Programmeringsprojekt: Fase 1

Hlynur Æ. Guðmundsson, Mathias B. Jensen, and Valdemar B. Reib

Fasekoordinator - Valdemar B. Reib

Institut for Matematik og Datalogi, Syddansk Universitet

November 2024

1 Problemstilling

I dette projekt er problemstillingen tredelt i faser, som giver anlæg til progressiv udvikling. I første fase har gruppen fået til opgave at udvikle de grundlæggende moduler, som resten af faserne kommer til at bygge på. Der skal til hvert modul laves et antal af funktioner, således at det færdige produkt af fase 1, kan danne et sorteringsnetværk ud fra de to modulers funktionalitet. Dette mulliggøre i sidste ende sortering af en liste heltal.

2 Introduktion

I dette projekt skal der udvikles et komparatornetværk igennem tre forskellige faser. I denne rapport dokumenterer, diskutere og forklare vi hvordan første fase er blevet gennemført. Her har udfordringen været at kreere to moduler i Python ved navn *Comparator.py* og *Network.py*, som tilsammen tillader gruppen at sortere en liste af heltal ved hjælp af komparatorer.

3 Fremgangsmåde

Da gruppen skulle udvikle to moduler, med flere forskellige funktioner har gruppen gjort brug af nogle teknikker som har gjort arbejdet lettere. Gruppen valgte at opdele versionerne i koden i forskellige revisioner, og løbende gemme dem på github[Azu24]. Da det tillod gruppen at dele kode og samle forskellige versioner. Ved at gøre brug af disse revisioner blev det også nødvendigt med fordeling af arbejdet. Gruppen har gjort brug af online møder, og fysiske møder i kombination for at sikre at hele gruppen fik en status. Dertil fordelte gruppen arbejdet ligeligt og byttede internt arbejdsopgaver hvis et individ sad fast på en enkelt funktion. Ved at møde regelmæssigt blev der planlagt arbejdsfordeling og forventning til næstkommende møde.

4 Opbygning af kode

I denne sektion vil der forklares nogle udvalgte funktioner og opbygningen af de pågældende moduler. De to moduler *Comparator.py* og *Network.py* er opbygget bag princippet omkring dataklasser og abstrakte datastrukturer. *Comparator.py* og *Network.py* har hver en dataklasse med henholdsvis to heltal i og j, samt en liste. Dette muliggjorde nem iteration af listerne. Dette tillader gruppen at udvide, rette og tilføje ting til modulerne uden videre. Dertil er projektet bygget op om ideen omkring kontraktbaseret programmering, der har gjort samarbejde omkring projektet nemt, da funktionerne skal returnere noget bestemt, kan hvert enkelt individ lave funktioner der bruger de andre funktioner uden at tænke over hvordan den pågældende funktion er kreeret.

5 Opbygning af *Comparator.py*

Komparator modulet har til formål at danne et objekt med to variabler der svarer til kanalerne de er kreeret i mellem. Modulet er bygget op rundt om flere forskellige funktioner, som tillader os at gøre brug af den effektivt i *Network.py* modulet. Komparator modulet har et par interessante funktioner, herunder *"apply()"*, *"all_comparators()"*, *"std_comparators()"*, *"to_program()"*.

5.1 *Apply()*

Apply er en funktion der varetager selve funktionaliteten af en komparatoren. Den gør brug af komparatorens variabler og en liste, til at sammenligne størrelsesorden på de to elementer i listen, komparatoren er på. Denne funktion er kreeret således at der i *Network.py* kan kaldes på den rekursivt for at løbe en hel liste igennem. Dette vil tillade gruppen at sortere en liste ved brug af flere komparatorobjekter.

5.2 *All_comparators()*

All_comparators er en funktion der tillader gruppen at hente en liste med alle de mulige komparatorer på n-kanaler. Ved at gøre brug af en *bubblesort* struktur i funktionen med brug af to *for each* lykker, for at sikre at alle kombinationer af komparatorer opnås.

5.3 *Std_comparators()*

Std_comparators er en funktion der ligeledes *all_comparators* returnere en liste med komparatorer på baggrund af n-kanaler. I denne funktion tjekkes alle de kombinationer for om de er en standardkomparator, hvilket betyder at i-variablen skal være strengt mindre end j-variablen. På den måde sikres der at listen der returneres, kun indeholder alle de mulige kombinationer af standardkomparatorer til n-kanaler.

5.4 *To_program()*

To_program er en funktion der tillader gruppen at kreere et Python program som kan køres direkte. Dette forventes brugt i de senere faser for at analysere og optimere. I komparatormodulet sættes det op således at *Network.py* kan gøre brug af det.

6 Opbygning af *Network.py*

Netværksmodulet har til formål at danne strukturen, som komparatormodulet skal agere på. Modulet er ligeledes komparatormodulet, opbygget af flere forskellige funktioner, som tillader os at bruge komparatormodulet effektivt til at simulere et sorteringsnetværk.

Netværksmodulet inkluderer funktioner, hvis funktionalitet efterligner mange af funktionerne i komparatormodulet, hvor hovedforskellen er, at de gentages for hver komparator i netværket. Udover dem er der fire essentielle funktioner i netværksmodulet, som hedder "*apply() "*, *"outputs()"*, *"all_outputs()"*, *"is_sorting()"* og *"to_program()"*.

6.1 Opbygning af *apply*()- og *outputs*()-funktionerne

Funktionerne *apply*() og *outputs*() har til formål af sortere, fra mindst til størst, en liste af lister indeholdende heltal. Funktionernes væremåde emulerer hvordan et sorteringsnetværk som helhed kan sortere talværdier.

Netværksmodulets udgave af *apply*()-funktionen efterligner i høj grad udgaven i komparatormodulet, men der er en essentiel forskel. Funktionen påkræver et ikke tomt netværk indeholdende komparatorer og en liste af heltal. Listen er den samling af tal, som der sorteres. Den fungerer ved at benytte en *for-each* lykke til at gentage *apply*()-funktionen fra komparatormodulet for hver komparator i input netværket. Dette resulterer i at alle komparator i et netværk bliver brugt en gang hver, som simulere hvordan en enkelt kanal påvirkes af flere komparatorer. Netværksmodulets udgave af *apply*()-funktionen kan dermed sortere en hel liste af heltal, i stedet for en andel, hvis et netværk har de rette komparatorer.

Funktionen *outputs*() bygger videre på *apply*()-funktionen og udvider dets omfang således, at et netværk kan sortere flere lister af heltal, i stedet for kun en ad gangen. Udover det påkræver funktionen en liste af lister med heltal som input i stedet for en enkelt liste med heltal. Funktionen fungerer ved, at den for hver liste i input listen kalder *apply*()-funktionen med listerne af heltal fra input listen som argument. Funktionen vil trinvis gennemgå og sortere alle listerne med heltal indtil der ikke er flere tilbage. Når funktionen er færdig, så er alle tallene i alle listerne blevet flyttet rundt på af alle komparatorer, så længe der bruges et gyldigt komparatornetværk. De tre funktioner og deres ansvarsområder er illustreret på den følgende figur

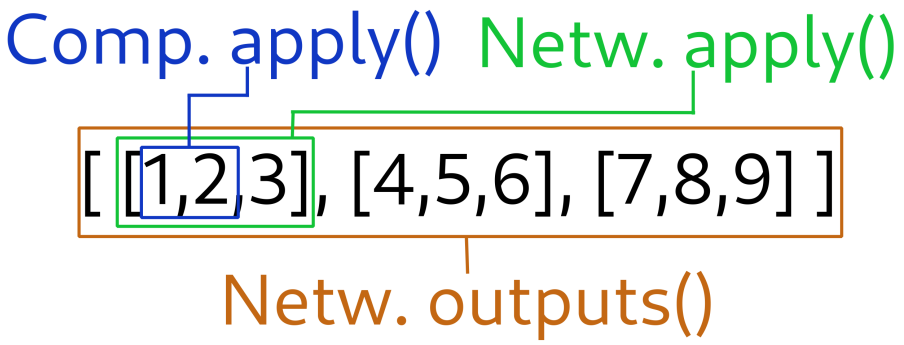

Figure 1: Diagram over omfang af funktioner. (Egen tilvirkning)

6.2 Opbygning af *permutations*() og *all_outputs*()

For at kunne teste vores sorteringsnetværk kreerede vi funktionen *all_outputs*() og en tilhørende funktion *permutations*().

Formålet med disse to funktioner er, at opstille alle permutationer af tallene 0 og 1, i form af en liste af lister med heltal, hvorefter et input netværk vil forsøge at sortere tallene i listerne. Efter netværket har sorteret tallene, så vil funktionen fjerne gentagne lister i tilfælde af, at listerne indeholder de samme tal i samme rækkefølge. Funktionerne muliggøre, at vi kan teste vores sorteringsnetværk og analysere deres væremåder, uden at skulle teste uendelige mange talværdier.

All_outputs()-funktionen gør brug af *permutations.py*-modulet sammen med *outputs*()-funktionen. Modulet *permutations.py* indeholder en enkelt funktion ved navn

permutations(), som opstiller selve permutationerne af 0 og 1 ved hjælp af *list-comprehension*, som Python understøtter. Funktionen påkræver et enkelt argument, som beskriver længden af permutationerne. For eksempel hvis længden var 3, så vil funktionen returnere:

$$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,1]]$$

Det ligger seperat i sit eget modul som et resultat af arbejdsfordelingen, men det kunne godt integreres i *network.py*-filen ud større udfordringer. Alle permutationerne af 0 og 1 bruges i funktionen $all\_outputs()$, som udover at kræve et heltal til $permutation()$-funktionen, også kræver et ikke tomt komparatornetværk. Netværket bruges til at sortere alle permutationerne ved hjælp af $output()$-funktionen. Efter at listerne bliver sorteret, så vil mange af listerne med gentagelser af hinanden, da de indeholder samme antal af 0'er og 1'er, der nu står i samme rækkefølge. Gentagelserne bliver trinvis frasorteret ved at sammensætte en simpel $for - each$ lykke med en $if\ not\ in$ commando. De vil tilsammen kun tilføje unikke indslag i en liste. Ved brug af eksemplet fra før, så vil funktionen trinvis fungerer således:

Trin 1: Opstil Permutationer
$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,1]]$

Trin 2: Sorter 0'er og 1'er
$[[0,0,0],[0,0,1],[0,0,1],[0,0,1],[0,1,1],[0,1,1],[1,1,1]]$

Trin 3: Fjern Gentagelser
$[[0,0,0],[0,0,1],[0,1,1],[1,1,1]]$

Tilsammen tillader funktionerne gruppen at sammenligne og analysere et komparatornetværk. For eksempel hvis gruppen havde to netværk, som havde de samme komparatorer men i forskellige rækkefølge, ville det være meget relevant at analysere, hvorvidt de sortere ens. Tilsvarende hvis vi havde et netværk, der manglede nogle komparatorer, så ville vi kunne opfange det ved, at analysere hvordan netværket sortere.

6.3 Opstilling af *is_sorting*()

For at kunne udnytte netværket effektivt i projektet, er det vigtigt at vide, hvilke evner og begrænsninger et individuelt netværk har. Derfor kreerede vi funktionen $is_sorting()$. Funktionen analysere om et inputnetværk kan korrekt sortere en specifikt antal kanaler, som også er et input i form af et heltal. For at kunne tjekke om et netværk kunne sortere et bestemt antal kanaler, så opstillede vi nogle minimumskrav.

- 1) Et netværk skal ikke være tomt

- 2) Skal som udgangspunkt ikke indeholde standardkomparatorer, men et gyldig netværk kan indeholde dem, så længe at fejlsorteringen bliver rettet af standardkomparatorer.

- 3) Den maksimale kanal af et netværk skal være lig med det antal af kanaler vi er interesseret i at undersøge om vores netværk kan sortere. Hvis et netværks maksimale kanal er mindre end vores antal af kanaler vi analysere, så vil et netværk ikke kunne flytte og sortere værdierne i den øverste kanal. Hvis et netværks maksimale kanal er større, så vil den heller ikke kunne sortere korrekt, da komparatorer vil forsøge at flytte værdier fra og til ikke-eksisterende kanaler.

Efter at have defineret nogle minimums krav, så skal funktionen også kunne yderligere analysere sorteringsnetværk. For selvom et netværk opfylder alle tre minimumskrav, så er det ikke garanteret, at det vil kunne sortere et bestemt antal kanaler korrekt. I planlægningsprocessen kom vi frem til to gyldige strategier, for at tjekke om et netværk kan sortere et antal kanaler eller ej.

- 1) Analysere om et netværk indeholder alle de nødvendige komparatorer i den rigtige rækkefølge for, at kunne sortere et antal kanaler korrekt. Matematisk tjekker vi om alle de rigtige komparatorer er en delmængde af et netværk. Teoretisk kunne det være en hurtig og effektiv løsning, men strategien har en stor ulempe. Hvis man tillader et netværk at have gentagne eller ikke standardkomparator er mængden af gyldige netværk uendelige. Hvis et netværk har de rette komparator til sidst i rækkefølgen, så vil de altid kunne rette på fejlene af de forrige, så netværket som helhed kan sortere korrekt. Det vil være intensivt at gennemgå et netværk slavisk, når der er uendelige mange gyldige muligheder. Hvis man forbyder ikke standardkomparatorer, så vil det ikke gøre den største forskel, da et netværk er tilladt at have kopier af samme komparatorer.

- 2) Lad komparatornetværk forsøge at sortere alle permutationer af 0 og 1 og derefter analysere om netværket har sorteret tallene korrekt. Strategien er forholdsvis mere simpel end den første, men den kan være intenst, når vi bearbejder mange kanaler og netværk med komparatorer. Dog undgår strategien hele problematikken med, at analysere om ikke standardkomparatorer bliver fuldstændig overskrevet af standardkomparatorer.

I sidst ende implementerede vi den anden strategi samt minimumskravende. Dette gjorde vi ved hjælp af funktionerne fra de tidligere afsnit.

6.4 *To_program*()

To_program funktionen er en funktion som ligeledes den fra komparatormodulet returnere en liste af kommandoer som kan køres direkte i Python. Den gør brug af komparatormodulets version af *to_program*, for at kunne printe en udvidelse af denne version. Dette tillader gruppen at fokusere på essensen af programmet, som vi forventer skal bruges i de senere stadier af projektet.

7 Test og evaluering

I denne sektion vil vi benævne hvordan at koden er blevet testet, og dertil også evalueret på baggrund af de forventelige resultater. Den kontraktbaseret struktur sikrer at hver funktion er klart beskrevet til at returnere specifikke ting, så som *int, str, list*[*int*]. Dette har tilladt gruppen at opskrive *DOCTEST* til hver funktion, som tydeligt angiver hvad den pågældende funktion gør. Dette er så udvidet i en separat test fil kaldet *test.py*, som har lagt an til flere test, med mere konkrete eksempler. Dette giver en samlet ide omkring hele programmets funktionalitet, ved blot at kører test filen.

7.1 Test af kode

I testen af *apply*() funktionen i *network.py* lavede vi nogle tests, for at se om funktionen virkede rigtig, derefter lavede vi mere komplekse netværk og lister. En af de tests inkluderede det samme netværk og liste fra projekbeskrivelsen (jf. 2). Dette viste gruppen at funktionen virkede korrekt, da sorteringen af den speciffike liste er betinget af det specifikke netværk af komparatorer. Testen af *to_program*() i *network.py* var mere kompliceret, da det ønskede resultat krævede at vi testede det med *exec*() funktionen, denne funktion læser Python kode fra en streng og kører det, ved brug af *exec*() skulle *aux* variablen første deklareres, fordi den kun eksisterede som et argument kaldet af *to_program*()

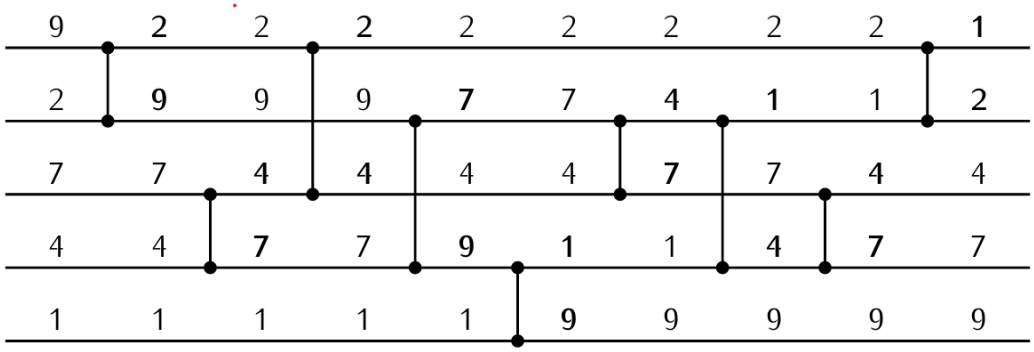

Figure 2: Komparator Netværk[Cru24]

Vi har struktureret vores test.py fil til at inkludere alle funktioner i første fase af projektet, som giver et samlet overblik. Filen er delt i to med *print*() funktioner som tydeligt angiver hvad der testes. I for eksempel test af *all_outputs*() gør vi brug af *print*() funktioner til tydeligt at vise forskellen på et sorteret og ikke sorteret output. Der gøres brug af vores hjælpefunktion permutations.py til at danne alle de permutationer, og derefter kaldes *all_outputs*() for at sortere de permutationer. Denne struktur bruges generelt i *test.py*. Et andet eksempel på dette ses i test af *is_sorting*() som også er struktureret for at tydeliggøre de forskellige scenarier hvor funktionen returnere *false*. Ved at holde den struktur og tilgang, sikre vi os at de pågældende funktioner virker som forventet.

