WISEflow
Europe/Oslo(CEST)
18 Dec 2022

Forsvaret

Bacheloroppgave

OPG3301

Predefinert informasjon

Startdato:	05-12-2022 09:00 CET	**Termin:**	2022 HØST
Sluttdato:	19-12-2022 08:00 CET	**Vurderingsform:**	Norsk 6-trinns skala (A-F)
Eksamensform:	Oppgave		
Flowkode:	1627 OPG3301 1 O 2022 HØST SKSK		
Intern sensor:	Alexander Sauter		
Intern sensor:	Christophe Massacand		

Deltaker

Navn:	Jone Måsøy
Kandidatnr.:	
FHS-id:	jmasoy@mil.no

Gruppe

Gruppenavn:	JEA-Navigation
Gruppenummer:	3
Andre medlemmer i gruppen:	André Killingmoe, Eirik Kiland

Sjøkrigsskolen

Bacheloroppgave

Hvordan kan maskinlæring brukes som hjelpemiddel til ruteplanlegging i Sjøforsvaret?

av

Jone Måsøy, Herman André Lindstad Killingmoe og Eirik Kiland

Levert som en del av kravet til graden:

Militære studier med fordypning i ledelse -

Marineingeniør våpensystemer, elektronikk og data

Innlevert: desember 2022

Godkjent for offentlig publisering

(Blank side)

I. Publiseringsavtale

En avtale om elektronisk publisering av bachelor/prosjektoppgave

Kadettene har opphavsrett til oppgaven, inkludert rettighetene til å publisere den.

Alle oppgaver som oppfyller kravene til publisering, vil bli registrert og publisert i Bibsys Brage når kadettene har godkjent publisering.

Oppgaver som er graderte eller begrenset av en inngått avtale, vil ikke bli publisert.

Vi gir herved Sjøkrigsskolen rett til å gjøre denne oppgaven tilgjengelig elektronisk, gratis og uten kostnader.	X Ja	Nei
Finnes det en avtale om forsinket eller kun intern publisering? (Utfyllende opplysninger må fylles ut)	Ja	X Nei
Hvis ja: kan oppgaven publiseres elektronisk når embargoperioden utløper?	Ja	Nei

II. Plagiaterklæring

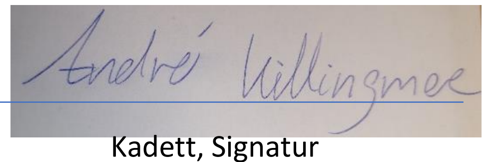
Vi erklærer herved at oppgaven er vårt eget arbeid og med bruk av riktig kildehenvisning. Vi har ikke nyttet annen hjelp enn det som er beskrevet i oppgaven. Vi er klar over at brudd på dette vil føre til avvisning av oppgaven.

Dato: 14.12.2022

Jone Måsøy

Kadett, Navn

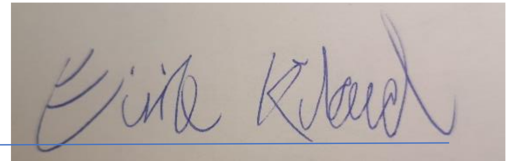
Kadett, Signatur



Herman André Lindstad Killingmoe

Kadett, Navn

Kadett, Signatur



Eirik Kiland

Kadett, Navn

Kadett, Signatur

III. Forord

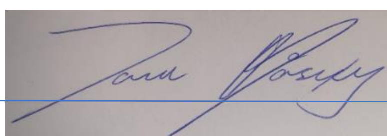
Denne bacheloroppgaven er skrevet av Eirik Kiland, André Killingmoe og Jone Måsøy i perioden juni 2022 til desember 2022. Den er skrevet som en del av kravet til studiet: Militære studier med fordypning i ledelse - Marineingeniør våpensystemer, elektronikk og data.

Oppgaven tar for seg maskinlæring og utforsker hvordan man kan bruke dette til å planlegge og lage maritime seilingsruter. Oppgavens tema er valgt basert på forfatternes interesse for fagfeltet og nysgjerrighet knyttet til mulighetsområdet for maskinlæring. Det legges stor vekt på teorien bak maskinlæring, og hvordan dette kan anvendes hos navigatørene i Sjøforsvaret i dag. En større grad av tilgjengelig datakraft har gjort at maskinlæring kan løse flere oppgaver som mennesket bruker betydelig tid på å løse. Det er en av disse oppgavene vi ser nærmere på i dette studiet.

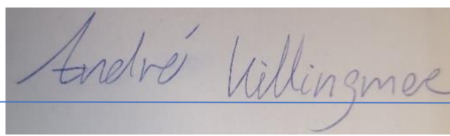
Vi vil rette en stor takk til Alexander Sauter og Christophe Massacand for god veiledning og hjelpsomme diskusjoner gjennom hele prosessen. Takk til Petter Lunde ved Navigasjonskompetansesenteret for gode tips angående det navigasjonstekniske. Vi sender også en takk til våre medkadetter for å bidra til et godt og sterkt klassemiljø gjennom årene på Sjøkrigsskolen.

Bergen, Sjøkrigsskolen, 13.12.2022.

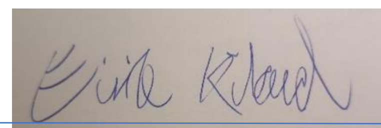
Jone Måsøy



André Killingmoe



Eirik Kiland



IV. Sammendrag

Maskinlæring er en type kunstig intelligens som gjør det mulig for en datamaskin å lære og gjøre forutsigelser basert på data, uten å bli eksplisitt programmert. Fremgangsmåten er å gi maskinen en stor mengde data, for deretter å la maskinen selv finne mønstre og lage regler for hvordan den skal løse oppgaven. På denne måten kan maskinen lære å gjøre ting som å gjenkjenne bilder, forutsi fremtidige hendelser eller løse komplekse problemer. Samtidig kan maskinlæring også medføre en rekke utfordringer, særlig innen personvern og etikk. Mange er også bekymret for at maskiner kan bli så avanserte at de overgår menneskets evne til å forstå eller kontrollere dem. Det er derfor viktig å ha kunnskap om hvordan maskinlæring fungerer, for å bruke det på en bærekraftig og ansvarsfull måte.

Denne oppgaven bruker reinforcement learning, som er en type maskinlæring der en datamaskin lærer gjennom å prøve og feile. Dette gjøres ved at maskinen får en belønning eller straff for hvert valg den tar. Den justerer seg deretter for å gjøre det valget som gir størst gevinst. På denne måten kan maskinen lære å ta optimale valg i et gitt miljø.

Reinforcement Learning kan brukes innen flere ulike områder, som for eksempel robotikk, spill og finansielle prognoser. Det er mange forskjellige algoritmer for reinforcement learning, inkludert Q-learning og Monte Carlo Tree Search. Disse algoritmene kan brukes til å løse komplekse problemer som å finne den beste strategien i et spill, eller å lære og bevege seg rundt i et miljø. Maskinlæringsalgoritmen i denne oppgaven gjør nettopp dette. Programmet konstruerer en labyrint fra et dybdekart som maskinlæringsalgoritmen skal bevege seg i. Gjennom å prøve og feile skal algoritmen lære seg hvordan labyrinten (sjøkartet) ser ut og deretter finne raskeste vei fra A til B.

Algoritmen kan kjøres direkte i programmeringsspråket Python. For å bruke algoritmen har det blitt lagd et grafisk brukergrensesnitt for å gjøre det hele mer brukervennlig. Med brukergrensesnittet kan man enkelt plotte inn start- og sluttpunkt i et kart og bekrefte det med en knapp. Dette gjør gradene til punktene om til koordinater som algoritmen leser av. Knappen starter også maskinlæringsalgoritmen som begynner å kartlegge området rundt start og slutt. Den vil etter en viss tid finne raskeste rute fra start til slutt med god sikkerhetsavstand til land. Deretter plotter den til slutt ruten i brukergrensesnittet.

For at maskinlæringsalgoritmen skal kunne anvendes i Sjøforsvaret, mangler det noen funksjoner før programmet er en komplett løsning. Per nå blir det ikke presentert noen data i brukergrensesnittet som kan brukes til å navigere, annet enn distansen og den visuelle ruten. Med implementering av

GPS, AIS og kursretning for de ulike strekningene vil programmet ikke bare være et godt planleggingsverktøy, men også et godt verktøy for navigatørene under transitt. Derfor er dette et konseptbevis på at maskinlæring kan nyttes til ruteplanlegging i Sjøforsvaret.

V. Oppgaveformulering

«Ved å ta i bruk teknologi på en smart måte vil vi kunne løse oppgavene mer effektivt og med større kraft, alene og sammen med allierte og partnere.»

Teknologien Forsvaret trenger (ffi.no)

Forsvaret forskningsinstitutt (FFI) ble opprettet av Stortinget i 1946, som en erkjennelse av hvor viktig den teknologiske utviklingen er for Forsvaret.

Vi har valgt å bidra til denne utviklingen ved å lage en ruteplanlegger. For å lage dette verktøyet har vi valgt å bruke maskinlæring. Hovedsakelig vil oppgaven basere seg på én spesifikk gren innenfor maskinlæring, reinforcement learning. Således er problemstillingen formulert som:

Hvordan kan maskinlæring brukes som hjelpemiddel til ruteplanlegging i Sjøforsvaret?

Dermed skal denne oppgaven undersøke hvordan maskinlæring kan brukes som hjelpemiddel til å planlegge ruter i det maritime miljøet.

VI. Innholdsfortegnelse

Innhold

I. Publiseringsavtale 2

II. Plagiaterklæring 3

III. Forord 4

IV. Sammendrag 5

V. Oppgaveformulering 6

VI. Innholdsfortegnelse 7

VII. Figurliste 10

VIII. Formelliste 11

Oversikter 12

Nomenklatur 12

Forkortelser 15

Matematisk notasjon 15

1. Introduksjon 17

1.1 Bakgrunn 17

1.2 Konseptuell idé 18

1.3 Mål 18

1.4 Begrensninger 19

1.5 Metode 19

1.6 Struktur 20

2. Teori 21

2.1 Maskinlæring 21

2.2 Reinforcement Learning 22

2.2.1 Introduksjon 22

2.2.2 Begreper 22

2.2.3 Exploitation versus exploration dilemma 23

2.2.4 Prediksjons- og kontrollproblemet 25
2.3 Vår metode for løsning av RL-problemet 26
2.3.1 Agenten i miljøet 26
2.3.2 Mål, gevinst og utbytte 28
2.3.3 Strategi og verdifunksjoner 29
2.3.4 Bellmanligningen 30
2.3.5 Optimal strategi 31
2.3.6 Oppsummert 32
2.4 Q-learning 33
2.5 Brukergrensesnitt 35
3. Utvikling av konseptet 36
3.1 Konseptuelle krav 36
3.2 Miljø 37
3.3 Graphical User Interface 40
4. Tester og resultater 44
4.1 Datafremstilling 44
4.1.1 Matriserepresentasjon 44
4.1.2 Fast dybderepresentasjon 45
4.1.3 Et labyrintkart av piksler 46
4.1.4 Gevinst-kart 47
4.1.5 Oppløsning og tidsbruk 48
4.2 Trening og implementering av RL-algoritmen 50
4.2.1 Rute mk1 51
4.2.2 Rute mk2 52
4.2.3 Rute mk3 53
4.3 Brukergrensesnitt 56
4.4 Oppsummering av resultater 58
5 Drøfting 59

5.1 Under utvikling 59
5.1.1 Datafremstillingen 59
5.1.2 RL-algoritmen 61
5.1.3 Graphical User Interface 64
5.2 Konseptuell idé 64
5.2.1 Sjøforsvarets nytteverdi 64
5.2.2 Live-oppdatering 65
5.3 Begrensninger ved maskinlæring 66
6 Konklusjon og forslag til videre arbeid 67
6.1 Konklusjon 67
6.2 Forslag til påbygning og videre arbeid 69
Bibliografi 72
Vedlegg A – Førsteutkast CostMap.py 74
Vedlegg B – Andreutkast CostMap.py 74
Vedlegg C – Brukermain.py 75
Vedlegg D – Main.py 80
Vedlegg E – NewLabyrinth.py 86
Vedlegg F – Sauterlek.py. 90
Vedlegg G – VBR.py 94

VII. Figurliste

Figur 1 Evaluation & Improvement 24
Figur 2 Epsilon Greedy Action 24
Figur 3 Prosessen i generalized policy iteration 25
Figur 4 Hele beslutningsprosessen i en MDP.. 26
Figur 5 En ordinær Markovkjede 27
Figur 6 Grafisk fremstilling av verdifunksjonene. 30
Figur 7 Q-learning sin posisjon i reinforcement learning-verden. 33
Figur 8 Kartutsnittet til miljøet. 37
Figur 9 Kartutsnittet som et rutekart. 37
Figur 10 Oppstartsskjermen til brukergrensesnittet. 41
Figur 11 Brukergrensesnittet med sjøkart-filter 42
Figur 12 Brukergrensesnittet i bruk 43
Figur 13 Kartutsnittet for dataene. .... **Feil! Bokmerke er ikke definert.**
Figur 14 Dataen representert i et 3D-plot 45
Figur 15 Scatter-plot 46
Figur 16 Kostnadskart 46
Figur 17 Gevinstkart 90x90 piksler. 47
Figur 18 Gevinstkart 150x150 piksler. 48
Figur 19 Gevinstkart 60x60 piksler. 49
Figur 20 Total gevinst per episode.. 49
Figur 21 Rute mk1. 51
Figur 22 Rute mk2 52
Figur 23 Rute mk3 53
Figur 24 Dynamisk dybde eks. 1 54
Figur 25 Dynamisk dybde eks. 2 55
Figur 26 Eksempel på første rute 56
Figur 27 Ruteplanleggeren med glidebryterfunksjonen. 57
Figur 28 Det endelige brukergrensesnittet 57

VIII. Formelliste

Formel 1 Overgangssannsynligheten i en MDP 27
Formel 2 Utbytte 28
Formel 3 Diskontert utbytte 29
Formel 4 Diskontert utbytte, rekursivt skrevet 29
Formel 5 Tilstands-verdifunksjonen 29
Formel 6 Handlings-verdifunksjonen 30
Formel 7 Bellmanligningen for tilstands-verdifunksjonen 31
Formel 8 Bellmanligningen for handlings-verdifunksjonen 31
Formel 9 Optimal tilstands-veridfunksjon 31
Formel 10 Optimal handlings-verdifunksjon 31
Formel 11 Tilstands-verdifunksjonen skrevet fra handlings-verdifunksjonen 32
Formel 12 Q-funksjonen 34
Formel 13 x-koordinatens plassering 37
Formel 14 y-koordinatens plassering 37

Oversikter

Nomenklatur

Her beskrives ord som gjennom oppgaven er oppført i kursiv tekst. Ordene er sortert etter temaene Markovbeslutningsprosess, maskinlæring og øvrig.

Ord	*Forklaring*
Tema:	*Markov beslutningsprosess*
Absorberende tilstander	En absorberende tilstand er en tilstand det ikke er mulig å forlate, og som avslutter beslutningsprosessen. Et eksempel på dette er når en sjakkspiller setter motstanderen i sjakk matt. Dette vil avslutte spillet og det er ikke mulig å gjøre flere trekk.
Agent	Den eller det som skal ta en beslutning.
Beslutningsprosess	Alt samspill mellom beslutningstaker og miljø.
Beslutningstaker	Samme som *agent.*
Diskontering	Å omregne en fremtidig verdi til nåverdi.
Diskret	Strukturen støtter ikke eller behøver ikke kontinuitet.
Ekvivalensklasser	En ekvivalensklasse er en klasse av tilstander som kan nås av hverandre.
Episode	Hele beslutningsprosessen fra start til den absorberende tilstanden.
Ergodisk	En ikke-reduserbar aperiodisk og tidshomogen markovkjede med et endelig antall tilstander N.
Gevinst	Resultatet til en beslutning og steg gjort av agenten.
Handling	En beslutning og steg gjort av agenten.
Ikke-reduserbar	For at en Markovkjede skal være ikke-reduserbar, må enhver tilstand kunne nås fra en hvilken som helst annen tilstand. Det må altså ikke være flere *ekvivalensklasser* i overgangsstrukturen
Kostnad	En negativ gevinst.

Miljø	Verdenen agenten eksisterer i og samspiller med. I denne oppgaven rutekartet, som er basert på dybdekart.
Optimal strategi	Den mest gunstige strategien. Det vil si den som gir størst gevinst.
Handlings-verdifunksjon	Den forventede gevinsten hvis agenten gjør handlingen i henhold til strategien.
Tilstands-verdifunksjon	Den forventede gevinsten hvis agenten velger gitt tilstand i henhold til strategien
Off-Policy algoritme	Bruker forskjellige strategier for å handle (acting policy) og for å oppdatere (updating policy).
Overgangssannsynlighet	Sannsynligheten for å gå i tilstanden s' med gevinst r fra tilstand s med handling a.
Overgangsstruktur	Alle mulige overgangssannsynligheter i miljøet.
Periodisitet	Beskriver hvor mange overganger det minimum kreves før man er tilbake til tilstanden man var i ved utgangspunktet. Skal det være en periodisitet må det være inne i én ekvivalensklasse
Rekurrente tilstander	En tilstand som prosessen med all sannsynlighet vil vende tilbake til.
RL-problem	Beskriver problemer som kan deles inn i diskrete tilstander.
Steg	Overgangen fra en gitt tilstand til en annen gitt tilstand med tilhørende handling og gevinst.
Strategi	«Planen» til agenten for å oppnå høyest mulig gevinst.
Tilstand	Situasjonen i miljøet til et gitt tidspunkt (steg).
Transiente tilstander	Det motsatte av en rekurrent tilstand, altså en tilstand hvor det er usikkert at en kommer tilbake til.
Utbytte	Den samlede kumulative gevinsten fra et gitt steg til og med absorpsjonssteget T.
Verdi	Det forventede utbyttet fra en gitt tilstand S_t ved et gitt steg. (Eventuelt også med handling a)
Verdifunksjon	Funksjonen som regner ut verdien fra en gitt tilstand s. (Eventuelt også med handling a)

Tema:	*Maskinlæring*
Hyperparametere	Konstanter som settes før treningen begynner.
K-means clustering	En unsupervised learning-algoritme som brukes for å løse «clustering»-problemer
Q-learning	Metoden innenfor Reinforcement Learning som denne oppgaven baserer seg på.
Reinforcement Learning	Erfaringsbasert maskinlæringsmetode som baserer seg på prøving og feiling.
Temporal-Difference Learning	Reinforcement Learning-metode som lærer hvordan å forutse basert på fremtidige verdier.
Trening	Kjøring av maskinlæringsalgoritmen.

Tema:	*Øvrig*
Autonomi	Egenskap som kjennetegner egne indre avgjørelser uavhengig av påvirkning utenfra.
Fysisk miljø	Det faktiske og virkelige miljøet som det virtuelle miljøet er basert på. For eksempel Sotraleden.
Iterasjon	En utregning.
Virtuelt miljø	Miljøet konstruert i koden basert på dybdekart.
Easting	Gridreferansesystem. Vertikale linjer som går fra topp til bunn på et kart. Deler opp kartet fra vest til øst. Oppgaven bruker det offisielle geodetiske datumet i Norge, EUREF89.
Northing	Gridreferansesystem. Horisontale linjer som går fra venstre til høyre på et kart. Deler opp kartet fra nord til sør. Oppgaven bruker det offisielle geodetiske datumet i Norge, EUREF89.

Forkortelser

Forkortelse	*Betydning*
GPI	Generalisert strategi-iterasjon. *(Generalised policy iteration)*
GUI	Grafisk brukergrensesnitt. *(Graphical User Interface)*
HMI	Brukergrensesnitt. *(Human Machine Interface)*
RL	Reinforcement Learning.
TD	Temporal-Difference Learning.

Matematisk notasjon

Notasjon	*Forklaring*
$P(A\|B)$	Betinget sannsynlighet. Sannsynligheten for A gitt B.
$P(X = x)$	Sannsynligheten for at variabel X har verdi x.
$\max_a f(a)$	Verdien for $f(a)$ der $f(a)$ har sin høyeste verdi med hensyn på a.
ε	Sannsynligheten for å handle tilfeldig. En handling uavhengig av strategien.
α	Hyperparameter brukt i TD- og Q-learning.
γ	Hyperparameter for diskonteringsrate.
s, s'	Tilstander
a	En handling.
r	En gevinst.
t	Et steg.
S_t	En tilstand ved det gitte steget t.
A_t	An handling ved det gitte steget t.
R_t	En gevinst ved det gitte steget t.

π	Strategien
$\pi(s)$	Strategien fra den gitte tilstanden s.
$\pi(a\|s)$	Sannsynligheten for å velge handling a gitt tilstand s i strategi π.
$v_\pi(s)$	Tilstands-verdifunksjonen under strategien π.
$q_\pi(s,a)$	Handling-verdifunksjon under strategien π.
$v_*(s)$	Optimal tilstands-verdifunksjon.
$q_*(s,a)$	Optimal handlings-verdifunksjon.
$V(s)$	Estimat for tilstands-verdifunksjon.
$Q(s,a)$	Estimat for handlings-verdifunksjon.

1. Introduksjon

1.1 Bakgrunn

Maskinlæringens historie starter i 1943, med den første matematiske modellen for et nevralt nettverk. (Norman, 2022) Syv år senere konstruerte Alan Turing en test kjent som *The Turing Test*. For å bestemme om en maskin har ekte intelligens, må maskinen kunne lure et menneske til å tro at maskinen også er et menneske.

Maskinlæring slik det i dag er kjent, ble først igangsatt i et eksperiment som het «Preceptron» i 1957. Her utviklet Frank Rosenblatt en elektrisk komponent bygget opp i samsvar med biologiske prinsipper, som viste en evne til å lære. Dette beviste at datamaskiner kan få en evne til å lære av egne erfaringer og løse problemer, i likhet med biologiske skapninger. Eksperimentet startet historien og utviklingen innen maskinlæring (Sannsyn, 2020).

Selv om det første maskinlærings-eksperimentet ble påbegynt i 1957, er anvendelsen og bruksområdene fortsatt regnet som ny kunnskap. Maskinlæring blir i dag benyttet i selvkjørende biler, sjakkalgoritmer, forskning og statistikkalgoritmer. Forsvaret har også benyttet maskinlæring ved tidligere forskning. For eksempel har dette blitt brukt i en oppgave ved Sjøkrigsskolen, der man så på hvordan maskinlæring kan benyttes for å klassifisere objekter i forbindelse med navigasjon (Mathiesen & Lowzow, 2017). Maskinlæring inngår også i en annen oppgave: Maskinlæring i praktisk beslutningstaking (Strandvold & Leines, 2018).

Det finnes mange forskjellige typer maskinlæring. De fire hovedkategoriene er supervised learning, unsupervised learning, semi-supervised learning og *reinforcement learning*. Reinforcement learning kan defineres som (her er det valgt å bruke en engelsk definisjon):

“Reinforcement Learning (RL) is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences” (Bhatt, 2018).

Denne oppgaven skal se nærmere på bruken av maskinlæring og reinforcement learning, for å hjelpe Sjøforsvarets navigatører med å planlegge seilingsruter. I dag bruker navigatørene betydelig tid på å planlegge ruten de skal seile, før de har påbegynt seilingen. Oppgaven skal dermed undersøke om maskinlæring og reinforcement learning kan løse problemstillingen: **Hvordan kan maskinlæring brukes som hjelpemiddel til ruteplanlegging i Sjøforsvaret?**

1.2 Konseptuell idé

Det er viktig for oppgavens helhet å gå inn på bakgrunnen til konseptet. Det sier mye om hvordan oppgaven kom ble løst, samt behovet for akkurat en slik løsning.

Det startet med at vi så hvor mye tid navigasjons-kadettene brukte på å planlegge rutene sine før de skulle seile. Dette var som oftest ruter som de aldri hadde seilt før, og som de skulle vurderes på gjennom seilasen. Maskinlæring ble vurdert til å ha et potensiale for å løse navigatørenes strevsomheter. Å gi navigasjonsdetaljen et verktøy som kan spare Sjøforsvaret utallige timer med planlegging, vil gi økt operativ effekt på alle Sjøforsvarets fartøy. Verktøyet kan også anvendes på forskjellige bruksområder i Sjøforsvaret, som for eksempel SAR-operasjoner og mineryddingsoperasjoner. Altså alle andre operasjoner som ikke kun handler om å komme seg kjappest mulig fra A til B. Dette utforsker oppgaven opp mot algoritmen, og presenterer flere forslag til hvordan dette kan jobbes med videre. Anbefalt videre arbeid kommer i delkapittel 6.2.

En *autonom* maritim ruteplanlegger er en type teknologi som bruker kunstig intelligens for å planlegge ruter for autonomt maritimt utstyr, for eksempel selvstyrte båter. Vår konseptuelle vinkling er fokusert mot å lage en ruteplanlegger som kan brukes som hjelpemiddel for navigatørene i Sjøforsvaret. Ideen bak denne teknologien er å gi autonomt maritimt utstyr evnen til å planlegge en optimal rute i et gitt havområde uten menneskelig inngripen. Dette kan gi forslag til ruter som navigatørene kan bruke som utgangspunkt for videre planlegging. En optimal ruteplanlegger bruker en kombinasjon av kartdata og maskinlæring for å lage en plan som tar hensyn til faktorer som dybde, strøm og gjeldende sjøregler. Målet er å hjelpe navigatørene til å navigere sikkert og effektivt i ulike havomgivelser.

1.3 Mål

Oppgavens mål er å utvikle et beslutningssystem som best mulig tar hensyn til to ting; effektiv og sikker navigering. For at ruteplanleggeren skal fungere, tas det utgangspunkt i at man har tilgjengelig kartdata for området man skal seile i. Området er kartlagt og man vet hvor man skal og hvor man starter. Dette er helt grunnleggende for at reinforcement learning skal kunne anvendes i ruteplanlegging. Hvordan tar man så avgjørelser på hvor man skal seile? Oppgaven vår bruker en maskinlæringsalgoritme som fungerer som *beslutningstaker* i *miljøet* som blir skapt med kartdataene tilgjengelig. Beslutningssystemet skal ta selvstendige avgjørelser i ruteplanleggingen for å oppnå disse to tingene på et tilfredsstillende nivå. Ut fra utviklingen av konseptet og testingen underveis, skal det også drøftes hvor stor nytteverdi dette kan ha for Sjøforsvaret, samt hvordan dette kan implementeres.

1.4 Begrensninger

For å kunne fokusere mer på selve maskinlæringen og *beslutningsprosessen* begrenses antall parametere i miljøet til kun dybde. Den konseptuelle idéen benytter maskinlæringen som et hjelpemiddel for navigatørene. Den skal kjapt komme med forslag til ruter. Derfor ser oppgaven bort fra faktorer som navigasjonsregler, strøm, vær og vind.

1.5 Metode

Arbeidet starter med en presentasjon av teorien som er nødvendig for å forstå hvordan maskinlæring fungerer, herunder spesielt reinforcement learning. Denne typen maskinlæring var en måte for oss å muliggjøre selvstendig beslutningstaking gjennom en programvare, noe som er nødvendig for at ruteplanleggeren skal kunne ta selvstendige valg og planlegge ruten på egen hånd. Programmeringen skjer i programmeringsspråket Python. Python ble valgt fordi det er et språk som ofte blir brukt i maskinlæringsalgoritmer, og var det språket som ble benyttet i «Maskinlæring i praktisk beslutningstaking» av Strandvold og Leines. Mer om dette senere i kapittelet.

Datafremstillingen som er miljøet for beslutningstakere, er konstruert virtuelt i Python. Denne leser direkte inn kartdata for sjøområder med tilhørende dybdeverdier, som setter mye av premissene som maskinlæringsalgoritmen må forholde seg til. Datafremstillingen og hvordan verdiene i den er vektlagt, har blitt betraktelig utviklet sammen med problemstillingen. Oppgaven tar også for seg et fungerende brukergrensesnitt som er det eneste brukeren av programmet trenger å forholde seg til. Mer om dette videre i oppgaven.

I maskinlæring er det en gjenganger at optimalisering og justering av *hyperparametere* (konstantene som gjør at datamaskinen lærer) er en ekstremt tidkrevende prosess som inneholder en hel del omfattende testing. For å ikke bruke mye tid på dette, baseres en del av hyperparameterne på en tidligere bacheloroppgave ved Sjøkrigsskolen, «Maskinlæring i praktisk beslutningstaking» av Strandvold og Leines. Strandvold og Leines har basert sitt sett med hyperparametere på en kildekode fra Lazyprogrammer Inc. som således blir det originale utgangspunktet for programmets hyperparametere (Lazy Programmer INC, 2018).

