Optimering af passagerkilometer i tog
Førsteårsprojekt - Online planlængning 62a

Nicolai Aarestrup Jørgensen - *nicjo14@student.sdu.dk*
Danny Rene Jensen - *danje14@student.sdu.dk*
Martin Stig Bondesen - *marbo14@student.sdu.dk*
Tenna Boberg Laursen - *telau14@student.sdu.dk*
Caroline Berntsen Knudsen - *cknud14@student.sdu.dk*

2. juni 2015

Resumé

Denne rapport tager afsæt i pladsreservationsproblemet i tog, hvor vi fokuserer på at optimere passagerkilometer. Problemet består i, at en række forespørgsler om rejser, givet med start- og slut-station, skal håndteres i forhold til placering i et tog. Toget kører en lige strækning med et antal stationer, og med et antal sæder.

Til at løse problemet har vi set på to versioner af problemet: *fair* og *unfair* håndtering af input, hvor fair dækker over, at en passager, der *kan* placeres på tidspunktet for forespørgslen, får en plads uanset dennes sammenhæng med øvrige passagerer.

Den unfair version af problemet har vi forsøgt at løse ved flere forskellige *offline* metoder. Offline dækker over, at alle forespørgsler først hentes ind, inden en beslutning om fordeling af passagererne træffes. Til den fair version af problemet har vi udviklet to forskellige algoritmer, hvoraf den ene kun arbejder *online*, og den anden arbejder halvvejs online, halvvejs offline. Online håndtering vil sige, at input ikke bearbejdes i sin helhed, men at forespørgsler derimod bliver behandlet, som de kommer i løbende rækkefølge.

For at afgøre, hvilke algoritmer, der klarer sig bedst i forhold til udbytte (her passagerkilometer), har vi analyseret disse og sammenlignet dem med hinanden ved forskellige tests på tilfældigt data. Det har af disse vist sig, at de to online algoritmer klarer sig næsten lige godt gennemsnitligt. De offline algoritmer klarer sig naturligvis langt bedre. Resultaterne vidner om, at *fairness*-kravet i høj grad påvirker udfaldet.

Forord

Denne rapport er blevet udarbejdet som et led i førsteårsprojektet på SDU. Her har hver enkelt studerende på Det Naturvidenskabelige Fakultet afgivet ønsker om, hvilket projekt de ville arbejde med, hvorefter de studerende er blevet sat i grupper efter projektønske. Man har dermed ikke haft direkte indflydelse på gruppesammensætningen, men man fik derimod mulighed for at få udbygget sit netværk på universitet, fagligt såvel som socialt. Det har i vores tilfælde været en overvejende god oplevelse, da vi alle er gået til opgaven med et åbent sind. Derudover har vi fået god vejledning og støtte af vores projektvejleder Lene Monrad Favrholdt, som har kommenteret vores projekt undervejs og vejledt os, når der har været problemer.

God læsning.

Indhold

1 Indledning

Vi har i dette projekt arbejdet med emnet *online planlægning*, som indbefatter algoritmer, der får et input stykke for stykke, hvor de så skal behandle de stykker i den rækkefølge, de får dem givet. I dette projekt fokuserer vi på *pladsreservationsproblemet* i tog. Problemet består i, at en (ukendt) mængde passagerer forespørger rejser i et tog, defineret ved start- og slut-station. Toget har n sæder til rådighed og kører en lige, ensrettet strækning gennem k stationer fra 1 til k. Mellem to stationer kan således være placeret maksimalt n personer. Problemet kræver i sin natur, at input - forespurgte rejser - håndteres *fair*. Dette betyder, at enhver passager, der på tidspunktet for forespørgslen har mulighed for at blive placeret på en plads, skal placeres på en plads. Det er altså med dette krav ikke tilladt at afvise passagerer, der reelt kan sidde i toget på det givne tidspunkt.

Problemet kan betragtes på baggrund af forskellige objektfunktioner. I dette projekt vil vi prøve at optimere *passagerkilometer*. For enkelthedens skyld antager vi, at afstanden mellem to på hinanden følgende stationer altid er netop 1 enhed (1 passagerkilometer).

For at have et grundlag for sammenligning vil vi se på en alternativ version af problemet, hvor håndtering af input ikke kræves at være fair. Dette kan opnås ved at kræve, at alle forespørgsler skal kendes, før tildelingen af sæder udføres. Dette kunne for eksempel gøre sig gældende, hvis der var et afgrænset tidsrum, hvor billetter kunne bestilles. Denne sædetildeling skal så ske *offline*, hvilket betyder, at alle forespørgsler kan bearbejdes i en helhed.

Herefter vil vi se, om vi kan løse den oprindelige version af problemet, der kræver, at input håndteres på fair vis. Dette vil vi gøre med en *online* metode, hvor rejser bliver forespurgt én for én og skal håndteres i denne rækkefølge.

Det kan give anledning til problemer at håndtere forespørgslerne online. Eksempelvis kan man forestille sig en passager, som skal rejse kort og spærrer en plads for en senere passager, der vil rejse længere over en del af den samme strækning. I sidste ende vil flere af denne slags situationer nemlig resultere i færre passagerkilometer, end hvad der kunne have været opnået. Vi vil se på, hvad en *worst case* for en online-algoritme er.

Vi vil forsøge at generalisere de forskellige algoritmers udbytte (sommetider refereret til som *performance*) i forhold til passagerkilometer ved at foretage tests med forskellige værdier for de parametre, der påvirker resultatet. Desuden vil vi forsøge at få et billede af, hvad der er den bedste løsning i praksis, ved at analysere dem i forhold til deres køretider.

2 Gennemgang af algoritmer

I denne sektion forklares de forskellige algoritmer, vi har implementeret til løsning af de forskellige versioner af problemet. Hertil bruges pseudokode som hjælpemiddel til at forstå algoritmernes virkemåde.

Vi har forsøgt selv at udvikle algoritmerne uden benyttelse af litteratur, fordi vi den vej ikke blev påvirket til at tænke i særlige baner ved at følge den publicerede viden om algoritmer inden for emnet. I stedet har vi startet med selv at forsøge at sammenkoble og udnytte vores egen viden og intuition omkring algoritmer og problemets natur. Dette er også sket med henblik på bedre at kunne forstå vanskelighederne ved de udviklede algoritmer.

2.1 Brute force

Brute force tjekker alle mulige kombinationer af passagerer for hvert sæde og gemmer det bedste løbende. Det bedste lokale resultat bruger den som løsning for hvert enkelt sæde.

En brute force-metode virker kun, hvis alle passagererne allerede er kendt. Derfor er dette en offline algoritme, da den må vælge iblandt alle passagerer. Herefter kan den tage dem, der vil passe bedst, og derved afvise nogle af dem, der kom først, hvis de ikke passer ind i den bedst mulige sammensætning. Det betyder også, at brute force-algoritmen ikke er en fair algoritme. Pseudokoden følger herunder (det antages, at *requestList* er tilgængelig for alle delmetoder af CALL):

```
1:  Input: number of stations k, number of seats n
2:  passsengersOnSeat ← new two-dimentional list
3:  localBest ← new one-dimentional list
4:  procedure CALL(requestList)
5:      for each seat s in seats do
6:          tempList ← new one-dimensional list
7:          counter ← 0
8:          localCount ← 0
9:          CHAIN(0, endstation, tempList, counter, localCount)
10:         for each passenger p in localBest do
11:             Add passenger p to seat s in passengersOnSeat
12:             DELETE(p)
13:         end for
14:     end for
15: end procedure
```

CALL ser på hvert sæde én efter én. Her laver den en ny tom *list1* og nulstiller *counter* og *list2count*. Nu kan den køre CHAIN for at finde de passagerer, der skal sidde på sædet. Derefter skal CALL slette disse passagerer fra *requestList* ved hjælp af metoden DELETE.

```
1: procedure CHAIN(end, max, tempList, counter, localCount)
2:     if end ≥ max then
3:         while end ! = max do
4:             for each passenger p in requestList do
5:                 pStart ← start station of p
6:                 if pStart = end then
7:                     Add p to tempList
8:                     pDistance ← distance of travel of p
9:                     counter ← counter + pDistance
10:                    CHAIN(requstlist[i] end, max, tempList, counter, localCount)
11:                    Remove last element from tempList
12:                    counter ← counter - requestlist[i] distance
13:                end if
14:            end for
15:            end ← end+1
16:        end while
17:    else
18:        if counter > localCount then
19:            localCount ← counter
20:            localBest ← tempList
21:        else if counter = localCount and length of tempList < length of localBest then
22:            localCount ← counter
23:            localBest ← tempList
24:        end if
25:    end if
26: end procedure
```

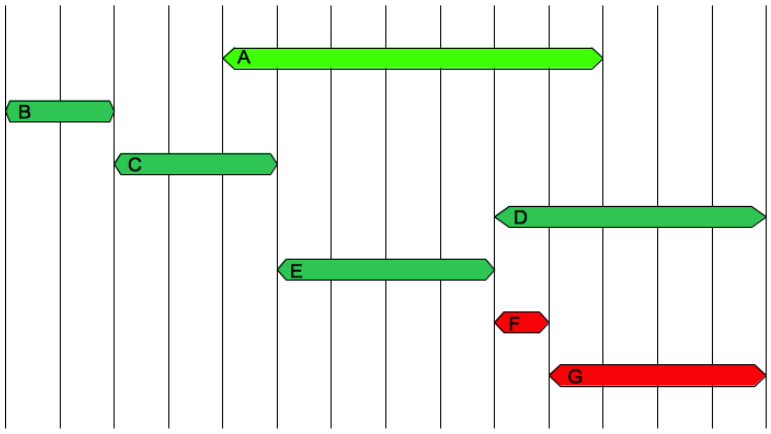
Brute force-metoden er designet sådan, at den kører alle mulige kombinationer igennem for hvert sæde. Det vil sige, at denne brute force-algoritme ikke er *global*, men *lokalt optimal*, da den kun kigger på ét sæde ad gangen.

Algoritmen kigger først, om der er nogen passagerer, der starter i den første station. Hvis der gør, tager algoritmen den første passager, gemmer denne, og kigger på dens slut-station. Hvis den finder en passager der, går den videre med den. Hvis der ikke er nogen passager, som starter på den givne station, går den videre til næste station. Når den er nået til enden, går den en passager tilbage og kigger efter en ny. Dette bliver algoritmen ved med, indtil den har testet alle mulige kombinationer, og dermed har den fundet den mest optimale til det givne sæde i forhold til passagerkilometer. Til sidst sletter den de fundne passagerer fra listen og går videre med næste sæde.

```
1: function DELETE(target)
2:     upper ← (number of passengers in requestList)−1
3:     lower ← 0
4:     current ← (upper + lower)/2
5:     while upper > lower and current ≠ target do
6:         if current > target then
```

```
7:              upper ← current−1
8:          else
9:              lower ← current+1
10:         end if
11:         current ← (upper + lower)/2
12:     end while
13:     if current = target then
14:         Remove target from requestlist
15:     end if
16: end function
```

Funktionen DELETE laver en binær søgning igennem *requestList* for at prøve at finde den passager, den leder efter. Den gør dette ved at kigge på indekset for passagereren i passagerlisten. Når passageren bliver fundet, vil den blive slettet fra *requestList*.



