Polyteknisk eksamens projekt

Evolutionære multiagent-systemer i computerspil

Afleveret d. 3. december 2007

Casper la Cour, s022892

Mads Terp, s022001

Abstrakt

Kunstig intelligens i computer spil spiller en stadig større rolle i forhold til at gøre computerspil interessante. De traditionelle metoder til agentstyring i spil bliver sværere og sværere for programmørerne at overskue, efterhånden som de laves mere komplekse, og grænsen for hvad de kan strækkes til, lader efterhånden til at være nået. I dette speciale undersøges mulighederne for at styre agenter i spil ved hjælp af beslutningstræer hvis indhold optimeres med en genetisk algoritme. Idéen hermed er, at det ikke er nødvendigt for en programmør selv at specificere agenternes adfærd. I stedet skal adfærden fremkomme ved at indstille beslutningstræerne med den genetiske algoritme. På den måde spares programmøren for arbejdet. Der laves en implementering af et sådan system for at påvise potentialet ved denne teknik. Opgaven er ikke tænkt som en lejlighed til at udvikle et egentligt spil, men til at påvise et eventuelt potentiale med et *proof of concept*. Ud fra målinger og test af det udviklede agentstyringssystem ses det, at agenterne bliver bedre som følge af evolutionen, og hovedkonklusionen er, at metoden har vist sig at have potentiale. Især fordi der angiveligt vil kunne opnås endnu mere overbevisende resultater, hvis man udvider systemet så agenterne bliver mere komplekse, end dem det har været muligt at implementere i dette projekt.

Indholdsfortegnelse

Forkortelser og ordforklaringer:

Der gives her en række forklaringer på ord og forkortelser, som bruges i forbindelse med denne rapport. Første gang begreberne bruges i rapporten vil de være markeret med fed skrift. Efterfølgende vil der ikke være nogen markering af dem. Nogle af begreberne har bredere betydning, end blot indenfor spil, men når de nævnes i denne rapport, er det altid i forbindelse med **FPS** spil, som projektet beskæftiger sig med.

Agent	Agenter er her en mod- eller medspiller, som er styret af computeren. Agenter benævnes også som **NPC**'er eller **Bot'**s, men ordet agent indikerer som regel en mere teoretisk betydning, hvor Bot og NPC oftest findes i meget praktiske sammenhænge.
Agentstyring	Den måde en agents handlinger styres på.
AI	**A**rtificial **I**ntelligence (Kunstig Intelligens)
Bot	Bot er en forkortelse af ordet robot, men i spilsammenhæng menes en computerstyret agent, altså det samme som en NPC.
Callback	I forbindelse med dette projekt er et callback når der kaldes fra C++ til en funktion i scriptkoden.
Drag-and-drop	Drag and drop vil sige at man ved hjælp af musen trækker et objekt hen til en ønsket placering og slipper det der. På den måde kan objekter placeres uden f.eks. at angive koordinater eller lign.
FPS	**F**irst **P**erson **S**hooter. FPS spil er en type af spil, hvor man styrer en personrolle, typisk med et udsyn fra personens øjne. Spillet foregår som regel i en virtuel 3d verden, og målet er ofte at skyde andre spillere – enten menneskelige, eller computerstyrede.
FSM	**F**inite **S**tate **M**achine (endelig tilstands maskine).
GA	**G**enetiske **A**lgoritmer.
Gametick	Er game enginens puls. Der udføres et gametick hver 30'te millisekund som opdaterer grafikken og fysikken i spillet. Alle grafiske objekter opdateres i her. F.eks. opdateres AI spillerens position på banen.
Gen	Se under **kromosom**.
GDC	**G**ame **D**evelopers **C**onference.
Hakke	Når et spil 'hakker' betyder det, at grafikken kører i 'hak' og ikke flydende, som den bør. Det sker typisk, når for meget CPU kraft skal bruges til at løse andre problemer end rendering af grafik.

Kromosom	Når der tales om kromosomer i denne opgave er det kromosomer i forbindelse med genetiske algoritmer. Et kromosom består af en samling gener. Kromosomet er et bud på en løsning eller et individ som den genetiske algoritme arbejder med.
Multi-agentsystem	I dette projekt menes hermed blot et system med flere agenter som agerer i samme miljø.
NPC	**N**on **P**layer **C**haracter. En personrolle i et spil, hvis handlinger styres af computeren.
Pathfinding	Den måde hvorpå en agent finder rundt på en given bane i spillet. Indendørs pathfinding betegner, at agenten skal finde rundt inde i et bygningsobjekt, og skal kunne gå rundt i gange, på trapper og flere komplicerede steder. Udendørs pathfinding betyder, at agenten skal finde rundt i et åbent miljø, hvor der findes flere bygningsobjekter.
Planning	En teknik inden for kunstig intelligens der benyttes til at planlægge en sekvens af handlinger, som fører til et ønsket mål.
TGE	**T**orque **G**ame **E**ngine. Se under Torque.
Torque	Game enginen der benyttes i dette projekt hedder Torque og er udviklet af Garage Games. Enginen implementerer al rendering af grafik, al fysikken i spillets virtuelle verden og netværks muligheder. På den måde kan dette projekt fokusere på selve udviklingen af agenter i spil.

Kapitel 1 Indledning

I dette kapitel indledes og afgrænses opgaven. Forskellige typer specifikationer gives.

1 Indledning

Denne rapport er udarbejdet i forbindelse med et polyteknisk eksamensprojekt, vedrørende evolutionære **multi-agent** systemer i spil, på Danmarks Tekniske Universitet, med Thomas Bolander som vejleder, og Rasmus Hartvig fra IO Interactive, som ekstern vejleder. I rapporten gennemgås og analyseres de idéer, og det programmel, der er udviklet i forbindelse med eksamensprojektet. Projektet indbefatter således både denne rapport og et implementeret agentsystem i et 3D computer spil, som er konstrueret med game enginen, **Torque**.

Da der i dette system indgår flere **agenter**, som spiller med og mod hinanden, er der tale om et multi-agentsystem. Begrebet multi-agentsystemer ses i forskellige forbindelser, men i denne sammenhæng dækker begrebet alene over, at der er flere agenter, som arbejder sammen og mod hinanden, dog uden at der på noget tidspunkt foregår forhandlinger mellem agenterne, som man nogle gange ser det, i forbindelse med begrebet multi-agentsystemer. Projektet omhandler et agentsystem, hvor agenterne er styret af beslutningstræer, der er optimeret af genetiske algoritmer. Agenterne skal agere i en 3D verden, i et **FPS** (First Person Shooter) spil, hvor det typisk handler om at skyde hinanden. I det følgende gives baggrunden for hvorfor projektet har sin berettigelse.

1.1 Baggrund

For at et øge underholdningen i computerspil er det ofte sådan, at den menneskelige spiller møder computerstyrede spillere i spillets virtuelle verden. Disse computerstyrede spillere kaldes ofte agenter eller **NPC**'ere (**N**on **P**layer **C**haracters). Baggrunden for opgaven findes i, at det i praksis er svært at designe og programmere disse med- og modspillere i computerspil, der er styret af computeren, hvis de samtidigt skal agere intelligent. For at øge underholdningsværdien er det nemlig vigtigt, at de computerstyrede agenter virker overbevisende på den menneskelige spiller, så der gives en følelse af, at det ligeså godt kunne være mennesker, der styrede modspillerne.

Computerspilsmarkedet vokser sig større og større og den interaktive underholdnings industri er ved at overhale filmindustrien i årlig omsætning.[1] Samtidigt giver udviklingen af nye og hurtigere computere øget mulighed for at udvikle stadig mere avancerede metoder indenfor alle computerspillets facetter. I den forbindelse har også 'kunstig intelligens i computerspil' fået mere fokus og spås en stor fremtidig rolle i interaktiv underholdning af bl.a. et af de førende sites indenfor spil **AI** (AI Depot), hvor det udtales "*With the increasing commotion about the importance of Artificial Intelligence in games, there is little doubt that it will be a key component of best-sellers in the near future.*"[2] Desuden kan **[AN]** citeres for følgende "*With increasing CPU power available for AI computations, NPC behavior will become more and more sophisticated.*"[3] Motivationen for at beskæftige sig med et emne som dette, er alene af den grund forståelig. Dertil kommer, at der i mange spil i dag stadig benyttes simple metoder[4], hvorfor der er et potentiale i at forske i området.

Samtidigt tyder noget på, at spilanmeldere er begyndt at forvente, at der også er brugt tid på AI'en i de nye computerspil. Hos GameSpot, som er et af de mest kendte anmelder sites for computerspil, kommenteres oftere og oftere på spillenes AI. F.eks. kommenterer GameSpot på AI'en i både det

[1] Fra **[Rambøll]**
[2] Citat fra **[AI-DEPOT2]**
[3] Citat fra **[AN]** side 60
[4] Kilde: **[MIT]**

nye ”Kane & Lynch: Dead Men”, ”BlackSite: Area 51” samt ” Half-Life 2” og ”Unreal Tournament 3”, og det går ud over anmeldelsen, hvis ikke AI’en er interessant.[5]

Desuden er genetiske algoritmer et område, som traditionelt ikke har været udnyttet i forbindelse med spil, hvilket understreges i følgende citat, fra et af de førende spil sites, ’Get In The Game News’, *”Genetic algorithms are one of a variety of AI techniques that have been extensively explored by academics but have yet to push their way into game development. However, they offer opportunities for developing interesting game strategies in areas where traditional game AI is weak.*”[6] Hvilket også motiverer til at se nærmere på teknologien i spilsammenhæng.

Formentlig derfor, har det også været svært at finde konkrete eksempler på computer spil som anvender genetiske algoritmer. Det er lykkedes at finde enkelte, men ikke til direkte styring af NPC’ere, som det ses i dette projekt, og slet ikke sammen med beslutningstræer. Det kan naturligvis ikke udelukkes fuldstændigt, at dette ikke skyldes spilproducenternes forsøg på at holde det hemmeligt, men omvendt er det forholdsvis let at finde eksempler på brug af anden AI til NPC styring, og udtalelser om, at GA endnu ikke anvendes i spil AI. Et eksempel herpå er allerede givet ved forrige citat, men også på aigamedev.com kan man finde udtalelser, der understøtter denne fornemmelse. Her udtales bl.a. *”Modern game developers certainly uses specific optimization techniques heavily, in particular for character animation, evolutionary algorithms still haven’t found a regular place in the development process — including for game AI.*”[7]

På gameai.com[8] er en liste over 50 spil, som ifølge sitet skulle indeholde spil, der gør brug af særligt interessante AI metoder. Heriblandt finder man enkelte spil, der hævder at benytte genetiske algoritmer, men i alle tilfældene er der tale om strategispil, som typisk er tur baseret, og dermed ingen FPS spil, som er den type spil, der kigges på i dette projekt. Samtidigt anvender ingen af spillene beslutningstræer. Faktisk er det ikke lykkedes at finde andre eksempler på denne kombination til **agentstyring**.

Det at kombinere agentstyring med genetiske algoritmer giver den fordel, at der er mulighed for at få agenter, der opfører sig forskelligt og er på forskellige niveauer i sværhedsgrad. På den måde kan den menneskelige spiller opleve agenter med forskellig adfærd. Dette er med til at øge følelsen af, at agenterne ligeså godt kunne være styret af andre mennesker. Med optimering ved hjælp af genetiske algoritmer, kan man opnå dette, idet agenterne bliver evalueret efter hvor gode de er til at vinde over det andet hold, og dermed haves et fingerpraj om hvor svære de er at vinde over - også for en menneskelig spiller. Dvs. at en mindre øvet menneskelig spiller kan komme til at spille mod agenter fra én generation, med en lavere fitness værdi, og en mere øvet menneskelig spiller kan prøve kræfter med agenter, som har opnået flere fitness point - typisk fra en senere generation.

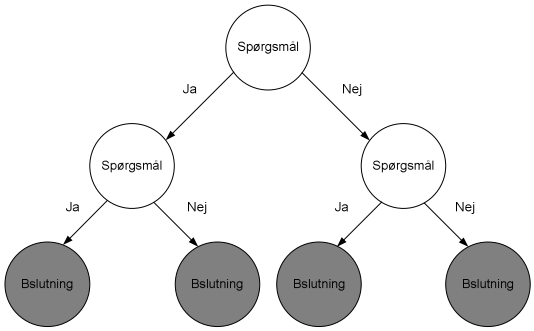
I afsnit ’5.1.3 Alternativer til GA’ sammenlignes genetiske algoritmer med andre potentielle metoder til optimering af beslutningstræerne.

[5] Kilde: **[GS]**
[6] Citat fra **[GIGNEWS]**
[7] Kilde: **[AIGAMEDEV]**
[8] Kilde. **[GAMEAI]**

1.2 Problemstilling

I dette projekt designes et system, hvor **agenter** i 3D **FPS** spil optimeres med en genetisk algoritme. Systemet skal kunne håndtere mange input og handlemuligheder for agenterne. Til at koble input og handlinger sammen, og dermed definere agentens adfærd, benyttes beslutningstræer. Det er hensigten at disse agenters adfærd, i så høj grad som muligt, skal optimeres uden menneskelig indblanding, da denne indblanding reelt vil begrænse antallet af mulige input og handlemuligheder for agenten, idet beslutningstræerne bliver svære at overskue for mennesker, når de bliver meget store. Samtidig skal agenterne opføre sig intelligent. Intelligent vil i dette projekt sige, at de skal klare sig godt, når de spiller med og mod andre spillere. Konkret betyder dette, at agenternes 'intelligens' bliver målt på den fitness de opnår, når de spiller mod modstandere, f.eks. agenter styret af **FSM**'er. Optimeringen sker med en metaheuristik (genetiske algoritmer), der skal bedømme hvor gode agenterne er, og derudfra forsøge at fremavle nye bedre agenter i næste generation af agenter. Dermed skal metaheuristikken optimere, hvordan de konkrete beslutningstræer skal designes, for at agenterne klarer sig så godt som muligt i spillet.

For helt kort at gøre det klart hvad et beslutningstræ er, ses et binært beslutningstræ på Figur 1. Som det også ses på figuren, repræsenterer knuderne i træet spørgsmål og kanterne et svar på det stillede spørgsmål. I ethvert blad findes herefter en beslutning, som passer til de svar, der er givet for at nå bladet. I dette projekt stilles spørgsmålene efter en bestemt form, nemlig som: 'er denne inputværdi større end en given grænseværdi?'. Samtidigt låses spørgsmålene på hvert niveau i træet, således at hver knude i et givet niveau i træet spørger om det samme. Hver knude i træet får tilknyttet en grænseværdi, som indstilles af den genetiske algoritme. Mere om beslutningstræer i afsnit '3.2 Beslutningstræer'.



Figur 1: Viser et binært beslutningstræ.

Den genetiske algoritme skal altså designes således, at den optimerer beslutningstræernes grænseværdier og træets beslutninger (blade). Der findes mange forskellige teknikker indenfor genetiske algoritmer, og det er også meningen, at en delmængde af disse skal afprøves, for at se hvilke teknikker, der bedst løser optimeringsproblemet. Kort fortalt fungerer den genetiske algoritme, ved at simulere Darwins evolutions teori – "Survival of the fittest". Idéen er, at man har en population af agenter (beslutningstræer) der er tilfældigt initialiseret. Denne population evaluerer man, for at finde ud af hvilke træer der er bedst. Derefter bruger man evalueringen til at vælge nogle af de bedste træer ud, og så kombinere disse træer til nye og (forhåbentligt) bedre træer. Derefter starter processen med evalueringen forfra, og der fortsættes indtil træerne er blevet 'gode nok'.

Herved er det computeren, i stedet for programmøren, der kommer til at bruge en masse tid på at optimere agenterne. Derfor har en del af opgaven også bestået i at spare så meget som muligt på

regnekraften, så den genetiske algoritme kan gennemføre træningen inden for rimelig tid. Da den genetiske algoritme i sig selv næsten ikke tager tid at afvikle, er der ikke lavet køretidsanalyser af denne. Regnekraften bruges derimod på at lade agenterne spille mod hinanden så de kan evalueres, og der er derfor gjort talrige bestræbelser på at få denne proces til at foregå så hurtigt som muligt.

Da der er uanede muligheder for at konstruere den genetiske algoritme samt muligheder for at styre denne, bærer opgaven meget præg af at det konstant har været nødvendigt at tage valg som begrænser agentsystemet. Det har været en af de sværeste udfordringer ved opgaven at træffe disse valg, og vil samtidigt være det i forbindelse med eventuel videreførelse af opgavens idéer.

Det ønskes altså med denne opgave, at lave et *proof of concept*, som skal vise, om det, med denne kombination af to forskellige grene indenfor **AI,** er muligt at lave et system, hvor programmøren ikke behøver at fastdefinere hver enkelt agents adfærd i detaljer. Sidst skal det siges, at agentsystemet demonstreres med game enginen Torque, der hjælper til at gøre det lettere, at lægge projektets fokus i agenternes styring, ved at levere en 3D virtuel verden som agenterne kan agere i.

1.3 Udviklingen inden for AI i spil

I dette afsnit ses kort på hvilke løsninger der i dag anvendes i kommercielle spiludgivelser. Da der er relativt meget hemmelighedskræmmeri angående AI i spilbranchen, er det desværre kun muligt at give et begrænset indblik i hvordan AI programmeringen foregår. Det vurderes dog, at tendenserne som ses i forbindelse med dette indblik, overordnet set er forholdsvis repræsentative for hele spilbranchen.

En af de simpleste, men mest anvendte, måder at lave NPC'ernes AI systemer på, er ved hjælp af FSM'er. Denne opfattelse styrkes ved at se udtalelser omkring FSM'er fra folk, der har beskæftiget sig med emnet som" *...It has been the method of choice when programming computer agents for many years now...*"[9] En påstand som bakkes op af PhD Jeff Orkin fra MIT Media Lab, der selv har været med til at udvikle adskillige AI'er i spilbranchen[10], med udtalelsen " *If the audience of **GDC** was polled to list the most common A.I. techniques applied to games, undoubtedly the top two answers would be A* and Finite State Machines (FSMs).*"[11]

Små FSM'er giver ikke særligt intelligente agenter, men de er til gengæld lette både at programmere og debugge, hvilket kan være en vigtig fordel i forhold til at overholde deadlines i et spil firma. F.eks. kan det nævnes, at FSM'er blev benyttet til at styre modstandere i de første 'Hitman' spil fra spilfirmaet 'IO Interactive'.[12]

I agentsystemer, der er styret af FSM'er eller lignende simple metoder, er et af de problemer som man oftere og oftere ser, at få de computerstyrede spillere til at opføre sig 'intelligent nok' til stadig at imponere den menneskelige spiller. For hvis tidligere spils FSM'er skal overgås med nye FSM'er, der virker mere intelligente, kræver dette nemlig efterhånden meget store og komplekse FSM'er, som dermed bliver svære at konstruere og især at overskue. Der skal i disse systemer tænkes over alle de mulige situationer en agent kan befinde sig i, og hvilken handlingstagen hver

[9] Citat fra **[Tisher]** side 30
[10] Kilde: **[MIT2]**
[11] Citat fra **[MIT]** side 1.
[12] Kilde: **[Lind]**

enkelt situation kræver, for at agenten virker intelligent. *"Det er mit indryk at spilproducenter før i tiden har valgt mest at satse på FSM'er og andre tilsvarende simple agentstyringsmetoder, men at billedet efterhånden lader til at vende, således at vi i fremtiden vil se mere avancerede metoder."* udtaler Rasmus Hartvig som er hovedansvarlig for det kommende Hitman spils AI hos IO Interactive.

Et eksempel på mere avanceret AI ser man i spillet *F.E.A.R.* hvor man har erkendt, at det ikke var muligt arbejde med FSM'er i den størrelse, der var nødvendige for at opnå den ønskede kompleksitet. Her har man i stedet valgt at benytte **planning** til at styre agenterne. Tilgangsvinkelen med planning er radikalt anderledes end den i dette projekt, idet der er valgt at lave mere reaktive agenter her. Ikke desto mindre illustrerer valget af planning frem for FSM'er i *F.E.A.R.* at FSM'erne har deres begrænsninger, og at grænsen er ved at være nået.[13]

Et andet eksempel fra den virkelige verden er rallyspillet 'Colin McRae rally 2', der har brugt et neuralt netværk til at styre 'bilagenterne'. Følgende citat af Jeff Hannan, der har lavet det neurale netværk i 'Colin McRae rally 2', begrunder ideen. *"I tried to create a set of rules to control the car, but was unsuccessful. I thought that neural networks would help me find a quick solution, by modelling the way I drove the car. In the end, the neural networks made it very easy to create the AI."*[14]

1.4 Rapport strukturen

Rapporten indeholder 11 kapitler, hvor indledningen er det første. Efter at have defineret problemstillingen, kommer der i kapitel 2 en introduktion til kunstig intelligens i den type computerspil, som dette projekt beskæftiger sig med. Dette kapitel er ikke direkte relevant for en læser, der kun er interesseret i hvordan de genetiske algoritmer er designet til optimeringen af systemet. Kapitlet giver dog en introduktion til de problemtyper, der findes i forbindelse med agenter i spil og de specifikke problemtyper, der er i forbindelse med dette projekt og giver derfor en læser, der ikke tidligere har arbejdet med kunstig intelligens i spil, bedre forudsætninger for at læse resten af rapporten. Efter denne introduktion gives en mere udførlig beskrivelse af hvordan beslutningstræer og tilstandsmaskiner fungerer i kapitel 3.

Kapitel 4 beskæftiger sig med de genetiske algoritmer på generelt niveau, og siden hen ses hvordan de tilpasses, så de kan optimere evolutionære agenter i kapitel 5. I kapitel 6 beskrives kort implementeringen af genetiske algoritmer, hvorimod beskrivelsen af implementeringen af hjælpefunktioner og lignende spil specifik kode udelades fra hovedrapporten – og findes i stedet i appendiks, da dette ikke er fokusområdet for projektet.

Efter denne indførelse i projektet og de valgte løsninger ses kort på softwaretest og herefter på parameter tuning og analyse, som præsenteres i henholdsvis kapitel 7 og 8. Kapitel 9 fortæller om projektets fremtidige potentiale, som efterfølges af konklusionen i kapitel 10. Kapitel 11 indeholder litteraturlisten og kapitel 12 indeholder appendiks.

[13] Kilde: **[MIT]**
[14] Citat fra **[McRae]**

Kapitel 2 Kunstig Intelligens i FPS spil

I dette kapitel beskrives, hvor i FPS spil kunstig intelligens normalt findes, og hvilke klassiske problemstillinger, der findes indenfor området.

2 Kunstig intelligens og agenter i 3D FPS computerspil

I dette afsnit gives en introduktion til nogle af de problemer, der skal løses i forbindelse med implementeringen af agenter styret af kunstig intelligens i computerspil. Endvidere gives en kort indførelse i game enginen Torque og dens virtuelle verden.

2.1 Agenter, sensorer og handlemuligheder

Fælles for alle computerstyrede agenter er, at de har brug for en række *sensorer*, der fortæller agenten noget om hvordan den verden, agenten befinder sig i, ser ud. F.eks. kunne agenten have en sensor der fortæller den, hvor mange fjender der er i syne, hvor langt der er til nærmeste health kit m.fl. Undertiden kaldes disse sensorer for input til agenten. Agenterne har desuden en række handlemuligheder, som de kan udføre; dette kan f.eks. være at sigte på en fjende, at skyde og bevæge sig m.fl. Udover dette findes der mange forskellige typer af agenter, der kan benyttes i forbindelse med spil, som hver især har sine fordele, ulemper og kompleksitetsniveauer.[15] Disse forskellige typer af agenter er samlet under begrebet 'agentteorier', som ikke kun dækker agenter i spil, men som bredere afdækker agentsystemer i forbindelse med kunstig intelligens som område. En dybere beskrivelse af agentteorierne falder dog udenfor denne rapports tilsigtede dækningsområde. Nærmere beskrivelser af agentteorier kan f.eks. findes i **[AI]**. I de næste afsnit beskrives i stedet nogle af de velkendte problemtyper, som agenter i spil skal løse.

2.2 Agentstyring

I forbindelse med styring af agenter i et FPS spil, er der brug for en række forskellige metoder, som agenten kan benytte sig af, når denne skal interagere med den menneskelige spiller i den virtuelle verden spillet foregår i. Grundlæggende har agenten brug for en måde at beslutte hvilke handlinger den skal udføre, givet de input den får gennem sine sensorer; området kaldes herefter for *agentstyring*. Da agentstyring definerer agentens adfærd, er dette uundværligt i spil. Måden agenterne styres på i forbindelse med dette projekt er ved at benytte beslutningstræer til at sammenkoble agentens opfattelse af hvordan verden ser ud (via dens input), med hvilke handlinger den skal udføre – mere om dette i afsnit '3.2 Beslutningstræer'.

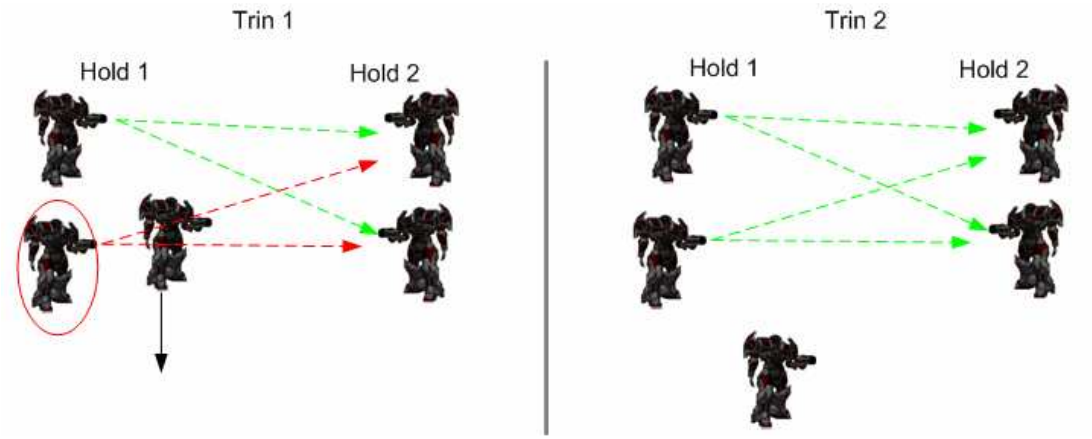
2.3 Multiagentsystemer i spil

I de fleste spil er der mange agenter på samme bane på samme tid, og i avancerede systemer samarbejder agenterne ofte. I denne form for multiagentsystem, hvor agenterne samarbejder, og danner en form for hold, har agenterne enten brug for en måde at kommunikere på, eller en anden måde, at få oplysninger om de andre agenter på. Som regel er der højst otte agenter på hvert hold, da banerne ofte har en størrelse, der ikke gør det ønskværdigt, at have flere agenter på banen ad gangen. I forbindelse med optimeringen af agenter, er det derfor smart at holde in mente, at agenterne skal lære at begå sig i et lignende miljø.

Forskellige udfordringer kan opstilles for denne type agenter, hvor det mest fundamentale er, at de ikke må modarbejde hinanden, hvis de er på samme hold – eksemplificeret med Figur 2. Det er her kritisk, at agenten markeret med den røde ring ikke skyder før holdkammeraten er udenfor skudlinien – og samtidig har agenten brug for en måde at kommunikere til agenten, der står i vejen,

[15] Fra **[AI]**, **[AGP1]** og **[AGP2]**

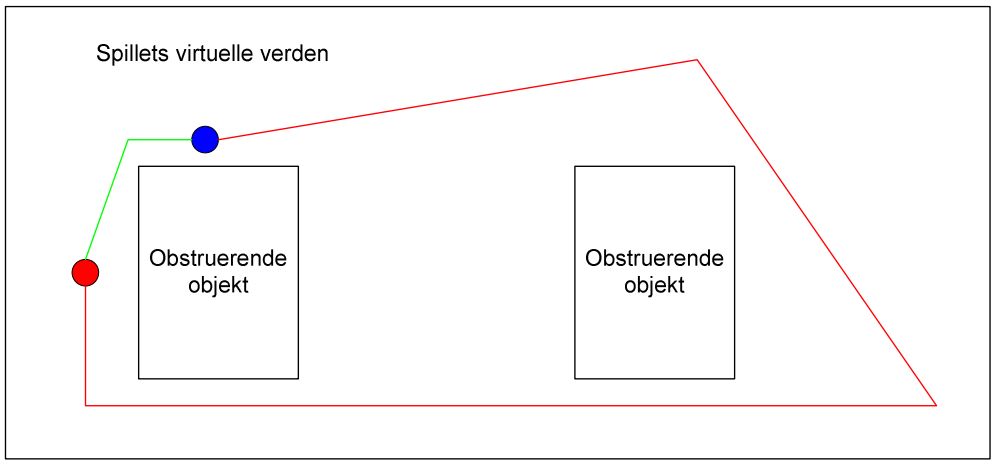
at denne skal flytte sig. Når denne agent har flyttet sig, er der frit udsyn for agenten markeret med den røde ring, som nu kan skyde. Agenterne kan gøres næsten ubegrænset avanceret, og næste skridt kunne være at lade dem udvikle en fælles taktik for, hvordan de bedst vinder over et andet hold. Dette samarbejde vil i denne opgave komme til udtryk ved at agenterne deler information om bl.a. hvilken strategi de hver især anvender, hvor de befinder sig, og hvor de har set fjenden.



Figur 2: Illustration af to hold, der kæmper mod hinanden. De stiplede linier indikere skud fra hold 1 mod hold 2 – de grønne er der hvor agenterne skal skyde, og de røde der hvor de ikke skal skyde, da der her står en medspiller i skudlinien. Agenten fra hold 1 markeret med den røde ring, skal altså vente med at skyde til holdkammeraten er væk fra skudlinien. I Trin 2 har agenten flyttet sig, og skudlinien er fri.

2.4 Pathfinding

I forbindelse med FPS spil, er en af de fundamentale, men samtidig væsentlige metoder, som agenten har brug for, en måde at bevæge sig på, i den virtuelle verden. Efterfølgende kaldes denne problemstilling for **pathfinding**. I 'Torque' game enginen har hver agent på forhånd en funktion, der gør, at agenten kan løbe hen mod en position på banen i en lige linie. Denne tager dog ikke højde for, at der kan være omstændigheder, der gør, at agenten ikke kan komme hen til den ønskede position. F.eks. kunne der være en bygning i vejen, eller en dyb kløft, der gjorde, at det slet ikke var muligt at komme hen til den ønskede position. Det eneste funktionen gør, er basalt set at sætte agenten til at gå i retningen (i en lige linie) mod det punkt han gerne vil hen til. Pathfindings problemet er på ingen måde simpelt at løse, idet agenten skal undgå at løbe ind i vægge, træer, vand, samt objekter der kan bevæge sig, som f.eks. andre agenter. Derudover er der terrænforskelle der gør, at agenten ikke kan komme op fra f.eks. en kløft. Samtidigt skal NPC'en kunne bevæge sig rundt i den verden spillet foregår i, på en realistisk og intelligent måde, for at overbevise den menneskelige spiller. Med en sti, der får agenten til at se intelligent ud, menes en kort sti, der ikke indeholder unødvendige omveje, som et menneske ikke ville benytte. Denne situation er illustreret på Figur 3 og farvekoderne er beskrevet i figurteksten.



Figur 3: Illustration af valg af en 'god' sti. Blåt punkt: Her starter agenten. Rødt punkt: Her vil agenten hen. Grøn linie: En 'god' sti, gør at agenten virker 'klog'. Rød linie: En 'dårlig' sti, gør at agenten virker 'dum'.

En sti, der får agenten til at se intelligent ud, kan vælges ud fra flere forskellige forudsætninger, men ingen af de mulige metoder er simple i en 3D verden. I dette projekt benyttes en graf til at repræsentere hvor agenten kan gå hen, - hvert punkt i grafen svarer til en nøje udvalgt position på banen og hver vægtet kant i grafen repræsenterer så, at der er en direkte sti mellem de to punkter med en afstand svarende til vægten på kanten. Herefter benyttes algoritmen A* (a stjerne) til at finde korteste sti mellem agentens startposition og den position agenten gerne vil hen til. Problemet er naturligvis at indlægge de positioner på banen, der gør, at agenten kan bevæge sig frit. Som nævnt er dette område på ingen måde simpelt at løse og der er mange faldgruber, men da dette område ikke har hovedfokus i dette projekt, henvises den interesserede læser i stedet til appendiks '12.1 Pathfinding' for uddybende forklaring af hvordan problemet er løst i dette projekt.

2.5 Kort om game enginen Torque Game Engine (TGE)

Torque game enginen har en række forskellige fordele og ulemper hvor af nogle af dem belyses i følgende afsnit. Valget er faldet på Torque primært fordi der tidligere er opnået kendskab til denne game engine, og det dermed blev lettere at koncentrere sig om den genetiske algoritme og de beslutningstræer, som er opgavens egentlige fokus. En anden væsentlig grund er at Torque er forholdsvis veldokumenteret og samtidigt ligger på et prisniveau som er overkommeligt.

TGE er en 3D game engine, der er skrevet med programmeringssproget C++ og assembler - og er opbygget af 2286 klasser med 20.160 funktioner og indeholder 494.990 liniers kode.[16] Derudover er der implementeret et scripting sprog, som bruges til at programmere specifikke spil eller baner, ved at implementere funktioner, der er specifikke for det spil der laves, og ved at have adgang til at kalde en række generelle funktioner, som er implementeret i TGE kildekoden.[17] Det siger sig selv, at med så stor en kodebase, så tager det en del tid at sætte sig ind i hvordan enginen hænger sammen og hvordan man får de forskellige dele til at tale sammen. Derudover har enginen nogle bugs, som også har gjort debuggingen tungere – herunder kan f.eks. nævnes, at den funktion der returnerer en agents rotation, ikke virker i bestemte intervaller. Hertil har det dog været til stor

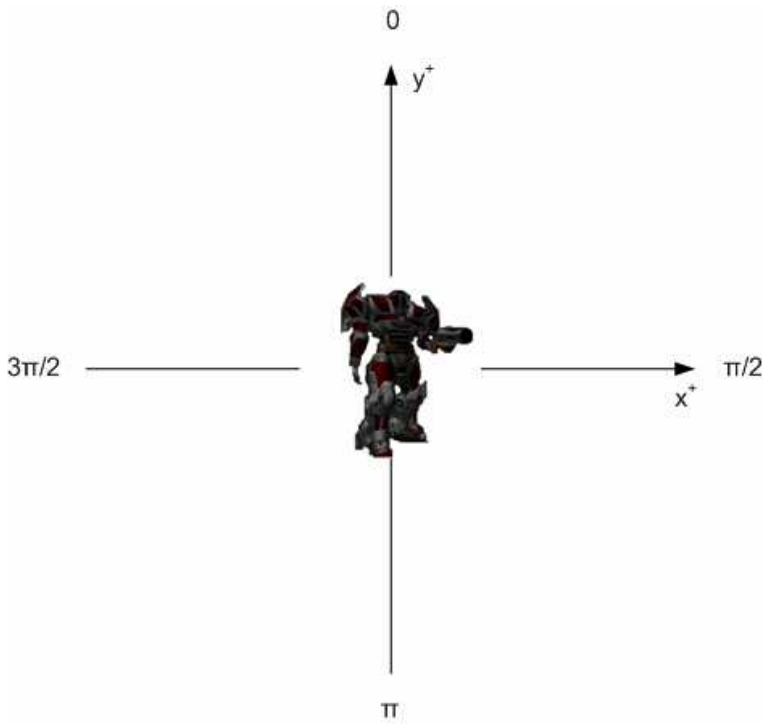
[16] Fra **[TorqueDoc1]**
[17] Fra **[TorqueDoc2]**

hjælp at kunne søge i fora vedrørende Torque og enginens dokumentation, **[GarageGames]** og **[TDN]**.

2.5.1 Torques "spilleregler"

Som nævnt er Torque en 3D game engine. Torques 'opgave', i forbindelse med dette projekt, er at gøre det muligt, at implementere et agentsystem, der optimeres af genetiske algoritmer, uden at skulle programmere grafik rendering, fysikken i den virtuelle spilverden, samt at tegne forskellige modeller af bygninger, spillere og andre objekter, der er brug for i forbindelse med spil.

Den virtuelle verden Torque implementerer, repræsenteres ved et 3D koordinatsystem, hvor akserne, ligesom i 'den virkelige verden', repræsenterer frem/tilbage, til højre/venstre og op/ned. Endvidere har hvert enkelt objekt, der er placeret i denne virtuelle verden, en rotation. Denne rotation måles ud fra at y-aksens retning repræsenterer rotationsvinklen 0, og ellers på samme måde som retninger bestemmes ud fra enhedscirklen. X-aksens retning vil derfor repræsentere en rotation på Π/2 osv., se Figur 4.[18] Endvidere kan spilleren også roteres i z aksen.



Figur 4: Objekternes rotation gives i forhold til dette koordinatsystem, hvor retningen mod y uendelig, vil returnere en rotation, der er lig med 0.

I Torque skelnes der endvidere mellem selve spillet, og en given 'bane'. Spillet er det der starter, når man eksekverer TorqueDemo.exe filen. Når selve spillet er startet, kommer man ind til en start menu, hvor man så kan vælge, at starte en given bane. Når banen er startet styrer man sin personrolle i spillet – og det er i den forbindelse, at agenter, styret af kunstig intelligens, kommer ind i billedet.

[18] Fra **[TDN]**

2.5.2 Baner i Torque

Til at lave en bane i Torque, er der implementeret en grafisk brugergrænseflade, hvor man kan se den bane, man er i gang med at lave, direkte. Ofte er det en god ide at lave terrænet først, hvortil der er en række hjælpemidler til rådighed. Bl.a. kan man, på et givet areal, hæve og sænke terrænet efter ønske, farve det således at det ligner græs, grus eller bjerge, udglatte terrænet, så det kommer til at ligne bakker eller gøre det helt fladt, så det er lettere at placere bygninger, m.m.

Herefter kan man efter **'drag-and-drop**' metoden, hente forskellige slags objekter ind på banen. Det kan være bygninger, træer, våben eller andre slags objekter. Alt i alt gør dette værktøj det forholdsvist let at lave en bane, når man har sat sig ind i hvordan de forskellige dele af editoren virker.

Selvom det ikke er så kompliceret at lave baner i Torque tager det alligevel lang tid, da der er mange objekter at vælge mellem, og man ikke kan se dem før man har sat dem ind på banen, hvilket gør man bliver nødt til at prøve sig frem og så slette de objekter man ikke vil bruge alligevel. Derfor er der i forbindelse med dette projekt også kun implementeret én test bane, som agenterne kan udvikle sig på.

På Figur 5 ses 3 billeder af den bane som blev udviklet og benyttet i forbindelse med arbejdet med agentstyringssystemet. Banen er samtidigt den bane som agenterne spiller mod hinanden på og gennemgår deres evolution på. De to røde ringe på det midterste billede angiver holdenes startpositioner.

Figur 5: viser den bane som blev udfærdiget i Torque til brug i dette projekt fra tre vinkler.

2.6 Agenternes input og handle muligheder

Som tidligere nævnt har agenterne brug for nogle inputfunktioner og nogle handlemuligheder for at kunne opføre sig intelligent. For at give agenterne de muligheder de har brug for, har det været nødvendigt at implementere de fleste af disse inputfunktioner og funktioner til at udføre diverse handlinger i Torque. I dette afsnit gives en kort beskrivelse af de sensorer og de handlemuligheder agenterne har fået til rådighed gennem de implementerede funktioner.

2.6.1 Input funktioner (agentens sensorer)

Som standard har agenter i Torque kun nogle få redskaber til rådighed. Disse inkluderer, at man kan få sin health- og ammunitionsstatus til ethvert tidspunkt. Man kan endvidere få at vide, om der er et

objekt i vejen mellem to givne punkter på banen, ved at lave et såkaldt ’raycast’. Man kan forestille sig, at man skyder en stråle fra det ene punkt i en lige linje hen til det andet punkt, hvorefter man får at vide, hvad strålen eventuelt har ramt på vejen.

Udover disse standardfunktioner, er der i forbindelse med dette projekt implementeret en lang række funktioner, der gør, at agenten kan få de resterende inputs, som agenten skal bruge til at træffe beslutninger om hvilke handlinger den skal udføre. I det følgende vil nogle eksempler på dette projekts mere centrale funktioner blive gennemgået kort.

Blandt de første funktioner det var nødvendigt at implementere, findes flere forskellige funktioner til at undersøge sigtbarheden af diverse objekter. Det er naturligvis helt centralt, at agenten kan se andre agenter, bygninger, træer osv. For blot at nævne et stedet det bruges, er det når agenten skal bestemme hvem den skal skyde på. Men også for at se hvor den kan gå henne, om andre kan se agenten selv osv. Endvidere skal agenten have mulighed for at kende afstande hen til en masse forskellige objekter, det være sig medspillere, modspillere, healthobjekter, ammunitionsobjekter, gemmesteder og andre objekttyper. Det er naturligvis også vigtigt, at agenten kan få at vide, hvad agentens holdkammerater laver, hvad de kan se, hvem de skyder på osv. for at kunne lægge en god holdstrategi. Desuden har agenterne brug for at vide hvilke andre agenter der skyder; både med- og modspillere. Der er også brug for at agenterne ved hvem der hjælper hvem på holdet. Der er som nævnt flere funktioner, som bruges til input til agenternes træer; hvilke typer af input agenterne mere har brug for, bliver beskrevet nærmere i afsnit ’5.3’ og frem.

2.6.2 Handlemuligheder

Torque er ikke er indrettet til at agenter skal kunne interagere med objekter på banen såsom at flytte rundt på møbler eller vælte dem for f.eks. at skabe dækning. Agenterne i Torque har kun nogle simple handlemuligheder. De der er brugt i forbindelse med dette projekt er funktioner, der gør, at agenten kan trykke på aftrækkeren på sit våben, kan sigte på en position angivet ved koordinater, og endeligt, at agenten kan gå mod et punkt på banen i lige linie og stoppe ved punktet, når den er nået frem. Agenterne kan desuden hoppe over små forhøjninger, men denne funktionalitet benyttes ikke her. Nogle af de handlingsfunktioner, der er implementeret i dette projekt gennemgås i det følgende.

De fleste af handlingsfunktionerne er væsentligt mere komplekse end inputfunktionerne, og har derfor krævet et vist tidsforbrug af lave. Som eksempel herpå kan f.eks. nævnes, at det har det været nødvendigt at implementere en funktion der gør, at en agent kan pathfinde til et givet punkt på banen og herved undgå at støde ind objekter. Heldigvis kunne dette pathfindingssystem baseres på det, der blev udviklet i forbindelse med vores polytekniske midtvejsprojekt (bachelor projekt). Det var dog nødvendigt at lade systemet gennemgå en større tilpasningsproces, før det kunne anvendes i forbindelse med denne opgaves multiagentsystem, da pathfindingen bl.a. ikke var konstrueret til at kunne håndtere mere end én agent ad gangen. Andre dele af systemet måtte også laves helt om. Den interesserede læser henvises til appendiks ’12.1 Pathfinding’.

