View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NORA - Norwegian Open Research Archives

UNIVERSITETET I OSLO
Institutt for informatikk

Metoder for likhetsvurdering av innleverte obligatoriske oppgaver i Java

Hovedoppgave

Christian Kringstad Kielland

15. juli 2006

Forord

Da denne oppgaven ble påbegynt i 2004 virket konstruksjonen av et sammenligningssystem for obligatoriske oppgaver som et relativt kjapt og greit prosjekt. I løpet av de to siste årene har det vokst seg større og blitt utvidet med ny funksjonalitet, og vi har funnet stadig nye idéer som måtte testes. I løpet av 2005 ble prosjektet så omfattende at brukergrensesnittet og tilhørende funksjonalitet ble skilt ut som en egen hovedoppgave.

Arbeidet med denne oppgaven har vært spennende og til tider slitsomt. Selv om kjøring av tester og tabulering av resultater har en tendens til å bli ensformig, har det vært interessant å få muligheten til tenke ut nye måter å gjøre ting på. De varierte temaene jeg har fått mulighet til å sette meg inn i har også vært en kime til inspirasjon. Ikke bare handler denne oppgaven om avsløring av kopiering, men vi får i tillegg ny kunnskap om strukturene bak kopieringen. Programmeringen av systemet har også vært en nyttig erfaring.

Arbeidet med denne oppgaven ville vært svært mye vanskeligere om det ikke hadde vært for veilederen min, Arne Maus, og mine foreldre. De har alle vært svært tålmodige selv om arbeidet har tatt sin tid, og jeg vil benytte sjansen til å rette en takk til Arne for verdifull diskusjon og tilbakemelding og til mine kjære foreldre for deres støtte.

Jeg håper denne oppgaven vil være til nytte for Ifi og for andre som måtte ha interesse av temaene som behandles her.

– Christian Kringstad Kielland

Innhold

Forord **iii**

1 Innledning **1**

1.1 Oversikt over oppgaven 1

1.2 Begreper og ordforklaringer 3

2 Beskrivelse av oppgaven **5**

2.1 Bakgrunn for javasammenligneren 5

2.1.1 Det grunnleggende problemet 6

2.1.2 Hvordan redusere kopieringsproblemet? 6

2.1.3 Problemstillingen 7

2.2 Definisjon av problemområdet 8

2.2.1 Definisjon av plagiat 9

2.2.2 Definisjon av likhet 9

2.3 Ønskede egenskaper ved systemet 10

2.3.1 Nøyaktighet 10

2.3.2 Pålitelighet 11

2.3.3 Effektivitetshensyn 11

2.3.4 Robusthet 11

2.3.5 Brukbarhet 12

2.3.6 Valg av algoritme 12

2.3.7 Kjennskap 12

2.3.8 Mulige sikkerhetsproblemer 12
2.3.9 Andre problemsituasjoner 14
2.3.10 Konklusjon 14
2.4 Oppsummering 14

3 Metoder for likhetsvurdering 15
3.1 Lengste felles delstreng 15
3.2 Likhetspåvisning med YAP3 16
3.3 Likhetspåvisning med CBR-teknikker 17
3.4 Likhetspåvisning med symboltelling 18
3.5 Oppsummering 18

4 Systemet fra et brukerperspektiv 21
4.1 Et utkast til et praktisk sammenligningssystem 21
4.1.1 Mulige utvidelser av systemet 23
4.2 Utkast til system 23
4.2.1 Funksjonalitet i GUI 24
4.2.2 Funksjonalitet i asynkront system 24
4.3 Fordeler og ulemper 24
4.4 Bruksmønstre 27
4.4.1 Studentenes tenkte bruksmønster 28
4.4.2 Gruppelærerens tenkte bruksmønster 29
4.4.3 Kursledelsens tenkte bruksmønster 30
4.5 Definisjon av resultatdata 31
4.5.1 Resultatdata til gruppelærerene 31
4.5.2 Statistikk fra systemet 32
4.6 Avsluttende betraktninger 32

5 Systemspesifikasjon 35
5.1 Evalueringsfunksjonen 35

5.1.1 Potensialet for å narre systemet 36
5.1.2 Sammenligningskriterier 36
5.1.3 Ukorrekt klassifisering 37
5.2 Oversikt over algoritmen 37
5.2.1 Symboltelling 37
5.2.2 Måling av euklidsk avstand 38
5.2.3 Vinkelanalyse av dimensjonstallene 40
5.2.4 En mulig implementasjonsbeskrivelse 42
5.3 Sammenligning av fri tekst 43
5.4 Oppsummering 43

6 Implementasjon av en prototype 45

6.1 Databasen 45
6.1.1 Tabellen for programdata 45
6.1.2 Tabellen for vektorer 46
6.1.3 Tabellen for likhetsverdier 47
6.2 Sammenligningsprogrammet 47
6.2.1 Brukergrensesnittet 47
6.2.2 Sammenligningsmekanismen 50
6.3 Diskusjon av implementasjonsvalg 50
6.3.1 Formatering av innleverte obliger 50
6.3.2 Tilgang til filer fra systemet 50
6.4 Oppsummering 51

7 Symbolkombinasjoner 53

7.1 Innledende kommentarer 53
7.1.1 Prosedyre for valg av symbolsett 53
7.2 Redegjørelse for forsøkene 54
7.2.1 Forsøksmaterialet 55
7.2.2 Hvordan å velge symboler 55

7.2.3 Programstruktur 56
7.3 De innledende forsøkene 58
7.3.1 Fremgangsmåte for forsøkene 59
7.3.2 Materialet for forberedende forsøk 59
7.3.3 Første forberedende test 59
7.3.4 Andre forberedende test 60
7.3.5 Det tredje forsøket 63
7.3.6 Normalisering av likhetsverdiene 63
7.3.7 Avskjæring av sammenligningsprosessen 65
7.4 Vinkelanalyse 65
7.4.1 Vinkel vs. euklid 67
7.4.2 Normalisering av vektorvinkelen 68
7.4.3 Valg av sammenligningsmetode 68
7.5 Optimalisering av symbollistene 68
7.5.1 Tilnærming med *steepest hill* 69
7.5.2 Fastsetting av terskelverdier 72
7.6 Oppsummering 72

8 Om terskelverdier **73**
8.1 Forventninger om kopiering 73
8.2 Diskusjon om datagrunnlaget 74
8.2.1 Mistankekategorier 74
8.3 Beskrivelse av undersøkelsene 77
8.3.1 Svakheter i metoden 77
8.3.2 Personvernsproblematikk 78
8.4 Setting av terskelverdiene 78
8.5 Resultater av dataanalysen 79
8.5.1 Høye terskelverdier 79
8.5.2 Moderate terskelverdier 82

8.5.3 Lave terskelverdier 83
8.6 Konklusjoner 85
8.6.1 Hvem kopierer? 85

9 Validering av resultater 87
9.1 Kategorisering av resultater 87
9.1.1 Maskinens kategorier 88
9.1.2 Kategorier for skjønnsvurdering 88
9.1.3 Manuell vurdering 90
9.1.4 Størrelsesforskjellens betydning 93
9.1.5 Innsnevring av terskelverdiene 94
9.1.6 Mindre kompliserte obliger 96
9.2 Nettverkskopiering 96
9.3 Kopieringstilfeller og karakterer 97
9.4 Konklusjoner 97
9.4.1 Omfanget av kopieringsproblemet 98

10 Systemet i drift 99
10.1 Innsnevring av problemområdet 99
10.2 Nødvendige endringer 99
10.2.1 Om valg av programmeringsspråk 100
10.2.2 Brukergrensesnittet 100
10.3 Konvergering av to systemer 101
10.3.1 Kort beskrivelse av Joly-systemet 101
10.3.2 Joly-systemets oppgaver 101
10.3.3 Algoritmesubsystemets grensesnitt 102
10.3.4 Endringer i systemet 105
10.4 Avsluttende bemerkninger 106
10.5 Oppsummering 106

11 Oppsummering **107**
11.1 Sammenfatning av arbeidet 107
11.2 Konklusjoner 108
11.2.1 Har vi nådd våre målsetninger? 109
11.2.2 Kartlegging av kopieringsproblemet 111
11.3 Videre arbeid 111

Kapittel 1

Innledning

Denne hovedoppgaven tar for seg problemet med maskinell vurdering av likheten mellom forskjellige javakodefiler. Denne algoritmen skal implementeres som en sentral komponent i et sammenligningssystem for bruk i forbindelse med vurderingen av obligatoriske oppgaver i kurs ved Institutt for Informatikk ved Universitetet i Oslo.

Datamaterialet oppgaven baserer seg på, består av de aller fleste innleverte besvarelser av den fjerde obligatoriske oppgaven i begynnerkurset i informatikk (INF1000) høsten 2004. Besvarelsene av de andre og tredje obligatoriske oppgavene fra det samme kurset er i mindre grad benyttet som datagrunnlag. Dette utgjør over 1200 besvarelser.

1.1 Oversikt over oppgaven

Oppgaven er inndelt som følger:

Kapittel 1: Innledning Dette kapittelet. I tillegg til denne korte oversikten over innholdet i de forskjellige kapitlene, presenterer vi en liste over begreper som brukes i oppgaven og hvilken betydning de har i denne sammenhengen.

Kapittel 2: Oppgavebeskrivelse Her presenterer vi bakgrunnen og begrunnelsen for dette arbeidet. Vi går gjennom utfordringene oppgavene presenterer og hvordan vi planlegger å møte dem. Vi gir en oversikt over metodene og målene for arbeidet.

Kapittel 3: Sammenligningsmetoder I dette kapittelet undersøker vi forskjellige algoritmer for sammenligning av javakode. Algoritmene vil bli analysert og vurdert med henblikk på kravene som stilles til systemet. Vi velger en av disse algoritmene som grunnlag for det videre arbeidet.

Kapittel 4: Brukerperspektiv på systemet Her diskuterer vi problemstillinger i forhold til brukergrensesnittet, hvem som skal bruke systemet og hva slags informasjon systemet vil gi. Da denne problemstillingen er såpass omfattende, begrenser vi oss til en kartlegging av de elementene som er relevante for den videre utviklingen av algoritmeimplementasjonen. Problemstillingen blir nærmere behandlet av Steensen og Vibekk i [SV06].

Kapittel 5: Systemspesifikasjon Vi går nærmere inn på hva slags funksjonalitet systemet må tilby, hvilke potensielle problemer som kan oppstå i bruk og hvordan disse kan løses. Vi gir en nærmere beskrivelse av den valgte sammenligningsmetoden og diskuterer spørsmål rundt implementasjonen.

Kapittel 6: Implementasjon Her diskuterer vi avgjørelsene vi har tatt i forhold til implementasjonen av systemet. Vi diskuterer hvilket programmeringsspråk som bør benyttes, alternative metoder for å implementere algoritmen, organisering av databasen, grensesnittet mot Joly samt effektivitetsspørsmål.

Kapittel 7: Symbolkombinasjoner Med bakgrunn i et mindre utvalg av besvarelser forsøker vi å komme frem til brukbare sammenligningskriterier. Vi beskriver en testmetode i fire trinn for å få best mulige resultater ut fra ressursene vi har til rådighet. Det settet av symboler vi sitter med etter disse undersøkelsene vil brukes i de påfølgende kapitlene.

Kapittel 8: Omfanget av fuskingen og diskusjon om terskelverdier Vi foreslår en testmodell for å finne ut hvordan kopieringstilfellene fordeler seg blant studentene med utgangspunkt i de resultatene vi har oppnådd hittil. Vi vurderer påliteligheten i disse resultatene på bakgrunn av tidligere anslag om problemet. Vi foreslår også terskelverdier for sammenligningsresultatene.

Kapittel 9: Validering av sammenligningsresultatene Her tester vi resultatene fra de to foregående kapitlene på bakgrunn av det fullstendige

datagrunnlaget. Vi sammenligner systemets vurderinger av par av obligatoriske besvarelser (obliger) med vår egen vurdering for et representativt utvalg av obligparene for å finne ut hvorvidt det er samsvar mellom de maskinelle og manuelle resultatene. Vi foretar en siste kalibrering av systemets terskelverdier.

Kapittel 10: Systemet i drift Etter at en annen hovedoppgave resulterte i et brukergrensesnitt for sammenligningssystemet ble det nødvendig med justeringer i algoritmesubsystemet. I dette kapittelet beskriver vi overgangen fra vårt eget testgrensesnitt til Joly og integreringen av de to komponentene.

Kapittel 11: Oppsummering Vi oppsummerer funnene som er gjort i denne oppgaven. Vi foreslår også måter å videreføre arbeidet. Det bør vurderes alternative måter å møte denne problemstillingen på og andre bruksområder for systemet slik det er implementert. I tillegg foreslår vi alternative bruksområder av algoritmen.

1.2 Begreper og ordforklaringer

Algoritmesubsystemet Sammenligningssystemet består av to komponenter, algoritmesubsystemet er ansvarlig for selve sammenligningen og for å skrive innleveringsinformasjon til databasen. Den andre komponenten utgjør brukergrensesnittet og relaterte funksjoner og er implementert i Joly-systemet.

Dummy-metoder Metoder som er lagt inn i programkoden kun for å narre sammenligningssystemet. Slike metoder utfører oftest trivielle eller ingen oppgaver, og blir for det meste ikke engang kalt.

Euklidsk avstand Om to obliger karakteriseres ved hver sin N-dimensjonale symbolvektor med felles startpunkt, betegner dette begrepet avstanden mellom vektorenes endepunkter. Et potensielt mål på likhet.

Likhetsverdi Sammenligningssystemets vurdering av likheten ("avstanden") mellom to obliger. Mellom obliger av tilnærmet lik størrelse betyr lavere verdier større likhet. Begrepet *likhetsverdi* er ikke knyttet til en bestemt utregningsalgoritme, men brukes synonymt med både *euklidsk avstand* og *vektorvinkel*.

Mistankerapporter Hver gang systemet sammenligner to obliger og både størrelsesforskjellen og likhetsverdien faller under definerte terskelverdier

blir en mistankerapport generert. Rapporten består av de to obligidentifikatorene, likhetsverdien og størrelsesforskjellen. Hvor mange mistankerapporter som genereres for en gitt oblig angir dermed hvor mange andre obliger den ligner på.

Oblig Populær forkortelse av *obligatorisk oppgave*. Dette er oppgaver som studentene må få godkjent for å kunne gå opp til eksamen. Obliger er vanligvis programmeringsoppgaver som utføres uten tilsyn. Det er vanlig at studenter hjelper hverandre med disse oppgavene, noe som er greit så lenge det ikke resulterer i kopiering. Her bruker vi begrepet *oblig* om de individuelle besvarelsene på en obligatorisk oppgave.

Rensing Sammenligningssystemet har en enkel preprosessor/skanner som renser hver kodefil bl. a. ved å fjerne kommentarer og faste strenger.

Størrelsesforskjell Størrelsen på en oblig er målt i antall bytes i kodefilen etter rensing. Størrelsesforskjellen er forholdet mellom størrelsesdifferansen og størrelsen på den største obligen angitt i prosent. Lavere verdier øker likhetsverdiens gyldighet.

Symbol I forbindelse med sammenligningsalgoritmen snakker vi om opptelling av *symboler*. Symboler er forskjellige signifikante tekststrenger av vilkårlig størrelse.

Symbolvektor Den N-dimensjonale vektoren som regnes ut for hver oblig når N er antallet forskjellige symboler som telles. Antall forekomster av et symbol representerer koordinatet til symbolvektoren i den dimensjonen symbolet representerer. En symbolvektor starter i origo og vinkelen mellom to symbolvektorer er ett mål på likheten mellom de respektive obligene.

Terskelverdi Brukt i forbindelse med likhetsverdi og størrelsesforskjell. En mistankerapport genereres om de to verdiene er lavere enn de gitte terskelverdiene.

Vektorvinkel Om to obliger karakteriseres ved hver sin N-dimensjonale symbolvektor med felles startpunkt, betegner dette begrepet vinkelen mellom vektorene. Et potensielt mål på likhet.

Kapittel 2

Beskrivelse av oppgaven

For å kunne håndtere den valgte problemstillingen på en fornuftig måte er det nødvendig å kartlegge grunnlaget for oppgaven. Hva slags behov er det vi forsøker å dekke, hvilke hindringer kan vi bli stilt overfor og hvordan ser vi for oss at disse hindringene kan overstiges.

I dette kapittelet presenterer vi en oversikt over disse temaene og gjør noen grunnleggende forutsetninger for det videre arbeidet.

2.1 Bakgrunn for javasammenligneren

Institutt for informatikk tilbyr hvert semester flere kurs i javaprogrammering. Disse kursene krever at studentene får godkjent et antall obligatoriske oppgaver, populært kalt obliger. Disse obligene skal løses av hver student og leveres til studentens gruppelærer innen en gitt tidsfrist. Gruppelærerene retter de innleverte obligene og registrerer dem som enten godkjent eller ikke godkjent. Studenter som ikke får sine obliger godkjent får vanligvis levere på nytt et antall ganger. Ved semesterslutt må hver student ha alle obligene godkjent for å få gå opp til eksamen.

De obligatoriske oppgavene har flere funksjoner. De skal teste studentens kunnskaper jevnlig i løpet av semesteret slik at man kan regne med at de som går opp til eksamen har et kunnskapsnivå som er høyt nok til at de kan regne med å stå. Denne testingen medfører også at studentene oppmuntres til å jobbe med stoffet gjennom hele semesteret, dermed unngås skippertaksstudering. Dette bidrar til en mer effektiv læringsprosess og tilbakemeldingen fra studentene indikerer at de lærer svært mye av arbeidet med obligene.

Inntil for noen få år siden var det vanlig praksis at studentene leverte obli-

gene skriftlig til gruppelæreren ved å legge dem i en åpen hylle hvor gruppelæreren kunne hente den etter at leveringsfristen var passert. Gruppelæreren kunne dermed rette obligene i ro og mak, kommentere i margen om nødvendig og levere dem tilbake til studentene, enten på en gruppetime eller via den samme hyllen. Dette fungerte for det aller meste tilfredsstillende.

2.1.1 Det grunnleggende problemet

Høsten 2003 ble det rapportert om flere obliger som forsvant fra innleveringshyllene. De forsvunne obligene kom gjerne til rette igjen på et senere tidspunkt og man oppdaget at de sannsynligvis hadde blitt kopiert etter at lignende besvarelser ble rettet av andre gruppelærere. Man har tidligere også hatt mindre problemer med fusk i form av kopiering, men hvorvidt problemet har øket i omfang er usikkert da vi mangler eksakte data.

På grunn av disse forsvunne obligene og at det etterhvert har blitt for lite plass til hyller har man på kurset INF1010 bestemt at alle innleveringer skal skje elektronisk, dvs. via epost. Dette hindrer folk i å kvarte andres obligbesvarelser, men kopiering skjer ofte ved elektronisk utveksling av kode med eller uten vitende eller vilje fra den opprinnelige opphavsmannen. Dermed vil ikke elektronisk innlevering hindre alle tilfeller av kopiering.

Sentrale kursholdere ved Institutt for informatikk anslår at ikke mindre enn 20% av studentene på begynnerkursene fusker på obligene. Dette anslaget er blant annet basert på antall uvettige eksamensbesvarelser. I tillegg hender det at studenter ikke får med seg små endringer i oppgaveteksten i de obligatoriske oppgavene fra semester til semester,[1] noe som tyder på at man bruker besvarelsene til studenter som har tatt kurset tidligere. 20% fusk er allikevel betraktelig lavere enn flere estimater som er gjort i andre undersøkelser, bl. a. i [HF04] og i [BR04].

2.1.2 Hvordan redusere kopieringsproblemet?

Vi kan konstatere at det eksisterer et problem med fusk på programmeringskursene og at fusk ofte gir seg uttrykk i lignende obligbesvarelser. Vi vet at flere av disse tilfellene blir oppdaget av gruppelærere som både har rettet original og kopi, men vi må anta at det også forekommer tilfeller som ikke blir oppdaget, for eksempel kopiering blant studenter på forskjellige

[1] En oppgave omhandler et system for registrering av flypassasjerer. Et semester ble antall seterader i flyene endret fra 10 til 11, men en del studenter leverte allikevel besvarelser med ti seterader.

undervisningsgrupper. Dette medfører et behov for en effektiv metode for påvisning av kopiering av obligatoriske programmeringsoppgaver.

Selv om konsekvent påvisning av kopiering er verdifullt ville det vært bedre om kopiering ikke foregikk i det hele tatt. Dermed må man være oppmerksom på at en fullstendig strategi for å komme kopieringsproblemet til livs inkluderer teknikker for forebygging i tillegg til påvisning. Effektive forebyggingsteknikker fordrer kunnskap om hvem det er som kopierer og hvorfor de gjør det. Disse problemstillingene er utbredt innen informatikkundervisning og blir utførlig beskrevet bl. a. i [Wag00] og i [Rob02].

Da denne oppgaven fokuserer på påvisning av kopitilfeller vil ikke disse problemstillingene bli videre behandlet her.

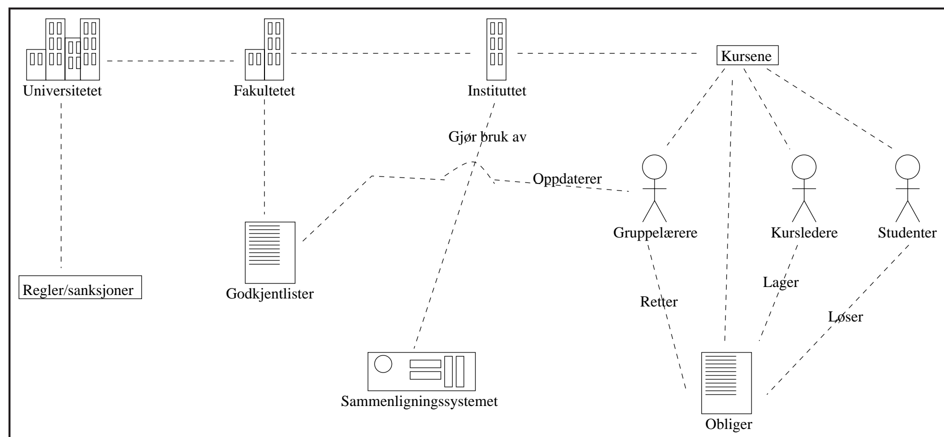
2.1.3 Problemstillingen

Vi har identifisert behovet for en mer effektiv metode for å oppdage fusk i form av kopiering av kode og det er dette behovet vi forsøker å møte med denne hovedoppgaven. Oppgaven består i å finne kriterier for hva som kan regnes for plagiat, finne algoritmer som kan brukes av et programsystem for å analysere innleverte obliger, programmere et analysesystem som implementerer denne algoritmen og bygge det inn i et koordinert system for elektronisk innlevering av obliger. Systemet må gi gruppelærere tilstrekkelig informasjon om analyseresultatene for hver oblig han skal rette og det må kunne foreta analyser på grunnlag av innleverte besvarelser fra tidligere semestre. Systemet må ivareta sikkerhet, være pålitelig og ta hensyn til relevante krav om personvern. Det bør også være mulig å hente statistisk informasjon ut fra systemet. Sist, men ikke minst, må systemet være lett å bruke for alle som må benytte seg av det.

Oppgaven vil forsøke å belyse problemene med å sammenligne javakodefiler av forskjellig lengde. Forskjellige sammenligningsmetoder vil bli beskrevet og vurdert. Endelig vil oppgaven foreslå et system for sammenligning av obliger i begynnerkurset ved Institutt for informatikk.

Et automatisert sammenligningssystem vil gjøre vurderingsprosessen raskere og vil lette arbeidet med retting av obliger. En gruppelærer vil raskt kunne få svar på om det er sannsynlig at obligen er kopiert og eventuelt hvilke lagrede obliger som ligner.

Fordelen ved maskinell sammenligning er at det går svært mye raskere å sammenligne et stort antall obliger på denne måten enn for hånd. Det er klart at maskinell sammenligning bare vil kunne gi uttrykk for sannsynligheten for at en oblig er kopiert. Vi forutsetter at lærer/sensor fortsatt vil måtte undersøke maskinens funn.



Figur 2.1: Modell av problemområdet

2.2 Definisjon av problemområdet

For å kunne definere problemet er det nødvendig å beskrive problemområdet i detalj, illustrert i figur 2.1. Problemet er som før nevnt kopiering av obliger studenter imellom og vi begrenser oss til besvarelser på oppgaver i Java ved Institutt for informatikk. Denne begrensningen betyr allikevel ikke at vi ikke behøver å ta hensyn til verden utenfor instituttet. Godkjentlistene, som vil måtte oppdateres etter at obliger er rettet og eventuelt fusk er påvist, er fakultetets ansvar. På samme måte må vi ta hensyn til at Universitetet har laget regler og sanksjoner for fusk som instituttet, og denne oppgaven, må ta hensyn til.

Hvert javaprogrammeringskurs ved instituttet har et antall obligatoriske oppgaver som studentene må få godkjent. Oppgavene er laget av kursledelsen og hver student må levere sine besvarelser til en gruppelærer som retter besvarelsene. Det vil i de aller fleste fall være gruppelærerene som oppdager eventuelt fusk i forbindelse med disse oppgavene, men de kan overlate en konfrontasjon med studenten til kursledelsen.

Det foreslåtte sammenligningssystemet må ta hensyn til de begrensningene som ligger i samspillet mellom elementene i problemområdet og samtidig legge forholdene til rette for mer effektiv evaluering av de sentrale funksjonene, i dette tilfellet obligsammenligning.

2.2.1 Definisjon av plagiat

Problemet vi tar opp består i å finne ut hvorvidt en bit med javakode er et plagiat av en annen. Dette er et komplisert problem. En bit med kode anses for å være plagiat hvis opphavsmannen har tatt utgangspunkt i et allerede eksisterende arbeid skapt av en annen part og resultatet i stor grad ligner på det opprinnelige arbeidet.

Om en gitt bit med kode kan sies å ligne på en annen er ofte avhengig av subjektive bedømmelser og kan sjelden besvares objektivt og kategorisk. Hvis man kan etablere at et gitt arbeid ligner på et annet, foregående arbeid er det rimelig å anta at det senere arbeidet kan være et plagiat av det foregående.

Om vi finner to biter med kode som ligner på hverandre må vi være klar over at det er en sjanse for at to opphavsmenn kan ha skrevet lignende kode uavhengig av hverandre. Dette er i enda større grad tilfelle når koden utgjør en obligbesvarelse, programmererne har mottatt samme undervisning og har lignende modningsgrad og programmeringsforståelse samt at det kan ha blitt lagt føringer på løsningene i form av f. eks. et programskjelett.

Gitt muligheten for at to biter med kode kan være tilfeldig like, og at likhet er en subjektiv størrelse, er det viktig at vi ikke lar maskinen autokratisk bestemme hvorvidt plagiat har forekommet eller ikke. Vi må begrense maskinen til å indikere om det kan eksistere likhet mellom to arbeider, men la det være opp til en bruker av systemet å bekrefte eller avkrefte hvorvidt det er snakk om kopiering.

2.2.2 Definisjon av likhet

I denne oppgaven bruker vi ordet likhet om graden av sammenfall mellom to biter med javakode. Hvordan likhet skal defineres i forhold til den aktuelle problemstillingen er et av hovedproblemene som må undersøkes. Selv om likhet ovenfor har blitt definert som en subjektiv størrelse, er vi nødt til å definere begrepet kvantitativt slik at det kan brukes i et datasystem. Vi må foreslå kriterier for likhet som kan benyttes i en sammenligningsmaskin.

I og med at likhet er et subjektivt begrep vil man kunne gjøre sammenligninger langs mange dimensjoner. I tillegg vil visse sammenligningsdimensjoner som er sentrale i noen tilfeller være irrelevante i andre sammenhenger. For eksempel vil man neppe kunne dra nytte av de samme sammenligningskriteriene når man sammenligner javakode som når man sammenligner prosa, eller bytekode for den del.

Siden det er umulig å bestemme med absolutt sikkerhet hvorvidt et arbei-

de er kopiert eller ikke. Dermed kan en binær resultatskala ikke benyttes. Vi velger heller å bruke en skala med likhetsverdier i intervallet $[0, \infty\rangle$. Tilfellet hvor to kodebiter er identisk like skal derfor gi et sammenligningsresultat på 0, mens to helt ulike obliger bør gi et relativt høyt tall. I praktisk bruk er vi sjelden interessert i annet enn den nederste delen av skalaen.

2.3 Ønskede egenskaper ved systemet

Det er mange problemer som må tas stilling til når man vurderer hvordan et slikt sammenligningssystem skal lages. Systemet må gi pålitelige resultater, det må være effektivt, det må være robust og det må være brukervennlig. Det er også nødvendig at alle studenter og gruppelærere bruker systemet, så utbredelse er et viktig kriterium.

De følgende avsnittene beskriver sentrale egenskaper ved systemet. Der det refereres til epostbaserte egenskaper er dette for å åpne for en epostbasert implementasjon.

2.3.1 Nøyaktighet

Det ideelle systemet skal kunne bestemme om to gitte kodefiler har samme opphav eller ikke. I tilfeller der det ikke dreier seg om originaler og blåkopier er det ikke mulig for et sammenligningssystem å avgjøre definitivt om det har foregått kopiering og i alle tilfeller må en endelig likhetsvurdering til syvende og sist gjøres av et menneske. Det maskinen kan gjøre er å beregne hvor stor sannsynligheten er for likhet mellom to obliger. Dette gjør det mulig for en person å sammenligne en mengde besvarelser uten å bruke alt for mye tid.

Resultatenes nøyaktighet avhenger ofte av kompleksiteten til algoritmene som brukes. Mer komplekse algoritmer krever større systemressurser som fører til at det tar lenger tid å komme frem til et resultat. Dermed er det nødvendig å finne et akseptabelt kompromiss mellom nøyaktighet og effektivitet.

