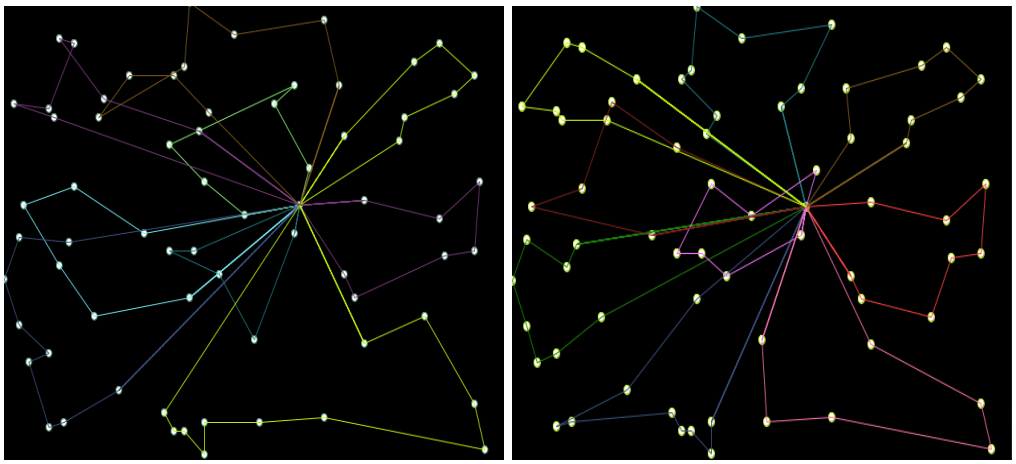
Capacitated Vehicle Routing Problem

Udarbejdet af:
Ebubekir Kayhan (69005)
Frederik Lyngsøe (69008)
Mads Sejer Pedersen (67892)
Martin List Syberg (67853)
Mikkel Elmelund Esbersen (67879)
Sebastian Nørager (69391)



Vejleder: Line Reinhardt

Roskilde Universitet

Nat Bach
2 Semester
Roskilde Universitet
Danmark
2020-01-04

1 Abstract

The transport sector is an important part of the modern world, so optimizing the sector leads to major savings in relation to shipping costs, travelling costs etc... This paper describes the *Vehicle Routing Problem*, and explores different ways of solving the *Vehicle Routing Problem*. Namely by using two different algorithms the *Clarke and Wright* alogrithm and our implementation of a *Cluster first, Route second algorithm*, where we use *K-means clustering* algorithm and afterwards 2-opt. The results show that over multiple iterations *K-means* can calculate shorter solutions with dataset A and with fewer routes than *Clarke and Wright*, but *Clarke and Wright* generates shorter solutions on dataset B. The calcutaion process of *K-means* and 2-opt takes longer. While *Clarke and Wright* on average generates solutions faster.

Indhold

List of Algorithms

2 Introduktion

Transport sektoren er en stor del af det moderne samfund, og mængden af energi, og penge der bruges i dette felt er meget højt. Derfor har det en stor interesse for mange at effektivisere transportsektoren. Hvis eksempelvis et fragt firma har mange forskellige kunder, og mange forskellige transportruter, så er det en fordel for dem, at de ruter, som bruges, er optimeret. Optimeringen af disse ruter kan løses i forbindelse med diverse problemstillinger - dette projekt har specifikt valgt at have fokus på problemstillingen *CVRP*, der opsætter problemet om at udregner forskellige ruter på én graf der alle starter og ender i samme knude, samtidig med at alle knuderne bliver besøgt. Præcis ligesom et fragt firma der skal udbringe en masse pakker til forskellige kunder, hvor ruterne starter og ender det samme sted nemlig depotet. *Vehicle Routing Problem*, kan f.eks løses med *Clarke and Wright*, og *K-means* med 2-opt, hvilket er de algoritmer projektet gør brug af. Projektet bruger Augerat et al. [6] datasættet, for at sammenligne løsningingen, og benchmarke algoritmerne mod hinanden, og den optimale løsning.

Optimeringen af *Vehicle Routing Problem* har ledt os til følgende problemformulering:

3 Problemformulering

Hvilken algoritme er bedst, *Clarke and Wright* eller *K-means* med 2-opt i forhold til at løse CVRP for Augerat et al. datasættet?

- *Hvordan implementerer vi Clarke and Wright, og K-means med 2-opt*
- *Hvad er løsningstiden og løsningskvaliteten for CVRP med henholdsvis Clarke and Wright, og K-means med 2-opt?*
- *Hvor effektiv er vores implementation i forhold til Augerat et al. datasættet?*

4 Semesterbinding

Semesterbindingen lyder som følgende "Samspil mellem teori, model, eksperiment og simulering i naturvidenskab". Projektet vil fokusere på sammenhængen mellem teorien, samt løsningen af *Vehicle Routing Problem*, i forlængelse af dette undersøger vi effektiviteten af 2 metoder med eksperimenter. Da projektet undersøger løsninger af *Vehicle Routing Problem*, ved hjælp af en række eksperimenter, så bliver semesterbindingen opfyldt. Projektet viser dermed samspillet mellem teori, og eksperiment i naturvidenskab. Projektets målgruppe er 2. semestersstuderende med matematik- og programmeringsbaggrund.

5 Vehicle Routing Problem

Dette projekt vil undersøge hvordan man eventuelt kunne løse et *CVRP*. For at kunne løse *CVRP* skal man først forstå *VRP*. *VRP* betegner problemstillingen, "Hvad er de optimale ruter for et sæt biler, som skal besøge et sæt knuder?", hvor alle knuder besøges én gang. *VRP* kunne eksempelvis bruges af en virksomhed, der har deres varer samlet på ét depot. I dette scenarie skal der laves ruter til antallet af køretøjer der bruges, hvor man prøver at minimere ruternes længde. Der er mange forskellige ruter der skal udregnes, og til tider har alle disse utallige muligheder. Der er så mange muligheder, at antallet kan blive så stort, at det er meget tidskrævende for en computer at udregne alle de forskellige løsninger, og derfor bruger man heuristikker til at løse problemet. En heuristik er en teknik hvor der prioriteres hastighed frem for nøjagtighed, og man kan se en heuristik som en genvej til gode løsninger uden at gennemgå alle mulige løsninger. I et *VRP* skal alle ruterne starte og slutte i det samme sted, nemlig depotet. Kompleksiteten af problemet kan afhænge af både antallet af kunder og antallet af køretøjer der er til rådighed. En virksomhed har typisk ikke mulighed for et uendeligt antal af biler, så denne parameter er ofte en restriktion for løsningen af problemet.
Generelt kan man beskrive *Vehicle Routing Problem* med disse tre restriktioner (i,ii,iii) [15].

i Alle knuder/kunder i grafen er besøgt én gang

ii Alle ruter/køretøjer starter og ender i depotet

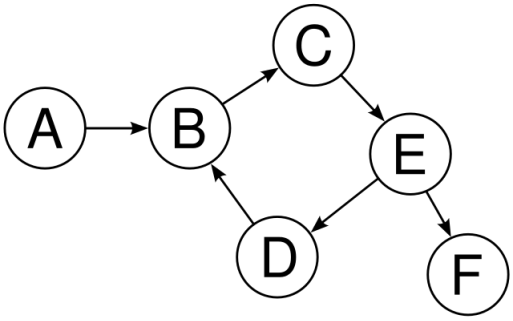
iii Bestemte restriktioner er tilfredstillet (Capacity, Time Windows...)

6 Grafteori

Hele ideen omkring *VRP*'s problemstilling "Hvad er de optimale ruter for et sæt biler som skal besøge et set knuder?"er at få oversat problemet til noget der kan tydes på en graf. Ideen omkring hvad en graf egentlig er, og definitionen herom, stammer faktisk fra en anden problemstilling værende kendt som *The Königsberg Bridge Problem*. *The Königsberg Bridge Problem* er kort fortalt et problem som gjorde sig gældende før grafteori var forstået. Problemet omhandlede et antal broer og øer, hvor de undrede sig over om det var muligt, at krydse alle broer præcis én gang, og ende samme sted som de startede. Det blev opdaget at problemet ikke var hvordan broerne så ud, så tegningen kunne oversættes til noget så simpelt som punkter og streger herimellem [21].

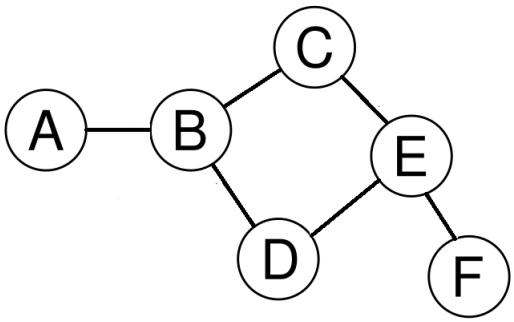
Figur 1: Illustration af overgang fra tegning til graf [20]

En graf består altså af x antal knuder, der i forhold til *VRP* har forskellige koordinater - knuderne kan passende anses som kunderne. Alle disse punkter er forbundet via kanter, her kan en kant eksempelvis ses som værende én vej man kan følge for at komme fra den ene knude til den anden. Inden for disse grafer skelnes der mellem to typer af grafer, de ensrettede, og dem der ikke er ensrettede. Når en graf er ensrettet betyder det, at en kant kun har én retning man skal følge, dermed er det altså ikke muligt at følge den i begge retninger [16]. Denne egenskab på kanterne kan visualiseres som pile, hvor pilens retning netop er den retning, som er den eneste mulige retning at følge.



Figur 2: Illustration af en ensrettet graf [16]

Den anden type af grafer er dem som ikke er ensrettede, og det betyder at kanterne ikke har en specifik retning. Uden en specifik retning, så er det muligt at følge hver kant begge retninger. Visualiseringen af sådan en graf ville have kanter uden pilespidser, så det vil hermed bare være streger, som vist i figur 3.



