18.11.2022

Isak Steinmo Hansen

Uit norges arktiske universitet

Dokumentasjon A\*

Obligatorisk oppgave 2 Intro AI og ML

**Oppgavebeskrivelse:**

Denne oppgaven var å implementere en A\* funksjon i ett eksisterende PyGame program hvor dijkstras algoritme allerede er implementert for å finne korteste vei til et punkt fra et annet punkt.

**Implementasjon av A\***

For å implementere A\* ble det benyttet de allerede innebygde metodene som er i programmet. Dette inkluderte blant annet klasser som Edge og Vertex som da inneholdt attributtene f, g og h. Det som kjennetegner A\* er at når den skal finne distansen til target\_node, så bruker den både en kalkulasjon av distansen som er G, men også en estimering av distansen (H). Resultatet av dette blir da den totale kostnaden F. Dette er i motsetning av for eksempel Dijkstra som ikke bruker heuristikk, eller greedy-first som bare bruker Heuristikk.

For å dokumentere koden går jeg igjennom den steg for steg:

**Initialisering av koden:**

Et bilde som inneholder tekst

Automatisk generert beskrivelseDet første som gjøres er å sette verdiene g, h og f. Dette gjøres i henhold til beskrivelsen gitt i lysbildene om A-Stjerne. Her angir g startvekten som vi setter til 0, h angir heuristikken fra startnoden til målnoden og f angir den nåværende totalkostnaden g + h.

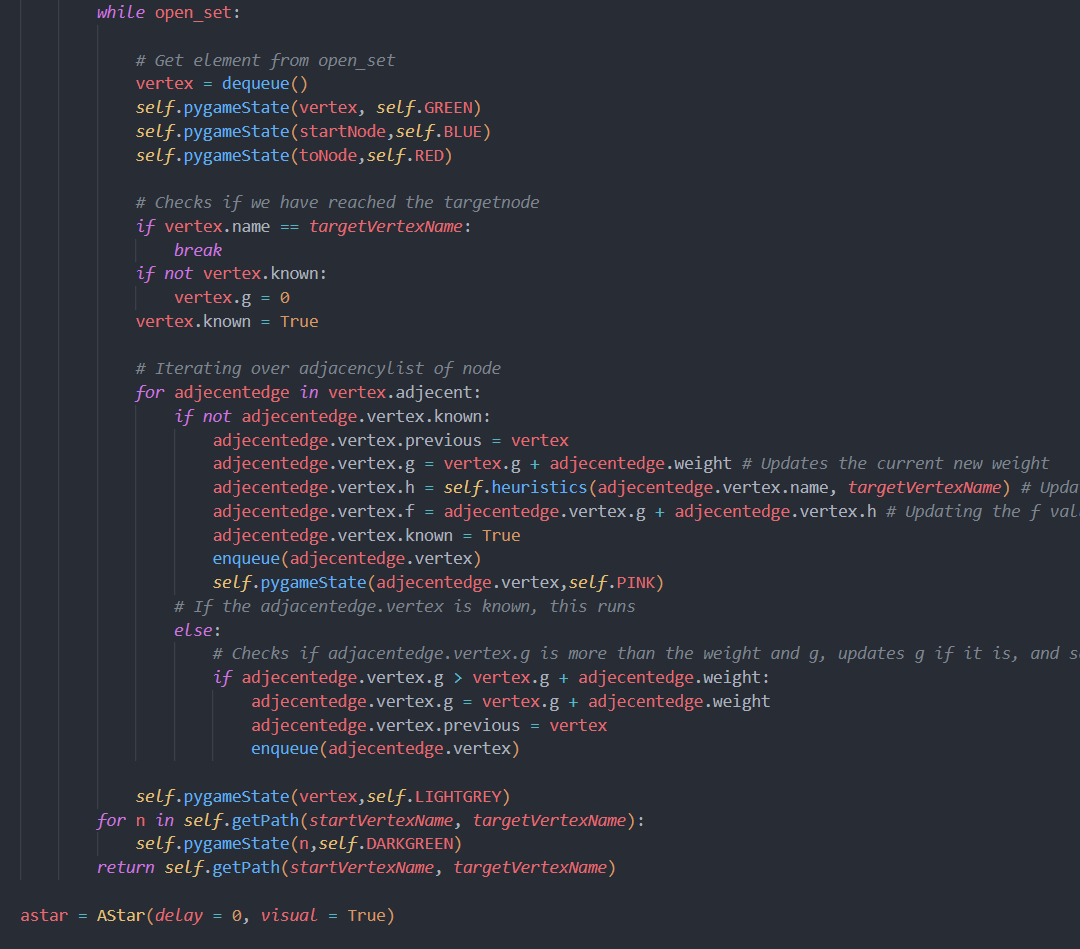
Figur Initialisering

Videre defineres open\_set ved å bruke HeapDict klassen som er en prioritetskø som inneholder de ubehandlede nodene. Legger også inn startnoden i open\_set for videre behandling. Som man ser i figur 1 er også funksjonene enqueue og dequeue definert, enqueue her er definert med prioritet på f verdien. I henhold til beskrivelsen av A-Stjerne.

**Loopen som itererer over noder og behandler dem til at målnode er funnet:**

Videre kjøres også en while løkke som vil kjøre så lenge open\_set ikke er tom. Der hentes først noden ut fra open\_set, sjekker om den er lik targetnode som i såfall vil si at programmet er ferdig, men ellers kjører videre. Etter denne sjekken kaller programmet på known verdien til en gitt node.

Neste som gjøres er å iterere over alle adjacent-vertecies til vertex, sjekker om denne ikke er kjent, hvis ikke kjent så oppdateres g ved å bruke initialverdien vertex.g + adjacentedge.weight. H oppdateres ved å bruke heuristikk funksjonen med nåværende node og selvfølgelig target-noden. F oppdateres også ved de nye g og h verdiene. Known verdien settes så til true og noden legges inn i open\_set.

Hvis den adjacentedge vi ser på i øyeblikket er kjent fra før sjekkes først om g verdien til noden vi nå ser på er større enn g til vertex + vekten til den gitte adjacentedge. Da settes en ny g verdi til adjacentedge.vertex som er summen av vertex.g og adjacentedge.weight. adjacentedge.vertex legges så inn på open\_set for oppdatering.

Figur Kode

Dette gjøres da helt til open\_set ikke innholder noen flere noder. Så kalles den allerede implementerte metoden getPath for å lage den raskeste stien fra startnode til målnode

**Resultater**

Ved kjøring av programmet får man et kart over besøkte noder som minner veldig om hvordan A\* vil oppføre seg. Dette er ved at den er ganske «firkantet» i motsetning til Greedy-first som ville ha sett ut som flere triangler, eller Dijkstras som ville søkt gjennom alt

Bildet av resultat:

Et bilde som inneholder tekst, klokke

Automatisk generert beskrivelse