1.6 Struktur

Denne oppgaven tar først et dypdykk i maskinlæringsteorien som danner grunnlaget til beslutningstakeren i ruteplanleggeren. Her vil reinforcement learning og tilhørende teori bli særlig prioritert. Det vil gi leseren bedre kunnskap om hvordan reinforcement learning fungerer, og betingelsene for at ruteplanleggeren skal ta optimale valg. Videre går oppgaven inn på utviklingen av konseptet, hvor den diskuterer bakgrunn, ulike krav og forklarer hvordan konseptet fungerer. Etter det, presenteres funn og bemerkninger gjort under tester, i kapittelet tester og resultater. Etter dette vil resultatene drøftes sammen med den konseptuelle idéen til oppgaven. Mot slutten følger en konklusjon hvor det også blir gitt anbefalinger til videre arbeid med problemstillingen.

2. Teori

Teorien i denne delen av oppgaven tar for seg hele spekteret av teori som blir brukt gjennom oppgaven. Først presenteres maskinlæring generelt, før det blir et dypdykk i reinforcement learning.

I teorien om maskinlæring brukes flere engelske fagbegreper for å beskrive fenomener som skjer i datamaskinen. For oppgavens del er det besluttet å beholde disse på engelsk, da de engelske begrepene er mer nøyaktige og det er de som brukes når man skal diskutere maskinlæring internasjonalt. Det anbefales å sjekke nomenklaturen hvis det er begreper man trenger mer informasjon om.

2.1 Maskinlæring

Hva er maskinlæring?

Denne engelske definisjonen reflekterer godt de ulike aspektene med maskinlæring:

«The use and development of computer systems that are able to learn and adapt without following explicit instructions, by using algorithms and statistical models to analyse and draw inferences from patterns in data» (Selig, 2022).

Maskinlæring beskriver det som foregår fra kjør trykkes i et maskinlæringsprogram, til resultatet kommer ut. Som nevnt i kap. 1.1 finnes det flere forskjellige maskinlæringsmetoder; supervised, unsupervised, semi-supervised og reinforcement learning.

Supervised learning er en metode som gir brukeren mer kontroll, som igjen fører til at det blir mindre feil i resultatet som maskinlæringsalgoritmen kommer frem til. Supervised learning-algoritmer bruker det som har blitt lært fra tidligere data på ny data, ved å bruke merkede eksempler. Systemet trenes opp og prøver å forutsi output-verdien ved en ny input-verdi. Fordelen med denne metoden er at man kan sammenligne resultatet algoritmen kom med, med det ønskede resultatet man vil ha. Eksempler på dette er klassifiseringsproblemer, der man forsøker å si om dataene gir 0 eller 1 som output. Dette kan eksempelvis være å diagnostisere om en person har kreft eller ikke kreft, basert på de tilgjengelige dataene.

Unsupervised learning er en algoritme som brukes når informasjonen som er tilgjengelig for å trene algoritmen ikke er klassifisert eller merket. Metoden prøver å finne en funksjon som kan beskrive en skjult struktur fra de umerkede dataene. Det som er spesielt med unsupervised learning, er at algoritmen aldri vil vite med sikkerhet hva riktig output skal være. Eksempler på metoden er K-means

clustering, som går ut på å klassifisere en mengde data innenfor riktig kategorier, noe som et menneske ville brukt lang tid på å få gjort (Myrianthous, 2022).

Semi-supervised learning er en miks av supervised og unsupervised learning. Det er nyttig når data i datasettet har forskjellig merking. Da må man bruke metoden for å klare å tolke og forstå alle dataene. Metoden er mye brukt til objektdeteksjon og bildeklassifisering. Bildegjenkjenning handler om å hente ut såkalte *information classes* fra et *multiband raster bilde* (Myrianthous, 2022).

Reinforcement learning er det området av maskinlæring programmet kommer til å benytte seg av i denne oppgaven. Reinforcement learning handler i hovedsak om å regne ut den mest optimale *handlingen* i en gitt *tilstand*. (Geeksforgeeks, 2022). Videre om dette i neste delkapittel.

2.2 Reinforcement Learning

2.2.1 Introduksjon

Idéen om læring gjennom interaksjon med miljøet, er gjerne det første man tenker på når man blir spurt om hvordan læring skjer. Når et barn leker, veiver med armene, ser seg rundt og ikke har en «lærer» eller voksen ved siden av seg, bruker det bare sansene sine for å skape et bilde om hvordan verden rundt seg fungerer. Sansene i dette eksempelet vil fungere som et sensorsystem. Får barnet holde på for seg selv lenge nok, vil det mest sannsynlig få en bedre forståelse av hvordan tingene rundt seg fungerer. Barnet vil kunne finne ut at man kan bruke en bøtte til å bære sand, men det vil få sin helt egen måte å gjøre det på. Barnet vil etablere sin egen forståelse av verden (Sutton & Barto, 2014, 2015).

I dette kapittelet går oppgaven mer i dybden i reinforcement learning. I reinforcement learning ser man på når datamaskiner skal lære ved hjelp av å interagere med miljøet rundt seg. Mer spesifikt, så er reinforcement learning mye mer fokusert på *gevinst*-rettet læring fra interaksjon enn maskinlæringstilnærminger, eksempelvis nevnt under delkapittel 2.1 maskinlæring.

Herfra og utover i bacheloren kommer reinforcement learning til å bli sitert som RL. Dette for å lettere snakke om og rundt fagbegrepet.

2.2.2 Begreper

Miljø, beslutningstaker, tilstand, handling og gevinst er sentrale begreper i RL. Gevinst blir tilbakemeldingen beslutningstakeren får av miljøet den lærer fra. Beslutningstakeren er den som skal læres opp til å ta beslutninger. En beslutningstaker vil være *agenten* i miljøet. Dette kan for eksempel

være et romskip i et spill, en sjakkspiller i et sjakkspill, eller en robot. *Miljøet* kan beskrives som det området beslutningstakeren får bevege seg i. Dette kan for eksempel være et sjakkbrett eller en labyrint. Med dette miljøet kommer det også regler som beslutningstakeren må følge. Handling er et *steg* som kan eller vil bli gjennomført. Det kan for eksempel være å flytte bonden på et sjakkbrett én eller to fram fra utgangsposisjonen. Handling kan også forklares som de ulike mulighetene du har. Tilstand er en situasjon i et bestemt miljø på et bestemt tidspunkt. Eksempler på dette vil være mulige hindringers posisjon i et kart, beslutningstakerens posisjon eller alle nåværende posisjoner for brikkene på et sjakkbrett (Strandvold & Leines, 2018).

Sutton og Barto beskriver i deres introduksjon til maskinlæring om fire sub-elementer i et RL-system. Et av sub-elementene, *gevinst*, ble dekket i forrige avsnitt. Det neste sub-elementet er summen av miljøet, handling og tilstand, som sammen utgjør *modellen*. De to andre viktige elementene er *strategi* og *verdifunksjon*. En strategi definerer beslutningstakerens oppførsel og handlemåte i et gitt scenario. Eksempler på hva en strategi kan være er en enkel funksjon eller en oppslagstabell, men det kan også være en komplisert søkeprosess. En strategi vil være kjernen hos en RL-beslutningstaker, da den alene bestemmer beslutningstakerens oppførsel.

I motsetning til gevinst, som indikerer at noe er bra i ett tilfelle, vil en verdifunksjon si noe om hva som er bra på sikt. Verdifunksjonen med fokus på en gitt tilstand, vil gi den totale mengden gevinst man kan forvente å få inn i fremtiden ved å starte i akkurat den tilstanden. For eksempel kan en tilstand ha en lav umiddelbar gevinst, men ettersom den ofte blir fulgt av tilstander som har høy umiddelbar gevinst, får tilstanden en høy *verdi*. Dette kan også stemme andre veien. Verdifunksjonen tar for seg et større bilde, og har ikke bare fokus på den umiddelbare gevinsten. Dette er et helt sentralt prinsipp i RL-prosessen, ettersom algoritmen gradvis blir bedre og bedre (Sutton & Barto, 2014, 2015, s. 21).

2.2.3 Exploitation versus exploration dilemma

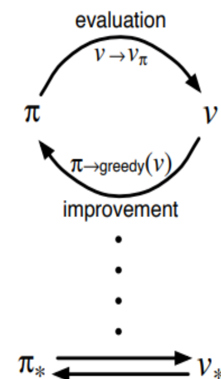
Etter hvert som RL har blitt bedre forstått, er det tydelig at et nøkkelkonsept i alle RL-algoritmer er avveiningen mellom utforsking (exploration) og utnytting (exploitation). Med andre ord må en velge mellom å utforske mulige bedre løsninger, eller å utnytte de løsningene man allerede har kommet frem til. Problemet er at det ikke er mulig å gjøre begge disse samtidig.

Et godt eksempel som beskriver dilemmaet, er diamantgraving. Se for deg at du og en venn er ute og graver etter diamanter. Dere graver to forskjellige hull, med en liten avstand fra hverandre. Din venn finner en diamant og drar hjem. Du ser dette og blir grådig, og tenker at du også skal finne en diamant. Du bytter derfor hull og graver videre i hullet til vennen din. Handlingen du nå har tatt er en

grådig handling, og strategien din kan beskrives som en grådig strategi. I noen tilfeller kan det hende flere diamanter ligger nær hverandre, mens i andre tilfeller kan det hende de ligger svært spredt. En grådig strategi kan altså feile og resultere i at du ikke finner en diamant. I dette tilfellet vil den grådige strategien være utnytting, og den strategien som ikke tar hensyn til tidligere funn, er utforskning.

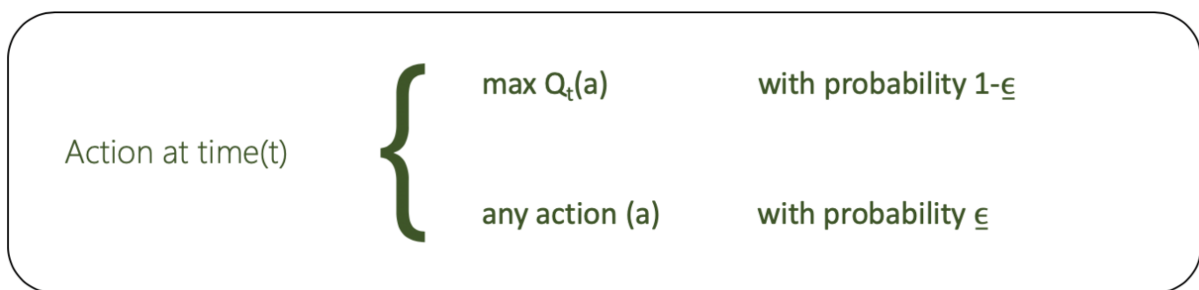
Det er viktig å se på hvilken informasjon du hadde da kompisen din fant diamanten. Du visste bare dybden han fant diamanten på. Med så lite informasjon om fremtidige tilstander og fremtidige gevinster, vil det være et dilemma om du skal bruke den informasjonen du har, eller om du skal fortsette å utforske ukjente handlinger for en større gevinst.

I en RL-algoritme vil agenten fungere slik som du gjorde i eksempelet over. Om agenten skal fortsette å grave hull der det er funnet diamanter før, eller grave på ukjente steder, er en avveining som må gjøres underveis (Khan, 2022). Denne avveiningen er illustrert i figur 1.



Figur 1 Evaluation & Improvement

En metode for å bestemme hvorvidt RL-algoritmen skal velge mellom å utforske eller utnytte, er å velge tilfeldig. Det kan være en som foretrekker å utforske kontra utnytte. Eksempelvis kan du trille en terning. Hvis terningen lander på én, skal algoritmen velge å utnytte (exploitation), ellers skal den velge å utforske (exploration). Denne metoden kalles *Epsilon Greedy Action* og blir representert i figur 2:



Figur 2 Epsilon Greedy Action

Med denne metoden vil beslutningstakeren med en sannsynlighet lik epsilon (ε) utføre den utforskende handlingen. Den vil dermed med en sannsynlighet lik 1 – ε velge den grådige metoden.

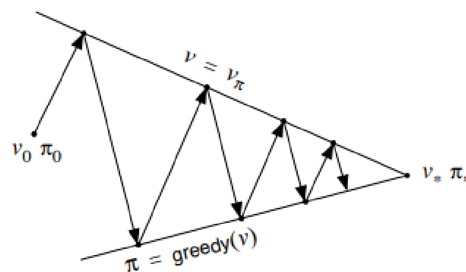
(GeeksforGeeks, 2022)

Med delvis eller ingen informasjon om fremtidige gevinster, vil Epsilon-greedy tilnærmingen gi best resultater. Det er fordi den balanserer mellom utnyttelse av nåværende kunnskap og utforsking av ukjente handlinger (Khan, 2022).

2.2.4 Prediksjons- og kontrollproblemet

Metodene for å løse *RL-problemer* er forskjellige, men det er i hovedsak to hovedidéer som gjelder. Disse er prediksjons- og kontrollproblemet. Kontrollproblemet gjengir hvordan den optimale strategien $\pi_*(s)$ blir estimert, med hensyn på tilstands-verdifunksjonen $v_\pi(s)$. Prediksjonsproblemet omhandler hvordan man utregner tilstandsverdifunksjonen $v_\pi(s)$ for strategien $\pi(s)$.

For å løse prediksjons- og kontrollproblemet vil man kombinere de gjennom noe som heter *Generalized Policy Iteration* (s.104 (Sutton & Barto, 2014, 2015)). Policy Iteration består av to prosesser som interagerer med hverandre samtidig, der den ene gjør verdifunksjonen konsistent med den nåværende strategien (*policy evaluation*), og den andre gjør strategien grådig med hensyn til den nåværende verdifunksjonen (*policy improvement*). Terminologien Generalized Policy iteration (GPI) blir brukt for å referere til den generelle idéen om å la policy improvement og policy evaluation interagere uavhengig av detaljnivået og detaljspesifikasjoner av de to prosessene. Prosessen i GPI illustreres i figur 3 (Sutton & Barto, 2014, 2015, s. 105).



Figur 3 Prosessen i generalized policy iteration

Figur 3 viser hvordan verdifunksjoner v(s) og strategifunksjoner $\pi(s)$ interagerer med hverandre frem til de er optimale og dermed konsistente med hverandre.

Dette er en veldig forenklet fremstilling av hvordan GPI faktisk fungerer, men figurene er enkle å forstå. I figur 3 representerer de diagonale linjene v og π en løsning til en av de to målene, policy improvement og policy evaluation. Figuren viser hvordan målene interagerer ettersom linjene ikke er

ortogonale. Å bevege seg mot et mål, betyr også at man beveger seg bort fra det andre målet. Men til slutt vil hele prosessen nærme seg et felles mål. Dette målet er totalt optimalisert, og det er uunngåelig at man beveger seg mot denne til slutt (Sutton & Barto, 2014, 2015, s. 105).

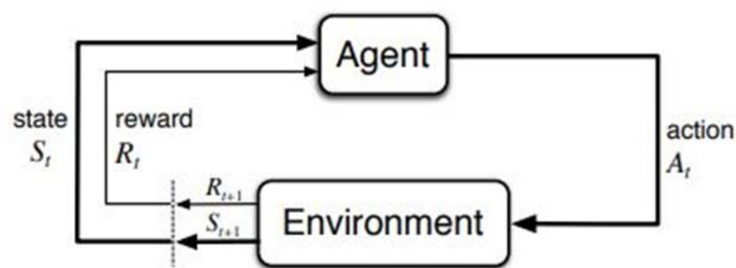
Det finnes flere forskjellige måter å utføre GPI på, og GPI legger grunnlaget for hvordan RL-metoder flest fungerer. Forskjellene ligger nyansert på detaljnivå, og oppgaven kommer derfor ikke til å diskutere disse ytterligere.

2.3 Vår metode for løsning av RL-problemet

En endelig Markov beslutningsprosess (MDP) er en viktig del av teorien rundt RL. I denne oppgaven diskuteres RL-problemer som deles inn i *diskré* tilstander og ulike avgjørelser som fører til en overgang fra én tilstand til en annen. Dette delkapittelet forklarer og bryter ned matematikken bak en endelig MDP og beskriver hvordan agenten tar beslutninger i miljøet.

2.3.1 Agenten i miljøet

Som forklart tidligere i oppgaven så handler RL om noe så enkelt som at en agent, som er en beslutningstaker, skal lære seg å samhandle med et miljø gjennom erfaring (Bhatt, 2018). Denne erfaringen får den ved at miljøet hele tiden gir tilbakemelding (en gevinst) på om beslutningen var god eller dårlig. Hele dette samspillet kalles for *beslutningsprosessen* og er illustrert i figur 4.



Figur 4 Hele beslutningsprosessen i en MDP. Beslutningstakeren (Agent) samhandler (action) med miljøet (environment) og havner i en tilstand (state). Dette gir en gevinst (reward) til beslutningstakeren (Sutton & Barto, 2014, 2015).

Beslutningstakeren og miljøet samhandler i diskrete tidsverdier, t = 0, 1, 2, 3, ... For hvert steg i tiden får beslutningstakeren en verdi for miljøets tilstand S_t med tilhørende gevinst R_t. En ny handling fra beslutningstakeren, A_t, gir en ny tilstand S_{t+1} og gevinst R_{t+1}. Store bokstaver

representerer bestemte tilstander, handlinger og gevinster bundet til et bestemt steg. Små bokstaver beskriver en vilkårlig tilstand, handling eller gevinst.

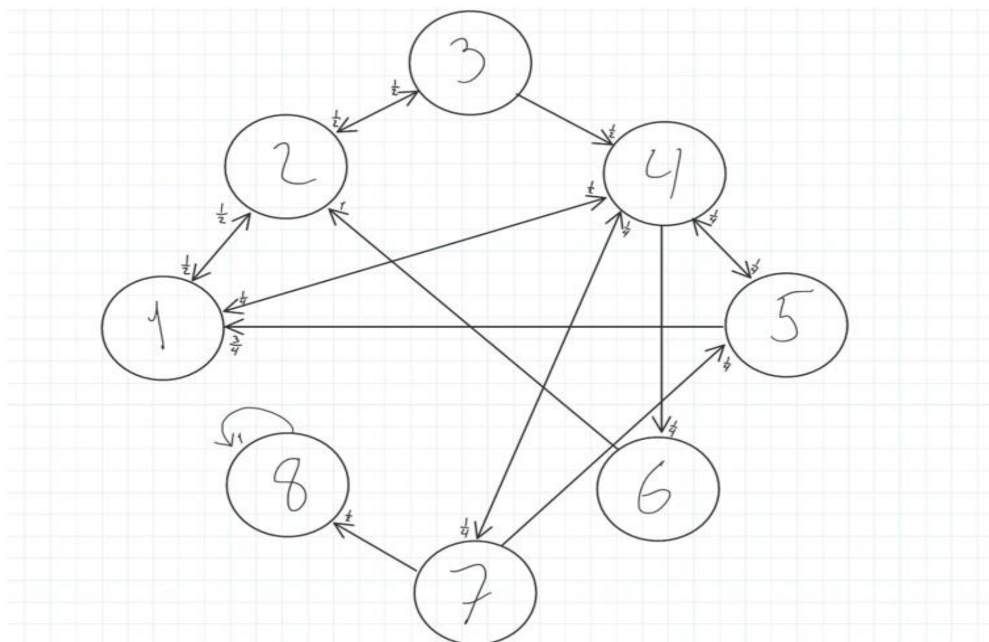
Dette er beskrivelsen av en vilkårlig MDP. Denne oppgaven tar kun for seg en endelig MDP. Dette betyr at mengden mulige tilstander, handlinger og gevinster er begrenset til et bestemt antall elementer. På grunn av dette er det en sannsynlighetsfordeling *p* for hvert valg av tilstand, *s*, og handling, *a*. Sannsynlighetsfordelingen er gitt som sannsynligheten *p* for en tilfeldig tilstand s' og en gevinst r, gitt en tilstand s og en handling a. Dette kalles for *overgangssannsynligheten* og beskrives her som hos Strandvold & Leines, 2018, s.13:

$$p(s',r|s,a) = p(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a)$$

Formel 1 Overgangssannsynligheten i en MDP

Alle overgangssannsynlighetene slått sammen, er MDP'en sin *overgangsstruktur.* Overgangsstrukturen er en beskrivelse av hele miljøet og forteller hvordan en beslutningstaker kan bevege seg mellom de ulike tilstandene.

Figur 5 viser et eksempel på en overgangsstruktur i en vanlig Markovkjede. Her representerer hver sirkel en tilstand, pilene viser hvilken tilstand beslutningstakeren kan velge og tallene på enden av pilene er overgangssannsynlighetene. Dette er en *ikke-reduserbar* Markovkjede. Det er fordi tilstand 8 er en *absorberende tilstand* og ikke mulig å forlate. Det kan man se ved at den eneste pilen ut fra tilstand 8 går tilbake til 8 og overgangssannsynligheten her er 1 (100%).



Figur 5 En ordinær Markovkjede

Overgangsstrukturen i figur 5 viser at overgangssannsynlighetene for alle mulige tilstander kun avhenger av foregående handling og tilstand. Hvor beslutningstakeren har vært, spiller ingen rolle for hvor den skal videre (så fremt den ikke går inn i den absorberende tilstanden). Skal man kunne bruke dette rammeverket til nytte, må hver tilstand inneholde informasjon om hvordan tidligere beslutninger har påvirkning på fremtiden. Har en tilstand denne informasjonen har den Markov-egenskapen (Sutton & Barto, 2014, 2015, s. 64). I denne oppgaven er Markov-egenskapen oppfylt, men det finnes ulike metoder innenfor RL hvor dette ikke er nødvendig. Disse skal ikke diskuteres ytterligere i denne oppgaven.

2.3.2 Mål, gevinst og utbytte

Målet til beslutningstakeren er hele tiden å maksimere sin gevinst. Dette gjøres ikke nødvendigvis ved å ta de valgene som umiddelbart gir størst gevinst, men ved å ta beslutninger som til slutt gir størst gevinst over tid (Bhatt, 2018). På samme måte som generaler og admiraler ikke bryr seg om å vinne slagene, men om å vinne krigen. På grunn av dette er det helt avgjørende hvordan man gir beslutningstakeren gevinsten. Gevinsten må bli satt ut ifra målet en ønsker å oppnå, ikke hvordan en selv mener beslutningstakeren skal nå målet. Dette fungerer på samme måte som oppdragsbasert ledelse i Forsvaret. Skulle man trent en beslutningstaker til å vinne en krig, kan man ikke gi den gevinst for å vinne et slag. Dette for å ikke risikere at beslutningstakeren blir god på å vinne slagene på bekostning av det faktiske målet, vinne krigen.

Målet til beslutningstakeren er, som tidligere nevnt, å maksimere gevinsten sin på lang sikt. Mer formelt kan man si at målet faktisk er å maksimere forventet *utbytte*, G_t. I sin enkleste form er G_t definert som summen av alle gevinstene:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ \dots \ + \ R_T$$

Formel 2 Utbytte

Hvor T er det siste tidssteget, dvs. steget inn i en absorberende tilstand. Dette gir mening i programmer eller spill hvor det er en naturlig progresjon i forhold til tid eller steg. Et eksempel er ruteplanleggeren, hvor det siste steget vil være da beslutningstakeren når den absorberende tilstanden - destinasjonen. Beslutningsprosessen fra start til destinasjonen, kalles for en *episode*.

Skulle en episode ikke ha noen absorberende tilstand eller ha antall steg lik uendelig, vil det være umulig å kalkulere forventet utbytte G_t, slik oppgaven definerer det. Dette løses ved å innføre det som kalles *diskontering*. Diskontering gjør at fremtidig utbytte maksimeres. Det gjøres ved å velge handlingen, A_t,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Formel 3 Diskontert utbytte

Hvor diskonteringsraten γ er en parameter, $0 \leq \gamma \leq 1$.

Ved å justere γ mot null prioriteres nærstående gevinster fremfor gevinster langt frem i tid. Ved γ = 1 vil den ikke gjøre forskjell på nære eller fjerne gevinster. Denne oppgaven forutsetter en absorberende tilstand. Den kommer derfor ikke til å bruke eller gå dypere inn på diskontert utbytte.

Formel 3 kan skrives om for å vise de rekursive egenskapene som kommer til nytte i algoritmene videre i oppgaven. Formel 4 viser hvordan de påfølgende utbyttene henger sammen;

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Formel 4 Diskontert utbytte, rekursivt skrevet

2.3.3 Strategi og verdifunksjoner

For hvert steg vil beslutningstakeren kartlegge sannsynlighetene for å velge hvilken tilstand den vil velge neste gang. Denne kartleggingen kalles for beslutningstakerens strategi og beskrives matematisk som π_t, hvor $\pi_t(a|s)$ er sannsynligheten for at $A_t = a$ hvis $S_t = s$. Strategien i denne oppgaven baserer seg på at fra hver enkelt tilstand $a_0(s)$ er det kun én riktig handling. Dette kalles for en deterministisk strategi og kan beskrives gjennom sannsynligheten $\pi(a_0(s)|s) = 1$. Strategien kan, basert på en sannsynlighetsfordeling, ses på som et kart over hver tilstand og dens respektive handling.

Enhver tilstand i en strategi har et forventet utbytte knyttet til strategiens handling. Dette beskrives gjennom verdifunksjonen, også kjent som tilstands-verdifunksjonen $v_\pi(s)$ for strategien π, definert som:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right]$$

Formel 5 Tilstands-verdifunksjonen

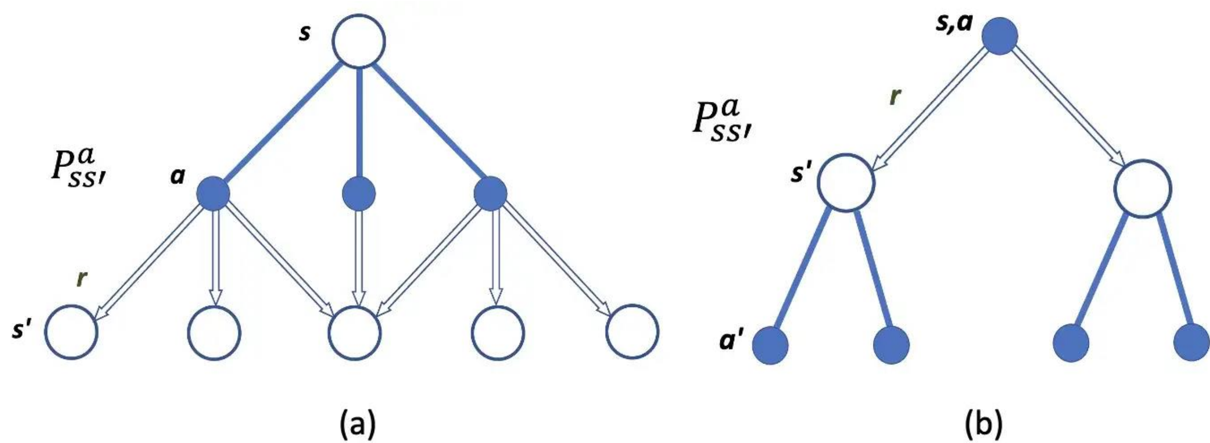
RL-problemet trenger også en annen funksjon kjent som handlings-verdifunksjonen. Den er svært lik verdifunksjonen $v_{\pi(s)}$, men det er en viktig forskjell. Der tilstands-verdifunksjonen forutsetter at strategien π følges for alle handlinger, forutsetter handlings-verdifunksjonen at handlingen a utføres,

og deretter at strategien følges. Handlings-verdifunksjonen beskriver dermed strategiens forventede utbytte, etter at en gitt handling a er utført. Handlings-verdifunksjonen kan beskrives gjennom funksjonen $q_\pi(s, a)$:

$$Q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]$$

Formel 6 Handlings-verdifunksjonen

Figur 6 illustrerer hvordan verdifunksjonene regnes ut, prosessen til verdifunksjonene, samt deres variabler og samspill.



Figur 6 Grafisk fremstilling av tilstands-verdifunksjonen og handlings-verdifunksjonen. P er sannsynligheten for at handling a, i tilstand s, ender opp i tilstand s' (med gevinst r). (Torres, 2020).

2.3.4 Bellmanligningen

For såkalte verdibaserte agenter er Bellmanligningen et av de mest sentrale elementene i RL. Ligningen brukes til å dekomponere verdifunksjonen inn i to deler; den umiddelbare gevinsten samt de diskonterte fremtidige verdiene (Torres, 2020).

Bellmanligningen for verdifunksjonen kommer fram ved å kombinere verdifunksjonen v_π med de rekursive egenskapene til G_t som ble presentert i Formel 4.

