min}
14	24	0,1231
14	28	0,05820
14	32	0,03778
16	27	0,1628
16	32	0,08592
16	37	0,02599
20	30	0,2173
20	40	0,1205
20	50	0,07960
40	70	0,2493
40	80	0,04327
40	90	0,03067

Tabell 5.28: Skillnad mellan M7b och M8.

5.5.6 Jämförelse mellan de olika metodtyperna

Tabell 5.29 visar de olika metodtypernas delta-medelvärden jämfört med M8.

Antal noder	Antal bågar	M1	M2b	M3	M4b	M9
14	24	0,8241	0,8300	0,8236	1,1663	2,0821
14	28	0,9354	0,8728	1,4649	1,978	2,4091
14	32	0,9728	0,8981	2,132	2,8607	2,8902
16	27	0,5830	0,6280	0,5860	0,9350	2,397
16	32	1,039	1,040	1,242	1,926	2,966
16	37	1,082	1,031	2,206	3,053	3,378
20	30	0,0902	0,1171	-0,1713	0,06460	2,5668
20	40	0,8697	0,8699	0,9307	1,5421	3,3218
20	50	1,3305	1,2747	2,7437	3,7449	4,7992
40	70	-1,483	-1,479	-1,798	-1,591	2,561
40	80	-1,037	-1,058	-1,312	-0,9300	2,179
40	90	-0,9380	-0,9610	-0,7920	-0,2790	2,541

Tabell 5.29: Deltavärden jämfört med M8.

5.6 Analys

5.6.1 Analys av Diametermetoderna

Spannervillkoret utgörs av en kvot mellan avstånd i grafen och avstånd geometriskt. Detta är samtidigt de två sätt att mäta diameter på som heuristikerna M2a och M2b använder. Om kvoten betraktas så finns det två sätt att minimera den, antingen minska täljaren, dvs. avståndet i grafen, eller öka nämnaren, dvs. det geometriska avståndet. När diametern växer ökar det geometriska avståndet och båda heuristikerna drar nytta av detta.

Heuristik M2a väljer att utöka den spannerö som byggs genom att lägga till den nod som ger minst ökning av den geometriska diametern. Den aktuella nod som läggs till i en iteration uppfyller spannervillkoret, men hur är det sedan? Om kvoten ovan betraktas betyder det att noder som senare läggs till troligare får problem då nämnaren minimeras i varje steg. Därför blir det så att spanneröar som byggts av M2a sannolikare har en högre spannerkonstant och dessutom bär med sig mindre möjligheter att lägga till nya noder än öar byggda av M2b.

Heuristik M2b väljer att i varje steg minimera möjligheten till stora täljare eftersom den minimerar diametern med avseende på avstånd inom ön. Detta arbetar direkt mot att underlätta för framtida noder att klara spannervillkoret.

5.6.2 Analys av Multiklustermetoderna

Samma resonemang som för skillnaden mellan M2a och M2b gäller här. Men det verkar dessutom vara någonting mer som påverkar då skillnaden växer med antalet bågar. Fler bågar borde betyda fler val i varje steg, men så även för M2 heuristikerna. Skillnaden kan öka antingen genom att M4a presterar sämre, M4b bättre, eller båda orsakerna samtidigt. Klart är att M4 heuristikerna rör sig åt motsatt riktning jämfört med alla andra heuristiker när fler bågar får användas. Det ser ut som att M4a ökar snabbare och det verkar vara detta som gör att skillnaden ökar. Kanske leder ökade valmöjligheter till fler dåliga val. Speciellt kan detta drabba M4a som redan gör sämre val enligt resonemanget ovan. Nu får den ökade möjligheter att göra dessa dåliga val, M4b lyckas dock välja lite mindre dåligt

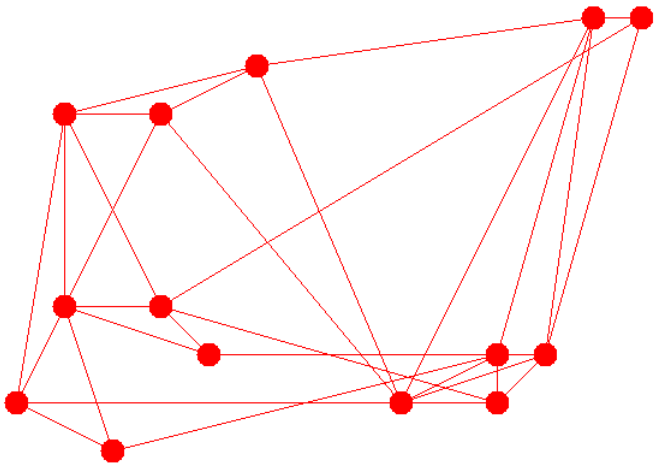
5.6.3 Analys av lokala och globala heuristiker

Det är intressant att se vilka heuristiker som lyckas bäst, lokala eller globala. Här avses att jämföra M1 med M3 och M2b med M4b. Tabell 5.23 visar en nedre gräns för skillnaden mellan M1 och M3. Enligt tabellen kan det inte anses säkert att M1 är bättre än M3 på hälften av testserierna. Å andra sidan är det stor skillnad på de testserier där man säkert kan säga att M1 är bättre. Det finns också ett tydligt mönster där M1 blir relativt bättre inom varje testserie då fler bågar finns att tillgå. Detta beror främst på att M3 presterar sämre när antalet bågar ökar. Figur 5.12 på sidan 75 visar vad som kan ligga bakom. Ju fler bågar, ju fler korta bågar vilket leder till fler små öar. Medan M1 bygger en ö åt gången kommer M3 att bygga små öar som växer tills de inte kan växa längre vilket leder till att alla noder med bågar som lämnar öarna blir flygplatser. Troligen är det så att när M1 bygger sin ö kommer den att absorbera många av de småöar som M3 bygger. Men medan M1 införlivar dessa i sin stora ö kommer M3s öar ofta att växa tills de når en maxstorlek. Sedan kan dessa många öar inte slås samma vilket leder till fler flygplatser. Sammantaget ger fler bågar att M3 skapar fler småöar vilket leder till fler kontaktpunkter med andra öar och fler flygplatser.

Samma fenomen verkar gälla mellan M2b och M4b. Här är det dock mycket större skillnad där extremfallet är *20 noder 50 bågar* testserien där M2b presterar 2,34 flygplatser bättre per graf i genomsnitt. Det bildas helt enkelt för många små öar med låg diameter som senare inte kan sättas ihop till större öar. Troligen är de för stora för att kunna sättas ihop. Om man jämför ö-storleken är det troligt att M4b och M3 producerar många öar med ungefär lika många noder medan M1 och M2b producerar öar av större variation, troligen några stora och några små vilket leder till färre totala flygplatser.

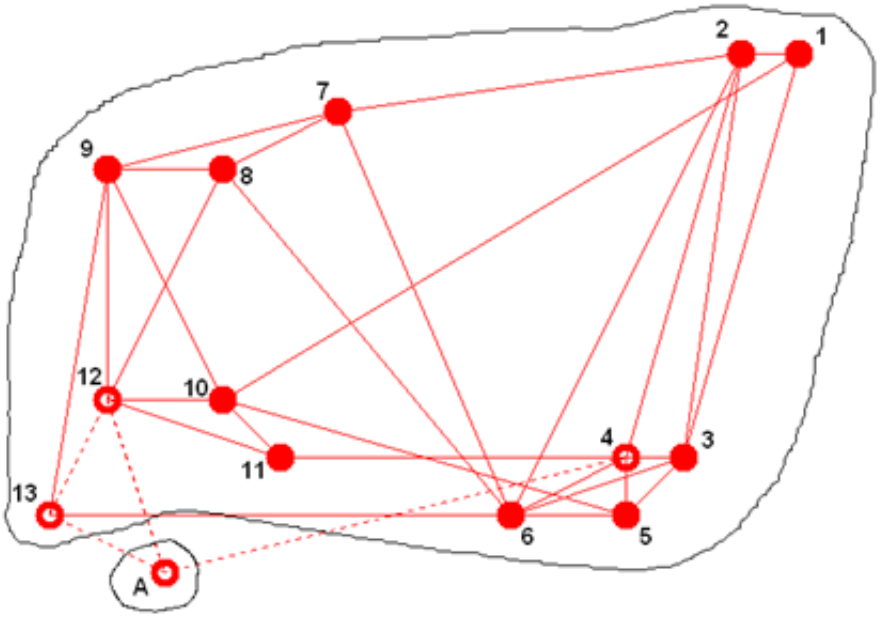
Ett exempel ges av graf nummer 9922 ur serien *14 noder 32 bågar*. Grafen visas i figur 5.33.

