Kungliga Tekniska Högskolan

Examensarbete inom datalogi, grundnivå
(DD143X)

AI för Hive

Författare:
Urban Pettersson
urbanpe@kth.se

Handledare:
Per Austrin

29 april 2014

Sammanfattning

I projektet implementerades och utvärderades strategier för brädspelet Hive. Rapporten går igenom regler samt systemet som implementerades för att kunna skapa strategier. Strategierna utvärderas baserat på hur de presterar mot varandra. Två sökalgoritmer implementerades: Minmax och Alpha-Beta Pruning. Alpha-Beta med djup 4 presterade bäst mot andra strategier samt i individuella tester.

Abstract

Strategies for the board game Hive was implemented and evaluated in this project. The report covers the rules as well as the system that was implemented in order to build strategies. The strategies are evaluated based on how they perform against each other. Two search algorithms were implemented: Minmax and Alpha-Beta Pruning. Alpha-Beta with depth 4 had the best performance against other strategies as well as in individual tests.

Innehåll

1 Inledning **1**

2 Problembeskrivning **1**

3 Metodbeskrivning **1**

4 Hive **2**
4.1 Regler . . . 2
4.2 Pjäser . . . 4
4.2.1 Drottning . . . 4
4.2.2 Spindel . . . 4
4.2.3 Myra . . . 4
4.2.4 Skalbagge . . . 4
4.2.5 Gräshoppa . . . 4

5 Implementation av bassystemet **5**
5.1 Piece . . . 5
5.2 Board . . . 6
5.3 Handler . . . 6
5.4 Player . . . 7
5.5 Felsökning av struktur . . . 7

6 Trädalgoritmer **7**
6.1 Evalueringsfunktion . . . 8

7 Strategier **9**
7.1 Random . . . 9
7.2 Aggro . . . 9
7.3 Greedy . . . 10
7.4 Minmax . . . 10
7.5 Alpha-Beta pruning . . . 10

8 Resultat **10**
8.1 Branching factor . . . 10
8.2 Utvärdering av ett drag . . . 10
8.3 Matchresultat . . . 10

9 Diskussion **11**

10 Slutsatser **12**

Referenser **13**

1 Inledning

Brädspel har en lång och intressant historia. De äldsta tros vara över fem tusen år gamla [1]. Spel som schack och go har spelats i över ett millenium. Spelen har genom åren fascinerat lärda så väl som kungligheter. Stormästare har kommit och gått.

År 1770 förändrades allt. Världens första schackspelade maskin introducerades: Turken [2]. Maskinen spelade väl och slog under åren många kända personer. Det hela var självfallet bara en bluff, en schackspelare satt undanstuvad i maskinen och kunde se allt som föregicks på brädet.

1996 körde stormästaren Garry Kasparov en serie matcher mot IBMs nykonstruerade schackdator Deep Blue [3]. Den skapades specifikt för att slå Kasparov, något den misslyckades med. Året därpå spelade de två än en gång och den här gången vann Deep Blue. Sedan dess har schackdatorerna med lätthet kunnat slå världens bästa spelare.

För strategispelet go är situationen annorlunda. Antalet drag växer snabbt och de bästa programmen har som bäst nått amatörnivåer [4]. Genom att minska på brädets storlek så ökar programmens chans att vinna.

Brädspel för två spelare med fullständig information samt utan slump är perfekta för AI att analysera. AIn behöver då enbart undersöka alla drag som kan göras av spelarna för ett visst antal drag framåt. Problem uppstår när antalet drag som måste analyseras växer snabbt. Detta begränsar hur djupt i sökträdet som AIn kan söka efter optimala värden.

Brädspelet Hive släpptes 2001. Det är likt schack och go ett spel för två spelare samt med perfekt information. Spelet är i antalet drag ett slags mellanting mellan schack och go. Detta i kombination med den intressanta spelstilen gör det till ett utmärkt mål för AI.

2 Problembeskrivning

I det här projektet kommer algoritmer implementeras och utvärderas för Hive. Algoritmerna kommer bestå av såväl enklare algoritmer som greedy samt svårare såsom minmax. Dessa algoritmer kommer utvärderas genom körningar mot varandra. För trädalgoritmerna kommer också data sammanställas över hur snabbt träden växer och vilken tidsåtgång en extra nivå i trädet kräver.

