Maksimal parring i todelte grafer
Obligatorisk opgave i DADS

Martin Geisler <gimpster@gimpster.com>
Jérémy Auneau <jeremy.auneau@skjoldhoej.dk>

Uge 13. maj 2002

Indhold

Kapitel 1

Introduktion og terminologi

Den obligatoriske opgave går ud på at skrive et Java-program, der kan finde en *størst mulig pardannelse* i en ikke-orienteret, todelt graf. Opgaven er opdelt i tre delopgaver, som lægger op til en skridtvis konstruktion af et Java-program. Disse delopgaver løses i de næste tre kapitler: Først konstrueres en abstrakt algoritme, som bevises korrekt; dernæst designes der effektive datastrukturer; og til sidst foretages der en implementation i Java. Det færdige program vil blive brugt til at løse to konkrete skemalægningsproblemer hentet fra *det virkelige liv*.

1.1 Parringer, dækninger og parringsgrafer

En ikke-orienteret graf $G = (V, E)$ med n knuder og m kanter er *todelt*, hvis knudemængden V kan deles i to disjunkte delmængder V_1 og V_2, således at alle kanterne i E forbinder en knude i V_1 med en knude i V_2. Vi vil sommetider angive G på formen $G = (V_1, V_2, E)$ når vi ønsker at markere, at knudemængden V er foreningen af to disjunkte knudemængder V_1 og V_2.

En *parring* er en delmængde P af kanterne i E, hvor alle kanterne i P er knudedisjunkte.

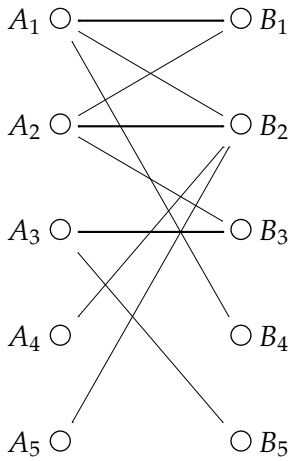
En *dækning* er en delmængde D af knuderne i V, således at alle kanterne i E er incidente med mindst én knude fra D.

Grafen i figur 1.1 på den følgende side er et eksempel på en todelt graf. De fede kanter angiver parringen $P = \{\{A_1, B_1\}, \{A_2, B_2\}, \{A_3, B_3\}\}$ af størrelse 3. Et eksempel på en dækning er knuderne $D = \{A_1, A_2, A_3, B_2\}$.

For en vilkårlig parring P og dækning D gælder der, at enhver kant i P er dækket af en knude i D, hvilket giver os følgende simple men meget brugbare observation:

$$|P| \leq |D|.$$

For en givet todelt graf G og parring P definerer vi den tilhørende *parringsgraf* til at være en orienteret graf $G' = (V, E')$, hvor kanterne i E' nu



Figur 1.1: Et eksempel på en todelt graf. Det er kun de fede kanter, der indgår i parringen P.

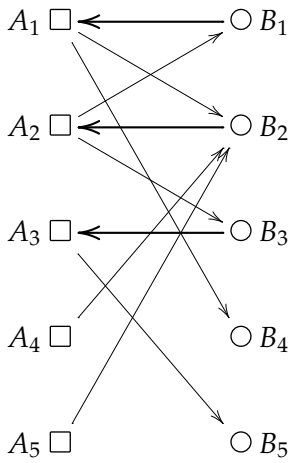
er orienterede. Kanter, der indgår i parringen P, er orienteret fra V_2 til V_1 og de øvrige kanter er orienteret fra V_1 til V_2 — vi vil lade kanterne der indgår i parringen være federe end de normale kanter. Knuder i V_1 lader vi være firkantede og knuder i V_2 runde. For grafen i figur 1.1 har vi således parringsgrafen på figur 1.2 på den følgende side. I dette tilfælde er der lige mange firkantede og runde knuder ($n_1 = n_2$ hvor $n_1 = |V_1|$ og $n_2 = |V_2|$) men det er ikke altid tilfældet.

Vi kalder de firkantede knuder med indgrad 0 for *initialknuder*, mens de runde knuder med udgrad 0 kaldes *terminalknuder*. En vej $\square - - \succ \bigcirc$ fra en initialknude til en terminalknude kaldes en *skiftevej*.

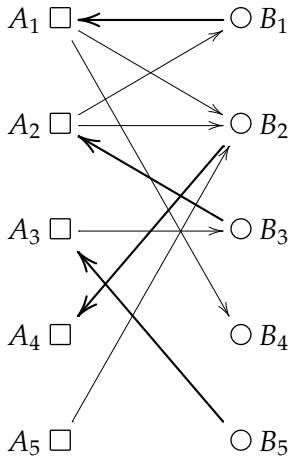
Det er nemt at indse, at hvis der findes en skiftevej i en parringsgraf, så kan man, ved at ændre orienteringen på alle kanterne på skiftevejen, opnå en parring der er én større. Argumentet er, at hvis der på skiftevejen er p parringskanter der går fra V_2 til V_1, så må der være $p + 1$ kanter der går fra V_1 til V_2. Der vil altid være denne forskel på 1, da vi totalt set bevæger os fra V_1 til V_2. Når vi vender alle knuderne om, så ser man, at der nu vil være p kanter der går fra V_1 til V_2 mens der er $p + 1$ parringskanter fra V_2 til V_1.

I grafen i figur 1.2 på næste side udgør $(A_4, B_2, A_2, B_3, A_3, B_5)$ en skiftevej, og hvis vi ændrer orienteringen langs denne vej, så for vi parringsgrafen i figur 1.3 på den følgende side, der repræsenterer en parring, der er én større.

Den obligatoriske opgave går ud på at bruge denne teknik til at finde den størst mulige parring. Vi bemærker, at den sidst opnåede parring indeholder fire kanter, og at den derfor er størst mulig idet der også findes en dækning i grafen bestående af fire knuder.



Figur 1.2: Parringsgrafen hørendende til grafen i figur 1.1.



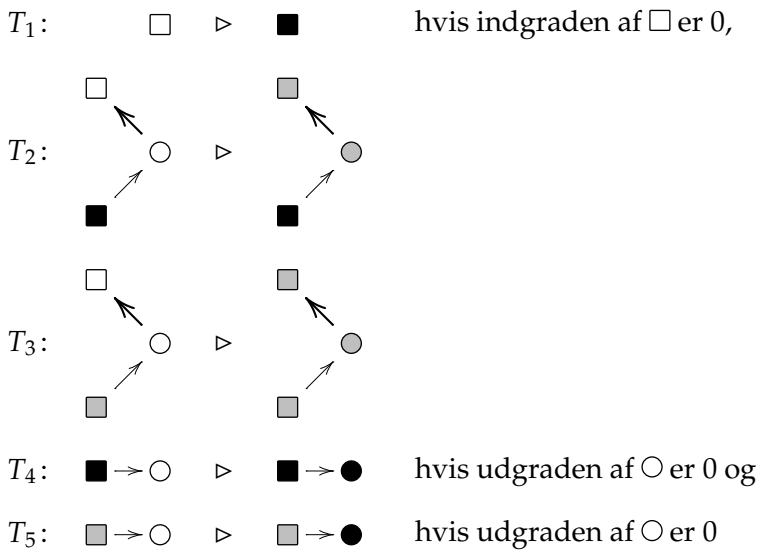
Figur 1.3: Parringsgrafen efter at orienteringen er ændret langs alle kanterne på skiftevejen.

Kapitel 2

Konstruktion af algoritmen

Vi vil i denne del af rapporten konstruere algoritmen til at finde den størst mulige pardannelse. Algoritmen benytter sig af et transitionssystem der farver knuder i parringsgrafer. Ufarvede knuder kaldes *hvide*, skraverede knuder *grå*, og udfyldte knuder er *sorte*. De farvede grafer kaldes *brogede*.

De fem transitioner der indgår i transitionssystemet er:



2.1 Tre udsagn om transitionssystemet

Vi lader nu P være en parring i en graf G med tilhørende parringsgraf G'. I en slutkonfiguration for en proces, der starter med den hvide graf G', gælder der følgende:

U_1**:** Alle grå og sorte knuder kan nås fra en firkantet sort knude,

U_2**:** Antallet af kanter i P er lig summen af hvide firkanter og grå cirkler i parringsgrafen.

U_3**:** Knudemængden bestående af □, ◍ og ● dækker alle kanter i G.

Vi skal bruge de tre udsagn U_1, U_2 og U_3 til at argumenterer for algoritmens korrekthed, og vil derfor starte med at bevise dem.

2.1.1 Bevis for U_1

Vi viser U_1 ved at vise, at det faktisk er en invariant under udførslen af transitionssystemet. Når vi starter med en parringsgraf G', så er alle knuder hvide, og U_1 holder derfor trivielt. Antag nu at U_1 holder — vi vil vise, at U_1 også holder efter en af transitionerne. Der er fem tilfælde:

1. Da vi ikke skaber nye grå knuder eller sorte cirkler bevares U_1.
2. De to nye grå knuder kan begge nås fra den sorte firkant, og U_1 er således bevaret.
3. Ifølge vores antagelse, så kan den grå firkant nås før transitionen. Derfor kan de to nye grå knuder også nås, da de ligger i forlængelse af en sti der går fra en sort firkant til den første grå knude.
4. Den nye sorte cirkel kan nås fra den sorte firkant.
5. Den grå firkant kan ifølge vores antagelse nås fra en sort firkant. Derfor kan den sorte cirkel også nås efter transitionen, da den ligger i forlængelse af stien fra den sorte firkant til den grå firkant.

Vi har nu vist, at U_1 bevares af alle transitionerne i transitionssystemet samt, at den er sand når vi starter. Derfor er U_1 også sand i slutkonfigurationen.

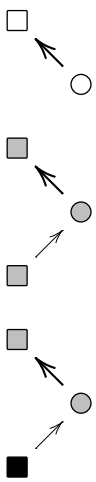
2.1.2 Bevis for U_2

Vi skal vise $U_2 : |P| =$ antal □ + antal ◍. Vi vil først vise, at hver □ giver anledning til en parringskant. Hvis der findes en hvid firkant når transitionssystemet er kommet til en død konfiguration, så ved vi, at indgraden er større end 0, idet den første transition ellers ville kunne bruges.

Vi ved også, at indgraden er lig 1, der ikke går kanter fra to forskellige cirkler til samme firkant. Derfor er den indgående kant en parringskant.

De grå cirkler giver også alle anledning til en parringskant. Det skyldes, at de kun fremkommer som resultat af transition 2 og 3. Man ser, at der i begge tilfælde går en parringskant fra en grå cirkel til en grå firkant.

Disse to typer af kanter er de eneste der kan gå fra højre til venstre. Enhver kant fra højre mod venster må starte i enten en ○, ◍ eller ●.