Figur 5.34 visar hur M2b hanterar den här grafen. Noderna adderas i nummerordning tills det inte går längre. Det blir en ö med 13 noder och en ö med 1 nod. Figur 5.35 och 5.36 visar i två steg hur M4b löser problemet. Till att börja

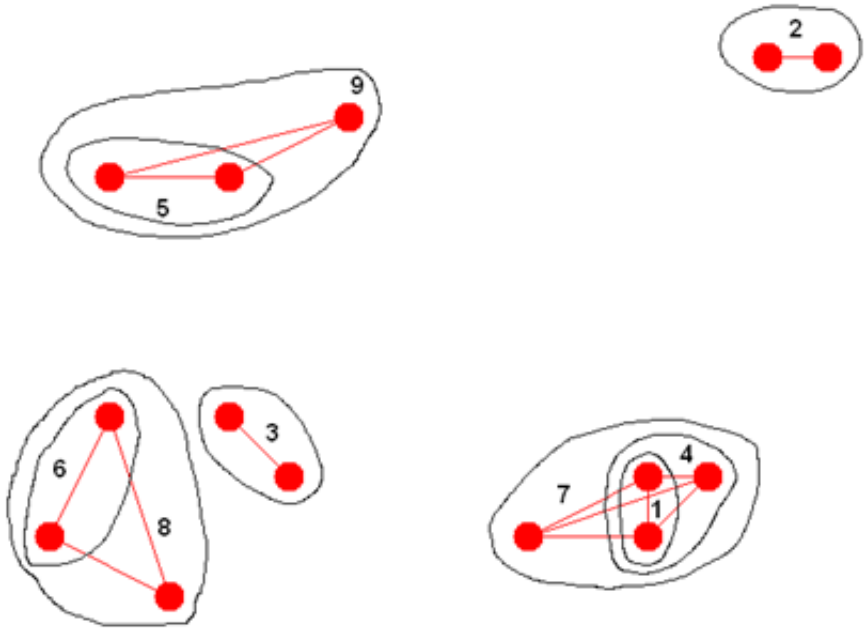


Figur 5.33: Graf 9922 ur testserien *14 noder 32 bågar*.

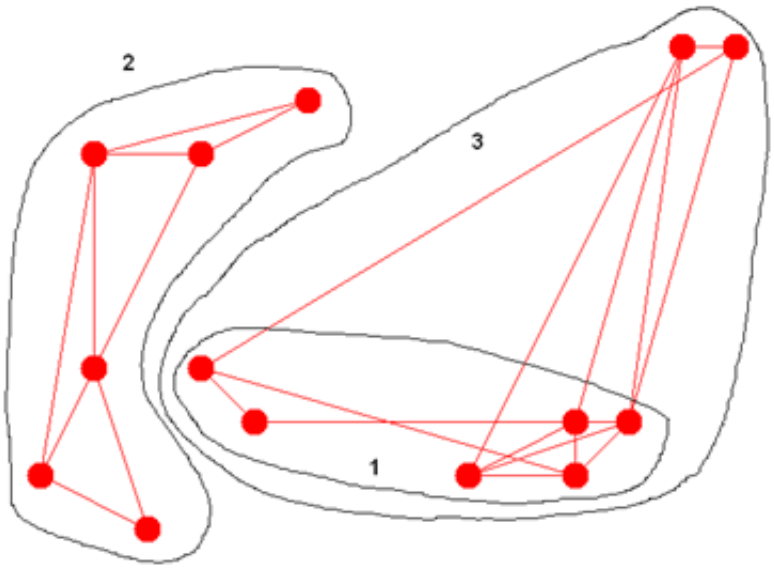
med bildas flera öar. De första 6 stegen bildar 5 olika öar. Sedan slås dessa ihop, jämt fördelat så att när slutet nås, vilket ses i figur 5.36 så finns det jämnstora öar. Dessa har många kopplingar emellan sig vilket syns på slutresultatet i figur 5.37. Heuristik M2b presenterar en lösning med 4 flygplatser medan heuristik M4b måste använda 11 stycken.



Figur 5.34: Heuristik M2b löser graf 9922.



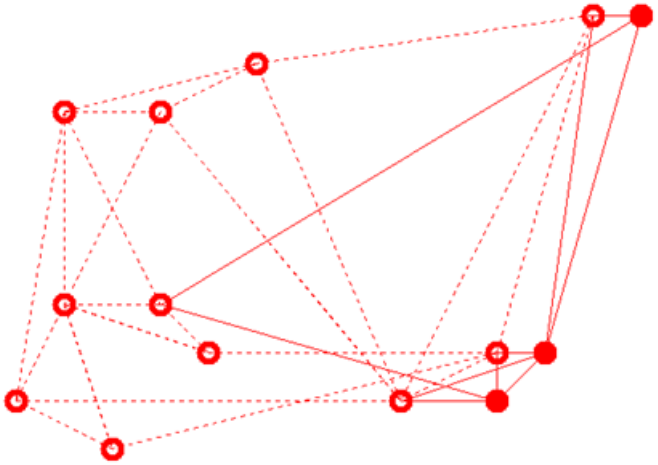
Figur 5.35: De 9 första stegen för heuristik M4b på graf 9922.



Figur 5.36: De sista tre stegen för M4b.

5.6.4 Analys av flödesheuristikerna

Tabell 5.25 visar hur stor skillnad lokaloptimeringen gör jämfört med den icke-optimerade lösningen. Det ser ut som om lokaloptimeringen blir mer effektiv ju fler



Figur 5.37: Slutresultat M4b.

bågar som finns i grafen. Fler bågar innebär ju fler möjligheter för optimering så resultatet var väntat.

Tabell 5.26 visar skillnaderna mellan att börja med de lätta respektive de svåra problemen först när globaloptimeringen exekveras. Resultatet pekar klart på att det är bättre att lösa de svåra problemen först. Troligen kan lösningen av dessa också ta hand om en del av de lätta problemen. Om heuristiken däremot löser de lätta problemen först verkar de svåra kvarstå och leda till fler flygplatser.

Tabell 5.27 visar skillnaden mellan den bästa flödesheuristiken, M8 (översiktsmetoden), och den lokaloptimerande. M8 är klart bättre i alla lägen. Detta är inte förvånande då M8 indirekt använder sig av den lokaloptimerande metoden och dessutom tillämpar översikt för att använda denna bättre. Det som var oväntat var att skillnaden minskar med ökat antal bågar inom varje segment. Troligen blir översiktsfördelen lägre då det blir färre problempar som den kan utnyttja i sin översikt.

