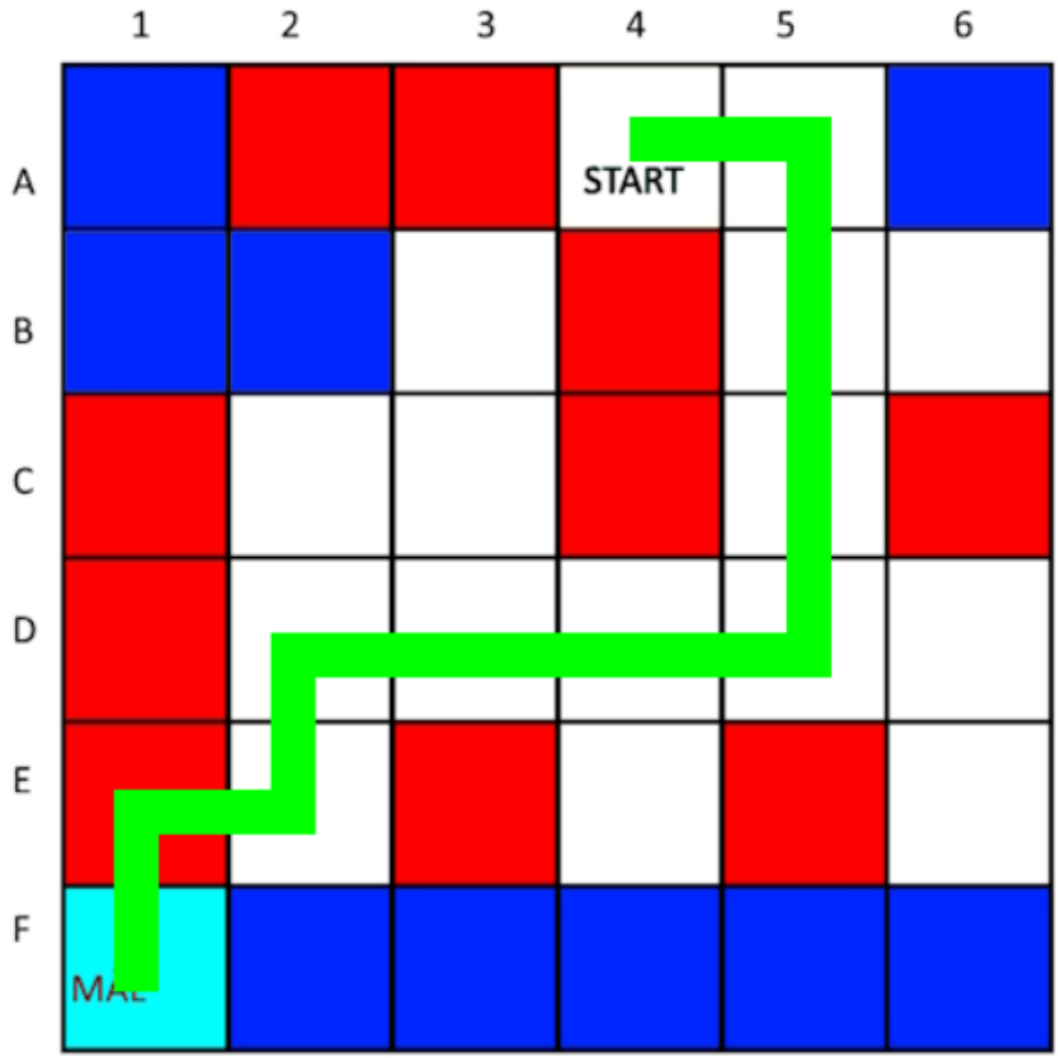
Traversering av ukjent terreng ved hjelp av Q-Learning



Snorre A. Solberg

Innholdsfortegnelse

Figurliste

Tabelliste

Kodeutdragsliste

1 Introduksjon

Hovedtema i denne rapporten er bruk av Q-learning til finne den mest effektive veien for en robot å traversere et ukjent terreng, men innledningsvis undersøkes det også hvor enkelt det er å finne den beste ruten ved hjelp av Monte Carlo simulering. Terrenget roboten trenes på er delt inn i et rutenett på 36 ruter. Roboten har evnen til å traversere alle rutene i terrenget, men noen av rutene er vanskeligere å komme seg gjennom enn andre, og betegnes da som å ha en høyere kostnad. Hindringene i rutene består av bratt/ulendt terreng og vann, hvorav vann regnes som farligere grunnet ukjente trusler som beveger seg under vannflaten. Selve roboten kan bevege seg i retning opp, ned, høyre og venstre, altså ikke diagonalt. Programmet rapporten handler om gjennomfører etter en suksessfull Q-learning prosess en visualisering av den mest effektive ruten gjennom terrenget ved hjelp av Python-modulen pygame.