7.2 Evaluering af kode

I løbet af udviklingen af de to moduler *Comparator.py* og *Network.py*, er der blevet taget en del valg, og ændringer. Dertil er der blevet brugt ChatGPT[Ope24] som hjælpeværktøj til sparring under udvikling af de forskellige funktioner osv. I denne sektion vil vi fokusere på at diskutere nogle af de valg.

7.3 Valg af løsningsforslag

En af de største valg der blev taget, var hvilken programmerings paradigme der ville fokuseres på. De vi i gruppen overordnet set havde mest erfaring med imperativ, er der en klar overvægt af det format. Dertil er der i vores modul permutations.py brugt funktionel programmering som løsningsforslag. Dette startede også originalt som en imperativ løsning. Dog var denne løsning delvist udformeret med ChatGPT som hjælpeværktøj, og derfor blev den senere omskrevet til en funktionel løsning.

I løbet af udviklingen er der også sket fejl, da programmet er komplekst, kan der ske uventet ting under test og udvikling. En af de største fejl var den første implementering af *Comparator.py* modulets funktion *apply*() da den gjorde brug af to for-each lykker som essentielt set er en *bubblesort* algoritme. Det resulterede i at output fra funktionen, var en allerede sorteret liste. Det blev opdaget senere hen i testforløbet, da valget af komparator var intet sigende. Det er et godt eksempel på hvorfor vi har valgt at fokusere så meget på testning af koden undervejs. Da vi ellers kunne stå i en situation hvor fremtidige funktioner fejler uventet, og i værste tilfælde de næste to faser. Et andet godt eksempel omhandlede funktionen *outputs*(), som bruges flere steder i *Network.py* da den er fundamentet i for eksempel *all_outputs*(). Da vi valgte at skrive et adskilt modul til at danne permutationerne af 0 og 1, baseret på n, oplevede vi et problem i testning af samtlige funktioner, da vi konsekvent fik en *Out of Range* fejl. Efter lang tid opdagede vi at problemet lå i implementeringen af de funktioner. Modulet *permutations.py* returnerede en liste med lister, og i implementeringen af *outputs*() blev den allerede samlet i en liste. Det gjorde at vores komparatorer ikke kunne få fat i de elementer der skulle sorteres, da det var en tredimensionel liste og ikke todimensionel som kontrakten beskrev. Den type fejl er ekstremt svær at finde da det bygger på flere elementer der fejler. Da hver funktion bruger den foregående. Dette er endnu et eksempel på hvor testning undervejs af koden forhindrede fremtidige problemer.

8 Konklusion

I denne rapport har vi gennemgået den første fase i det tredelte projekt. Der er kreeret to moduler i Python *Comparator.py* og *Network.py* som udgør første fase. Funktionaliteten af de to moduler er blevet tydeliggjort igennem eksempler på test og opbygning. Dertil er gruppens fremgangsmåde dokumenteret og diskuteret løbende. Det ses igennem udførlige tests og dokumentation at det færdige produkt af Fase 1 opfylder de krav der er stillet i problemstillingen, og ligger dertil grund til de næstkommende faser.

9 Litteraturliste

References

[Azu24] AzureX. *SDU-Programmeringsprojek*. Accessed: 08-11-2024. 2024. URL: https://github.com/xXmemestarXx/SDU-Programmeringsprojekt/tree/AzureX/.

[Cru24] Luis Cruz-Filipe. *Gruppeprojekt — Efterår 2024 — Introduktion*. PDF document. 2024.

[Ope24] OpenAI. *ChatGPT*. Accessed: 08-11-2024. 2024. URL: https://openai.com/chatgpt.

10 Bilag

10.1 comparator.py

```python
"""
comparator.py
Made by H8G03: Valdemar, Mathias og Hlynur
"""

from dataclasses import dataclass

"""
Implements a custom dataclass that uses two variables i and j as integers.
This enables creating a instance of a comparator that symbolizes'
the gates of a comparator network.
"""
@dataclass
class Comparator:
    i: int
    j: int

def make_comparator(i: int, j: int) -> Comparator:
    """
    Takes the arguments i and j and returns a Comparator instance with the arguments.

    DOCTEST
    i = 0
    j = 2
    >>> make_comparator(i, j)
    Comparator(0, 2)
    """
    return Comparator(i, j)

def get_i(c: Comparator):
    """
    Auxillary function to get the i-value of a Comparator
    """
    return(int(c.i))

def get_j(c: Comparator):
    """
    Auxillary function to get the j-value of a Comparator
    """
    return int(c.j)

def get_i_and_j(c: Comparator):
    """
    Auxillary function to get the contents of a Comparator
    as a tuple
    """
    return (int(c.i), int(c.j))

def min_channel(c: Comparator) -> int:
    """
    Compares the value of i and j in c and returns the lower value
```

```
52
53      DOCTEST
54      i = 0
55      j = 2
56      >>> min_channel(Comparator(i, j))
57      0
58      """
59      if(c.i > c.j):
60          return c.j
61      else:
62          return c.i
63
64  def max_channel(c: Comparator) -> int:
65      """
66      Compares the value of i and j in c and returns the higher value
67
68      DOCTEST
69      i = 0
70      j = 2
71      >>> max_channel(Comparator(i, j))
72      2
73      """
74      if(c.i < c.j):
75          return c.j
76      else:
77          return c.i
78
79  def is_standard(c: Comparator) -> bool:
80      """
81      Checks if c is a standard comparator (it sets the lowest value on the lowest channel)
82
83      DOCTEST
84      i = 0
85      j = 2
86      c = make_comparator(i, j)
87      >>> is_standard(c)
88      True
89      """
90      return (c.i < c.j) and (c.i != c.j)
91
92  def apply(c: Comparator, w: list[int]) -> list[int]:
93      """
94      Uses a comparator to compare 2 elements in a list of integers and
95      swaps them if needed.
96
97      DOCTEST
98      c = make_comparator(i, j)
99      w = [3,4,2,5]
100     >>> apply(c, w)
101     [2,3,4,5]
102     """
103
104     if(is_standard(c) and w[c.i] > w[c.j]):
105         w[c.i], w[c.j] = w[c.j], w[c.i]
106     return w
107
```

```
108 def all_comparators(n: int) -> list[Comparator]:
109     """
110     Returns a list of all possible comparators on n-channels.
111 
112     DOCTEST
113     n = 3
114     >>> all_comparators(n)
115     [Comparator(i=0, j=1), Comparator(i=0, j=2), Comparator(i=1, j=0),
116      Comparator(i=1, j=2), Comparator(i=2, j=0), Comparator(i=2, j=1)]
117     """
118     comparators=[]
119     # This iterates the nested part of the for loop,
120     # and ensures we print all combinations of i and j.
121     for i in range(n):
122         for j in range(n):
123             # Ensures that i and j are not equal, to comply with the defintion of a comparator
124             if i != j:
125                 comparators.append(Comparator(i,j))
126     return comparators
127 
128 def std_comparators(n: int) -> list[Comparator]:
129     """
130     Returns a list of all standard comparators on n-channels.
131     Standard mean that the j-value is always larger than the
132     i-value.
133 
134     DOCTEST
135     n = 3
136     >>> std_comparators(n)
137     [Comparator(i=0, j=1), Comparator(i=0, j=2), Comparator(i=1, j=2)]
138     """
139     comparators=[]
140     # This iterates the nested part of the for loop,
141     # and ensures we print all combinations of i and j.
142     for i in range(n):
143         for j in range(n):
144             # Checks if i and j are not equal and is a standard comparator
145             if(i != j and is_standard(Comparator(i,j))):
146                 comparators.append(Comparator(i,j))
147     return comparators
148 
149 def to_program(c: Comparator, var: str, aux: str) -> list[str]:
150     """
151     Returns a list of instructions that simulates the Comparator.
152 
153     DOCTEST
154     i = "0"
155     j = "1"
156     var = "new_list"
157     aux = "temp"
158     c = make_comparator(i, j)
159     >>> to_program(c, var, aux)
160     ['if new_list[0] > new_list[1]:', 'temp = new_list[0]',
161     'new_list[0] = new_list[1]', 'new_list[1] = temp']
162     """
163     return [
```

```
    f"if {var}[{c.i}] > {var}[{c.j}]:",
    f"    {aux} = {var}[{c.i}]",
    f"    {var}[{c.i}] = {var}[{c.j}]",
    f"    {var}[{c.j}] = {aux}"]
```

10.2 permutations.py

```
import functools as func

def permutations(n) -> list[list[int]]:
    """
    Strategy:
    Start with all zeros and then edit
    the lists one-by-one
    """

    """Start by making an empty list"""
    all_permu = []

    """Then make a list that contain n amount of 0's """
    all_zero = ["0"] * n

    """Calculate hvor many permutations of n there is"""
    amount_permu = 2**n

    """
    Start by going through the lists of 0's one-by-one
    until reaching the number of permutations
    """

    i = 0
    while i < amount_permu:

        v = format(i,"b").zfill(n)

        all_permu.append(list(map(lambda x,y : int(x) | int(y),all_zero,v,)))
        i = i + 1

    return all_permu
```

10.3 network.py

```
"""
network.py
Made by H8G03: Valdemar, Mathias og Hlynur
"""

from dataclasses import dataclass
import comparator as Comp
from permutations import * #Note: We made this module ourselves

"""
```

```
11 We start by importing the dataclass functionality,
12 the comparator module and our own permutations module.
13 """
14
15 @dataclass
16 class Network():
17     """
18     The network is implemented as a dataclass,
19     whose only property is to create a list of comparator objects.
20     """
21     network: list
22
23 def to_string(net: Network) -> str:
24     """
25     Writes the contents of a Network object to a String
26     Note: Useful for testing
27
28     DOCTEST
29     >>> net.network = [obj_1, obj_2]
30     '[obj_1, obj_2]'
31     """
32     return (f"{net.network}")
33
34 def empty_network() -> dataclass:
35     """
36     Creates an empty Network object that only contains an empty list
37     """
38     return Network([])
39
40 def append(c: Comp, net: Network) -> None:
41     """
42     Appends a comparator c to the Network, net.
43
44     Iterates on net and does not make a copy of it in memory.
45
46     DOCTEST
47     >>> append(Comparator_1,Network)
48     >>> append(Comparator_2,Network)
49     Network = [Comparator_1, Comparator_2]
50     """
51     net.network.append(c)
52
53 def size(net: Network) -> int:
54     """
55     Returns the amount of Comparators in a Network
56     in the form of an integer
57
58     DOCTEST
59     Network.network = [Comparator_1,Comparator_2]
60     >>> size(Network)
61     2
62     """
63     return len(net.network)
64
65 def max_channel(net: Network) -> int:
66     """
```

```
    Returns the largest j-value in the network. The main difference in not using
    max.channel() from comparator.py is that we avoid
    non-standard Comparators.

    Instead it uses get_j() from comparator as an auxiliary function
    Therefore Comparator.py needs to be imported.

    Requirement:
    size(net) > 0

    DOCTEST
    Comparator_1.i = 2
    Comparator_1.j = 4

    Comparator_2.i = 1
    Comparator_2.j = 5

    Comparator_3.i = 2
    Comparator_3.j = -1

    append(Comparator_1,Network)
    append(Comparator_2,Network)
    append(Comparator_3,Network)

    >>> max_channel(Network)
    5
    """

    max = Comp.get_j(net.network[0])

    """
    We start by equating the maximum channel of the network to
    the maximum channel of the first comparator of the network.

    This also ensures that if there is only one comparator in the
    network, the following for-each loop is skipped.
    """
    for i in range(1,size(net)):
        """
        Thereafter we check the maximum channel of each Comparator one-by-one
        except the first one, since we already checked it.
        """

        if Comp.get_j(net.network[i]) > max:
            """
            If the function finds a bigger number via the for-loop
            and if-statement, it overwrites the previous maximum channel
            and continues to check the other Comparators
            """
            max = Comp.get_j(net.network[i])
    return max

def is_standard(net: Network) -> bool:
    """
    Checks whether the input Network only
    contains standard comparators or not
```

```
123
124     Note:
125     A standard comparator is a comparator where
126     its j-value is larger than its i-value
127
128     Uses is.standard() from comparator module as an
129     auxiliary function. Therefore Comparator.py
130     needs to be imported.
131
132     Requirement:
133     size(net) > 0
134
135     DOCTEST
136     Net_1 = empty_network()
137     Com_1 = comparator_1.make_comparator(7,1)
138     Com_2 = comparator_1.make_comparator(3,5)
139     append(Com_1,Net_1)
140     append(Com_2,Net_1)
141
142     >>> is_standard(Net_1)
143     False
144     """
145
146     for i in range(0,size(net)):
147         """
148         We go through the list of Comparators and check one-by-one if they
149         are standard or not. If at least one of them is non-standard then
150         the whole function returns false.
151         """
152
153         if Comp.is_standard(net.network[i]) is False:
154             return False
155
156     """
157     If the for-each loop reached the end of the list without ever finding at
158     least one non-standard Comparator then the function returns True.
159
160     None are False => All are True
161     """
162     return True
163
164 def apply(net: Network, w: list[int]) -> list[int]:
165     """
166     Sorts a single list of integers using comparators
167     in a network.
168
169     Requirement:
170     size(net) > 0
171     len(w) > 0
172
173     DOCTEST
174     net = empty_network()
175     Com_1 = Comp.make_comparator(1,3)
176     Com_2 = Comp.make_comparator(2,3)
177
178     w = [1,2,4,3]
```

```
179
180     append(Com_1, net)
181     append(Com_2, net)
182
183     >>> apply(net, w)
184     [1, 2, 3, 4]
185     """
186
187     for i in range(0, size(net)):
188         """
189         For every Comparator in the network, apply the
190         Comparators on the list, so every Comparator is
191         used at least once
192         """
193         Comp.apply(net.network[i],w)
194
195     return w
196
197 def outputs(net: Network, w: list[list[int]]) -> list[list[int]]:
198     """
199     Returns a sorted list of lists containing no duplicates
200     The list themselves are not sorted
201
202     Requirement:
203     size(net) > 0
204     len(net) > 0
205
206     DOCTEST
207     net = empty_network()
208
209     Com_1 = Comp.make_comparator(0,1)
210     Com_2 = Comp.make_comparator(1,2)
211
212     append(Com_1,net)
213     append(Com_2,net)
214
215     v = [[36,25,25],[36563236,63425,4433660]]
216
217     >>> outputs(net, v)
218     [[25,25,36],[63425,4433660,36563236]]
219     """
220
221     for i in range(0,len(w)):
222         """
223         Sorts the individual lists inside the w
224         by using apply() as an auxiliary function
225         """
226         apply(net,(w[i]))
227
228         """
229         Depends on if we want to also remove the duplicate
230         numbers in the list
231         """
232
233         """
234         Python supports the datatype sets which are unorderd
```

```
        collections of elements with no duplicates.
        Python can convert a list to a set using the set()
        function which will also remove any duplicate elements.

        But by converting the set back to a list using the
        list() function, we essentially get the original list
        without duplicates, since we didn't change the order
        of the set or modified it in any other way.

        w[i] = set(w[i])
        w[i] = list(w[i])
        """
    return w


def all_outputs(net: Network, n: int) -> list[list[int]]:
    """
    Returns all permutations of 0 and 1 of n length,
    essentially the same as counting from 0 to n in binary,
    and then sortes them and removes repeats.

    Requirement:
    n > 0

    DOCTEST
    >>> all_outputs(1)
    [[0],[1]]

    >>> all_outputs(3)
    [[000],[001],[010],[011],[100],[101],[111]]
    """

    permu = permutations(n)

    permu = outputs(net,permu)

    permu_no_dupe = []

    for i in range(0,len(permu)):
        if permu[i] not in permu_no_dupe:
           permu_no_dupe.append(permu[i])

    return permu_no_dupe


def is_sorting(net: Network, size: int) -> bool:
    """
    We will refer the variable size to letter n

    Checks wheter a sorting network is able to correctly sort
    a network with n amount of channels with the comparators
    it has.

    DOCTEST
    Net_1 = empty_network()
    Com_1 = Comp.make_comparator(1,3)
```

```
    Com_2 = Comp.make_comparator(2,4)
    append(Com_1,Net_1)
    append(Com_2,Net_1)

    Net_2 = empty_network()
    Com_3 = Comp.make_comparator(0,1)
    com_4 = Comp.make_comparator(0,2)
    Com_5 = Comp.make_comparator(1,2)
    append(Com_3,Net_2)
    append(Com_4,Net_2)
    append(Com_5,Net_2

    >>> is_sorting(Net_1, 3)
    False

    >>> is_sorting(Net_2, 3)
    True
    """
    """
    Checking the minimum criteria for the network.
    """
    if len(net.network) == 0:
        return False

    if max_channel(net) != (size-1) : #Code using 0-indexing
        return False

    """
    First we create all permutations of 0 and 1
    of n length and place it in a list
    """
    permu = all_outputs(net,size)

    """
    We check if the list of lists of integers are
    sorted correctly be checking every integer
    one-by-one. If a previous value is larger
    than the following value then the list
    is sorted incorrectly. If all previous values
    are less than the following values then the list
    is sorted correctly.
    """

    for i in range(0,len(permu)-1):
        for j in range(0,len(permu[i])-1):
            if permu[i][j] > permu[i][j+1]:
                return False
    return True

def to_program(net: Network, var: str, aux: str)-> list[str]:
    """
    Returns a list of instructions that simulates the Comparator network.

    DOCTEST
    net = empty_network()
    Com_1 = Comp.make_comparator(0,1)
```

```
    Com_2 = Comp.make_comparator(2,3)
    Com_3 = Comp.make_comparator(0,2)
    append(Com_1, net)
    append(Com_2, net)
    append(Com_3, net)
    >>> to_program(Net1, var, aux)
    [['if var[0] > var[1]:', 'aux = var[0]', 'var[0] = var[1]', 'var[1] = aux'],
    ['if var[2] > var[3]:', 'aux = var[2]', 'var[2] = var[3]', 'var[3] = aux'],
    ['if var[0] > var[2]:', 'aux = var[0]', 'var[0] = var[2]', 'var[2] = aux']]
    """
    # Note: The output from this program can be run by the command exec()
    # and should result in the same execution as apply()
    returned_list = []

    # Iterates through the entire network
    for i in range(size(net)):
         # Iterates through the list from to_program() in comparator.py
        for j in range(len(Comp.to_program(net.network[i], var, aux))):
            # Appends each line as a new element in returned_list
            returned_list.append(Comp.to_program(net.network[i], var, aux)[j])
            # Next line is only needed if you need to be able to run the code
            # Creates a new line to seperate each output from to_program() in comparator.
            returned_list.append("\n")
    return returned_list
```