Med nøyaktighet mener vi at systemet bør finne flest mulige kopier og klassifisere færrest mulig originale arbeider som suspekte. Hvis enhver kopi har en likhetsverdi under 2 og åpenbart originale arbeider har verdier over 10, kan vi hevde at systemet tilfredsstiller nøyaktighetskravet fordi det er mulig, med god margin, å differensiere mellom kopierte og originale obliger ut fra likhetsverdien alene.

2.3.2 Pålitelighet

Et annet viktig krav til systemet er at absolutt alle innleverte løsninger må lagres i databasen, godkjente eller ikke. Så lenge en student kopierer en løsning som ikke er lagret i databasen vil ikke systemet kunne påvise likhet med mindre en annen kopi allerede er lagret. Det er dermed viktig at samtlige medlemmer av den gruppen som har ansvar for bruken, benytter seg av systemet.

Med pålitelighet menes òg at systemet skal ha minimalt med nedetid. Dersom det allikevel er nede skal det gis tilbakemelding til brukeren om alle besvarelser som ble forsøkt registrert, men ikke lagret.

2.3.3 Effektivitetshensyn

Hvor effektivt systemet må være avhenger av hvordan det implementeres. Et interaktivt system bør ikke bruke mer enn et par sekunder på å presentere et resultat, mens et helautomatisk system basert på epost gjerne kan bruke lenger tid på analysen.

Når det gjelder tidsbruk er det viktig at maskinen er så rask som overhodet mulig. Premissene for systemet inkluderer et antall obliger i størrelsesorden 10 000 som sammenlignes med den innleverte obligen. Man antar at begynnerkurset har maksimalt 1000 studenter hvert år og at de obligatoriske oppgavene endres ikke sjeldnere enn hvert tiende år. I praksis er det kanskje ikke snakk om mer enn 3000 obliger maksimalt, på grunn av at oppgavene skiftes relativt ofte og at mange studenter ikke leverer obliger.

Det er et krav at maskinen skal kunne gjøre alle disse sammenligningene på få sekunder, det blir dermed nødvendig å benytte "one pass"-algoritmer, generelt av en orden ikke overstigende $\mathcal{O}(n)$.

I utgangspunktet skal systemet konstrueres slik at det kan benyttes av alle programmeringskurs, uavhengig av hvilke programmeringsspråk kursene benytter. Det er mulig å begrense sammenligningen til obliger i samme kurs, men på grunn av at de samme oppgavene kan bli gitt i forskjellige kurs (særlig ved omlegging av kurstilbudet) er ikke dette hensiktsmessig. På den annen side er det åpenbart at man kun sammenligner innleverte obliger med andre obliger skrevet i samme programmeringsspråk.

2.3.4 Robusthet

Hvis systemet skal få den nødvendige utbredelse er det nødvendig at det er robust. Systemet bør bygges med teknologi som kan skaleres i nødvendig

grad. I et epostsystem må det alltid være mulig å sende epost til systemet. Om systemet er nede må det være mulig å lagre innkommende epost for senere prosessering og analyse.

2.3.5 Brukbarhet

For at brukere skal komme fortest mulig i gang med å bruke systemet må det være lett å bruke og man må få gjort det man skal med færrest mulige tastetrykk. Et epostsystem bør både kunne støtte epost sendt direkte fra brukers epostprogram og epost sendt fra en kryptert webside.

2.3.6 Valg av algoritme

Kjernen i systemet er algoritmen for likhetsanalyse. Denne algoritmen må tilfredsstille flere krav, blant annet må den være effektiv og den må kunne gi pålitelige resultater. Det er flere algoritmer som må vurderes, blant annet nøkkelordbaserte algoritmer, sammenligning av kodelengde, lengste felles delstreng og metodekalltrær. Man kan tenke seg at flere av disse metodene blir brukt i kombinasjon.

2.3.7 Utbredelse av kjennskap til systemet

I utgangspunktet vil det være ønskelig at studentene har minst mulig å gjøre med analysesystemet. Vi antar at effekten på studentenes arbeidsmetoder vil bli størst dersom de er klare over at de blir testet, men ikke hvordan analysealgoritmen fungerer. På denne måten kan usikkerheten rundt systemet ivaretas samtidig som studentene vil være forsiktige med å fuske. For å bevare denne avstanden mellom studentene og systemet kan det være nyttig å la dem levere obligene elektronisk til gruppelæreren som registrerer samtlige obliger i systemet.

2.3.8 Mulige sikkerhetsproblemer

I et system som dette vil det alltid være en mulighet for at brukere kan kompromittere sikkerheten, enten med overlegg eller ukyndighet. Dette kan være at brukere skaffer seg tilgang til databasen og sletter eller endrer lagret informasjon, det kan være personer som mater virus inn i databasen. I tillegg er spam eller søppelpost alltid et problem man er nødt til å håndtere i et epostsystem.

Tilgang til systemet

I utgangspunktet skal tilgangen til systemet være begrenset til gruppelærere og/eller en eposttjener. I et interaktivt system skal alle gruppelærerene ha tilgang til å lese fra databasen og til å legge data til databasen gjennom systemet, men ikke til å slette eller endre informasjon. Så langt det er mulig bør endring av informasjon ikke føre til sletting av informasjon da det er nyttig å ha tilgang til historikk. Endring av informasjon bør heller skje i form av at oppdatert informasjon blir lagt til databasen og merket som gjeldende. På denne måten vil det være større mulighet for å rekonstruere informasjonsmengden om det skulle bli nødvendig.

Brukere bør kun ha tilgang til de delene av systemet de har behov for. Gruppelærere i et kurs bør f. eks. ikke kunne lese informasjon fra databasen som har med et annet kurs å gjøre.

Faren for virusinfeksjon av systemet

Når det gjelder virus introdusert i databasen vil dette neppe kunne skje i form av vedlegg til epost. Systemet vil kun lagre javakode i databasen, ikke binærkode. Systemet vil uansett ikke kjøre noen slags form for innmatede programmer, hverken binærkode eller bytekode generert gjennom kompilering av innleverte obliger (se avsnitt 4.1.1). Om systemet kompilerer de innsendte programmene vil bytekodefiler som genereres kastes og kun kompilatorens utskrift til `stdout` vil tas vare på.

Spamproblematikk

I utgangspunktet skal epost adressert til epostsystemet være formattert på en gyldig måte, definert og gjenkjennelig av systemet. Epost som ikke konformerer til dette formatet blir kastet og vil ikke sende kvittering tilbake til avsender. Systemet vil se etter vedlegg av typen `Text/PLAIN` eller `Text/(X-)JAVA`, alle andre typer vedlegg vil bli kastet.

Søppelpost vil sannsynligvis ikke være formattert på en måte som konformerer til systemets krav, og vil for det aller meste kastes. Ugyldig formattert post som kastes av systemet skal ikke generere epostkvitteringer siden dette antagelig vil øke mengden av søppelpost som mottas.

2.3.9 Andre problemsituasjoner

Andre tilstander som kan skape problemer, om ikke direkte forårsake sikkerhetsfeil, er brukere som mater samme kode inn i systemet flere ganger. Slike dubletter vil ikke påvirke påliteligheten eller nøyaktigheten til systemet, men kan påvirke effektiviteten hvis dette skjer ofte nok. En åpenbar løsning på dette problemet er å begrense alle brukere til å levere kun én løsning på hver oppgave. Dette er allikevel ingen god løsning da de fleste studenter er nødt til å levere en eller flere obliger om igjen etter å ha fått den første løsningen underkjent. Om det er ønskelig bør det være mulig å begrense antall innleveringer av samme oblig fra samme student. De færreste studenter behøver mer enn 3 forsøk på samme oppgave, så systemet kan tillate 3-5 forskjellige versjoner av samme besvarelse. Denne problematikken kan håndteres ved å legge inn ekstra informasjon i hodet i javafilen, f. eks. et versjonsnummer.

2.3.10 Konklusjon

Kravene som stilles til systemet er altså at det skal være robust og pålitelig. Det må være lett å bruke og tilby de nødvendige operasjoner. Det skal ikke være mulig å kompromittere systemsikkerheten utenfra. Algoritmen som brukes må være rask og pålitelig. Systemet må, kort sagt, raskt kunne gi pålitelig informasjon til brukeren.

2.4 Oppsummering

Etter å ha gitt en innføring i bakgrunnen for dette arbeidet har vi definert to av de mest grunnleggende begrepene, *likhet* og *plagiat*, slik de vil brukes videre i denne oppgaven. Videre har vi kartlagt de behovene systemet vårt må tilfredsstille, disse inkluderer bl. a. *nøyaktighet*, *pålitelighet*, *effektivitet*, *robusthet*, *brukbarhet* og *sikkerhet*.

Disse punktene vil utgjøre grunnlaget for det videre arbeidet med sammenligningssystemet.

Kapittel 3

Metoder for likhetsvurdering

Flere metoder har blitt utviklet for å påvise likhet mellom programmer. Vi vil se på noen av disse metodene og vurdere hvorvidt de passer til vårt formål.

Metoden vi velger bør ideelt være enkel å implementere, gi eksakte resultater samt kjøre effektivt.

3.1 Lengste felles delstreng

Flere sammenligningssystemer baserer seg på deteksjon av en lengste felles delstreng og/eller antall felles delstrenger over en viss lengde. Denne metoden vil detektere gjenbruk av kodesegmenter uavhengig av hvordan de er stokket om hverandre i forsøk på å endre kodens utseende. Ved å fjerne kommentarer og skifte ut alle variabelnavn, faste strenger og tegn med en generell identifikator før sammenligning unngår man også å bli villedet av endrede navn på disse elementene. En slik fremgangsmåte er beskrevet i "Detecting Copied Submissions in Computer Science Workshops" [GH89]. Her blir koden først analysert og renset for kommentarer. Faste strenger og tegn blir erstattet av uniforme symboler blant andre endringer. Denne prosessen minner om oppgaven til koderenseren vi beskriver i avsnitt 5.2.4. Til slutt leter man etter felles delstrenger blant to rensede kodefiler.

Ulempen med sjekking av felles delstrenger er at metoden kan være tidkrevende, særlig hvis brukt for å sjekke en oblig mot for eksempel 10 000 forekomster i en database. Hvis en sammenligning tar anslagsvis 1/100 sekunder vil en sammenligning av 30 innleverte obliger mot 10 000 forekomster ta rundt 50 minutter, noe som ikke er i nærheten av optimalt. Tiden det vil ta å rense 30 innleveringer for kommentarer og erstatte navn og

faste strenger med faste identifikatorer kan måles i sekunder og vil dermed være insignifikant.

3.2 Likhetspåvisning med YAP3

YAP3 [Wis96] er den tredje versjonen av et sammenligningssystem for programtekster. Felles for alle versjonene av YAP er at sammenligningen foregår i to faser.

I den første fasen renses programtekstene for kommentarer og faste strenger i tillegg til videre uniformering. Denne fasen er relativt uendret mellom YAP-versjonene. Etter tekstrensingen stokkes metodene om i henhold til kallrekkefølge. Den resulterende symbolsekvensen blir deretter sammenlignet med de tilsvarende symbolsekvensene for andre programtekster i neste fase.

Det er i fase nummer to vi ser endringene mellom versjonene der forskjellige algoritmer brukes for å sammenligne symbolsekvensene. Første versjon av YAP [Wis92] bruker UNIX' `sdiff` som implementerer "lengste felles delstreng"-sammenligning, YAP2 bruker Heckels algoritme, mens en algoritme kalt Running Karp-Rabin Greedy String Tiling (RKR-GST) ble utviklet for bruk i YAP3.

Lengste felles delstreng er problematisk å bruke siden man enkelt kan bryte opp ellers identiske tekstsekvenser ved å stokke om setninger. Heckels algoritme, som den er implementert i YAP2, har en lignende svakhet. Denne algoritmen starter med å lete etter identiske biter med tekst i to filer og søker etter likhet i tilstøtende tekstbiter slik at identiske tekstblokker kan identifiseres uansett hvor i teksten de befinner seg. Resultatet av sammenligningen er basert på den totale lengden av identiske tekstblokker. Problemet er som i første versjon at stokking av setninger og introduksjon av ekstra setninger bryter opp blokkene og reduserer nøyaktigheten.

RKR-GST-algoritmen brukt i YAP3 er en videreføring av de forrige algoritmene i det den også baserer seg på sammenligning av delstrenger, kalt "tiles". For å øke effektiviteten til algoritmen sammenligner den hash-verdier utregnet for delstrengene. Denne fremgangsmåten har vist seg å gi mer nøyaktige resultater enn de foregående versjonene av YAP, men er også relativt komplisert.

3.3 Likhetspåvisning med CBR-teknikker

Cunningham og Mikoyan [CM93] foreslår en sammenligningsstrategi inspirert av CBR-teknikker (Case Based Reasoning). Først må man finne gode kriterier for å karakterisere tilfellene (programtekstene), deretter kan man klassifisere dem for så å kunne hente frem de mest sannsynlige kopikandidatene for hver sammenligning.

Forfatterne legger vekt på viktigheten av å finne gode representasjonskriterier for programtekstene. Om karakteriseringen er lite gjennomtenkt vil det ikke hjelpe om algoritmen er aldri så god. Sammenligningssystemet som beskrives i denne artikkelen karakteriserer programtekster blant annet ved funksjonskalltrær. Denne metoden går ut på å følge alle funksjonskallene i programmet og konstruere et tre med rot i `main()`. Hver gang parseren finner et funksjonskall går den videre til denne funksjonen og følger funksjonskallene derfra.

Forfatterne beskriver en metode å gjøre dette på ved å komprimere treet til en streng av påfølgende metodekall bestående av blant annet metodenavnet og et tall som representerer hvilket nivå kallet forekommer på. Rekkefølgen på metodekallene i strengen defineres av en dybde-først-traversering av treet.

Fordelen med denne metoden er at trestrukturen i seg selv kan identifisere et program, uavhengig av variabel- og funksjonsnavn. Ved å sammenligne maksimal dybde og bredde i to trær har man en indikasjon på hvorvidt de to trærne har opphav i lignende programmer. Denne sammenligningen kan for eksempel gjøres ved å finne felles delstrenger i to trær, som er over en viss lengde.

Tanken er at de mest grunnleggende programstrukturene gir en bedre karakteristikk av programmet enn forekomster av nøkkelord. Visse nøkkelord, de som angir datastrukturer, karakteriserer allikevel programmet bedre enn identifikatorer, og kan gi relativt gode karakteristikker.

Denne metoden virker relativt komplisert, men vil ikke kreve betydelig mer av ressurser enn en lengste-felles-delstrengoperasjon for hver sjekk. Allikevel kan dette være snakk om opptil 10 000 slike operasjoner for en sammenligning om vi forutsetter at sammenligningsdatabasen inneholder $\sim$10 000 innleveringer. Se forøvrig avsnitt 3.1.

Å bruke funksjonskalltrær som grunnlag for programkarakterisering i vårt system vil antagelig være lite hensiktsmessig i og med at det krever implementasjon av en relativt komplisert preprosessor.

3.4 Likhetspåvisning med symboltelling

Som et alternativ til allerede etablerte sammenligningsmetoder foreslår vi en enkel metode basert på opptelling av utvalgte symboler (nøkkelord i Java) i en programkodefil. Kriteriene for valg av symboler må være nøye gjennomtenkt, jfr. avsnitt 3.3, siden metodens nøyaktighet avhenger sterkt av kvaliteten på symbolutvalget.

For å kunne gjøre raske sammenligninger av to programmer basert på antall symbolforekomster i hvert program kan vi beskrive mengden av symbolforekomster som en N-dimensjonal vektor mellom origo og et punkt med koordinater beskrevet ved symbolantallene, gitt at vi teller opp forekomstene av N symboler. Vi kan tenke på denne vektoren som programmets fingeravtrykk. Symboltellingen kan gjøres i løpet av én gjennomgang av filen samtidig som den renses og den reelle størrelsen måles. Dette er svært effektivt sammenlignet med YAP og andre delstrengbaserte algoritmer og bevarer samtidig en av fordelene ved disse metodene, nemlig at stokking av kodeelementer ikke påvirker resultatet.

Et mulig mål på likheten mellom to programmer er avstanden mellom endepunktene til de respektive vektorene. Jo nærmere hverandre endepunktene ligger dess større likhet kan vi anta programmene har til hverandre siden kort avstand indikerer et relativt likt antall forekomster av de forskjellige symbolene i de sammenlignede programmene.

Et annet mulig likhetsmål er vinkelen mellom vektorene. Avstandsmålet er en funksjon av både vektorlengdene og vinkelen, men vektorlengden er hovedsaklig en funksjon av programstørrelsen og ved bruk av denne metoden vil det være av størst interesse å sammenligne programmer av noenlunde samme størrelse da stor størrelsesforskjell som oftest indikerer liten grad av likhet.

Siden symboltelling klart er mer effektivt enn de foregående metodene, og i tillegg er mindre komplisert, velger vi å implementere denne metoden i det foreslåtte analysesystemet.

To varianter av denne metoden blir nærmere beskrevet i kapittel 5.

3.5 Oppsummering

For å få gode resultater fra delstrengbaserte metoder synes det å kreve klart langsommere algoritmer enn det som er hensiktsmessig å bruke i vårt prosjekt. Metoden beskrevet i avsnitt 3.4 vil antagelig gi resultater på linje med YAP3, gitt en grundig utvelgelse av nøkkelord og en algoritme som kan

utnytte informasjonen effektivt. Som nevnt kommer vi derfor til å bruke symboltelling som sammenligningsmetode videre i oppgaven.

Kapittel 4

Systemet fra et brukerperspektiv

Vi har i de foregående kapitlene redegjort for de forutsetningene vi vil legge til grunn for arbeidet. I dette kapittelet vil vi beskrive et første utkast til et sammenligningssystem, vi vil undersøke nærmere de bruksmønstre systemet vil måtte møte og dermed komme frem til et sett av funksjoner som bør implementeres.

Det mest grunnleggende er å definere hva slags informasjon systemet skal motta, hvordan det skal behandle informasjonen og hva slags informasjon det skal gi tilbake til brukeren.

Vi vil skissere et grunnleggende brukergrensesnitt med forslag til hvordan informasjon mates inn i systemet og hvordan tilbakemelding presenteres til brukeren. Ytterligere nødvendig funksjonalitet vil også beskrives.

Denne beskrivelsen av brukergrensesnittet vil inngå i grunnlaget for spesifikasjonen av sammenligningsfunksjonene i systemet.

4.1 Et utkast til et praktisk sammenligningssystem

For å kunne bruke systemet i praksis ved en utdanningsinstitusjon, i dette tilfellet Institutt for informatikk ved Universitetet i Oslo, er det behov for en måte å lagre innleverte obliger. Den mest åpenbare måten å gjøre dette på er å benytte seg av en database hvor man både lagrer kildefilen i sin helhet og eventuelt de forskjellige nøkkelorddataene. I tillegg vil det være relevant å registrere innleveringssemester (siden oppgavetekstene kan endres), dato, godkjenningsdato og annet. Andre data kan tenkes å være

relevante avhengig av hvor nært maskinen skal integreres med godkjenningssystemet og godkjentlistene.

Det er et mål at innleveringen av obliger skal automatiseres. Dermed er det behov for et grensesnitt mot sammenligningsmaskinen og databasen slik at hver gruppelærer kan sjekke sine studenters obliger for likheter mot resten av obligene i databasen. Studentene skal levere inn kildekode via epost og man skal kunne mate en oblig inn i databasen ved et enkelt håndgrep (eller tastetrykk). Derfor er det viktig at man innfører et standardformat for kildekoden til obligene hvor relevant innleveringsinformasjon legges i kommentarer i begynnelsen av kildekoden. Eksempelvis kan dette se slik ut:

```
//Forfatter:   ola@ifi.uio.no
//Kurs:        INF1010
//Gruppe:      11
//Oppgave nr.: 6
```

Et provisorisk forslag som blir benyttet i kurset INF1010 våren 2004 er hodet
`// <brukernavn> g-<gruppenr> o-<oppgavenr>`.

På denne måten kan systemet kjenne igjen og plukke ut relevant informasjon fra kildekoden, som den så lagrer i databasen før sammenligning og vurdering utføres. Hvis hodeformatet er feil vil det resultere i en feilmelding når filen forsøkes lagret i databasen, med mulighet til å ignorere hodet og legge inn dataene manuelt. Avhengig av hvor automatisert systemet er kan det returnere obligen til studenten med beskjed om å rette hodet før obligen regnes som innlevert. Ulempen ved denne fremgangsmåten er at studentene erfaringsmessig ofte formatterer denne informasjonslinjen galt, noe som kan være frustrerende.

Hvis systemet er så automatisert at obliger sendt til en spesiell epostadresse automatisk blir lagt inn i databasen, kan man tenke seg at sammenligningen blir utført med en gang og resultatet blir meddelt gruppelæreren på epost uten at han behøver å foreta seg noe i forhold til sammenligningssystemet. Om dette er mulig å få til gjenstår å se.

Uavhengig av automatiseringsgrad vil det være behov for å underrette gruppelæreren når en oblig som tidligere har blitt registrert som original senere blir registrert lik en nyinnlevert oblig. Dette vil antagelig skje ved at gruppelæreren som retter den nyinnleverte obligen underretter sin kollega. Et helautomatisk system vil sende epost til de berørte parter.

4.1.1 Mulige utvidelser av systemet

Etter å ha spesifisert de grunnleggende kravene for et brukbart obliginnleveringssystem kan det være interessant å se på mulige utvidelser av systemet. En naturlig tilleggsmodul kan tilby statistisk informasjon over hvor mange tilfeller av kopiering man har funnet, en modul kan også tilby funksjonalitet for søking blant innleverte obliger.

Det kan også være interessant å se på en nærmere integrering av obliginnleveringssystemet mot godkjentlistesystemet.

Det kan være interessant for gruppelæreren å vite om innleverte obliger kompilerer. Selv om det vanligvis går raskt å kompilere et javaprogram kan det være nyttig å få denne informasjonen fra systemet sammen med likhetsinformasjon. Dette vil antagelig kun være nyttig i et epostbasert system i og med at kompileringstiden vil gjøre ventetiden i et interaktivt system for lang. Kompilering kunne muligens implementeres som en valgmulighet i et interaktivt system. Et epostbasert system vil kompilere koden etter å ha undersøkt den for likhet. Gruppelæreren vil, sammen med likhetsinformasjonen, få informasjon om hvor mange feilmeldinger kompilatoren gir.

4.2 Utkast til et innleverings- og sammenligningssystem

Det er visse funksjoner systemet må tilby uansett hvilket grensesnitt brukerne presenteres for.

- Systemet må inneholde funksjonalitet for lagring av innlevert kode. Vi antar at en databaseløsning vil fungere tilfredsstillende. Til å begynne med antar vi at MySQL vil tilfredsstille systemets behov.

- Systemet må lagre informasjon om når en oblig er levert, hvilket kurs og hvilken oppgave innleveringen gjelder, informasjon om hvilken gruppelærer som er ansvarlig for innleveringen og hvem som har skrevet den.

- Systemet må tilby funksjonalitet for å sammenligne en ny oblig med allerede eksisterende besvarelser for samme kurs og oppgave.

- Systemet må gi tilbakemelding på om den nye obligen ligner på allerede lagrede obliger. Den må liste opp de obligene som ligner mest og angi hvor mye de ligner.

- Om en allerede lagret oblig ligner på en nettopp innlevert oblig bør systemet melde fra til ansvarlige gruppelærere for begge obliger, så sant begge er levert samme semester.

4.2.1 Funksjonalitet for et system med grafisk grensesnitt

- Systemet skal presentere brukeren for et grensesnitt med funksjonalitet for å lagre en oblig i databasen.
- Samtidig som den lagres skal den sjekkes mot alle relevante obliger og sammenligningsdata skal også lagres.
- Systemet skal presentere et resultat som detaljert ovenfor.

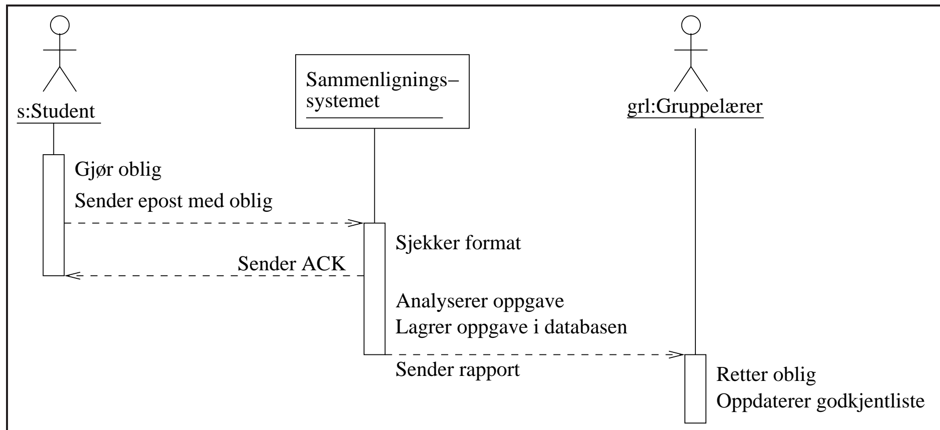
4.2.2 Funksjonalitet for et system basert på epost eller annen asynkron kommunikasjon

Om det er aktuelt å lage et slikt system må det ha følgende funksjonalitet.

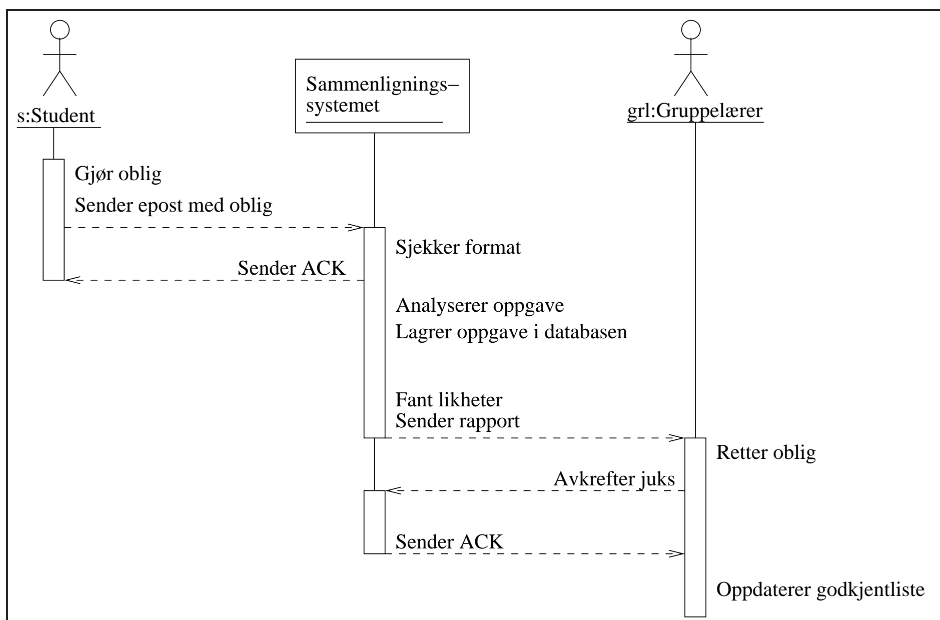
- Systemet skal tillate at studenter sender inn obliger som vedlegg til epost til en gitt adresse. Eposten og vedleggene må være formattert på en gyldig måte.
- Systemet skal kunne håndtere innkommende epost, lagre relevante vedlegg (javakode) i databasen og utføre en sammenligning.
- Systemet skal sende bekreftelse på mottatt epost til studenten.
- Systemet må sende den opprinnelige eposten videre til ansvarlig hjelpelærer med vedlagt sammenligningsresultat (se ovenfor).
- Systemet må sende epost til eventuelle ansvarlige gruppelærere for obliger den nye innleveringen viser seg å ligne på.