Figur 1: Her ses et input, hvor der er to sæder ($n = 2$) til rådighed. De grønne bjælker svarer til de forespørgsler, algoritmen har reserveret plads til, mens de røde er de forespørgsler, der ikke er plads til.

I Figur 1 ses en illustration af algoritmens virkemåde. Her vil brute force finde B først, da det er den eneste, der starter i den første station. Derefter vil den finde C, E og så D. Efter den er kommet til endestationen, vil den gå hen til starten af D og se, om der er andre muligheder. Det er her, hvor den finder F og G. Men disse er ikke bedre end D, så den gemmer stadig D. Hvis vi går tilbage, til efter den har fundet B og kigger igen, så finder den ingen andre. Det betyder, at algoritmen kigger på næste station, hvor der ingen er. Her kigger den på næste igen, hvor den finder A og bygger videre på denne, hvis det er muligt. Sådan kommer algoritmen igennem alle mulige kombinationer på hvert sæde.

2.2 Dynamisk

> Find for hver station den optimale passager-sammensætning ved at kigge på de tidligere stations fundne optimale sammensætninger.

Da brute force-metoden er virkelig tung og ikke kan tage særlig mange passagerer og sæder, kan man ikke bruge denne på større input. Derfor blev der lavet en dynamisk algoritme. Ligesom brute-force er denne algoritme designet som en offline algoritme, da denne også skal kende alle forespørgsler, inden algoritmen kan gå igang. Algoritmen er heller ikke fair, da denne også kan sige nej til nogle af de første, hvis de ikke passer ind i den bedste løsning. Pseudokoden følger herunder:

```
1:  Input: number of stations k, number of seats n
2:  passsengersOnSeat ← new two-dimentional list
3:  procedure CALL(requstlist)
4:      sorted ← new two-dimensional list of length k
5:      for each passenger p in requestList do
6:          Add p to sorted
7:      end for
8:      for each seat s in seats do
9:          best ← new two-dimensional list of length k
10:         counter ← 0
11:         best ← CHAIN(startstation, endstation, best)
12:         Add all passengers from best[endstation] to seat s in passsen-
    gersOnSeat
13:         Delete all passengers in best[endstation] from sorted
14:     end for
15: end procedure
```

I CALL sorterer algoritmen alle passagerer i et dobbelt-array efter deres slutstationer. Herefter kører algoritmen igennem for det antal sæder, der er, hvor den finder den mest optimale passager-fordeling på det givne sæde ved hjælp af CHAIN. Efter den optimale fordeling er fundet, skal disse også slettes fra den sorterede liste.

```
1:  procedure CHAIN(min, max, best)
2:      localBest ← new one-dimensional list
3:      for each station station from 1 to max do
4:          for each passenger p in sorted on station do
5:              pStart ← start station of p
6:              pDistance ← distance of travel of p
7:              if pStart = 0 then
8:                  localBest ← p
9:              else
```

```
10:                 localDistance ← total distance of travel of localBest
11:                 checkDistance ← pDistance + total distance of travels
    in best[pStart]
12:                 prevDistance ← total distance of travels in best[station-
    1]
13:                 localbest ← CompBest(p, best, localBest)
14:                 localbest ← CompPrev(p, best, localBest)
15:             end if
16:         end for
17:         if no passengers start in station then
18:             best[i] ← best[station-1]
19:         else
20:             best[i] ← localBest
21:         end if
22:     end for
23:     return best
24: end procedure
```

Chain virker ved at kigge på anden station og se, om der er nogen, der slutter der. Hvis der er nogen, der slutter i den station, kigger den på, om den person kører fra starten af. Hvis denne gør det, erstatter den, hvad algoritmen har i dens *localBest*-liste, som holder den bedste samling af passagerer, som den har fundet, inden den er færdig med at kigge på den givne station. Når algoritmen så er færdig med at kigge på en station, gemmer algoritmen den bedst fundne sammensætning af passagerer i listen *best* og går videre til næste station. Hvis der er nogen, der starter i denne station, som ikke kører fra start, bruger algoritmen CompBest, hvorefter den bruger CompPrev. Hvis der slet ikke er nogen, der starter i stationen, som algoritmen kigger på, så bliver den bedste liste fra forrige station lagt over til denne station.

```
1: procedure CompBest(passenger, best, localBest)
2:     pStart ← start station of passenger
3:     if checkDistance < localDistance then
4:         localBest ← passenger and best[pStart]
5:     else if checkDistance = localDistance then
6:         if length of best[pStart]+1 < length of localBest then
7:             localBest ← passenger and best[pStart]
8:         end if
9:     end if
10:    return localbest
11: end procedure
```

CompBest sammenligner den nye passagers rejse-distance plus dennes start-stations distance med den lokalt bedste fundne distance, som ligger i *localBest*. Hvis den nye passager har en længere distance end den tidligere fundne, erstatter CompBest passagererne som ligger i *localBest* med den nye passager og dennes start-stations bedste passagerer. Hvis den nye passagers samlede distance og den foregåenes beste distance er lig med hinanden kigger *CompBest* på

hvilken af de to sider har færrest passager og gemmer den.

```
1: procedure CompPrev(passenger, best, localBest)
2:     if prevDistance > localDistance then
3:         localBest ← best list in previous station
4:     else if prevDistance = localDistance AND best list from previous
   stations size < localBest size then
5:         localBest ← best list from prevous station
6:     end if
7:     return localbest
8: end procedure
```

CompPrev sammenligner den lokalt bedste kombination af passagerer i den givne station med den foregående stations bedste. Hvis den foregående stations bedste liste og den lokalt bedste har samme passager distance , så vil *CompPrev* kigge på hvilken én, der har færrest antal passager, hvorefter den gemmer den, som lokalt bedste.

Hvis vi kigger på samme eksempel som i brute force (Figur 1):

Algoritmen kigger på anden station først, men her stiger ingen passagerer af, så derfor går den videre til næste station. Her er der en passager, så den gemmer B i et array på tredje plads. Nu kører algoritmen igen igennem de stationer der ikke har nogen passagerer der stiger af, og skriver den forige stations bedst op, det vil sige at B kommer til at stå i arrayet fra plas tre til plads fem. Så kommer den hen til sjette station. Her stiger C af derfor kigger den på om C starter i den første station, det gør den ikke derfor kigger Algoritmen på arrayet med de foregående bedste fund på Cs startstation, her finder den B så derfor skriver algoritmen B og C til arrayet på sjette plads. Sådan bliver den ved, på tiende station skriver den B, C og E. På ellevte station skriver den B, C, E og F. Så finder den A og finder ud af at på As startplads står B som den bedste så ligger den de to sammen og sammenligner med den forgående stations bedste, her står der jo B, C, E og F som er bedre end A og B, derfor skriver den B, C, E og F igen. Til sidst finder algoritmen D og på Ds start station står der B, C og E. Nu kigger slagoritmen på G og ser om den vil kunne bygge noget der er bedre end før men den finder kun noget der er lige godt som den kunne med D men der er flere passager i denne derfor gemmer den B, C, E og D. Disse er den mest optimale på det givne sæde.

2.3 Forward-Backward

Forward-Backward vil danne to "kæder" af passagerer, der følger lige efter hinanden - én forfra og én bagfra - og sammenligne dem for at finde den bedste af disse kæder.

Forward-Backward er en offline algoritme designet med idéen om, at den skulle være et kompromis mellem køretid og optimering, idet at den ikke i alle tilfælde finder den lokale mest optimale sædeplacering, men at køretiden gør op for

denne lille forskel. Herunder følger pseudokoden for algoritmen (det antages, at *requestList* er tilgængeligt for alle delprocedurer):

```
1: input: number of stations k, number of seats n
2: function CALL(requestList)
3:     placement ← new two-dimensional list
4:     for number of seats n do
5:         specialList ← CHAIN(0, 0, k)
6:         for each passenger p in specialList do
7:             Add p to placement on seat n
8:             DELETE(p)
9:         end for
10:    end for
11: end function
```

Funktionen *Call* tager sig af det, der skal ske én gang for hvert sæde, som at starte funktionen *Call*, gemme passagererne når de er fundet og at slette disse passagerer bagefter.

```
1: function CHAIN(currentStation, localMinStation, localMaxSstation)
2:     (countForward, list1) ← FORWARDCHAIN(localMinStation, local-
   MaxStation)
3:     (countBackward, list2) ← BACKWARDCHAIN(localMinStation, lo-
   calMaxStation)
4:     if countForward > countBackward or (countForward = count-
   Backward and length of list1 < length of list2) then          ▷
   Comparison
5:         countBest ← countForward
6:         bestList ← list1
7:     else
8:         countBest ← countBackward
9:         bestList ← list2
10:    end if
11:    for each passenger p in requestList do   ▷ Check for special case
12:        if travel distance of p ≥ countBest and start station of p ≥
   localMinStation and end station ofp ≤ localMaxStation then
13:            specialPassenger ← p
14:            countBest ← travel distance of specialPassenger
15:            check ← true
16:        end if
17:    end for
18:    if check = true then
19:        SPECIALCASE(specialPassenger)
20:    else
21:        for Each passenger p in bestList do
22:            Add p to specialList
23:        end for
24:    end if
```

```
    Return specialList
end function
```

Funktionen Chain kører først funktionerne ForwardChain og BackwardChain for at finde en kæde forfra og bagfra, herefter vil den sammenligne de to kæder og vælge kæden med den længste distance (passagerkilometer). Hvis det er tilfældet, at kæderne har samme distance, vil den vælge den, der bruger færrest passagerer. Når den bedste kæde er valgt, vil det blive tjekket, om der er en enkelt-passager, der kører længere, end den længste som blev fundet. Hvis den finder en passagerer, der gør dette, vil funktionen *SpecialCase* blive kørt. Ellers vil den bedste kæde blive lagt over i en anden list og returneres.

```
function ForwardChain(localMinStaion, localMaxStation)
    currentStation ← localMinStation
    while currentStation < localMaxStation do
        for Each passenger p in requestList do
            if start station of p = currentStation and end station of p ≤ localMaxStaion and counter < distance of p then
                temp ← P
                counter ← distance of p
            end if
        end for
        if temp = Empty passenger then
            currentStation ← currentStation + 1
        else
            Add temp to list1
            currentStation ← end station of temp
            countForward ← countForward + distance of temp
            temp ← empty passenger
        end if
    end while
    Return (countForward and list1)
end function

function BackwardChain(localMinStaion, localMaxStation)
    currentStation ← localMaxStation
    while currentStation > localMinStation do
        for Each passenger p in requestList do
            if end station of P = currentStation and start station of p ≥ localMinStation and counter < travel distance of p then
                temp ← p
                counter ← distance of p
            end if
        end for
        if Passenger temp = empty passenger then
            currentStation ← currentStation - 1
        else
            Add temp to list2
```

```
35:             currentStation ← start station of temp
36:             countBackward ← countBackward + travel distance of
    temp
37:             temp ← empty passenger
38:         end if
39:     end while
40:     Return (countBackward and list2)
41: end function
```

Funktionen ForwardChain vil starte med at kigge på start-stationen og se, hvilken passager der kører længst fra denne station. Herefter vil den sætte denne passager ind i en liste og ændre *currentStation* til at være den station, hvor passageren står af. Hvis der ikke findes nogle passagerer, som starter på den station, der der bliver kigget på, vil der i stedet blive lavet et hul, og så vil der i stedet blive set på den næste station.

