UMEÅ UNIVERSITET
Department of Computer Science
Directed Acyclic Graph

June 10, 2022

Directed Acyclic Graph

Programspråk

version 2.0

Name Mathias Hallberg, Gustaf Söderlund
Username c19mhg, et14gsd

Graders
Henrik Björklund, Oscar Kamf

Contents

1 Ändringar

- Rapporten är renskriven.
- Förbättrat polymorphism med att implementera typklasser och instanser.
- Beskrivning till Algoritm finns nu med i rapporten.
- Ett försök till refaktorisering. Istället för att ha allt under en Where-clause har vi delat upp det i funktioner.
- Ändrat namn på Python-filerna, för att köra Haskell testerna har vi alltid skrivit in *:s Script.hs* och det har funkat bra för oss. Vet ej om det finns något bättre sätt men det funkar. Vi tolkade det även som att när man kunde göra en större graf för att få med edge-cases var bara en rekommendation och något att lägga i åtanke till nästa gång.

2 Introduktion

I denna laboration har en riktad acyklisk graf implementerats med hjälp av det funktionella programspråket Haskell, samt ett annat programmeringsspråk som inte är av den funktionella typen. I laborationen har man valt att använda Python där man kommer utnyttja en objekt-orienterad lösning.

Inom implementation av grafen ska man kunna lägga till noder, kanter, få ut en topologisk ordning av grafen och till sist få ut den längsta vägen från nod *A* till nod *B*. Lösningen ska även vara polymorfisk, det vill säga att man ska kunna använda sig utav av mer än en datatyp i lösningen. Där man har valt att använda heltal och karaktärer för att representera vikterna av kanterna samt noderna.

3 Beskrivning av DAG

I denna sektion kommer en beskrivning av den riktade acykliska grafen.

3.1 Gränsyta

Gränsytan är implementerad i två programmeringsspråk från vardera programmeringsparadigm där funktionerna ser annorlunda ut, men har samma funktionalitet. För att kunna skapa en graf, få ut den topologiska ordern samt den längsta vägen från nod A till nod B. Har implementationen i det funktionella språket, haskell gjorts på ett sätt så att man behöver skicka in den skapade grafen i ett funktionsanrop. I t.ex. *add_ vertex()* kallas det som *add_ vertex(DAG)*, där *DAG* representerar grafen. I Python som använder en objekt-orienterad lösning kallar man samma funktion genom att göra ett följande funktionsanrop *DAG.add_ vertex()*.

I *add_ vertex()* tar man in en vikt och därefter används ett ID, i form av heltal som genereras av programmet. Det ID som har genererats returneras för att användas vid ett senare tillfälle.

```
v = add_vertex(w)
```

I *add_ edge()* kommer en kant genereras mellan två noder. Indatat till funktionen är nod *a*, nod *b* samt en vikt *w*. Vikten *w* representerar vikten på den kant som sätts mellan noderna. Kanterna mellan noderna har en riktning. Funktionen misslyckas om kanten skapar en cykel.

```
add_edge(a,b,w)
```

Topological order får ut en topologisk ordning från en graf. Det innebär att man får ut en lista där en ordning från vänster till höger med alla noders ID listas. Det ID längst till vänster i listan är den noden som inte har någon inkommande nod. Därefter tas den noden bort helt från grafen och man börjar om. Om vägarna mellan noderna ses som uppgifter måste det ske i en viss ordning och topological ordering hittar en väg som är möjlig att ta.

```
vlist = topological_ordering()
```

Funktionen *weight_ of_ longest_ path()* returnerar vikten av den längsta vägen från en startnod a till slutnoden b. Indatat f och g är två funktioner som bestämmer vikten av vägen.

```
w = weight_of_longest_path(a,b,f,g)
```

3.2 beskrivningen av implementation

I denna sektion kommer en beskrivning av implementationen samt algoritmerna som har använts i programmet.

3.2.1 Kahn's Algorithm

Kahn's Algoritm är en algoritm man har använt sig utav för att räkna ut topological order samt efterliknat den för att få ut den längsta vägen i grafen. Första steget i algoritmen är att kolla hur många ingående vägar alla noder har i en graf. Det vill säga att om man går från nod 1 till nod 2 kommer nod 2 att ha totalt en ingående väg. Detta sparas undan då man enbart i början vill ha noderna som har 0 ingående vägar. Detta då dom kommer hamna först i listan av en topological order. Man sparar då undan noderna som har noll stycken ingående för att repetera det föregående steget med en ny graf som inte innehåller det undansparade värdena. När man har den nya grafen utan dom som föregående hade noll vägar in i sig kommer det ha tillkommit en eller flera noder med noll stycken ingående vägar. Detta repeterar man tills att processen är klar. Man kan även på detta vis testa om grafen är riktad och acyklisk då detta kommer att kunna genomföras och inte krascha.