10.4 test.py

```
import network as Netw
import comparator as Comp

print(f"")
print(f"------- test comparator.py -------")
print(f"")
print(f"------- test 1 begin -------")
print(f"")
print(f"Testing make_comparator()")
print(f"")
i = 0
j = 2
c = Comp.make_comparator(i, j)
print(f"i = 0, j = 2 => {c}")

print(f"")
print(f"------- test 1 end -------")
print(f"")

print(f"------- test 2 begin -------")
print(f"")
print(f"Testing min_channel() & max_channel()")
print(f"")
i = 0
j = 2
c = Comp.make_comparator(i, j)
print(f"{c} => min = {Comp.min_channel(c)}, max = {Comp.max_channel(c)}")
```

```
28
29  print(f"")
30  print(f"------- test 2 end -------")
31  print(f"")
32
33  print(f"------- test 3 begin -------")
34  print(f"")
35  print(f"Testing is_standard()")
36  print(f"")
37  i = 0
38  j = 2
39  c = Comp.make_comparator(i, j)
40  print(f"i = {i} and j = {j} => {Comp.is_standard(c)}")
41  i = 2
42  j = 0
43  c = Comp.make_comparator(i, j)
44  print(f"i = {i} and j = {j} => {Comp.is_standard(c)}")
45  i = 2
46  j = 2
47  c = Comp.make_comparator(i, j)
48  print(f"i = {i} and j = {j} => {Comp.is_standard(c)}")
49
50  print(f"")
51  print(f"------- test 3 end -------")
52  print(f"")
53
54  print(f"------- test 4 begin -------")
55  print(f"")
56  print(f"Testing apply()")
57  print(f"")
58  comp = Comp.make_comparator(i = 0, j = 0)
59  w = [3,4,2,5]
60  print(f"Normal: [3,4,2,5] => {Comp.apply(comp, w)}")
61  v = [1,2,4,5]
62  print(f"Sorted: [1,2,4,5] => {Comp.apply(comp, v)}")
63  z = [1,3,4,4]
64  print(f"Duplicated Sorted: [1,3,4,4] => {Comp.apply(comp, z)}")
65  k = [2,1,1,3]
66  print(f"Duplicated: [2,1,1,3] => {Comp.apply(comp, k)}")
67  print(f"")
68  print(f"------- test 4 end -------")
69  print(f"")
70
71  print(f"------- test 5 begin -------")
72  print(f"")
73  print(f"Testing all_comparators()")
74  print(f"")
75  n = 3
76  print(f"n = 3 => {Comp.all_comparators(n)}")
77  n = 0
78  print(f"n = 0 => {Comp.all_comparators(n)}")
79  n = -3
80  print(f"n = -3 => {Comp.all_comparators(n)}")
81  print(f"")
82  print(f"------- test 5 end -------")
83  print(f"")
```

```
84
85  print(f"------- test 6 begin -------")
86  print(f"")
87  print(f"Testing std_comparators()")
88  print(f"")
89  n = 3
90  print(f"n = 3 => {Comp.std_comparators(n)}")
91  n = 0
92  print(f"n = 0 => {Comp.std_comparators(n)}")
93  n = -3
94  print(f"n = -3 => {Comp.std_comparators(n)}")
95  print(f"")
96  print(f"------- test 6 end -------")
97  print(f"")
98
99  print(f"------- test 7 begin -------")
100 print(f"")
101 print(f"Testing to_program()")
102 print(f"")
103 i = "i"
104 j = "j"
105 to_program_c = Comp.make_comparator(i = str(i), j = str(j))
106 print(Comp.to_program(Comp.Comparator(i, j), var = "network", aux = "temp"))
107 print(f"")
108 print(f"------- test 7 end -------")
109
110 print(f"")
111 print(f"------- test network.py -------")
112 print(f"")
113 print(f"------- test 1 start -------")
114 print(f"")
115 print(f"Testing to_string()")
116
117 Net = Netw.empty_network()
118 Com_1 = Comp.make_comparator(0,1)
119 Com_2 = Comp.make_comparator(1,2)
120 Netw.append(Com_1,Net)
121 Netw.append(Com_2,Net)
122
123 print(f"The network object to convert: {Net}")
124 print(f"Converted to string: {Netw.to_string(Net)}")
125
126 print(f"")
127 print(f"------- test 1 end -------")
128 print(f"")
129
130 print(f"")
131 print(f"------- test 2 start -------")
132 print(f"")
133 print(f"Testing empty_network()")
134 print(f"Test a empty network {Netw.empty_network()}")
135
136
137 print(f"")
138 print(f"------- test 2 end -------")
139 print(f"")
```

```
140
141 print(f"")
142 print(f"------- test 3 start -------")
143 print(f"")
144
145 Net = Netw.empty_network()
146 Com_1 = Comp.make_comparator(0,1)
147 Com_2 = Comp.make_comparator(1,2)
148
149 print(f"Testing append()")
150 print(f"Before append(): {Net}")
151 Netw.append(Com_1,Net)
152 Netw.append(Com_2,Net)
153 print(f"After append(): {Net}")
154
155 print(f"")
156 print(f"------- test 3 end -------")
157 print(f"")
158
159 print(f"")
160 print(f"------- test 4 start -------")
161 print(f"")
162
163 print(f"Testing size()")
164 Net = Netw.empty_network()
165 print(f"Before append, {Net}")
166 print(f"Before append, size: {Netw.size(Net)}")
167
168 Com_1 = Comp.make_comparator(0,1)
169 Com_2 = Comp.make_comparator(1,2)
170 Netw.append(Com_1,Net)
171 Netw.append(Com_2,Net)
172
173 print(f"After append, {Net}")
174 print(f"After append, size: {Netw.size(Net)}")
175
176 print(f"")
177 print(f"------- test 4 end -------")
178 print(f"")
179
180 print(f"")
181 print(f"------- test 5 start -------")
182 print(f"")
183
184 print(f"Testing max_channel()")
185
186 Net = Netw.empty_network()
187 Com_1 = Comp.make_comparator(0,1)
188 Netw.append(Com_1,Net)
189 print(f"Network to test: {Net}")
190 print(f"Max channel: {Netw.max_channel(Net)}")
191
192 Com_2 = Comp.make_comparator(1,2)
193 Com_3 = Comp.make_comparator(3,4)
194
195 Netw.append(Com_2,Net)
```

```
196 Netw.append(Com_3,Net)
197
198 print(f"Network to test: {Net}")
199 print(f"Max channel: {Netw.max_channel(Net)}")
200
201 print(f"")
202 print(f"------- test 5 end -------")
203 print(f"")
204
205 print(f"")
206 print(f"------- test 6 start -------")
207 print(f"")
208
209 print(f"Testing is_standard()")
210
211 Net = Netw.empty_network()
212 Com_1 = Comp.make_comparator(0,1)
213 Com_2 = Comp.make_comparator(1,2)
214 Com_3 = Comp.make_comparator(4,3)
215 Netw.append(Com_1,Net)
216 Netw.append(Com_2,Net)
217 Netw.append(Com_3,Net)
218
219 print(f"")
220 print(f"Network to test: {Net}")
221 print(f"Does the network only contain std comparators: {Netw.is_standard(Net)}")
222 print(f"")
223 Net_1 = Netw.empty_network()
224 Com_1 = Comp.make_comparator(0,1)
225 Com_2 = Comp.make_comparator(4,5)
226 Com_3 = Comp.make_comparator(2,5)
227 Netw.append(Com_1,Net_1)
228 Netw.append(Com_2,Net_1)
229 Netw.append(Com_3,Net_1)
230
231 print(f"Network to test: {Net_1}")
232 print(f"Does the network only contain std comparators: {Netw.is_standard(Net_1)}")
233
234 print(f"")
235 print(f"------- test 6 end -------")
236 print(f"")
237
238 print(f"")
239 print(f"------- test 7 start -------")
240 print(f"")
241
242 print(f"Testing apply()")
243
244 Net = Netw.empty_network()
245 Com_1 = Comp.make_comparator(1,2)
246 Com_2 = Comp.make_comparator(3,4)
247 Com_3 = Comp.make_comparator(0,1)
248
249
250 Netw.append(Com_1,Net)
251 Netw.append(Com_2,Net)
```

```
252 Netw.append(Com_3,Net)
253
254 v = [1,2,0,4,3]
255
256 print(f"Testing sorting the network with correct comparators")
257 print(f"")
258 print(f"{Net}")
259 print(f"")
260 print(f"List before: {v}")
261 print(f"")
262 print(f"Each step taken for sorting the list")
263 print(f"")
264 print(f"List after: {Netw.apply(Net,v)}")
265 print(f"")
266
267 Net_1 = Netw.empty_network()
268 Com_1 = Comp.make_comparator(1,3)
269 Com_2 = Comp.make_comparator(1,2)
270 Com_3 = Comp.make_comparator(3,4)
271 Com_4 = Comp.make_comparator(2,3)
272
273 Netw.append(Com_1,Net_1)
274 Netw.append(Com_2,Net_1)
275 Netw.append(Com_3,Net_1)
276 Netw.append(Com_4,Net_1)
277
278 w = [1,2,0,4,3]
279
280 print(f"Testing sorting the network with wrong comparators")
281 print(f"")
282 print(f"{Net_1}")
283 print(f"")
284 print(f"List before: {w}")
285 print(f"")
286 print(f"List after: {Netw.apply(Net_1,w)}")
287
288 print(f"")
289 print(f"------- test 7 end -------")
290 print(f"")
291
292 print(f"")
293 print(f"------- test 8 start -------")
294 print(f"")
295
296 print(f"Testing outputs()")
297
298 Net = Netw.empty_network()
299 Com_1 = Comp.make_comparator(0,1)
300 Com_2 = Comp.make_comparator(0,2)
301 Com_3 = Comp.make_comparator(1,2)
302
303 Netw.append(Com_1,Net)
304 Netw.append(Com_2,Net)
305 Netw.append(Com_3,Net)
306
307 w = [[36,25,25],[36563236,63425,4433660]]
```

```
308 print(f"Testing sorting the network with correct comparators")
309 print(f"")
310 print(f"{Net}")
311 print(f"List before: {w}")
312 print(f"")
313 print(f"List after: {Netw.outputs(Net,w)}")
314 print(f"")
315
316 print(f"")
317 print(f"------- test 8 end -------")
318 print(f"")
319
320 print(f"")
321 print(f"------- test 9 start -------")
322 print(f"")
323
324 print(f"Testing all_outputs()")
325
326 Net = Netw.empty_network()
327 Com_1 = Comp.make_comparator(0,1)
328 Com_2 = Comp.make_comparator(0,2)
329 Com_3 = Comp.make_comparator(1,2)
330
331 Netw.append(Com_1,Net)
332 Netw.append(Com_2,Net)
333 Netw.append(Com_3,Net)
334
335 print(f"Network to test with: {Net}")
336 print(f"All permutations unsorted: {Netw.permutations(3)}")
337 print(f"All permutations sorted: {Netw.all_outputs(Net, 3)}")
338
339 print(f"")
340 print(f"------- test 9 end -------")
341 print(f"")
342
343 print(f"")
344 print(f"------- test 10 start -------")
345 print(f"")
346
347 print(f"Testing is_sorting()")
348
349 print(f"")
350 print(f"For us to check is_sorting(), we need to check the cases where,")
351 print(f"Evertyhing is OK, and then all the false cases possible, being")
352 print(f"len == 0, max_channel != size-1, all if statements is true,")
353 print(f"but one comparator is missing")
354 print(f"")
355
356 Net = Netw.empty_network()
357
358 com1 = Comp.make_comparator(0,1)
359 com2 = Comp.make_comparator(0,2)
360 com3 = Comp.make_comparator(1,2)
361
362 Netw.append(com1,Net)
363 Netw.append(com2,Net)
```

```
364 Netw.append(com3,Net)
365
366 print(f"First case where the three criteria are OK")
367 print(f"Max: {Netw.max_channel(Net)}")
368
369 print(f"The network contains: {Netw.to_string(Net)}")
370 print(f"")
371
372 print(f"Is the network able to sort a network of size {3}: {Netw.is_sorting(Net,3)}")
373
374 Net = Netw.empty_network()
375
376 print(f"")
377 print(f"Second case where the network is empty")
378 print(f"Max: 0")
379
380 print(f"The network contains: {Netw.to_string(Net)}")
381 print(f"")
382
383 print(f"Is the network able to sort a network of size {0}: {Netw.is_sorting(Net,0)}")
384
385 print(f"")
386 print(f"Third case where the network max_channel is not equal to size-1")
387
388 Net = Netw.empty_network()
389
390 Com_1 = Comp.make_comparator(0,1)
391 Com_2 = Comp.make_comparator(0,2)
392 Com_3 = Comp.make_comparator(1,3)
393
394 Netw.append(Com_1,Net)
395 Netw.append(Com_2,Net)
396 Netw.append(Com_3,Net)
397
398
399 print(f"Max: {Netw.max_channel(Net)}")
400
401 print(f"The network contains: {Netw.to_string(Net)}")
402 print(f"")
403
404 print(f"Is the network able to sort a network of size {4}: {Netw.is_sorting(Net,5)}")
405
406 print(f"")
407 print(f"Fourth case where the network is missing one singular comparator")
408
409 Net = Netw.empty_network()
410
411 Com_1 = Comp.make_comparator(0,1)
412 Com_2 = Comp.make_comparator(1,2)
413
414 Netw.append(Com_1,Net)
415 Netw.append(Com_2,Net)
416
417 print(f"Max: {Netw.max_channel(Net)}")
418
419 print(f"The network contains: {Netw.to_string(Net)}")
```

```
420 print(f"")
421
422 print(f"Is the network able to sort a network of size {3}: {Netw.is_sorting(Net,3)}")
423
424 print(f"")
425 print(f"------- test 10 end -------")
426 print(f"")
427
428 print(f"")
429 print(f"------- test 11 start -------")
430 print(f"")
431
432 print(f"Testing to_program()")
433 print(f"")
434
435 Net = Netw.empty_network()
436 v = [9,5,8,23]
437 len_v = len(v)
438 for i in range(len_v - 1):
439     for j in range(len_v - 1):
440         Com = Comp.make_comparator(j, j+1)
441         Netw.append(Com, Net)
442 outputs = Netw.to_program(Net, "v", "aux")
443 print(outputs)
444
445 empty_string = ""
446
447 for x in range(len(outputs)):
448     empty_string += outputs[x]
449
450 print(f"list before: {v}")
451 aux:int
452 exec(empty_string)
453 print(f"list after: {v}")
454
455 print(f"")
456 print(f"------- test 11 end -------")
```

Programmeringsprojekt: Fase 2

Hlynur Æ. Guðmundsson, Mathias B. Jensen & Valdemar B. Reib

Fasekoordinator - Mathias B. Jensen

Institut for Matematik og Datalogi, Syddansk Universitet

2024-11-29

Indholdsfortegnelse

1 Problemstilling

I denne rapport behandler vi fase 2 af eksamensprojektet i kursuset DM574: Introduktion til programmering. I fase 2 bygges der videre på den første inplementering af et komparatornetværk, hvor vi udvider det således at der kan konstrukteres et sortingsnetværk på n-kanaler, som kan benyttes ved hjælp af en CLI (*Command Line Interface*).