Funktionen BackwardChain virker på samme måde som *ForwardChain*, men hvor den i stedet ser, om der er nogen, der står af på slut-stationen og ændrer *currentStation* til denne passagers startstation.

```
1: function SpecialCase(specialPassenger)
2:     Add passengerspecialPassenger to specialList
3:     Delete(specialPassenger)
4:     tempMin ← start station of specialPassenger
5:     tempMax ← end station of specialPassenger
6:     Chain(localMinStation, localMinStation, tempMin)
7:     Chain(tempMax, tempMax, localMaxStation)
8: end function
```

Funktionen SpecialCase bliver kørt, hvis der er en enkelt-passager, som kører længere end nogle af de kæder, som Chain finder. Først bliver denne passager tilføjet til listen, derefter bliver passageren slettet fra *requestList*, så den ikke kan blive valgt igen. Herefter vil funktionen Chain blive kørt på hver side af *specialPassenger* for at se, om den kan få andre passagerer til at passe med *specialPassenger*. Til sidst returneres listen.

```
1: function Delete(target)
2:     upper ← (Number of passengers in requestList)−1
3:     lower ← 0
4:     current ← (upper + lower)/2
5:     while upper > lower and current ≠ target do
6:         if current > target then
7:             upper ← current−1
8:         else
9:             lower ← current+1
10:        end if
11:        current ← (upper + lower)/2
12:    end while
13:    if current = target then
```

```
14:         Remove target from requestList
15:     end if
16: end function
```

Funktionen DELETE laver en binær søgning igennem *requestList* for at prøve at finde den passager den leder efter. Den gør dette ved at kigge på indekset af passagererne i listen. Når passageren bliver fundet vil den blive slettet fra *requestList*

2.4 Online/Offline

> Tjek for hver forespørgsel, om passageren kan være der (om kapaciteten n overskrides mellem de ønskede stationer), og godkend i så fald denne. Ved forespørgslernes ophør, placér de godkendte passagerer i rækkefølge efter start-station på det første ledige sæde.

Online/Offline-algoritmen består af en online del og en offline del. Følgende viser detaljeret pseudokode for online/offline-algoritmen (det antages, at *seats* er tilgængelig for enhver procedure):

```
1: Input: number of stations k, number of seats n
2: density ← new list of length k − 1
3: seats ← new two-dimensional list
4: procedure CALL(requestList)
5:     passengers ← new list
6:     for each passenger p in requestList do
7:         start ← start station of p
8:         end ← end station of p
9:         i ← start
10:        isPossible ← true
11:        while i < end and isPossible = true do
12:            if density[i] ≥ n then
13:                isPossible ← false
14:            end if
15:            i ← i + 1
16:        end while
17:        if check = true then
18:            Add p to passengers
19:            for j=start to end−1 do
20:                density[j] = density[j]+1
21:            end for
22:        end if
23:    end for
```

```
24:     Sort passengers by increasing start station
25:     PlacePassengers(passengers)
26: end procedure
```

Med online/offline-algoritmen skal det simuleres, at forespørgsler kommer løbende. Dette er gjort ved at lade algoritmens primære funktion, Call, tage en liste af alle forespørgsler, som den så gennemgår i stigende rækkefølge. Desuden får algoritmen givet antallet af stationer, k, samt antal pladser, n.

Antallet af passagerer placeret på hver enkelt station vedligeholdes i listen *density*. For hver forespørgsel tjekkes det, om den pågældende rejse vil få n til at overskride mindst ét sted mellem de stationer, rejsen dækker over. Hvis dette er tilfældet, bliver passagereren afvist. Hvis dette ikke er tilfældet, bliver passageren derimod godkendt, og værdierne i *density* bliver opdateret (sådan at kapaciteten mellem de involverede stationer øges med én).

Til sidst i Call-funktionen, når alle forespørgsler er gennemgået, sorteres passagererne efter stigende start-tid. Herefter kaldes PlacePassengers, som skal fordele alle de godkendte passagerer på pladserne.

Herunder ses den detaljerede pseudokode for fordelingen af passagererne:

```
1: procedure PlacePassengers(passengers)
2:     for each seat s in seats do
3:         for each passenger p in passengers do
4:             start ← start station of p
5:             prevEnd ← end station of previous p
6:             if s is empty or start ≥ prevEnd then
7:                 Add p to s
8:                 Remove p from passengers
9:             end if
10:        end for
11:    end for
12: end procedure
```

Denne algoritme kaldes en *First Fit*-algoritme, som det også er defineret i [1], da den vil forsøge at placere hver passager på det første sæde, så det andet sæde etc., indtil et ledigt sæde findes.

Beviset for, at denne algoritme får placeret alle de godkendte passagerer på et sæde, er ført i *afsnit 3.2*.

2.5 Online

Placér hver passager på det sæde, der skaber mindst muligt mellemrum (antal stationer) i forhold til allerede placerede passagerer.

Følgende viser detaljeret pseudokode for online-algoritmen (det antages her, at k, n og *seats* er tilgængelig for enhver delprocedure):

```
1:  Input: number of stations k, number of seats n
2:  seats ← new two-dimensional list
3:  procedure BEST-FIT(requestList)
4:      for each passenger in requestList do
5:          possibleSeats ← GETPOSSSEATS(passenger)
6:          if possibleSeats is not empty then
7:              minLeft ← k
8:              minRight ← k
9:              minSeat ← possibleSeats[1]
10:             for each seat s in possibleSeats do
11:                 (left, right) ← NEARESTSPACE(passenger, s)    ▷ Find
    nearest spaces on seat
12:                 (minLeft, minRight, minSeat) ← BESTSPACE(left,
    right, minLeft, minRight, s, minSeat)          ▷ Update best seat and
    distances so far
13:             end for
14:             Add passenger to seat minSeat
15:         end if
16:     end for
17: end procedure
```

Online-algoritmen får givet antal pladser n samt antal stationer k som input. Desuden får selve BEST-FIT-proceduren givet listen af forespurgte rejser, der skal simulere en rækkefølge, der kommer løbende.

For hver forespørgsel findes først den mængde af pladser, passageren har mulighed for at sidde på. Er denne tom, kan passageren ikke få en plads. Ellers skal passageren pladseres bedst muligt blandt disse.

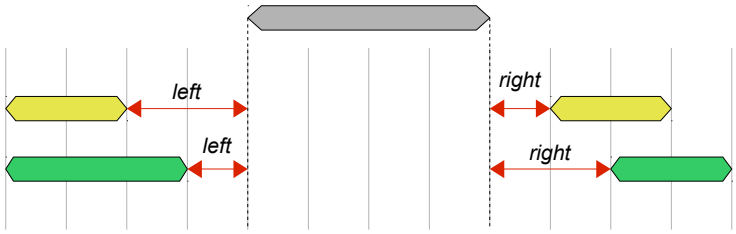
Hvilke sæder, passageren har mulighed for at sidde på (uden overlap), findes med funktionen GETPOSSSEATS:

```
1:  procedure GETPOSSSEATS(passenger)
2:      possibleSeats ← new list
3:      start ← start station of passenger
4:      end ← end station of passenger
5:      for i = 1 to n do
6:          isPoss ← true
7:          for j = 1 to length of list at seat i do
8:              pStart ← start time of passenger j at seat i
9:              pEnd ← end time of passenger j at seat i
10:             if pStart < end and pEnd > start then
11:                 isPoss ← false
12:             end if
13:         end for
```

```
14:         if isPoss = true then
15:             Add seat number to possibleSeats
16:         end if
17:     end for
18:     return possibleSeats
19: end procedure
```

Funktionen gennemgår hver passager på hvert sæde og tjekker, om der er overlap med den nye passager. Hvis alle passagerer på et sæde er uden overlap med den nye passager, bliver sædet tilføjet til listen af mulige sæder, *possibleSeats* (*linje 15* i GetPossSeats).

For at placere en passager bedst muligt i forhold til optimering af passagerkilometer antages det, at skabelse af mindst muligt mellemrum til andre passagerer er bedst. På denne måde kan man forestille sig, at længere fri strækninger forbliver tomme til lange rejser. Denne metode er en *Best Fit*-algoritme, som det fremgår af [1].



Figur 2: Illustration af værdierne for *left* og *right*. De farvede bjælker illustrerer passagerer, der er placeret mellem de stationer, der er farvet i det horisontale område. Den grå sekvens er her den passager, der ønsker at få plads i toget. De gule og de grønne sekvenser repræsenterer allerede placerede passagerer, der sidder er fordelt på to sæder. Værdierne for *left* og *right* er markeret med røde pile.

Hver passager, der er placeret på hver plads, skal gennemgås. For hver plads skal findes værdier for afstanden henholdsvis til højre og venstre og den nye passager (*left* og *right*) hen til den nærmeste anden passager placeret på samme plads. Disse findes ved for hver passager på sædet at opdatere disse værdier, hvis passageren har en mindre afstand til den nye passager til henholdsvis højre eller venstre. *left*- og *right*-værdierne er illustreret i Figur **??** (kald til metoden nearestSpace på *linje 15* i call-funktionen):

```
1: procedure NearestSpace(passenger, s)
2:     start ← start station of passenger
3:     end ← end station of passenger
4:     left ← start
5:     right ← k−end
6:     for each passenger p at seat s do
7:         pStart ← start station of p
```

```
8:          pEnd ← end station of p
9:          if start > pEnd and start−pEnd < left then
10:             left ← start−thisEnd
11:         end if
12:         if end < pStart and pStart−end < right then
13:             right ← thisStart−end
14:         end if
15:         if left = 0 and right = 0 then          ▷ A best seat is found
16:             return (left, right)
17:         end if
18:     end for
19:     return (left, right)
20: end procedure
```

Et specialtilfælde er, hvis *left* og *right* begge er 0. Så er situationen bedst tænkelig (da passageren passer perfekt ind), og derfor stoppes gennemløbet (*linje 11-13* i NearestSpace).

Under gennemgangen af sæderne sørger BestSpace for, at der bliver der vedligeholdt to variabler; *minLeft* og *minRight*. De holder de bedste fundne værdier indtil videre i gennemløbet. Desuden bliver det sæde (nummer), som værdierne blev fundet på, gemt (*minSeat*).

Måden, den afgør, om et givent sæt af *left*- og *right*-værdier er bedre end sættet repræsenteret ved *minLeft* og *minRight*, er defineret i metoden BestSpace. Hvis den mindste af værdierne *left* og *right* er mindre end både *minLeft* og *minRight* - eller hvis det mindste mellemrum er lig enten *minLeft* eller *minRight*, og det største mellemrum er mindre end den anden værdi (*minLeft* eller *minRight*), - skal værdierne for *minLeft* og *minRight* opdateres (og desuden *minSeat*):

```
1: procedure BestSpace(left, right, minRight, minLeft, seat, minSeat)
2:     minSpace ← left
3:     maxSpace ← right
4:     if left > right then
5:         minSpace ← right
6:         maxSpace ← left
7:     end if
8:     if (minSpace < minLeft and minSpace < minRight) or (minSpace = minLeft and maxSpace < minRight) or (minSpace = minRight and maxSpace < minLeft) then
9:         minLeft ← left
10:        minRight ← right
11:        minSeat ← seat
12:    end if
13:    return (minLeft, minRight, minSeat)
14: end procedure
```

Til sidst indsættes passagereren på sædet med nummeret *minSeat*, som er det bedst fundne sæde. Hvis alle sæder er tomme, vil *minSeat* automatisk indeholde nummeret på det første sæde, sådan at den første passager vil blive placeret her.