Ungefär samma princip sker när man kollar det längsta vägen av från en nod till en annan. Man tar in noderna och kollar från topologiska ordningen från startnoden, där måste man hantera om det kan uppstå att det finns en längre väg som inte startnoden går genom till höger i listan av den topologiska ordningen. Där kollar man genom alla noder som går från startnoden till slutnoden och sparar undan alla vägar medan man tar bort alla andra noder. När alla vägar som man möjligtvis kan gå börjar man kolla på vikterna av dom. Man börjar med startnoden och skapar en lista där adderar man ihop vikten det tog för att komma till nästa nod. Skulle det komma någon annan väg som kan vara längre överskrids den gamla och om den inte är längre låter man den vara. Detta sker

med samma princip från när man räknade ut topological order då man tar bort noder ur listan som är i en topologisk order och modifierar grafen som man räknar på. På detta sätt kommer man få ut den längsta vägen associerad med slutnoden som kan skrivas ut till användaren.

Samma princip görs i både Python och Haskell förutom att all listhantering och andra diverse saker är allmänt enklare att förstå sig på i Python. Den enda saken är att i Python sparar man inte värdet av längsta vägen utan allt sparas som en lista som summeras i slutet.

3.2.2 Encapsulation

För att uppnå encapsulation i Haskell har programmet implementerats i en *module*. Datatyperna blir inte grupperad i ett "objekt", istället är de grupperad i samma module. Där exporterar man alla funktioner som är nödvändiga för att kunna använda sig i programmet. Det vill säga att alla de fyra funktionerna för att bygga upp och få ut begärda resultat finns med. Alla hjälpfunktioner som dom fyra funktionerna använder sig av blir då på så sätt gömda från användaren. Detta kan man kolla genom att skriva *:browse* när man har laddat in programmet. Resultatet av det blir att man bara får upp det fyra funktionerna med deras signatur.

En alternativ lösning till detta hade kunnat varit att bygga programmet med flera olika filer med vardera modul, men på grund av saknad erfarenhet valde man att arbeta i en modul i en fil istället. Anledningen till detta är då man inte behöver importera egenskrivna moduler. Det blir enklare och flyter på när man själv skriver att ha allt i ett dokument utan att behöva hålla koll på andra filer när man vill se vart det har gått fel.

I python uppnås encapsulation genom att varje datatyp och dess metoder är bundlade i en egen klass. På grund av mer erfarenhet i objekt-orienterad programmering valde man att dela upp programmet i flera olika klasser. Genom att använda klasser för att uppnå encapsulation i python behöver användaren inte veta hur klassen är implementerad eller lagrad. Användaren behöver bara veta vad som skickas in i klassen och vad man får tillbaka. En alternativ lösning till encapsulation i python hade varit att göra allt i en och samma fil och skapa flera klasser, men det känns inte naturligt att göra när man arbetar med objekt-orienterad programmering därför valde man att arbeta i flera klasser.

3.2.3 Information hiding

För att uppnå information hiding i haskell har man datatyperna DAG, Edge och Vertex. Detta är dom enda datatyperna som användaren behöver veta existerar. Detta exporteras från modulen så att man kan använda och kolla på hur dem ser ut, allt annat man manipulerar i de olika värden och identifierare behöver man inte veta om, då det kommer få rätt resultat om programmet används korrekt. När man till exempel räknar ut den topologiska ordern jobbar man mycket med

listor och då behöver inte användaren veta att man gör det utan bara att det behövs skickas in en graf så kommer den att få ut en lista över nodernas ID i en lista som är topologisk.

Anledningen till varför man valde att arbeta på det här sättet var av den anledningen att vid tidigare laborationer har samma strategi använts. Vilket är bra då man har en viss kunskap om hur det fungerar och det blir enklare att göra framsteg i koden. Nackdelen är att man inte testar på andra strategier för att utvecklas som programmerare.