4.3 Diskusjon av fordeler og ulemper med de respektive systemene

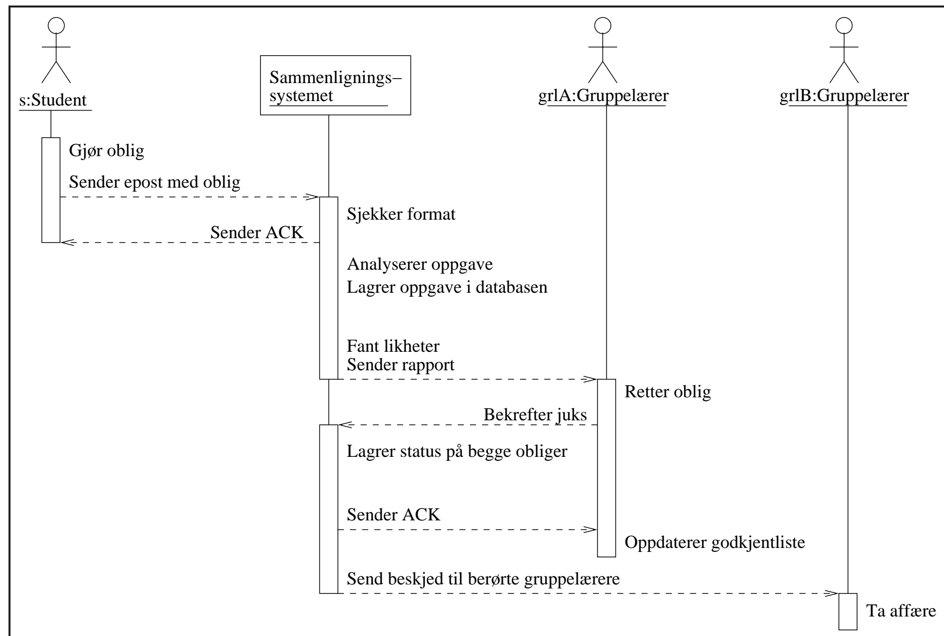
Et interaktivt system vil kreve at gruppelæreren aktivt mater innleverte obliger til systemet. Selv om dette nok krever litt arbeid vil han antagelig ha følelsen av større kontroll over hvordan systemet virker og hva slags informasjon han har tilgang til. Studenten vil på den annen side i mindre grad



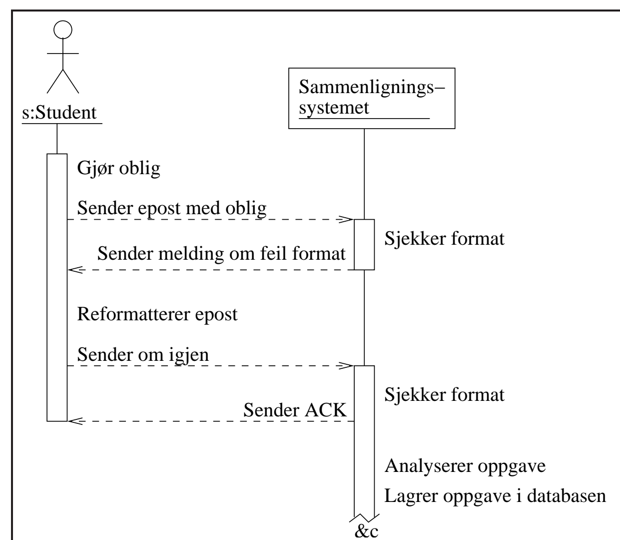
Figur 4.1: Diagram over et solskinnsscenario hvor ingenting går galt med epostsystemet



Figur 4.2: Epostsystemet melder om likheter, men gruppelærer avkrefter kopiering



Figur 4.3: Kopiering bekreftes av gruppelæreren



Figur 4.4: Epostsystemet reagerer på feil format i epost eller javakode. Ingen kvittering gitt

behøve å forholde seg til et spesielt format på epost og javakode, og vil sende epost direkte til gruppelærer, noe som muligens vil kunne oppfattes som tryggere for studenten. Et interaktivt system vil med andre ord kun forutsette bruk av gruppelærere, som er en mye mindre gruppe brukere enn studentgruppen, men det krever også en aktiv innsats av gruppelærerene å bruke det. Det inviterer ikke direkte til bruk slik et epostbasert system vil gjøre, og dette kan føre til at terskelen for å ta systemet i bruk vil være høyere.

For å få gruppelærerene til å registrere obliger i systemet må nødvendigheten av dette gjøres eksplisitt av kursledelsen og nevnes i gruppelærerkurset. Det å registrere 100+ obliger i systemet betyr ikke nødvendigvis mye merarbeid for gruppelærerne om systemet konstrueres slik at registrering av en mengde obliger krever færrest mulig tastetrykk.

Et epostbasert system vil forutsette at studenten sender epost direkte til systemet, på et gyldig format. Dette bør ikke være en uoverkommelig vanskelighet, men kan kanskje by på noen problemer for en ny bruker av systemet. For gruppelæreren derimot vil et epostbasert system sannsynligvis fungere godt dersom det oppfyller kravene om å gi best mulig relevant informasjon. Dette systemet forutsetter aktiv bruk av studenten, noe som sikres ved at obliger bare regnes som innleverte om de er levert via epost til systemet. For at systemet skal kunne fungere etter intensjonen er det nødvendig at det innføres obligatorisk innsending av obliger til systemet for alle studentene på kurs hvor systemet brukes. Gruppelærerene vil ha en mer passiv rolle i forhold til systemet idet de hovedsaklig vil være mottagere av informasjon fra systemet. Denne informasjonen vil være nyttig i arbeidet med å rette obliger og den vil sendes på epost til gruppelærerene slik at informasjonen er tilgjengelig når de finner det for godt å benytte den.

4.4 Bruksmønstre

Vi har tidligere sett på systemets oppgaver og vil nå ta for oss de forskjellige aktørenes roller. Aktørene har blitt definert til å inkludere studenter som leverer obliger, gruppelærere som retter obliger og kursledere som har behov for statistikk fra systemet. Vi undersøker bruksmønstre for to tenkte versjoner av systemet, en basert på et grafisk brukergrensesnitt og en annen basert på epost.

Beskrivelsene nedenfor forutsetter et sammenligningsprogram for javakode og en database som ligger i bunnen av det tenkte systemet. Disse grunnleggende elementene vil ikke kunne manipuleres direkte av noen brukere, men vil kunne nås via forskjellige grensesnitt tilgjengelige for de forskjel-

lige brukerne. Det er disse grensesnittene som diskuteres her. Forskjellige scenarier som kan inntreffe ved bruk av systemet skisseres i figurene på side 25-26.

4.4.1 Studentenes tenkte bruksmønster

Beskrivelsen av studentenes arbeidsprosedyre kan vanligvis reduseres til følgende: De løser oppgaven, de leverer inn obligen og de får den tilbake rettet og godkjent eller med beskjed om å levere en gang til.

Uavhengig av om det foreslåtte systemet tas i bruk, og uavhengig av hvilken versjon som velges, vil høyst ett av stegene beskrevet ovenfor måtte endres. Studentens innlevering av obligen kan enten skje ved at han leverer direkte til gruppelæreren eller at han leverer til systemet via et elektronisk grensesnitt.

Om studenten selv skal ha ansvaret for å registrere obligen sin i systemet vil det enten være snakk om et epostgrensesnitt eller et web-grensesnitt.

Epostgrensesnittet

I dette tilfellet vil studenten måtte sende en velformet epost til en spesiell epostadresse. Eposten må inneholde javakoden, inkludert som et vedlegg av typen "Text/(X-)JAVA" og eventuelt andre deler av innleveringen som egne vedlegg. Eposten må inneholde informasjon om studentens brukernavn (dette vil kunne hentes fra eposthodet om den er sendt fra en UiO-adresse), oppgavens nummer, hvilket kurs det er snakk om og hvilken gruppelærer som skal rette obligen.

Når dette er gjort skal systemet sende en epost tilbake til studenten med bekreftelse på at innleveringen er mottatt. Om innleveringen er galt formattert skal kvitteringseposten be om at obligen leveres inn på nytt og henvise til formatbeskrivelsen.

Web-grensesnittet

Alternativt kan man se for seg et web-basert grensesnitt hvor studentene logger inn med brukernavn og passord (helst UiO-passord) og presenteres for en side hvor man får se hvilke kurs man er oppmeldt i og kan velge ett av dem, man får også beskjed om hvilken gruppe man er registrert på (denne informasjonen kan hentes fra `user` eller `ng`). Her kan studenten endre gruppen han vil levere til, han har muligheten til å skrive en kommentar til innleveringen og kan laste opp de filene som er relevante for innleveringen.

Når studenten klikker på Send vil javafilene bli analysert av systemet, all informasjonen studenten har gitt vil bli samlet i en pen epost og sendt til gruppelæreren sammen med resultatet fra analysen.

Dette vil nok være enklere for studentene å forholde seg til enn å komponere eposter etter et sett med regler.

Konklusjon

Tatt i betraktning at det er et mål at studentene i størst mulig grad skal være uvitende om systemets funksjoner og bruksområde vil vi ikke lage et grensesnitt for studentene mot systemet. Dette betyr at ekstraarbeidet denne løsningen påfører gruppelærerene må gjøres minimal gjennom løsninger i andre deler av analysesystemet.

Om et grensesnitt for studentene allikevel skulle implementeres ville et grafisk grensesnitt være det beste ut fra et brukervennlighetshensyn.

4.4.2 Gruppelærerens tenkte bruksmønster

Om vi skal redusere beskrivelsen av gruppelærerens arbeidsprosedyre på samme måte som studentens, ovenfor, vil vi kunne beskrive den som følger: Motta en innlevert oblig, rette obligen, registrere den som godkjent eller ikke og returnere den til studenten med en tilbakemelding.

Som for studenten er det kun ett av disse stegene som vil endres om en versjon av systemet tas i bruk, nemlig mottakelsen av en innlevert oblig. Om studenten leverer direkte til gruppelæreren vil gruppelæreren måtte mate alle de innkomne obligene inn i systemet og deretter observere resultatene av analysen via et grafisk grensesnitt. Dette kan medføre en del arbeid for gruppelæreren. Om studentene selv leverer inn til systemet, på en av måtene skissert ovenfor, vil gruppelæreren motta studentenes innlevering på epost, med analyseresultatet og eventuell lignende, lagret kode som vedlegg.

Vi går altså ut fra at gruppelærerne må legge inn alle innleveringer på egen hånd. Denne løsningen vil forskåne studentene fra enhver kontakt med sammenligningssystemet, men vil kreve et eget grafisk grensesnitt for gruppelærerenes aktivitet. Det vil også legge mer arbeid på gruppelærerene i en periode hvor de allerede har det travelt med retting.

Gruppelæreren som aktiv bruker

Her vil gruppelæreren måtte legge hver besvarelse han har fått inn i systemet på egenhånd. Grensesnittet vil måtte tilby mulighet til å velge filer, det vil måtte presentere resultatet raskt og oversiktlig. I tilfeller hvor innlevert kode ligner på allerede lagret kode bør innlevert og tidligere registrert kode kunne vises i forskjellige vinduer for enkel sammenligning. Responstiden er viktig i en slik interaktiv løsning og gruppelæreren skal ikke måtte behøve å vente i alt for lang tid før analysen er ferdig.

Vi antar i utgangspunktet at det er uheldig å påføre gruppelærerene det ekstraarbeidet denne løsningen medfører, det bør derfor være et mål å gjøre arbeidet med databaseregistreringen så effektivt som mulig. Blant annet bør det være mulig å be systemet om å lagre/analysere alle javafiler på et filområde og brukeren bør kunne utføre sammenligningen med så få operasjoner som mulig. I tillegg kan det være interessant å se på muligheter for å lagre/analysere filer via kommandolinjen i Unix.

En nødvendig funksjon i et gruppelærerdrevet system må være at systemet gir informasjon til andre gruppelærere hvis studenters obliger er blitt funnet lignende nylig innleverte obliger. Slik informasjon kan gis via epost.

Konklusjon

Med utgangspunkt i konklusjonen i avsnitt 4.4.1 velger vi å implementere et grafisk grensesnitt for gruppelærerene mot analysesystemet. Grensesnittet må være så effektivt og enkelt som mulig. Et forslag til grensesnittmodell kan sees i figur 4.5.

4.4.3 Kursledelsens tenkte bruksmønster

De ansvarlige lærerene for et gitt kurs har generelt begrenset befatning med prosessen med å rette obliger, selv om det i noen kurs holdes jevnlige gruppelærermøter hvor kursledelsen får informasjon om de forskjellige gruppelærerenes erfaringer. For oversiktens skyld kan det være nyttig for kursledelsen å få en statistisk representasjon av relevante innleveringsdata fra systemet. Dette kan implementeres som en egen modul med tilgang til å lese informasjon fra databasen, men adskilt fra algoritmesubsystemet.

Et tenkt grensesnitt for denne statistikken bør kunne brukes over WWW og må tillate brukeren å velge kurs og muligens også enkeltgrupper. Statistikken bør presenteres på en så oversiktlig måte som mulig.

```
christkk logget inn for gruppe 13

Velg kurs     inf1010        Filer
Velg oblig    1              ~/obliger/o1/o1-1.java
Velg gruppe   13             ~/obliger/o1/o1-2.java
Velg filer    Browse...      ~/obliger/o1/o1-3.java
Analyser                     ~/obliger/o1/o1-4.java
                             ~/obliger/o1/o1-5.java
                             ~/obliger/o1/o1-6.java
                             ~/obliger/o1/o1-7.java
                             ~/obliger/o1/o1-8.java

Analyserer o1-1.java - Ingen signifikant likhet funnet
Analyserer o1-2.java - Ingen signifikant likhet funnet
Analyserer o1-3.java - Ingen signifikant likhet funnet
Analyserer o1-4.java - Ingen signifikant likhet funnet
Analyserer o1-5.java - Ingen signifikant likhet funnet
Analyserer o1-6.java
      Ligner på o1-h02-34.java
      Ligner på o1-h02-26.java
Analyserer o1-7.java - Ingen signifikant likhet funnet
Analyserer o1-8.java - Ingen signifikant likhet funnet
```

Figur 4.5: Skisse av web-grensesnitt for gruppelærere

4.5 Definisjon av resultatdata

Hver gang systemet blir matet med informasjon produserer det resultatdata og dirigerer disse til opptil flere mottagere. Hva slags resultatdata som blir gitt, og hvordan den blir levert avhenger av hvilke grensesnittløsninger man velger.

4.5.1 Resultatdata til gruppelærerene

Gruppelærerene vil måtte forholde seg til et web-grensesnitt hvor de selv registrerer alle obligene de har fått fra studentene, se figur 4.5. I de tilfellene en gruppelærer underviser i flere kurs vil han bli bedt om å velge kurs, om han har flere grupper i samme kurs må han velge gruppe. Om gruppelæreren kun underviser i ett kurs eller én gruppe vil systemet finne de nødvendige data uten at brukeren behøver å taste det inn.

Brukeren vil bli bedt om å angi de programkodefilene som skal analyseres og filnavnene vil dukke opp i et vindu for referanse.

Når brukeren har angitt filnavnene, trykker han på analyser-knappen og

resultatet for hver fil vil vises i et beskjedfelt nederst i vinduet. Dersom ingen likhet blir funnet går systemet videre og sjekker neste fil. Dersom likhet blir funnet blir det skrevet ut en lenke til filen som blir sjekket og til filene som ligner i beskjedvinduet, filene somligner vil bli rangert etter likhet slik at gruppelæreren kan velge å se på de som er mest relevante. Deretter går programmet videre og sjekker neste fil.

Dersom gruppelæreren ønsker å sjekke om det kan være snakk om kopiering kan han klikke på lenkene i beskjedvinduet som vil bringe javakoden opp i nye nettleservinduer. Han dermed sammenligne koden på en enkel måte.

4.5.2 Statistikk fra systemet

Hver gang systemet foretar en analyse av kildekodefiler vil systemet registrere hvilke filer som ligner på hverandre og hvilke som ikke ligner på noen andre. Denne informasjonen kan kombineres med innleveringstidspunktet for hver oblig for å produsere statistikk over hvor mange innleverte obliger hvert semester (eller totalt) som ligner på andre. Denne informasjonen kan også brytes ned på gruppenivå, men hvorvidt det finnes et behov for så detaljert informasjon er usikkert. Personvernhensyn vil også spille en rolle i utformingen av statistikken fra systemet.

Uansett vil dette kunne være interessant for kursledelsen, som vil ha tilgang til statistikken gjennom en nettleser.

I utgangspunktet vil det kunne være interessant å presentere informasjon over hvor mange innleverte obliger som er blitt klassifisert som mistenkelige for hvert semester og totalt, og hvor mange obliger som er blitt levert i det hele tatt. Denne statistikken må kunne lages for en hvilken som helst gitt likhetsverdi. For det meste vil det være statistikken for inneværende semester som er av størst interesse.

Videre vil systemet kunne presentere informasjon over hvordan likhetsverdien på de mistenkelige obligene fordeler seg, altså hvor mange som har en euklidsk avstand under 3, 2, 1 eller kategorisert etter vektorvinkel.

Informasjonen denne statistikken bygger på kan hentes rett fra databasen og presenteres grafisk i en nettleser.

4.6 Avsluttende betraktninger

På grunn av kravet til oversiktlighet og hensynet til nye studenter vil vi gå ut fra at det beste er å implementere et system med web-basert grafisk bru-

kergrensesnitt. Vi antar at gruppelærerene er primærbrukere av systemet og forutsetter at kursledelsen i de forskjellige kursene motiverer gruppelærerene til å bruke systemet. Dette forutsetter også at alle obliger leveres elektronisk fra student til gruppelærer.

Vi tar utgangspunkt i denne beskrivelsen av brukergrensesnittet i arbeidet med å lage sammenlignings- og lagringsfunksjonene i systemet.

Kapittel 5

Systemspesifikasjon

Vi har nå en god oversikt over problemområdet og hvilken funksjonalitet sammenligningssystemet må tilby. Vi har valgt en lovende algoritme og foreslått en lagringsplattform. Etter å ha diskutert systemets funksjoner fra et brukerperspektiv må vi se nærmere på hvordan funksjonaliteten skal implementeres.

I dette kapittelet vil vi diskutere forskjellige implementasjonsvalg. Disse foreleggene vil utgjøre grunnlaget for implementasjonen og undersøkelsene vi vil beskrive i de neste kapitlene.

5.1 Evalueringsfunksjonen

Man kan snakke om to hovedtyper av algoritmer for tekstanalyse: Nøkkelordbaserte sammenligningsalgoritmer og de som baserer seg på å finne en lengste felles delstreng. Eksempler på disse metodene ble presentert i kapittel 3. Den siste metoden går ut på å sjekke to filer med kode og finne den lengste delstrengen de har felles. I denne sammenhengen kan det være nyttig å i tillegg finne alle felles delstrenger over en viss lengde. Denne metoden vil antagelig kreve for mye tid.

Den andre hovedtypen baserer seg på sammenligning av forekomster av nøkkelord eller nøkkelfraser. Den enkleste sammenligningen man kan gjøre er å telle antall `{` i hver kildekodefil og anta at et likt antall er suspekt. Andre nøkkelord kan være for eksempel `class`, `for(`, `[`, `public`, `void` et c.

Fordelen ved en slik sammenligning av nøkkelord er at maskinen kun behøver å lese gjennom én kildekodefil (den innleverte obligen som sjekkes) én gang. Antallet forekomster av aktuelle nøkkelord og -fraser vil da bli lagt

i minnet og kan bli skrevet til fil eller til en database for referanse ved senere sammenligning. På denne måten vil hver innleverte oblig ha en samling av aktuelle nøkkelorddata på en fil eller i en database, disse kan maskinen lese rett inn i minnet når den startes opp, og derved redusere tiden en gitt sammenligning tar.

Hva slags statistiske metoder som bør tas i bruk for å velge ut signifikante nøkkelord og -fraser må utredes nøye. I denne forbindelse vil det være interessant å se på om det bør legges større vekt på forekomsten av visse nøkkelord enn på andre.

5.1.1 Potensialet for å narre systemet

Et problem dukker opp hvis studentene finner ut hvordan sammenligningsprogrammet virker, noe de garantert kommer til å gjøre. De vil da finne måter å omgå sammenligningsalgoritmene. Den letteste måten å narre nøkkelordmetoden på er ved å legge inn dummy-metoder i koden. En mulig måte å finne slike dummies på er å undersøke om de blir referert andre steder i koden. Blir de ikke det er de sannsynligvis dummies.

Slike kvalitetssikringsmetoder kan selvfølgelig også omgås om man kjenner til dem, og dermed kreves nye sikringsmetoder. Problemet oppstår når sikringsmetodene kompliserer maskinen i en slik grad at tidsforbruket øker. I verste fall kan man risikere at "one pass"-kravet må oppgis, noe som ikke er akseptabelt. På den annen side vil man, med strenge nok sikringsmetoder, tvinge studenten til å skrive om mye av koden hvis han insisterer på å kopiere kode som er registrert av maskinen.

Etterhvert kommer man til en grense hvor studenten, for å sikre at hans kopieringsforsøk ikke blir avslørt, er nødt til å besitte den grad av programmeringsevner og kunnskap som oppgaven krever. I møtet med dette grenseområdet endrer problematikken seg til å dreie seg om hindring av fusking på prinsipiell basis og pedagogisk politikk, noe som faller utenfor denne oppgavens interessefelt.

5.1.2 Sammenligningskriterier for obliger av forskjellig lengde

Ved sammenligning av programkode må det også tas hensyn til kildekodens lengde. Den første oppgaven i begynnerkurset kan vanligvis løses på ti linjer eller mindre. Det er åpenbart at svært mange besvarelser av slike enkle oppgaver vil være mer eller mindre like, rett og slett fordi det er så få måter å løse dem på. Obliger av en slik lengde vil av den grunn ofte være uinteressante å analysere.

I programmer på hundre linjer eller mer vil dette problemet stort sett ikke oppstå siden det vil være mulig å velge forskjellige datastrukturer for et formål i programmer av denne størrelsen. Selv om det er mulig å finne strukturelle forskjeller i ganske korte programmer vil hovedstrukturene fortsatt være de samme. Det vil neppe være hensiktsmessig å sammenligne på grunnlag av antall klasser før programmene kommer opp i 1000 linjer eller mer, siden klassestrukturen ofte er gitt eksplisitt eller implisitt i oppgaveteksten.

5.1.3 Ukorrekt klassifisering av like og ulike obliger

På grunn av at systemet kun indikerer tilfeller av potensielle kopier vil originale arbeider noen ganger klassifiseres som mulige kopier. Samtidig kan det skje at faktiske kopier unngår mistanke på grunn av mangler i sammenligningsalgoritmen. Det er viktig at antallet slike tilfeller reduseres til et minimum, noe som krever utstrakt testing av sammenligningskriteriene. Dette vil vi komme tilbake til i kapittel 7.

5.2 Oversikt over sammenligningsalgoritmens funksjonsmåte

Vi har valgt å benytte symboltelling[1] som grunnlag for sammenligningsalgoritmen, så vi skal se mer inngående på denne metoden.

5.2.1 Symboltelling

En av de enkleste metodene for likhetsanalyse består i å telle forekomster av forskjellige symboler i en programkodefil. Disse tallene beskriver koordinater i et N-dimensjonalt rom og vektoren fra origo til punktet definert av symboltallene benevner vi “symbolvektoren”.

Symbolene må defineres slik at de på en best mulig måte representerer strukturelle egenskaper i koden. I javakode vil det være naturlig å registrere antall venstre krøllparanteser, `{`, samt nøkkelord som `while`, `for`, `public`, `static`, `void` et c. Hvor mange komponentvektorer som skal genereres må bestemmes ved å prøve ulike kombinasjoner av symboler. Om man registrerer for få symboler vil man oftere utsettes for tilfeldige likheter, men om man registrerer for mange vil man kunne oppleve at små

[1] Symbol: En del av en tekst, et tegn, en tegnkombinasjon, et “token” eller “item”.

forskjeller i hver av komponentvektorene til sammen resulterer i en uforholdsmessig stor likhetsverdi i forhold til den faktiske likheten.

Når vi har talt opp alle symbolene i en fil, vil filen lagres i en database sammen med informasjon om symbolvektoren. Neste steg vil være å sammenligne symbolvektoren med vektorene til alle de andre filene i databasen. Vi går ut fra at det mest tidkrevende aspektet er spørringene mot databasen, men ellers skal denne metoden være svært effektiv, se avsnitt 3.4.

5.2.2 Måling av euklidsk avstand

Den ene måten å sammenligne filene på er å finne den euklidske avstanden mellom filene P og Q, se figur 5.1. Vi bruker N symbolvektorer som gir oss verdiene $d_{p_1}, d_{p_2}, \ldots, d_{p_n}$ og $d_{q_1}, d_{q_2}, \ldots, d_{q_n}$. Da kan vi beskrive sammenligningen som følger:

$$D_{pq} = \sqrt{\frac{\sum_{i=1}^{n}(d_{p_i} - d_{q_i})^2}{n}} \tag{5.1}$$

Resultatet er den euklidske avstanden, D_{pq}, som indikerer hvor like de to filene er, lave tall indikerer større likhet, mens høye tall indikerer signifikante forskjeller.

Denne metoden forutsetter at kopien har opphav i én original besvarelse og at det ikke blir lagt til eller fjernet alt for mye fra koden. For å påvise likhet mellom enkeltelementer (f. eks. metoder) i to programmer vil det antagelig være nødvendig å finne største felles delstreng. Men i besvarelser av oppgaver av denne typen kan man nesten ikke unngå at lignende elementer går igjen i forskjellige besvarelser. Dette er en svakhet vi vil akseptere i og med at systemets hensikt er å påvise tilfeller av mindre sofistikert kopiering.

Som følge av forutsetningen gjort ovenfor forstår vi at en sammenligning av filer av svært forskjellig størrelse vil gi en relativt høy euklidsk avstand, D_{pq}, og dermed indikere liten sannsynlighet for kopiering. Dermed kan vi begrense hvilke filer som sammenlignes. Det har for eksempel ingen hensikt å sammenligne en fil med en annen som er dobbelt så stor siden den store filen vil ha omtrent dobbelt så mange forekomster av hvert symbol som den mindre. $d_{p_i} \approx 2d_{q_i}$. Dette vil gi omtrent det samme resultatet som å sammenligne den mindre filen med en tom fil siden $d_{p_i} - d_{q_i} \approx d_{q_i} - 0$ for en gitt i.

Hvor stor størrelsesforskjell to filer kan ha før det ikke lenger er hensiktsmessig å sammenligne dem må undersøkes videre. Vi kan i utgangspunktet anta at en differanse i netto filstørrelse[2] på over 30% vil garantere ulikhet,

[2] Størrelsen på filen etter at alle kommentarer og blanke tegn er fjernet.

men vi kan sannsynligvis begrense dette ytterligere uten å ofre nøyaktighet.

Normalisering av likhetsverdien

Vi støter på et problem i tolkningen av likhetsverdien. Terskelen for hvilke verdier som indikerer mistenkelig likhet varierer nemlig avhengig av størrelsen på programmene som sammenlignes. Resultatet av sammenligning av to store, nesten like programmer kan være en likhetsverdi som er relativt høy sammenlignet med resultatet for to små programmer. Det blir dermed umulig å tolke verdiene om de er det eneste vi har å gå ut fra. Om vi setter en terskelverdi for euklidsk avstand uten å tenke over størrelsen på programmene som analyseres, og verdier under og over denne terskelen er hhv. mistenkelige og ikke mistenkelige, vil verdien for små programmer nesten garantert tolkes som mistenkelig, mens verdien for store programmer nesten aldri vil tolkes som mistenkelig. Dette er uholdbart.

Siden terskelverdien avhenger delvis, om ikke totalt, av de sammenlignede programmenes størrelse kan ikke den euklidske avstanden alene indikere mistenkelig likhet. Vi trenger et mål med stabil tallstørrelse slik at terskelverdien også er stabil. For å få til dette må likhetsverdiene normaliseres. Den normaliserte verdien vil alene kunne indikere mistenkelig likhet uavhengig av de sammenlignede filenes karakteristika, ved å sammenlignes med terskelverdien.

En enkel metode for normalisering er å dividere likhetsverdien på filstørrelsen til den minste av de to filene som sammenlignes. Hvorvidt denne metoden gir gode resultater finner vi ved å undersøke et testutvalg av filer.

En annen mulig metode består i å normalisere dimensjonstallene før man regner ut euklidsk avstand. Dette kan gjøres ved å regne ut forholdstall for hvert symbol. Dette forholdstallet vil kunne defineres ved antall forekomster av et symbol (S_i) dividert på totalt antall forekomster av alle symboler (S), eller S_i/S. Denne løsningen vil neppe gi en fornuftig likhetsverdi for to sammenlignede filer, men den kan kanskje benyttes til å lage en profil for programmets programmeringsstil. En slik profil kan for eksempel brukes til å finne ut om lignende programmeringsstiler er benyttet i forskjellige programmer. Om en av obligene levert av en student har en programmeringsstil som avviker fra de andre kan dette være en indikasjon på kopiering.

Før vi bestemmer oss for en normaliseringsmetode bør vi finne ut hvordan likhetsverdiene for store og små programmer forholder seg til hverandre. På bakgrunn av dette kan vi finne den normaliseringsmetoden som er mest hensiktsmessig.

5.2.3 Vinkelanalyse av dimensjonstallene

Som et alternativ til den euklidske avstanden kan vi regne ut vinkelen mellom symbolvektorene. Jo mindre vinkelen er dess større sannsynlighet er det for at obligene er like. Siden sannsynligheten er liten for at to obliger av vidt forskjellig størrelse skal ha samme opphav, bør en vurdering av vektorvinkelen ta utgangspunkt i at obligene er av sammenlignbar størrelse.

Når vi vurderer den euklidske avstanden bruker vi kun endepunktene til vektorene, men vektorene eksisterer i et N-dimensjonalt rom, med felles startpunkt i origo og dermed er det mulig å bestemme vinkelen mellom dem. Cosinus til denne vinkelen α beregnes ut fra følgende formel hvor N er antall dimensjoner (dvs. antall symboler), P og Q er de to vektorene som skal sammenlignes.

$$cos\ \alpha = \frac{\sum_{i=1}^{N} P_i \cdot Q_i}{\sqrt{\sum_{i=1}^{N} P_i^2} \cdot \sqrt{\sum_{i=1}^{N} Q_i^2}} \tag{5.2}$$

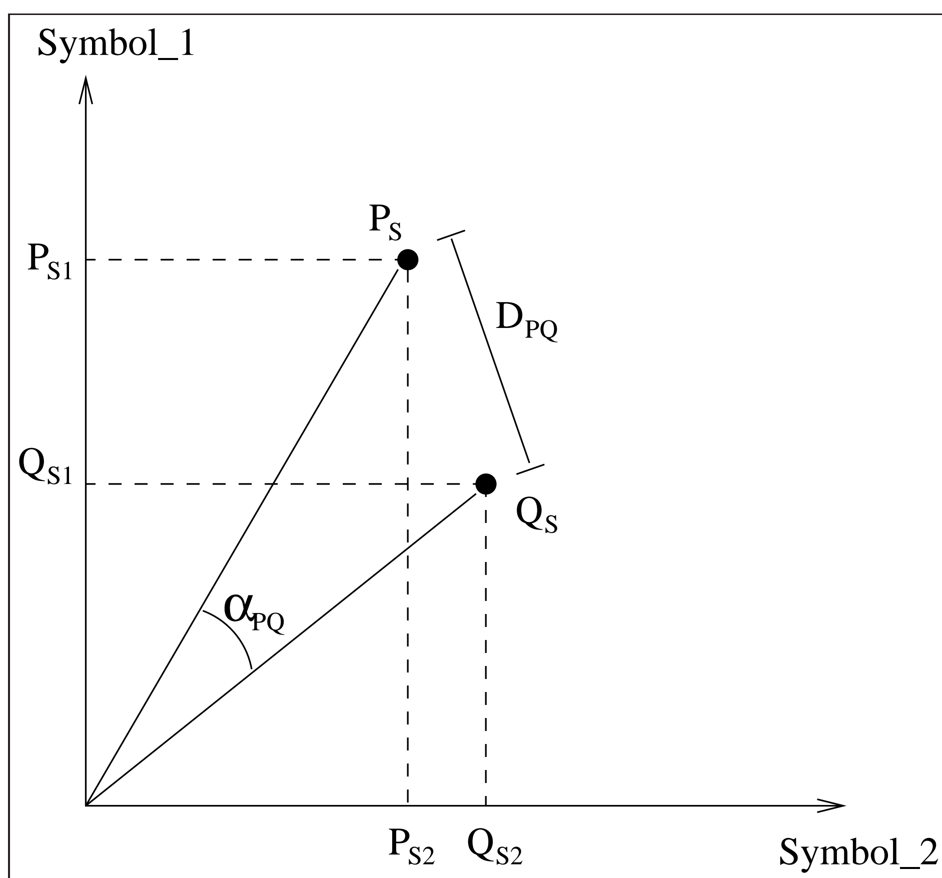
Figur 5.1 viser hvordan sammenligningen av symbolvektorene for to programmer, P og Q, kan visualiseres for $n = 2$. D er den euklidske avstanden og α er vinkelen mellom vektorene. Flere symboler (dimensjoner) ville gjort figuren mer interessant, men desto mer uoversiktlig.