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

$$= E_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$

$$= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

Formel 7 Bellmanligningen for tilstands-verdifunksjonen

Her kommer sammenhengen mellom forventet utbytte til en tilstand samt dens suksessive tilstander fram, ved å sette inn for G_t fra formel 4. Det samme kan også gjøres for handlings-verdifunksjonen.

$$q_\pi(s, a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, a')|S_t = s, A_t = a]$$

Formel 8 Bellmanligningen for handlings-verdifunksjonen

De to siste ligningene er de to som hver utgjør sin den del av Bellmanligningen. Ved å ta Bellmanligningen for verdifunksjonen, kan man danne et lineært ligningssett som kan løses eksakt for å finne den optimale tilstands-verdifunksjonen $v_*(s)$, som også vil være tilstandens høyeste mulige verdi. På samme måte kan den optimale funksjonen $q_*(s, a)$ utarbeides for handlings-verdifunksjonen $q_\pi(s, a)$.

Bellmanligningen forenkler utregningen av verdifunksjonen, slik at man slipper å summere for hvert enkelte steg. Dette gjøres ved at man finner den optimale løsningen til et komplekst problem, ved å utnytte de rekursive egenskapene til G_t og dele opp problemet i flere mindre og enklere delproblemer (Torres, 2020).

2.3.5 Optimal strategi

Målet til agenten er å maksimere den totale og kumulative gevinsten over tid. Strategien som sørger for å maksimere den totale kumulative gevinsten kalles for den optimale strategien, og noteres π_*. Ettersom dette er den strategien som best oppfyller målet til agenten, vil dette også være løsningen på RL-problemet. Det kan eksistere flere samtidige optimale strategier, ettersom forskjellige handlinger kan føre til samme sluttilstand, men den optimale verdifunksjonen vil det aldri eksistere mer enn én løsning for (Dobillas, 2022).

Det å løse en MDP er ensbetydende med å finne den optimale tilstandsfunksjonen. Matematisk kan dermed funksjonen beskrives som:

$$v_*(s) = \max_\pi v_\pi(s)$$

Formel 9 Optimal tilstands-verdifunksjon

Formelen over viser oss at $V_\pi(s)$ er den høyeste gevinsten en for tilstand i systemet. (Jing, 2022)

På samme måte viser den matematiske funksjonen til den optimale handlingsfunksjonen hva som blir den høyeste gevinsten i tilstanden etter en gitt handling:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

Formel 10 Optimal handlings-verdifunksjon

$v(s)$ kan også defineres via $q(s, a)$ slik at verdien for en tilstand er lik verdien for den maksimale handlingen fra denne tilstanden:

$$v(s) = \max_a q(s, a)$$

$$v_*(s) = \max_a q_*(s, a)$$

Formel 11 Tilstands-verdifunksjonen skrevet fra handlings-verdifunksjonen

Det å kunne løse Bellmanligningene og finne den eksakte optimale strategien, vil ofte være tilnærmet umulig. Strandvold og Leines trekker fram i sin oppgave noen eksempler som viser vanskeligheten med en eksakt optimal strategi, men også hvor god en tilnærmet optimal strategi kan være. Selv om det i et sjakkspill finnes flere mulige trekk enn atomer i universet (rundt 10^{120} mot rundt 10^{80}) klarte RL-agenten AlphaZero å slå daværende verdensmester Stockfish, i Top Chess Engine i 28 av 100 spill. Resten endte i remis. (Strandvold & Leines, 2018, s. 17)

For å unngå problemet med å finne en eksakt løsning, finnes det flere metoder. Typisk må man inngå et kompromiss der man for eksempel velger seg ut noen situasjoner der agenten yter mindre optimalt. Dette er gjerne situasjoner med lav sannsynlighet for å skje, slik at man kan ha en optimal og pålitelig strategi ellers (Sutton & Barto, 2014, 2015).

Denne oppgaven unngår problemet med den eksakte løsningen ved å bruke prøve-og-feilemetoden. I et miljø som endrer seg basert på ønsket rute, vil det være vanskelig å få en eksakt løsning ettersom dette forutsetter en full kjennskap og forståelse for miljøet og dets dynamikk. Prøve-og-feilemetoden utforsker miljøet og estimerer verdifunksjonene basert på samspillet og erfaringene som samles i løpet av agentens beslutninger og interaksjoner med miljøet. På denne måten samles det inn data som kan estimere løsningen til Bellman-ligningene.

2.3.6 Oppsummert

Det å løse RL-problemet handler om å finne den mest optimale strategien, og dermed om å løse Bellmanligningene innenfor rammene gitt av Markov beslutningsprosess. Agenten interagerer med miljøet og samler på denne måten inn data i form av tilstand, handling og gevinst, som videre brukes for å kalkulere et forventet utbytte eller verdi. Det er det forventede utbytte som Bellmanligningene optimerer, for å få de største mulige verdiene til en optimal strategi. Denne strategien vil resultere i en maksimal total og kumulativ gevinst som oppfyller målet til agenten, slik at RL-problemet er løst.

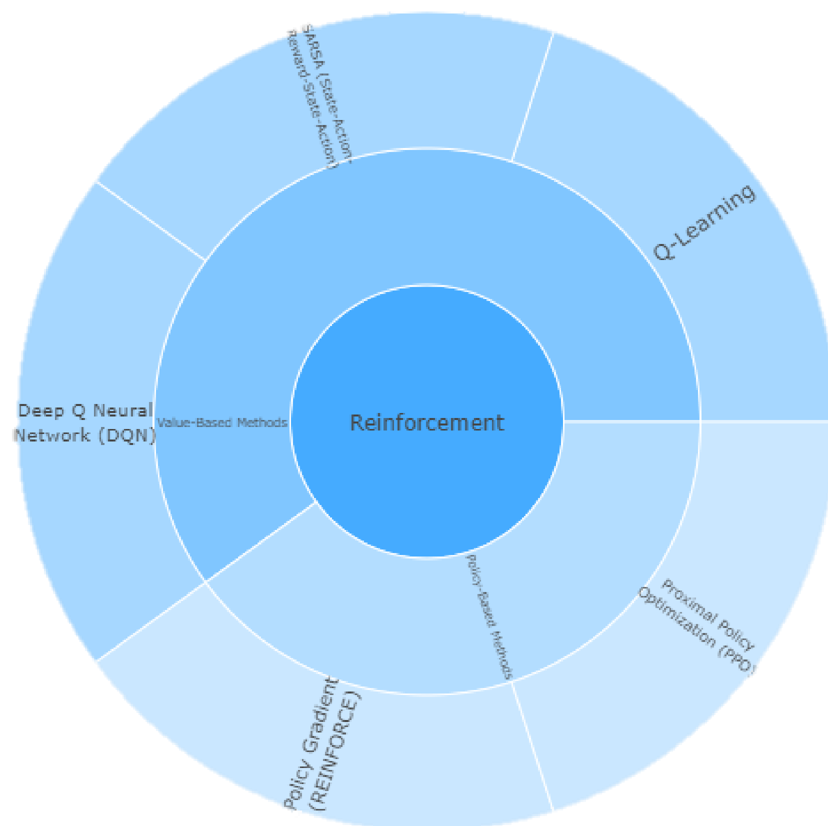
Her må viktigheten til hva agenten *egentlig* er påpekes. Beslutningstakeren jobber kun med å optimalisere et matematisk regnestykke. Den vet ikke hvorfor eller hva start og slutt er. Den optimaliserer regnestykket kun basert på de gitte parameterne. Som Strandvold og Leines skriver:

«Den [agenten] er med andre ord på ingen måte «smart» eller «intelligent» fordi den ikke reflekterer rundt hvorfor det den gjør fungerer, selv om det fra utsiden virker slik i og med at den er i stand til å ta gode/optimale beslutninger» (Strandvold & Leines, 2018, s. 18).

2.4 Q-learning

For å løse RL-problemet i denne oppgaven, benytter oppgaven seg av en metode innenfor *temporal-difference learning* (TD) som heter *Q-learning*. Felles for TD-løsninger er bruken av erfaringsbasert læring for å oppdatere verdifunksjonene mens den trener. Ved hjelp av teknikker nevnt i forrige delkapittel, estimerer TD-løsninger kontinuerlig beste handling, og handler før den får den endelige løsningen. På denne måten finner beslutningstakeren en erfaringsbasert løsning som muliggjør læring i hvert trekk gjennom hele episoden (Shyalika, 2019).

I figur 7 ser vi Q-learning sin plassering i reinforcement learning-verden. Q-learning faller inn under de verdi-baserte metodene. Mer om dette i de neste avsnittene.



Figur 7 Viser Q-learning sin posisjon i reinforcement learning-verden (Dobillas, 2022).

For Q-learning-algoritmen er det viktig å huske at den er en verdi-basert metode, og dermed det som kalles for en *off-policy*-algoritme. Dette betyr at den under trening bruker forskjellige strategier for å handle (handlingsstrategien) og for å oppdatere Q-funksjonen (handlings-verdifunksjonen). Enklere forklart vil dette gi oss muligheten til å oppdatere verdiene til strategien for å finne den optimale strategien, samtidig som agenten ikke må følge denne strategien. Dermed estimeres en optimal strategi, samtidig som agenten kan utforske nye handlinger og lete etter bedre verdier. Det motsatte kalles en *on-policy*-algoritme, hvor samme strategi benyttes for handling og oppdatering av verdier. Da må man godta at agenten alltid vil følge strategien. (Dobillas, 2022)

Q-funksjonen beskriver hvordan Q-learning-algoritmen trener:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Formel 12 Q-funksjonen

Venstresiden $Q(S_t, A_t)$ er den oppdaterte verdien for det spesifikke tilstands-handlingsparet. Det første leddet på høyre side $Q(S_t, A_t)$ er den nåværende verdien for det samme tilstands-handlingsparet. For å oppdatere den nåværende verdien brukes gevinsten for den påfølgende handlingen R_{t+1}, men legger til den høyeste verdien fra den neste tilstanden $\gamma max_a Q(S_{t+1}, a)$ diskontert ved gamma, og trekker fra den nåværende verdien $Q(S_t, A_t)$. Variabelen alfa brukes for å kontrollere hvor stor påvirkning hver oppdatering skal ha for den nye verdien. Ved å se på verdiene i klammeparentesen, ser man at den enten kan bli null, et positivt eller et negativt tall. Dermed vil den nye Q-verdien til tilstands-handlingsparet enten øke, synke eller forbli det samme. (Dobillas, 2022)

I maskinlæringsverdenen kalles det å kjøre algoritmen, som $Q(S_t, A_t)$ rett over, for *trening*. Dette er fordi hver gjennomgang estimerer verdifunksjonene mer og mer mot den optimale løsningen. Dermed kan hver handling og hver episode estimere en bedre rute, og på denne måten trener algoritmen og agenten på RL-problemet.

Dette er metoden programmet bruker for å løse RL-problemet i oppgaven. Selv om programmet tar inn flere variabler og parametere, er prinsippet det samme. Ved hjelp av teorien i oppgaven, samt utviklingen av konseptet i neste kapittel, burde det være mulig å gjenkjenne og forstå kodene for programmene som er vedlagt. Programmet Main.py er hovedprogrammet som kjøres og inneholder koden for trening.

2.5 Brukergrensesnitt

Et brukergrensesnitt er et grensesnitt som skal gjøre det mulig for mennesker å samhandle med en maskin. Det finnes to typer brukergrensesnitt til datamaskiner; grafisk brukergrensesnitt (GUI) og tekstbasert brukergrensesnitt (TUI). Et GUI er et grensesnitt for dataprogrammer som gjør det mulig å samhandle med datamaskinen gjennom tastatur, datamus og lignende. GUI er grafisk fremstilt og veldig ofte mer intuitivt og brukervennlig enn motparten TUI. TUI baserer seg på å skrive inn kommandoer i shell-applikasjoner og krever gjerne mer fra brukeren i form av datakjennskap. (Churchville, 2021)

Det meste fra et operativsystem til installerte programmer bruker GUI. Dette gir brukere tilgang til funksjoner og informasjon i programmene. En GUI bruker grafiske elementer som ikoner, bilder og knapper for å la brukeren samhandle med programmet. Dette gjør at selv lite datakyndige brukere kan bruke avanserte programmer. Det finnes også noen nedsider med GUI. Et enkelt og brukervennlig program gir gjerne mindre fleksibilitet ettersom bare forprogrammerte instruksjoner kan bli utført. Sammenlignet med en TUI, trenger en god GUI ofte mye mer harddiskplass, den er mer komplisert å lage og kommandoer tar gjerne lenger tid å utføre. Derfor er det viktig å tilpasse brukergrensesnittet til brukerne som faktisk skal bruke det. (Hope, 2021)

3. Utvikling av konseptet

Dette kapittelet vil se på hvordan utviklingen av konseptet har skjedd. Dette er viktig fordi det sier noe om hvorfor løsningene gjennom oppgaven er som de er.

3.1 Konseptuelle krav

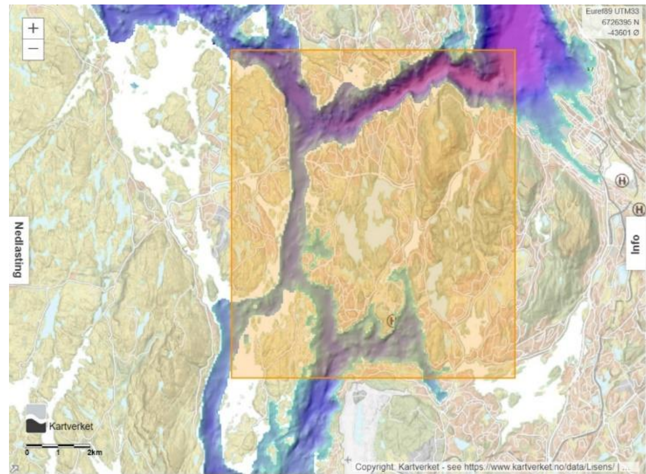
Oppgaven setter en del krav til konseptet, som programmet og algoritmen dermed må oppfylle. Kravene er satt på bakgrunn av oppgavens mål og begrensninger, og er beskrevet i tabell 1 under.

Enhet	*Krav*	*Beskrivelse av måloppnåelse*
Agent	Korrekt Avgjørelse	Agenten tar en korrekt avgjørelse når avgjørelsen fører til den mest effektive ruten til målet. Den skal også oppdage og omgå hindringer i kartverket.
	Kjapp beslutning	Programmet kan ikke bruke for lang tid til utregning. Det må oppfattes som relativt kjapt og smidig, slik at det blir en faktisk tidsbesparelse for navigatørene.
	Takler dilemmaer	Dersom det finnes flere relativt like ruter, eller flere suboptimale ruter, skal agenten velge den som fremstår som mest effektiv og hensiktsmessig.
Kostnadskart	Leser kartet riktig	Programmet må lese kartet riktig og nøyaktig, slik at kostnadskartet gir troverdige og representative data til miljøet agenten opererer i.
	Tilrettelagt for sekvensiell bevegelse	Miljøet må være konstruert slik at agentens handlinger kan spores og utføres sekvensielt.

Tabell 1 Systemkrav

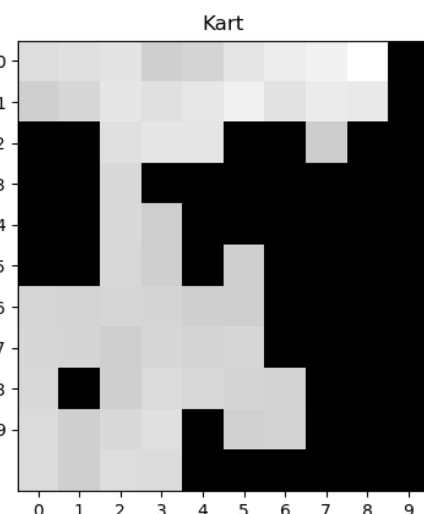
3.2 Miljø

Oppgaven bruker et *virtuelt miljø* basert på åpne digitale kart fra Kartverket. Disse kartene er detaljerte dybdekart som kun gir oss dybdedata i sjøen og ikke land.[1] Figur 8 viser kartutsnittet dataene er basert på, og området valgt for testing. Land er her grafisk fremstilt, men ligger ikke i matrisen som ble lastet ned fra Kartverket. Derfor settes det en kunstig stor høyde til alle de områdene som ikke har data, nemlig land.



Figur 8 Kartutsnittet til miljøet.

I figur 9 kan presenteres et rutekart med oppløsning 10x10 piksler, etter at land er gitt en kunstig høyde. På denne måten kan det settes en stor negativ gevinst, eller kostnad, for å kjøre på land. Slik unngår beslutningstakeren disse rutene som representerer land.



Figur 9 viser Kartutsnittet fra figur 8 fremstilt i et rutekart med 10x10 piksler.

Rutekartet blir initialisert ved at xyz-dataene fra Kartverket plottes inn i tilsvarende riktig piksel basert på denne formelen:

$$ix = \frac{X_{[i]} - \min x}{dx}$$

Formel 13 x-koordinatens plassering

der ix er x-koordinaten for plassering, $\min x$ er tallverdien for den minste x-koordinaten i datasettet, og dx er den største verdien til x minus den minste verdien til x delt på antall piksler det ønskes å ha i x-retning. Akkurat det samme har blitt gjort for tilsvarende y-verdier:

$$iy = \frac{Y_{[i]} - \min y}{dy}$$

Formel 14 y-koordinatens plassering

Dette plassert inn i en tom matrise, gir alle verdiene som trengs for å konstruere et rutekart for vannområdet. Siden det kun er data for vannområder, vil de verdiene som fortsatt er tomme i

[1] Kartverkets dybdekart er fritt tilgjengelig, og våre data ble lastet ned direkte fra deres nettside: https://dybdedata.kartverket.no/DybdedataInnsyn/

matrisen være land. Landområdene settes lik null, mens vannområdet overskrives med hver piksels snittdybde. Slik er det laget et dynamisk system, der man kan endre på antall piksler i hver retning for å kunne teste med forskjellige oppløsninger i kartet.

Ettersom algoritmen beveger seg i et rutekart så beveger den seg sekvensielt. Dette betyr at den går med diskré verdier. For at algoritmens sekvensielle bevegelse skal fungere må det virtuelle miljøet være tydelig definert inn i et grid-system i form av piksler. Det virtuelle miljøet må altså representeres som en tolkning av det gitte fysiske området, ved hjelp av MDP-teorien.

Dataene fra Kartverket kommer som en lang liste med tilhørende x-, y- og z verdier i form av en matrise, sortert med sørvestligste- og nordøstligste-punkt henholdsvis først og sist. Dette grid-systemet baserer seg på kartreferansen *easting* & *northing*, altså hvor langt øst for et punkt det er og hvor langt nord fra ekvator det er. Dette baserer seg på kartreferansen EUREF89[2]. Gridsystemet gir også en verdi, z, for snittverdien til dybden i den gitte piksel.

Et kartutvalg som tilsvarer Bergen kommune, har data i underkant av 15 000 tallverdier som piksler. For å effektivisere utregning skaleres derfor kartutsnittet ned slik at det i stedet blir et rutekart på underkant av 150x150 piksler. Dette rutekartet vil være noe mindre nøyaktig enn ved full skala, men vil fortsatt gi en god nok nøyaktighet til å kunne vise navigerbare leder og passasjer som agenten kan benytte. Hadde rutekartet vært ned mot 10x10 piksler eller 20x20 piksler ville det kunnet utelukke enkelte leder og passasjer, og i verste fall gi forslag på ruter som ikke er traverserbare i virkeligheten.

Ved hjelp av dette grid-systemet kan hver enkelt rute få en gevinst basert på rutens egenskaper, i dette tilfellet dybde. Gevinsten representerer «fordelen» ved å havne i den gitte ruten, uavhengig av hvilken rute du kommer fra. Slik danner programmet et virtuelt miljø som agenten kan traversere sekvensielt mot målet.

Det virtuelle miljøet er direkte basert på teorien. Agenten befinner seg i miljøets tilstand S_t ved steg $t.$ Tilstandsrommet S er definert som alle tilstandene agenten kan befinne seg i. Hver enkelt tilstand har sitt handlingsrom. Handlingsrommet til agenten i rutekartet, er å bevege seg opp, ned, til venstre eller til høyre. Dette er kjent som handlingsrommet $A(s)$. På denne måten har algoritmen de dataene den trenger for at agenten skal kunne samhandle sekvensielt med miljøet, og dermed kunne

[2] EUREF89 er den geodetiske referanserammen, tidligere datum, for Norge og andre europeiske land.

beregne sannsynligheter for mulige traverseringer og strategier. Alt basert på et ønske om høyest mulig gevinst – som tilsvarer kortest mulig vei i dypest mulig farvann.

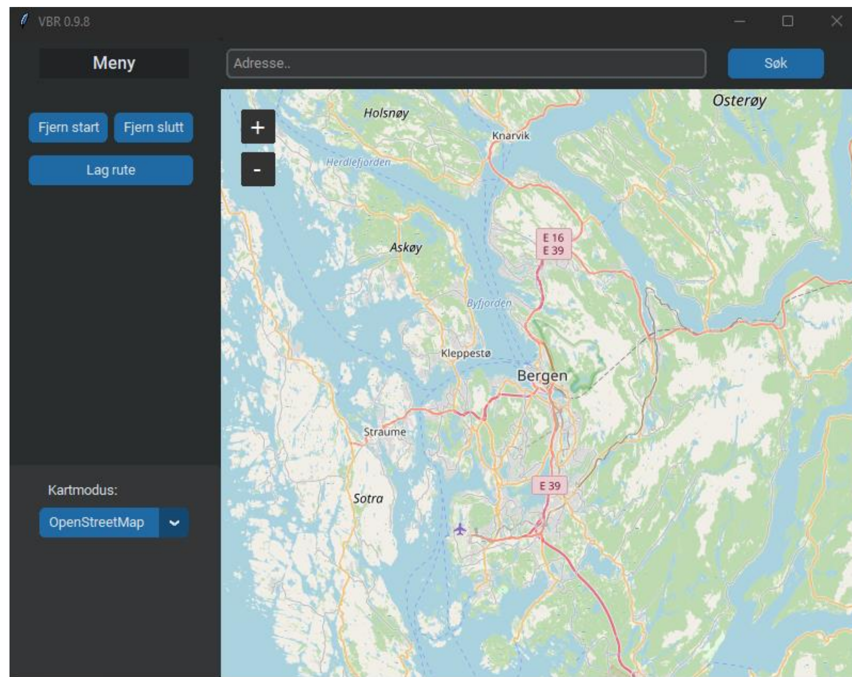
3.3 Graphical User Interface

Tidligere i oppgaven ble det forklart hva et brukergrensesnitt er og hvorfor det er nødvendig i programmer. Brukergrensesnittet til programmet i denne oppgaven består av en GUI. Dette er basert på at programmet skal bli brukt av navigatører og at det derfor skal kunne brukes uten noen særlige forkunnskaper i programmering. Ved å kunne kjøre algoritmen gjennom en GUI, kan brukerne raskt og effektivt bruke ruteplanleggeren som et verktøy til navigasjon.

I denne oppgaven er det brukt programmeringsspråket Python for å lage en GUI. Python har et standard grensesnitt-verktøy som heter Tkinter. Tkinter er et eldre bibliotek, og oppgaven bruker derfor CutomTkinter, et moderne Python brukergrensesnitt-bibliotek basert på tkinter. Dette biblioteket er lagd av Tom Schimansky.[3]

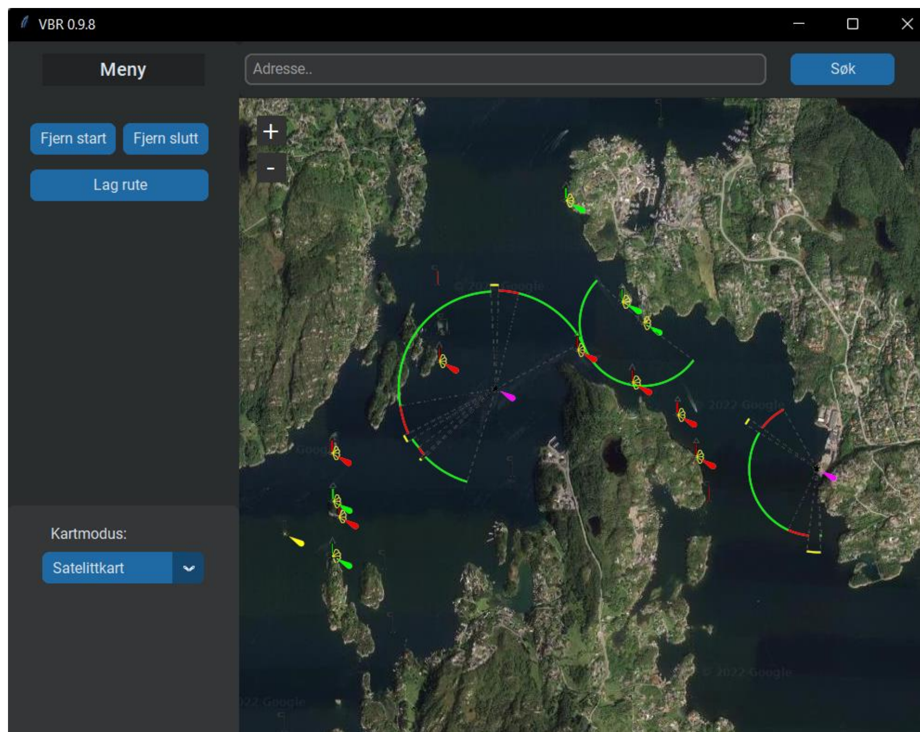
For å lage et GUI for en ruteplanlegger, krever det visse funksjoner for at den skal være operasjonell. Det må være et kart som brukeren kan samhandle med, og brukeren må kunne legge inn start- og sluttpunkt for ruten. Dette er løst med TkinterMapView som er et interaktivt kart for Python Tkinter-biblioteket. TkinterMapView henter ulike kart fra Google Maps, som for eksempel satellittkart eller veikart. Det støtter også filtre man kan legge over kartet, som OpenSeaMap. Dette er fordelaktig i en maritim ruteplanlegger.
For å få lagt inn start- og sluttpunkt benytter ikke menyen seg av knapper, men at man direkte høyreklikker på stedet man ønsker å sette start/sluttpunkt. På denne måten er det enkelt å velge et nøyaktig sted for ønsket start og slutt for ruten. Punktet blir lagt til en egen liste i Python. Skal man legge til enda et start/sluttpunkt, blir det gamle punktet slettet ved at listen overskrives.

[3] Biblioteket er lagt ut på GitHub: https://github.com/TomSchimansky/CustomTkinter



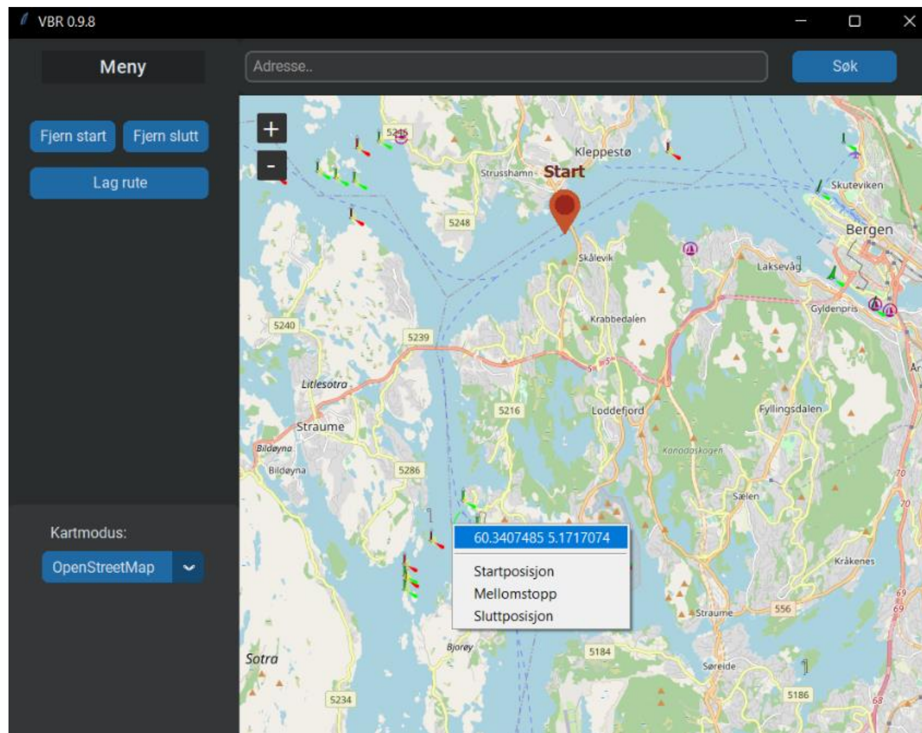
Figur 10 Oppstartsskjermen til brukergrensesnittet.

