Artificiell intelligens i datorspel

Robert Sirén
Matrikelnummer 36117
Åbo Akademi
Fakulteten för naturvetenskaper och teknik
Handledare Anna-Marie Soini
17.04.2014

Sammanfattning

TODO

Innehållsförteckning

TODO

1. Inledning

TODO

2. Vad innebär AI i datorspel?

Enligt Neil Kirby[1] är artificiell intelligens i datorspel till en viss mån annorlunda från AI i allmänhet. Datorspelen kräver nämligen att en bra AI är både trovärdig men samtidigt också effektiv i exekveringen. Ett rollspel skulle t.ex. inte vilja ha datorstyrda figurer som hopplöst fortsätter kollidera med ett staket då spelaren står bakom detta staket och skjuter med en pilbåge, givet att figurerna i sammanhanget är smartare än vad deras handlingar berättar. Ett trovärdigare beteende skulle vara att figurerna går runt staketet eller över den om spelet tillåter det.

Alternativt så kan det vara i trovärdighetens ändamål att figurerna inte är alltför smarta. Man kan argumentera att eftersom människor gör misstag så kanske bör AI också göra misstag då och då. Enligt Turing-testet så är en AI trovärdig om dess beteende inte kan skildras från en människas och därifrån så är en trovärdig AI inte perfekt i sitt beteende. Detta ska dock inte göra AI:n meningslöst. Prioritet ett är ändå att spelet ska vara roligt och AI har möjligheten att dra nytta av spelets interaktivitet för att förbättra spelarens nöje.

Samtidigt ska dessa figurer inte slösa datorresurser när de t.ex. söker en trovärdig stig till spelaren. Datorresurserna bör användas till nyttigt arbete. Med nyttigt arbete avses här att arbetet ska leda till en förändring i spelets tillstånd så att detta märks av spelaren. Om en datorfigur inte renderas när den söker efter en stig kanske det inte är så viktigt att man kontrollerar att hinder uppstår på stigen som t.ex. att figuren blir attackerad av en annan figur eller att ett stenras tvingar figuren att ta en mindre omväg.

Ett annat exempel skulle vara om två figurer simulerar ett brädspel som spelaren kan se. Frågan är då hur viktigt denna simuleringen är. Om den bara är en del av scenbilden är det inte så nödvändigt att brädspelet spelas korrekt. Simuleringen kunde enkelt bytas ut mot en animering. Datorprogram i allmänhet kräver ändå att prestandan är godtagbar när de är färdiga.

Men en viktig del av AI är att AI:n kan reagera på förändringar i sin miljö. Detta kan vara allt från en enkel insignal från spelaren till beaktande av fientliga figurer i en datorstyrd figurs omgivning då figuren prioriterar dess nya attackmål.

1 Kirby, Neil, *Introduction to Game AI*, Course Technology/Cengage Learning, 2010

3. Vanliga problemställningar för AI i datorspel

I likhet med program i allmänhet ska programkoden lösa ett problem. Till detta ändamål behövs algoritmer. Vilken som passar bäst in beror på omständigheterna men det finns olika strategier. Innan man kan börja måste man bestämma vad som ska göras. Detta kan åskådliggöras med en finit tillståndsmaskin.

3.1. Finit tillståndsmaskin (FSM)

En FSM är formellt en 5-tuppel bestående av en ändlig mängd av tillstånd, ett ändligt alfabet, en övergångsfunktion samt ett starttillstånd och en ändlig mängd sluttillstånd. Alfabetet i detta fall representerar indata. Tillstånden representerar en AI:s beteende. En mycket enkel FSM är ett värmeelement i tabell 1.

	1. Start/Slut: Slå av effekten	2. Slå på effekten
Temperatur för låg	2	2
Temperatur för hög	1	1

Tabell 1: FSM för ett värmeelement. Nästa tillstånd bestäms av kombinationen av nuvarande tillstånd och indata. Tabellen kan också åskådliggöras grafiskt.

Om denna FSM klassificeras som AI kan argumenteras men denna metod kan appliceras i andra sammanhang såsom implementeringen av en rutin för hur en AI ska bete sig när den möter en fientlig figur.