2 Introduktion

I dette projekt er problemstillingen tredelt i faser, som giver anlæg til systematisk problemløsning. I anden fase har vi som gruppe fået til opgave at udvikle de grundlæggende moduler, som varetager dannelsen af sorteringsnetværk på n antal kanaler. Der skal til hvert modul laves et antal af funktioner, således at det færdige produkt af fase 2, kan danne et sorteringsnetværk til et bestemt antal kanaler.

3 Fremgangsmåde og Strategi

Den største ændring i vores fremgangsmåde sammenlignet med fase 1, er at der i højere grad benyttes redskaberne fra funktionelt programming og i nogen grad rekursiv programming. Efter at have fået feedback fra fase 1, opdagede vi at der var en kritisk misforståelse af komparatornetværkets væremåde. Vi lavede funktioner og moduler der laver bivirkninger og reviderer inputværdier, som de ikke skal, da det er vigtigt for projektet som helhed, at vores program gemmer og husker de oprindelige inputværdier, så de blandt andet kan genbruges flere gange. Metoderne fra funktionelt og rekursiv programming har den fordel, at de som udgangspunkt ikke vil revidere inputværdier og skabe data, der ikke er den samme hukommelse som inputværdien. Til gengæld kan programmer lavet med funktionelt og især rekursivt programming være hukommelsemæssigt intense. Da målet i fase 2 er at implementere sorteringsnetværker og ikke optimere dem til perfektion, så er ulempen ikke lige så betydningsfuld som i andre sammenhænge. I de tilfælde hvor vi ikke har brugt funktionelt eller rekursiv programming, har vi sat fokus på, at vores program laver kopier af inputværdier og behandler dem i stedet for, at være sikre på, at vores inplementering ikke laver uønsket

bivirkninger.

Vores misforståelse af funktionerne fra fase 1 har medført, at vores første inplementering af komparatornetværker ikke stemmer overens med programmingskontrakten og kravene fra fase 2. Derfor har vi aktivt valgt at gøre brug af vores undervisers implementere af komparatornetværker. Det har den store fordel, at vi har muligheden for, at korrekt kunne implementere sortingsnetværker med *Command Line Interfaces*. Ulempen er dog at vi skal arbejde med en implementere, som vi ikke har lige så meget indsigt i og skal derfor tildele noget af vores arbejdstid for at forstå det.

Ligesom i fase 1, så har vi benyttet softwaren *Git* og hjemmesiden *GitHub* [Azu24] til, at kunne samarbejde, programmere og fordele arbejdsopgaverne i praksis. Vores forbrug af *Git* er ikke perfekt, da vi ikke har meget erfaring med det. Udover det har vi benyttet hjemmesiden og LLM *Large Language Model* ChatGPT [Ope24] til, at omformuelere projektoplægget og programkontrakten, så vi kunne få en tydeligere ide om, hvad vi skulle programmere. Det har til dels være nyttigt da gruppen er flersproget. Kun enkelte tilfælde har vi benyttet ChatGPT til at lave kodeforslag, men alle gangene har vi afvist forslagene, da de enten ikke virkede eller benyttede metoder vi ikke havde erfaring med og kunne bedømme kvaliteten af.

4 Programmering af *filter.py*

Modulet filter har til hovedeformål at definere en datastruktur ved navn *Filter*. Strukturen benyttes i høj grad af de senere moduler til, at opbevarer vigtig data således, at vi kan til sidst i fasen opstille et sorteringsnetværk. Datastrukturen skal som minimum indeholde et netværk, som beskrevet af *network.py*-modulet, og alle dens binære permutationer, som er opstillet som en liste af lige lange lister. I løbet fasen skal der kunne tilføjes flere komparatorer, som er defineret i *comparator.py*-modulet, til datastrukturen, hvor der skal kunne gøres forskel på nyttige og redundante komparatorer. Funktioner der beskriver væremåden af Filter datastrukturen er beskrevet i de følgende kapitler.

4.1 *make_empty_filter()*

Formålet med *Make_empty_filter()* er at returnerer et tomt filter, bestående af et tomt netværk og alle de binære permutationer af den givne inputværdi

n. I praksis er et Filter opstillet af en dataclass, der indeholder et netværk, en liste af binære permutationer og den mængde kanaler Filteret er lavet til at kunne behandle. Alternativt kunne vi havde brugt en liste, hvor vi reserverer pladserne i listen til de forskellige værdier, men det frarådes at bearbejde for mange forskellige datatyper i samme liste. Filteret kreeres ved at bruge funktionen fra *network.py*-modulet, ved navn *empty_network()* og *_all_binary_inputs()*, som tilsammen danner det tomme filter og den tilhørende liste af permutationer. Udover det tilføjes også størrelsen, som er lig emd inputværdien n, som i praksis er et heltal.

4.2 *Net() & Out()*

Net() og *Out()* funktionerne giver muligheden for henholdsvis at aflæse det pågældende netværk ud eller de pågældende binære permutationer. Dette opnås ved brug af det objekt orienterede paradigme, da Filteret er et objekt med attributter, kan vi tilgå det givne netværk eller de givne binære permutationer, ved at kalde *"f.n"* eller *"f.out"*.

4.3 *Is_redundant()*

Is_redundant() er en funktion der tjekker om en given komparator er redundant ved at tilføje den til et givent filters netværk, og dertil tjekke om de binære permutationer ændres. Hvis dette er tilfældet dømmes komparatoren ikke-redundant, og ellers redundant. Dette opnås ved at tage en kopi at det nuværende stadie af netværket og de binære permutationer. Derefter tilføjes der den givne komparator, og der forsøges at sortere de binære permutationer. Til sidst tjekkes der om de tidligere binære permutationer er ens til de nye. Hvis dette er tilfældet, er komparatoren som nævnt redundant, og ellers ikke-redundant. Dette tillader gruppen at frasortere de ikke-redundate komparatorer. Dette bruges i samspil med *add()*-funktionen i *generate.py* modulet til at udvide netværket.

4.4 *Add()*

Add()-funktionen tilføjer en komparator i enden af netværket på et filter. Dette gøres med et simpelt kald til Filter med opdateret argumenter, der indeholder den ønskede komparator. Funktionen bruges i samspil med *is_redundant()* i *generate.py*-modulet, til at udvide alle de pågælende filtre.

4.5 *Is_sorting()*

Is_sorting() funktionen tjekker om det givne netværk i et filter er et korrekt sorteringsnetværk, ved at tjekke om netværket kan sortere alle binære permutationer fra mindst til størst. I praksis gøres det ved at aktivere *network.py*-modulets *is_sorting()*-funktion, hvor vi giver et Filters netværk.

5 Opstilling af *generate.py* og *extend()*-funktionen

Efter at have defineret filterdatastrukturen skal det behandles i *generate*-modulet. Modulet består af en enkelt funktion som hedder *extend()*. Funktionen kræver en liste af filtre og et positivt heltal, som beskriver mængden af kanaler filtrerne er lavet til at behandle. Funktionens endemål er at udvide alle netværkerne i filtrene med en ikke-redundant standardkomparator på alle mulige måder. Matematisk er funktionen tilsvarende til, at opstille delmængden af det kartesiske produkt mellem alle netværkene i filtrene og alle standardkomparatorerne. Det er en delmængde fordi outputtet skal kun indeholde filtre udvidet med ikke-redundante komparatorer. For eksempel hvis man startede med en liste, der kun indeholdte det tomme filter, som er lavet til tre kanaler og aktiverede *extend()*-funktionen med listen som input, så ville man få en ny liste, der indeholdte tre filtre. Den første filter indeholder komparatoren mellem kanal 0 og 1, den anden har komparatoren for 1 og 2 og den sidste har komparatoren til kanal 0 og 2. Hvis man aktiverede *extend*-funktionen igen, men med listen lavet af den forrige *extend()*-funktionskald, så ville filtret der indeholdte komparatoren mellem kanal 0 og 1 ikke blive udvidet med samme komparator, da den nye komparator ville være redundant. Det ikke-udvidet filter ville ikke blive inkluderet i funktionsproduktet. Funktionens væremåde er afbildet på nedenstående flowchart:

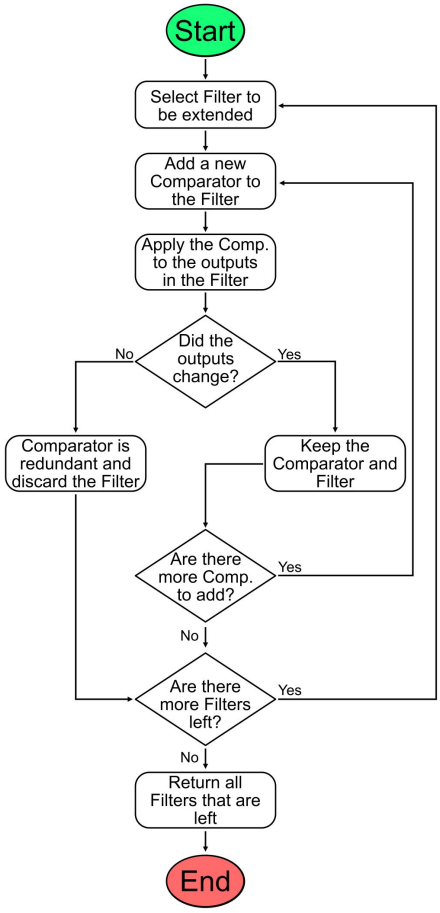

Fig. 1: Trinvis gennemgang af *extend()*-funktionen. Egen tilvirkning

I praksis valgte vi at implementere *extend()*-funktionen ved hjælp af funktionel programmering. Vi benyttede funktionerne *map()* til at danne de individuelle kartesiske produkter, *reduce()* til at sammensætte produkterne til en enkelt liste, og til sidst *filter()*-funktionen til at fjerne de redundante filtre. Desværre var der nogle udfordringer med at bruge funktionel programmering. Funktionerne *map()* og *filter()* understøtter brug af logiske operatorer, såsom *if*-sætninger og sammenligninger. I vores specifikke tilfælde vil vi kun konstruere og beholde filtre med ikke-redundante komparatorer, hvor vi forkaster resten. I vores implementering med *map()*-funktionen, så hvis man kun tog højde for de relevante komparatorer, så vil funktion-

sproduktet af filtrene og de redundante komparatorer være lig med Pythons *None*-datatype, da de ikke er defineret i den anonyme funktion. For at undgå at skulle behandle *None*-datatypen, så valgte vi at definere funktionsproduktet, filtrene og de redundante komparatorer til, at være lig med den tomme liste. Vi vælger at bruge den tomme liste, da det kan nemt af frasortere fra resten af funktionsproduktet ved hjælp af *filter()*-funktionen, og dertil forebygger at vi ikke blander for mange datatyper sammen. Hvis man var mere erfaren inden for funktionel programmering, kunne man højst sandsynligt opstille en pænere løsning, men vores første prioritet er at lave en funktion der virker og spiller sammen med resten af modulerne.

6 Opbygning af *network_finder.py*

Modulet *Network_finder.py*-modulet har til formål, at være en rugergrænsefladen, hvor en brugeren kan opstille et sorteringsnetværk for en mængde kanaler de selv bestemmer. Modulet skal kunne behandle inputværdier fra brugeren og til sidst fremvise et sorteringsnetværk i form af samling af Python-kode, der beskriver hvordan netværket fungerer. Brugergrænsefladen er opbygget som et CLI (*Command Line Interface*), hvor der gøres stor brug af Pythons *print*-funktion. I modulet har vi defineret en funktion ved navn *make_sorting_network()*, som opstiller selve sorteringsnetværket til brugeren. Den er inplementeret rekursivt, hvor den skal have en liste med et tomt Filter og mængden af kanaler som startværdi. Funktionen starter med at tjekke om nogen at Filtrene i listen indeholder et sorteringsnetværk ved hjælp af *is_sorting()* og *map()*, hvis det ikke er tilfældet så udvides alle Filtrene i listen med ikke redundante komparatorer ved hjælp af *extend()*-funktionen. Funktionsproduktet af *extend()*-funktion bliver derefter behandlet af *prune()*-funktionen fra *prune.py*, hvor til sidst produktet af *prune()* bliver sendt til at blive behandlet af *make_sorting_network()* igen. Hvis man finder et sorteringsnetværk i listen af Filtre, så returneres det første Filter der har et sorteringsnetværk. Efter man har fundet et sorteringsnetværk til den mængde kanaler som brugeren ønsker, bruges *to_program()* fra netværksmodulet til, at fremvise hvordan sorteringsnetværket virker.

6.1 Køretiden af *network_finder.py*

Køretiden for *make_sorting_network()* er meget svær at beregne. Først ville vi finde ud af, hvad køretiden er for en kørsel (en kanal med 1 filter). Der blev brugt Generativ AI [Ope24] til en estimering på $O(2^n \cdot n)$, som er meget langsom. Dog er køretiden med flere kanaler og flere filtre $O(2^n \cdot n \cdot size(w) \cdot log(size(w))$ hvor $size(w)$ repræsenterer antallet af Filtre. Denne køretid er ekstremt lang. Med $n = 3$ kanaler er antallet af filtre 4, med $n = 4$ filtre er der allerede 340, og med $n = 5$ er antallet af filtre kommet op på cirka 1,2 millioner efter 12 timer, og processen er stadig ikke færdig.

7 Test og evaluering

I denne sektion vil der gennemgås hvordan, vores program blev testet, og dertil også evalueret på baggrund af de forventet resultater. Ved at gøre brug af den kontraktbaserede struktur, som projektet gøre brug af, kan vi teste vores funktioner og moduler systematisk for, at kunne bedømme hvor vidt de er i overensstemmelse med kontrakten og den forventede udfald. Dette ses igennem brug af *DOCTEST*, *preconditions* og de opstille test. De omtalte *DOCTEST* og *preconditions* kan findes til hver defineret funktion i modulerne. De systematiske test findes i filen *test.py*. textitDOCTEST'ene, *preconditions* og testene tilsammen sikre os, at hver enkelt funktion opfører sig som forventet.

7.1 Test af kode

I testen af koden blev der valgt at undlade testen af modulet *network_finder.py*. Da modulet kun håndterer selve brugergrænsefladen og indeholder en enkelt funktion ved navn *make_sorting_network()*. Da opbygningen af modulet besværliggør test af modulet, da kald af funktion *make_sorting_network()* afventer et input fra brugeren, før resten af programmet udføres. I praksis gør det, at testfilen ikke viser alle de andre resultater, før der indgives et input. Derfor har gruppen fravalgt at teste funktionen og ser i stedet korrektheden igennem kørsel af modulet adskilt. I testen af *extend()*-funktionen i *generate.py*-modulet lavede gruppen en test, hvor der først bliver vist 3 filtre med en komparator tilføjet. Derefter kaldes *extend()*-funktion og resultat printes trinvist, så hvert enkelt udvidet filter vises. Dette tillader gruppen

at se forskellen, samt at kompleksiteten af *extend()*-funktionen tydeliggøres. Strukturen af hele testfilen er ens til den af første fase. Testene er inddelt efter modul og underopdelt i individuelle funktioner. Dette tillader at gruppen kan se eventuelle forskelle på det kontraktbaserede resultat og resultatet i praksis.

7.2 Evaluering af kode

Ved at gøre brug af systematiske test, kunne gruppen konkludere at dannelsen af et sorteringsnetværk på n-kanaler var muligt, med en praksis øvre grænse på 5 kanaler. Da køretiden vokser superekseponentielt med antallet af kanaler, var det ikke muligt for gruppen at få dannet et netværk på 5 kanaler. Der blev forsøgt med at lade programmet kører i 38 timer, og den sidste iteration var endnu ikke gennemført, efter en estimering af den asymptotiske køretid, indså gruppen at der ville gå meget lang tid før den kunne færdiggøres. Som tidligere nævnt ramte programmet 1,2 millioner i antallet af filtre, derfor tager programmet så lang tid, da den skal udvide 1,2 millioner filtre. Selvom gruppen ikke definitivt kunne sige at programmet virkede for 5 kanaler, blev der i projektbeskrivelsen for fase 2 også nævnt, at det var forventeligt at det kunne køres op til 4 eller 5 kanaler [Cru24]. Dermed kunne gruppen uden yderligere indsats sikre sig, at opgaven var løst. I fase 3 forventes der, at der implementeres et modul ved navn *prune.py* som ville varetage *prune*-delen af *generate-prune* algoritmen. Dette ville drastisk reducere køretiden, da der i denne fase kun kaldes på *extend()*-funktionen. Dette gør at programmet danner et sorteringsnetværk, og dertil udvider netværket, uden at optimere netværket yderlige, dette sænker køretiden voldsomt, da der skal testes med det flere kombinationer.