Figur 10 viser oppstartsskjermen til GUI. Her er det prøvd å holde brukergrensesnittet så minimalistisk og enkelt som mulig. Øverst er det et søkefelt hvor brukeren kan skrive inn adresse, bynavn eller grader for å finne et sted. Dette er ikke en nødvendig funksjon, men det kan være praktisk å bruke for å navigere i kartet. Under søkefeltet er kartet. Det bruker en ved å klikke og dra frem og tilbake. Man kan også zoome inn/ut ved hjelp av datamus eller med pluss og minus knappene i hjørnet av kartet. Nederst til venstre kan man velge kartmodus. Her kan man velge mellom OpenStreetMap, vanlig kart og satellittkart. I figur 11 nedenfor er et eksempel på hvordan satellittkart ser ut.



Figur 11 brukergrensesnittet med sjøkart-filter.

Figur 11 viser også sjøkarts-filteret OpenSeaMap som er lagt over kartet. Filteret synes ikke når man har zoomet ut veldig mye, men detaljene i filteret kommer frem når brukeren zoomer inn.
Ved å høyreklikke på kartet vil det dukke opp en liten meny. Da får man opp alternativer om å sette ut ulike markører på kartet som representerer start-, mellom- og sluttpunkt av ruten. Dette er satt i en egen meny fordi det er praktisk å kunne trykke nøyaktig der man vil ha en markør. I figur 12 er det et eksempel hvor det har blitt satt ut en startmarkør ved Askøybrua og høyreklikkmenyen er satt opp rett nord for Bjorøy.



Figur 12 brukergrensesnittet i bruk.

Øverst til venstre er det lagt til tre knapper. Fjern start og fjern slutt vil fjerne eventuelle markører brukeren har satt ut. På denne måten kan en fjerne start- og sluttmarkører uten å måtte sette ut nye. Under fjern knappene er «lag rute». Den trykker brukeren på når hen har satt ut ønsket start- og sluttpunkt. Det første som skjer, er at gradene til punktene blir oversatt til koordinater som algoritmen kan lese. Disse blir lagt i variabler som algoritmen leser av, og brukes til å kalkulere ruten. Når ruten er lagd blir koordinatene til hvert vendepunkt i ruten oversatt til grader og sendt tilbake til kartet. Mer om resultatene av ruten under 4.3 Brukergrensesnitt

4. Tester og resultater

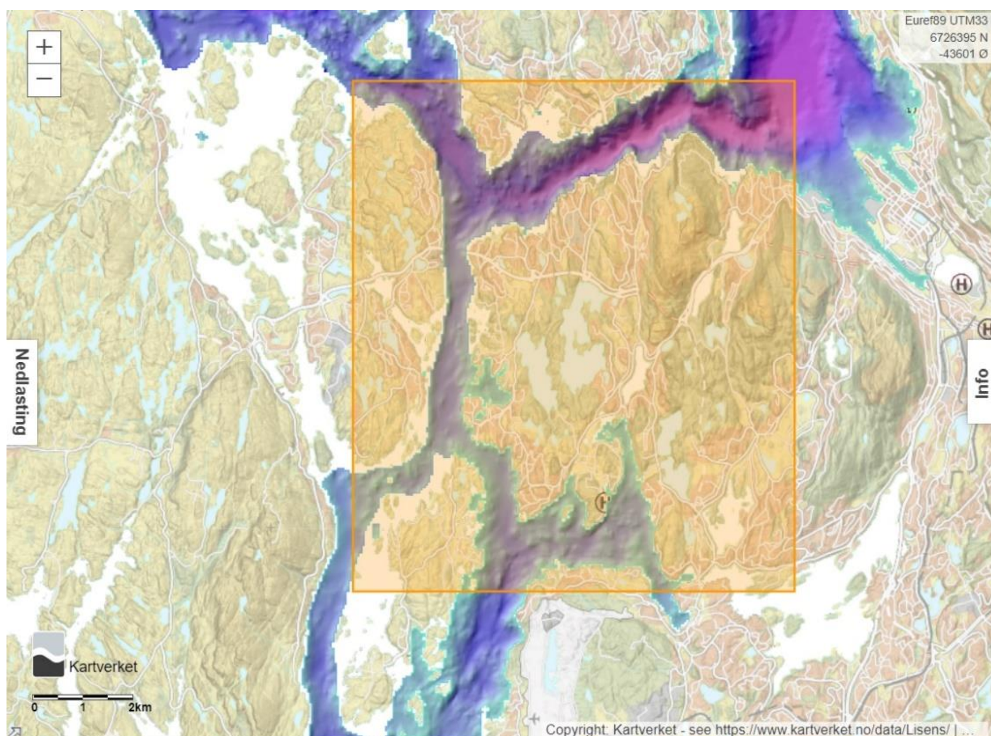
Dette kapittelet tar for seg systematisk og kronologisk testingen og resultatene gjennom utviklingen av programvaren. Kapitelet ser nærmere på de forskjellige løsningene underveis i oppgaven, noe som er viktig for å forklare hvordan sluttproduktet ser ut. Det presiseres igjen at testingen og resultatene gjelder for utviklingen av kostnadskartet, den grafiske fremstillinga av dataene, brukergrensesnittet og for treningen og implementeringen av RL-algoritmen.

4.1 Datafremstilling

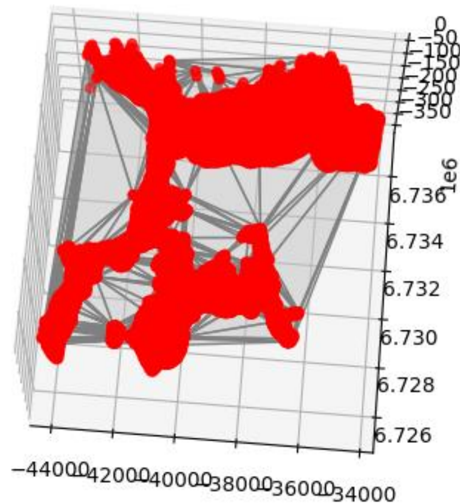
Dette delkapittelet vil se nærmere på hvordan den grafiske fremstillingen av dataene ble til. Dette gjøres både for brukerinteraksjon og for utvikling av verktøyet. Oppgaven ser også nærmere på førsteutkastet og resultatene fra idéer underveis.

4.1.1 Matriserepresentasjon

Oppgavens førsteutkast på en matriserepresentasjon at kostnadskartet skulle være basert på en matrise. En tredimensjonal matrise som skulle fungere som grunnlag for et miljø for agenten å bevege seg i. Data fra om området Bergen fra Kartverket i xyz-format, easting, northing og dybde, hentes og legges inn i en numpy-fil. Det er viktig å spesifisere at her er det kun koordinatene for der det er vann og dermed dybde. Dataene fra Kartverket har ingen koordinater for landområder rundt. Området kan ses i figur 13.



Figur 13 Kartutsnittet for dataene. Veldig likt det i figur 8.

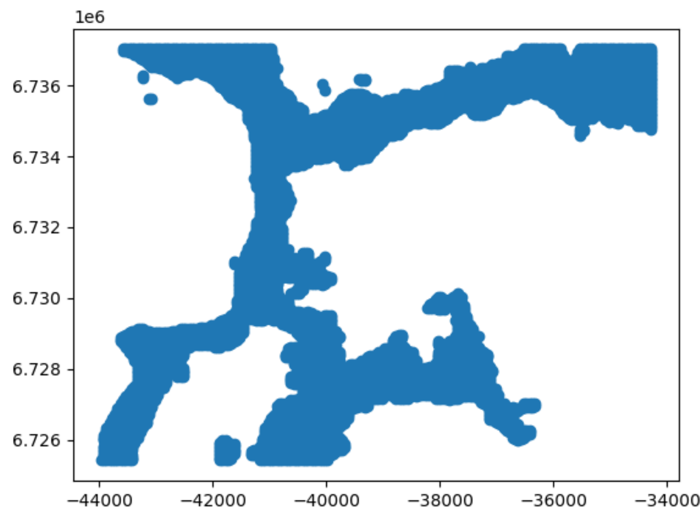


Figur 14 Dataen representert i et 3D-plot.

Figur 14 viser den første fremstillingen av kartdataene. Her er vannet fra figur 13 representert i rødt. Ser man nærmere, ser man at det er samme kartutsnittet presentert med dybdeverdien langs z-aksen. Figur 14 viser en grafisk fremstilling av dataene, men den er vanskelig å bruke. Det er mange og store tallverdier klemt inn i en liten figur, noe som gjør det vanskelig å representere hvor man er på et kart, og hva tallverdiene faktisk betyr. Kildekoden for denne fremstillingen finner du i vedlegg A.

4.1.2 Fast dybderepresentasjon

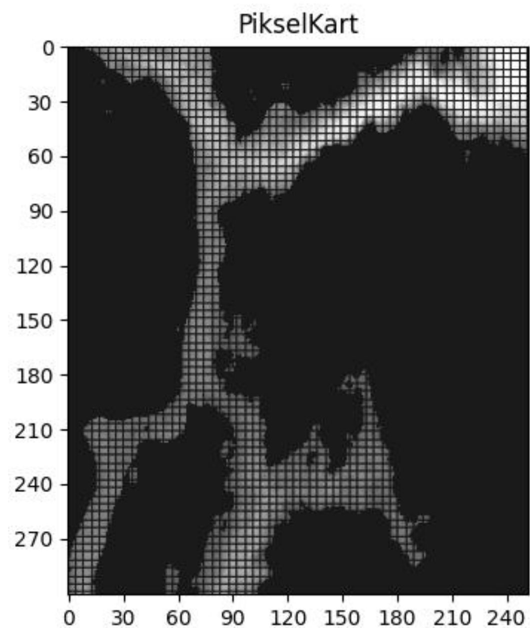
Neste figur viser en fremstilling av kartdataene, hvor kartet er plottet todimensjonalt og ikke tar hensyn til dybden. Den plotter bare for x og y, Easting og Northing, for å skape en enkel representasjon av hvordan kartet ser ut. Easting sitt nullpunkt er standardisert i henhold til EUREF89. Kartet ligger altså 34 000-44 000 meter vest for x-aksens nullpunkt. Northing er antall meter fra ekvator. Her er det oppgitt langs y-aksen fra 6,726 millioner til 6,736 millioner meter. Dermed er førsteutkastet i 4.1.1 mer komplisert. Bakgrunnen for forenklingen var å forsøke å simulere en labyrint, og dermed en navigasjonsrute for RL-algoritmen. Derimot er det kun tallverdier for havområder, og siden det dermed ble «hull» i matrisen der det er land, kunne ikke resultatet brukes for RL-algoritmen. Figur 15 viser hvordan funksjonen «scatter» er brukt i Python for å lage en enkel grafisk representasjon. Kildekoden for denne grafiske fremstillingen finner du i vedlegg B.



Figur 15 Samme område som figur 13, men illustrert som et scatter-plot

4.1.3 Et labyrintkart av piksler

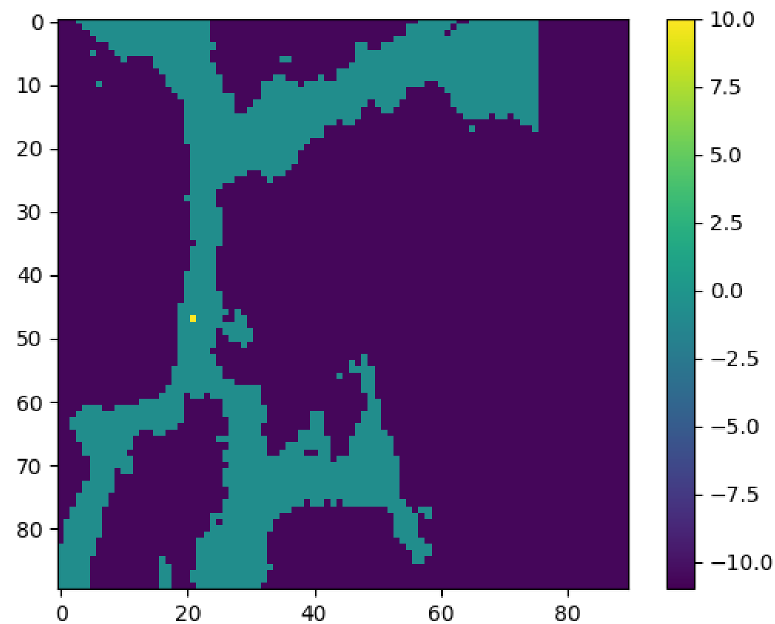
Figur 16 representerer et labyrintkart av piksler basert på 4.1.1 og 4.1.2. Dette er en grafisk representasjon av kostnadskartet. Her er det etablert en matrise, der tall i matrisen representerer dybdeverdi, og hvert tall ligger geografisk riktig plassert i matrisen basert på x- og y-koordinater fra numpy-fila. Den todimensjonale representasjon av vestlige Bergen er beholdt, men hver piksel i bildet representeres med en fargekode for hvor dypt det er. Jo dypere snittdybden i én piksel er, jo lysere er pikselen på kartet. Verdien 100 er satt for alle landområder.



Figur 16 Kostnadskart

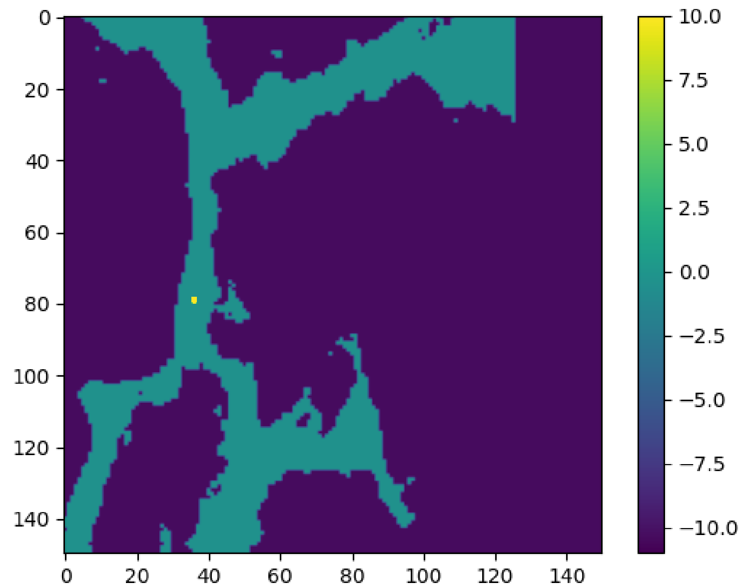
Dette gir en veldig mørk farge, men de mørke områdene gir en god visuell representasjon av hvor utsnittet er tatt fra. Sammenlignet med kartutsnittet i figur 13, er dette en lik fremstilling av dataene gjennom et kostnadskart. Koden for kartet kan ses i vedlegg C.

4.1.4 Gevinst-kart



Figur 17 Gevinstkart 90x90 piksler.

I figur 17 er gevinsten man får for hver piksel lagt inn i rutene som en fargekode. Fargekoden er i relasjon med dybden som er gitt et område. Her er dybden på land null, som gir en gevinst på minus ti. Gevinsten for piksler med havområder som er traverserbare, altså over ti meter dyp, har gevinst på null. Den gule prikken man ser i figuren er oppgitt mål-posisjon. Når agenten når denne fra en gitt startposisjon, har en *iterasjon* blitt gjennomført. Slik som verdimåleren til høyre viser, har alle rutene som er på land en negativ gevinst mindre enn -10. Mål har en gevinst som er lik 10. Langs aksene er det rader og kolonner i tallmatrisen som figur 17 er basert på. Tallmatrisen har like mange rader og kolonner i hver retning som det er piksler i hver retning. Oppløsningen på dette kartet er 90x90 piksler, noe som gir et mindre detaljert kart. Figur 18 viser samme kart, men med en oppløsning på 150x150 piksler.



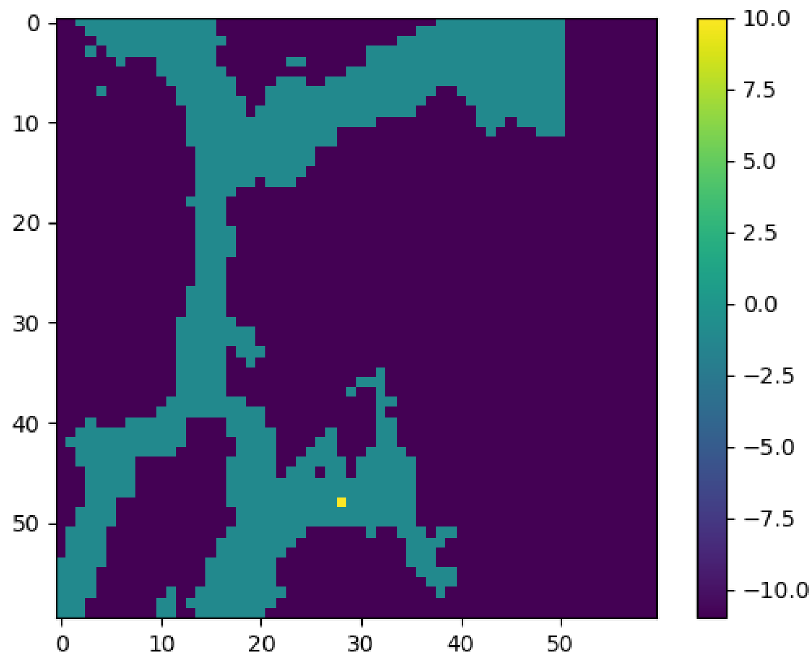
Figur 18 Gevinstkart 150x150 piksler.

Dette er et mer oppløst og detaljrikt kart. Dette kommer synlig frem langs landområdene. Det er dermed lettere å se nyansene i kartet opp mot det originale kartet, representert i utdraget i figur 13.

4.1.5 Oppløsning og tidsbruk

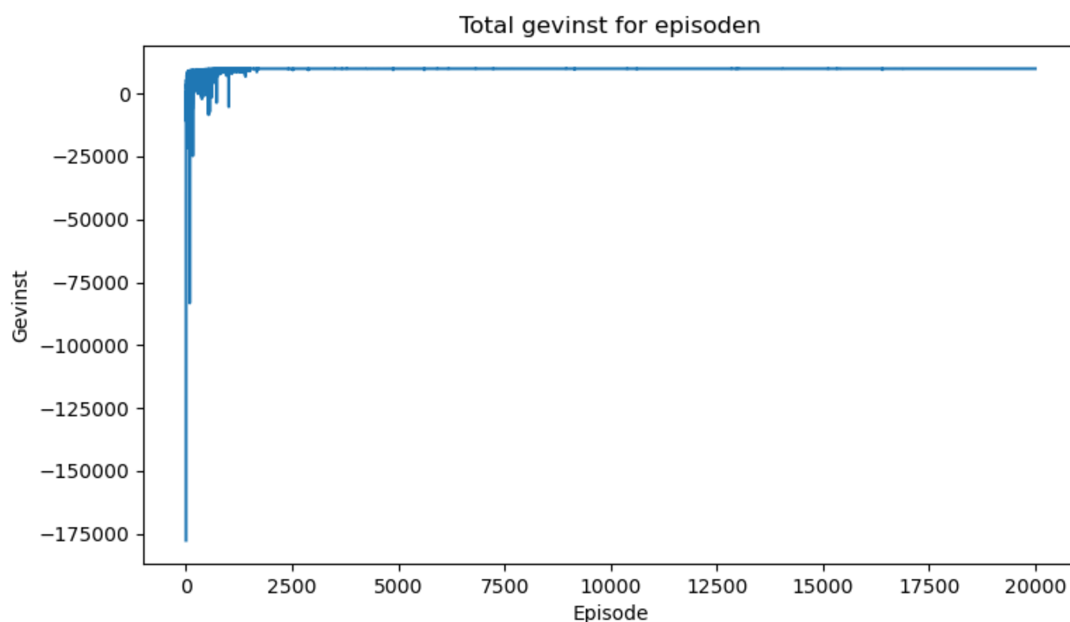
For å sikre at ruteplanleggeren gir en sikker og effektiv rute, er det en forutsetning at rutekartet representer det fysiske miljøet så godt som mulig. For å få til dette er et rutekart som er høyoppløst nok til å kunne vise smale sund, farlige grunner og andre fysiske hindringer, avgjørende for rutevalget.

I figur 17, 18 og 19 vises eksempler på rutekart med en oppløsning på henholdsvis 90x90, 150x150 og 60x60 ruter. Ved å teste en rute med samme start- og sluttilstand i de tre forskjellige oppløsningene, ser man oppløsningens betydning for tidsbruken til algoritmen.



Figur 19 Gevinstkart 60x60 piksler.

Strandvold og Leines har ved hjelp av regresjonsanalyse funnet en funksjon som viser at antall episoder for konvergering er lineært med antall mulige tilstander. De kom fram til en funksjon $f(x) = 18{,}66x + 5932{,}01$ for deres miljø, noe som betyr at det blir rett i underkant av 19 nye episoder per ekstra rute (Strandvold & Leines, 2018, s. 46). Denne oppgaven kjører derimot det samme antallet episoder for treningen til de forskjellige oppløsningene. Dette er fordi deres funksjon gir et unødvendig høyt antall episoder for høyere oppløsninger, der for eksempel et rutekart på 150x150 piksler gir totalt antall episoder på rundt 425 000.



Figur 20 Total gevinst per episode. Ser den konvergerer rundt 2000 episoder.

Figur 20 viser hvordan algoritmen konvergerer mye tidligere enn funksjonen til Strandvold og Leines tilsier. Den konvergerer rett rundt 2 000 episoder, men for enkelthetens skyld ble det faste tallet på antall episoder algoritmen skal kjøre, satt til 20 000. Dette var for å legge til rette for større, mer høyoppløste og mer kompliserte miljøer som krevde flere episoder.

Tabell 2 viser hvordan tidsbruken øker betydelig med oppløsningen til rutekartet, ettersom antallet strategier som må prøves ut vil øke tilsvarende. Hver gang man dobler antall piksler på hver akse, firedobles antallet tilstander i hvert rutekart.

Rutekartets oppløsning	Antall mulige handlinger	Tid brukt [sekunder]
60x60	14 400	17
90x90	32 400	29
150x150	90 000	82

Tabell 2 Rutekart og tid brukt for å beregne den samme ruten med forskjellige oppløsninger.

For antall episoder trent, er det også viktig å legge til en sikkerhetsmargin ved å runde opp antallet episoder som kjøres. Dette gjøres for å sikre at beste løsning blir funnet med sikkerhet, selv om det kan være usikkerhet knyttet til hvor nøyaktig verdifunksjonen har konvergert. På denne måten kan man unngå at løsningen ikke er optimal, til tross for at det kan være en viss usikkerhet knyttet til hvor mange episoder som er nødvendig.

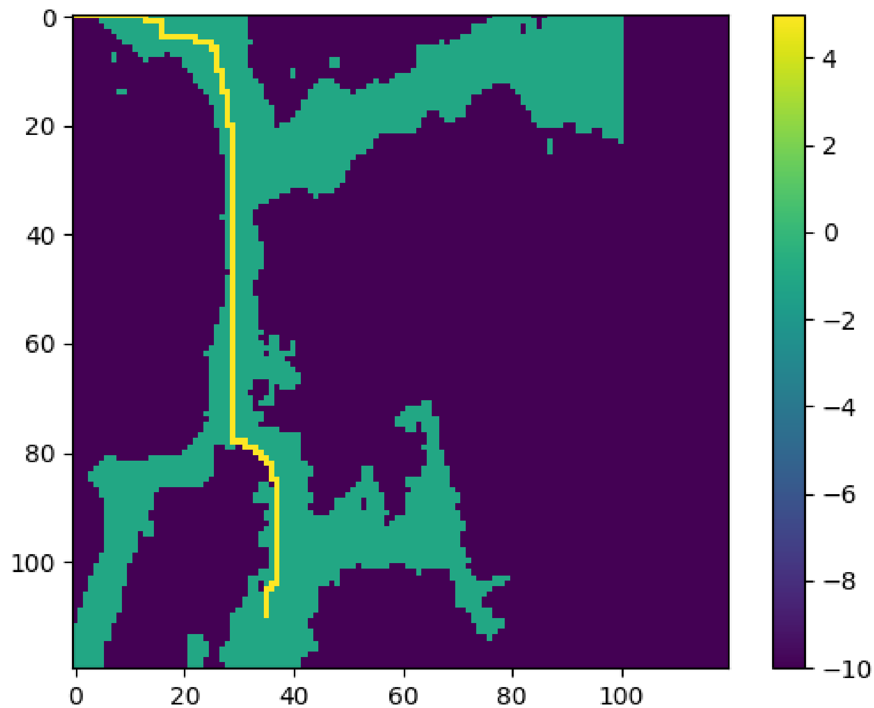
4.2 Trening og implementering av RL-algoritmen

Dette delkapittelet går nærmere inn på de ulike resultatene gjennom algoritmens utvikling. Delkapittelet vil ikke fremlegge de faktiske variablene og kodeteksten som ble endret på, men vil forklare hva som ble endret maskinlæringsmessig, samt i algoritmen fra forrige resultat. Tabell 3 viser de forskjellige parameterne som er viktige for hver mark av ruten. Her kan man også se forskjeller ved hver enkel rute.

Rute	***Varierende parametere***
Rute mk1	- Låst startpunkt, starter i (0,0). - Hvis dybden i en piksel er dypere enn 10 meter anser rute mk1 den som traverserbar. - Manuell setting av målposisjon
Rute mk2	- Låst startpunkt, starter i (0,0). - Diskret dybdeverdier. Piksler med dybde mellom 0-10 meter gis en negativ gevinst, dybde mellom 10-30 en annen og dybde over 30 meter gis ingen negativ gevinst.
Rute mk3	- Flyttbart startpunkt, kan starte hvor en vil. - Dybden er dynamisk, det gis negativ gevinst i piksler med hensyn på dybde i henhold til en dynamisk kostnadsformel.

Tabell 3 De forskjellige rutene

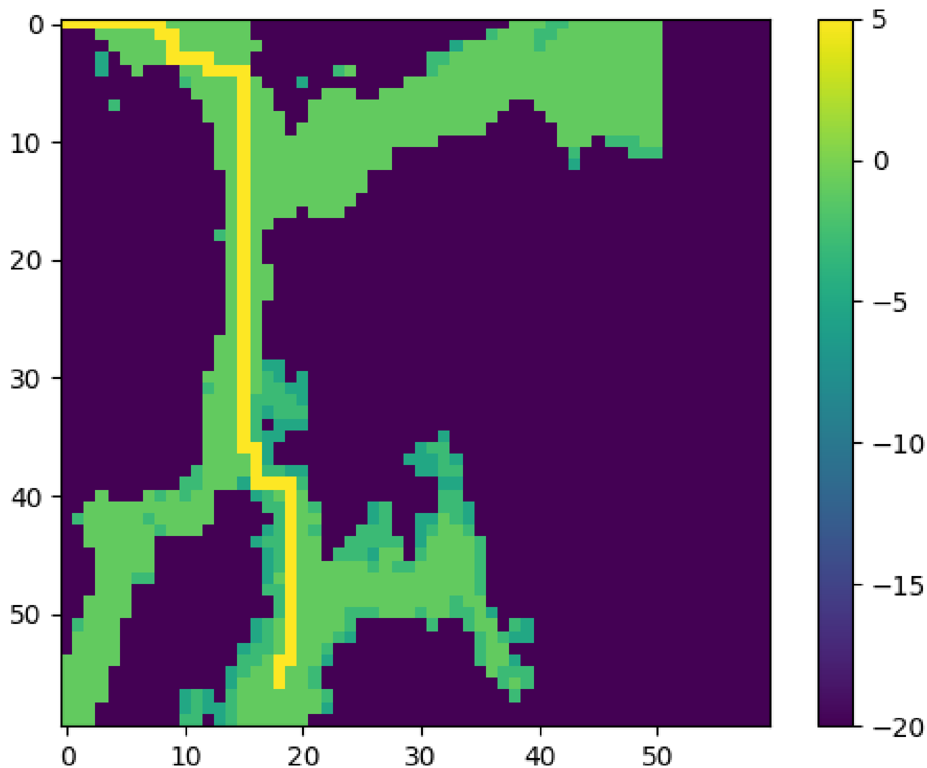
4.2.1 Rute mk1



Figur 21 Rute mk1, legg merke til hvordan den tar raskeste rute langs med land.