At lede efter fjender på banen er en anden af de mere centrale handlingsfunktioner, som det har været nødvendigt at implementere – bl.a. fordi funktionen også skulle koordinere med andre holdkammerater, så man leder forskellige steder på banen. Dertil kommer funktioner, som at gemme sig et sted på banen, at samle health kit op, at samle ammunition op og at sigte og skyde på fjender. Flere handlingsfunktioner er implementeret, der henvises til afsnit ’5.3’, for en beskrivelse af de anvendte handlingsfunktioner.

Kapitel 3 Strukturer til reaktive agentsystemer

I dette kapitel gennemgås teori for forskellige metoder til agentstyring. Herunder bl.a. FSMer og beslutningstræer.

3 Systemer til agentstyring

I dette kapitel ses på forskellige strukturer, der kan benyttes til agentstyring. De tre metoder der kigges på er endelige tilstandsmaskiner, beslutningstræer og vægtmatricer. Alle tre metoder kan principielt vælges til optimering med genetiske algoritmer, men de tre valgmuligheder er hver især forbundet med en række fordele og ulemper. Mere om dette i afsnit '5.1 Alternative løsningsovervejelser'.

3.1 Endelige tilstandsmaskiner

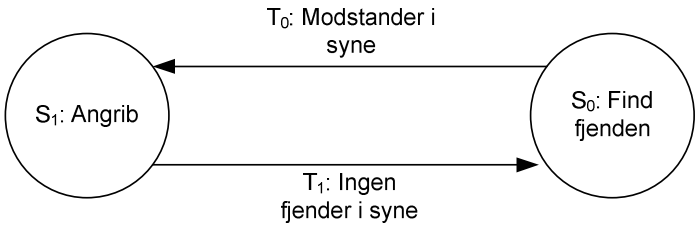
En klassisk måde at styre agenter på er at benytte endelige tilstandsmaskiner. Hver tilstand kunne så bestå af en række handlinger som agenten skulle udføre. En tilstand kunne f.eks. være 'attack' og handlingerne til tilstanden være at skyde på nærmeste fjende og forfølge fjenden. Hver transition kan så bestå af et eller flere input, som f.eks., hvor mange skud agenten har tilbage, hvor meget health den har eller hvor langt der er til nærmeste fjende. Agenten opdaterer så løbende sin situation ved at tjekke de givne input i den tilstand den befinder sig i, efter et givet tidsinterval.

3.1.1 Teori om endelige tilstandsmaskiner (FSM´er)[19]

En endelig tilstands maskine (FSM) er en abstrakt maskine, der består af et endeligt antal tilstande og transitioner mellem disse tilstande. Hver transition beskriver, ud fra logiske udtryk, hvordan man kommer fra én tilstand til en anden. Maskinen kan befinde sig i en vilkårlig tilstand til ethvert givet tidspunkt. Endvidere kan FSM'en bearbejde input resulterende i et tilstandsskifte, et bestemt output, eller en bestemt handling.

Endelige tilstandsmaskiner kan repræsenteres med orienterede grafer, der består af en række punkter (tilstande) og en række orienterede kanter (transitioner) mellem disse. Hver kant er mærket med et eller flere logiske udtryk, der fortæller hvordan man kommer fra en tilstand til en anden via den givne transition (kant).

Et simpelt eksempel på en sådan graf, ses i Figur 6. I dette eksempel repræsenterer FSM'en et meget simpelt agentstyringssystem. Agenten går i tilstand S_0 rundt og leder efter fjender. Når en modstander kommer i syne bliver udtrykket i T_0 sandt, og FSM'en skifter tilstand til S_1. I tilstanden S_1 skyder agenten efter sin modstander. Hvis modstanderen slipper ud af agentens syne (eller dør) opfyldes udtrykket i T_1, og FSM'en skifter tilstand til S_0, hvor agenten igen vil lede efter modstandere.



Figur 6 – Tilstands graf

[19] Definitionen er baseret på kilderne **[DIE]**, **[AI-DEPOT]** og **[AGP2]** s. 284

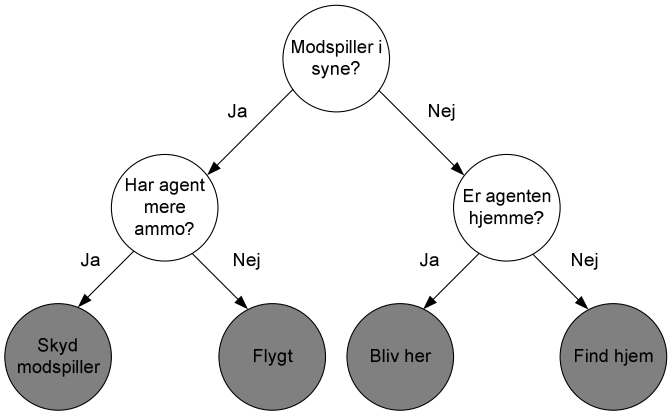
Under tiden defineres tilstands maskiner også som sprog genkendere (som regel under det engelske navn (Finite Automata) – den interesserede læser henvises her til anden litteratur, f.eks. findes en definition af Finite Automata's i **[CS]**. I forbindelse med spil, defineres tilstandsmaskiner dog som regel som ovenfor.

3.2 Beslutningstræer

I dette afsnit gives en beskrivelse af hvad beslutningstræer er og den grundlæggende teori omkring dem. I den forbindelse gives også et eksempel på hvorledes beslutningstræer kan benyttes til at styre agenter med.

3.2.1 Grundlæggende teori om beslutningstræer[20]

Beslutningstræer definerer veje gennem sine knuder svarende til en række par af spørgsmål og svar. Hver knude i træet repræsenterer et spørgsmål, og hver kant der går ud til knudens børn repræsenterer en besvarelse af spørgsmålet. Ved at starte i roden af træet og vælge svar til knuderne fås en sti ned gennem træet til et blad. Hvert blad i træet repræsenter en beslutning. Et simpelt eksempel er givet nedenfor, i Figur 7.



Figur 7: Eksempel på hvordan et beslutnings træ kan bruges til at styre en agent (vælge hvilke handlinger der skal udføres)

Man kan også sige at beslutningstræer er en måde at løse problemer på, ved at stille en række spørgsmål, for på den måde at udelukke forkerte løsninger, således, at man til sidst har 'det rigtige svar'. I dette projekt er 'det rigtige svar' et sæt af handlinger, som altså bestemmes af, hvilke svar der er givet på spørgsmålene ned gennem træet. Hvordan beslutningstræerne er tilpasset i denne opgave belyses senere i kapitel '5 Kombination af GA og beslutningstræer'. I dette projekt er spørgsmålene formuleret på en bestemt måde, nemlig om en værdi (fra et input) er større eller mindre end en bestemt grænseværdi, der er tilknyttet hver enkelt knude. Et spørgsmål kunne således være: 'har agenten mere ammunition end 10?', hvor '10' er den omtalte grænseværdi.

Et beslutningstræ er altså kendetegnet ved at være et træ, som har følgende egenskaber:

- Enhver knude i træet tester for en egenskab (spørgsmålet)

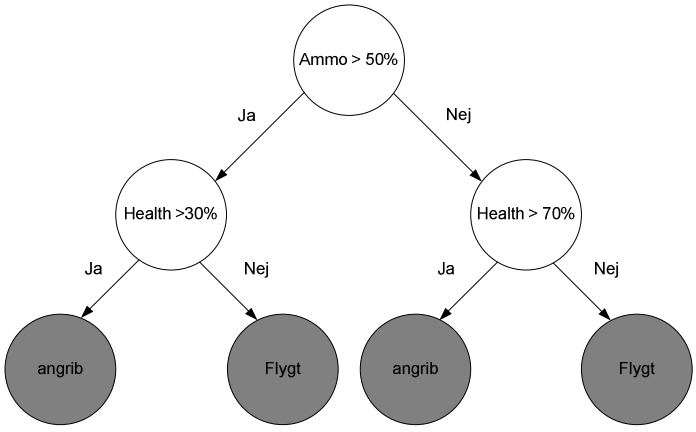
[20] Baseret på **[DT] og [AI]** s. 653-664

- Enhver kant i træet svarer til en bestemt egenskab (svaret, input fra sensorer)
- Ethvert blad i træet modsvarer en beslutning (beslutningen, sæt af handlinger)

I forbindelse med FPS spil, kan man forestille sig, at agenten skulle afgøre hvilken handling, der skulle være den næste, ud fra en række spørgsmål. Svarene til spørgsmålene kunne findes i en række forskellige input, som agenten får via dens sensorer.

Træet kan naturligvis udbygges efter behag og man kan herved øge kompleksiteten af agentens adfærd. Et af problemerne ved at anvende beslutningstræer til agentstyring, er at hele agentens adfærd skal fastprogrammeres fra start. Beslutningstræet skal indeholde alle tænkelige scenarier for agenten og tage højde for alle typer input – hvilket kan gøre det svært at programmere, hvis agentens adfærd ønskes meget kompleks. Netop dette er baggrunden for at anvende genetiske algoritmer til at optimere på disse træer, da man herved undgår at skulle fastprogrammere meget store beslutningstræer, men i stedet bruger en metaheuristik til at søge efter en hensigtsmæssig træstruktur. For at mindske søgerummet, er det valgt at fastlåse de spørgsmål der stilles, således at det 'kun' er grænseværdierne og handlingerne, der skal optimeres af de genetiske algoritmer.

Da det ikke fra start er oplagt hvilke spørgsmål der passer godt sammen, er det nødvendigt at stille alle spørgsmål ligegyldigt hvilken vej ned gennem træet der vælges. Konkret gøres dette ved at have et spørgsmål, som er det samme på hele niveauet i træet, således at alle spørgsmål med samme afstand til roden, tester for den samme egenskab. På Figur 8 ses et simpelt eksempel på et sådan beslutningstræ. I træet undersøges først om der er ammunition nok, dvs. mere end 50 %. Hvis der er det, undersøges det om der som minimum er lidt 'health' tilbage (mindst 30 %), hvorefter der angribes såfremt dette også er tilfældet. Er der mindre end 30 % health tilbage flygter agenten. Hvis der i den første test i stedet viste sig ikke at være så meget ammunition, undersøges det, om der til gengæld er rigeligt med 'health' (mindst 70 %), hvilket vil betyde, at agenten stadig vælger at angribe. Hvis der hverken er meget ammunition eller mindst 70 % 'health', så flygter agenten.



Figur 8: Beslutningstræ hvor der testes for samme egenskab på et givent niveau i træet.

Beslutningstræer ses ikke så ofte i spilsammenhæng, men man kan godt finde eksempler på at der er brugt *Learning Decision Trees* ifølge **[AIGAMEDEV2]**. For at benytte denne metode kræves det at man har eksempler, som man kan træne træet med, for det på den måde kan lære ud fra eksemplerne. Det lykkedes ikke at finde et godt eksempel på et spil, som anvender almindelige beslutningstræer, hvilket nok skyldes, at det i er mere oplagt at anvende en FSM når

FSM'en/beslutningstræet ikke skal oplæres. Det var heller ikke muligt at finde eksempler på beslutningstræer som indstilles med en genetisk algoritme.

3.3 Vægtede lineære funktioner

En anden tilgang til at løse agentstyringsopgaven, er at lave et system med vægtede inputs som vælger en handling ud fra inputtene og deres tilhørende vægte. Et sådan system er skitseret nedenfor i Tabel 1.

Handlinger \ Input:	i_1	i_2	i_3	…	i_n
1: Løb mod fjende, skyd mod fjende	$w_{11} \cdot i_1$	$w_{12} \cdot i_2$	$w_{13} \cdot i_3$	…	$w_{1n} \cdot i_n$
2: Gå mod ammo, skyd mod fjende	$w_{21} \cdot i_1$	$w_{22} \cdot i_2$	$w_{23} \cdot i_3$	…	$w_{2n} \cdot i_n$
3: Løb mod health kit, skyd ikke	$w_{31} \cdot i_1$	$w_{32} \cdot i_2$	$w_{33} \cdot i_3$	…	$w_{3n} \cdot i_n$
…	…	…	…	…	…
m: Handlingskombination m	$w_{m1} \cdot i_1$	$w_{m2} \cdot i_2$	$w_{m3} \cdot i_3$	…	$w_{mn} \cdot i_n$

Tabel 1: Viser en skitse af et vægtet inputsystem som vælger agenthandlinger.

Inputtene er stadig baseret på informationer om f.eks. ammo level, health level, afstande til modstandere og tilsvarende oplysninger. Ideen med systemet er, at agenten benytter de samme input som med beslutningstræet, men vælger handlinger ud fra hvordan vægtene i systemet er indstillet. Dette gøres ved at gange inputtenes værdier sammen med deres vægte og herefter summere hen over rækkerne en ad gangen. Ved derpå at undersøge hvilken række, der giver den højeste sum, kan der udtages en række, som svarer til den handling agenten skal foretage sig. På denne måde kan inputtenes relevans altså vægtes i forhold til at udføre en given handling. I forbindelse med et sådan vægtsystem ville der kunne optimeres på de vægte som ganges på inputtene med f.eks. en genetisk algoritme.[21]

For den interesseret læser findes et eksempel på hvordan systemet kan anvendes i forbindelse med skak i **[AI]** side 150 og frem. Det har ikke været muligt at finde eksempler på at teknikken har været brugt i forbindelse med FPS spil.

En vurdering af fordele og ulemper ved at benytte de tre forskellige agentstyringssystemer i sammenhæng med genetiske algoritmer gives i afsnit '5.1 Alternative løsningsovervejelser', hvor valget af beslutningstræer i kombination med genetiske algoritmer begrundes. Før denne argumentation gives, er det dog nødvendigt at kigge nøjere på teori om genetiske algoritmer.

[21] Inspireret af **[AI]** side 171-173.

Kapitel 4
Genetiske Algoritmer

I dette kapitel ses på hvad genetiske algoritmer er, og hvilke variationsmuligheder der er inden for emnet. I kapitlet gives en generel indførelse i teorien bag den heuristiske metode.

4 Genetiske algoritmer

I dette kapitel gennemgås principperne bag genetiske algoritmer, og der gives et overblik over fordele og ulemper ved at benytte algoritmerne. Der gives desuden en beskrivelse af nogle af de typiske generelle varianter indenfor genetiske algoritmer. I kapitel 5, vil teorien bag beslutningstræer blive kombineret med beslutningstræer og agenter i spil.

4.1 Generelt om genetiske algoritmer

Genetiske algoritmer er en metaheuristisk søgeteknik, som kan benyttes til at lede efter løsninger i søgerum, der er for store til at man kan undersøge alle løsningsmulighederne. Man hører ofte at teknikken blev kendt efter John Holland benyttede den i 1970, og senere udgav en bog om teknikken i 1975. Men faktisk blev teknikken opfundet tre uafhængige steder i 1960'erne af henholdsvis Edward Forel, Schwefel and Rechenberg og så altså John Holland[22].

Fordi genetiske algoritmer er en generel søgeteknik, kan den anvendes i mange forskellige sammenhænge og bruges til at lede efter løsninger på meget forskellige optimeringsproblemer. Et klassisk optimeringsproblem hvor genetiske algoritmer kan bruges er '*traveling salesman*'[23] problemet.

4.2 Teorien bag teknikken

Som nævnt er der utroligt mange variationsmuligheder inden for genetiske algoritmer. I dette afsnit gennemgås den fundamentale teorien for genetiske algoritmer og de klassiske variationsmuligheder, samt dem som har relevans for dette projekt.

Grundideen med algoritmerne er at efterligne den biologiske evolution. Det gøres ved at modellere den naturlige udvælgelse, samt krydsning og mutation af gener. Hver løsning til det problem der løses, repræsenteres så i et **kromosom**, der altså indeholder informationer om hvordan problemet kan løses. Algoritmerne fungerer så ved at udvælge nogle forskellige tilfældige startpunkter (nogle genetiske udgangspunkter for algoritmerne). Disse udgangspunkter gemmes som kromosomer, som algoritmen så forsøger at optimere på. Dvs. at i sammenhæng med *traveling salesman* problemet, ville kromosomerne være en eller anden form for rutebeskrivelse, eller en sorteret liste, hvor sorteringen angiver den rækkefølge som kunderne skal besøges i.

Da kromosomudgangspunktet til start vælges helt tilfældigt, vil den løsning som kromosomet repræsenterer i starten typisk ikke være en særlig god løsning. Ideen er herefter, at man benytter en fitnessfunktion til at bedømme hvor gode de forskellige kromosomer er. Hvis man f.eks. har et kromosom med de kunder den omrejsende sælger skal besøge, og rækkefølgen er helt tilfældig, er der meget lille sandsynlighed for, at dette giver den korteste rute. Men fitness funktionen kan nu benyttes til at måle på hvor gode de forskellige løsninger er, ved at beregne hvor lange ruter de forskellige kromosomer svarer til.

[22] Kilde: **[TS]**
[23] Klassisk optimerings problem hvor en sælger ønsker at finde den korteste rute til at besøge en række kunder som er lokaliseret forskellige steder.

Når fitness funktionen har vurderet alle kromosomerne, udvælges nogle af dem, med vægtet sandsynlighed, således at bedre løsninger har større sandsynlighed for at blive udvalgt end dårligere løsninger. De udvalgte løsninger krydses med hinanden til nye løsninger, i et forsøg på at få det bedste fra to verdener. I *traveling salesman* problemet håber man altså på, at man kan foretage en krydsning af to løsninger ved at sammensætte en god delløsning fra et kromosom med en god delløsning fra et andet kromosom, som til sammen herefter giver en bedre løsning end de to løsninger var hver for sig.

Efter krydsningen dannes små mutationer på de nye løsninger, som fremkom ved krydsningen. Der haves nu en række nye løsninger, som fitnessfunktionen kan anvendes på. Ideen er at gentage proceduren med udvælgelse baseret på en fitness måling, efterfulgt af krydsning og mutation, indtil der opnås et resultat, som opfylder de krav, der haves til løsningen.

Den generelle genetiske algoritme ser således ud, men findes i utroligt mange forskellige afskygninger.

Den generelle genetiske algoritme

```
P := InitialiseRandomPopulation()
while (f[getBestIndividual(P)] < minRequirement)
        for i := 1 to I
                f[i] := eval(i)
        P' := select(P, f)
        P' := crossover(P')
        P' := mutate(P')
        update(P, P')
return getIndividualFromPopulation(getBestIndividual(P))
```

I algoritmen laves først en population, *P*, som initialiseres med tilfældige individer. Herefter køres et while-loop, indtil der findes et individ i populationen *P*, der opfylder minimumskravet, *minRequirement*. For hvert individ i populationen beregnes en fitness værdi, som gemmes på pladsen med individets indeks i *f*-arrayet. Derefter foretages den vægtede udvælgelse, som udtager et antal individer fra populationen. Individerne krydses og muteres, for til sidst at genindtræde i populationen og erstatte nogle af de gamle individer. Desuden skal det nævnes, at *I* er antallet af individer i populationen, og *getBestIndividual* returnerer indeks for det foreløbigt bedste individ. Sidst returnerer algoritmen det bedste individ, som altså repræsenterer den bedste løsning, fra populationen.

Det skal her siges, at algoritmerne kan implementeres og tweakes efter flere strategier, alt efter hvilket problem man vil løse. Ofte er afvejningen om man vil målrette søgningen meget, med risiko for at man ender i et lokalt optimum, kontra om man vil lave søgningen bredere, men samtidigt også langsommere. Konkret er det ofte spørgsmålet om man vil gå meget målrettet efter at videreføre de bedste kromosomers gener, eller om man vil forsøge at få en adspredelse, der gør, at man kommer dybere ud i søgerummet. Det giver sig især udslag i udvælgelsen af kromosomer, der skal benyttes til den nye generation, men også i krydsningen og mutationen, kan søgning guides eller gøres bredere.

4.2.1 Fitness funktionen

Noget af det vigtigste i forbindelse med at benytte genetiske algoritmer er, at man kan finde en god fitness funktion, eller i det hele taget at kunne finde en fornuftig måde at evaluere løsningerne/individerne på. Uden dette falder grundlaget for at lave algoritmen til jorden, da man så ikke ved hvilke løsninger der er gode. I dette afsnit beskrives kort hvordan en fitness funktion typisk fungerer.

Fitness funktionen er meget problemafhængig. Funktionen skal være i stand til at evaluere hvor god en løsning er. I *traveling salesman* er det ikke så stort et problem at finde en fitness funktion, idet det er relativt oplagt at vælge længden af den samlede rute som fitnessværdi. Da der her er tale om et minimerings problem skal en kortere sti give en højere fitness værdi. I det følgende gennemgås hvorledes udvælgelsen sker på grundlag af fitness værdier.

Skulle man i stedet få en computer til at komponere et stykke musik eller male et maleri, ville det blive betydeligt sværere konstruere en fornuftig fitness funktion. For det er enormt svært at opstille regler for hvornår et kreativt stykke arbejde er godt, og der kan endda være meget delte meninger om det. Så selvom mange mennesker godt kan blive enige om at bestemte værker er stor kunst, kan det være utroligt svært at sætte fingeren på alle de ting, som netop gør det godt.

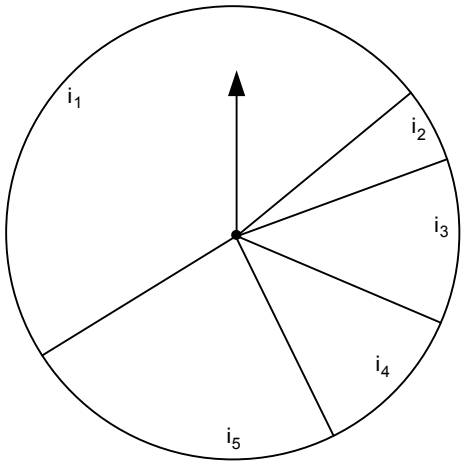
4.2.2 Udvælgelse

Når fitness funktionen har et mål på hvor godt hvert af individerne i populationen har klaret sig, foregår udvælgelsen så de bedste individers gener kan videreføres (*'suvival of the fittest'*). Der er mange forskellige metoder til at foretage denne udvælgelse. I dette afsnit gennemgås nogle af de mest populære af dem.

4.2.2.1 Proportional Selection[24]

I proportional selection vægtes hvert individ efter hvor stor en andel af populationens samlede mængde fitness point de har fået. Denne andel angiver derefter sandsynligheden for at et individ udvælges til krydsning. Man kan tænke på metoden som en roulette, der drejer rundt og lander et tilfældigt sted. Hvert individ modsvarer et felt på rouletten, som har størrelse efter hvor stor en andel af populationens samlede mængde fitness point individet har scoret. Når det første af to individer er valgt bør dette fjernes fra rouletten, så det ikke vælges igen, da et individ krydset med sig selv blot giver samme resultat som udgangspunktet.

[24] Kendes bl.a. fra **[SM]** afsnit 4.1.1.1



Figur 9: Viser proportional selection roulette.

I det følgende er opstillet pseudokode, som angiver hvordan man kunne designe denne form for udvælgelse.

Proportional Selection

```
Sum := 0
for i := 1 to I
        Sum := Sum + fitness(Individual[i])
r := randomFloat(0, Sum)
i2 := null
i1 := s := 0
while (r > s)
        i1 := i1 +1
        s := fitness(Individual[i1])
while i1 ≠ i2
        s :=  i2 := 0
        r := randomFloat(0, Sum)
        while (r > s)
                i2 := i2 +1
                s := s + fitness(Individual[i2])
```

I pseudokoden benyttes *Sum* til at udregne den samlede sum af fitness værdier for hele populationen. *i*, i_1 og i_2 er heltalsvariable som benyttes til at angive indekset for et individ, og *I* er antallet af individer i populationen. *r* er en decimaltalsvariabel, som benyttes til at angive et tilfældigt tal, og *s* angiver også en sum af fitness værdier.

Algoritmen fungerer ved at summen af alle de fitness point, der er givet findes i den første for-løkke lægges sammen i *sum*. Herefter vælges et tilfældigt tal, *r*, mellem 0 og den samlede sum. Dette tal svarer til viseren på roulettehjulet. Der køres herefter et while loop, som har til formål at undersøge hvilket individ viseren peger på. Dette gøres ved at tage fitness værdien fra det første individ og lægge den over i *s*. Se på om denne værdi er større eller mindre end *r*. Hvis *s* er større end *r* betyder det at viseren peger på individet. Hvis ikke lægges det næste individs fitness værdi sammen med værdien i *s*, og der undersøges igen om *s* er større end *r* hvilket i så fald betyder, at viseren pegede

på det andet individ. Sådan fortsættes til det første individ der skal udtages er fundet. Algoritmen vælger en ny værdi for *r* og gentager med endnu et while loop, som fungerer på samme måde og finder det andet individ. I denne version laves et tjek på om de to valgte individer er ens. I så fald gentages den sidste del af koden, som vælger det andet individ indtil der er valgt to forskellige individer.

Koden udvælger altså kun to individer til krydsning. Ønskes der flere individer skal koden kaldes gentagende gange indtil det ønskedes antal forældre par er fundet.

Proportional selection har desværre den ulempe, at den ofte får populationen til at konvergere på for tidligt et tidspunkt, hvilket kan bevirke, at populationen sidder fast i et lokalt ekstremum.[25] Dette hænger sammen med, at hvis ét individ har fået en høj fitness, så er der også stor sandsynlighed for, at dette individ vælges som forælder til mange af næste generations børn.

4.2.2.2 Linear Fitness Ranking (LFR)[26]

Med Linear Fitness Ranking benytter man ikke den rå fitness værdi direkte i udvælgelsesfunktionen. I stedet rangordnes alle individerne i populationen, således at det bedste individ i en population på N individer får en rang på N. Den næstbedste får en rang på N-1 osv. Det dårligste individ får dermed en rang på 1. På denne måde bliver det mindre sandsynligt at et enkelt individ vælges hver gang, da størrelsen på individernes sandsynlighedsandele kommer til at ligge forholdsvis tæt på hinanden. Det kan stadig lade sig gøre at vælge den bedste hver gang, men eftersom det kun vil være dobbelt så sandsynligt, at vælge det bedste individ frem for det gennemsnitlige, er det ikke så sandsynligt, at det bedste individ indgår i hovedparten af krydsninger, selv hvis der er tale om populationer med et enkelt meget dominerende individ.

Til denne form for udvælgelse gives ingen pseudokode, da udvælgelsen foregår på samme måde, som den vil gøre uden LFR, efter individerne er rangordnet. Selve rangordningen laves ved at sortere individerne efter deres fitness værdiers størrelse, hvilket kan gøres med en af de mange kendte sorterings algoritmer som f.eks. *heap sort*[27] eller en anden effektiv sorterings algoritme.

Tilsvarende LFR kan man også udføre en såkaldt *Sigma scaling*[28]. Denne form for skalering minder i sine egenskaber om LFR, men da den ikke benyttes i dette projekt, vil den ikke blive nærmere belyst her. Der kan læses mere om denne metode i kilden **[APG2]**.

4.2.2.3 Tournament selection[29]

Ved tournament selection udvælges to individer tilfældigt med uniform sandsynlighed. Individerne sammenlignes og individet med den bedste fitness udvælges til krydsning. Denne simple metode at foretage udvælgelsen på, har gode egenskaber i form af, at den dels forebygger en for tidlig konvergens, og samtidigt er let at parallelisere.[30]

---

[25] Kilde **[TS]**
[26] Kendes bl.a. fra **[TS]**
[27] Der kan læses mere om algoritmen i kilden **[IA]** på side 127
[28] Kilde: **[APG2]** side 651
[29] Fra bl.a. **[SM]** afsnit 4.1.1.1
[30] Fra kilde: **[TS]**

Tournament selection findes i flere forskellige versioner. I det følgende gives pseudokoden til én måde at opskrive algoritmen på.

Tournament Selection

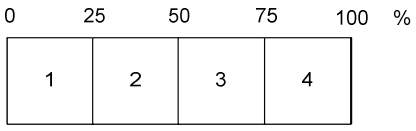
```
i1 := randomInt(1, I)
I2 := null
while(i1 ≠ i2)
        i2 := randomInt(1, I)
if (fitness(i1) > fitness(i2))
        return i1
else
        return i2
```

Her angiver i_1 og i_2 igen heltalsvariable, som benyttes til at angive numre på individer. Algoritmen starter med at vælge et tilfældigt heltal mellem 1 og antallet af individer, *I*. Tallet gemmes i i_1. Herefter vælges endnu et tilfældigt heltal mellem 1 og *I*, og dette gentages, indtil der er fundet et tal der er anderledes end i_1. i_1 og i_2 repræsenterer to individer, hvis fitness værdier sammenlignes, og det bedste individ returneres. Man kunne her også vælge at udtage to individer uden tilbagelægning. Det ville give samme resultatet.

I denne version af algoritmen vælges kun ét individ. Da man som regel ønsker at finde to forskellige individer til krydsning, kan den varieres til f.eks. at udtage fire individer uden tilbagelægning og sammenligne dem parvis, for på den måde at få udvalgt to forskellige individer.

Algoritmen ses også nogle gange i versioner hvor der udtages flere end to individer til den pulje hvorfra den bedste tages, men det mindsker muligheden for at mindre gode individer også vælges ind i mellem, hvilket kan være med til at gøre udvælgelsen for elitær.[31]

Tournament selection kan imidlertid også benyttes i forbindelse med udvælgelse af individer, som skal erstattes af den nye generations individer. Man vælger her på samme måde to individer, der sammenlignes hvorefter den dårligste af de to udvælges til at blive erstattet. Ved kun at udtage to individer til den turnering/pulje, der sammenlignes individer fra, får de bedste individer i populationen (den bedste halvdel) gennemsnitlig tre gange så stor sandsynlighed for at overleve, som de dårligste individer (den dårligste halvdel). Dette kan indses ved at tænke populationen som en uniform fordeling, som den vist på Figur 10.



Figur 10: Viser en uniformfordeling opdelt af skillelinjer ved de tre almindelige kvartiler.

I fordelingen er alle individerne lagt ind sorteret efter deres fitnessværdier. Tallene øverst angiver hvor mange procent af populationen som ligger til venstre for skillelinjerne. Det individ med middelfitnessværdien vil ligge ved 50 % skillelinjen. Derved vil det gennemsnitlige individ blandt

[31] Et eksempel herpå findes i kilden **[UCC]**

den dårligste og den bedste halvdel ligge ved henholdsvis 25 % og 75 % skillelinjerne. Da det gennemsnitlige individ fra den dårligste halvdel har bedre fitness end 25 % af fordelingen og dårligere end 75 %, vil det have ¼ sandsynlighed for at overleve en turnering. Det gennemsnitlige individ fra den bedste halvdel vil have bedre fitness end 75 % af populationen og dermed ¾ sandsynlighed for at overleve en turnering. Det ses altså, at det gennemsnitlige individ fra den bedste halvdel har tre gang så stor sandsynlighed for at overleve turneringen, som det gennemsnitlige fra den dårligste halvdel.

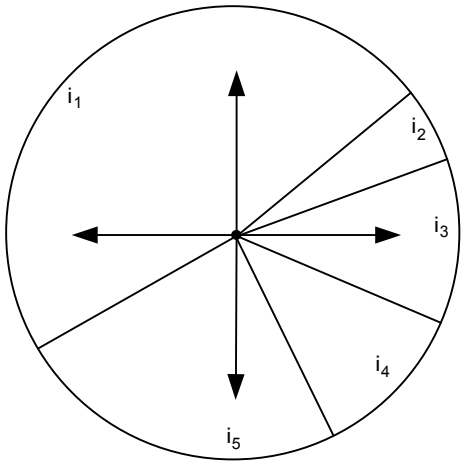
En anden egenskab ved tournament selection er, at den sikrer, at det bedste individ i populationen ikke ødelægges ved at blive muteret og krydset med andre individer, da et individ kun kan blive erstattet, såfremt der findes et andet individ i populationen som er endnu bedre. Ideen om at bevare den eller de bedste individer kaldes også ofte for elitisme[32].

Tournament selection egner sig godt til at udtage individer hvorfra man ikke på forhånd har fitness værdier for alle individerne i populationen og det samtidigt tager lang tid at evaluere hvert enkelt individ. Med tournament selection kan man nøjes med at evaluere to individer hver gang man skal udtage et individ, da man kan vælge to individer, som evalueres og sammenlignes uden at have evalueret andre individer i populationen.[33]

4.2.2.4 Stochastic Universal Sampling (SUS)[34]

I denne udvælgelsesmetode laver man en roulette ligesom i 'proportional selection', men til forskel for proportional selection, sætter man her flere visere på, og angiver dermed flere individer ad gangen. Viserne sættes på med ens afstand mellem hinanden. På Figur 11 ses den samme roulette som blev brugt i Figur 9 i forbindelse med proportional selection, men der er nu sat 4 visere på, og metoden udvælger derfor 4 individer til krydsning. Som det ses på figuren vælges individet i_1 to gange, men det er også det maksimale antal gange den kan udvælges i denne situation, og det er selvom den næsten har halvdelen af populationens samlede fitness points. Denne egenskab ved metoden hjælper med til at undgå for tidlig konvergens. Det kan stadigt godt lade sig gøre, at det bedste individ vælges af alle visere. Hvis det bedste individ skal vælges af alle visere skal det have en andel af den samlede fitness, der er større end $\frac{n-1}{n}$, hvor n angiver antallet af visere.

[32] Kilde: **[ER]**
[33] Kilde: **[AGP]** side 653.
[34] **[TS]**



Figur 11: Viser Stochastic Universal Sampling roulette med 8 visere.

Hvordan denne udtagelsesmodel kan opskrives som pseudokode, gives der i det følgende et eksempel på.

```
Stochastic Universal Sampling:

Sum := 0
for i := 1 to l
        Sum := Sum + fitness(Individual[i])
r := randomFloat(0, Sum/n)
hand := sh := s := l := 0
while(hand < n)
        s := s + fitness(Individual[i])
        while(sh < s)
                selection[hand] := i
                hand := hand + 1
                sh := s + (Sum/n)
        i := i + 1
return selection
```

Summen af populationens samlede fitness point gemmes i *Sum*, ligesom det gøres i proportional selection. *n* angiver antallet af visere på rouletten. Ideen er herefter at der med *n* visere er nødt til at være én viser i den første *n*'te del af rouletten. Dermed er det kun nødvendigt at finde ud af hvor i det første af *n* lige store intervaller, viseren peger. Når den første viser haves, vides det hvor de resterende visere er, da disse vil ligge forskudt med samme afstand mellem dem, fordelt over resten af rouletten. Der køres derfor to while loops. Det yderste while loop kører indtil alle visere er knyttet til et individ, og det inderste while loop sikrer, at der ikke gås videre til næste individ før alle de visere som skal knyttes til et individ er fundet.

4.2.2.5 LFR-SUS

Denne metode kombinerer Linear Fitness Ranking og Stochastic Universal Sampling. Det gøres ved, at man først tildeler individerne rang efter deres fitness værdier, som i LFR. Herefter laver man et roulettehjul, hvor andelene er lavet efter rangen. På den måde bliver det bedste individs andel af

rouletten altså ca. dobbelt så stor som det gennemsnitlige individs andel. Man sætter så et antal visere på, som svarer til det antal individer det ønskes at udtage. Når disse to teknikker kombineres, betyder det ofte at udvælgelsen af agenterne fremme adspredelse og ikke elitisme, da det bliver sværere at vælge et individ til som forælder til mange børn.[35]

Rangordningen foregår helt som i Linear Fitness Ranking afsnittet, og LFR-SUS fungerer fuldstændigt som Stochastic Universal Sampling når individerne først har fået fitness baseret på rangordningen. Af den grund er det valgt at undlade pseudokode her.

LFR-SUS er generelt særligt velegnet, hvis enkelte individer dominerer det samlede antal fitness point, på en måde, der vil lede til, at den næste generation af individer vil indeholder store dele af denne ene agents kromosom og dermed lede til for tidlig konvergens for algoritmen. Til gengæld er LFR-SUS svær at parallelisere, hvilket kan gøre den mindre egent i sammenhænge hvor dette er nødvendigt.[36]

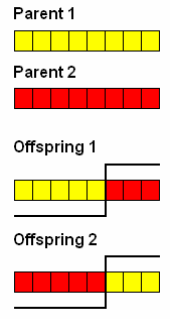
4.2.3 Krydsningen

Krydsningen er en central del i genetiske algoritmer. Der er mange forskellige metoder, som kan anvendes til at foretage en krydsning. I det følgende ses 4 forskellige eksempler på hvordan krydsning kan foregå. De første tre metoder benyttes til at krydse arrays. Den sidste benyttes i situationer, hvor man har viden om hvilke gener, der er beslægtede. I det pågældende afsnit eksemplificeret ved geometrisk placerede punkter.[37]

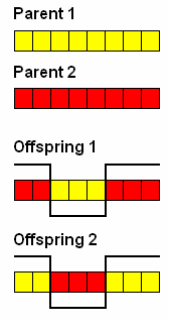
Idéen med krydsningen er at det ønskes at sammensætte to eksisterende løsninger til en ny. Herved håbes det at man får gode gener fra de to forskellige forældre som tilsammen giver en endnu bedre løsning. Man taler i denne forbindelse om *sexual cross-over* når to løsninger sammensættes til én.[38]

De tre metoder til at krydse arrays med er illustreret på figurerne nedenfor. Her angiver farven gul, at der er tale om en arrayværdi fra den ene forælder, og farven rød at det er en fra den anden forælder. Hver af metoderne beskrives i det følgende.

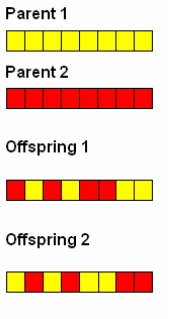
One point Cross-over:



Two point Cross-over



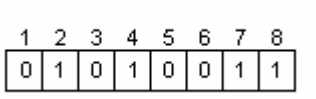
Uniform Cross-over:



[35] **[AGP2]** side 651
[36] Kilde: **[TS]**
[37] Kilde: **[SM]** side 101
[38] Kilde: **[AGD]**

4.2.4 Uniform Cross-over

Uniform Cross-over (også kaldet multipoint Cross-over) fungerer ved at der først dannes en bitmaske, som afgør, for hver enkelt position, om arrayværdien skal hentes fra forælder 1 eller 2. I dette eksempel ser masken ud, som vist i Figur 12 nedenfor. Til afkom 1 skal der benyttes forælder 2, der hvor der står 0, og der skal benyttes forælder 1 der hvor der står 1, og omvendt for afkom 2.



Figur 12: Denne maske angiver med 0 og 1 værdier, om der skal benyttes genet fra forælder 1 eller 2.

I det følgende gives et eksempel på hvordan pseudokoden for Uniform Cross-over kan se ud.

Uniform Cross-over

```
for i := 1 to I
        if(mask[i])
                Child1[i]  := Parent1[i]
                Child2[i]  := Parent2[i]
        else
                Child1[i]  := Parent2[i]
                Child2[i]  := Parent1[i]
```

i angiver indekset på den plads der kopieres fra i *Parent* arrayene samt til i *Child* arrayene. *I* angiver længden på arrayene.

4.2.5 One point Cross-over og two point cross-over

Disse to metoder er blot specialtilfælde af uniform krydsning. Alligevel beskrives de kort, da netop 'One point cross-over' er implementeret i forbindelse med dette projekt. I 'One point cross-over', krydses to arrays ved først at vælge en tilfældig position i arrayet. Når afkommet så dannes, vælges arrayværdier fra den ene forælder, frem til den tilfældigt valgte position. Herefter benyttes arrayværdierne fra den anden forælder. I dette tilfælde er det den sjette plads i arrayet, der er blevet valgt. Typisk laver man to børn, hvor det andet barn hele tiden får arrayværdier fra den anden forælder end det første barn får. Det svarer til at bitmasken i uniform cross-over består af en streng af enten kun 1-taller eller kun 0'er, efterfulgt af en streng af det omvendte (negerede).

I det følgende ses et eksempel på hvorledes pseudokoden til at lave denne krydsning ser ud.

One Point Cross-over

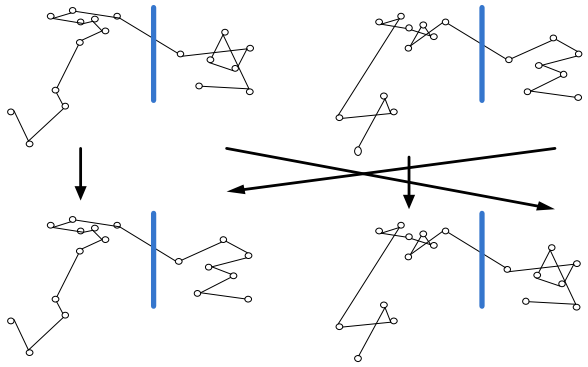
```
p := randomInt(1, l)
for i := 1 to p-1
        Child1[i]  := Parent1[i]
        Child2[i]  := Parent2[i]
for i := p to l
        Child1[i]  := Parent2[i]
        Child2[i]  := Parent1[i]
```

i angiver indekset på den plads der kopieres fra *Parent* arrayene samt til *Child* arrayene. *I* angiver længden på arrayene. *randomInt(a, b)* angiver en diskret uniform fordeling fra *a* til *b*.

Den eneste forskel mellem one point cross-over og two point cross-over er, at der to gange skiftes mellem hvilken forældre generne/decimaltallene hentes fra i two point cross-over. Bitmasken for two point cross-over består derfor af tre strenge, hvor den førstes og den sidstes celler har samme indhold.

4.2.5.1 Geometrisk krydsning

Den geometriske krydsning adskiller sig lidt fra de tre andre krydsninger, som er gennemgået her. Krydsningen kræver, at man har kendskab til hvilke gener, der er beslægtet med hinanden, og derfor kan inddeles i områder. Krydsningen foregår ved, at man vælger gener fra et geometrisk område fra den ene forælder, f.eks. højre side, og lægger generne i højre side hos barnet. Man tager dernæst gener fra et andet geometrisk område, f.eks. fra venstre side hos den anden forældre, og lægger disse gener i venstre side hos barnet, således at barnet i et område kun har gener fra den ene forælder, og i et andet område kun har fra den anden forælder.[39] Idéen er illustreret på Figur 13.



Figur 13: Viser en geometrisk krydsning af rutebeskrivelser for traveling salesman problemet.