2 Teori

Denne teoridelen beskriver i hovedsak Q-learning, som er en type Reinforcement Learning, og konseptene det bygger på. Kritisk for funksjonaliteten er for eksempel Markov-prosesser, Monte Carlo Metoden og Bellman-likningen. Det forutsettes i denne rapporten at leseren har en grunnleggende forståelse for programmeringsspråket Python og dets funksjonalitet, og det vil derfor ikke gjennomgås i denne seksjonen.

2.1 Reinforcement Learning

Reinforcement Learning er en av tre hovedretninger innen maskinlæring. Det består av en *agent* (eller flere), ett sett med tilstander (*state*) den kan være i og et sett med tillatte handlinger (*action*) per tilstand. Når agenten utfører en handling i en tilstand veksler den til en annen tilstand, dette kalles et steg. For hvert tilstand/handling par er det assosiert en numerisk belønning (*reward*). Disse belønningsverdiene er lagret i det som kalles en Reward-matrise, hvor hver rad er en tilstand og hver kolonne er en handling. Verdt å merke seg er at det også er mulig å gi negativ belønning for uønskede handlinger.

En viktig forutsetning for Reinforcement learning er at oppgaven kan modelleres som en *Markov Decision Process* (MDP), som er et matematisk rammeverk for å modellere beslutninger. En sentral egenskap ved denne prosessen er at Markov-egenskapen, som sier at fremtiden kun er avhengig av nåtiden, og er da altså fullstendig uavhengig av fortiden.[1] I MDP-modellen er hver node en tilstand, som kan være fysiske (som posisjon) eller mer abstrakte (som værtilstander). For å endre tilstand må det utføres en handling, representert ved kantene mellom nodene. Hvilke handlinger som er mulige å utføre, kan variere mellom tilstandene. Markov-egenskapen innebærer at hvordan man havnet i nåværende tilstand er irrelevant; det eneste som betyr noe videre, er hvilke handlinger man kan foreta seg. Når man kombinerer en MDP med en Reward-matrise, får man en Markov Reward Process (MRP).

Agentens mål i Reinforcement Learning er å maksimere den totale belønningen den får på veien fra utgangstilstanden til måltilstanden. For å gjøre dette tar agenten i betraktning den maksimalt oppnåelige belønningen for fremtidige tilstander. Dette er en vektet sum av de forventede belønningene for alle fremtidige steg, med utgangspunkt i nåværende tilstand.[2]

2.2 Q-Learning

Q-learning er type *model free* reinforcement learning. Model free betyr at agenten ikke kjenner til miljøet fra før av. Den må ved hjelp av en *policy* utforske miljøet for å finne den beste veien til mål. I Q-learning benyttes generelt tre typer policy: Monte Carlo, Greedy (Grådig) og Epsilon-Greedy, som beskrives i mer detalj i neste avsnitt. Noe som er viktig å merke seg er at Q-learning er *off-policy*, dette betyr at man ikke er tvunget til bruke samme policy gjennom hele prosessen. Det kan for eksempel benyttes én policy som fokuserer på utforskning under trening og én policy som utnytter denne informasjonen til å finne den mest effektive ruten når behovet for utforskning er tilfredsstilt.

Informasjonen agenten innhenter når den utforsker lagres i det som kalles en Q-matrise. Denne matrisen beskriver den mest lønnsomme handlingen å utføre i hver tilstand, og har samme dimensjoner som Reward-matrisen. Q-matrisen initialiseres gjerne med samme verdi i alle posisjoner, for eksempel 0. Verdiene oppdateres ved hjelp av Bellman-likningen, vist i likning 1.