Ved å se på vektorene for de to filene under sammenligning vil vi legge merke til om det er en stor vinkel mellom dem. Hvis dette er tilfellet, men den euklidske avstanden likevel er relativt lavt kan det være en indikasjon på at vi har å gjøre med et falskt positivt resultat.

I tilfeller hvor vektorvinkelen er liten, men den euklidske avstanden er høy tyder det på at programmeringsstilen for filene er lik, mens programstørrelsen er forskjellig. Forholdet mellom antall forekomster av hvert symbol i de to filene vil med andre ord være relativt likt om vinkelen er liten.

I en eventuell utvidelse av systemet kan man bruke vektorvinkelen til å finne ut om samme programmeringsstil er brukt i forskjellige programmer. Om for eksempel programmeringsstilen i en oblig varierer sterkt fra de andre innleveringene til en gitt student vil det være naturlig å undersøke om studenten kan ha fusket under arbeidet med den aktuelle obligen.

Hvilket kriterium som gir de mest nøyaktige resultatene, euklidsk avstand, vektorvinkel eller en kombinasjon av begge, vil de videre undersøkelsene vise.



Figur 5.1: Vektorene for to programmer P og Q

Effektivitet ved utregning av vektorvinkel

Tiden det tar for maskinen å regne ut vektorvinkelen er ikke merkbart forskjellig fra tiden det tar å regne ut den euklidske avstanden. Skulle vinkelberegningen allikevel by på problemer kan de to kvadratrotoperasjonene utelates om man i stedet kvadrerer telleren, dette kan spare litt regnekraft. I så fall har man regnet ut $cos^2\alpha$ som like gjerne kan brukes, selv om man mister den visuelle/intuitive tilknytningen til vinkelen som er nyttig i forbindelse med å bestemme terskelverdier. Har man allerede bestemt terskelverdier, må disse omjusteres for å ta hensyn til endringen, men eller er dette sikkert en grei løsning.

I denne sammenhengen har vi valgt å beholde disse operasjonene for illustrasjonens skyld. Velger man å bruke $cos^2\alpha$ kan man samtidig utelate arcus cosinus-operasjonen som brukes i vår implementasjon.

5.2.4 En mulig implementasjonsbeskrivelse

For å kunne vurdere den reelle størrelsen på en oblig, er det nødvendig å fjerne alle kommentarer og faste tekststrenger fra koden. Dette gjøres greit ved hjelp av en koderenser skrevet i Perl. Dette skriptet kan også brukes til å telle opp forekomstene av symboler i kildekoden. I og med at denne tellingen kun behøver å gjøres første gangen en kildefil sjekkes er det kun filene som blir sjekket mot databasen som vil bli gjennomgått. Hvis vi antar at en gruppelærer ikke har mer enn 30 studenter som leverer obliger, vil han sjelden sjekke mer enn 30 filer om gangen. Vi antar dermed at akseptabelt tidsforbruk kan regnes ut fra 30 javafiler av en gitt størrelse.

I kapittel 4 foreslo vi å benytte databasen MySQL som lagringsmedium for innleverte obliger. MySQL er enkel å bruke, den er billig og kan håndtere store datamengder effektivt. I tillegg tilbyr både Perl, Java og PHP lett tilgjengelige grensesnitt mot MySQL. Det virker derfor som om MySQL er den mest nærliggende løsningen for lagringsmedium.

Hver av kodefilene må etter rensing bli lagret i en database sammen med den tilhørende informasjonen om antall symbolforekomster. Deretter må hver kildefil sjekkes mot alle de andre som er lagret i databasen i henhold til metoden illustrert ved ligning 5.1. For hver kildefil vi mater til systemet finner vi at databasen må gjøre én `INSERT`-operasjon og en moderat komplisert utregning for hver forekomst i databasen. Ved eventuelle treff vil databasen i tillegg måtte utføre et antall `SELECT`-operasjoner. Hvor mye tid disse operasjonene vil ta avhenger av databasen vi velger å bruke.

5.3 Sammenligningsmetoder for frie tekster

I tillegg til sammenligning av programkode kan det være interessant å vurdere utvidelse av systemet til å håndtere sammenligning av frie tekster. Vi nevnte YAP3-systemet i kapittel 3, et eksempel på en metode som også håndterer prosatekster. Opptelling av en håndfull nøkkelord kan nok gi tilfredsstillende resultater for programtekster, men vil neppe fungere for sammenligning av prosa på grunn av vanskeligheten med å finne et godt utvalg av nøkkelord.

Inntil videre anser vi denne problemstillingen for å falle utenfor denne oppgavens fokus, selv om det kan være interessant i forbindelse med en eventuell videreføring av prosjektet.

5.4 Oppsummering

I dette kapittelet har vi gitt en oversikt over implementasjonsavgjørelsene og vi har beskrevet to måter for analyse av symbolvektorer.

Vi har beskrevet algoritmen vi vil bruke og den kan i korthet oppsummeres som følger:

- Velg to programmer for sammenligning.
- Vurder størrelsesforskjellen, om programmene har svært forskjellig størrelse går vi ut fra at de er forskjellige og sammenligningen utføres ikke.
- Finn antallet av hvert symbol for hvert program.
- Regn ut vinkelen mellom de N-dimensjonale vektorene.
- Hvis vinkelen er mindre enn terskelverdien, gi en mistankerapport.

Kapittel 6

Implementasjon av en prototype

I dette kapittelet beskriver vi implementasjonen av databasen og sammenligningssystemet slik de foreligger. Denne implementasjonen utgjør en prototype som vil brukes i de eksperimentene som beskrives i de neste kapitlene. Prototypen har all den funksjonaliteten som er nødvendig for sammenligning og lagring av obliginnleveringer.

6.1 Databasen

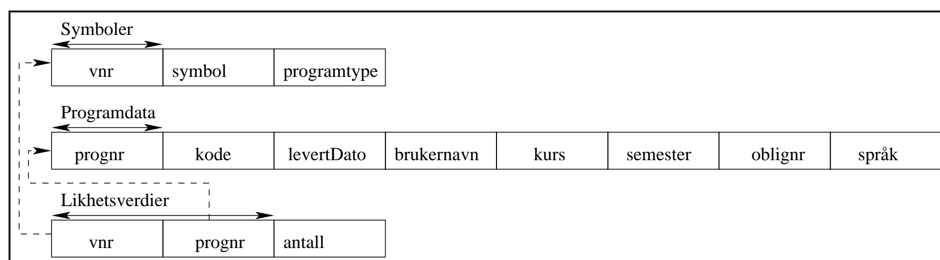
Som nevnt i tidligere avsnitt vil vi benytte MySQL til å lagre innleverte obliger, samt informasjon om likhetsverdier og symbolantall. Databasen er en relativt ukomplisert konstruksjon og krever ikke kraftigere teknologi enn det MySQL kan tilby, i tillegg er MySQL gratis, noe som er et tungt argument for ethvert ikke-kommersielt system.

Databasen består av tre tabeller, Programdata, Vektorer og Likhetsverdier. Disse tabellene inneholder alle nødvendige data for å foreta analyser og sammenligninger av de forskjellige programmene.

Lagring av brukerdata krever en utvidelse av databasen som vi vil komme tilbake til senere i oppgaven.

6.1.1 Tabellen for programdata

Kjernen i databasen er tabellen som inneholder programdata. Hver innlevering vil bli identifisert med et løpenummer når den registreres i databas-



Figur 6.1: Definisjon av forsøksdatabasen

en og den opprinnelige koden vil bli lagret i sin helhet som et `MEDIUMTEXT`-element. Selv om en obliginnlevering kan defineres entydig ved forfatterens epostadresse, kurskode og oblignummer, vil det være mer praktisk å la primærnøkkelen til en kildefil i databasen bestå av et løpenummer, så vi kan lagre flere innleveringer av samme oppgave fra samme student.

I tillegg har vi behov for ytterligere informasjon, blant annet nummeret på obligen, dato og tid registreringen skjedde, hvilket kurs innleveringen er del av, brukernavnet til forfatteren og hvilket semester kurset gikk. For at sammenligningen skal bli så pålitelig som mulig må vi også registrere antall kodelinjer. Antall kodelinjer brukes som grunnlag for å angi relativ størrelse på programmet. Det kreves antagelig en del testing for å finne optimale terskelverdier, men som et utgangspunkt kan vi foreslå at alt under 100 linjer ansees som et lite program, 100-1500 linjer regnes som et mellomstort program og alt over 1500 linjer regnes som store programmer. Med tanke på videre utvikling av systemet kan det også være interessant å registrere hvilket programmeringsspråk koden er skrevet i. Figur 6.1 viser en modell av databasen.

6.1.2 Tabellen for vektorer

På grunn av at forskjellige typer programmer har forskjellige karakteriske kvaliteter er det naturlig å bruke forskjellige kriterier i sammenligningen av f. eks. store vis-à-vis små programmer. Mange nøkkelord vil variere med programmeringsspråket, mens strukturelle nøkkelord som f. eks. `class`, `public`, et c. vil være relativt betydningsløse i små programmer i forhold til i store. Av denne grunn vil det være interessant å definere flere sett av kriterier for sammenligning av programmer av forskjellig lengde eller type.

Hvert nøkkelord kalles for et *symbol* og består av en tekststreng med lengde større enn null, uten blanke tegn. Informasjon om hvilke(n) programtype(r) dette kriteriet skal brukes på registreres i et `SET`-objekt.

Denne tabellen kommer etter hvert til å være relativt statisk med et antall på forhånd definerte symboler, men det er selvfølgelig mulig at det vil bli nødvendig å definere flere symboler etter at systemet er satt i drift.

6.1.3 Tabellen for likhetsverdier

Til slutt har vi en tabell hvor likhetsverdiene for de forskjellige kodefilene er lagret. Vi antar at det fra tid til annen kan være nødvendig å analysere en kodefil med utgangspunkt i symbolvektorer basert på flere enn ett symbolsett. En årsak kan bl. a. være om programmet ligger på grensen mellom stort og mellomstort. Systemet regner derfor ut likhetsverdier for hvert symbol, uavhengig av størrelsen på programfilen.

Hver likhetsverdi identifiseres ved hjelp av selve symbolstrengen og programnummeret for hvert symbol i tabellen. Likhetsverdien reflekterer kun resultatet av gjennomlesingen av koden og eventuell likhet med andre programmer i databasen avsløres ikke før de relevante likhetsverdiene til to programmer blir sammenlignet med hverandre.

6.2 Sammenligningsprogrammet

Tidligere antydet vi at Perl kunne egne seg til å rense kodefilene, men vi endte opp med å skrive koden i PHP. Grunnet blant annet at systemet vil ta imot kodefiler for lagring og analyse via WWW vil det være naturlig å gjøre mesteparten av databehandlingen på web-tjeneren for å unngå tilleggskostnadene ved å kalle eksterne programmer (dette er en av ulempene med CGI). En analyse av fordeler og ulemper ved CGI kan finnes i [Zha99]. PHP-tolkeren har den styrken at den er kompilert inn i Apache-tjeneren og vil dermed levere HTML-kode tilbake til web-klienten raskere enn et Perl/CGI-skript ville gjort. PHP har i tillegg et enklere og mer intuitivt grensesnitt mot MySQL.

6.2.1 Brukergrensesnittet

Systemet består, i tillegg til MySQL-databasen, av et antall PHP-filer. Brukeren får altså tilgang til systemet gjennom en nettleser. Etter å ha logget inn blir brukeren, i dette tilfellet gruppelæreren, først bedt om å skrive inn stien til filmappen hvor de nylig innleverte javafilene ligger. Han vil så presenteres for en side med en liste over alle filnavn i denne filmappen som ender på `.java` og har mulighet til å velge hvilke filnavn han vil lagre i databasen.

Etter å ha valgt filnavn behøver brukeren kun vente mens hver fil analyseres. Hver javafil som er gjenstand for analyse listes opp i nettleservinduet og alle filer som ligner på filen under analyse listes opp med tilhørende likhetsverdi, se figur 6.2.

Hvert filnavn som listes opp er en hyperlenke som, ved å trykkes på, åpner et nytt nettleservindu med programkoden filen inneholder. På denne måten er det mulig for brukeren å åpne den nylig innleverte koden i ett vindu og åpne et annet vindu med den lignende koden. Slik kan de to filene lett sammenlignes. Om nødvendig kan andre vinduer også åpnes med kode fra andre lignende filer. Hvor mange nettleservinduer som er åpne til en hver tid er selvfølgelig opp til brukeren.

På grunn av at prototypen foreløpig er ment for eksperimentell bruk tilbyr brukergrensesnittet mangelfull sikkerhet og implementerer heller ikke tilbakemelding til andre enn primærbrukeren. Ellers er presentasjonen av sammenligningsinformasjon adekvat for eksperimentell bruk og utgjør et godt utgangspunkt for det endelige brukergrensesnittet.

Notifikasjon til flere gruppelærere

På grunn av at en fil kan vise seg å ha likheter med en senere lagret fil bør vi implementere en epostfunksjon som automatisk sender informasjon om dette til den aktuelle gruppelæreren. Epost blir kun sendt dersom den allerede lagrede filen er registrert levert samme semester. Implementasjonen bør sikre at samme gruppelærer ikke får flere eposter om forskjellige like filer, eller forskjellige eposter om samme suspekte fil. Ideelt sett skulle informasjon om alle suspekte filer vært samlet i én epost.

Det blir gjort et relativt stort antall SQL-spørringer når en gruppe med filer sendes for lagring og analyse, totalt $M(AVG(N)+2)+1$ spørringer. Sett at vi har et stort antall filer som skal undersøkes ($M = 30$) og et gjennomsnitt på 15 symboler ($AVG(N) = 15$) vil vi totalt foreta 511 spørringer. 30 (M) av disse er SELECT-spørringer som vil returnere svært mange forekomster.

Vi har valgt å gjøre alle utregningene i PHP-programmet og bruker databasen kun som kilde for relevante data. Det er mulig at vi ved å lage utregningsprosedyrer i databasen hadde kunnet redusere antall spørringer i programmet, men dette hadde blitt komplisert da MySQL har begrenset støtte for lagrede funksjoner og prosedyrer.

Et annet viktig problem er hvilke symboler sammenligningen skal ta utgangspunkt i. Dette vil vi diskutere i kapittel 7.

Filer registrert - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

file:///ifi/fe Go

Filer registrert

Resultat av filregistrering

Vent noen minutter mens filene blir analysert...

Fil #2: /hom/christkk/ttt/O2-02-0.java - 3364 bytes
Kode for filen /hom/christkk/ttt/O2-02-0.java - 1537 bytes parsed

O2-02-0.java **Vektorvinkel: 0.00°** Endre status til "ikke juks"

Fil #3: /hom/christkk/ttt/O2-03-0.java - 2691 bytes
Kode for filen /hom/christkk/ttt/O2-03-0.java - 1433 bytes parsed

O2-03-0.java **Vektorvinkel: 0.00°** Endre status til "ikke juks"

Fil #4: /hom/christkk/ttt/O2-04-0.java - 1823 bytes
Kode for filen /hom/christkk/ttt/O2-04-0.java - 768 bytes parsed

O2-04-0.java **Vektorvinkel: 0.00°** Endre status til "ikke juks"
O2-04-a.java **Vektorvinkel: 1.07°** Endre status til "ikke juks"

Fil #5: /hom/christkk/ttt/O2-04-a.java - 1538 bytes
Kode for filen /hom/christkk/ttt/O2-04-a.java - 762 bytes parsed

O2-04-0.java **Vektorvinkel: 1.07°** Endre status til "ikke juks"
O2-04-a.java **Vektorvinkel: 0.00°** Endre status til "ikke juks"
O2-04-0.java **Vektorvinkel: 1.07°** Endre status til "ikke juks"

Fil #8: /hom/christkk/ttt/O3-03-0.java - 58460 bytes
Kode for filen /hom/christkk/ttt/O3-03-0.java - 25355 bytes parsed

Fil #9: /hom/christkk/ttt/O3-04-0.java - 32016 bytes
Kode for filen /hom/christkk/ttt/O3-04-0.java - 14427 bytes parsed

Fil #10: /hom/christkk/ttt/O4-01-0.java - 24619 bytes
Kode for filen /hom/christkk/ttt/O4-01-0.java - 12096 bytes parsed

Fil #11: /hom/christkk/ttt/O4-01-a.java - 17450 bytes
Kode for filen /hom/christkk/ttt/O4-01-a.java - 12185 bytes parsed

O4-01-0.java **Vektorvinkel: 0.00°** Endre status til "ikke juks"

Done

Figur 6.2: Presentasjon av mistankerapporter.

6.2.2 Sammenligningsmekanismen

Etter at brukeren har angitt hvilke filer som skal undersøkes hentes alle relevante symboler ut av databasen og relevant innleveringsinformasjon (se avsnitt 6.1.1) blir hentet ut fra hodet til den første valgte kodefilen. Kodefilen strippes deretter for kommentarer samtidig som alle forekomster av de forskjellige symbolene telles opp. Kodefilen lagres i databasen, med kommentarer intakt og med relevant innleveringsinformasjon og hvert av symbolene med tilhørende verdier for denne filen blir likeledes lagret.

Etter at filen er lagret hentes symbolantallene for alle lignende, tidligere lagrede kodefiler ut av databasen for sammenligning. Lignende kodefiler er definert som filer programmert i samme språk som ikke er mindre enn halvparten så korte eller mer enn dobbelt så lange enn kodefilen under analyse. En likhetsverdi blir funnet på bakgrunn av ligning nr. 5.1 på side 38 og denne blir skrevet ut i web-leseren om verdien antyder uforholdsmessig stor likhet mellom de to filene.

6.3 Diskusjon av implementasjonsvalg

Vi har for det meste fulgt de premissene vi la til grunn for implementasjonen i tidligere kapitler, men noen av disse har vi måttet endre ettersom nye behov har kommet for dagen. Andre forutsetninger har vi vært nødt til å spesifisere nærmere.

6.3.1 Formatering av innleverte obliger

Vi forutsetter at all innlevert kode inkluderer en kommentarlinje med innleveringsinformasjon som første linje i programmet. Denne må se ut som følger: `// brukernavn g-m o-n k-kurskode/semester`.
Her skal `brukernavn` erstattes med studentens brukernavn, `m` skal erstattes med gruppenummer og `n` med oblignummer. Dette formatet gjør det enkelt å finne og lagre denne informasjonen i databasen. Hvis denne linjen enten ikke finnes, ikke er første linje eller er galt formattert vil programmet skrive ut en feilmelding for filen og ikke gjøre noe med den. Deretter fortsetter programmet med å behandle neste fil.

6.3.2 Tilgang til filer fra systemet

Løsningen som er foreslått implementerer en egen struktur for filvelging som forutsetter at alle filer som skal analyseres ligger i samme filsystem

som vevtjeneren, det er flere grunner til dette. For det første fant vi at funksjonen for filvelging i HTML-skjemaer var for snever. De aller fleste vevlesere tillater kun å velge én fil fra den lokale disken, mens vi hadde behov for å velge flere. Dette kunne løses ved å laste opp én fil om gangen, men vi anså dette for å være uholdbart tungvint. Vi kunne ha inkludert et større antall knapper for filvalg i samme skjema, men dette ville blitt uoversiktlig og svært lite brukervennlig. Løsningen vi foreslår er svært mye mer brukervennlig enn andre alternativer og vi anser ikke kravet om at filene må ligge på Ifis filsystem for å være noe reelt problem i og med at alle gruppelærerene har brukerkontoer på Ifi.

6.4 Oppsummering

Vi har i dette kapittelet beskrevet implementasjonen av en fungerende prototype av sammenligningssystemet. Prototypen implementerer den valgte algoritmen og kan brukes i de neste kapitlenes tester av nøyaktighet og optimale symbolutvalg.

Kapittel 7

Undersøkelse av forskjellige symbolkombinasjoner

I dette kapittelet beskriver vi symbolutvelgelsesprosessen og de forskjellige testene vi utfører for å komme frem til et tilnærmet optimalt symbolsett. Vi foreslår en testprosedyre og følger steg 1-3 i denne prosedyren. Dette vil gi oss et godt utgangspunkt for fastsetting av passende terskelverdier i neste kapittel.

De første undersøkelsene utførte vi for å finne den mest nøyaktige sammenligningsmetoden, vurdering av enten euklidsk avstand eller vektorvinkelen. Deretter gjorde vi et utvalg av lovende symboler til utgangspunktsettet. Utgangspunktsettet ble deretter redusert til et best mulig utvalg ved at vi utførte forsøk hvor vi optimaliserte symbolsettet etter "steepest hill"-algoritmen som beskrives i avsnitt 7.5.1.

7.1 Innledende kommentarer

Som tidligere nevnt er det ikke åpenbart hvilke kombinasjoner av symboler som vil gi de mest nøyaktige klassifikasjonsresultatene. I dette kapittelet vil vi beskrive resultatene av forsøkene vi har gjort med forskjellige kombinasjoner på en samling av obliger.

7.1.1 Prosedyre for valg av symbolsett

For å komme frem til et brukbart symbolsett utfører vi testing i flere steg. Disse testene vil bli nærmere beskrevet i dette og de påfølgende kapitlene.

1. Først foreslår vi forskjellige mengder av mulige symboler.

2. Så konstruerer vi et testsett av programmer hvor noen er mer eller mindre varierte kopier. Vi kjører programmene gjennom sammenligningsalgoritmen med de forskjellige symbolsettene. Hensikten er å finne ut om metoden i det hele tatt virker fruktbar, og i så fall finne symbolsettet som gir de beste resultatene.

3. Vi tar utgangspunkt i symbolsettet vi fant i forrige steg og forsøker å redusere det til vi får optimale resultater med færrest mulig symboler. Da vi ikke har ressurser til å gjøre fullstendige tester av alle mulige symbolkombinasjoner bruker vi en *steepest-hill*-metode hvor vi fra mengden av N symboler finner det settet av $N-1$ symboler som gir det beste resultatet. Vi gjentar reduksjonen av symbolsettet helt til vi ikke lenger ser noen forbedring av ytterligere reduksjoner.

4. Når vi nå har funnet et brukbart symbolsett, må vi fastsette terskelverdiene for hva som skal tolkes som mistenkelig. Verdiene som må avgrenses er størrelsesforskjellen mellom programmene og vektorvinkelen. Vi bruker et stort utvalg av genuine innleverte obliger og registrerer resultatene fra sammenligningssystemet med forskjellige terskelverdier. Vi forventer å finne flere mistenkelige tilfeller blant kandidater med dårlig eksamenskarakter. På bakgrunn av disse undersøkelsene fastsetter vi terskelverdiene.

5. Siste steg er verifikasjon av resultatene. Vi går gjennom et representativt utvalg av de genererte mistankerapportene og registrerer antall falske og sanne positive og negative rapporter. Resultatene vi får i dette steget vil fortelle oss hvor eksakt metoden er når det gjelder å rapportere mistenkelige obliger.

7.2 Redegjørelse for forsøkene

Vi bruker obliger i kursene INF1010 og INF1000 samlet inn i løpet av vår- og høstsemestrene 2004. De fleste obligene er ikke kopier og brukes som kontrollmateriale. Av noen av obligene har vi laget kopier hvor vi har lagt inn større eller mindre endringer av den typen som vanligvis forekommer i obligfusk.

Endringene som gjøres av fuskeren varierer fra å skifte ut de faste strengene i `System.out.print`-setningene, via enkel endring av variabel- og metodenavn, til endring av rekkefølgen av klasser og metoder. Noen av de mer oppfinnsomme fuskerne vil blant annet legge til ekstra variable og blokker

som enten ikke har noen funksjon eller gjør enkle kall på utskriftsmetoder og lignende.

Fjerning av elementer fra originalkoden vil kun gjøres av de mest durkdrevne fuskerne siden de fleste fuskerne vil frykte at dette vil hindre koden i å kompilere og kjøre riktig.

7.2.1 Forsøksmaterialet

Obligene som brukes er besvarelser på oppgave nr. 2 (Dronningoppgaven), 3 (en GUI-oppgave) og 4 (et spamfilter) fra INF1010 våren 2004.

Filnavnene er på formen `Om-nn-x.java` hvor `m` er et nummer som angir hvilken oppgave det er snakk om, `nn` er et løpenummer som angir forskjellige besvarelser på oppgaven og `x` er enten 0 om besvarelsen er original, eller en bokstav om det er en kopi. `Om-nn` er forskjellig for hver originale javafil. Kopier har samme `Om-nn`-verdi som originalen, men forskjellig `x`. Navnet på en original kan dermed se ut som følger `O2-01-0.java`, mens kopier av denne vil ha navnene `O2-01-a.java`, `O2-01-b.java`, et c. En annen original besvarelse av samme oppgave vil ha navnet `O2-02-0.java`.

7.2.2 Hvordan å velge symboler

Der endringene i kopien kun gjelder faste strenger og tegn, variabel- og metodenavn eller forandring av rekkefølgen på setninger og blokker, vil systemet klassifisere kopien og originalen som helt like, dvs. at både vektorvinkelen og den euklidske avstanden er 0. Dette på grunn av at egendefinerte navn i kildekoden ikke tas hensyn til i sammenligningsprosessen. Der fuskeren legger elementer til den opprinnelige koden vil resultatet bli mindre nøyaktig siden algoritmen som benyttes baserer seg på telling av tekstsymboler som kan forekomme i disse nye elementene.

Vi kan anta at ekstra setninger ofte blir lagt til mens ekstra blokker sjeldnere legges til. Derfor vil det være lurt å telle symboler som

- `{`
- `public`
- `static`
- `void`
- `if(`

- `else`
- `while(`
- `for(`

siden disse hovedsaklig opptrer som prefikser til blokker. De vil dermed kunne si oss noe om kodens struktur, jfr. [CM93].

Symboler som

- `.`
- `;`
- `"`
- `(`

og lignende kan derimot ofte opptre i setninger, og vil derfor ikke egne seg like godt som analysegrunnlag.

7.2.3 Hvordan frekvensen av forskjellige symboler kan beskrive programmets struktur

De to settene av symboler i avsnitt 7.2.2 blir henholdsvis kategorisert som strukturbeskrivende og størrelsesbeskrivende. Vi kan si at mens frekvensen av symboler som `;` har sammenheng med antall kodelinjer i et program, kan frekvensen av symbolet { si noe om hvor fint oppdelt programmet er.

På samme måte kan det finnes korrelasjon mellom frekvensen av forskjellige symboler. Hvis forholdet mellom frekvensen av symbolet `else` og symbolet `if(` i samme program som regel er 0,8, og dette forholdet holder for et tilfeldig utvalg av programmer, indikerer dette korrelasjon mellom de to symbolene. Hvis korrelasjonen er sterk, det vil si at forholdet er rimelig konstant, vil det være uinteressant å telle opp forekomstene av begge symbolene siden man kan anslå frekvensen av det ene symbolet ut fra frekvensen av det andre.

For å kunne velge symboler på et fornuftig grunnlag er det nødvendig å diskutere de forskjellige symbolenes konnotasjoner.

- { Som nevnt ovenfor sier krøllparantesen noe om hvor fint oppdelt programmet er. Den brukes i de fleste programmeringsspråk til å indikere begynnelsen på en ny blokk som kan være en klasse, en metode eller en annen gruppering av kodelinjer. Jo høyere frekvens av

krøllparanteser per linje, desto finere oppdelt er programmet. Frekvensen av krøllparanteser kan derfor bidra til å identifisere en programmeringsstil, og vil antagelig være essensiell for å påvise likhet.