Figur 21 viser den første ruten koden klarte å tegne. Ruten ligger i et 120x120 piksler stort gevinst-kart. I denne programversjonen er startpunktet låst til punkt (0,0), som er øverst til venstre i figuren, og regner seg frem til mål som her er i posisjon (35,110). Rute mk1 tar ikke hensyn til dybder dypere enn 10 meter, og ruten tar derfor raskeste vei, uavhengig av om det er nærme land eller midt i leden.

4.2.2 Rute mk2



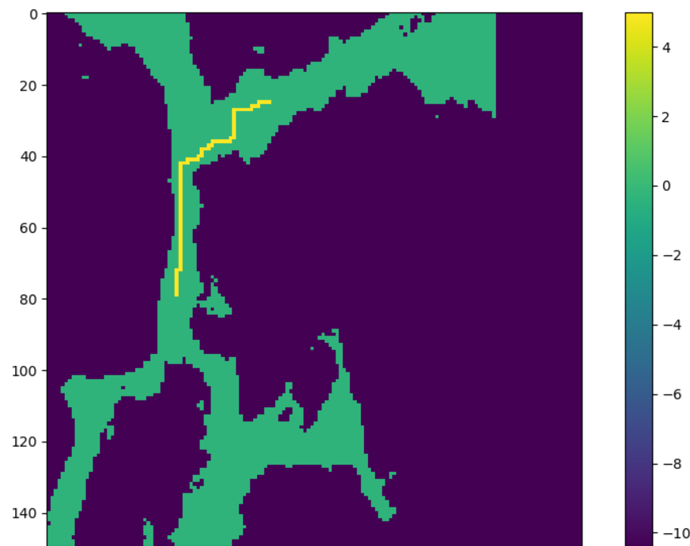
Figur 22 Rute mk2, legg merke til de større kostnadene der det er grunnere farvann.

Figur 22 består av et gevinstkart på 60x60 piksler. Det kommer frem på oppløsningen sammenlignet med rute mk1 i figur 21. Ruten her er derimot påvirket av en ekstra parameter. Det er lagt inn ekstrakostnader for grunt farvann, som man kan se i de grå og mørkegrønne pikslene nært landområdene. Her gis ruten en ekstrakostnad for å reise over, noe som gir insentivet til å heller kjøre nærmere midten av leden for å komme seg til mål. Det er også tatt hensyn til hvor stor denne kostnaden skal være, slik at den ikke prioriterer å gå 15 ruter i motsatt vei for å finne en ny vei til mål, og således blir stående fast uten å fullføre iterasjonen. Altså at den ikke kommer til mål.

Ekstrakostnaden for dybder blir satt ved å gi alle dybder mellom 0 og 10 meter en gevinst på -5 for en handling gjennom pikselen. Tilsvarende fikk dybdene mellom 10 og 30 meter en gevinst på -3. Piksler med samlet dybde over 30m fikk ingen negativ gevinst, som førte til at disse pikslene ikke hadde en tilleggskostnad, og heller ble foretrukket.

Å gjøre det på denne måten gjorde at programmet slet med å øke skaleringen. Ved et 90x90 piksler gevinst-kart, eller et 120x120 piksler gevinst-kart, klarte ikke algoritmen å kjøre iterasjoner, fordi den aldri kom til mål. Dette problemet blir belyst i delkapittel 5.1.2.

4.2.3 Rute mk3

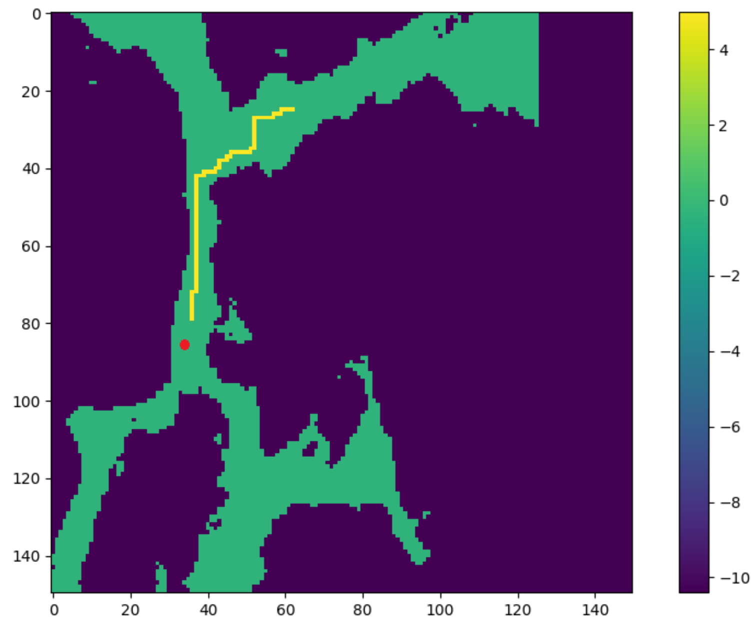


Figur 23 Rute mk3, her i et høyoppløst rutekart.

Figur 23 viser en rute i et gevinstkart med høy oppløsning. Her er oppløsningen 150x150 piksler. I rute mk3 kan programmet endre start og sluttposisjon, ulik de tidligere rutene som bare kunne endre målposisjon. Det er satt en ny startposisjon for å vise at dette funker. Her starter ruten nord i kartet og traverserer mot sør der målet er. Denne ruten er ganske forskjellig fra rute mk2 og fungerer på en annen måte. Her settes ikke gevinster i alle pikslene bare ved hjelp av dybde, men det legges til en negativ steg-gevinst i hver piksel som avhenger av dybden. Denne steg-gevinsten er dynamisk ettersom den ganges med en kostnadsformel med dybden som en variabel. Denne ruteplanleggeren er derfor mer komplisert, men også mer dynamisk. Således kan oppløsningen endres til å bli bedre, noe som ikke gikk i rute mk2.

I gevinstkartet til denne ruten er det ingen synlige grå områder som tilsvarer grunne farvann. Disse områdene er fortsatt til stede, men kostnaden er såpass liten at forskjellen kun blir synlig ved å plassere musepekeren over pikslene. Dette bringer oss videre til figur 24.

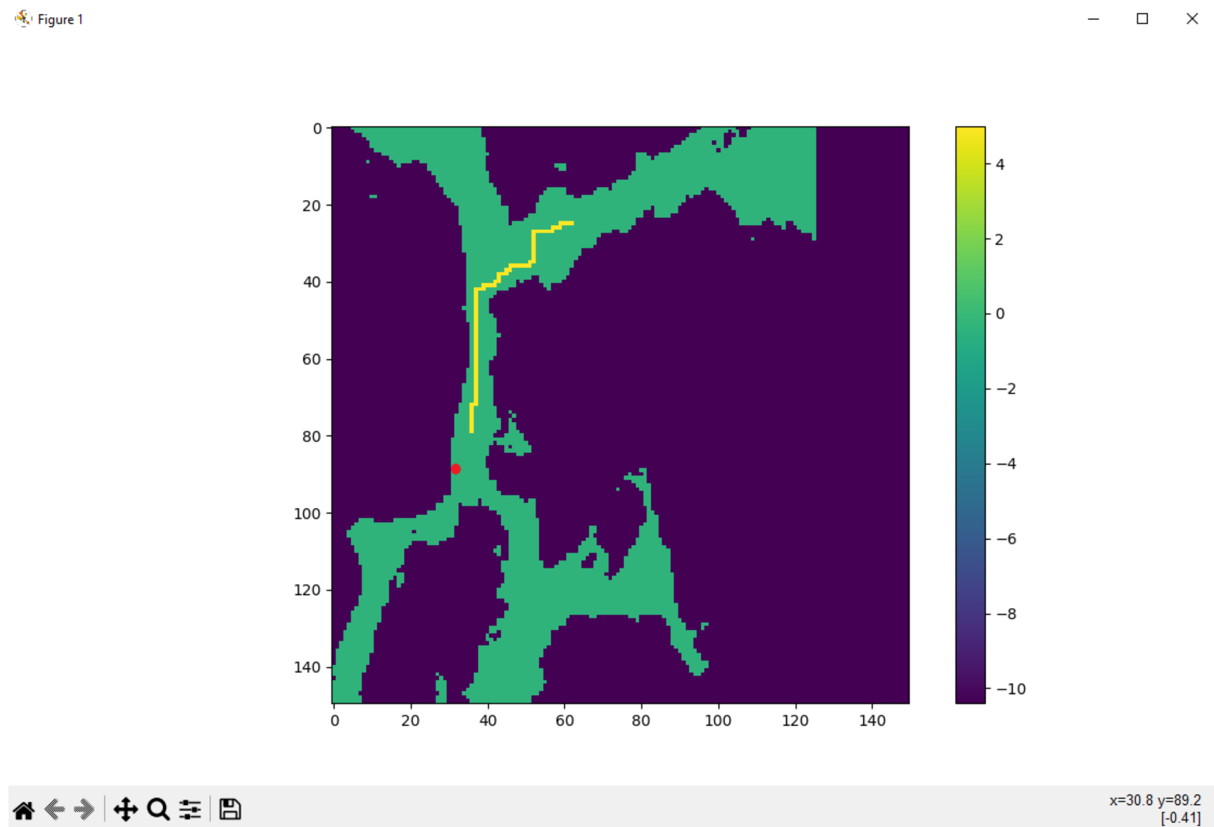
Figure 1



x=33.3 y=82.7
[-0.40]

Figur 24 Gevinsten vises nederst i høyre hjørnet.

Figur 24 er helt lik figur 23, foruten den røde prikken i kartet som representerer musepekeren. Nederst til høyre i figur 24 vises x- og y-koordinatene relativ til figuren, samt en skalar under. Denne skalaren representerer kostnaden for å traversere gjennom akkurat denne pikselen. Der den røde prikken er nå, er det dypt og den befinner seg et godt stykke fra land. Derfor er denne skalaren satt til verdi $-0{,}40$. Formelen algoritmen brukte for å komme frem til $-0{,}40$, kan ses i kildekoden. $-0{,}40$ er viktig å merke seg til neste figur, for å legge merke til forskjellen mellom farvann nært og et stykke fra land.

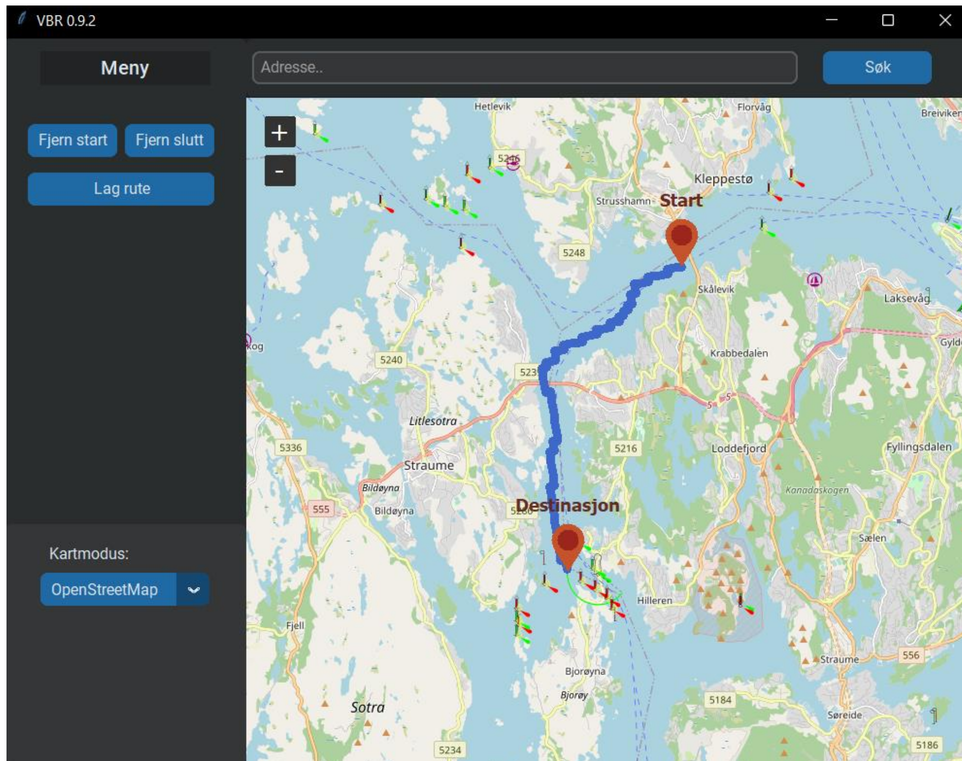


Figur 25 Legg merke til gevinsten på -0,41 nede i hjørnet til høyre.

I figur 25 er markøren flyttet nærmere land, dette kan man også se ved å sammenligne x- og y-koordinatene til figur 25 med figur 24. Det som er viktig å få frem, er at her er skalaren som representerer kostnaden for pikselen lik $-0{,}41$. Derav er det en forskjell på $-0{,}01$ mellom dypt vann og grunt vann i gevinstkartet. Dette utgjør en forskjell for beregningen av ruten, og algoritmen vil oppnå størst gevinst ved å holde seg unna landområder. Ruten i figur 25 synliggjør dette. Istedenfor å gå den korteste veien fra nord til sør, helt inntil land der sundet blir smalere, beregnes ruten i god avstand til land.

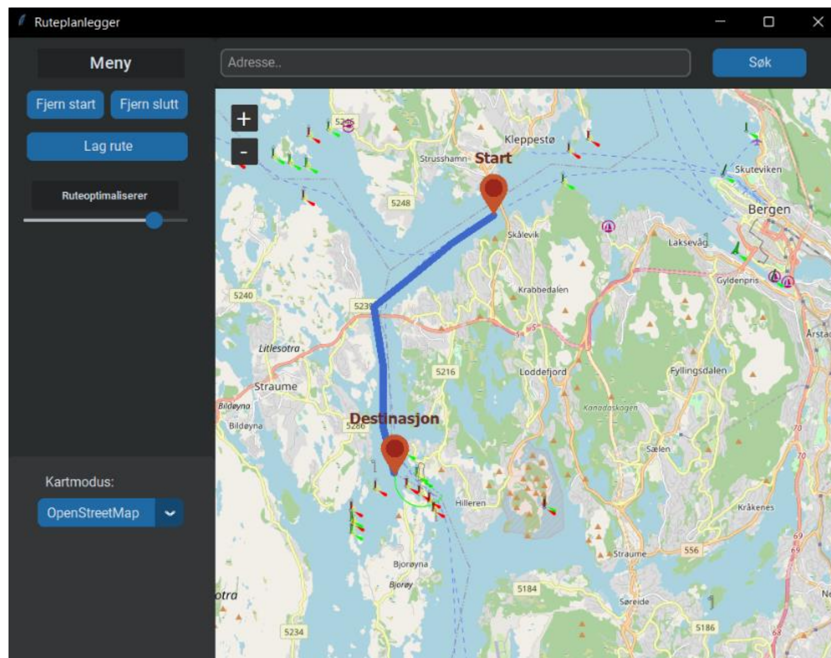
4.3 Brukergrensesnitt

Dette delkapittelet viser testene som er gjort av brukergrensesnittet. Testene viste umiddelbart nødvendigheten av noen nye funksjoner. Disse funksjonene blir gjort rede for i dette delkapittelet.



Figur 26 Eksempel av første rute

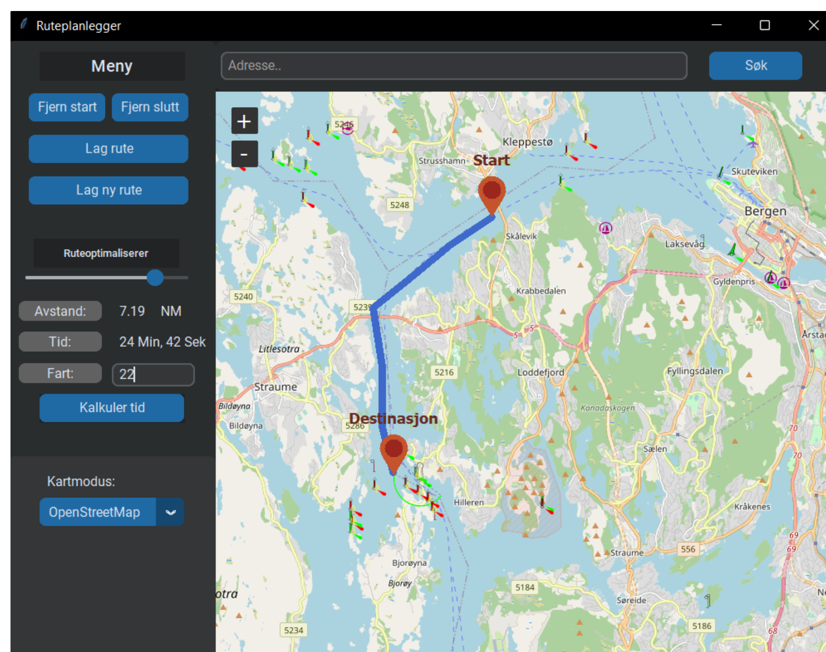
Figur 26 viser eksempel på hvordan de første rutene så ut i brukergrensesnittet. Den sekvensielle bevegelsen til algoritmen gjør ruten svingete, samt at den vil gi en distanse mye lenger enn den reelle avstanden mellom start og slutt. Dette er uhensiktsmessig og vil ikke være til noen nytte. For å løse dette er det lagt til en funksjon som fjerner et antall av punktene algoritmen har bevegd seg gjennom. Funksjonen fungerer med en glidebryter i brukergrensesnittet og gjør at brukeren selv kan velge hvor mange punkter som skal fjernes. På denne måten kan brukeren velge hvordan ruten skal se ut fra behov.



Figur 27 Ruteplanleggeren med glidebryterfunksjonen. Her bruker brukergrensesnittet kun hvert 16. punkt fra algoritmen.

Figur 27 viser implementeringen av glidebryterfunksjonen. Her er det ingen antydning til den sekvensielle bevegelsen til maskinlæringsalgoritmen. Dette gjør at ruten er mer realistisk og at distansen fra start til slutt i større grad samsvarer med virkeligheten.

Implementeringen av denne funksjonen gjør det mulig å bruke distansen til å måle hvor lang tid det vil ta å seile ruten. For at brukeren skal kunne nytte seg av dette er det også lagt til en vei/fart/tid-kalkulator. Denne krever kun at brukeren nytter seg av glidebryterfunksjonen og legger inn fart.



Figur 28 Det endelige brukergrensesnittet, her med vei/fart/tid-kalkulator

4.4 Oppsummering av resultater

Gjennom kapittel 4 skjer det en gradvis utvikling av ruteplanleggeren på alle områder. Ruteplanleggeren blir tildelt flere funksjoner i brukergrensesnittet og blir mer komplisert ved at parameterne forbedres gjennom de ulike versjonene. Presenteringen av forskjellig tidsbruk ved forskjellig oppløsning på rutekartet, er viktig når det nå gås inn på drøftingen av resultatene. Disse blir også særs aktuelle når det diskuteres hvordan maskinlæring kan brukes som hjelpemiddel til ruteplanlegging i Sjøforsvaret.

5 Drøfting

I dette kapittelet vil oppgaven ta for seg hva som har blitt gjort, og det systemet som har blitt konstruert. Hvorfor resultatet i testingen har blitt som de er, og hvorfor systemet har de egenskapene som er beskrevet, er noe det kommer til å ses nærmere på. Problemene som ble møtt underveis vil også diskuteres, for å skape et godt grunnlag for forståelse og for videre arbeid med produktet. Konseptet og løsningen oppgaven har kommet frem til vil også drøftes, for å konkludere best mulig mot slutten av oppgaven.

5.1 Under utvikling

I dette delkapittelet vil det ses nærmere på testingen og resultatene i kapittel 4.

5.1.1 Datafremstillingen

Rådataene brukt gjennom denne oppgaven er kartdata fra Kartverket. I xyz-format leverte Kartverket ut data for havarealet i området markert ut, noe som resulterte i at man kun hadde data der algoritmen i utgangspunktet kunne kjøre. Den første utfordringen ble å gi dataene rundt havområdet en verdi, altså gi landområdene rundt sjøen en verdi slik at programmet klarte å skille mellom land og vann.

Ved å ta utgangspunkt i figur 15 i del-kapittel 4.1.2, der man ser dataene plottet for x- og y-koordinatene, ble dette ordnet. Slik blir dybden og z-aksen noe man ikke ser rent fysisk. Dette gjorde det enkelt å sette inn en verdi lik én, for alle områdene i rutekartet uten en dybdeverdi, ettersom dette er land. Det var enklest å jobbe med kvadratiske datafremstillinger, noe av koden brukt baserte seg på å regne ut likt for dataene i y og x. Dette passet bra og påvirket ikke de dataene som allerede eksisterte, fordi for hver x-koordinat var det også en tilsvarende y-koordinat.

Løsningen med dynamisk testing uten å endre på datasettet og skalarene, har vært nyttig fordi det gir ruteplanleggeren en mulighet til å bli forbedret på samme området. Ved endring av datasettet vil man kanskje kunne få problemer som ikke har noe med algoritmen å gjøre, men med korrupte datasett. En løsning for å muliggjøre et konseptbevis har blitt fokusert på, for at oppgaven skal svare best mulig på problemstillingen.

Å bruke snittverdien i hver piksel for å representere dybden, kan derimot være problematisk i enkelte tilfeller der pikselen ligger over et område som både har vann og land. Pikselens verdi blir således regnet ut fra kanskje 10 meter dybde, altså -10 meter, i halvparten av pikselen og positiv 5 meters dybde, der dybden i hele pikselen blir representert med -10 + 5 = -5 / 2 = -2.5 meter dybde.

For piksler med både høye landområder og grunne vannområder, ville det vært fordelaktig med en «ceiling»-funksjon, en funksjon som setter cellen til den høyeste verdien. For lave høyder i landområder med veldig dyp dybde i vannareal rundt, kunne det vært bedre med en «floor»-funksjon. En «floor»-funksjon tar utgangspunkt i den laveste verdien i pikselen, og setter denne for hele pikselen. Dette er gunstig fordi det å seile inntil land, ikke er problematisk så lenge det er dypt nok frem til synlig land. Det mest optimale for at alle pikslene i rutekartet skal samsvare best med hverandre, hadde vært å regne ut hver piksels snittdybde, men da med et uendelig antall piksler. Dette er noe som ikke er fysisk mulig, og det ville ført til at beregningstid for algoritmen hadde blitt veldig stor. Dette er fordi det er antall piksler som sier hvor mange muligheter algoritmen har.

5.1.2 RL-algoritmen

Ruten som er kommet frem til i 4.2.3 Rute mk3, representerer den mest komplekse ruten som kan bli anvendt i brukergrensesnittet. Det er en rute som gir raskest mulig vei fra punkt A til punkt B, med hensyn på ulik dybde og tilstrekkelig avstand til land. Det er flere grunner til hvorfor ruten skal holde seg unna landområder. Den første er at man skal unngå å seile nært land, ettersom det er en økt risiko for å grunnstøte. Den er kanskje selvsagt for oss mennesker, men den er ikke selvsagt for en ruteplanlegger som planlegger for et perfekt seilas. Med perfekt seilas menes et seilas som følger ruten nøyaktig, og ikke viker for vær og vind, samt andre fartøy. Når det er mennesker som skal seile strekningen, er det fordelaktig med god margin til landområder. Dette gir også et større mulighetsrom for å svinge med skipet og gjøre unnvikelsesmanøvrer. Det gir også rom for å løse problemer som kan oppstå underveis, for eksempel at maskineriet i båten slutter å fungere, eller at roret blir ødelagt. Dersom en situasjon skulle oppstå, har man bedre tid på å agere før man eventuelt går på land.

En annen tanke er at ruten skal passe til alle Sjøforsvarets fartøyer. Hadde ruten gått helt inntil land hadde det kanskje ikke vært et problem for korvettene eller stridsbåtene våre, men det hadde blitt et problem for større fartøy som fregatter og KNM Maud. Derfor er det fordelaktig at ruten kan brukes av samtlige fartøyer. Det som hadde vært interessant å se på i et videre arbeid, er å tilpasse kostnadsfunksjonen og ruten til hvert enkelt fartøy. Korvettene kan for eksempel få ruter som ikke fregattene kan seile, men som er mer optimale for korvettens taktiske bruksområder. Mer om dette i delkapittel 6.5.6.

Et problem oppstod hvis det var ubalanse mellom kostnad og gevinst i miljøet til agenten. Dersom agenten fikk en for stor gevinst for en handling til en gitt tilstand, kunne agenten gå frem og tilbake i de samme tilstandene. Siden agenten kontinuerlig søker etter en størst mulig gevinst, kan den estimere at den høyeste gevinsten oppnås ved å gjentatte ganger gå til den samme tilstanden. Konsekvensen ble at agenten «gikk seg fast», og ikke fullførte iterasjonen. Derfor var det viktig å ha dynamiske kostnader og gevinster, slik at man unngår at agenten «går fast».

5.1.2.1 Tidsbruk

Et viktig moment for ruteplanleggeren er hvor lang tid den bruker på å regne ut den optimale ruten. Tidsbruken for algoritmens utregning har betydning for hvor raskt programmet kan gi et resultat, dermed hvor effektivt det kan brukes i praktiske situasjoner. Jo raskere programmet kan utføre utregningene, desto mer nyttig kan det være for navigatørene som trenger å planlegge ruter på kort tid. Det kan også ha betydning for hvor godt programmet kan lære og forbedre seg over tid, da det vil kunne gjøre flere utregninger i løpet av en kortere tidsperiode. Er RL-algoritmen for treg vil programmet oppleves upraktisk for brukeren. Videre kan et tregt system også begrense bruksområdene til programmet, slik at man for eksempel kun kan bruke det i forkant av seilas, og ikke underveis.

Antall episoder har en direkte innvirkning på tidsbruken til algoritmen. Desto flere episoder algoritmen har, desto mer tid vil det ta å gjennomføre læringsprosessen. Dette skyldes at algoritmen må utføre flere handlinger og interagere med miljøet flere ganger. Dette for å tilegne seg nok erfaring slik at den kan tildele nøyaktige verdier til handlinger i miljøet.

I oppgaven til Strandvold og Leines forklarer de hvordan de i testene fant at antall episoder for konvergens kunne beregnes, gitt at rutekartet og kompleksiteten i miljøet var kjent (Strandvold & Leines, 2018, s. 55). Imidlertid er det en risiko for at treningen kan avsluttes for tidlig dersom det estimerte antallet episoder viser seg å være for lite. Derfor kan det være lurt å legge inn en sikkerhetsmargin på antall episoder, slik at treningen fortsetter litt lengre enn det estimerte antallet, for å sikre at verdifunksjonen har konvergert. En annen løsning kan være å implementere metoder for å kontrollere at verdifunksjonen har konvergert, slik at treningen avsluttes når dette er bekreftet. På denne måten kan man unngå å trene videre etter at treningen har konvergert, uten å risikere at treningen avsluttes for tidlig.

Det er sett på muligheten for å lagre og huske estimatene fra verdiligningene til algoritmen. Dette ville ført til at tidsbruken for miljøer der agenten allerede hadde trent, minket betraktelig. Dersom algoritmen derimot møtte et nytt ukjent miljø, ville den brukt full tid ettersom den måtte ha regnet ut alle verdiene. Samtidig kan det også være en risiko ved å bruke tidligere estimater i et gjenkjent miljø. Hvis estimatene hindrer utforskningen og får agenten til å holde seg til de gamle strategiene, kan dette føre til at agenten ikke lærer å tilpasse seg endringer i miljøet, og kan dermed miste sin evne til å navigere effektivt. Derfor må man være forsiktig med å bruke tidligere estimater i et kjent miljø, og sørge for at agenten fortsetter å utforske og lære for å holde seg oppdatert.

En siste faktor for tidsbruken til algoritmen, er den tilgjengelige datakraften. Den har en direkte innvirkning på tidsbruken, ved at desto mer datakraft som er tilgjengelig, desto raskere vil algoritmen

kunne utføre en regneoperasjon. Dette betyr at det overordnede systemet som denne løsningen skal implementeres i, bør ha god datakraft tilgjengelig for å sikre at treningen går så raskt som mulig.

Det er også verdt å nevne at en effektiv programmering kan bidra til å optimalisere tidsbruken til algoritmen. Gjennom god strukturering og organisering av koden, samt valg av effektive datastrukturer, kan man sørge for at treningen går så raskt som mulig uten å ofre noe på nøyaktigheten. På denne måten kan man få mest mulig ut av datakraften som er tilgjengelig.