Noe som er litt annerledes fra det som defineres i 2.1 i praksis er at i Q-learning er det kun belønningen for gjeldene tilstand/handling par pluss den høyeste oppnåelige belønningen i neste tilstand som tas i betraktning i hver utregning av ny Q-verdi. Dette betyr at de fremtidige tilstandene utover dette ikke er kjent i dette steget. Med denne virkemåten vil belønningen fra måltilstanden allikevel forplante seg bakover mot starttilstanden etterhvert som agenten utforsker området og modifiserer Q-matrisen steg for steg, og dette vil føre til at den mest effektive veien har de høyeste Q-verdiene.

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot max_{a'} Q(s',a')) \tag{1}$$

Der

- s er nåværende tilstand
- a er nåværende valgte handling
- s' er neste tilstand
- a' er neste handling
- α er *Learning Rate* (Læringsrate)
- γ er *Discount Factor* (Diskonteringsfaktor)

Denne ligningen introduserer parametrene α og γ, som kalles *hyperparametre*. Disse parametrene kontrollerer hvor hurtig agenten lærer og hvor mye fremtidige belønninger påvirker nåværende beslutning. α representerer andelen av den nye Q-verdien som kommer fra den utregnede nye verdien, i forhold til den eksisterende Q-verdien, altså kontrollerer den direkte hvor hurtig agenten kan lære for hver handling den utfører. γ bestemmer hvor optimistisk agenten skal være til at fremtiden bringer belønninger. Dette velger altså hvor stor innvirkning beste handling i neste tilstand skal ha på Q-verdien for nåværende tilstand/handling par.[3]

En høy verdi for α gjør at agenten hurtig kan tilpasse seg endringer, som for eksempel endrede belønninger eller endringer i miljøet den utforsker. Dette er lønnsomt om det er aktuelt med slike endringer i et *dynamisk miljø*, men om agenten opererer i det som kalles et *statisk miljø*, altså et miljø som ikke kommer til å endre seg, er det bedre å bruke en lavere verdi for α. Når det gjelder γ, i et dynamisk miljø vil ikke fremtidige belønninger alltid være like sikre, og det vil være logisk for agenten å være skeptisk til dette. I et statisk miljø er fremtidige belønninger forutsigbare, og da lønner det seg også å ha en høy verdi for γ, slik at agenten forsøker å effektivt jobbe mot denne langsiktige belønningen.

2.3 Policy

Monte Carlo Metoden er en type algoritme som benytter tilfeldige verdier i prosessen for å estimere en numerisk løsning på et matematisk problem.[4] Det kan for eksempel brukes til å utforske en MDP for å finne hvilken tilstand agenten mest sannsynlig vil ende opp i etter et visst antall steg. Dette gjøres ved å gjennomføre t steg flere ganger for å danne et statistisk grunnlag det kan beregnes sannsynligheten for hver sluttilstand utifra. I reinforcement learning kan det for eksempel brukes til å la en agent utforske et landskap/ en labyrint for å finne mest lønnsomme vei gjennom. Man vil da ikke nødvendigvis limitere antall steg, men heller se etter maksimal oppnådd belønning når agenten ankommer målområdet.

Greedy algorithm er en type algoritme som forsøker å løse problemer ved å utføre den lokalt beste handlingen i en viss tilstand. Den vil da altså velge handlingen som fra den gjeldene tilstanden gir høyest belønning. Nedsiden med denne typen algoritme er at den blir kortsiktig, den fokuserer på oppnå størst mulig belønning i neste steg, i motsetning til å jobbe langsiktig mot en større belønning på et senere tidspunkt. Det kan sammenlignes med å øse vann ut av en båt istedenfor å tette hullet når den lekker. Hvis det gis positiv belønning for å redusere vannmengden vil det gis en liten belønning ofte, men den langsiktige løsningen hadde åpenbart vært å tette hullet slik at det ikke trenger inn mer vann.