I traveling salesman problemet kunne man forestille sig, at denne krydsning kunne have en fordel, da man her typisk vil have domæne specifik viden om hvilke gener, der geometrisk er placeret tæt på hinanden. Dette kræver, at man repræsenterer problemet på en måde så denne viden kan udnyttes i krydsningen. Denne problematik er det imidlertid valgt ikke at komme nærmere ind på her, da det ligger uden for denne rapports tilsigtede fokusområde, men Figur 13 illustrerer som sagt idéen. Her

[39] Kilde: **[TS]**

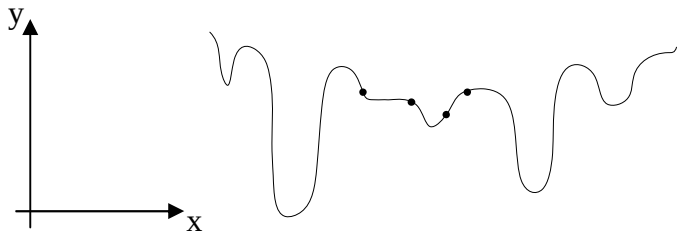
krydses to rutebeskrivelser, således at der opstår to nye hvoraf den venstre ser ud til at være bedre, end de to forældre hver for sig.

Da denne form for krydsning er meget problemspecifik, er det valgt ikke at give eksempel på pseudokode for krydsningen.

4.2.6 Mutation

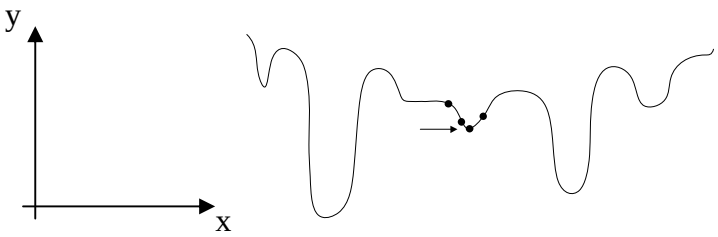
En anden væsentlig del af den generelle genetiske algoritme er mutation. Mutation benyttes for ikke blot at få en blanding af de løsninger man allerede kender, da der ikke er nogen garanti for at kombinationer af disse, indeholder den optimale løsning. Ved at introducere tilfældigheder på generne, mindsker man ved hjælp af mutation risikoen for, at man havner i et lokalt ekstremum, i stedet for at finde det globale ekstremum. I dette afsnit gennemgås nærmere hvordan mutationen kan afhjælpe sådanne problemer og hvordan mutation kan udføres.

Mutation udføres fordi det ønskes at introducere ny genmasse, dvs. nye egenskaber i genstrengene, som ikke er set i de foregående generationer[40]. Oprindeligt blev mutation introduceret, som en baggrunds faktor, hvor mutation kun blev foretaget med relativ lille sandsynlighed for hver gen, der kunne muteres, men forsøg har vist, at det ofte godt kan betale sig, at arbejde med større mutations rater, som så mindskes efterhånden, som evolutionen konvergerer.[41] På Figur 14 ses et eksempel på et løsningsrum, hvor fire tilfældige udgangspunkter for løsninger er indtegnet. Problemet er et minimeringsproblem, så det ønskes at finde det sted på kurven som har den mindste værdi.



Figur 14: løsningsrum for problem med 4 tilfældigt genererede løsninger

Hvis man udelukkende krydser disse løsninger med hinanden risikerer man, at havne i et lokalt minimum, som det ses på Figur 15. I dette eksempel benyttes der en krydsning som giver en x-værdi for løsningen, der ligger et sted mellem de to forælders x-værdier.



Figur 15: viser en algoritme som er konvergeret mod et lokalt minimum i stedet for det globale.

[40] Kilde: **[SEPAM]** side 2
[41] Kilde: **[ECC]**

Ved at benytte mutation kan man få indført ny arvemasse, som kan afhjælpe problemet, så man finder ned til det globale minimum. I det følgende gennemgås to forskellige metoder.

4.2.6.1 Tilfældig mutation

Denne form for mutation fungerer ved at tilfældigt udvalgte arrayværdier tillægges en vis mængde støj. Det kan enten vælges at danne en helt tilfældig værdi og lægge ind i arrayet i den celle som skal muteres, eller det kan vælges blot at ændre den eksisterende en lille smule. Hvis der er tale om et array med decimaltalværdier kan det f.eks. gøres ved at lægge et tal mellem -x og x til den eksisterende værdi, hvor x angiver den maksimal afvigelse fra udgangspunktet. I denne pseudokode for algoritmen ændres den eksisterende værdi lidt med funktionen *alter*. Koden er givet som følger:

Tilfældig mutation

```
for i := 1 to I
        r := randomFloat(0, 1)
        if(r < p_m)
                C_i := alter(C_i)
```

I er antallet af pladser i arrayet og *i* er indeks for en specifik celle i arrayet. *r* er en tilfældig værdi, der udregnes for hver celle i arrayet. p_m er den mutationsrate, som afgør, hvor tit der skal udføres mutationer. C_i er den celle i arrayet, som har indeks *i*. *randomFloat*(*a*, *b*) angiver en kontinuert uniform fordeling med endepunkterne *a* og *b*. *alter*(C_i) ændrer en lille smule på indholdet af cellen i *C* med indeks *i*.

Arrayet løbes igennem fra 1 til *I*. For hver celle i arrayet vælges et tilfældigt tal, *r*, fra en uniform fordeling mellem 0 og 1 ved hjælp af kommandoen *randomFloat*(0, 1). Hvis det valgte tal er mindre end mutationsraten (p_m), muteres der på værdien i arrayet.[42]

4.2.6.2 Ordnet mutation

Denne mutation foregår ved at to tilfældige celler i arrayet udvælges som endepunkter. Disse endepunkter angiver hvor der skal udføres mutation. Der laves mutationer på tilfældigt valgte celler, mellem disse to endepunkter. Pseudokoden for mutationsmetoden ser således ud:

Ordnet mutation

```
i_1 := randomInt(1, I)
i_2 := randomInt(1, I)
if(i_1>i_2)
        swap(i_1, i_2)
For all i := i_1,.., r_2
        r := randomFloat(0, 1)
        if(r<p_m)
                C_i := alter(C_i)
```

[42] Kilde: **[CI]** side 140

i_1 er det første endepunkt og i_2 er det andet endepunkt. r er en tilfældig værdi, der udregnes for hver af cellerne mellem de to endepunkter i strengen. *randomInt*(a, b) angiver en diskret uniform fordeling, med endepunkterne a og b. *randomFloat*(a, b) angiver en kontinuert uniform fordeling, med endepunkterne a og b. swap(a, b) bytter om på værdierne i a og b.

Først vælges to endepunkter fra en diskret uniform fordeling fra 1 til antallet af celler i arrayet vha. kommandoen *randomInt*(1, I). For hver af bittene mellem de to valgte endepunkter beregnes, som før, en tilfældig værdi fra en kontinuert uniform fordelingen mellem 0 og 1, og det afgøres vha. mutationsraten p_m, om cellen skal ændres[43]

4.2.6.3 Guidet mutation

Mutationen minder om *tilfældig mutation*, men hvor man i standard udgaven af *tilfældig mutation* muterer tallene til mere eller mindre tilfældige værdier, kigger man i *guidet mutation* på den bedste løsning fra forrige generation og muterer værdierne, så de kommer til at minde mere om de tilsvarende værdier hos det bedste individ.[44] Derfor kræver guidet mutation, at man har noget problemspecifik viden, der gør, at man ved hvordan man kan komme til at ligne den løsning man prøver at guide genet hen imod mere. F.eks. kan guidet mutation benyttes, hvis man har et array med decimaltalsværdier, som skal muteres. Hvis man antager, at der er en lineær sammenhæng mellem værdierne og den løsning de repræsenterer, således at værdier der er tæt på hinanden også er tæt på hinanden som løsning, så kan guidet mutation opskrives som følger.

Guidet mutation

```
for i := 1 to I
        r := randomFloat(0, 1)
        if(r<p_m)
                C_i := guide(C_i, best_i)
```

C er et decimaltalsarray. De celler i arrayet som skal muteres, sættes nu til gennemsnittet mellem den nuværende værdi, og værdien fra den celle med samme indeks, som findes hos det bedste individ fra forrige generations array. Præcist hvor meget man retter de værdier, der skal muteres kan naturligvis variere for de forskellige versioner af algoritmen.

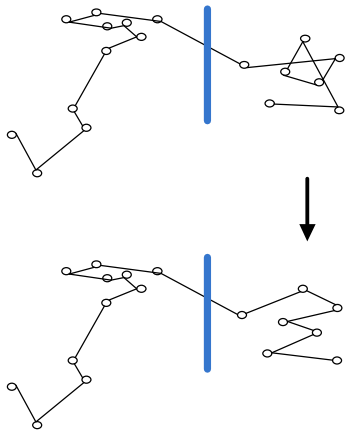
For at sætte et konkret eksempel på, kunne man forestille sig, at et gen der skulle muteres havde værdien 0.5, mens det tilsvarende gen hos den hidtil bedste løsning havde værdien 0.8. Hvis man så går ud fra, at det problem, der skal løses er sådan sammensat, at jo tættere værdierne er på hinanden, jo mere ligner løsningerne også hinanden, så kunne man guide genet fra værdien 0.5 til f.eks. 0.6, da den herved er tættere på 0.8. Hvis der ikke er en direkte sammenhæng mellem hvor store værdierne er og hvor tætte de tilsvarende løsninger er, så må man finde en anden måde at guide på.

4.2.6.4 Geometrisk mutation

Ved *geometrisk mutation* vælger man et område, inden for hvor der muteres i stedet for at vælge enkelte gener tilfældigt ud. For traveling salesman problemet kunne man vælge at ændre på

[43] Kilde: **[CI]** side 140
[44] Kilde: **[PB]**

rækkefølgen af punkter der ligger geometrisk tæt på hinanden. Et eksempel er givet på Figur 16 hvor der med fordel ville kunne muteres på rækkefølgen af de punkter som ligger til højre i for den blå linje på figuren.



Figur 16: Viser en geometrisk inddeling i to halvdele af rutebeskrivelser for traveling salesman problemet.

Ligesom ved *geometrisk krydsning* er det altså nødvendigt at vide, hvilke gener som ligger geometrisk tæt på hinanden i forbindelse med udvælgelsen af gener som skal muteres. Man kan endvidere kombinere geometrisk og guidet mutation, således at et helt område i kromosomet guides hen mod det bedste hidtidige kromosoms tilsvarende område.

Netop fordi denne form for mutation er problemspecifik vil pseudokoden ligesom ved geometrisk krydsning også være forskellig fra problem til problem.

4.3 Yderligere variationsmuligheder

Når man laver en genetisk algoritme, er der som tidligere nævnt utrolig mange måder at gøre dette på. I dette afsnit ses på to klassiske metoder til at variere strukturen af den samlede population.

4.3.1 Racer

Den første af metoderne er en opdeling af populationen i racer. Metoden benyttes som regel til optimering, hvor problemet har flere dimensioner. Det vil altså sige, når der skal optimeres på mere end én evne, således at fitness funktionen måler på mere end én ting. Man laver så typisk en 'rovdyr' - og en 'bytte' –race (bedre kendt under det engelske navn, prey and predator), som kæmper mod hinanden. Rovdyrracens individer måles på hvor gode de er til at fange individer fra bytteracen, og omvendt måles bytteracens individer på hvor gode de er til at undslippe rovdyrene. På denne måde kan optimeringen af de egenskaber, der optimeres på, specialiseres.[45] En feature er her, at hver race lære af hvordan den anden race udvikler sig, og kan derfor tilpasse sig denne strategi.

[45] Kilde: **[CC]**

4.3.2 Øer

En anden velkendt metode til at variere de genetiske algoritmer på, går ud på at opdele populationen i øer. På hver enkelt ø foregår der så en evolution. Fordelen er her, at ved at dele populationen op, kan man sikre, at et enkelt individ ikke kommer til at dominere hele populationen. Af og til kan man så lade øerne kæmpe mod hinanden, for at se hvem der er bedst. På den måde får man både fremavlet individer, der klarer sig godt mod de andre på deres egen ø, men samtidigt individer, som også klarer sig godt mod andre øer. Dette hjælper som regel til at undgå, at man sidder fast i et lokalt ekstremum, eller at man får for specialiserede løsninger. Derudover kan man lade beboere besøge hinanden, eller helt flytte fra ø til ø (visitation og immigration). Dette foregår typisk ved at enkelte gode individer, fra øer som klarer sig godt, besøger øer, som ikke er helt så godt med, for så at indgå i en krydsning der. På den måde er individerne med til at indføre ny arvemasse på den mindre gode ø. Individerne sendes herefter hjem igen i forbindelse med visitation, og ved migration lader man individerne blive boende, for at opnå en mere massiv og langsigtet påvirkning.[46]

[46] Kilde: **[CI]** side 141-144

Kapitel 5 Kombination af GA og beslutningstræer

I dette kapitel beskrives hvorledes teorien fra kapitlet om beslutningstræer og kapitlet om genetiske algoritmer sammenkobles til det agentsystem som er udviklet i forbindelse med dette projekt.

5 Kombination af GA og beslutningstræer

I dette kapitel beskrives hvorledes den genetiske algoritme og de konkrete beslutningstræer er konstrueret ud fra teorien fra henholdsvis Kapitel 4 om genetiske algoritmer og teorien fra afsnit 3.2 om beslutningstræer. Først sammenholdes de i Kapitel 3 præsenterede metoder dog, og fordele og ulemper ved systemerne i forhold til at benytte dem i kombination med genetiske algoritmer, diskuteres.

5.1 Alternative løsningsovervejelser

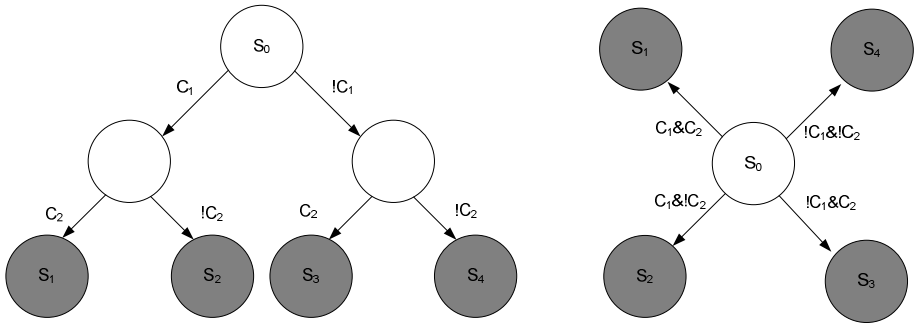
Under dette projekt er flere forskellige løsningsmodeller blevet overvejet. I dette afsnit ses på flere af de alternative løsningsmetoder som blev diskuteret, men som endte med ikke at indgå i dette projekts løsningsmodel.

5.1.1 FSM'er optimeret med metaheuristik

Et alternativ til den valgte løsning med beslutningstræer, der optimeres med GA, er at udskifte beslutningstræerne med FSM'er, der mere sædvanligt anvendes i spilsammenhæng. Der kunne i den forbindelse optimeres på FSM'erne ved brug af enten en genetisk algoritme eller en anden form for metaheuristik. I dette afsnit beskrives hvordan en sådan løsning kunne se ud, samt hvilke fordele og ulemper den har.

Umiddelbart virker FSM'er og beslutningstræer som to forholdsvis forskellige ting, men de har alligevel en række egenskaber, som gør dem ens på mange punkter. For det første gælder det, at et træ er en afgrænset type af grafer, som bl.a. er kendetegnet ved at være acyklisk, og kanterne i beslutningstræet kan siges at være orienterede, da det i forbindelse med valg af handlinger/beslutninger ikke giver mening at gå tilbage i træet til tidligere knuder. Der er altså kun en retning man kan gå i træet, nemlig fra rod til blad. FSM'er, der også tit ses repræsenteret som grafer, kan derimod sagtens have cykler. Kanterne i FSM'en er også orienterede, men til gengæld kan der gå kanter i begge retninger mellem to tilstande, hvilket i princippet giver det samme som en ikke orienteret kant (dog typisk med forskellige kriterier for at gå hver sin vej langs kanten). Alt i alt kan man derfor sige, at FSM'erne lader til lader være at være mere generelle i deres struktur end beslutningstræerne.

Det er desuden nogenlunde oplagt at et beslutningstræ kan simuleres af en FSM. For at gøre dette skal FSM'en have én starttilstand, som repræsenterer roden i træet samt én tilstand for hvert af bladene i beslutningstræet. For hvert blad i beslutningstræet, skal der så være en transition i FSM'en, som går fra starttilstanden til den tilstand, som repræsenterer bladet. Denne transition repræsenterer betingelsen, som opstår ved at konjugere alle betingelserne, der optræder i beslutningstræet på stien fra roden ned til bladet. Et eksempel er vist på Figur 17.



Figur 17: viser et simpelt beslutningstræ til venstre, og til højre en simulering af dette med en FSM

Ved at lade en FSM simulere et beslutningstræ fås dog en mindre overskuelig struktur i FSM'en end beslutningstræet havde, hvorfor der ikke ser ud til at være nogen fordel forbundet med det i forhold til dette projekt. Det er ikke lige så oplagt at holde styr på alle transitionerne (bestående af tilsvarende konjugerede kanter i træet) i en FSM struktur, som når de er opstillet i et beslutningstræ.

Omvendt virker det mere besværligt at skulle simulere en FSM med et beslutningstræ, blandt andet fordi FSM'erne kan have cykler. Samtidigt kan man sige, at FSM'en på sin vis har en form for hukommelsesmulighed indbygget, idet at den kan skifte mellem tilstande ved forskellige begivenheder, og ved at befinde sig i disse forskellige tilstande, reagere forskelligt på samme inputs. Denne egenskab er umiddelbart ikke så oplagt at simulere med et beslutningstræ. Eksempelvis kunne en FSM agent første gang der blev skudt (uden at agenten rammes) skifte til en tilstand hvor den var sur, og hvis det skete igen, kunne den skifte videre til en tilstand, hvor den skød igen. Hvis denne form for hukommelse skal simuleres i et beslutningstræ er det nødvendigt at beslutningstræet gemmer information om, at der er blevet skudt ved f.eks. ved at sætte et flag. Når beslutningstræet herefter køres igen, skal flagets tilstand gives til træet, som et af inputtene, således at det kan føre til et andet blad end første gang der blev skudt.

Igen er det ikke så oplagt at forsøge at tvinge et beslutningstræ til at simulere en FSM, hvilket betyder, at det nok vil være mere oplagt at vælge FSM'en i situationer, hvor det ønskes at give agenterne denne form for hukommelse.

I denne opgave er det valgt at arbejde med beslutningstræer som agentstyringsmodel, fordi det blev vurderet, at det ikke var nødvendigt med den form for indbygget hukommelse i agentstyrings-strukturen, som FSM'erne kan benyttes til, og fordi at beslutningstræerne giver en pæn og overskuelig struktur at arbejde med i forhold til de input agenten får.

Det kan dog ikke afvises, at man vil kunne lave en anden pæn struktur med FSM'er, som måske endda kan være bedre på nogle punkter, men præcist hvordan denne skulle se ud, er ikke oplagt. En mulighed der kort blev overvejet, var at have en FSM, som indeholdte en tilstand for hver mulig handlingskombination og så transitioner mellem samtlige tilstande. Med denne model er det i højere grad op til den genetisk algoritme at indstille FSM'en smart, da der ikke er så meget der er givet på forhånd. Denne idé blev droppet da alene antallet af tilstande ville blive utroligt højt, nemlig givet ved følgende formel.

$$n = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

Her angiver n antallet af tilstande og hvert M_i antallet af afhængige handlinger af en bestemt handlingstype. Handlingstyperne skal være indbyrdes uafhængige. F.eks. kan M_1 være måden agenten bevæger sig på, og M_2 den destination agenten bevæger sig hen til. Agenten vil både kunne vælge at gå eller løbe hen til sin destination, men kun en af delene ad gangen. Samtidigt kan agenten uafhængigt af måden den bevæger sig på, frit vælge en destination, men kun en destination ad gangen. M_3 kunne herefter være det våben agenten holder om osv. Agenten ender således med at have mange tilstande som desuden minder meget om hinanden, i og med at de bare er forskellige kombinationer af en begrænset mængde handlinger. Samtidigt vil antallet af transitioner mellem tilstandene blive (n-1)*n hvis der skal være transitioner fra enhver tilstand, som fører til enhver anden tilstand.

Alt i alt blev det vurderet, at beslutningstræet giver en fornuftigt struktur at arbejde med, da det er muligt at have relativt mange handlingskombinationer (i bladene) samtidigt med at agenterne hurtigt kan finde ud af hvilken handlingskombination de skal vælge, når spillet afvikles. I det følgende afsnit ses i på fordele og ulemper ved et system med vægtede lineære funktioner som også kaldes for vægtmatrix systemet i denne rapport.

5.1.2 Vægtede lineære funktioner optimeret med metaheuristik

Dette system ville ligesom de andre nævnte muligheder kunne indstilles af en genetisk algoritme, eller en anden metaheuristik, ved at ændre på vægtene, indtil der haves en fornuftig afvejning af inputtene for hver enkelt handlemulighed.

Systemet kæder inputtene sammen på en radikalt anderledes måde end de bliver i beslutningstræet. En forskel der er værd at bemærke ligger i, at inputtene i det vægtede system kan supplere – og træde i stedet for hinanden, hvilket kan være både en fordel og en ulempe, afhængigt af den pågældende situation.

For at illustrere et eksempel på et sted hvor det kunne være en fordel, kan man tænke sig en situation med en agent som kun har health og ammunition niveauer givet som inputs, og træffer en beslutning om at angribe såfremt den mindst har 150 i health og ammunition niveau lagt sammen. Med beslutningstræstrukturen ville man her være nødt til vælge f.eks. at lade agenten angribe hvis den både har over 75 i health level og 75 i ammunition level. Agenten kan altså lave en samlet afvejning med systemet af vægtede lineære funktioner så den også vælger at angribe i tilfælde af at den har 100 i health og 70 i ammunition, hvilket også kunne være fornuftigt.

Modsat kan det også være uhensigtsmæssigt at benytte et sådant system, såfremt inputtene er bedre tjent med at være uafhængige af hinanden. F.eks. er det ikke oplagt, at det at der er kort hen til et ammunitions kit kan opveje at agenten ikke har så meget i health. Dette gælder specielt, når mange af disse typer af vægtede input lægges sammen, da mange af dem kunne være fornuftige parvis. F.eks. ammunition og health eller det at der er mange fjender samlet på et sted kan opvejes af at der er kort afstand hen til egne holdkammerater, men det at finde en strategi ud fra en samlet vægt, når der er mange input, bliver mindre oplagt, vurderes det.

Et andet problem med denne agentstyringsform ligger i afhængigheden mellem vægtene i systemet. For man kan have et perfekt system af vægte som bliver ødelagt ved blot at ændre en vægt, da dette kan betyde, at man altid kommer til at vælge den samme handlingskombination. Denne overfølsomhed overfor ændringer i enkelte vægte ville formentlig kunne vanskeliggøre den

metaheuristiske søgning. Man kan forstille sig en situation hvor søgningen kunne være på vej mod en rigtig god løsning, men en lille mutation ødelægger systemet, da en dårlig kombinationen af handlinger så vælges for ofte – herefter bliver hele kromosomet skrottet, på trods af at 95 % af kromosomet var rigtigt godt.

Dette problem findes naturligvis også i beslutningstræer, men er her mest kritisk i tilfælde hvor de knuder som er placeret øverst i træet muteres til dårlige værdier, som betyder et fravalg af mange underliggende knuder. Dette vurderes dog at være et mindre kritisk problem, da sandsynligheden for at mutere en knude øverst i træet er lille, da der her kun er få knuder, hvor man i det vægtede system i princippet kan komme til ødelægge systemet ved at mutere en hvilken som helst knude, da dette kan gøre at en funktion vælges hver gang, som den med størst samlet sum.

Et vægtet system vil også blive stort, da antallet af funktioner, *n*, i systemet er givet ved antallet af handlingsmuligheder,

$$n = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

hvor M_i er defineret på samme måde som i afsnit 5.1.2. Selvom man opdeler systemet med valg af strategi først vil systemet blive større end beslutningstræerne hvis man, som i denne opgaves system, har mange handlingskombinationer. Man kan sige, at det afgøres af om antallet af funktioner givet ved formlen ovenfor bliver større eller mindre end 2 opløftet i antallet af input.

Det er naturligvis ikke muligt at vide på forhånd om vægtsystemet vil fungere lige så godt – eller måske endda bedre end beslutningstræer, men umiddelbart vurderes det, at ulemperne overskygger fordelene, når det sammenholdes med en løsning med beslutningstræer. Beslutningstræernes største ulempe er, at de hurtigt vokser sig store, hvis man f.eks. har 30 spørgsmål (input) man gerne vil bruge til at afgøre hvilke handlinger man skal udføre, får man et træ med 2^{30} knuder, som det ikke er svært at forestille sig, ville det, at optimere på et sådant træ være en stor opgave. Ved at dele spørgsmålene op, således at nogle af dem bliver stillet i et træ, der afgør en overordnet strategi og de andre i et træ, der repræsenterer den givne valgte strategi, kan dette problem til en vis grad afhjælpes.

Samlet set vurderes det, at beslutningstræerne på forhånd derfor den mest lovende struktur samtidig med at den er meget intuitiv at forstå, hvorfor det er valgt at benytte denne.

5.1.3 Alternativer til GA

Uanset om der arbejdes med optimering af en FSM, et beslutningstræ eller et system af vægtede funktioner, vil der kunne optimeres på dette ved hjælp af forskellige metaheuristikker. I dette afsnit sammenlignes genetiske algoritmer med alternative metaheuristikker og der argumenteres for valget af GA i dette projekts sammenhæng.

I forhold til at udskifte den genetiske algoritme med en anden metaheuristik er der flere andre muligheder man kunne overveje. To af de klassiske er *simulated annealing* og *tabu search*. Projektet kunne også have været gennemført med en af disse metoder i stedet for den genetiske algoritme, men den genetiske algoritme har en række fordele, som gør, at det er mest oplagt at anvende den frem for en anden metaheuristik netop til denne opgaves problemstilling.

For det første haves en stokastisk evalueringsfunktion da udfaldet af kampe mellem agenterne kan variere, for hvis man lader dem kæmpe flere gange uden at ændre deres gener kan udfaldet godt variere. Dette skyldes, bl.a. at det kan varieres præcis hvornår agenterne begynder at agere i forhold til gameticks'ne, men også at multitrådede programmer per definition er ikke-deterministiske. I den forbindelse giver det optimeringen en robusthed, at arbejde med en population, som hverken fås *simulated annealing* eller *tabu search*.[47]

En egenskab ved optimerings problemet der spiller ind, er det faktum, at der ikke er retningsinformation i søgerummet. Til denne slags problemer er genetiske algoritmer også at foretrække frem for metaheuristikker som anvender 'hill climber/gradient decent' tilgangen, hvor man prøver netop at udnytte retningsinformation til søgningen.

Foruden disse metoder kunne man også forestille sig at *neurale netværk* eller *learning decision trees* kunne have været anvendt. Problemet med disse to metoder er, at der kræves en række eksempler på god NPC adfærd, som systemet kan trænes med. Dette haves ikke, og hvis sådanne skal designes, mistes idéen med projektet; nemlig at det skal være muligt at have et så komplekst system, at programmøren ikke kan overskue hvornår agenten skal udføre hvilke handlinger og at systemet derfor på egen hånd skal 'opdage' de gode løsninger.

Rent optimeringsmæssigt er argumenterne for at vælge genetiske algoritmer også mange. For det første er søgerummet meget stort. Mulighederne for at kombinere de kontinuerte variabler, der repræsenterer grænseværdier i de enkelte knuder i træet, er store allerede ved forholdsvis små træer. F.eks. er der i et træ med ti niveauer $2^{10} - 1 = 1023$ kontinuerte variable, hvis værdier skal optimeres – og udover disse, skal der så tilknyttes de 'rigtige' handlinger til hvert blad i træet. Når søgerummet bliver så stort, er det naturligt at vælge en metaheuristik der er god til virkelig hårde problemer, og det er genetiske algoritmer.[48]

En anden grund til, at det er en god ide, at bruge genetiske algoritmer er, at optimeringsproblemet er et multiobjektiv problem, idet agenterne skal være gode til flere ting på én gang. For at virke overbevisende på den menneskelige spiller skal agenten, som minimum, både være god til at angribe modstandere, men samtidigt også god til at overleve. Der er naturligvis mange andre mulige formål, f.eks. kunne det være at samarbejde med andre agenter på samme hold, eller at hjælpe den menneskelige spiller m.fl. Ifølge **[TS]** er dette netop en af de ting genetiske algoritmer er gode til.

En tredje grund til at benytte genetiske algoritmer er, at problemet er af multikriteriums karakter – dvs. at der ikke kun er én opskrift på gode agenter, men derimod masser kombinationer af input og handlinger, der vil skabe gode agenter. Det er derfor på forhånd svært at lægge en strategi for hvordan agenterne skal designes, da det ikke er muligt at vide, hvilken strategi, der vil føre til den bedste type agenter. Ved at benytte genetiske algoritmer, er det meningen at undersøge en repræsentativ del af søgerummet og herved se hvilke kombinationer, der er gode og hvilke der er knap så gode.[49]

[47] Kilde: **[TS]**
[48] Fra **[TS]**
[49] Fra **[TS]**

5.2 Afvejning fri/styret AI

Da agenternes styringssystem skulle modelleres var der forskellige hensyn der indgik i overvejelserne. På den ene side var det ønskeligt at lave et system, der var så selvstændigt som muligt, forstået på den måde, at systemet helst skulle finde en adfærdsmodel med så lidt menneskelig indblanding som muligt, således at menneskelig indblanding ikke udelukkede rigtig gode løsninger, som måske ikke var oplagte for almindelig menneskelig intuition. På den anden side var der en begrænsning i forhold til tid og regnekraft, som gav anledning til at lave et mindre frit system, hvor søgning guides mere og som ikke ville tage helt så længe at oplære. Balancen mellem en målrettet søgning og et helt frit system er derfor mest et spørgsmål om, hvor meget tid der er til rådighed.

Afvejningen er blevet til et system, hvor bl.a. spørgsmålene i træerne er fastdefineret af os. Endvidere er det defineret, at hver niveau i træet har hver sit spørgsmål, og alle knuder på samme niveau i træet derfor spørger om samme spørgsmål. Dette er gjort for at sikre, at der kan spørges om alle spørgsmål, ligegyldigt hvilken sti ned gennem træet der tages. Træet er nemlig låst på den måde, at alle større end tegn vender samme vej i spørgsmålene. Dvs. at man altid går til højre i træet, når en værdi (input) f.eks. er større end den grænseværdi, der passer til spørgsmålet. Grunden til at spørgsmålene er låste til hvert niveau hænger sammen med, at det er ønsket at have mulighed for at stille alle spørgsmål, men samtidig ville det gerne undgås, at samme spørgsmål blev stillet flere gange, hvilket let kunne ske efter krydsningen af forskellige grene. Med en fast ramme for hvordan spørgsmålene stilles, sikres det altså, at når man krydser agenter, så krydser man indenfor samme domæne. Endvidere gør det, det også lettere at fortolke træet, når man det har en fastdefineret struktur, og derved bliver det lettere at designe det system der skal føde træet med input og reagere på dets beslutning.

Når den genetiske algoritme er færdig med at optimere på beslutningstræerne, haves dermed et hold af agenter, som styres af beslutningstræer, hvor strukturen, spørgsmålene og handlemulighederne er menneskeskabt/defineret, men hvor beslutningstræets grænseværdier og konkrete kombinationer af handlinger er indstillet af den genetiske algoritme. Hermed haves et system som er en mellemting mellem på den ene side at have et fuldstændigt fast defineret adfærdsmønster i form af et fastlåst beslutningstræ og på den anden side en agent, som er defineret ud fra en genetisk algoritmes optimering.

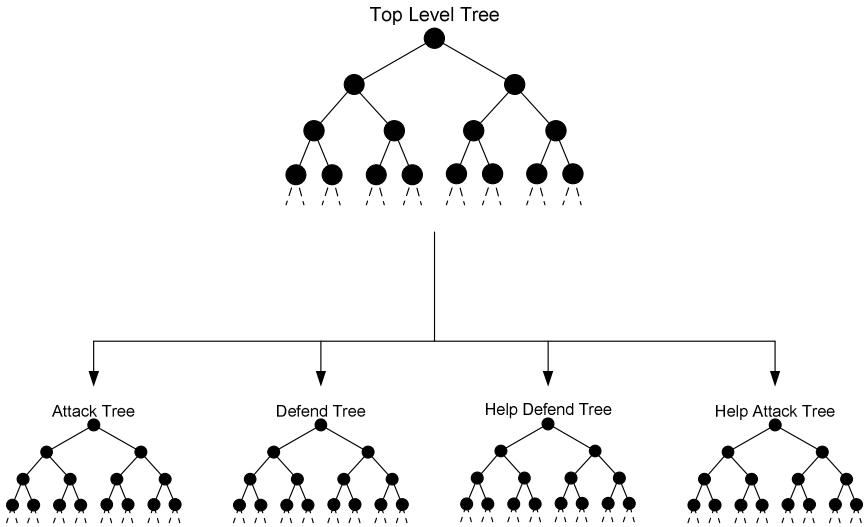
Da der som sagt er en væsentligt influerende regnekraft- og tidsfaktor at tage hensyn viste det sig nødvendigt at opdele beslutningstræerne i fem forskellige beslutningstræer. Beslutningstræerne blev dermed opstillet i en struktur, hvor ét træ tager sig af at vælge en overordnet strategi og hvert af de fire sidste træer står for en sådan strategi. Når agenten har valgt en strategi forsvinder den information, som findes i de input der danner grundlag for valg af strategi, og samtidigt er der forskel på hvilke input der gives til undertræerne. Dermed træffer agenten den endelige beslutning ud fra færre input, men det blev vurderet, at det var den mest fornuftige måde at gøre det på – både for at mindske de kombinatoriske problemer de helt store træer kan skabe, men også ud fra en antagelse om, at ikke alle input er nødvendige for alle beslutninger (strategier). Når strategierne er f.eks. er ’angrib’ og ’forsvar dig’, er det naturligt, at der i disse undertræer vil spørge om forskellige ting. Der findes eksempler på andre som også anvender denne form for hierarkiske struktur i forbindelse med beslutningstræer, men det har ikke været muligt at finde spilrelaterede eksempler herpå. I det følgende ses på hvorledes det samlede system af beslutningstræer er konstrueret, og hvorledes hvert af træerne fungerer.

5.3 Den overordnede træstruktur

I dette afsnit gennemgås den modellering, der er foretaget i henhold til at lave et system af beslutningstræer, som kan styre agenterne, når de skal begå sig i spillet. I de følgende fem afsnit gennemgås hvert enkelt træ mere udførligt.

Bl.a. fordi størrelsen af træerne stiger eksponentielt med antallet af spørgsmål (niveauer i træerne), er det, som nævnt, valgt at opdele agenternes adfærd i fem mindre træer, som tager sig af forskellige situationer. Det sker ud fra en antagelse om, at alle inputs til agenterne ikke er lige relevante i alle situationer. F.eks. vil det typisk være mere relevant at kigge på afstanden til nærmeste gemmested eller health kit, hvis man kun har 5 % health tilbage, end undersøge hvilken fjende der er mindst isoleret. Ud fra den idé er der valgt en struktur, hvor agenten først vælger en overordnet strategi, primært ved at kigge på dens egen sårbarhed og placering på banen i forhold til andre agenter samt de andre agenters strategier. Dette første træ, som vælger agentens overordnede strategi kaldes for *top level* træet, og har kun til formål at vælge den bedste af følgende fire strategier: 'angrib alene' (*Attack*), 'hjælp holdkammerat med at angribe' (*Help Attack*), 'forsvar sig selv' (*Defend*), eller 'hjælp holdkammerat med at forsvare sig' (*Help Defend*). Der er lavet et undertræ til hver af de fire strategier. *Attack træet* repræsenterer en angrebsstrategi, hvor agenten på egen hånd angriber fjenden uden at koordinere med eventuelle hold kammerater. *Defend træet* fungerer som modstykke til attack træet ved at det repræsenterer en strategi, hvor agenten på egen hånd vælger at forsvare sig og flygte. *Help Attack* strategien benyttes, når en agent ønsker at hjælpe en anden agent med at angribe og tilsvarende benyttes *Help Defend* strategien til at hjælpe en agent, som har besluttet sig for at flygte. Den mere nøjagtige adfærd kan derpå bestemmes i et af disse 4 træer ved hjælp af så relativt få inputs, at træernes størrelser holdes på et rimeligt niveau.

Man kunne naturligvis også forsøge at dele træerne op i endnu flere undertræer, men faren er naturligvis, at man kommer til at mangle nogle kombinationer af informationer, der kunne være gode til at træffe en beslutning. Samtidigt er det heller ikke helt oplagt hvordan et system med f.eks. træer i 4-5 niveauer skulle se ud. På Figur 18 ses en oversigt over sammenhængen mellem de fem træer.



Figur 18: Ufuldstændig skitse af træstrukturen hvor kun toppen af træerne ses. *Top level træet* tager 12 inputs og giver valg af undertræ som output. Undertræerne spørger herefter på yderligere inputs afhængigt af hvilket undertræ, der er valgt og giver agentens handlinger som output.

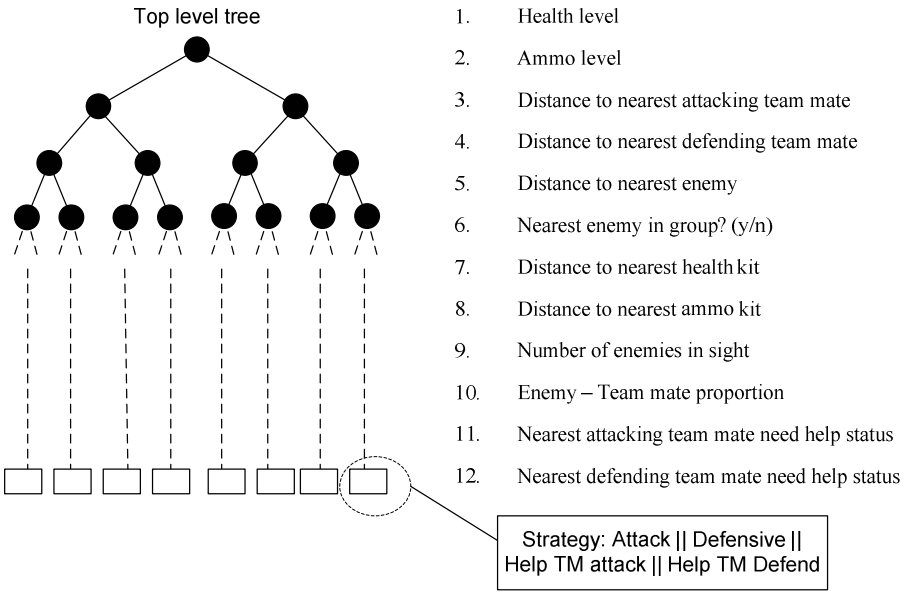
Hvert af de fire undertræer har varierende spørgsmål, som afhænger af den strategi de repræsenterer. Hvis en agent f.eks. har valgt *defend* strategien, antages det at være mere interessant at kigge på hvor det nærmeste health kit er, end afstanden mellem den nærmeste af agentens angribende hold kammerater og kammeratens mål. Dette er naturligvis en subjektiv vurdering som det på forhånd ikke kan vides med sikkerhed om er den bedste, men for de fleste af inputtene er der alligevel et klart incitament til hvor de hører mest hjemme – det samme gælder for handlemulighederne.

Når agenten har svaret på de forskellige spørgsmål og er nået ned i bunden af undertræet, findes der et sæt af handlinger her. Dette sæt indeholder en destination, en ankomsts retning, en skydefunktion og en måde at bevæge sig på, samt valg af våben. Destinationshandlingen afgør om agenten skal bevæge sig hen til en destination og i givet fald hvilken. Ankomstretningen angiver hvilken vej agenten vælger at ankomme til destinationen, da det kan være meningsfyldt at snige sig ind på en modstander bagfra eller forsøge at omringe denne. Skydefunktionen afgør om agenten skal skyde efter noget og i givet fald hvad agenten skal skyde på. Bevægelseshandlingen afgør hvilken måde agenten skal bevæge sig på. Altså om agenten f.eks. skal løbe eller kravle hen til sin destination. Endelig styrer våbenhandlingen hvilket våben agenten skal bruge til at skyde med, såfremt den har besluttet sig for at åbne ild. Disse fem handlingstyper er ens for alle fire undertræer, men de handlinger, der kan vælges mellem afhænger af strategien. Dvs. at man f.eks. ikke kan vælge at gå hen til nærmeste fjende i *defend* træet, men sagtens kan i *attack* træet. I *defend* træet kan man til gengæld vælge at gå hen til nærmeste health kit, hvilket ikke er en valgmulighed i *attack* træet. Det skal her siges, at selvom der er implementeret mulighed for disse fem handlemuligheder i træernes struktur, så har det reelt ikke været muligt at nå at lave funktioner, der gør det muligt at vælge en bestemt ankomst retning, at vælge en bestemt måde at bevæge sig på og vælge mellem forskellige våben. Funktionerne er ren spilprogrammering og er derfor blev nedprioriteret frem for at designe de genetiske algoritmer. Funktionerne er naturligvis vigtige i forhold til at agenterne skal kunne lægge smarte strategier, men til gengæld forstyrre dette ikke billedet af hvordan de genetiske algoritmer skal optimere på agenterne. Der vil således umiddelbart kunne tilføjes disse funktioner i forbindelse med fremtidigt videre arbejde med systemet, uden at der skal ændres nævneværdigt i træstrukturen og den genetiske algoritme.

I de følgende afsnit kan der læses om hvilke konkrete inputs, der gives til hvert enkelt træ, og hvilke handlingsmuligheder hvert af træerne kan give som output.

5.4 'Top-level' træet

Det første træ som agenten beslutter sin adfærd ud fra er som sagt *top-level* træet. Dette træ består af et spørgsmål omkring hvor sårbar agenten er, dvs. hvor meget health og ammunition den har tilbage, samt spørgsmål omkring hvor agenten befinder sig i forhold til både fjender, holdkammerater og health- og ammunitions kits. Det spørges også til hvor meget holdkammerater har brug for hjælp. Det vurderes, at disse informationer er tilstrækkelige til at kunne afgøre hvad agenten overordnet bør gøre. Altså om agenten f.eks. er for sårbar til at angribe og derfor må flygte.



Figur 19: viser *top level* træet som vælger strategi. Til højre ses de 10 inputs som træet tager beslutningen ud fra.

På Figur 19 ses en model af *top level* træet med de 12 inputs specificeret i rækkefølge efter hvad niveau de ligger på i træet. I det følgende gennemgås hvordan inputtene beregnes.