Enligt Kirby[2] är fördelen med denna metod att den ger en bra överbild av beteendet. Varje tillstånd har sina egna algoritmer som ger upphov till det sökta beteendet. Om dessa algoritmer är komplexa kan ett strategimönster användas för att enkelt hantera dessa tillstånd.

Nackdelen är att om alla tillstånd, eller åtminstone de tänkbara tillstånden, inte hanteras av tillståndsmaskinen så finns det en möjlighet att spelaren kan leda AI:n till sådana omständigheter där AI:n beter sig icke-trovärdigt. Dock om man beaktar alla

2 Kirby, Neil

tillstånd kan detta leda till en kombinatorisk explosion av tillstånd och indata. Bördan ligger då hos programmeraren att dela upp AI:s beteende till så få men så omfattande tillstånd som möjligt.

Implementering av en FSM kan göras på olika sätt men ett förslag är att använda ett tillståndsmönster för större tillståndsmaskiner. Exempelvis, en datorstyrd figurs AI bestäms av ett antal tillstånd.

```
public class NPC extends GameObject{
     public NPC(){};
     private State currentState;
     public void setState(State newState, Event event){
          this.unregister(currentState);
          currentState = newState;
          this.register(currentState);
          currentState.execute(event);
     }
}
```

```
public abstract State implements Observer{
     private Map<Event, State> transitionMap;
     public abstract void execute(Event event);
     public void onEvent(Event event){
          State s = transitionMap.get(event);
          if (s != null) {
               event.Caller().setState(s, event);
          }
     }
}
```

Med ovanstående kod skickar figuren ett `Event`-objekt ner till tillståndet. Objektet innehåller all nödvändig information för att bestämma om en övergång ska utföras. Om så är fallet, utifrån övergångsfunktionen `transitionMap`, meddelar tillståndet figuren om ett nytt tillstånd samt orsaken. Figuren avregistrerar det gamla

tillståndet från figurens lyssnare och registrerar det nya samt exekverar detta tillstånd med informationen i `Event`-objektet. Övergångsfunktionen konstrueras när tillståndet skapas i t.ex. ett `Factory`-objekt. Den kan t.ex. hämtas direkt från en textfil eller databas.

För små tillståndsmaskiner är mönstret kanske inte nödvändigt. Ett exempel skulle vara om endast ett tillstånd behövs såsom en dum men sökande missil. Dessa kan, i vissa sammanhang, använda en regulator. Anta att spelet har ett underliggande system som simulerar spelfysiken varje gång en ny ram ska renderas. En missil skjuts och ska följa efter ett mål *T*. Systemet simulerar fysiken genom att uppdatera föremålens positioner baserade på föremålens datavariabler. Följande pseudokod visar hur detta skulle kunna ske.

```
public void updatePhysics(float timeStep) {
     targetData = T.getPosition();
     physicsData = controller(self.getPhysicsData(),
                    targetData, timeStep);
     this.simulate();
}
```

Här skulle t.ex. missilen uppdatera hur mycket den ska vinkla sin raketavgas och på så sätt styra missilen. Regulatorn bestämmer hur mycket baserat på någon reglerlag. Dock så är detta så simpelt att koden inte beaktar om det finns något hinder framför missilen. För detta relegeras denna kod till en subrutin inom ett tillstånd.

3.2. Stigsökning

Ett mycket klassiskt problem som om och om igen stöts på är stigsökning: Hur ska man hitta en optimal stig så snabbt som möjligt? I ett euklidiskt plan är den kortaste stigen en rak sträcka. Men detta beaktar inte hinder och fart. För detta vänder sig programmerare till grafteorin. Grafer består av noder och kanter. Noderna representerar platser i någon sökrymd, kanterna representerar avstånd mellan noder. Dessa kanter brukar också viktas i t.ex. den tid det tar att traversera mellan två noder och kan vara

riktade.

Forskningen i detta område är stort. På grund av dess många tillämpningsområden har flera olika algoritmer utvecklats. De två mest kända algoritmerna är Dijkstras algoritm och A*-algoritmen. Den första är en girig algoritm medan den andra är en utveckling av Dijkstras algoritm med hjälp av heuristik.