Epsilon-Greedy er en kombinasjon av Monte Carlo og Greedy policy. Det brukes en parameter ϵ, som er en sannsynlighets-parameter, til å velge mellom *Exploration* (utforskning) eller *Exploitation* (Utnyttelse). På grunn av denne balansen mellom Exploration og Exploitation er denne metoden ofte foretrukket over de to andre i praksis. Det genereres en tilfeldig verdi mellom 0 og 1, om dette tallet er større enn ϵ vil det utføres en tilfeldig handling (Exploration). Ellers vil det utføres en greedy handling (Exploitation). Dermed kan ϵ defineres som andelen av handlingene som utføres tilfeldig. I en Q-learning prosess vil det vanligvis være lønnsomt å bedrive mest utforskning i starten av prosessen for så å gjøre en større andel informerte valg når mer informasjon er tilgjengelig. En vanlig måte å utføre dette på er med en *decay-funksjon*, som reduserer ϵ lineært eller eksponentielt for hver epoke.

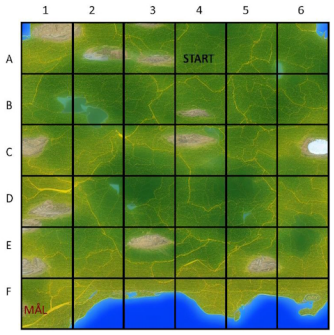
3 Metode

Tolkninger som ligger til grunn for arbeidet beskrevet i denne rapporten er at det er mye farligere å kjøre under vann enn gjennom ulendt terreng, og at vanlige ruter uten utfordrende terreng også har en kostnad grunnet for eksempel tidsbruk, men den er da liten sett i forhold til de to andre typene terreng. Forsøk på å forlate det definerte området straffes med høy negativ belønning og ingen forflytning.

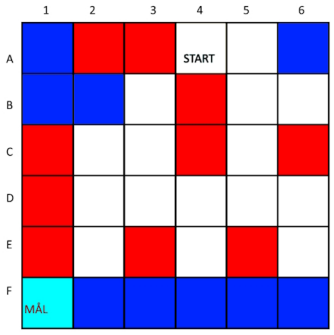
3.1 Utarbeiding av Reward-Matrise

Terrenget roboten skal traversere er vist i figur 1. Utifra dette er det utarbeidet et nytt kart som kun inneholder fargekodet informasjon om hvor krevende terrenget er, vist i figur 2. Her er ruter med vann blå, ruter med ulendt terreng røde, ruter uten spesielle merknader hvite, og målruten lys blå. Med utgangspunkt i tolkningene beskrevet ovenfor og dette nye kartet ble det besluttet at kun måltilstanden skulle gi positiv belønning, og denne ble satt til 50. Blå ruter skal ha en kostnad på -15, røde -5 og hvite -1. Straffen for å forsøke å forlate det definerte området er -100, og roboten vil da også bli stående i samme tilstand.

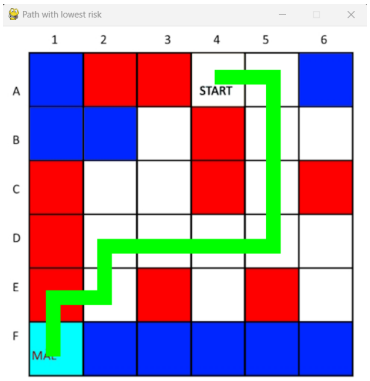
Reward-matrisen vil være en 36x4 matrise, siden det er 36 mulige tilstander og 4 mulige handlinger å utføre i hver av dem. Med rot i denne Reward-matrisen er det mulig å definere den beste ruten gjennom terrenget, som en slags fasit for roboten. Denne ruten er vist i figur 3, og vil gi roboten en *score* på 37. Dette betyr at under disse forholdene er det ikke mulig for roboten å oppnå en høyere score enn 37.



Figur 1: Terrenget roboten skal traversere.



Figur 2: Kart over terrenget fargekodet etter kostnad.



Figur 3: Beste rute basert på gjeldene kriterier.

3.2 Hyperparametre

Denne Q-learning prosessen benytter fem hyperparametere: α, γ, ϵ, min_ϵ og $decay\_rate$. Siden terrenget ikke vil endre seg underveis i prosessen kan miljøet defineres som statisk. Som beskrevet i seksjon 2.2 i teoridelen vil det i et slikt miljø lønne seg å benytte en lav α og en høy γ. ϵ beskriver forholdet mellom Exploration og Exploitation. Når roboten starter har den ingen informasjon om miljøet, og det er derfor lønnsomt med mer Exploration, men senere i prosessen er det mer lønnsomt å utføre en større andel handlinger basert på tillegnet kunnskap. På grunn av dette er det besluttet å starte med en høy verdi for ϵ, som reduseres eksponentielt for hver Q-learning epoke ved hjelp av en decay-funksjon. Hvordan $decay\_rate$ og min_ϵ knytter inn i dette beskrives nærmere i forklaringen av metodene i Robot klassen i seksjon 3.3.1.

