INF-2101: Algoritmer

Obligatorisk oppgave 2

- You are likely to be eaten by a Grue!

Gruppe 3

Adnan Begovic
Magnus W. Bjørklund
Ole-Morten Tangen

Tromsø, 5. oktober 2012

Abstrakt
Å implementere en AI-spiller som kan bevege seg gjennom en labyrint for å finne et mål, samtidig som den bygger opp en grafrepresentasjon av labyrinten etterhvert som spilleren ser mer og mer av labyrinten.

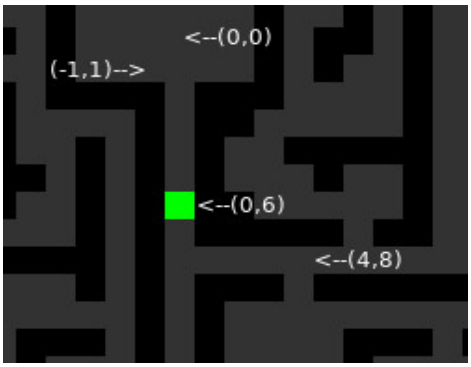
1. Introduksjon:

Oppgaven går ut på å finne Toby i en labyrint innen en gitt tid, mens man samtidig passer på feller og på å ikke bli spist av en Grue. Mens man beveger seg gjennom labyrinten skal man også bygge opp en grafrepresentasjon av labyrinten.

Prekoden vi fikk utlevert inneholdt server, kartverktøy og klient som har metodene som kommuniserer med serveren. Det er metodene i klienten vi benytter oss av for å interagere med labyrinten.

2. Design:

Vi definerer startposisjonen som (0,0) i grafoppbygningen siden vi ikke vet hvor vi starter i forhold til labyrintens ytterkant. Deretter bygger vi hver posisjon i horisontale retningen som første element i tuplen og hver posisjon i den vertikale retningen som andre element. Dersom grafen ble visualisert, ville den tatt form som et rutenett, der hver node er en rute i labyrinten.

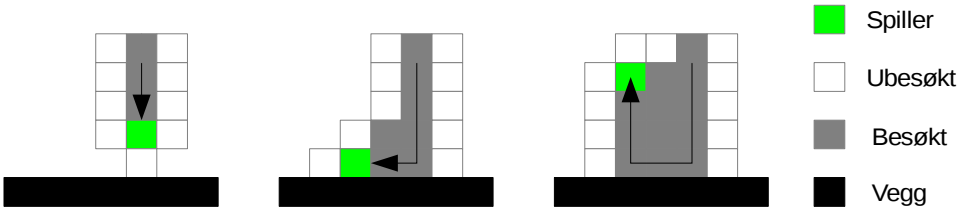


Vi beveger oss rundt i labyrinten, her undersøkes det for hver retning om det er mulig å gå, og vi beveger oss i den første ledige retningen, derfor er det vektet slik at den går sør, nord, vest, øst og lagrer alle intersections i en stack, dersom det ikke fins en vei videre så går vi raskeste vei tilbake til forrige intersection vi kom fra ved hjelp av Dijkstra's algoritme. Dersom det ikke fins noen tidligere intersection så benytter vi Trémaux algoritme for å undersøke området bakover og se om det finns en annen vei å gå.

3. Implementasjon:

Bevegelse:

Implementasjonen begynner alltid med å gå sørover i labyrinten. For hver ubesøkt rute vil algoritmen se seg rundt og legge de nye rutene inn i grafen dersom de ikke er vegger, registrere nåværende rute som besøkt og rutene rundt som ubesøkt. Algoritmen prioriterer ruter som ikke er besøkt. For hver nye rute som blir besøkt så undersøkes det om rutene som er tilkoblet denne også er tilkoblet en tidligere besøkt rute, dersom den er det så vil også den ruten som er mellom de besøkte bli satt som besøkt. Dette medfører at man slipper å se igjennom hver eneste rute, man får i stedet muligheten til å bare bevege seg over annenhver rute i ett rom, siden man i grunn alt vet hva som befinner seg på rutene imellom.