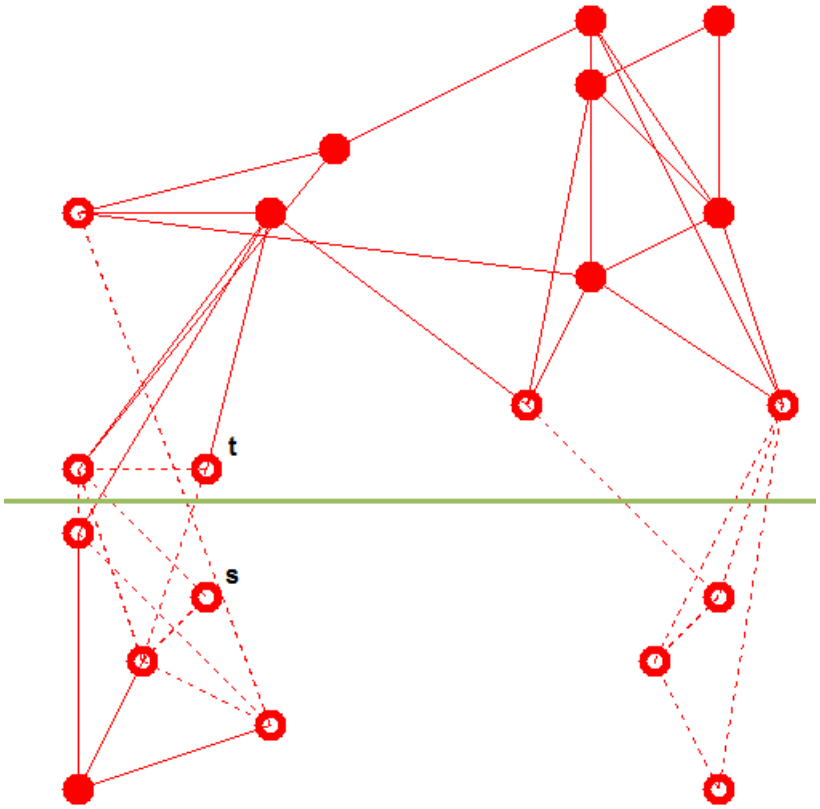
Tabell 5.28 visar skillnaden mellan översiktsmetoden och den globaloptimerande metoden. Översiktsmetoden är generellt sett bättre på alla testserier. Detta beror på att M8 arbetar smartare. M7 kommer om den löser de först synliga problemparen och sedan upptäcker nya att backa och försöka lösa alla paren samtidigt från en ren graf. När M8 ställs inför samma problem, och det gör den exakt eftersom den exekverar M7 på de första synliga problemen, så kommer den att ta in informationen om de nya problemen, men först försöka lösa de ursprungliga med denna extra information. Om detta går kan den sluta och har alltså inte behövt separera fler noder genom att införa flygplatser. Det är därför M8 presterar bättre än M7 i många fall.

Även här ser vi att skillnaden minskar för fler bågar. Detta beror troligen som

förut på att antalet problempar minskar vilket ger mindre utrymme för M8 att använda sin översiktsmetod. Detta leder till att den mer och mer arbetar som M7 och resultaten närmar sig varandra.

5.6.5 Analys av Sveplinjemetoden

Ur tabell 5.29 syns direkt att heuristik M9, sveplinjemetoden, presterar dåligt. Faktiskt presterar den sämst av alla heuristiker. Detta beror troligen på att den är för grov. Problemet ligger i att den skär hela grafen när den försöker dela upp i två delar. Detta kan behövas, men oftast räcker det att skära en liten del mellan noderna som skall separeras. Figur 5.38 visar ett exempel där metoden misslyckas på grund av den här anledningen.



Figur 5.38: Sveplinjemetoden misslyckas.

Noderna s och t behöver separeras. Trots att den vänstra delen av grafen är oberoende av den högra så skär heuristiken båda delarna vilket leder till att 5 onödiga flygplatser skapas. Heuristiken behöver 18 flygplatser av 20 möjliga noder för att lösa den här grafen. Majoriteten av de övriga heuristikerna löser den med 9

flygplatser vilket också är det optimala värdet. Vidare utveckling av denna metod diskuteras i kapitel 5.7.1.

5.6.6 Analys av exekveringstider

Tabell 5.19 visar exekveringstider för alla heuristikerna. Det är föga förvånande att den giriga metoden är snabbast. I varje iteration väljer den ut en ny nod för att antingen utöka den befintliga ön eller för att påbörja en ny ö. Det blir alltså högst n iterationer. Diametermetoden med euklidiskt avstånd är nästan lika snabb. Den väljer också en nod, men det går lite långsammare att beräkna diametern av grafen än avståndet till den. Diametermetoden som mäter diametern via bågarna är dock mycket långsammare. En faktor 10 som beror på att den i varje iteration beräknar kortaste vägen inom ön mellan alla noder via Floyds algoritm som är $O(n^3)$.

M3 är nästan lika snabb som M1. I varje iteration väljer den också en nod att lägga till men den tittar på alla bågar medan M1 bara betraktar de bågar som går ut ifrån den aktuella ön.

M4 heuristikerna tar lite längre tid än de tidigare. Detta beror på att de testar många kombinationer. Varje båge som inte använts testas i varje iteration. Detta skulle kunna optimeras genom att spara värden som inte påverkas av iterationens val och endast räkna om de som påverkas av valet. Även här kommer Floyds algoritm in i bilden och gör M4b långsammare än M4a.

M5 är den första av flödesalgoritmerna och den tar längre tid än de hittills nämnda metoderna. Här kommer många beräkningar in i bilden då flödet skall beräknas och stängas av. Att beräkna flödesgrafen tar tid. Detta är ännu mer synligt i exekveringstiden för M6 som efter att ha löst grafen som M5 dessutom testar nya vägar att stoppa flödet och därför räknar om flera gånger.

M7 och M8 optimerar vidare och detta kostar tid. Eftersom M8 kör både M7a och M7b är det kanske förväntat att den tar längre tid, men det leder också till bra val som kortar exekveringstiden. Detta syns då den bara använder cirka 64% av den tid det tar att exekvera båda M7a och M7b på hela grafen.

M9 är nästan lika snabb som de första heuristikerna. Detta beror på att den trots att den gör många beräkningar även lyckas separera många noder i varje iteration, om än till ett högt pris.

Sammantaget kan sägas att skillnaden mellan de snabba och långsamma heuristikerna blev stor, det skiljer ca 400 gånger i körtid mellan M1 och M8. Trots detta slår M1 heuristik M8 på samtliga testsviter med 40 noder.

5.7 Slutsats och vidare arbete

I kapitel 5.5.1 visar jag att heuristik M2b ger ett bättre resultat än heuristik M2a. I kapitel 5.5.2 visas att M4b ger ett bättre resultat än M4a. Slutligen visas i kapitel 5.5.4 att M2b är bättre än M4b vilket leder till att heuristik M2b är den bästa bland de fyra heuristiker som använder diametermått. Jag har alltså bevisat att

när det gäller att bygga spanneröar så är det bättre att mäta diametern av en ö via grafens bågar än via det euklidiska avståndet.

Jämförelsen i kapitel 5.5.3 mellan M1 och M3 är inte lika klar. Analysen görs i kapitel 5.6.3 och den kommer fram till att M3 är bättre på glesa grafer medan M1 är bättre på täta.

Kapitel 5.5.5 undersöker flödesheuristikerna. Det visas här att M8 i alla lägen är den bästa flödesheuristiken. Slutligen visas i 5.5.6 att sveplinjemetoden är den i särklass sämsta heuristiken.

En sak som inte presenteras i arbetet men som kan utläsas ur testserierna är att M8 som använder sig av både M7a och M7b presterar ett högre resultat än deras minimum. Även om M7b är bättre än M7a kommer på grund av grafernas slumputseende flera fall uppstå där M7a har tur och presterar bättre. M8 har inte så stor tur att den kan mäta sig mot minimum av två heuristiker som angriper problemet olika. En lärdom av detta är att flera olika metoders minimum alltid kommer att vara bra. Om en heuristik skall skrivas kan det löna sig att blanda lösningsmetoder, då vissa metoder klarar vissa grafer speciellt bra, och sedan ta den bästa lösningen av dessa.