7.3 Valg af løsningsforslag

Overordnet havde gruppen succes med vores fremgangsmåde og strategi. Selvom gruppen som helhed ikke er lige så erfaren med rekursiv og funktionel programmering, opstillede vi et sorteringsnetværk op til 4 kanaler. Ud fra resultaterne af vores test, undgik vi at lave uønsket bivirkninger i vores forskellige funktioner. Dette skyldes både at vi er blevet mere erfaren og har en dybere forståelse af projekter, men også fordi vi gjorde brug rekursiv og funktionel programmering, der som udgangspunkt ikke laver bivirkninger. Her i fase 2 gjorde vi også mindre brug af ChatGPT [Ope24], da vi i højere

grad gjorde brug af de forskellige metoder gennemgået i kurset, i stedet for at få ChatGPT til at opstille kodeforslag. Der var dog tilfælde, hvor vores erfaring inden for funktionel programmering blev udfordret, hvor vi endte med at opstille fungerende men ikke pæne løsninger. Dog da vores test er i overensstemmelse med kontrakten forventer vi, at vores moduler vil være tilstrækkelige i den næste fase.

8 Konklusion

I denne rapport er den anden fase i det tredelt eksamensprojekt blevet gennemgået. Der er blevet programmeret tre moduler ved navn *filter.py*, *generate.py* og *network_finder.py*. Disse moduler kan sammen med første fase, danne et sorteringsnetværk på n antal kanaler. Funktionaliteten af de tre moduler er blevet vist igennem systematiske tests. Ydeligere så er gruppens fremgangsmåde, valg og tankegang blevet forklaret og diskuteret. Ved at evaluere fase 2 igennem de systematiske test, ses der at det færdige produkt, løser problemstillingen tilfredsstillende, og danner derved grundlag til fase 3.

9 Litteraturliste

[Azu24] AzureX. *SDU-Programmeringsprojek*. Accessed: 08-11-2024. 2024. URL: https://github.com/xXmemestarXx/SDU-Programmeringsprojekt/tree/AzureX/.

[Cru24] Luis Cruz-Filipe. *Gruppeprojekt - Efterår 2024 — Introduktion*. PDF document. 2024.

[Ope24] OpenAI. *ChatGPT*. Accessed: 08-11-2024. 2024. URL: https://openai.com/.

10 Bilag og kildekode

10.1 filter.py

```
"""
filter.py

Written by Hlynur, Mathias and Valdemar H8G03

DM574 Exam project
"""

"""
Importing 3rd party libaries
"""
from dataclasses import *

"""
Importing our lecturer's modules
"""
import network as Netw
import comparator as Comp

@dataclass
class Filter:
    """
    We define a singular Filter as a
    dataclass that contains a network,
    its binary outputs and the amount
    of channels the Filter works on.

    We use a dataclass since it is not
    recommend to use a list containing
    different datatypes
    """
    netw: Netw.Network
    outp: list[list[int]]
    size: int

```

```
36 def make_empty_filter(n: int) -> Filter:
37     """
38     Preconditions: n > 0
39 
40     Returns a filter with an empty network and all binary
41     permutations of length n.
42 
43     DOCTEST
44     >>> make_empty_filter(2)
45     [[], [[0, 0], [1, 0], [0, 1], [1, 1]]]
46     """
47     net = Netw.empty_network()
48     out = Netw._all_binary_inputs(n)
49     size = n
50     return Filter(net,out,size)
51 
52 def net(f: Filter) -> Netw.Network:
53     """
54     Returns the Network of a Filter
55 
56     DOCTEST
57     test_filter = make_empty_filter(2)
58     >>> net(test_filter)
59     []
60     """
61     copy = f.netw
62     return copy
63 
64 def out(f: Filter) -> list[list[int]]:
65     """
66     Returns the outputs of a Filter
67 
68     DOCTEST
69     test_filter = make_empty_filter(2)
70     >>> out(test_filter)
71     [[0, 0], [1, 0], [0, 1], [1, 1]]
72     """
73     copy = f.outp
74     return copy
75 
```

```
76  def size(f: Filter) -> int:
77      """
78      Returns the size of the filter
79
80      DOCTEST
81      test_filter = make_empty_filter(2)
82      >>> get_size(test_filter)
83      2
84      """
85      copy = f.size
86      return copy
87
88  def is_redundant(c: Comp.Comparator, f: Filter)-> bool:
89      """
90      Checks if the Comparator, c, would be redundant if it were
91      to be added to the Network in the Filter, f.
92
93      DOCTEST
94      n = 3
95      filt_test = filt.make_empty_filter(n)
96      filt_test = filt.add(2, filt_test)
97      Filter(n=[2], out=[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]])
98      >>> is_redundant(2, filt_test)
99      True
100     """
101     copy_net = net(f)
102     copy_per = out(f)
103
104     copy_net = copy_net + [c]
105
106     new_per = Netw.outputs(copy_net, copy_per)
107     return f.outp == new_per
108
109 def add(c: Comp.Comparator, f: Filter) -> Filter:
110     """
111     Appends a Comparator to the end of a Network in
112     a Filter
113
114     DOCTEST
115     test_comp = Comp.make_comparator(0, 2)
```

```
116     test_filter = make_empty_filter(3)
117     >>> test_filter = add(test_comp, test_filter)
118     [[5], [[0, 0, 0], [0, 1, 0], [0, 0, 1], [1, 0, 1], [0, 1, 1], [1, 1, 1]]]
119     """
120     new_net = Netw.append(c,net(f))
121     new_out = Netw.outputs(new_net, out(f))
122     same_size = size(f)
123     return Filter(new_net,new_out,same_size)
124
125
126 def is_sorting(f: Filter) -> bool:
127     """
128     Checks if the network in the filter is a sorting network.
129
130     DOCTEST
131     n = 3
132     filt_test = filt.make_empty_filter(n)
133     filt_test = filt.add(2, filt_test)
134     filt_test = filt.add(5, filt_test)
135     filt_test = filt.add(8, filt_test)
136     Filter(n=[2, 5, 8], out=[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]], size=3)
137     >>> is_sorting(filt_test)
138     False
139     """
140     return (size(f) < 2) or Netw.is_sorting(net(f),size(f))
```

10.2 generate.py

```
1  """
2  generate.py
3
4  Written by Hlynur, Mathias and Valdemar H8G03
5
6  DM574 Exam project
7  """
8
9  """
10 Importing 3rd party libaries
```

```
11 """
12 import functools as Func
13 from dataclasses import *
14
15 """
16 Importing our own and our lecturer's modules
17 """
18 import comparator as Comp
19 import network as Netw
20 import filter as Filt
21
22 """
23 Importing definitions of data structures
24 """
25 Comparator = int
26 Network = list[Comparator]
27 Filter = list[Netw.Network, list[list[int]]]
28
29 def extend(w: list[Filter], n: int) -> list[Filter]:
30     """
31     Preconditions: n > 0 and len(w) > 0
32
33     Adds all non-redundant standard comparators for
34     n amount of channels to the Filters in
35     the input list, w.
36
37     This is equal to calculating a subset of the
38     cartesian product of w and all standard
39     comparators for n amount of channels
40
41     DOCTEST
42     n = 2
43     filt_test = Filt.make_empty_filter(n)
44     w = [filt_test]
45     filt_test2 = Gene.extend(w, n)
46     >>> filt_test2 = extend(w,2)
47     [Filter(n=[2], out=[[0, 0], [0, 1], [1, 1]], size=2)]
48     """
49
50     """
```

```
51      Start by getting all the standard Comparators
52      """
53      stdComp = Comp.std_comparators(n)
54
55      """
56      The following map functions makes a subset
57      of the cartesian product of the input Filters
58      and all standard comparators for n amount of
59      channels. It is a subset since we only keep
60      the non-redundant comparators and the original
61      Filters.
62
63      We need to use the list() function multiple
64      times since Python's version of the map
65      function returns an object.
66      """
67      #carte_prod = list(map(lambda f: list(map(lambda c: _check_and_add(c,f),stdComp)),w))
68
69      carte_prod = list(map(lambda f: list(map(lambda c: Filt.add(c,f)
70                  if not Filt.is_redundant(c,f) else [],stdComp)),w))
71      """
72      Since the lambda function needs to always return some
73      some type value, the lambda function will return
74      the empty list when a Comparator is redundant. If we
75      removed the line then the auxillary function would
76      return None if a Comparator is redundant.
77      """
78
79      combined_filters = Func.reduce(lambda x,y: x+y,carte_prod)
80
81      """
82      Python always requires an else clause when using
83      if-statements in a map. There _add_and_check()
84      returns a empty list when a Comparator is redundant. We
85      remove all the 0's using the filter function
86      """
87      extended_filters = list(filter(lambda x: x != [],combined_filters))
88
89      return extended_filters
```

10.3 network_finder.py

```
"""
network_finder.py

Written by Hlynur, Mathias and Valdemar H8G03

DM574 Exam project
"""

"""
Importing 3rd party libaries
"""
from dataclasses import *

"""
Importing our own and our lecturer's modules
"""
import comparator as Comp
import network as Netw
import filter as Filt
import generate as Gene
import prune as Prun


"""
network_finder is essentially a small Command Line Interface program, or CLI.
Therefore the user interface, UI, and the user experience, UX,
needs to be taking into acoount. The simplest way to do this is by printing
guiding helpful messages, so the user knows when
and how they are using the program wrong.
"""

def make_sorting_network(f: list[Filt.Filter], n: int, i: int) -> Filt.Filter:
    """
    Preconditions: n > 0 and len(f) > 0

    Checks if there is one or more sorting networks in the filter list, and then returns t
    """
    if any(list(map(Filt.is_sorting, f))):
```

```
39          return list(filter(lambda x: Filt.is_sorting(x) == True, f))[0]
40
41      i = i + 1
42      extended_filters = Gene.extend(f, n)
43      clean_list = Prun.prune(extended_filters, n)
44      return make_sorting_network(clean_list, n, i)
45
46  done = False
47
48  while not done:
49      print(
50          """
51          .----------------------------------------.
52          |                                        |
53          |   Welcome to Network Finder program |
54          |     by Hlynur, Mathias & Valdemar  |
55          '----------------------------------------
56          """)
57
58      print("Tip: Time needed to calculate is proportional to amount of channels ")
59
60      channel_amount = int(input("Please enter how many channels you want to sort: "))
61
62      while 1 > channel_amount:
63          """
64          Checks wheter the input is valid number or not
65          """
66          if 0 >= channel_amount:
67              print("Please enter a number greater than zero")
68
69          else:
70              print("Invalid input")
71          channel_amount = int(input())
72
73
74      all_filters = [Filt.make_empty_filter(channel_amount)]
75
76      print(f"Finding a sorting network for {channel_amount} channels... \n")
77
78      sorting_network = make_sorting_network(all_filters, channel_amount, 0)
```

```
79
80      print(f"Found a sorting network for {channel_amount} channels with size {Netw.size(Fil
81
82      print(f"An implementation of the sorting network in Python would look like: \n")
83
84
85      program_string = Netw.to_program(Filt.net(sorting_network),'','')
86
87      for i in range(0,len(program_string)):
88          print(program_string[i])
89
90
91      done = True
```

10.4 test.py

```
1  import comparator as Comp
2  import network as Netw
3  import filter as Filt
4  import generate as Gene
5
6  print(f"")
7  print(f"------- test filter.py -------")
8  print(f"")
9  print(f"------- test 1 begin -------")
10 print(f"")
11 print(f"Testing make_empty_filter()")
12 print(f"")
13
14 filt_test = Filt.make_empty_filter(0)
15 filt_test2 = Filt.make_empty_filter(2)
16 print(f"{filt_test} => {filt_test}")
17 print(f"{filt_test2} => {filt_test2}")
18
19 print(f"")
20 print(f"------- test 1 end -------")
21 print(f"")
22
```

```
23 print(f"------- test 2 begin -------")
24 print(f"")
25 print(f"Testing net()")
26 print(f"")
27
28 comp_test = Comp.make_comparator(0, 1)
29 filt_test = Filt.make_empty_filter(0)
30 filt_test2 = Filt.make_empty_filter(2)
31 filt_test2 = Filt.add(comp_test, filt_test2)
32 print(f"{filt_test} => {Filt.net(filt_test)}")
33 print(f"{filt_test2} => {Filt.net(filt_test2)}")
34 print(f"")
35 print(f"------- test 2 end -------")
36 print(f"")
37
38 print(f"------- test 3 begin -------")
39 print(f"")
40 print(f"Testing out()")
41 print(f"")
42
43 filt_test = Filt.make_empty_filter(0)
44 filt_test2 = Filt.make_empty_filter(2)
45 print(f"{filt_test} => {Filt.out(filt_test)}")
46 print(f"{filt_test2} => {Filt.out(filt_test2)}")
47 print(f"")
48 print(f"------- test 3 end -------")
49 print(f"")
50
51 print(f"------- test 4 begin -------")
52 print(f"")
53 print(f"Testing size()")
54 print(f"")
55
56 filt_test = Filt.make_empty_filter(0)
57 filt_test2 = Filt.make_empty_filter(2)
58 print(f"{filt_test} => size(filter) => {Filt.size(filt_test)}")
59 print(f"{filt_test2} => size(filter) => {Filt.size(filt_test2)}")
60 print(f"")
61 print(f"------- test 4 end -------")
62 print(f"")
```

```
63
64 print(f"------- test 5 begin -------")
65 print(f"")
66 print(f"Testing is_redundant()")
67 print(f"")
68
69 filt_test = Filt.make_empty_filter(3)
70
71 filt_test2 = Filt.make_empty_filter(3)
72 filt_test2 = Filt.add(2, filt_test)
73
74 # Doing this to minmize length of the statement
75 print_help = Filt.is_redundant(2, filt_test)
76 print_help2 = Filt.is_redundant(2, filt_test2)
77
78 print(f"{filt_test} => Testing a single comparator => {print_help}")
79 print(f"{filt_test2} => Testing duplicated comparators => {print_help2}")
80 print(f"")
81 print(f"------- test 5 end -------")
82 print(f"")
83
84 print(f"------- test 5 begin -------")
85 print(f"")
86 print(f"Testing add()")
87 print(f"")
88
89 filt_test = Filt.make_empty_filter(2)
90 filt_test2 = Filt.make_empty_filter(2)
91 filt_test2 = Filt.add(2, filt_test2)
92
93 print(f"{filt_test} => add(2, filter) => {filt_test2}")
94
95 print(f"------- test 5 end -------")
96 print(f"")
97
98 print(f"------- test 6 begin -------")
99 print(f"")
100 print(f"Testing is_sorting()")
101 print(f"")
102
```

```
103 filt_test = Filt.make_empty_filter(4)
104 filt_test = Filt.add(2, filt_test)
105 filt_test = Filt.add(5, filt_test)
106 filt_test = Filt.add(8, filt_test)
107
108 filt_test2 = Filt.make_empty_filter(3)
109 filt_test2 = Filt.add(2, filt_test2)
110 filt_test2 = Filt.add(5, filt_test2)
111 filt_test2 = Filt.add(8, filt_test2)
112
113 print(f"{filt_test} => is_sorting(filter) => {Filt.is_sorting(filt_test)}")
114 print(f"{filt_test2} => is_sorting(filter) => {Filt.is_sorting(filt_test2)}")
115
116 print(f"------- test 6 end -------")
117 print(f"")
118
119 print(f"")
120 print(f"------- test generate.py -------")
121 print(f"")
122 print(f"------- test 1 begin -------")
123 print(f"")
124 print(f"Testing extend()")
125 print(f"")
126
127 F1 = Filt.add(2,Filt.make_empty_filter(3))
128 F2 = Filt.add(5,Filt.make_empty_filter(3))
129 F3 = Filt.add(8,Filt.make_empty_filter(3))
130
131 all_filters = [F1,F2,F3]
132
133 for i in range(0,len(all_filters)):
134     print(f"Filter {i+1} network:{Filt.net(all_filters[i])} => {Filt.out(all_filters[i])}\
135
136 new = Gene.extend(all_filters,3)
137
138 print("After Extending")
139
140 for i in range(0,len(new)):
141     print(f"Filter {i+1} network:{Filt.net(new[i])} => {Filt.out(new[i])}\n")
142
```

```
143 print(f"")
144 print(f"------- test 1 end -------")
145 print(f"")
```

Programmeringsprojekt: Fase 3

Hlynur Æ. Guðmundsson, Mathias B. Jensen & Valdemar B. Reib

Fasekoordinator - Mathias B. Jensen

Institut for Matematik og Datalogi, Syddansk Universitet

2024-12-20

Indholdsfortegnelse

1 Problemstilling

I denne rapport behandles fase 3 af eksamensprojektet i kurset DM574: Introduktion til programmering. I fase 3 skal produktet fra de forgående faser optimeres ved, at implementere et modul ved navn *prune.py*, der vil forkorte køretiden på programmet og muliggøre behandling af flere kanaler.

2 Introduktion

I dette projekt er problemstillingen tredelt i faser, som muliggøre systematisk problemløsning. I tredje fase har vi som gruppe fået til opgave at udvikle et enkelt modul ved navn *prune.py* som indeholder tre forskellige funktioner ved navn *boring_prune()*, *score_prune()* og *vector_prune*. Disse tre forskellige funktioner er tre forskellige implementeringer af *prune*-delen af *generate-prune*-algoritmen. Dette modul tillader gruppen at forkorte køretiden af programmet, da der efter implementering ikke længere udvides på redundante netværk, som for eksempel semantiske dubletter.

