A^*

Klas Rogne Kanestrøm

November 2020

1 Introduksjon

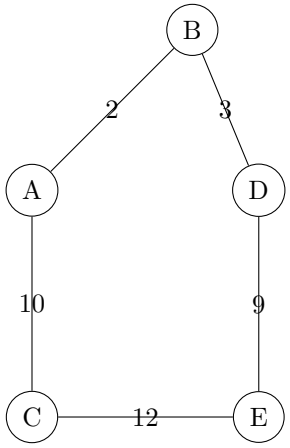
I den obligatoriske oppgaven er det lagt opp til at vi skal programmere opp A stjerne søkealgoritmen til et eksisterende "spill". Spillet er satt opp som en labyrint hvor målet er å finne den korteste veien fra en start node til en slutt node. Den obligatoriske oppgaven hadde allerede en metode for Dijkstra's algoritme, som gjorde implementasjonen av A^* betraktelig enklere. A^* er essensielt Dijkstra med et ekstra lag kompleksitet i form av en heuristisk[1] funksjon. Når jeg implementerte A^* tok jeg i stor grad bruk den allerede implementerte Dijkstra's algoritmen, og det jeg fant mest krevende var egentlig å implementere det sammen med ulike pygame[2], states fordi dette er en modul jeg ikke har sett så veldig mye på. I den første obligatoriske oppgaven hvor vi skulle implementere et kalman filter trenger vi ikke å ta i bruk pygame komponenter.

2 Dijkstra

Når jeg startet implementeringen av A^* valge jeg å sette opp Dijkstra's algoritmen og bygge ut fra den. Hovedideen i Dijkstra er at når man starter på en node vil den neste noden man besøker være den med den laveste kostnaden med traversering fra kilden. Så tenk deg at vi har en kø med noder og for hver iterasjon, tar vi et element fra køen med den laveste kostnaden så langt. Men den enkleste måten å forstå en algoritme på er å gå gjennom et eksempel. Under har jeg satt opp en graf, hvor jeg skal ta i bruk Dijkstra for å finne den korteste veien fra A til E. Her vil jeg forklare skritt hvordan algoritmen fungerer.

[1] heuristikk er en tommelfinger regel eller en god guid å følge når man løser et problem. Målet med heuristikk er å utvikle en enkel prosess som genererer et nøyaktig resultat på et akseptert mengde med tid.

[2] pygame er et set med python moduler designet for å lage video spill



Vi starter på A, og ser inn i køen og naboene til A ser vi:

```
queue: {
    B, kostnad : 2
    C, kostand : 10
    }
```

her velger vi å gå til den naboen med lavest kostnad som i dette tilfeller et B. Når vi nå befinner oss i B ser vi på nabo nodene A og D, og vi ser at node D ikke har noen kostnad satt hos seg så langt, så vi beregner den. Kostanden for node D er kostnaden så langt for B pluss kostnaden på edgen mellom B og D, som er 3. Dermer har vi en total kostnad på 5. Når vi putter dette inn i køen vår ser det slik ut:

```
queue: {
    C, kostnad : 10
    D, kostand : 5
    }
```

Vi følger den samme prosessen som over, vi tar ut node D fra køen fordi den har den laveste kostnaden så langt, selv om C var lagt inn fra før. Når vi er i D ser vi at vi når E med en kostnad lik 14, vi legger E til i køen og vi velger skal nå velge den neste noden vi skal besøke:

```
queue: {
    E, kostnad : 14
    C, kostand : 10
    }
```

Her ser vi at C er den laveste kostanden, og derfor tar vi ut C av køen. C har to naboer, A som er start noden som vi har besøkt før og E. Dersom vi går fra C til E vil det ha en kostnad på 22. Som er større en den nåværende kostanden på 14 og dermed forander vi den ikke. Dermed få vi den kortest avstanden på A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E. Nå har vi sett hvordan Dijkstra fungerer gjennom et eksempel, og har jeg satt opp implementasjonen som en pseudo kode.

```
#c(i,j) er linkkostnaden fra node i til node j
#D(v) er oyeblikkskostnaden paa stien mellom kilde og destinasjon v
#N er et set med noder

#Initialisering
N = {A}
for all nodes v
    if v adjacent to A
        then D(v) = c(A, v)
        else D(v) = None

Loop
find w not in N such that D(w) is a minimum
    add w to N
    update D(v) for all v adjacent to w and not in N:
        D(v) = min(D(v), D(w) + c(w, n))
    #Ny kostnad til v er enten gammel kostnad til v
    #eller kjent least-cost-path til w pluss kostnad fra w til v
until all nodes in N
```

Listing 1: pseudo kode Dijkstra

3 A^*

Vi nevnte over at A^* implementerte en heuristisk funksjon. Det denne funksjonen prøver å gjøre er å finne destinasjonen våres forte ved å unngå å gå ned unødvendige noder. For å gjøre dette trenger man noen estimat. Dersom alle noder vet omtrent hvor langt ned i "shortest path" denne destinasjonen er, kan vi bruke dette som ekstra info i estimatet vårt, og på denne måten finne veien fortere.

Ved hjelp av denne heuristikk kan vi lage oss en funksjon, som gir oss et estimat av den totale kostnaden på destinasjonen fra start noden til destinasjonen. Denne funksjonen vi bygger ved hjelp av heuristikken ser slik ut: $f(n) = g(n) + h(n)$
Her er:
- $f(n)$ = totalt estimerte kostnaden gjennom node n
- $g(n)$ = den totale kostnaden fra start til n
- $h(n)$ = estimerte kostnaden fra n til målet. Heuristikken.

$g(n)$ har vi allered implementert gjennom dijkstra algortimen, så dersom vi nå er i stand til å finne $h(n)$ kan vi finne frem til $f(n)$. Måten man implementerer det på er at man essensielt bruker dijkstra men bruke $h(n)$ funksjonen når man henter den neste noden fra frontier'en, da får man A^*. Dermed blir A^* en mer informert dijakstra. Det å ha en god heuristisk funksjon kan hjelpe oss med å unngå å gå innom unødvendige noder. Den heuristikk funksjonen vi tar i bruk i selve implementasjonen vår er en manhatten heuristikk funksjon som gjør at vi kan bevege oss i fire ulike retninger på rutenettet vårt. Standford.edu har lagt ut en veldig grundig beskrivelse av hvordan dette heuristikk funksjonen fungerer[3].

[3]https://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html

```
while openSet is not empty
    current = get node from openSet while removing the (currnet)
    node from openSet
    if current == targetnode
        break
    add current to closedSet
    for every edge in current.adjecent
        if edge.node not marked for update before (not in openSet)
            if edge.node not already updated before (not in the
    closedSet)
                edge.node.previous = current
                edge.node.g = current.g + edge.weight
                edge.node.h = heuristics(edge.node, targetnode)
                edge.node.f = edge.node.g + edge.node.f
                add the edge.node to openSet
            else
                if edge.node > current.g + edge.weight
                    edge.node.previous = current
                    edge.node.g = current.g + edge.weight
                    edge.node.f = edge.node.g + edge.node.f
                if edge.node have been updated previously (edge.
    node exists in closedSet)
                    remove edge.node from closedSet
                    and insert.node for update (insert edge.node
    into openSet)
    return
```

Listing 2: pseudo kode A^*