5.1.2.2 Oppløsning

Et annet område som har en direkte innvirkning på hastigheten til algoritmen, er oppløsningen til rutekartet. Særlig blir dette et problem med større rutekart, ettersom tiden øker eksponentielt med størrelsen på rutekartet, som vist i kap. 4.1.5. En mulig løsning for dette kan være å implementere flere lag for beslutningstaking i ruteplanleggeren. Ved å bruke flere lag kan ruteplanleggeren dele opp treningen i mindre deler og dermed redusere tidsbruken. For eksempel kan det første laget brukes til å trene på et mindre oppløst rutenett for å lære grunnleggende strategier over store avstander. Det andre laget kan brukes til å trene på et høyoppløst rutenett over et mindre område, for å lære mer avanserte strategier. På denne måten kan ruteplanleggeren lære på en mer effektiv måte, og dermed unngå at treningen tar for lang tid.

Oppløsningen til rutekartet er avgjørende for å kunne konstruere et miljø som best mulig representerer det fysiske miljøet. Dersom fokuset ligger for mye på hastighet og enkelhet, og man velger en lav oppløsning på for eksempel 10x10, kan dette føre til at enkelte leder og passasjer blir utelukket. Enda verre vil det være hvis farer som skjær, holmer og andre hindringer «forsvinner» i skaleringen. Desto høyere oppløsning et rutekart har, desto mer detaljert vil det være og desto bedre vil det gjenspeile det fysiske miljøet. Ettersom høy oppløsning også er mer ressurskrevende å generere og krever mer minne for å lagres, må det gjøres en vurdering mellom nøyaktighet og hastighet.

Det er viktig å huske på at et rutekart bare er et verktøy for å representere det fysiske miljøet, og at det ikke kan erstatte det fysiske miljøet selv. Det er derfor viktig å vurdere hva som er viktigst for formålet med rutekartet og velge en oppløsning som passer best for dette. Hvis formålet er å navigere i et område med mange små detaljer, som skjær og holmer, kan det være lurt å velge en høy oppløsning slik at disse detaljene ikke går tapt. Hvis formålet derimot er å navigere i et større område med færre detaljer, kan en lavere oppløsning være tilstrekkelig.

5.1.3 Graphical User Interface

Brukergrensesnittet passer godt med hva oppgaven behøver. Brukeren kan enkelt sette ut nøyaktige markører og deretter fjerne dem. Det er praktisk å navigere rundt i kartet og brukergrensesnittet har nyttige funksjoner som søk, vei/fart/tid-kalkulator og en funksjon som gjør brukeren i stand til å endre antall punkter i ruten. Brukergrensesnittet gir brukeren de verktøyene som trengs for å bruke maskinlæringsalgoritmen på en effektiv måte. Likevel er ikke programmet perfekt og det er ingen måte for navigatørene å bruke ruten til annet enn å finne raskeste led, avstand og tidsbruk. Med sjøkartsfilteret kan man se lykter, skjær og andre sjømerker, men programmet bruker det ikke til noe. Dette hadde kunnet blitt brukt ved at programmet fikk inn flere parametere som representerte de ulike lyktene, skjærene og sjømerkene. Ved en funksjon som tolker disse og gir en ekstra gevinst/kostnad i hver piksel hvor disse er navigasjonsrelevante, ville programmet kunne anvende sjøkartsfilteret. Siden dette ikke er tilfelle, gjør det at programmet egentlig ikke har den nytteverdien den potensielt kan ha.

For å ta det til neste steg, er en idé å legge til informasjon på hver strekning i brukergrensesnittet. Hver strekning kan inneholde informasjon om hvilken kurs man skal holde, hvor lang tid man skal bruke i en gitt fart per kurs og hvor mange grader man skal svinge for å komme på neste kurs. På denne måten vil brukeren i større grad kunne bruke ruteplanleggeren til å navigere. En annen idé er å legge til GPS. Med GPS vil ruteplanleggeren kunne gi tilbakemeldinger på hvor du befinner deg i ruten. Den kan også måle fart og dermed gi kontinuerlig tilbakemelding på hvor langt og lenge det er til neste sving. Man kan også legge til informasjon om hvilke lykter man passerer og hvilke av de som kan brukes til navigering. På denne måten kan man utnytte potensialet i ruteplanleggeren og navigatørene kunne fått mer utbytte av å bruke programmet.

5.2 Konseptuell idé

I dette kapittelet drøftes det hvordan løsningen på sikt kan bli brukt i Sjøforsvaret, og hvordan dette kan bli implementert ved en eventuell ferdigstilling av ruteplanleggeren. Dette kapittelet går mot det man kaller et konseptbevis.

5.2.1 Sjøforsvarets nytteverdi

Gjennom tre og et halvt år på sjøkrigsskolen er det observert hva navigatørene i kullet vårt bruker tid på. En arbeidsoppgave som tar mye tid av deres hverdag, er ruteplanlegging. Før hvert seilas de har på skolen planlegger de rutene de skal seile i detalj. Hvor de skal turne, hva de skal peile på, hvor mye de skal turne og andre ting som er relevant når man skal seile et fartøy. Navigatørene om bord på Sjøforsvarets fartøy må også planlegge ruter i detalj før de skal seiles. Å få inn en ruteplanlegger

som kan gjøre dette for dem, i alle fall kunne lage et utgangspunkt for en rute de lager selv, kan potensielt spare mye tid og frigjøre tid til andre oppgaver ombord.

Så er spørsmålet, kan en ruteplanlegger erstatte ruteplanleggingsegenskapene som en offiser fra Sjøkrigsskolen har opparbeidet seg gjennom tre år med skolegang? Ruteplanleggeren det er kommet frem til er et førsteutkast for hva som er mulig å få til med maskinlæring. Med videre utvikling av algoritmen, vil man kunne legge til flere parametere som kan øke kompleksiteten i ruten og maskinlæringen, samt gi et mer spesifikt ønske om hva det er man vil få til. En parameter kunne for eksempel vært; i hvor stor grad ønsker man å holde seg nært land? Parameterne kan i bunn og grunn gjøre store endringer i hvordan ruten kommer til å se ut, altså hvordan algoritmen setter kostnad i hver piksel. En positiv konsekvens av dette vil være at man får flere parametere man kan tilpasse til militær navigasjon. Navigatøren kan legge inn hva hen vil ruteplanleggeren skal ta hensyn til, og hva den skal nedprioritere. Militær navigasjon er ikke lik sivil navigasjon. Man skal ofte ta hensyn til taktiske perspektiver, og ikke bare komme seg kjappest mulig fra A til B. Militær navigasjon kan også inneha andre forutsetninger enn det et sivilt skip har, som for eksempel undervannsbåter, som navigerer på en helt annen måte enn det overflateskip gjør. Her kunne det vært gjort tilpasninger i ruteplanleggeren slik at parameterne tok hensyn til at du var under vann.

Implementering av en ruteplanlegger-løsning i Sjøforsvaret er en prosess som kan være omfattende og tidskrevende. Dette fordi det er et system som fungerer i dag, og endringer er alltid vanskelige og tøffe for folk og utstyr å tilpasse seg til. Dette hadde påvirket direkte hvordan navigatørene jobber, og ville vært en løsning som det hadde vært lurt å lære navigatørene å bruke så tidlig som mulig. I tillegg til innføringen av det faktiske systemet som ruteplanleggeren skal fungere på, er det nyttig med kursing av Sjøforsvarets navigatører i ruteplanleggeren. Tilsvarende kunne det vært aktuelt å innføre et kurs på Sjøkrigsskolen som gjorde kadetter kjent med ruteplanleggeren.

Oppgavens ruteplanlegger, med hensyn til det diskutert tidligere i delkapittelet, kan bidra til å helhetlig forbedre bro-teamet i en besetning. Sjøforsvaret kan derfor ha nytte av å implementere ruteplanleggeren.

5.2.2 Live-oppdatering

Programmet vårt har ingen form for live-oppdatering av ruten mens man seiler. Programmet kan dermed ikke ta hensyn til andre fartøy rundt seg underveis. Den har heller ingen måte å bruke GPS eller andre metoder for å oppdatere egen posisjon etter hvert som man seiler ruten. Dette fører til at ruteplanleggeren er statisk og ikke gjør tilpasninger i ruten automatisk underveis i transitt. Ruteplanleggeren kan derfor kun brukes som et planleggingsverktøy i forkant. Dette er en idé som

ses på som svært relevant å bygge på videre på. Live-oppdatering og samspill med GPS og AIS-posisjoner til andre fartøyer vil nok være neste steg i utviklingen av en ruteplanlegger som Sjøforsvaret kan implementere. Med GPS og AIS vil ruteplanleggeren hele tiden kunne ta høyde for egen posisjon, men også andre fartøy som interferer med ruten. Her kan man også legge inn data fra radar som kan plukke opp andre objekter som ikke bruker AIS, som for eksempel mindre båter og skjær. Mer om dette i delkapittel 6.5.7.

5.3 Begrensninger ved maskinlæring

En av de viktigste begrensningene man har når man lager en ruteplanlegger ved hjelp av maskinlæring, er at algoritmen kun er så god som det miljøet den får trent på. Maskinlæring krever store datamengder for å kunne trene effektive algoritmer. Dersom dataene ikke er tilfredsstillende relevante eller korrekte, kan dette føre til at algoritmene og løsningene ikke fungerer som ønskelig. Det kan også være utfordringer knyttet til tilgjengelighet av kartdata og kvalitet på treningen i miljøet, som kan begrense hva man kan gjøre med maskinlæring. Hvis algoritmen ikke får gjennomført nok iterasjoner, vil ruten den produserer ikke være god nok.

En annen begrensning er at maskinlæring ikke alltid kan ta hensyn til alle relevante faktorer og variabler som kan påvirke seilingsruter. Maskinlæring er god til å analysere og simulere data, men det kan være ulike faktorer som ikke er inkludert i dataene, eller som er vanskelige å måle eller kvantifisere. Dette kan føre til at algoritmene og løsningene gir ufullstendige eller upresise resultater, slik at man må supplere maskinlæring med andre metoder for å få en fullstendig oversikt over faktorer og variabler som kan påvirke seilingsruter.

En tredje begrensning er at maskinlæring kan være dyrt og komplekst å implementere og vedlikeholde. Utvikling og trening av avanserte algoritmer og teknologiske løsninger krever både tid og ressurser, og det kan være utfordrende å finne riktig balanse mellom *kostnader* og fordeler. Det kan også være at teknologien endres eller forbedres over tid, og at man må oppdatere og forbedre algoritmene og løsningene for å holde tritt med ny utvikling. Dette kan kreve ekstra innsats og ressurser, og det kan være utfordrende å sikre at maskinlæringen forblir effektiv og oppdatert.

6 Konklusjon og forslag til videre arbeid

6.1 Konklusjon

Denne oppgaven skulle svare på problemstillingen **«Hvordan kan maskinlæring brukes som hjelpemiddel til ruteplanlegging i Sjøforsvaret?».** Gjennom utviklingen av konseptet, tester og resultater, har oppgaven belyst at det er mange hindringer, men også muligheter, når det kommer til en ruteplanlegger i Sjøforsvaret. Resultatene fra de gjennomførte testene, viser at systemet har klart å oppfylle de ulike kravene som var satt, noe som tilsier at systemet klarer å gi en tilfredsstillende rute fra A til B, med god sikkerhetsavstand til land.

Kanskje en av de største utfordringene med en autonom maritim ruteplanlegger er å garantere sikkerhet og pålitelighet. Dette skyldes at havområdene kan være svært komplekse og dynamiske, med en rekke faktorer som kan påvirke sikkerheten og påliteligheten til ruteplanleggeren. Som diskutert i kapittel 5, vil oppløsningen av pikslene ha mye å si på hvor nøyaktig ruten som algoritmen kommer frem til er. At snittdybden i hver piksel er brukt som utgangspunkt, kan ha store konsekvenser for grunne punkter i pikselen. Dette vil i stor grad påvirke navigatørens evne til å navigere trygt og effektivt. Derfor må ruteplanleggeren i fremtiden være bedre i stand til å håndtere slike faktorer på en sikker og pålitelig måte for å sikre at fartøy navigerer trygt og unngår ulykker.

En konsekvens som følger med oppløsningen og økt nøyaktighet, er økt tidsbruk. Algoritmen vil med økt nøyaktighet og flere piksler i kartet, bruke mye lengre tid på å regne ut en tilfredsstillende rute. Det er derfor viktig å vurdere det fysiske miljøet samt formålet til ruten, for å velge en oppløsning som blir en tilfredsstillende avveining mellom tid og presisjon.

En identifisert faktor som er avgjørende for riktig tilpasning av ruten, er hvilke gevinster som skal gis de ulike dybdene i miljøet. Tidligere i delkapittel 5.1.2.1 ble problematikken med balansen mellom kostnad og gevinst belyst. Hvis balansen mellom kostnad og gevinst var dårlig, fikk det negative konsekvenser for ruten. Gevinster kan derfor ikke settes slik at agenten ikke vil gå over uønskede steg i miljøet, fordi dette vil føre til at agenten «går seg fast» når det er bare uønskede steg tilgjengelig. På samme måte kan ikke gevinster settes slik at forbudte områder fører til en for mild kostnad. Dette kan føre til at agenten tar «snarveier» over forbudte områder, ettersom dette vil gi en høyere total gevinst på sikt, og dermed bli den optimale strategien.

Løsningen med denne ruteplanleggeren er ikke tilfredsstillende og nøyaktig nok til å bli anvendt, men konseptet ses på som veldig aktuelt. Denne bacheloroppgaven har funnet at maskinlæring er et

nyttig verktøy som kan anvendes i ruteplanlegging, og at grunnlaget for morgendagens ruteplanlegging mest sannsynlig vil benytte seg av maskinlæring.

6.2 Forslag til påbygning og videre arbeid

Gjennom bacheloroppgaven er det brukt mye maskinlæring og det er forklart hvordan det kan anvendes. Det er også blitt oppdaget at mulighetene for hva det kan brukes til er uendelige. Verden går mot en fremtid fylt med teknologi og kunstig intelligens, noe som tilsier at datamaskiner blir smartere og smartere. Selv etter bacheloroppgaven er vi på et grunnleggende nivå når det kommer til maskinlæring. Men noen av mulighetene som er oppdaget, vil settes fokus på her.

6.2.1 Mineryddingsoperasjoner

Implementering av veipunkter i algoritmen, og ikke bare start og slutt for ruten, er noe det er sett på for algoritmen. Ved å si at ruten til algoritmen skal gå igjennom visse veipunkter, og at disse ikke er uunngåelige, vil algoritmen kunne brukes til å automatisere mineryddingsoperasjoner. Gevinsten til veipunktene kan baseres på sannsynligheten for at en mine befinner seg i området. Algoritmen kan dermed produsere en rute optimalisert for minejakt. Dette vil være en anvendelse av maskinlæring som kan frigjøre tid hos navigatørene i minevåpenet.

6.2.2 Search and Rescue

SAR, eller «Search and Rescue»-operasjoner, er noe algoritmen vår kan brukes til i fremtiden med implementering av veipunkter. Det vil også være relevant med koordinering av ferdiglagde ruter mellom flere forskjellige enheter, slik at ikke samme område sjekkes flere ganger. Ved hjelp av en parameter som sier hvor mange enheter man har tilgjengelig, kan ruteplanleggeren automatisk planlegge koordinerte ruter for eksempelvis ti enheter. Det kan effektivisere SAR generelt. Dersom det ikke er behov for skjerming/hemmelighold, kunne ruten også blitt publisert online slik at alle kan se hvor de andre har planlagt å kjøre. Dette bringer oss litt innpå neste område for påbygning.

6.2.3 Relativ oppdatering i forhold til andre fartøyer

Ruteplanleggeren kan legges inn i et nytt program, som koordinerer med GPS-posisjon på alle synlige fartøy, og legger inn en bevegelig kostnad for hver av disse fartøyene. Slik vil algoritmen unngå å kollidere med motgående, eller tregere farkoster, fordi den vil planlegge for å unngå dem. Dette vil si at programmet må ha en form for oppdatering underveis i ruten, slik at man kobler ruteplanleggeren opp mot egen posisjon.

6.2.4 Lagring av gamle ruter

Fartøy i Sjøforsvaret må ofte seile ruter de har seilt før. Enten de skal på vedlikehold, tilbake til basen, eller at et minefartøy skal søke over samme område på nytt. Hvis algoritmen brukes til å lage en slik rute, hadde det vært nyttig hvis ruten hadde kunne blitt lagret i en database. Ruten kan senere og ved behov lastes ned for å anvendes på nytt. Da trenger man ikke bruke tid på å lære opp algoritmen for hver gjennomkjøring, og man vil spare tid.

6.2.5 Legge til flere funksjoner i algoritmen

Akkurat nå regner algoritmen ut raskeste vei fra A til B med hensyn på dybde. Man kunne for eksempel ha lagt til en funksjon for å finne mest mulig kost-effektiv vei, med henholdsvis flere variabler som den tar hensyn til. Slik vil man kunne spare kostnader ved å bruke mindre drivstoff, ved å for eksempel seile medstrøms. Dette vil kreve tilpasning av selve algoritmen, siden den må legge til en gevinst eller fjerne en ekstra kostnad basert på andre parametere, som vær eller vind.

6.2.6 Tilpasse ruteplanleggeren til hver fartøysklasse

Hver fartøystype i Sjøforsvaret har ulike egenskaper og begrensninger basert på størrelse, tyngde og type skrog. Dette gir de forskjellige fartøyene ulike muligheter når det kommer til navigering. Ettersom det allerede eksisterer dybdedata fra havbunnen, kan det implementeres et brukervalg som bestemmer hvilket fartøy det skal lages rutes for. Denne kan ta hensyn til dypgang, lengde og bredde. Dette gjør at maskinlæringsalgoritmen kan lage en bedre tilpasset rute til hver spesifikk fartøysklasse.

6.2.7 Live-oppdatering i samspill med AIS, GPS, værmelding og radar

For å kunne lage en mer dynamisk ruteplanlegger som hele tiden tar hensyn til miljøet rundt seg, trenger programmet mer data fra området det skal operere i. Med datainput fra AIS vil programmet hele tiden kunne ta hensyn til andre fartøy i området. Den vil ikke bare kunne tilpasse ruten i forhold til andre fartøy, men også gi navigatøren en varsling om hvilke fartøy som potensielt kan interferere med ruten. Dette vil også kunne åpne for samhandling mellom flere fartøyer som benytter ruteplanleggeren. Skal man seile i formasjon med en flåte, vil ruteplanleggeren kunne gi hvert fartøy beskjed om hvor det skal legge seg for å komme inn i ønsket formasjon. Sammen med input fra radar vil man hele tiden kunne få nøyaktige målinger om avstand til de forskjellige fartøyene i formasjonen. Dette vil også gi ruteplanleggeren en bedre forståelse av nøyaktig hvor man befinner seg.

Med værmelding vil ruteplanleggeren kunne ta hensyn til været i et område. Er det mye vind og høye bølger vil ruteplanleggeren kunne ta hensyn til dette og, heller velge en rute som tar litt lenger tid. Dette åpner også for at den kan komme med flere forslag til ruter. Navigatøren kan da selv velge ønsket rute ut fra formålet med transitteringen.

Bibliografi

Bhatt, S. (den 19 03 2018). *Reinforcement Learning 101. Learn the essentials of Reinforcement Learning*. Hentet fra Towards Data Science: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

Churchville, F. (09 2021). *What is user interface (UI)?* HHentet fra Tech Target: https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI

Dobillas, S. (den 04 10 2022). *Q-learning Algorithm: How to Successfully Teach an Intelligent Agent to Play A Game*. Hentet fra Towards Data Science: https://towardsdatascience.com/q-learning-algorithm-how-to-successfully-teach-an-intelligent-agent-to-play-a-game-933595fd1abf

GeeksforGeeks. (den 23 August 2022). *Epsilon-Greedy Algorithm in Reinforcement Learning*. Hentet fra Geeksforgeeks: https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/?fbclid=IwAR21C_n7w4ZRx-nfxuihHT1kF6EeUiDD7ZL5WCx2oCDG2geDJeBZCs16-ag

Geeksforgeeks. (den 18 10 2022). *Reinforcement learning*. Hentet fra geeksforgeeks: https://www.geeksforgeeks.org/what-is-reinforcement-learning/

Hope, C. (den 04 12 2021). *What is a GUI (Graphcial User Interface)?* Hentet fra Computerhope: https://www.computerhope.com/jargon/g/gui.htm

Jing, H. (den 17 12 2022). *Reinforcement Learning - The Value Function*. Hentet fra Towards Data Science: https://towardsdatascience.com/reinforcement-learning-value-function-57b04e911152

Khan, T. (den 1 11 2022). *REINFORCEMENT LEARNING – EXPLORATION VS EXPLOITATION TRADEOFF*. Hentet fra AI ML ANALYTICS: https://ai-ml-analytics.com/reinforcement-learning-exploration-vs-exploitation-tradeoff/?fbclid=IwAR0olGWyP6KHddczsWQUC1e7reL_wRinW77O0nYEw8HAtxcpWW6ofm-J6r0

Lazy Programmer INC. (den 20 11 2018). *Artificial Intelligence: Reinforcement learning in Python*. Hentet fra Udemy.

Mathiesen, J. K., & Lowzow, J. (2017). *Hvordan benytte maskinlæring i navigasjon.* Bergen: Sjøkrigsskolen.

Myrianthous, G. (den 18 10 2022). *Supervised vs Unsupervised learning*. Hentet fra towardsdatascience: https://towardsdatascience.com/supervised-vs-unsupervised-learning-bf2eab13f288

Norman, J. M. (den 16 12 2022). *McCulloch & Pitts Publish the First Mathematical Model of a Neural Network*. Hentet fra History of Information: https://www.historyofinformation.com/detail.php?entryid=782

Sannsyn. (2020). *Maskinlæring finnes i mange former – her er de viktigste*. Hentet fra sannsyn.com: https://sannsyn.com/no/maskinlaering/

Selig, J. (den 18 10 2022). *Machine learning*. Hentet fra Expert AI: https://www.expert.ai/blog/machine-learning-definition/

Shyalika, C. (den 15 11 2019). *A Beginners Guide to Q-Learning. Model-Free Reinforcement Learning*. Hentet fra Towrads Data Science: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c

Strandvold, S., & Leines, H. P. (2018). Maskinlæring i praktisk beslutningstaking. i S. Strandvold, & H. P. Leines, *Maskinlæring i praktisk beslutningstaking* (s. 103). Bergen: Sjøkrigsskolen.

Sutton, R. S., & Barto, A. G. (2014, 2015). Reinforcement Learning: An Introduction In progress. i R. S. Andrew G. Barto, *Second Edition, in progress.* London, England: The MIT Press.

Torres, J. (2020, 05 11). *The Bellman Equation. V-function and Q-function | by Jordi TORRES.AI | Towards Data Science*. Hentet fra Towards Data Science: https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7

Vedlegg A – Førsteutkast CostMap.py

Filen kan hentes fra GitHub:

Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/Forsteutkast_CostMap.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)

```
1.  import numpy as np
2.  import pandas as pd
3.  import matplotlib.pyplot as plt
4.  from mpl_toolkits.mplot3d import axes3d
5.
6.
7.
8.
9.  data_array = np.load('data.npy')
10. #Teller baklengs [::-1] for å starte med første koordinater
11. Lat, Lon, Depth = data_array[::-1,0], data_array[::-1,1], data_array[::-1,2]
12.
13. # Get X, Y, Z
14.
15.
16.
17. X = Lat
18. Y = Lon
19. Z = Depth
20.
21. #Plotter figuren basert på X, Y og Z
22.
23. fig = plt.figure()
24. ax = fig.add_subplot(111, projection='3d', autoscale_on=True)
25. ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
26. ax.scatter(X, Y, Z, c='red')
27. plt.show()
```

Vedlegg B – Andreutkast CostMap.py

Filen kan hentes fra GitHub:

Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/Andreutkast_CostMap.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)

```
1.  import numpy as np
2.  import pandas as pd
3.  import matplotlib.pyplot as plt
4.
5.  #Henter ut dataene fra .npy fila og legger de i en numpy-array
6.  data_array = np.load('data.npy')
7.
8.  #Kaller første kolonne for easting, andre for northing og tredje for depth
9.  easting, northing, depth = data_array[:,0], data_array[:,1], data_array[:,2]
10.
11.
12. #Plotter dataene ved hjelp av funksjonen scatter
13. plt.scatter(easting, northing)
```

```
14. plt.show()
```

Vedlegg C – Brukermain.py

Filen kan hentes fra GitHub:

[Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/Brukermain.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)]

```
1.  #### INFORMASJON ####
2.  # Filnavn: Brukermain.py
3.  # Forfatter: Jone Måsøy, André Killingmoe og Eirik Kiland
4.  # Inspirert av Maskinlæring i praktisk beslutningstaking av Simen Strandvold og Hans
    Petter Leines
5.
6.  # Git: https://github.com/beachviolence/bachelor.git
7.
8.
9.  # Beskrivelse:
10. # Denne koden henter miljøet, trener på det og
11. # returnerer anbefalt handling og evt. strategien
12. # Det er denne koden som fungerer opp mot GUI'en
13.
14. #### INKLUDERTE BIBLIOTEK ####
15.
16. # Lisenser for Numpy kan leses på
17. # https://docs.scipy.org/doc/numpy-1.10.0/license.html
18. import numpy as np
19.
20. # Lisenser for MatplotLib kan leses på
21. # https://matplotlib.org/users/license.html
22. import time
23. from os.path import exists
24. import os
25. from pyproj import Proj, transform
26. import geopy.distance
27. import math
28.
29. # Inkluderte .py filer
30. from newlabyrinth import grid_world, print_values
31. from sauterlek import dy, dx, minx, miny, convertStartSlutt
32.
33. #### HYPERPARAMETERE ####
34. GAMMA = 0.9
35. ALPHA = 0.1
36. EPSILON = 0.1
37.
38. #### GLOBALE VARIABLER ####
39. # Alle mulige handlinger
40. ALL_POSSIBLE_ACTIONS = ('U', 'D', 'L', 'R')
41.
42. # Skrur av og på debugverktøy som grafer
43. DEBUG = True
44. # Div lagrede variabler
45. PREV_POLICY = {}
46. PREV_STATE = ()
47. PREV_Q = {}
48.
49. # Antall episoder den skal trene ved neste trening
50. SECOND_EPISODES = 4000
```

```
51.
52.
53. #### FUNKSJONER ####
54. # Returnerer argmax og maxverdien fra en dict
55. def max_dict(d):
56.   max_key = None
57.   max_val = float('-inf')
58.   for k, v in d.items():
59.     if v > max_val:
60.       max_val = v
61.       max_key = k
62.   return max_key, max_val
63.
64. # Velger en tilfeldig handling med sannsynlighet = epsilon
65. def random_action(a, eps=0.1):
66.   p = np.random.random()
67.   if p < (1 - eps):
68.     return a
69.   else:
70.     return np.random.choice(ALL_POSSIBLE_ACTIONS)
71.
72. # Henter opp miljøet fra grid_world + startposisjonen
73. def run(epoch,Startposisjon,Sluttposisjon):
74.   startY,startX = convertStartSlutt(Startposisjon,Sluttposisjon)
75.   startstate = (startY,startX)
76.   grid, learn = grid_world(startstate)
77.
78.   # Bestemmer om miljøet er likt eller om det skal
79.   # læres på nytt
80.   if learn:
81.     return q_learn(grid, epoch)
82.   else:
83.     return PREV_POLICY, grid.current_state()
84.
85.
86.
87. #
88. def q_learn(grid, epoch):
89.
90.   global PREV_Q
91.
92.
93.
94.   # Starter timer i debug modus
95.   if DEBUG: t0 = time.time()
96.   trigstart = False
97.
98.
99.
100.        # Importerer start-tilstand
101.        start_state = grid.current_state()
102.
103.        # Initsialiserer Q(s,a) ved første kjøring
104.        # hvis ikke henter den Q(s,a) fra forrige
105.        if epoch == 0:
106.          Q = {}
107.          states = grid.all_states()
108.          for s in states:
109.            Q[s] = {}
110.            for a in ALL_POSSIBLE_ACTIONS:
111.              Q[s][a] = 0
112.        else:
113.          Q = PREV_Q
114.          states = grid.all_states()
115.
116.        # Holder kontroll på hvor mange ganger Q[s] er oppdatert
```

```
117.          update_counts = {}
118.          update_counts_sa = {}
119.          for s in states:
120.            update_counts_sa[s] = {}
121.            for a in ALL_POSSIBLE_ACTIONS:
122.              update_counts_sa[s][a] = 1.0
123.
124.          # Init variabler
125.          t = 1.0
126.          deltas = []
127.          sum_reward = []
128.
129.          # Bestemmer antall episoder det skal trenes
130.
131.          episode_func = 20000
132.
133.          # Brukes ved løsning av komplekse miljø hvor det trengs
134.          # flere episoder
135.
136.          if epoch == 0:
137.            num_episodes = episode_func
138.          else:
139.            num_episodes = SECOND_EPISODES
140.
141.          # Starter trening
142.          for it in range(int(num_episodes)):
143.            if it % 100 == 0:
144.              t += 1e-2
145.            if it % 2000 == 0:
146.              if DEBUG:
147.                print("it:", it)
148.
149.            # Istedenfor å generere en epsisode, spiller vi en
150.            # episode inne i treningen
151.
152.            # Henter startstate og setter
153.            s = start_state
154.            grid.set_state(s)
155.
156.            # Den første (s, r) er start-tilstanden og 0 siden vi ikke
157.            # får en gevinst. Den siste (s, r) er den absorberende tilstanden
158.            # og den site gevinsten, og er per definisjon 0
159.
160.            # Init-verdier
161.            if not os.path.exists('NX_fil.npy'):
162.              a, _ = max_dict(Q[s])
163.            else:
164.              a, _ = max_dict(PREV_Q[s])
165.            biggest_change = 0
166.            sum_episode_rewards = []
167.
168.            # Gjennomfører én episode
169.            while not grid.game_over():
170.              # Utfører en tilfeldig handling med sannsynlighet eps
171.              a = random_action(a, eps=EPSILON)
172.
173.              # Beveger seg til valgt tilstand
174.              r = grid.move(a)
175.
176.              # Legger episodens gevinster i en liste
177.              sum_episode_rewards.append(r)
178.
179.              # Neste state
180.              s2 = grid.current_state()
181.              ###################
182.              #print(s, s2)
```

```
183.            # Oppdaterer tellingen av Q(s,a)
184.            update_counts_sa[s][a] += 0.005
185.
186.            #### OPPDATERINGEN AV Q(s,a)
187.
188.            # Tar vare på gamle verdien av Q(s,a)
189.            old_qsa = Q[s][a]
190.
191.            # Finner argmax_a(Q(s2,a)) og max_a(Q(s2,a))
192.            a2, max_q_s2a2 = max_dict(Q[s2])
193.
194.            # Gjør oppdateringen av Q(s,a) i henhold til formel 12, side 20
195.            Q[s][a] = Q[s][a] + ALPHA*(r + GAMMA*max_q_s2a2 - Q[s][a])
196.
197.            # Registrerer den største endringen gjort i Q(s,a) i løpet av episoden
198.            biggest_change = max(biggest_change, np.abs(old_qsa - Q[s][a]))
199.
200.            # Holder kontroll på hvor mange ganger Q(s) er oppdatert
201.            update_counts[s] = update_counts.get(s,0) + 1
202.
203.            # Neste tilstand blir nåværende tilstand
204.            s = s2
205.            # Neste handling blir nåværende handling
206.            a = a2
207.
208.          # Legger største forandring av Q i en liste til bruk i graf
209.          deltas.append(biggest_change)
210.
211.          # Legger samlet gevinst pr. episode i en liste til bruk i graf
212.          sum_reward.append(sum(sum_episode_rewards))
213.
214.        # Finner strategien fra Q
215.        # og tilstands-verdifunksjonen V fra handlings-verdifunksjonen Q
216.        policy = {}
217.        V = {}
218.        for s in grid.actions.keys():
219.          a, max_q = max_dict(Q[s])
220.          policy[s] = a
221.          V[s] = max_q
222.
223.        # Lagerer strategien til bruk i neste iterasjon
224.        global PREV_POLICY
225.        PREV_POLICY = policy
226.
227.
228.
229.        # Brukes til debugging og demonstrering
230.        if DEBUG:
231.          # Stopper timer og utskrift av tid brukt
232.          t1 = time.time()
233.          print("Time elapsed: {}".format(t1-t0))
234.
235.          # Finner laveste delta
236.          lowest_delta = min(deltas)
237.          print(lowest_delta)
238.
239.
240.
241.          # Printer tid brukt i hver tilstand
242.          print("update counts:")
243.          total = np.sum(list(update_counts.values()))
244.          for k, v in update_counts.items():
245.            update_counts[k] = float(v) / total
246.          print_values(update_counts, grid)
247.
248.          # Visualiserer miljø med gevinster
```

```
249.            rew = np.zeros((grid.height, grid.width))
250.            for i in range (grid.height):
251.              for j in range(grid.width):
252.                rew[i,j] = grid.rewards.get((i,j), 0)
253.
254.            # Visualiserer verdifunksjonen
255.            val = np.zeros((grid.height, grid.width))
256.            for i in range (grid.width):
257.              for j in range(grid.height):
258.                val[i,j] = V.get((i,j), 0)
259.
260.
261.            #For å telle hvor mange piksler ruta går over
262.            l = 0
263.             # Visualiserer strategien fra start til mål
264.            route = rew
265.            pos_x, pos_y = start_state[1],start_state[0]
266.
267.            #Definerer lister for omregning til latitude og longditude
268.            NorthList =[]
269.            EastList = []
270.            NorthEastArray = ()
271.
272.            while route[pos_y,pos_x] != 5:
273.              route[pos_y,pos_x] = 5
274.              if policy.get((pos_y,pos_x)) == 'U':
275.                pos_y -= 1
276.              elif policy.get((pos_y,pos_x)) == 'D':
277.                pos_y += 1
278.              elif policy.get((pos_y,pos_x)) == 'L':
279.                pos_x -= 1
280.              elif policy.get((pos_y,pos_x)) == 'R':
281.                pos_x += 1
282.              l +=1
283.
284.              ######### Regner tilbake til Northing og Easting. X- og Y-verdier
    ####################################
285.              reverse_ix = -round((pos_y * dx) + minx)
286.              reverse_iy = round((pos_x * dy) + miny)
287.
288.              # Legger disse til en liste
289.              NorthList.append(reverse_ix)
290.              EastList.append(reverse_iy)
291.
292.            #Omgjør xyz-koordinater til grader
293.            inProj = Proj('+proj=utm +zone=33 +ellps=WGS84')
294.            outProj = Proj('+proj=longlat')
295.            EastList_np = np.array(EastList)
296.            EastList_np = np.column_stack(EastList_np)
297.            NorthList_np = np.array(NorthList)
298.            NorthList_np = np.column_stack(NorthList_np)
299.
300.
301.            #Initialiserer listene fyllt med 0er med like mange verdier som det
    finnes koordinater
302.            # i ruta
303.            x2 = np.zeros((1,len(NorthList_np)))
304.            y2 = np.zeros((1,len(NorthList_np)))
305.
306.            # Transformerer hver koordinat fra xyz til grader og legger disse i en
    array
307.            for i in range(len(NorthList_np)):
308.              x2,y2 = transform(inProj,outProj,EastList_np[i],NorthList_np[i])
309.            NorthEastArray = np.column_stack((y2, x2))
310.
311.
```

```
312.            #Gjør om arrayen til liste slik at HMI kan lese det enkelt
313.            FerdigList =  NorthEastArray.tolist()
314.
315.            #Regner ut avstanden til ruta
316.            Distanse = 0
317.            for i in range(len(FerdigList)):
318.              if i == len(FerdigList) - 1:
319.                break
320.              koordinat1 = FerdigList[i]
321.              koordinat2 = FerdigList[i+1]
322.              EnkelDistanse = geopy.distance.geodesic(koordinat1, koordinat2).km
323.              Distanse = Distanse + EnkelDistanse
324.
325.            #Regner ut hvor lang tid ruta tar ved en viss hastighet
326.            # Distanse / fart = tid
327.            NautiskMil = Distanse / 1.852
328.            Knop = 15.5
329.            Tid = NautiskMil / Knop
330.            Antallminutter = Tid * 60
331.            Antallsekunder = ((Antallminutter % 1)*100) * 60/100
332.
333.            print(math.trunc(Antallminutter), ' min ', Antallsekunder, ' s')
334.
335.            #Lagrer så ruta og distansen til fil som HMI kan lese fra
336.            np.save('Ruta.npy', FerdigList)
337.            np.save('Distanse.npy', NautiskMil)
338.
339.          # Returnerer strategien og start-tilstand
340.          return policy, start_state
```

Vedlegg D – Main.py

Filen kan hentes fra GitHub:

[Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/main.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)]

```
1.  #### INFORMASJON ####
2.  # Filnavn: brukermain.py
3.  # Forfatter: Jone Måsøy, André Killingmoe og Eirik Kiland
4.  # Inspirert av Maskinlæring i praktisk beslutningstaking av Simen Strandvold og Hans
    Petter Leines
5.
6.
7.  # Beskrivelse:
8.  # Denne koden henter miljøet fra newlabyrinth.py, trener på det og
9.  # returnerer anbefalt handling og strategi
10. # Dette er en visualiseringsfil for ruta, ikke en fil som brukes i programmet.
11.
12. #### INKLUDERTE BIBLIOTEK ####
13.
14. # Lisenser for Numpy kan leses på
15. # https://docs.scipy.org/doc/numpy-1.10.0/license.html
16. import numpy as np
17.
18. # Lisenser for MatplotLib kan leses på
19. # https://matplotlib.org/users/license.html
```

```
20. import matplotlib.pyplot as plt
21.
22. import time
23. from os.path import exists
24. import os
25. from convertbng.util import convert_bng, convert_lonlat
26. from convertbng.util import convert_epsg3857_to_wgs84
27. from pyproj import Proj, transform
28. import geopy.distance
29. import math
30.
31. # Inkluderte .py filer
32. from newlabyrinth2 import grid_world, print_values, print_policy
33. from sauterlek2 import dy, dx, minx, miny
34.
35. #### HYPERPARAMETERE ####
36. GAMMA = 0.9
37. ALPHA = 0.1
38. EPSILON = 0.1
39.
40. #### GLOBALE VARIABLER ####
41. # Alle mulige handlinger
42. ALL_POSSIBLE_ACTIONS = ('U', 'D', 'L', 'R')
43.
44. # Skrur av og på debugverktøy som grafer
45. DEBUG = True
46. trigstart = False
47.
48. # Div lagrede variabler
49. PREV_POLICY = {}
50. PREV_STATE = ()
51. PREV_Q = {}
52.
53. # Antall episoder den skal trene ved neste trening
54. SECOND_EPISODES = 4000
55.
56. #### FUNKSJONER ####
57.
58. # Returnerer argmax og maxverdien fra en dict
59. def max_dict(d):
60.   max_key = None
61.   max_val = float('-inf')
62.   for k, v in d.items():
63.     if v > max_val:
64.       max_val = v
65.       max_key = k
66.   return max_key, max_val
67.
68.
69. # Velger en tilfeldig handling med sannsynlighet = epsilon
70. def random_action(a, eps=0.1):
71.   p = np.random.random()
72.   if p < (1 - eps):
73.     return a
74.   else:
75.     return np.random.choice(ALL_POSSIBLE_ACTIONS)
76.
77.
78. # Henter opp miljøet fra grid_world
79. def run(epoch):
80.   grid, learn = grid_world()
81.
82.   # Bestemmer om miljøet er likt eller om det skal
83.   # læres på nytt
84.   if learn:
85.     return q_learn(grid, epoch)
```

```
86.   else:
87.     return PREV_POLICY, grid.current_state()
88.
89.
90.
91. # Selve læringsfunksjonen
92. def q_learn(grid, epoch):
93.
94.   global PREV_Q
95.
96.   # Starter timer i debug modus
97.   if DEBUG: t0 = time.time()
98.   trigstart = False
99.
100.          # Importerer start-tilstand
101.          start_state = grid.current_state()
102.
103.          # Initsialiserer Q(s,a) ved første kjøring
104.          # hvis ikke henter den Q(s,a) fra forrige
105.          if epoch == 0:
106.            Q = {}
107.            states = grid.all_states()
108.            for s in states:
109.              Q[s] = {}
110.              for a in ALL_POSSIBLE_ACTIONS:
111.                Q[s][a] = 0
112.          else:
113.            Q = PREV_Q
114.            states = grid.all_states()
115.
116.          # Holder kontroll på hvor mange ganger Q[s] er oppdatert
117.          update_counts = {}
118.          update_counts_sa = {}
119.          for s in states:
120.            update_counts_sa[s] = {}
121.            for a in ALL_POSSIBLE_ACTIONS:
122.              update_counts_sa[s][a] = 1.0
123.
124.          # Init variabler
125.          t = 1.0
126.          deltas = []
127.          sum_reward = []
128.
129.          # Bestemmer antall episoder det skal trenes
130.          episode_func = 20000
131.
132.          # Brukes ved løsning av komplekse miljø hvor det trengs
133.          # flere episoder
134.          if epoch == 0:
135.            num_episodes = episode_func
136.          else:
137.            num_episodes = SECOND_EPISODES
138.
139.
140.          # Starter trening
141.          for it in range(int(num_episodes)):
142.            if it % 100 == 0:
143.              t += 1e-2
144.            if it % 2000 == 0:
145.              if DEBUG:
146.                print("it:", it)
147.
148.            # Istedenfor å generere en epsisode, spiller vi en
149.            # episode inne i treningen
150.
151.            # Henter startstate
```

```
152.            s = start_state
153.            grid.set_state(s)
154.
155.            # Den første (s, r) er start-tilstanden og 0 siden vi ikke
156.            # får en gevinst. Den siste (s, r) er den absorberende tilstanden
157.            # og den site gevinsten, og er per definisjon 0.
158.
159.            # Init-verdier
160.            if not os.path.exists('NX_fil.npy'):
161.              a, _ = max_dict(Q[s])
162.            else:
163.              a, _ = max_dict(PREV_Q[s])
164.            biggest_change = 0
165.            sum_episode_rewards = []
166.
167.
168.            # Gjennomfører én episode
169.            while not grid.game_over():
170.
171.              # Utfører en tilfeldig handling med sannsynlighet eps
172.              a = random_action(a, eps=EPSILON)
173.
174.              # Beveger seg til valgt tilstand
175.              r = grid.move(a)
176.
177.              # Legger episodens gevinster i en liste
178.              sum_episode_rewards.append(r)
179.
180.              # Neste state
181.              s2 = grid.current_state()
182.
183.              # Oppdaterer tellingen av Q(s,a)
184.              update_counts_sa[s][a] += 0.005
185.
186.
187.              #### OPPDATERINGEN AV Q(s,a)
188.
189.              # Tar vare på gamle verdien av Q(s,a)
190.              old_qsa = Q[s][a]
191.
192.              # Finner argmax_a(Q(s2,a)) og max_a(Q(s2,a))
193.              a2, max_q_s2a2 = max_dict(Q[s2])
194.
195.              # Gjør oppdateringen av Q(s,a) i henhold til formel XXXX, side XXXX
196.              Q[s][a] = Q[s][a] + ALPHA*(r + GAMMA*max_q_s2a2 - Q[s][a])
197.
198.              # Registrerer den største endringen gjort i Q(s,a) i løpet av episoden
199.              biggest_change = max(biggest_change, np.abs(old_qsa - Q[s][a]))
200.
201.              # Holder kontroll på hvor mange ganger Q(s) er oppdatert
202.              update_counts[s] = update_counts.get(s,0) + 1
203.
204.              # Neste tilstand blir nåværende tilstand
205.              s = s2
206.              # Neste handling blir nåværende handling
207.              a = a2
208.
209.            # Legger største forandring av Q i en liste til bruk i graf
210.            deltas.append(biggest_change)
211.
212.            # Legger samlet gevinst pr. episode i en liste til bruk i graf
213.            sum_reward.append(sum(sum_episode_rewards))
214.
215.
216.          # Finner strategien fra Q
217.          # og tilstands-verdifunksjonen V fra handlings-verdifunksjonen Q
```

```
218.          policy = {}
219.          V = {}
220.          for s in grid.actions.keys():
221.            a, max_q = max_dict(Q[s])
222.            policy[s] = a
223.            V[s] = max_q
224.
225.
226.          # Lagrer strategien til bruk i neste iterasjon
227.          global PREV_POLICY
228.          PREV_POLICY = policy
229.
230.
231.          # Brukes til debugging og demonstrering
232.          if DEBUG:
233.
234.            # Stopper timer og utskrift av tid brukt
235.            t1 = time.time()
236.            print("Time elapsed: {}".format(t1-t0))
237.
238.            # Finner laveste delta
239.            lowest_delta = min(deltas)
240.            print(lowest_delta)
241.
242.            # Plotter forandringen i delta
243.            plt.plot(deltas)
244.            plt.show()
245.
246.            # Plotter oppnådd gevinst pr episode
247.            plt.plot(sum_reward)
248.            plt.show()
249.
250.            # Printer tid brukt i hver tilstand
251.            print("update counts:")
252.            total = np.sum(list(update_counts.values()))
253.            for k, v in update_counts.items():
254.              update_counts[k] = float(v) / total
255.            print_values(update_counts, grid)
256.
257.            # Visualiserer miljø med gevinster
258.            rew = np.zeros((grid.height, grid.width))
259.            for i in range (grid.height):
260.              for j in range(grid.width):
261.                rew[i,j] = grid.rewards.get((i,j), 0)
262.            plt.imshow(rew)
263.            plt.clim(-11,10)
264.            plt.colorbar()
265.            plt.show()
266.
267.            # Visualiserer verdifunksjonen
268.            val = np.zeros((grid.height, grid.width))
269.            for i in range (grid.width):
270.              for j in range(grid.height):
271.                val[i,j] = V.get((i,j), 0)
272.            plt.imshow(val)
273.            plt.colorbar()
274.            plt.show()
275.
276.
277.            # For å telle hvor mange piksler ruta går over
278.            l = 0
279.            # Visualiserer strategien fra start til mål
280.            route = rew
281.            pos_x, pos_y = start_state[1],start_state[0]
282.
283.            # Definerer lister for omregning til latitude og longditude
```

```
284.            NorthList =[]
285.            EastList = []
286.            NorthEastArray = ()
287.
288.            # Gir hver pixel i ruta gevinsten 5 for å vise den grafisk.
289.            while route[pos_y,pos_x] != 5:
290.              route[pos_y,pos_x] = 5
291.              if policy.get((pos_y,pos_x)) == 'U':
292.                pos_y -= 1
293.              elif policy.get((pos_y,pos_x)) == 'D':
294.                pos_y += 1
295.              elif policy.get((pos_y,pos_x)) == 'L':
296.                pos_x -= 1
297.              elif policy.get((pos_y,pos_x)) == 'R':
298.                pos_x += 1
299.              l +=1
300.
301.              # Regner tilbake til Northing og Easting. X- og Y-verdier
302.              reverse_ix = -round((pos_y * dx) + minx)
303.              reverse_iy = round((pos_x * dy) + miny)
304.              NorthList.append(reverse_ix)
305.
306.              EastList.append(reverse_iy)
307.
308.
309.            # Omgjør xyz-koordinater til grader
310.            inProj = Proj('+proj=utm +zone=33 +ellps=WGS84')
311.            outProj = Proj('+proj=longlat')
312.            EastList_np = np.array(EastList)
313.            EastList_np = np.column_stack(EastList_np)
314.            NorthList_np = np.array(NorthList)
315.            NorthList_np = np.column_stack(NorthList_np)
316.
317.            # Initialiserer listene fyllt med 0er med like mange
318.            # verdier som det finnes koordinater i ruta
319.            x2 = np.zeros((1,len(NorthList_np)))
320.            y2 = np.zeros((1,len(NorthList_np)))
321.
322.            # Transformerer hver koordinat fra xyz til grader og legger disse i en
    array
323.            for i in range(len(NorthList_np)):
324.              x2,y2 = transform(inProj,outProj,EastList_np[i],NorthList_np[i])
325.            NorthEastArray = np.column_stack((y2, x2))
326.
327.            #Gjør om arrayen til liste slik at HMI kan lese det enkelt
328.            FerdigList =  NorthEastArray.tolist()
329.
330.            #Regner ut avstanden til ruta
331.            Distanse = 0
332.            for i in range(len(FerdigList)):
333.              if i == len(FerdigList) - 1:
334.                break
335.              koordinat1 = FerdigList[i]
336.              koordinat2 = FerdigList[i+1]
337.              EnkelDistanse = geopy.distance.geodesic(koordinat1, koordinat2).km
338.              Distanse = Distanse + EnkelDistanse
339.
340.
341.            #Regner ut hvor lang tid ruta tar ved en viss hastighet
342.            # Distanse / fart = tid
343.            NautiskMil = Distanse / 1.852
344.            Knop = 15.5
345.            Tid = NautiskMil / Knop
346.            Antallminutter = Tid * 60
347.            Antallsekunder = ((Antallminutter % 1)*100) * 60/100
348.
```

```
349.            print(math.trunc(Antallminutter), ' min ', Antallsekunder, ' s')
350.
351.            #Lagrer så ruta og distansen til fil som HMI kan lese fra
352.            np.save('Ruta.npy', FerdigList)
353.            np.save('Distanse.npy', NautiskMil)
354.
355.            print(NautiskMil, ' ',  'nm')
356.
357.            # Viser ruta grafisk
358.            plt.imshow(route)
359.            plt.colorbar()
360.            plt.show()
361.
362.            # Printfunksjoner
363.            print("rewards:")
364.            print_values(grid.rewards, grid)
365.            print("values:")
366.            print_values(V, grid)
367.            print("policy:")
368.            print_policy(policy, grid)
369.
370.          # Returnerer strategien og start-tilstand
371.          return policy, start_state
372.
373.        # Kjører trening på valgt miljø og printer anbefalt handling
374.        if __name__ == "__main__":
375.          epoch = 0
376.          while True:
377.            input("Trykk enter")
378.            policy, s, m = run(epoch)
379.            action = policy.get(s)
380.            print("recommended action:")
381.            print(action)
382.            epoch += 1
```

Vedlegg E – NewLabyrinth.py

Filen kan hentes fra GitHub:

[Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/newlabyrinth.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)]

```
1.  #### INFORMASJON ####
2.  # Filnavn: newlabyrinth.py
3.  # Forfatter: Jone Måsøy, André Killingmoe og Eirik Kiland
4.  # Inspirert av Maskinlæring i praktisk beslutningstaking av Simen Strandvold og Hans
    Petter Leines
5.
6.  # Beskrivelse:
7.  # Denne koden generer miljøet som er lesbart
8.  # for brukermain.py eller main.py koden.
9.
10. # Importerte biblioteker
11. import numpy as np
12. from sauterlek import nx, ny, pixels, mazes
13.
14. #### HYPERPARAMETERE ####
```

```
15. # Gevinster
16. GOAL_REWARD = 10000
17. LAND_REWARD = -10
18. SHALLOW_REWARD = -5
19. KINDASHALLOW_REWARD = -3
20. WAYPOINT_REWARD = 0
21.
22. STEP_COST = -(60/nx) #For å gjøre denne dynamisk
23. DYPGANG = -10
24. vekting = 1.2/nx   #For å gjøre denne dynamisk
25.
26.
27. # Størrelse på manuelt miljø og antall hinder ønsket
28. GRID_HEIGHT = nx
29. GRID_WIDTH = ny
30. NUM_OBSTACLES = GRID_HEIGHT
31.
32. # Initsialiserer tilfeldig eller manuelt miljø
33. RANDOM_REWARDS = False
34. MAZE = mazes(2)
35.
36. ## GLOBALE VARIABLER ##
37. PREV_REWARDS = {}
38. last_action = None
39. firstscan = True
40.
41.
42. #### KLASSE SOM INNEHOLDER MILJØET ####
43. class Grid:
44.     def __init__(self, height, width, start):
45.         self.height = height
46.         self.width = width
47.         self.i = start[0]
48.         self.j = start[1]
49.
50.     # Setter gevinster og lovlige handlinger
51.     def set(self, rewards, actions):
52.         self.rewards = rewards
53.         self.actions = actions
54.
55.     # Setter tilstanden
56.     def set_state(self, s):
57.         self.i = s[0]
58.         self.j = s[1]
59.
60.     # Returnerer nåværende tilstand
61.     def current_state(self):
62.         return(self.i, self.j)
63.
64.
65.
66.     # Beveger beslutningstakeren i miljøet
67.     def move(self, action):
68.         global last_action
69.         global firstscan
70.         # Sjekker om det er en lovlig handling
71.         ekstrakost = 0
72.         if action in self.actions[(self.i, self.j)]:
73.             if action == 'U':
74.                 self.i -= 1
75.             elif action == 'D':
76.                 self.i += 1
77.             elif action == 'L':
78.                 self.j -= 1
79.             elif action == 'R':
80.                 self.j += 1
```

```
81.         # Returnerer gevinst for neste tilstand
82.         return (self.rewards.get((self.i, self.j), 0))
83.
84.     # Sjekker om episoden er ferdig
85.     def game_over(self):
86.         # True om beslutningstakeren er i mål
87.         return self.rewards[self.i, self.j] == GOAL_REWARD
88.     # Retunerer en dict med alle tilstander
89.     def all_states(self):
90.         return set(self.actions.keys()) | set(self.rewards.keys())
91.
92. #### KONSTRUKSJON AV MILJØ ####
93.
94. # Konstruerer miljø som en dict
95. def make_grid(y, x, num_obstacles = 0,startstate = (1,1)):
96.     g = Grid(y, x, startstate)
97.     rewards = {}
98.
99.     # Legger til mulige handlinger i hver tilstand
100.         actions = {
101.             startstate: ('U', 'D', 'R', 'L'),
102.             (y-1,x-1): ('U', 'L'),
103.             (0, x-1): ('D', 'L'),
104.             (y-1, 0): ('U', 'R'),
105.             (0, 0): ('D', 'R')
106.         }
107.
108.         for i in range(1, x-1):
109.             actions[(0,i)] = ('D', 'R', 'L')
110.             actions[(y-1,i)] = ('U', 'R', 'L')
111.         for j in range(1, y-1):
112.             actions[(j,0)] = ('U', 'D', 'R')
113.             actions[(j,x-1)] = ('U', 'D', 'L')
114.         for k in range(1,x-1):
115.             for l in range(1,y-1):
116.                 actions[(l,k)] = ('U', 'D', 'R', 'L')
117.
118.         # Legger til steg-gevinst i hver tilstand
119.         for i in range(x):
120.             for j in range(y):
121.                 samletreward = 0
122.                 if pixels[(j,i)] > DYPGANG:
123.                     samletreward = samletreward + LAND_REWARD + STEP_COST
124.                 elif pixels[(j,i)] < 9*DYPGANG:
125.                     samletreward = samletreward + STEP_COST
126.                 else:
127.                     dybde = pixels[(j,i)]
128.                     samletreward = samletreward + STEP_COST +
    vekting*(dybde/(8*DYPGANG) - 1 - 1/8)
129.                 rewards[(j,i)] = samletreward
130.
131.         # Leser og tolker labyrinter hentet fra filen sauterlek.py
132.         if not RANDOM_REWARDS:
133.             labyrint = MAZE
134.             k = 0
135.             l = 0
136.             for i in labyrint:
137.                 for j in i:
138.                     if j==10: rewards[(l,k)] = GOAL_REWARD
139.                     k= k+1
140.                 k = 0
141.                 l = l+1
142.
143.         # Lager tilfeldige hindringer i miljøet
144.         else:
145.             for _ in range(num_obstacles):
```

```
146.                    rewards[(np.random.randint(0,y-1),np.random.randint(0,x-1))] =
    LAND_REWARD
147.                for _ in range(num_obstacles):
148.                    rewards[(np.random.randint(0,y-1),np.random.randint(0,x-1))] =
    SHALLOW_REWARD
149.                rewards[(y-1,x-1)] = GOAL_REWARD
150.                rewards[startstate] = STEP_COST
151.
152.            # Setter handlinger og gevinster i klassen Grid
153.            g.set(rewards, actions)
154.
155.            # Returnerer et element av klassen Grid
156.            return g
157.
158.        #### MANUELT MILJØ ####
159.        def manual_grid(startstate):
160.            # Lager grid [y,x] stor
161.            g = make_grid(GRID_HEIGHT,GRID_WIDTH,NUM_OBSTACLES,startstate)
162.            # Sier at den skal trene hver gang
163.            train = True
164.            # Setter startposisjon
165.            s = startstate
166.            g.set_state(s)
167.            # Returnerer element av klassen grid og variablen train
168.            return g, train
169.
170.        # Funksjonen hentet opp i main.py som inneholder
171.        # hele miljøet i objektet grid
172.        def grid_world(startstate):
173.            return manual_grid(startstate)
174.
175.        #### PRINT FUNKSJONER ####
176.        def print_values(V, g):
177.            for i in range(g.height):
178.                print("-------------------")
179.                for j in range(g.width):
180.                    v = V.get((i,j), 0)
181.                    if v >= 0:
182.                        print("  %.2f  | " % v, end="")
183.                    else:
184.                        print(" %.2f  |" % v, end=" ")
185.                print("")
186.
187.        def print_policy(P, g):
188.            for i in range(g.height):
189.                print("-------------------")
190.                for j in range(g.width):
191.                    a = P.get((i, j), ' ')
192.                    print("  %s  |" % a, end="")
193.                print("")
```

Vedlegg F – Sauterlek.py.