De første to inputs kan aflæses direkte, da der her bare er tale om agentens health og ammunition level. Agenten har adgang til fælles information om hvilke holdkammerater, der følger hvilke strategier, samt hvilke fjender, der er observeret af holdkammeraterne og den selv. Afstanden til nærmeste angribende, og nærmest flygtende holdkammerat, samt nærmeste fjende kan derfor udregnes med en simpel afstandsformel efter agenten har spurgt hvor medspillere der flygter og angriber befinder sig, samt hvor fjenden er observeret. For at få en tallet som en relativ værdi/rate mellem 0 og 1, tages disse afstande i forhold til hvor stor banen er. Afstanden til health- og ammunition kits kan tilsvarende beregnes. Agenten kender deres faste positioner og skal derfor kun aflæse sin egen position for at kunne bestemme hvilke kits der ligger tættest på.

Nearest enemy in group inputtet beregnes ved at se på om en given fjende har andre medspillere i nærheden. Det gøres ved at måle den givne agents afstand til de andre kendte fjender og se hvor mange af dem, der ligger indenfor en vis radius.

Antallet af synlige fjender deles med det højeste mulige antal af fjender for at også at få denne værdi som et relativt tal/rate mellem 0 og 1. Desuden beregnes forholdet mellem de to hold som et relativt tal mellem 0 og 1, hvor 0 svarer til, at der kun er fjender tilbage og 1 svarer til at det kun er holdkammerater tilbage.

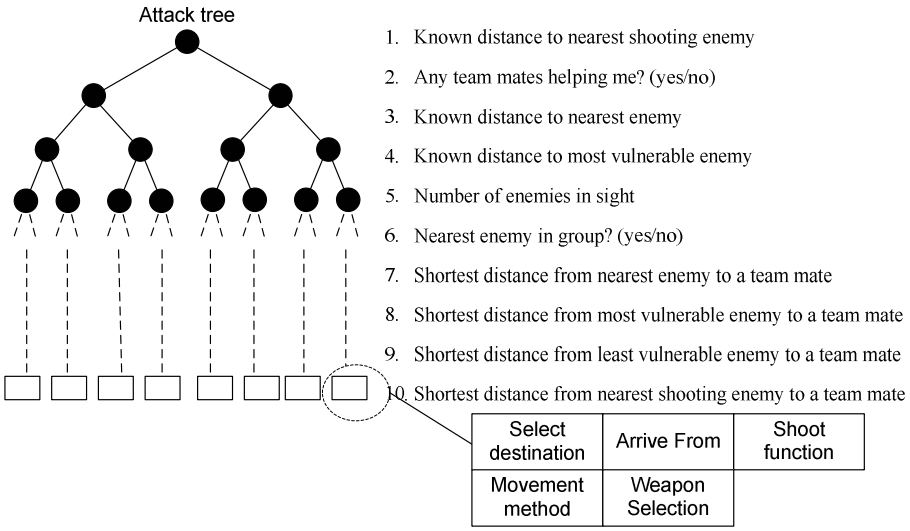
De to sidste input angiver mål for hvor stort et behov der er for hjælp hos de to nærmeste holdkammerater som henholdsvis angriber alene og forsvarer sig alene. Inputtene bliver bestemt ud fra om agenten skyder eller ej, og hvor meget health og ammo agenten har tilbage. Inputtet beregnes med funktionen *getHelpStatus*, der kigger på om agentens medspillere har brug for hjælp, ved at se på dennes health status, ammunitions status og skyde status.

Hvordan kromosomerne bliver lavet ud fra træerne bliver forklaret i afsnit ’5.9 Optimering af beslutningstræer, der styrer NPC’er’, men foreløbigt skal det nævnes, at der i dette træ bliver et kromosom med længde på 8191 variabler (2^{12} - 1) + (2^{12}), hvor 4095 af variablerne er grænseværdier (decimal tal) og 4096 af variablerne er pointere til strategier (heltal).

5.5 ’Angrib alene’ træet

I dette træ bestemmes hvilken fjende, der skal angribes, samt hvorledes dette angreb skal foregå. Beslutningen træffes ved at kigge på afstanden til de nærmeste og mest sårbare fjender, hvor den mest sårbare fjende i denne sammenhæng defineres som den fjende, der er mest isoleret fra sine medspillere.

En oversigtsliste over de inputs, som træet bruger til at beslutte hvilke handlinger agenten skal tage kan ses på Figur 20. I det følgende gennemgås de mere udførligt.



Figur 20: viser attack træet som afgør hvorledes agenten skal angribe fjenden. Til højre ses træets 6 input.

Afstand til den nærmeste og til den mest sårbare fjende beregnes på samme måde som i top level træet. Desuden beregnes afstanden til den nærmeste fjende som skyder. Denne fjende vil dog i praksis ofte være den samme som den nærmeste. Agenterne samarbejder om at bemærke om en fjende skyder eller ej, og antages at være skydende, når den har skudt for nyligt. Hvis en agent holder en pause med at skyde vil den dermed holde op med at fremstå som en skydende fjende.

Agenten har også mulighed for at få information om hvorvidt der er andre agenter, der har besluttet sig for at hjælpe agenten selv. Raten gives som antallet af agenter, der hjælper i forhold til hvor mange agenter, som er på holdet og dermed kunne have valgt at hjælpe.

Antallet af fjender der er i syne er det antal fjender som agenten selv kan se og dermed skyde på. Inputtet gives også som en rate hvor det samlede fjender der maksimalt kan være divideres op i antallet af dem der er i syne, for at få et tal mellem 0 og 1.

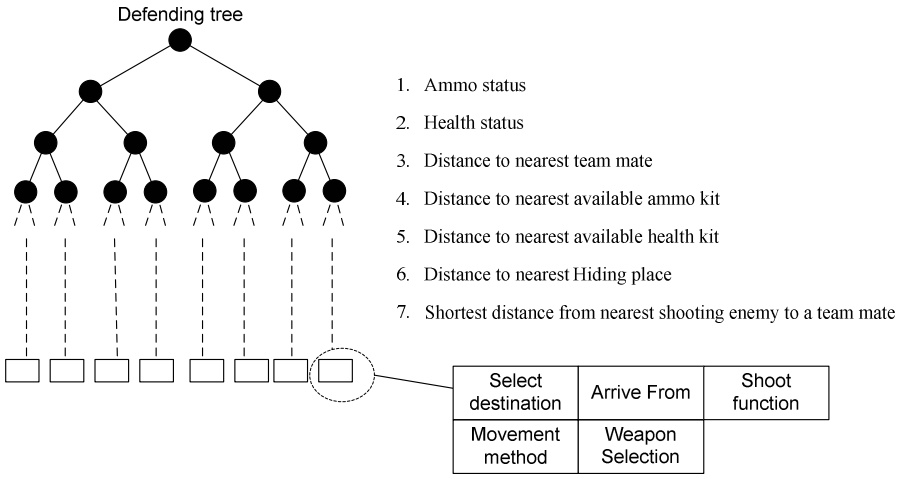
De fire sidste inputs angiver om der er risiko for at agenten kan komme til at ramme en holdkammerat ved at skyde på forskellige fjender, for på den måde at give agenten mulighed for at

vælge at skyde efter en anden fjende. Raten er et mål for hvor kort afstanden er mellem fjenden og holdkammeraten. Denne rate udregnes for alle fire undertræer for hver af de modstandere, der kan skydes efter. Idéen er så at træerne forhåbentligt bliver optimeret således, at agenterne ikke vælger at skyde efter en modstander, hvis risikoen for at ramme en medspiller er for stor.

I angrebstræet er de mulige handlinger begrænset til de handlinger, som findes relevante i forhold til at gennemføre et angreb. Blandt destinationsfunktionerne er bl.a.: gå til mest sårbare fjende, led efter fjender og gå til nærmeste fjende. Blandt skydefunktionerne er også den nærmeste og den mest sårbare fjende på som mulige mål, da disse modstandere er oplagte at skyde efter, når man i forvejen kan gå hen til dem.

5.6 Forsvars træet

I dette beslutningstræ bestemmes hvordan en agent på bedst mulig vis forsvarer sig selv i en given situation. Agenten kigger her på inputs, som vedrører afstanden til nærmeste health- og ammunitions kit. Derudover kigges på hvor tæt agenten er på et sted, hvor den kan gemme sig for fjenden og hvor langt væk den nærmeste fjende er.



Figur 21: Viser forsvarstræet som styrer agenternes forsvarsadfærd. De 6 forsvars input ses vil højre.

I forsvarstræet undersøges igen hvor meget health og ammunition agenten har tilbage ligesom i top level træet. Desuden beregnes afstanden til nærmeste hold kammerat, gemme sted, samt health og ammunition kits. Alle afstandene deles med banens størrelse for at få dem som rater.

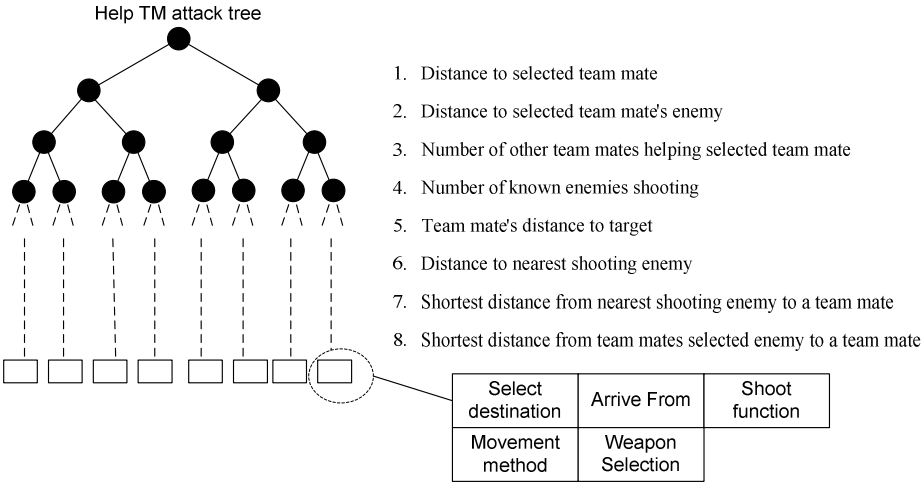
I forsvars træets handlingsmuligheder indgår især forskellige muligheder for at flygte til nærmeste health eller ammunitions kit, samt at gå til nærmeste holdkammerat eller gemmested. Skydefunktionen er begrænset til enten ikke at skyde eller kun at skyde på den nærmeste modstander som skyder, da funktionen anses for at skulle bruges mindre aggressivt i tilfælde hvor agenten flygter eller kun forsvarer sig.

5.7 'Hjælp holdkammerat med at angribe'–træet

I dette beslutningstræ ses på hvordan det kan lade sig gøre, at hjælpe en holdkammerat med at angribe en modstander. Der ses på hvor langt der er til den holdkammerat man vil hjælpe og hvor

langt der er til den fjende, som agenterne sammen skal angribe. Desuden ses på hvor langt der er mellem fjenden og holdkammeraten.

Igen ses den samlede liste af inputs som træet får på følgende Figur 22 med nummerering efter hvad niveau i træet de anvendes på.



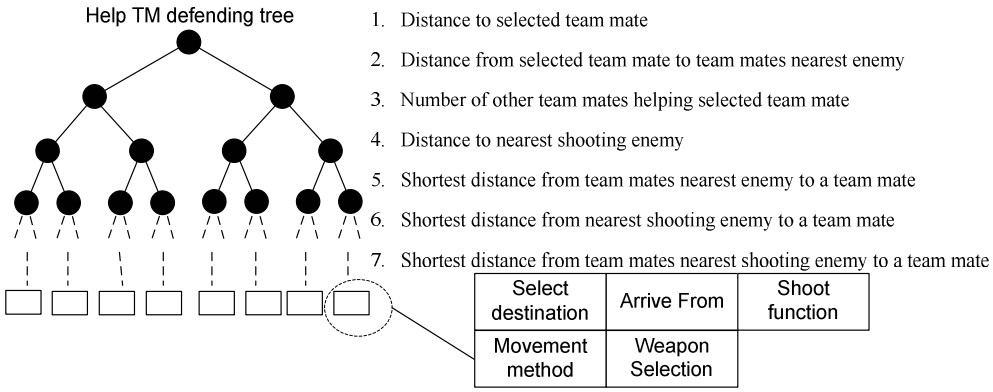
Figur 22: viser *Help Team Mate* træet. Træets inputs ses til højre.

Det ses at flere af inputtene igen er afstande. Disse afstande udregnes tilsvarende måde som det er tilfældet i de andre undertræer. Derudover ses også på antallet af fjender som skyder og på antallet af andre agenter, som har planlagt at hjælpe den samme agent. Begge disse antal deles også med antallet af henholdsvis fjender eller holdkammerater.

I *help attack* træet er handlingerne også tilpasset strategien. Det er dermed kun muligt at blive stående, bevæge sig hen til den holdkammerat man vil hjælpe eller hen til dennes valgte fjende. Endvidere kan man bl.a. vælge at skyde på samme fjende, som ens holdkammerat, eller vælge at skyde på den nærmeste skydende fjende.

5.8 'hjælp holdkammerat med at forsvare sig' –træet

I dette træ kigger agenten også på afstanden til den holdkammerat, som den ønsker at hjælpe. Derudover ses på om andre holdkammerater har valgt at hjælpe til, samt afstanden til den nærmeste fjende som skyder.



Figur 23: Viser *Help Team Mate defend* træet. Inputs er listet til højre.

Igen består inputtene hovedsageligt af afstande. Derudover indgår antallet af andre agenter, der hjælper til også som input.

Handlingerne til dette træ afspejler igen strategien. Agenten kan f.eks. vælge at blive stående eller gå hen til holdkammeraten. Samtidigt kan den primært vælge at skyde efter de fjender som udgør en trussel for holdkammeraten.

Det skal her siges, at det ikke er alle handlingsfunktioner der er beskrevet i ovenstående afsnit om træerne. En oversigt over samtlige inputs, som tages i betragtning, samt alle de mulige handlinger der er for hvert af træerne, kan ses i appendiks ’12.3 Input til-, handlinger fra- og begrænsning i beslutningstræerne’.

5.9 Optimering af beslutningstræer, der styrer NPC’er

I dette afsnit ses først på hvilke valg, der er truffet i forbindelse med optimeringen og af agentsystemet. Dernæst ses på hvordan teorien fra afsnittet om genetiske algoritmer og beslutningstræer anvendes i en kombination, hvor den genetiske algoritme optimerer på de beslutningstræer, som styrer agenterne.

5.9.1 De forudgående valg til algoritmen

I et projekt af denne type findes der utallige muligheder for at konstruere en genetisk algoritme og ideer til hvordan denne kan optimere et agentstyringssystem. Derfor har det ikke været muligt at afprøve alle ideer og muligheder for variationer af den genetiske algoritme. I det følgende gennemgås hvilke valg, der er truffet med hensyn til den genetiske algoritme. I kapitlet om parameter tuning vil parametre og variationsmuligheder af disse blive diskuteret.

Det er valgt, at designe systemet med racer, for at målrette den evolutionære træning af agenterne. På den måde gøres det muligt at benytte ’prey and predator’-ideen til at fremavle forskellige egenskaber hos agenterne. Valget synes oplagt, da agenterne har flere mål de gerne vil opfylde; de vil gerne både angribe modstanderen, men samtidig vil de også gerne overleve. I dette projekts sammenhæng betyder det, at angrebstræer skal træne mod forsvarstræer, for at kunne lære at slås mod deres *naturlige fjender*. Reelt betyder det, at agenterne i de første generationer fastlåses, sådan at predator agenterne kun kan vælge mellem angrebstræet og ’hjælp med at angribe’, og prey

agenterne kun kan vælge forsvars træet eller 'hjælp med at forsvare' træet. Når agenterne har nået et vist niveau, sættes de to agenttyper sammen, således at angrebstræet og 'hjælp med at angribe', kombineres med forsvarstræet og 'hjælp med at forsvare'. Herefter skal top-level træet naturligvis trænes, så den rigtige strategi bliver valgt i den rigtige situation.

Foruden brugen af racer, er det endvidere valgt at arbejde med ø-systemer, da dette giver mulighed for at dele populationen op i flere mindre underpopulationer. Dette har flere fordele. For det første er det en god måde at gøre plads til flere forskellige typer af agenter, hvilket er godt i et multikriteriums problem. Øer giver nemlig bedre mulighed for at få forskellige løsninger, da man undgår, at hele populationen bliver domineret af én agents struktur, fordi en given agents afkom primært vil holde sig til agentens egen ø. Samtidig giver det mulighed for at have en større population, hvilket ellers ikke er så let i forbindelse med sådanne systemer i FPS spil. Det er nemlig ikke hensigtsmæssigt at træne med flere end otte agenter på hvert hold, da agenterne gerne skulle lære at agere i en verden, der svarer til den, som der reelt er i sådanne typer af spil. Og da der i disse typer af spil sjældent er over otte agenter på hvert hold, er det den situation, som agenterne gerne skulle optimeres til. Ved at have øer, kan man have flere hold á otte agenter, og dermed en større population. I forbindelse med ø implementeringen er der også lavet ø-migration, ø-visitation samt ø turneringer.

Man kunne også lave en større population ved at lade holdene træne på skift uden at opdele med øer, men det ville give problemer når fitness pointene skulle sammenlignes, da et hold med 8 ens super agenter ville resultere i at agenterne fik ca. samme halv gode fitness, samtidigt med at en halv god agent på et hold sammen med 7 andre virkelig dårlige agenter kunne få en super god fitness, hvis de andre agenter kom til at ofre sig – eller hvis modstanderne bare var for dårlige. Ved at lade alle individerne indgå i samme pulje til udtagelse, ville den halvdårlige agent urimeligt tit blive udvalgt. Samtidig ville man heller ikke få den samme adspredelse mellem agenterne, som øer giver.

Allerede i forbindelse med disse forudgående valg til algoritmen, fås en lang række parametre som skal indstilles. Dette er også en af de større udfordringer i dette projekt, som vil blive belyst i kapitel '8 Parameter tuning og analyse af resultater'.

5.9.1.1 Træningens forløb

Agenterne træner altså en række generationer mod forskellige typer at modstandere. Præcis hvornår hvilke agenttyper træner mod hvilke andre, er parametre til algoritmen og gennemgås derfor først i afsnittet om parameter tuning. Dog opsummeres den overordnede idé i det følgende.

Agenterne starter med at blive opdelt i to hold – et der består af angrebs agenter, der har 'angrebs' træet og 'hjælp med at angrib' træet til rådighed, og et andet hold, der består at 'forsvar' træet og 'hjælp med at forsvare' træet. Endvidere deles den samlede population op i øer, således at hver ø består af otte angrebs agenter og otte forsvars agenter. De første *n* generationer træner de to hold mod agenter der blot står stille – altså hver hold hver for sig. Dette er for at give agenterne en chance for at lære positionerne på banen, uden at de får dårlig fitness fordi de bliver slagtet af bedre agenter. De næste generationer træner hver hold med agenter styret har FSM'er som beskrives i afsnit '5.10 FSM'. Efter dette begynder de to hold at træne mod hinanden – altså forsvars agent mod angrebs agent (prey mod predator). For at holde styr på om agenterne rent faktisk bliver bedre, får hver hold dog lov til at træne mod FSM'erne hver femte eller tiende generation (alt efter hvilken træning der tales om – mere om dette i parameter tunings afsnittet). Det er også efter at have trænet

mod disse FSM'er, at agenternes fitness værdier bliver sammenlignet med de tidligere generationers bedste fitness værdier, således at det hold som til sidst (når algoritmen terminerer) vælges som det der har klaret sig bedst, er blevet målt mod agenter af samme type som andre kandidater, nemlig mod fastdefinerede FSM styrede agenter.

5.9.2 Algoritmen

I dette afsnit gennemgås modelleringen af den genetiske algoritme. Der ses på hvordan genereringen af beslutningstræer foregår, og hvordan de optimeres.

Nedenfor ses pseudokoden for den genetiske algoritme, der er implementeret. Det skal siges, at der er implementeret flere forskellige slags krydsning og flere forskellige slags mutation – her er medtaget branch crossover og guidet mutation, da det fra start er de valg som virker mest lovende, ud fra, dels en forestilling om at træerne kan indeholde gode grene og områder som det er bedst ikke at klippe over i forbindelse med krydsning, og dels en forestilling, om at det kan være smart at forsøge at efterligne de agenter, som tidligere har været succesfulde i mutationen. Da det ikke er muligt at vide hvilken kombination af krydsning og mutation, der vil være den mest effektive testes alle kombinationer i kapitel 8.

Koden ligner den, der allerede er beskrevet i starten afsnit '4.2 Teorien bag teknikken', men i denne version er de konkrete metoder til udvælgelse, krydsning og mutation medtaget.

Den genetiske algoritme

```
P := InitialiseRandomPopulation()
while (f[getBestIndividual(P)] < minRequirement)
      for i := 1 to I
            f[i] := fitnessEvaluation(i)
      P' := LFRSUS(P, f)
      P' := branchCrossover (P')
      P' := guidetMutation(P')
      Update with tournament selection(P, P')
return getIndividualFromPopulation(getBestIndividual(P))
```

Det første algoritmen gør, er at initialisere en population for at have et udgangspunkt at arbejde ud fra. Dette gøres ved at oprette et antal øer bestående af agenter fra de to racer, der styres af hver deres beslutningstræer, og så indsætte tilfældige tal som grænseværdier i træerne. Der indsættes samtidigt tilfældige handlingspointere til funktioner, der kan kaldes som del af et handlingssæt. For grænseværdierne vælges der kun tal mellem 0 og 1, da alle input er normaliserede. Ligeledes indsættes kun pointere til funktioner, som er relevante for den type træ, som der er tale om. I forbindelse med angrebstræet, vil det sige funktioner som f.eks. *lookForEnemy*, *shootAtNearestEnemy*, *shootAtMostVulnerableEnemy* og lignende. Herefter evalueres agenterne, ved at lade dem spille mod hinanden.

5.9.2.1 Evaluering af agenterne og fitness funktionen

Nu haves altså en række tilfældige beslutningstræer, som styrer agenterne. Disse agenter vil typisk ikke være særligt intelligente, da indholdet i deres beslutningstræer er tilfældigt sammensat.

Evalueringen (tiden agenterne spiller) varer 90 sekunder og herefter undersøger algoritmens fitness funktion, hvor meget hver enkelt agent har fået i fitness. Agenterne får fitness point når de forårsager skade på agenter fra modsatte hold og omvendt trækkes der fitness point fra, hvis de selv bliver skadet eller hvis de kommer til at skade en medspiller. Fitnessfunktionen udregner så ud fra disse oplysninger en vægtet fitnessværdi. Vægtene bliver parametre til algoritmen – mere om dette senere. Funktionen ser altså således ud.

$$fitnessEvaluation(agent) = w_1 \cdot damagedone - w_2 \cdot healthlost - w_3 \cdot teamdamagedone,$$

hvor w_1, w_2, w_3, er vægtparametrene.

Fitnessfunktion kan naturligvis varieres meget, bl.a. efter hvilke egenskaber hos agenterne man ønsker at fremavle. Det er intuitivt oplagt at forsøge at belønne angrebsagenterne mere for at forsage skade på det andet hold, end man straffer dem for selv at være blevet beskadiget og omvendt for forsvarsagenterne. På den måde burde det blive lettere at fremavle mere aggressive egenskaber hos angrebsagenterne, som helst ikke skal være bange for at gå til angreb og omvendt fremavles evnen til ikke at miste health hos forsvars agenterne. Desuden skal det her nævnes, at vægtene i fitness funktionen naturligvis kun bruges til at udregne den fitness, der afgør hvilke agenter der bliver brugt som forældre til næste generation. Hver gang agenterne træner mod FSM'erne ses på om der er opnået bedre resultater end tidligere og dette gøres ved at kigge på den rene fitness som udregnes uden vægte ud fra følgende funktion.

$$pureFitnessEvaluation(agent) = damagedone - healthlost - teamdamagedone$$

Endvidere er der mulighed for at evaluere holdet af agenter på forskellige måder, når det gælder om at algoritmen skal finde den bedste endelige løsning. Enten kan man tage den bedste agent og sige at den repræsenterer det bedste fra holdet, og så kopiere denne agents adfærd, således at et hold med otte ens agenter fås. En anden strategi er at tage hele holdet og lægge deres fitnessværdier sammen, og på den måde se hvilket hold af agenter, der klarer sig bedst. Det er ikke oplagt hvilken strategi der er bedst, da det kan være at otte forskellige typer af agenter komplementere hinanden på en god måde og derfor samlet set er bedre end otte ens agenter, der dog har fået højere fitness end resten af deres hold. Derfor er der implementeret begge dele i forbindelse med dette projekt. Endvidere kunne man også forestille sig, at en kombination af disse strategier, hvor man f.eks. tog de fire bedste agenter og kopierede dem, således at et hold ville bestå af fire forskellige agenttyper med to af hver, kunne være fordelagtig. Mulighederne for forskellige kombinationer er mange, men da projektet kun har et vist omfang, er det valgt kun at implementere de to førstnævnte strategier.

5.9.2.2 Udvælgelsen

Når agenterne er færdige med at kæmpe og de har fået målt deres fitness, udvælges agenter til den næste generation – og dermed til krydsning og mutation. Idet populationen er forholdsvis beskeden, på hver ø, grundet det faktum, at spillet kun køres med otte agenter på hvert hold, er det valgt at lade udvælgelsen foregå ved hjælp af LFR-SUS. På denne måde undgås det, at gener fra ét, eller enkelte, meget stærke individer fuldstændigt dominerer den nye generation, da brugen af rangordningen i LFR-SUS giver bedre adspredelse i populationen og forebygger for tidlig konvergens[50], hvilket ellers kan være svært at undgå ved mindre populationer.

[50] Kilde: **[AGP2]** side 650.

Efter agenterne er valgt, krydses og muteres disse, og når der er genereret nye børn, benyttes tournament selection til at vælge de agenter fra den gamle generation, der nu skal udskiftes – på denne måde kan den bedste agent aldrig blive udskiftet (med mindre alle agenter udskiftes). Derved fås elitisme som beskrevet i afsnit '4.2.2.3 Tournament selection' hvilket også anbefales i en artikel om forbedring af genetiske algoritmer i spil sammenhæng.[51]

Samtidigt har gode agenter generelt større sandsynlighed for at overleve end dårlige agenter, når tournament selection anvendes. Desuden giver det er større adspredelse end bare at vælge de dårligste agenter og så udskifte dem. Sidstnævnte kan også være 'farligt', da søgningen for det første bliver meget målrettet og for det andet risikerer man at smide nogle gode gener væk, der blot har klaret sig mindre godt, fordi der et andet sted i kromosomet er nogle dårlige gener. Tournament selection og LFR-SUS er forklaret henholdsvis i afsnit '4.2.2.3 Tournament selection' og '4.2.2.5 LFR-SUS'.

5.9.2.3 Krydsningen

Der er implementeret to forskellige slags krydsning, nemlig one-point-crossover og geometrisk krydsning. One-point-crossover er modelleret således, at den som parameter tager, hvor stor en andel af kromosomet, der skal tages fra den ene forælder (og dermed også implicit hvor stor en del der skal tages fra den anden). Herefter tages den første andel af kromosomet fra den ene forælder og resten fra den anden, og så haves et nyt barn. Dette gøres to gange, således at 'barn nummer 1' får del 1 fra 'forælder 1' og del 2 fra 'forælder 2' og 'barn nummer 2' får del 2 fra 'forælder 1' og del 1 fra 'forælder 2'.

Den geometriske krydsning, også kaldt 'random branch', er modelleret således, at en parameter fortæller algoritmen hvor mange grene i træet, der skal vælges til krydsning. Selve krydsningen foregår ved at udvælge tilfældige grene ned gennem beslutningstræerne, som bruges til en del af det nye barn. Resten af generne kommer så fra den anden forælder. Der laves også her to børn på samme måde som beskrevet i forbindelse med one point crossover. Grunden til at valget faldt på geometrisk krydsning er, at man kunne forestille sig, at man let kunne komme til at ødelægge gode egenskaber hos agenten ved helt tilfældigt at vælge gener til krydsning, som det f.eks. gøres i uniform crossover. Håbet er at bevare en større del af de gode egenskaber ved, at der krydses hvor der både vælges de grænseværdier, der gjorde man nåede frem til specifikke handlinger, samt de handlinger der er blevet udført. Modelleringen er udført således, at der først vælges et tilfældigt niveau i træet, så en tilfældig knude i det valgte niveau, og derefter en tilfældig sti ned gennem træet. Det skal dog her siges, at da der tilfældigt vælges grene, kan starten af den valgte gren naturligvis blive valgt et uheldigt sted og der således alligevel vil blive ødelagt en god egenskab. Afvejningen er hvor meget man skal guide søgningen og dermed gøre søgningen hurtig og mere målrettet, men dermed heller ikke nå så vidt omkring i søgerummet.

5.9.2.4 Mutation

Mht. mutation er der også her implementeret to forskellige metoder. En tilfældig mutation og en guidet mutation. Førstnævnte vælger ud fra en mutationsrate, om et gen skal muteres eller ej. Hvis genet skal muteres, så bliver det muteret tilfældigt op eller ned – dog er der to parametre, der

[51] Kilde: **[AGP2]** side 649.

bestemmer hvor meget der maksimalt kan muteres med og hvor lidt der mindst skal muteres med. Den guidede mutation guider søgningen hen mod den bedste agent, ved at mutere værdierne på en måde, så de kommer tættere på den bedste agents tilsvarende værdier. I dette tilfælde er det den bedste agent på holdet i den givne generation. Hvor meget et udvalgt gen skal muteres hen mod det tilsvarende gen hos den bedste agent, styres af en parameter, der fortæller hvor stor en andel af forskellen på de to gener, der skal lægges til eller trækkes fra genet. Som nævnt benytter begge mutationsmetoder en mutationsrate, der bestemmer hvor mange gener der skal muteres på.

5.9.2.4.1 Faldende mutationsrate

Det er valgt at benytte en faldende mutationsrate funktion, ud fra idéen om, at jo længere henne i træningen vi er, jo tættere på en god løsning er vi og desto mindre skal generne muteres. Konkret ser mutationsrate funktionen således ud.[52]

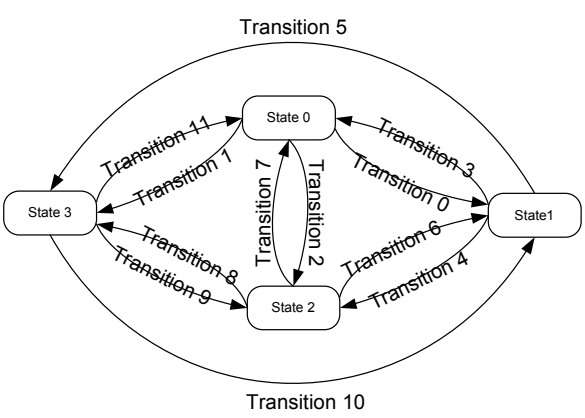
$$p_m(t) = \left(2 + \frac{l-2}{T-1}t\right)^{-1},$$

hvor t er generationsnummeret, T er samlet antal generationer og l er længden på kromosomet. Som det ses, er det afgørende hvor langt henne i træningen man er (generationsnummeret) og hvor stort kromosomet er, for hvor stor en andel af kromosomet man muterer. I det kommende afsnit beskrives det simple FSM system, som der benyttes til at træne og vurdere træningen af agenterne med.

5.10 FSM som sammenligningsgrundlag

FSM'en styrer en agenttype, der forsøger at overleve på bedst mulig vis. Den bruges som nævnt til at kickstarte den genetiske algoritme, men også til at give et fast holdepunkt i forbindelse med evalueringen af agenterne. De FSM styrede agenter prøver at gemme sig fra agenter på modsatte hold. Hvis agenterne mister for meget health eller ikke har så meget ammunition tilbage, prøver de at finde dette – agenterne vægter højest at finde health, hvis de mangler begge dele. Hvis de ikke har nogen fjender i syne og har både 'nok' health og ammunition, bliver de hvor de er. Modellen er vist nedenfor.

[52] Fra **[decMutRate]**, hvor den anbefales, hvis ikke man vil benytte en selv adapterende mutationsrate.



Transition 0: (H <= 60)
Transition 1: (E.I.) & (D)
Transition 2: (A < 5)
Transition 3: (E.I.) & (A > = 5) & (H > 60)
Transition 4: (H > 60) & (A < 5)
Transition 5: (!E.I.) & (H > 60)
Transition 6: (H <= 60)
Transition 7: (E.I.) & (A > = 5) & (H > 60)
Transition 8: (!E.I.) & (A >= 5)
Transition 9: (A < 5)
Transition 10: (H <= 60)
Transition 11: (E.I.) & (A > = 5) & (H > 60)

State 0: Look for hiding place
State 1: Look for health
State 2: Look for ammo
State 3: Hide

Notation
E.I. = Enemy insight, A = Ammo,
H = Health, ! = Not, & = And, D = destination reached

Figur 24: Model af FSM'en. Transistioner og tilstande er forklaret til højre.

FSM'en har fire forskellige tilstande den kan befinde sig i. Den første hedder 'Look for hiding place', og som navnet antyder, vil den i denne tilstand prøve at løbe hen til et sted, hvor den kan gemme sig. Hvis det lykkes at finde et uforstyrret gemmested skifter agenten til tilstanden 'Hide' og bliver herefter hvor den er, i et forsøg på at gemme sig. Foruden disse to tilstande, har agenten tilstandene ved navn 'Look for health' og 'Look for ammunition'. Når agenten er i 'Look for ammunition' tilstanden, prøver den at finde ammunition på kendte steder på banen, og på samme måde vil den i 'Look for health' tilstanden lede efter health kits på banen. I samtlige tilstande, skyder agenten efter modstandere, som den kan se.

Kapitel 6 Implementering af agentsystemet

I dette afsnit ses på hvorledes det er valgt at implementere den valgte løsningsmodel.

6 Implementering af agentsystemet

I dette kapitel forklares hvordan implementeringen er gennemført. Det gennemgås først hvordan den overordnede struktur ser ud, og herefter tages der et kig på hvordan hovedkomponenterne i implementeringen er konstrueret. Da selve implementeringen ikke er så central en del af opgaven, holdes dette afsnit relativt kort (i forhold til det reelle tidsforbrug der har været i forbindelse med dette), hvorfor der kun findes enkelte spilspecifikke programmerings-detaljer i dette kapitel. I stedet holdes abstraktionsniveauet generelt på et overordnet plan. Al kode, som er udviklet i forbindelse med dette projekt findes i appendiks '12.4 Kode'.

6.1 Torque & strukturen

I dette afsnit gives et overblik over den overordnede struktur for den kode, som har med agentstyringen at gøre. Der ses på hvordan de forskellige klasser kommunikerer indbyrdes, og der gives desuden et indblik i de vigtigste objekters sammensætning. En mere detaljeret gennemgang af hver af de centrale klasser findes tillige i hvert af deres respektive afsnit i det følgende.

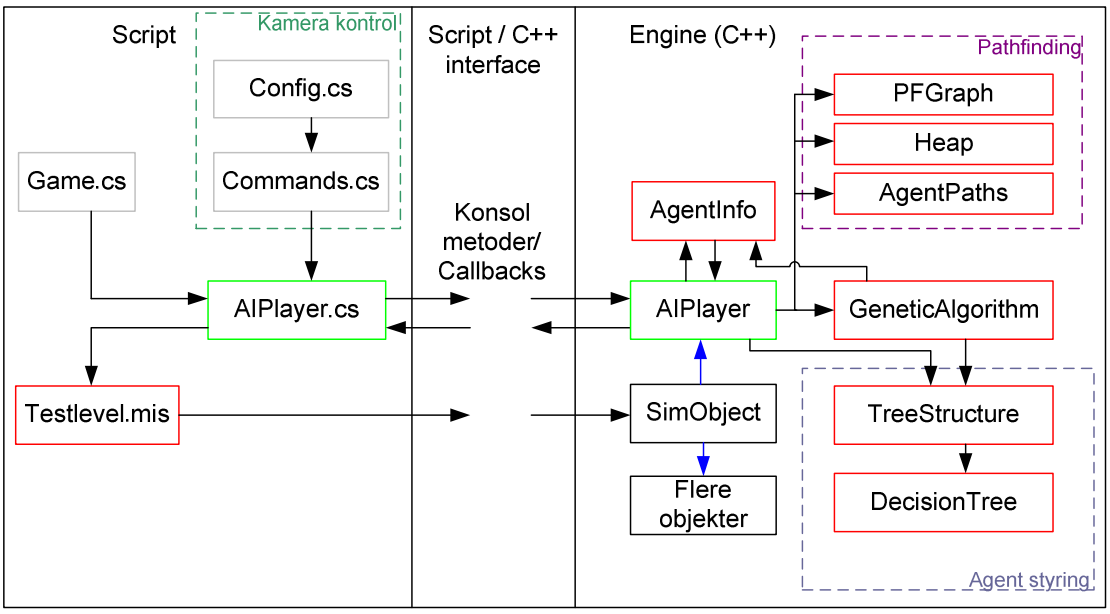
Som nævnt tidligere, er agentsystemet udviklet som en udvidelse til game enginen Torque, som er skrevet i C++. Torque implementerer dog også et scriptsprog med C++ lignende syntaks, som benyttes til at programmere bl.a. banespecifikke forhold. I scriptsproget kan der også lægges beregninger og styring af agenter, men da Torques scriptsprog ikke er så beregningsmæssigt effektivt, er det smartere at lægge de tunge beregninger over i C++ delen af enginen. Herunder har det i praksis vist sig, at være nødvendigt at lægge næsten al koden der, så spillet (og dermed evalueringen af agenterne) kan køre så hurtigt som muligt.

Torque er endvidere lavet således, at den bruger flere tråde til at holde styr på bl.a. grafik, fysik og memory manager. Endvidere er det sådan, at main tråden styrer **gameticks**, som sker hver 30 millisekund. Hver gametick sørger så for at opdatere hele spillets virtuelle verden – og i den forbindelse også de agenter, der styres af beslutningstræerne. På den måde styres alle agenter af samme tråd, hvilket giver en række fordele i og med, at agenternes fælles informationer (delte hukommelse) således ikke behøver trådsikring og kritiske sektioner, og da der i dette projekt blev arbejdet på maskiner som 'kun' havde en kerne, er det derfor valgt, at beholde strukturen med én samlet tråd til at styre agenterne for simpelhedens skyld.

Tilbage i scriptdelen ligger således kun forskellige initialiseringsprocesser samt banespecifikke objekter som f.eks. bygningsobjekter, healthkitobjekter, våbenobjekter, agentobjekter osv. Alle disse objekter er samlet i en missionsfil, der således indeholder alle de objekter, som findes på banen og dermed alle de objekter, der skal renderes af enginen.

Til at binde scriptkode og C++ kode sammen, er der fra start indbygget såkaldte 'konsol-metoder' i Torque, som er implementeret i C++ og herefter kan kaldes fra scriptet. Hertil er der i løbet af dette projekt tilføjet en række lignende konsol-metoder, der binder den C++ kode der er udviklet, sammen med den kode der er udviklet i scriptet. Metoderne implementeres i C++ og kan herefter kaldes fra scriptet. Desuden er der implementeret en metode til at eksekvere scriptfunktioner fra C++ koden – denne bruges ofte til at lave callbacks.

Nedenfor ses et diagram, der viser hvordan den del af enginen dette projekt har beskæftiget sig med, hænger sammen. Filerne der indeholder klasserne PFGraph, Heap, Agentpaths, AgentInfo, GeneticAlgorithm, TreeStructure, DecisionTree samt TestLevel er alle tilføjet enginen i forbindelse med dette projekt. De resterende klasser er en del af enginen som standard; dog skal det siges, at en del af filerne er revideret og udvidet væsentligt, herunder især AIPlayer filerne. Andre filer, som f.eks. Game.cs og Config.cs er kun ændret i lille grad, mens SimObject klassen slet ikke er ændret. På figuren er de klasser, der er implementeret i forbindelse med dette projekt røde, de klasser der er væsentligt modificeret og udvidet er grønne og de klasser der i lille grad er ændret er markeret med grå. Udover disse klasser er der implementeret header filer til filerne i C++ koden.



Figur 25: Viser de centrale klasser og deres relationer. Sorte pile viser hvilke objekter, der kalder funktioner i andre klasser og/eller laver instanser af klassen. De blå pile illustrerer hvilke klasser der arver egenskaber fra objektet. Endvidere er det markeret med grøn, lilla og grå-blå hvilke klasser der henholdsvis bruges til kamera kontrol, pathfinding og agentstyring.

Fra Game.cs initialiseres spillet og de forskellige scripts eksekveres herfra. Herunder også AIPlayer.cs, der har som hovedopgave at koble alle de nødvendige banespecifikke oplysninger sammen med AIPlayer klassen i C++ koden, således, at agenterne får mulighed for at få information om den verden de befinder sig i. Endvidere indeholder AIPlayer.cs en AIManager, der sørger for at koble AIPlayer objekter fra C++ sammen med de personroller, der renderes i spillet. Hver personrolle tilknyttes således et AIPlayer objekt, der bliver ansvarlig for at styre denne karakter.

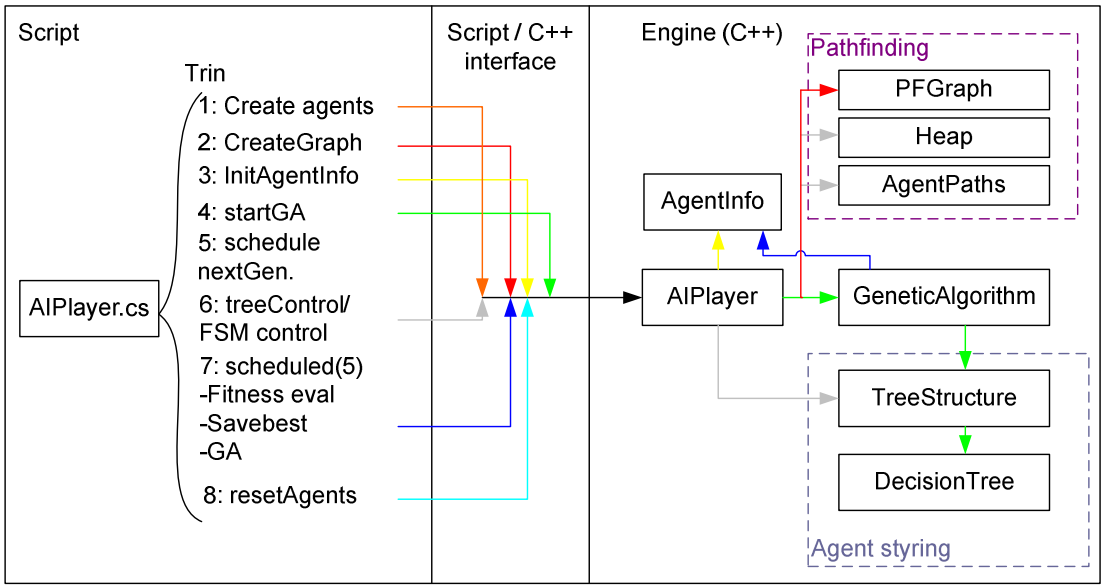
Fra scriptet (AIPlayer.cs) foretages en række kald til C++ funktioner, der her til formål at kopiere en række informationer over i C++ delen af enginen. Her er blandt andet tale om informationer i form af 'sim objekter', som hurtigere kan tilgås ved at være tilgængelige i C++ koden. Det er både agent objekter, de genstande som agenterne kan indsamle, men også en samling koordinater til diverse knudepunktspositioner på banen. Informationen sendes fra script filen *aiPlayer.cs* til C++ filen *aiPlayer.cc*. Det er bl.a. herfra, der ved hjælp af de overførte informationer, udregnes de inputs, som beslutningstræerne anvender i forbindelse med handlingsvalg.