3.2.1. Dijkstras algoritm

Given en mängd noder V och viktade kanter $E \subseteq V^2 \times \mathbb{R}$, konstueras en graf $G = (V, E)$. För detta ändamål beaktas endast icke-negativa vikter. Grafen antas vara sammanlänkad. Enligt Ariel Felner[3] utförs Dijkstras algoritm på följande sätt.

1. Låt $S \subseteq V$ och $U \subseteq V$. Låt $S \leftarrow \emptyset, U \leftarrow V$.
2. Låt startnoden $s \in V$, avstånd$[s] \leftarrow 0$. Alla andra noder i U initialiseras till att ha ett oändligt avstånd. Alla noders föregående noder initialiseras till 0.
3. Så länge som $U \neq \emptyset$:
    1. Välj $u \in U$ så att $\forall x \in U$: avstånd$[u] \leq$ avstånd$[x]$.
    2. Låt $S \leftarrow S \cup \{u\}, U \leftarrow U \setminus \{u\}$.
    3. För alla närliggande noder $v \in U$ till u med kant e:
        1. Om avstånd$[u]$ + vikt$[e]$ < avstånd$[v]$:
            1. Låt avstånd$[v] \leftarrow$ avstånd$[u]$ + vikt$[e]$.
            2. Låt föregående$[v] \leftarrow u$.
4. Kortaste stigen fås genom att traversera bakåt via föregående[] från målnoden.

Ovanstående kod har modifierats av mig för att tillåta en att traversera stigen. I praktiska tillämpningar bestäms denna implementations tidskomplexitet främst av implementering av U. Implementerat som en länkad lista skulle den ha tidskomplexiteten $O(|E| + |V|^2)$. Å andra sidan skulle en prioritetskö med logaritmisk insättning och uttagning snabba upp algoritmen till linjäritmisk tid.

3 Felner, Ariel, *Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm,* Proceedings, The Fourth International Symposium on Combinatorial Search (SoCS-2011)

Framöver hittar algoritmen alltid den kortaste stigen. Nackdelen är dock att antalet noder som måste besökas är stor. Så stor att för de flesta realtidsimplementeringar, som t.ex. datorspel, är denna algoritm för långsam. En effektivare algoritm är A*-algoritmen.

3.2.2. A*-algoritmen

För att snabba upp Dijkstras algoritm används nu istället en heuristisk algoritm, A*-algoritmen. Skillnaden mellan denna och Dijkstras är att algoritmen nu, tillsammans med närliggande noders avstånd, uppskattar den kortaste stigen till målnoden. Detta kräver att man vet något om grafens implementering. I t.ex. ett euklidiskt plan vet man koordinaterna till noderna. En rak linje från en nod till målnoden kunde då vara en uppskattning. Enligt Xiao Cui[4] utförs algoritmen på följande sätt.

1. Låt $S \subseteq V$ och $U \subseteq V$. Låt $S \leftarrow \emptyset$.
2. Låt startnoden $s \in V$, $f[s] \leftarrow 0$, avstånd$[s] \leftarrow 0$. Låt $U \leftarrow \{s\}$.
3. Så länge som $U \neq \emptyset$:
    1. Välj $u \in U$ så att $\forall x \in U: f[u] \leq f[x]$.
    2. Låt $U \leftarrow U \setminus \{u\}$, $S \leftarrow S \cup \{u\}$
    3. Om u är slutnoden, stoppa algoritmen.
    4. För alla närliggande noder v som nås med en kant e från u:
        1. Om $v \in S$, skippa v.
        2. Annars om $v \notin U$:
            1. Låt $U \leftarrow U \cup \{v\}$.
            2. Låt föregående$[v] \leftarrow u$.
            3. Låt avstånd$[v] \leftarrow$ avstånd$[u] +$ vikt$[e]$.
            4. Låt $f[v] \leftarrow$ avstånd$[v] + h[v]$.
        3. Annars om $v \in U$:

Fortsättning på nästa sida.

