Crash & Turn AI

Arvid Magnusson, Viktor Sjögren

6 augusti 2022

Innehåll

1 Introduktion

Vägsökning inom spel är en stor tillämpning där Artificiell Intelligens (AI) kan utnyttjas för att ta fram olika intressanta beteenden. Flera varianter har tagits fram med samma mål att kunna navigera i en terräng, men hur agenten beter sig på vägen varierar baserat på den valda metoden. Detta problem är heller inte unikt för datorspel utan återfinns i en massa olika områden och branscher. Det kan till exempel användas inom ekonomi för att hitta tillvägagångssättet med lägst kostnad, eller inom logistikbranschen för att hitta de kortaste och mest tidseffektiva rutterna.

I denna rapport beskrivs ett projektarbete där en Crash & Turn algoritm har implementerats. Tanken med algoritmen var att utveckla en fiende-AI som med hjälp av vägsökning kan jaga en spelare som rör sig genom en värld med hinder. Beteendet ska replikera ett simpelt beteende som kan användas för enklare ändamål.

2 Bakgrund och Teori

2.1 Vägsökning i spel

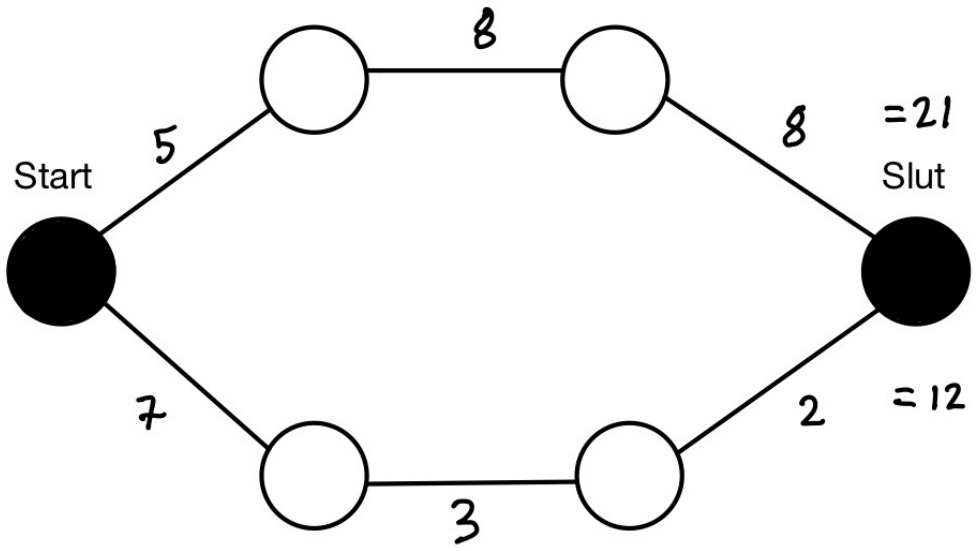
Vägsökningens uppgift i spel är att hitta en väg från en punkt till en annan, oavsett hur banan ser ut. Detta ska ofta ske utifrån några kriterier, till exempel att vägen ska vara den kortast möjliga, eller ta kortast tid till målet eller också vara den "billigaste" vägen att gå.

För att lösa detta problem så har en mängd olika metoder presenterats genom historien[1]. *Breadth First Search* (BFS) är en simpel algoritm som expanderar som en ring utåt ett steg i taget tills den har hittat en väg till målet. Den är dock inte så effektiv eftersom den kollar alla möjliga vägar. *Dijkstra's Algoritm* bygger vidare på BFS genom att låta oss prioritera vägar som är associerad med en låg kostnad. På det sättet behöver vi inte utforska samtliga vägar. Den mest populära metoden för vägsökning i spel är dock A*. Den presenterades redan 1968 av tre forskare på Stanford Research Institute[5]. Det är en modifikation på Dijkstra's algoritm och är optimerad för att hitta vägen till en enda destination. Det som gör A* så populär är att den garanterat hittar den kortaste vägen till målet, på ett relativt beräkningseffektivt sätt.