Hvis kanten starter i en hvid cirkel, så må den ende i en hvid firkant, da der ikke er nogen transition der farver en hvid firkant når dens indgående kant kommer fra en hvid cirkel, og alle firkanter har enten indgrad 0 eller 1. Dette tilfælde er altså dækket af U_2.

Kanten kan også starte i en grå cirkel, og vil så ende i en grå firkant ifølge transition 2 og 3. Igen ser vi, at U_2 tager dette tilfælde med.

Endelig kan kanten *ikke* starte i en sort cirkel, da transition 4 og 5 fortæller os, at alle sorte cirkler har udgrad 0.

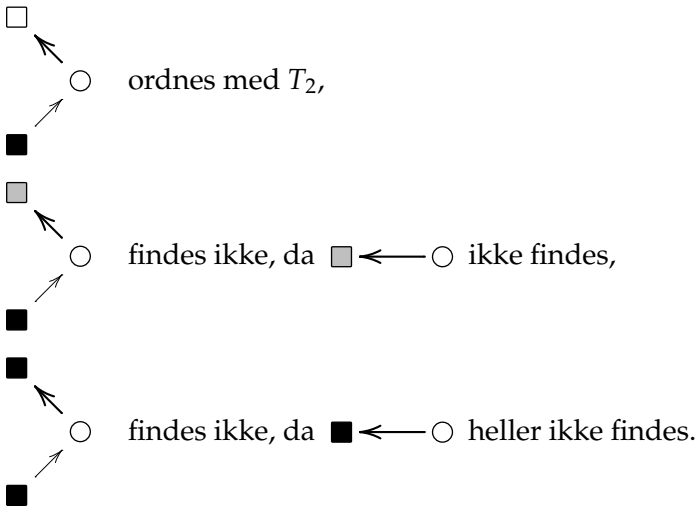
2.1.3 Bevis for U_3

Vi viser, at U_3 holder i slutkonfigurationen ved at gennemføre et modstridsargument. Der er nemlig fire kanter der ikke må forekomme hvis U_3 skal være sand: □ ⟶ ○, ■ ⟶ ○, □ ⟵ ○ og ■ ⟵ ○. Vi starter med den sidste mulighed først:

Kanten ■ ⟵ ○ kan ikke forekomme i slutkonfigurationen da enhver ■ har indgrad 0, idet alle ■ stammer fra □ med indgrad 0.

Kanten □ ⟵ ○ kan heller ikke forekomme i en slutkonfiguration. Det skyldes, at □ så kun skulle have indgående kanter fra ○. Men □ bliver kun lavet af transitionerne 2 og 3, og i begge tilfælde har □ en indgående kant fra ●.

Kanten ■ ⟶ ○ findes ikke, da ○ i så fald skulle have en udgrad der var større end 0, da T_4 ellers ville kunne anvendes. Det giver tre muligheder:



Kanten □ ⟶ ○ findes heller ikke — det indses ved at bruge samme argumentation som ved ■ ⟶ ○.

2.2 Bevis af algoritmens gyldighed

Det er nu på tide at præsentere den algoritme vi vil bruge til at finde den maksimale parring:

Algoritme 1 StørstMuligPardannelse(G)

Input: En ikke-orienteret, todelt graf G
Output: P der er den størst mulige parring i G
  Lad G' være en vilkårlig parringsgraf for G
  Farv G' ved hjælp af transitionssystemet
  // Invariant: $I = (U_1 \wedge U_2 \wedge U_3)$
  while der findes en vej ■ – – ➢ ● i G **do**
    Vend orienteringen af alle kanter langs ■ – – ➢ ●
    Farv alle knuderne i G' hvide
    Farv G' ved hjælp af transitionssystemet.
  end while
  Sæt P lig mængden af kanter i G' der går fra højre mod venstre

Vi skal argumentere for, at invarianten er gyldig og for, at algoritmen er korrekt. Den første bevisbyrde er $\{In\}C\{I\}$ hvor $\{In\}$ er inputkravene og C er kommandoen der farver G' ved hjælp af transitionssystemet. Men det er trivielt, da vi netop har bevist, at $I = (U_1 \wedge U_2 \wedge U_3)$ er sand når transitionssystemet, der farver grafen, er nået til en slutkonfiguration.

Vi skal nu vise, at **while**-løkken er korrekt. En bevisbyrde af formen $\{U\}$**while** b **do** $C\{V\}$ giver anledning til følgende tre bevisbyrder:

basis:	$U \Rightarrow I$
invarians:	$\{I \wedge b\}C\{I\}$
konklusion:	$I \wedge \neg b \Rightarrow V$.

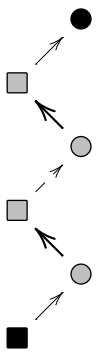
Her er U blot lig I, I er vores invarians og V er outputkravene.

Basis er derfor triviel: $I \Rightarrow I$. Andet trin, invarians, følger også let. Vi skal vise, at

$$(I(G') \wedge \text{der findes en vej } \blacksquare - - \succ \bullet \text{ i } G) \Rightarrow I(\text{opdateret } G').$$

Det sidste vi gør i **while**-løkken er, at farve den opdaterede G' ved hjælp af transitionssystemet. Vi ved, at I holder hver gang vi har anvendt transitionssystemet, så det viser implikationen.

Man skal dog lige undersøge, om den opdaterede G' virkelig er en parringsgraf. For det første kan man nemt konstatere, at hver anden kant på skiftevejen er en parringskant, se figuren til højre. Med andre ord har knuderne på skiftevejen skiftevist indgrad 1 (startpunktet på en parringskant) og udgrad 1 (endepunktet på en parringskant), hvis man ser bort fra initialknuden med indgrad 0 og terminalknuden med udgrad 0.

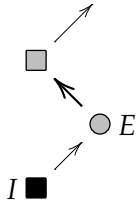


Ændringen af orienteringen på alle kanterne langs skiftevejen medfører, at alle kanter, der oprindeligt pegede mod højre, nu peger mod venstre og de er dermed blevet til parringskanter, forudsat, at de stadig opfylder kravet om at være knudedisjunkt med alle andre parringskanter i parringsgrafen. Tilsvarende er de oprindelige parringskanter nu orienterede i den modsatte retning, dvs. mod højre, og de er derfor ikke længere parringskanter.

De nye kanter der peger fra højre mod venstre skal opfylde, at deres startpunkt har udgrad 1, og at deres endepunkt har indgrad 1. Hvis de ikke gør det, så betyder det, G' ikke længere er en parringsgraf, og derfor vil transitionssystemet heller ikke kunne anvendes på grafen.

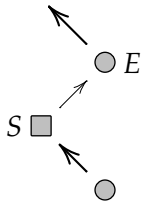
Før reorinteringen var disse kanter delt i 4 grupper, og i det følgende vil vi se på parringsgrafen før kanterne blev vendt om:

- Startpunktet I er initialknuden (indgrad 0) og endepunktet E er startpunktet for en parringskant (udgrad 1):

  Efter reorienteringen af kanten får initialknuden indgrad $0 + 1 = 1$ og dens modsatte knude E får udgrad 2. Parringskanten med startpunkt i E bliver også reorienteret, da den ligger på skiftevejen, og så bliver udgraden af E lig $2 - 1 = 1$. Kanten (E, I) peger nu fra højre mod venstre og opfylder, at dens startpunkt E har udgrad 1, og at dens startpunkt I har indgrad 1. Altså er kanten (E, I) en parringskant.

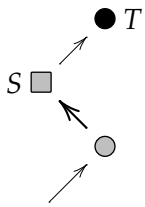
- Startpunktet S er endepunktet for en parringskant (indgrad 1) og endepunktet E er startpunktet for en parringskant (udgrad 1):

  Parringskanten med endepunkt i S reorienteres og dermed falder S's indgrad til 0. Derefter vendes kanten fra S til E, således at indgraden af S bliver 1 og udgraden af E bliver $1 + 1 = 2$. Parringskanten med startpunkt i E vendes og dermed bliver dens udgrad igen $2 - 1 = 1$.

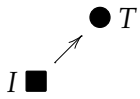
  Kanten fra E til S har efter reorienteringen et startpunkt med udgrad 1 og et endepunkt med indgrad 1. Desuden peger den fra højre mod venstre, den opfylder derfor kravet for at være en parringskant.

- Startpunktet S er endepunktet for en parringskant (indgrad 1) og endepunktet T er terminalknuden (udgrad 0):

  Parringskanten med endepunkt i S vendes, således at indgraden af S bliver 0. Derefter roteres kanten fra (S, T), og indgraden af S bliver lig $0 + 1 = 1$ og udgraden af T bliver $0 + 1 = 1$. Ligesom for tilfælde nummer to kan vi konkludere, at kanten (T, E) er en parringskant.

- Startpunktet I er initialknuden og endepunktet T er terminalknuden:

  Efter reorienteringen af kanten (I, T) er indgraden af I lig $0 + 1 = 1$ og udgraden af T er $0 + 1 = 1$. Dermed er (T, I) en parringskant.

Efter transformationen har vi en graf hvor kanterne der peger mod venstre er knudedisjunkte og en sådan graf er per definition en parringsgraf.

Vi skal nu vise er konklusionen i **while**-løkken:

$$(I \wedge \text{der findes ikke en vej } \blacksquare - - \succ \bullet \text{ i } G) \Rightarrow$$
$$P \text{ er største mulige parring.}$$

Da der ikke findes en vej $\blacksquare - - \succ \bullet$ i G, så findes der heller ikke nogen sorte cirkler. Det skyldes U_1, der specielt siger, at enhver sort cirkel vil kunne nås fra en sort firkant, hvilket ville betyde, at der alligevel var en vej $\blacksquare - - \succ \bullet$ i G.

For at vise, at parringen P er den størst mulige parring efter at algoritmen er afsluttet, så antager vi (for at få en modstid), at der findes et P' sådan, at $|P'| > |P|$. Vi ved fra U_3, at knudemængden bestående af $\square$, $\circ$ og $\bullet$ dækker alle kanter i G. De udgør dermed en dækning D. Men da der ikke er nogen sorte cirkler tilbage når algoritmen er slut, så har vi ifølge U_2, at $|P| = |D|$.

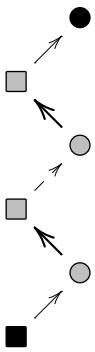
Vi ved også, at $|P| \leq |D|$ for enhver parring P. Der gælder altså:

$$|D| = |P| < |P'| \leq |D|,$$

Dermed har vi fået vores modstrid, og vi konkluderer, at P var den maksimale parring i G.

Vi har nu vist at algoritmen er korrekt — vi mangler stadig at vise, at den er gyldig, altså at den terminerer, hvilket er det samme som at vise, at **while**-løkken terminerer, da vi antager, at udførslen af de andre linier i algoritmen terminerer. Vi skal senere se, at vores algoritme der farver grafen terminerer, ligesom vi skal se, at algoritmen der vender kanterne langs skiftevejen terminerer.