3 Metodbeskrivning

För att kunna skapa strategier måste först ett system implementeras som det går att spela på Hive på. Detta kräver full förståelse för hur spelet fungerar. Systemet måste ha kapacitet att skapa giltiga drag för spelet som strategier

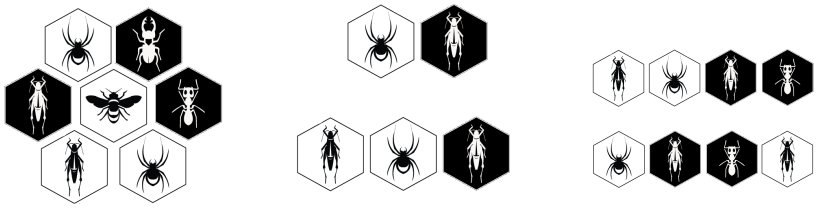
sedan kan fatta beslut på. Strategier kommer utvärderas i form av matcher mot varandra samt individuella data.

4 Hive

Hive är ett spel för två personer [5]. Spelarna turas om att göra varsitt drag. Spelet saknar bräde och byggs istället upp av de hexagonala pjäser som spelarna placerar ut. Slump saknas helt och hållet och ingen information är dold.

4.1 Regler

Varje spelare har elva pjäser: en drottning, två spindlar, två skalbaggar, tre gräshoppor och tre myror. Spelaren som först lyckas omringa den fientliga drottningen vinner. Drottningen behöver ej enbart vara omringad av pjäser av motsatt färg. För att nå detta mål så turas spelarna om att utföra drag. Ett drag kan vara en av två olika händelser: en pjäs läggs till på brädet eller en existerande pjäs flyttas.



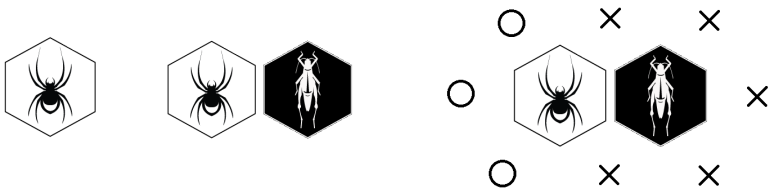
(a) Svart seger (b) Vit lägger till en pjäs (c) Vit flyttar en pjäs

Figur 1: Seger och drag. Bild 1a och 1c är ej realistiska situationer utan visar enbart koncept.

Första draget består av att spelare ett placerar ut en pjäs. Därefter placerar spelare två ut en pjäs bredvid den första. Nya pjäser som placeras ut får enbart placeras i förbindelse med spelarens egna pjäser, inga fientliga pjäser får finnas bredvid den valda positionen. Drottningen får senast placeras ut som fjärde pjäs.

I vissa Hive-turneringar är det förbjudet att placera ut drottningen först. Detta för att motverka att två spelare placerar drottningarna bredvid varandra och därmed kan spela mot oavgjort om läget börjar se illa ut. I implementationen är det förbjudet att placera ut drottningen som första pjäs.

Pjäser som finns utplacerade får flyttas så snart den egna drottningen placerats ut. De fem olika pjästyperna rör sig på olika sätt men de lyder också under allmänna regler. En pjäs får ej dela brädet i två delar, inte ens om den så sluter samman brädet när den nått sin slutdestination. I figur 3 så visas ett exempel på ett drag som bryter mot regeln. Det är vits tur att



(a) Vit börjar (b) Svart svarar (c) Möjliga drag för vit

Figur 2: Början av spelet. I bild 2c visar cirklarna platser där vit får placera ut en pjäs, kryssen är otillåtna platser.

göra ett drag i figur 3a. Drottningen är ute så vit får flytta på pjäser. Vit myra får ej flyttas då detta kommer leda till figur 3b. Detta trots att myrans slutdestination åter igen binder samman spelplanen.

(a) Vits drag (b) Delad Hive (c) Otillåtet drag

Figur 3: Exempel på regelbrott mot regeln om sammanhängande spelplan.

En pjäs måste också lyda regeln för naturliga rörelser. Detta innebär att den måste kunna skjutas in på sin plats utan att lyftas från bordet. Gräshoppan står dock helt över denna regel och skalbaggen berörs sällan av den. I figur 4a är det vits tur. Vit myra kan ej skjutas in på den centrala positionen i figur 4b då vit drottning och vit spindel är i vägen. Figur 4c visar en liknande situation som kan uppstå ett par drag senare från figur 4a. Vit kan i det fallet skjuta myran ett steg åt höger.