Informationerne gemmes forskellige steder alt efter hvilken form for information det drejer sig om. Nogle informationer skal være fælles tilgængelige for alle agenter på det samme hold, mens andre er individuelle for hver af agenterne. For at agenterne kan dele information om f.eks. hvor fjenden sidst er set, og hvem der undersøger hvilke gemmesteder på banen, er der oprettet fælles statisk klasseobjekt kaldet *agentInfo*. Alle *AIPlayer* objekter har så en pointer til dette statiske objekt, og kan herved tilgå de gemte informationer. Kommunikationen med dette fællesobjekt sker, som nævnt, hovedsageligt via *aiPlayer*. Dette objekt er altså centralt i kommunikationen mellem agenter på samme hold. Endeligt gemmes information om agenternes fitness også i dette objekt – dvs. oplysninger om hvor meget health de har taget fra modstandere, hvor meget health de har mistet og hvor meget team damage de har lavet – denne information bliver videregivet til *GeneticAlgorithm* via et *AIPlayer* objekt.

Foruden *agentInfo* objektet har *aiPlayer* også adgang til træstrukturen og den genetiske algoritme, der er implementeret i henholdsvis *TreeStructure.cc* og *geneticAlgorithm.cc*. *TreeStructure.cc* benytter desuden filen *DecisionTree.cc*, som implementerer det generelle beslutningstræ. Ved hjælp heraf laver *TreeStructure.cc* alle fem beslutningstræer og kæder dem sammen. *TreeStructure.cc* fodres med inputs fra *aiPlayer.cc*, som passer til *top level* træet. Herefter tages en beslutning om hvilket undertræ, der skal benyttes i den givne situation. Når der er valgt et undertræ spørges der tilbage til *aiPlayer* for at få de sidste fornødne inputs, som er påkrævet i det valgte undertræ. Når undertræet har disse inputs, svarer det med handlinger, som skal udføres af agenten. Handlingerne sendes tilbage til *aiPlayer*, som sætter agenten i gang. Agenten får så nye handlinger, når det er agentens tur til at bruge et gametick på at beslutte et sæt af handlinger gennem TreeStructure.

6.2 Rækkefølgen af udførelsen af koden

Den genetiske algoritme startes også via et *AIPlayer* objekt. Dette gør nemlig at det kan startes fra konsollen i spillet efter det er startet. Den genetiske algoritme kommunikerer naturligvis med *TreeStructure* objekterne, da disse fungerer som agenternes kromosomer, og det er disse, der skal optimeres. Endvidere har klassen brug for at kunne læse *AgentInfo* objektet, da der her gemmes informationer, der bruges til at udregne fitnessværdier; dette sker som nævnt via et *AIPlayer* objekt.



Figur 26 Viser rækkefølgen koden eksekveres i, efter banen er startet.

På Figur 26 ses det netop beskrevne system. Figuren viser rækkefølgen af hvordan koden eksekveres. Når spillet (trin 1) startes kaldes *AIPlayer.cs* og denne kobler agenterne sammen med *AIPlayer* objekter fra C++ koden. Derefter, i trin 2, initialiseres grafen, der bruges i forbindelse med pathfinding på den givne bane. Herefter initialiseres alle de datastrukturer, der skal bruges til at holde styr på agentens viden, både den individuelle og den fælles viden som agenterne på samme hold deler (trin 3). Nu kan den genetiske algoritme startes, og det sker ved et kald til *startGA* funktionen (trin 4). Funktionen sørger for, at den genetiske algoritme initialiserer nye kromosomer (nye *TreeStructure* objekter) og sætter agenter koblet sammen med disse til at træne mod hinanden, indtil de skal evalueres (trin 5). Når denne periode er gået, stoppes agenterne (trin 6), den genetiske algoritme kalder evaluerings funktionen og sætter næste ø til at træne eller laver en ny generation af agenter (trin 7). Endelig resettes agenterne (trin 8); dvs. de bliver sat tilbage til deres startposition på banen, deres health og ammunition sættes til startværdierne og *AgentInfo* objektet nulstilles. Herefter går algoritmen til trin 5, og fortsætter til antallet af generationer er det som fra start er fastsat. I de følgende underafsnit gennemgås hver af de væsentligste klasser fra Figur 25 enkeltvis.

6.3 Script

I dette underafsnit ses kort på hvilke dele af opgaven, som er løst ved hjælp fra script sprog, og hvad baggrunden for disse valg er. I forbindelse med script koden ses der kun på *aiPlayer.cs*, da langt det meste af koden, som er relevant for agentstyringen, findes her. Af andre overordnede ting der er lavet i script kan nævnes kamera styring (så kameraet kan tilknyttes forskellige *AIPlayer* objekter i forskellige vinkler) og binding af taster til styring af agenterne.

I forbindelse med script delen af enginen ligner syntaksen, som C++, med undtagelser. F.eks. benyttes %-tegn foran lokale variable og $-tegn foran globale variable. Desuden er det ikke nødvendigt at deklarere variablene – alle variable kan indeholde typerne float, integer og string. Dette er også med til at gøre, at man eksplicit skal fortælle, hvis man f.eks. vil sammenligne strings. Dette gøres ved operatoren ’$=’. Konkatenering af strenge foregår ved brug af @-tegnet. Hvis man fra scriptet vil kalde funktioner i C++ koden, skal man implementere konsol metoder i C++ (mere om det senere), herefter kan de kaldes fra scriptet ved at man har et objekt fra C++ koden i scriptet (f.eks. et AIPlayer objekt) og så benytter ’.’ mellem objektnavn og metode navn – altså ligesom man ville gøre i C++.

6.3.1 aiPlayer.cs

Som tidligere nævnt er scriptsproget beregningsmæssigt langsomt, og det har derfor været hensigten at lægge så meget af koden i C++ delen som muligt. Når der alligevel er nogle ting der er lagt i scriptet skyldes dette, at det har været en nødvendighed at løse enkelte opgaver ved hjælp af scriptprogrammering. Et eksempel herpå findes i forbindelse med at man starter enginen og indlæser den bane man vil benytte. Her indlæses alle de agenter og genstande som er på banen i scriptet. Det var derfor nødvendigt i dette projekt at lave scriptfunktioner, som i forbindelse med at agenterne oprettes, laver kald til C++ kode og dermed registrerer de objekter, som skal kunne benyttes fra C++ koden. Denne operation sker kun en gang, hvilket gør tidsforbruget mindre kritisk.

Selve træningen startes også fra scriptet ved at kalde en funktion ved navn *startGA*. *startGA* funktionen starter evolutionen og kalder funktionen *nextGen*. *nextGen* sætter agenterne i gang med at slås med hinanden og sørger for at agenterne periodisk flyttes tilbage til deres udgangspositioner og får fuld health og ammunition, hver gang der startes en ny træningsrunde. Samtidigt nulstilles

informationer i fælles objektet 'agentInfo' også, således at agenterne ikke længere ved hvor fjenden sidst er set og hvilke gemmesteder, der er undersøgt.

Foruden den periodiske nulstilling af agenterne og objektkopieringen i starten er der enkelte informationer, som findes i scriptet hvor det er nødvendigt med et **callback** fra C++ koden til scriptet. Et sådan callback udføres bl.a. i forbindelse med, at der indsamles inputs til top level træet. Et af inputtene er en rate for hvor meget ammunition agenten har tilbage. Timingen for hvornår inputtet skal bruges, ligger i C++ koden, så derfor er det nødvendigt til at lave et callback i denne situation. Heldigvis er denne operation ikke særligt beregningskrævende, og udgør dermed ikke nogen nævneværdig del af det samlede tidsforbrug. Det kræver ingen ekstra tilføjelser i scriptkoden at man skal kunne kalde en funktion fra C++.

6.4 C++

I dette afsnit ses på den del af koden som er tilføjet i C++ delen af enginen. Der tages udgangspunkt i de klasser som findes i oversigten i Figur 25 hvorfra hver klasse gennemgås i individuelle afsnit.

Selvom Torque er implementeret i C++ er det ikke alle de funktioner man normalt kender fra C++ som kan anvendes. På en lang række punkter er det nødvendigt at benytte til C funktioner eller Torques egne funktioner. Dette skyldes at Torque enginen er lavet med henblik på at være platformuafhængig, hvilket betyder at man f.eks. ikke har STL (standard library) til rådighed, da denne konflikter med Torques memory manager, som ikke umiddelbart kan slås fra uden tab af funktionalitet.[53] Forskellen kommer bl.a. til udtryk ved at det ikke er muligt at anvende 'strings' i C++ delen af Torque, hvor man i stedet må arbejde med char arrays, ligesom det er nødvendigt at bruge funktionen 'dPrinft' til hvis det ønskes at skrive til terminalvinduet. Disse ændringer har besværliggjort arbejdet i C++ delen af Torque en smule.

6.4.1 AIPlayer.cc

aiPlayer.cc fungerer som hjertet i agentstyringssystemet. Det er herfra funktionen *getAIMove* ligger, og denne kaldes hver gang der forekommer et gametick, hvorefter spillets tilstand bliver opdateret. Dvs. at det er i *AIPlayer* objektet, at agenten skal kommunikere med resten af enginen for at reagere i spillets virtuelle verden (f.eks. at bevæge sig eller skyde mm.). Dette gør det også til et optimalt sted at finde ud af om agenten skal beslutte sig for at ændre taktik – dvs. gennemløbe træstrukturen og få nogle nye handlinger, som den skal udføre. For at spare på regnekraften (og fordi det ikke er nødvendigt at opdatere den enkelte agents taktiske valg hver gametick), er det implementeret således, at agenternes beregninger er distribueret ud på forskellige gameticks. I praksis er det gjort ved at f.eks. agent 1 får gametick nummer 1, agent 2 får nummer 2 osv. Ved hjælp af simpel modulo regning fås så et system hvor agenterne på skift får mulighed for at beslutte sig for en ny taktik. Endvidere indeholder klassen en række funktioner til at fortolke de handlingspointere, der returneres fra *TreeStructure*, når en agent køre ned gennem beslutningstræstrukturen, således at agenten rent faktisk begynder at udføre de handlinger, som den har besluttet sig for.

aiPlayer.cc er todelt idet den består af dels en række C++ funktioner som kan kaldes fra de andre funktioner der er implementeret i C++, og dels består af en række konsolmetoder, som kan kaldes

[53] Fra **[TorqueDoc3]**

fra scriptet. Blandt konsolmetoderne findes primært de metoder, som bruges til at kopiere objekter fra scriptet til C++ koden, og de metoder, som benyttes til at nulstille de forskellige agent- og holddata. Altså de funktioner, som kaldes af script funktionen *nextGen* hver gang agenterne starter en ny generation. Nedenfor er givet et eksempel på hvordan en konsolmetode kan se ud.

```
ConsoleMethod( AIPlayer, trackDamage, void, 4, 4, "(S32 agentNum, S32 damage)"
              "Used to keep track of the damage the agents do for fitness eval.")
{
       S32 agentNum = dAtoi(argv[2]);
       S32 damage = dAtoi(argv[3]);
       object->aInf->trackDamage(agentNum, damage);
}
```

Første argument, ”AIPlayer” angiver objekttypen, som kan kalde denne konsolmetode fra scriptet – i dette tilfælde, altså et AIPlayer objekt. Andet argument, ”trackDamage” er navnet på konsolmetoden og skal bruges til at kalde funktionen fra scriptet. Næste argument ”void” angiver hvilken type metoden skal returnere. De næste to argumenter ”4” og ”4” angiver hvor mange argumenter konsolmetoden mindst og højst skal kaldes med. Heri er medregnet objektet og metodenavnet – hvorfor disse tal altid mindst er 2. Når tallene er ”4” og ”4” betyder det så, at selve funktionskaldet fra scriptet mindst og højst skal tage 2 argumenter yderligere. De to næste strenge bruges blot til dokumentation til scriptere – således, at man kan få at vide hvilke typer variable, der forventes medsendt og en beskrivelse af hvad metoden gør. Argumenterne der er medsendt fra scriptet fås så fra argv arrayet og fortolkes i dette tilfælde som integers (S32 dvs. signed 32 bit) med funktionen dAtoi. Floats kan fortolkes lignende med dAtof. *trackDamage* benyttes til at holde styr på hvem der har skudt på hvem. Kaldet til funktionen fungerer ved hjælp af en pointer til *AIPlayer* objektet, *object*, som har en pointer videre til det statiske *AgentInfo* objekt, *aInf*, hvori *trackDamage* funktionen ligger.

Når der skal kaldes funktioner i scriptet fra C++ koden er det også i denne klasse det sker. F.eks. er der brug for, at den genetiske algoritme kan kommunikere til scriptet, hvilke agenttyper der skal kobles sammen med de grafiske objekter af agenterne. Dette sker via *AIPlayer* objektet, som vist nedenfor.

```
Con::executef(this, 4, "setOpponents", Con::getIntArg(opponentNum),
Con::getIntArg(treeTeamNum), Con::getIntArg(genNum));
```

Con er et statisk konsolobjekt, hvorfra der kan kaldes funktioner fra scriptet ved hjælp af funktionen executef. Det første argument funktionen tager, er det objekt der indeholder den funktion der kaldes – her *AIPlayer* i form af `this`. 4 angiver hvor mange argumenter scriptfunktionen tager. En scriptfunktion som er implementeret i et objekt tager altid mindst ét argument, nemlig objektet selv. Strengen ”setOpponents” angiver navnet på script funktionen, og herefter kommer de tre yderligere argumenter, som scriptfunktionen setOpponents tager.

6.4.1.1 Funktioner til styring af agenterne

Der er som sagt implementeret en masse funktioner til at styre agenterne og indsamle inputs til deres beslutningstræer. Her er tale om funktioner, der måler afstanden mellem agenter, undersøger hvilke agenter som kan se hinanden, samt hvor meget health agenterne har tilbage og den slags oplysninger. Endvidere omhandler denne kategori selvfølgelig også funktioner der implementerer de handlemuligheder agenterne har. Selvom disse har taget en del tid at lave, vil de ikke blive

beskrevet yderligere en i modelleringsafsnittene, da det ikke er her projektets fokus har ligget. Det skal dog siges, at langt de fleste af disse funktioner er placeret i AIPlayer.cc.

For at nedsætte tidsforbruget så meget som muligt er det valgt at lave et fælles system som afgør om der skal indsamles nye inputs til agenterne, eller om det foreløbigt kan springes over. F.eks. er det ikke nødvendigt, at se på hvor stor afstand der er til de forskellige fjender, hvor grupperet fjenden er, samt hvem af fjenderne der er nærmest, hvis der stadig ikke er nogen der kan *se* fjenden. Idéen er så, at i stedet for at bruge regnekraft på at udregne input o.a. når det ikke er nødvendigt, så kan evalueringen i stedet speedes op. Systemet som afgør hvornår agenterne skal indsamle inputs og trække svar ud af beslutningstræerne kaldes *eventChecker* og vil blive behandlet i det følgende.

6.4.1.2 *eventChecker* systemet

eventCheker systemet fungerer i to trin. Det første trin består i at der hver sekstende gametick skiftevis for de to hold udføres et fælles tjek for, om der er sket noget, der ændrer agenternes fælles situation. Såfremt det er tilfældet sættes et flag, som angiver, at der er sket noget som har indflydelse for alle agenterne på holdet. I det fælles tjek ses bl.a. på, om der er et nyt antal af fjendtlige agenter i syne og om nogen af agenternes holdkammerater er døde m.fl.

Det andet trin i *eventChecker* består af et individuelt tjek for hver agent, og dette er som tidligere beskrevet distribueret ud på forskellige gameticks, således at hver agent har et gametick til rådighed. Her ses på om agenten er nået frem til den position, som agenten var på vej hen til, og om der er sket ændringer i health– og ammunitionsniveauet. Hvis dette er tilfældet sættes et individuelt flag som angiver, at agenten skal revurdere sin situation.

Efter de to trin er udført laves et tjek på, om et af de to flag er sat, og hvis mindst et af dem er sat vil der herefter blive indsamlet nye inputs til agenten og trukket et sæt af handlinger ud af agentens beslutningstræ.

6.4.1.3 Pathfinding

Det er også i AIPlayer objektet den centrale del at pathfindingen finder sted. Det er nemlig her A* er implementeret, som benyttes til at finde korteste sti i pathfindings grafen, så agenterne ved hvilke punkter de skal gå mellem for at komme rundt på banen. Selve implementeringen af A* er genbrugt fra et tidligere projekt og vil derfor ikke blive beskrevet yderligere her. For den interesserede læser henvises i stedet til appendiks ’12.1 Pathfinding’ med start på side 101.

6.4.2 AgentInfo

AgentInfo anvendes til at dele information mellem agenterne, således at alle agenter på det samme hold er klar over hvad deres medspiller laver. Her gemmes bl.a. information om hvilke agenter der undersøger de forskellige gemmesteder, når holdet leder efter fjenden, og hvilen strategi hver af agenterne følger. Objektet indeholder desuden information om hvor fjenden befinder sig eller sidst er observeret. Der er ingen spektakulære implementeringsdetaljer i denne klasse, som hovedsageligt består af en masse pointere til datastrukturerer og funktioner til at tilgå samme.

6.4.3 GeneticAlgorithm

GeneticAlgorithm indeholder selve den genetiske algoritme og står dermed for træningen af agenterne. Klassen indeholder en række funktioner til at starte parametertuning, og den endelige træning af agenterne. Endvidere indeholder klassen funktioner til at foretage forskellige former for krydsning, mutation og udvælgelse. Hertil kommer en række funktioner, der holder styr på hvor langt i træningen algoritmen er kommet og funktioner der holder styr på øerne, visitationen migrationen og turneringerne. Endelig indeholder klassen funktioner til at udregne og gemme statistikker, agentkromosomer og parametre i eksterne filer, så disse kan tilgås senere. I det følgende gennemgås hovedfunktionerne fra *GeneticAlgorithm* og de klasser som tilsammen står for evolutionen.

6.4.3.1 Parametertuningsalgoritmen & den endelig test

Den genetiske algoritme og træningen af agentsystemet indeholder en masse forskellige parametre. Det er ikke oplagt hvilke kombinationer af disse parametre der er bedst til at optimere beslutningstræerne, hvorfor det er nødvendigt at undersøge det. Ideelt ville det være muligt at afprøve alle forskellige parameterkombinationer, men da dette reelt er umuligt pga. den lange evaluerings tid, er en heuristisk metode nødvendig. Mere om dette i parameter tunings kapitlet. Her skal det blot siges, at den algoritme, der så er valgt til at finde den bedste parameter kombination er en slags grådig algoritme, hvor den starter med en given værdi for hver parameter, og systematisk afprøver 3 forskellige værdier til hver parameter. Hvis den nye parameterindstilling er bedre end den gamle gemmes den nye værdi og sådan fortsætter algoritmen til den har afprøvet 3 værdier for hver parameter. Metoden er implementeret i funktionen greedyParameterTuner. Den endelige test benytter så denne bedste kombination af parametre for den specifikke genetiske algoritme, der testes. I alt afprøves fire kombinationer af genetiske algoritmer; nemlig kombinationerne mellem to slags mutation og to forskellige slags krydsning – men meget mere om dette i parameter tuningsafsnittet. Til at holde styr på de forskellige parametre er der lavet et objekt (defineret som en struct i header filen), hvor alle de forskellige parametre gemmes – på denne måde holdes de samlet og kan forholdsvis let udskiftes, da der laves en global instans af objektet.

6.4.3.2 Fitness, udvælgelse, krydsning og mutation

Hver gang en træning session, hvor et hold af agenter har spillet mod en modstander, er færdig, kaldes funktionen advanceTraining. Denne holder styr på hvor i træningen algoritmen er. Dvs. at den holder styr på hvilket hold på hvilken ø mod hvilken modstander, der netop har trænet og hvem der skal træne som de næste. Endvidere holder den styr på hvornår det er tid til at lave ø-vistation, ø-migration og ø-turneringer. Den holder også styr på, hvornår det er tid til at lave næste generation af agenter og når det er tilfældet, kaldes funktionen createNewGeneration, der står for at kalde fitness evalueringen, udvælgelsen, krydsningen, mutationen og endelig udskiftningen af agenter på det givne hold. Da der i dette projekt har været flere forskellige bud på hvordan konkrete implementeringer af udvælgelse, krydsning og mutation kunne anvendes smart, er det i implementeringen valgt at gøre det let at skifte mellem forskellige algoritmer til de forskellige områder. Derfor er genereringen af den nye generation implementeret med funktionspointere, således at man kan kalde f.eks. udvælgelses funktionen med en pointer til den præcise implementering af udvælgelse man vil benytte. Der er så lavet en definition (se nedenfor) af udvælgelses funktions pointere der siger, at funktionen skal returnere en pointer til en S32 (signed 32 bit integer) – dvs. et sted i hukommelsen, hvor der er gemt integers, der repræsenterer de

udvalgte agenter. Endvidere er det defineret, at funktionen skal tage tre argumenter, nemlig et array med fitness værdier, antallet af fitness værdier i arrayet og antallet af agenter der skal udvælges.

```
typedef S32* (Selector::*SelectorFunctionPtr)(S32*, S32, S32);
```

Hver gang der skal udvælges agenter ud fra deres fitness kan nedenstående funktion nu kaldes med en pointer til den udvælgelsesfunktion der skal benyttes.

```
S32* GeneticAlgorithm::select(S32* fitnessValues, S32 numToSelect,
          SelectorFunctionPtr selectorPtr){

          return (selector.*selectorPtr)(fitnessValues,
          parameters[teamNum][islandNum].population_size, numToSelect);
}
```

Et funktionskald til ovenstående funktion kan så f.eks. se således ud.

```
select(fitnessValues, numToSelect, &Selector::LFRSUS)
```

Samme princip gør sig gældende for implementeringen af metoderne til krydsning og mutation. På denne måde bliver det let at udskifte de forskellige algoritmer og det bliver også lettere at lave en automatisk test af de forskellige algoritmer, hvor de dynamisk udskiftes og afprøves.

6.4.3.2.1 FitnessEvalutaion

Selve beregningen af agenternes fitness funktion er implementeret fuldstændigt som beskrevet i afsnit 5.9.2.1. Udover denne beregning foregår der også andet i fitness funktionen. Bl.a. gemmes bedste fitness værdi til brug i guidet mutation. Hvis agenterne træner mod FSM'erne og samtidigt opnår bedre fitnessværdier end tidligere set, enten for en enkelt agent eller for holdet samlet set, gemmes agentens eller agenternes kromosomer også, så de efterfølgende kan hentes ind i spillet igen.

6.4.3.2.2 Selector

Selector klassen bruges til at udvælge de individer, som skal anvendes til krydsningen og de individer, som skal erstattes af nye individer. Der er implementeret en *LFR-SUS* til at udvælge individer til krydsning og en *Tournament selection* til at udvælge de individer som skal erstattes. I det følgende gennemgås hvorledes metoderne er implementeret.

6.4.3.2.2.1 LFR-SUS

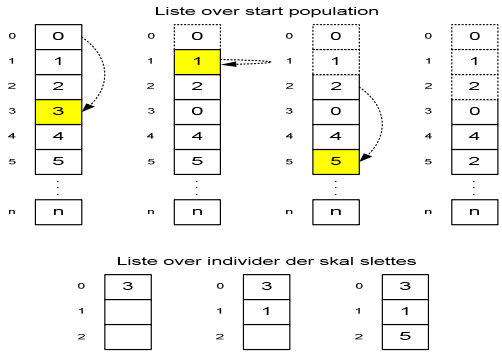
LFR-SUS-funktionen kaldes med en pointer til et array som indeholder individernes fitnesspoint, samt antallet af individer i populationen og det antal individer der skal udtages. Selve udtagelsen af individerne minder meget om pseudokoden fra afsnittet '4.2.2.4 Stochastic Universal Sampling (SUS)'. Det er derfor valgt ikke at gennemgå koden slavisk her. I stedet ses kort på de forskelle der er i forhold til pseudokoden. Den komplette kildekode findes i appendiks 12.4 såfremt det har interesse.

Implementeringen af *LFR-SUS* afviger fra pseudokoden ved at være udvidet med tilføjelsen af et kald til funktionen *randomizeOrder*. Funktionen sikrer, at de individer som udtages indgår tilfældigt i forældrepar, for parrene dannes efterfølgende ved at udtage dem fra listen sådan som de ligger placeret i denne.

Rangordningen i *LFR-SUS* er lavet med en naiv sorterings algoritme, da der i praksis aldrig arbejdes med populationer som indeholder mere end 8 individer, og derfor ikke er noget nævneværdigt at spare ved at anvende mere avancerede sorterings algoritmer. Koden kan naturligvis også læses i appendikset.

6.4.3.2.2.2 Tournament selection

Tournament selection er implementeret så den minder meget om pseudokoden i afsnit ’4.2.2.3 Tournament selection’, men udtager her det dårligste af to individer til sletning i stedet for det bedste af to til krydsning. Funktionen kaldes med det antal individer det ønskes at udtage samt med antallet af individer i population og en pointer til et array, som indeholder fitnessværdierne ligesom i *LFR-SUS*. Funktionen fungerer ved at to individer udtages fra en populations liste. Listen har fra start samme nøgleværdier som indekset. Det sikres, at det ikke er det samme individ, der vælges to gange. Det med den laveste fitness værdi kopieres over i en anden liste, som indeholder de individer der vælges til at blive erstattet af nye individer. Herefter overskrives nøglen til det individ, som skal slettes, med nøglen fra det første individ i populationslisten. Såfremt det var det første individ, der blev udvalgt ændrer denne operation ikke noget. Der ses efterfølgende kun på individerne mellem den anden og den sidste plads i populationslisten, begge inklusive. Udvælgelsen af to forskellige individer gentages, og det dårligste af dem kopieres til ’erstatningslisten’ og overskrives herefter med individet fra den anden plads i listen. Sådan fortsættes indtil det ønskede antal individer til erstatning er fundet, hvorefter erstatningslisten returneres. Se eventuelt Figur 27. Den faktiske kode findes i appendiks 12.4.



Figur 27: viser populations- og erstatningslisten, og hvordan individer der udtages til sletning fjernes fra populationslisten så de kun vælges til sletning én gang. Indekset er til venstre for cellerne og nøglerne i cellerne. Cellerne med stiplede linier kan ikke udtages til turneringen. Gule celler angiver individer der tabt turneringen. Læst fra venstre mod højre tæller hver søjle en iteration i algoritmen.

6.4.3.3 Krydsningen og mutationen

Der er som nævnt implementeret både to former for krydsning og to former for mutation. Fælles for krydsningerne er, at de tager to *TreeStructure* pointere som parametre (forældrene) og ud fra disse to forældre returneres så to andre *TreeStructure* objekter (børnene). Implementeringen af one-point-

crossover og random-branch-crossover følger ideerne fra afsnit 5.9.2.3 fuldstændigt og vil derfor ikke blive beskrevet yderligere her. Tilfældig og guidet mutation er begge implementeret så de både muterer på grænseværdier og på handlingspointerne. Forskellen på at mutere grænseværdier og handlingspointere er naturligvis blot, at førstnævnte er float værdier hvor sidstnævnte er integer værdier. I den tilfældige mutation medgives endvidere en øvre og en nedre grænse værdien af det nye muterede gen – således at der ikke kan muteres til ulovlige værdier. For handlingspointernes vedkommende betyder det, at der ikke må muteres til pointerværdier som ikke peger på en funktion. For guidet mutation er dette problem selvløst, da de gener man muterer hen mod i forvejen er lovlige.

6.4.3.4 Populationen

For at holde styr på hele populationen, som består af øer der er inddelt i hold, er det valgt at lave en TreeStructure*** variabel, der netop hedder population. På den måde kan man mappe den enkelte agent i populationen ved at kende hans hold, hvilken ø han er på og hans unikke nummer på øen, som vist nedenfor.

```
population[team][island][agentNumber]
```

Der er så implementeret en *getAgent* funktion, som kan kaldes fra et *AIPlayer* objekt, som returnerer det *TreeStructure* objekt, som passer til et hold og et agent nummer. På den måde kan beslutningstræet fortolkes i *AIPlayer* objektet. Ø nummeret findes i *GeneticAlgorithm* objektet, da det er her der holdes styr på hvilken ø, der er ved at blive trænet.

6.4.4 TreeStructure

I dette underafsnit ses på *TreeStructure*, som benyttes til at holde styr på beslutningstræerne og den struktur, som træerne er forbundet med. Samtidigt er det herfra agenternes træer kan gemmes i eksterne filer, så deres adfærd kan hentes frem igen på et senere tidspunkt.

TreeStructure's vigtigste funktion hedder *decideAction*. Funktionen benyttes, som navnet antyder, til at bestemme hvilken handling agenten skal tage. Funktionen kaldes med de inputs, som er indsamlet til *top level* træet, og bestemmer ud fra dem hvilket undertræ som skal vælges. Dette foregår ved hjælp af et kald til den instans af *DecisionTree* der repræsenterer top level træet – denne returnerer så et tal mellem 0 og 3, der angiver indekset på det valgte undertræ.

Når undertræet er valgt foretages opslag i *agentInfo* objektet samt kald til funktioner i *aiPlayer* for at få inputtene til det valgte undertræ. Grunden til at disse inputs først indsamles efter der er valgt undertræ er, at hver af undertræerne skal have forskellige inputs, og for at spare beregninger indsamles kun de nødvendige inputs.

Der foretages endnu et kald til *DecisionTree,* som herefter returnerer det handlingssæt, som agenten skal følge. Handlingerne sendes videre tilbage til *aiPlayer*, sammen med information om hvilket undertræ handlingspointerne passer til. Herefter kan agenten udføre de givne handlinger.

Der er desuden implementeret et låsesystem til når der skal trænes efter *Predator vs. Prey* teknikken. Hvis en agent er sat til at opføre sig som *Predator*, kan der kun vælges mellem *attack* og

help attack træerne. Omvendt kan en agent, som agerer *prey* kun vælge mellem *defend* og *help defend* træerne.

TreeStructure benyttes desuden til at indlæse og gemme agenternes træsystemer, sådan at agenternes adfærd kan lagres i en fil til senere brug. Funktionerne hertil hedder *saveAgent* og *loadAgent*. De kaldes begge med agentens nummer og et mappenavn. Filen navngives efter agentens nummer og bliver placeret i en mappe med det mappenavn, som funktionen bliver kaldt med.

6.5 Optimering af tidsforbruget

Under udvikling af dette system, har begrænsning af tidsforbruget vist sig at være en af de store udfordringer, da algoritmens resultat afhænger af hvor lang tid den får til at lede efter løsninger, eller omvendt hvor meget algoritmen (især evalueringen) kan speedes op. Derfor er der i løbet af projektet gjort en række tiltag for at få algoritmen afviklet hurtigst muligt. I dette afsnit ses på nogle af de steder der er optimeret.

Torque lægger som udgangspunkt op til at udføre en del funktioner i scriptet, da informationer om spillets virtuelle verden er meget lettere tilgængeligt her, og samtidigt er der lavet flere af de hjælpefunktioner, som bruges til agenthandlinger i scriptet. Desværre viste scriptet sig at være for langsomt til de fleste af disse, hvorfor der er en del af denne funktionalitet der er flyttet til C++ koden. Der er implementeret over 20 funktioner til udregning af input. Heriblandt fungerer nogle ved at gemme information om f.eks. hvilke fjender holdet kan se, så andre agenter kan aflæse dette i konstant tid, mens andre er individuelle for hver agent og returnerer inputtene direkte.

Det er generelt forsøgt at genbruge så mange beregninger af data som muligt, for at optimere på systemet. Det har især været vigtigt spare på raycasts da disse er beregningsmæssigt meget dyre og tidskrævende. I denne forbindelse er der udviklet 'light' versioner af nogle af funktionerne. Et eksempel herpå er *findNearestEnemyInSightLE*, som er en light udgave af funktionen *findNearestEnemyInSight*. Idéen er, at når der allerede er kendskab til hvilke agenter der kan ses, er det ikke nødvendigt undersøge dette igen. Derfor gemmes der information om hvilke agenter der er synlige i et bit array i *agentInfo* objektet. *findNearestEnemyInSightLE* benytter bit arrayet til at afgøre hvilke agenter, der skal måles afstand til. På tidspunkter hvor en sådan liste ikke findes eller skal opdateres, er det naturligvis stadig nødvendigt at lave raycasts.

eventCheker systemet som er beskrevet i afsnit 6.4.1.2, er et andet eksempel på et sted, der er optimeret ved at udføre operationer som er fælles for et helt hold af agenter og dermed spare på blandt andet raycasts i forhold til at udføre disse for hver agent.

Alt i alt har det forskellige optimeringstiltag betydet at spillet er gået fra at kunne speedes op til at køre ca. 3,5 gange realtid, til at kunne køre ca. 20 gange så hurtigt som realtid. En hastighedsforøgelse på over 5 gange, hvilket vurderes at have været indsatsen værd.

Kapitel 7
Test

Dette kapitel omhandler de overvejelser der er gjort i forbindelse med valg af tests og hvilke tests der er udført. Endvidere præsenteres resultatet af testene.

7 Test

I forbindelse med software udvikling er det altid en vigtig del af arbejdet at teste om softwaren fungerer. Det skal kort nævnes, at der i denne rapport arbejdes med forskellige former for tests. Ordet optræder i to forskellige hovedsammenhænge. Den ene sammenhæng er i forbindelse med softwaretest, hvormed der menes en test af programmets korrekthed, altså om programmet fungerer efter hensigten. Den anden sammenhæng er i forbindelse med parameter tuning og den endelig test, som omhandler afprøvning og tuning af forskellige parameterindstillinger og træning af agentsystemet. Dette kapitel omhandler alene softwaretest.

7.1 Teori om software test

Softwaretest inddeles normalt i to hovedtyper af test. Den ene type kaldes *strukturel test* og den anden kaldes *funktionel test*. I de følgende to underafsnit ses på hver af de to testtyper. Det er her valgt kun at give en kort introduktion til de metoderne, men yderligere information om testtyperne findes i kilden **[sestoft]**. I afsnit ’7.2 Den valgte testprocedure’ belyses herefter hvordan denne rapport forholder sig til de to testtyper, og hvilke former for tests, det er valgt at anvende i forbindelse med projektet.

7.1.1 Strukturel test

I den strukturelle test tages et nærmere kig på den kode der testes. Det undersøges om al koden bruges eller om der eventuelt kan reduceres i den, samt om den kan gøres enklere. Herunder indgår blandt andet kontrol af hvor mange gange der loopes i while løkker, og om if-else konstruktioner både bliver sande og falske ind i mellem. Derfor kaldes denne form for test også *intern test* eller *white-box test*, da testeren har indblik i koden. Testeren udarbejder i denne forbindelse en *test case suite* som består af en samling inputs, til den funktion der testes, og en samling forventede outputs. Inputtene skal være udtømmende, således at der under testen kommes rundt om al koden og på det den måde demonstreres at alle forgreninger i koden kan komme ud for at blive afviklet.[54]

7.1.2 funktionel test

I den funktionelle test laves en test af hvordan programmet opfører sig under afvikling. Forskellen på denne test og den strukturelle test er, at den funktionelle test ikke tager udgangspunkt i koden, men snarere i slutbrugerens oplevelse. I den funktionelle holdes koden skjult, hvorfor denne testtype også kaldes for *black-box test* eller *ekstern test*. Også her laves en *test case suite*, men der tages udgangspunkt i det problem som koden skal løse, og der laves så både legale og ugyldige inputs, for at teste hvor godt programmet håndterer disse.[55]

7.2 Den valgte testprocedure

I dette afsnit beskrives det hvordan der er softwaretestet i projektet, og der argumenteres desuden for det valgte testforløb. Derudover ses på hvorledes de udførte tests af programmet er forløbet, og der gives en vurdering af brugbarheden af det færdige produkt på grundlag af testresultaterne og

[54] Kilde: **[Sestoft]**
[55] Kilde: **[Sestoft]**

observationer. Det skal dog her nævnes, at software tests ikke har været en del af hovedfokuset i dette projekt.

Da det skulle bestemmes hvilke former for test der skulle udføres i forbindelse med dette projekt, stod det hurtigt klart, at en strukturel test kunne udelukkes. Denne beslutning skyldes primært, at det er en enorm langsommelig proces at udføre strukturel test, og at udbyttet ved en sådan test ikke vil kunne stå mål med de tidsmæssige omkostninger, der er ved at udføre testen.

Den funktionelle test kunne til gengæld umiddelbart virke mere oplagt at begive sig i kast med. Dels kan den funktionelle test som oftest udføres væsentligt hurtigere, og man kan sige, at det i dette projekt er helt anderledes interessant at få bekræftet at samtlige funktioner fungerer efter hensigten ved hjælp af en funktionel test, end det er interessant at finde ud af om en while-løkke kan erstattes med en if-sætning ved hjælp af en strukturel test. Begge dele er naturligvis relevant under software projekter, men det er meget normalt for større software projekter, at man prioriterer kun at lave den funktionelle test, da den strukturelle test er alt for tidskrævende.

Da der ikke er udarbejdet egentlige testskemaer for en sådan funktionel test, skyldes dette at der er en række egenskaber ved projektet, som har betydet at det alligevel ikke har været helt så oplagt at bruge tid på testen.

En af grundene til at den funktionelle test er undladt er, at der arbejdes med en metaheuristisk søgning og dermed indgår en vis tilfældighedsfaktor. Det vil sige, at når der skal opstilles skemaer for hvilke forventede outputs der knytter sig til et sæt inputs, er det meget svært at vide hvordan disse skal konstrueres.

En anden årsag til fravalget af testen ligger i, at selvom den funktionelle test er væsentligt mindre tidskrævende end den strukturelle, har den alligevel stadig et enormt omfang i forhold til de tidsmæssige ressourcer for dette projekt, og det blev derfor vurderet, at omkostningerne ved at opstille en systematisk funktionel test i skemaer og tabeller ikke kan måle sig med det udbytte der fås ved at udføre testen.

Dertil kommer, at der under udviklingen af dette program løbende er blevet udført tests af om softwaren opførte sig efter hensigten ved at teste enkelte funktioner straks efter implementeringen af disse, og igen i sammenhæng med andre funktioner, når dette var muligt. Denne test er dog sket uden at opstille testskemaer og ikke helt så systematisk, men den har alligevel gjort det mindre kritisk med en endelig funktionel test.

En anden faktor som har spillet ind på beslutningen om at undlade den funktionelle test er, at programmet automatisk gennemgår en form for stresstest i forbindelse med at individerne trænes. Under træningen speedes afviklingen af spillet nemlig op så det kører ca. 20 gange så hurtigt som ved den normale hastighed spillet afvikles i. Spillet har så fået lov til at køre i over 35 timer, uden at der opstod problemer. Det svarer til at have spillet i realtid i over 700 timer uden at opleve fejl, der fik programmet til at crashe.

Under dette stresstestforløb blev der desuden holdt nøje øje med hukommelsesforbruget, for på den måde at sikre, at der ikke forekom memory leaks. Det faktum at der ikke er memory leaks beviser naturligvis ikke at programmet nødvendigvis vil kunne køre i meget længere tid end de 35 timer som det har kørt uden at gå ned, men det er stadig en forudsætningerne for, at det vil kunne lade sig

gøre, da programmet med sikkerhed vil gå ned, hvis der forekommer memory leaks, så snart hukommelsen er brugt op.

Alt i alt konkluderes det, ud fra de forskellige tests, at programmet fungerer efter hensigten. Der er som sagt ikke lavet skemaer hvor testresultater kan aflæses, men det observeres, at programmet ikke fejler under stresstesten, og at agenterne samtidigt bliver bedre og bedre desto mere de for lov til at træne Endvidere opstår der ikke memory leaks og forøget hukommelses forbrug. De udførte tests giver naturligvis igen garanti for at der ikke kan være enkelte mindre fejl i koden, men det viser i hvert fald at programmet overordnet set fungerer efter hensigten, og gør dermed sandsynligheden for fejl i koden væsentligt mindre. Endvidere gives de tests der løbende er udført under udviklingen en væsentlig indikation på, at systemet virker efter henseende.

Kapitel 8
Parameter tuning og analyse af resultater

I dette kapitel ses på hvordan parameter tuningen er forløbet og på de resultater som algoritmen har opnået.

8 Parameter tuning og analyse af resultater

I dette kapitel kigges på hvilke parametre, der kan indstilles på i forbindelse med optimeringen af agentsystemet. Grunden til det er vigtigt at parameter tune de genetiske algoritmer er, at algoritmens performance afhænger deraf. Det er naturligt at optimeringen af beslutningstræerne ændres, når man f.eks. ændrer på hvor meget der skal muteres, hvor stor en del, der skal stamme fra hvilken forælder i forbindelse med krydsning, hvor mange individer, der skal udskiftes i hver generation osv. For så at finde ud af hvilken kombination af disse parametre, der giver den bedste optimering, laver man først en parameter tuning for hver specifik genetiske algoritme man vil teste, og efterfølgende tester man så algoritmerne med de parametre, der passer bedst til hver algoritme. I dette projekt testes der fire forskellige algoritmer (store dele af algoritmerne er ens – men mutationen og krydsningen gøres på forskellige måder, jf. afsnit 5.9.2.3 og afsnit 5.9.2.4). Hver af disse algoritmer skal så parameter tunes, så hver algoritme får det bedst mulige udgangspunkt til at optimere beslutningstræerne. Herefter kan algoritmerne sammenlignes. Dette gøres ved at lave en endelig træning, hvor algoritmerne får lov til at benytte de bedste parametre, der er fundet og så træne i flere generationer end ved parameter tuningen. I denne opgave er valgt træne i 600 generationer, men tallet kunne også have været større, hvis man havde mere tid til at forsøge, at finde en bedre løsning. Herefter udtages de bedste hold fra disse træninger og disse hold får så lov til at dyste mod hinanden. På den måde findes det bedste hold, og dermed hvilken algoritme, der har været bedst til at optimere beslutningstræerne. For at få et bedre statistisk grundlag burde man både foretage parametertuningen og den endelige test flere gange, og så bruge et gennemsnit af performancen som grundlag for at afgøre hvilke parametre og algoritmer der var bedst. Desværre var det ikke muligt at nå i løbet af projektet, da en enkelt afvikling af algoritmerne allerede er forholdsvis tidskrævende. Dette svækker selvfølgelig det statiske grundlag for de resultater, der præsenteres i dette afsnit, som dog alligevel kan bruges til at give en indikation af hvor godt algoritmerne fungerer.