3 Fremgangsmåde og Strategi

Der er i fase 3 sket flere ændringer til gruppens fremgangsmåde. Vi har i fase 3 valgt at undgå ChatGPT, da den kunne give anlæg til misforståelser af problemstillingen, som gruppen ellers tidligere har fået fortolket gennem ChatGPT. Dertil har gruppen lagt større fokus på imperativ programmering på baggrund af, at de tidligere faser har givet mere erfaring indenfor modulæritet og bivirkninger, så gruppen kan undgå de faldgruber. Dertil er det som tidligere nævnt ændret i det paradigme som gruppen arbejder med. En af de andre væsentlige ændringer kommer sig af gruppens tilgang til samarbejde. Da gruppen i de tidligere faser har uddelegeret arbejdet og gået hvert til sit, med få fysiske fremmøder. Da fase 3 er den mest komplekse og abstrakte fase, ville gruppen øge de fysiske fremmøder, for at gøre det lettere

for hvert gruppemedlem at sparre, og generere ideer til modulet. Igennem feedback har gruppen fået kortlagt at fokuspunktet bliver at skrive bedre kode. Da rapporten har lagt stabilt i begge faser, men kodekvaliteten er gået ned. Derfor gøres der større brug af fysiske fremmøder. Ligesom de sidste 2 faser har gruppen gjort brug af *Git* og *Github* [Azu24] for at dele arbejdet og holde styr på de forskellige versioner af modulet. Selvom brugen ligeledes sidst ikke er perfekt.

4 Programmering af *prune.py*

I denne sektion vil der gennemgås de forskellige løsningsforslag til problemstillingen. Disse løsningsforslag er henholdsvis *boring_prune*, *score_prune* og *vector_prune*. Disse løsningsforslag har hver deres styrker og svagheder, som gennemgås i evalueringsafsnittet. Nedenstående vil udviklingen af de 3 implementeringer forklares og uddybes.

4.1 *boring_prune()*

Den første implementering der blev kreeret var *boring_prune*, som har til formål at fjerne semantiske dubletter, som en metode til at skære køretiden af programmet ned. Gruppens implementering gør brug af en liste med booleske værdier, og en ekstra tom liste. Der løbes igennem den givne liste, og dertil tjekkes om det pågældende index og det næste index i listen har de samme output i filtret. Ved at gøre dette og rette de booleske værdier i den booleske liste, opnås der til sidst en liste med sandt og falsk, alt efter om det skal beholdes eller ej. Dette bruges i sidste ende til at tilføje de pågældende filtre der skal beholdes til den ekstra liste, og den returneres så. Denne metode blev inspireret af en tidligere øvelsestime i programmering, hvor opgaven lød på at implementere Eratosthenes' si for at finde primtal. Selve strukturen gør brug af en boolesk liste, og dertil gennemgås listen systematisk. Ved at ændre pladserne i den booleske liste efter, om tallet er et primtal eller ej, opnås der samme resultat med en liste, der kun indeholder primtal og i gruppens tilfælde de filtre, der ikke har semantiske dubletter.

4.2 Vores egne *prune*-metoder

For at kunne opstille praktiske og effektive *prune*-metoder, valgte vi at undersøge væremåden af selve opstillingen af sorteringsnetværkene og hvilke sammenhænge der indgår.

En af de vigtigste sammenhænge er sammenspillet mellem mængden af komparatornetværk, der beholdes hver gennemgang af *extend()* og *prune()*-cyklussen, og størrelsen af det endelige sorteringsnetværk. I vores uformelle undersøgelser observerede vi, at mængden af komparatorer over det optimale antal [Cod+14], eller afvigelse som vi definerer det, steg i takt med, jo færre komparatornetværker programmet beholdte hver cyklus. Den logiske forklaring på dette er, at vores implementering af *boring_prune()* også fjernede komparatornetværk, der havde potentialet til at blive en af de optimale sorteringsnetværker i forhold til den givne størrelse. Derfor vil det være en stor fordel at designe vores *prune*-metoder således, at programmet i højere grad beholder komparatornetværk, der har potentialet for at blive til det mindst mulige sorteringsnetværk, og frasorterer dem med lavt potentiale.

Til gengæld er det et vilkår, at der vil opstå afvigelser i resultaterne, når man inkluderer heuristikker og andre logiske genveje. Jo mere omfattende heuristikker vi benytter i vores design, jo mere regnetid kan vores program spare i sidste ende, men til gengæld vil der også være en større mulighed for, at vi opnår et resultat, der har en meget stor afvigelse sammenlignet med den optimale mængde af komparatorer. Der er ikke noget definitivt svar på, hvad en god balance mellem tid sparet og afvigelse er, da vi ikke har andre implementeringer af sorteringsnetværker vi kan sammenligne med. Derfor vil vi først kunne finde en fornuftig balance, når vi har lavet de første funktionelle prototyper af vores egne *prune*-metoder. Vores ambition er, at vores program kan opstille optimale netværk til flere kanaler, end vi kunne i fase 2 af projektet, og opnå en hurtigere køretid end *boring_prune()*.

4.2.1 *score_prune()*

Vi fik to ideer til *score_prune()*

- En algoritme baseret på den grådige algoritme, hvor vi frasortere alt bortset fra det filtrer der på nuværende tidspunkt har den største sandsynlig for, at blive til et sorteringsnetværk.
- En algoritme, der frasortere alle filtrer, der har gentagne komparatorer. Siden sorteringsnetværk der indeholder unikke komparatorer er en delmængde af alle sorteringsnetværk, så kan man spare tid ved at frasortere de andre.

4.2.2 *score_prune*

Score_prune var vores første egen implementering af *prune* og virker ved, at afprøve netværkene i filterne på forskellige lister og tildele dem en score, som afgøre hvilke filtre skal fjernes. *Score_prune* gemmer ikke en procent af filterne, men det gemmer de filtre som har en score tæt på high scoren. Antallet af filtre, som bliver fjernet, varierer fra iteration til iteration, fordi vi ikke fjerner en fast procentdel. Ved at kigge nærmere på hvordan *Score_prune* virker, når filterlisten bliver større, kan man se at scoren for hvert filter kommer tættere på hinanden. Gruppen har nogle teorier om hvorfor det sker, den bedste teori er at filterne bliver alt for ens i senere iterationer, og derfor får en ens score, fordi *Score_prune* gemmer for ens filtre i tidligere iterationer. Vi undersøgte det ikke mere i dybden fordi vi endte med at fokusere på *vector_prune()* som virkede bedre for flere kanaler.

Der er nogle hjælpefunktioner som giver en score til filterne, den første som kører er *calc_score()* som finder ud hvor høj filterens scorer er. Denne funktion er implementeret med *map()*-funktionen og gemmer det i *score_list*. I næste linje bruger vi *score_list* med *reduce()*-funktionen til at finde den højeste score af alle filterne, som vi bruger i næste del med en *while*-løkke til at finde alle filterne som har en score tæt på high scoren og gemmer det i den returnerede liste.

4.2.3 *Calc_score(), how_sorted() og make_list()*

Calc_score() bruger 2 hjælpefunktioner til at opbygge 5 forskellige lister til at teste filterne på. Den første er *make_list()* som returnerer en liste af 5 lister af længde n, hvor n er antallet af kanaler. Vi bruger så *how_sorted()* til at få scoren fra alle listerne med deres respektive 'weights', fordi vi ville sikre at listerne varierer indbyrdes. *Make_list()* kreerer 5 forskellige lister af længde n, som ses nedenstående.

- Liste 1: er en omvendt liste
- Liste 2: veksler mellem ulige numre og 0
- Liste 3: binær liste som begynder på 0
- Liste 4: sorteret liste
- Liste 5: tager halvdelen af listen og laver den omvendt og gør det samme med den anden halvdel

Disse lister ses repræsenteret visuelt nedenfor.

For n channels og hvor m = n//2
Liste 1: [n, n-1, n-2, ..., 1]
Liste 2: [1, 0, 3, 0, ..., n]
Liste 3: [0, 1, 0, 1, ..., 0]
Liste 4: [0, 1, 2, 3, ..., n]
Liste 5: [m, m-1, ..., 1, n, n-1, ..., m+1]

Fig. 1: De forskellige lister

How_sorted() er en simpel rekursiv funktion som tjekker hvert element med den næste, hvis den første er lavere, får den et point.

4.2.4 Variable variation

I *score_prune* er det nogle variabler som kan ændres til at få forskellige resultater. Det er svært at finde ud hvilke konfiguration af de variabler der er bedst. I *make_list()* har vi forskellige vægt for hver liste, årsagen til det er at nogle lister er mere kompliceret at sortere, for eksempel den omvendte liste og den halv omvendte liste, men vi ville også have de andre lister med til at "udvande" scoren og til at få mere mangfoldighed.

Det andet sted som har variabilitet, er variablen *score_prox* i *score_prune()*-funktionen, denne variable er vigtig for at smide filterne væk, når listen af filterne bliver større.

4.2.5 *vector_prune()*

En anden og mere matematisk tilgang til at bedømme og beholde filtrer efter deres potentiale, er ved at gøre brug af vektormatematik. Som beskrevet i kodekontrakten [Pro24] indeholder hvert filter en liste af lister af heltal, som er en repræsentation af dens tilhørende komparatornetværk og output. Så længe repræsentationerne er i overensstemmelse med komparatornetværkene og er en god beskrivelse, kan man bedømme filtrer på baggrund af deres output. Matematisk set er en liste af lige lange lister svarende til en matrice, og dermed vil man, i teorien, kunne sammenligne filtrer ved hjælp af metoder, som for eksempel boolesk matricemultiplikation. Et eksempel af hvordan matricerepræsentationen af et filters output kan se ud ses på figur 2.

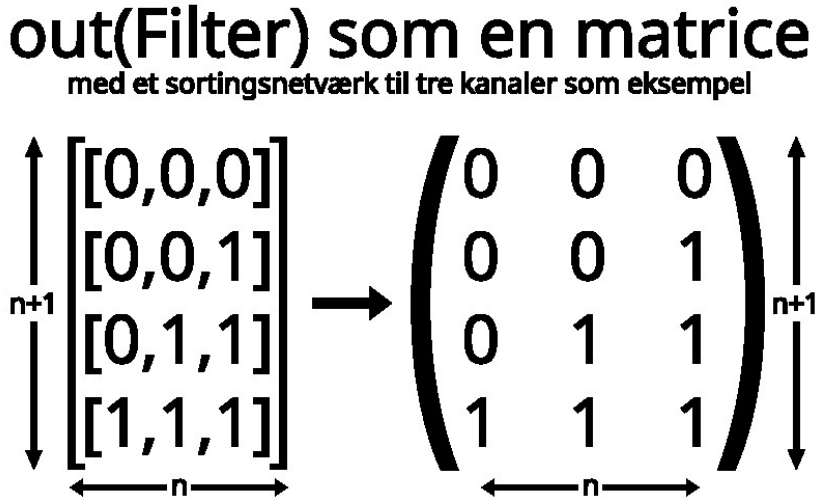

Fig. 2: Matrice repræsentationen af et sorteringsnetværk til tre kanaler

Dog har outputtene i filtrene en opstilling, der forhindrer matriceoperationer. Da *add()*-funktionen fra *Filter*-modulet automatisk forkorter outputtene, når en komparator tilføjes. Dette betyder, at efter hver gennemgang af *extend()*-funktionen vil outputtene i filtrene have forskellig størrelse. Eksempelvis vil matricerepræsentationen af det tomme filter have en højde lig med 2^n, mens et sorteringsnetværk vil have en højde på $n + 1$, hvor n er antallet af kanaler filtrene er lavet til. Idet matriceoperationer påkræver et bestemt størrelsesforhold mellem højden og bredden, vil operationerne dermed være utilgængelige. Man kunne undgå problemstillingen ved at forlænge matricerne således, at de opfylder størrelseskravet, men det vil sandsynligvis være spild af regnekraft, da man skulle gøre det for alle filtre. Et bedre alternativ vil være at omdanne matricerne til vektorer. Man kan opfatte en vektor som en matrice med én enkelt række og modsat kan man også anse en matrice som en samling af vektorer. Man kan undgå, at matricerne har forskellige antal rækker ved at summere vektorerne, så hvert filter har en enkelt sumvektor. Et eksempel kan ses på figur 3

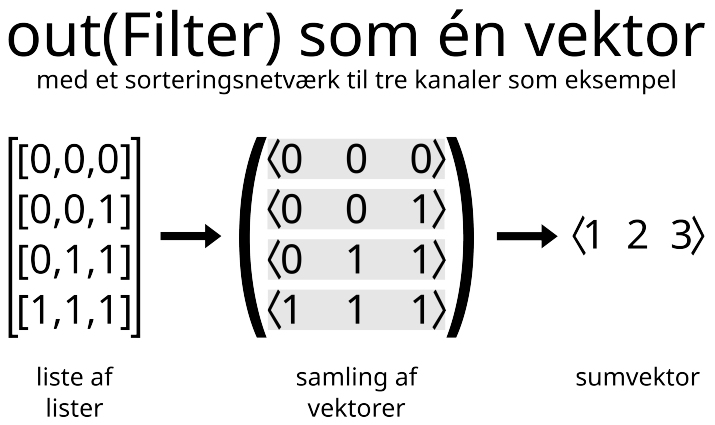

Fig. 3: Vektor repræsentationen af et sorteringsnetværk til tre kanaler

Når man repræsenterer de forskellige filtre som vektorer, vil det tilgængeliggøre nye sammenligningsmetoder. Man kan kvantitativt sammenligne stedvektorer ved at beregne deres afstand til hinanden eller til en konstant vektor. Konstanten i denne sammenhæng kunne for eksempel være sumvektoren dannet af et sorteringsnetværks output, da den har et fast mønster, der afhænger af antallet af kanaler. Man kan dermed beregne, hvor anderledes et filters komparatornetværk er fra et sorteringsnetværk ved at tage udgangspunkt i afstanden mellem deres vektorrepræsentationer. Jo mindre afstanden, jo mere ligner filtrets komparatornetværk et sorteringsnetværk. Denne tilgang har nogle fordele og ulemper. Den første fordel er, at sammenligningsmetoden naturligt vil prioritere filtre med et kort output, da der er færre tal, som summeres. En anden og meget stor fordel er, at metoden vil prioritere, at 0- og 1-tallene i outputtene er på de samme pladser som i sorteringsnetværkets output. I det tilfælde, hvor to filtre har samme størrelse output, vil filtret med det output, som har flest 0'er forrest i listerne af heltal, have en kortere distance fra sorteringsnetværkets sumvektor, eller sammenligningsvektoren, som vi navngiver den. For eksempel i *score_prune* tog vi udgangspunkt i, hvor mange par et komparatornetværk kunne sortere korrekt, og ikke om disse par selv var i rigtige rækkefølge. Vektormetoden vil derimod favorisere, at alle par er sorteret, som i sorteringsnetværkets output. En sidste fordel i praksis er, at metoden tager udgangspunkt i et sort-

eringsnetværks output og ikke i indholdet af komparatornetværkene. Sammenligningsmetoden vægter ikke bestemte komparatorer mere end andre, og vægter i stedet deres sammenspil. Hvis man vægter enkelte komparatorer mere end andre, kan det resultere i, at der opstilles færre potentielle sorteringsnetværker af *extend()*-og-*prune()* cyklussen. I værste tilfælde opstilles der så få, at programmet ikke vil kunne lave det mindste og mest optimale sorteringsnetværk. Derfor vil det være fortrukket at tage udgangspunkt i komparatorernes sammenspil. Til gengæld, har vektormetoden nogle ulemper. Den første er, når man beregner sumvektoren til et filter, at man så ikke kan identificere, hvilket filter vektoren tilhører, idet sumvektorerne, ligesom outputtene, ikke er unikke. Forskellen mellem sumvektorerne og outputtene er, at sumvektorerne ikke er en del af *Filter*-datastrukturen. Dette medfører, at implementeringen af sammenligningsmetoden skal huske, hvilke sumvektorer tilhører hvilke filtre. En anden ulempe er, at sammenligningsmetoden vil være ressourcemæssig dyr. Da man skal opstille outputtene af et sorteringsnetværk og beregne dem og alle filtrenes sumvektor, og til sidst afstandene. På grund af dette kan sammenligningsmetoden hurtigt blive regnemæssigt intens og dermed dårligt egnet til store datamængder. Udover det, er det kun sorteringsnetværkets output, der kan genbruges, mens resten skal beregnes hver cyklus. For at kunne realisere metoden vil det være nødvendigt at kraftigt reducere datamængden ved at beholde forholdsvis få filtrer, hvis man vil finde sorteringsnetværker til flere kanaler hurtigere end *boring_prune*.