3 Analyse af algoritmer

I dette afsnit gås i dybden med de forskellige algoritmer. Der bliver kigget på køretiden og et bevis på, at online/offline-algoritmen får placeret alle godkendte passagerer på et sæde. Desuden bliver online-algoritmens worst case-udbytte analyseret.

3.1 Køretid

Følgende notation bliver brugt under analysen af køretiderne:
N = antal sæder
K = antal stationer
P = antal passagerer
Antal passagerer på ét sæde kan skrives som K, da der aldrig kan sidde flere passagerer på et sæde, end der er antal stationer.

Brute Force

Brute force kigger på alle muligheder for hvert sæde. På hver station vil den kigge alle passagerer igennem, og på stationen efter vil den prøve at sætte alle andre passagerer i forlængelse af den forrige passager. Tiden, det tager den at sætte alle passagerer i forlængelse af hinanden, kan udtrykkes som P^K. Da den kører denne proces igennem for hvert sæde, vil det blive til $N \cdot P^K$. Udover det er der er også en slette-proces, der finder sted, hver gang der bliver fundet et sæt passagerer til det pågældende sæde. Sletningen er en binær søgning, som finder passageren i listen, men da alle elementer til højre for dette element i listen skal rykkes én plads til venstre, tager dette $P + \log P$ tid, hvilket den skal gøre for hver passager på sædet. Dog kan antal passagerer ikke overstige antal stationer, som betyder vi kan skrive dette som K, mens sletningen kommer til at være $K \cdot (P + \log P)$. Den samlede køretid kommer derfor til at være $N \cdot (P^K + K \cdot (P + \log P))$. Da vi kun er interesseret i det dominerende led, kommer køretiden til at være:

$$O(N \cdot P^K)$$

Dynamisk

Dynamisk vil for hvert sæde opbygge et array af længde K, ved at den for hver delstrækning vil se på alle passagererne, og se, hvad den bedste mulighed er for den nuværende delstrækning. For at finde den bedste mulighed, kigger den et array igennem af længde K, hvori der ialt er P elementer, hvilket giver det en tid på $P + K$. Da den gør dette for alle sæder, bliver det $N \cdot (P + K)$. Når det oprindelige array er fyldt ud, vil dynamisk have fundet passagerne for ét sæde. Derefter slettes passagerne, men da dynamisk gemmer hvor i listen dens passagerer kommer fra, samt at der samlet kun kan rykkes P elementer, da den arbejder i delmængder af P, tager dette kun $K + P$ tid, hvilket giver en samlet tid på $N \cdot (2(K + P))$. Herefter fjernes konstanterne, da man ikke er interesseret i dette ved O-notation:

$$O(N \cdot (K + P))$$

Forward-Backward
Forward-Backward prøver først at gå stationerne igennem fra starten, og se hvilke passagerer, den kan placere i forlængelse af hinanden. Dette tager $P \cdot K$ tid. På samme måde kigger den bagfra, som også tager $P \cdot K$. Herefter vil den se, om der er nogen enkelt-rejsende, der er længere end denne kæde, hvilket tager P tid. Dette kan opstå op til K gange, hvor den så vil køre den tidligere del, $P \cdot K + P$, igennem igen, som betyder en tid på $K \cdot (K \cdot P + P)$. Der bliver også kørt den samme sletningsproces som i brute force, hvilket tager $K \cdot (P + \log P)$. Disse ting gør den for hvert sæde, hvilket bertyder en samlet tid på $N \cdot (K \cdot (K \cdot P + P) + K \cdot (P + \log P))$. Når man får ganget ud og ser på det mest dominerende led, vil det se således ud:

$$O(N \cdot K^2 \cdot P)$$

Online/Offline
For at se, om der er plads til passageren i online/offline-algoritmen, vil den kigge et array igennem af længde K for hver passager, hvilket tager $P \cdot K$ tid. Efterfølgende vil den fylde sæderne op ved at køre listen af passagererne igennem for hvert sæde, hvilket har tiden $N \cdot P$. Når en passager har fundet en plads, vil den blive slettet fra listen, men da den allerede ved, hvor i listen den er, tager dette kun P tid, hvilket giver en samlet tid på $2P \cdot (K + N)$. Den samlede tid kommer til at se således ud, da vi ikke er interesseret i konstanterne:

$$O(P \cdot (K + N))$$

Online
Online vil for hver passager kigge alle sæderne igennem for at se, hvor passageren kan sidde. Det gør den ved at se på alle passagererne på sædet, som tager $P \cdot K \cdot N$ tid. Herefter vil den kigge på alle sæder, som passageren kan sidde og se, hvor det er bedst at placere passageren. Det gør den ved at kigge alle passagerer igennem på hvert sæde, $N \cdot K$, som giver en samlet tid på $P \cdot 2(K \cdot N)$. Igen er vi ikke interesseret i konstanterne, så det kommer til at se således ud:

$$O(P \cdot K \cdot N)$$

3.2 Bevis for korrekthed af online/offline-algoritme

Påstand: First Fit-algoritmen i online-metoden får placeret alle godkendte passagerer på et sæde. Mere præcist: når den største *density* (som defineret i forklaringen algoritmen) er $\leq n$, vil First Fit kunne placere passagererne på n sæder.

Alle passagerer er sorteres ud fra start-stationer. Da selve pladsfordelingen foregår offline, kan enhver togrute modelleres som et graffarvningsproblem ved brug af intervalgrafer, hvor en knude er en passagers accepterede rejseforespørgsel, og enhver kant er et overlap mellem to af disse, således at en kant symboliserer, at to passagerer begge rejser over mindst én delstrækning på samme tid.

Hver knude kan nu farves, således, at hver farve markerer et sæde. For alle kanter gælder da, at de ikke må forbinde to knuder med samme farve, hvilket ellers ville svare til, at to passagerer skulle side på sammen sæde på samme tidspunkt, hvilket skal undgås.

Antag, at der er placeret p rejse-forespørgsler allerede, og lad q være rejse-forespørgsel nummer $p+1$. En farve m vil da tages i brug, hvis og kun hvis q overlapper med tidligere placerede rejse-forespørgsler i strækningen fra q's startstation til q's startstation+1, hvor densiteten tidligere var på $m-1$. Dette gør sig gældende, da enhver rejse-forespørgsel forinden ville have en startstation $\leq q$'s startstation (givet ved sorteringen af passagererne), og derved kan der kun være et overlap, hvis der i hvert fald er overlap i denne delstrækning.

Derved kan det altså konstateres, at når der eksisterer m overlap (svarende til m forskellige farver farver i en intervalgraf), er netop m sæder er tilstrækkeligt for First Fit. □

4 Resultater

For at finde ud af, hvordan algoritmer klarer sig på forskellige slags input, er foretaget flere tests. Resultaterne af disse tests bliver fremlagt i det følgende.

4.1 Tilfældigt test-data

For at sammenligne udbyttet ("performance") af de forskellige algoritmer med hensyn til antal passagerkilometer, er lavet et test-program. Dette program kører algoritmerne på tilfældigt genereret input gentagne gange (det samme input til hver algoritme), og til sidst beregner den for hver algoritme et gennemsnitligt udbytte. Alle resultater undervejs lagres desuden for at kunne analysere på dette i en sammenhæng bagefter. Det "værste" fundne resultat - i form af valgte passagerer - for hver algoritme vedligeholdes undervejs, sådan at disse kan undersøges efterfølgende.

Tre forskellige tests med følgende parametre er blevet udført:

nr.	**antal sæder**	**antal stationer**	**antal forespørgsler**	**teoretisk maks.**[a]
1	5	20	50	100
2	5	20	20	100
3	5	20	15	100

[a] *Teoretisk maksimale* dækker over *antal sæder·antal stationer*, der er udtryk for den maksimalt mulige mængde passagerkilometer.

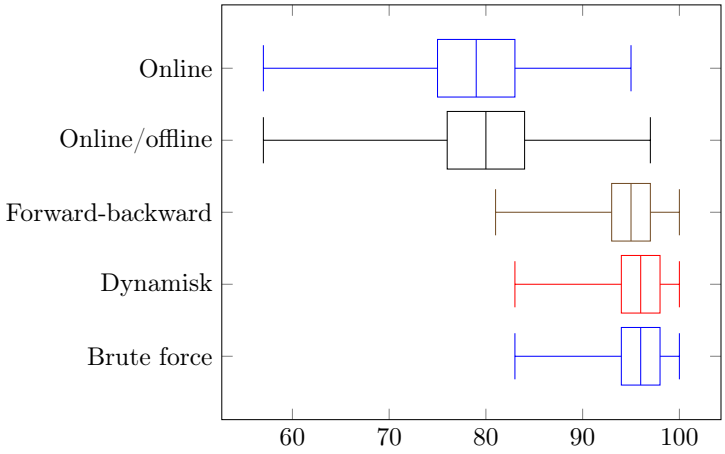
Værdierne for antal sæder og antal stationer er faste, mens antal forespørgsler er valgt med henblik på at teste programmerne med forskelligt forhold mellem antal sæder og antal forespørgsler. Som det ses, er forholdene henholdsvis 1:10, 1:4 og 1:3.

De følgende resultater er opnået ved **999** gennemløb. Resultaterne i hver kolonne er de gennemsnitligt opnåede passagerkilometer:

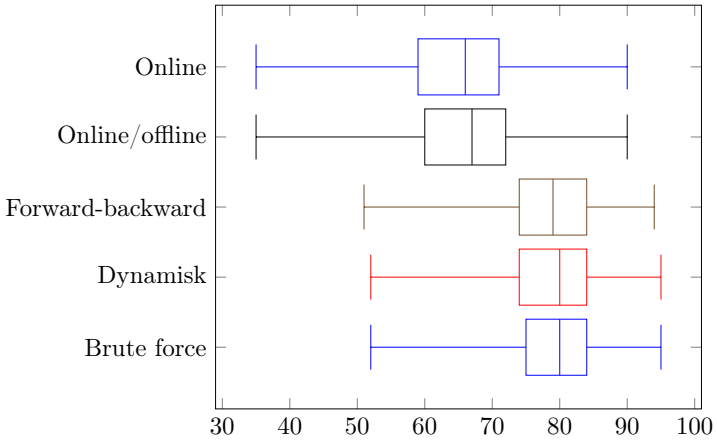
nr.	**Forespurgt**	**Brute force**	**Dynamisk**	**Forward-backward**	**Online/offline**	**Online**
1	287,7	95,6	95,6	94,9	79,9	78,9
2	113,7	79,1	79,1	78,9	66,0	64,9
3	85,8	70,2	70,2	70,1	60,9	60,0

Offline-algoritmerne (*Forward-backward*, *Dynamisk* og *Brute force*) får altså gennemsnitligt nogenlunde samme resultat. Dette gælder også for online- og online/offline-algoritmen.