Som vores termineringsfunktion μ vil vi bruge antallet af kanter der går fra venstre mod højre, altså de kanter der ikke er parringskanter. Antallet er et ikke-negativt heltal og det aftager for hvert gennemløb af **while**-løkken. μ aftager, da kanterne langs skiftevejen bliver vendt om i hvert gennemløb af **while**-løkken. Hvis der er n kanter fra venstre mod højre langs skiftevejen, så er der $n-1$ kanter fra højre mod venstre. Når kanterne vendes bliver der kun $n-1$ kanter fra venstre mod højre.



Da antallet af kanter aldrig bliver negativt konkluderer vi, at **while**-løkken terminerer, og dermen terminerer hele algoritmen også.

2.3 Bevis for Königs sætning

Sætningen siger:

> I enhver todelt graf er størrelsen af den maksimale parring lig størrelsen af den minimale dækning.

Vi ved at vi kan finde den maksimale parring P i en graf G ved at udføre vores algoritme. Vi har set, at parringen giver anledning til en dækning D og at $|P| = |D|$. Vi antager nu for en modstrid, at der findes en dækning D' sådan, at $|D'| < |D|$. Men det ville betyde at

$$|P| \leq |D'| < |D| = |P|,$$

hvilket giver os den ønskede modstrid. Derfor er D den minimale dækning.

Kapitel 3

Design af datastrukturer

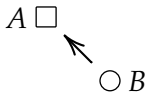
Den abstrakte algoritme, der blev udviklet i kapitel 2, skal implementeres i JAVA. Vi skal derfor designe en hensigtsmæssig og effektiv realisation af parringsgrafen $G' = (V_1, V_2, E')$. Målet er, at kunne

- Farve G' ved hjælp af transitionssystemet med en udførselstid på $O(m + n)$ og
- Vende kanterne langs en skiftevej med en udførselstid der er proportional med længden af ■ – – ➢ ● , altså $O(|m'|)$ hvor m' er kanterne der indgår i skiftevejen.

3.1 Repræsentation af parringsgrafen G'

Vi vil repræsentere G' ved hjælp af to lister, der indeholder henholdsvis knuderne V_1 og V_2. De indgående og udgående kanter for hver knude er gemt i selve knuden.

Denne struktur vil være en *naboliste*, idet hver knude har lister over de indgående og udgående kanter, samtidig med, at hver kant ved hvilke to knuder den forbinder. Vi vil altså kunne gå langs en parringskant fra en knude $B \in V_2$ til den tilhørende knude i $A \in V_1$ i tid $O(1)$, idet knuden B kun har én udgående kant. På samme måde kan vi gå tilbage langs en parringskant fra A til B i tid $O(1)$.



Når vi har fat i en kant, så vil vi også kunne vende denne om i tid $O(1)$, da vi kan fjerne og tilføje en kant til en knude i konstant tid. Det skyldes, at listerne med de ind- og udgående kanter i hver knude, understøtter de såkaldte *locators*.

3.2 Farvning af en parringsgraf

Vi farver grafen når vi kører transitionssystemet, hvilket gøres ved hjælp af algoritme 2 på næste side: runTransitionSystem(G').

Algoritme 2 runTransitionSystem(G')

Input: En parringsgraf $G' = (V_1, V_2, E')$
Output: Den farvede graf G' samt en tilfældig terminalknude V_0

```
    M1 ← ∅ // Ventende knuder fra V1
 2: M2 ← ∅ // Ventende knuder fra V2
    // Vi starter med at finde initialknuderne
 4: for all V ∈ V1 do // Transition 1
      if V har indgrad 0 then
 6:     Farv V sort
        for all E ∈ V.outEdges do
 8:       M2 ← M2 ∪ {E.to}
        end for
10:   end if
    end for
12: // Vi har nu fundet og farvet initialknuderne
    while M1 ≠ ∅ ∨ M2 ≠ ∅ do // Der er stadig knuder i M1 eller M2
14:   while M1 ≠ ∅ do // Transition 2 og 3
        Vælg et V ∈ M1
16:     M1 ← M1 \ {V}
        Farv V grå
18:     for all E ∈ V.outEdges do
          if E.to er hvid then // Vi kigger kun på ubesøgte knuder
20:         M2 ← M2 ∪ {E.to}
          end if
22:     end for
      end while
24:   while M2 ≠ ∅ do // Transition 2, 4 og 5
        Vælg V ∈ M2
26:     M2 ← M2 \ {V}
        if V har udgrad 0 then // Transition 4 og 5
28:       Farv V sort
          V0 ← V // En tilfældig terminalknude
30:     else // Transition 2 og 3
          Farv V grå
32:       for all E ∈ V.outEdges do
            if E.to er hvid then // Vi kigger kun på ubesøgte knuder
34:           M1 ← M1 ∪ {E.to}
            end if
36:       end for
        end if
38:   end while
    end while
```

Algoritmen arbejder med to mængder M_1 og M_2. Disse mængder indeholder knuder fra henholdsvis V_1 og V_2, og knuderne har den egenskab, at de kan bruges i en transition.

Når grafen er helt hvid er det kun transition 1 der kan anvendes. Derfor starter vi i linie 4–11 med at finde og farve alle initialknuder sorte. Alle knuder, der kan nås fra en initialknude, kan nu indgå i en transition og de gemmes derfor i M_2 i linie 8.

Vi har nu fået startet en eller flere skifteveje. Anden del af algoritmen gentages indtil der ikke længere dukker nye knuder op, altså så længe bare en af M_1 og M_2 indeholder ventende knuder. Første gang vil M_1 være den tomme mængde, men M_2 vil indeholde naboknuderne til initialknuderne.

Fra linie 14 til 23 gennemgås knuderne i M_1. Vi starter med at vælge og fjerne en tilfældig knude i M_1. Knuderne er alle firkanter fra V_1, og de skal alle farves grå, idet de er den øverste firkant i enten transition 2 eller 3. Knuderne i M_1 er nemlig indsat i linie 33, så vi ved at de har en indgående kant fra en cirkel.

I linie 24–37 bliver knuderne i M_2 gennemgået. Disse knuder har alle en indgående kant, da de enten stammer fra linie 8 eller linie 20. Vi skal derfor blot undersøge, om deres udgrad er 0 sådan at vi kan anvende transition 4 eller 5. Hvis deres udgrad er forskellig fra , så anvendes transition 2 eller 3 og alle naboknuderne samles i M_1.

Hele algoritmen har udførselstid $O(m + n)$. Vi besøger hver knude højst én gang, da en knude aldrig indsættes i M_1 eller M_2 hvis den er farvet, hvilket indikerer at den allerede er besøgt. Hver gang vi besøger en knude, så følger vi alle de udgående kanter fra knuden. I en orienteret graf kan to knuder ikke dele en udgående kant, så det betyder, at hver kant også kun besøges højst én gang. Den totale udførselstid bliver derfor $O(m + n)$.

3.3 Opdatering af skiftevej

Når vi skal opdatere kanterne langs en skiftevej i en farvet graf, så bliver vi nødt til at lave et konstant stykke arbejde ved hver kant, ellers kan vi ikke opnå en tidskompleksitet på $O(|m'|)$ hvor m' er de kanter vi skal vende.

3.3.1 Den næsten rigtige turnEdges

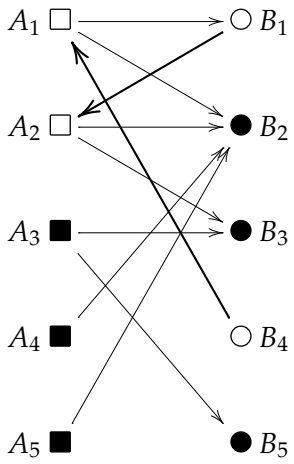
Vores første algoritme tog udgangspunkt i en terminalknude, hvorefter den så prøvede at finde vej tilbage mod en initialknude. Terminalknuden skulle komme fra runTransitionSystem, der jo netop farver terminalknuderne sorte og derfor sagtens kan returnere én af dem til senere brug.

Vi mente, at vi kunne gå tilbage langs en tilfældig indgående kant til terminalknuden, da alle indgående kanter skulle komme fra enten en sort

eller en grå firkant. Vores U_1 siger så, at denne firkant kan nås fra en sort firkant, altså er vi på vej mod en initialknude.

Vi ville fortsætte på samme måde, idet den indgående kant i en grå firkant kommer fra en grå cirkel som så også kan nås fra en sort firkant ifølge U_1. Da der er et endeligt antal grå knuder, så vil processen til sidst terminere.

Men det viste sig, at de indgående kanter i de sorte og grå cirkler godt kan komme fra en hvid firkant. I figur 3.1 har B_3 to indgående kanter: én kant fra den hvide A_2 og én fra den sorte A_3. Vi kan derfor ikke uden yderligere information slutte os til, hvilket af de to kanter vi skal følge for at komme til en initialknude.



Figur 3.1: Et eksempel på, at man ikke uden videre kan afgøre hvilket vej man skal følge fra en terminalknude B_3 for at komme til en initialknude A_3.

3.3.2 Den forkerte turnEdges

Vi skiftede derfor taktik, og prøvede så i stedet på at finde en skiftevej givet en initialknude, altså en sort firkant. Ifølge transition 2 og transition 4 går der altid en kant fra en sort firkant til en sort eller grå cirkel. Det samme gælder for grå firkanter, på grund af transition 3 og 5. Desuden er grå cirkler altid startpunkt for én og kun én parringskant rettet mod venstre. Algoritmen ville gå igennem tre trin, hvor den første gang starter ved den sorte initialknude:

1. Vælg en tilfældig udgående kant hen til en grå eller sort cirkel.
2. Hvis endepunktet er en sort cirkel, har vi fundet en skiftevej.

3. Hvis endepunktet er en grå cirkel, så er denne grå cirkel nødvendigvis startpunktet for en parringskant, der er rettet mod venstre. Følg denne parringskant, og start forfra med udgangspunkt i den nye firkant.

Denne algoritme virker desværre heller ikke. Problemet er, at kan vi ikke være sikre på, at vi ender i en sort cirkel. Vi ved fra U_1, at alle grå og sorte knuder kan nås fra en sort firkant, men det betyder *ikke* at man altid kan nå en sort knude fra en sort firkanter. En enlig firkant med indgrad 0 vil jo blive farvet sort af T_1, men den er ikke startpunkt for en skiftevej.

Der er desuden mulighed for, at algoritmen stopper ved en grå firkant, som ikke har udgående kanter. Vi er derfor tilbage i en situation hvor vi skal undersøge mere end én udgående kant for hver knude, hvilket resulterer i, at udførelsestiden ikke bliver proportional med længden af skiftevejen sådan som opgaven kræver det.

3.3.3 Den rigtige turnEdges

Vi kan nu konstatere, at vi bliver nødt til at starte ved en terminalknude, da disse er de eneste faste holdepunkter vi har for en skiftevej. Og ved at benytte U_1 ved vi, at vi er på rette vej, så længe vi går til en grå knude.