Med detta sagt så är det inte alltid man är intresserad av att en AI alltid tar den optimala vägen i spel. Det kan se onaturligt ut. Vi vill ofta att en AI ska efterlikna hur en människa skulle agera och en människa tar inte alltid den optimala vägen, speciellt inte på mer komplexa layouter. För dessa ändamål kan en algoritm som Crash & turn fungera utmärkt.

2.2 Lokal eller global metod

När det kommer till vägsökningsalgoritmer så är termerna lokal och global metod viktiga att förstå. En lokal metod tar bara hänsyn till det som ligger i dess direkta närhet. Detta illustreras lättast med en väldigt simpel nodgraf, se figur 2.1. Den lokala metoden ser bara ett steg framåt och kommer därför välja att gå uppåt eftersom 5 är billigare än 7. Detta kommer dock resultera i att den totala vägen till slutet är dyrare än om den hade valt att gå nedåt första steget. En global metod däremot kommer kolla på hela grafen för att hitta den globalt bästa vägen.



Figur 2.1: *Simpel nodbaserad graf för att illustrera lokal vs global metod. Nummrena anger kostnaden att gå den vägen. Lokal metod kommer gå den övre vägen medans global väljer den undre.*

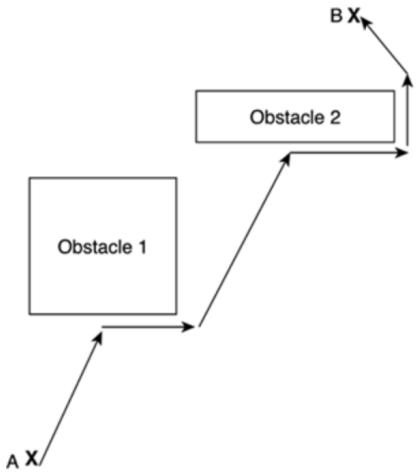
2.3 Crash & Turn

Crash & turn är en lokal metod som är starkt baserad på heuristik och som ganska mycket liknar hur ett djur skulle bete. Det är inte en exakt algoritm som till exempel A* är. Den följer istället mer utav ett koncept som kan implementeras på olika sätt. Beskrivningen här baserar sig på Yaldex beskrivning av metoden[4].

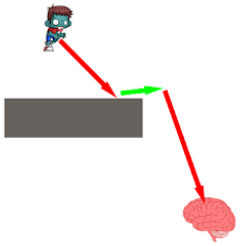
Vi utgår från figur 2.2 för att förklara konceptet. Agenten startar i punkt A och ska gå till punkt B. Den försöker först gå direkt mot punkten så länge den kan, så kallat fågelvägen. Efter ett tag så ligger ett hinder i vägen. Den kommer då gå antingen höger eller vänster tills hindret inte längre ligger mellan agenten och punkt B. Agenten fortsätter sedan att gå rakt mot punkten. Detta mönster upprepas tills agenten nått slutpunkten.

Eftersom Crash & turn inte är en exakt algoritm så kommer dess beteende avgöras av hur algoritmen implementeras. Till exempel kan riktningen den går längs vid kollision väljas på olika sätt:

- *Random.* Agenten väljer helt enkelt sida slumpmässigt. Detta fungerar oftast förvånansvärt bra eftersom agenten ska vara lite dum och inte ta den optimala vägen. Det uppstår



Figur 2.2: *Illustrerar konceptet för Crash & turn. Tagen från Yaldex[4].*



Figur 2.3: *Agenten väljer riktning som avviker minst från den initiala riktningen.*

dock situationer där den väljer den uppenbart mycket sämre riktningen vilket kan se riktigt dåligt ut.

- *Föredra ingångsriktning.* Agenten väljer den riktning som avviker minst från riktningen den har när den träffar hindret. Detta illustreras i figur 2.3.

Det finns dessutom olika metoder för hur den ska bete sig när den träffar ett objekt. Ett tillvägagångssätt är att en punkt beräknas som ligger strax bredvid objektet som agenten går till och därefter försöker gå mot slutpunkten igen. En annan metod är att, för varje bildruta, låta agenten ta ett ministeg i sidled och sedan testa att gå mot slutpunkten igen. Ligger hindret fortfarande i vägen så upprepas detta tills den kan.