Da projektet er stort og algoritmerne kan implementeres på mange måder og efter mange strategier, er der rigtigt mange parameter værdier og kombinationer af samme, der bør testes. Dette faktum, kombineret med, at evalueringen af nye agenter tager forholdsvis lang tid, fordi agenterne er nødt til at spille mod hinanden for at blive evalueret, gør, at det ikke har været muligt at teste alle kombinationer af parameterindstillinger og algoritmevalg. Derfor er det nødvendigt at afgrænse testområdet (parameter tuningen), hvilket gøres i det kommende afsnit. En liste over alle parametre og algoritme valg findes i Tabel 2.

Parametre						
Hovedgruppe	**Parameter gruppe**	**Parameter**	**Type**	**Mulige værdier**	**Test værdier**	**Start værdi**
Genetiske algoritmer	Population	Samlet antal agenter pr. generation	Heltal	{1, ∞}	Låst	2*4*8 = 64
Genetiske algoritmer	Population	Antal agenter pr. generation pr. ø	Heltal	{1, 8}	Låst	8
Genetiske algoritmer	Population	Antal øer	Heltal	{1, ∞}	Låst	4
Genetiske algoritmer	Population	Antal agenter pr. migration mellem øer, pr. ø	Heltal	{0, 8}	Låst	1
Genetiske algoritmer	Population	Antal generationer mellem migrationer	Heltal	{0, 8}	{5, 7, 10}	5
Genetiske algoritmer	Population	Antal agenter pr. visitation mellem øer	Heltal	{0, 8}	Låst	1
Genetiske algoritmer	Population	Antal generationer mellem visitationer	Heltal	{1, ∞}	{6, 8, 11}	6
Genetiske algoritmer	Population	Antal generationer mellem turneringer	Heltal	{1, ∞}	{4, 9, 12}	4
Genetiske algoritmer	Udvælgelse	Hvor mange agenter pr. generation skal udskiftes	Heltal	{1, 8}	{4, 6, 8}	4
Genetiske algoritmer	Krydsning (branch)	Hvor mange grene pr. kromosom	Heltal	{1, antal gener}	{6, 8, 10}	3
Genetiske algoritmer	Krydsning (branch)	Hvor mange gener pr. gren	Heltal	Specifikt for gren	Låst	tilfældigt
Genetiske algoritmer	Krydsning (one point)	Hvor mange gener fra forælder 1	Decimal	{0, 1} (andel)	{0.1, 0.2, 0.3}	0.1
Genetiske algoritmer	Mutation	Mutations rate (hyppighed)	Decimal	{0, 1} (andel)	Låst	Faldende rate
Genetiske algoritmer	Mutation	Øvre grænse for mutations forskydningsrate	Decimal	{0, 1} (andel)	{1, 0.4, 0.1}	1
Genetiske algoritmer	Mutation	Nedre grænse for mutations forskydningsrate	Decimal	{0, 1} (andel)	{0, 0.01, 0.05}	0
Genetiske algoritmer	Mutation (guidet muation)	Mutations forskydningsrate (andel af forskel der skal lægges til/trækkes fra)	Decimal	{0, 1} (andel)	{0.25, 0.17, 0.13}	0.25
Genetiske algoritmer	Mutation (tilfældig muation)	Mutations forskydningsrate (hvor meget der skal muteres)	Decimal	{0, 1} (andel)	Låst	tilfældigt
Genetiske algoritmer	Fitness	Vægt ganget på skade	Heltal	{0, ∞}	{1, 3, 5}	1
Genetiske algoritmer	Fitness	Vægt ganget på health	Heltal	{0, ∞}	{1, 3, 5}	1
Genetiske algoritmer	Fitness	Vægt ganget på team damage	Heltal	{0, ∞}	Låst	1
Genetiske algoritmer	Træning	Antal generationer før kombinering af angrebs og forsvars agenter	Heltal	{0, ∞}	Låst	30
Genetiske algoritmer	Træning	Antal generationer der skal trænes mod agenter der står stille	Heltal	{0, ∞}	Låst	5
Genetiske algoritmer	Træning	Antal generationer der skal trænes mod agenter styret af FSM'er	Heltal	{0, ∞}	Låst	5 (+dem fra parameteren nedenunder)
Genetiske algoritmer	Træning	Antal generationer mellem træning mod FSM'er	Heltal	{0, ∞}	Låst	5
Genetiske algoritmer	Træning	Antal generationer agenterne trænes	Heltal	{0, ∞}	Låst	50
Forskellige algoritmer	Evolutionær algoritme	Generational EA (bakterie evolution)	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Evolutionær algoritme	Steady state EA (elefant evolution)	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Udvælgelse, agenter til reproduktion	LFR-SUS	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Udvælgelse, agenter til udskiftning	Tournament selection	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Krydsning	One-point	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Krydsning	Geometrisk (branch)	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Mutation	Tilfældig	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Mutation	Guidet	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Andet	Racer (prey/predator)	Boolean	{ja, nej}	Ja	-
Forskellige algoritmer	Andet	Ikke specialiseret modstander	Boolean	{ja, nej}	Ja	-

Tabel 2: Viser alle parametre og algoritme valg, der er benyttet i forbindelse med dette projekt.

8.1 Afgrænsning af test

Tiden evalueringen af agenter tager, afhænger naturligvis af tiden et spil tager. I de fleste FPS multiplayer spil, som kan sammenlignes med den type spil de evolutionære agenter skal begå sig i, er standardspilletiden ud fra egne erfaringer vurderet til at være et sted mellem ét og fem minutter per bane. Det giver derfor ikke mening at lade agenterne spille i kortere tid end 1 minut, da de herved ikke vil kunne nå at deltage på den tilsigtede måde i spillet. Ligesom med nogle af parametrene, er spilletiden blevet fastlåst, således at denne er 90 sekunder i alle de udførte test. Det betyder samtidig, at en evaluering af en generation principielt ikke kan gennemføres hurtigere end de 90 sekunder, plus det tid selve algoritmen tager. Dog har det vist sig, at spillet kan simuleres lidt over tyve gange hurtigere en realtid, bl.a. pga. de optimeringer, der er beskrevet i afsnit '6.5'. Selvom spilletiden altså er formindsket med ca. en faktor tyve, betyder det alligevel, at den samlede test tid hurtigt bliver meget stor, hvis man skal lave udtømmende test med alle kombinationer.

En hurtig udregning viser, at hvis der 'blot' skal testes kombinationerne af parametre, som har med de genetiske algoritmer at gøre, med 3 forskellige værdier og med ét datasæt på én ø og kun én gang hver, vil det tage 3^{21} parameterkombinationer (21 parametre der kan indstilles). Dette tal skal herefter ganges med de 50 generationer, som træningen kører, og dette skal så ganges med den tid det tager at evaluere agenterne (90/20) sekunder. Omregnet fås ca. 74 000 år, hvis der benyttes en 3.4ghz computer med 2gb ram, som tids målingerne er udført på. Hertil kommer, at det på forhånd ikke kan vides hvilke algoritmer, der løser opgaven bedst, hvorfor flere test bliver nødvendige. Endvidere bør man træne flere gange med samme parametre og algoritmer og så udregne middelværdien og spredningen af hvordan de klarer sig, for at lave en optimal sammenligning af parameter- og algoritme valg. Ikke overraskende er dette ikke muligt i dette projekt, hvorfor det har været nødvendigt at finde en heuristik til parametertuningsproblemet.

Valget er faldet på en løsning med en grådig algoritme, der i stedet for at afprøve alle kombinationer af parametre, først forsøger at finde den bedste værdi til parameter 1 (ud fra 3 mulige værdier), dernæst gemmes denne bedste værdi og benyttes som valgt værdi til parameter 1 i resten af tuningen. Herefter afprøves tre værdier for parameter 2, og den bedste gemmes og bruges efterfølgende - og sådan fortsættes der til alle parametre er blevet testet med tre forskellige værdier og den bedste er blevet valgt for hver af disse.

Grunden til denne grådige og simple tilgang er valgt, er netop fordi det ønskes at implementere noget, der 'hurtigt' (det tager stadig ca. 10 timer at lave en parameter tuning), kan finde frem til en parameter kombination, der er nogenlunde fornuftig. Det betyder også, at resultaterne fra parametertuningen kun kan bruges som en indikation på hvordan parametrene indstilles smart, men uden et stærkere grundlag, kan det ikke siges præcist hvilken parameter kombination, der er den bedste.

Som det ses i Tabel 2, er der 23 forskellige parametre der kan indstilles, men da nogle af dem hører til specifikke algoritmer, kan højst 21 af dem være aktuelle ad gangen. Hertil kommer, at nogle af parametrene fastlåses til bestemte værdier af andre årsager – f.eks. er antallet af agenter på hvert hold fastlåst til otte, da der i denne type spil ikke er flere end otte agenter på banen ad gangen – og agenterne skal trænes til den situation, som spillet skal spilles i. Samtidigt giver det heller ikke mening at have færre agenter på hver hold, da der så vil være meget lidt genmateriale til rådighed for algoritmen. Hvis man vil have en større population kan man bruge flere øer, hvilket til gengæld gør, at det tager længere tid at evaluere en generation af agenter, da spillet så skal spilles det antal

gange, som der er øer. Afvejningen er derfor klassisk og står mellem hvor stor en del af søgerummet man vil undersøge kontra hvor meget tid man vil bruge. Det er besluttet at benytte 4 øer med 2 hold på hver, hvilket i alt giver 64 agenter som trænes.

Der er flere parametre der er fastlåste ud fra samme argumentation; det er angivet i parameter skemaet i Tabel 2, hvilke der er låst. I alt er der 11 parametre, der tunes med 3 værdier hver. Algoritmevalgene er på forhånd blevet låst på baggrund af argumentationen i afsnit '5.9 Optimering af beslutningstræer, der styrer NPC'er'. De parametre der skal testes og de parameterværdier der er valgt til parameter tuningen, er valgt ud fra dels små strikprøve tests samt en fornemmelse af hvilke parametre, der er mest afgørende for agenternes udvikling gennem generationerne. Denne fornemmelse bunder i rådgivning fra metaheuristik-ekspert Thomas Stidsen[56]. I de følgende afsnit bliver de konkrete udførte tests og resultater præsenteret.

8.2 Tuningen, testene og resultaterne

I dette afsnit præsenteres resultaterne fra selve parametertuningen, den endelige træning og turneringen mellem agenterne.

8.2.1 Parametertuningen

Selve parametertuningen er, som nævnt, lavet med tre forskellige værdier for hver af de valgte parametre. Parametertuningen er så udført én gang på hver algoritme. Resultaterne af parameter tunings testene findes i Tabel 3.

Parameter	Testværdier	Valgt til GA1	Valgt til GA2	Valgt til GA3	Valgt til GA4
Antal generationer mellem migrationer	{5, 7, 10}	10	10	7	10
Antal generationer mellem visitationer	{6, 8, 11}	6	6	6	6
Antal generationer mellem turneringer	{4, 9, 12}	9	9	4	9
Hvor mange agenter pr. generation skal udskiftes	{4, 6, 8}	4	4	4	6
Hvor mange grene pr. kromosom	{6, 8, 10}	10	8	-	-
Hvor mange gener fra forælder 1	{0.1, 0.2, 0.3}	-	-	0.3	0.1
Øvre grænse for mutations forskydningsrate	{1, 0.4, 0.1}	0.1	0.1	0,4	1
Nedre grænse for mutations forskydningsrate	{0, 0.01, 0.05}	0.05	0.05	0.05	0.05
Mutations forskydningsrate (andel af forskel der skal lægges til/trækkes fra)	{0.25, 0.17, 0.13}	0.25	-	0.13	-
Vægt ganget på skade	{1, 3, 5}	3	3	5	1
Vægt ganget på health	{1, 3, 5}	1	3	1	1

Tabel 3: GA1 er den genetiske algoritme med branch crossover og guidet mutation. GA2 er den genetiske algoritme med branch crossover og tilfældig mutation. GA3 er den genetiske algoritme med one point crossover og guidet mutation. GA4 er den genetiske algoritme med one point crossover og tilfældig mutation.

Det skal hertil siges, at paramenterne 'Vægt ganget på skade' og 'Vægt ganget på health', kun vælges til den specifikke race. Dvs. at 'Vægt ganget på skade' kun benyttes til agenter, der fungerer som 'predator' i prey/predator træningen og omvendt benyttes 'Vægt ganget på health' kun til de

[56] Rådgivning fra Lektor ved IMM, DTU, Thomas Stidsen. Metaheuristik-ekspert og underviser bl.a. i PhD kursus 02719 Meta-Heuristic optimization.

agenter, der fungerer som 'prey'. På den måde skulle agenternes specifikke angrebs og overlevelses egenskaber gerne blive fremavlet.

Træningen med de bedste parametre

I dette afsnit beskrives den endelige test med de bedst fundne parameterkombinationer fra Tabel 3. Det bemærkes, at hver af algoritmerne kun er trænet én gang hver, hvilket selvfølgelig er i underkanten, da algoritmernes start løsninger er tilfældigt genereret. For at give et bedre statistisk grundlag at vurdere ud fra, burde man lade algoritmerne optimere flere populationer og måle den gennemsnitlige bedste fitness og middelværdi. Det har der dog ikke været tid til i forbindelse med dette projekt. Resultaterne skal nok derfor snarere ses som en indikation af hvordan algoritmerne performer, frem for et egentligt facit. Denne endelig træning er udført for følgende algoritmer.

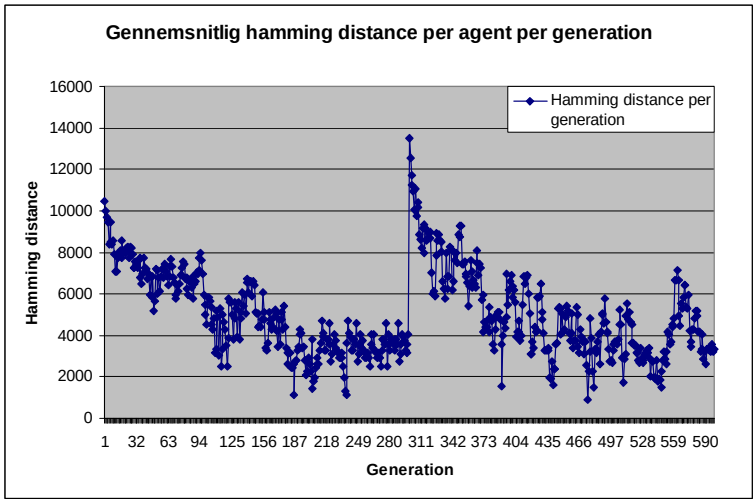
- GA1 med branch krydsning og guidet mutation
- GA2 med branch krydsning og tilfældig mutation
- GA3 med one point krydsning og guidet mutation
- GA4 med one point krydsning og tilfældig mutation

Hver algoritme har trænet i 600 generationer, hvoraf de første 15 er mod agenter, der blot står stille. De næste fem er mod FSM'erne og de næste op til generation 300 er angrebs agenterne mod forsvars agenterne med træning mod FSM'erne hver tiende generation. Endeligt er de sidste 300 generationers træning med agenter, der er kombineret angrebs og forsvars agenter. Dvs. at agenten nu har alle fire 'undertræs-strategier' til rådighed at vælge mellem.

Som træningen forløber, kommer træerne til at ligne hinanden mere og mere, som en naturlig konsekvens af udvælgelsen og krydsningen af de bedste agenter. I den forbindelse er det interessant at kigge på hvor tæt på hinanden de egentligt kommer, for at sikre, at søgningen ikke bliver for specialiseret (agenterne for hurtigt kommer til at minde meget om hinanden). Når agenterne bliver for ens vil de typisk ikke blive bedre af at blive trænet i flere generationer, da det her meget er samme genmateriale, der vil være til rådighed hos agenterne, hvorfor deres adfærd ikke vil ændres. Til at måle afstanden mellem agenternes træer benyttes hamming distance[57]. Denne udregnes som den absolutte forskel på kromosomerne gen for gen. Dvs. at grænseværdier og handlingspointere i træerne sammenlignes og forskellen på hver enkelt grænseværdi og hver enkelt handlingspointer lægges så sammen. Den samlede sum er så hamming distancen mellem de to sammenlignede træer.

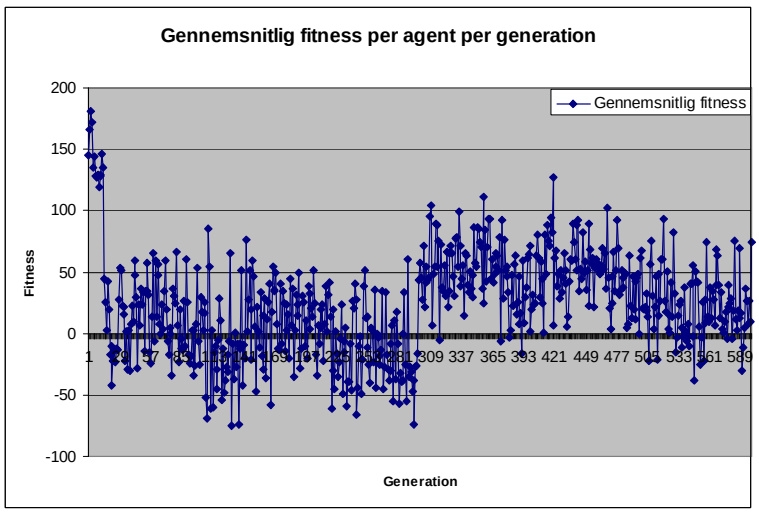
Nedenfor ses en figur, der viser den gennemsnitlige hamming distance for predator racen i træningen med algoritmen med one point krydsning og tilfældig mutation. Det ses af figuren, at hamming distancen, ganske som forventet, falder indtil generation 300, hvor agenternes træer sættes sammen. Her stiger hamming distancen naturligt nok drastisk, da racerne kombineres til én agenttype (race). Efterfølgende ses det, at hamming distancen igen begynder at falde, efterhånden som optimeringen af træerne sker. Det ses også af figuren, at hamming distancen ikke kun er nedadgående (også andre steder end ved generation 300) – det forklares ved, at populationen tilføres nyt genmateriale fra mutation, ø-migration og ø-visitation periodisk. Tilsvarende kurver findes for træningen med de øvrige algoritmer.

[57] Hamming distance kendes bl.a. fra **[AI]** s. 735



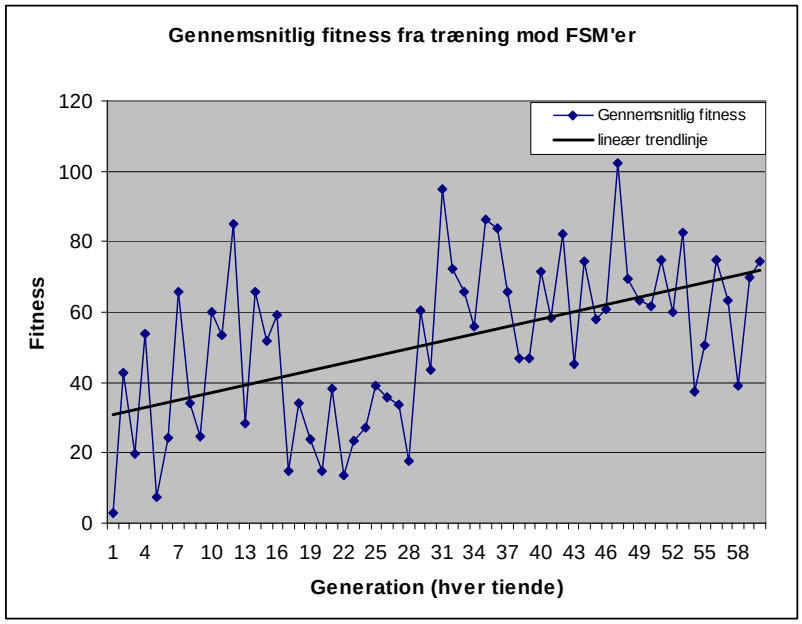
Figur 28:Viser den gennemsnitlige hamming distance mellem agenterne i generation. Data er fra den endelig træning med one point krydsning og tilfældig mutation

Mht. agenternes fitnessværdier under træningen, er disse plottet nedenfor. De første 15 generationer træner agenterne mod modstandere, der blot står stille og ikke giver modstand – derfor er agenternes fitness værdier naturligt høje her. Det skal siges, at de værdier der er plottet er den rene fitness – dvs. uden vægte ganget på skade, health eller holdskade. Som forventet, i en prey and predator træning, svinger fitness værdierne meget. Idéen med prey and predator træningen, er netop, at de to racer skal tilpasse sig hinandens nye strategier og på den måde udvikle hinanden. Derfor er det meget fint, at de to racer skiftes til at få 'gode' og 'dårlige' fitness værdier (holdenes fitness værdier afhænger jo også af hvor godt modstanderen klarer sig). På Figur 29 er fitness værdierne plottet for angrebs agenterne. 0 på figuren svarer til at agenten selv har taget lige så meget skade som den har skadet andre agenter, hvilket betyder at agenter kan få et negativt antal fitness point. Som det ses ligger klarer de to hold sig ca. lige godt de første 300 generationer, herefter sættes agenterne sammen hvilket set ud til at være til angrebsagenternes fordel, men det lader til at forskellen udjævnes hen mod slutningen af evolutionen. Det kunne her tyde på, at 'prey' racen har adapteret sig til 'predatornes' strategi.



Figur 29:Viser den gennemsnitlige fitness per agent per generation i den endelig træning med algoritmen, hvor der benyttes branch krydsning og guidet mutation.

For at vurdere om agenterne så er blevet bedre eller de bare har skiftedes til at være dårlige, er der gjort to ting. For det første er der, som nævnt, under træningen af agenterne, periodisk trænet mod de fast definerede FSM'er. Dette er gjort hver tiende generation for denne endelige træning af agenterne – og fitnessværdierne for agenterne i disse generationer er plottet nedenfor. For det andet gennemføres en turnering, hvor de bedste agenter fra træningen med hver af algoritmerne, spiller mod hinanden og FSM'erne, for at se hvem der er bedst.



Figur 30: Viser fitness værdier fra træningen med geometrisk krydsning og guidet mutation. Det er kun fitness værdier fra træningen mod FSM'er der er plottet.

Selvom der er kraftige udsving i fitnessværdierne ses det dog, at den gennemsnitlige fitness per agent øges efterhånden som træningen skrider frem. En del af udsvinget hænger formentlig sammen med måden den genetiske algoritme optimerer på, hvor sammensætning af to gode kromosomer ikke nødvendigvis giver et super kromosom – og mutationen af gener ikke altid gør kromosomet bedre. Man kan derfor sige, at svingningerne kan være en naturlig konsekvens af at algoritmen prøver at lede forskellige steder i søgerummet. Det vigtigste er derfor, at optimeringen i gennemsnit gør, at fitnessværdierne bliver bedre, hvilket også er tilfældet i figuren. Endvidere kan noget af udsvinget muligvis forklares af den stokastiske fitnessfunktion, der gør det sværere at vurdere agenternes præstation. Af det kan udledes, at det havde været bedst at lade agenterne kæmpe mod hinanden flere gange i hver generation, og tage et gennemsnit af deres fitnessværdier, for at få en mere korrekt måling af disse. Dette har der imidlertid ikke været tid til, da det vil betyder en markant forøgelse af tidsforbruget.

Endvidere er det sandsynligt, at agenternes fitness kan forbedres mere ved at lade dem træne yderligere, men da dette naturligvis vil tage længere tid, er det desværre ikke nået i løbet af dette projekt. Endeligt bemærkes det, at agenterne bliver bedre efter de gives flere handlemuligheder, hvilket kunne underbygge, at en yderligere udvidelse af søgerummet, i form af flere handlemuligheder og inputs, vil kunne give en yderligere forbedring af agenterne.

8.2.2 Turnering mellem agenterne, der er trænet med de forskellige algoritmer

For at afgøre hvilken algoritme, der er bedst til at optimere på det valgte agentsystem, er der designet en turnering, hvor agenter fra hver algoritmes træning spiller mod hinanden. Turneringen er en pulje turnering hvor alle møder alle 10 gange (fem gange fra hver startposition på banen) i en kamp på 90 sekunder. Vinderen af hver kamp er det hold, der får den højeste samlede fitness og vinderen af turneringen er det hold der vinder flest kampe. Nedenfor er angivet en tabel, hvor resultaterne af turneringen fremgår. I tabellen fremgår det, hvor mange ud af de 10 kampe hver hold vandt; f.eks. ses det, at GA1 vandt seks gange over GA2 og GA2 vandt fire gange over GA1. Under sejre er listet hvor mange sejre agenterne har og i parentes hvor mange kampe holdet har vundet. Det samme gælder for uafgjorte og tabte kampe.

	GA1	GA2	GA3	GA4	FSM	Sejre	Uafgjorte	Tabte
GA1		6-4	6-4	6-4	9-1	4 (27)	0 (0)	0 (13)
GA2	4-6		3-7	4-6	7-2	1 (18)	0 (1)	3 (21)
GA3	4-6	7-3		4-6	9-1	2 (24)	0 (0)	2 (16)
GA4	4-6	6-4	6-4		9-1	3 (25)	0 (0)	1 (15)
FSM	1-9	2-7	1-9	1-9		0 (5)	0 (1)	4 (34)

Tabel 4: GA1 er den genetiske algoritme med branch crossover og guidet mutation. GA2 er den genetiske algoritme med branch crossover og tilfældig mutation. GA3 er den genetiske algoritme med one point crossover og guidet mutation. GA4 er den genetiske algoritme med one point crossover og tilfældig mutation. FSM er den implementerede tilstandsmaskine.

Som det ses, er GA1 den algoritme, der har trænet de agenter, som har klaret sig bedst i turneringen. Således konkluderes det, at branch krydsning og guidet mutation har givet de bedste agenter i denne træning. Det bemærkes endvidere, at alle agenterne, der er trænet af de genetiske algoritmer har slået agenterne, der er styret af den implementerede tilstandsmaskine. Endvidere skal det siges, at der i kampene var en overvægt af sejre, når agenterne startede på et bestemt af de to startsteder. Ud af de tres kampe, der blev spillet uden agenter styret af tilstands maskiner, blev de 39 af dem vundet fra samme startposition. Det tyder derfor på, at denne startposition er mere fordelagtig end den anden.

For at måle hvor meget bedre agenterne er blevet under træningen har vi også ladet det vindende hold spille mod agenter med tilfældigt sammensatte kromosomer, svarende til agenter fra generation 1. Hver kamp blev spillet mod agenter med nye tilfældige kromosomer, for at give et bedre statistisk grundlag at vurdere ud fra – det kunne jo være at de tilfældige kromosomer blev initialiseret på en særlig heldig eller uheldig måde. Her vandt turneringens vindende hold overlegent 10-0 i kampe, som også blev gennemført således, at hvert hold startede fem gange på hver startposition på banen. Endvidere havde det vindende hold i gennemsnit ca. 1,5 gange så meget i fitnessværdier, som agenterne fra generation 1, hvilket bekræfter, at træningen har båret frugt. 1,5 gange lyder måske ikke af så meget, men hertil skal det siges, at agenterne får fitness point blot for at overleve (ikke miste health), hvorfor det er svært at vinde meget stort over det andet hold, hvis blot enkelte agenter fra det andet hold formår at gemme sig og derved undgår at miste health.

Spørgsmålet er selvfølgelig så om de trænede agenter rent faktisk er bedre og sjovere at spille mod end dem en spiludvikler kan hardcodes i en FSM. Da vi ikke har nogle sådan FSM'er at lade agenterne spille mod, og vi ikke har haft mulighed for at afsætte tid til en brugertest-periode, hvor agentsystemet kunne blive vurderet så objektivt som muligt af folk ude fra, bliver det i stedet en

subjektiv vurdering. Umiddelbart virker det som om, at de trænede agenter ikke vil være overlegne i forhold til agenter styret af FSM'er i spil som f.eks. Hitman[58]. Det skal dog siges, at det ikke kan konkluderes, at de evolutionære agenter ikke *kan* blive bedre, men de funktioner, der reelt er til rådighed for agenterne i forbindelse med dette projekt, er ikke nok til at en spiludvikler ikke vil kunne hardcode en FSM, der kan det samme. Der hvor de evolutionære agenter rigtigt vil kunne komme til sin ret, er når agenterne får flere handlemuligheder og input, og det derved bliver svært for en programmør at definere agentens adfærd. F.eks. ville flere oplysninger om banen, flere forskellige måder at bevæge sig på (så som at løbe, gå, snige, kravle, hoppe mfl.), forskellige våben at vælge mellem, forskellige pathfinding muligheder, så agenten kan ankomme til et sted fra flere retninger osv. ganske givet gøre agenterne bedre.

Selvom agenterne har haft noget begrænsede handlemuligheder skal det dog siges, at der undervejs har været tegn på at de evolutionære agenter finder frem til nogle smarte strategier, som ikke umiddelbart kunne forventes på forhånd. F.eks. ser man indimellem agenter, der gemmer sig bag en mur, for så indimellem at komme ud fra gemmestedet for at skyde, og så igen løbe tilbage for ikke selv at blive ramt af modstanderens skud.

Et andet spørgsmål man kan stille sig selv er hvor godt det kombinatoriske problem med at indstille træets værdier er blevet løst. Det er et svært spørgsmål at besvare, da det er svært at analysere præcis hvor godt det er lykkedes at optimere på grænseværdierne og handlingspointerne bare i dette projekt. Problemet er netop, at man på forhånd ikke kan sige noget om, hvordan de optimale kromosomer skal se ud, og man ved derfor ikke hvor tæt agenterne er kommet på dette potentiale. De eneste reelle muligheder der for at sige noget om, hvor godt det kombinatoriske problem med at indstille vægtene i træerne på er løst, er enten at se hvor meget bedre agenternes fitness værdier er blevet, eller at lave en subjektiv vurdering af agenternes adfærd. Den subjektive vurdering er givet ovenfor. Mht. fitness værdierne er disse en del bedre end dem de får til at starte med – men igen er det svært at sige, hvor langt de er fra deres potentiale.

Den genetiske algoritme har formået, at optimere agentsystemet til et stadie, hvor det er væsentligt bedre end udgangspunktet. Spørgsmålet er så om dette også vil kunne lade sig gøre, når agenterne får flere handlemuligheder og input og søgeproblemet derfor bliver væsentligt større. Som minimum må man forvente, at dette vil kræve, at agenterne skal træne i længere tid (flere generationer) og at populationen formentligt også skal forstørres. Ikke desto mindre er netop genetiske algoritmer kendt som en metode, der er god til at optimere på hårde problemer med kæmpe store søgerum[59], hvorfor man alligevel kan forestille sig, at det kunne lade sig gøre, at optræne også væsentligt mere komplekse agenter.

Konklusionen på træningen er derfor, at agenterne bliver afgørende bedre en de simple FSM'er, der er implementeret i forbindelse med dette projekt og at træningen samtidig har udviklet agenterne væsentligt fra deres udgangspunkt, men at agenternes muligheder ikke er mange nok til, at agenten reelt vil adskille sig fra det, en udvikler vil kunne hardcode i en FSM. Med flere input og især flere handlemuligheder vil agenten formentligt kunne få lejlighed til at adskille sig fra sådanne FSM systemer. Sidst konkluderes det, at algoritmen med den guidede mutation sammen og den geometriske krydsning (branch) er den bedste, af de afprøvede, til at optimere træerne, men at forskellen mellem algoritmernes performance ikke er stor.

[58] Hitman er lavet af IO Interactive som dette projekt er lavet i samarbejde med. I Hitman er NPC'erne styret af FSM'er.
[59] Udsagn fra bl.a. **[TS]**

Kapitel 9
Perspektivering og fremtidigt potentiale

Kapitlet indeholder en analyse af hvordan, der kan arbejdes videre med de idéer og resultater, som dette projekt har udmøntet sig i. I den forbindelse, gives et bud på, hvilke metoder, der kunne være spændende at arbejde med i forhold til mere avanceret kunstig intelligens i spil.

9 Perspektivering og fremtidigt potentiale

I dette afsnit ses der på hvilket potentiale genetiske algoritmer kombineret med beslutningstræer har inden for agentstyring i spil, og der gives en vurdering af om det er noget der vil komme mere af i fremtiden. Herunder vil de faldgruber, der er ved at arbejde med et system som dette også blive diskuteret. Desuden opsamles nogle af de ideer, der er til en eventuel videreudvikling af projektets løsningsmodel, som kunne være med til at gøre det samlede system bedre.

Efter at have implementeret dette projekts agentstyringssystem, ligger det nu forholdsvis fast hvor brugbar denne kombination er i praksis i spil. Potentialet for agenterne begrænses naturligvis af den mængde funktioner (handlemuligheder og input), som det har været muligt at nå at implementere i løbet af et projekt af denne størrelse. Men med udgangspunkt i dette projekts begrænsede omfang af implementeringer af disse, har metoden bestemt vist sig at være effektiv, da optimeringen af agenterne er forløbet med løbende forbedring af fitness, som håbet.

Det er klart, at der inden for den tid som var afsat til dette projekt vil kunne laves en (eller mange) forholdsvis avancerede FSM'er. En sådan FSM ville også kunne komme til at fremstå intelligent på den lille og afgrænsede bane, som blev anvendt i dette projekt. Man kan dog forestille sig, at et sådant system kun ville fungere godt i præcis det domæne som FSM'en er programmet til at agere i. Valgte man at placere FSM'en på en radikalt anderledes indrettet bane, kunne man derfor forestille sig, at den ville komme til kort. Netop det faktum, at det er svært at lave en generel FSM model, som passer til alle miljøer, giver også anledning til at ønske sig en mere fleksibel løsningsmodel. Evolutionære agenter vil kunne tilpasses individuelle miljøer automatisk uden menneskelig styring, ved at træne på forskellige baner og i forskellige miljøer. Man kan endda flytte agenter, som er færdigtrænet på én bane til en anden og fortsætte træningen ud fra det udgangspunkt, som allerede er opnået på den første bane. Derved kommer deres første evolution til at fungere som et godt udgangspunkt (en god startløsning) for den anden evolution, da det trods alt må antages, at der er visse ligheder i agentens adfærd, selv på meget forskellige baner. Derved kunne man forestille sig, at der også var visse genegenskaber som med fordel kan anvendes som udgangspunktet for den næste evolution. Således kan man altså lave et system, som tilpasser agenterne til samtlige baner i et komplet spil, uden yderligere menneskelig indblanding, og dermed bliver det en meget skalerbar løsningsmodel.

Projektet har vist, at de evolutionære agenter opnår fine resultater ved at træne mod hinanden – taget deres handlemuligheder i betragtning. F.eks. er de væsentligt bedre end implementerede FSM'er. Dog skal det siges, at FSM'erne er implementeret på relativt kort tid, og selvfølgelig vil kunne forbedres meget, hvis man investerede mere tid i at designe et system, der udnyttede samme funktioner, som de evolutionære agenter har til rådighed. De er naturligvis også meget bedre end tilfældigt genererede agenter. Spørgsmålet er naturligvis, hvor gode de er i forhold til fastdefinerede FSM'er, som har samme input og handlemuligheder til rådighed. Som nævnt tidligere, er der ikke noget håndfast bevis på hvordan de klarer denne situation, men vores umiddelbare erfaring siger os, at en spiludvikler ville kunne lave en FSM, der som minimum ville være lige så effektiv som de evolutionære agenter, der er trænet i forbindelse med dette projekt. Pointen er dog her, at dette skyldes, at det stadig her vil være muligt at overskue antallet af funktioner og handlemuligheder og konvertere dem til en fornuftig kombination af tilstande og transitioner. Man kan forestille sig, at når agenten får væsentligt flere af disse, at det så bliver svært, at gennemskue præcis hvilke sammenhænge, der vil føre til en god agent, samtidigt med at tidsforbruget til at hardcode hver enkelt FSM naturligvis også vil stige. Det skal siges, at de genetiske algoritmer selvsagt herved også vil få et større problem at optimere på, men som nævnt i forrige kapitel er genetiske algoritmer

netop er kendt for at være gode til at løse store komplekse optimeringsproblemer, og derfor kan man forvente, at der med en større population og flere generationer, vil kunne opnås en fornuftig løsning på det tilsvarende større problem.[60] I det følgende afsnit diskuteres nogle af de faldgruber der er ved det evolutionære agentsystem.

9.1 Faldgruber ved agentsystemet

I dette underafsnit diskuteres nogle af de faldgruber, der er ved at anvende og designe et evolutionært agentsystem. Som førnævnt, vil træerne blive større og større, jo flere handlemuligheder og især jo flere input de får. Hver gang man tilføjer et input (spørgsmål i træet), så tilføjer man også et niveau i træet og gør det dobbelt så stort. Når man tilføjer handlemuligheder bliver træet som sådan ikke større, men til gengæld bliver der flere værdier, som den genetiske algoritme skal vælge imellem. Derved bliver optimeringsproblemet sværere og sværere at løse, når man tilføjer flere muligheder for agenterne, som til gengæld så får mulighed for at få en mere kompleks adfærd. En idé, som allerede er benyttet i dette projekt, er så at opdele træet i undertræer, og på den måde gøre træerne mindre. En anden idé er at målrette søgningen yderligere med metoder som er mere elitære end de valgte. Faren ved dette er naturligvis at havne i et lokalt maksimum. For at mindske denne risiko, kunne man så øge populationen. En idé til at øge populationen kunne være at lave flere øer, men samtidigt ville flere øer kræve mere tid, hvilket igen giver en afvejning af tidsforbruget overfor hvor sikker man vil være på at finde en god løsning.

En anden faldgrube ved systemet er, at det er svært at teste hvordan træerne opfører sig i alle situationer. Netop det, at træerne er store og komplekse gør, at agenterne kan opføre sig godt i 99 procent af tiden, men i det sidste stykke tid, kommer agenten måske ind i en specielt dårlig del af træet, hvor dens adfærd er uhensigtsmæssig. Dette berører også en anden udfordring ved systemet – nemlig at det er utroligt svært manuelt at gå ind og rette i træet, hvis man ser, at en agent har opført sig på en utilsigtet måde, da det er svært at vide hvor i træet man skal rette og hvilke andre konsekvenser det eventuelt vil få. Her er man enten nødt til at acceptere, at agenten i den givne situation opfører sig uhensigtsmæssigt, eller også så lade agenterne træne videre, og håbe, at agenten lære at agere anderledes i den givne situation herefter. Men samtidigt er idéen med de genetiske algoritmer i dette projekt netop at de skal kunne danne beslutningstræer, som er så komplekse, at det som programmør alligevel ikke er muligt fuldstændigt at overskue disse. Endeligt er en af idéerne med at benytte et system som dette også, at den menneskelige spiller gerne må blive overrasket over hvordan NPC'en opfører sig i bestemte situationer, da dette kan være med til at give flere udfordringer for spilleren.[61]

En anden udfordring man har, når man designer agentsystemer som dette, er hvornår man skal stoppe træningen af agenterne. Det er svært at vide, hvornår agenterne vil opføre sig fornuftigt i (næsten) alle situationer; dette hænger naturligvis også sammen med, at det er svært at teste alle agentens mulige beslutningsudfald. En måde man kunne gøre det på, var at kigge på et antal generationer tilbage, og så se hvor meget fitness værdierne i gennemsnit var steget over de sidste x generationer. Når fitness værdierne så ikke steg længere, kunne man stoppe træningen. En fare herved, er naturligvis at ender i et lokalt maksimum, for optimeringsproblemet.

[60] Det at genetiske algoritmer netop er gode til at løse store og komplekse optimeringsproblemer, er bl.a. kendt fra **[TS]**.
[61] Fra anmeldelse **[GSKL]**

Endeligt kan det, at definere en fitness funktion, der kan optimere agenterne så de bliver sjovere at spille mod være svært. I dette projekt har fitness funktionen været defineret ud fra at agenterne skulle blive bedre til at dræbe deres modstandere, men man kunne forestille sig, at der også kunne være andre parametre, der kunne være med til at gøre agenterne sjovere at spille mod. Hvis systemet skal bruges i forbindelse med et kommercielt spil, er det i hvert fald vigtigt at være opmærksom på denne udfordring.

I forbindelse med metaheuristiske søgninger bør man normalt køre både parameter tuningen og den endelige test flere gange for at få et gennemsnit af disse til at vurdere hvor godt algoritmen fungerer til slut og for hver af de individuelle parameterindstillinger. Dette skyldes de forskellige steder i algoritmerne hvor tilfældigheder spiller ind på resultaterne. I dette projektet har det ikke været muligt at nå indenfor tidsrammen. Dette udgør potentielt en risiko for at nogle af resultaterne i denne opgave kan være misvisende. Igen er det en af de ting man i kommerciel sammenhæng vil være nødt til at afsætte tid til, således at der kan opnås større sikkerhed for de resultater som findes.

I løbet af dette projekt har det hele tiden været nødvendigt at begrænse opgaven til det der var realistisk. Selvom der i et kommercielt spil vil være bedre tid og flere ressourcer til AI udvikling, er det stadig nødvendigt at træffe valg og især fravalg som begrænser udviklingsprojektet til en realistisk størrelse. I den forbindelse er der imidlertid også en risiko for at træffe uheldige fravalg som gør at systemet ikke når sin potentielle højde.

I det følgende afsnit ses på hvilke forbedringsmuligheder, der er i forhold til at videreudvikle og forbedre agentstyringsmodellen.

9.2 Forbedringsforslag

I dette afsnit behandles nogle af de ideer, der er kommet frem under projektet, som det ikke har været muligt at afprøve pga. af den begrænsede tid, og for nogle af idéerne pga. af de begrænsede ressourcer.

En af de mest oplagte udvidelsesmuligheder for projektet ville naturligvis være at implementere de handlingsfunktioner, som det ikke var muligt at nå at få med. Her tænkes først og fremmest på valg af bevægelses metode (movement method) og valg af våben (weapon selection). Men også en handlingsfunktion til at bestemme hvorfra agenter skulle ankomme (arrive from). Foruden disse funktioner kunne man også tænke sig destinations- og skydefunktionerne udvidet med en række yderligere funktioner at vælge imellem, samt antallet af input.

9.2.1 Fitness afmålingen

Et af de problemer der er ved opgavens system er at evalueringen ikke kan undgås at være stokastisk. For at imødekomme dette problem kan man lave et system hvor agenterne under træningen får lov til at anvende de samme beslutningstræer i flere kampe inden der gås videre til næste generation, hvorefter det så er muligt at benytte en gennemsnits fitness værdi som udgangspunkt for agenternes sandsynligheder for at blive valgt til krydsning og erstatning. Dette ville give mere troværdige fitnessværdier og dermed formentligt også give en mere stabil forbedring fra generation til generation, men naturligvis også tage meget længere tid.