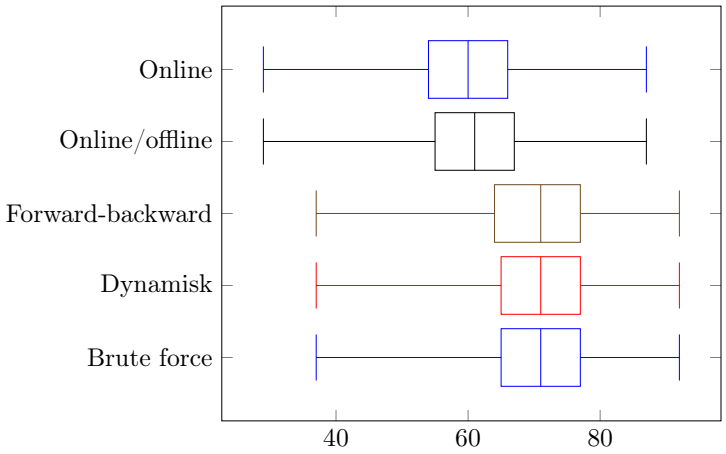
For at få et overblik over den samlede mængde af resultater, som hver af algoritmerne opnår under gennemløbene, ses et boksplot af test nr. 1 i Figur 3, test nr. 2 i Figur 4 og test nr. 3 i Figur 5.



Figur 3: Boksplot, der viser resultatet af test nr. 1.



Figur 4: Boksplot, der viser resultatet af test nr. 2.



Figur 5: Boksplot, der viser resultatet af test nr. 3.

Af boksplotsene ses det, at offline-algoritmerne får en større spredning af resultater (større afstand mellem mindste værdi og største værdi), jo færre forespørgsler der er i forhold til antal sæder. Som et resultat heraf kommer offline-algoritmernes mindste værdier tættere på online- og online/offline-algoritmernes mindste værdier.

Der er udført yderligere to tests for at kunne sammenligne disse med test nr. 1 og test nr. 2 ved at lade forholdet mellem antallet af sæder og antallet af forespørgsler være det samme:

nr.	**antal sæder**	**antal stationer**	**antal forespørgsler**	**teoretisk maks.**
4	50	20	500	1000
5	50	20	200	1000

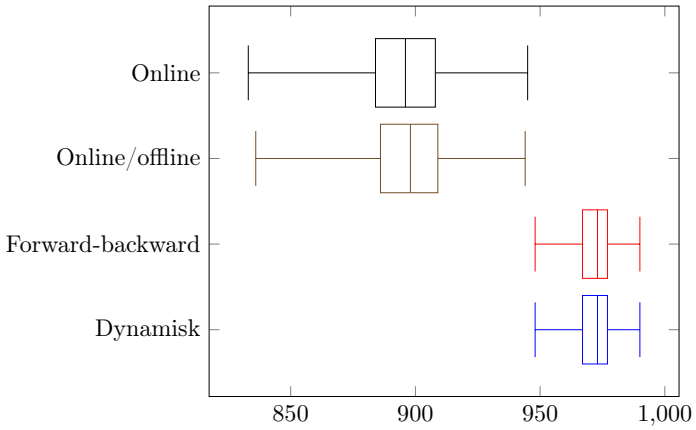
De overordnede resultater af ovenstående tests ses her:

nr.	**Forespurgt**	**Brute force**	**Dynamisk**	**Forward-backward**	**Online/offline**	**Online**
4	2870,7	-	971,8	971,8	896,9	895,3
5	1154,4	-	870,7	868,5	781,7	777,1

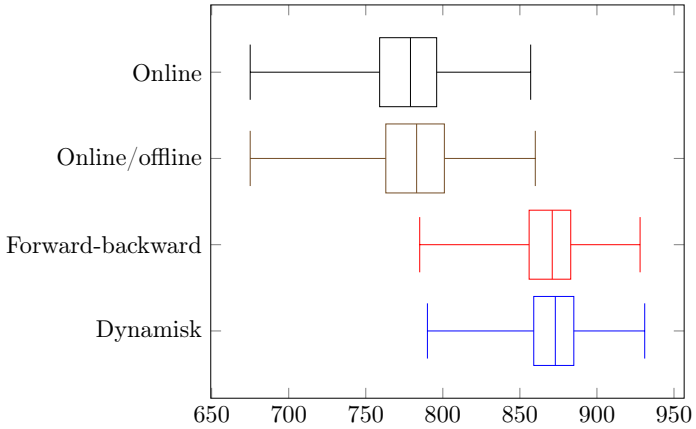
Som det er antydet i tabellen, er brute force udelukket fra disse tests, da dens køretid stiger drastisk for større værdier af antal sæder og antal passagerer (se Afsnit 3 for argumentationen bag dette).

Igen er gennemsnittene næsten ens for offline-algoritmerne, og næsten ens for online-algoritmerne.

Boksplots for resultaterne ses i henholdsvis Figur 6 og Figur 7.



Figur 6: Boksplot, der viser resultatet af test nr. 4.



Figur 7: Boksplot, der viser resultatet af test nr. 5.

Den nedre grænse for alle algoritmerne ser ud til forholdsmæssigt at være næsten lig deres median fra den forrige test med samme forhold (test nr. 4 sammenlignes med test nr. 1 og test nr. 5 sammenlignes med test nr. 2).

Desuden ser det ud til, at offline-algoritmernes største værdier er blevet forholdsmæssigt lidt mindre.

4.1.1 Sandsynlighedsfordeling af forespørgsler

Ud fra den måde, hvorpå vi genererer tilfældige passagerer (start- og slut-stationer), kan sandsynligheden for de forskellige rejselængder og start/slut stationer udregnes ved hjælp af formlerne nedenfor (fra [2]).

Tilfældige rejser bliver genereret på følgende måde: En tilfældig start-station s fra 1 til $k-1$ vælges, og herefter vælges en tilfældig slut-station fra $s+1$ til k.

For at regne slut ud fra startstationen bruges denne formel:

$$\frac{1}{j} \cdot \frac{1}{i} \mid i \in \{j, j-1, \ldots, 1\}$$

$j = k - 1$, altså antallet af stationer fratrukket 1.

Hvis man overordnet skal lave en matematisk formel for at udregne sandsynligheden for at få en hvis længde af passagerers rejse vil den se sådan ud:

$$s = \sum_{i=0}^{j-x} \frac{1}{j^2 - i \cdot j}$$

hvor x er den ønskede rejselængde og j er samme som forrige formel.

MatLab-scriptet[1] giver to resultater: først sandsynligheden for de forskellige rejselængder, og derefter gives sandsynligheden for slut-station ud fra start-station. Bemærk, at den første forekommende startstation er defineret som station 1 i det følgende.

Resultatet der giver sandsynligheden for de forskellige rejselængder, findes ved for eksempel en rute bestående af to stationer. I dette tilfælde er der kun en mulighed, nemlig en rejse af længde 1. I dette tilfælde vil der først blive printet 1 for rejser af længde 1, og 0 for rejser af længde 2, hvilket der aldrig vil forekomme.

Ved en rute bestående af tre stationer, vil der være to mulige start-stationer. Fra station 1 vil der være $\frac{1}{2}$ chance for både station 2 og for station 3 som slut-station. Vælges station 2 som start-station, da vil station 3 vælges som slut-station, givende en fordeling af længder på,

$\frac{3}{4}$ sandsynlighed for en rejse af længde 1, og

$\frac{1}{4}$ sandsynlighed for en rejse af længde 2.

Sandsynligheden for slutstationer gives ud fra startstationer. For at eksemplificere dette, kan man kigge på en tog-rute med fem stationer, altså hvor der er fire delstrækninger af længde 1. Da kan startstationen være enten $1, 2, 3$ eller 4.

Denne tog rute giver os fire mulige start-stationer, altså må der nødvendigvis være $\frac{1}{4}$ chance for, at $\{1, 2, 3, 4\}$ bliver valgt som startstation. Dette giver herefter fire mulige cases,

Vælges station 2 som startstation, er der igen fire valgmuligheder for slut stationen, $\{2, 3, 4, 5\}$, hvilket giver en sandsynlighed for at disse skulle vælges på $\frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$.

Vælges station 1 som startstation, er der tre valgmuligheder for slut stationen, $\{3, 4, 5\}$, hvilket giver en sandsynlighed for at disse skulle vælges på $\frac{1}{4} \cdot \frac{1}{3} = \frac{1}{12}$.

Vælges station 3 som startstation, da er der to valgmuligheder for slut stationen, $\{4, 5\}$, hvilket giver en sandsynlighed for at disse skulle vælges på $\frac{1}{4} \cdot \frac{1}{2} = \frac{1}{8}$.

Vælges station 4 som startstation, er der kun en mulighed for slut station, $\{5\}$, hvilket giver en sandsynlighed for at denne skulle vælges på $\frac{1}{4} \cdot \frac{1}{1} = \frac{1}{4}$.

Hermed kan vi konkludere at der er større sandsynlighed for forespørgsler på kortere strækning end længere strækning.

4.2 Online-algoritmens udbytte

For at finde ud af, hvor "dårligt" online-algoritmen klarer sig i værste tilfælde, er der behov for at finde et input, der giver det værste resultat sammenlignet

[1] Der er blevet udviklet et MatLab-script i samarbejde med gruppen 62c, der har sammen emne og vejleder. Selve scriptet er at forefinde i Appendiks.

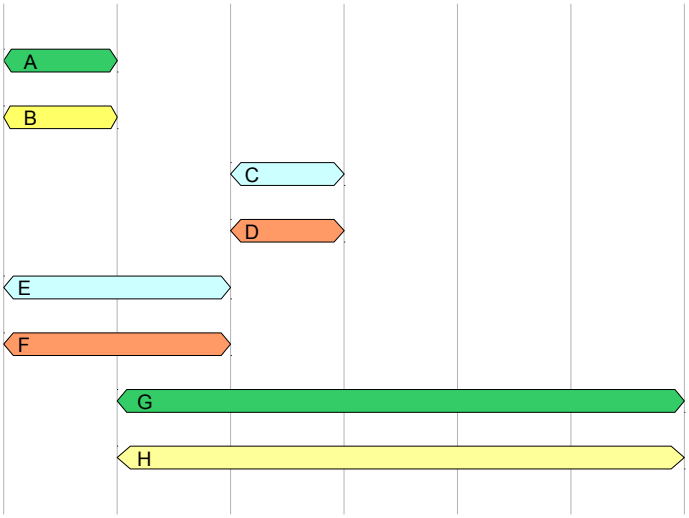
med den optimale løsning. Den optimale løsning er her givet ved online/offline-algoritmens resultat/output med samme inputsekvens.

Et worst case-eksempel er følgende, der følger af beviset for Theorem 10 i [1]:

Antal stationer, k, er i det følgende eksempel 6. Antal pladser, n, er 4:

	antal	**Forespørgsel**	**Online (placering)**	**Online/offline (placering)**
a	$n/2$	[1,2]	sæde 1	sæde 1
b		[1,2]	sæde 2	sæde 2
c	$n/2$	[3,4]	sæde 1	sæde 3
d		[3,4]	sæde 2	sæde 4
e	$n/2$	[1,3]	sæde 3	sæde 3
f		[1,3]	sæde 4	sæde 4
g	$n/2$	[2,k] = [2,6]	*afvist*	sæde 1
h		[2,k] = [2,6]	*afvist*	sæde 2

Situationen er illustreret i *figur 8*.



Figur 8: Illustration af worst case - Stationerne er repræsenteret horisontalt fra venstre mod højre, og tid vertikalt. Forespørgslernes farver svarer til de forskellige sæder, som online/offline-algoritmen placerer passagererne på.