Figur 3: Illustration af en graf uden retning [16]

Det skal hermed også forstås, at der ikke kun er forskel visuelt, men også måden knuderne kan sammensættes, og hermed den matematiske forståelse. Lad det gøre sig gældende, at grafen G består af et sæt af knuder, V, og et sæt af kanter, E. Ved den forståelse, så vil knuderne være et *ordnet par*. Et ordnet par betyder hvis, man som eksempel, kigger tilbage på figur 2, hvor V består af $V = \{A, B, C, D, E, F\}$, og nu hvor kanterne har bestemte retninger, så vil E bestå af $E = \{(A, B), (B, C), (C, E), (E, D), (D, B), (E, F)\}$. Her er sættet af kanter altså kun de retninger, som det er muligt at følge, nu hvor kanterne har en bestemt retning. Modsat den ensrettede graf, så vil man her kunne se forskel på sættet af kanter i den ikke ensrettede figur 3. I den ikke ensrettede graf, så vil V bestå af det samme $V = \{A, B, C, D, E, F\}$, men da kanterne ikke har en bestemt retning, så vil det nu være muligt at gå fra eksempelvis D til B og D til E, hvor det før kun var muligt at gå fra D til B. Med denne viden, så vil sættet af kanterne være $E = \{(A, B), (B, A), (B, C), (C, B), (C, E), (E, D), (D, E), (D, B), (E, F), (F, E)\}$ og her ser man altså, hvilken betydelig forskel de to typer af grafer har [16].

7 Kompleksitetsteori

For at løse *VRP* og finde de optimale løsninger for ruterne i forhold til deres distance, så skal der bruges algoritmer. Der er adskillige algoritmer til rådighed, så man skal tage højde for forskellige egenskaber som hver algoritme har. En af disse egenskaber er tidskompleksitet. Tidskompleksiteten af en algoritme, er hvor mange udregninger algoritmen bruger for at finde et svar. Typen og antallet af aktioner algoritmen skal bruge, påvirker tiden, og mængden af ressourcer algoritmen skal bruge, for at udføre udregningen [5]. Det er derfor optimalt at have et program, hvor kompleksiteten er lav, så programmet kan komme frem til svaret hurtigere. I *Vehicle Routing Problem* kan kompleksiteten blive høj, hvis man bruger en algoritme der søger efter den mest optimale løsning. Sådan en algoritme er eksakt[17], hvis den altid kommer frem til den optimale løsning. En eksakt algoritme er *deterministisk* dvs., at den altid kommer frem til det samme svar, med det samme input. I modsætning så er en algoritme ikke deterministisk hvis den udregner forskellige svar på det samme input [7]. En komplet algoritme vil sige, at den altid kommer frem til en løsning, hvis en løsning findes - denne løsning behøver ikke at være optimal, men den skal være lovlig. En lovlig løsning må ikke bryde nogle af de restriktioner der kunne være, f.eks at bilen besøger den samme knude to gange i *VRP*, hvis det skete ville løsningen **ikke** være lovlig (se sektion 5). Hvis en løsning ikke findes, så vil en komplet

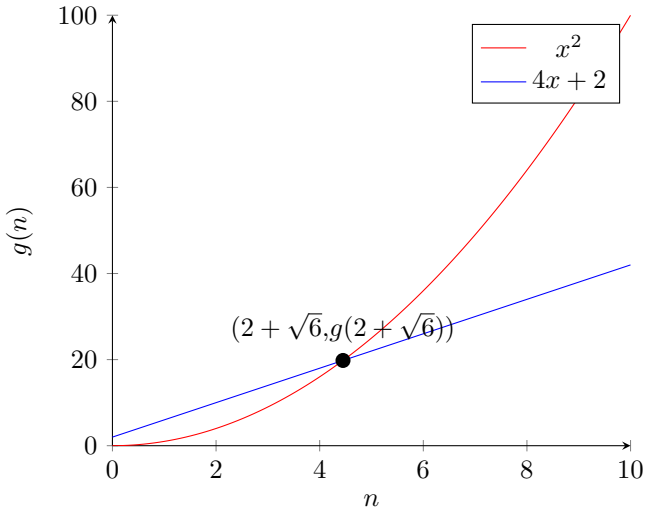
algoritme også rapportere dette - hermed vil der ikke opstå en situation for en komplet algoritme, hvor den ikke har mulighed for at give et form for svar [13]. Da kompleksiteten kan blive høj for sådanne algoritmer, så kan det være en fordel at have en algoritme der kan filtrere, hvilke løsninger den undersøger, så den udelukker de svar der giver et meget dårligt resultat. Det vil gøre at algoritmen skal lave færre udregninger, og man sparer dermed ressourcer på udregningen. Til at beskrive tidskompleksiteten af vores algoritmer har vi valgt at bruge *big-O-notation*. *Big-O* er defineret som (ligning 1):

$$O(g(n)) = \{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\} \quad (1)$$

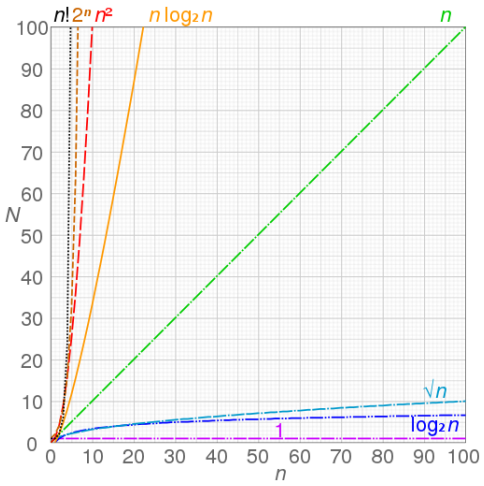
Så en funktions kompleksitet afhænger af hvad de største faktorer i funktionen er, det er denne faktor der styrer kompleksiteten af funktionen. For eksempel med $T(n)$ funktionen:

$$T(n) = n^2 + 4n + 2$$

Hvis man ser på vækstraterne i forhold til n^2 og $4n$ har n^2 en højere vækstrate, og det er den afgørende faktor for big-O [9]. Som eksempel når man lader $n \to 1000$, eftersom man går mod 1000 så vil de mindre udtryk i funktionen, $4 * n + 2$, udgøre en mindre værdi af T(n) jo højere n værdi der forekommer. Det er dog vigtigt at udpege, at der vil være værdier for n, hvor n^2 er lavere end $4n + 2$, og denne grænse går ved $2 + \sqrt{6} < n$. Når vi har en n som tilfredsstiller $2 + \sqrt{6} < n$, så vil kompleksiteten, og hermed *Big O notation* være: $O(n^2)$. For at vise at $T(n)$ tilhører $O(n^2)$, så vælger vi selv en konstant $c = 1$, dermed får vi: $T(n_0) = (n_0)^2 + 4 * n_0 + 2 \leq 1 * n_0^2$ Ved at konstanten er 1 så kan vi vælge et heltal n større end $2 + \sqrt{6}$ for hvilket ligningen 1 gælder. Dvs. at $T(n) \in O(n^2)$.



Kompleksitet kan beskrives, og illustreres på forskellige måder, figur 4 giver et overblik over diverse kompleksiteter:



Figur 4: Forskellige typer af kompleksiteter visuelt[19]

Det ses på figur 4, at der er stor forskel på hvor hurtigt de forskellige funktioner stiger i mængden af nødvendige udregninger. Især $n!$ som vokser faktorielt, f.eks $10! = 3628800$, og $12! = 479001600$, hvor n eksempelvis kunne være antallet af knuder på en graf. Ved at have en algoritme med lav kompleksitet, kan man dermed spare på mængden af iterationer.

F.eks. hvis du har graf med mange knuder (høj n-værdi), og du også bruger en algoritme med høj kompleksitet kan det betyde, at det ikke er muligt for en computer at beregne en lovlig løsning (i fornuftig tid), og derfor bruges forskellige heuristikker til at løse problemet med en tilnærmelsesvis løsning. I følge [2] er *CVRP* en del af *NP-hard* problemerne. Hvilket vil sige at man ikke har fundet ud af om man kan udregne den optimale løsning i polynomiel tid f.eks n^100, og ikke i faktoriel tid $n!$, hvor n bestemmer antallet af knuder. Derfor bruger man heuristikker til at finde en tilnærmelsesvis løsning i stedet, da man ikke kan udregne den optimale med de eksisterende algoritmer. Projektet vil specielt fokusere på brugen af *Clarke and Wright* algoritmen, og *K-means* med 2-opt til at løse *VRP*'et. Når en optimal eller tilnærmelsesvis løsning findes, så er det vigtigt at huske på, at selv en lille forbedring på hver køretøjs rute, kan give en større forbedring overordnet set.

7.1 Sammenhæng mellem Vehicle Routing Problem og Travelling Salesman Problem

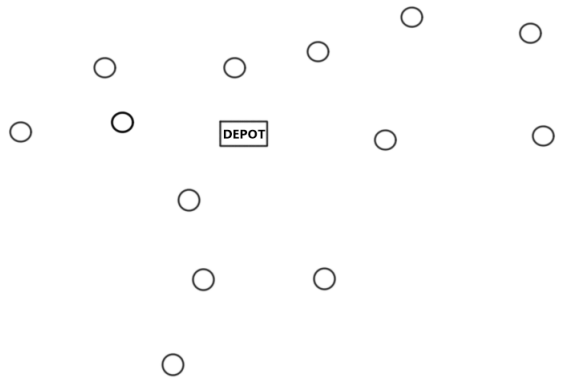
For at kunne yderligere forstå hvad *VRP* er, er det nødvendigt at kende til *Travelling Salesman Problem (TSP)*. For at finde en optimal rute for *TSP*, skal alle knuder besøges én gang med ét enkelt køretøj. Målet ved at løse *TSP*, er at finde den optimale rute, der går igennem alle knuder på grafen én gang, hvor ruten starter og ender det samme sted. Hvilket ville reducere distancen for køretøjet så der bliver sparet benzin, penge og tid. Når det gælder *TSP*, må mængden af kunder (knuder) kun præcist besøges én enkelt gang, dvs. at hvis man antager at ruten består af 10 kunder er der et krav om, at hver knude skal besøges, men kun én gang og derefter slutte på start- og slutpunktet A. Forskellen mellem *Travelling Salesman Problem* og *Vehicle Routing Problem* er, at et *VRP* er en slags udvidelse af en *TSP*. Et *VRP* kan producere adskillige ruter som forskellige køretøjer skal igennem, hvor en *TSP* kun har én enkel rute med ét enkelt køretøj. Som nævnt tidligere i rapporten har *VRP*'et restriktioner for at gøre 'opgaverne' specifikke. *Capacitated Vehicle Routing Problem (CVRP)* er et af dem. *CVRP*'et skal igennem kunder (knuder) hvor køretøjet/køretøjer levere varer. Varerne har nogle parametre som f.eks. vægt og volumen og køretøjerne har en kapacitet, altså en maksimumskapacitet, som betyder at når køretøjerne skal igennem en rute, må der kapaciteten aldrig overskrides. Transporten skal udføres på den mest tidsbesparende måde, mens kunderne besøges. Målet er at levere og tage imod varer på en måde hvor man spare tid og penge, og på samme tid må maksimumskapaciteten ikke overskrides. Derudover er det også essentielt at få minimeret rejsetiden eller mængden/længden af ruterne. Kort fortalt betyder denne restriktion, at alt efter hvor mange køretøjer der gøres i brug fra depotet, har køretøjerne en grænse for hvor mange varer der kan være i køretøjet. *CVRP* og *TSP* hænger sammen, og det er netop fordi nogle af kravene fra *TSP* også gælder i *CVRP.* F.eks.:

- Finde de mest optimale ruter i forhold til den samlede længde.
- Kravet om at alle kunder (knuder) i en rute skal præcist besøges én enkelt gang og ende ved depotet.