Information hiding uppnås i Python programmet genom att varje klass initieringsmetod använder dubbel understreck vilket ska göra metoden och dess attribut i objektet privat/otillgänglig från andra klasser. På så sätt "göms" data för användaren. Varför man valde att arbeta med information hiding på det här sättet var av den anledningen att det var den mest rekommenderade strategin på diverse inspirationskällor. Man sökte också efter alternativa strategier till att arbeta med information hiding på, men dessvärre var detta den enda strategin man kom över.

3.2.4 Polymorphism

I programmet har det bestämts att de två datatyperna som går att använda är heltal och karaktärer för att uppnå Polymorphism.

För att uppnå polymorphism i Haskell har en typklass implementerats där +, - och sum operatorerna/funktionerna definierats. Varje gång dessa operatorer/metoder används i programmet skapas en ny instans beroende på vilken datatyp som används, det vill säga heltal eller karaktärer. Anledningen till varför man valde att göra på det här sättet är för att det get större möjlighet att utveckla programmet i de fall man behöver lägga till fler datatyper, dessutom är det mycket enklare då man endast behöver lägga till ytterligare kod på en plats i programmet. En alternativ lösningen hade varit att försöka konvertera allt till heltal. Man hanterar karaktärer som deras ASCII värde för att jämföra om vilken väg som blir längst. Detta är givetvis inte ett bra sätt om man vill utveckla vidare på det då vissa typer kan bli krångliga att försöka göra om till heltal.

I Python använder man inte datatyper på sammasätt, en variabel blir den datatyp den blir tilldelad. Vill man använda integers anger man att vikten ska vara integer och samma sak med characters. I programmet fungerar detta bra, det finns en funktion där man behöver kontrollera vilken datatypen är för att kunna göra en beräkning.

En alternativ strategi som man kan använda i Python som diskuterats är att konvertera om allt till strängar och bara arbeta med strängar. Problemet med detta är att det har utförts i en tidigare labb i en annan kurs och de problem som uppstod vid det labbtillfället var att det blev en heldel konverteringar från strängar till andra datatyper vid just uträkning vilket kunde bli fel oftast.

3.2.5 Higher order functions

Higher order funktionerna är implementerade på så sätt att man kommer ta in en två separata funktioner där ena kommer att manipulera nodens vikt och den andra kommer att ändra kanternas vikt. När man kallar på funktionen som får ut den längsta vägen har man det två olika funktionerna som inparametrar. För att skicka in en funktion i Haskell kan man använda sig av en anonym funktion där till exempel om man vill addera 2 på vikten av kanter kommer man skicka in $(\backslash g- > g+2)$. I Python är det lämpligast att använda en lambda funktion som ser ut på följande sätt *lambda g: g + 2* där funktionen hade adderat 2 i detta fall.

När man tar in funktionerna kommer man att kontrollera och addera de olika funktionerna. Higher order funktionerna kommer då att fungera som helt vanliga funktioner för vikterna förutom att dom skickas in för att användas senare.

Ett annat alternativ är att man ändrar till en speciell signatur till en inbyggd funktion, det vill säga att man till exempel skickar in *(+3)* när man ska addera tre till vikterna. På detta vis kan det vara enklare att förstå vad man ska skicka in och vad man kommer få för resultat. Det kan dock leda till att funktionerna blir väldigt grundläggande då dom kommer bli en flaskhals av hur man hanterar funktionerna, vill man ha en mer avancerad funktion in i programmet kanske det inte funkar. Det blir då bättre att man får skriva sina egna lambda funktioner där man får leka så mycket man vill. Detta blir då principen som gäller för både Python och Haskell.

4 Tester

För att verifiera att implementationen fungerar har under implementationsfasen testning gjorts vid konstruktion av varje funktion. I denna sektion tas utförandet av testningen för de två programspråken upp.

Under konstruktion av varje funktion för de två programmen utfördes tester på dess funktionalitet. Hypotetiskt utdata som man förväntade att få tillbaka efter en körning skapades och konstruerade eget indata som man matade funktionerna med. Om hypoteserna stämde överens med det faktiska utdatat som returnerades från från funktionerna, förklarades funktionen som körbar.

Många gånger ritades skisser på en DAG och räknade ut för hand slutresultatet. I Figur 4 åskådliggörs ett exempel på en graf som används vid testning.