4 Cui, Xiao, *A*-based Pathfinding in Modern Computer Games*, IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, January 2011

1. Om $f[v] > \text{avstånd}[u] + \text{vikt}[e] + h[v]$:
    1. Låt $\text{avstånd}[v] \leftarrow \text{avstånd}[u] + \text{vikt}[e]$.
    2. Låt $f[v] \leftarrow \text{avstånd}[v] + h[v]$.
    3. Låt $\text{föregående}[v] \leftarrow u$.

Här är $h[x]$ den heuristiska funktionen som uppskattar avståndet från x till slutnoden. Återigen är det praktiskt att implementera U som en prioritetskö, t.ex. en trappa eller självbalanserande binärt sökträd. S bör också snabbt gå att söka igenom.

Enligt Cui är denna algoritm optimal om $h[x]$ aldrig överuppskattar det verkliga avståndet. En optimal algoritm anses vara en sådan algoritm som har en så låg tidskomplexitet som är teoretiskt möjligt för den valda heuristiska funktionen. Skulle den heuristiska funktionen överskatta avståndet, kan det hända att den stig man får inte är den kortaste. I de flesta fallen är den heuristiska funktionen också monoton d.v.s. $h[u] \leq \text{vikt}[e] + h[v]$, där e är en kant från u till v. Jämför med triangelolikheten. Enligt Masoud Nosrati[5] skulle en monoton heuristisk funktion betyda att ingen nod söks igenom mer än en gång. Dijkstras algoritm använder t.ex. $h[x] = 0$ för alla x.

Enligt Nosrati[6] är tidskomplexiteten beroende på valet av heuristisk funktion. Exponentiell tid är möjligt i värsta fall. Men om $|h[x] - h^*[x]| = O(\log h^*[x])$, där $h^*[x]$ är den exakta kostnaden från x till slutnoden, samt om sökrymden är ett träd garanteras en polynomiell tid. I de flesta fall inom datorspel är detta snabbare än Dijkstras algoritm.

3.2.3. Optimering av A*-algoritmen

Ett problem som A*-algoritmen har är att den, likt Dijkstras algoritm, tar upp mycket minne. Mängden U lägger till fler och fler noder i A*-algoritmen, Dijkstras algoritm laddar in hela grafen från början! För stora grafer blir detta då ett problem. I de flesta fall kan en effektiv datastruktur lindra besvären, men man kommer bara så långt

5 Nosrati, Masoud et.al., *Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches*, World Applied Programming, Vol (2), No (4), April 2012

6 Nosrati, Masoud

med den optimeringen. För att göra det bättre bör sökrymden minskas. Inom datorspelsindustrin används NavMesh-teknik (från engelskans *Navigation Mesh*, dvs navigationsnät).

Enligt Cui[7] delas sökrymden in med NavMesh till konvexa polygoner. För nästan kontinuerliga sökrymder, t.ex. ett euklidiskt 2D-plan i en datorvärld, är detta en väl användbar metod. Kort sagt navigerar man först mellan polygonerna tills man når samma polygon som målnoden finns i. Därefter räcker det oftast att man tar fågelvägen då skillnaderna mellan den och den kortaste stigen är obetydlig. Hur undviker man då hinder? Detta åstadkoms genom att polygonerna byggs runt hindren. En stor fördel med detta är att tekniken gör det enkelt att göra ändringar i navigationsnätet då t.ex. ett passage har blockerats av ett stenblock som nyligen fallit. Man behöver endast bygga om ett par polygoner runt t.ex. en kollisonslåda.

Ytterligare optimeringar skulle ske om algoritmen gjordes mer iterativ. IDA*-algoritmen är en sådan variant som använder sig av iterativ fördjupning. Denna algoritm är så gott som en variant av IDDFS, eller iterativ fördjupning, djupet-först sökning. Dock med en heuristik som bestämmer maximidjupet.

För övrigt är det värt att nämna att oavsett av allt detta förekommer det ofta ett problem som inte alltid är lättlöst: Spelaren föredrar ibland inte alltid den kortaste stigen. Ta ett strategispel som exempel. Spelarens skulle inte alltid vilja sända sina trupper igenom suboptimal terräng. Inte skulle spelaren heller vilja sända trupper igenom områden som inte kan försörja alla trupper med t.ex. mat. Där skulle trupperna påverkas av utnötning. Dessa effekter måste då beaktas under stigsökningen.