Vehicle Routing Problem (VRP) beskriver problemet om at optimere antallet, og længden af de forskellige ruter. Ved at have en graf med n antal knuder, disse knuder kan med fordel ses som kunder. *Vehicle Routing Problem* bruges ofte til at forbedre et udbringningsfirmas transportruter. Ved at forbedre ruterne kan firmaet spare penge, og samtidig formindske energi forbruget. Der er adskillige restriktioner af *VRP*'et, så som *Vehicle Routing Problem with Time Windows (VRPTW)*, hvor alle afleveringerne af pakkerne skal være indenfor et bestemt tidsrum. *VRPTW* kan sammenkobles med *CVRP* så man skal følge reglerne fra begge, så man får *CVRPTW*. Med disse restriktioner kan man gøre *VRP*'et endnu mere specifikt, hvis man for eksempel er en virksomhed hvor leveringen inden for et bestemt tidsrum er vigtigt.

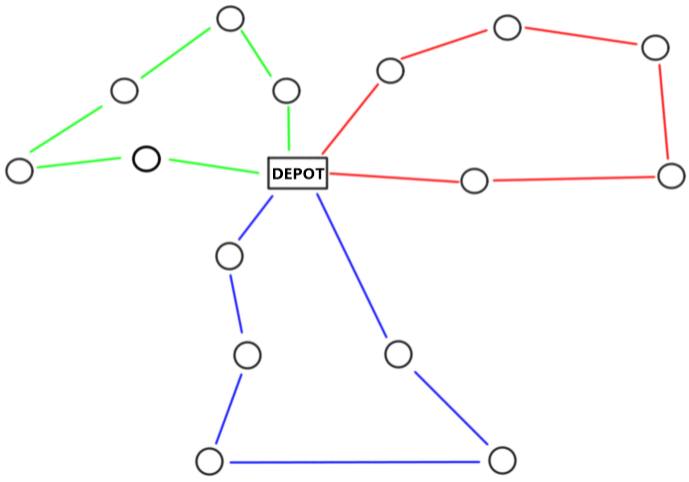
7.2 Konstruktionen af en Graf i forhold til VRP

Når man skal løse *Vehicle Routing Problem*, starter man med en graf, hvor man skal beregne et antal ruter som opfylder sætningerne, som også blev givet tidligere (i,ii,iii) på listen i sektion 5. I det generelle *VRP* kender man både kunderne, og længden mellem de forskellige kunder i forvejen. Altså er løsningen f.eks. ikke en *VRP* hvis man har en rute der er i den samme knude 2 gange, eller hvis en rute ender et andet sted end i depotet. På figur 5 er et eksempel på en graf med ét depot hvor alle knuderne er beskrevet i euklidisk rum, i $\mathbb{R}^2$, hvor vi har ortonormale vektorer som basis.



Figur 5: Eksempel på én graf med *14* knuder og ét depot

Med denne graf kan man herefter lave én *VRP* rute. Figur 6 illustrerer hvordan en *VRP* rute kunne se ud for denne graf med 3 køretøjer, og 14 knuder og et depot, hvor de forskellige farver illustrerer de forskellige køretøjers ruter.



Figur 6: Eksempel på *VRP* med 3 ruter fra depotet

Hvis antallet af køretøjer man har til rådighed bliver ændret, så vil det højst sandsynligt også medføre at alle ruterne ændres. Så hvis der f.eks. kun stilles 2 køretøjer til rådighed for figur 6 så kan der kun dannes 2

ruter, og deres set af knuder vil derfor blive større. Og hvis man f.eks. havde 4 køretøjer til rådighed så ville mængden af knuder de skulle besøge formindskes.

7.3 Klargørelse af Augerat et al. Data Set

Til løsningen af *CVRP* bruges et datasæt som vi fremover kalder Augerat et al. datasæt [6]. Augerat et al. datasæt er et benchmark(sammenligningsgrundlag) til *CVRP*, hvor i et enkelt datasæt er der beskrevet de forskellige parametre til at løse en *CVRP* såsom hvor meget kapacitet der er til hver enkelt køretøj, hvor hver kunde er henne, hvad tid de kan serviceres og hvor meget varerene fylder. For de forskellige datasæt kan findes optimale løsninger som dermed kan bruges til at afprøve forskellige algoritmer. Augerat et al. dataset består af et varierende antal af punkter, hvor hver punkt har én tildelt x og y værdi. Det første punkt i datasættet er depotet, som også er der hvor alle ruterne skal starte og slutte. Data punkterne har én værdi kaldet *Demand* tildelt, som kan forstås som antallet af varer på det enkelte punkt. De forskellige værdier er tilfældige, og er med til at gøre grafen mere virkelighedsnær. Augerat et al. datasæt bruges til at generere en graf, som kan løses med én eller flere algoritmer, hvor alle ruterne skal opfylde betingelserne givet tidligere [1]. Ved at man laver kanter imellem de forskellige punkter, som også kunne repræsentere den vej et køretøj skulle følge. Alle Augerat et al. datasæt er blevet løst i forskellige grader, - endda nogle med optimale løsninger. Datasættet bruges som et benchmark(sammenligningsgrundlag) for at teste forskellige heuristikkers løsningskvalitet og tid, mod enten den optimale løsning eller andres heuristikker. Det er også dette datasæt vi bruger til at danne en *CVRP*, og hermed for at afprøve vores algoritmer. Der findes 2 afgrene af Augerat datasættet, hhv. sæt A og sæt B. Sæt A's knuder er alle genereret tilfældigt, men sæt B's knuder er genereret så de ligger i mere 'naturlige' *clusters*.

8 Clarke and Wright

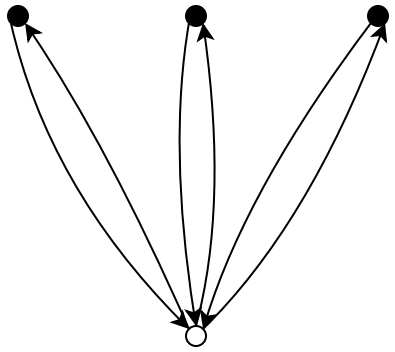
Den første algoritme, som projektet vil bruge til løsningen af *VRP* er *Clarke and Wright* algoritmen. *Clarke and Wright* algoritmen er en af de mest velkendte heuristikker for løsningen af *CVRP* [1], og har mange variationer der prøver at forbedre den [17]. Den bruges til at forbedre løsningen fra initial[2], som illustreres på figur 7, til den nye udregnede via brugen af algoritmen. Den ændrer både antallet og rækkefølgen af de knuder køretøjet besøger. Simpelt kan den forklares ved at man samler to forskellige ruter, til én rute i stedet. Når man har mange ruter, kan det forbedre den samlede længde af alle ruterne betydeligt, på baggrund af trekantsuligheden. Trekantsuligheden er at når man har en side af en trekant vil den altid være mindre end summen af de 2 andre sider. *Clarke and Wright* finder altid den samme løsning, og er dermed en deterministisk heuristik. Selvom man kører den mange gange på den samme graf får man altid den samme løsning [8].

Grund princip, *Clarke and Wright* savings algoritmen (CW) starter med n-antal knuder. Efter knuderne bliver defineret, så beregnes en initial løsning for alle knuder, hvor en rute bliver allokeret til hver knude som er forbundet tilbage til depotet (algortime 1). Herefter beregnes savings for alle 'knude-knude-depot' par (algortime 2). Det formindsker den samlede længde af ruten, hvilket er det heuristikken har til formål at gøre.

Vi har taget udgangspunkt i parallel-CW algoritmen. Som udgangspunkt fungerer CW på den måde at der beregnes savings med de euklidiske afstande for hver 'knude-knude-depot' par. Efterfølgende søger man så

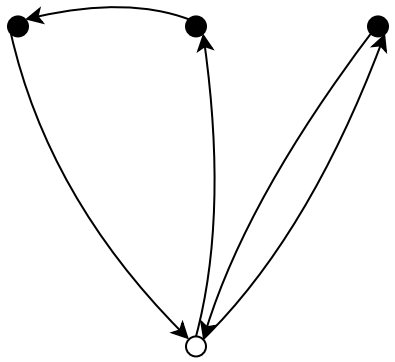
[1] Se afsnit 7.2

[2] Algoritme 1



Figur 7: Initial løsning for *Clarke and Wright*. [STEP 1]

efter hvor ens savings er størst, og kan dermed danne par der hvor afstanden er mindst.



Figur 8: Sammenfletning af 2 ruter til én [STEP 3]

På figur 8 ses det hvordan *Clarke and Wright* samler tre ruter til to ruter, hvilket formindsker antallet af biler man skulle bruge fra **tre** til **to**, og den samlede længde af ruten. Den kunne også sammenflette de to ruter til én, så der kun er behov for en bil i stedet for to.

Selve dannelsen af ruten kan ske under bestemte parametre, f.eks. om knuden/kunden passer ind i ruten uden at den overskrider kapaciteten.

8.1 Clarke and Wright Algoritme

Strukturen af grafen i vores program er baseret på to objekter: ruten og knuden. Selve knuden indeholder al information om lokation, kapacitetsbehov. Ruten indeholder en liste over knuder, samt. savings, og dens bekostning. Ruterne gemmes yderligere i en *arrayliste* over alle ruter (algoritme 1, linje 7), så vi kan sammenligne dem senere hen. Selve ruterne har også en *arrayliste* i sig hvor knuderne i bliver gemt og per algoritme 1, starter alle ruterne med depoterne i deres nulte indeks, og deres "start"knude i det første (hhv. linje 1 og 2). Til sidst lægger vi knuden i og ruten i et hashmap, hvor i er vores key, og ruten er vores value, som vi bruger senere hen i scanneren (algoritme 3).