Alternativt kunne der muligvis udvikles et system som tager et gennemsnit over de seneste par generationers fitnessværdier, således at en enkelt dårlig kamp ikke ødelægger det for en agent som har været suverænt bedst de sidste mange kampe.

9.2.2 Krydsning

I forbindelse med de overvejelser, der blev gjort omkring valg af krydsningsmetode opstod en idé om at lave et krydsningssystem som lavede en analyse af hvilke gener, der havde ført de vindende individer til sejr. På den måde kunne man altså lave en intelligent form for *sexual cross-over*, hvor systemet identificerede gode gener og gennemførte krydsningen ved at sammensætte gode gener fra de førende individer, udvalgt således at generne komplementerede hinanden. Den store hindring for dette er naturligvis at genkende hvilke gener der er specielt gode og hvilke gener der vil komplementere hinanden godt.

En idé kunne være, at man med en eller anden form for statistik over hvilke stier i beslutningstræerne, der blev valgt i situationer hvor agenterne fik og mistede fitness points, kunne forsøge at identificere de gener der blev brugt, når agenten konkret fik fitness point og hvilke der blev brugt, når agenten mistede fitnesspoint. Muligvis kunne man lave et system, der gav de seneste x brugte stier fitness point, ud fra idéen om, at agenten er havnet i en given situation pga. en række tidligere valg og ikke blot det seneste. Man kunne så vælge en række gode stier fra én agent og herpå finde en anden agent, der havde en række af andre gode stier for at kombinere de to til en ny agent til næste generation. Præcist hvor effektiv denne form for krydsning vil være, er der naturligvis ingen garanti for, men er helt klart en af de muligheder, der ville kunne undersøges hvis man valgte at arbejde videre med agentstyringssystemet. Også fordi, at denne metode potentielt kunne være med til at målrette søgningen ved hurtigere at udvælge og kombinere gode gener, hvilket kan blive nødvendigt, når beslutningstræerne bliver større.

Et andet forslag til krydsning går ud på at lave en version af branch cross-over som er mindre tilfældig. Funktionen skulle vælge den første knude i træet på samme måde som i den nuværende version, men i stedet for at vælge en tilfældig sti ned gennem træet fra knuden, skulle den tage alle knuder under den først valgte, eller i hvert fald en større sammenhængende 'klump', således at der fås en funktion der udvælger et sammenhængende område, som har mere end en knude fra hvert niveau i træet.

9.2.3 Mutation

I forbindelse med mutationen blev det overvejet at lave et system som ud fra hamming distancen afgjorde, om der skulle anvendes guidet mutation eller om der skulle anvendes tilfældig mutation. Idéen hermed er, at det ikke er hensigtsmæssigt at anvende guidet mutation hvis individerne i en population på for tidligt et tidspunkt i evolutionen allerede minder meget om hinanden. Guidet mutation vil i det tilfælde blot guide det nye afkom hen mod de andre agenter i populationen, og agenterne vil dermed formentlig blive ens og konvergere for tidligt. Dette kunne have den konsekvens, at den genetiske algoritme havner i et lokalt maksimum. Hvis individerne er 'for ens' kunne man i stedet vælge en tilfældig mutation, eller måske endda en mutation der guidede afkommet væk fra den tidligere generation.

Det er også muligt at agenternes performance kan forbedres yderligere ved at implementere en geometrisk mutations funktion. Funktionen skulle mutere et helt område af træet, i stedet for at

mutere på tilfældige gener i kromosomet. Denne metode kunne naturligvis også kombineres med guidet mutation, således at f.eks. en hel gren i træet blev muteret hen mod det foreløbige bedste individs tilsvarende værdier.

9.2.4 Parametre

I løbet af projektet har der også været idéer omkring at lave en mere omfattende parametertuningsalgoritme. Der er ingen tvivl om, at der ikke er tid til at gennemgå alle parameterkombinationer, som ellers er oplagt, når man har et mindre problem. I dette projekt var det endda nødvendigt at lave en meget snæversynet grådig algoritme til at parametertune med. I fremtidigt arbejde ville det være en mulighed at lave en grådig algoritme, som afprøvede flere forskellige værdier for hver enkelt parameter (f.eks. med bisektion eller en anden metode som prøver sig frem) og således sporede sig ind på mere præcise parameterindstillinger for på den måde at opnå et bedre endeligt resultat. Problemet er naturligvis, at holde tiden, det tager at gennemføre parametertuning nede, og metoden skal derfor helst konvergere hurtigt, hvorfor den grådige tilgang også er valgt i forbindelse med dette projekt. En anden simpel udvidelse kunne være, at en grådig algoritme blev ved med at prøve at øge parameterværdierne en ad gangen, indtil performance blev dårligere, og så brugte man den værdi, der blev afprøvet umiddelbart før performancen faldt.

9.3 Andre idéer

Et af de kritikpunkter man ofte hører i forhold til at overlade styringen til genetiske algoritmer er at man delvist mister kontrollen og derved risikerer at agenterne i helt særlige situationen kan komme til at opføre sig uintelligent. Det vil som nævnt under faldgrubber også være muligt med den model der arbejdes med i dette projekt, da programmøren ikke har fuldstændig overblik over hvordan det komplette beslutningstræ ser ud. En mulig løsning på problemet kunne i givent fald være at lade agenterne fortsætte træningen mod menneskelige spillere, således at hvis flere brugere fandt en uhensigtsmæssighed i form af en bestemt simpel taktik til at snyde agenterne og vinde over dem, så ville agenterne blive straffet gennem deres fitnessværdier. Man kunne forestille sig, at et spilfirma indbød (eller betalte for) en masse spillere til at teste deres nye AI. På den måde ville uhensigtsmæssigheder som udgangspunkt blive rettet i tilfælde af at flere brugere stødte på dem og udnyttede dem.

En anden idé, i forbindelse med træningen af agenterne kunne være at fastlåse modstander holdet i et antal generationer i stedet for at ændre begge hold mellem hver generation. F.eks. kunne predator agenterne træne mod fastlåste prey agenter i fem generationer og derefter kunne de bytte så det var predator agenterne der var fastlåste. Dette ville give en bedre mulighed for reelt at vide hvor meget bedre næste generation af agenter var. Man kunne også forestille sig, at det ville give agenterne mulighed for bedre at nå at adaptere sig til modstander holdets strategi.

For at lave en egentlig vurdering af om agenterne forøger spilleglæden kan man f.eks. lave en brugertest. Man kunne tænke sig, at når man havde udvidet systemet med flere handlemuligheder for agenterne, at der så med fordel kunne laves en brugertest af agentsystemet, for at teste om agenterne rent faktisk bliver sværere og/eller sjovere at spille mod, end agenter styret af FSM'er. En sådan test kunne ville give et fornuftigt bud på om systemet kunne blive en succes i et rigtigt spil.

Kapitel 10
Konklusion

Her samles der op på hvad projektet har beskæftiget sig med og hvilket potentiale projektet har i forhold til fremtidig kunstig intelligens i spil.

10 Konklusion

Efter at have arbejdet med genetiske algoritmer i sammenhæng med agenter i spil, er der nu opnået indblik i teknikkens potentiale og de muligheder, der er for fremtidigt arbejde, samt nogle af de problemer, som kan opstå i forbindelse med at vælge netop denne teknologi til at styre agenterne med. I dette kapitel gives der afslutningsvis en samlet vurdering af opgavens resultater og hvilken konklusion, der kan drages ud fra disse. Først skal det dog siges, at en stor del af opgaven har bestået i at prioritere og vælge mellem de utallige muligheder for kombinationer der har været mulighed for at implementere. Dette gælder både i forhold til agentens input og handlemuligheder, men i endnu større grad i forbindelse med design af den genetiske algoritme og testen af samme.

Agentsystemet er udviklet som en udvidelse til game enginen Torque og har derfor til en hvis grad også været underlagt Torques præmisser. Her tænkes især på de muligheder agenterne har haft til rådighed fra start. I relation til styringen af agenterne har Torque faktisk ikke leveret andre funktioner end health level, ammunitions level, raycasts, skydefunktion og positioner på objekter. Det har derfor været nødvendigt at investere en del tid i at udbygge enginen, så agenterne har fået mulighed for dels at få information om deres miljø, og dels at udføre forskellige handlinger. Denne proces har ikke altid været helt ligetil, da Torque er stor, kompleks, og ikke er helt fri for bugs. Når det er sagt, har enginen heldigvis et fornuftigt forum med dokumentation og andre brugeres tips, hvilket har været en stor hjælp i udviklingsfasen. En anden af de store udviklingsmæssige udfordringer i projektet har været at få optimeret koden, så evaluerings processen kunne speedes op.

Formålet projektet har været at lave et agentstyringssystem, som kunne vise potentialet ved at kombinere to yderpunkter indenfor kunstig intelligens, i form af genetiske algoritmer og beslutningstræer, og derved implementere et system som *proof of concept* for metoden. Der har undervejs været forskellige udfordringer med f.eks. at kunne træne træerne hurtigt nok, når træerne blev store og at få lavet en passende struktur på træerne, så både størrelsen og kørslen af dem blev realistisk at arbejde med på en computer. Løsningen på dette problem har dels været at dele træerne op i et *top-level* træ, der beslutter en overordnet strategi og fire undertræer, der repræsenterer hver af de fire strategier. Dette har været med til drastisk at reducere størrelsen på beslutningstræet og dermed også søgerummet. Samtidigt har det den gode egenskab, at der samlet skal spørges om færre spørgsmål under spillets afvikling, når agenten skal træffe en beslutning, fordi der efter valg af strategi, f.eks. 'forsvar dig', så ikke skal spørges om forhold, der har med angrebsstrategien at gøre. Dette er med til at gøre at kørslen af træet (herunder udregningen af inputtene) kan udføres hurtigere end ellers. Det skal dog siges, at der ved at opdele træerne i undertræer naturligvis fjernes nogle informationer og kombinationer, som ellers ville havde været mulige. En anden del af løsningen på det kombinatoriske problem har været at prøve at guide optimeringen forholdsvis meget. Det har været nødvendigt dels fordi træerne kan blive store og dels fordi evalueringen af agenterne tager lang tid. For at guide søgningen, er der implementeret en geometrisk krydsning og en guidet mutation. Det har siden vist sig, at netop denne kombination også er den, der har givet de bedste resultater i optimeringen af træerne, af de afprøvede algoritmer.

En anden udfordring har været at designe systemet således, at det var muligt at have en fornuftig stor population. Spil universet gør, at det ikke umiddelbart er hensigtsmæssigt at have mere end otte agenter på hvert hold på banen ad gangen. Løsningen har været at dele populationen ind i øer, hvorved det er gjort muligt, at lade to hold være én ø og så have flere sæt af hold i samme population. På den måde kan populationen gøres vilkårligt stor, men til gengæld kommer evalueringen af hver generation til at tage længere tid, da dette reelt betyder, at hver ø skal

evalueres hver for sig. Selve ø-strukturen giver dog også nogle andre fordele – f.eks. at det er mere usandsynligt, at søgningen ikke bliver alt for styret af én type agent. Det er mere sandsynligt, at hver ø vil bevæge sig i sin egen retning – hvilket er smart i et multikriteriums problem, hvor det ikke er sikkert, at der kun er én rigtig løsning, men formentligt snarere er en masse forskellige løsninger, der er gode på hver sin måde.

Desuden har parameter tuning af de fire testede algoritmekombinationer selvfølgelig haft fokus, da det er vigtigt for både sammenligningen af algoritmerne, men selvfølgelig også agenternes udvikling, at algoritmerne har de bedst mulige betingelser for at optimere systemet. Da der er mange parametre til algoritmerne og evalueringstiden af agenterne samtidigt er høj, er det ikke muligt at teste alle kombinationer af parametre, hvorfor der er implementeret en parametertunings algoritme, der arbejder med en grådig tilgang, hvilket gør, at den 'forholdsvist' hurtigt (på ca. 10 timer) giver en indikation på hvordan parametrene kan indstilles.

Efter denne parametertuning er algoritmerne så blevet testet i den endelige træning. Hver algoritme har fået lov til at træne en population af agenter i 600 generationer, hvorefter agenterne har indgået i en turnering, hvor de forskellige algoritmers agenter har dystet mod hinanden. Her har det vist sig, at algoritmen med geometrisk krydsning og guidet mutation har fungeret bedst. Endvidere har disse agenter også spillet mod hold af agenter fra et tidligt stadie i evolutionen, hvor de mest trænede agenter vandt overlegent. Alt dette er med til at bekræfte at optimeringen af agenterne har været en succes. Et andet spørgsmål er så, hvor gode disse agenter er i forhold til mere avancerede FSM'er, der udnytter samme input og handlemuligheder, som de evolutionære agenter. Det er svært at sige noget om, da det ikke har været muligt at lave en objektiv sammenligning, men vores umiddelbare bud er, at en spiludvikler vil kunne designe et system, som kan det samme som de evolutionære agenter, der er trænet i dette projekt kan.

Dette leder os så videre til om systemet kan udvides i tilstrækkelig grad til at agenterne får så mange input og handlemuligheder, at en spiludvikler reelt vil have svært ved designe et tilsvarende effektivt system med en FSM. Implementeringsmæssigt burde der ikke være de store problemer i det, da træstrukturen allerede er på plads, og det derfor blot handler om at tilføje de funktioner, der skal udregne input og udføre handlinger for hver agent. Spørgsmålet er så om de genetiske algoritmer kan håndtere at skulle optimere på dette større problem. Det kan naturligvis ikke siges med sikkerhed, men det er meget sandsynligt, at det i hvert fald vil kræve en længere træningsperiode og sikkert også en større population, da søgerummet bliver større. Potentielt kunne det sikkert betale sig at forsøge at målrette søgning endnu mere, f.eks. med sexual crossover og guidet geometrisk mutation for på den måde at komme de kombinatoriske problemer i forkøbet.

Efter at have implementeret det evolutionære agentsystem og have arbejdet med at optimere på det, står det klart at målet er nået. – Det er lykkedes at lave et system, hvor agenterne optimeres uden at en programmør skal definere agentens adfærd. Det konkluderes, at systemet har potentiale til at blive rigtigt godt, hvis input og handlemuligheder bliver så mange, at agentens adfærd bliver for kompleks til at en spiludvikler reelt kan designe et godt system. Det kræver naturligvis, at den genetiske algoritme er succesrig i dens søgning i dette nye større søgerum. Derfor vurderes det at opgaven kan bruges som et *proof of concept* for brugbarheden af kombinationen af en genetisk algoritme og beslutningstræer til agentstyring.

11 Litteratur liste

11.1 Bøger

[A*] "Artificial Intelligence: A New Synthesis", I.S.ed edition, 1998, af Nils Nilsson, Morgan Kaufmann Publishers

[AGP1] AI Game Programming Wisdom 1, Steve Rabin

[AGP2] AI Game Programming Wisdom 2, Steve Rabin

[AI] "Artificial Intelligence - A Modern Approach", Second Edition, Stuart Russell & Peter Norvig, Prentice Hall

[CC] Co-evolutionary Constraint Satisfaction, Jan Paredis.

[CI] Computational Intelligence **–** An Introduction, Andries P. Engelbrecht, WILEY

[SM] Search Methodologies – Introductory Toturials in Optimization and Decision Support Techniques, Edmund K. Burke & Graham Kendall

11.2 Artikler og hjemmesider

[AI-DEPOT] Artificial Intelligence Depot
http://ai-depot.com/FiniteStateMachines/FSM-Background.html

[AI-DEPOT2] Artificial Intelligence Depot
http://ai-depot.com/GameAI/Agent-Intelligence.html

[AIGAMEDEV] Game AI for Developers
http://aigamedev.com/discussion/evolutionary-algorithms-suitable

[AIGAMEDEV2] Game AI for Developers
http://aigamedev.com/reviews/top-ai-games

[AGD] AI Game Development: Synthetic Creatures With Learning And Reactive Behaviors
http://www.asigner.com/Books/new.riders.ai.game.development.synthetic.creatures.with.learning.and.reactive.behaviors/index.html?page=1592730043_ch35lev1sec3.html

[AN] A in Computer Games, af Alexander Nareyek, Guest Researcher, Carnegie Mellon University
http://www.4c.ucc.ie/web/upload/publications/article/nareyek-acmqueue04.pdf

[CS] Definition på sproggenkender (Finite Automaton)
http://www.cs.gsu.edu/~cscskp/Automata/FA/node4.html

[DIE] die.net

http://dict.die.net/finite%20state%20machine/

[DT] "Decision Trees - A Primer for Decision-making Professionals"
http://www.projectsphinx.com/decision_trees/index.html

[decMutRate] "Adaptive mutation rate control schemes in genetic algorithms", Dirk Thierens
www.cs.uu.nl/research/techreps/repo/CS-2002/2002-056.ps.gz

[ECC] Evolutionary Computation Comments on the History and Current State, Thomas Bäck, Ulrich Hammel, and Hans-Paul Schwefel

[ER] Evolutionary Robotics - The Use of Artificial Evolution in Robotics - A tutorial presented at IROS 2004, Mattias Wahde
http://www.me.chalmers.se/~mwahde/robotics/TechReports/TR-BBR-2004-001.pdf

[GAMEAI] The Game AI Page: Building Artificial Intelligence into Games
http://www.gameai.com/ai.html

[GarageGames] "Garage games homepage**"**
www.garagegames.com

[GIGNEWS] Get In the Game - Using Genetic Algorithms for Game AI, af Greg James
http://www.gignews.com/gregjames1.htm

[GS] GameSpot
http://www.gamespot.com/

[GSKL] Anmeldelse på Gamespot af "Kane and Lynch"
http://www.gamespot.com/users/LordAndrew/video_player?id=IHwzwTut5bwNvjXc

[McRae] "http://www.generation5.org/content/2001/hannan.asp", interview med Jeff Hannan

[MIT] Three States and a Plan: The A.I. of F.E.A.R. - Monolith Productions / M.I.T. Media Lab, Cognitive Machines Group, af Jeff Orkin.
http://www.cs.ualberta.ca/~bulitko/F06/papers/gdc2006_orkin_jeff_fear.pdf

[MIT2] Jeff Orkin - Curriculum Vitae
http://web.media.mit.edu/~jorkin/

[PB] Population Based Incremental Learning with Guided Mutation Versus Genetic algorithms Iterated Prisoners Dilemma, Timothy Gosling, Nanlin Jin and Edward Tsang.
http://cswww.essex.ac.uk/research/CSP/finance/papers/GosJinTsa-Pbil_vs_Ga-Cec2005.pdf

[Rambøll] Rambøll management
http://www.ramboll-management.com/dan/sites/ee_experience_economy/it+og+medier/default.htm

[SEPAM] "A Study of Execution Plan Aware Mutations for Genetic Cyclic Query Optimization", Victor Muntés, Josep Aguilar Saborit, Josep Lluis Larriba Pey y Calisto Zuzarte, 2004

http://www.dama.upc.edu/public/mutation04.pdf

[Sestoft] ”Systematic software test, af Peter Sestoft, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark”
http://www.itu.dk/people/sestoft/programmering/struktur.pdf

[TDN] ”Torque Developer Network”
http://tdn.garagegames.com/wiki/Torque_Game_Engine

[TorqueDoc1] “Engine coding in C++ part IV. Engine overview”
www.garagegames.com/docs/tge/general/ch06.php

[TorqueDoc2] ”TorqueScript Part III, Torque Script”
www.garagegames.com/docs/tge/general/ch05.php

[UCC] Noter til GA forelæsning ved “Department of Computer Science at University College Cork” http://www.cs.ucc.ie/~dgb/courses/tai/notes/handout12.pdf

11.3 Projekter og slides fra foredrag

[PMP] Polyteknisk midtvejs projekt ”Grundlæggende AI i computer game engines”, 2006, af Casper la Cour & Mads Terp

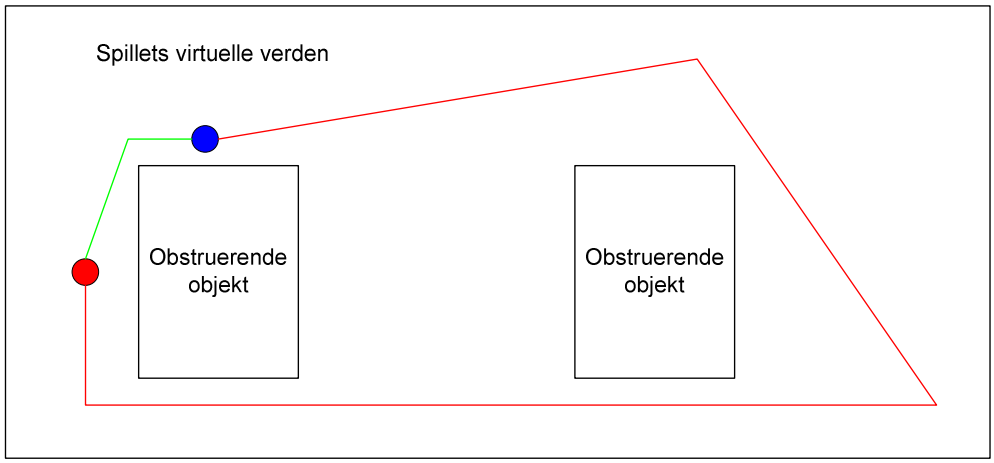
[Tisher] Semantic Memory Algorithm - A Standardized Memory Algorithm for Computer Game Agents, André Tischer, 2007
ftp://ftp.diku.dk/diku/image/publications/tischer.060801.pdf€

[TS] Sildes fra Thomas Stidsens forelæsning om evolutionære algoritmer.
http://www2.imm.dtu.dk/courses/02719/evo/evo.pdf

12 Appendiks

12.1 Pathfinding

Det skal her siges, at implementeringen af A* ikke er foretaget i forbindelse med dette projekt, men genbrugt fra et tidligere projekt. I forbindelse med FPS spil, er en af de fundamentale og væsentlige metoder agenten har brug for, en måde at bevæge sig på, i den virtuelle verden. Efterfølgende kaldes denne problemstilling for pathfinding. For at en NPC skal virke overbevisende, på en menneskelig computerspiller, skal NPC'en kunne bevæge sig rundt i den verden spillet foregår i, på en realistisk måde. Med dette menes, at NPC'en skal kunne bevæge sig, uden at løbe ind i andre objekter og NPC'en skal vælge en sti, der får den til at *se intelligent* ud, for den menneskelige spiller. En sti, der får agenten til at se intelligent ud, kan vælges ud fra flere forskellige forudsætninger, som bliver diskuteret i næste afsnit. Et kriterium er dog gældende for alle intelligente stier er dog, at agenten ikke vælger åbenbare omveje. Denne situation er illustreret på Figur 31 og farvekoderne er beskrevet i figurteksten.



Figur 31: Illustration af valg af en 'intelligent' sti. Blåt punkt: Her starter agenten. Rødt punkt: Her vil agenten hen. Grøn linie: En 'intelligent' sti. Rød linie: En sti, der gør at agenten ikke ser 'intelligent' ud.

Én kunne være, at agenten skulle forsøge at efterligne den menneskelige måde at navigere på, og vælge stier herefter. En anden måde er, at agenten skal vælge den kortest mulige sti, agenten har til rådighed på det givne tidspunkt. Denne optimale sti, adskiller sig fra den menneskelige måde at navigere på, ved at den bruger al information om verdenen, der påvirker sti valget, hvorimod den menneskelige måde at finde vej på, ikke har information om uudforskede områder til rådighed.

Fordelen ved at simulere den menneskelige måde at navigere på er, at den menneskelige computerspiller bruger samme måde at finde rundt på, og derfor opfatter agentens måde at finde rundt på, som værende på niveau med sin egen (hvis den implementeres på en overbevisende måde). Ulempen kan være, at agenten muligvis bruger længere tid på, at komme hen til det sted den skal – og dermed måske kommer for sent til en del af handlingen i spillet – f.eks. kunne det være agenten skulle hjælpe den menneskelige spiller, med at nedkæmpe fjender, men ikke nåede derhen før den menneskelige computerspillers karakter var død, fordi pathfindingen var for dårlig. En anden ulempe ved denne metode, er at det er sværere at modellere en agent, der skal simulere den menneskelige måde at navigere på, da mange parametre her er med til at afgøre, hvilken sti der

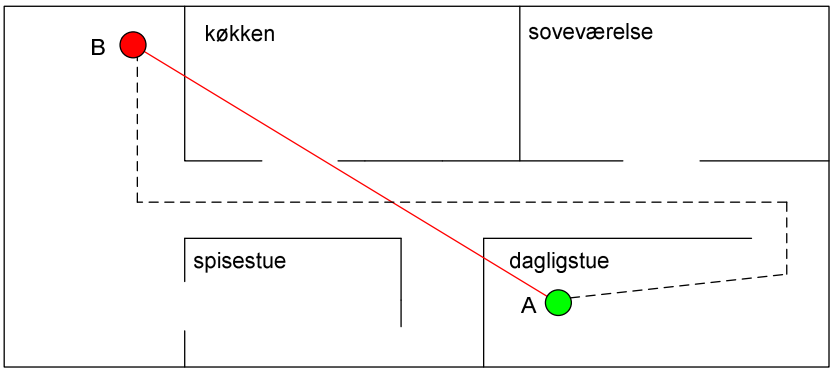
bliver valgt. Bl.a. skal det tages i betragtning, om man har fulgt stien før, om man har kendskab til lokalområdet, man vil bevæge sig ind i, om man har kendskab til forskellige ruters længde, m.fl. Herimod er det kun rutens længde, der afgør valget, når man benytter korteste sti til at finde vej, hvilket gør det lettere at modellere en løsning til problemet. Ulempen ved korteste sti metoden er omvendt, er agenten kan komme til at virke 'for klog', da den altid vælger den korteste (optimale) sti. Der er altså fordele og ulemper ved begge tilgange, til pathfindings problemet. I dette projekt er korteste sti metoden valgt, ud fra den vurdering, at det ikke reelt er et problem, at agenten bliver bedre til at finde rundt, end den gennemsnitlige menneskelige spiller. I FPS multiplayer spil, spilles samme bane ofte mange gange, og den menneskelige spiller lærer derved også banen at kende, og kan derved navigerer ligeså godt (kender også de korteste stier) som NPC'en.

12.1.1 Overgang fra spil til søgealgoritme

Når man ønsker at finde den korteste sti, fra et sted i den virtuelle verden til et andet, er det nødvendigt, at opdele spillet i områder, eller positioner, som agenten kan gå til og fra. For at illustrere dette tages først udgangspunkt i en 2d verden, hvorefter en illustration af de problemer der tilføjes ved at befinde sig i en 3d verden illustreres. Før forklaringen uddybes, skal det dog nævnes, at agenter i dette projekt kun har en mulighed for at bevæge sig mellem to givne positioner – nemlig 'lige ud' ved at gå den direkte vej.

12.1.1.1 Agent i en 2D verden

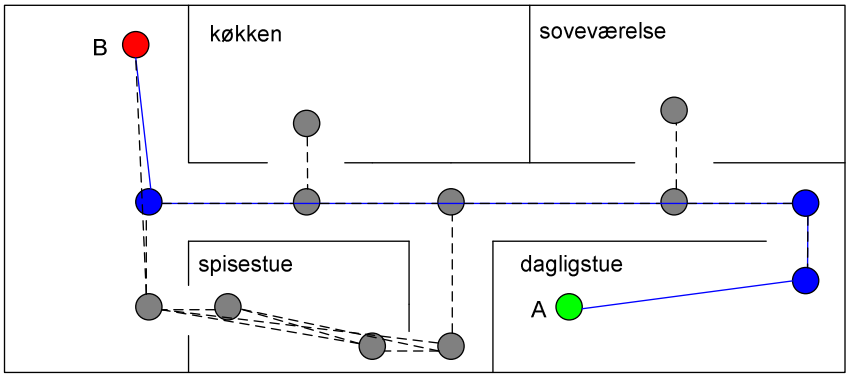
Hvis agenten skal kunne finde rundt i f.eks. et hus i et spil, er den nødt til at vide, hvor dørene er, og hvor der ligger f.eks. møbler og andet i vejen, som den er nødt til at gå udenom. Derfor giver det god mening at opdele huset i positioner, ud fra hvor der er objekter i vejen for agentens sti, ved at se på hvilke positioner agenten kan gå mellem i en lige linie. I Figur 32 ses et eksempel på et udsnit af et hus set oppefra. De sorte streger angiver vægge, som agenten ikke kan gå igennem. Agenten befinder sig nu på position A, og ønsker at bevæge sig fra A til B, uden at støde ind i en væg, og for så vidt muligt, uden at gå en omvej fra A til B.



Figur 32: Viser et hus i et spil set oppefra hvor en agent ønsker at finde fra A til B.

Hvis ikke agenten har nogen ide om, at den er nødt til at gå via et antal positioner for at komme fra A til B, vil den gå direkte via den røde linie, og støde ind i væggen. Agenten er derfor nødt til først, at gå lidt længere væk fra B, for at komme ud af dagligstuen, og herefter videre ad den stiplede linie, forbi køkken og spisestue, indtil den til sidst, kan se punktet B, og gå direkte derhen. For at

agenten kan gå via den stiplede linie, er agenten nødt til at vide, hvornår den skal skifte retning. Agenten er nødt til at have sådanne waypoints, således, at den, ved at følge en sekvens af forskellige waypoints, kan komme fra et hvilken som helst rum, til et hvilket som helst andet rum. Indeni hvert rum, skal agenten kunne se alle positioner i rummet, så den kan gå i en lige linie, fra waypointet derhen uden at støde ind i noget. Hvis der står møbler i vejen, som skygger for udsynet, skal der indlægges flere waypoints i lokalet, således at agenten, ved at gå den direkte vej mellem waypoints'ne, kan komme til samtlige positioner i lokalet (der er fysisk mulige). Det sker ved at det for enhver position i spillets virtuelle verden skal være muligt at gå til mindst et waypoint. På Figur 33, ses et eksempel på, hvordan disse waypoints kunne være indlagt, således at de opfylder nævnte betingelser.



Figur 33: De grå ringe er indlagte waypoints som agenten kan bevæge sig i mellem.

Waypoints'ne er placeret således, at uanset hvor i huset A og B placeres, kan agenten gå i en lige linie, mellem mindst ét af waypoints'ne og A, og ligeledes mellem mindst ét af waypoints'ne og B, uden at støde ind i en væg. Samtidigt, kan agenten gå mellem waypoints'ne, og komme fra ethvert waypoint til ethvert andet kun ved at gå mellem waypoints, uden at gå ind i vægge. Den vej, som agenten skal gå, er her indtegnet med blå, og de waypoints agenten skal gå mellem er også farvet blå. De stiplede linier angiver de waypoints, som agenten kan gå direkte mellem (de findes også bag den blå linie).

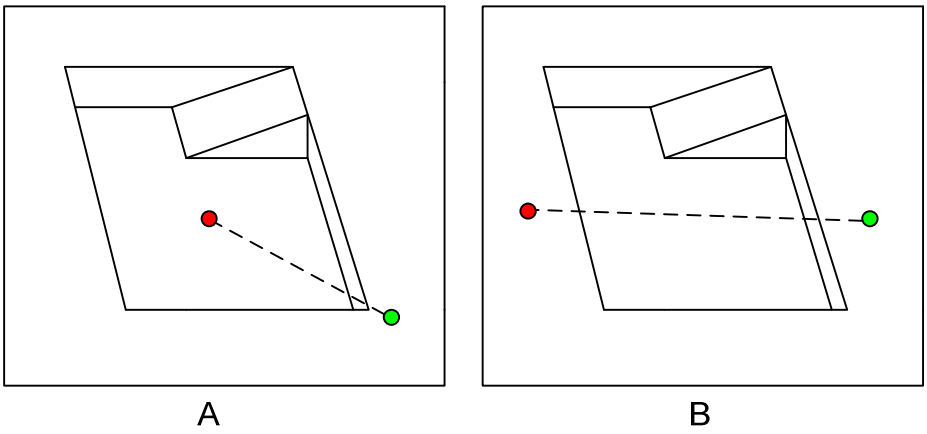
Pathfindings problemet mellem A til B, kan nu opstilles, som et korteste sti problem i en graf. Waypoints'ne og A og B, kan opfattes som punkter i grafen, og de linier mellem punkterne, som agenten kan gå efter, opfattes som kanter, hvor afstanden mellem punkterne er kanternes vægte, fordi det er den afstand, som agenten skal bevæge sig, for at komme fra det ene waypoint, til det andet. Dette kræver naturligvis, at, som minimum, punkterne A og B kan tilføjes til grafen dynamisk, så agenten under spillet kan bestemme sig for at gå mellem en hver position på banen og enhver anden og ikke er tvunget til at starte og slutte i et punkt der fra start er på grafen – agenten befinder sig jo næsten aldrig på præcis samme sted som et punkt i grafen. Selve punkterne er ikke noget problem at tilføje til grafen dynamisk – derimod kan det volde problemer at finde frem til hvilke kanter der går til og fra de dynamisk tilføjede punkter. For at kunne gøre dette er agenten nødt til at have nogle censorer der kan fortælle hvorvidt det er muligt at gå fra et punkt til et andet i en lige linie. Mere om løsningen på dette senere.

Problemet gøres altså til et generelt korteste sti problem. Til løsning af dette problem, findes en række kendte algoritmer, som agenten kan benytte, og i dette tilfælde er det oplagt at benytte

algoritmen A* (mere om dette senere). Måden waypoints'ne samt start og slutpunkterne på agenternes stier lægges ud på i projektets konkrete bane beskrives i et senere afsnit.

12.1.1.2 Agent i en 3D verden

Da agenten i dette projekt befinder sig i en 3D verden kompliceres pathfindingen dog væsentligt. Problemet løses stadig som et korteste sti problem og selve algoritmen til løsningen af dette problem kompliceres ikke af udvidelsen, og det gør graf repræsentationen heller ikke. Det er det at danne den rigtige graf der kompliceres. Komplikationen er især til stede, hvis man gerne vil kunne danne grafen automatisk ved at bruge agentens censorer til at detektere hvordan verden ser ud. I dette afsnit beskrives de problemer der er ved at danne en 'korrekt' graf i forbindelse med en 3D verden. Først og fremmest er der to grundlæggende ting der skal tages højde for, illustreret på figuren nedenfor.



Figur 34: Illustrerer to generelle problemer ved pathfinding i 3 dimensioner

På figuren vil agenten gerne gå mellem det grønne punkt og det røde punkt. På A skal det være muligt at gå fra det grønne punkt til det røde, men ikke fra det røde til det grønne, da agenten kan hoppe ned fra kanten, men ikke kravle opad samme. På B skal agenten hverken kunne gå fra det grønne til det røde punkt eller omvendt, da agenten ikke kan kravle opad væggen. Problemet er nu, at der ikke er objekter i vejen for agenten i en direkte linie mellem de to punkter, hvorfor agenten kunne foranlediges til at tro, at den kunne gå mellem punkterne.

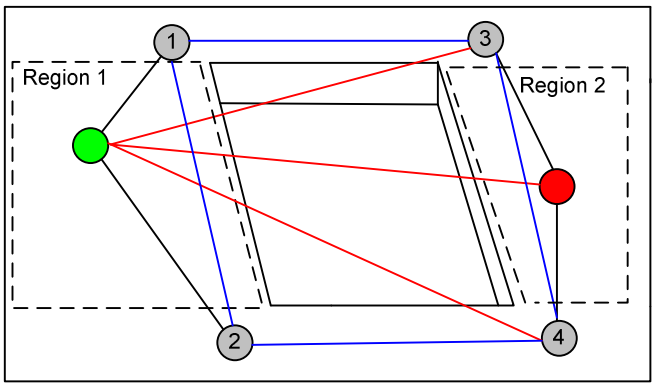
12.1.1.3 Domæne specifikke problemer

Pga. de censorer agenten har (og dermed de muligheder for at opleve den virtuelle verden) opstår der lang række problemer, når start og slutpunktet skal tilføjes til grafen. Fordi raycast'et er eneste reelle mulighed for at tjekke om der er en kant mellem to punkter gøres processen ekstra besværlig. Et af de problem mange problemer der kan opstå er, at agenten raycaster mellem to punkter i en given højde, men lige over eller under raycastet hænger der en snor. Raycastet finder ingen objekter mellem de to punkter, men når agenten forsøger at løbe mellem de to punkter, vil han løbe ind i snoren. For at løse det problem med raycasts ville agenten skulle kaste raycasts i en slags tunnel mellem de to punkter, så den var sikker på, at der ikke var et eneste sted på stien mellem punkterne, hvor der var noget i vejen. Problemerne ved Figur 34 ville også være meget svære at løse kun ved

brug af raycasts. Faktisk er dette problem et meget stort projekt i sig selv, hvorfor vi har valgt ikke at danne grafen automatisk, men i stedet manuelt lave en graf, der tager højde for disse problemer.

12.1.1.4 Generering af graf

Som nævnt i ovenstående afsnit dannes grafen ud fra punkter der manuelt er lagt på banen. Dertil har vi implementeret et system hvor det er muligt at placere punkter på banen og tilknytte nabo punkter til dette objekt. Da punkterne lægges manuelt på banen (af os) har vi lavet en begrænsning, der gør, at punkterne højst kan have otte naboer, for at begrænse arbejdet med at lave grafen. Samtidigt er nabo punkterne valgt således at det er muligt at komme ud i forskellige retninger i forhold til punktet på banen og på den måde sikres det at mulighederne for at komme rundt på banen er næsten lige så gode, som hvis punkterne havde alle punkter man kunne gå mellem som nabo punkter. På denne måde kan grafen konstrueres fuldstændigt som ønskes – kun med kanter mellem de punkter, som man rent faktisk kan gå mellem. Der er dog stadig et problem med start- og slut punktet, som skal lægges på grafen dynamisk, og derfor ikke kan tilføjes manuelt af programmøren. I stedet benyttes et system, der definerer hvilke punkter der sikkert kan tilføjes som nabopunkter i et fastdefineret område. På den måde deles banen op i regioner, hvor bestemte punkter er tilknyttet disse specifikke regioner. De punkter der er tilknyttet de forskellige regioner er tilknyttet ud fra en manuel betragtning af programmøren, på en måde så ligegyldigt hvor man står i regionen, så kan man altid gå i en lige linie hen til de fast definerede punkter, se illustrationen nedenfor.



Figur 35: Illustrerer hvordan punkter bliver tilføjet til grafen dynamisk ud fra den region punktet befinder sig i

På figuren ønskes det, at tilføje det grønne og det røde punkt til grafen. Den eksisterende graf består her af punkterne 1-4 og de blå kanter. Agenten befinder sig i det grønne punkt (startpunktet) og vil gerne til det røde punkt (slutpunktet) – dette skal selvfølgelig ske uden at agenten sidder fast i hullet i midten. Det grønne punkt befinder sig i region 1 der er markeret med stiplet linier – og til denne region er punkt nummer 1 og 2 tilknyttet (markeret med sorte linier). Dvs. at det grønne startpunkt kan tilføje punkt nummer 1 og 2 til sin nabo liste, hvorimod punkterne 3 og 4 ikke er tilknyttet regionen og derfor ikke bliver naboer til det netop tilføjede punkt i grafen. På samme måde bliver punkterne 3 og 4 naboer til slutpunktet. Det betyder så, at når agenten vil fra start til slutpunkt, så vil han først gå fra startpunkt til punkt 1, så til punkt 2 og herfra til slutpunktet. Hele banen vil være dækket af regioner, der har mindst 1 punkt i grafen tilknyttet, således at det altid vil være muligt at til og fra enhver region (og dermed enhver position på banen). Endvidere skal det nævnes, at regionen selvfølgelig skal defineres i alle tre dimensioner, således at hvis der er flere etager i en bygning (eller broer), så vil z koordinaten afgøre hvilken region man er i, og dermed hvilke punkter

der er nabo punkter til det punkt der skal tilføjes grafen. Til at finde ud af hvilke punkter i grafen agenten så skal gå mellem for at vælge den korteste sti, bruges algoritmen A*.

12.1.2 A* (A stjerne)

A* adskiller sig fra de andre korteste sti algoritmer, ved at den benytter en viden om søgerummet til at optimere søgningen. Derfor kaldes den en 'informeret søgealgoritme'. Denne information opnås vha. en heuristik funktion (h-funktionen), som er indeholdt i A*. En heuristik funktion er en funktion, der estimerer hvor stor omkostning er, forbundet med at komme til en ønskelig situation, fra en given situation. I A* er heuristikken et bud på, hvor langt der er fra den knude man betragter, til målet. Dette estimat skal være optimistisk, forstået på den måde, at det aldrig må overvurdere afstanden til målet (det forklares hvorfor senere).

A* benytter desuden af en *g*-funktion og en *f*-funktion. *g*-funktionen angiver, hvor langt der er fra startpunktet til et givet punkt, - eller mere præcist længden af den korteste kendte sti, til det punkt der betragtes, på et givent tidspunkt i afvikling af algoritmen. *f*-funktionen er *h*- og *g*-funktionernes værdier lagt sammen, og angiver dermed hvor langt der mindst er til målet, via den pågældende knude. Disse tre nøgle-værdier afgør hvorledes A* forløber, og når A* er afviklet, haves en korteste sti fra startpunktet til slutpunktet, givet en sti findes (argumentationen for dette kommer senere).

12.1.2.1 En kort gennemgang af hvorledes A* fungerer.

A* fungerer ved at kigge ud fra startpunktet til alle de punkter, der kan nås herfra og lægge dem på en liste der kaldes 'åben-listen', som indeholder alle de punkter der skal gennemgås. De forskellige kanter til nabopunkterne har en vægt som bruges til at sætte deres *g*-værdi med, fordi det er den omkostning, der er ved at gå fra startpunktet og ud til hver af dem. Herefter sættes punkterne til at have startpunktet som forælder, fordi de er nået via startpunktet. Der foretages herefter en beregning af, hvor langt der mindst er fra hvert af startpunktets nabopunkter, til målet, vha. heuristik funktionen. Denne beregning gemmes i punkternes *h*-værdi. Endeligt udregnes punkternes *f*-værdi, som angiver, hvor langt der mindst er til målet, via netop denne knude. Det gøres ved at lægge punktets *h*- og *g*-værdi sammen, og startpunktet lægges herefter på en liste som kaldes lukket-listen, som angiver, at algoritmen er færdig med at kigge på dette punkt. Herefter fortsætter A* iterativt med at vælge det punkt med laveste *f*-værdi fra åben-listen, og kigge ud fra dette punkt, til punktets nabo punkter og tilføje dem til åben-listen. Nabopunkternes *g*-værdier bliver herefter sat til kantvægten mellem nabopunktet og det punkt, der kigges ud fra, lagt sammen med *g*-værdien for punktet, der kigges ud fra. Dette sker dog ikke hvis det givne nabopunkt allerede har en *g*-værdi, som er mindre end den nye. Hvis punktet i forvejen har en større *g*-værdi overskrives både *g*-værdi og forælder statusen, da dette betyder, at der er fundet en kortere vej hen til punktet end den allerede kendte. På et tidspunkt vil A* kigge ud fra et punkt, som har slutpunktet som nabo og så vil A* have fundet en korteste sti. Hvis der ikke findes en vej i grafen, fra startpunktet til slutpunktet, vil A* stoppe, når alle punkter, der er forbundet til startpunktet, er kigget igennem.