Illustrasjon av spillerens bevegelser og besøkte ruter.

Tilbakesporing:

Dersom spilleren kommer i en posisjon der det ikke er noen ubesøkte ruter rundt den, og det finnes et kryss som ikke har vært undersøkt, vil den gå raskeste vei gjennom tidligere besøkte ruter tilbake til krysset. Her blir Dijkstras algoritme benyttet for å finne den raskeste veien mellom nåværende rute og tidligere kryss med ubesøkte veier. Algoritmen går gjennom alle nodene og regner seg frem til vekter fra node til node og vil for hvert steg velge den veien med lavest vekt. På denne måten får man den raskeste veien fra posisjonen man er på til posisjonen man skal gå tilbake til.

Dersom det ikke er noen tidligere kryss bruker vi Tremaux's algoritme til å undersøke rutene bakover for å se om vi finner en annen vei. Denne algoritmen vil her markere ruten vi står på som besøkt to ganger, og deretter bevege seg til en av de tidligere besøkte rutene, på denne måten vet man at dersom man kommer over en rute som er besøkt to ganger så vil også rutene bak være besøkt to ganger.

Håndtering av andre enheter:

Koden som legger til ubesøkte ruter håndterer også interaksjon med enheter i labyrinten. Dersom denne finner Toby i en av rutene rundt seg, vil spilleren velge å gå dit neste steg. Feller vil på samme måte bli desarmert så snart man ser disse i en nærliggende rute. Gruer håndteres derimot i bevegelseskoden. For hvert steg vil spilleren sjekke om det er en grue i samme rute, og tenne en fyrstikk hvis dette er tilfelle og spilleren har fyrstikker igjen.

4. Diskusjon:

Det vi valgte til å begynne med var Tremaux algoritme for å finne frem til Toby i labyrinten, dette er fordi det er en algoritme som, gitt nok tid alltid vil finne frem, i tillegg så er den lett å implementere hurtig. Etterhvert valgte vi å bygge på den med Dijkstra's algoritme for å forsøke å gjøre den raskere når den beveger seg i rom, og for å kunne bevege seg raskeste vei tilbake dersom den møter en blindvei.

Etterpå begynte vi å se på de andre vanskelighetsgradene, her valgte vi blant annet å desarmere feller så snart vi så dem, i stedet for å lagre posisjonen på alle fellene å kun desarmere de når vi måtte gå over den ruten, dette var fordi spilleren går over de fleste rutene uansett og at vi ikke følte vi hadde spart så mye tid på å implementere dette.

Gruer ble i en tidlig utgave av implementasjonen sjekket på samme måte som feller. Dette medførte at gruer av og til begynte å spise på spilleren uten at den ble oppdaget siden den beveget seg før spilleren fikk sett seg rundt. I andre tilfeller kunne en grue gjøre at spilleren brukte flere fyrstikker på rad uten at det hadde noen effekt. Den nåværende implementasjonen undersøker heller ruten spilleren står på og har så langt ikke vist seg å feile.

5. Konklusjon:

Programmet kjører etter forutsetningene gitt i oppgaven, og løser alle kartene som fulgte med prekoden. Denne oppgaven har gitt oss innblikk i tilgjengelige muligheter for stifinning og hvordan et par av disse kan implementeres i praksis.

6. Referanser:

Maze Solving Algorithm – Wikipedia: http://en.wikipedia.org/wiki/Maze_solving_algorithm

Dijkstra's Algorithm - Wikipedia: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

UC Berkeley - Graph algoritms: http://www.cs.berkeley.edu/~kamil/teaching/sp03/041403.pdf

Dijkstra's algorithm for shortest path: http://code.activestate.com/recipes/119466/