Den initielle løsningen har en tidskompleksitet på n (linje 3), da den gennemgår den initielle liste som repræsenterer n, og det er den initielle programmet optimerer.

Efter vi har beregnet den initielle løsningen beregner vi savings for alle knuderne i knudelisten ved algoritme 2, som bliver lagt til en en *arrayliste* $\rightarrow$ savingslist. Herefter sorteres denne savingslist for ruterne

Algorithm 1: Initial Løsning

input : En knudeliste, med alle knuder fra Augerat et al. data

output: En liste over alle initial ruter

```
1  initRoute (knudeListe = {knude_i...knude_n})
2      Initialiser ruter = {rute_0...rute_n};
3      foreach knude i ∈ knudeListe do
4          Initialiser, rute;
5          Tilføj depot, til rute;
6          Tilføj i, til rute;
7          Tilføj rute, til ruter;
8          Put Vertex i (key), Rute rute (value) i ruteIndex;
```

som er defineret ved *compareTo* som bliver kaldt når man sammenligner *savings* for hver rute til hinanden når vi kalder på sort() fra Collections biblioteket i native Java.

Algorithm 2: Savings Algoritme

```
1  calculateSavings ()
2      foreach knude i ∈ knudeListe do
3          foreach knude j ∈ knudeListe do
4              if i ≠ j ∧ i ≠ depot ∧ j ≠ depot then
5                  cost0i = |depot − i|;
6                  cost0j = |depot − j|;
7                  costij = |i − j|;
8                  savings = cost0i + cost0j - costij;
9                  Lav en ny Rute, tempRute;
10                 Tilføj i, til rute;
11                 Tilføj j, til rute;
12                 Tilføj savings til tempRute;
13                 Tilføj tempRute til savingsList
14 sort(savingsList);
```

Tidskompleksiteten af savings-algoritmen kan beregnes på baggrund af at et nested forloop (algoritme 2, linje 2-3) har kompleksiteten n^2, som laver en savingsList som er n^2 lang (linje 14). Sort() funktionen bliver kaldt på savingsList (algoritme 2, linje 15) og er en modificeret merge sort fra Java's eget biblioteket som har kompleksiteten $nlog(n)$.

Dermed bliver tidskompleksiteten for *savings-algoritmen* : $\mathbf{n^2 log(n^2)}$.

Selve *Clarke and Wright* algoritmen har vi implementeret for sig selv, hvor den er blevet modificeret for at tage forbehold for vores restriktioner - capacity. Efter vi har beregnet savingsList, - altså listen over hvilke vertices der har størst savings med hinanden, så kan vi scanne igennem den og begynde at merge de forskellige par.

Scanneren gør brug af ruteIndex, (algoritme 3, linje 5-6) for at finde hvilke rute som hver vertex tilhører

Algorithm 3: Scanner Algoritme

```
1  scanner ()
2      foreach rute ∈ savingsList do
3          Vertex i = vertex_0 ∈ rute;
4          Vertex j = vertex_n ∈ rute;
5          Rute ruteI = ruteIndex(i);
6          Rute ruteJ = ruteIndex(j);
7          if ruteI ≠ ruteJ then
8              indexI = index af vertexI i forhold til ruteI;
9              indexJ = index af vertexJ i forhold til ruteJ;
10             merge(ruteI,ruteJ,indexI,indexJ);
```

i forhold til de overordnet ruter, men *ikke* i forhold til savingsList. Hvor vores key er en vertex og value er ruten som vertex'en tilhører, og hvis ruterne som i og j tilhører ikke er ens, så skal de merges.

Algorithm 4: Merge Algoritme

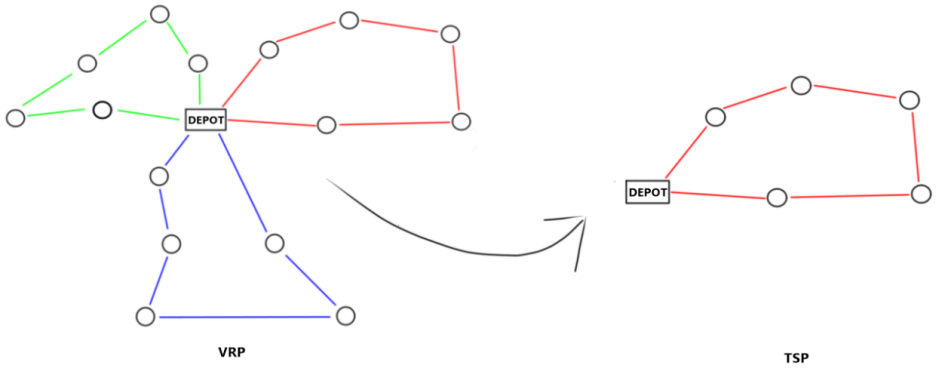
```
1  merge (ruteI,ruteJ,indexI,indexJ)
2      if indexI == 1 ∧ indexJ == index_n ∈ ruteJ then
3          Initialisér tempRuteI = {vertex_1...vertex_n} ∈ ruteI;
4          if ruteJ.cost + ruteI.cost < RUTECAPACITET then
5              Fjern depot fra tempRuteI;
6              Tilføj {vertex_1...vertex_n} ∈ tempRuteI til ruteJ;
7              Beregn den nye længde for ruteJ;
8              Hent ruteI fra Ruter og Initialisér ruteI;
9              foreach vertex ∈ ruteJ do
10                 Put vertex (key), ruteJ (value) i ruteIndex;
```

I merge (algoritme 4) starter vi med at kigge på om indekset af knuderne hhv. for indeks $i = 1$ og indeks j = sidste indeks fordi når vi skal ligge listerne af knuderne fra den ene rute til den anden skal vi sørge for at I og J ikke ligger i interiøret af ruten - altså I og J skal være nabo med depotet. Derefter tjekker vi om mergen overholder kapaciteten af ruterne (i forhold til $CVRP$), og hvis det bliver overholdt fjernes depotet fra *temp* ruten (det sørger for at depotet ikke bliver lagt i rute J flere gange), og så kan vi tilføje alle punkterne fra rute I til J. Vi initialiserer rute I så den bliver tom, og til sidst sørger vi for at knuderne har den korrekte rute gemt i ruteIndex. Da nøglerne i hashmaps er unikke, vil det slette den forrige rute som er gemt for knuden. Når alle knuderne er i en rute så er *Clarke and Wright* algoritmen færdig.

9 Cluster first, Route second

I projektet har vi implementeret en version af *Cluster first, Route second*, hvilket til vores bedste viden ikke er blevet lavet før. *Cluster first, Route second* er en metode med to faser, hvor der først beregnes nogle *clusters* (første fase), som grupperer datasættet. Efter grupperingen ser man på hver gruppe som værende

en rute som kan løses (anden fase). Her går vi altså fra at have en *VRP* til adskillige *TSP'er* - da hver gruppes rute kan ses som værende en *TSP*-rute (figur 9). En tredje fase kan implementeres, denne fase skal se tilbage på løsningen, og se om den er lovlig - hvis den ikke er, så reparere den så den bliver lovlig [17].



Figur 9: Illustration for *VRP* til *TSP*

Der kan være en fordel først at beregne *clusters* for alle knuder, og derefter beregne ruten for clusteret med en algoritme. Hvis vi ser på det fra et kompleksitetsteoretisk perspektiv, så deler du n op i flere dele, og det kan være en stor fordel, fordi de fleste *VRP* algoritmer typisk har eksponentiel vækst i deres kompleksitet. For eksempel er: $10! \neq 5! + 5!$. Hvor $5! + 5!$ repræsenterer 2 *clusters*, hvor de hver har 120 kombinationer (5!), hvorimod hvis man tager alle 10 knuder som et *cluster* er der $10! = 3,628,800$ kombinationer.

Vi har valgt at bruge *K-means* til den første fase for at danne vores *clusters* og hhv. 2-opt til den anden fase for at løse *clusterne*.

9.1 K-means

K-means algoritmen er en af disse *clustering* algoritmer. En *clustering* algoritme bruges til at gruppere datasæt. Og det er netop dét, som algoritmen her gør, *K-means* er en algoritmisk metode hvor, hvis ens datasæt kan repræsenteres på en graf (i forhold til afsnit 5), så kan den automatisk gruppere datasættet ind i k-antal *clusters*. K'et her skal forstås som værende en variabel med en numerisk-værdi. Det virker på den måde, at der startes med k-antal initial centroider, som putter data ind i hver sit *cluster* afhængig af punkternes distance til de forskellige centroider. Når alle punkter er blevet indsat i et *cluster*, så vil centroiderne opdatere sine positioner. Hvis man anser punkter som værende *instances*, så foregår metoden således på punkt form:

i *Each instance* d_i *is assigned to its closest centroid* [18].

ii *Each centroid* C_j *is updated to be the mean of its constituent instances* [18].

Altså bliver centroiderne først opdateret, når alle punkter er blevet indsat i clustrene. Dette projekt vil gøre brug af en mindre modificeret *K-means* på flere forskellige måder. En af disse modificeringen ses i stopkriteriet, da *K-means* algoritmen normalt har sit eget stopkriterie, når den ikke længere får indsat nye

punkter i gruppen efter den har rykket sig til den gennemsnitlige afstand [18]. Dog vil den modificerede *K-means* have et stopkriterie på baggrund af hvor mange opdateringer der skal ske for centroiderne, og hermed clustrene. Antallet af opdateringer er bestemt af en variabel. En anden modificering er at der gøres brug af clustre med en bestemt kapacitet, denne kapacitet er bestemt af en variabel. Det betyder at clustrene kun kan have et bestemt antal punkter inden i sig inden kapaciteten er fyldt op. Det betyder at der kan opstå situationer, hvor punkter ikke bliver taget højde for, hvis kapaciteten er for lav, og antallet af punkter er for høje - eller hvis antallet af clustre er for lav. Alt dette betyder at algoritmen kan være ret så nyttig at bruge, men den kan også omvendt være upræcis, og dens grupperinger kan være ubrugelige i visse datasæt – alt det afhænger af k-værdien (antallet af clustre), men også centroidernes initielle position. Nøglen til en god gruppering af et datasæt er at finde den optimale k-værdi, altså antallet af centroider, og dermed også antallet af grupper. En optimal K-værdi finder vi på baggrund af alle knudernes *demand*. Hermed er det ikke kun vigtigt at finde en K-værdi som skaber gode grupperinger, men endnu vigtigere er det at finde en K-værdi, som tager højde for alle punkterne, således at alle punkter er indsat i en af clustrene. Hvis man ved hvor stort et datasæt man har, og man ved hvad kapaciteten af clustrene er sat til, så kan man hurtigt finde en minimums værdi, således at alle punkter bliver indsat i clustrene. Herefter kan man eventuelt finde en bedre K-værdi ved flere gennemkørsler af forskellige K-værdier.
Når K-værdien er fundet, køres færdiggørelsen af *K-means*, og datasættet vil hermed være inddelt i grupper, og så kan problemet nu løses på andenvis. I stedet for at se hele sættet som et *VRP*, så kan man i stedet kigge på hver gruppe som et *Travelling Salesman Problem*, *TSP* - se figur 9.