Bland de övriga heuristikerna har visats att M2b är bättre än M4b och M1 bättre än M3. Hur står då M1 och M3 sig mot M8? Det ser ut som (se tabell 5.29 på sidan 104) att inom varje segment blir M8 bättre ju fler bågar som finns, dvs. när grafen tätnar. M1 och M3 gynnas däremot av fler noder, dvs. när grafen blir glesare. M1 och M3 är svåra att separera, för få antal noder presterar M1 hela tiden bättre medan för de stora nodantalen återhämtar M3 sig.

Slutsatsen blir att för täta grafer bör man välja en heuristik av typ M8 medan man för glesa grafer bör välja en heuristik av typ M3. M1 hamnar utanför då den slår M3 på de typer av grafer då M8 ändå är bäst.

5.7.1 Vidare arbete

Det finns möjlighet att testa ett stort antal kombinationer av de nämnda heuristikerna, och kanske även helt nya. När det gäller de presenterade heuristikerna finns det som nämnts utrymme att förbättra sveplinjemetoden. Kanske bör snittet som läggs starta i en punkt på vektorn mellan s och t och sedan kan det växa utåt så mycket som behövs tills separation nås. Ett annat alternativ är att betrakta grafen som en polygon och då kanske snittet bara behöver skära genom en konvex del av denna. Denna idé passar bra in på problemet i figur 5.38 där endast den vänstra delen skulle skäras när s och t separeras.

Litteraturförteckning

[1] Mattias Andersson, Joachim Gudmundsson, and Christos Levcopoulos. Approximate distance oracles for graphs with dense clusters. *Comput. Geom. Theory Appl.*, 37(3):142–154, 2007.

[2] Gunnar Blom and Björn Holmquist. *Statistikteori med tillämpningar.* Studentlitteratur, Lund, www.studentlitteratur.se, 1998.

[3] Gilles Brassard and Paul Bratley. *Fundamentals of algorithmics.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[4] Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-sat. *Theor. Comput. Sci.*, 329(1-3):303–313, 2004.

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001.

[6] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil. On two problems in the generation of program test paths. *IEEE Trans. Softw. Eng.*, 2(3):227–231, 1976.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[8] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[9] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[10] Christos M. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, Massachusetts, 1994.

[11] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *PODC '87: Proceedings of the sixth annual ACM Symposium on*

Principles of distributed computing, pages 77–85, New York, NY, USA, 1987. ACM.

[12] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

Bilaga A

Redovisning av beräkningar

A.1 Antalet vägar mellan s och t

Här utreds hur många vägar det finns mellan två noder i en komplett graf. En komplett graf betraktas då den representerar det värsta fallet. Det finns en möjlig väg mellan s och t av längd 1 (med längd menas antalet bågar). Av längd 2 finns det $n-2$ olika vägar, vilken som helst av de $n-2$ noderna kan väljas som nod på vägen. Med längd 3 finns det $(n-2)(n-3)$ vägar eftersom ordningen mellan noderna längs vägen är viktig. Tabell A.1 visar hur antalet vägar ökar då längden på dem ökar.

Antal bågar	Antal vägar
1	1
2	$n-2$
3	$(n-2)(n-3)$
...	...
$n-2$	$(n-2)\cdot\ldots\cdot 2$
$n-1$	$(n-2)!$

Tabell A.1: Antalet vägar av olika längd.

Observera att de två nedersta uttrycken i tabellen är lika stora. Vidare syns ur tabellen att uttrycket vid k bågar kan skrivas $\frac{(n-2)!}{(n-k-1)!}$. Summeras detta över antalet bågar fås enligt uttryck A.1 resultatet $O((n-2)!)$. I beräkningen utnyttjas $\sum_{k=0}^{\infty} \frac{1}{k!} = e$.

$$\sum_{k=1}^{n-1} \frac{(n-2)!}{(n-k-1)!} = (n-2)! \sum_{k=1}^{n-1} \frac{1}{(n-k-1)!} = (n-2)! \sum_{k=0}^{n-2} \frac{1}{k!} \leq$$
$$\leq (n-2)! \sum_{k=0}^{\infty} \frac{1}{k!} = (n-2)! \cdot e = O((n-2)!) \qquad \text{(A.1)}$$

A.2 Tidskomplexitet för delgrafalgoritmen

Totalt sett finns 2^{n-2} olika delgrafer. Eftersom Dijkstras algoritm i värsta fall tar $O(n^2)$ tid så kommer olika mycket tid att spenderas i de olika delgraferna. Största delgrafen ger längst tid eftersom den har störst antal noder, men det finns bara en sådan. Det finns däremot flest grafer med $n/2$ antal noder och här ges en fingervisning om resultatet. I syfte att jämföra den här algoritmen med nästa kommer alla detaljer att redovisas och en exakt värstafallstid beräknas. Tabell A.2 visar en översikt över antalet grafer vid olika antal noder.

Antal noder	Antal grafer	Komplexitet för Dijkstras algoritm
0	1	0
1	$\binom{n-2}{1}$	1
2	$\binom{n-2}{2}$	2^2
3	$\binom{n-2}{3}$	3^2
...	...	...
$n-2$	$\binom{n-2}{n-2}$	$(n-2)^2$

Tabell A.2: Antalet grafer vid olika antal noder i grafen.

Uttryck A.2 och A.3 visar i två steg hur komplexiteten beräknas (0 noder har lämnats utanför beräkningen).

$$\sum_{k=1}^{n-2} \binom{n-2}{k} k^2 = (n-2) \sum_{k=1}^{n-2} \binom{n-3}{k-1} k = (n-2) \sum_{k=0}^{n-3} \binom{n-3}{k} (k+1) \qquad \text{(A.2)}$$

$$\sum_{k=0}^{n-3} \binom{n-3}{k} (k+1) = \sum_{k=0}^{n-3} \binom{n-3}{k} + \sum_{k=0}^{n-3} \binom{n-3}{k} k = 2^{n-3} +$$
$$+(n-3) \sum_{k=0}^{n-4} \binom{n-4}{k} = 2^{n-3} + (n-3)2^{n-4} = (n-1)2^{n-4} \qquad \text{(A.3)}$$

Tidskomplexiteten blir $(n-2)(n-1)2^{n-4}+1$ vilket ur asymptotisk synpunkt är $\Theta(n^2 2^n)$. Det krävs ungefär lika mycket tid att skapa delgrafen som att köra Dijkstras algoritm, ungefär $O(n^2)$. Detta påverkar inte den asymptotiska komplexiteten.

I beräkningen utnyttjas $\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}$ för att bli av med k-faktorerna.