I sagens natur findes A* med forskellige udgaver af *h*-funktionen, alt efter hvilket specifikt søgeproblem der skal findes en korteste sti til. I vores opgave bestemmes *h*-værdien som den direkte afstand mellem to punker i det tredimensionelle rum. Denne heuristik er god til 3D Pathfinding i spil, fordi man her som oftest kan bevæge sig frit, men aldrig på en måde så man kommer lettere hen til et punkt end den direkte retning. Dette gør, at denne heuristik aldrig vil overvurdere hvor stor afstanden er mellem to punkter og den er derfor sikker at benytte. Samtidigt er den direkte linie

også den vej agenten vil tage hvis det er muligt, hvorfor heuristikken ikke kan blive mere præcis uden at risikere at komme til at overvurdere afstanden til slutpunktet.

12.1.2.1.1 Pseudokode

I det følgende ses hvorledes A* kan beskrives med pseudokode dog med hoben undladt. Hoben opfører sig som en standard min-hob[62].

```
A*(G, s, goal)                                  //is called with a graph a start and a goal node.

1  List P ← Ø                                   //open-list is initialized.
2  List Q ← Ø                                   //closed-list is initialized.
3  g[s] := 0                                    .
4  h[s] := abs(s, goal)                         //calculate straight line distance from s to goal
5  f[s] := h[s]
6  P := s                                       // starting point is added to the open-list
7  While P ≠ Ø
8       c := ExtractMin(P)                      //heap extracts node with smallest f-value.
9        add c to Q                                  //adds c to the closed-list
10        Do For each vertex v ∈ Adj[c]
11             If v ∉ Q
12                  If v ∉ P
13                       add v to P                  //adds v to the open-list
14                       Update(v, c)
15                  else if (g[v] > g[c] + abs(v, c))
16                       Update(v, c)
17                  if c = = goal
18                        return BacktrackPath(G, s, goal)      //goal node reached uses parent status to
trace path
19  return false                                 //if no path has been found
```

```
Update(v, c)                                    //updates the g-, h- and f-values

1  g[v] := g[c] + abs(v, c)
2  h[v] := abs(v, goal)
3  f[v] := h[v] + g[c]
4  P[v] := c                                    //makes node c parent of node v
5  UpdateHeap(v, f[v])                          //updates heap so min vertex is in top. O(lg V)
```

12.1.2.1.2 Intuitiv forklaring på hvorfor A finder korteste sti*

Algoritmen og dens korrekthed bygger på den grundlæggende idé, at *f* værdien skal være et optimistisk bud på hvor langt der er til målet. Dvs. at den faktiske afstand fra et startpunkt via et givet punkt og til slutpunktet aldrig må være mindre end punktets *f* værdi. Endvidere benyttes korteste afstand mellem to punkter både som heuristik og som kantvægte mellem to nabopunkter i

[62] Hoben svarer til beskrivelsen af en min-hob i [IA] fra side 129 og frem.

grafen. Dette gør, at når algoritmen når til næstsidste punkt på stien (dvs. punktet der kan se slutpunktet) så er dette punkts *f* værdi det samme som længden af den korteste sti.

Algoritmen vælger hele tiden det punkt med mindst *f* værdi og kigger ud på dets naboer for at se om det kan se slutpunktet. På et tidspunkt vil algoritmen komme til et punkt der kan se slutpunktet (såfremt der eksisterer en sti i grafen) og fordi alle *f* værdierne er optimistiske (og det punkt med den mindste er valgt), betyder det, at de andre punkter (med større *f* værdier) dermed ikke kan indgå i en kortere sti end den fundne. For et egentligt bevis for A* og dens korrekthed henvises til **[A*]** afsnit 9.2.2.

*12.1.2.1.3 Køretiden for A**

Køretiden for A* afhænger af implementeringen af åbenlisten, hvor tiden det tager at udføre tre forskellige funktioner er betydende, nemlig: tiden det tager at udtrække mindste element fra listen (linie 8 i pseudokoden), tiden det tager at indsætte et nyt element i listen (linie 13 og 14 i pseudokode) og tiden det tager at formindske værdien af et element der allerede er på listen (linie 16 i pseudokode).

Der er to oplagte implementeringsmetoder af åbenlisten. Enten kan åbenlisten implementeres blot som en liste (array), hvor det vil tage O(*V*) tid at udtrække mindste element (da man er nødt til at søge hele listen igennem for at være sikker på at have fundet det mindste element). Eller også kan listen implementeres som en prioritetskø i form af en min hob, hvor en standard ExtractMin operation kan gøres i O(*lg(V)*) tid, fordi hoben skal opdateres efter det mindste element er udtrukket. Til gengæld tager en standard InsertKey operation og en standard DecreaseKey operation også O(*lg(V)*) i en min hob. Disse operationer bruges af A* når nye punkter skal sættes på åbenlisten eller når der er fundet en kortere sti til et punkt der allerede er på åbenlisten. Disse funktioner kan begge gøres i konstant (O(*1*) tid) i liste implementeringen.

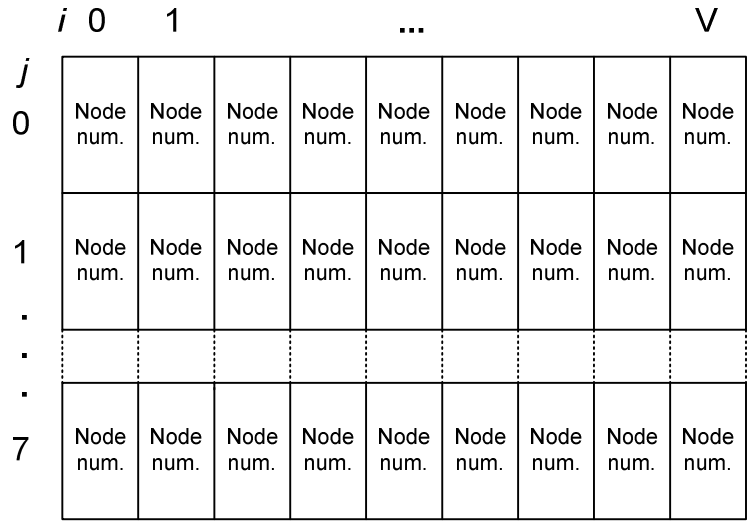
Hvilken af de to implementeringer man skal vælge afhænger af hvordan grafen. Hvis grafen er tynd, betyder det, at for hvert punkt der kigges på (fra åbenlisten) så skal der kigges ud på relativt få andre punkter, hvorfor udtrækningen af mindste element bliver overskyggende for køretiden. Derfor betyder det ikke så meget at det at indsætte et punkt eller ændre et punkts værdi på listen tager O(*lg(V)*) tid. Derimod bliver det vigtigt hurtigt at kunne udtrække mindste element på åbenlisten (dvs. i O(*lg(V)*) tid i stedet for i O(*V*) tid). I tilfældet med tynde grafer kan det derfor bedst betale sig, at benytte en min hob som prioritetskø i forbindelse med åbenlisten. Hvis problemet der skal løses derimod er repræsenteret i en tæt graf skal der indsættes og formindskes værdier på åbenlisten op imod *V* gange for hver gang mindste element udtrækkes fra åbenlisten. Her vil det derfor være oplagt at benytte en liste implementering af åbenlisten. Værste tilfælde køretiden er derfor bedst hvis man benytter en 'array implementering', hvor den samlede køretid bliver $O(V^2)$. Samme er for 'hob implementeringen' O(*E lg(V)*). I grafen der benyttes i forbindelse med pathfindingen i dette projekt, haves en tynd graf, da det på forhånd er defineret, at hver punkt højst kan have 8 naboer og grafen har flere end firs punkter. Reelt bliver forskellen derfor O(80*8*lg(80)) overfor $O(80^2)$. Derfor er åbenlisten implementeret som en prioritetskø i form af en min hob.

12.1.2.2 Modellering af graf til pathfinding

Til at repræsentere grafen til pathfinding er der brug for en repræsentation, der kan håndtere orienterede grafer. Grunden til at den netop skal kunne repræsentere *orienterede* grafer skyldes, at

det at man kan gå fra punkt A til B ikke automatisk medfører at man kan gå fra B til A (se Figur 34). Som regel benyttes enten en nabomatrix eller en hægtet liste til at repræsentere en graf i en computer. I dette projekt ser grafen dog ud på en sådan måde, at hvert punkt højst kan have otte naboer, hvorfor ingen af de to nævnte løsninger er oplagte, da nabomatricen her bruger for meget unødvendigt plads og de hægtede lister i praksis er for langsomme (der henvises til de resultater, der er fundet i forbindelse med **[PMP]** afsnit 7). Endvidere har vi ikke brug for at kunne slå enkelte punkter op, da A* kigger på *alle* nabopunkter for hvert punkt på åbenlisten, hvilket gør, at A* i forvejen er nødt til at gennemgå alle naboerne til et givent punkt og på den måde kender indekset til den liste, der indeholder naboer. Samme indeks kan bruges til at slå vægten på kanten mellem de to punkter op.

Derfor benyttes i stedet 3 lister til at holde styr på grafen, nemlig en liste til at holde styr på hvilke nabopunkter et punkt har, en liste der fortæller vægten på kanterne til disse naboer og endelig en liste der mapper fra et punktnummer til en position på banen. De 2 første lister har samme struktur og da hvert punkt højst kan have otte naboer er listerne så lange som antallet af punkter (*V*) i grafen og otte høje. I den ene liste kan man slå punktnummeret op på en nabo. Punktnummeret er indekseret *i* og dens naboer er indekseret med *j*. Man kan derfor slå *i*'s naboer op med (*i*, *j*), hvor *j* er nummeret på en nabo (mellem 0 og 7) og få punkt nummeret på en nabo. Såfremt punktet har færre end 8 naboer, er de sidste pladser i arrayet udfyldt med '-1' taller, således at hvis et punkt har seks naboer, så returnerer (*i,* 6) og (*i,* 7) begge '-1' – på den måde kan A* stoppe med at kigge på et punkts naboer, når den ser et '-1', da den så ved, at punktet ikke har flere. Hvis punktet *i*'s første nabo er punktnummer 5, så returnerer (*i,* 0) nummer 5. Arrayet kalder vi adjArray og ses nedenfor.



Figur 1: adjArray, fortæller hvilke naboer hvert af punkterne har.

Det er imidlertid ikke tilstrækkeligt at vide hvilke punkter der har hvilke naboer. Det er også nødvendigt at kende vægtene på kanterne mellem de punkter der er naboer. Dette er opbygget fuldstændigt som adjArray, men i stedet for at returnere punktnummeret på naboen, så returneres afstanden til naboen i stedet.

Sidst er der naturligvis brug for en måde at komme fra et punktnummer i grafen til en egentlig position på banen. Dette gøres ved simpelt ved at have et array af objekter der er indekseret med

punktnumrene i grafen. Objekterne indeholder så tre værdier, nemlig x, y og z koordinaterne der fortæller hvor punktet er placeret på banen.

12.1.2.3 Implementeringen af grafen

Grafen som A* finder korteste sti i er implementeret fuldstændig som beskrevet i modelleringsafsnittet. Filerne graf objektet er implementeret i hedder 'pfGraph.h' og 'pfGraph.cc', som henviser til 'PathFinding Graph'. Grafen består af et objekt som indeholder tre to dimensionelle arrays, der fortæller hvilke punkter der er naboer til hvert af punkterne indeholdt i grafen, hvor langt der er til hver af disse naboer og endelig hvor disse punkter befinder sig på banens koordinat system. De tre arrays kaldes for adjArray, distanceMatrix og nodeToWorld og er henholdsvis af typerne int, float og Point3F. Point3F er et objekt der indeholder x,y og z koordinater. Desuden implementerer objektet en række funktioner til at oprette, sætte og hente værdier i grafens forskellige arrays. Eksempelvis allokeres hukommelsen til nodeToWorld arrayet i følgende funktion:

```
void pfGraph::createNTW(S32 numNodes){
           nodeToWorld = (Point3F *) dMalloc(numNodes*(sizeof(Point3F)));
}
```

S32 er en signed integer, og variablen numNodes fortæller hvor mange punkter (nodes) der er i grafen. dMalloc er Torques implementering af malloc, som i c benyttes til at allokere hukommelse dynamisk. dMalloc benytter Torques memory manager til at allokere hukommelse dynamisk. Torques memory manager er implementeret på en sådan måde at den allokere en 'klump' hukommelse på 8mb ad gangen (dynamisk) og når man så benytter f.eks. dMalloc, så får man noget af dette allerede allokerede hukommelse. På den måde undgås det at man hele tiden bruger tid på at allokere og frigøre hukommelse dynamisk. Dette er vigtigt, fordi bagvedliggende kode skal afvikles så hurtigt som overhovedet muligt i runtime (mens spillet spilles), så spillet ikke hakker.

Da samme graf benyttes af alle agenter, er der lavet individuelle pladser til hver agents start og slutpunkter i grafen, således, at vi undgår problemer med parallelle processer, der skriver til samme variabler. Alle grafens punkter (undtagen agenternes start og slutpunkter) læses kun af agenterne og agenternes start og slut punkter læses kun af hver enkelt agent, hvorfor det ikke er nødvendigt at implementere atomiske funktioner der låser variablerne med f.eks. semaforer når variablerne er i brug. De individuelle pladser i arraysne får indeksene, startpunkt: `(numOfNodes + agentNum*2)`, slutpunkt: `(numOfNodes + agentNum*2 + 1)`, hvor `numOfNodes` er antallet af faste punkter i grafen og `agentNum` er agentens individuelle unikke nummer, hvor den første agent har nummer 0 den anden agent nummer 1 osv.

12.1.2.4 Implementering af A*

Implementeringen af A* findes i 'AIPlayer.h' og 'AIPlayer.cc'. Hovedfunktionen hedder `aStar` og kaldes med et start- og et slutpunkt for en given agent. Disse to tal mapper til punkternes indeks i grafen. Det første der sker er at en hjælpefunktion ved navn `initNodeInfoList` kaldes med slut og startpunktet. Denne hjælpefunktion kalder en funktion som opretter hoben i den størrelse som grafen har, således at der er plads til alle grafens punkter på hoben (åbenlisten), hvis det skulle blive nødvendigt. Samtidigt lægges startpunktet på hoben. En mere detaljeret beskrivelse af hvordan hoben fungerer følger i afsnit '12.1.2.4.1 Implementeringen af Hoben'. Udover at oprette hoben

initialiseres et array kaldet `nodeInfoList`. Dette array indeholder informationer om hver enkelt af punkterne, hvor punktnummeret i grafen bruges som indeks til at få fat i informationerne om det enkelte punkt. På denne måde kan informationerne gemt i arrayet får i konstant tid, hvis man har punktnummeret på det punkt man vil udtrække informationerne for. I arrayet er det angivet om et punkt optræder på en af A*'es to lister, åben- og lukketlisten. Desuden indeholder listen hvilket andet punkt der på pågældende tidspunktet er forælder til hvert af punkterne, såfremt de har en. Ligeledes er der angivet der *g*-, *h*- og *f*-værdier her, og endeligt er antallet af punkter i den sti som fører til punkterne angivet. Den sidstnævnte information skyldes at der ikke kan passes arrays mellem scripts og C++ delene i Torque. Det er derfor er nødvendigt at vide hvor lang stien er, så den kan hentes af scriptet ved at køre en for-løkke som kører lige så mange gange som der er punkter på stien. `nodeInfoList` arrayet initialiseres ved at sætte startpunktets *g*-værdi til 0, beregne afstanden mellem start- og slutpunkt og sætte denne værdi til startpunktets *h*- og *f*-værdi. Det er ikke muligt dynamisk at lave et 2 dimensionalt array i C (uden det er en pointer af pointers). Derfor er arrayet implementeret som et almindeligt et dimensionalt `float` array, der laves dynamisk, alt efter størrelsen på grafen. Størrelsen er 7 gange så stort som antallet af punkter i grafen, da arrayet indeholder 7 værdier for hvert punkt. På denne måde fås informationerne i arrayet ved at slå op på følgende måde:

```
[Node*numOfColumns+Index]
```

Hvor `Node` angiver punktets nummer, `numOfColumns` angiver bredden på arrayet (her 7) og `Index` angiver hvilken information eller værditype (f.eks. er indeks 2 punktets h værdi), der henvises til for det pågældende punkt. Der er lavet konstanter med passende variabel navne til hver af værdierne, f.eks.:

```
const static S32 hIndex = 2;
```

Således kan arrayet opfattes som et dobbelt array hvor punktnummeret er det ene indeks og de forskellige værdityper er det andet indeks.

Efter `nodeInfoList` er initialiseret er det næste der sker i A*, at der køres en while-løkke som kører så længe der er flere elementer på hoben (åben listen). I denne løkke udtrækkes det mindste element fra hoben, altså det punkt med mindste *f*-værdi. Punktnummeret fra dette punkt gemmes i en variabel der kaldes 'selectedNode'. Denne variabel indeholder så det punkt som der i modelleringsafsnittet siges at 'der kigges ud fra' under afvikling af A*. Derudover tilføjes punktet til lukketlisten, vha. en funktion som hedder 'addToClosedList' som kaldes med 'selectedNode'. Denne funktion går ind i 'nodeInfoList' arrayet og flytter 'selectedNode' fra åben listen over i lukketlisten med følgende to linier:

```
nodeInfoList[node*numOfColumns+closedListIndex] = 1;
nodeInfoList[node*numOfColumns+openListIndex] = 0;
```

Den første linie sætter et 1-tal på pladsen med indeks 'node' og 'closedListIndex', hvor 'node' er det punktnummer som funktionen kaldes med. 1-tallet betyder således at punktet indgår i lukketlisten. Den anden linie fungerer på tilsvarende måde, men skriver altså 0 på pladsen som angiver om punktet er på åbenlisten, fordi det ikke længere optræder her.

Det bemærkes at informationen om hvorvidt et punkt optræder på en liste eller ej, kan gives ved kun at benytte én bit, og at der ved at blande den ind i 'nodeInfoList' arrayet ofres en hel 32-bit's integer

på hvert punkt på både åben- og lukketlisten. Her kunne man altså have sparet plads ved at lave et specifikt bit-array til denne information, men da det blev vurderet at koden ville blive lettere at overskue på denne måde, blev det besluttet at samle informationerne om punkterne i et array.

Herefter skal A* se på alle naboer til `selectedNode`, hvilket foregår i en ny while-løkke, der kigger i grafen så længe der er flere naboer til punktet – dvs. så længe punktnummeret på naboen ikke returneres til '-1'. Der huskes her tilbage på den måde grafen er repræsenteret på, hvor arrayet der indeholder naboer til et punkt er lavet således, at naboerne altid står på de første pladser i arrayet, således, at hvis der f.eks. er 3 naboer, så står de på pladserne 0, 1 og 2 i arrayet. Dermed vides det, at hvis man kommer til et naboindex i grafen, hvor der ikke er et punktnummer (dvs. der står '-1') så er der ikke flere naboer til dette punkt. Hver nabo til `selectedNode` kaldes så `adjacencyNode` og afstanden mellem de to punkter hentes så fra grafens `distanceMatrix`.

Det undersøges nu i en if-sætning om det fundne nabopunkt er indeholdt i lukketlisten ved at kigge i `nodeInfoList`. Hvis det ikke er det fortsætter programmet ind til if-sætningens indhold. Her spørges der på samme måde om nabopunktet er på åbenlisten. Hvis ikke det er tilfældet kaldes en funktion, som hedder 'addToOpenList' som tilføjer nabopunktet til åbenlisten. Udover dette sættes 'selectedNode' som forælder til nabopunktet og længden af stien samt *g*-, *h*- og *f*-værdierne beregnes og gemmes i `nodeInfoList`. Heuristikken (*h* værdien) udregnes ved at kalde funktionen `straightLineDistance`, som udregner 'lige linie afstanden' mellem punkterne.

Hvis punktet var på åbenlisten undersøges om der kan findes en kortere sti til nabopunktet via `selectedNode`. `selectedNode`'s *g*-værdi slås op i `nodeInfoList` og lægges til den vægt der er fundet mellem punkterne. Denne sum sammenlignes med nabopunktets *g*-værdi som også slås direkte op i 'nodeInforList' arrayet. Er der fundet en kortere sti kaldes funktionen `shorterPathUpdate`, som ændrer nabopunktets forælder til `selectedNode` og opdaterer *g*- og *f*-værdierne samt stiens længde.

Endeligt laves et check på om nabopunktet er slutpunktet. Hvis dette er tilfældet bruges forælder informationen fra `nodeInfoList` til at backtracke tilbage til startpunktet ved hjælp af en funktion som kaldes `backtrackPath`. Denne funktion gemmer alle forælder værdierne til punkterne i stien og 'backtracker' på denne måde stien fra slut til startpunkt. Stien gemmes herefter i et objekt der indeholder alle arrays der indeholder alle agenternes stier, således at stien kan hentes fra scriptkoden. Stiens længde returneres således at scriptkoden ved hvor mange kald der skal foretages for at hente den komplette sti fra C++ delen af Torque.

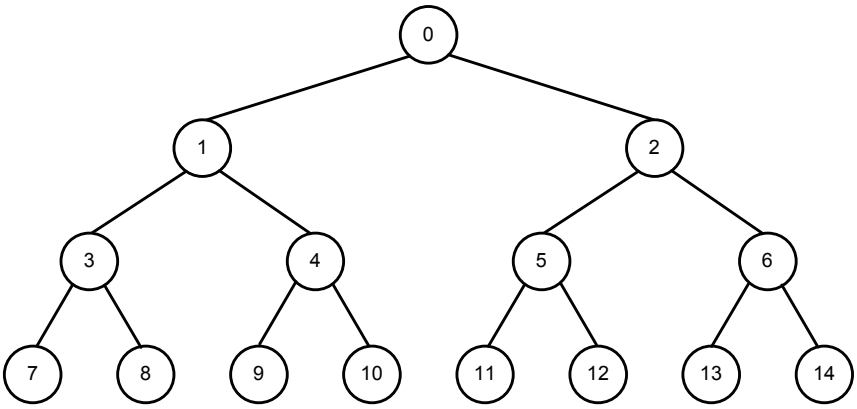
12.1.2.4.1 Implementeringen af Hoben

I dette afsnit ses på hvorledes hoben, som benyttes til en del af åbenlisten i implementeringen af A*, i praksis kom til at se ud.

Hoben ligger implementeret i filen Heap.cc. Den opererer med tre hoved arrays som hedder henholdsvis, `placeToKey`, `placeToNode` og `nodeToPlace`. `placeToKey` arrayet mapper til key værdien (*f*-værdien) givet en plads i hoben. `placeToNode` mapper til punktnummeret i grafen givet en plads i hoben, og `nodeToPlace` mapper til et punkts plads i hoben, givet dets nummer. Disse tre arrays oprettes når funktionen `createHeap` kaldes med et antal punkter, som der skal kunne være på hoben. Herefter fyldes `placeToKey` med 0'er og i `placeToNode` og `nodeToPlace` lægges indekset ind som værdi. Det betyder nu, at alle pladserne i hoben har den samme key-værdi nemlig 0, og det

svarer til at alle de punkter der ligger på hoben har *f*-værdi 0. Da der til start ikke er nogen punkter på hoben betyder dette ikke noget. For at holde styr på hvor mange punkter, der er på hoben opererer hoben med en variabel som hedder `heapSize` som tælles op hver gang der lægges et element på hoben og som tælles ned når minimum trækkes ud.

For at lette forståelsen af hvordan hoben fungerer vises først hvordan hobens pladser er nummereret på Figur 36. I denne gennemgang er det punkternes *f*-værdi som opfattes som key-værdien, da det er *f*-værdiernes størrelse hoben ønskes at sortere punkterne efter, og denne værdi normalt kaldes for hobelementernes key-værdi.



Figur 36: Viser en visualisering af hoben, med nummerering af pladserne.

For at kunne afgøre hvilken anden plads i hoben der svarer til forælderen, højre barn eller venstre barn, findes tre funktioner, som givet et pladsnummer, returnerer pladsindekset for henholdsvis forælder, højre - og venstre barn til pladsnummeret. Disse tre hjælpefunktioner beskrives i det følgende.

Den første af hjælpefunktionerne hedder `getParent`. Den kaldes med et pladsindeks og returnerer forælderens pladsindekset. Hvis den f.eks. kaldes med pladsindeks 5 vil den returnere 2, da pladsnummer to er forælder til plads nummer 5, som det også fremgår af Figur 36. udregningen af forælderpladsen foregår i følgende linie kode:

```
return (int) (placeInHeap-1)/2;
```

Der trækkes 1 fra `placeInHeap` som indeholder det pladsindeks som funktionen blev kaldt med. Herefter divideres med to og resultatet konverteres til et heltal, hvilket betyder at der rundes ned til nærmeste heltal (trunkeres).

I eksemplet med 5 vil regnestykket derfor se således ud:

$$\lfloor (5-1)/2 \rfloor = \underline{2}$$

Her giver det altså 4 uden at der rundes ned, hvis der forsøges med 10 fås:

$$\lfloor (10-1)/2 \rfloor = \underline{4}$$

I dette tilfælde giver (10-1)/2 resultatet 4,5 som så rundes ned til 4. hvilket også passer med oversigten på Figur 36.

Den anden af de tre hjælpefunktioner er `getLeftChild`. Denne giver det barn som man finder i Figur 36 ved at bevæge sig nedad i hoben mod venstre. Denne funktion returnerer pladsindekset på det venstre barn givet et indeks på en plads i hoben. Beregningen af indekset sker i følgende linie kode:

```
return placeInHeap * 2 + 1;
```

Også her angiver `placeInHeap` pladsindekset for det element som funktionen kaldes med. Dette indeks ganges med 2 og der lægges 1 til resultatet. Tages pladsen med indeks 5 som eksempel fås følgende beregning:

$$5 \cdot 2 + 1 = \underline{\underline{11}}$$

Som det også fremgår på Figur 36 er plads nummer 11 det venstre barn til pladsen 5.

Endeligt giver hjælpefunktionen `getRightChild` det højre barn. Denne funktion fungerer helt som `getLeftChild` bortset fra at der lægges 2 til i stedet for 1. Tages tallet 5 igen som eksempel fås følgende udregning:

$$5 \cdot 2 + 2 = \underline{\underline{12}}$$

Det ses også her, at pladsen med indeks 12 er barn til plads nummer 5, men denne gang altså det højre.

I det følgende gennemgås de videre funktioner som hoben benytter sig af. Når A* vil lægge et element på hoben kaldes en funktion som hedder ’insertKey’. Funktionen kaldes med et punktnummer og en *f*-værdi. *f*-værdien lægges ind på den første frie plads i `placeToKey` arrayet ved med følgende kommando:

```
placeToKey[heapSize] = fValue;
```

Første gang der lægges et element på hoben er `heapSize` lig med 0, og elementet vil derfor blive lagt på plads nummer 0 i arrayet. Herefter lægges punktnummeret ind i `placeToNode` med følgende kommando:

```
placeToNode[heapSize] = node;
```

Første gang dette gøres vil `node` indeholde startpunktets nummer. Herefter lægges `heapSize`’s værdi ind i `nodeToPlace` arrayet med følgende kommando:

```
nodeToPlace[node] = heapSize;
```

Første gang er `heapSize` som sagt 0 og derfor kommer der til at stå 0 på den plads i arrayet som har startpunktets nummer. Derved mapper `nodeToPlace` og `placeToNode` til hinanden.

Hvis det ikke er det eneste element, der er på hoben er det muligt, at det er mindre end nogle af de elementer, der er højere oppe i hoben. Da det er indsat på den første frie plads i hoben vil der aldrig kunne være elementer som ligger under den plads som punktet lægges ind på. Derfor kaldes funktionen `updateUpwards`, og `heapSize` tælles op fordi hoben nu er et element større.

Funktionen `updateUpwards` kaldes med en plads i hoben. Den kigger så om det element der er ovenover i hoben, altså forælderen til punktet, har en større key-værdi end det punkt som står på den plads, funktionen er blevet kaldt med. For at `updateUpwards` kan vide hvilken plads i `placeToKey` arrayet som svarer til punktets forælder beregnes forælderens pladsindeks ved at kalde `getParent`. Det indeks som `getParent` returnerer bruges til at slå op i `placeToKey` for at sammenligne *f*-værdierne. Hvis *f*-værdien for det punkt som er forælder er størst byttes om på punkterne ved at kaldes funktionen 'exchange'.

Funktionen 'exchange' kaldes med pladsindeksene for de to punkter, der ønskes ombyttet. I følgende kodestump foregår ombytningen:

```
//Bytter key værdierne om
temp = placeToKey[placeInHeap1];
placeToKey[placeInHeap1] = placeToKey[placeInHeap2];
placeToKey[placeInHeap2] = temp;

//Bytter placeToNode værdierne om
temp = placeToNode[placeInHeap1];
placeToNode[placeInHeap1] = placeToNode[placeInHeap2];
placeToNode[placeInHeap2] = temp;

//Bytter nodeToPlace værdierne om
temp = nodeToPlace[placeToNode[placeInHeap1]];
nodeToPlace[placeToNode[placeInHeap1]] = nodeToPlace[placeToNode[placeInHeap2]];
nodeToPlace[placeToNode[placeInHeap2]] = temp;
```

Variablene `paceInHeap1` og `placeInheap2` er pladsindekset på de to punkter som skal byttes rundt. De benævnes i de følgende, som henholdsvis punkt1 og punkt2. Funktionen starter med at lægge *f*-værdien for punkt1 over i en midlertidig opbevarings variabel kaldet 'temp'. Herefter lægges *f*-værdien for punkt2 over i `placeToKey` på punkt1's plads i arrayet, og efterfølgende lægges *f*-værdien fra punkt1 som blev gemt i temp, over i `placeToKey` på punkt2's plads. Hermed er *f*-værdierne ombyttet.

Det næste der sker er, at der skal byttes om på punkterne i 'placeToNode' arrayet. Dette gøres ved først at få punktnummeret for punkt1 lagt over i temp. Herefter lægges punktnummeret for punkt2 over i `placeToNode` på punkt1's plads. Og til sidst altså punkt1's punktnummer fra 'temp' over i `placeToNode` på punkt2's plads. Hermed mangler kun `nodeToPlace` arrayet at blive opdateret.

Det sidste trin er lidt mere omstændeligt, fordi `nodeToPlace` og `placeToNode` nu ikke mapper til hinanden længere. Derfor foregår den sidste ombytning ved at punkt1's pladsindeks bruges til at slå punkt1's punktnummer op i `placeToNode`, og dette indeksnummer slås så op i `nodeToPlace` arrayet og lægges over i `temp`. Herefter slås punkt2's punktnummer op i `placeToNode` som bruges til at slå op i `placeToNode` med, og indekset lægges over på den plads som fås ved at bruge slås punkt1's punktnummer op i `placeToNode` og herefter slå op i `nodeToPlace`. Endeligt lægges

punkt1's pladsindeks fra `temp` over i den plads som fås ved at slå punktnummer op med `placeToNode` og herefter `nodeToPlace`.

Efter at kaldet af `exchange` er udført kalder `updateUpwards` sig selv igen, da det kunne være at punktet skulle rykkes længere op i hoben, og proceduren gentages til punktet enten er nået toppen eller der ligger et punkt med mindre *f*-værdi over det. Hoben opdateres altså rekursivt.

Når A* skal bruge det mindste element fra hoben kaldes funktionen ved navn `extractMinimum()`. Denne funktion kaldes uden argumenter og returnerer punktnummeret på det punkt fra hoben som har den mindste *f*-værdi. Funktionen afvikles ved først at få punktnummeret til toppunktet i hoben ved at tage 'placeToNode' af 0. Dette indeks gemmes herefter i en variable som kaldes `topNode`. Nu skal punktet fjernes og hoben opdateres. Derfor kaldes `exchange` på toppunktet og det sidste element i hoben, altså det med det højeste pladsindeks som vil være `heapSize-1`. Herefter tælles `heapSize` én ned, således at den plads hvor toppunktet blev flyttet ned ikke længere indgår i hoben. Derefter kaldes en funktion, som hedder `updateDownwards` således at det punkt der netop er lagt på hoben kommer til at ligge rigtigt i forhold til dets *f*-værdi. Og efter hoben er opdateret returneres `topnode` som indeholder punktnummeret på det punkt som før var toppen og nu er blevet fjernet fra hoben.

`updateDownwards` kaldes også med en 'plads i hoben. Den undersøger om der findes to børn til pladsen ved at se om `heapSize` er mindst lige så stort som det pladsindekset for det højre barn, som fås vha. `getRightChild`. Hvis begge børn findes sammenlignes de. Dette gøres ved at kalde `getLeftChild` og `getRightChild` og bruge de to pladsindeks som de retunerer til at slå op i `placeToKey` og se hvilket barn der har den mindste *f*-værdi. Det mindste barns plads gemmes i en variabel kaldet `minChild` og der sammen lignes igen *f*-værdier for `minChild` og punktet som `updateDownwards` blev kaldt med for at se om punktet skal flyttes nedad i hoben. Hvis dette er tilfældet kaldes `exchange` for at bytte punkterne og `updateDownwards` kalder sig selv igen for at se om punktet skal rykkes længere ned.

Hvis punktet ikke har et højrebarn undersøges om der findes et venstre. Er dette tilfældet sammenlignes der på samme måde som før *f*-værdier for at se om punktet som `updateDownwards` skal bytte plads med det venstre barn. Nu vides det så at der ikke er flere børn, og derfor behøver funktionen ikke længere kalde sig selv selvom punktet og dets venstre barn skal ombyttes. Hvis punktet ikke har nogen børn eller er mindre end begge børn slutter funktionen også.

Endeligt kan det også ske at A* finder en ny sti til et punkt som er kortere og dermed giver punktet en mindre *f*-værdi. I denne situation skal punktets *f*-værdi opdateres, og det kan så betyde, at punktet skal flyttes opad i hoben. Til det formål findes der en funktion i hoben som hedder `decreaseKey`.

`decreaseKey` kaldes med en den nye *f*-værdi og punktets nummer. Der slås så op i `nodeToPlace` for at få punktets placering i hoben, og herpå med pladsindekset i `placeToKey` for at få den plads hvorpå *f*-værdien skal overskrives med en nye *f*-værdi. Derefter kaldes `updateUpwards`, som sørger for at punktet placeres korrekt i hoben efterfølgende.

12.2 Agenternes handlingsfunktioner - implementering

For at agenterne kan indsamle inputs og foretage handlinger, har det været nødvendigt at implementere en lang række funktioner. I dette underafsnit vil enkelte af disse funktioner blive kort beskrevet.

For at en agent kan vælge en sti ned gennem sit beslutningstræ kræves det at han kan svare på de spørgsmål, som er opstillet i træet. Et af de central spørgsmål er her hvad agenten kan se af fjender. Da det er meningen at holdet skal arbejde mere og mere sammen på sigt, er det også interessant at se på hvilke fjender ens medspillere kan se. Til det formål er der lavet to funktioner ved navn *getEnemiesInSight* og *countEnemiesInGroupsSight*.

Foruden disse funktioner findes der også en række funktioner som afgør hvilken af de fjender, der er kendskab til på et givent tidspunkt, og hvilken fjende der er henholdsvis den nærmeste fjende, mest og mindst sårbare fjende, samt om fjenden er del af en gruppering af fjender, eller om den står mere isoleret. Derudover findes også en mindre funktion til udregne afstanden mellem en agent og en given fjende.

Udover at indsamle inputs fra banen skal agenterne kunne handle ud fra de input som er observeret. Dette gøres som bekendt ved at der i bunden af beslutningstræet findes en række pointere som peger på de funktioner der skal kaldes. Alene den del der har med pathfindingen at gøre er en længere historie som vi ikke vil komme ind på her. Til gengæld vil vi kort beskrive nogle af de andre vigtigste funktioner.

For at agenterne overhovedet kan skyde på hinanden er der implementeret en funktion som muliggør dette. Denne funktion hedder *shootAtEnemy* og er relativt simpel. En lidt mere kompleks funktion er funktionen *lookForEnemy* som bruges til at finde fjender såfremt ingen kan ses, eller hvis agenten vurderer at der er for lang til nærmeste kendte fjende, og derfor hellere vil lede efter fjender i det område som den befinder sig i. Funktionen benytter sig af en række underfunktioner som blandet andet udnytter mutexlåse til at beskytte et array med en liste over hvilke gemmesteder der nyligt er undersøgt, så agenterne effektivt kan finkæmme banen på en koordineret måde.

Foruden muligheden for at lede efter fjender er der også implementeret funktioner til at lede efter health kits og ammonition. Disse er uundværlige for de forsvars FSMer som bruges til at træne angrebstræerne mod. Til denne opgave er der implementeret to funktioner ved navn *findNearestHealth* og en ved navn *findNearestAmmo*. Begge disse funktioner benytter en mere generel funktion som hedder *findNearestItem*, og for at agenterne senere kan blive mere intelligente er der implementeret en metode som endnu ikke bruges der kan finde de to objekter (af to forskellige typer) som samlet set er nærmest. På den måde kan en agent f.eks. planlægge den korteste rute til at finde både et health kit og et ammunitions kit hvis den skal bruge begge dele.

Det har under implementeringen været nødvendigt at flytte en lang række af disse funktioner fra enginens scriptsprog over i selve enginen som er skrevet med C++. Dette skyldes at enginens script sprog viste sig at være alt for langsomt, men grundet tidspres er denne overførselsprocedure endnu ikke færdiggjort. Det er naturligvis også et sted vi vil være nødt til at sætte ind i det fremtidige arbejde.

12.3 Input til-, handlinger fra- og begrænsning i beslutningstræerne

12.3.1 Top level tree

Functions (overall strategy)

1. Help team mate attacking.
2. Help team mate defending.
3. Attack on own
4. Defend

Input

1. Ammo * health
2. Distance to nearest attacking team mate (100% hvis ingen findes)
3. Distance to nearest defending team mate
4. Distance to nearest enemy
5. Nearest enemy in group? (y/n)
6. Distance to nearest health or ammo
7. Number of enemies in sight
8. Any other team mates helping selected team mate (y/n)

Possible constraints: (‘*Functions*, ‘*Input’*)
(1, **2 if value = ‘-1’**) & (2, **3 if value = ‘-1’**)

12.3.2 Attacking tree

Selection destination

1. Do not move
2. Move to nearest enemy
3. Move to most vulnerable enemy (most isolated)
4. Look for enemy
5. Move to least vulnerable (largest group of enemies)

Arrive from

1. North
2. East
3. South
4. West
5. Shortest path
6. Nearest 'out of sight of opponent player' point

1-4: Kan ikke bruges med selection destination nummer 1 og 4.

Shoot functions

1. Do not shoot
2. Shoot at nearest enemy
3. Shoot at most vulnerable enemy
4. Shoot at nearest shooting enemy
5. (og måske også Least vulnerable enemy)

Input

1. Known distance to nearest shooting enemy
2. Any team mates helping me? (yes/no – in practice done by rate of 1 or 0)
3. Known distance to nearest enemy
4. Known distance to most vulnerable enemy
5. Number of enemies in sight
6. Nearest enemy in group? (yes/no – in practice done by rate of 1 or 0)

Possible constraints: ('*Selection destination'*, '*Arrive from'*, '*Shoot functions'*, '*Input'*)

(**2**,-,3,-) & (**4**,-,{2;3;4},-) & ({2;3;5},-,{2;3;4},**3 if value ='-1'**) & (-,-,4,**1 if value ='-1'**) & (-,-,{2;3;4},**5 if value ='0'**) & ({2;3;5},-,{2;3;4}, **4 if value ='-1'**) & (-,-,4, **1 if value ='-1'**)

12.3.3 Defending tree

Select destination

1. Do not move
2. Move to nearest available health kit
3. Move to nearest available ammo kit
4. Move out of sight
5. Move to nearest team mate

Arrive from

1. Shortest path

Shoot functions

1. Do not shoot
2. Shoot at nearest shooting enemy

Input

1. Ammo status
2. Health status
3. Distance to nearest team mate
4. Distance to nearest available ammo kit
5. Distance to nearest available health kit
6. Hiding place (building) in range (yes/no)

Possible constraints: (‘*Selection destination’*, ‘*Arrive from’*, ‘*Shoot functions’*, ‘*Input’*)
(5,-,-,**3 if value =’-1’**) & (2,-,-,**5 if value =’-1’**) & (3,-,-,**4 if value =’-1’**) &
(4,-,-,**6 if value =’0’ (no))**

12.3.4 Help defending tree

Select destination

1. Do not move
2. Move to selected team mate

Arrive from

1. shortest path

Shoot functions

1. Do not shoot
2. Team mate's nearest enemy
3. Nearest shooting enemy (to you)
4. Team mate's nearest shooting enemy

Input

1. Distance to selected team mate
2. Distance from selected team mate to team mates nearest enemy
3. Any other team mates helping selected team mate (y/n)
4. Distance to nearest shooting enemy

Possible constraints: ('*Selection destination'*, '*Arrive from'*, '*Shoot functions'*, '*Input'*)
(-,-,{2;3;4},2 if value ='-1') & (-,-,{3;4}, **4 if value ='-1'**)

12.3.5 Help attacking tree

Select destination

1. Do not move
2. Move to selected team mate
3. Move to selected team mate's target

Arrive from

1. Same direction as most team mates
2. Opposite direction than most team mates
3. Different direction than most team mates
4. shortest path

Shoot functions

1. Do not shoot
2. Nearest shooting enemy
3. Selected team mate's enemy

Input

1. Distance to selected team mate
2. Distance to selected team mate's enemy
3. Any other team mates helping selected team mate (y/n)
4. Any known enemies shooting (y/n)
5. Team mate's distance to target
6. Distance to nearest shooting enemy

Possible constraints: ('*Selection destination'*, '*Arrive from'*, '*Shoot functions'*, '*Input'*)
(**2**,{1;2;3},-,-) & (3,-,3,**2 if value = '-1'**) & (-,-,2,**4 if value = '0'(no)**) & (3,-,3, **5 if value = '-1'**) & (-,-,2, **6 if value = '-1'**)

12.4 Kode

Koden er vedlagt som separat appendiks, med titel: ’Evolutionære agenter – Bilag kildekode’.