9.2 Implementering af K-means & Travelling Salesman Problem

Implementeringen af *K-means* og heraf strukturen af denne i programmet er først af alt baseret på alle knuderne og hermed de *centroids*, der bliver dannet i *K-means*. Den første del af *K-means* kalder vi for den initiale del af *K-means*, denne del ses i Algorithm 5.

Algorithm 5: *K-means*, Initial

```
1 for int i = 0; i < k; i++ do
2     float randY = randomNumbGen(0,højesteKnudeYposition);
3     float randX = randomNumbGen(0,højesteKnudeXposition);
4     initialiser Centroide c;
5     c.position = (randX,randY);
6     tilføj c til en liste over alle centroider;
```

Næste del af algoritmen afgør hvilken centroide knuden er tættest på. Denne afgørelse sker ved at man kigger på hver knuder og tjekker distancen mellem den pågældende centroide og knuden, hvis distancen mellem knuden og den pågældende centroide er bedre end den hidtil gemte centroide, så vil den pågældende centroide nu være den gemte centroide. Når alle knuder har fået tildelt en gemt centroide, så bliver knuderne indsat i den gemte centroides *cluster*. Denne del er også beskrevet i algoritme 6.

Den første del af *K-means* starter med at danne K-antal centroids, hvor hver af disse centroids får skabt et *cluster* til sig selv. Centroids kan anses som værende en form for magnet, da det er et punkt som skal have de nærmeste knuder sat ind i sit *cluster*, hvor *clusteret* er en unik arrayliste til en centroid. Hver *centroid* starter med en tilfældig position, som er begrænset af selve vinduet af programmet. Herefter starter programmet

Algorithm 6: *K-means, Distance Calculator*

```
Input: knudeListe, kListe
foreach centroide c ∈ kListe do
    Initialiser c's cluster;
    Initialiser kapaciteten af clusteret;
foreach knude ∈ knudeListe do
    Initialiser tempCentroid;
    foreach centroide nyCentroide ∈ kListe do
        if d(gemtCentroide, knude) ∈ knude ≥ d(nyCentroide, knude) then
            d(gemtCentroide, knude) ∈ knude = d(nyCentroide, knude);
            tempCentroid = nyCentroide;
    if tempCentroid != null then
        Tilføj knuden til tempCentroid's cluster;
        Opdater kapaciteten af tempCentroid;
    initialiser d(gemtCentroide, knude) ∈ knude
```

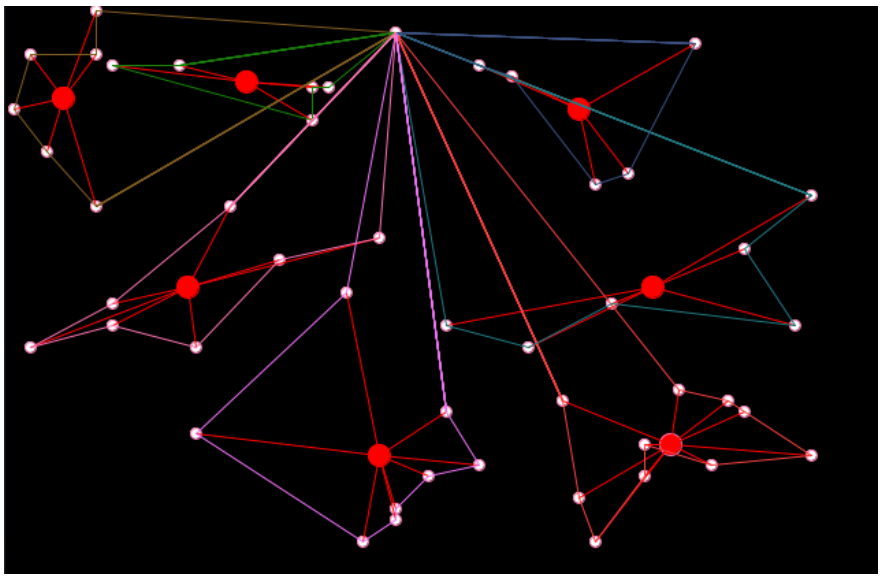
søgningen igennem hver *centroid* for hvert punkt, og kigger på distancen mellem hver knude og hver *centroid*. Når søgningen er færdig, vil hver knude blive gemt i sin nærmeste *cluster*.
Her afsluttes den initielle del, og herefter starter "*m*eans"delen af *K-means*. Når alle punkter er gemt i *clusters*, så vil hvert enkelt centroid forny sin position i forhold til den gennemsnitlige afstand til hvert punkt gemt i *clusteret*. Dette gøres ved at tage hvert punkts position og hermed tages gennemsnittet af disse, både x og y og indsætter det som værende centroidens nye position. Herefter skal *clusteret* tømmes for punkter, således at der kan komme nye punkter ind i disse *clusters*. Når disse *clusters* er blevet tømt, så kan koden begynde på ny, og det vil den fortsætte med at gøre indtil stopkriteriet bliver ramt. Stopkriteriet for den *K-means* der bruges i projektet afgøres af det ønskede antal af gennemkørsler - det kunne være eksempelvis 3 eller 4 gennemkørsler. Disse grupper som er blevet dannet af *K-means* indtil videre, bliver der bygget videre på, da de hver især vil blive betragtet som *TSP*-ruter, og hermed vil der yderligere blive brugt en algoritme til at løse disse.

Den anden del som lige er blevet beskrevet, means delen, kalder vi for en *merge*-metode, som er den del hvor positionen bliver opdateret i forhold til *clusterets* punkter. Måden hvorpå kode-delen af denne metode fungerer på er således:

Disse grupperinger har nu åbnet op for muligheden til at anse ruterne som værende ruter for sig, og hermed som *TSP*-ruter.

Algorithm 7: *K-means, Update Centroid*

```
Input: knudeListe, kListe
foreach centroide ∈ kListe do
    Initialiser tempx, tempy;
    foreach knude ∈ centroidens cluster do
        tempx = tempx + knudens x-koordinat;
        tempy = tempy + knudens y-koordinat;
    if centroid.Cluster.size() != null then
        tempx = tempx/centroid.Cluster.size();
        tempy = tempy/centroid.Cluster.size();
        centroid.position.x = tempx;
        centroid.position.y = tempy;
```

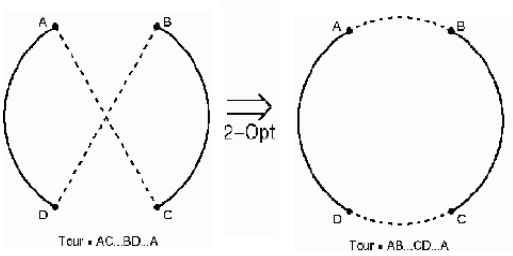


Figur 10: Visuelisering af K-means med 2-opt hvor ruter og clusters bliver vist.

9.3 2-opt og Brute force

2-opt er en algoritme der har til formål at forbedre en ikke optimal initiel rute, ved at ændre hvilke noder der forbindes. 2-opt ændrer to kanter ad gangen, algoritmen gør det ved at bytte rundt på kanter i tilfælde af, at de to kanter krydser hinanden. 2-opt er baseret på ideen at det gør en rute længere hvis der er to kanter der krydser hinanden. Efter at 2-opt har byttet rundt på alle kanter der krydser, er algoritmen hermed færdig. Ruten burde dermed være blevet kortere. Det er også muligt at ændre mere end to kanter ad gangen, for eksempel med tre 3-opt eller 4-opt [14].

2-opt algoritmen bliver brugt på de forskellige *clusters* der er dannet af *K-means* til at udregne en *TSP* på disse *clusters*. Algoritmen er et eksempel på hvordan vi har implementeret 2-opt algoritmen.



Figur 11: Eksempel på et 2-opt bytte
[10]

Algorithm 8: Implementeringen af 2-opt

Input: kListe

```
1  for knudeI ∈ kListe do
2      for knudeJ = knudeI + 2 ∈ kListe cluster do
3          d1 = |i + i_{+1}| + |j, j_{+1}| ;
4          d2 = |i + j| + |i_1, j_{+1}|;
5      if d2 < d1 then
6          int min = minimize((i_{+1}mod(n), (j_{+1}mod(n));
7          int max = maximize((i_{+1}mod(n), (j_{+1}mod(n));
8          new arrayList[] middle = copy kListe from min to max ;
9          invert arrayList[] middle ;
10         new arrayList[] left = copy kListe from 0 to min ;
11         new arrayList[] right = copy kListe from max to n ;
12         copy arrayList(left) to kList starting from 0 to left.length ;
13         copy arrayList(middle) to kList starting from 0 to middle.length ;
14         copy arrayList(right) to klist starting from left.length+right.length to right.left ;
```

Brute force er en løsnings metode der går igennem alle muligheder, og vælger den bedste af dem, altså at man udregner alle løsninger for et problem. Vi implementerer ikke *Bruteforce*, i vores program. Ved at bruge *Brute force* er resultatet altid det optimale for den enkelte TSP, da man har været igennem alle muligheder. Et eksempel på hvordan algoritmen fungerer: Man har en pinkode på [2039], hvis man vil have *Brute force*, til at udregne pinkoden checker den [0000], [0001], [0002], osv. I dette tilfælde ville det tage 2039 iterationer før pinkoden er fundet. Problemet ved *Brute force* er hvis antallet af udregninger den skal udfører er så højt at det tager for lang tid, f.eks ved at udregne alle rute for en graf med 20 knuder. Da der i dette tilfælde er $n! = 2.432 * 10^{18}$ muligheder. Vi bruger *Brute force* på de forskellige *clusters* dannet af *K-means*, da de typisk ikke har en knudemængde på over 10, og når det er under 10 er antallet af udregninger ikke alt for stort.