- `public` Dette nøkkelordet brukes kun i forbindelse med deklarasjoner av variabler og metoder og kan knyttes til programmeringsstil. Frekvensen av dette symbolet kan være betydningsfullt i analysen av studenters oppgaver.
- `static` Et av Javas nøkkelord som ofte misbrukes av begynnerstudenter. En høy frekvens av dette symbolet indikerer en umoden forståelse av objektorientering. I kopierte programmer vil dette symbolet høyst sannsynlig forekomme bortimot like ofte i hver kopi.
- `void` Dette symbolet brukes i metodedeklarasjoner og gir informasjon om antall metoder som ikke returnerer noen verdi. Symbolet vil ikke nødvendigvis korrelere med de to foregående symbolene, men en programmeringsoppgave vil ofte være så nøye spesifisert at antallet void-metoder kan variere lite fra besvarelse til besvarelse. Vi benytter dette symbolet i analysen inntil videre.
- `if(` Dette symbolet angir en forgrening i programmet. Forutsetningene i programmeringsoppgaven nødvendiggjør et visst antall forgreninger i alle oppgavebesvarelsene. I tillegg legger studentene inn et forskjellig antall if-setninger på bakgrunn av sin særegne programmeringsstil. Dermed kan vi anta at dette symbolet er en god likhetsindikator siden antallet if-setninger i forskjellige besvarelser på samme oppgave kan variere betydelig.
- `else` Dette symbolet er antagelig mindre betydningsfullt enn `if(` siden det nødvendigvis er en sterk korrelasjon mellom dem. Det kan allikevel ha verdi som likhetsindikator da forskjellige programmeringsstiler i varierende grad bruker ekstra if-setninger som erstatning for else-grener.
- `while(` Løkkekonstruksjonene sier noe om programmets struktur. Om man ønsker å kopiere koden til en medstudent vil man ofte velge å gjøre endringer som er mindre komplisert enn å fjerne eller legge til løkker. Vi vil gå ut fra at antall while-setninger er en betydningsfull likhetsindikator.
- `for(` Det som gjelder for `while(` gjelder også for `for(`. Disse to symbolene kan til en viss grad byttes om i et program, men slike endringer vil ofte virke påfallende for en som leser koden.
- = Likhetstegnet forekommer svært ofte i programmer. Det brukes i tilordningssetninger og sammenligninger og korrelerer antagelig med

både `if` og `for`. Vi antar i utgangspunktet at det er interessant å undersøke dette symbolet i analysen.

- `.` Punktumet brukes til navigering i programmets klasse-/objekthierarki. Et høyt antall punktum per kodelinje kan ofte indikere at programmet har en lav grad av objektorientering og mangler en naturlig delegering av funksjoner mellom klassene. Dette symbolet kan dermed antas å ha verdi som likhetsindikator.

- `;` Semikolonet brukes i Java for å avslutte en programsetning, men også for å adskille de tre kriteriene i deklarasjonen av en for-løkke. Vanligvis avsluttes en linje med semikolon, mens hver for-deklarasjon inneholder to semikolon. Hvis vi antar at hver for-deklarasjon står på en linje for seg kan dette symbolet antas å korrelere med summen av antall linjer i programmet og antall for-løkker. På den annen side vil forholdet mellom antall linjer og antall semikolon til en viss grad avhenge av hvor fint oppdelt programmet er, siden første linje av en blokk avsluttes med en krøllparantes og ikke et semikolon. Det er derfor riktigere å si at antallet semikolon korrelerer med differansen mellom antall linjer og antall krøllparanteser.

- `"` Anførselstegnet angir start og slutt for en fast tekststreng og korrelerer dermed 100% med antall faste strenger. Vi kan også anta at det til en viss grad korrelerer med antall utskriftssetninger. Utskriftssetningene er blant de kodeelementene som i størst grad blir manipulert i kopierte oppgaver, det er dermed tvilsomt om dette symbolet har noen verdi som likhetsindikator.

- `(` Paranteser har forskjellige bruksområder, blant annet i metodedeklarasjoner, if-setninger, for- og while-løkker. De kan også brukes i aritmetiske og logiske uttrykk. Siden symbolet kan brukes i mange forskjellige kontekster er risikoen relativt stor for at et likt antall paranteser i to programmer kan skyldes tilfeldigheter heller enn faktisk likhet mellom programmene.

7.3 De innledende forsøkene

I forsøkene i dette avsnittet kjører vi et utvalg av konstruerte obligbesvarelser gjennom sammenligningssystemet. Siden formålet med denne testen er å gjøre en innledende vurdering av forskjellige symbolsett tar vi utgangspunkt i et utvalg av forskjellige obliger og lager tre kopier av hver av to obliger. Kopiene modifiseres i forskjellig grad.

Resultatene i dette avsnittet består av utregninger av ***euklidske avstander*** slik de defineres i ligning 5.1 på side 38.

7.3.1 Fremgangsmåte for forsøkene

Det finnes noen sentrale spørsmålene forsøkene vil kunne gi svar på.

1. Hvordan skal likhetsverdiene normaliseres?
2. Hvor nøyaktig er algoritmen?
3. Hvilke symbolkombinasjoner gir de beste resultatene?
4. Er det mulig å begrense sammenligningene?

For å svare på første spørsmål må vi justere verdiene slik at like resultater indikerer samme grad av likhet uavhengig av filstørrelse. Den resulterende verdien skal indikere likhet og dermed sannsynlighet for kopiering. Et resultat på 0 skal indikere svært høy sannsynlighet for at de sammenlignede filene er identisk like. En mer omfattende diskusjon finnes i avsnitt 7.3.6.

For å svare på de to neste spørsmålene kjører vi et antall originale og kopierte programmer gjennom sammenligneren og registrerer likhetsverdiene for hver kombinasjon av filer og for hvert sett av symboler brukt i sammenligningen. En endelig vurdering gjøres i oppsummeringen i slutten av kapittelet.

Det fjerde spørsmålet krever beviser for at det er unødvendig å sammenligne besvarelser fra to forskjellige, gitte kategorier. Det kan for eksempel være snakk om størrelseskategorier. Dette belyses videre i avsnitt 7.3.7.

7.3.2 Materialet for forberedende forsøk

I følgende tester brukes 13 besvarelser av obligatorisk oppgave nr. 4, hvorav 6 er kopier av forskjellig kompleksitet. `O4-0{1,2}-a.java` har kun endret variabelnavn og rekkefølgen på setninger og blokker. `O4-0{1,2}-b.java` er endret ytterligere ved ekstra setninger, hovedsaklig `System.out.print`-setninger. `O4-0{1,2}-c.java` har i tillegg blitt utvidet med flere overflødige if-setninger, while-løkker og metoder som ikke utføres.

7.3.3 Første forberedende test

Symbolene som i første omgang brukes i sammenligningsprosessen er `{ ; , public while class`. Dette er symbolsett nr. 1 og gir resultat-

ene i tabell 7.1.

Vi legger merke til at filene `O4-01-*.java` gir innbyrdes likhetsverdi under 2,3, mens filene `O4-02-*.java` gir innbyrdes likhetsverdi under 3,4. Siden disse er kopier av hverandre antar vi at 3,4 kvalifiserer som en lav verdi for dette settet med symboler. På den annen side blir verdien mellom `O4-02-0` og `O4-07-0` 2,98, like lavt som for kopiene, selv om disse (etter manuell gjennomgang) ikke er særlig like.

Filene `O4-01-*` får gjennomgående høye likhetsverdier mot de andre filene, rundt 40. Dette er på grunn av størrelsesforskjellen mellom filene. Filene `O4-01-*` er rundt 40% til 60% større enn de andre filene og inneholder dermed flere av de aktuelle symbolene.

Avstanden mellom filene `O4-0{2,3,4,5,6,7}-0.java` ligger for det meste rundt 10. Her ser vi at i tillegg til verdien 2,98 er også verdiene 4,69 (`O4-04-0` og `O4-07-0`) og 6,29 (`O4-02-0` og `O4-04-0`) bemerkelsesverdig lave, innenfor det dobbelte av den høyeste kopiverdien. Resten av verdiene ligger relativt klart utenfor området som indikerer kopiering.

Vi antar at likhetsverdier over ti indikerer tilstrekkelig grad av ulikhet for dette symbolsettet.

Grunnen til at noen originaler faller innenfor mistankeområdet er at koden inneholder omtrent like mange av de seks symbolene. Dette kan tyde på at vi bør velge andre symboler, men på den annen side kan vi ikke gardere oss fullstendig mot falske positive siden to forskjellige filer ved en tilfeldighet kan inneholde like mange av hvert av symbolene.

7.3.4 Andre forberedende test

Vi legger til symbolene `static else void String`. Symbolsett nr. 2 består dermed av symbolene `{ ; , public while class static else void String`. Resultatene registreres i tabell 7.2.

På grunn av at vi i dette forsøket bruker 10 symboler i stedet for seks skulle man anta at den euklidske avstanden ble større i dette forsøket. Grunnen til at det ikke er tilfelle er at vi dividerer verdien på N i ligning 5.1. Den høyeste verdien for kopiene er 2,06, mens `O4-02-0`, `O4-07-0` og `O4-04-0` fortsatt virker mistenkelig like med verdier innenfor det dobbelte av den høyeste kopiverdien. Resten av verdiene ligger mellom fem og ti, mens verdiene for `O4-01-*` mot de andre ligger mellom 20 og 30.

Vi finner at resultatene blir reprodusert relativt konsekvent med begge symbolsettene vi hittil har brukt. Det ser videre ut som om likhetsverdier over

	O4-01-0	O4-01-a	O4-01-b	O4-01-c	O4-02-0	O4-02-a	O4-02-b	O4-02-c	O4-03-0	O4-04-0	O4-05-0	O4-06-0	O4-07-0
O4-01-0	0,0												
O4-01-a	0,0	0,0											
O4-01-b	1,17	1,17	0,0										
O4-01-c	2,26	2,26	1,34	0,0									
O4-02-0	40,80	40,80	41,87	43,04	0,0								
O4-02-a	40,80	40,80	41,87	43,04	0,0	0,0							
O4-02-b	39,43	39,43	40,49	41,66	1,54	1,54	0,0						
O4-02-c	37,48	37,48	38,55	39,71	3,40	3,40	2,17	0,0					
O4-03-0	40,09	40,09	41,13	42,31	8,12	8,12	7,62	8,12	0,0				
O4-04-0	39,56	39,56	40,67	41,78	6,29	6,29	6,43	6,62	8,99	0,0			
O4-05-0	34,12	34,12	35,19	36,36	7,53	7,53	6,36	5,13	9,49	6,43	0,0		
O4-06-0	51,96	51,96	53,04	54,18	11,99	11,99	13,47	15,23	17,43	13,93	18,38	0,0	
O4-07-0	40,72	40,72	41,80	42,95	2,98	2,98	3,62	4,73	9,76	4,69	7,01	11,41	0,0

Tabell 7.1: Euklidsk avstand mellom obligene ved bruk av symbolsett nr. 1 i avsnitt 7.3.3.

	O4-01-0	O4-01-a	O4-01-b	O4-01-c	O4-02-0	O4-02-a	O4-02-b	O4-02-c	O4-03-0	O4-04-0	O4-05-0	O4-06-0	O4-07-0
O4-01-0	0,0												
O4-01-a	0,0	0,0											
O4-01-b	0,73	0,73	0,0										
O4-01-c	1,38	1,38	0,82	0,0									
O4-02-0	24,53	24,53	25,16	25,87	0,0								
O4-02-a	24,53	24,53	25,16	25,87	0,0	0,0							
O4-02-b	23,70	23,70	24,33	25,04	0,92	0,92	0,0						
O4-02-c	22,53	22,53	23,17	23,87	2,06	2,06	1,34	0,0					
O4-03-0	24,06	24,06	24,68	25,39	4,98	4,98	4,68	4,95	0,0				
O4-04-0	23,80	23,80	24,47	25,14	4,07	4,07	4,14	4,28	5,57	0,0			
O4-05-0	20,57	20,57	21,21	21,91	4,58	4,58	3,89	3,20	5,89	4,20	0,0		
O4-06-0	31,25	31,25	31,90	32,59	7,32	7,32	8,19	9,28	10,63	8,45	11,10	0,0	
O4-07-0	24,48	24,48	25,14	25,84	2,51	2,51	2,80	3,41	6,09	3,15	4,57	6,96	0,0

Tabell 7.2: Euklidsk avstand mellom obligene ved bruk av symbolsett nr. 2 i avsnitt 7.3.4.

seks-syv indikerer en tilstrekkelig grad av ulikhet på bakgrunn av dette symbolsettet.

7.3.5 Det tredje forsøket

Vi tar konsekvensen av diskusjonen i avsnitt 7.2.2 og gjør enda en forberedende analyse på bakgrunn av symbolsett nr. 3 som består av `{ public static class void if( else while( for(` og registrerer resultatene i tabell 7.3.

Vi bruker nå ni symboler som vi mener burde være bedre egnet til å påvise likhet enn de foregående settene. Vi finner at høyeste likhetsverdi for kopiene er redusert til 1,15, at verdiene for `O4-01-*` mot de andre ligger i overkant av ti og resten i underkant av fem.

Hvis vi legger til grunn at verdier under det dobbelte av høyeste kopiverdi er mistenkelige finner vi at `O4-03-0`, `O4-02-0` og `O4-07-0` ligner på hverandre, `O4-07-0` ligner videre på `O4-05-0` som igjen ligner på `O4-04-0`. Det er allikevel verdt å legge merke til at den laveste verdien for en falsk positiv tross alt er over 50% høyere enn den høyeste kopiverdien.

Vi ser allikevel noen endringer fra de første forsøkene med hensyn på hvilke filer som klassifiseres som tilnærmet like.

7.3.6 Normalisering av likhetsverdiene

En mulig måte å normalisere verdiene på er å dividere det på den gjennomsnittlige størrelsen av de to besvarelsene som sammenlignes eller eventuelt dividere det med størrelsen på den minste eller største besvarelsen. Dette resulterer i en svært liten verdi og vi multipliserer det derfor med 10 000 for å gjøre sammenligningen med andre normaliserte verdier enklere for oss selv.

Å dividere med den gjennomsnittlige eller største filstørrelsen er ikke hensiktsmessig siden dette vil resultere i at høye likhetsverdier vil reduseres proporsjonalt med at størrelsesforskjellen øker.

Å dividere med den minste filstørrelsen vil resultere i at vi unngår den uheldige utjevningen av resultatene beskrevet ovenfor. I tillegg vil de normaliserte verdiene fra forskjellige sammenligninger ligge nærmere hverandre. Det vil være lettere å si at normaliserte verdier over 100 indikerer stor usannsynlighet for kopiering enn å si det samme for de absolutte likhetsverdiene. I tabell 7.4 har vi normalisert verdiene fra tabell 7.3.

Vi ser at de normaliserte verdiene gjennomgående er større enn de unorma-

	O4-01-0	O4-01-a	O4-01-b	O4-01-c	O4-02-0	O4-02-a	O4-02-b	O4-02-c	O4-03-0	O4-04-0	O4-05-0	O4-06-0	O4-07-0
O4-01-0	0,0												
O4-01-a	0,0	0,0											
O4-01-b	0,0	0,0	0,0										
O4-01-c	0,75	0,75	0,75	0,0									
O4-02-0	13,35	13,35	13,35	14,03	0,0								
O4-02-a	13,35	13,35	13,35	14,03	0,0	0,0							
O4-02-b	13,35	13,35	13,35	14,03	0,0	0,0	0,0						
O4-02-c	12,36	12,36	12,36	13,03	1,15	1,15	1,15	0,0					
O4-03-0	13,32	13,32	13,32	14,01	1,75	1,75	1,75	2,21	0,0				
O4-04-0	10,38	10,38	10,38	11,07	4,66	4,66	4,66	4,30	4,47	0,0			
O4-05-0	11,25	11,25	11,25	11,93	3,25	3,25	3,25	2,95	3,56	2,22	0,0		
O4-06-0	14,97	14,97	14,97	15,67	2,32	2,32	2,32	3,40	2,74	5,34	4,20	0,0	
O4-07-0	12,14	12,14	12,14	12,84	2,24	2,24	2,24	2,31	2,10	2,97	2,17	3,03	0,0

Tabell 7.3: Euklidsk avstand mellom obligene ved bruk av symbolsett nr. 3 i avsnitt 7.3.5.

liserte. Unntaket er sammenligningen av O4-01-c mot O4-01-0,a,b hvor de normaliserte verdiene er $\sim$18% lavere, dette er positivt siden disse er kopier av hverandre. Normalverdiene for O4-02-c mot O4-02-0,a,b er $\sim$40% høyere enn de ikkenormaliserte og indikerer at normalisering av likhetsverdiene på denne måten ikke nødvendigvis gir bedre resultater, på den annen side er resultatene for normalverdiene heller ikke dårligere.

Hvis vi skal normalisere likhetsverdiene virker divisjon med minste filstørrelse foreløpig som en fruktbar måte å gjøre dette på. For å tilpasse de normaliserte verdiene til en brukbar skala kan det være nødvendig å endre multiplikasjonsfaktoren på 10 000.

7.3.7 Avskjæring av sammenligningsprosessen

På grunn av at sammenligningsalgoritmen ser på hver besvarelse i sin helhet er det ikke mulig å finne ut om en oppgave er delvis kopiert og delvis original. Siden det er besvarelsene som helhet som blir sammenlignet følger det at store størrelsesforskjeller oppgavene imellom nærmest vil garantere negative sammenligningsresultater.

Vi kan anta at en besvarelse A, som er dobbelt så stor som en annen besvarelse B (etter at kommentarer og blanke tegn er fjernet) ikke vil klassifiseres som en kopi av B av sammenligningssystemet. Dette på grunn av at de ekstra kodelinjene nødvendigvis må inneholde et antall forekomster av symbolene som telles opp.

Jevnt over vil en besvarelse A, sammenlignet med en dobbelt så stor besvarelse B, ved hjelp av ligningen på side 38, gi samme resultat som en sammenligning av A med en fil av lengde null. Dette fordi d_a i gjennomsnitt vil være halvparten så stor som d_b for en gitt dimensjon d, som gir samme resultat som om $d_b = 0$. Vi lar dermed være å sammenligne besvarelser med andre besvarelser som er dobbelt så store eller halvparten så små.

7.4 Vurdering av vektorvinkelen

I forsøkene ovenfor så vi at vurdering på bakgrunn av euklidsk avstand (normalisert eller ikke) kan gi et ikke ubetydelig antall falske positive resultater. Vi må derfor forsøke å analysere verdiene på en annen måte. Etter samtale med veileder har vi kommet frem til en alternativ måte å analysere symbolvektorene på, vi regner rett og slett ut vinkelen mellom de to symbolvektorene. Se avsnitt 5.2.3.

	O4-01-0	O4-01-a	O4-01-b	O4-01-c	O4-02-0	O4-02-a	O4-02-b	O4-02-c	O4-03-0	O4-04-0	O4-05-0	O4-06-0	O4-07-0
O4-01-0	0,0												
O4-01-a	0,0	0,0											
O4-01-b	0,0	0,0	0,0										
O4-01-c	0,62	0,62	0,61	0,0									
O4-02-0	18,78	18,78	18,78	19,74	0,0								
O4-02-a	18,58	18,58	18,58	19,52	0,0	0,0							
O4-02-b	17,97	17,97	17,97	18,89	0,0	0,0	0,0						
O4-02-c	16,09	16,09	16,09	16,96	1,62	1,62	1,62	0,0					
O4-03-0	16,03	16,03	16,03	16,86	2,46	2,43	2,36	2,88	0,0				
O4-04-0	16,84	16,84	16,84	17,96	7,56	7,56	7,56	6,96	7,25	0,0			
O4-05-0	15,81	15,81	15,81	16,76	4,57	4,57	4,57	4,15	5,00	3,60	0,0		
O4-06-0	30,26	30,26	30,26	31,68	4,69	4,69	4,69	6,87	5,54	10,79	8,49	0,0	
O4-07-0	19,55	19,55	19,55	20,68	3,61	3,61	3,61	3,72	3,38	4,82	3,49	6,12	0,0

Tabell 7.4: Verdiene fra tabell 7.3 på normalisert form

	Sett #1		Sett #2		Sett #3	
	Pos	Neg	Pos	Neg	Pos	Neg
Euklidsk avstand						
Falske	4	0	5	0	5	0
Reelle	12	17	12	16	12	16
Vektorvinkel						
Falske	4	0	2	0	1	0
Reelle	12	17	12	19	12	20

Tabell 7.5: Sammendrag av sammenligningsresultatene for de tre symbolsettene gitt ved euklidsk avstand og vektorvinkel i kategoriene *falske positive*, *falske negative*, *reelle positive* og *reelle negative*.

7.4.1 Vurdering av vektorvinkelen opp mot euklidsk avstand

Om to vektorer har omtrent lik lengde, men koden allikevel er forskjellig, er det sannsynlig at vinkelen mellom de to vektorene er relativt stor. Det kan forekomme at både den euklidske avstanden og vinkelen er relativt små på tross av at programmene er forskjellige, men sannsynligheten for at begge kriteriene skal gi et falskt positivt resultat samtidig er langt mindre enn for at et enkelt kriterium resulterer i en falsk positiv.

Vi definerer et falskt positivt resultat som en sammenligning av to ulike obliger hvor likhetsverdien (euklidsk avstand eller vektorvinkel) allikevel er lavere enn det dobbelte av den høyeste likhetsverdien for en positiv sammenligning i den samme testen. Alternativt: $FP < 2P_{max}$ hvor FP er en falsk positiv og P_{max} er den høyeste positive likhetsverdien.

I tabell 7.5 ser vi hvor mange av sammenligningene som faller i kategoriene falske positive, falske negative, reelle positive og negative for vurdering av henholdsvis euklidsk avstand og vektorvinkel for de tre forsøkene. Settene 1 og 2 er delvis overlappende, mens sett 3 er nærmest disjunkt fra de to første. Terskelen for en mistenkelig liten vektorvinkel er satt til $5°$, men bør antagelig strammes inn.

Det høyeste reelle positive resultatet for sett 1 er $1,26°$. Tilsvarende for settene 2 og 3 er hhv. $1,31°$ og $1,23°$. Vi hadde kunnet fjerne alle falske positive ved å sette terskelverdien til $3°$. Denne vinkelen hadde fortsatt vært over dobbelt så stor som største vinkel for reelle positive.

Vi ser av tabellen at vinkelsammenligningen jevnt over gir færre falske positive. Falske negative forekommer aldri, men dette er på grunn av at vi definerer terskelverdiene ut fra den høyeste reelle positives verdi. I et par tilfeller finner vi at sammenligningen av to filer som gir en mistenkelig lav

euklidsk avstand også resulterer i en relativt stor vektorvinkel, noe som indikerer at den lave verdien for euklidsk avstand indikerte mistenkelighet uten grunn.

7.4.2 Normalisering av vektorvinkelen

Vektorvinkelen vil være lettere å normalisere enn den euklidske avstanden siden den faller innenfor et endelig intervall uansett programmets størrelse og andre faktorer. I tillegg vil ulikheter i symbolforekomster mellom to sammenlignede programmer gi større utslag dess kortere de respektive vektorene er. Om vi utfører to sammenligninger av hhv. små og store programmer som begge gir den samme euklidske avstanden vil vektorvinkelen mellom de to små programmene være større enn vinkelen mellom de store. Så vi ser at utregning av vektorvinkelen allerede innebærer en implisitt normalisering. Om dette er tilstrekkelig vil vi finne ut i de kommende testene.

7.4.3 Valg av sammenligningsmetode

Som følge av disse observasjonene kan vi konkludere med at euklidsk avstand er en dårligere indikator på likhet enn vektorvinkelen. Vi vil derfor utelukkende benytte vektorvinkelsammenligning i de videre forsøkene siden dette tydeligvis gir mer pålitelige resultater.

7.5 Optimalisering av symbollistene

For å redusere ressursbruken til sammenligningsalgoritmen bør vi bruke så få symboler som mulig i analysen. Vi har sett at av forsøkssettene er det noen som skiller seg positivt ut og settet som inneholder alle symbolene gir best resultater. Vi ønsker å finne den minste delmengden av dette settet som best bevarer eller øker nøyaktigheten til det fulle settet.

Vi definerer nøyaktigheten til et symbolsett på bakgrunn av sammenligninger av javafilene som er beskrevet i avsnitt 7.3.2. Nøyaktigheten er definert som $\frac{N_{min}}{P_{max}}$ hvor N_{min} er den minste vinkelen blant de negative sammenligningene og P_{max} er den største vinkelen blant de positive sammenligningene.

7.5.1 Tilnærming med *steepest hill*

Ideelt ville vi undersøkt alle kombinasjonene av de utvalgte 15 symbolene, men det utgjør en mengde av $\sum_{i=1}^{15} \frac{15!}{i!(15-i)!} = 24\ 512$ kombinasjoner. Det er åpenbart alt for mange til at det er realistisk å undersøke dem for hånd. I stedet starter vi med å fjerne ett og ett symbol og finne ut hvilket sett av 14 symboler som er det beste. Av det utvalgte settet sammenligner vi så de 14 resulterende settene av 13 symboler og velger ut det beste som grunnlag for videre undersøkelse. Ved å benytte denne metoden reduserer vi antallet kombinasjoner til $\sum_{i=1}^{15} i = 120$ som er håndterbart selv om det innebærer en del arbeid.

Denne prosessen forutsetter at symbolsettet med 15 symboler er suboptimalt, men at det fortsatt inneholder de symbolene som utgjør det optimale settet. Dermed vil vi kunne oppnå bedre resultater med færre symboler. Disse antagelsene er fundert på observasjonene om forskjellige symboler i avsnitt 7.2.3. Vi antar videre at resultatene fra et progressivt smalere symbolsett vil beskrive en relativt jevn kurve. Det vi dermed gjør ved å fjerne ett og ett symbol er å finne veien mot det tilnærmet optimale symbolsettet ett skritt av gangen. Av alle mulige skritt velger vi det som gir umiddelbart best resultat, og derfra velger vi igjen det beste skrittet av mulighetene som nå er åpne.

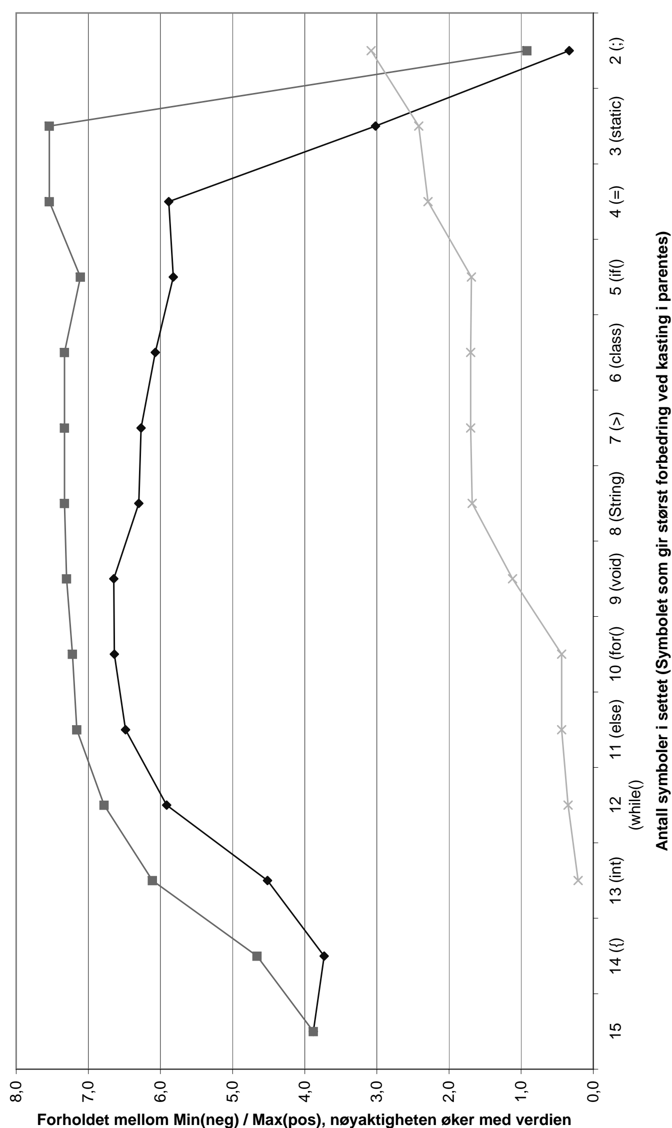
Denne metoden kalles *steepest hill climbing* [Wik06] og er kjent fra målsøkingsproblemstillinger innenfor AI-feltet, for eksempel i evolusjonssimuleringer hvor utprøving av alle utviklingsretninger er urealistisk og man velger det utviklingssteget som til en hver tid virker mest lovende.

Vi fortsetter utvelgelsen helt til vi ikke lenger ser noen økning i nøyaktigheten og vi har et optimalt symbolsett. Riktignok vil dette symbolsettet kun være optimalisert for testing av obligene de har blitt prøvd ut på, men vi antar at det vil vise seg like bra for et hvilket som helst utvalg av oppgaver. Videre forsøk vil vise om dette er tilfelle.

Testresultater

Overraskende nok viste det seg at å utelate symbolet $\{$ fra settet ga størst gevinst med god margin. Vi oppnådde faktisk en nøyaktighetsgevinst på 20% i forhold til det fulle symbolsettet.

Av grafen i figur 7.1 er det tydelig at vi med fordel kan kutte drastisk i antall symboler i settet. Det er grafen for maksimalverdier vi er interessert i, grafen nedenfor viser gjennomsnittsverdien for alle settene med et gitt antall symboler og brukes som en indikasjon på påliteligheten til resultatene



Figur 7.1: Nøyaktigheten til progressivt mindre symbolsett etter utvelgelse ved steepest hill-algoritmen. Utgangspunktet var symbolsettet `S = int while( else for( void String > class if( = { static ; , public`. Øverste kurve beskriver det beste resultatet for et sett av denne størrelsen, kurven i midten beskriver gjennomsnittsverdien av alle settene av denne størrelsen. Den nederste kurven beskriver settet av symboler som er forkastet, dette settet blir progressivt større.

for symbolsett av en gitt størrelse. Den nederste grafen viser nøyaktigheten til settet av de symbolene som etterhvert blir forkastet, det er ikke interessant utover å utgjøre et grunnlag for sammenligning med et svært dårlig symbolutvalg.