Tabell 1: Hyperparameter verdier benyttet til Q-learning.

Parameter	Verdi
α	0.1
γ	0.8
ϵ	1
min_ϵ	0.1
$decay\_rate$	0.995

3.3 Programmering

3.3.1 Klassen Robot

Attributter

- De fem hyperparametrene
- y: Robotens y-koordinat
- x: Robotens x-koordinat
- best_path: En liste som inneholder den beste ruten roboten har funnet
- best_score: Scoren til den beste ruten
- r_matrix: Reward-matrisen som beskriver miljøet
- q_matrix: Robotens Q-matrise

Metoder

Beskrivelsen av klassens metoder er kopiert direkte fra klassens docstring, som er skrevet for å være tilstrekkelig med informasjon til å forstå virkningen av hver metode, samt hvilke input/output variabler de har. Forklaringen av metodene er oppgitt på engelsk, da dette er en vanlig konvensjon innen programmering. En ting verdt å merke seg er at når en metode nevner en tilstands *index* er dette fordi R/Q-matrisene er bygget opp som en liste av lister, og derfor vil hver tilstand aksesseres ved bruk av indeksering fra 0-35 (36 tilstander). Dette fremkommer tydeligere om man ser på koden, derfor er det ikke beskrevet ytterligere i docstringen.

```
Methods
-------
set_next_state(action: int) -> int:
    Updates robot position based on action (up, down, right, left)
    Checks if action is legal and returns the new states index.

get_action_greedy(state: int) -> int:
    Returns the action with the highest Q-value for current state.
    If multiple actions have same value, selects one of them
    randomly.

get_action_mc() -> int:
    Randomly selects and returns an action (0-3), simulating Monte
    Carlo policy.

get_action_eg(state: int) -> int:
    Returns Monte Carlo or greedy action based on epsilon value.

epsilon_update() -> None:
    Decreases epsilon by decay_rate towards min_epsilon.

monte_carlo_exploration(epochs: int) -> None:
    Explores terrain with Monte Carlo policy. Starts in (0, 3),
    resets in (5, 0). Repeats for specified number of epochs, then
    prints best score achieved.

one_step_q_learning() -> None:
    Executes one step of q-learning and updates the q-matrix.

q_learning(epochs: int) -> None:
    Runs q_learning for specified number of epochs. Starts epoch in
    random state, ends epoch when reaching (5, 0). Reduces epsilon
    for each epoch.

has_reached_goal() -> bool:
    Returns True if target state == (5, 0) is reached.

reset_random() -> None:
    Resets robot position to random state.

get_best_path() -> None:
    Finds best path and score using a greedy policy on the
    q-matrix.
```

Tre av disse metodene er ganske sentrale for oppgaven og kan beskrives litt nærmere:

monte_carlo_exploration

```python
def monte_carlo_exploration(self, epochs: int) -> None:
    scores = []
    for _ in range(epochs):
        self.y = 0
        self.x = 3
        sum_reward = 0
        finished = False
        while not finished:
            action = self.get_action_mc()
            sum_reward += self.r_matrix[6 * self.y + self.x][action]
            self.set_next_state(action)
            finished = self.has_reached_goal()
        scores.append(sum_reward)
    print(max(scores))
    return None
```

Kodeutdrag 1 : monte_carlo_exploration kodeutdrag.