A.3 Tidskomplexitet för splittringsalgoritmen

Rekursionsekvationen för splittringsalgoritmens tidskomplexitet blir $T(n) = n^2 + 2T(n-1)$, $T(2) = 1$. Där det tar konstant tid att köra algoritmen på bara noderna a och b. n^2 kommer från värstafallskomplexiteten av Dijkstras algoritm och kontrollen av om det är en spannerväg. Konstanten framför denna term påverkar inte komplexiteten. Uttrycket kan förenklas via

$$\begin{aligned} T(n) = n^2 + 2T(n-1) = n^2 + 2(n-1)^2 + 4(n-2)^2 + \cdots \\ \cdots + 2^{n-2} = \sum_{k=0}^{n-3} (n-k)^2 2^k + 2^{n-2} \end{aligned} \tag{A.4}$$

Här syns att summan ger rätt värde ner till dess att rekursionen avbryts, så sista värdet lämnas utanför summan. Nu förenklas summationsgränsen

$$\begin{aligned} &\sum_{k=0}^{n-3}(n-k)^2 2^k = \\ &\quad = \sum_{k=0}^{n}(n-k)^2 2^k - 2^n(n-n) - 2^{n-1}(n-(n-1)) - 2^{n-2}(n-(n-2)) = \\ &\quad = \sum_{k=0}^{n}(n-k)^2 2^k - 2^{n-1} - 2\cdot 2^{n-2} = \sum_{k=0}^{n}(n-k)^2 2^k - 2^n \end{aligned} \tag{A.5}$$

Den nyss beräknade summan kan skrivas om som

$$\sum_{k=0}^{n}(n-k)^2 2^k = n^2 \sum_{k=0}^{n} 2^k + \sum_{k=0}^{n} k^2 2^k - 2n \sum_{k=0}^{n} k 2^k \tag{A.6}$$

Dessa summor angrips nu en efter en. Uttryck A.7 visar vad som gäller för den geometriska summan.

$$n^2 \sum_{k=0}^{n} 2^k = n^2(2^{n+1} - 1) \tag{A.7}$$

De övriga summorna beräknas med hjälp av *finite calculus* som presenteras i Graham et. al [8]. Här följer en kort beskrivning av begrepp som hämtas ur denna bok.

Definition A.3.1 *x* **upphöjt till** *m* **fallande** *skrivs $x^{\underline{m}}$. Detta uttryck definieras som $x^{\underline{m}} = x(x-1)\dots(x-m+1)$.*

Finite calculus definierar egna versioner av *calculus* derivata och integralbegrepp.

Definition A.3.2 Differensoperatorn Δ *definieras enligt $\Delta f(x) = f(x+1) - f(x)$.*

Speciellt gäller att $\Delta(x^{\underline{m}}) = mx^{\underline{m-1}}$ och $\Delta(2^x) = 2^x$. Nu definieras motsvarigheter till definita och indefinita integraler.

Definition A.3.3 *För den* **indefinita summan** *gäller*

$$g(x) = \Delta f(x) \Leftrightarrow \sum g(x)\delta x = F(x) + C \tag{A.8}$$

där C är en funktion $p(x)$ sådan att $p(x+1) = p(x)$.

Definition A.3.4 *Låt $g(x) = \Delta f(x)$, då definieras den* **definita summan** *enligt*

$$\sum_{x=a}^{b} g(x)\delta x = [f(x)]_a^b = f(b) - f(a) \tag{A.9}$$

För den definita summan gäller att om f och g är som i definition A.3.4 så är

$$\sum_{x=a}^{b} g(x)\delta x = \sum_{k=a}^{b-1} g(k) \tag{A.10}$$

Definition A.3.5 Skift operatorn E *definieras enligt $Ef(x) = f(x+1)$.*

Nu kan en motsvarighet till partiell integration formuleras. Den kan antingen vara definit eller indefinit. Den indefinita versionen visas i uttryck A.11

$$\sum u\Delta v = uv - \sum Ev\Delta u \tag{A.11}$$

Om $u(x) = x$ och $\Delta v(x) = 2^x$ ger den partiella integrationen att

$$\sum x2^x\delta x = x2^x - \sum 2^{x+1}\delta x = x2^x - 2^{x+1} + C \tag{A.12}$$

Applicerat på den högra summan i uttryck A.6 så ger det

$$\sum_{k=0}^{n} k2^k = \sum_{x=0}^{n+1} x2^x\delta x = \left[x2^x - 2^{x+1}\right]_0^{n+1} = \\ = ((n+1)2^{n+1} - 2^{n+2}) - (0 \cdot 2^0 - 2^{0+1}) = (n-1)2^{n+1} + 2 \tag{A.13}$$

För att beräkna den mittersta summan i A.6 utnyttjas $x^2 = x^{\underline{2}} + x^{\underline{1}} = x^{\underline{2}} + x$. Det ger

$$\sum x^2 2^x \delta x = \sum (x^{\underline{2}} + x)2^x\delta x = \sum x^{\underline{2}}2^x\delta x + \sum x2^x\delta x \tag{A.14}$$

Här blir den första summan

$$\sum x^{\underline{2}}2^x\delta x = \\ = x^{\underline{2}}2^x - \sum 2x^{\underline{1}}2^{x+1}\delta x = (x^2 - x)2^x - 4\sum x2^x = \\ = x^2 2^x - x2^x - 4(x2^x - 2^{x+1}) + C = 2^x(x^2 - 5x + 8) + C \tag{A.15}$$

Sammantaget ger det

$$\sum x^2 2^x \delta x = 2^x(x^2 - 5x + 8) + 2^x(x-2) + C = 2^x(x^2 - 4x + 6) + C \tag{A.16}$$

Detta leder till att den mittersta summan i A.6 blir

$$\sum_{k=0}^{n} k^2 2^k = \sum_{x=0}^{n+1} x^2 2^x \delta x = \left[2^x(x^2 - 4x + 6)\right]_0^{n+1} = 2^{n+1}(n^2 - 2n + 3) - 6 \tag{A.17}$$

Nu kan uttrycket för $T(n)$ sättas samman av delarna i utryck A.7, A.13 och A.17.

$$T(n) = n^2(2^{n+1} - 1) + (2^{n+1}(n^2 - 2n + 3) - 6) - \\ -2n((n-1)2^{n+1} + 2) - 2^n = 5 \cdot 2^n - n^2 - 4n - 6 \tag{A.18}$$

Algoritmen är alltså av komplexiteten $O(2^n)$.

Om de olika beräkningsstegen studeras syns att det blir 2^{n-2} grafer med storlek 2 noder som algoritmen till slut skall köras på när rekursionen når botten. Men det finns egentligen bara en sådan graf med noderna a och b. Det samma gäller i steget innan där det är 2^{n-3} grafer av storlek 3 som skall undersökas. Men det finns bara $n-2$ grafer av den här typen så det är alltså samma grafer som återkommer

exponentiellt ofta. Det går därför att snabba upp algoritmen genom att se till så att den inte testar samma graf mer än en gång. Om algoritmen började nerifrån och beräknade först alla grafer med två noder, sedan tre osv. och hela tiden höll föregående resultat lagrat skulle algoritmen ta upp ungefär $O(2^{n-2\log(n)})$ utrymme. Detta inses genom att titta på största binomialkoefficienten som är den centrala binomialkoefficienten, $\binom{n-2}{n/2-1}$. Tabellen för alla grafer är störst då den med storlek $n/2-1$ noder nyss har byggts upp. Här används Stirlings approximation, $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$, som återfinns till exempel i Graham et al. [8]. Det leder till att

$$\binom{n}{n/2} = \frac{n!}{(n/2)!^2} \geq \frac{\left(\frac{n}{e}\right)^n}{\left(n \cdot \left(\frac{n}{2e}\right)^{n/2}\right)^2} = \frac{2^n}{n^2} = 2^{n-2\log(n)} \tag{A.19}$$

Här antas n vara ett jämnt tal för enkelhets skull. Ordo-notationen gör att detta fortfarande är $O(2^{n-2\log(n)})$ om n substitueras mot $n-2$ som gäller för den centrala binomialkoefficienten enligt ovan.

Finns det någon punkt mellan ingen tabell och en full tabellering som ger optimal tidskomplexitet, och är den bättre än $O(2^n)$?

En tabell konstrueras med resultat från alla grafer av storlek x noder. Den får storlek $\binom{n-2}{x}$ eftersom noderna a och b måste finnas med. Att exekvera den tidigare algoritmen tar för varje graf $O(2^x)$. Det ger en ny algoritm med komplexitet

$$\begin{aligned} T(n) = O(2^x)\binom{n-2}{x} + n^2 + 2(n-1)^2 + \cdots + \\ +2^k(n-k)^2 + \cdots + 2^{n-x-1}(x+1)^2 + 2^{n-x}O(x) \end{aligned} \tag{A.20}$$

Den första termen består här i att konstruera tabellen med alla resultat och den sista termen motsvarar att alla 2^{n-x} beräkningarna slås upp i en tabell. Om grafernas värde lagras i en trädstruktur så tar det i värsta fall $O(x)$ att hämta dem ur denna då trädet inte säkert är balanserat. Övriga termer är precis samma som i den tidigare beräkningen, A.4.