9.4 K-means med 2-opt Tidsompleksitet

For at finde tidskompleksiteten af *K-means*, så finder man tidskompleksiteten af de forskellige dele, og lægger dem sammen. Den første del af *K-means* algoritmen 6, har en tidskompleksitet på k, da den ikke gør brug af n, og bruger et *for-loop* fra 0 til k (antallet af *centroids*). Den anden del af *K-means* algoritmen 13, fra linje 1-3,

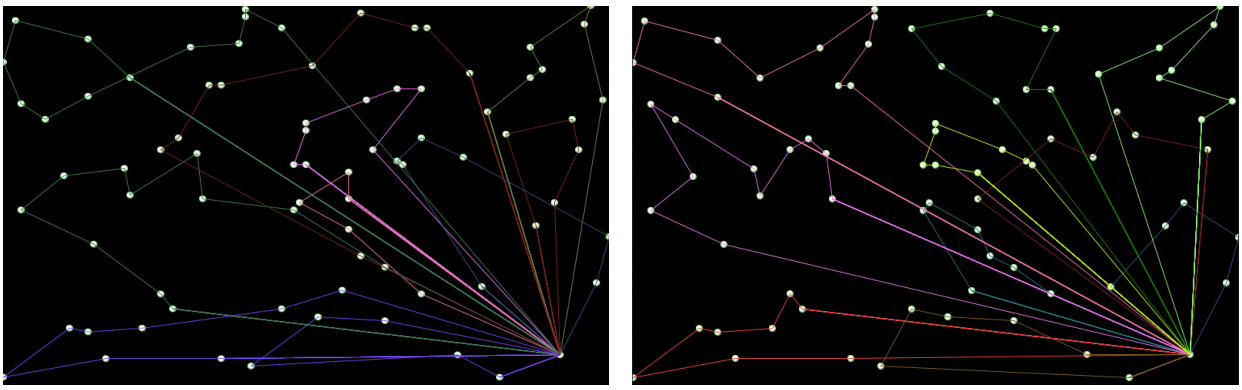
har en tidskompleksitet på k, da *for-loopet* itererer igennem alle centroids (k). Den næste del af algoritmen linje 4-13, har et *nested for-loop*, der gennemgår alle *centroids* i listen for hver knude og beregner deres distance, så den beregner $k * n$ Den sidste del af *K-means* algoritmen 10, har et *nested for-loop*, der itererer igennem alle *centroids* i listen, og udregner en *tempx* og *tempy*, for hver knude, dermed er tidskompleksiteten af dette $k*n$. 2-opt algoritmen 14, fra linje 1-2 har et *nested for-loop*, hvor den sammenligner alle knuder med hinanden altså $n*n$. Så får den dermed en tidskompleksitet på n^2 Vi beregner den samlede tidskompleksitet for *K-means* med 2-opt til at være $k + n * k + k * n + n^2$, og herefter med big-O-notation, at $\mathbf{O(n^2)}$.

10 Struktur af programmet

Projektets program er skrevet i *Java*, og har blandt andet brugt *Proccesing* som *library/bibliotek*, altså har vi brugt funktionerne fra *Proccesing* i programmet, og ikke kun funktionerne fra standard *Java*. For at samarbejde omkring programmeringen så har vi benyttet os af *IntelliJ*, og *GitHub*, hvilket har gjort det muligt at programmere online. Programmet er opdelt i forskellige *classes* og består af en *Core*, for bedre at have overblik over hele programmet. Eksempelvis har alle algoritmerne hver deres *class*, så de nemt kan bruges. Programmet gør især brug af *Arrayliste*, fremfor et *Array*, *Arrayliste* har den fordel at de ikke har en fikseret størrelse. Det er nødvendigt for programmet, da ruterne ikke altid har den samme længde, og man ville få en fejl-meddelelse med et *Array*. *ArrayList* bruges både i *Clarke and Wright*, og *K-means* algortimen til at håndterer grafen og ruten. I *Clarke and Wright* algoritmen benytter vi også data strukturen *Hash-Maps*. *Hashmaps* fungerer på samme måde som *arrays*, fordelen er dog at man kan bruge alt som nøgle, så længe nøglerne er unikke. Mens *arrays* kun kan bruge en *integer* som index. *Hashmaps* har af den grund dog ikke en naturlig rækkefølge og kan derfor ikke itereres på samme måde som arrays.

11 Databehandling

CVRP datasæt fra Augerat et al. [1] Vi har kørt vores *Clarke and Wright* algoritme og vores *K-means* sammen med vores 2-opt algoritme på datasættet [6].



(a) Clarke & Wright ruter

(b) *K-means* og 2-opt ruter

Figur 12: Ruter fra datasæt *A n80 k10*

Som vises på figur 12a dannes der ruter ud fra depotet som er nede i højre hjørne. Med vores *Clarke and Wright* får vi i alt 10 ruter på dette datasæt som har en sammenlagt længde på **1865** som kan aflæses på

tabel 1. Dette er desværre ikke den optimale længde som bliver opgivet i datasættet til at være 1763.

Med vores *K-means* og 2-opt har vi kørt over samme datasæt flere gange da vores algoritme har en vis tilfældighed i at *K-means clusteret's* koordinater er placeret tilfældigt til at starte med hvorefter *clusterne* bliver rykket til centrummet af alle de kunder som bliver forbundet.

På figur 12b kan man se en løsning hvor alle kunderne bliver besøgt, denne løsning blev fundet hvor vi kørte vores algoritme igennem 50000 gange, hvor alle kunderne blev besøgt og den korteste løsning var på 1836. På tabel 1 kan man se at den gennemsnitlige løsning var betydeligvis større end *Clarke and Wright*.

Det ses på tabel 1 at differencen mellem den optimale og det vores program har udregnet ikke er så stor igen omkring 5% for de to algoritmer. Dog er afvigelsen mellem den gennemsnitlig værdi for *K-means* med datasæt $n32 - k5$, og den optimale på 13.6 % $(891 - 784)/784 = 0.136$, hvilket ikke er specielt godt. Så hvis man kun har mulighed for at lave en iteration af en af algoritmerne, ville det være en fordel at bruge *Clarke and Wright*, da den er deterministisk, og altid kommer frem til det samme resultat. *Clarke and Wright* udregner en rute længde der er kortere end *K-means* gennemsnits resultat.

For at sammenligne algoritmerne på andet end deres rutelængder har vi også målt hvor lang tid det tager at gennemkøre dem. På tabel 1 kan man se at *K-means* med 2-opt tager længere tid tid end *Clarke & Wright*, men dette er for 50000 gennemkørsler, så hvis man er oppe på en stor mængde af *K-means* gennemkørsler kan det godt tage betydeligvist længere tid end *Clarke & Wright*. Men hvis man kigger på en enkelt gennemkørsel fra A-n80-k10 på tabel 1 tager det $\frac{20{,}97ms}{3093{,}3ms/50000} = 339$ altså 339 gange så lang tid at bruge *Clarke and Wright* hvilket er betydeligt længere, men nok ret ubetydeligt for en bruger.

Det sidste sammenligningsgrundlag vi bruger er deres gennemsnitlige afvigelse fra det optimale fra hvert datasæt, så vi kan se hvor gode *K-means* og *Clarke & Wright* er til at tilnærme sig den optimale løsning ved tilfærdigt genereret positioner for knuderne (datasæt A), og knuder i *clusters* (datasæt B).

Afvigelserne i tabel 1 og 2 er udregnet med formlen fra ligning 2

$$\frac{(\text{Målt} - \text{Optimal})}{\text{Optimal}} * 100 = \text{Afvigelse i procent} \tag{2}$$

Datasæt A		*Clarke & Wright*				*K-Means* med 2-opt ud af 50000				
Datasæt	Optimal længde	Længde	Afvigelse fra optimal	Antal ruter	Tid (ms)	Bedste længde	Afvigelse fra optimal	Gns. længde	STD	Tid (ms)
n32-k5	784	844	7.7 %	5	7.34	837	6.8%	891	32	928.0
n33-k5	661	694	5.0 %	5	10.49	696*	5.3%	766	36	958.4
n33-k6	742	776	4.6 %	7	8.39	749	0.9%	864	60	1038.1
n34-k5	778	811	4.2 %	6	8.39	796	2.3%	911	58	1016.1
n36-k5	799	845	5.8 %	5	10.49	819	2.5%	895	41	1048.6
n37-k5	669	705	5.4 %	5	9.44	724*	8.2%	786	35	1106.2
n37-k6	949	980	3.3 %	6	10.49	981*	3.4%	1084	50	1151.3
n38-k5	730	784	7.4 %	6	11.53	765	4.8%	866	54	1220.5
n39-k5	822	910	10.7%	5	13.63	868	5.6%	1007	62	1377.8
n39-k6	831	883	6.2 %	6	10.49	870	4.8%	964	46	1209.0
n44-k6	937	981	4.7 %	6	10.49	1012*	8.1%	1115	43	1323.3
n45-k6	944	1043	10.5%	8	13.63	984	4.2%	1170	84	1478.5
n45-k7	1146	1214	5.9 %	7	11.53	1205	5.1%	1301	54	1415.6
n46-k7	914	938	2.6 %	7	11.53	956*	4.6%	1065	35	1355.8
n48-k7	1073	1124	4.8 %	7	13.63	1111	3.5%	1223	36	1452.3
n53-k7	1010	1093	8.2 %	8	12.58	1096*	8.6%	1244	51	1735.4
n54-k7	1167	1202	3.0 %	7	11.53	1208*	3.5%	1359	60	1808.8
n55-k9	1073	1110	3.5 %	9	14.68	1101	2.7%	1234	54	1760.6
n60-k9	1354	1419	4.8 %	9	13.63	1398	3.3%	1509	48	2012.2
n61-k9	1034	1105	6.9 %	10	13.63	1112*	7.5%	1293	61	2099.2
n62-k8	1288	1352	5.0 %	8	15.73	1338	3.9%	1497	50	2156.9
n63-k9	1616	1702	5.3 %	10	12.58	1684	4.2%	1861	81	2211.4
n63-k10	1314	1358	3.3 %	10	17.83	1348	2.6%	1496	59	2148.5
n64-k9	1401	1463	4.4 %	9	14.68	1443	3.0%	1605	82	2312.1
n69-k9	1159	1209	4.3 %	9	19.40	1187	2.4%	1329	56	2519.2
n80-k10	1763	1865	5.8 %	10	20.97	1836	4.2%	2044	85	3093.3

Tabel 1: Data fra datasæt (A,random). (* angiver hvor *K-means* har højere længde end *CW*) For hvert datasæt er der en K værdi som er antallet af ruter for *K-means*. Optimale værdier fra ref [3].