3 Metod

3.1 Programvara och ramverk

Implementationen av algoritmen gjordes i Javaskript (JS) tillsammans med ramverket p5.js. P5.js är ett hjälpbibliotek som tillåter för bland annat enklare uppsättning av grafiska komponenter [2]. P5 kommer med en setup-funktion som körs en gång när programmet startas samt en draw-funktion som körs varje bildruta. Med användning av JS och p5.js kunde programmet köras direkt lokalt via webben. Koden skrevs i programutvecklingsmiljön Visual Studio Code för enklare hantering av felkällor samt för att tillåta realtids-samarbete via *Live Share* [3].

3.2 Programstruktur

Programmet delades upp i 5 JS filer, samt en .html och .css fil för att hålla koden ren.

- **main.js**. Här skrevs all allmän kod som behövdes för att programmet skulle köras.
- **crashAndTurn.js**. I denna fil skrevs koden som hade med algoritmen specifikt att göra, exempelvis vart agenten går efter kollision.
- **avatar.js**. I denna fil skrevs endast kod som hade med agenten att göra, exempelvis agentens gå-animation.
- **obstacle.js**. I denna fil skrevs endast kod relaterat till hindrena och hur de ritas ut.
- **map.js**. Här skrevs färdiga koordinater och storlekar på olika banor.

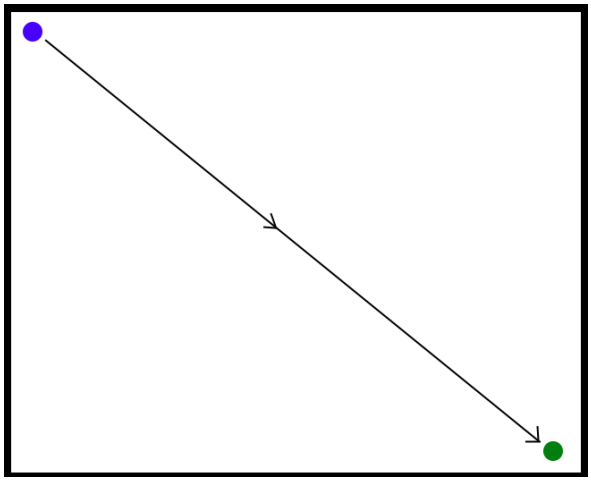
3.3 Vår implementering av Crash & Turn

Algoritmen implementerades utifrån beskrivningen i stycket 2.3.

3.3.1 Första fasen

Implementeringen började med att sätta upp en *canvas* där agenten skulle kunna röra sig i. För att fokus skulle ligga på algoritmens logik förenklades agentobjektet till en cirkel med

färg och en fast punkt ritades ut som motsvarade målet för agenten att gå till. Med dessa temporära förenklingar beräknades fågelvägen till målpunkten och sedan testades ifall agenten kunde gå ifrån vänsterkanten på *canvasen* till slutpunkten längst ner till höger, som illustreras i figur 3.1. För att få en animation på agenten så beräknades en normaliserad vektor som pekade mot slutpunkten som agenten skulle röra sig längs, vilket gjorde att för varje bildruta omritades agentobjektet med den normaliserade vektorns bidrag. På det sättet fås effekten av att agenten "går" mot målet.



Figur 3.1: *Första implementationen av programmet, fågelvägen och animation av agenten.*

Nästa steg som implementerades var att definiera hinder. För att undvika att behöva hantera komplexa former beslöts det att hindrena enbart skulle vara instanser av rektanglar där det enda som varierar är längd och bredd. Hinderobjekten lades in i en array där de sista 4 i arrayen alltid består utav väggarna på kartan. Efter hindrena var implementerade kunde de ritas ut i *canvasen* men agenten hade inte någon form av kollisionsdetektion, vilket gjorde att det naturliga nästa steget var att implementera kollision mellan agenten och ett hinder.