Uttrycket kan förenklas genom

$$\begin{aligned} n^2 + 2(n-1)^2 + 4(n-2)^2 + \cdots + 2^k(n-k)^2 + \cdots + \\ +2^{n-x-1}(x+1)^2 + 2^{n-x}O(x) = 2^{n-x}O(x) + 2^{n-x}\sum_{k=1}^{n-x}\frac{(x+k)^2}{2^k} \end{aligned} \tag{A.21}$$

Sedan kan summan begränsas ovanifrån genom

$$\sum_{k=1}^{n-x}\frac{(x+k)^2}{2^k} \leq x^2\sum_{k=0}^{\infty}\frac{1}{2^k} + 2x\sum_{k=0}^{\infty}\frac{k}{2^k} + \sum_{k=0}^{\infty}\frac{k^2}{2^k} = 2x^2 + 4x + 6 \tag{A.22}$$

Här utnyttjades att

$$\sum_{k=0}^{n} \frac{1}{2^k} = \frac{1 - \left(\frac{1}{2}\right)^n}{1 - \left(\frac{1}{2}\right)} \to 2 \quad \text{då} \quad n \to \infty \tag{A.23}$$

$$\left.\begin{array}{l} \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \\ x = 1/2 \end{array}\right\} \quad \text{ger} \quad \sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 \tag{A.24}$$

$$\left.\begin{array}{l} \sum_{k=0}^{\infty} k^2 x^k = \frac{x(x+1)}{(1-x)^3} \\ x = 1/2 \end{array}\right\} \quad \text{ger} \quad \sum_{k=0}^{\infty} \frac{k^2}{2^k} = \frac{\frac{1}{2} \cdot \frac{3}{2}}{(1-\frac{1}{2})^3} = 6 \tag{A.25}$$

Uttrycken A.23-A.25 fås från teorin för genererande funktioner som presenteras i Graham et al. [8].

Med hjälp av detta kan $T(n)$ skrivas $T(n) = O(2^x)\binom{n-2}{x} + 2^{n-x}O(x^2)$.

Beräkningen av en asymptotisk komplexitet för T påverkas inte av det som döljer sig i ordo notationen så istället betraktas

$$T(n) = 2^x \binom{n-2}{x} + x^2 2^{n-x} \tag{A.26}$$

Här syns att ett stort x ökar den första termen och minskar den andra. Det minsta värdet antas därför då de är lika stora. Det efterfrågas ett x som löser

$$2^x \binom{n-2}{x} = x^2 2^{n-x} \tag{A.27}$$

Om binomialkoefficienten betraktas så gäller med hjälp av Sterlings approximation för $x!$

$$\binom{n-2}{x} = \frac{(n-2)(n-1)\cdots(n-x+1)}{x(x-1)\cdots 2 \cdot 1} \leq \frac{n^x}{(x/e)^x} = \left(\frac{ne}{x}\right)^x \tag{A.28}$$

Här går det att få bort n ur basen genom att sätta $x = n/c$ för något c. Det ger

$$\binom{n-2}{n/c} = (ce)^{n/c} \tag{A.29}$$

Genom att uppskatta binomialkoefficienten fås en bättre approximation och ett x värde som bättre approximerar lösningen till ekvationen.

$$\binom{n-2}{x} \leq \frac{1}{n}\binom{n}{x} \leq \frac{1}{n} \cdot \frac{n \cdot \left(\frac{n}{e}\right)^n}{\left(\frac{x}{e}\right)^x \left(\frac{n-x}{e}\right)^{n-x}} \tag{A.30}$$

Här har täljaren uppskattas uppåt och nämnaren nedåt utgående från Stirlings approximation för fakultet. Om e förkortas bort och x sätts till $x = n/c$ fås

$$\frac{n^n}{\left(\frac{n}{c}\right)^{\frac{n}{c}} \left(\frac{n(c-1)}{c}\right)^{\frac{n(c-1)}{c}}} = \frac{n^n}{\frac{n^n}{c^n} \cdot (c-1)^{\frac{n(c-1)}{c}}} = \frac{c^n}{(c-1)^{\frac{n(c-1)}{c}}} \tag{A.31}$$

Nu kan ekvation A.27 betraktas i ett nytt läge, nu är den en ekvation med c som okänd eftersom $x = n/c$. Logaritmeras båda leden fås

$$\frac{n}{c} + n \log c - \frac{n(c-1)}{c} \log (c-1) = 1 - \frac{n}{c} + 2 \log \frac{n}{c} \tag{A.32}$$

Division med n ger

$$\frac{1}{c} + \log c - \frac{c-1}{c} \log (c-1) = 1 - \frac{1}{c} + \frac{2}{n} \log \frac{n}{c} \tag{A.33}$$

Här syns att för stora n spelar inte den ursprungliga x^2 faktorn så stor roll. Det är i detta steg motivationen finns för att ordo-uttrycken kunde tas bort. På grund av logaritmeringen skulle dessa bli en konstant delat med n.

Så för ekvationen gäller då $n \to \infty$ att

$$\frac{2}{c} + \log c - \frac{c-1}{c} \log (c-1) - 1 = 0 \tag{A.34}$$

Vilket ger $c \approx 5.863876183$.

Sätts detta in i $x = n/c$ så fås asymptotiskt likhet mellan de två termerna och den högra ger lättast storleken på komplexiteten som blir $O(2^{0.82946434n})$. Det kan nämnas att faktorn x^2 framför inte betyder något asymptotiskt då den är av storleksordningen $\log n$ vilket är försvinnande lite (t.ex. kan ett uttryck i n med koefficient mindre än minsta decimalen ovan kompensera för dess bortgång). Nu har en ännu bättre algoritm än den förra som var $O(2^n)$ hittats. Om den ersätter den förra i rekursionsekvationen fås en ännu bättre algoritm. Detta kan upprepas fler gånger och ger en följd av algoritmer som konvergerar mot en asymptotisk komplexitet som nu skall bestämmas. Om det i uttrycket som c löstes ut ur nu utnyttjas att det finns en algoritm som tar $2^{n(1-1/c)}$ fås följande rekursionsuttryck för allt bättre algoritmer

$$\frac{(2 - 1/c_{n-1})}{c_n} + \log c_n - \frac{c_n - 1}{c_n} \log (c_n - 1) - 1 = 0 \tag{A.35}$$

På numerisk väg beräknas att $c_n \to 5.617962663$ då $n \to \infty$. Detta ger en slutlig tidskomplexitet på $O(2^{0.82199953n})$ eller $O(1.767854485^n)$. Utrymmesmässigt krävs $\binom{n-2}{n/c}$. Hur stor är då denna term? Det gäller i uttryck A.27 att vänster led ungefär motsvarar höger och höger har beräknats till $O(2^{0.82199953n})$ så då är