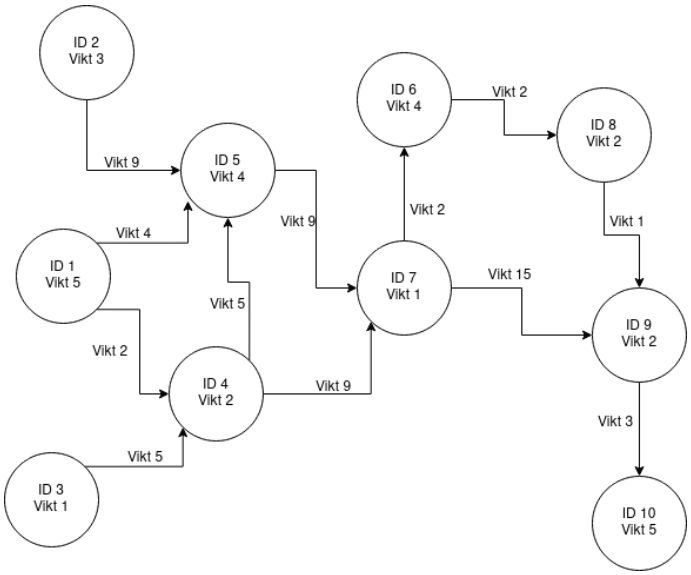

Figure 1: En av de grafer som använts vid testning.

I grafen som representeras av figur 4 presenteras en rad olika tester som utförts. Nedanför presenteras tester på en topologisk order samt längsta vägen mellan en rad olika noder.

```
topological_order dag11
weight_of_longest_path 1 10 dag11 (\f -> f) (\g -> g)
weight_of_longest_path 5 9 dag11 (\f -> f) (\g -> g)
weight_of_longest_path 1 10 dag11 (\f -> f + 6) (\g -> g - 3)
```

Dag11 är graden och man kollar från nod 1 till 10, 5 till 9 samt skickar in higher order funktioner för att testa om det fungerar. Resultatet blir då följande.

```
[1,2,3,4,5,7,6,8,9,10]
[5,2,2,5,4,9,1,15,2,3,5]
[4,9,1,15,2]
[11,-1,8,2,10,6,7,-1,10,-1,8,-2,8,0,11]
```

Där man kan se att den topologiska ordern stämmer, samt att de längsta vägarna följer den väg som faktiskt är längst. Man kan även se att det funkar när man skickar in higher order funktioner då den ändrar den längsta vägen till den som faktiskt är längst.

Här testas även att det fungerar med polymorphism där man använder karaktärer istället för heltal som vikter för noder och kanter. För enkelhetensskull testades fallet om något hade en vikt 1 så fick den karaktären 'a', 2 fick 'b' och så vidare. Testerna som gjordes beskrivs nedanför

```
topological_order dag11a
weight_of_longest_path 1 10 dag11a (\f -> f) (\g -> g)
weight_of_longest_path 1 10 dag11a (\f -> f + 2) (\g -> g + 6)
```

Resultatet av testerna är följande.

```
[1,2,3,4,5,7,6,8,9,10]
"ebbediabdbbabce"
"ghdkfochfhdgdig"
```

Man kan se utifrån resultatet att vägarna följer testresultaten med heltal och därmed är implementerad på ett korrekt sätt. Topological ordern kommer att vara i heltal då man beslutat över att alla ID ska vara det men man kan även se att det stämmer.

Det finns ett flertal situationer som kan uppstå i en DAG som behövdes testas.

I en DAG får grafen inte vara cyklisk, det vill säga att det går att gå tillbaka till en nod som den redan besökt. Testningen för detta utfördes genom att konstruera en graf med beteendet då man kan vandra tillbaka till en nod man redan besökt. Om man skulle testa att lägga till en kant mellan 6 till 2 i Figur 4 som medvetet skulle göra den cyklisk vill man att användaren prompt ska få reda på att den skulle bli det. Om man skulle testa det med följande exempel.

```
add_edge (fst v6) (fst v2) 1 dag11
```

Skulle detta synas i terminalen.

```
*** Exception: Tried to enter edge which makes DAG cyclic
```

Vilket stämmer bra överens med vad som man förutspått.