Men vi har brug for ekstra information for at kunne vælge den rigtige indgående kant når vi står ved en cirkel. Det essentielle er, at vi kan vælge den rigtige kant i tid $O(1)$. Vi modificerer derfor runTransitionSystem sådan, at den gemmer information om den kant der ledte op til hver grå og sort cirkel med. Når runTransitionSystem farver en cirkel grå eller sort, så har den fulgt en af de sidste fire transitioner, og i alle tilfælde vil vi kunne følge den indgående kant tilbage mod en grå eller sort firkant.

Hvis vi kommer tilbage til en sort firkant, så er vi færdige. Ellers står vi ved en grå firkant der har en indgående kant fra en grå cirkel, da de grå firkanter kun laver ved transition 2 og 3. Vi står nu igen ved en grå cirkel, og processen kan så fortsætte.

Den endelige turnEdges (se algoritme 3 på næste side) får derfor en udførselstid på $O(|m'|)$, idet den for hver cirkel på skiftevejen direkte kan følge den rigtige kant tilbage til en firkant. Der er præcis én indgående kant til hver firkant på skiftevejen, så der behøver vi ikke nogen ekstra information — vi kan direkte gå tilbage til en den rigtige cirkel.

Algoritme 3 turnEdges(G', V_0)

Input: En parringsgraf G' og en terminalknude V_0 i G'
Output: G' med kanterne langs en tilfældig skiftevej til V_0 vendt om

```
    M ← ∅ // Kanter der skal vendes om
 2: E1 ← V0.prev
    V1 ← E1.from
 4: M ← M ∪ {E1}
    while V1 ikke er sort do
 6:    Vælg E2 ∈ V1.outEdges // Der er kun én
       V2 ← E2.from
 8:    Vælg E1 ∈ V2.prev
       V1 ← E1.from
10:    M ← M ∪ {E1, E2}
    end while
12: for all E ∈ M do
       E ← (E.to, E.from)
14: end for
```

Kapitel 4

Implementation i Java

Vi skal nu implementere og afprøve den abstrakte algoritme udviklet i kapitel 2 ved hjælp af de datastrukturer fra kapitel 3 i Java.

Selve koden til programmet findes på side 28. Her vil vi i stedet beskrive hvordan vi har testet programmet. Den første test var grafen fra kapitel 1. Vi har allerede undersøgt den og ved, at den maksimale parring indeholder fire kanter. Det er heldigvis også det vores algoritme kommer frem til, se figur 4.1 på den følgende side, hvor algoritmens fem trin er vist.

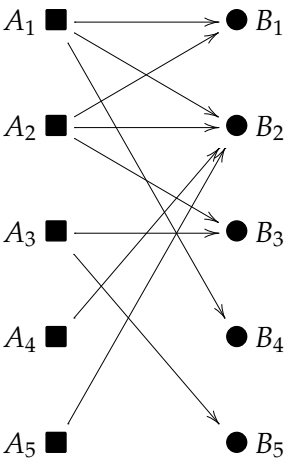
Efter første trin, som ses på figur 4.1(a), er grafen blevet farvet helt sort, da alle firkanterne har indgrad 0 og alle cirklerne har udgrad 0. Vi kan derfor lave en vilkårlig kant om til den første parringskant, og algoritmen vælger kanten (A_1, B_1).

På 4.1(b) er grafen farvet på ny og vi kan nu lave to forskellige skifteveje med udgangspunkt i A_2: (A_2, B_1, A_1, B_2) og (A_2, B_1, A_1, B_4). Af mere eller mindre tilfældige årsager, så vælger vores program den sidste skiftevej. Det illustrerer en interessant ting ved vores algoritme: det er svært på forhånd at sige, hvilke kanter den vil vælge. Det er ikke fordi den er nondeterministisk, men derimod fordi den er rimelig kompleks: kanterne bliver hele tiden lagt på og taget af to stakke, så det er svært at overskue hvad der sker.

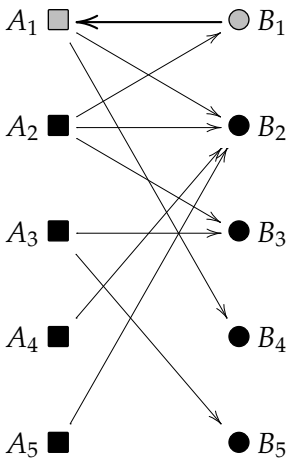
Efter at skiftevejen får vi figur 4.1(c). Her ser vi, at knuderne A_1, A_2 og B_1 nu ikke længere bliver farvet. Det skyldes, at de alle tre har både indgående og udgående kanter, således at de ikke kan være hverken startpunkt eller endepunkt for en skiftevej.

Men (A_3, B_3) kan lavet til en parringskant, hvilket bringer os til 4.1(d). Her bliver (A_4, B_2) til en parringskant, og der er nu ikke flere sorte cirkler, hvilket betyder, at der heller ikke er flere skifteveje, idet alle skifteveje skal ende i en sort cirkel.

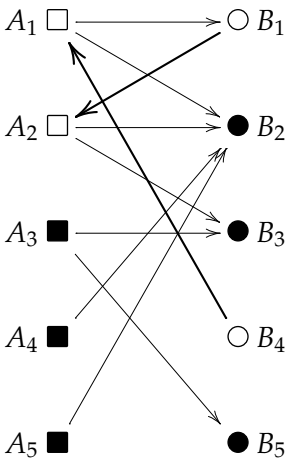
Programmet fandt altså de fire parringskanter ved at farve grafen fem gange — helt som forventet, da programmet bliver nødt til at farve en ekstra gang til sidst sådan at det kan konstatere, at der ikke er flere sorte cirkler.



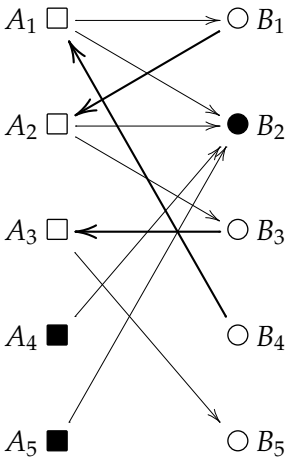
(a) Grafen som den ser ud efter den første farvning. Vi kan nu vende kanten (A_1, B_1) om.



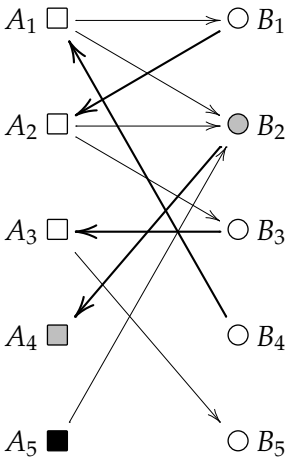
(b) Kanten (B_1, A_1) er nu lavet til en parringskant, og den indgår også i skiftevejen (A_2, B_1, A_1, B_4).



(c) Kanten (A_3, B_3) kan nu vendes om, således at den bliver en parringskant.



(d) Der er nu kun to interessante kanter tilbage, hvoraf kun den ene vil blive til en parringskant: (A_4, B_2) og (A_5, B_2).



(e) Nu er (A_4, B_2) vendt om, og der er ikke flere sorte cirkler. Vi har således fundet den sidste parringskant.

Figur 4.1: De fem trin i runTransitionSystem kørt på grafen fra kapitel 1. Som forventet, så finder vi fire parringskanter. Bemærk, at der stadig er en sort firkant tilbage i 4.1(e), selvom der ikke er flere sorte cirkler.

4.1 Skemaplanlægning på Vorrevangsskolen

Et problem som hvert år stilles for verdens forskellige uddannelsessteder er planlægningen af et hensigtsmæssig skema givet hvilke fag klasserme skal instrueres i og lærernes kompetencer.

Vi vil her undersøge muligheden at bruge vores program til at løse et skemaplanlægningsproblem fra Vorrevangsskolen. Givet en ekstern repræsentation af en todelt graf (U, W, E), hvor U angiver lærerne, W klasserne og en kant $E = (U_i, W_j)$ at lærer U_i kan undervise klasse W_j, er vi interesserede i at finde ud af, hvor mange klasser kan undervises samtidigt og af hvilke lærere? Grafen for problemet findes i figur 4.2 på næste side.

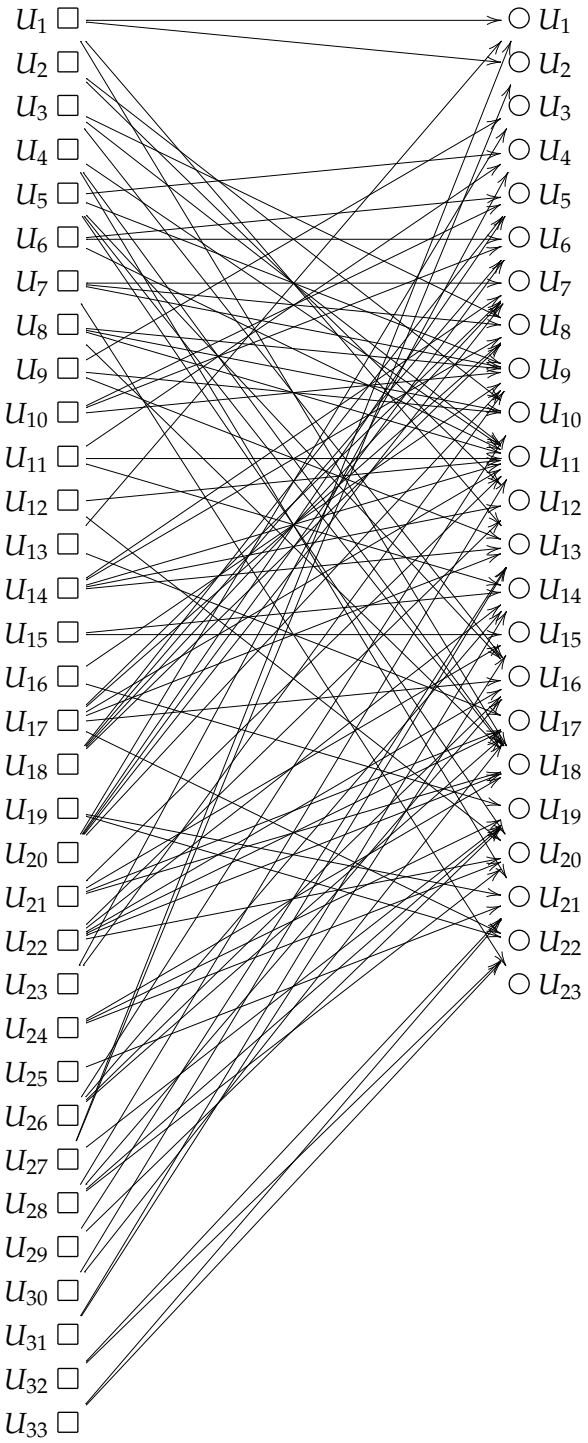
Skal opgaven løses ved håndkraft vil det kræver et vist overblik og et overskud af tålmodighed. Generelt set kan den slags effektiviseringsproblemer, hvor mængden af de mulige konfigurationer stiger meget hurtigt i forhold til antallet af elementer, kun løses med ved hjælp af en elegant algoritme kørende på en datamaskine.

