Bio-Inspired AI (Project 1 Report) *by Stian Tornholm Grimsgaard (stiantg@stud.ntnu.no)*

1. Chromosome representation

I min antakelse av denne oppgaven, så tilsvarer et kromosom en fullstendig løsning av problemet. Jeg antar også at et kromosom består av flere gener, der ètt gen er et kjøretøy.

I dette prosjektet har jeg valgt å representere et kromosom som en egen klasse (Chromosome.java). Denne klassen består flere kjøretøy (Vehicle). Hvert kjøretøy blir lagt i en ArrayList, da dette er en passende datatype.

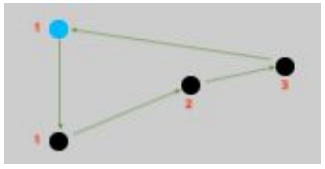
```
public class Vehicle {

    private Depot startDepot;
    private Depot endDepot;
    private ArrayList<Customer> customers;
```

Hvert kjøretøy består hovedsakelig av et start-depot og et end-depot. Det består også av en liste av kunder knyttet til kjøretøyet. Her er også en ArrayList blitt tatt i bruk.

Fordelen med denne representasjon er at den er enkel å arbeide med og forholde seg til. Man kan enkelt få tilgang til all ønskelig informasjon ved å kalle på funksjoner knyttet til objektene. Dette er også den "riktige" måten å jobbe på i Java, da språket er et objektorientert språk.

Ulempen med denne representasjonen er at det hele tiden vil være mange objekter i omløp. Dette kan føre til treghet i systemet, og i verste fall et fullt heap. Jeg har også støtt på situasjoner der jeg må utføre såkalt "deep copying", som vil si å utføre en fullstendig duplikasjon. Dette hadde vært enklere med representasjonen som nevnes neste.

En annen mulig representasjon for denne oppgaven hadde vært å bruke en ArrayList med Integers. Det første og siste tallet i listen hadde representert Id-en til start- og slutt-depot, mens alt i mellom hadde representert Id-ene til kundene.



I bildet til venstre har både start- og end-depot Id = 1. Kunden har Id = 1, 2 og 3. En representasjon av dette kjøretøyet kunne dermed vært: **[1, 1, 2, 3, 1]**

Fordelen med denne representasjonen er at den trolig er plassbesparende, i forhold til å opprette mange objekter. Den kan også ha en fordel i utførelse av operasjoner som mutation og crossover. En mutation operasjon kunne for eksempel bare vært å valgt en tilfeldig depot/kunde og byttet siffer til en verdi som ikke oversteg den høyeste Id-en i problemet.

Ulempen med denne løsningen er at den etter min preferanse, er vanskeligere å forholde seg til når man koder. Hver gang man skal hente en kunde eller et depot, så må man utføre en form for "mapping", der man henter tilleggsinformasjon (som koordinater), fra en Id. Her kunne man for eksempel hatt en database som representerte et kunde/depot objekt.

Jeg valgte å gå for førstnevnte representasjon, da den var mest intuitiv og enklest å jobbe med, i tillegg til at det er en naturlig fremgangsmåte i språket Java. Hvis ytelse spilte en viktig rolle, kunne det derimot vært interessant å teste den andre representasjonen, for å se om den gir bedre ytelse.

2. Crossover & Mutation operators

Crossover operasjonen jeg har valgt er en variant av Best Cost Route Crossover (BCRC), der jeg lager to barn av to foreldre. Dette er en optimalisert crossover-metode, som henter kunder fra hver forelder og legger kundene på den best mulige plassen i henhold til distanse, på den motsatte forelder.

En kunde vil aldri bli innsatt på et kjøretøy, hvis det bryter constraints. Dette er håndtert ved at den underveis kalkulerer distanse og load ved innsetting av ny kunde, og fortsetter til neste kjøretøy hvis constraints brytes. Dermed vil crossover-metoden ikke produsere ugyldige barn. Hvis den derimot skulle gjøre det (noe som foreløpig ikke har skjedd), vil den returnere foreldrene som barn. Det vil dermed ikke skje en crossover for de angitte foreldrene. Hvis dette hadde blitt tilfellet, kunne det blitt håndtert ved å for eksempel prøve å utføre crossover med andre foreldre.

Jeg har valgt å bruke 4 ulike **mutasjoner**, der èn av de utføres ved en gitt tilfeldighet:

1) Swap-mutation: Her swappes to kunder fra to ulike kjøretøy.
2) Inverse-mutation: Her inverteres en sekvens av kunder på et tilfeldig kjøretøy.
3) Rearrange-mutation: Her flyttes alle kunder over på andre kjøretøy, hvis det gir en kortere distanse.
4) Rerouting-mutation: Her flyttes kunder fra ètt tilfeldig kjøretøy over på andre ruter.

I alle mutasjonen er det lagt inn restriksjoner som gjør at en mutasjon vil reverseres etter et gitt antall forsøk, hvis den bryter constraints.

3. Parameter relationships

Jeg ønsker å se på forholdet mellom population size og mutation rate. Data er hentet fra forsøk på P01, med en helt tilfeldig initiell populasjon etter 1000 generasjoner.

Population Size	Mutation Rate	Best fitness
1000	0	612.683
1000	0.05	613.401
100	0.05	643.895
100	0	710.839
10	0.05	735.223
10	0	778.439

Tabellen viser at man som regel får et bedre resultat, hvis man inkluderer en mulighet for mutasjon. Dette var ikke tilfellet i første kjøring, men her vil jeg argumentere for at en populasjon på 1000 gir en såpass høy grad av mangfold at mutasjon ikke vil spille en like stor rolle.

Når man derimot kommer ned på små populasjoner, spiller mutasjon en større rolle. Dette er fordi de mindre populasjonene har et mye mindre mangfold, og vil dermed ha et større krav til å oppnå mutasjoner underveis for å unngå lokale maksimum.

Population Size:
Tidlig fase: Størrelsen på populasjonen vil ha en størst betydning i den tidlige fasen av algoritmen. Dette er fordi en stor populasjon vil gi mye tilfeldighet, som tidlig kan lede algoritmen inn på riktig spor mot en optimal løsning. En liten populasjon kan lede algoritmen inn i lokale maksimum.
Senere fase: I de senere fasene vil også en stor populasjon ofte gi bedre resultater, da crossover og mutation vil oppstå på flere individer, men en stor populasjon krever mer beregningskraft, da det er flere individer å utføre operasjoner på. Dermed må man ofte begrense seg, hvis man ikke har god tid.

Mutation Rate:
Tidlig fase: I en tidlig fase vil mutasjonsraten ha relativt liten innvirkning, da populasjon allerede har mye tilfeldighet. I verste fall kan en høy mutasjonrate gi algoritmen for mye støy, som kan hindre fremgang.
Senere fase: I de senere fasene konvergerer algoritmen, da den har for lite tilfeldighet til å oppnå nye løsninger. Her vil mutasjonen spille en viktig rolle, da den bidrar til å opprettholde tilfeldighet i algoritmen og pushe algoritmen mot bedre løsninger. Hvis man muliggjør “parameter control”, kan det dermed være fordelaktig å øke mutasjonsraten mot slutten av en kjøring.