Att detektera kollision kunde enkelt göras med if-satser som jämför agentens position med alla rektanglar i hinder-arrayen. Det enda som visade sig vara problematisk med att just detektera kollision var hur överlappande hinder skulle hanteras, vilket gjorde att det lämnades oklart för tillfället. När väl en detektion hade identifierats så kollas det ifall sidan av objektet som träffats är orienterat horisontellt eller vertikalt. Baserat på vilket, sätts den nya riktningen på agenten till (1,0) eller (-1,0) respektive (0,1) eller (0,-1). Detta motsvarar att agenten går rakt åt vänster/höger eller nedåt/uppåt längs med hindret. Vilken riktning som väljs baseras på vilken av de två metoderna *Random* eller *Föredra ingångsriktning* som presenterades i stycke 2.3. I detta stadie lades två knappar till som i realtid ändrar viken metod som agenten följer för att lättare se skillnaderna på beteendet.

När väl riktningen har satts till någon av dessa, så testas ifall fågelvägen är fri till målpunkten för varje bildruta. Om fågelvägen fortsätter att vara blockerad kommer riktningen bevaras

som den är men så fort fågelvägen är öppen kommer riktningen sättas till vektorn som motsvarar fågelvägen. Under varje bildsekvens som agenten körs så kollas alltid först ifall agentens position är ovanpå slutpunkten, vilket ifall är sant avslutas programmet då målet är uppfyllt.

Denna version som implementerades visade sig ha problem med att logiken för vilken väg som väljs vid kollision inte var konsekvent. För att försöka förbättra detta implementerades if-satser som skulle i teorin omöjliggöra att agenten kan byta till en annan riktning när en riktning har valts efter kollision. Problemet löstes inte utav detta utan berodde på något annat. En omskrivning utav if-satserna gjordes sedan där jämförelser mellan flyttal gjordes på ett noggrannare sätt för att identifiera ifall problemet kunde komma utifrån *numerical errors*. Testning med noggrannare flyttal *checks* visade sig inte ha hjälpt utan problemet kvarstod.

3.3.2 Andra fasen

Eftersom implementationen av algoritmen som beskrivits i föregående avsnitt gav tydliga problem bestämdes det att en omskrivning av hur agenten beter sig vid kollision behövdes göras. På detta sätt kunde problemet som uppstod förut eventuellt undvikas. Tanken med omskrivningen var att istället för att sätta riktningen på agenten till något bestämt, skulle en punkt sättas som agenten går till.

För att denna metod skulle funka behövde det säkerställas att punkten som bestäms är varken utanför banan, inuti ett objekt eller orimligt satt så att agentens vägval upplevs som dåraktigt. Det algoritmen gör är att kolla vilket objekt som kollisionen uppstått med och därefter ta reda på vilken riktning agenten ska gå i. Därefter så beräknas en punkt agenten ska gå till utifrån vilket objekt den har kolliderat med. Detta illustreras i figur 3.2. Denna destinationspunkt ligger alltid en liten bit utanför objekt och även lite snett bakåt då detta visade sig ge bäst resultat.

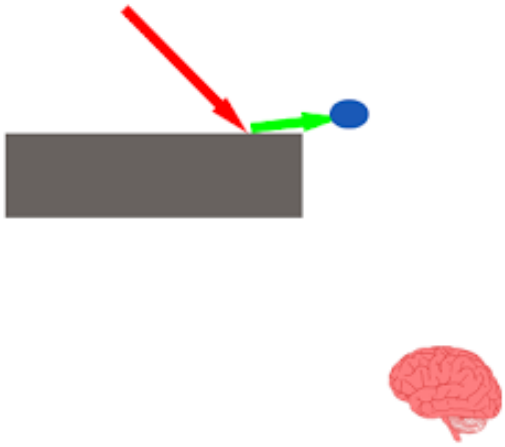
Denna metod visade sig inte ha samma problem med att agenten blev instabil och väljer att byta väg vid orimliga tillfällen. Däremot har denna metod konsekvensen att agenten går mot den beräknade punkten även ifall fågelvägen är öppen för den att gå.