Omformulere vi Vorrevangsskolens skemaplanlægningsproblem til et problem der vedrører en todelt graf, så er vores mål at finde den største delmængde af kanterne således at hver kant er knudedisjunkt. En lærer kan nemlig ikke undervise to klasser på samme tid, og et hold kan ikke undervises at to forskellige lærer samtidigt. Set fra denne synsvinkel er den ovennævnte mængde per definition den største mulig parring.

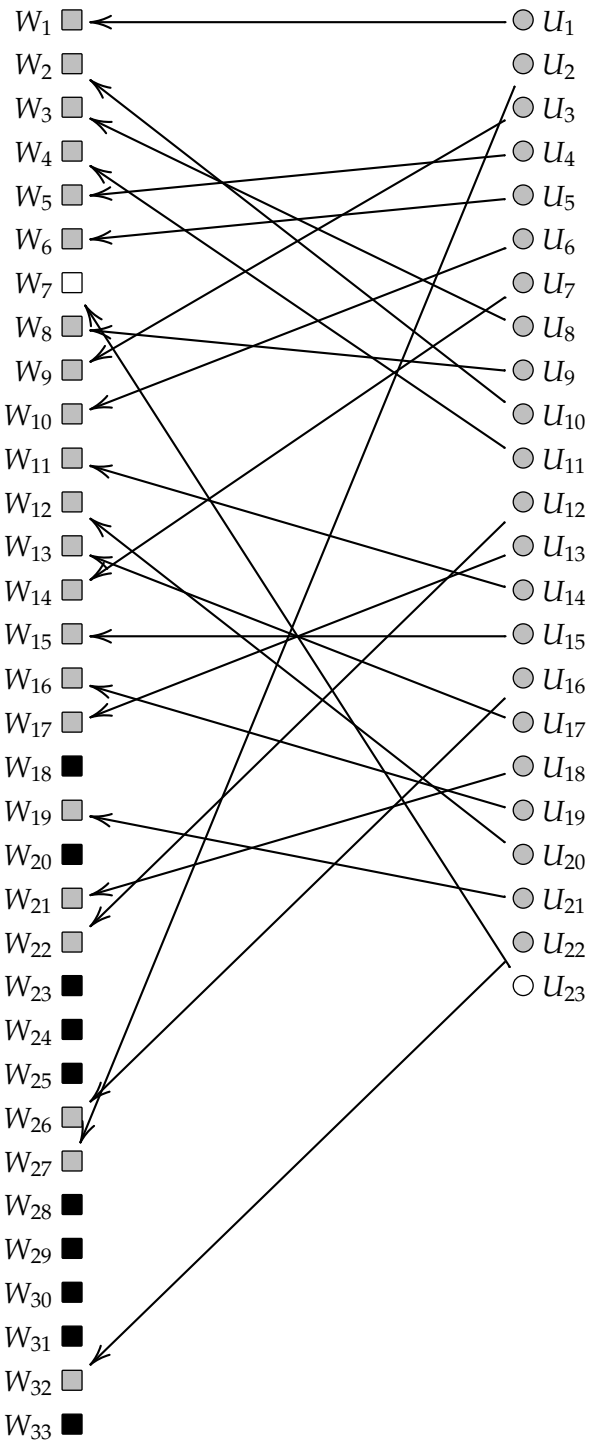
Vi udfører algoritmen MaximumMatching med grafen der var givet i filen `vorrevang.gph` som input. Resultatet findes i figur 4.3 på side 22, og det giver straks et svar til problemet.

Ud fra grafen kan vi slutte, at der eksisterer en parring som dækker mængden W af klasserne, hvilket betyder, at alle klasser kan blive undervist på samme tid. Da kanterne i den resulterende parring dækker alle knuder kan vi med sikkerhed konkludere, at det er en største parring. Tilføjelse af en ekstra kant i parringen vil nemlig medføre, at en samme klasse „dækkes“ af to lærere. Algoritmen fortæller os også umiddelbart, hvilke lærere der skal undervise hvilke klasser.

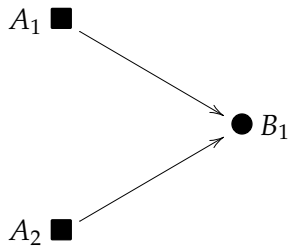
Til gengæld er der på ingen måde blevet sagt, at vi har fat i den *eneste* maksimale parring. Der kan eksistere flere mulige maksimale parringer, da kanterne der udgør en skiftevej efter hver iteration af algoritmen, ikke nødvendigvist vælges entydigt. Det indser vi let ved at betragte figur 4.4 på side 23. En algoritme der giver flere mulige maksimale parringer kunne være interessant, hvis f.eks. den givene planlægning ikke er tilfredsstillende for lærernes side.



Figur 4.2: Grafen for skemaplanlægningsproblemet — problemet er rimelig komplekst, da der er ialt 104 kanter i grafen



Figur 4.3: Resultatet efter at algoritmen er kørt på problemet. Man ser nemt, at alle klasserne har en lærer.



Figur 4.4: Den maksimale parring er ikke entydigt bestemt ved grafens udseende. Det ses lettest ved et modeksempel: der er to muligheder for den maksimale parring i denne graf: $\{A_1, B_1\}$ og $\{A_2, B_1\}$.

4.2 Repræsentanter til skolekomedien

Vi skal bruge vores program til at løse et praktisk problem: I Vorrevangsskolens 5.b går der 18 elever:

> Anders, Bodil, Camilla, Doris, Esben, Frida, Gert, Hassan, Ida, Jane, Klaus, Liv, Morten, Nicolai, Ofelia, Peter, Rasmus og Sine.

Klassen er ved at øve sig på en skolekomedie, hvor de skal være på scenen efter planen i tabel 4.1. 5.b's klasselærer vil gerne have valgt en elevrepræsentant for hver scene til at stå for rekvisitter. Repræsentanten for en scene skal selv være med i scenen, og ingen elev må repræsentere mere end én scene.

Scene	Medvirkende
1	Doris, Esben, Hassan, Jane, Peter og Sine
2	Liv og Sine
3	Hassan, Klaus og Peter
4	Frida, Hassan, Nicolai, Ofelia, Rasmus og Sine
5	Bodil, Hassan, Liv, Peter og Sine
6	Hassan, Peter, Rasmus og Sine
7	Bodil, Hassan, Liv og Peter
8	Anders, Ida, Klaus, Morten og Nicolai
9	Bodil, Peter og Sine
10	Camilla, Doris, Gert, Ofelia og Rasmus
11	Bodil, Klaus, Liv og Peter
12	Hassan, Klaus og Sine
13	Alle

Tabel 4.1: Plan over de medvirkende i skolekomedien på Vorrevangsskolen.

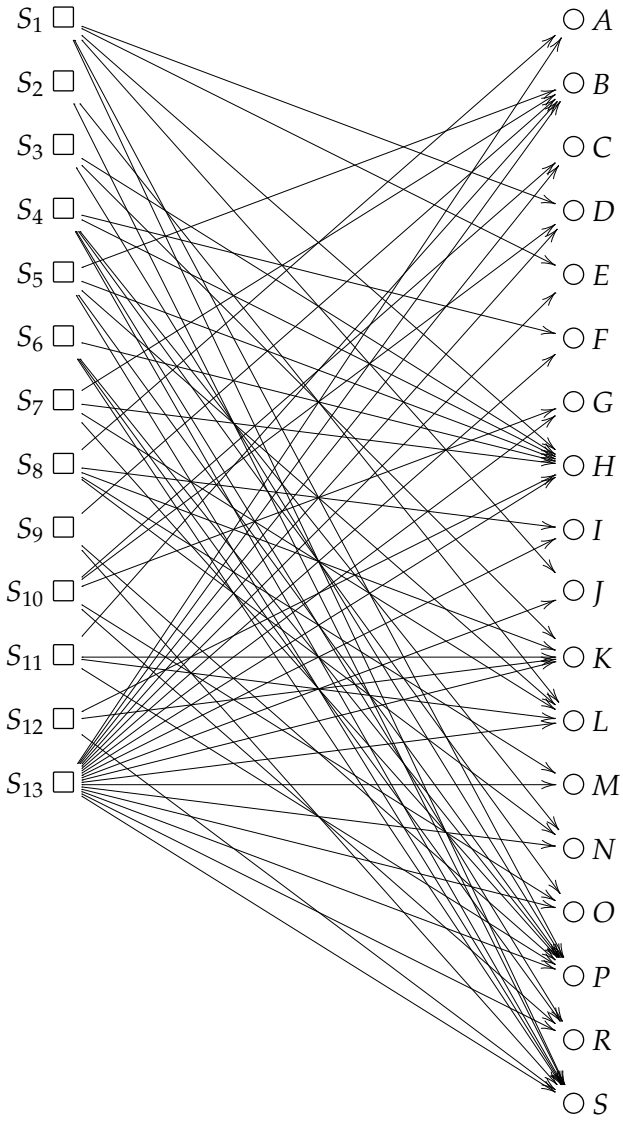
Problemer er nu at bestemme, om det overhovedet kan lade sig gøre at vælge 13 repræsentanter, og i givet fald hvordan de skal vælges.

Vi vil naturligvis også løse dette problem ved at omformulere det som et spørgsmål om maksimal parring i en todelt graf. På figur 4.5 på næste side ser man en graf der repræsenterer problemet.

