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.

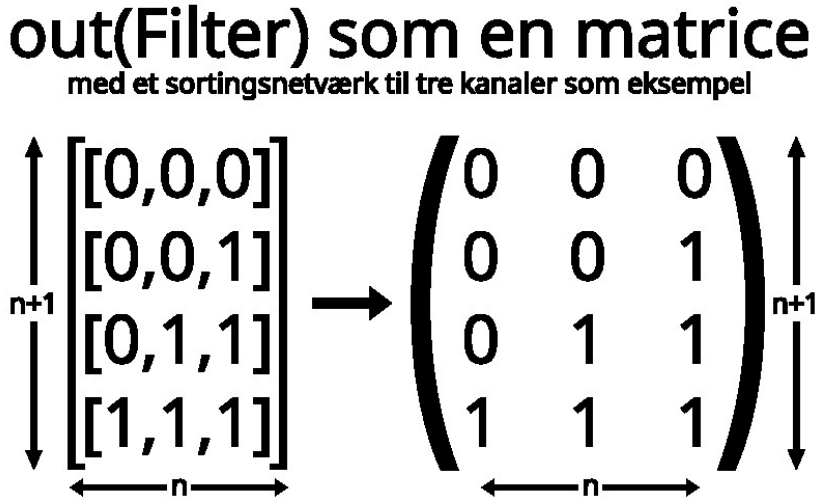

Fig. 2: Matrice repræsentationen af et sorteringsnetværk til tre kanaler

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

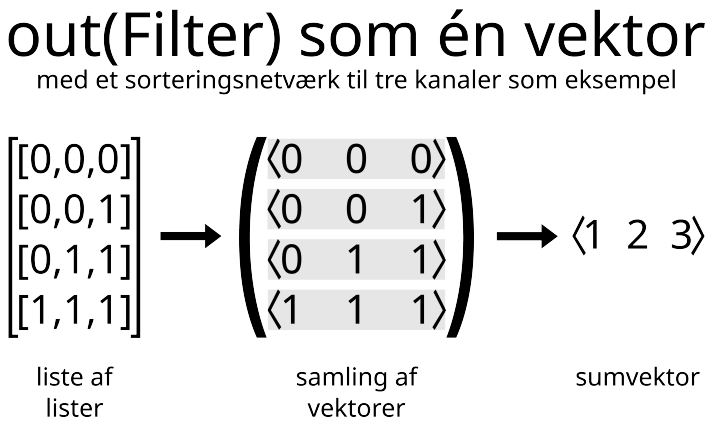

Fig. 3: Vektor repræsentationen af et sorteringsnetværk til tre kanaler

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.

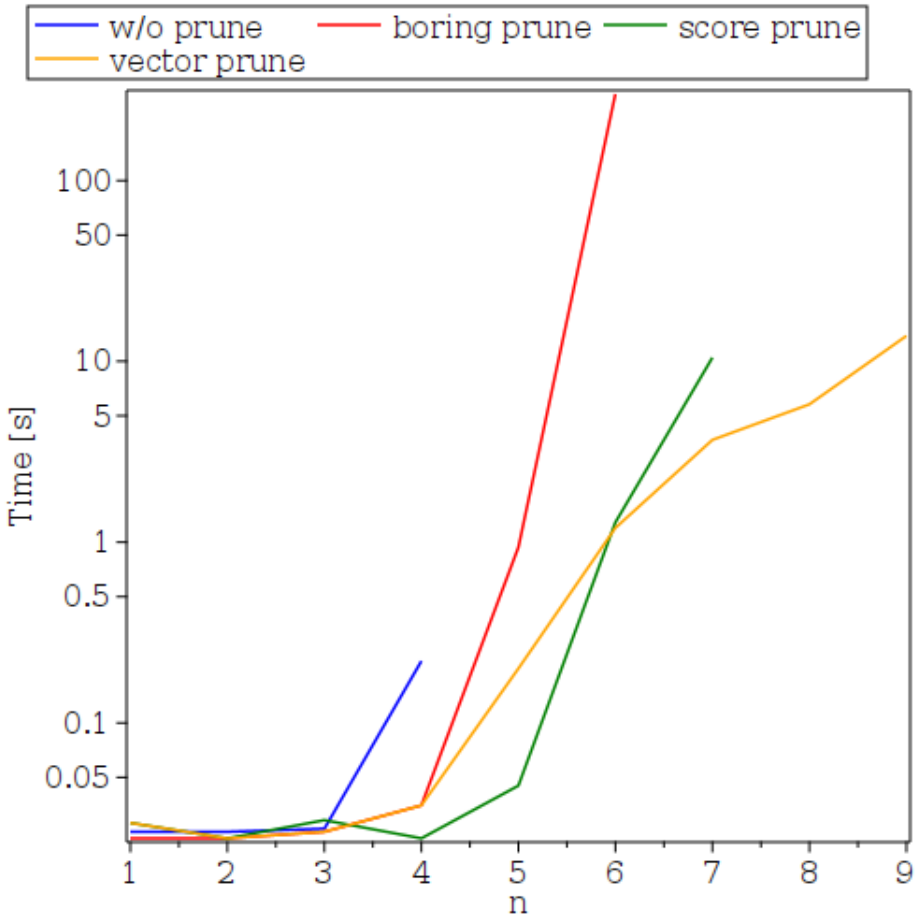

Fig. 4: Samlet køretid for de forskellige implementationer

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))
```