3.3.3 Övrig implementation

Utöver huvudfokuset som var att implementera en Crash & Turn AI, lades också den allra sista tiden av projektet till grafiska komponenter och ljud. Detta innebar att en 2D-sprite animation lades till som en textur till agenten i form av en gående zombie. Animationen gjordes enkelt med en for-loop som itererar igenom en bild-array. Spelarens karaktär lades enkelt till som en statisk bild i form av en hjärna, där positionen på spelaren styrs med

datamusen. Hindrena fick en enkel stenmur-textur på sig och *canvasen* fylldes med en grå färg som bakgrund.

Till sist lades även en bakgrundslåt till i spelet samt start och stopp knappar som gjorde att spelet kunde pausas. När agenten fångat spelaren skapades också en "game over" text som talar om för spelaren att spelet är över.



Figur 3.2: *Illusterar hur en destinationspunkt beräknas när agenten kolliderar med ett objekt.*

4 Intressanta problem

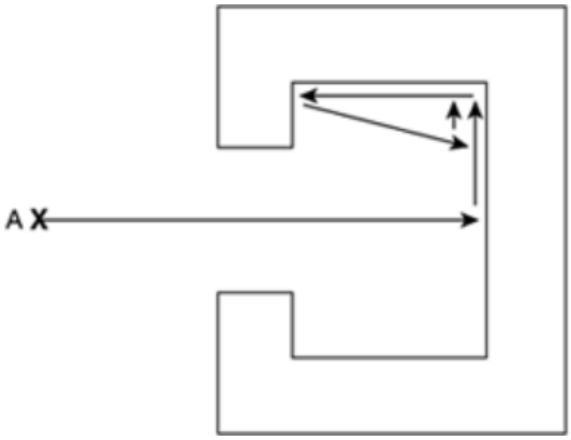
4.1 Kollisionshantering

Tanken i början av projektet var att kollisionshantering inte skulle vara något större problem. Vi tänkte att det bara skulle vara att kolla varje bildruta om ett ministeg mot slutpunkten skulle göra att vi rörde oss in i ett objekt. I teorin ett enkelt problem men som i praktiken inte fungerade klockrent. Det stora problemet var att det var svårt att felsöka exakt vart problemet låg då buggen med kollision var väldigt inkonsekvent. Den kunde långa perioder fungera perfekt och navigera runt på banan utan problem, men helt plötsligt kunde den byta håll när den höll på att gå runt ett objekt eller i vissa fall rent av gå igenom objektet. Vi provade att lägga till flera checks och villkor för att förhindra dessa saker men inget fick bort det helt. Detta ledde till att vi bytte metod till den som beskrevs under 3.3.

Som nämndes i 3.3.1 lämnades kollision med överlappande objekt oklart, vilket i slutändan aldrig löstes. Istället beslöts det att hur banorna utformas får vara begränsade till att alltid se till att objekt noggrant placeras ut där de aldrig går in i varandra.

4.2 Det konkava problemet

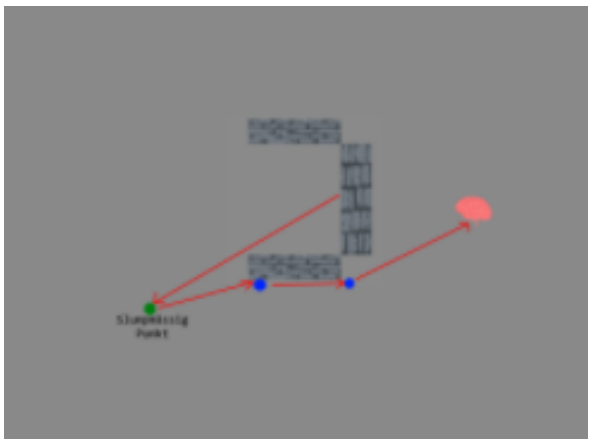
Konkava objekt är algoritmens akilleshäl. När agenten kommer in i ett konkavt objekt är det i princip omöjligt för den att ta sig ut om man inte modifierar algoritmen på något sätt. Beteendet för en icke-modifierad algoritm kan se ut enligt Figur 4.1. Den kommer alltså fastna i en loop inuti objektet.