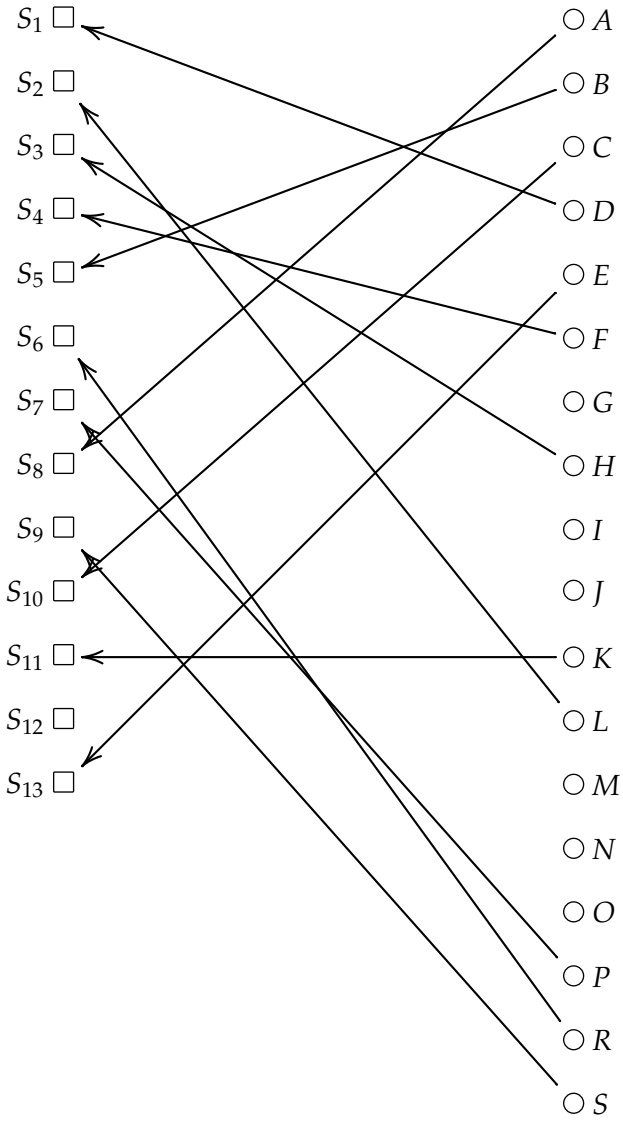
Hver scene er knyttet til de elever der skal være med i den pågældende scene. For at løse problemet, så skal vi finde den maksimale parring, og undersøge, om alle scenerne er med i denne. Vi ved nemlig at knuderne i den maksimale parring alle er forskellige, sådan at kravet om, at hver elev højst repræsenterer én scene er opfyldt.

Den maksimale parring er fundet på figur 4.6 på side 26. Vi har kun taget parringskanterne med for at gøre det nemmere, at se hvilke scener der har en elevrepræsentant. Det fremgår, at scene 12 ikke har nogen elevrepræsentant.

Da vi har fundet den maksimale parring, og vi kan konstatere, at den består af 12 kanter, så ved vi, at det er umuligt at vælge elevrepræsentanter for alle scenerne, da der er 13 scener. Hvis klasselæreren dropper scene 12, så går kabalen op. De parringskanter vi allerede har fundet ændrer sig nemlig ikke når vi fjerner en knude.



Figur 4.5: Graf over problemet med at vælge repræsentanter til skolekomedien. De udgående kanter fra hver scene går til de elever der er med i scenen. En graf som denne giver et fint indtryk af kompleksiteten af de problemer vores algoritme kan løse.



Figur 4.6: Grafen efter at den maksimale parring er fundet. Bemærk, at der går en parringskant til alle scenerne med undtagelse af scene 12. Det fortæller os, at det ikke er muligt at vælge 13 elevrepræsentanter ud fra de givne krav.

Kapitel 5

Konklusion

Vi skulle designe en algoritme og dens tilhørende datastrukturer og derefter implementere det i JAVA. Det lykkedes os at designe og bevise algoritmen, og det var heller ikke noget problem at implementere algoritmen i JAVA når pseudokoden først var skrevet.

Ved først at bevise algoritmens korrekthed ud fra principperne om gyldige udsagn spredt i pseudokoden, så viste det sig faktisk, at resten af arbejdet var nemt. Vi vidste, at tingene ville virke når vi konvertere pseudokoden til JAVA, for vi havde *bevist*, at det var korrekt. Vi behøvede således ikke at begynde med en besværlig afprøvning sådan som det ofte er tilfældet når man har lavet et program ved at prøve sig lidt frem.

Det, at vores program er korrekt, hænger selvfølgelig på, at vores bevis for dets korrekthed selv er korrekt, ligesom det hænger på, at vi har konverteret pseudokoden til ækvivalent kode i JAVA. Men sådan er det jo altid når man beviser noget: hvis selve beviset ikke er korrekt, så er resultatet ubrugeligt.

Derefter brugte vi programmet til at løse to praktiske planlægningsproblemer — begge problemer involverede så mange faktorer, at det ikke ville have været muligt at løse dem i hånden. Men ved hjælp af vores program var vi i stand til at løse dem hurtigt og effektivt.

Opgaven var spændende og tilpas udfordrende, idet vi skulle løse et ikke-trivielt problem. Først på det meget abstrakte plan, dernæst på et mere konkret plan, og tilsidst et praktisk plan hvor vi så på realistiske problemer fra hverdagen.

Bilag A

Kildekode

Det endelige program kom til at bestå af syv klasser hvoraf vi lavede de tre fra bunden, mens resten er mere eller mindre uændrede i forhold til da vi fik dem.

Vi har valgt, at skrive på engelsk i JAVA-koden, og derfor har klasserne også fået engelske navne. Der er derfor ikke en klasse der hedder ParringsGraf men derimod en klasse der hedder BipartiteGraph da er det engelske ord for en todelt graf. Men da det var et krav, at man skulle kunne køre det færdige program som

```
% java MaxParring data.gph
```

så er der dog en klasse der hedder MaxParring — klassen sørger blot for at kalde main i MaximumMatching.

A.1 MaxParring

Som allerede nævnt, så er denne klasse kun en *wrapper* omkring MaximumMatching.

```
2  import java.util.Enumeration;

4  /**
     Hovedprogrammet. Denne klasse eksistere kun for man kan kalde
6    programmet som <code>java MaxParring &lt;data&gt;</code>.
   */
8  public class MaxParring {

10   /**
       Starter programmet.
12
       Programmet tager op til to argumenter:
14
```

```
        <pre>
16      Usage: java MaximumMatching &lt;DATA&gt; [OPTIONS]

18      A graph is read from the file DATA and the maximum matching is
        reported along with the nodes in the matching. The program has the
20      following optional arguments:

22      −l, −−latex GRAPH Save the graph to the file graph−GRAPH−begin.tex
                       as an Xy−pic matrix. This is done just after
24                     it is constructed. The graph is also saved to
                       graph−GRAPH−end.tex when the maximum
26                     matching has been found.
        −h, −−help      Prints this message and exit with exitcode 0.
28      </pre>

30      @param args kommandolinieparametre.
    */
32  public static void main(String[] args) {
        MaximumMatching.main(args);
34  }
}
```

A.2 MaximumMatching

Det er i denne klasse at tingene foregår. Den laver en ny graf, og beder denne om at hente dataene i det filnavn der blev givet på kommandolinien. Hvis det går galt fordi der ikke var et sådant filnavn, så udskrives der en kort forklaring og programmet afsluttes med returværdi 1.

```
2   import java.util.Enumeration;

4   /**
      The main program.
6
      This program will find the number of nodes in the maximum pairring
8     in a graph. The graph is loaded from the file specified on the
      command−line.
10  */
    public class MaximumMatching {
12
      private static String outputFilename = null;
14
      /**
16      Starts the program. The program takes the following arguments:
        <pre>
18      Usage: java MaximumMatching &lt;DATA&gt; [OPTIONS]
```

```
20      A graph is read from the file DATA and the maximum matching is
        reported along with the nodes in the matching. The program has the
22      following optional arguments:

24      -l, --latex GRAPH Save the graph to the file graph-GRAPH-begin.tex
                         as an Xy-pic matrix. This is done just after
26                       it is constructed. The graph is also saved to
                         graph-GRAPH-end.tex when the maximum
28                       matching has been found.
        -h, --help       Prints this message and exit with exitcode 0.
30      </pre>

32      @param args the command-line arguments.
    */
34  public static void main(String[] args) {
        try {
36          BipartiteGraph G = new BipartiteGraph();
            /* We check for arguments: */
38          for (int i = 0; i < args.length; i++) {
                if (args[i].equals("-l") || args[i].equals("--latex")) {
40                  /* LaTeX output. */
                    i++;
42                  outputFilename = args[i];
                } else if (args[i].equals("-h") || args[i].equals("--help")) {
44                  /* Help message. */
                    usage();
46                  System.exit(0);
                } else {
48                  /* Everything else should be a filename. */
                    G.loadDataFromFile(args[i]);
50              }
            }
52          /* We will now calculate the maximum matching in G: */
            System.out.println(calculateMaximumMatching(G));
54      } catch (ArrayIndexOutOfBoundsException e) {
            usage();
56          System.exit(1);
        }
58  }

60  /** Prints a short usage message. */
    public static void usage() {
62      /* Spaces needed for alignment of the option: */
        String s = "                        ";
64      System.out.println("Usage: java MaximumMatching " +
                           "<DATA> [OPTIONS]\n");
66      System.out.println("A graph is read from the file DATA and the " +
                           "maximum matching is\nreported along with the " +
                           "nodes in the matching. The program has\n" +
68
```

```
                    "the␣following␣optional␣arguments:");
70
        System.out.println(
72          "\n-l,␣--latex␣GRAPH␣␣␣␣" +
            "Save␣the␣graph␣to␣the␣file␣graph-GRAPH-begin.tex\n" +
74          s + "as␣an␣Xy-pic␣matrix.␣This␣is␣done␣just␣after\n" +
            s + "it␣is␣constructed.␣The␣graph␣is␣also␣saved␣to\n" +
76          s + "graph-GRAPH-end.tex␣when␣the␣maximum\n" +
            s + "matching␣has␣been␣found."
78      );

80      System.out.println("\n-h,␣--help␣␣␣␣␣␣␣␣␣␣" +
                    "Prints␣this␣message␣and␣exit␣with␣exitcode␣0.");
82  }

84  /**
      Calculates the maximum matching.
86
      @param G the graph that should be examined.
88    @return a message that tells the user the number of nodes in
      the maximum matching along with the nodes themselves.
90  */
    public static String calculateMaximumMatching(BipartiteGraph G) {
92      int i = 0;

94      if (outputFilename != null)
            G.toLaTeX("graph-" + outputFilename + "-begin.tex");
96
        Node n = G.runTransitionSystem();
98      while (n != null) {
            G.turnEdges(n);
100         G.removeColors();
            n = G.runTransitionSystem();
102         i++;
        }
104
        if (outputFilename != null)
106         G.toLaTeX("graph-" + outputFilename + "-end.tex");

108     return "The␣maximum␣matching␣consists␣of␣" + i + "␣nodes." +
            "The␣edges␣are:\n" + G.getMatches();
110 }
}
```

A.3 BipartiteGraph

Denne klasse repræsenterer vores parringsgraf og indholder de metoder man skal bruge for at køre transitionssystemet og så videre. De to metoder

runTransitionSystem og turnEdges er dog allerede beskrevet i kapitel 3 på side 12 og JAVA-koden udtrykker jo blot algoritmerne beskrevet der.

Metoden loadDataFromFile indlæser en graf fra en fil. Grafen skal selvfølgelig være gemt i det rigtige format:

$$
\begin{array}{l}
\langle n_1 \rangle \\
\langle n_2 \rangle \\
\langle m \rangle \\
\langle A_{i_1} \rangle \ \langle B_{j_1} \rangle \\
\langle A_{i_2} \rangle \ \langle B_{j_2} \rangle \\
\qquad \vdots \\
\langle A_{i_m} \rangle \ \langle B_{j_m} \rangle
\end{array}
$$

hvor n_1 og n_2 er antallet af knuder i henholdsvis V_1 og V_2 og m er antallet af kanter.