De tre første stegene bedrer resultatene seg kraftig før de så begynner å flate ut. Vi ser marginale forbedringer de neste fem stegene før kurven begynner å indikere redusert nøyaktighet når vi står igjen med fem symboler i settet. Merkelig nok viser resultatene en bemerkelsesverdig bedring for settene med fire og tre symboler, før kurven kollapser fullstendig når settet reduseres til to symboler.

Ved å sammenligne med gjennomsnittskurven ser vi at den kollapser allerede når settet reduseres til tre og selv om gjennomsnittsresultatet for et sett med fire symboler virker brukbart ligger det under de fleste større settene.

Vi går ut fra at de gode resultatene for settene av fire og tre symboler skyldes tilfeldigheter og at det optimale settet består av de seks symbolene `if public static , ; =`. Dette er neppe de symbolene vi ville valgt ut før disse testene. Ingen av dem sier egentlig noe om programstruktur, men resultatene tyder på at de allikevel karakteriserer programmene tydelig i denne sammenhengen.

I ettertid ser vi at det nok hadde vært bedre å velge det største symbolsettet med tilnærmet optimal verdi, altså det med ni elementer i stedet for det med seks elementer. Med flere elementer i settet ville vi antagelig kunne kuttet ned på antallet falske positive resultater.

Forbehold

Datagrunnlaget for *steepest hill*-testen er det samme som ble brukt for å komme frem til symbolsettet vi nå reduserer. Det er viktig å være klar over at vi tester symbolsettet på det samme grunnlaget som vi brukte for å velge ut symboler opprinnelig. Dette er en svakhet i eksperimentet, men vi antar at det resulterende symbolsettet er godt nok. Vi vil gjøre ytterligere tester på grunnlag av resultatene fra dette kapittelet og kvaliteten på resultatene fra de videre testene vil fortelle oss om dette symbolsettet er brukbart.

Obligene som brukes som datagrunnlag i disse testene er et konstruert utvalg. De representerer ikke den reelle variasjonen i obliginnleveringer i INF1000. Hensikten med disse testene var heller ikke å vurdere likhet mellom obliger, men å finne et grunnlag for å vurdere likhet. Dermed hadde vi behov for et kjent, variert datagrunnlag. Symbolsettet vi har kommet frem til vil bli brukt til likhetsvurdering på det fulle utvalget av reelle innleveringer i de to neste kapitlene. Endelig vurdering av symbolsettets kvalitet

vil måtte gjøres etter at de resultatene er klare.

På grunn av disse forbeholdene kan vi ikke være sikre på at det utvalgte symbolsettet er det beste av de vi har undersøkt. Vi vet bare at det er best på dette datagrunnlaget. Det er mindre sannsynlig at vektorvinkelen blir mistenkelig liten ved en tilfeldighet dess flere dimensjoner vektorene består av. Derfor er det godt mulig at et sett med ni symboler ville gitt bedre resultater for et bredere utvalg av obliger. Dette har vi dessverre ikke ressurser til å teste og må derfor begrense oss til de dataene vi har. Vi understreker i tillegg at mindre symbolsett reduserer antallet regneoperasjoner som må til for å bestemme vektorvinkelen.

7.5.2 Fastsetting av terskelverdier

Ut fra definisjonen av nøyaktigheten til et symbolsett i avsnitt 7.5 kan vi sette foreløpige terskelverdier for rapportering av mistenkelige obliger. Vi har vært inne på dette tidligere i kapittelet og har foreslått at terskelverdien tilsvarer to ganger verdien til vinkelen for den minst mistenkelige positive sammenligningen, $2P_{max}$. Siden den verdien er relativt lav bruker vi i stedet halvparten av verdien til den mest mistenkelige negative sammenligningen, omtrent $2°$. Terskelverdien for størrelsesforskjell ble foreslått å settes til 50% i avsnitt 7.3.7. Dette må antagelig snevres betydelig inn, men det er et greit utgangspunkt.

7.6 Oppsummering

I dette kapittelet har vi etter utstrakt testing kommet frem til et utvalg av symboler vi vil bruke som grunnlag for videre tester.

Testene har også avslørt at den euklidske avstanden, som vi hittil har antatt var det beste målet for likhet, ikke er god nok. Som følge av dette bruker vi vektorvinkelen som likhetsmål. Vi har også beskrevet metoden vi benytter for å regne ut vektorvinkelen

Resultatene fra dette kapittelet har gjort oss i stand til å foreslå terskelverdier. Disse terskelverdiene vil utgjøre utgangspunktet for undersøkelsene i de neste kapitlene.

Kapittel 8

Terskelverdier og omfanget av kopieringsproblemet

I dette kapittelet analyserer vi resultatene av eksamener og obliger levert i INF1000 høsten 2004. I motsetning til i forrige kapittel vil vi ikke se på individuelle besvarelser, men fokusere på tallenes helhetsbilde.

Formålet med dette er å komme frem til fornuftige terskelverdier siden vi ikke kan gjøre reelle sammenligninger uten å ha fastsatt disse terskelverdiene. I tillegg forsøker vi å finne ut om de som kopierer obliger får dårligere karakterer på eksamen enn de som skriver sine innleveringer selv.

8.1 Forventninger om kopiering

Før vi starter de undersøkelsene som utgjør hoveddelen av dette kapittelet er det nødvendig å klargjøre våre oppfatninger om hvem og hvor mange som bedriver kopiering.

Vi vet at kopiering forekommer og at problemet hovedsaklig viser seg i kopiering av obliger. Studier foretatt ved andre utdanningsinstitusjoner, blant annet i Storbritannia og USA, viser at svært mange av oppgavebesvarelsene i tilsvarende kurs har blitt kopiert og modifisert i større eller mindre grad. Se [HF04] og [BR04].

Som tidligere nevnt er innlevering av obligatoriske oppgaver en sentral del av informatikkundervisningen ved UiO. Eksamen er lagt opp slik at de studentene som har produsert godkjente besvarelser på obligene skal kunne stå, og obligene fungerer som en kvalifiseringsmekanisme for eksamensdeltakelse. I en ideell situasjon hvor kopiering ikke forekommer ville

strykprosenten på informatikkursene vært null siden potensielle strykkandidater ikke ville fått obligene sine godkjent i utgangspunktet.

Strykprosenten i begynnerkursene i informatikk har ligget rundt 30% så det er naturlig å anta at dette tallet representerer en nedre grense for mengden av kopieringstilfeller. I tillegg til strykkandidatene finnes det eksempler på kandidater som har levert gode obliger, men likevel får dårlige karakterer på eksamen. Disse tilfellene kan også betraktes som mistenkelige.

8.2 Diskusjon om datagrunnlaget

Som nevnt antar vi at minst 30% av studentene leverer helt eller delvis kopierte obliger. Vi antar også at de fleste kopistene vil få dårligere karakterer enn gjennomsnittet siden de mangler den programmeringserfaringen det gir å arbeide selvstendig med obligene. Det er dermed sannsynlig at kandidater som har levert gode obliger, men dårlige eksamener, helt eller delvis har kopiert obliger.

Det er nok trygt å anta at ikke alle som kopierer obliger mener å fuske, men har gått i kopieringsfellen i forbindelse med samarbeid om oppgaven. Disse kandidatenes obliger kan generere mange mistankerapporter, men kandidatene kan gjøre det bra på eksamen fordi de begge har dratt nytte av samarbeidet. I disse tilfellene vil mistankerapportene være korrekt generert og ikke falske positive siden sammenligningssystemet kun vurderer likheten mellom obliger og ikke årsakene til likhet eller motivene bak kopieringen.

For å kunne anslå omfanget av kopieringen med størst mulig grad av sikkerhet ble gruppelærerne bedt om å vurdere kvaliteten på obligene de rettet. Hver godkjent oblig ble registrert med en karakter, "God", "Middels" eller "Dårlig". Det er disse tre karakterene vi referer til med begrepet ***obligkarakterer*** i dette kapittelet. Det var i tillegg nødvendig å få tillatelse fra det Matematisk-Naturvitenskapelig fakultet til å kryssreferere obligkarakterene mot studentenes eksamenskarakterer. Vi fikk tillatelse til dette for denne spesielle undersøkelsen med forbehold om at koblingene mellom de to registrene ikke ville bli opprettholdt utover tiden det tok å utføre undersøkelsen.

8.2.1 Mistankekategorier

På bakgrunn av våre forventninger kan vi dele opp kandidatene i to kategorier, "ikke mistenkelig" og "mistenkelig". En foreløpig oppdeling kan være følgende: Eksamenskandidater med karakterene A og B er utelukket

Eksamens-resultat	Obligkarakterer		
	God	Middels	Dårlig
A	-	-	-
B	-	-	-
C	M	-	-
D	M	M	M
E	M	M	M
F	M	M	M
Syk	M	M	M
Ikke møtt	M	M	M

Tabell 8.1: Oversikt over mistankekategorier. Vi forutsetter at studenter som har levert tilfredsstillende obliger er i stand til å få C eller bedre på eksamen. De med gode obliger burde få enten A eller B på eksamen.

fra mistanke, mens kandidater som har fått en eksamenskarakter dårligere enn C ansees å falle under mistanke. Tabell 8.1 angir hvilke kombinasjoner av obligkarakter og eksamenskarakter som legges til grunn for inndelingen, mens tabell 8.2 angir hvor mange obliginnleveringer som faller i de forskjellige kategoriene. Diagrammet i figur 8.1 synliggjør hvordan obligene fordeler seg. Vi legger merke til at studenter med gode eksamenskarakterer jevnt over leverer bedre obliger enn studenter som gjør det dårligere på eksamen.

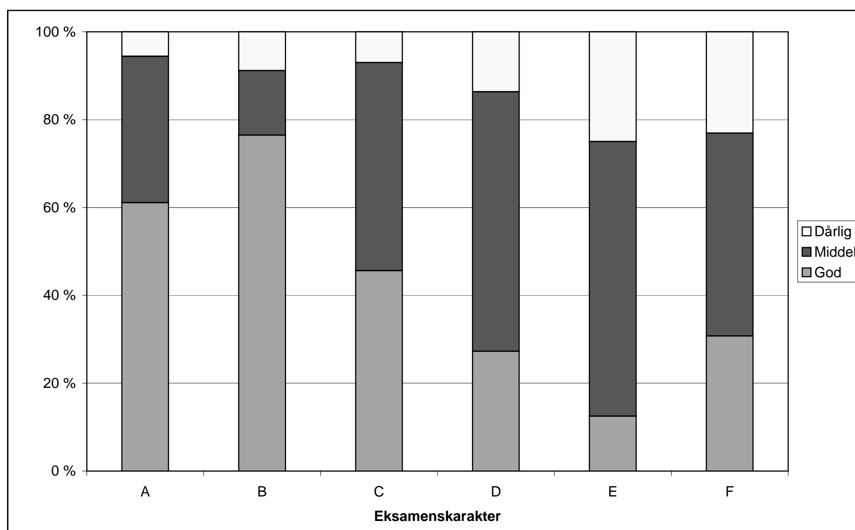
Legg også merke til at hver mistankerapport er knyttet til to obliger, levert av to forskjellige studenter, gjerne med to forskjellige eksamenskarakterer. Den samme mistankerapporten blir dermed talt opp to ganger enten i forskjellige eller samme eksamenskarakterkategori. På samme måte er det mulig at forskjellige gruppelærere har vurdert lignende obliger forskjellig.

Det er i tillegg verdt å legge merke til at kategoriene "Syk" og "Ikke møtt" inneholder svært få forekomster. Resultatene er derfor relativt upålitelige. Denne upåliteligheten forsvinner når disse kategoriene omfattes av større kategorier siden gjennomsnittstallene da regnes ut på bakgrunn av det totale antall obliger i alle underkategoriene.

Flere av obliginnleveringene er ikke tatt med i denne oversikten på grunn av at sensurdataene er mangelfulle. Flere studenter som har levert obliger er ikke registrert i sensurdataene og på samme måte finnes det eksamensresultater for kandidater som ikke er registrert med obligresultater. I dette kapittelet undersøker vi besvarelsene på oblig 4 levert av studenter vi har både sensurdata og obligkarakterer for. Dette utgjør 189 besvarelser, 54% av de innleverte besvarelsene vi har registrert for oblig 4. Vi vurderer dette

Eksamens-resultat	Obligkarakterer			Sum
	God	Middels	Dårlig	
A	11	6	1	18
B	26	5	3	34
C	26	27	4	57
D	6	13	3	22
E	2	10	4	16
F	12	18	9	39
Syk	2	0	0	2
Ikke møtt	0	0	1	1
Sum	85	79	25	189

Tabell 8.2: Tabellen viser antall obliger som har fått vurderingen *God*, *Middels* og *Dårlig* fordelt på hvilken karakter studenten som leverte obligen fikk på eksamen.



Figur 8.1: Anskueliggjøring av sannsynligheten for at dårlige obliger korresponderer med dårlige eksamenskarakterer. Hver søyle angir prosentandelen av de tre obligkarakterene på obliger levert av studenter som får en gitt eksamenskarakter. Dataene er fra tabell 8.2.

datagrunnlaget som tilstrekkelig.

8.3 Beskrivelse av undersøkelsene

Som nevnt er det primære målet med undersøkelsene i dette kapittelet å komme frem til optimale terskelverdier for symbolsettet vi har valgt å bruke for sammenligningene. I forrige kapittel fant vi et utgangspunkt for terskelverdiene på bakgrunn av de innledende undersøkelsene. Vi går derfor til å begynne med ut fra at en vektorvinkel mindre enn 2° og en størrelsesforskjell mindre enn 50% bør rapporteres som mistenkelig.

Undersøkelsene blir gjort ved å kjøre alle obligene som utgjør datagrunnlaget (dvs. de obligene hvor vi har studentens eksamensresultater) gjennom sammenligningssystemet. Ved å telle opp alle mistankerapportene som genereres for hver oblig kan vi koble antallet rapporter til studentenes eksamenskarakterer, dermed vet vi hvilke karakterer som oftest gis til studenter med mange eller få mistankerapporter knyttet til obligene sine.

Vi forutsetter at eksamenskarakteren er den beste indikatoren på om en student har kopiert sine obliger eller ikke. Dvs. at vi forventer relativt få mistankerapporter på obliger levert av A-kandidater og relativt mange flere på obliger levert av strykkandidater. Vi går ut fra at de terskelverdiene som gir tydeligst utslag mellom gode og dårlige eksamenskarakterer er bedre enn de som gir små variasjoner. Vi antar også at lave terskelverdier er bedre enn høye da sannsynligheten for falske positive resultater øker når terskelverdiene heves.

I tillegg til å etablere terskelverdiene ønsker vi å se nærmere på hva undersøkelsene forteller om hvem og hvor mange som kopierer obliger. Siden systemet fortsatt er under bygging kan ikke resultatene fra disse undersøkelsene brukes til å bekrefte omfanget av kopieringen. Meningen med disse undersøkelsene er å finne ut hvorvidt resultatene systemet gir stemmer overens med våre forventninger. Jo bedre resultatene stemmer overens med våre forventninger, dess bedre kan vi anta at systemet faktisk fungerer. Det er allikevel verdt å legge merke til at undersøkelsene i dette kapittelet fokuserer på å gjenkjenne trender i større datamengder, ikke direkte sammenligning av obligpar, det gjøres i neste kapittel.

8.3.1 Svakheter i metoden

Det å vurdere systemets pålitelighet ut fra våre forventninger og å vurdere forventningenes gyldighet ut fra hvor godt de blir bekreftet av resultatene fra systemet er selvfølgelig et eksempel på sirkulær logikk. Dermed kan

vi ikke konkludere bastant etter disse undersøkelsene. Vi hevder allikevel at forventningene våre er fornuftige og at eventuelt samsvar mellom disse og resultatene presentert i dette kapittelet til en viss grad vil bekrefte både våre forventninger og at systemet fungerer etter hensikten.

8.3.2 Personvernsproblematikk

I diskusjonen om datagrunnlaget beskriver vi hvordan vi kobler studenters eksamenskarakterer og obligkarakterer for å generere det nødvendige statistiske materialet. I neste kapittel går vi enda lenger og kobler studenter til hverandre i et forsøk på å påvise kopieringsnettverk. For å kunne utføre disse koblingene har det vært nødvendig å gå via studentenes brukernavn siden det er den eneste identifikatoren som er felles for eksamens- og obligregistrene.

Vi gjør oppmerksom på at alle data som kan knyttes til individuelle studenter er anonymisert i denne oppgaven og de koblede registrene ble slettet etter at vi genererte de nødvendige dataene. For å gjenskape datagrunnlaget vil det være nødvendig å gjenopprette koblingene mellom de to registrene.

8.4 Setting av terskelverdiene

Vi bruker resultatene for sammenligningssystemet fra forrige kapittel til å analysere dataene. Vi bruker det antatt optimale symbolsettet med en terskelverdi for vektorvinkelen på $2°$. Denne verdien er mer enn tre ganger større enn vinkelen mellom største forskjell for kopierte obliger $(0,58°)$, men fortsatt mindre enn halvparten så stor som minste vinkel mellom originale programmer $(4,25°)$. Dette burde være et godt utgangspunkt for terskelverdien.

Likheten mellom to programmer må nødvendigvis bli mindre jo mer størrelsesforskjellen øker. Positive resultater rapportert av sammenligningssystemet vil dermed med økende grad av sannsynlighet være falske positive jo mer størrelsesforskjellen øker. Det er derfor nødvendig å begrense sammenligning av programmer av svært forskjellig størrelse. I utgangspunktet lar vi terskelverdien for størrelseforskjell være 50%, det vil si at det største programmet ikke er over 50% større enn det minste programmet. Størrelsen er regnet ut etter at kommentarer, mellomrom og faste strenger er fjernet fra programmet, dermed blir dens pålitelighet som indikator større.

Etter å ha foretatt innledende undersøkelser basert på disse terskelverdiene vil vi gjøre flere tester med lavere verdier, henholdsvis $1,16°/$10% og $0,8°/$5%, for å finne ut hvordan og i hvilken grad resultatene endrer seg.

8.5 Resultater av dataanalysen

Vi starter med å se på fordelingen av mistankerapporter blant kandidatene fordelt på deres eksamenskarakterer. En mistankerapport er en forekomst i systemdatabasen som angir vektorvinkelen mellom to registrerte obligbesvarelser. I disse undersøkelsene bryr vi oss ikke om vektorvinkelen for de enkelte mistankerapporter, kun om antallet rapporter.

Vi må forvente at en del av mistankerapportene fra systemet er falske positive vi ikke har greid å luke ut. Vi kan anta at falske positive rapporter er spredt jevnt over alle kategorier. Om vi legger denne antagelsen til grunn kunne vi ha kuttet bunnen av stolpediagrammene i figurene 8.3 og utover og dermed gjort forskjellene i hvert diagram tydeligere.

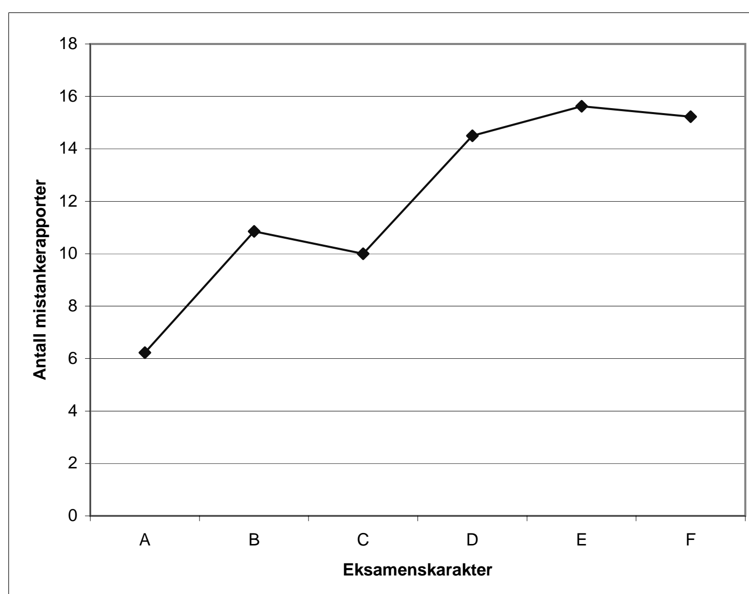
8.5.1 Resultater for høye terskelverdier

De følgende resultatene er basert på mistankerapporter der vektorvinkelen mellom et programpar er mindre enn 2° og størrelsesforskjellen er mindre enn 50% (det største programmet er mindre enn 50% større enn det minste).

Fordeling av mistankerapporter per karakter

Diagrammet i figur 8.2 viser hvordan antall mistankerapporter varierer med eksamenskarakteren obligforfatteren fikk. Verdiene regnes ut for hver eksamenskarakter ved å legge sammen alle mistankerapportene generert for obliger levert av studenter som fikk denne eksamenskarakteren, summen divideres på det totale antallet obliger levert av studenter som fikk denne eksamenskarakteren.

Det gjennomsnittlige antallet stiger jevnt fra 6 til 16 over karakterene A til E, og obliger fra studenter med karakterene E og F genererer 2,5 ganger så mange mistankerapporter som obliger fra A-kandidatene. Disse tallene samsvarer med det vi ville forvente og indikerer at sammenligningssystemet gir brukbare resultater. På den annen side ser vi at gjennomsnittet for karakteren C er lavere enn for B, noe som skulle indikere at kopiering foregår i større grad blant B-kandidater enn blant C-kandidater. Dette kan være et utslag av et samarbeid mellom to studenter som begge drar nytte av samarbeidet. Alternativt kan det rett og slett være at kopistene velger sine kilder med omhu, at de foretrekker å kopiere de flinkere studentene. Eller det kan være at terskelverdiene i dette tilfellet er for høye. Som tidligere nevnt kan ikke systemet se forskjell på original og kopi, begge blir implisert i mistankerapporten.



Figur 8.2: Antall mistankerapporter (Y) generert fra obliger levert av studenter med disse eksamenskarakterene (X). Terskelverdiene er $2°/50\%$

Gjennomsnittlig antall mistankerapporter per oblig fordelt på obligens karakter og kandidatens eksamenskarakter

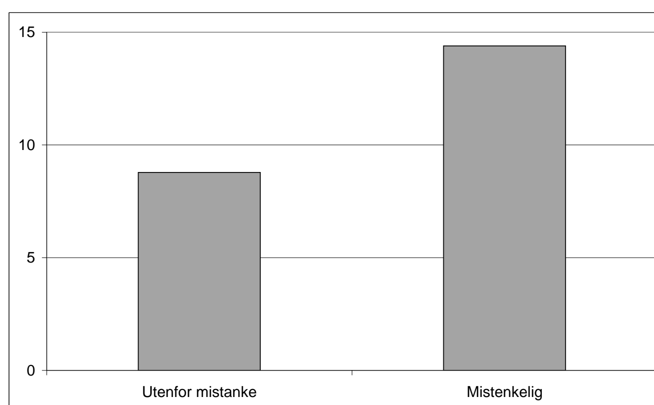
Her gjør vi en litt mer finkornet oppdeling av dataene, basert på mistankekategoriene i tabell 8.1. I dette tilfellet tar vi utgangspunkt i de enkelte obligene og antall mistankerapporter hver av dem genererer. Vi regner deretter ut gjennomsnittlig antall genererte mistankerapporter for alle obliger i hver oblig- og eksamenskarakterkategori for å finne ut om vi ser noen signifikant forskjell mellom kategoriene. Verdiene for de forskjellige kategoriene finnes i tabell 8.3.

I et forsøk på å gjøre resultatene enda tydeligere har vi regnet ut tilsvarende gjennomsnittsverdier for kategoriene “ikke mistenkelige” og “mistenkelige” obliger definert i avsnitt 8.2.1. Disse resultatene vises i stolpediagrammet i figur 8.3.

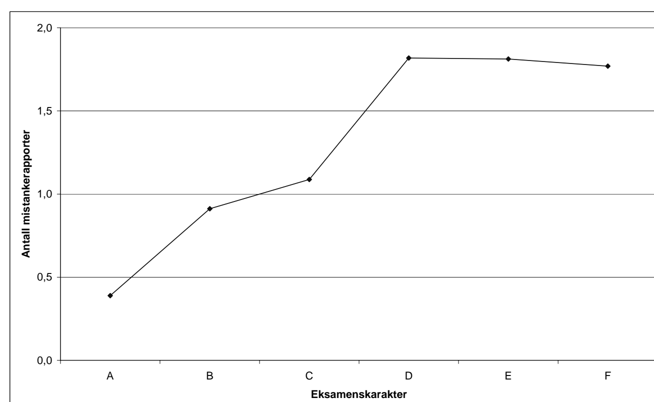
Vi legger merke til at obliger i kategorien “Utenfor mistanke” i gjennomsnitt har færre mistankerapporter enn den gjennomsnittlige mistenkelige obligen. Gjennomsnittet for kategorien “Mistenkelig” er 64% høyere enn for kategorien “Utenfor mistanke.” Disse resultatene møter våre forventninger.

Eksamens-resultat	Obligkarakterer		
	God	Middels	Dårlig
A	5,1	8,7	4,0
B	11,1	5,6	18,0
C	12,4	8,5	4,5
D	4,3	19,4	13,7
E	26,5	14,3	13,5
F	12,7	19,8	9,4
Syk	13,5	-	-
Ikke møtt	-	-	13,0

Tabell 8.3: Gjennomsnittlig antall mistankerapporter per levert oblig i hver kategori.



Figur 8.3: Gjennomsnittlig antall mistankerapporter (Y) per oblig fordelt på mistankekategoriene (X) beskrevet i tabell 8.1. Terskelverdiene er $2°/50\%$



Figur 8.4: Gjennomsnittlig antall mistankerapporter (Y) generert fra obliger levert av studenter med disse eksamenskarakterene (X). Terskelverdiene er $1,16°/10\%$

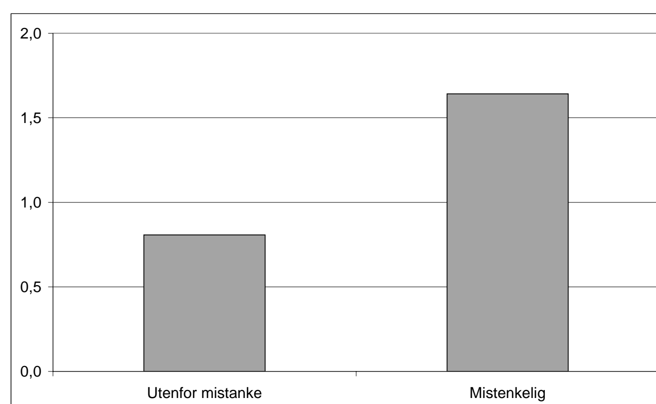
8.5.2 Resultater for moderate terskelverdier

Resultatene i dette avsnittet er basert på mistankerapporter der vektorvinkelen mellom et programpar er mindre enn $1,16°$ og størrelsesforskjellen er mindre enn 10%. Verdien $1,16°$ er dobbelt så stor som den største vektorvinkelen for kopierte programmer i forrige kapittel, mens en størrelsesforskjell på 10% kan antas å utelukke mange falske positive rapporter uten å overse faktiske kopieringstilfeller.

Vi forventer at trendene som er etablert i forrige avsnitt blir mer markante etter at utvalget av mistankerapporter begrenses i henhold til disse terskelverdiene. Dette ville i så fall bekrefte at lavere terskelverdier øker systemets nøyaktighet.

Gjennomsnittlig fordeling av mistankerapporter per karakter

Diagrammet i figur 8.4 tilsvarer diagrammet i avsnitt 8.5.1. Vi ser en lignende trend i dette diagrammet med en skarp stigning fra C til de tre dårligste karakterene, men obliger fra E-/strykkandidater genererer nå nesten fem ganger så mange mistankerapporter som obligene fra A-kandidatene. Fordelingen av mistankerapporter mellom karakterene er med disse terskelverdiene nærmere den økningen vi forventet.



Figur 8.5: Gjennomsnittlig antall mistankerapporter (Y) per oblig fordelt på mistankekategoriene (X) beskrevet i tabell 8.1. Terskelverdiene er $1,16°/10\%$

Gjennomsnittlig antall mistankerapporter per oblig fordelt på obligens karakter og kandidatens eksamenskarakter

Vi bruker den samme oppdelingen av dataene som i avsnitt 8.5.1 med resultater som vist i figur 8.5.

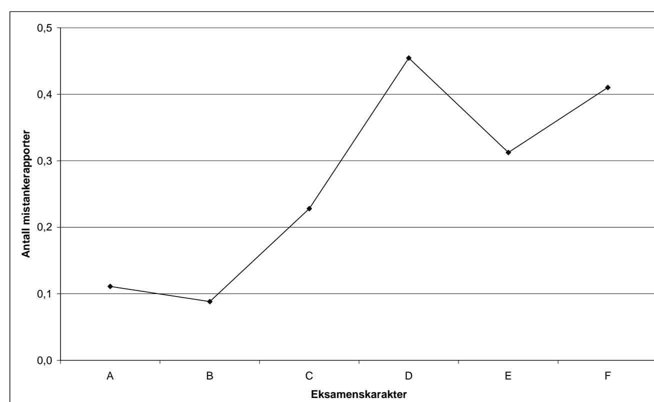
Her ser vi at trenden fra avsnitt 8.5.1 er blitt mer markant. Med de gjeldende terskelverdiene finner vi at de mistenkelige obligene genererer 103% flere mistankerapporter enn de som faller utenfor mistanke. Dette tyder på at innsnevringen av terskelverdiene gir bedre resultater.

8.5.3 Resultater for lavere terskelverdier

Vi gjentar de foregående sammenligningene i dette avsnittet for å finne ut om trendene holder for enda lavere terskelverdier. Terskelverdiene er i disse sammenligningene satt til vektorvinkel mindre enn $0,8°$ og størrelsesforskjell mindre enn 5%.