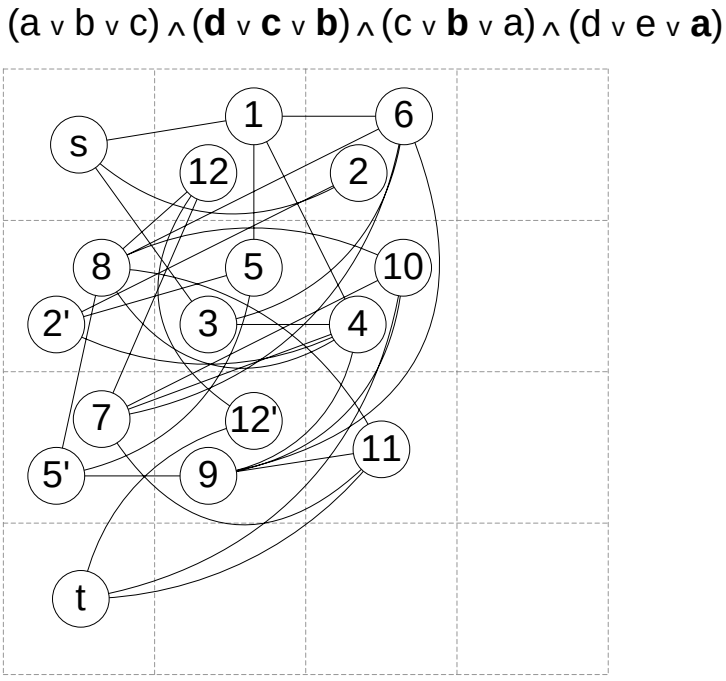
$$O(\binom{n-2}{n/c}) = O(2^{0.82199953n - n/5.617962663}) = O(2^{0.6439990580n}) = O(1.562655^n) \tag{A.36}$$

Utrymmet blir alltså något mindre än exekveringstiden, men fortfarande exponentiellt.

A.4 Jämförande exempel

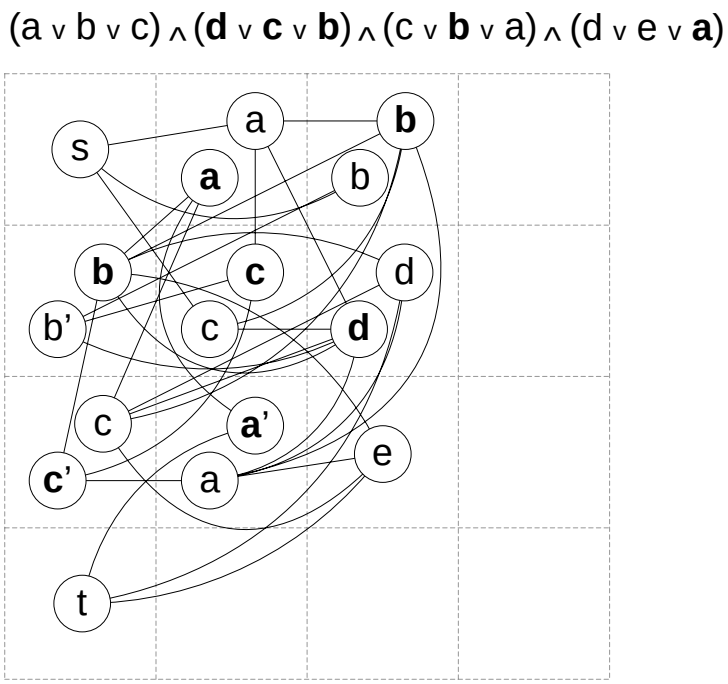
Graferna som visas för att klargöra skillnaderna mellan de olika sätten att reducera ett 3SAT3V2L problem återges med detaljer här. 3SAT3V2L uttrycket är det som ges i figur 3.5 på sidan 30.

Den första reduktionen utgår från grafen i figur 3.6 och tar bort bågar mellan literaler med motsatt värde. Dessa bågar får inte användas eftersom en väg mellan dessa skulle innebära en tilldelning av två olika sanningsvärden till en variabel. Sedan flyttas noderna om så att de förbjudna paren hamnar i samma rutor. Figur A.1 visar hur det ser ut.



Figur A.1: Reduktion via PwFP uttrycket.

Figur A.2 visar grafen med literalerna utplacerade.



Figur A.2: Direktreduktion från 3SAT uttrycket.

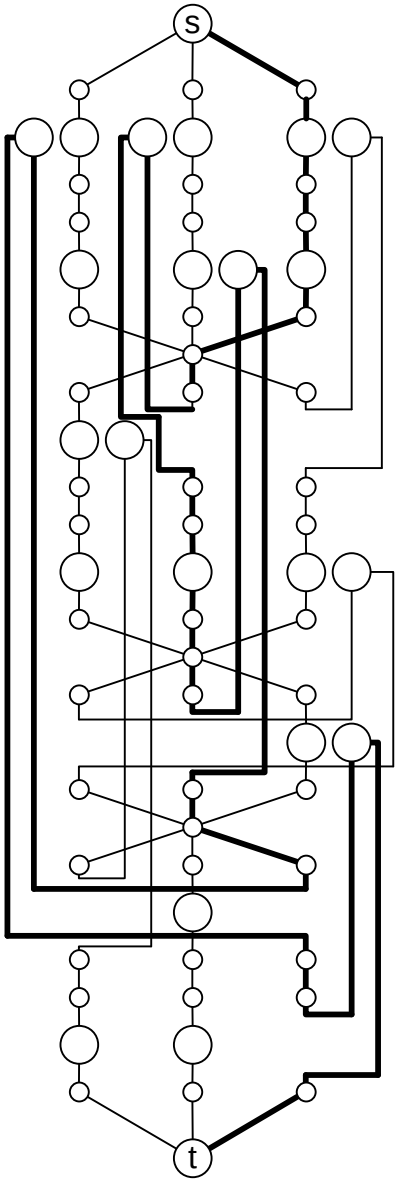
Om nodernas storlek approximeras med noll och deras avstånd inom en ruta också approximeras med noll går det lätt att räkna ut avstånden i grafen då centrumavstånden hela tiden kan användas. Om alla vägar tillåts mellan de olika lagren i figur 3.6 fås följande tabell.

Lager	Längsta båge	Längd
1	$s \to 2 \to 2'$	$2 + \sqrt{5}$
2	$2' \to 5 \to 5'$	$1 + \sqrt{2}$
3	$6 \to 7$	$2\sqrt{2}$
4	$7 \to 12 \to 12'$	$2 + \sqrt{5}$
5	$10 \to t$	$2\sqrt{2}$

Tabell A.3: Längsta bågar mellan lagren i figur 3.6.

Den längsta spannervägen uppskattas därför till summan av dessa som blir ungefär 16.54 vilket avrundas till 17. Den kortaste tillåtna vägen passerar noder som ligger i rutor intill varandra på ett avstånd av 1. Detta innebär att spannerkonstanten måste vara minst 17 för att alla möjliga lösningar skall tillåtas. Det räcker

att noder som placeras i samma ruta hamnar inom $2/17$ eftersom kortaste vägen går via en intilliggande ruta och tillbaka vilket ger vägen längd 2. Denna får inte tillåtas även om 17 gånger längre vägar tillåts. Om Θ-uttrycket använts hade det behövts en spannerkonstant på 99 för att hitta en spannerväg igenom. Det syns att denna värstafallsuppskattning hamnar långt ifrån exemplets värden.



Figur A.3: Längsta vägen genom grafen.

När det gäller direktreduktionen från 3SAT visar det sig ganska snabbt att det bara finns två långa vägar. Mätning ger dem dessutom samma längd, 32ε. Figur A.3

visar den ena av dessa. Om η väljs så $\eta = 1$ så blir $\varepsilon = \sqrt{5}$ och vägens längd $32\sqrt{5}$. Då denna väg måste tillåtas passera noder som ligger på avstånd 1, i nästa kolumn, kommer detta också att bli kravet på stretchkonstanten. Avrundat till närmsta heltal blir det 72. Bakåtnoder bör bryta mot detta om deras negation passeras. Kortaste tillåtna avståndet mellan två otillåtna bågar är $2\eta = 2$ som motsvarar att vägen går ner och vänder på nästa lager. Detta innebär att bakåtnoderna måste placeras inom $2/72 = 1/36$ från sin negation.