Vi har imidlertid udviddet formatet sådan, at man også kan specificere navnene på de enkelte knuder. Hvis der er flere linier tilbage i filen efter at de m kanter er indlæst, så vil de første n_1 linier angive navnene på knuderne i V_1 og dernæst kan der komme n_2 linier med navnene på knuderne i V_2. Hvis der mangler linier, så vil de resterende knuder blot hedde A_x og B_x hvor x er et løbenummer. Denne udvidelse er fuldt ud kompatibel med det oprindelige format, og den har gjort det meget nemmere at generere de forskellige grafer i denne rapport.

Graferne i rapporten er lavet ved et kald til toLaTeX. Denne metode tager et filnavn, og vil så udskrive den LaTeX-kode der skal til for at repræsentere grafen. Koden er lavet til XY-pic og kan simpelthen bruges sammen med `\[\input{fil}\]`.

```
import java.util.Enumeration;
import java.io.*;

/**
   A æreprsentation of a bipartite graph.

   The graph is represented only by the nodes – the edges are not
   stored explicit in the graph. The nodes are put into two lists,
   <i>V1</i> and <i>V2</i>.

   @author Martin Geisler <gimpster@gimpster.com>
   @author ééJrmy Auneau <jeremy.auneau@skjoldhoej.dk>
*/
public class BipartiteGraph {

    private Node[] v1 = new Node[0];
    private Node[] v2 = new Node[0];
```

```
18      private int n1 = 0;
        private int n2 = 0;
20
        /**
22         Run the graph coloring algorithm.

24         The graph is colored by application of the coloring transition
           system. The algorithm runs in <i>O</i>(<i>|m| + |n|</i>), where
26         <i>m</i> and <i>n</i> are the edges and nodes from the graph
           which nodes are those from <i>V1</i> and <i>V2</i>.
28
           @return a random terminal node
30      */
        public Node runTransitionSystem() {
32         Sequence m1 = new Sequence(); // Edges from V1 to V2
           Sequence m2 = new Sequence(); // Edges from V2 to V1
34         Node v0 = null; // A random initial node.

36         for (int i = 0; i < n1; i++) {
              if (v1[i].inDegree() == 0) {
38               v1[i].color = "black";
                 Enumeration e = v1[i].outEdges();
40               while (e.hasMoreElements()) {
                    m1.insertLast((Edge)e.nextElement());
42               }
              }
44         }

46         while (m1.size() != 0 || m2.size() != 0) {
              while (m1.size() != 0) {
48               Edge e = (Edge)m1.removeLast();
                 if (e.to.outDegree() == 0) {
50                  e.to.color = "black";
                    /* We pick a random terminal node: */
52                  v0 = e.to;
                    /* We also store the edge from which we came, so
54                     that we can find our way back again later: */
                    e.to.prev = e;
56               } else {
                    e.to.color = "gray";
58                  /* We store the node which we came from, so that
                       we can find our way back again later: */
60                  e.to.prev = e;
                    Edge edge = e.to.anyOutEdge(); // There is exactly one.
62                  if (edge.to.color.equals("white")) {
                       m2.insertLast(edge);
64                  }
                 }
66            }
```

```
68          while (m2.size() != 0) {
                Edge e = (Edge)m2.removeLast();
70              e.to.color = "gray";
                Enumeration enum = e.to.outEdges();
72              while (enum.hasMoreElements()) {
                    Edge edge = (Edge)enum.nextElement();
74                  if (edge.to.color.equals("white")) {
                        m1.insertLast(edge);
76                  }
                }
78          }
        }
80      return v0;
    }
82
    /**
84      Turns the edges around.

86      The edges are reorientated along a random path leading back
        from the terminal node <code>v0</code> to an initial node. It
88      runs in <i>O</i>(<i>|m'|</i>) where <i>m'</i> are the edges that lie
        on the path.
90
        The terminal node <code>v0</code> <i>must</i> come from a call
92      to {@link #runTransitionSystem} because that method also stores
        information needed to find a path back to an initial node.
94
        @param v0 a terminal node as returned by runTransitionSystem()
96      @see #runTransitionSystem()
    */
98  public void turnEdges(Node v0) {
        Sequence m = new Sequence();
100     Edge e1 = null; // An Edge from V1 to V2
        Edge e2 = null; // An Edge from V2 to V1
102     Node v1 = null; // A Node in V1
        Node v2 = null; // A Node in V2
104
        e1 = v0.prev;
106     v1 = e1.from;
        m.insertLast(e1);
108
        while (!v1.color.equals("black")) {
110         e2 = v1.anyInEdge(); // There is exactly one edge.
            v2 = e2.from;
112         e1 = v2.prev;
            v1 = e1.from;
114         m.insertLast(e1);
            m.insertLast(e2);
```

```
116         }

118         Enumeration e = m.elements();
            while (e.hasMoreElements()) {
120             Edge edge = (Edge)e.nextElement();
                Node a = edge.from;
122             Node b = edge.to;
                edge.remove();
124             new Edge(b, a);
            }
126
        }
128
        /**
130        Colors the nodes white

132        All the nodes composing the graph are run through. The color of
           the nodes is systematically changed to white. The algorithm
134        runs in <i>O</i>(<i>|n|</i>), where <i>n</i> are the nodes
           listed in <i>V1</i> and <i>V2</i> (which means the total amount
136        of nodes in the graph).
        */
138     public void removeColors() {
            for (int i = 0; i < n1; i++)
140             v1[i].color = "white";
            for (int i = 0; i < n2; i++)
142             v2[i].color = "white";
        }
144
        /**
146        Loads an input graph from a file

148        Input data are read from a file and the corresponding graph is
           constructed. The algorithm runs in <i>O</i>(<i>|n| + |m|</i>) where
150        <i>m</i> are the edges listed in the input file and <i>|n|</i>
           the number of nodes composing the graph.
152
           @param filename an input data file describing a graph.
154     */
        public void loadDataFromFile(String filename) {
156
            try {
158             BufferedReader input =
                    new BufferedReader(new FileReader(filename));
160
                n1 = Integer.parseInt(input.readLine());
162             v1 = new Node[n1];
                for (int i = 0; i < n1; i++)
164                 v1[i] = new Node("A" + (i+1), i);
```

```
166             n2 = Integer.parseInt(input.readLine());
                v2 = new Node[n2];
168             for (int i = 0; i < n2; i++)
                    v2[i] = new Node("B" + (i+1), i);
170
                int  m = Integer.parseInt(input.readLine());
172             for (int i = 0; i < m; i++) {
                    String line = input.readLine();
174                 int s = line.indexOf(" ", 0);
                    String a = line.substring(0, s);
176                 String b = line.substring(s+1, line.length());
                    new Edge(v1[Integer.parseInt(a)],
178                         v2[Integer.parseInt(b)]);
                }
180
                /*
182               Extension: it is possible to specify the names of the
                  nodes in the lines that follow the last edge. The
184               format is: one name per line, starting with the names
                  of the nodes in V1.
186             */
                for (int i = 0; i < n1 && input.ready(); i++) {
188                 v1[i].name = input.readLine();
                }
190
                for (int i = 0; i < n2 && input.ready(); i++) {
192                 v2[i].name = input.readLine();
                }
194         } catch (IOException e) {
                System.err.println("There was an IO error while " +
196                                "reading the file - aborting...");
                System.exit(1);
198         } catch (NumberFormatException e) {
                System.err.println("There was a number format error while " +
200                                "reading the file - aborting...");
            }
202     }

204     /**
          Produces a graphical LaTeX output of a graph. The output is an
206       Xy-pic matrix which can be included in another LaTeX document.

208       @param filename the LaTeX output will be saved here.
        */
210     public void toLaTeX(String filename) {
            toLaTeX(filename, false);
212     }
```

```
214     /**
          Produces a graphical LaTeX output of a graph. The output is an
216       Xy-pic matrix which can be included in another LaTeX document.

218       @param filename the LaTeX output will be saved here.
          @param only_pairs boolean that indicates whether the produced
220       graph should contain all the edges or only the pair edges.
        */
222     public void toLaTeX(String filename, boolean only_pairs) {
          Enumeration enum = null;
224       String str = "\\xymatrix@!0{\n";
          int i = 0;
226       int extra_columns = 8;
          while (i < n1 || i < n2) {
228         if (i < n1) {
              str += "␣␣{\\rbox{" + v1[i].name + "\\;}" +
230             "\\"␣+␣v1[i].color.charAt(0)␣+␣"box}";
              if (!only_pairs) {
232             enum = v1[i].outEdges();
                while (enum.hasMoreElements()) {
234               Edge e = (Edge)enum.nextElement();
                  Node n = e.to;
236               str += "␣\\ar[" + (n.rankLaTeX - i) +
                        "," + extra_columns + "]␣";
238             }
              }
240         }

242         str += "␣";
            for (int c = 0; c < extra_columns; c++) {
244           str += "&";
            }
246         str += "␣";

248         if (i < n2) {
              enum = v2[i].outEdges();
250           while (enum.hasMoreElements()) {
                Edge e = (Edge)enum.nextElement();
252             Node n = e.to;
                str += "\\ar@*{[thicker]}[" + (n.rankLaTeX - i) +
254                 ",-" + extra_columns + "]␣";
              }
256           str += "{\\"␣+␣v2[i].color.charAt(0)␣+␣"circ"␣+
                "\\lbox{\\;" + v2[i].name + "}}";
258         }

260         i++;
            if ((i < n1 && n1 > n2) || (i < n2 && n2 > n1)) {
262           str += "␣\\\\\n";
```

```
            }
        }

        str = str + "\n}\n";

        BufferedWriter out;
        try {
            out = new BufferedWriter(new FileWriter(filename));
            out.write(str);
            out.close();
        } catch (IOException e) {
            System.err.println("There␣was␣an␣IO␣problem␣with␣the␣file␣" +
                               filename + "␣-␣no␣LaTeX␣output␣produced.");
        }
    }

    /**
       Returns a string representation of the graph.

       This should only be used when debugging.

       @return an enumeration of the nodes in <i>V1</i> and <i>V2</i>.
    */
    public String toString() {

        Enumeration e = null;
        String result = "Nodes␣in␣V1:\n";
        for (int i = 0; i < v1.length; i++) {
            result = result + v1[i] + ":\n␣␣In:␣␣";

            e = v1[i].inEdges();
            while (e.hasMoreElements())
                result = result + (Edge)e.nextElement() + "␣";

            result = result + "\n␣␣Out:␣";

            e = v1[i].outEdges();
            while (e.hasMoreElements())
                result = result + (Edge)e.nextElement() + "␣";

            result = result + "\n";
        }

        result = result + "Nodes␣in␣V2:\n";
        for (int i = 0; i < v2.length; i++) {
            result = result + v2[i] + ":\n␣␣In:␣␣";

            e = v2[i].inEdges();
```

```
312             while (e.hasMoreElements())
                    result = result + (Edge)e.nextElement() + "␣";
314
                result = result + "\n␣␣Out:␣";
316
                e = v2[i].outEdges();
318             while (e.hasMoreElements())
                    result = result + (Edge)e.nextElement() + "␣";
320
                result = result + "\n";
322         }

324         return result;
        }
326
        /**
328        Get a string representation of the edges that are part of the
           matching.
330
           This is done by finding the edges that goes from V2 to V1. It
332        is done in <i>O</i>(<i>|V1|</i>) since we only have to ask each
           node in V1 once to find out if there is an incoming edge. If
334        so, then it takes a constant amount of time to acquire the
           edge.
336
           @return a string representation of the edges in the matching,
338        or the empty string if the matching is the empty set.
        */
340     public String getMatches() {
            String str = "";
342
            for (int i = 0; i < n1; i++) {
344             if (v1[i].inDegree() == 1) {
                    str += ",␣" + v1[i].anyInEdge().toString();
346             }
            }
348         return str.substring(2);
        }
350
} // BipartiteGraph
```