Gjennomsnittlig fordeling av mistankerapporter per karakter

Diagrammet i figur 8.6 tilsvarer diagrammet i avsnitt 8.5.1. Trenden fra de foregående sammenligningene har blitt utydelig, selv om obliger fra



Figur 8.6: Gjennomsnittlig antall mistankerapporter (Y) generert fra obliger levert av studenter med disse eksamenskarakterene (X). Terskelverdiene er $0,8°/5\%$

E-/strykkandidatene fortsatt genererer nesten fire ganger flere mistanke-rapporter enn A-kandidatenes obliger.

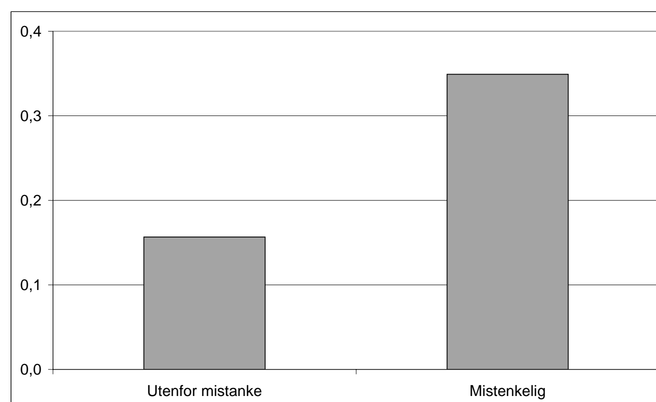
Det kan virke som disse resultatene lider av en for sterk begrensning av dataene.

Gjennomsnittlig antall mistankerapporter per oblig fordelt på obligens karakter og kandidatens eksamenskarakter

Vi bruker den samme oppdelingen av dataene som i avsnitt 8.5.1 med resultater som vist i figur 8.7.

Forskjellen mellom mistankekategoriene har øket til 123%. På dette nivået ser vi altså at de gjeldende terskelverdiene ser ut til å gi bedre resultater enn de foregående.

Jevnt over virker det som om disse terskelverdiene gir de beste resultatene, så vi vil benytte dem i undersøkelsene videre.



Figur 8.7: Gjennomsnittlig antall mistankerapporter (Y) per oblig fordelt på mistankekategoriene (X) beskrevet i tabell 8.1. Terskelverdiene er $0,8°/5\%$

8.6 Konklusjoner

Det virker som om de beste resultatene fulgte av terskelverdier på $0,8°$ for vektorvinkelen og 5% for størrelsesforskjellen. Vi så tegn til at resultatenes forutsigbarhet ble svekket på visse punkter da vi senket terskelverdiene til dette nivået, noe som indikerer at de ikke bør snevres ytterligere inn. Vi velger å bruke disse terskelverdiene videre.

Vi har nå etablert alle verdiene vi har behov for før valideringen av resultatene gjøres i neste kapittel.

8.6.1 Hvordan fordeler kopieringstilfellene seg?

Det ser ut som om sammenligningssystemet møter forventningene våre. Antallet mistankerapporter øker med graden av mistanke, noe som stemmer godt med våre antagelser.

Vi hadde ikke forutsett at systemet skulle gi såpass mange mistankerapporter på obliger vi hadde plassert utenfor mistanke. Falske positive mistankerapporter står for en del av dette, men vi kan heller ikke se bort fra at obligsamarbeid (hvor begge partnerene gjør en innsats) faktisk øker sjansen for å få en god karakter på eksamen. På den annen side er det sannsynlig at de som kopierer velger originalbesvarelser fra flinke studenter. Dette sannsynliggjør at obliger fra flinke studenter, som blir rapportert som mis-

tenkelige, for det meste er originaler som er lånt ut til kopiering heller enn faktiske kopier.

Om vi antar at fordelingen av falske positive mistankerapporter er konstant over mistankekategoriene og ser bort fra et likt antall mistankerapporter i hver kategori vil vi se at de eksisterende trendene blir tydeligere og forskjellene mellom kategorienes verdier vil øke.

Selv om vi har fått visse indikasjoner på at kopieringstilfellene korresponderer med eksamensresultater må det nærmere undersøkelser til før vi kan si noe bastant om dette. De som er interessert i dette temaet oppfordres til å gå videre med dette arbeidet.

Kapittel 9

Validering av sammenligningsresultatene

I forrige kapittel undersøkte vi hvor mange obligpar som var mistenkelige, definert ved at de falt innenfor på forhånd definerte terskelverdier for størrelsesforskjell og likhet.

For å validere verdien av disse resultatene plukker vi ut noen av de mistenkelige obligparene og sammenligner dem mot hverandre. Jo flere av parene som faktisk er like, dess bedre var de opprinnelige resultatene.

For å få et brukbart utvalg plukker vi kun mistankerapporter hvor begge obligene er besvarelser på Oblig 4 fra høsten 2004. Obligene vil derfor i utgangspunktet ha tydelige likheter hva angår klasse- og datastruktur. Studentene har fått de samme hintene fra gruppelærere og på forelesninger i tillegg til at oppgaven legger visse føringer på hvordan løsningen skal se ut. Likhet og ulikhet må derfor defineres strammere enn det som kanskje er naturlig siden to programmer som løser den samme oppgaven på tilstrekkelig forskjellige måter vil vurderes som ulike.

9.1 Kategorisering av sammenligningsresultatene

Datagrunnlaget for undersøkelsene i dette kapittelet utgjøres av mistankerapporter generert fra 352 besvarelser. 2254 av de 61776 potensielle sammenligningene er mistenkelige nok til å generere mistankerapporter. 401 av disse rapportene fordeler vi i kategorier etter grad av usikkerhet: Usikre, Ganske Sikre og Sikre. Denne kategoriseringen gjøres utelukkende på bakgrunn av maskinens sammenligningsresultater og involverer ikke menneskelig skjønn. Kategoriseringsskjemaet er vist i tabell 9.1

Fra hver av disse kategoriene plukker vi ca. 20 rapporter som vi gransker nøye. Vi gir vår personlige vurdering av likhetsgraden mellom obligene involvert i hver av rapportene i utvalget. Vi kategoriserer hver av de undersøkte rapportene i henhold til hvor stor grad av likhet vi finner.

Målet med disse undersøkelsene er å vurdere nøyaktigheten i sammenligningssystemets rapporter, og samtidig å fastsette endelige terskelverdier for mistankerapportering.

9.1.1 Maskinens kategorier

Usikre rapporter karakteriseres ved at størrelsesforskjellen mellom de to sammenlignede obligene ligger mellom 10% og 50% og at vektorvinkelen ligger mellom $1,16^{\circ}$ og $2,0^{\circ}$. (Ingen rapporter har størrelsesforskjell større enn 50% eller vektorvinkel større enn $2,0^{\circ}$ da disse rapportene svært sjelden har noen som helst verdi og dermed ikke blir generert.) Størrelsesforskjellen er alene nok til å indikere at en av obligene inneholder mer kode enn den andre og at de dermed ikke kan være identiske. Det er fortsatt mulig at for eksempel ekstra utskriftslinjer er lagt til i en av obligene og i disse tilfellene kan en lav vektorvinkel indikere mistenkelig likhet. I de fleste fall vil allikevel en lav vektorvinkel bero på tilfeldigheter når størrelsesforskjellen er såpass stor.

Ganske sikre rapporter karakteriseres ved at størrelsesforskjellen ligger mellom 5-10% og vektorvinkelen mellom $0,8^{\circ} - 1,16^{\circ}$. I disse tilfellene vil størrelsesforskjellen enten bero på tilfeldigheter eller på at noen få linjer eller linjeskift er lagt til en ellers identisk oblig. I det første tilfellet vil vektorvinkelen for det aller meste være stor nok til at rapporten faller utenfor denne kategorien, men rapportene som faller i denne kategorien er ofte verdt å undersøke da mange av dem antagelig forårsakes av mistenkelig likhet som ofte grenser mot identisk likhet.

Sikre rapporter karakteriseres ved at størrelsesforskjellen ligger under 5% og vektorvinkelen under $0,8^{\circ}$. Sannsynligheten for at dette skjer ved en tilfeldighet er svært liten, så falske positive vil ikke forekomme særlig ofte. De fleste av rapportene i denne kategorien handler om grove tilfeller av kopiering.

9.1.2 Kategorier for skjønnsvurdering

Vi plukker tilfeldig ut rundt 20 mistankerapporter fra hver av kategoriene beskrevet ovenfor og vurderer likheten i hvert tilfelle. Hvert obligpar vi undersøker blir plassert i en av de følgende kategoriene på bakgrunn av en

Maskinens kategorier

Sdiff	Vektorvinkel		
	$< 0,8°$	$0,8°$-$1,16°$	$1,16°$-$2,0°$
<5%	Sikre 85		
5% -10%		Ganske sikre 140	
10% -50%			Usikre 176

Tabell 9.1: Sdiff angir størrelsesforskjellen i prosent. Vi analyserer kun de 401 mistankerapportene som faller i de tre kategoriene angitt i skjemaet. De 1853 rapportene som faller i de seks andre kategoriene er uinteressante i denne sammenhengen. Tallet under kategoribenevnelsen angir hvor mange rapporter som faller i denne kategorien.

skjønnsvurdering.

Ikke lik Obligene er ulike. Vi vil etter en rask sammenligning konkludere med at dette åpenbart ikke dreier seg om et tilfelle av kopiering.

Litt lik Noen metoder ligner på hverandre. Strukturen er relativt lik. Tilfellene i denne kategorien er svært sannsynlig ikke resultat av kopiering, men heller av et begrenset samarbeid, noe studentene oppfordres til.

Ganske lik Vi finner noen forskjeller i metodene. Strukturen er lik. Disse tilfellene er litt for like, men det er ikke åpenbart at denne likheten er et resultat av kopiering. Mest sannsynlig dreier det seg om et litt for nært samarbeid.

Nær identisk Metode- og variabelnavn samt kommentarer og faste strenger er endret mellom obligene. Bruken av mellomrom og linjeskift varierer. Dette er åpenbare tilfeller av kopiering, selv om kopisten i hvert fall har tatt seg bryet med å forsøke og skjule opphavet til programmet han leverer.

Identisk Obligene er i det alt vesentlige identiske. Dette er et åpenbart kopieringstilfelle og studenten har ikke engang gjort nevneverdige forsøk på å endre programmet han har lånt.

Mistankerapporter for obligpar som vurderes som *ikke like* eller *litt like* definerer vi som falske positive. De som vurderes som *nær identiske* eller *identiske* defineres som reelle positive. *Ganske like* vurderinger er tvilstilfeller.

9.1.3 Manuell vurdering av maskinens mistankerapporter

Vi deler først maskinens mistankerapporter inn i kategoriene *usikre*, *ganske sikre* og *sikre* rapporter, basert på likhetsverdi og størrelsesforskjell. Vi tar for oss rapportene fra hver av mistankekategoriene, en for en, og gjør en manuell vurdering av et utvalg av rapportene fra hver kategori. Den manuelle vurderingen resulterer i en av likhetsvurderingene i forrige avsnitt.

Vurdering av *usikre* mistankerapporter

Mistankerapportene i denne kategorien er definert ved at størrelsesforskjellen mellom obligene er større enn 10% og vektorvinkelen er større enn $1,16^\circ$, men mindre enn 50%/$2,0^\circ$ siden obligpar med verdier høyere enn dette ikke gir opphav til mistankerapporter. Det er totalt 176 mistankerapporter som faller i denne kategorien. Ved gjennomgang vurderte vi 20 obligpar, altså 11% av det totale antall forekomster. Disse ble klassifisert som følger:

Av mistankerapportene i denne kategorien kan alle betegnes som falske positive. Det viser seg at disse terskelverdiene er for høye.

Usikre mistankerapporter

Vurdering	Antall		Kopi/ikke kopi	%
Ikke like	19			
Litt like	1	20	Falske positive	100%
Ganske like	0	0	Tvilstilfeller	0%
Nær identiske	0			
Identiske	0	0	Reelle kopier	0%
Alle	20	20		100%

Tabell 9.2: Manuell vurdering av maskinens usikre mistankerapporter. Kolonnene til høyre klassifiserer vurderingene som falske eller reelle positive.

Vurdering av *ganske sikre* mistankerapporter

Mistankerapportene i denne kategorien er definert ved at størrelsesforskjellen ligger mellom 5-10% og vektorvinkelen mellom $0,8^\circ$-$1,16^\circ$. Vi vurderte 17 obligpar av de 140 mistankerapportene som faller i denne kategorien, dvs. 12% av forekomstene. Disse ble klassifisert som følger:

De tre parene som ble vurdert som ganske like ville berettige en nærmere undersøkelse, men 82% klart falske positive er for mye til å forsvare selv

disse relativt lave terskelverdiene.

Ganske sikre mistankerapporter

Vurdering	Antall		Kopi/ikke kopi	%
Ikke like	8			
Litt like	6	14	Falske positive	82%
Ganske like	3	3	Tvilstilfeller	18%
Nær identiske	0			
Identiske	0	0	Reelle kopier	0%
Alle	17	17		100%

Tabell 9.3: Manuell vurdering av maskinens ganske sikre mistankerapporter. Kolonnene til høyre klassifiserer vurderingene som falske eller reelle positive.

Vurdering av *sikre* mistankerapporter

Mistankerapportene i denne kategorien er definert ved at størrelsesforskjellen ligger under 5% og vektorvinkelen under $0,8°$. Vi vurderte 19 av de 85 obligparene som faller i denne kategorien, utvalget utgjør 22% av det totale antall forekomster. Disse ble klassifisert som følger:

Med denne senkingen av terskelverdiene ser vi at antallet falske positive er redusert ned mot 50%. Dette er fortsatt for høyt, men utgjør allikevel en signifikant forbedring i forhold til de mer usikre utplukkene.

Sikre mistankerapporter

Vurdering	Antall		Kopi/ikke kopi	%
Ikke like	6			
Litt like	4	10	Falske positive	52%
Ganske like	2	2	Tvilstilfeller	11%
Nær identiske	3			
Identiske	4	7	Reelle kopier	37%
Alle	19	19		100%

Tabell 9.4: Manuell vurdering av maskinens sikre mistankerapporter. Kolonnene til høyre klassifiserer vurderingene som falske eller reelle positive.

	Kategorier for maskinvurdering			
	Usikre	Ganske sikre	Sikre	Alle
Terskel-verdier	10%-50% $1,16°$-$2,0°$	5%-10% $0,8°$-$1,16°$	<5% $< 0,8°$	<50% $< 2,0°$
Ikke lik	19 (95%)	8 (47%)	6 (32%)	33 (59%)
Litt lik	1(5%)	6 (35%)	4 (21%)	11 (20%)
Ganske lik	-	3 (18%)	2 (11%)	5 (9%)
Nær identisk	-	-	3 (16%)	3 (5%)
Identisk	-	-	4 (21%)	4 (7%)
Sum	20 (100%)	17 (100%)	19 (100%)	56 (100%)

Tabell 9.5: Resultatene av vurderingen av de utvalgte mistankerapportene viser graden av likhet mellom obligpar rapportert i hhv. usikre, ganske sikre og sikre mistankerapporter. Terskelverdiene er størrelsesforskjell/vektorvinkel.

Mistenkelige obliger

	# Totalt	# Utvalgt	% Lavt anslag	% Høyt anslag
Usikre	176	20	0%	0%
Ganske sikre	140	17	0%	18%
Sikre	85	19	37%	48%
Sum	401	56	13%	21%

Tabell 9.6: Sammenfatning av mistankerapportene fra besvarelser på oppgave 4. “Lavt anslag” angir prosentvis hvor mange av mistankerapportene som har opphav i faktiske kopieringstilfeller, de reelle positive. “Høyt anslag” inkluderer også tvilstilfellene, de “Ganske sikre” mistankerapportene.

Oppsummering av likhetsvurdering av mistankerapporter

Tabellene 9.5 og 9.6 viser resultatene av vår vurdering av de utvalgte mistankerapportene. 13% av de 401 mistankerapportene kommer av åpenbar kopiering. Dette øker til 21% om vi legger til mistankerapporter fra obligpar som er verdt å undersøke nærmere. De fleste mistankerapportene med opphav i reelle kopieringstilfeller var hentet fra kategorien for sikre rapporter, noe som indikerer at terskelverdiene for denne kategorien ikke var satt for lavt. Faktisk viser det seg at terskelverdiene med fordel kan settes enda lavere, som beskrevet i avsnitt 9.1.5.

Med utgangspunkt i disse resultatene kan vi beregne et alternativt estimat for omfanget av kopieringsaktiviteten blant studentene som leverte besvar-

elser på den fjerde oppgaven. Vi har to anslag, et lavt basert på obligparene vurdert som “Nær identiske” eller “Identiske” og et høyt hvor vi i tillegg tar med obligparene vurdert som “Ganske like”.

Vi antar at frekvensen av likhet blant hele mengden av mistankerapporter tilsvarer frekvensen blant utvalget. Vi antar videre at sammenligninger som ikke genererer mistankerapporter ikke representerer falske negative siden frekvensen av kopieringstilfeller blant utvalget av usikre rapporter allikevel er null.

Vårt høye anslag over studenter involvert i kopiering, Ant_H, er basert på dataene i tabell 9.6. Anslaget er gitt ved $Ant_H = 0{\cdot}176 + 0,18{\cdot}140 + 0,48{\cdot}85 = 66$. Det lave anslaget, Ant_L, er gitt ved $Ant_L = 0,37 \cdot 85 = 31$.

Siden hver rapport involverer to obliger og vi vet at tilfeller av nettverkskopiering er svært uvanlig (se avsnitt 9.2), antar vi at hver mistankerapport som representerer et faktisk kopieringstilfelle involverer to studenter.

Blant 352 innleverte obliger (fra 352 forskjellige kandidater) kan vi dermed anslå at minst $2\frac{Ant_L}{352} = 18\%$ kandidater er involvert i kopiering. Det er naturlig å anta at halvparten skriver originalen og låner den ut, mens den andre halvparten kopierer. Et høyere anslag er $2\frac{Ant_H}{352} = 38\%$ med samme fordeling mellom originalforfatter og kopist. Om vi antar at det faktiske tallet ligger midt imellom disse ytterpunktene stemmer det godt overens med våre opprinnelige antagelser.

9.1.4 Størrelsesforskjellens betydning

Alternativt utvalg av mistankerapporter

Vurdering	Antall	Kopi/ikke kopi	%
Ikke like Litt like	14	Falske positive	48%
Ganske like	0	Tvilstilfeller	0%
Nær identiske Identiske	15	Reelle kopier	52%
Alle	29		100%

Tabell 9.7: Manuell vurdering av mistankerapporter med størrelsesforskjell $< 20\%$ og vektorvinkel $< 0,5°$.

I den foregående kategoriseringen undersøker vi ikke mistankerapporter hvor størrelsesforskjellen er høy og vektorvinkelen lav eller vice versa. Siden en for høy vektorvinkel tydeligvis leder til dårligere resultater undersøker vi fordelingen av mistankerapportene med størrelsesforskjell mindre

enn 20% og vektorvinkel mindre enn $0,5°$. Dette utgjør 29 rapporter alt i alt og vi undersøker alle sammen. Resultatene oppsummeres i tabell 9.7.

I denne vurderingen kategoriserte vi ingen rapporter som *ganske like*. Ved nærmere ettertanke er det sannsynlig at man ved skjønnsvurdering avgjør ganske raskt om to obliger er like eller ikke, og etter lang erfaring med retting av obliger er det sjelden man er i tvil. Dette er antagelig grunnen til fordelingen vi finner mellom kategoriene.

I etterpåklokskapens lys må vi nok innrømme at fem kategorier for skjønnsvurdering var for mange. Når rutiner for skjønnsvurdering skal implementeres i systemet vil brukerne (gruppelærerne) kun ha to valg:

Ikke mistenkelig som betyr at rapporten ikke vil forfølges da det antagelig ikke er snakk om kopiering.

Mistenkelig som betyr at det er nødvendig å undersøke tilfellet nøyere blant annet ved å prate med de involverte studentene.

Å bestemme graden av mistenkelighet med større nøyaktighet enn dette er ikke nødvendig da disse to reaksjonsformene er de eneste som er aktuelle.

9.1.5 Innsnevring av terskelverdiene

Vi ser på verdiene til de individuelle mistankerapportene som vist i tabell 9.8. ***Ulike*** betegner parene vurdert som *Ikke like* eller *Litt like*, ***Like*** betegner parene vurdert som *Identiske* eller *Nær identiske*, *Sdiff* betegner størrelsesforskjellen og *Vinkel* angir vektorvinkel.

Tabell 9.9 viser bare mistankerapportene hvor størrelsesforskjellen er lavere enn 5%.

Vi ser av tabell 9.8 at av obligparene med størrelsesforskjell over 5% er 11 av 12 falske positive, mens kun tre av 17 er falske positive når størrelsesforskjellen er under 5%. Senker vi terskelverdien for størrelsesforskjell til 2% ser vi at vi har like mange falske positive, men vi mister noen av de reelle kopitilfellene. Det virker som om 5% er en god terskelverdi for størrelsesforskjell.

Med utgangspunkt i rapportene med størrelsesforskjell opp til 20% ser vi at 13 av 18 (72%) av rapportene med vektorvinkel over $0,3°$ er falske positive, mens kun en av 11 er falsk positiv for vektorvinkler under $0,3°$. For rapportene med vektorvinkel under $0,3°$ har ni stykker en vektorvinkel nær null.

Med utgangspunkt i rapportene med størrelsesforskjell under 5% (representert i tabell 9.9) ser vi at falske positive blir helt eliminert når terskel-

Vurdering av rapporter med varierende størrelsesforskjell

	Ikke lik/Litt lik (14)	Identisk/Nær identisk (15)
$Sdiff > 5\%$	11	1
$Sdiff < 5\%$	3	14
$Sdiff < 2\%$	3	11

Vurdering av rapporter med varierende vektorvinkel

	Ikke lik/Litt lik (14)	Identisk/Nær identisk (15)
$Vinkel > 0,3°$	13	5
$Vinkel < 0,3°$	1	10
$Vinkel \approx 0$	0	9

Tabell 9.8: Fordelingen av mistankerapporter med størrelsesforskjell mindre enn 20% og vektorvinkel mindre enn $0,5°$ fra utvalget i avsnitt 9.1.4.

Vurdering av rapporter med varierende vektorvinkel

	Ikke lik/Litt lik (3)	Identisk/Nær identisk (14)
$Vinkel > 0,3°$	3	5
$Vinkel < 0,3°$	0	9
$Vinkel \approx 0$	0	8

Tabell 9.9: Fordelingen av mistankerapporter med størrelsesforskjell mindre enn 5% og vektorvinkel mindre enn $0,5°$ fra utvalget i avsnitt 9.1.4.

verdien for likhet reduseres fra $0,5^\circ$ til $0,3^\circ$, men vi mister fem av 14 reelle mistankerapporter. Det er en reduksjon på 36% reelle kopitilfeller.

9.1.6 Mindre kompliserte obliger

Alle disse dataene bygger på undersøkelser av besvarelser på den fjerde oppgaven i INF1000. Ved å hente ut tilfeldige mistankerapporter fra alle innleverte oppgaver finner vi for det første at besvarelsene på oppgave 2 er likere enn oppgave 3 som igjen er likere enn oppgave 4. Dette er som forventet da mindre kompleksitet i mindre grad åpner for egen kreativitet.

I et tilfeldig utvalg gjelder over halvparten av mistankerapportene besvarelser på oppgave 2, mens det er nesten dobbelt så mange rapporter knyttet til oppgave 3 som til oppgave 4 selv om antallet reelle kopitilfeller var relativt konstant for hvert sett med obliger. Dette indikerer at systemet vil generere desto flere falske positive mistankerapporter for mindre komplekse obliger enn for mer komplekse.

Det kan virke som om en senkning av terskelen for vektorvinkelen vil kunne bedre resultatene for mindre kompliserte obliger. Ved å bruke en terskelverdi for likhet på $0,3^\circ$ ved sammenligning av besvarelser på oppgave 3 og $0,1^\circ$ for oppgave 2 greier vi å påvise de mistenkelige tilfellene samtidig som vi utelukker de fleste falske positive rapportene. Disse resultatene er kun basert på det allerede sammenlignede utvalget av mistankerapporter, så ytterligere undersøkelser er nødvendig for å bekrefte dem. Det er kan være fruktbart å undersøke hvorvidt en terskelverdi for likhet kunne baseres på størrelsen til de sammenlignede programmene, for eksempel $VEKTORVINKEL_{terskel} = \frac{ANTLINJER}{1000}$ hvor $ANTLINJER$ er linjeantallet til det minste programmet.

9.2 Nettverkskopiering

I og med at hver enkelt oblig blir registrert med eget nummer i databasen kan vi kartlegge i hvilken grad nettverkskopiering forekommer. Ved å ta utgangspunkt i de 15 rapportene vurdert som ***identiske/nær identiske*** i avsnitt 9.1.4 finner vi at det i de aller fleste tilfeller handler om par av studenter som leverer mistenkelig like obliger, kun i to tilfeller finner vi trioer av studenter som leverer mistenkelig like obliger.

Vi kan med stor grad av sikkerhet slå fast at nettverkskopiering ikke er et problem. Der kopiering forekommer lages det for det meste kun én kopi av den originale obligen. Organisert fusking kan dermed utelukkes.

	Obligkarakterer				
	God	Middels	Dårlig	Alle	%
A				0	0%
B	1			1	2%
C	2	4	1	7	8%
D		1	1	2	4%
E		3		3	10%
F	5	6		11	13%
	8	14	2	24	7%

Tabell 9.10: Mistenkelige besvarelser på oppgave 4 i et tilfeldig utvalg. Tallene angir antallet mistenkelige obliger i hver kategori, %-tallene angir andelen av mistenkelige obliger som ble levert av studenter med denne eksamenskarakteren.

9.3 Kopieringstilfeller og karakterer

Siden vi har tilgang til obligkarakterer og eksamenskarakterer kan vi også undersøke hva slags karakterer de studentene som er involvert i kopiering får. Resultatene som vises i tabell 9.10 er produsert på bakgrunn av et tilfeldig utvalg av mistankerapporter med størrelsesforskjell under 20% og vektorvinkel under $0,8°$. Tallene angir hvor mange obliger som er involvert i tilfeller som er vurdert som mistenkelig. Siden det er enkeltobliger som telles kan det tenkes at den samme mistankerapporten blir talt to ganger, representert ved begge obligene som er involvert. Dette er gjort siden forfatterne av de to obligene kan ha fått forskjellig eksamenskarakter (og kanskje obligkarakter) og man må ta høyde for dette når man leser resultatene. Den siste kolonnen angir hvordan mistenkelige obliger fordeler seg på eksamenskarakterer.

Vi ser at svært få av de mistenkelige obligene er levert av studenter som har gjort det godt på eksamen, mens desto flere er levert av strykkandidatene, dette er som forventet. Det er verdt å legge merke til at et betydelig antall er levert av C-kandidatene. Selv om dette kan komme av manglende representativitet i utvalget kan årsaken også være at disse studentene er trygge på å stå på eksamen uten å legge særlig arbeid i obligene.

9.4 Konklusjoner

Vi ser at systemets vurdering stemmer bedre og bedre overens med resultatene av den manuelle likhetsvurderingen for lavere terskelverdier. Resulta-

tene vi etterhvert kom frem til viste seg å være svært tilfredsstillende etter utprøving på det fullstendige datagrunnlaget.

Den optimale terskelverdien for størrelsesforskjell ser ut til å være rundt 5% for oppgave 4. Dette stemmer overens med resultatet vi kom frem til i forrige kapittel.

Gitt en størrelsesforskjell under 5% vil man kunne så godt som eliminere alle falske positive ved å bruke en terskelverdi for likhet på $0,3°$. Man vil ved bruk av disse verdiene måtte akseptere å gå glipp av nærmere 40% av reelle kopitilfeller, noe som ikke er særlig effektivt.

Men kan fange opp de aller fleste reelle kopitilfellene ved å sette terskelverdien for likhet til $0,5°$, men må da akseptere at rundt 20% av mistankerapportene er falske positive. Vi mener 20% er til å leve med selv om dette tallet øker for mindre komplekse obliger. Denne verdien er betydelig lavere enn $0,8°$ som vi konkluderte var den riktige terskelverdien i forrige kapittel. Vi revurderer vår konklusjon og benytter denne verdien heretter.

For mindre kompliserte obliger kan det være hensiktsmessig å redusere terskelverdiene ytterligere. For obliger under en viss størrelse bør terskelverdien til vektorvinkelen justeres dynamisk for hver sammenligning som en funksjon av den minste obligens størrelse.

9.4.1 Omfanget av kopieringsproblemet

Dette har vært et tilbakevendende tema i denne oppgaven, men det er først nå vi har faktiske data å basere et estimat på. Våre tidligere antagelser om at rundt 30% av studentene var involvert i kopiering har vist seg å stemme ganske godt. Disse personene utgjør langt flere enn de få som hittil har blitt oppdaget. Vi håper bruken av dette sammenligningssystemet kan bidra til å redusere kopieringen ytterligere.

Kapittel 10

Systemet i drift

Vi har nå en prototype av systemet som fungerer og leverer brukbare resultater. Prototypen er allikevel ikke klar til å settes i drift på grunn av begrensninger i det valgte programmeringsspråket. I tillegg mangler vi et brukergrensesnitt.

I dette kapittelet vil vi ta for oss utviklingen av prototypen til et fungerende system som kan settes i drift. Vi vil gå gjennom de utfordringene vi møter i forbindelse med konvergeringen mellom de to komponentene som utgjør systemet og presentere en endelig definisjon av det inn/ut-grensesnittet algoritmesubsystemet tilbyr.