I praksis er sammenligningsmetoden, som vi navngiver *vector_prune()* således: Først opstilles sammenligningsvektoren ved hjælp af en simpel *for-each*-lykke. I første prototype blev outputtet af et sorteringsnetværk opstillet med en rekursiv funktion, da outputtene har et fast mønster, som kan ses på nedenstående ligning.

$$out(Sortnet(n)) = [\ ([0]\cdot n),\ ([0]\cdot(n-1)+[1]\cdot 1),\ \ldots,\ ([0]\cdot(n-n)+[1]\cdot n)\ ]$$

$$out(Sortnet(3)) = [\ [0,0,0],\ [0,0,1],\ [0,1,1],\ [1,1,1]\ ]$$

Dog opdagede vi at sumvektoren af outputtene altid er lig med vektoren, der har følgende mønster formel.

$$vec(out(Sortnet(n))) =< 1,2,...,n >$$

$$vec(out(Sortnet(3))) =< 1,2,3 >$$

Derfor kan man beregne sammenligningsvektoren uden at opstille outputtene til et sorteringsnetværk. Derefter beregner vi sumvektoren til hvert filter, som *vector_prune()* modtager fra *extend()*-funktionen. Dette gøres ved hjælp af to *while*-løkker, hvor vi beregner én dimension af den resulterende vektor ad gangen. Dette gentages for hvert filter, hvor til sidst sumvektorerne samles i en liste, som har samme rækkefølge som listen af filtre. På den måde huskes hvilken sumvektor tilhører hvilket filter. I det sidste trin af sammenligningsmetoden, beregnes hver sumvektors afstand fra sammenligningsvektoren ved hjælp af Pythagoras' læresætning, *map()*-funktionen og Pythons indbygget *sum()*-funktion. Funktionerne bruges blandt andet fordi programmet skal finde den kvadratiske afstand i hver dimension og summere dem, som kan implementeres meget kort og præcist med *map()* og *sum()*. Alternativt kan man bruge *reduce()*-funktionen i stedet for *sum()*. En vigtig del af trinnet er, at programmet ikke tager kvadratroden af den summeret kvadratiske afstande. Det gøres fordi vi vil gerne undgå at arbejde med *floating point*-datatypen, som Pythons kvadratrodfunktion vil lave, og foretrækker at alle programmets talværdier er heltal, så de forbliver sammenlignelige. Udover det vil det ikke gøre nogen praktisk forskel, så længe at alle de beregnet afstande er kvadratiske, udover at spare én aritmetisk operation over. Efter at *vector_prune()* har kvantitativt sammenlignet hvor forskellig filtrene er fra et sorteringsnetværk, kan programmet behandle filtrene således at *vector_prune()* beholder de bedste og fjerner de værste. I praksis har vi gjort dette ved, at sortere filtrene fra kortest kvadratisk afstand til længst, og derefter *list slicing* til at beholde den første og bedste brøkdel af filtrene. Det er en forholdsvis dyr og langsom metode, da sortering oftest er tidskrævende, men det er en meget simpel metode, der implementeres og fejlsøges uden et stort tidsforbrug. I programmet benyttes *Bubble-Sort*-algoritmen med lille ændring, så den sortere både listen af filtre og kvadratiske afstande samtidigt, så elementernes en-til-en relation beholdes. Dette gøres ved at sammenligne heltallene i listen af afstande og bytte om på elementerne i begge lister. Når *Bubble-Sort*-algoritmen er færdig bliver den sorteret liste af filtre *sliced* fra den første plads, som indeholder den sumvektor der er tættest på sammenligningsvektoren, til og ikke med en talværdi, som dynamisk beregnes. For at modgå metodens svaghed ved behandling af store datamængder, beholdes en mængde der er invers proportional med længden af inputlisten af filtre. Dette resultere i, at *vector_prune()* beholder forholdsvis mange filtre når inputlisten er kort, og meget få når listen er lang. Dette tillader at *vector_prune()* at altid beholde de matematiske bedste filtre

og holde arbejdsmængden forholdsvis konstant lavt. Et nemmere alternativt ville være at altid beholde dén bedste eller et fast antal, som for eksempel de hundrede bedste. Dog viste vores første prototyper, at beholde et fast antal er dårligt egnet til, hvis *vector_prune()* skal kunne behandle flere kanaler, hvor arbejdsmængderne ikke er lige store. Derfor valgte vi at gøre mængden af filtre vi beholder dynamisk.

Funktionen og metoden *vector_prune()* tager en matematisk tilgang til, at kvantitativt bedømme, hvilke filtrer der skal beholdes i *extend()*-og-*prune()* cyklussen. Den har den grundantagelse at outputtene er en præcis repræsentation af sammenspillet i et komparatornetværk. Det er en kvantitativ metode, som både har fordele og ulemper, som skal udnyttes og behandles i den praktiske implementering. Hvor effektiv den er i praksis kun analyseres, ved at teste den i sammenspil med de andre moduler.

5 Test og evaluering

I denne sektion vil der gennemgås hvordan, vores program blev testet, og dertil også evalueret på baggrund af forventelige resultater. I fase 3 er der kun et modul med en funktion ved navn *prune()* som skal optimere *network_finder.py*-modulet. Funktionen *prune()* gør brug af *boring_prune()*, som er en simpel implementation af *prune*-funktionaliteten. Den anden implementation *score_prune()* er gruppens bud på en god implementation af *prune*-funktionaliteten. Derfor er der ikke stor fokus på den kontraktbaseret struktur, men mere modulariteten. Da de tidligere faser har haft store problemer med modulairteten, da der blev kreeret hjælpefunktioner og de givne funktioner der skulle kreeres til modulet, blev implementeret således at de ikke holdt den kontraktbaseret struktur. Derudover er gruppens fase 2 meget langsom, og har nogle tydelige faldgruber, som først er blevet tydelig gjort gennem feedback, og test af fase 3. Disse fejl rettes til bedste evne og vil benævnes til det mundtlige forsvar.

5.1 Test af kode

I testen af modulet *prune.py* blev der modsat de tidligere faser ikke benyttet en separat testfil, men derimod blev der testet ved hjælp af *print* og *bash*-kommandoen *time*. Den første test af koden sker ved brug af *boring_prune* som er den første og mest simple implementation af *prune*-funktionaliteten. Derefter kommer testen af *score_prune* og *vector_prune*. Nedenstående gennemgås de to metoder for test af koden.

5.1.1 *Print()*

Ved at redigere i *network_finder.py* kunne gruppen printe længden af listen der indeholdte alle filtre, og gentage dette efter hvert funktionskald, da funktionen *make_sorting_network()* er implementeret rekursivt. Dertil printes selv samme liste efter hver funktionskald til *prune()*. Da dette printes side om side, kan gruppen følge med i hvordan at størrelsen af listen henholdsvis vokser og mindskes, hvor der til sidst eksistere et enkelt filter i listen, som er det sorteringsnetværk som returneres som svar. Selve *print*-strukturen ses i sektion 9.2 fra linje 41-49. Alt efter om der ønskes test med *prune* eller uden, kan den udkommenterede linje 48 rettes, og dertil kommenteres linje 42. Dette har gjort det let at teste forskellen, og dertil også se forskellen. Da der kaldes på modulet *prune.py* kan der skiftes mellem de tre forskellige implementationer af *prune* uden at rette noget i *network_finder.py*.

5.1.2 *Time*

For at skabe et overblik over forskellen mellem at bruge *prune.py* og ikke at benytte det, gjorde gruppen brug af en *bash*-kommando ved navn *time* som notere tiden det tager at køre en given kommando. I dette tilfælde bruges syntaksen *time python3 network_finder.py* for at køre programmet med tidsfunktionaliteten. Kommandoen returnerer i afslutning af kørslens tre værdier, som er henholdsvis *real, user, sys* hvor *real* er total tid fra kommandoen eksekveres til den er færdig, dette tal bruges ikke da tiden det tager at indtaste tallet som *network_finder.py* bruger kan varigere. Det er

ikke en stor variation, men for at sikre at tiden der bedømmes udelukkende er total CPU tid, så kombineres tiden fra *user* og *sys*, da *user* er tiden der bruges på overfladen, som er visning på skærmen, hvor *sys* er tiden som CPU'en bruger til at skrive fra register til register. Disse værdier kombineres og danner en graf som tager udgangspunkt i et givent n og køretiden med og uden de forskellige *prune*-implementationer. Denne graf ses nedenstående.

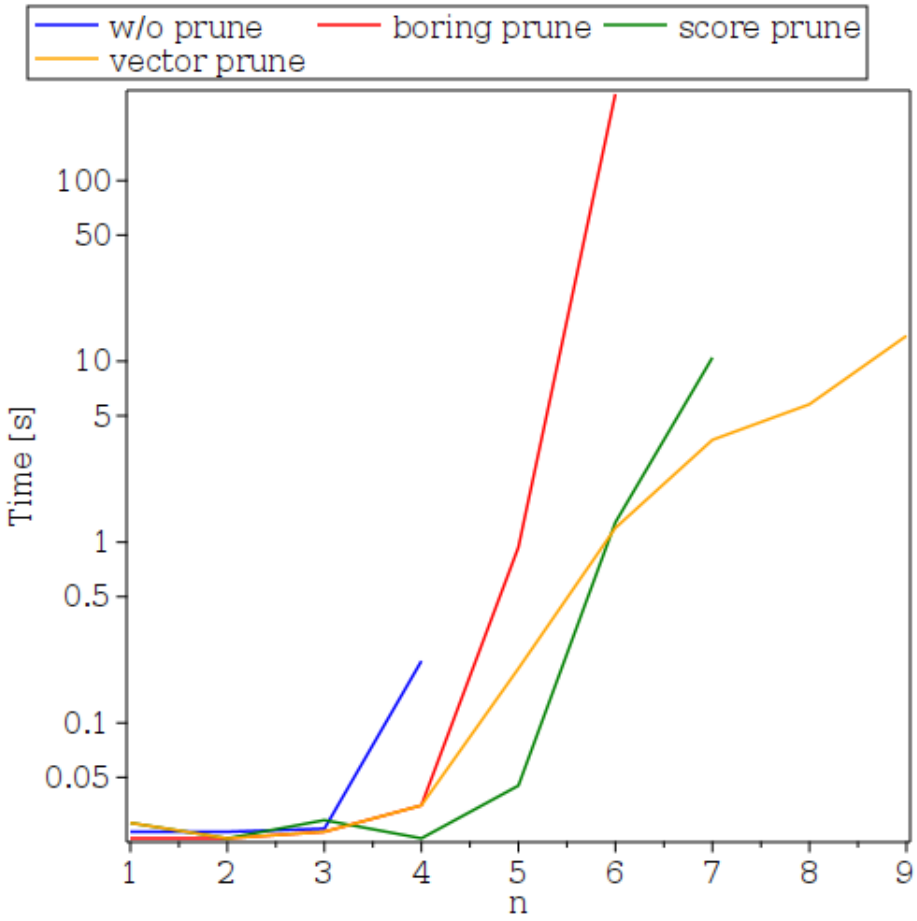

Fig. 4: Samlet køretid for de forskellige implementationer

Der ses tydeligt en forskel i køretiden og dertil også antallet af kanaler der kunne findes et sorteringsnetværk for. Da køretiden af denne algoritme er supereksponentiel, har det ikke været muligt at få et resultat for 5 kanal-

er uden *prune* (*den blå linje*) og 7 kanaler med *boring_prune* (*den røde linje*). Ved den grønne linje ses der en endnu større forskel mellem brug af *score_prune* og kørsel uden modulet. Der observeres at antallet af kanaler, der kunne behandles indenfor rimelig tid opnås inde for kun 10 sekunder, og er dertil også ved $n = 7$, dog kunne $n = 8$ ikke opnås. Den sidste implementation *vector_prune* (*den gule linje*) ses entydigt som den bedste, da den både har tilladt gruppen at finde et sorteringsnetværk for 9 kanaler, med en køretid der er næsten ens til den af *score_prune* for 7 kanaler.

Ved at sammenligne de forskellige implementationer kan man se en tydelig forskel på antallet af kanaler der kan køres, samt køretiden for dette. Der kan altså konkluderes en væsentlig forskel fra fase 2 til fase 3, i det at gruppen har optimeret programmets køretid og dertil som bivirkning har muliggjort behandling af flere kanaler.

5.2 Evaluering af kode

I denne sektion vil produktet af fase 3 evalueres. Der er i testsektionen set en klar forbedring af køretiden og antallet af kanaler. Da køretiden er supereksponentiel er det forventeligt at der ikke kan opnås flere kanaler end 4 eller 5 uden brug af *prune*-modulet. Der ses tydeligt en reduktion af køretid, og antallet af kanaler der er muligt stiger. Grunden til dette er ikke kun optimering af koden, men skyldes primært at der frasorteres en del netværk, og iblandt disse netværk kan man risikere at fjerne et muligt sorteringsnetværk. Der er altså et forhold mellem køretiden og korrektheden. Dette forhold ses nedenstående i tabeller.

5.2.1 *Tabeller*

n	Køretid [s]	Størrelse fundet	Optimale størrelse	Fejl
1	0,025	0	0	0
2	0,025	1	1	0
3	0,026	3	3	0
4	0,220	5	5	0
5	DNF	DNF	DNF	DNF
6	DNF	DNF	DNF	DNF
7	DNF	DNF	DNF	DNF

Tabel 1: *No prune*

n	Køretid [s]	Størrelse fundet	Optimale størrelse	Fejl
1	0,023	0	0	0
2	0,023	1	1	0
3	0,025	3	3	0
4	0,035	6	5	1
5	0,937	10	9	1
6	300,785	15	12	3
7	DNF	DNF	DNF	DNF

Tabel 2: *Boring prune*

n	Køretid [s]	Størrelse fundet	Optimale størrelse	Fejl
1	0,028	0	0	0
2	0,023	1	1	0
3	0,029	3	3	0
4	0,023	5	5	0
5	0,045	9	9	0
6	1,287	15	12	3
7	10,495	20	16	4
8	DNF	DNF	DNF	DNF

Tabel 3: *Score prune*

n	Køretid [s]	Størrelse fundet	Optimale størrelse	Fejl
1	0,028	0	0	0
2	0,023	1	1	0
3	0,025	3	3	0
4	0,035	5	5	0
5	0,200	9	9	0
6	1,199	12	12	0
7	3,679	16	16	0
8	5,787	19	19	0
9	13,833	26	25	1
10	DNF	DNF	DNF	DNF

Tabel 4: *Vector prune*

De ovenstående tabeller viser de forskellige resultater for implementationerne. Kolonnen med fejl er regnet ved at finde differensen mellem størrelsen af det fundne netværk, og den kendte optimale størrelse [Cod+14]. Dette viser tydeligt gruppens fremgang i både køretid og korrekthed. Hvis man udelukkende vurdere på baggrund af de data der er fremlagt, ville den bedste implementation være *vector_prune*. Der vil nedenstående diskuteres andre synsvinkler for hvordan de forskelle implementation kan vurderes.

5.2.2 Diskussion af implementationerne

Der ses i ovenstående afsnit en faktuel forskel på de forskellige implementation hvor den mest effektive og korrekte implementation er *vector_prune*. I dette afsnit vil der udforskes andre parametre at måle en implementation på.

Hvis man kigger på *boring_prune()* i forhold til de fremviste data, kan man tydeligt set at den er mindst effektiv. Dog er den mest simpel og sikker i det at den tager udgangspunkt i sekventiel søgning af listen. Da den er dannet ud fra konceptet om *bubblesort*-algoritmen, vides der med sikkerhed at der ikke overses noget, da hvert enkelt element sammenlignes. Det er en simpel implementering af *prune*-delen af *generate-prune* algortimen, da den bare frasortere en efter en. Dog er denne løsning meget langsom, som tydeligt kan

forberedes ved at bruge nogen heuristikker.

Score_prune() er et rigtig godt eksempel på hvordan man kan forberede en simpel frasortering, ved at aflægge ideen om sekventiel søgning, men at fokusere på en vægting af hvert filters korrekthed. Ved at bruge vægtning som en heuristik kan der tages udgangspunkt i en komparator som objekt, og tillader gruppen af beholde en vis modularitet. Da denne implementering ikke afhænger af andre faktorer end *comparator*-modulet og *network*-modulet for at vægte filteret. Da den følger kontrakten, kan der frit udskiftes mellem forskellige implementeringer af de to moduler.

Vector_prune() er den mest effektive løsning, når man fokuserer på data og statistik. Dog er der flere faktorer der gør implementeringen mindre gunstig. For eksempel gøres der brug af vektormatematik, som implicere en utvetydig tilgang, da der ikke er plads til usikkerheder på samme måde. Det tager udgangspunkt i en repræsentation af komparatorer og ikke datastrukturen i sig selv, som resulterer i at både fremgangsmåde og idéen afhænger meget af kodekontrakten og ikke teorien om sorteringsnetværker. Hvis man valgte at revidere i kodekontrakten, for eksempel *outputs*-funktionen fra komparatormodulet ikke forkorter lister, kunne det resultere i at *vector_prune* ikke længere ville virke korrekt. Funktionen *score_prune()* derimod afhænger ikke i lige så høj grad af selve kodekontrakten.