Denne metoden brukes til å teste utforskning av terrenget ved hjelp av Monte Carlo simulering. Av kodeutdrag 1 kan det sees hvordan agenten for hvert steg velger en tilfeldig handling, får belønningen for handlingen, oppdaterer robotens posisjon og sjekker om den nye posisjonen er måltilstanden. Når måltilstanden nås legges den totale belønningen oppnådd fra starttilstand til måltilstand til i listen scores. Etter et valgt antall repetisjoner av denne prosessen printes den største verdien i scores til terminalen.

one_step_q_learning

```python
def one_step_q_learning(self) -> None:
    current_state = self.y * 6 + self.x
    action = self.get_action_eg(current_state)
    next_state = self.set_next_state(action)
    reward = self.r_matrix[current_state][action]
    self.q_matrix[current_state][action] = (
        (1 - self.alpha) * self.q_matrix[current_state][action] + self.alpha
        * (reward + self.gamma * max(self.q_matrix[next_state]))
    )
    return None
```

Kodeutdrag 2 : one_step_q_learning kodeutdrag.

Denne metoden brukes til å utføre hvert enkelt steg med Q-learning. I kodeutdrag 2 kan det sees at roboten velger en handling basert på Epsilon-Greedy policy, oppdaterer posisjonen, sjekker belønningen for handlingen og til sist oppdaterer Q-matrisen.

q_learning

```python
def q_learning(self, epochs: int) -> None:
    for _ in range(epochs):
        self.reset_random()
        while not self.has_reached_goal():
            self.one_step_q_learning()
        self.epsilon_update()
    return None
```

Kodeutdrag 3 : q_learning kodeutdrag.

Denne metoden utfører hovedprosessen i Q-learning. Kodeutdrag 3 viser hvordan roboten for hver epoke resettes til en tilfeldig posisjon, så lenge målet ikke er nådd gjennomføres ett steg med Q-learning, når målet nås oppdateres ϵ.

3.4 Utførelse og visualisering av Q-learning

I en egen fil kalt q-learning.py initialiseres roboten med valgte hyperparametre og Q-learning kjøres for valgt antall epoker. Videre finnes den beste ruten og tilhørende score ved hjelp av get_best_path, som printes til terminalen, og så visualiseres denne ruten ved hjelp av pygame.

4 Resultater

I denne seksjonen presenteres resultatene av de ulike testene.

4.1 Monte Carlo Exploration

Tabell 2: Score oppnådd ved bruk av Monte Carlo

Iterasjoner	Score
10	-381
100	-3
1000	25
10000	33
100000	35

4.2 Q-learning

Det er ikke mye resultat å vise til for Q-learning annet enn den animerte visualiseringen av den beste ruten.[5] Alle tilfeller der det kjøres nok epoker av Q-learning til at programmet ikke havner i en *deadlock* situasjon i get_best_path resulterer i at roboten finner den beste veien, som definert i figur 3. Om det ikke kjøres nok epoker vil roboten ikke klare å definere en beste rute, fordi den ikke har nok informasjon å gå på. Resultatet av testing av variasjon av antall epoker tyder på at 200 epoker skal være nok. Forsøk med å bytte policy til å utelukkende bruke greedy policy gir omtrent samme resultat som Epsilon-Greedy, utenom ved lavere antall epoker, hvor den noen ganger kun finner den nest beste ruten (score = 35).

5 Diskusjon

5.1 Reward-Matrise

Første utkast ble utarbeidet med ingen kostnad for å traversere normalt terreng, og ga belønningene 0 for hvite, -1 for røde, -5 for blå og +10 for mål. Dette ble senere endret til -1 for hvite, -5 for røde, -15 for blå og +50 for mål for å straffe unødvendige omveier i enkelt terreng. Dette tilsier at i et eventuelt større terreng ville vi være villige til å krysse en blå rute om det sparer oss 15 hvite ruter og en rød rute om vi sparer 5 hvite. Dette gir en ny beste score på 37. Dette gir fortsatt samme beste rute, men nå er det også straff for å gjøre unødvendige forflytninger mellom hvite ruter.

5.2 Monte Carlo Exploration

Av tabell 2 kan det sees at selv ved 100 000 iterasjoner av Monte Carlo Exploration finner roboten fortsatt ikke ruten som gir den høyeste scoren. Det trengs gjerne rundt 10000 iterasjoner for å oppnå et resultat jeg ville betegnet som særlig godt. Siden den beste ruten har 10 steg burde den teoretisk sett være funnet etter $4^{10} = 1\ 048\ 576$ iterasjoner, som tar ganske lang tid å gjennomføre. Siden det i Monte Carlo simulering er like stor sjanse for å bevege seg hver retning betyr dette at roboten stadig vekk også forsøker å gå utenfor rutenettet, og dermed oppnår dårlig score.