Som det ses af ovenstående, får online/offline-algoritmen alle passagerer med, hvilket giver 16 passagerkilometer. Online-algoritmen får derimod kun 8 passagerkilometer, hvilket er netop $\frac{4}{(k+2)} \Rightarrow \frac{4}{8} = \frac{1}{2}$ gange det optimale resultat. Dette mål for online-algoritmen er dens *competitive ratio* for *accomodating sekvenser*. Dette er et mål for forholdet mellem en online-algoritmes worst case og det tilsvarende optimale resultat (hvor alle forespørgsler *kan* imødekommes), som det er defineret i [1].

Asymptotisk kan dette mål beskrives som $\Theta(\frac{1}{k})$, og online-algoritmens udbytte af det generelle worst case-input bliver altså i stigende grad værre, jo flere stationer problemet indebærer.

5 Diskussion

I det følgende afsnit diskuteres resultaterne med udgangspunkt i Afsnit 4.

I vores tests kan vi se at boksplottene får et bedre resultat, når der er flere forespørgsler per sæde, hvilket er, fordi der er større chance for, at der bliver genereret nogle passagerer, der passer bedre ind sammen med hinanden, dette gælder både offline- og online-algoritmerne.

Som nævnt ovenover, sker der en stor forandring, når man ændrer på forholdene mellem sæder og passagerer, men der er også en stor forskel, når man ændrer i størrelsen af input (men bevarer forholdene). Som det kan ses i de forskellige boksplots, rammer offline-algoritmerne ikke deres teoretiske maksimale, når der kommer flere passagerer. Dette kan forklares ved at se på, hvordan vores passagerer bliver genereret, da vi fandt ud af at chancen for, at en passager har startstation i station 1 er $\frac{1}{k-1}$. Ved første eksempel (5 sæder, 20 stationer og 50 passagerer) betyder det, at hvis den skal ramme det teoretiske max, skal der være mindst 5 personer, som har en startstation i 1, da den hver gang genererer en passager, som har chancen $\frac{1}{19}$ og at chancen for dette er ens for hver passager, skal den være "heldig" for at få startstation 1, dvs. at den skal være "heldig" 5 gange ud af 50. Når den får et større input (men med samme forhold) (50 sæder, 20 stationer og 500 passagerer), skal den være "heldig" 50 gange ud af 500, hvilket har mindre sandsynlighed end med det mindre input.

Vi regnede med, at den dynamiske algoritme altid ville få samme resultat som brute force-algoritmen, men det har vist sig ikke at være tilfældet. De to algoritmer arbejder på hver deres måde, og der sker sommetider en lille afvigelse i udfaldet som et resultat af dette. Brute force-algoritmen gennemgår alle muligheder hvor den vælger den første bedste rute, og gemmer denne, men det gør den dynamiske ikke. Den dynamiske algoritme gemmer det bedste resultat for hver station, og vælger den første som slutter i stationen, hvorefter den prøver at lave en samling af passagerer, og af denne grund kan der opstå forskelligheder. Den dynamiske gemmer måske 3 passagerer i en given station, og når den så kigger på den næste, kan disse 3 opnå det samme som 3 andre ville kunne opnå i brute force-algoritmen, eller omvendt. Herunder følger et eksempel på denne forskel.

Følgende rejser forespørges (notationen [*start-station*, *slut-station*] anvendes her):

> [19, 20], [12, 18], [0, 12], [17, 20], [0, 17], [18, 20], [0, 13], [15, 18], [18, 20], [0, 8], [9, 20], [13, 15]

Her vil brute force vælge følgende på det første sæde:

> [0, 12], [12, 18], [18, 20]

Hvor den dynamiske vil vælge disse intervaller på det første sæde:

> [0, 17], [17, 19], [19, 20]

De to algoritmer opnår at fylde sædet op lige godt i forhold til passagerkilometer, men hvis der er flere end ét sæde til rådighed, vil de andre sæder få et andet udfald også:

Bruteforce sæde 2:

[0, 8], [9, 20]

Dynaisk sæde 2:

[0, 13], [13, 15], [15, 18], [18, 20]

Som man kan se, så får den dynamiske i dette tilfælde et bedre udfald, fordi den ikke brugte passageren [18, 20] på første sæde. Den brugte ikke denne passager fordi at passageren [19, 20] kom før [18, 20]. Der er en chance for, at algoritmerne muligvis ville kunne få et bedre overordnet udfald, hvis den ikke brugte nogle af disse passagerer. Dette sker, fordi begge disse algoritmer er *lokalt* optimale og ikke *globalt* optimale. De ser kun på ét sæde ad gangen. Hvis de kiggede på alle sæder på én gang, ville de muligvis ikke får den mest optimale på første sæde alene men få et bedre globalt udfald. Men da de begge ikke er globale, og at de sommetider fylder forskellige passagerer på et sæde, vil det være muligt, at de to algoritmer får et forskelligt resultat og dermed også en lille forskel i passagerkilometer. Det samme kan ske med Forward-Backward-algoritmen. Den er i mange tilfælde næsten lige så god som Bruteforce og Dynamisk, men det sker en sjælden gang imellem, at Forward-Backward faktisk kan være bedre end de to andre algoritmer.

6 Konklusion

Efter at have testet algoritmerne, kan vi konkludere, at kravet om, at en algoritme skal være fair, er en afgørende faktor i forhold til antallet af passagerkilometer. Her har offline-algoritmerne været brugt som sammenligningsgrundlag, selvom det naturligvis ikke er samme version af problemet, de løser. Der kender den nemlig alle forespørgsler, hvorefter den placerer dem strategisk bedst. Dog forsøger den online algoritme også at placere passagererne strategisk godt, men har kun mulighed for at træffe et valg for den nuværende forespørgsel. Testresultaterne bærer præg af måden, vi har valgt at generere tilfældige passagerer på. Derfor kunne udfaldet af testene have forholdt sig anderledes med en anden måde at generere passagerer på.

Vi havde regnet med, at fairness-kravet havde en betydning, men den afgørende effekt på online/offline-algoritmen var overraskende. Dog er det vigtigt her at bemærke, at online-algoritmen i værste tilfælde kan få et udbytte, der er langt værre end online/offline-algoritmens udbytte.

På denne baggrund kan vi slutteligt konkludere, at online/offline-algoritmen umiddelbart ville være den bedste løsning på den oprindelige version af problemet til optimering af passagerkilometer i tog, hvor fairness-kravet er til stede.

7 Perspektivering

I en virkelig situation ville det naturligt være krævet, at pladsreservation i tog bliver håndteret på fair vis. Ellers kunne man eksempelvis risikere, at en passager, der faktisk havde reserveret sin rejse som den første, blev nedprioriteret,

hvis dennes rejse er kortere end alle de følgende rejser. Af den grund vil de præsenterede offline-algoritmer i dette projekt altså ikke umiddelbart kunne anvendes i praksis i lige netop en sådan situation. Til gengæld er online- og online/offline-algoritmerne begge mulige måder at håndtere rejse-forespørgsler på i tog. Dog skal man være opmærksom på, at online-algoritmen (jævnfør worst case-tilfældet præsenteret i Afsnit 4.2) kan få et meget dårligt resultat i visse tilfælde. Dette kunne eventuelt forbedres ved at tillade, at passageren kan få tildelt en anden plads inden togets afgang, hvis der på den måde kan blive plads til flere passagerer (og derved flere passagerkilometer).

Selvom offline-algoritmerne ikke er direkte anvendelige til denne problemstilling, kan man forestille sig andre lignende problemer, der ikke har dette fairness-krav. Eksempelvis kan problemet overføres til lokalebooking (hvor stationer er tid og sæder er lokaler), hvor ønsket om, at der ikke står lokaler tomme er vigtigst, eller planlægning af arbejdsopgaver, der skal udføres over et tidsrum, og hvor der ønskes flest muligt arbejdstimer afviklet.

Litteratur

[1] J. Boyar and Kim K. S. Larsen. The seat reservation problem. *Algorithmica*, 1999.

[2] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* Seventh Edition. McGraw-Hill, 2013.

8 Appendiks

8.1 Procesanalyse

Projektarbejdet er blevet udført sådan, at vi i gruppen selv har forsøgt at besvare problemformuleringen uden brug af litteratur til at starte med. På den måde havde vi en ambition om ikke at lade os påvirke af færdigudviklede og anerkendte idéer (i forhold til algoritmer, der løser problemet), uden vi først selv havde arbejdet med det ud fra vores intuition. Det gav os en frihed, fordi vi alle kun kom med den viden, vi havde fra vores respektive kurser fra vores studier, og alle idéer var derved velkomne. Arbejdet med at udvikle algoritmerne forløb ved dels at beskrive idéerne for hinanden på et konceptuelt plan, og herefter blev de konkretiseret ved at programmere dem direkte, så de kunne testes og rettes til.

Da vi havde nået frem til de algoritmer, vi havde brug for, ønskede vi at analysere dem. I den forbindelse søgte vi hjælp i en artikel, der omhandlede netop vores emne, og på den måde fik vi også sat navn på blandt andet delmetoder i vores algoritmer.

Arbejdet med projektet i gruppen har i de fleste henseender gået efter hensigten. Dog har der også været udfordringer undervejs af forskellig karakter. Blandt andet det forhold, at vi har været fire datalogi-studerende og én matematik-studerende, har været vanskeligt at indstille projektarbejdet efter. Dels har det været svært at uddele arbejdet på en rimelig måde; at vurdere, hvad der passer bedst til den enes matematik-kundskaber, og hvordan datalogi-arbejdet (der måske i virkeligheden har været den største del) skal strække sig til fire personer. Det har dog alligevel lykkedes os at adskille de matematik-relevante dele og de mere datalogi-relevante dele (herunder især programmering). For den ene matematik-studerende i gruppens vedkommende har det været svært at vide, hvad hun har skullet bidrage med, fordi projektet hurtigt tog en drejning, der i høj grad drejede sig om programmering. På den måde har de øvrige gruppemedlemmer været meget optagede af denne del, og der har på det tidspunkt ikke været noget konkret for hende at arbejde med. Dog fik vi, efterhånden som projektet skred frem, en mere klar idé om, hvilke matematik-relevante emner, der kunne arbejdes med, og derved fik hun noget at arbejde med. I den forbindelse har det også været en opgave i gruppen at formidle de forskellige aspekter, der ligger uden for den omvendte gruppes fagområde, til hinanden på en forståelig måde. Dette har vi forsøgt at imødekomme på flere måder. Vi har eksempelvis sørget for, at alle har været en del af en fælles gruppe-chat, som man har kunnet kontakte alle gruppens medlemmer i. Desuden har vi gjort stort brug af tegninger, eksempler og tavle-forklaringer, så idéer er blevet beskrevet så let-tilgængeligt som muligt for de øvrige i gruppen.

Socialt har arbejdet fungeret godt, og vi har alle følt os inkluderet i projektarbejdet. Alle gruppemedlemmer har for det meste været åbne, hvis der er noget, man har været utilfreds med eller ikke har forstået.

Projektarbejdet er primært blevet udført ved fælles fremmøde, sådan at man altid har haft andre at sparre med mundtligt. Dog har der også været enkelte opgaver, som er blevet udført adskilt fra gruppen for at sikre, at alle rapportens dele kunne nå at blive produceret. I den forbindelse har det også for det meste været muligt at få kontakt til de øvrige gruppemedlemmer (skriftligt), da alle har været opmærksomme på at stå til rådighed.