A.4 Node

Klassen Node er en af de færdige klasser, men vi har dog tilføjet lidt til den. Det er helt centralt for algoritmen, at det er muligt at farve knuderne. Vi har derfor tilføjet et feltet color af typen String til at gemme knudens farve. Alle knuder starter med at være hvide. På samme måde har vi tilføjet feltet

prev som indeholder den kant vi fulgte da vi besøgte knuden. Feltet rank indeholder knudens placering i enten V_1 eller V_2 og bruges kun af toLaTeX til at generere graferne.

Man kunne sagtens have implementeret disse dekoreringer ved at lave Node udvide en klasse der implementerer en **public void** set(String, Object) til at sætte disse dekoreringer, og en **public** Object get(String) metode til at få fat på værdien af en tidligere dekorering. Det ville heller ikke ændre på tidskompleksiteten, da antallet af dekoreringer ikke afhænger af antallet af kanter eller antallet af knuder: der er højst de tre nævnte dekoreringer.

Men i vores tilfælde vil det bare gøre tingene unødigt komplicerede at skulle igennem et ekstra lag for at få fat i informationerne.

Der er også to nye metoder: anyInEdge og anyOutEdge og det returnerer simpelthen en tilfældig indgående eller udgående kant. Her betyder „'tilfældig"' faktisk den første kant, men det er ikke vigtigt i denne sammenhæng.

```
2  import java.util.Enumeration;

4  /**
     A representation of a node. Created: Fri Mar 31 09:16:17 2000
6
     @author Hans Kyndesgaard
8    @version 1
   */
10 public class Node {
     Sequence out = new Sequence();
12   Sequence in = new Sequence();
     /* The name of the node. */
14   String name = "";
     /* The rank used by the toLaTeX method in BipartiteGraph. */
16   int rankLaTeX = 0;
     String color = "white";
18   Edge prev = null;

20   /**
       Creates a new Node.
22     @param name You can give each node a name. This name can be
       very handy when debugging. <i>O</i>(<i>1</i>).
24   */
     public Node(String name, int rankLaTeX) {
26       this.name = name;
         this.rankLaTeX = rankLaTeX;
28   }

30   /**
       Get the outgoing edges.
32     @return an enumeration of all the edges going out of
```

```
     <code>this</code> node. <i>O</i>(<i>1</i>).
34   */
     public Enumeration outEdges() {
36       return out.elements();
     }
38
     /**
40     Returns a random outgoing edge.
       @return an outgoing edge. <i>O</i>(<i>1</i>).
42   */
     public Edge anyOutEdge() {
44       return (Edge)out.elements().nextElement();
     }
46
     /**
48     Get the incoming edges.
       @return an enumeration of all the edges going in to
50     <code>this</code> node. <i>O</i>(<i>1</i>).
     */
52   public Enumeration inEdges() {
       return in.elements();
54   }

56   /**
       Returns a random incoming edge.
58     @return an incoming edge. <i>O</i>(<i>1</i>).
     */
60   public Edge anyInEdge() {
       return (Edge)in.elements().nextElement();
62   }

64   /** Get the number of incoming edges.
       @return the number of edges going into <code>this</code> node.
66     <i>O</i>(<i>1</i>).
     */
68   public int inDegree(){
       return in.size();
70   }

72   /**
       Get the number of outgoing edges.
74     @return the number of edges going out of <code>this</code>
       node. <i>O</i>(<i>1</i>)
76   */
     public int outDegree(){
78       return out.size();
     }
80
     /**
```

```
82          The string representation is just the given name.
            <i>O</i>(<i>1</i>)
84      */
        public String toString(){
86          return name + "␣[" + color.charAt(0) + "]";
        }
88
} // Node
```

A.5 Sequence

Den eneste ændring vi har lavet i klassen Sequence er, at vi har tilføjet en removeLast metode. Den returnerer det sidste element hvis der er et, ellers får man en fejl. Men denne metode kan man direkte bruge en Sequence som en stak hvis man skulle få brug for det — vi bruger removeLast til hurtigt at få fat i og samtidig fjerne et vilkårligt element fra sekvensen.

```
2   import java.util.NoSuchElementException;

4   /**
        A sequence. This is implemented using a double−linked list with
6       locators which means that most operations run in time <i>O</i>(<i>1</i>).

8       Created: Fri Mar 31 09:33:27 2000

10      @author Hans Kyndesgaard
        @version 1
12  */
    public class Sequence {
14      private class SNode implements Locator{
            SNode prev;
16          SNode next;
            Object element;
18          boolean isContained;

20          public SNode(SNode prev, SNode next, Object element){
                this.prev = prev;
22              this.next = next;
                this.element = element;
24              this.isContained = true;
            }
26
            public Object element(){
28              return this.element;
            }
30
            public Object container(){
```

```
32              return Sequence.this;
            }
34
            public boolean isContained(){
36              return this.isContained;
            }
38      } //SNode

40      SNode first;
        SNode last;
42      Object stop = new Object();
        int size;
44
        /** Constructs an empty sequence. <i>O</i>(<i>1</i>) */
46      public Sequence() {
            first = new SNode(null, null, stop);
48          last = new SNode(first, null, stop);
            first.next = last;
50      }

52      public int size(){
            return size;
54      }

56      /**
           Inserts an object at the last posistion of the sequence.
58         <i>O</i>(<i>1</i>).
           @param o the object to be insterted.
60      */
        public Locator insertLast(Object o){
62          SNode n = new SNode(last.prev, last, o);
            n.prev.next = n;
64          n.next.prev = n;
            size++;
66          return n;
        }
68
        /**
70         Removes the last element in the sequence.
           @throws NoSuchElementException if the sequence is empty.
72         @return the last element in the sequence.
        */
74      public Object removeLast() {
            if (size > 0) {
76              SNode n = last.prev;
                Object obj = n.element;
78              n.prev.next = last;
                last.prev = n.prev;
80              n.isContained = false;
```

```
            size--;
82          return obj;
        } else {
84          throw new NoSuchElementException("The Sequence is empty.");
        }
86  }

88  /**
      Removes an element from the sequence given that elemnts
90    locator. <i>O</i>(<i>1</i>).
      @param l The locator of the element to be removed.
92  */
    public void remove(Locator l){
94      if (l instanceof SNode && l.isContained() && l.container() == this) {
            SNode sn = (SNode) l;
96          sn.prev.next = sn.next;
            sn.next.prev = sn.prev;
98          sn.isContained = false;
            size--;
100     }
    }
102
    /**
104   Make a string representation of the sequence. It is just the
      string representation of all the elements.
106   @return the constructed string.
    */
108 public String toString(){
        StringBuffer result = new StringBuffer("{");
110     java.util.Enumeration e = elements();
        while (e.hasMoreElements()) {
112         result.append(e.nextElement().toString());
            if (e.hasMoreElements()) {
114             result.append(", ");
            }
116     }
        result.append("}");
118     return result.toString();
    }
120
    /**
122   Makes an Enumeration of all the elements of the sequence. This
      method uses <i>O</i>(<i>1</i>) time and so does all of the
124   methods of the constructed Enumeration.
      @return the constructed enumeration
126 */
    public java.util.Enumeration elements(){
128     return new SEnumeration();
    }
```

```
130
    private class SEnumeration implements java.util.Enumeration{
132     SNode current;
        public SEnumeration(){
134         current = first.next;
        }
136
        public boolean hasMoreElements(){
138         return current != last;
        }
140
        public Object nextElement(){
142         if (hasMoreElements()) {
                Object result = current.element;
144             current = current.next;
                return result;
146         } else {
                throw new NoSuchElementException("No more elements " +
148                                                "in Enumeration.");
            }
150     }
    } //SEnumeration
152
} // Sequence
```

A.6 Edge

Klassen Edge er uændret.

```
/**
2   A representation of an edge.

4   Created: Fri Mar 31 09:31:36 2000

6   @author Hans Kyndesgaard
    @version 1
8 */
public class Edge {
10    Node from;
      Node to;
12    Locator fromLocator;
      Locator toLocator;
14
      /**
16       Creates an edge from the <code>from</code>-node to the
         <code>to</code>-node. Even though the two nodes are adjacent
18       an edge will be made between them. <i>O</i>(<i>1</i>).
         @param from The origin of the newly created edge.
```

```
20        @param to The destination of the newly created edge.
      */
22    public Edge(Node from, Node to) {
          this.from = from;
24        this.to = to;
          fromLocator = from.out.insertLast(this);
26        toLocator = to.in.insertLast(this);
      }
28
      /**
30        Removes <code>this</code> edge. The data structures in the
          nodes are updated accordingly. <i>O</i>(<i>1</i>).
32    */
      public void remove(){
34        from.out.remove(fromLocator);
          to.in.remove(toLocator);
36    }

38    /**
          Produce a stringrepresentation of a edge. This is just a pair
40        of node representations. <i>O</i>(<i>1</i>).
      */
42    public String toString(){
          return "(" + from + ",␣" + to + ")";
44    }

46  } // Edge
```

A.7 Locator

Interfacet Locator er uændret.

```
  /**
2   Locator.java

4   Created: Fri Mar 31 09:43:51 2000

6   @author Hans Kyndesgaard
    @version 1
8 */
  public interface Locator {
10
      /**
12        Returns the element. <i>O</i>(<i>1</i>).

14        @return the element associated with this locator.
      */
16    public Object element();
```

```
18      /**
           Returns true if the element is still contained.
20         <i>O</i>(<i>1</i>).

22         @return true if the element is still in a container.
        */
24      public boolean isContained();

26      /**
           Returns the container of the element. <i>O</i>(<i>1</i>).
28
           @return the container of the element associated with this
30         locator.
        */
32      public Object container();

34  } // Locator
```