Ved et kigge på de forskellige implementationer fra andre vinkler, ses der en forskel mellem dem der ikke afsløres af statistikkerne. For at vælge den korrekte løsning skal der tages stilling til hvad der er brug for i problemstillings konteksten, fra opgavebeskrivelsen [Cru24]

5.3 Valg af løsningsforslag

For at vælge det korrekte løsningsforslag til problemstillingen, skal der tages stilling til hver implementations styrke. Det er tydeligt at *vector_prune* er den mest effektive og den med højst korrekthed på baggrund af de data der er fremlagt. Dog er *score_prune* den der er mest modulære, og passer bedst til kontraktens beskrivelse. Den mest sikre, men langsomste implementation er *boring_prune*. Der defineres i projektbeskrivelsen at formålet med fase 3

af projektet er "Fase 3 udvikler denne algoritme med teknikker fra Kunstig Intelligens til at kunne håndtere højere værdier af n"[Cru24]. Hvis der tages udgangspunkt i dette, er det *vector_prune* der er det bedste løsningsforslag til problemstillingen, da dens implementering *prune*-funktionaliteten og kan håndtere den højeste værdi for n.

6 Konklusion

I denne rapport er den tredje og sidste fase i det tredelte eksamensprojekt blevet gennemgået. Der er blevet programmeret et modul *prune.py* som indeholder 3 forskellige implementeringer af *prune*-delen til *generate-prune* algoritmen. Disse 3 implementeringer er blevet testet og omdiskuteret således at der kunne vælges en enkelt implementering *vector_prune* som det bedste løsningsforslag til problemstillingen, da det både gør programmet hurtigere, tillader behandling af flere kanaler, samt at sikre en vis korrekthed.

7 Figurer og tabeller

Liste over figurer

Liste over tabeller

8 Litteraturliste

[Azu24] AzureX. *SDU-Programmeringsprojekt*. Accessed: 20-12-2024. 2024. URL: https://github.com/xXmemestarXx/SDU-Programmeringsprojekt/tree/AzureX/.

[Cod+14] Michael Codish o.fl. "Twenty-Five Comparators Is Optimal When Sorting Nine Inputs (and Twenty-Nine for Ten)". Í: *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*. IEEE, nóv. 2014, bls. 186–193. DOI: 10.1109/ictai.2014.36. URL: http://dx.doi.org/10.1109/ICTAI.2014.36.

[Cru24] Luis Cruz-Filipe. *Gruppeprojekt — Efterår 2024 — Introduktion*. PDF document. 2024.

[GJR24] H. Æ. Guðmundsson, M. B. Jensen og V. B. Reib. *Programmeringsprojekt: Fase 2*. Accessed: 20-12-2024. 2024. URL: https://github.com/xXmemestarXx/SDU-Programmeringsprojekt/blob/main/Fase%202/group03/report03.pdf.

[Pro24] DM574 Group Project. *Introduction to Programming: Group Project — Fall 2024 — Phase III*. Accessed: 2024-12-17. 2024. URL: %5Curl%7Bhttps://github.com/xXmemestarXx/SDU-Programmeringsprojekt/blob/main/Fase%5C%203/project3.pdf%7D.

9 Bilag og kildekode

9.1 prune.py

```
1  import filter as Filt
2  import functools
3  import network as Netw
4
5  def prune(w:list[Filt.Filter], n: int) -> list[Filt.Filter]:
6      """Returns the implementation of prune used, change the function to either
7        'boring_prune', 'score_prune' or 'vector_prune.' to use different implementations.""
8      return vector_prune(w, n)
9
10
11 def boring_prune(w, n):
12     """
13     Returns all filters with unique outputs from input list w.
14     Implementation is based on the sieve of Eratosthenes algorithm.
15     Using a boolean list and two for-each-loops, the function looks
16     through the given list and compare the outputs for the filters,
17     where it crosses out filters if the outputs are non-unique
18
19     reg:
20     len(w) > 0
21     """
22
23     keep = [True for i in range(0, len(w)+1)]
24     pruned = []
25
26     for i in range(0,len(w)):
27         if keep[i]:
28             pruned.append(w[i])
29             for j in range(i,len(w)):
30                 if Filt.out(w[i]) == Filt.out(w[j]):
31                     keep[j] = False
32     return pruned
33
34 def how_sorted(v: list[int]) -> int:
```

```
35      """
36      Gives score to a filter for how well it sorts a list.
37      Every list gets 1 for each two elements that are sorted,
38      a perfectly sorted list gets a score equal to it's length minus one.
39      DOCTEST
40
41      test_list=[3,6,2,4]
42      >>> how_sorted(test_list)
43      2
44
45      """
46      if(len(v)<=1):
47          return 0
48      elif(v[0] <= v[1]):
49          return 1 + how_sorted(v[1:])
50      else:
51          return how_sorted(v[1:])
52
53  def make_list(n: int) -> list[list[int]]:
54      """
55      Makes 5 lists,
56      1st. reverse list: [n, n-1, n-2, n-3,....., 1]
57      2nd. alternating half sorted w. duplicated zeros: [1, 0, 3, 0, 5,...., n]
58      3rd. binary alternating list [0, 1, 0, 1,...., 0]
59      4th. sorted list [0, 1, 2, 3, 4, 5,....., n]
60      5th. reversed halved list [4, 3, 2, 1, 9, 8, 7, 6, 5] if n==9
61
62      Returnes a list of these 5 lists.
63
64      DOCTEST
65
66      >>> make_list(3)
67      [[3,2,1], [1,0,3], [0,1,0], [0,1,2], [1,3,2]]]
68
69      """
70
71      ret_list = [[],[],[],[],[]]
72      ret_list[0] = list(range(n,0,-1))
73      ret_list[1] = list(map(lambda x: x if x%2!=0 else 0, range(1,n+1)))
74      ret_list[2] = list(map(lambda x: 1 if x%2==0 else 0, range(1,n+1)))
```

```
75      ret_list[3] = list(range(n))
76      ret_list[4] = list(list(range(n//2,0,-1)) + list(range(n,n//2,-1)))
77      return ret_list
78
79  def calc_score(f: Filt.Filter,n: int) -> int:
80      """
81      Calculates the total score of a filter by using apply() from network on each list,
82      score is infalted for some lists using weigths.
83
84      DOCTEST
85
86      n = 3
87      Comp_1 = Comp.make_comparator(0, 1)
88      Comp_2 = Comp.make_comparator(1, 2)
89      Filter = Filt.make_empty_filter(n)
90      Filter=Filt.add(Comp_1,Filter)
91      Filter=Filt.add(Comp_2,Filter)
92      >>> calc_score(Filter,3)
93      14.4
94
95      """
96
97      list_weigth=[1, 1.2, 1.5, 2]
98      whole_list= make_list(n)
99      pts_list1 = how_sorted(Netw.apply(Filt.net(f),whole_list[0])) * list_weigth[3]
100     pts_list2 = how_sorted(Netw.apply(Filt.net(f),whole_list[1])) * list_weigth[0]
101     pts_list3 = how_sorted(Netw.apply(Filt.net(f),whole_list[2])) * list_weigth[1]
102     pts_list4 = how_sorted(Netw.apply(Filt.net(f),whole_list[3])) * list_weigth[0]
103     pts_list5 = how_sorted(Netw.apply(Filt.net(f),whole_list[4])) * list_weigth[2]
104     return ( pts_list1 + pts_list2 + pts_list5 + pts_list3 + pts_list4 + pts_list5 )
105
106 def score_prune(w: list[Filt.Filter],n: int) -> list[Filt.Filter]:
107     """
108     Uses a point system to find out which filters will advance,
109     point system consists of using calc_score() to find the
110     highest score of all the filters, and only passsing the
111     ones that are close to that high_score,
112     how close it is proportional to
113     length of w, the larger the list of filters the higher percentage
114     of filters will get pruned.
```

```
115     """
116
117     # creates list by using map and calc_score on w
118     score_list= list(map(lambda x: calc_score(x,n), w))
119     # finds the higest value in score_list
120     high_score= functools.reduce(lambda a,b: a if a>b else b,score_list)
121
122     returned_list=[]
123     i=0
124     score_prox=high_score-(50/len(w))
125     # prunes w by matching scores for each filter with the score_list.
126     while( i < (len(score_list))):
127         if(score_list[i] < score_prox):
128             i=i+1
129         else:
130             returned_list.append(w[i])
131             i=i+1
132
133     return returned_list
134
135 def add_vectors(v: list[list[int]]) -> list[int]:
136     """
137     Adds all the vectors in v and returns a single vector
138
139     reg:
140     all vectors need to be the same length
141
142     len(v) > 0
143     """
144     new_v = [0 for x in range(0,len(v[0]))]
145
146     i = 0
147
148     while i < len(v):
149         j = 0
150
151         while j < len(v[0]):
152             new_v[j] = new_v[j] + v[i][j]
153             j = j + 1
154
```

```
155         i = i + 1
156
157     return new_v
158
159 def make_comparison_vector(n: int) -> list[int]:
160     """
161     Quickly makes the vector that would result
162     from adding all the vectors in the
163     outputs of a sorting network for n channels
164
165     reg:
166     n > 0
167     """
168     res = []
169     for i in range(1,n+1):
170         res.append(i)
171     return res
172
173 def filter_vector(f: Filt.Filter):
174     """
175     Takes a Filters outputs and returns a
176     new vector.
177
178     DOCTEST
179     Filt.out(Filter)= [[0, 0, 0, 0, 0],
180                        [0, 0, 0, 1, 0],
181                        [1, 0, 0, 1, 0],
182                        [0, 0, 0, 0, 1],
183                        [0, 0, 0, 1, 1],
184                        [1, 0, 0, 1, 1],
185                        [0, 0, 1, 1, 1],
186                        [1, 0, 1, 1, 1],
187                        [0, 1, 1, 1, 1],
188                        [1, 1, 1, 1, 1]]
189     >>> filter_vector(Filter)
190     [4, 2, 4, 8, 7]
191     """
192     return add_vectors(Filt.out(f))
193
194 def sqr_eu_dist(cv: list,v: list) -> int:
```

```
195     """
196     Calculates the square distance between two
197     vector using Pythagoras' theorem
198     """
199     return sum(list(map(lambda x,y: (x-y)**2,cv,v)))
200
201 def all_dis(cv: list[int],fv: list[Filt.Filter]) -> list[int]:
202     """
203     Calculates distances between a constant vector and
204     all the vectors generated from a list of Filters
205
206     DOCTEST
207
208     filt_test = Filt.make_empty_filter(5)
209     filt_test = Filt.add(Comp.make_comparator(0,4), filt_test)
210     filt_test = Filt.add(Comp.make_comparator(1,4), filt_test)
211
212     filt_test2 = Filt.make_empty_filter(5)
213     filt_test2 = Filt.add(Comp.make_comparator(2,4), filt_test2)
214     filt_test2 = Filt.add(Comp.make_comparator(1,4), filt_test2)
215     filt_test2 = Filt.add(Comp.make_comparator(2,3), filt_test2)
216
217     fv = [filt_test,filt_test2]
218     cv = make_comparison_vector(5)
219
220     >>> all_dis(cv,fv)
221     [291, 151]
222     """
223     all_distances = []
224     for i in range(0,len(fv)):
225         all_distances.append(sqr_eu_dist(cv,filter_vector(fv[i])))
226     return all_distances
227
228 def sort_dist_and_filt(v: list[int],w: list[Filt.Filter]) -> list[Filt.Filter]:
229     """
230     Bubble sorts the Filter according to their distances from the
231     comparison vector. Their distances is in another list
232     and not part of the Filter data structures themselves,
233     therefore two lists are required.
234
```

```
235     The main difference from bubble sort is that we sort two lists
236
237     req:
238     len(v) = len(w)
239     len(v), len(w) > 0
240
241     DOCTEST
242
243     v = [1,5,8,3]
244     w = [Filter_1,Filter_2,Filter_3,Filter_4]
245
246     >>> sort_dist_and_filt(v,w)
247     [Filter_1,Filter_4,Filter_2,Filter_3]
248     """
249     dis = v[0:] #to avoid changing input list
250     fil = w[0:] #to avoid changing input list
251
252     for i in range(0,len(fil)): #v and w should be equally long
253         for j in range(0,(len(fil)-1)-i):
254             if dis[j] > dis[j+1]:
255                 dis[j], dis[j+1] = dis[j+1], dis[j]
256                 fil[j], fil[j+1] = fil[j+1], fil[j]
257     return fil
258
259 def vector_prune(w: list[Filt.Filter], n: int):
260     """
261     Makes vector from the outputs in the list of Filter,
262     whereafter the function calculates the distances between
263     the vectors and a constant vector created from a sorting
264     networks outputs.
265
266     Then sortes the Filters according to their distances from
267     the comparison vector and returns the first fraction of them
268     determined by the denominator
269     """
270     cov = make_comparison_vector(n)
271
272     dis = all_dis(cov,w)
273
274     pru = sort_dist_and_filt(dis,w)
```

```
275
276     # Slices the list invers proportionally, so we keep fewer
277     # Filters the longer the list of Filter gets
278     return pru[0: int(1/(len(pru))*100000)]
```

9.2 network_finder.py

[GJR24]

```
"""
network_finder.py

Written by Hlynur, Mathias and Valdemar H8G03

DM574 Exam project
"""

"""
Importing 3rd party libaries
"""
from dataclasses import *

"""
Importing our own and our lecturer's modules
"""
import comparator as Comp
import network as Netw
import filter as Filt
import generate as Gene
import prune as Prun


"""
network_finder is essentially a small Command Line Interface program, or CLI.
Therefore the user interface, UI, and the user experience, UX, needs to be taking
into acoount. The simplest way to do this is by printing guiding helpful messages,
so the user knows when and how they are using the program wrong.
"""

def make_sorting_network(f: list[Filt.Filter], n: int, i: int) -> Filt.Filter:
    """
    Checks if there is one or more sorting networks in the filter list,
    and then returns the first sorting network.
    """
    if any(list(map(Filt.is_sorting, f))):
```

```
37          # for i in range(0,len(list(filter(lambda x: Filt.is_sorting(x) == True, f)))):
38          #     print(list(filter(lambda x: Filt.is_sorting(x) == True, f))[i])
39          return list(filter(lambda x: Filt.is_sorting(x) == True, f))[0]
40
41      i = i + 1
42      extended_filters = Gene.extend(f, n)
43      clean_list = Prun.prune(extended_filters, n)
44      print(f"Iteration: {i}")
45      print(f"EXTENDED: {len(extended_filters)}, PRUNED: {len(extended_filters)-len(clean_li
46
47      print("")
48
49      # return make_sorting_network(extended_filters, n, i)
50      return make_sorting_network(clean_list, n, i)
51
52  done = False
53
54  while not done:
55      print(
56          """
57          |                                      |
58          |   Welcome to Network Finder Program |
59          |     by Hlynur, Mathias & Valdemar  |
60          `----------------------------------------
61          """)
62      print("Tip: Time needed to calculate is proportional to amount of channels ")
63
64      channel_amount = int(input("Please enter how many channels you want to sort: "))
65
66      while 1 > channel_amount:
67          """
68          Checks wheter the input is valid number or not
69          """
70          if 0 >= channel_amount:
71              print("Please enter a number greater than zero")
72
73          else:
74              print("Invalid input")
75          channel_amount = int(input())
76
```

```
77
78      all_filters = [Filt.make_empty_filter(channel_amount)]
79
80      print(f"Finding a sorting network for {channel_amount} channels... \n")
81
82      sorting_network = make_sorting_network(all_filters, channel_amount, 0)
83
84      print(f"Found a sorting network for {channel_amount} channels with size {Netw.size(Fi
85
86      print(f"An implementation of the sorting network in Python would look like: \n")
87
88
89      program_string = Netw.to_program(Filt.net(sorting_network),'','')
90
91      for i in range(0,len(program_string)):
92          print(program_string[i])
93
94
95      done = True
```

9.3 make_sorted_outputs.py

```
"""
make_sorted_outputs.py

Written by Valdemar H8G03

DM574 Exam project
"""

"""
Importing 3rd party libaries
"""
from dataclasses import *

"""
Importing our own and our lecturer's modules
"""
import comparator as Comp
import network as Netw
import filter as Filt
import generate as Gene
import prune as Prun

def make_sorted_outputs(n: int):
    """
    makes a sorted lists of lists of ints that
    are equal to the outputs in a sorting network
    for n amount of channels
    """
    if n == 0:
        return []
    else:
        base = [0 for i in range(0,n)]

        #print(base)

        whole = [base]

        return whole + _make_sorted_outputs(whole[0],n-1)
```

```
39
40 def _make_sorted_outputs(v: list[int], n: int):
41     """
42     Auxillary function which takes a list of 0's and
43     changes the n'th index to 1 and then calls itself
44     """
45     new_base = v[0:]
46     new_base[n] = 1
47     new_n = n-1
48
49     #print(new_base)
50
51     if new_n == -1:
52         return [new_base]
53     else:
54         return [new_base] + _make_sorted_outputs(new_base,new_n)
55
56 # print(make_sorted_outputs(0))
57
58 # print(make_sorted_outputs(1))
59
60 # print(make_sorted_outputs(2))
61
62 # print(make_sorted_outputs(3))
63
64 # print(make_sorted_outputs(4))
```