Projektet har i høj grad båret præg af samarbejde, men der har også været

mulighed for individuel udfoldelse, idet nogle af opgaverne er blevet uddelegeret. I størstedelen af projektperioden har arbejdet foregået ved en opdeling, hvor omkring to personer har arbejdet med det samme.

Effektivitetsniveaet har varieret meget igennem projektet. Man har tydeligt kunnet mærke, hvilke dele af projektet gruppen har fundet mest interessant, og dette har påvirket engagementet. Eksempelvis er den formidlingsmæssige del af projektet, rapport og poster, ikke gledet frem lige så uproblematisk som idégenereringen og programmeringen i starten af projektet.

Vi har i gruppen mødtes med vores vejleder én gang om ugen på møder á en-to timers varighed. Her har vi stort set hver gang haft en dagsorden klar, hvor vi havde formuleret spørgsmål og problemer forinden. Til mødet blev disse så fremlagt og diskuteret, og vores vejleder hjalp os i den rigtige retning. Foruden møderne har vi været i kontakt med vores vejleder via e-mail, hvor hun har givet os respons og gode idéer til vores videre arbejde.

Alt i alt har projektet bestået af overvejende gode erfaringer, men naturligvis har der også været situationer, man bør tackle anderledes. Læreprocessen, hvor vi først genererede idéer, og først bagefter formaliserede dem og brugte litteratur, har fungeret rigtig godt og givet os frie rammer til at tænke og arbejde i. Desuden har det virket meget positivt, at vi i gruppen har mødtes så meget, som vi har - på den måde har man til enhver tid kunnet spørge en anden om hjælp og få svar med det samme. Desuden kan man påvirke hinanden til at arbejde - men desværre har det også været erfaringen, at det omvendte kan være tilfældet. Hvis en eller flere gruppemedlemmer har været ukoncentrerede, har det nemt kunnet smitte af på resten, og dermed har vi i de tilfælde ikke fået produceret så meget, som vi burde. Her skal man til en anden gang være bedre til at stoppe op, når man kan mærke, at der ikke længere er fokus på relevant arbejde.

8.2 Sandsynlighedsfordeling af forespørglser

```
function [] = sand(x)
format rat
a = zeros(1,x);
b = zeros(1,x);
% zeroes bestemmer, om svaret kommer ud vandret eller
    lodret

for i=0:(x-1) ;
     a(i+2) = 1/(x^2-i*x)+ a(i+1) ;
     b(i+1) = (1/x) * (1/(i+1)) ;
i = i+1;
end


% Hvad der printes:
a = fliplr(a), b = fliplr(b)

% a er sandsynligheden ved rejselaengderne, og
% b er sandsynligheden ved slutstationen ud fra
    startstationen.
```

8.3 Java filer

Brute force

```
1  import java.util.ArrayList;
2
3  public class Bruteforce {
4  int counter = 0;
5  int passKm = 0;
6  int people = 0;
7  int min = 0;
8  int capacity = 0;
9  int stations = 0;
10 ArrayList<Passenger> passengerlist;
11 int localCount;
12 ArrayList<ArrayList<Passenger>> seats = new ArrayList<
       ArrayList<Passenger>>();
13 public ArrayList<Passenger> List1;
14 public ArrayList<Passenger> localBest;
15
16 public Bruteforce(int capacity, int stations) {
17     this.capacity = capacity;
18     this.stations = stations;
19     for(int i = 0; i < capacity; i++){
20         ArrayList<Passenger> newSeat = new ArrayList<
               Passenger>();
21         seats.add(newSeat);
22     }
23 }
24
25 public int call(ArrayList<Passenger> requests){
26     ArrayList<Passenger> requests2 = new ArrayList<
           Passenger>();
27     requests2.addAll(requests);
28     this.passengerlist = requests2;
29     for (int i = 0; i < capacity; i++) {
30         List1 = new ArrayList<Passenger>();
31         localCount = 0;
32         counter = 0;
33         Chain(0, stations, passengerlist);
34         people += localBest.size();
35         seats.get(i).addAll(localBest);
36         for(int l = 0; l < localBest.size(); l++){
37             delete(localBest.get(l).name, passengerlist);
38         }
39         passKm = localCount+passKm;
40     }
41     return passKm;
```

```
42  }
43
44  public void Chain(int end, int max, ArrayList<Passenger>
        passengerlist) {
45      if (end < max){
46          while(end != max) {
47              for (int i = 0; i < passengerlist.size(); i
                    ++) {
48                  if (passengerlist.get(i).start == end) {
49                      List1.add(passengerlist.get(i));
50                      counter = counter + passengerlist.get
                            (i).distance;
51                      Chain(passengerlist.get(i).end, max,
                            passengerlist);
52                      List1.remove(List1.size()-1);    // we
                             remove the passenger to move back
                             to where that passenger started
53                      counter = counter - passengerlist.get
                            (i).distance;  // we remove
                            passenger.distance because conter
                            is not a local variable
54                  }
55              }
56              end++;
57          }
58      }else {
59          if (counter > localCount) {
60              localCount = counter;
61              localBest = new ArrayList<Passenger>(List1);
62          }
63          else if (counter == localCount && List1.size() <
                localBest.size()) {
64              localCount = counter;
65              localBest = new ArrayList<Passenger>(List1);
66          }
67      }
68  }
69
70  public void delete(int seat, ArrayList<Passenger>
        passengerlist){
71      int upper = passengerlist.size()-1;
72      int lower = 0;
73      int current = (upper + lower)/2;
74      while(upper > lower && passengerlist.get(current).
            name != seat){
75          if(passengerlist.get(current).name > seat){
76              upper = current-1;
77          } else if (passengerlist.get(current).name < seat
                ){
78              lower = current+1;
```

```
79              }
80              current = (upper + lower)/2;
81          }
82          if (passengerlist.get(current).name == seat){
83              passengerlist.remove(current);
84          }
85      }
86
87      public ArrayList<ArrayList<Passenger>> getPassengers(){
88          return seats;
89      }
90      }
```

Dynamisk

```
1   import java.util.ArrayList;
2
3   public class Dynamic {
4   int passKm = 0;
5   int people = 0;
6   int min = 0;
7   int seats = 0;
8   int endstation = 0;
9   ArrayList<ArrayList<Passenger>> finalSeats = new
        ArrayList<ArrayList<Passenger>>();
10  public ArrayList<Passenger> passengerlist = new ArrayList
        <>();
11  ArrayList<ArrayList<Passenger>> sorted = new ArrayList<
        ArrayList<Passenger>>();
12
13  public Dynamic(int seats, int endstation) {
14      this.seats = seats;
15      this.endstation = endstation;
16      for(int i = 0; i < seats; i++){
17          ArrayList<Passenger> newSeat = new ArrayList<
                Passenger>();
18          finalSeats.add(newSeat);
19      }
20  }
21
22  public int call(ArrayList<Passenger> requests){
23      ArrayList<Passenger> requests2 = new ArrayList<
            Passenger>();
24      requests2.addAll(requests);
25      this.passengerlist = requests2;
26      for (int i = 0; i <= endstation; i++) {
27          sorted.add(new ArrayList<Passenger>());
28      }
```

```
29      for (int i = 0; i < passengerlist.size(); i++) {
30          sorted.get(passengerlist.get(i).end).add(
                passengerlist.get(i));
31      }
32      for (int i = 0; i < seats; i++) {
33          int counter = 0;
34          ArrayList<ArrayList<Passenger>> best = new
                ArrayList<ArrayList<Passenger>>();
35          for (int j = 0; j <= endstation; j++) {
36              best.add(new ArrayList<Passenger>());
37          }
38          Chain(min, endstation, best);
39          finalSeats.get(i).addAll(best.get(endstation));
40          for(int j = 0; j < best.get(endstation).size(); j
                ++) {
41              counter = counter + best.get(endstation).get(
                    j).distance;
42              sorted.get(best.get(endstation).get(j).end).
                    remove(best.get(endstation).get(j));
43          }
44          passKm = counter+passKm;
45      }
46      return passKm;
47  }
48
49  public void Chain(int min, int max, ArrayList<ArrayList<
        Passenger>> best) {
50      ArrayList<Passenger> localBest = new ArrayList<
            Passenger>();
51      for (int i = 1; i <= max; i++) {
52          boolean check = true;
53          for (int j = 0; j < sorted.get(i).size(); j++) {
54                  check = false;
55                  if (sorted.get(i).get(j).start == 0) {
56                      localBest = new ArrayList<Passenger
                            >();
57                      localBest.add(sorted.get(i).get(j));
58                  } else {
59                  int localDistance = 0;
60                  for (int k = 0; k < localBest.size(); k
                        ++) {
61                      localDistance = localDistance +
                            localBest.get(k).distance;
62                  }
63                  int checkDistance = sorted.get(i).get(j).
                        distance;
64                  for (int k = 0; k < best.get(sorted.get(i
                        ).get(j).start).size(); k++) {
65                      checkDistance = checkDistance + best.
                            get(sorted.get(i).get(j).start).
```

```
                    get(k).distance;
66                  }
67                  if (checkDistance > localDistance) {
                            //Here CompBest starts
68                      localBest = new ArrayList<Passenger
                            >();
69                      localBest.add(sorted.get(i).get(j));
70                      localBest.addAll(best.get(sorted.get(
                            i).get(j).start));
71                  } else if (checkDistance == localDistance
                        ) {
72                      if ((best.get(sorted.get(i).get(j).
                            start).size()+1) < localBest.size
                            ()) {
73                          localBest = new ArrayList<
                                Passenger>();
74                          localBest.add(sorted.get(i).get(j
                                ));
75                          localBest.addAll(best.get(sorted.
                                get(i).get(j).start));
76                      }
77                  }

                        //Here CompBest ends
78                  localDistance = 0;
79                  for (int k = 0; k < localBest.size(); k
                        ++) {
80                      localDistance = localDistance +
                            localBest.get(k).distance;
81                  }
82                  int prevDistance = 0;
83                  for (int k = 0; k < best.get(i-1).size();
                        k++) {
84                      prevDistance = prevDistance + best.
                            get(i-1).get(k).distance;
85                  }
86                  if (prevDistance > localDistance)
                                //Here CompPrev starts
87                      localBest = new ArrayList<Passenger
                            >();
88                      localBest.addAll(best.get(i-1));
89                  }else if (prevDistance == localDistance
                        && localBest.size() > best.get(i-1).
                        size()) {
90                      localBest = new ArrayList<Passenger
                            >();
91                      localBest.addAll(best.get(i-1));
92                  }

                        //Here CompPrev ends
```

```
93                  }
94              }
95              if (check && i > 0) {
96                  best.get(i).addAll(best.get(i-1));
97              }else {
98                  best.get(i).addAll(localBest);
99              }
100         }
101 }
102
103 public ArrayList<ArrayList<Passenger>> getPassengers(){
104     return finalSeats;
105 }
106 }
```

