Karaktersatt Oblig-2

Jon-Fredrik Hopland

Denne oppgaven går ut på å implementere en A* algoritme. I denne teksten skal jeg forklare hvordan algoritmen jeg har implementert fungerer.

VI starter først med å sjekke om vi har en start-node og en slutt-node. Hele poenget med A* er å finne den korteste veien mellom to punkt, så uten start-node og/eller slutt-node så gir det ingen mening å kjøre resten av programmet.

```
def AStarSearch(self, startVertexName=None, targetVertexName=None):
    self.initPygame()
    # Check to see that start vertex is in Graph
    if startVertexName not in self.vertecies:
        raise KeyError("Start node not present in graph")
    # check to see if target vertex is in Graph
    if targetVertexName not in self.vertecies:
        raise KeyError("Cannot find target node in graph")
```

Videre setter vi noen initialverdier.

```
startNode = self.vertecies[startVertexName]
startNode.g = 0
startNode.h = self.heuristics(startVertexName, targetVertexName)
startNode.f = startNode.g + startNode.h
toNode = self.vertecies[targetVertexName]
```

- StartNode er start-noden
- startNode.g er kostnaden fra start-noden og frem til en node n. startNode.g har derfor initialverdi = 0
- startNode.h er den heuristiske verdien i en node n. Den heuristiske verdien er den estimerte distansen/kostnaden mellom noden man er i og slutt-noden (mål-noden). Den

forteller oss med andre ord hvor nærme vi er målet. Heuristikkmetoden bruker Manhatten-distansen, altså den kvadratiske avstanden

- startNode.f er totalkostnaden i en node n
- toNode er slutt-noden

```
import heapdict
openSet = heapdict.heapdict()
closedSet = []

def enqueue(data):
    openSet[data] = data.f

def dequeue():
    return openSet.popitem()[0]
```

Her oppretter vi en prioritetskø «openSet» og en tom liste «closedSet». OpenSet brukes til å lagre ubehandlede noder, der vi er interesserte i å hente ut den noden med lavest f-verdi. Funksjonen enqueue legger til en node i openSet, men funskjonen dequeue henter ut den første noden i openSet. ClosedSet er en liste der vi lagrer noder som er ferdigbehandlet.

```
enqueue(startNode)

while openSet:
    current = dequeue()
    self.pygameState(current, self.GREEN)
    self.pygameState(startNode, self.BLUE)
    self.pygameState(toNode, self.RED)

    if current == toNode:
        break

    current.known = True
```

Her ser vi at vi starter med å legge start-noden til i openSet.
Deretter kjører vi en while-løkke så lenge vi har ubehandlede noder.
Deretter tar vi ut første noden i prioritetskøen og kaller denne noden «current».
Viss noden vi behandler er den samme som slutt-noden, så har vi nådd målet. Derfor kjører vi en «break». Så setter vi current.known = True.

```
for adjecentedge in current.adjecent:
    if not adjecentedge.vertex.known:
        if adjecentedge.vertex not in closedSet:
            adjecentedge.vertex.previous = current
            adjecentedge.vertex.g = current.g + adjecentedge.weight
            adjecentedge.vertex.h = self.heuristics(
                adjecentedge.vertex.name, targetVertexName)
            adjecentedge.vertex.f = adjecentedge.vertex.g + adjecentedge.vertex.h
            enqueue(adjecentedge.vertex)
            self.pygameState(adjecentedge.vertex, self.PINK)
        else:
            if adjecentedge.vertex.g > current.g + adjecentedge.weight:
                adjecentedge.vertex.previous = current
                adjecentedge.g = current.g + adjecentedge.weight
                adjecentedge.vertex.f = adjecentedge.vertex.g + adjecentedge.vertex.h
            if adjecentedge.vertex in closedSet:
                closedSet.remove(adjecentedge.vertex)
                enqueue(adjecentedge.vertex)

closedSet.append(current)
```

Ovenfor ser vi selve algoritmen. Korti fortalt så for hver gang A* behandler en ny node så kalkulerer den kostnaden f til nabo-nodene og går til den nabo-noden med laveste f. Videre går vi gjennom algoritmen steg for steg.

For hver kant i nabo-nodene til current node så sjekker vi om noden til kanten ikke er markert for oppdatering, altså at adjecentedge.vertex.known er False. Viss den heller ikke er i closedSet, altså ikke behandlet, så kjører vi koden inne i den if-setningen.

Vi setter så previous til å være current.
Så regner vi ut kostnaden fra start-noden til noden. Det er kostnaden til current (current sin korteste veide vei) + vekten til kanten.
Så regner vi ut den heuristiske verdien til noden ved å regne ut heuristikken fra noden til slutt-noden.
Deretter regner vi ut totalkostnaden f ved å legge sammen g og h, og legger noden til i openSet.

Dersom noden ikke er i openSet, men den er i closedSet så sjekker vi om den korteste veide veien g i noden er større enn current.g + vekta til kanten.

Viss det er tilfellet så oppdaterer vi adjecentedge.vertex.previous, adjecentedge.vertex.g og adjecentedge.vertex.f på samme måte som overnfor.

I den siste if-setning så sjekker vi om noden har allerede blitt oppdatert, altså om den er i closedSet. Viss den allerede er i closetSet, så fjerner vi den fra closedSet og legger den til i openSet slik at den oppdateres på nytt. Grunnen til at vi gjør det er at noden har fått endret vekt og prioritet og derfor skal den re-traverseres.

Helt til slutt legger vi current til i closedSet siden den er nå behandlet.

Vi ser at med denne implementasjonen av A* så besøker vi ferre noder enn med dijkstras algoritme. Disse to algoritmene er veldig like, men A* er mer effektiv og raskere på grunn av heuristikk-metoden, som gir prioritet til noder som skal være nærme målet enn andre.