Filen kan hentes fra GitHub:

[Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/sauterlek.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)]

```
1.  #### INFORMASJON ####
2.  # Filnavn: sauterlek.py
3.  # Forfatter: Jone Måsøy, André Killingmoe og Eirik Kiland
4.
5.  # Beskrivelse:
6.  # Denne koden generer matrisene som er lesbart
7.  # for brukermain.py eller main.py koden.
8.
9.  import numpy as np
10.
11. # Laster inn data fra datafilen og legger disse i variabler
12. data_array = np.load('data.npy')
13. data_array = data_array.astype(float)
14. Y, X, Z = data_array[:,0], data_array[:,1], data_array[:,2]
15. X = -X
16.
17. # Setter oppløsninga i kartet lik 150 * 150 ruter
18. nx = 150
19. ny = 150
20.
21. # Setter min og max verdier som brukes i mellomregning
22. minx = np.min(X)
23. maxx = np.max(X)
24. miny = np.min(Y)
25. maxy = np.max(Y)
26.
27. # Regner ut dx og setter dy lik.
28. dx = ((maxx-minx)/nx)
29. dy = dx
30.
31. # Initialiserer matriser som vi senere putter verdier i
32. pixels = np.zeros((nx+1,ny+1))
33. nvalues = np.zeros((nx+1,ny+1))
34.
35. i = 0
36.
37. dypest = 0
38.
39. for k in Z:
40.     # Plassering er ix og iy
41.     ix = round((X[i]-minx)//dx)
42.     iy = round((Y[i]-miny)//dy)
43.     # Finner antall og summen av alle som passer i denne plasseringen.
44.     nvalues[ix,iy] = nvalues[ix,iy] +1
45.     pixels[ix,iy] = pixels[ix,iy] + Z[i]
46.     if Z[i] < dypest:
47.         dypest = Z[i]
48.
49.     i = i + 1
50.
51.
52.
53.
```

```
54. i = 0
55. # Omgjør summen til gjennomsnitt
56. for k in pixels[:,1]:
57.     j = 0
58.     for l in pixels[1,:]:
59.         if nvalues[i,j] > 0:
60.             pixels[i,j] = round(pixels[i,j]/nvalues[i,j])
61.             #pixels[i,j] = 1
62.         else:
63.             pixels[i,j] = 0
64.         j = j + 1
65.     i = i + 1
66.
67. # Funksjon som setter startposisjonen for HMIen
68.
69. def convertStartSlutt(Startposisjon,Sluttposisjon):
70.     #Regner ut startposisjonen
71.     Startposisjon = Startposisjon.astype(float)
72.     dataY = Startposisjon[:,0]
73.     dataX = Startposisjon[:,1]
74.     dataY = -dataY
75.
76.     startY = round((dataY[0]-minx)//dx)
77.     startX = round((dataX[0]-miny)//dy)
78.
79.     #Regner ut målposisjon
80.     Sluttposisjon = Sluttposisjon.astype(float)
81.     dataYs = Sluttposisjon[:,0]
82.     dataXs = Sluttposisjon[:,1]
83.
84.     dataYs = -dataYs
85.     sluttY = round((dataYs[0]-minx)//dx)
86.     sluttX = round((dataXs[0]-miny)//dy)
87.
88.     #Setter målposisjon
89.     pixels[sluttY, sluttX] = 10
90.
91.
92.     return startY, startX
93.
94. # Funksjon for valg av labyrint
95.
96. def mazes(num):
97.     labyrint0 = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
98.                 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
99.                 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
100.                      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
101.                      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
102.                      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
103.                      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
104.                      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
105.                      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
106.                      [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
107.                      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
```

```
108.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
109.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
110.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
111.                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
112.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
113.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
114.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
115.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
116.                            [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
117.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
118.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
119.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
120.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
121.                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
122.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
123.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
124.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
125.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
126.                            [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
127.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
128.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
129.                            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5]]
130.
131.            labyrint1 = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
132.                            [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
133.                            [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
134.                            [0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
135.                            [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
136.                            [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1,
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0],
137.                            [0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1,
    0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
138.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1,
    0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
139.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1,
    0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0],
140.                            [0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1,
    0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0],
```

```
141.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1,
    0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0],
142.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1,
    1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0],
143.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0],
144.                            [0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1,
    1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
145.                            [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0],
146.                            [0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1,
    1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
147.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 5, 1, 0, 1,
    0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0],
148.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0],
149.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
150.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1,
    1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
151.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
    0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0],
152.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
    0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0],
153.                            [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0,
    0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0],
154.                            [0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,
    0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0],
155.                            [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
    0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0],
156.                            [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
    0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0],
157.                            [0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
158.                            [0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0],
159.                            [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
160.                            [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0],
161.                            [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
162.                            [0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
163.                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
164.
165.            labyrint2 = pixels
166.
167.            if num == 0: return labyrint0
168.            if num == 1: return labyrint1
169.            if num == 2: return labyrint2
```

Vedlegg G – VBR.py

Filen kan hentes fra GitHub:

[Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging/VBR.py at main · JEA-Navigation/Bachelor-Hvordan-kan-maskinl-ring-brukes-som-hjelpemiddel-til-ruteplanlegging (github.com)]

```
1.  #### INFORMASJON ####
2.  # Filnavn: VBR.py
3.  # Forfatter: Jone Måsøy, André Killingmoe og Eirik Kiland
4.
5.  # Beskrivelse:
6.  # Her lages brukergrensesnittet.
7.
8.  from runpy import run_path
9.  from telnetlib import theNULL
10. import numpy as np
11. import customtkinter
12. import geopy.distance
13. import numpy
14. import tkintermapview
15. from pyproj import Proj, transform
16. from Brukermain import run
17.
18.
19.
20. # CTkinter vinduet
21. VBR = customtkinter.CTk()
22. VBR.geometry(f"{800}x{600}")
23. VBR.title("JEA Navigation")
24.
25.
26. # Initialiserer listene og noen variabler vi bruker i programmet
27. Startpunktgrafisk = []
28. Mellompunktgrafisk = []
29. Sluttpunktgrafisk = []
30.
31. Startpunkt = []
32. Mellompunkt = []
33. Sluttpunkt = []
34.
35. EastList = []
36. NorthList = []
37.
38. Ruteliste = []
39. Rutegrafisk = []
40.
41. global Rute
42. Distanse = float
43. glider = int
44.
45.
46.
47. # Venstre vinduramme oppe og nede
48.
49. frame_left = customtkinter.CTkFrame(master = VBR,width=200)
50. frame_left.pack(fill="y", side="left")
51.
52. frame_left_low = customtkinter.CTkFrame(master = frame_left, width = 200)
53. frame_left_low.pack(side="bottom", fill="x")
```

```
54.
55.
56. # Funksjon for å velge type kart:
57.
58. def Endre_kart(Nytt_kart):
59.         if Nytt_kart == "OpenStreetMap":
60.             Kart.set_tile_server("https://a.tile.openstreetmap.org/{z}/{x}/{y}.png")
61.         elif Nytt_kart == "Vanlig kart":
62.             Kart.set_tile_server("https://mt0.google.com/vt/lyrs=m&hl=en&x={x}&y={y}
    &z={z}&s=Ga", max_zoom=22)
63.         elif Nytt_kart == "Satelittkart":
64.             Kart.set_tile_server("https://mt0.google.com/vt/lyrs=s&hl=en&x={x}&y={y}
    &z={z}&s=Ga", max_zoom=22)
65.
66. Kartmodus = customtkinter.CTkLabel(frame_left_low,
    text="Kartmodus:").place(x=1,y=10)
67.
68. Kartvelger = customtkinter.CTkOptionMenu(frame_left_low, values=["OpenStreetMap",
    "Vanlig kart", "Satelittkart"],
69.                                                                           command=Endre
    _kart).place(x=30,y=40)
70.
71.
72. # Funksjonen som fjerner markørene. marker.delete() fjerner markørene grafisk,
    .clear() fjerner innholdet fra listene.
73. def fjernstartmarkor():
74.     for marker in Startpunktgrafisk:
75.             marker.delete()
76.     for marker in Mellompunktgrafisk:
77.             marker.delete()
78.     Startpunkt.clear()
79.     Mellompunkt.clear()
80.
81. def fjernsluttmarkor():
82.     for marker in Sluttpunktgrafisk:
83.             marker.delete()
84.     Sluttpunkt.clear()
85.
86.
87.
88. # Deklarerer menyoverskrift og knappene som ligger i venstre ramme
89.
90. Overskrift = customtkinter.CTkLabel(text="Meny", text_font=("Roboto Medium", -16),
    corner_radius=10).place(x=30,y=10)
91.
92. Fjernstartmarkør = customtkinter.CTkButton(master=frame_left,
93.                                             text="Fjern start", width=40,
94.                                             command=fjernstartmarkor)
95. Fjernstartmarkør.place(y=50, x=20)
96.
97.
98. Fjernsluttmarkør = customtkinter.CTkButton(master=frame_left,
99.                                             text="Fjern slutt", width=40,
100.                                                   command=fjernsluttmarkor)
101.       Fjernsluttmarkør.place(y=50, x=100)
102.
103.
104.       # Funksjonen som fjerner ruten, og resten av knappene tilhørende ruten man
    fjerner, dersom brukeren velger lag ny rute
105.
106.       def slett_gammel_rute():
107.           for marker in Sluttpunktgrafisk:
108.                   marker.delete()
109.           for marker in Startpunktgrafisk:
110.                   marker.delete()
111.           for marker in Mellompunktgrafisk:
```

```
112.                    marker.delete()
113.            Startpunkt.clear()
114.            Mellompunkt.clear()
115.            Sluttpunkt.clear()
116.
117.            for rute in Rutegrafisk:
118.                rute.delete()
119.
120.            Ny_rutelager.place_forget()
121.            Avstandslabel.place_forget()
122.            NMlabel.place_forget()
123.            Tidlabel.place_forget()
124.            Fart.place_forget()
125.            Fartsvariabel.place_forget()
126.            Beregningsknapp.place_forget()
127.            Slider.place_forget()
128.            Avstanden.place_forget()
129.            Tidsvariabel.place_forget()
130.
131.        # Funksjonen som gjør om start- og sluttmarkør til koordinater og legger de i
    npy. fil.
132.
133.        def lag_rute():
134.            StartpunktX, StartpunktY = zip(*Startpunkt)
135.            SluttpunktX, SluttpunktY = zip(*Sluttpunkt)
136.
137.
138.            inProj = Proj(proj='longlat')
139.            outProj = Proj(proj='utm', zone=33, ellps='WGS84')
140.
141.            # initialiserer listene fyllt med 0er med like mange verdier som det
    finnes koordinater
142.            # i ruta
143.            startx2 = numpy.array(StartpunktX) #startpunktX
144.            starty2 = numpy.array(StartpunktY) #startpunktY
145.            sluttx2 = numpy.array(SluttpunktX) #sluttpunktX
146.            slutty2 = numpy.array(SluttpunktY) #sluttpunktY
147.
148.            # Transformerer hver koordinat fra grader til xyz
149.            outstart = transform(inProj,outProj,starty2,startx2)
150.            outslutt = transform(inProj,outProj,slutty2,sluttx2)
151.
152.            # Legger de transformerte gradene i en variable
153.            printstartarray = numpy.column_stack((outstart[1], outstart[0]))
154.            printsluttarray = numpy.column_stack((outslutt[1], outslutt[0]))
155.
156.            # Lagrer de transformerte gradene inn i hver sin numpyfil
157.            numpy.save('printstartarray.npy', printstartarray)
158.            numpy.save('printsluttarray.npy', printsluttarray)
159.
160.            # Printer for bekreftelse
161.            datastart = numpy.load('printstartarray.npy')
162.            dataslutt = numpy.load('printsluttarray.npy')
163.
164.            print(datastart)
165.            print(dataslutt)
166.
167.            epoch = 0
168.            while epoch == 0:
169.                policy, s = run(epoch,printstartarray,printsluttarray)
170.                action = policy.get(s)
171.                print("recommended action:")
172.                print(action)
173.                epoch += 1
174.
175.
```

```
176.
177.
178.        # Knapp som igangsetter lag_rute-funksjonen
179.
180.        Rutelager = customtkinter.CTkButton(master=frame_left,
181.                                             text="Bekreft start/slutt",
    width=155,
182.                                             command=lag_rute)
183.        Rutelager.place(y=90, x=20)
184.
185.
186.        # Knapp for å lage ny rute
187.
188.        Ny_rutelager = customtkinter.CTkButton(master=frame_left,
189.                                             text="Lag ny rute", width=155,
190.                                             command=slett_gammel_rute)
191.
192.
193.        # Funksjonen som tegner ruten når algoritmen har lagd den ferdig
194.        def Ruten_tilbake():
195.
196.            global Distanse
197.            global Rute
198.            global NMlabel
199.            global Tidlabel
200.            global Fart
201.            global Beregningsknapp
202.            global Avstandslabel
203.            global Avstanden
204.            global OverfDistanse
205.            Ruteliste = numpy.load('Ruta.npy')
206.            Kartrute = Ruteliste
207.
208.        # Finner distansen til ruta basert på avstanden mellom hvert punkt er.
209.
210.            Distanse = 0
211.            for i in range(len(Kartrute)):
212.                if i == len(Kartrute) - 1:
213.                    break
214.                koordinat1 = Kartrute[i]
215.                koordinat2 = Kartrute[i+1]
216.                EnkelDistanse = geopy.distance.geodesic(koordinat1, koordinat2).km
217.                Distanse = Distanse + EnkelDistanse
218.
219.            # Begrenser distansen til to desimaler
220.            Distanse = format(Distanse, ".2f")
221.
222.            # Denne legger ruten i en liste og tegner den
223.            Rutegrafisk.append(Kart.set_path(position_list=(Kartrute)))
224.
225.        #Linjene under navngir og plasserer alle knapper og labels som skal frem
    etter funksjonen har kjørt
226.
227.            Ruteoptimaliserer = customtkinter.CTkLabel(text="Ruteoptimaliserer",
    text_font=("Roboto Medium", -11), corner_radius=10).place(x=25,y=190)
228.
229.
230.            Slider.place(x=10,y=220)
231.            Slider.set(0)
232.
233.            Avstanden = customtkinter.CTkLabel(master=frame_left,
234.                                                   text="Avstand:",
235.                                                   height=20, width=80,
236.                                                   corner_radius=6,fg_color=(
    "white", "gray38"))
237.            Avstanden.place(x=10, y=250)
```

```
238.
239.            Avstandslabel = customtkinter.CTkLabel(master=frame_left, text=Distanse,
240.                                                   height=20, width=40,
241.                                                   corner_radius=6)
242.            Avstandslabel.place(x=100, y=250)
243.
244.            NMlabel = customtkinter.CTkLabel(master=frame_left, text="NM",
245.                                                   height=20, width=20,
246.                                                   corner_radius=6)
247.            NMlabel.place(x=140, y=250)
248.
249.
250.            Tidlabel = customtkinter.CTkLabel(master=frame_left,
251.                                                   text="Tid:",
252.                                                   height=20, width=80,
253.                                                   corner_radius=6,
    fg_color=("white", "gray38"))
254.            Tidlabel.place(x=10, y=280)
255.
256.
257.            Fart = customtkinter.CTkLabel(master=frame_left, text="Fart:",
258.                                                height=20, width=80,
259.                                                corner_radius=6,  fg_color=("
    white", "gray38"))
260.            Fart.place(x=10, y=310)
261.
262.            Ny_rutelager.place(y=130, x=20)
263.            Fartsvariabel.place(x=100, y=310)
264.
265.            Beregningsknapp = customtkinter.CTkButton(text="Kalkuler tid",
    command=Kalkuler_tid)
266.            Beregningsknapp.place(x=30,y=340)
267.
268.
269.        Fartsvariabel = customtkinter.CTkEntry(master=frame_left,
270.                                           height=20, width=80,corner_radius=6)
271.
272.
273.        #Funksjonen som gjør at glidebryteren fungerer. Denne avgjør hvor mange
    punkter som skal være med i ruten.
274.
275.        def Endreglider(glider):
276.
277.            # Det første denne gjør er å fjerne den gamle ruten. Dette skjer hver
    gang brukeren rører på glidebryteren
278.            global OverfDistanse
279.            Rutegrafisk.clear
280.            for Rute in Rutegrafisk:
281.                    Rute.delete()
282.
283.            # Glider = 0 vil ta med alle punktene i ruten. Denne vil bli veldig
    svingete og dermed urealistisk lang
284.            if glider == 0:
285.                Ruteliste = numpy.load('Ruta.npy')
286.                kartrute = Ruteliste
287.
288.
289.                Distanse = 0
290.                for i in range(len(kartrute)):
291.                    if i == len(kartrute) - 1:
292.                        break
293.                    koordinat1 = kartrute[i]
294.                    koordinat2 = kartrute[i+1]
295.                    EnkelDistanse = geopy.distance.geodesic(koordinat1,
    koordinat2).km
296.                    Distanse = Distanse + EnkelDistanse
```

```
297.
298.                 Distanse = format(Distanse, ".2f")
299.                 # Lager ruten på nytt med nytt antall punkter.
300.                 Rutegrafisk.append(Kart.set_path(position_list=(kartrute)))
301.
302.                 Avstandslabel.configure(text=Distanse)
303.                 OverfDistanse = Distanse
304.
305.             # Glider = 1 vil kun beholde hvert 4. punkt
306.             elif glider == 1:
307.                 # Starter med å laste inn ruten fra numpyfilen
308.                 Ruteliste = numpy.load('Ruta.npy')
309.                 # [0::4] sier at den skal starte på 0. punkt (Dette er 1. punkt) og
    dermed hente ut hver 4. verdi fra listen.
310.                 kartrute = Ruteliste[0::6]
311.                 # Setter den nye lengden til Ruteliste inn i variabelen
    kartrute_lengde
312.                 kartrute_lengde = len(Ruteliste)
313.                 # Henter ut det siste punktet i ruten og legger den i variabelen
    DestinasjonsKartrute
314.                 DestinasjonKartrute = Ruteliste[kartrute_lengde - 1]
315.                 # Legger nye ruten inn i kartrute sammen med siste punktet i ruten.
    Denne er viktig å legge til ettersom man hopper over så mange punkter i ruten.
316.                 # Har man den ikke med kan sluttdestinasjonen bli helt feil sted.
317.                 kartrute = numpy.append(kartrute, [DestinasjonKartrute], axis=0)
318.
319.
320.
321.                 Distanse = 0
322.                 for i in range(len(kartrute)):
323.                     if i == len(kartrute) - 1:
324.                         break
325.                     koordinat1 = kartrute[i]
326.                     koordinat2 = kartrute[i+1]
327.                     EnkelDistanse = geopy.distance.geodesic(koordinat1,
    koordinat2).km
328.                     Distanse = Distanse + EnkelDistanse
329.
330.                 Distanse = format(Distanse, ".2f")
331.
332.                 Rutegrafisk.append(Kart.set_path(position_list=(kartrute)))
333.
334.                 Avstandslabel.configure(text=Distanse)
335.
336.                 OverfDistanse = Distanse
337.
338.             # Glider = 2 vil kun beholde hvert 8. punkt
339.             elif glider == 2:
340.                 Ruteliste = numpy.load('Ruta.npy')
341.                 kartrute = Ruteliste[0::12]
342.                 kartrute_lengde = len(Ruteliste)
343.                 DestinasjonKartrute = Ruteliste[kartrute_lengde - 1]
344.                 kartrute = numpy.append(kartrute, [DestinasjonKartrute], axis=0)
345.
346.                 Distanse = 0
347.                 for i in range(len(kartrute)):
348.                     if i == len(kartrute) - 1:
349.                         break
350.                     koordinat1 = kartrute[i]
351.                     koordinat2 = kartrute[i+1]
352.                     EnkelDistanse = geopy.distance.geodesic(koordinat1,
    koordinat2).km
353.                     Distanse = Distanse + EnkelDistanse
354.
355.                 Distanse = format(Distanse, ".2f")
356.
```

```
357.                Rutegrafisk.append(Kart.set_path(position_list=(kartrute)))
358.
359.                Avstandslabel.configure(text=Distanse)
360.                OverfDistanse = Distanse
361.
362.            # Glider = 3 vil kun beholde hvert 12. punkt
363.            elif glider == 3:
364.                Ruteliste = numpy.load('Ruta.npy')
365.                kartrute = Ruteliste[0::18]
366.                kartrute_lengde = len(Ruteliste)
367.                DestinasjonKartrute = Ruteliste[kartrute_lengde - 1]
368.                kartrute = numpy.append(kartrute, [DestinasjonKartrute], axis=0)
369.
370.
371.                Distanse = 0
372.                for i in range(len(kartrute)):
373.                    if i == len(kartrute) - 1:
374.                        break
375.                    koordinat1 = kartrute[i]
376.                    koordinat2 = kartrute[i+1]
377.                    EnkelDistanse = geopy.distance.geodesic(koordinat1,
    koordinat2).km
378.                    Distanse = Distanse + EnkelDistanse
379.
380.                Distanse = format(Distanse, ".2f")
381.
382.                Rutegrafisk.append(Kart.set_path(position_list=(kartrute)))
383.
384.                Avstandslabel.configure(text=Distanse)
385.                OverfDistanse = Distanse
386.
387.            # Glider = 4 vil kun beholde hvert 16. punkt
388.            elif glider == 4:
389.                Ruteliste = numpy.load('Ruta.npy')
390.                kartrute = Ruteliste[0::24]
391.                kartrute_lengde = len(Ruteliste)
392.                DestinasjonKartrute = Ruteliste[kartrute_lengde - 1]
393.                kartrute = numpy.append(kartrute, [DestinasjonKartrute], axis=0)
394.
395.
396.                Distanse = 0
397.                for i in range(len(kartrute)):
398.                    if i == len(kartrute) - 1:
399.                        break
400.                    koordinat1 = kartrute[i]
401.                    koordinat2 = kartrute[i+1]
402.                    EnkelDistanse = geopy.distance.geodesic(koordinat1,
    koordinat2).km
403.                    Distanse = Distanse + EnkelDistanse
404.
405.                Distanse = format(Distanse, ".2f")
406.
407.                Rutegrafisk.append(Kart.set_path(position_list=(kartrute)))
408.
409.                Avstandslabel.configure(text=Distanse)
410.                OverfDistanse = Distanse
411.
412.            # Glider = 5 vil kun beholde hvert 20. punkt
413.            elif glider == 5:
414.                Ruteliste = numpy.load('Ruta.npy')
415.                kartrute = Ruteliste[0::30]
416.                kartrute_lengde = len(Ruteliste)
417.                DestinasjonKartrute = Ruteliste[kartrute_lengde - 1]
418.                kartrute = numpy.append(kartrute, [DestinasjonKartrute], axis=0)
419.
420.
```

```
421.                Distanse = 0
422.                for i in range(len(kartrute)):
423.                    if i == len(kartrute) - 1:
424.                        break
425.                    koordinat1 = kartrute[i]
426.                    koordinat2 = kartrute[i+1]
427.                    EnkelDistanse = geopy.distance.geodesic(koordinat1,
    koordinat2).km
428.                    Distanse = Distanse + EnkelDistanse
429.
430.                Distanse = format(Distanse, ".2f")
431.
432.                Rutegrafisk.append(Kart.set_path(position_list=(kartrute)))
433.
434.                Avstandslabel.configure(text=Distanse)
435.                OverfDistanse = Distanse
436.
437.
438.        # definerer slideren og legger sliderverdien inn i variabelen glider nedenfor
439.        Slider = customtkinter.CTkSlider(master=frame_left,
440.                                    command=Endreglider,
441.                                    from_=0, to=5, number_of_steps=5,
442.                                    width=170)
443.        glider = Slider.get()
444.
445.
446.        # Funksjonen som kalkulerer tiden det vil ta å seile ruten. Fart er input fra
    brukeren.
447.        def Kalkuler_tid():
448.
449.            if OverfDistanse != 0:
450.                Distanse = OverfDistanse
451.
452.            knop = Fartsvariabel.get()
453.            Fart = float(knop)
454.            Avstand = float(Distanse)
455.
456.            Tid = Avstand / Fart
457.            Antallminutter = Tid * 60
458.            Antallsekunder = ((Antallminutter % 1)*100) * 60/100
459.            AM = int(Antallminutter)
460.            AS = int(Antallsekunder)
461.            Totaltid = (AM, "Min,", AS, "Sek")
462.            global Tidsvariabel
463.            Tidsvariabel = customtkinter.CTkLabel(master=frame_left, text=Totaltid,
464.                                                  height=20, width=80,
465.                                                  corner_radius=6)
466.            Tidsvariabel.place(x=100, y=280)
467.
468.
469.        # Toppramme
470.        frame_top = customtkinter.CTkFrame(master = VBR)
471.        frame_top.pack(fill="x", side="top")
472.
473.
474.        # Søkefunksjonen som tilhører søkebaren. Funksjonen som setter kartet over
    stedet man søker på.
475.        def Søkefunksjon(event=None):
476.            Kart.set_address(Søkefelt.get())
477.
478.
479.        Søkefelt = customtkinter.CTkEntry(master=frame_top,
480.                                    placeholder_text="Adresse..",
481.                                    width=450)
482.        Søkefelt.pack(padx=5, pady=10, side="left")
483.
```

```
484.        Søkefelt.entry.bind("<Return>", Søkefunksjon)
485.
486.        Søkeknapp = customtkinter.CTkButton(master=frame_top,
487.                                            text="Søk",
488.                                            width=90,
489.                                            command=Søkefunksjon)
490.        Søkeknapp.pack(side="left", padx=15)
491.
492.
493.
494.        # Høyre ramme
495.        frame_right = customtkinter.CTkFrame(master=VBR)
496.        frame_right.pack(fill="both", expand=True, side="right")
497.
498.        # Kartet
499.        Kart = tkintermapview.TkinterMapView(master=frame_right)
500.        Kart.pack(fill='both', expand=True)
501.
502.        # Sjøkartet (overlay)
503.        Kart.set_overlay_tile_server("http://tiles.openseamap.org/seamark//{z}/{x}/{y
    }.png")
504.
505.        # Setter startposisjon
506.        Kart.set_position(60.3888429, 5.3242385)  # Bergen
507.        Kart.set_zoom(10)
508.
509.
510.
511.
512.
513.
514.        # Høyreklikk funksjonen som setter startmarkør. Den vil først slette
    eventuelle markører som er satt fra før, slik at det kun er én startmarkør.
515.        # Dette skjer både grafisk og i listen markørene ligger i. Så printer den ut
    koordinatene og legger markøren i en liste.
516.        def add_startmarker_event(coords):
517.            for marker in Startpunktgrafisk:
518.                marker.delete()
519.                Startpunktgrafisk.clear()
520.                Startpunkt.clear()
521.            print("Markør - start:", coords)
522.            Startpunktgrafisk.append(Kart.set_marker(coords[0], coords[1],
    text="Start"))
523.            Startpunkt.append(coords)
524.
525.        Kart.add_right_click_menu_command(label="Startposisjon",
526.                                               command=add_startmarker_event,
527.                                               pass_coords=True)
528.
529.
530.        # Høyreklikk funksjonen som setter mellommarkør
531.        def add_mellommarker_event(coords):
532.                print("Markør - Mellomstopp:", coords)
533.                Mellompunktgrafisk.append(Kart.set_marker(coords[0], coords[1],
    text="Mellomstopp"))
534.                Mellompunkt.append(coords)
535.
536.
537.        Kart.add_right_click_menu_command(label="Mellomstopp",
538.                                               command=add_mellommarker_event,
539.                                               pass_coords=True)
540.
541.        # Høyreklikk funksjonen som setter sluttmarkør
542.        def add_sluttmarker_event(coords):
543.                for marker in Sluttpunktgrafisk:
544.                    marker.delete()
```

```
545.                    Sluttpunktgrafisk.clear()
546.                    Sluttpunkt.clear()
547.                print("Markør - destinasjon:", coords)
548.                Sluttpunktgrafisk.append(Kart.set_marker(coords[0], coords[1],
    text="Destinasjon"))
549.                Sluttpunkt.append(coords)
550.
551.        Kart.add_right_click_menu_command(label="Sluttposisjon",
552.                                          command=add_sluttmarker_event,
553.                                          pass_coords=True)
554.
555.
556.        # Høyreklikk funksjonen som aktiverer "Ruten_tilbake". Denne må brukes når
    algoritmen har lagd ruten
557.
558.        Kart.add_right_click_menu_command(label="Tegn rute",
559.                                   command=Ruten_tilbake)
560.
561.
562.        VBR.mainloop()
```