A.5 Beräkningar för heuristikerna

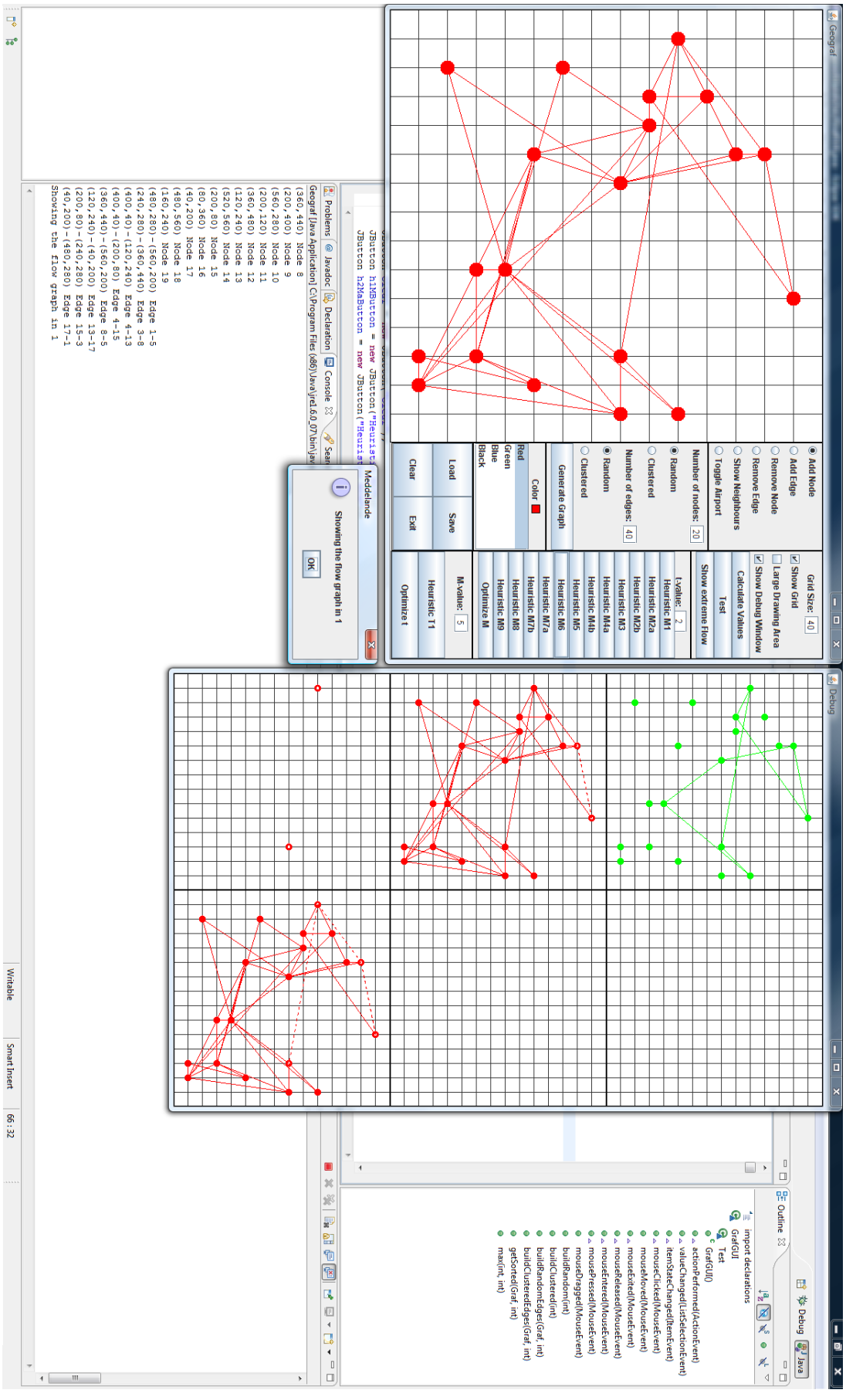
Följande metod används för att ta fram intervallskattningar för skillnaden i resultat mellan två heuristiker. Metoden ligger till grund för samtliga statistiska resultat som presenteras i kapitel 5.5.

Datan kan betraktas som ett antal stickprov i par. Det innebär att heuristikernas resultat jämförs parvis eftersom det inte går att jämföra mellan olika grafer. Då ett stort antal mätvärden används går det att utnyttja en normalapproximation för att bestämma intervallet. Analysen utgår från modellen att den mindre seriens fördelning är approximativt normalfördelad med fördelningen $N(m_j, \sigma_1)$ samt den större serien med fördelningen $N(m_j + \Delta, \sigma_2)$. Detta ger enligt Blom och Holmquist [2] intervallskattningen $I_\Delta = (\bar{z} \pm t_{\alpha/2}(f)d)$ där z är den mellan paren bildade differensen och $\bar{z}$ medelvärdet av dessa differenser. f är antalet frihetsgrader för vilket gäller att $f = n - 1$ där n är antalet par. Observera att i samtliga testserier är antalet frihetsgrader så stort att t-fördelningen sammanfaller med normalfördelningen. Vidare gäller att d är ett mått på spridningen vilket fås från den uppmätta standardavvikelsen genom $d = s/\sqrt{n}$. Signifikans *** eller 0.001 används då påståenden ges.

Ett Δ_{min}-värde ur tabellen för jämförelse mellan M2a och M2b, se tabell 5.21 på sidan 99, beräknas nu som exempel. Mätvärdena hämtas ur första raden i tabell 5.4. Intervallskattning av Δ ger $I_\Delta = (0.18 \pm 0.047)$. Med signifikans *** gäller alltså att metoden M2b i genomsnitt är minst 0.13 flygplatser bättre per graf på randomiserade grafer av typen *14 noder 24 bågar*. På samma vis ger motsvarande beräkningar på de andra typerna av grafer till slut tabell 5.21. Övriga intervall och Δ-värden som återfinns i de andra tabellerna i kapitel 5.5 beräknas på liknande sätt.

A.6 Verktyget

Följande blid visar det grafiska verktyg som använts i kapitel 5 för att generera grafer och köra test. Utförlig testning och debugging av heuristiker har även utförts via detta verktyg som är utvecklat med Eclipse SDK version 3.4.0.


Add Node
Add Edge
Remove Node
Remove Edge
Show Neighbours
Toggle Airport
Grid Size: 40
Show Grid
Large Drawing Area
Show Debug Window
Calculate Values
Test
Show extreme Flow
Number of nodes: 20
Random
Clustered
Number of edges: 40
Random
Clustered
Generate Graph
t-value: 2
Heuristic M1
Heuristic M2a
Heuristic M2b
Heuristic M3
Heuristic M4a
Heuristic M4b
Heuristic M5
Heuristic M6
Heuristic M7a
Heuristic M7b
Heuristic M8
Heuristic M9
Optimize M
M-value: 5
Heuristic T1
Optimize t
Color
Red
Green
Blue
Black
Load
Save
Clear
Exit
Meddelande
Showing the flow graph in 1
OK
(360,440) Node 8
(200,400) Node 9
(560,280) Node 10
(200,120) Node 11
(360,480) Node 12
(120,240) Node 13
(520,560) Node 14
(200,80) Node 15
(80,360) Node 16
(40,200) Node 17
(480,560) Node 18
(160,240) Node 19
(480,280)-(560,200) Edge 1-5
(240,280)-(360,440) Edge 3-8
(400,40)-(120,240) Edge 4-13
(400,40)-(200,80) Edge 4-15
(360,440)-(560,200) Edge 8-5
(120,240)-(40,200) Edge 13-17
(200,80)-(240,280) Edge 15-3
(40,200)-(480,280) Edge 17-1
Showing the flow graph in 1
Debug
Java
Outline
Writable
Smart Insert
66 : 32