Forward-Backward

```
1  import java.util.ArrayList;
2
3  public class FB {
4  int counter = 0;
5  int allKm = 0;
6  int people = 0;
7  int min = 0;
8  int seats = 0;
9  int endstation = 0;
10 public ArrayList<Passenger> List1;
11 public ArrayList<Passenger> List2;
12 public ArrayList<Passenger> bestList;
13 public ArrayList<Passenger> specialList;
14 public ArrayList<Passenger> passengerlist;
15 public ArrayList<ArrayList<Passenger>> finalSeats = new
       ArrayList<ArrayList<Passenger>>();
16 boolean check = false;
17 Passenger tempKald = null;
18
19 public FB(int seats, int endstation) {
20         this.seats = seats;
21         this.endstation = endstation;
22         for(int i = 0; i < seats; i++){
23                 ArrayList<Passenger> newSeat = new
                       ArrayList<Passenger>();
24                 finalSeats.add(newSeat);
25         }
26 }
27
28 public int call(ArrayList<Passenger> requests){
29         ArrayList<Passenger> requests2 = new ArrayList<
               Passenger>();
```

```
30          requests2.addAll(requests);
31          this.passengerlist = requests2;
32          for (int i = 0; i < seats; i++) {
33                  int passKm = 0;
34                  specialList = new ArrayList<>();
35                  counter = 0;
36                  Chain(0, min, endstation);
37                  people += specialList.size();
38                  finalSeats.get(i).addAll(specialList);
39                  for(int l = 0; l < specialList.size(); l
                        ++){
40                          passKm = passKm + specialList.get
                                (l).distance;
41                          delete(specialList.get(l).name);
42                  }
43                  allKm = passKm+allKm;
44          }
45          for(int i=0; i<passengerlist.size(); i++) {
46          }
47
48          return allKm;
49  }
50
51  public void Chain(int end, int minimum, int max) {
52      List1 = new ArrayList<Passenger>();
53      List2 = new ArrayList<Passenger>();
54      int countForward = 0;
55      int countBackward = 0;
56      int countBest = 0;
57      bestList = null;
58      Passenger temp = null;
59      while(end < max) { // create chain, going forward
60          counter = 0;
61          for(int i=0; i<passengerlist.size(); i++) {
62              if(passengerlist.get(i).start == end &&
                    passengerlist.get(i).end <= max) {
63                  if(counter < passengerlist.get(i).
                        distance) {
64                      temp = passengerlist.get(i);
65                      counter = passengerlist.get(i).
                            distance;
66                  }
67              }
68          }
69          if(temp == null) {
70              end++;
71          }else {
72              List1.add(temp);
73              end = temp.end;
74              countForward = countForward+temp.distance;
```

```
75                  temp = null;
76              }
77          }
78          end = max;
79          while(end > minimum) { // create chain, going
                Backward
80              counter = 0;
81              for(int i=0; i<passengerlist.size(); i++) {
82                  if(passengerlist.get(i).end == end &&
                        passengerlist.get(i).start >= minimum) {
83                      if(counter < passengerlist.get(i).
                            distance) {
84                          temp = passengerlist.get(i);
85                          counter = passengerlist.get(i).
                                distance;
86                      }
87                  }
88              }
89              if(temp == null) {
90                  end--;
91              }else {
92                  List2.add(temp);
93                  end = temp.start;
94                  countBackward = countBackward+temp.distance;
95                  temp = null;
96              }
97          }
98          if(countForward > countBackward) { // comparison
                between the chains
99              countBest = countForward;
100             bestList =  new ArrayList<>(List1);
101         }else if(countForward < countBackward) {
102             countBest = countBackward;
103             bestList = new ArrayList<>(List2);
104         }else {
105             if(List1.size() > List2.size()) {
106                 countBest = countBackward;
107                 bestList = new ArrayList<>(List2);
108             }else {
109                 countBest = countForward;
110                 bestList = new ArrayList<>(List1);
111             }
112         }
113         for(int i=0; i < passengerlist.size(); i++) { //
                check for specialcase
114             if(passengerlist.get(i).distance >= countBest &&
                    passengerlist.get(i).start >= minimum &&
                    passengerlist.get(i).end <= max) {
115                 bestList = new ArrayList<>();
116                 bestList.add(passengerlist.get(i));
```

```
117                 countBest = passengerlist.get(i).distance;
118                 tempKald = passengerlist.get(i);
119                 check = true;
120             }
121         }
122         if (check) { // specialcase
123             check = false;
124             for (int i = 0; i < bestList.size(); i++) {
125                 specialList.add(bestList.get(i));
126             }
127             for (int j = 0; j < specialList.size(); j++) {
128                 delete(specialList.get(j).name);
129             }
130             int minTemp = tempKald.start;
131             int maxTemp = tempKald.end;
132             Chain(minimum, minimum, minTemp);
133             Chain(maxTemp, maxTemp, max);
134         }else {
135             for (int i = 0; i < bestList.size(); i++) {
136                 specialList.add(bestList.get(i));
137             }
138         }
139     }
140 
141     public void delete(int seat){
142         int upper = passengerlist.size()-1;
143         int lower = 0;
144         int current = (upper + lower)/2;
145 
146         while(upper > lower && passengerlist.get(current).
                name != seat) {
147             if(passengerlist.get(current).name > seat) {
148                 upper = current-1;
149             } else if (passengerlist.get(current).name < seat
                    ) {
150                 lower = current+1;
151             }
152             current = (upper + lower)/2;
153         }
154         if (passengerlist.size() == 0) {
155             //null
156         }else if (passengerlist.get(current).name == seat) {
157             passengerlist.remove(current);
158         }
159     }
160 
161     public ArrayList<ArrayList<Passenger>> getPassengers(){
162         return finalSeats;
163     }
164     }
```

Online/offline

```java
import java.util.ArrayList;

public class OnlineOff {
    ArrayList<Passenger> List1 = new ArrayList<>();
    private ArrayList<ArrayList<Passenger>> seats = new
        ArrayList<ArrayList<Passenger>>();
    boolean check;
    int min = 0;
    int passKm = 0;
    int people = 0;
    int capacity = 0;
    int stations = 0;

    public OnlineOff(int capacity, int stations) {
        this.capacity = capacity;
        this.stations = stations;

        for(int i = 0; i < capacity; i++) {
            ArrayList<Passenger> emptyList = new
                ArrayList<Passenger>();
            seats.add(emptyList);
        }
    }

    public int call(ArrayList<Passenger> passengerlist) {
        int[] density = new int[stations+1];
        for (int i = 0; i < passengerlist.size(); i++) {
            int k = passengerlist.get(i).start;
            check = true;
            while(k <= passengerlist.get(i).end-1 &&
                check) {
                // check if there is available space for
                    this passengers
                if (density[k] >= capacity) {
                    check = false;
                }
                k++;
            }
            if (check) {
                List1.add(passengerlist.get(i));
                passKm += passengerlist.get(i).distance;
                people++;
                for (int j = passengerlist.get(i).start;
                    j < passengerlist.get(i).end; j++) {
                    density[j] = density[j]+1;
                }
```

```
42              }
43          }
44          ArrayList<Passenger> sortedList = sortPassengers(
                List1);
45          placePassengers(sortedList);
46          return passKm;
47      }
48
49      public ArrayList<Passenger> sortPassengers(ArrayList<
            Passenger> passengers) {
50          // sort passengers by non-decreasing start
                station
51          ArrayList<Passenger> sortedList = new ArrayList<
                Passenger>();
52          for(int i = 0; i < stations; i++) {
53              for(int j = 0; j < passengers.size(); j++) {
54                  if(passengers.get(j).start == i) {
55                      sortedList.add(passengers.get(j));
56                  }
57              }
58          }
59          return sortedList;
60      }
61
62      public void placePassengers(ArrayList<Passenger>
            passengers) {
63          // place passengers with First Fit
64          for(int i = 0; i < passengers.size(); i++) {
65              for(int j = 0; j < capacity; j++) {
66                  if(seats.get(j).size() == 0 || passengers
                        .get(i).start >= seats.get(j).get(
                        seats.get(j).size()-1).end) {
67                      seats.get(j).add(passengers.get(i));
68                      break;
69                  }
70              }
71          }
72      }
73
74      public ArrayList<ArrayList<Passenger>> getPassengers
            () {
75          return seats;
76      }
77  }
```

Online

```
1  import java.util.ArrayList;
```

```
2
3  public class Online {
4      int min = 0;
5      int capacity = 0;
6      int stations = 0;
7      int max = 0;
8      private ArrayList<ArrayList<Passenger>> seats = new
           ArrayList<ArrayList<Passenger>>();
9
10 public Online(int capacity, int stations) {
11     this.capacity = capacity;
12     this.stations = stations;
13     this.max = stations;
14
15     for(int i = 0; i < capacity; i++) {
16         seats.add(new ArrayList<Passenger>());
17     }
18 }
19
20 public int call(ArrayList<Passenger> passengers) {
21     for(int i=0; i < passengers.size(); i++){
22         ArrayList<Integer> possSeats = getPossSeats(
               passengers.get(i));
23         if(possSeats.size() == 0) {
24             // No available seat
25         }
26         else {
27             int start = passengers.get(i).start;
28             int end = passengers.get(i).end;
29             int minRight = max;
30             int minLeft = max;
31             int minSeat = possSeats.get(0);
32             for(int h = 0; h < possSeats.size(); h++) {
                   // for each possible seat
33                 int left = passengers.get(i).start;
34                 int right = max-passengers.get(i).end;
35                 int seat = possSeats.get(h);
36                 for(int k = 0; k < seats.get(seat).size()
                       ; k++) { // for each passenger on seat
37                     int pStart = seats.get(seat).get(k).
                           start;
38                     int pEnd = seats.get(seat).get(k).end
                           ;
39                     if(start >= pEnd && start-pEnd < left
                           ) {
40                         left = start-pEnd;
41                         // find distance to passenger
                               closest to the left on seat
42                     }
43                     if(end <= pStart && pStart-end <
```

```
                        right) {
44                          right = pStart-end;
45                          // find distance to passenger
                                closest to the right on seat
46                      }
47                      if(left == 0 && right == 0) {
48                          // perfect match
49                          break;
50                      }
51                  }
52                  int minSpace = left;
53                  int maxSpace = right;
54                  if(left > right) {
55                      minSpace = right;
56                      maxSpace = left;
57                  }
58                  // update min. values
59                  if((minSpace < minLeft && minSpace <
                        minRight) || (minSpace == minLeft &&
                        maxSpace < minRight) || (minSpace ==
                        minRight && maxSpace < minLeft)) {
60                      minLeft = left;
61                      minRight = right;
62                      minSeat = seat;
63                  }
64                  if(minLeft == 0 && minRight == 0) {
65                      break;
66                  }
67              }
68              seats.get(minSeat).add(passengers.get(i));
69              // add passenger to seat minSeat
70          }
71      }
72      return getPassKm(seats);
73  }
74
75  public ArrayList<Integer> getPossSeats(Passenger
        passenger) {
76      ArrayList<Integer> possSeats = new ArrayList<Integer
            >();
77      for(int i = 0; i < seats.size(); i++) {
78          boolean isPoss = true;
79          for(int j = 0; j < seats.get(i).size(); j++) {
80              if(!(seats.get(i).get(j).start >= passenger.
                    end || seats.get(i).get(j).end <=
                    passenger.start)) {
81                  isPoss = false;
82                  break;
83              }
84          }
```

```
85              if(isPoss) {
86                  possSeats.add(i);
87              }
88          }
89          return possSeats;
90      }
91
92      public int getPassKm(ArrayList<ArrayList<Passenger>> list
            ) {
93          int passKm = 0;
94          for(int i=0; i < list.size(); i++) {
95              for(int j=0; j < list.get(i).size(); j++) {
96                  passKm += list.get(i).get(j).distance;
97              }
98          }
99          return passKm;
100     }
101
102     public ArrayList<ArrayList<Passenger>> getPassengers() {
103         return seats;
104     }
105     }
```