Datasæt B		*Clarke & Wright*				*K-Means* med 2-opt ud af 50000				
Datasæt	Optimal længde	Længde	Afvigelse fra optimal	Antal ruter	Tid (ms)	Bedste længde	Afvigelse fra optimal	Gns. længde	STD	Tid (ms)
n31-k5	672	685	1.9 %	5	8.39	680	1.2%	691	13	977.3
n34-k5	788	809	2.6 %	5	8.39	794	0.7%	847	17	1094.7
n35-k5	955	1028	7.7 %	5	8.39	1021	6.9%	1040	15	914.4
n38-k6	805	834	3.6 %	6	8.39	860*	6.8%	933	24	1166.0
n39-k5	549	568	3.5 %	5	8.39	570*	3.9%	637	21	1283.5
n41-k6	829	905	9.2 %	7	12.58	907*	9.4%	1021	21	1149.2
n43-k6	742	763	2.8 %	6	8.39	756	1.9%	839	33	1350.6
n44-k7	909	956	5.1 %	7	12.58	947	4.2%	1009	29	1417.7
n45-k5	751	758	0.9 %	6	12.58	806*	7.3%	941	55	1589.6
n45-k6	678	733	8.2 %	7	12.58	764*	12.6%	830	15	1509.9
n50-k7	741	750	1.2 %	7	8.39	794*	7.2%	819	14	1505.8
n50-k8	1312	1361	3.7 %	8	16.78	1354	3.2%	1463	45	1740.6
n51-k7	1018	1192	17.1%	9	16.78	1140	12.0%	1273	32	1669.3
n52-k7	747	769	3.0 %	7	12.58	782*	4.6%	887	35	1644.2
n56-k7	707	737	4.2 %	7	12.58	765*	8.2%	804	16	1807.7
n57-k7	1144	1266	10.7%	8	12.58	1253	9.6%	1450	84	1929.4
n57-k9	1598	1662	4.0 %	9	16.78	1683*	5.3%	1769	41	1900.0
n63-k10	1496	1608	7.5 %	11	16.78	1567	4.7%	1683	51	2172.6
n64-k9	861	927	7.7 %	10	12.58	933*	8.4%	1079	50	2269.1
n66-k9	1316	1421	8.0 %	10	16.78	1367	3.9%	1455	58	2281.7
n67-k10	1032	1109	7.5 %	11	16.78	1078	4.5%	1201	64	2290.1
n68-k9	1272	1336	5.0 %	9	16.78	1320	3.8%	1457	60	2327.8
n78-k10	1221	1296	6.2 %	10	16.78	1301*	6.6%	1448	65	2822.8

Tabel 2: Data fra datasæt (B, *clusters*) vi har indsamlet data fra. (***** angiver hvor *K-means* har højere længde end *CW*) For hvert datasæt er der en K værdi som er antallet af ruter for *K-means*. Optimale værdier fra ref [3].

	Gns. afvigelse fra optimal	
Datasæt	*CW*	*K-means* med 2-opt
A	5.5%	4.5%
B	5.7%	6.0%

Tabel 3: Gennemsnitlig afvigelse fra optimal længde for vores algoritmer

12 Diskussion

En anden form for *VRP*, som kunne være relevant at undersøge, i stedet for at kigge på *CVRP*, kunne være *VRPTW*. Hvis man i stedet løste *VRPTW*, så ville algoritmerne også forholde sig anderledes til et givent datasæt. Det ville eksempelvis ses i *K-means*, når den skulle inddele datasættet i *clustre*. Dette kunne implementeres ved at give *K-means* øjne for hvorvidt et punkt havde mulighed for at blive indsat i *clusteret* i forhold til dets kapacitet, så der ikke endes op med et *cluster* som ikke kan gennemføres rutemæssigt grundet en overskredet kapacitet.

Så vi havde egentlig en forståelse af, hvordan tidsvinduer funktionelt skulle fungere som parameter for punkterne. Men med henblik på selve koden, så viste denne implementering sig dog at være en for stor mundfuld for os, så vi nåede ikke at implementere tidsvinduer, selvom det var intentionen.

12.1 Sammenligning af de to forskellige metoder

De forskellige ruter har alle sammen fået udregnet den samlede længde. På disse længder kan man se, at selvom *K-means* og 2-opt kan få en kortere løsning end *Clarke and Wright*, så får de gennemsnitligt betydeligvist længere ruter end *Clarke and Wright* løsningen. Man har selv mulighed for at forøge chancerne for at få dannet en bedre rute gennem *K-means*. Det kan gøres ved at ændre på den variable, som bestemmer hvor mange gange centroiderne skal opdatere deres *clustre* - jo højere værdi denne variabel har, jo flere gange skal der opdateres, og jo højere er chancen for at få bedre grupper. Dermed kan brugeren af programmet selv vurdere om man har brug for en hurtig, men stadig god løsning eller om man har tid til at vente på at *K-means*, udregner en bedre løsning. F.eks hvis man er et udbringnings firma der bruger den samme rute hver gang, ville det være bedst at bruge *K-means*, da man har tiden til at vente på at få en bedre rute, og forbedringen af længden ville gøre en forskel på længere sigt. I et andet tilfælde hvor ruterne ændrer sig hver dag ville det her være bedre at bruge *Clarke and Wright*, da den gennemsnitligt kommer frem til en løsning, der er bedre end *K-means*.

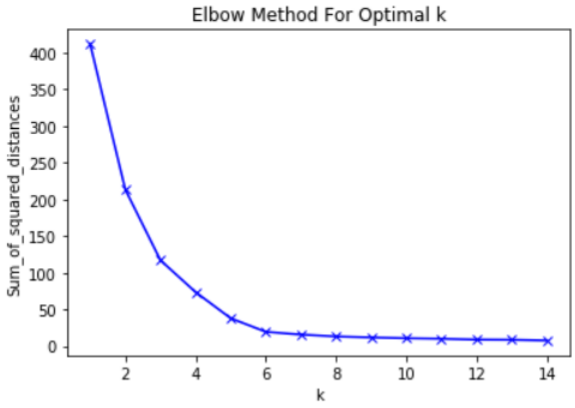
Resultaterne fra vores program ligger rimelig tæt op af de optimale løsninger, for de forskellige datasæt som der er udført udregninger på. Som man kan se på tabellerne for datasæt A (tabel 1) og B (tabel 2) ligger vores afvigelser fra det optimale mellem 0,7% og 17,1% for de to datasæt. Men hvis man kigger på det gennemsnitlige på tabel 3 ligger vi gennemsnitligt set omkring 5-6% fra det optimale.

12.2 K-means stopkriterie

I projektet bliver det kort nævnt, at den modificerede *K-means*, som bliver brugt i projektet, ser bort fra den *"det normale"* stopkriterie. Det normale stopkriterie var ved at se på, hvorvidt der blev indsat nye punkter i *clustrene*, eller om det var de samme punkter flere opdateringer i streg. Hvis der ikke blev indsat nye punkter i *clustrene*, så ville det betyde, at de perfekte grupper, ifølge algoritmen, er fundet og hermed vil den stoppe. Det kan dog forestilles, at hvis man er uheldig, så kan sådan et stopkritierie blive opnået for tidligt. Hvis stopkriteriet bliver opnået for tidligt, så kan det ske, at grupperne egentlig kunne forbedres en hel del mere, end det som de er endt med at blive til i sådan en situation. Derfor kunne det være spændende at have brugt en blanding af de to stopkriterier for at se, om det eventuelt ville danne et bedre stopkriterie.

12.3 K-means Optimal K

Projektets *K-means* har en K-værdi, som er fundet ved hjælp af viden omkring kapaciteten på de anvendte *clusters*, men en anden, mere anvendt metode til at finde den optimale K-værdi er *"The Elbow Method"*, albue metoden. Denne metode går ud på at prøve forskellige k-værdier, og hermed se på hver enkeltes *means*¸ som er den gennemsnitlige kvadreret afstand fra alle punkter til deres centroide. Man kan hermed lave en graf over *means* for hver k-værdi, og det burde danne en graf der ser således ud:

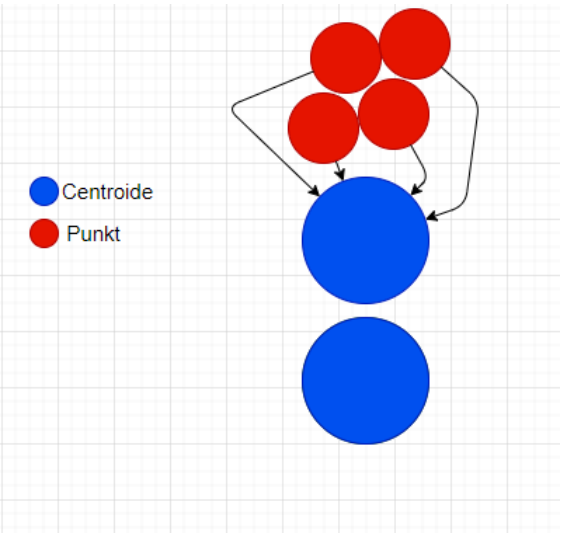


Figur 13: Eksempel på *elbow method*-graf [4]

I figur 13 kan man se at der dannes en *"albue"*, her vil man gerne have k-værdien, hvor albue kanten ligger på - 5 eller 6. Man vil ikke have en for stor k-værdi, men man skal ikke have en for stor *means*, eller afstand mellem punkter og *clusters*, så derfor vil man tage 5 eller 6, da de begge har en lav means i forhold til deres værdi [11]. Man kan heller ikke se en forskel fra 6 og op efter, som er betydelig i den grad at man ville vælge den k-værdi frem for 6 eller 5. Denne metode kan måske og anses som værende lidt af en gætte metode, i det at der ikke bliver fundet en bestemt K-værdi, men flere forskellige K-værdier, som sammenlignes. Og ud fra denne sammenligning skal man kunne vide, hvilken der passer bedst til sit datasæt.