Figur 4.1: *Problemet som uppstår då agenten går innanför en konkav struktur, bild tagen från Yaldex[4].*

För att lösa detta gav vi vår agent ett "minne". Vi sparar de fem senaste punkterna där agenten har kolliderat. Är det två eller färre punkter som är unika betyder det att agenten har kört fast och därför låter vi den gå till en "random" punkt. Denna punkt är inte 100% random utan beräknas genom att gå rakt bakåt (dvs motsatt spelarens position) och sedan läggs det på lite random förskjutning på den punkten. Denna lösning visade sig fungera relativt bra, men den löser tyvärr inte problemet alla gånger. Ju större konkavt objekt det är desto svårare blir det för agenten att ta sig ut. Detta ligger dock i algoritmens natur. Den kommer aldrig kunna lösa konkava objekt till till 100%. Vill man ha konkava objekt på sin bana är det bästa att skapa speciallfall för just dessa objekt så att agenten vet exakt hur den ska röra sig för att ta sig ut.

I Figur 4.2 ses det ett teoretisk fall då denna lösning skulle hjälpa agenten direkt ta sig ur det konkava problemet.



Figur 4.2: *Illustration av vår lösning till konkava former, när random punkt blir vald optimalt.*

5 Diskussion och Slutsats

Tanken i början av projektet var som tidigare nämnt att AI:n skulle, för varje bildruta, försöka gå fågelvägen mot slutpunkten även fast den precis hade kolliderat. Detta skulle göra agenten mer responsiv till hur spelaren rörde sig på banan. Speciellt vid kollision med stora objekt. Detta fick tyvärr slopas av anledningarna som beskrevs under 3.3 och 4.1. Dock så fungerar den metod vi bytte till väldigt väl och det är ytterst få gånger som fall uppstår då man önskar att agenten var mer responsiv.

Eftersom själva fysiken vid kollision inte var huvudfokuset för detta projekt så skulle detta problem ha kunnats lösas genom att använda ett spelramverk, till exempel Phaser, som har fysikhantering inbyggt.

Så som nämndes i teorikapitlet 2.3 så kan man använda olika heuristik för att bestämma vilken riktning agenten ska gå vid kollision. Tanken i början av projektet var att användaren skulle kunna välja vilken heuristik att använda med knappar. Dock efter att vi hade testat både *random* och *föredra ingångsriktning* så var det tydligt att den senare såg klart mycket bättre ut, speciellt om hindrena var relativt stora. Därför togs implementationen för att byta heuristik bort från programmet.

Denna algoritm liknar beteendet zombies har i Call of Duty zombies. Dessa zombies vet alltid vart spelaren är, oavsett om spelaren är synlig eller inte. Zombien går hela tiden raka vägen mot spelaren tills den krockar med ett objekt då den väljer ett håll för att gå runt. Detta beteende fungerar mycket bra när det är många zombies på banan samtidigt. Algoritmen är dessutom beräkningslätt vilket gör det möjligt att ha många agenter aktiva samtidigt.

Avslutningsvis så kan vi fastställa att denna algoritm fungerar väldigt bra för vissa ändamål. Den behöver en bana som inte är allt för komplex. Ganska många banor i spel är dock relativt simpla. Den skulle till exempel klara den stora majoriteten av ett open world-spel då dessa består till stor del av träd, stenar, kullar, hus osv. Den lämpar sig även bra till AI som inte ska vara så smarta, till exempel djur eller zombies som nämndes tidigare.

Litteraturförteckning

[1] Introduction to A*. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. Hämtad: 2021-12-17.

[2] p5.js javascript library for creative coding. https://p5js.org. Hämtad: 2021-12-20.

[3] Visual Studio Code live share. https://visualstudio.microsoft.com/services/live-share/. Hämtad: 2021-12-20.

[4] Yaldex game-programming path finding. http://www.yaldex.com/game-programming/0131020099_ch08lev1sec1.html. Hämtad: 2021-11-23.

[5] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.