10.1 Innsnevring av problemområdet

På grunn av at problemområdet vi har definert er relativt omfattende har det vist seg hensiktsmessig å utskille konstruksjonen av brukergrensesnittet og tilhørende funksjonalitet og problemstillinger i et eget prosjekt. Brukergrensesnittet vil derfor bli utviklet separat i forbindelse med Joly-prosjektet [SV06], vi vil dermed ikke implementere denne funksjonaliteten, men i denne sammenheng begrense oss til å beskrive og implementere et javagrensesnitt som Joly kan bruke for å kalle på metodene i sammenligningssystemet.

10.2 Endringer fra prototype til fungerende system

Det var hovedsaklig to aspekter som måtte endres for å utvikle prototypen til et driftklart system. Svakhetene forbundet med et web-basert bru-

kergrensesnitt og PHPs nære forbindelse med web-presentasjon gjorde det nødvendig å oversette koden til et annet programmeringsspråk og systemet måtte utstyres med adekvate brukergrensesnitt.

10.2.1 Om valg av programmeringsspråk

I de foregående kapitlene diskuterte vi egnetheten av forskjellige programmeringsspråk og endte opp med å programmere prototypen i PHP. Bruken av PHP tilbød lett tilgjengelige metoder for sentrale kodeelementer som databasetilgang, regulære uttrykk og filbehandling. Dette gjorde det relativt lett å implementere sammenligningsalgoritmen, foreta eksperimenter og skrive ut testresultatene på en oversiktlig måte for tabulering. Det største problemet med dette språket var at vevtjeneren PHP kjører på terminerer kjøringen av programmet om det ikke avslutter innen en viss tid. Denne tidsbegrensningen kan forandres ved å endre på innstillingene i vevtjeneren, men dette er uansett ingen hensiktsmessig løsning.

Grunnen til at systemet brukte såpass lang tid på å kjøre at det støtte på vevtjenerens tidsbegrensning har også å gjøre med at PHP ikke er særlig effektivt til denne typen oppgaver, noe som i seg selv tilsier at vi burde bruke en annen plattform for systemet når det skulle settes i drift.

Selv om et vevbasert system fungerte godt i testfasen var det ustabilt nok til at vi bestemte oss for å gå helt vekk fra dette og i stedet implementere sammenligningssystemet som en javapakke. Java ble valgt da det har god støtte for databasefunksjoner, det er stabilt og effektivt. Java ble også valgt som implementasjonsspråk for brukergrensesnittet, så kompatibilitet var enda en viktig grunn til reimplementeringen. Se avsnitt 10.2.2.

10.2.2 Brukergrensesnittet

I avsnittene 4.2.1 og 4.2.2 beskrives to forslag til brukergrensesnitt for sammenligningssystemet. I løpet av arbeidet med denne oppgaven har to andre master-studenter utviklet et brukergrensesnitt i prosjektet Joly [SV06]. Sammenligningssystemet beskrevet så langt i denne oppgaven har blitt utstyrt med et inn/ut-grensesnitt for integrering med Joly-systemet og problemstillingene relatert til brukergrensesnitt faller dermed utenfor denne oppgaven.

10.3 Konvergering av to systemer

Da denne oppgaven fokuserer på utvikling av en algoritme og kriterier for sammenligningssystemet, mens Joly-systemet fokuserte på kommunikasjon med brukerne og relaterte temaer var det nødvendig å integrere de to delene så sømløst som mulig.

Joly-systemet bygger på en databasestruktur som er mer kompleks enn den som er beskrevet i tidligere kapitler og som er benyttet i prototypen. Det ble derfor nødvendig å migrere dataene fra den opprinnelige databasen til Joly-databasen. På grunn av forskjellige måter å indeksere dataene og forskjeller i normaliseringsnivået mellom basene lot det seg kun gjøre å flytte over programdata (programkode, brukernavn, størrelse, innleveringsdato et c.). Sammenligningsdata gikk tapt i migrasjonen på grunn av at definisjonene av primærnøklene var annerledes definert i destinasjonsdatabasen. Dette var ikke et kritisk problem siden sammenligningsdataene kan gjenskapes på bakgrunn av programdataene.

Forskjellen i normaliseringsnivå kompliserte også omprogrammeringen av prototypen siden SQL-spørringene måtte skrives om. Det ble i tillegg nødvendig med flere spørringer per sammenligning da datastrukturen i destinasjonsdatabasen var mer kompleks enn prototypen. Vi forholdt oss til de endrede forutsetningene under antagelsen om at den økte kompleksiteten var nødvendig for å skape et velfungerende brukergrensesnitt.

10.3.1 Kort beskrivelse av Joly-systemet

Joly [SV06] er et innleveringssystem for obligatoriske oppgaver. Systemet skal håndtere innlevering og lagring av besvarelser, interaksjon med brukere, tilby verktøy for oppgavevurdering og statistikk. Ved å gjøre kall mot sammenligningssystemet vil Joly også fungere som et brukergrensesnitt for vårt system.

10.3.2 Joly-systemets oppgaver

Omskrivingen av algoritmesubsystemet bygger på følgende forutsetninger om Joly-systemets oppgaver:

- Joly håndterer all kommunikasjon med brukere.
- Joly tar seg av lagring av innleverte programmer og brukerinformasjon.

- Joly registrerer et identifikasjonsnummer for hvert innlevert program.
- Joly gjør et kall mot algoritmesubsystemet for hvert program som skal analyseres. Se avsnitt 10.3.3.
- Joly henter sammenligningsdata fra et objekt generert av algoritmesubsystemet. Se avsnitt 10.3.3.

Databasen som benyttes både av Joly og algoritmesubsystemet til hhv. lagring av bruker- og programdata og sammenligningsdata er definert som beskrevet i figur 10.1.

Tabellene som refereres av algoritmesubsystemet er Reportcase, Studentsolution, Checkelement og Elementoccurrence. Primærnøklene til noen av disse tabellene er løpenumre som ikke bærer essensiell informasjon og om vi hadde definert databasen hadde vi gjort følgende endringer:

1. Reportcase ville hatt primærnøkkelen (studentsolutionID_1,studentsolutionID_2) i stedet for løpenummeret reportcaseID.
2. Checkelement ville hatt primærnøkkelen (checkelement) i stedet for løpenummeret checkelementID siden checkelement allerede er entydig.
3. Elementoccurrence ville hatt primærnøkkelen (checkelementID,studentsolutionID) som er entydig i stedet for løpenummeret elementoccurenceID som ikke er informasjonsbærende i likhet med resten av disse løpenumrene.

Utover å forenkle databasen ville disse endringene gjort programmeringen av algoritmesubsystemet litt enklere.

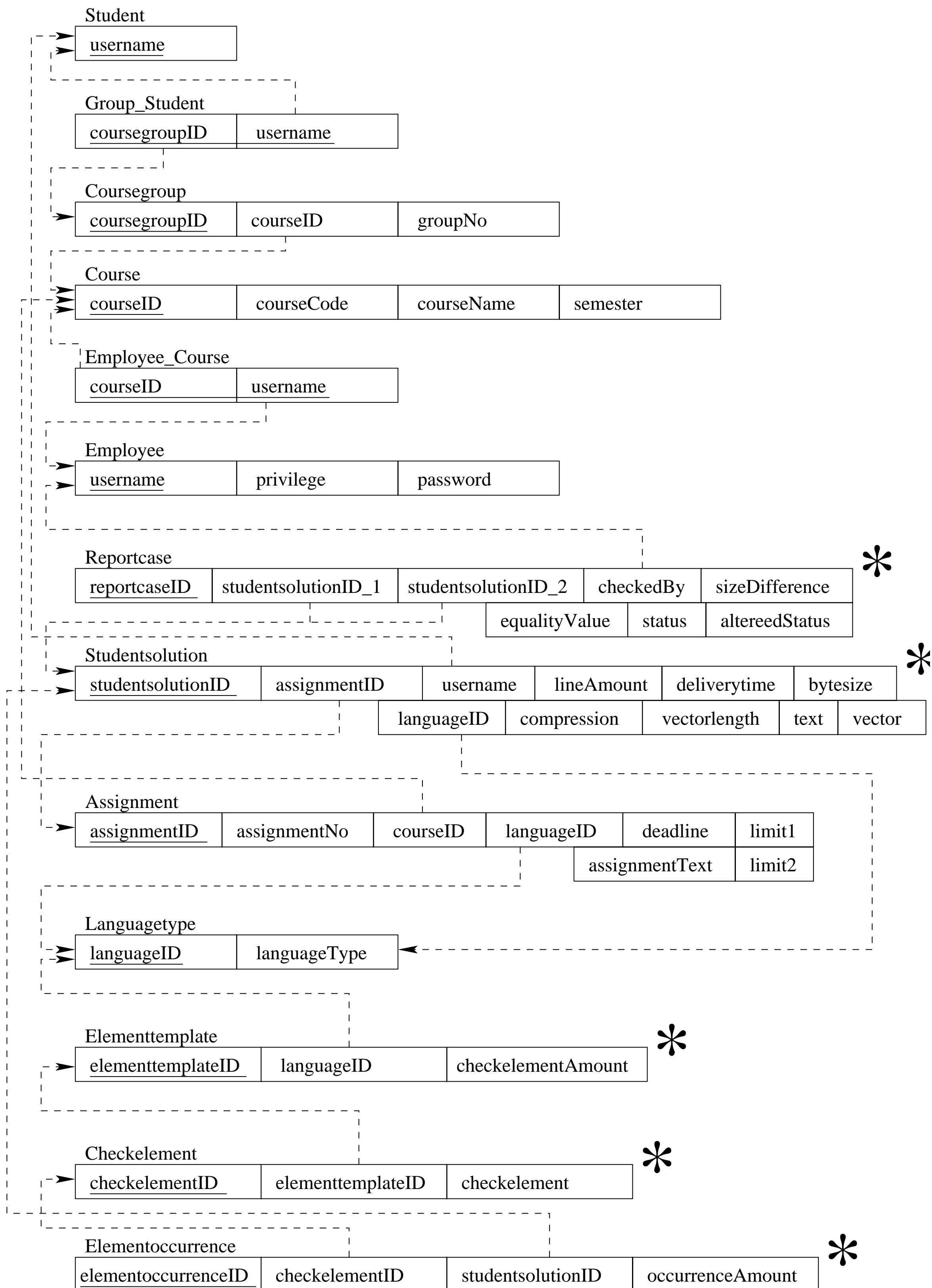
10.3.3 Algoritmesubsystemets grensesnitt

I dette avsnittet beskriver vi grensesnittet som tilbys av algoritmesubsystemet. Systemet består av tre klasser, JCLikhet, JCMistankerapport og JCParser. Joly befatter seg kun med de to første. Disse klassene er definert i de neste avsnittene.

Algoritmesubsystemets grensesnitt mot Joly utgjøres av metodene som er beskrevet i tabellene 10.2 og 10.1.

Følgende kall kan dermed gjøres fra Joly for å generere, lagre og hente alle mistankerapporter for program nummer 357:

```
Stack<JCMistankerapport> s=new JCLikhet(357,1).hentRapporter();
```



Figur 10.1: Definisjon av Joly-databasen.
* Markerer tabellene som brukes for lagring av sammenligningsinformasjon.

Gitt kallet ovenfor kan følgende kall gjøres for å hente vektorvinkelen til den øverste mistankerapporten i stacken:

```
double vvinkel=s.pop().hentLikhetsverdi();
```

Resten av de relevante verdiene er tilgjengelige via tilsvarende metoder.

Class JCMistankerapport

Objekter av denne klassen inneholder informasjon om likheten mellom to filer i tillegg til get-metoder for å hente ut informasjonen. Denne informasjonen er tilstrekkelig for å avgjøre graden av mistenkelighet i en sammenligning.

Field Summary

`double`	`likhetsverdi` Vinkelen mellom de to filenes N-dimensjonale vektorer
`int`	`program1` Databaseidentifikatoren til subjektfilen
`int`	`program2` Identifikatoren til filen som hentes fra databasen
`double`	`størrelsesforskjell` Størrelsesforholdet mellom det største og minste programmet

Method Summary

`int`	`hentSubjektprogram()` Returnerer dette programmets id-nummer.
`int`	`hentLagretprogram()` Returnerer id-nummeret til det andre programmet i sammenligningen.
`double`	`hentLikhetsverdi()` Returnerer likhetsverdien for sammenligningen.
`double`	`hentStørrelsesforskjell()` Returnerer størrelsesforskjellen.
`boolean`	`hentMistanke()` Returnerer enten mistenkelig eller ikke mistenkelig.
`void`	`print()` Skriver informasjonen fra mistankerapporten til `stdout`.

Tabell 10.1: Dokumentasjon av sentrale elementer i klassen JCMistankerapport

Class JCLikhet

JCLikhet er den sentrale klassen i algoritmesubsystemet. Ved å opprette et objekt av denne klassen hentes en kodefil fra databasen. Filen blir renset for

kommentarer, faste strenger og blanke tegn og deretter begynner likhetsanalysen. Til slutt genereres et antall mistankerapporter (se avsnitt 10.3.3) som gjøres tilgjengelig i JCLikhet-objektet.

Field Summary

`java.util.Stack` `<JCMistankerapport>`	`mrapp` inneholder alle genererte mistankerapporter for kodefilen representert ved dette JCLikhet-objektet.

Constructor Summary

`JCLikhet(int inf, int vs)`	Oppretter et JCLikhet-objekt for en fil som skal analyseres.
	`int inf`
	nummeret programfilen er lagret med i databasen.
	`int vs`
	nummeret på symbolsettet som skal brukes i analysen.

Method Summary

`java.util.Stack` `<JCMistankerapport>`	`hentRapporter()` Returnerer stacken med mistankerapporter for programmet som analyseres av dette objektet. Returverdi `Stack<JCMistankerapport>`

Tabell 10.2: Dokumentasjon av sentrale elementer i klassen JCLikhet

10.3.4 Nødvendige endringer i algoritmesubsystemet

Siden denne oppgaven handler om algoritmedelen av sammenligningssystemet vil vi kun diskutere Joly-delen der den har å gjøre med algoritmedelen. Joly tilbyr en bruker oversikt over hvilke lagrede programmer som ligner på et gitt innlevert program. Den informasjonen algoritmesubsystemet må gi til Joly må dermed inkludere de følgende elementene for hvert programpar under sammenligning:

1. Identifikasjonsnumrene for programmene.
2. Vektorvinkelen for de to programmene.
3. Størrelsesforskjellen mellom programmene.
4. Binær mistankevurdering.

Informasjonen i punktene ovenfor inkluderes som verdier i et objekt av klassen JCMistankerapport. Siden et gitt program ofte vil ligne på flere andre programmer vil algoritmesubsystemet generere flere mistankerapporter

hver gang det kalles. Algoritmesubsystemets grensesnitt mot Joly vil gjøre tilgjengelig et Stack-objekt som inneholder alle JCMistankerapportobjektene sortert etter likhetsverdi, med den laveste likhetsverdien (som indikerer størst sannsynlighet for likhet) øverst i Stacken. Joly kan dermed presentere mistankerapportene for brukeren på en hensiktsmessig måte.

10.4 Avsluttende bemerkninger

På grunn av at vi ikke har tilgang til Joly-systemet har den endelige integrasjonen ennå ikke funnet sted. Vi tror likevel at algoritmesubsystemets grensesnitt er så enkelt og greit at integrasjon mot et hvert brukergrensesnitt som kan gjøre bruk av den informasjonen algoritmesubsystemet tilbyr vil være trivielt.

10.5 Oppsummering

Med utgangspunkt i prototypen på et sammenligningssystem beskrevet og brukt i tidligere kapitler har vi utført modifikasjoner med henblikk på å integrere systemet mot en brukergrensesnittkomponent. Siden programmering av brukergrensesnittet faller utenfor denne oppgavens fokus og Joly-systemet foreløpig er utilgjengelig har vi utviklet et svært enkelt, men tilstrekkelig, inn/ut-grensesnitt som gjør all relevant informasjon fra algoritmesubsystemet tilgjengelig via metodekall.

Gitt at Joly-systemet er operativt er det kun behov for noen få ekstra kodelinjer for å integrere de to komponentene.

Kapittel 11

Oppsummering og tanker om videre arbeid

Vi sammenfatter de konklusjoner og resultater vi har kommet frem til i løpet av arbeidet med oppgaven og presenterer mulige forbedringer, alternative implementasjonsmetoder og alternative retninger for videre utbygning av systemet.

11.1 Sammenfatning av arbeidet

Vi har utført flere forskjellige undersøkelser i forbindelse med denne oppgaven. Hovedhensikten var å lage et system for sammenligning av obligatoriske oppgaver for bruk ved Institutt for informatikk.

Utviklingen av systemet foregikk i flere faser som kan oppsummeres som følger:

Beskrivelse av systemet. Før vi startet arbeidet med programmering og undersøkelser laget vi en utførlig beskrivelse av sammenligningssystemet, inkludert et forslag til brukergrensesnitt.

Konstruksjon av prototype. Dette arbeidet besto i å lage en database for lagring av grunnlagsdata, programmering av en koderenser og symbolteller. Prototypen ble skrittvis utvidet med sammenligningsalgoritmen i forbindelse med de påfølgende fasene. Den første versjonen av prototypen ble skrevet i Perl, men raskt oversatt til PHP.

Valg av metode. Vi vurderte flere forskjellige sammenligningsmetoder etter kriterier for bl. a. nøyaktighet og effektivitet og valgte den vi fant mest lovende. Denne metoden er basert på telling av forskjellige symboler i hvert

program, utregning av en flerdimensjonal vektor på bakgrunn av symbolantallene og vurdering av vektorene til programpar av sammenlignbar størrelse.

Valg av symbolsett. Kvaliteten på symbolsettet er en av de viktigste faktorene for metodens nøyaktighet, det var derfor nødvendig med grundig testing før vi valgte et symbolsett for de videre undersøkelsene. Vi vurderte flere forskjellige symboler og utførte innledende sammenligningstester med tre forskjellige symbolsett. På bakgrunn av disse testene valgte vi ut de 15 mest lovende symbolene. Dette settet av 15 symboler ble deretter redusert til 6 symboler gjennom utstrakt testing ved "steepest hill"-algoritmen.

Valg av metode for vektorsammenligning. Vi vurderte to forskjellige måter for sammenligning av vektorene til programpar. Utregning av den euklidske avstanden mellom endepunktene eller vinkelen mellom vektorene. Vinkelen viste seg å gi større nøyaktighet.

Karakterprofiler for kopierende studenter. På bakgrunn av eksamensresultater og obligvurderinger fra gruppelærere var det mulig å få resultater om hvilke karakterer som var vanligst blant studenter involvert i kopiering. Selv om det avtegnet seg forskjeller mellom de forskjellige karaktergruppene var resultatene for dårlige til å kunne si noe sikkert om hvem som kopierer. Derimot ga resultatene oss et godt utgangspunkt for å estimere terskelverdier for mistankevurderingen.

Fastsetting av terskelverdier. For å kunne kategorisere et programpar som mistenkelig likt eller ikke er det nødvendig å vurdere vinkelen mot en terskelverdi. Vi utførte tester ved å sammenligne et stort antall obligbesvarelser mot hverandre og undersøke et betydelig utvalg av de genererte mistankerapportene. Dataene fra disse undersøkelsene gjorde oss i stand til å fastsette endelige terskelverdier for optimal nøyaktighet i sammenligningene.

Sammenbygging av to systemer. I de siste fasene av arbeidet ble konstruksjonen av et brukergrensesnitt overtatt av andre studenter som laget systemet Joly. Dermed ble det nødvendig å programmere om systemet som en modul med grensesnitt mot en uavhengig GUI-komponent. Dette gjorde det nødvendig med enda en oversettelse av programmet, denne gangen til Java.

11.2 Konklusjoner

Vi har behandlet flere temaer i denne oppgaven. Vi har utviklet en algoritme for sammenligning av javakodefiler, vi har implementert denne algoritmen i et sammenligningssystem, vi har undersøkt omfanget av kopiering

av obligbesvarelser i begynnerkurset i informatikk, vi har beskrevet utfordringene ved å implementere en sammenligningsalgoritme samt utfordringene ved konvergering av to programkomponenter til et fungerende system.

11.2.1 Har vi nådd våre målsetninger?

I de første kapitlene beskrev vi noen av forutsetningene og kravene vi la til grunn for arbeidet med systemet. Her redegjør vi for hvordan disse forutsetningene og kravene er blitt møtt.

Oppfyller systemet de nødvendige krav?

I avsnitt 2.3 i den innledende oppgavebeskrivelsen angir vi de sentrale kravene systemet bør oppfylle. Her ser vi på hvert av disse kravene og hvordan systemet møter dem. På grunn av at systemet ble besluttet utviklet som to separate komponenter underveis i prosessen har noen av de foreslåtte kravene mistet mesteparten av sin relevans i forhold til algoritmesubsystemet.

Nøyaktighet De avsluttende testene beskrevet i kapittel 9 indikerer at 80% av de likhetstilfellene systemet rapporterer er reelle tilfeller av kopiering. De resterende 20% er falske positive. Siden alle rapporter må undersøkes av en gruppelærer bør det ikke forekomme for mange falske positive da dette kan redusere brukernes opplevelse av systemets nytteverdi. Vi tror én av fem grunnløse rapporter er trygt innenfor brukernes toleransegrense. Systemet vil en sjelden gang unnlate å rapportere reelle kopieringstilfeller, men dette kan ikke sies å redusere nøyaktigheten nevneverdig, det vil heller ikke påvirke brukernes opplevelse av systemets nytteverdi.

Pålitelighet Systemets pålitelighet, definert i avsnitt 2.3 er ivaretatt i algoritmesubsystemet så langt det er mulig. Svakheter i databaseprogramvaren eller Javas grensesnitt mot databasen kan påvirke påliteligheten, men vi tror det er usannsynlig. Sannsynligheten for redusert pålitelighet i Joly eller i forbindelse med Joly's kall mot algoritmesubsystemet har vi ikke tatt hensyn til. Algoritmesubsystemets pålitelighet er adekvat gitt at systemets grensesnitt blir benyttet. Ukorrekt bruk av grensesnittmetodene skal ikke være mulig utover bruk av ukorrekte parameterverdier.

Effektivitet De operasjonene som gjør størst utslag på tidsbruken er databaseoperasjoner (spørring, skriving og oppdatering). Systemet er

implementert med hensyn på å redusere antallet databaseoperasjoner til et minimum. Effektiviteten er tilfredsstillende med ∼1 200 lagrede innleveringer.

Robusthet Om vi ser bort fra databasen og brukergrensesnittet vil algoritmesubsystemet måtte håndtere et lineært økende antall grunnleggende regneoperasjoner for et økende antall lagrede innleveringer. Det skulle dermed skalere greit. Resten av punktene under robusthet er ikke lenger relevante da de må håndteres i brukergrensesnittkomponenten.

Brukbarhet Dette er nesten utelukkende et brukergrensesnittspørsmål. Algoritmesubsystemet ivaretar brukbarhet i et enklest mulig grensesnitt mot Joly-komponenten.

Sikkerhet Tilgang til systemet er nå håndtert av Joly. Virusinfisering av systemet bør hindres og håndteres på brukergrensesnittnivå om vi antar at den primære kilden til virusangrep er epostinnleveringer. Dette gjelder også for spamproblematikk.

I kapittel 4 beskriver vi et første utkast til brukergrensesnitt for å ha et grunnlag å forholde oss til i arbeidet med algoritmesubsystemet. Dette utkastet ble satt til side etter at arbeidet med Joly ble startet. De opprinnelige forutsetningene viste seg allikevel å være nyttige og dannet grunnlaget for algoritmesubsystemets grensesnitt mot Joly. Flere av spørsmålsstillingene i dette kapittelet er blitt videre utredet i [SV06].

I det hele tatt mener vi systemet oppfyller de kravene som opprinnelig ble stilt. Systemkravene fokuserer på avsløring av kopiering, noe vi mener å ha vist at systemet utfører effektivt.

Vil systemet bidra til å redusere kopiering av obliger?

Dette spørsmålet er et av de mest interessante å finne svar på siden det oppsummerer selve hensikten med dette prosjektet. Dessverre faller det ikke inn under denne oppgaven. Vi har foreslått en metode og laget et programsystem for å oppdage kopiering, men hvorvidt systemet vårt faktisk bidrar til å redusere antallet kopieringstilfeller gjenstår å se.

Dette kan være et interessant tema for undersøkelse etter at systemet har vært i drift ett eller to semestre og vi håper og tror svaret vil være positivt.

Denne evalueringen av systemet kan også inkludere brukernes opplevelse. Mener gruppelærerne at systemet hjelper dem å rette obliger raskere og bedre? Ser faglærerne noen effekt av redusert kopiering? Frykter studentene systemet så sterkt at de jobber hardere med stoffet?

Er systemet blitt satt i drift?

Systemet er, i det dette skrives, i ferd med å ferdigstilles og er planlagt satt i drift til høstsemesteret 2006. Algoritmesubsystemet som er beskrevet i denne oppgaven vil integreres med et brukergrensesnitt som kan tilfredsstille de krav som er beskrevet, det tidligere nevnte Joly-systemet.

Har vi bidratt til innsikt på andre områder?

I forbindelse med utviklingen av systemet har vi presentert noen kriterier for hva som karakteriserer likhet mellom obliger. Disse kriteriene har kommet som et resultat av undersøkelsene vi har gjort, men kanskje kan de også være til nytte i andre sammenhenger.

Kan kriteriene være til nytte for gruppelærere? Bør det eksistere felles retningslinjer for retting av obliger? Og bør de sammenfalle med våre kriterier? Dette er spørsmål som kan være interessante å stille i en eventuell videreføring av temaene fra denne oppgaven.

11.2.2 Kartlegging av kopieringsproblemet

Vi har ut fra en systematisk undersøkelse av de innleverte oppgavene i INF1000 estimert omfanget av kopieringsvirksomheten blant studentene på dette kurset. Vi anslår at rundt 30% av studentene er involvert i kopieringsvirksomhet, dette bør reduseres både av hensyn til studentene selv og troverdigheten til kurset.

11.3 Videre arbeid

Det er mange veier å føre dette arbeidet videre og vi har allerede vært inne på noen av dem. Den mest nærliggende oppgaven er å fullføre arbeidet med å sette systemet i drift, men også utarbeiding av ny funksjonalitet etter hvert som behov eventuelt måtte oppstå i praktisk bruk.

Algoritmen vi har brukt er basert på nøkkelordtelling, men selv om den er effektiv er den ikke nødvendigvis optimal. I kapittel 3 nevner vi flere andre metoder for likhetspåvisning, blant annet sammenligning av funksjonskalltrær og blokknøsting. En videre undersøking av disse alternative metodene for likhetsvurdering kan være av interesse.

En annen interessant mulighet vi dessverre ikke hadde mulighet til å forfølge i denne oppgaven er påvisning av karakteristiske kodestiler. Vi har en

hypotese om at programmer av forskjellig størrelse som har en liten vektorvinkel er skrevet med sammenlignbar kodestil. Vil vektorene til to forskjellige programmer skrevet av samme programmerer peke i samme retning? Hvis så er tilfelle, er det mulig å bruke dette til å gjenkjenne kodestilen til en programmerer om du allerede har noen av hans andre programmer? Dette kan være interessant å undersøke.

Som vi nevnte ovenfor kan det være nyttig å gjøre undersøkelser om utbredelsen av kopiering etter at dette systemet har vært i bruk noen semestre. Blir systemet brukt i det hele tatt? Gir det tilfredsstillende resultater?

Kan prinsippene som er beskrevet i denne oppgaven benyttes på andre områder? Det er fra tid til annen snakk om patentering av programmeringskonsepter og kodesnutter. Om vi ser bort fra de åpenbare problemstillingene rundt "fair use", kan algoritmen beskrevet i oppgaven være til nytte på dette feltet?

Det er i det hele tatt nok av veier å følge videre fra dette prosjektet, og nye master-studenter vil helt sikkert komme på flere idéer enn de vi har skissert her. Vi overlater herved fakkelen til dem.

Bibliografi

[BR04] Ruth Barrett og Austen Rainer. "With a Little Help from My Friends...". *Higher Education Academy, Subject Centre for Information and Computer Science*, 2004.

[CM93] Pádraig Cunningham og Alexander N. Mikoyan. Using CBR Techniques to Detect Plagiarism in Computing Assignments. Rapport, Trinity College, Dublin, 1993.

[GH89] Dick Grune og Matty Huntjens. Detecting Copied Submissions in Computer Science Workshops. Rapport, Vrije Universiteit De Boelelaan, Amsterdam, 1989.

[HF04] Mike Hart og Tim Friesner. Plagiarism and Poor Academic Practise – A Threat to the Extension of e-Learning in Higher Education? *Electronic Journal on e-Learning, Volume 2, Issue 1*, side 89–96, 2004.

[Rob02] Eric Roberts. Strategies for Promoting Academic Integrity in CS Courses. *32nd ASEE/IEEE Frontiers in Education Conference, Session F3G*, side 14–19, 2002.

[SV06] Therese Steensen og Hanne Vibekk. DHIS and Joly: Two Distributed Systems under Development: Design and Technology. Hovedfagsoppgave, Universitetet i Oslo, 2006.

[Wag00] Neal R. Wagner. Plagiarism by Student Programmers. *http://www.cs.utsa.edu/ wagner/pubs/plagiarism0.html*, 2000.

[Wik06] Wikipedia. Hill climbing. `http://en.wikipedia.org/wiki/Hill_climbing`, 2006.

[Wis92] Michael J. Wise. Detection of Similarities in Student Programs: YAP'ing May Be Preferable to Plague'ing. *SIGCSE'92, Kansas City, Missouri, USA*, side 268–271, 1992.

[Wis96] Michael J. Wise. YAP3: Improved Detection of Similarities in Computer Programs and Other Texts. *SIGCSE'96, Philadelphia, USA*, side 130–134, 1996.

[Zha99] Weigang Zhao. A study of web-based application architecture and performance measurements. *AusWeb99, Fifth Australian World Wide Web Conference*, 1999.