5.3 Q-Learning

Resultatene av Q-learning viser i utgangspunktet viktigheten av å innhente nok data til å kunne løse oppgaven på en god måte. For lite utforskning resulterer i at det ikke er mulig å finne en god rute til målet med en ren grådig algoritme. Om man bevarer en viss andel tilfeldige handlinger kan det være mulig å bryte ut av oscillering mellom to tilstander, men det har vel egentlig liten hensikt når det da åpenbart ikke finner en god vei uansett.

I forhold til forsøket med å kun benytte greedy policy gir det mening at dette skulle gå like fint som Epsilon-Greedy med nok epoker. Siden roboten resettes til en tilfeldig posisjon i terrenget ved starten av hver epoke er det mulig å starte nærme mål flere ganger, og på denne måten vil belønningen derfra forplante seg bakover mot startposisjonen. Dette har mye å gjøre med hvordan terrenget og Reward-matrisen er satt opp.

5.4 Feilkilder og Problematikk

Som nevnt i seksjon 4.2 kan roboten ende opp i en deadlock situasjon om ikke det utforskes lenge nok. Dette kan skje i get_best_path metoden om q matrisen ikke er tilstrekkelig utfylt ut for tilstandene. Om to nabotilstander for eksempel begge er hverandres beste valg å gå til vil roboten oscillere mellom disse to tilstandene.

Da denne koden kun skal brukes av meg har det ingen hensikt å bruke tid på legge inn beskyttelse mot deadlock tilstander og infinite-loops og lignende. Dette er i utgangspunktet kun relevant når roboten kjører get_best_path og det ikke har blitt gjennomført nok utforskning. Valgt løsning på problemet er å unngå det ved å sørge for å kjøre nok epoker. En annen løsning på problemet kunne vært å implementere en teller og maksverdi for antall steg i funksjonen.

5.5 Hva jeg ville gjort annerledes om jeg startet på nytt

Under arbeidet med oppgaven falt det meg inn at det kanskje er mer hensiktsmessig å starte i starttilstanden A4/ (0, 3) for hver Q-learning epoke når man bruker Epsilon-Greedy. Selv om den formelle algoritmen til Q-learning sier at det skal startes i en tilfeldig tilstand gir det mer mening å starte i startposisjonen hver gang, hvertfall i et så lite komplekst terreng.

6 Konklusjon

Å prøve å finne den beste ruten ved hjelp av Monte Carlo simulering er lite lønnsomt, men det er en meget effektiv metode å kartlegge et område på. Derimot konkluderes det med at implementasjonen av Q-learning er en fullstendig suksess. Roboten finner den beste ruten til målet hver gang, så lenge det kjøres nok epoker med Q-learning.

6.1 Videre arbeid

En mulighet for videreutvikling hadde vært å lage en funksjon som kunne lese av enten et fargekodet bilde av terrenget eller i det minste en liste over kostnad for å entre hver tilstand og automatisk opprette en Reward-matrise utifra dette. Det vil gjøre at roboten enkelt kan flyttes til et nytt terreng og starte prosessen med trening umiddelbart, gitt mindre modifikasjoner i koden som gjør at for eksempel maskimal bevegelse i x og y retning oppdateres dynamisk.

En annen tanke er rundt det å underveis i Q-learning prosessen teste etter den beste ruten og holde denne oppdatert, og slike holde rede på effektiviteten til Q-learning prosessen.

Referanser

[1] "Markov-prosess," *i Store norske leksikon*. (Apr. 24, 2024), [Online]. Available: https://snl.no/Markov-prosess (visited on 10/12/2024).

[2] S. Li, *Reinforcement Learning for Sequential Decision and Optimal Control*. Singapore: Springer Verlag, 2023.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. England: MIT Press, 1998.

[4] Ø. Grøn. "Markov-prosess," *i Store norske leksikon*. (Jul. 5, 2023), [Online]. Available: https://snl.no/Monte_Carlo-metode (visited on 10/12/2024).

[5] S. A. Solberg. "Q-learning demo." (Oct. 14, 2024), [Online]. Available: https://github.com/Pizzaman9988/karaktersatt-oppgave-1 (visited on 10/14/2024).