En anden ting der kunne havde være spændende var at ændre på *centroidernes* initielle placering. I projektet er den initielle placering bestående af en tilfældig x-værdi og en tilfældig y-værdi, som er begrænset af vinduets størrelse. Det kunne være interessant at ændre den initielle placering til at være lig et tilfældigt punkts position. Dette ville forhindre *centroider* i at placere sig ude i de yderste hjørne, hvor der oftest ikke er punkter. Og det ville også medføre en *"sikker"* start, altså at alle *centroider* vil have et punkt indsat i sit *cluster* - pånær hvis der er flere *centroider* end der er punkter. En situation hvor en initial centroide ikke får indsat et punkt i sit *cluster* kunne godt opstå, ved den måde som bliver brugt i projektet, en sådan situation kan illustreres som i figur 14.

I figur 14 ses det at den eneste *centroide*, den som er længst væk fra de røde punkter, ikke får nogen punkter i sit *cluster*, fordi alle punkterne er tættere på det andet *cluster*. Hvis vi i stedet havde valgt at sætte de initielle punkter som værende et tilfældigt punkts position, så ville denne situation ikke opstå, da hvert *cluster* altid mindst vil have ét punkt.



Figur 14: Illustration af en situation, hvor en centroide ikke får punkter til sit *cluster*

12.4 Simulated Annealing

Vi overvejede også at bruge *Simulated Annealing* til at udregne *CVRP*. *Simulated Annealing* er en meta-heuristik, da den bruges til at forbedre en anden heuristik. Den kan for eksempel bruges på *K-means*, til at forbedre resultatet. Den starter med at opsætte nogle parametre, en initial løsning, nedkølings konstant, en start temperatur, og en anden løsning end den initielle [12]. Temperaturen falder efter hver iteration, altså hver gang den har udregnet en ny løsning, og den bliver ved med at lave udregninger indtil temperaturen er faldet under en konstant. Stop kriteriet styres af dem der laver programmet, og et godt stop kriterier findes ved at afprøver en masse forskellige nedkølings konstanter, indtil man finder en god én. Efter hver iteration checker den om den nye løsning er bedre end den gamle, hvis den er det bliver den gemt separat. Nogle gange tager den også en løsning selvom den er dårligere end den gamle, dette gør den med en faldende chance der afhænger af hvad temperaturen er. Det gør at den for undersøgt et større område en hvis den bare stoppede når den havde fundet et lokalt optimum. Ved at bruge *Simulated Annealing* kan man få en algoritme, som fx *K-means* til at tage en beslutning som måske ikke giver et bedre resultat til at starte med, men i sidste ende er et bedre valg. F.eks. ved at søge efter den højeste bakke i et landsskab så er det ikke nok at teste den første bakke top, selvom man er det højeste sted på den bakke, da alle retninger man kan gå går nedad. Derfor bliver man også nødt til at teste de andre bakker, og se om der er nogle af dem som er højere.

Det ses også på illustrationen hvor der er stor forskel på de lokale maksimum og det globale. I dette tilfælde ville det ikke være nok at bruge en *"Hill Climber"*, algoritme til at finde en god løsning, da den ofte vil finde et lokal minimum, og ikke det globale. En *"Hill Climber"* er en algoritme der går igennem løsningerne en for en, og tager løsningen hvis den er bedre end den gamle indtil der kun er dårligere løsningerne på begge sider. Derfor vil den ofte ende på et lokal maksimum og ikke det globale, da man i mange tilfælde vil have mange lokale maksimum i ens løsningerne.

Algoritme 9 er et *Pseudokode* eksempel på *Simulated Annealing*, hvor det ses hvordan algoritmen fungerer i de forskellige trin:

Vi nåede ikke at implementere *Simulated Annealing*, da implementation af de to andre algoritmer tog længere tid end forventet. Vi startede med at implementere algoritmen men stoppede, da det var bedre at få

Algorithm 9: *Simulated Annealing*

```
Input: knudeListe
while T > T_slut do
    for i = 0; i < i_max; i++ do
        if S0 < S then
            S = S0
        else
            p = exp((S0 − S/T))
        if p < random(0, 1) then
            S = S0
        T = a ∗ T
```

de to andre algoritme implementeret ordentligt, inden vi gik i gang med at implementere *Simulated Annealing*. Hvis vi havde nået at implementere den så havde vi haft endnu en algoritme at sammenligne med. Det er svært at sige hvor vidt den havde været hurtigere til at generere en god løsning end de to andre algoritmer, eller om den havde været dårligere. *Simulated annealing* havde dog været god at sammenligne med *K-means*, da begge algoritmer er **ikke-deterministisk**, og her havde det været spændende at se hvilken en af dem der udregnede den korteste løsning, og hvilken af algoritmerne der var hurtigst til at udregne løsningen.

13 Konklusion

Hvorvidt *Clark & Wright* eller *K-means* med 2-opt er bedst til at løse *VRP* med Augerat et al. datasættet, er egentlig en smagssag i den forstand, at det afhænger af flere parameter. Fra databehandlingen ser man, at både *Clark & Wright* og *K-means* med 2-opt er i stand til at finde ruter med få procent afvigelse fra den optimale rute. *K-means* med 2-opt er den bedste til at finde den korteste rute på datasæt A (se tabel 3), selvom længden ikke er langt fra den rute som *Clark & Wright* finder. Hvor *Clarke & Wright* gennemsnitligt udregner kortere rute på datasæt B (se tabel 3). Den procentvise afvigelse for *K-means* og *Clarke & Wright* i forhold til deres gennemsnitlige længde for datasæt A og B er ret lille. Køretiden er også en faktor man skal tage højde for. Køretiden af datasæt A($n80 - k10$) med *Clarke & Wright* er 20,97 ms, mens *K-means* med 2-opt har en køretid på 3093,3 ms, på 50000 gennemkørsler. Det er vigtigt at huske, at man ikke skal regne med *K-means* med 2-opt finder en rute der er noget værd, hvis man kun laver én gennemkørsel. Man skal i stedet have mange gennemkørsler, måske endda 50000 eller mere. *K-means* køretid på 3093,3 ms er en del højere end *Clarke & Wright* køretid. *Clarke & Wright* og *K-means* med 2-opt skaber hver især forskellige mængder af ruter. Som eksempel, i datasæt B ($n51 - k7$), så laver *Clarke & Wright* en *CVRP* med 9 ruter, mens *K-means* skaber en *CVRP* med 7 ruter. Antallet af ruter kan også ses som antallet af lastbiler/chauffører som *VRP* løsningen kræver. Dermed er *K-means* i dette tilfælde en bedre løsning for en virksomhed, da virksomheden sparer to lastbiler/chauffører, hvilket løber op i mange ressourcer hvis man hver dag kunne sparer dette i f.eks et år.

14 Ordliste

I **ikke** alfabetisk rækkefølge

VRP	Vehicle Routing Problem
VRPTW	Vehicle Routing Problem Time Windows
CVRP	Capacitated Vehicle Routing Problem
TSP	Travelling Salesman Problem
n	$\{1, 2, 3, ..n\}$
CW	Clarke and Wright
K-means	Clustering Algoritme
SA	Simulated Annealing
Kompleksitet	Iterationer der kræves for en fuldendt løsning
K-værdi	Antallet af Clusters

15 PC Specifikationer

RAM	CPU	STYRESYSTEM
2x8GB 2400MHz	i7-8700K CPU @ 3.70GHz	Windows 10 64bit

Tabel 4: Systemsspecifikationerne som er beskrevet i tabellen, er fabriksbestemte og ikke de aktuelle clock-hastigheder opnået under forsøgene.

Litteratur

[1] Capacitated vrp instances. `http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-instances/`. Sidst besøgt 25/05-2020.

[2] *Handbook of Theoretical Computer Science : Volume A: Algorithms and Complexity.* Elsevier Science Publishers, Amsterdam.

[3] Known best results vehicle routing problem. `http://neo.lcc.uma.es/vrp/known-best-results/`. Optimale længder for CVRP instanser, sidst besøgt 25-05-2020.

[4] Tola Alade. How to determine the optimal number of clusters for k-means clustering, 2018. [Brugt billede : Online; Besøgt 01/05, 2020, https://blog.cambridgespark.com/how-to-determine-the-optimal-number-of-clusters-for-k-means-clustering-14f27070048f].

[5] N. Alon and J.H. Spencer. Wiley-interscience series in discrete mathematics and optimization. pages 353–354, 01 2008.

[6] Naddef D Belenguer J M Benavent E Corberan A Augerat, P and G Rinaldi. Computational results with a branch and cut code for the capacitated vehicle routing problem, Sep 1995.

[7] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, page 4, USA, 2009. USENIX Association.

[8] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[10] Milan Djordjevic, Milan Tuba, and Bojan Djordjevic. Impact of grafting a 2-opt algorithm based local searcher into the genetic algorithm. pages 485–490, 08 2009.

[11] Robert Glove. Using the elbow method to determine the optimal number of clusters for k-means clustering, 2017. [Brugt billede : Online; Besøgt 19/05 2020, https://bl.ocks.org/rpgove/0060ff3b656618e9136b].

[12] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–868, October 1989.

[13] Steven M. LaValle. *Planning Algorithms.* Cambridge University Press, USA, 2006.

[14] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Oper. Res.*, 21:489–516, 01 1973.

[15] C.-Y Liong, I. Wan, and Khairuddin Omar. Vehicle routing problem: Models and solutions. *Journal of Quality Measurement and Analysis*, 4:205–218, 01 2008.

[16] Lithmee. What is the difference between directed and undirected graph. https://pediaa.com/what-is-the-difference-between-directed-and-undirected-graph/#Directed%20Graph.

[17] Stefan Røpke. *Heuristic and exact algorithms for vehicle routing problems*. PhD thesis, 2006.

[18] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. Constrained k-means clustering with background knowledge. In Carla E. Brodley and Andrea Pohoreckyj Danyluk, editors, *ICML*, pages 577–584. Morgan Kaufmann, 2001.

[19] Wikipedia, the free encyclopedia. Big o notation,, 2020. [Brugt billede : Online; Besøgt 08/05, 2020, https://commons.wikimedia.org/wiki/File:Comparison_computational_complexity.svg#filelinks],.

[20] Wikipedia, the free encyclopedia. Seven bridges of königsberg, 2020. [Brugt billede : Online; Besøgt 10/05, 2020, https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg].

[21] Robin J Wilson. *Introduction to Graph Theory*. John Wiley, Sons, Inc., USA, 1986.