Ett annat problem som kan uppstå är om man försöker kolla längsta vägen på en väg som inte existerar. Om man försöker kolla längsta vägen från nod 1 till nod 2 vill man att användaren även där ska få veta att ett problem har uppstått. Följande testas.

```
weight_of_longest_path 1 2 dag11a (\f -> f) (\g -> g)
```

Och får ut följande resultat.

```
*** Exception: ERROR: tried getting longest path on nonexisting
↪  path
```

Vilket stämmer överens bra med vad man ville skulle hända

En annan situation som kan uppstå i en dag som man vill undvika är när man ska räkna ut längsta vägen från en startnod till en slutnod, men att startnoden har en eller flera ingående kanter. En graf med detta beteende åskådliggörs i figur 4 Det man vill undvika i denna graf är att beräkningen som görs i programmet

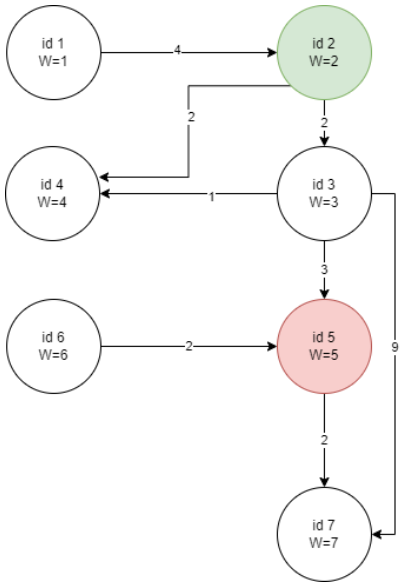

Figure 2: Räkna ut längsta vägen från en startnod (grönt) till en slutnod (rött). Startnoden har en ingående kant.

räknar fel, t.ex. att beräkningen görs från noden med ID 1 istället för ID 2.

Här nedan har testas topologisk order samt längsta vägen mellan en mängd noder i python.

```
topological_order_list = dag.topological_ordering()

f = lambda int : int
g = lambda int : int
longest_path = dag.weight_of_longest_path(v1,v10,f,g)

f = lambda int : int
g = lambda int : int
longest_path = dag.weight_of_longest_path(v5,v9,f,g)

f = lambda int : int+6
g = lambda int : int-3
longest_path = dag.weight_of_longest_path(v1,v10,f,g)
```

I funktionsanropen ovan kan man se *dag* som är ett DAG-objekt. En kontroll om topological order stämmer, samt om *weight_ of_ longest_ path()* fungerar

från nod 1 till 10, 5 till 9 och om man använder higher order funktioner från nod 1 till nod 10. Följande output ges utifrån dessa tester.

```
[1,2,3,4,5,7,6,8,9,10]
[5,2,2,5,4,9,1,15,2,3,5]
[4,9,1,15,2]
[11,-1,8,2,10,6,7,-1,10,-1,8,-2,8,0,11]
```

Här kan man se att den topologiska ordern och *weight_ of_ longest_ path()* stämmer. Man ser också att det stämmer för när man använder higher order functions.

Polymophism testas genom att ersätta heltal med karaktärer i vikter för noder och kanter. För enkelhetens skull ändrades att om något hade en vikt 1 så fick den karaktären 'a', 2 fick 'b' och så vidare. Testerna som gjordes syns nedanför.

```
topological_order_list = dag.topological_ordering()
```

```
f = lambda int : int
g = lambda int : int
longest_path = dag.weight_of_longest_path(v1,v10,f,g)
```

Följande resultat gavs.

```
[1,2,3,4,5,7,6,8,9,10]
['e', 'b', 'b', 'e', 'd', 'i', 'a', 'b', 'd', 'b', 'b', 'a', 'b',
↪  'c', 'e']
```

Man kan se att vägarna följer testresultaten med heltal och därmed är implementerad korrekt. Man valde att alla IDn ska vara i heltal och det är därför topological visas med det, men även topological ordern stämmer.

Här är ett test på om man försöker kolla längsta vägen på en väg som ej existerar. Skulle man använda grafen från Figur 4 och ta vägen från nod 1 till nod 2 kommer användaren få meddelat att det ej existerar en väg mellan dessa två noder.

```
f = lambda int : int
g = lambda int : int
longest_path = dag.weight_of_longest_path(v1,v2,f,g)
```

```
No path available
```

Här visas det felmeddelande som förväntades att få när man försöker gå en väg som ej existerar.

Eftersom en graf ej får vara cyklisk testas detta också för python programmet. Här nedan testar man lägga till en kant mellan nod 6 och 2 det ska göra grafen cyklisk.

```
dag.add_edge(v6, v2, 3)
```

Följande felmeddelande ges i terminalen.

```
Error: tried to add edge which makes graph cyclic.
 Continuing without adding the edge 5 -> 1
```

Alltså ett felmeddelande som säger att man försökt göra grafen cyklisk, programmet kommer att fortsätta, men den kanten tas inte med i grafen.