3.3. Markovkedjor

Ibland så räcker det med att AI beter sig slumpvist. Detta är idén bakom Markovkedjor. En Markovkedja är en (diskret) stokastisk process. I praktiken är den en FSM med stokastiska övergångar. Det som behövs är då en FSM där varje övergång har en sannolikhet och summan av alla utgående övergångar summeras till ett.

Markovkedjor används i stor del i textsyntes. I textsyntesen är problemet att skapa

7 Cui, Xiao

en text som en människa kan tro är skriven av en annan människa. En Markovkedja kan åstadkomma detta genom att slumpmässigt välja fraser enligt någon sannolikhetsfunktion. Den kanske mest enklaste implementationen är att skanna in ett antal texter. Varje ord, inklusive punktering, bildar då ett tillstånd. Övergångar fås genom att utföra en frekvensanalys av efterföljande ord. Sannolikheten för en övergång är då en klassisk sannolikhet av frekvensen av ett efterföljande ord dividerat med totala antalet observationer av alla andra efterföljande ord. Från ett initialtillstånd kan en stor mängd text genereras.

```
Map<String, Map<String,Integer>> analyzeText(File t) {
     TextStream stream = new TextStream(
          new FileReader(t,'ReadOnly'));
     stream.setDelimiter('whitespace');

     Map<String, Map<String,Integer>> map =
          new HashMap<String, Map<String,Integer>>();

     String prev = null;
     while (stream.hasNext()) {
          String current = stream.NextString();
          if (prev == null) {prev = s; continue;}
          if (map.get(prev) == null) {
               //Nytt tillstånd
               map.put(prev,
                    new HashMap<String,Integer>())
          }
          Map<String,Integer> transMap = map.get(prev);
          if (transMap.get(s) == null) {
               //Ny övergång
```

Fortsättning följer på nästa sida.

```
                transMap.put(s,1);
            }
            else{
                int score = transMap.get(s);
                transMap.put(s,score++);
            }
            prev = s;
        }
        return map;
}
```

Ovanstående pseudokod är ett exempel på hur en övergångsfunktion skulle konstrueras för en Markovkedja. Notera att övergångsfunktionen kan också skrivas till sin egna klass för att enkapsulera den underliggande koden och t.ex. spara totala antalet observationer som sin egen privata variabel. Exempel:

```
TransitionTable analyzeText(File t) {
     TextStream stream = new TextStream(
          new FileReader(t,'ReadOnly'));
     stream.setDelimiter('whitespace');

     TransitionTable table =  new TransitionTable();

     String prev = null;
     while (stream.hasNext()) {
          String current = stream.NextString();
          if (prev == null) {prev = s; continue;}
          table.addTransition(prev,s)
          prev = s;
     }    return table;
}
```

För att detta ska fungera väl måste man ha ett stort antal texter, möjligtvis inom det område man vill att texten ska omfatta. Genereringen följer då en Monte Carlo metod. Möjligtvis kan man också behöva beakta fler ord än ett åt gången. Ett tillstånd kan t.ex. bestå av två eller tre ord.

Markovkedjor har också andra tillämpningar. Anta att en datorstyrd figur har en "ledig tid" där den inte specifikt har något att göra. Istället för att stå på samma ställe eller vandra runt utan mening, skulle den kunna läsa in möjliga aktiviteter. Därefter, baserat på en Markovkedja, väljer den vilken aktivitet den föredrar att utföra.

3.4. Skedulering

TODO

3.5. Lookahead-metoder

TODO

4. Avslutning och kommentarer

TODO

5. Litteraturlista

Kirby, Neil, *Introduction to Game AI*, Course Technology/Cengage Learning, 2010

Felner, Ariel, *Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm,* Proceedings, The Fourth International Symposium on Combinatorial Search (SoCS-2011)

Cui, Xiao, *A*-based Pathfinding in Modern Computer Games*, IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, January 2011

Nosrati, Masoud et.al., *Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches*, World Applied Programming, Vol (2), No (4), April 2012