5 Fungerande exempel

Ett Praktiskt exempel man skulle kunna använda är hur man klär på sig på morgonen. Om man har följande graf

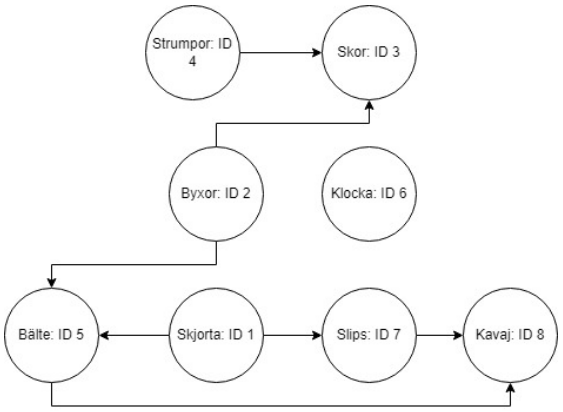

Figure 3: Ett praktiskt exempel över hur man skulle kunna implementera en DAG.

så kan man få ut en topologisk order över hur man skulle kunna klä på sig en morgon. För att få ut en topologisk order kallas grafen på följande vis.

```
topological_order dagr6a
```

Där visas en topologisk order på följande sätt.

```
[1,2,4,3,5,6,7,8]
```

Detta innebär att man får ut en ordning på följande sätt.
Skjorta → *Byxor* → *Strumpor* → *Skor* → *Bälte* → *Klocka* → *Slips* → *Kavaj*
Vilket skulle kunna ses som ett bra sätt att klä på sig om man vill.
Det blir svårt att testa längsta vägen för detta exempel men man kan kolla på om man vill börja med att sätta på sig en skjorta för att sedan sätta på sig en kavaj. Det vill säga att man kallar på det genom följande sätt.

```
weight_of_longest_path 1 8 dagr6a (\f -> f) (\g -> g)
```

Vägen kommer då att se ut på följande vis

```
[2,1,15,1,3]
```

Där man innan satt att sätta på en sig slips har en vikt av 15, av erfarenhet vet man att det tar en lång tid. Så om man vill vara så ineffektiv som möjligt med att sätta på sig skjortan och sluta med en kavaj först kommer man sätta på sig slipsen först istället för bältet.

6 Analys/Utvärdering

Vår lösning löser problemet med dem funktionerna som man skulle ha med och löser det på ett korrekt sätt. Vi kan kolla på testerna och se att det klarar av ett brett testfall med allt från att hantera polymorphiska typer till att ändra funktionerna i higher order funktionerna. I vårt fall tänker vi att man bara behöver hantera två olika datatyper i den polymorphiska delen, heltal och karaktärer. Detta då det skulle vara helt onödigt att kunna hantera någon annan datatyp. En väg enligt vår uppfattning kan givetvis ha heltal eller karaktärer som värden men att äventyra ännu längre och ha andra datatyper känns överflödigt och irrelevant till lösningen man skulle få fram.

När vi får ut vår längsta väg har vi även valt att representera vägen av en lista av nodernas vikt, samt kanter. Detta då det blir konstigt om man skulle använda sig utav bokstäver som datatyp till vikten. När vi kollar vikterna tar vi ASCII-koden för att jämföra vilken väg som är längst, det vill säga att $a + a$ väger mindre än $a + b$ då b har ett högre ASCII värde. När vi väl ska summera ihop listan skulle det bli något som inte är läsbart om man använder karaktärer och därför beslutades om att använda en lista för heltal för att programmet inte ska göra två helt olika saker.

7 Reflektion

Den erfarenhet och lärdomar vi tar med oss ifrån denna laborationsuppgift är att samma uppgift med samma algoritm går att implementera i olika programspråksparadigmer. Vi personligen tycker att det var mycket lättare att utföra uppgiften i Python jämfört med Haskell. Vilket kan bero på att vår erfarenhet med funktionella språk är mycket lägre jämfört med objektorienterade språk.

Vi hade problem att lämna in uppgiften i tid inför det första inlämningstillfället med anledningen av väldigt många tentor/inlämningar i andra kurser och denna kurs. Utöver det har arbetet gått väldigt bra, skönt att covid är över och att man kan träffas i skolan och arbeta tillsammans igen♡.