(a) Vits drag (b) Otillåtet drag (c) Liknande situation

Figur 4: Exempel på regelbrott mot regeln om naturliga rörelser samt en liknande situation där myran ej begränsas av regeln.

4.2 Pjäser

Det finns fem olika sorters pjäser: drottningen, spindeln, myran, skalbaggen och gräshoppan. Varje pjäs har ett unikt rörelsemönster.

4.2.1 Drottning

Drottningen rör sig ett steg i taget. Figur 5a visar hur ett exempel på möjliga drag för en drottning. Cirklarna representeras tillåtna positioner och kryssen otillåtna. Drottningen får ej placera sig ovanpå den vita myran.

4.2.2 Spindel

Spindeln rör sig tre steg i taget. Stegen måste alltid vara i angränsning till andra pjäser och en position får ej återanvändas. Figur 5b visar hur spindeln rör sig. Den har i exemplet två möjliga slutpositioner markerade med siffran 3. Spindeln kan ej avsluta sitt drag på position 1 genom att röra sig 1-2-1 då detta räknas som återanvänding av en ruta.

4.2.3 Myra

Myran kan röra sig till alla positioner den kan nå längs utsidan av brädet. Figur 5c visar vit myras rörelsemöjligheter. Den kan avsluta sitt drag på alla positioner markerade med cirklar. Kryssen visar otillåtna slutpositioner.

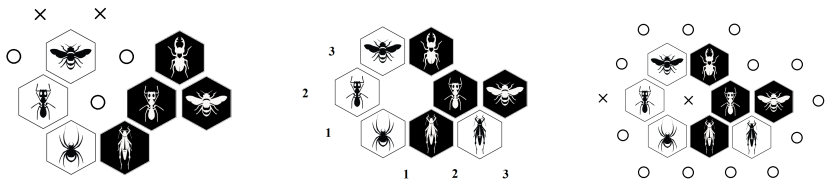
4.2.4 Skalbagge

Skalbaggen rör sig likt drottningen enbart ett steg i taget. Till skillnad från drottningen kan den placeras ovanpå andra pjäser och låsa fast dem. Det enda som hindrar skalbaggen är så kallade portar som består av två pjäser som ligger intill pjäsen och hindrar den från att röra sig framåt. I figur 6a visas skalbaggens möjliga flyttpositioner i form av cirklar. De blå cirklarna indikerar att skalbaggen kommer placeras ovanpå en annan pjäs.

Den vita drottningen samt den svarta myran agerar tillsammans port och hindrar skalbaggen från att i ett drag röra sig till den nedre vänstra positionen. Skalbaggen kan dock först flyttas ovanpå en av de två pjäserna för att nästa drag nå denna position.

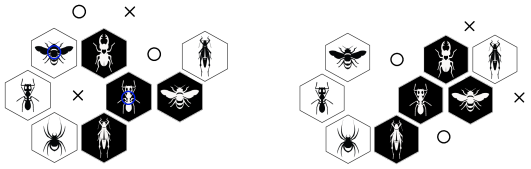
4.2.5 Gräshoppa

Gräshoppan kan enbart röra sig genom att hoppa över andra pjäser. Den flyttas i en riktning över alla pjäser som finns en viss riktning. Den kan ej röra sig i en riktning om det saknas angränsande pjäs där. I figur 6b visas exempel på hur gräshoppan rör sig. Den kan hoppa vänsterut över svart skalbagge eller neråt vänster över svart drottning. Den kan ej röra sig uppåt vänster eller neråt höger då den behöver pjäser att hoppa över.



(a) Vit drottning (b) Vit spindel (c) Vit myra

Figur 5: Exempel på rörelsemönster för drottning, spindel samt myra.



(a) Svart skalbagge (b) Vit gräshoppa

Figur 6: Exempel på rörelsemönster för skalbagge och gräshoppa.

5 Implementation av bassystemet

För att kunna implementera strategier så måste man först skapa ett system som kan hålla i pjäser och beskriva vad de får och inte får göra. Bassystemet kom att bestå av ett antal lager av klasser, detta för att se till att varje del skulle kunna förbättras och felsökas individuellt. För felsökning implementerades en klass som undersöker huruvida ett givet parti har följt reglerna eller ej.

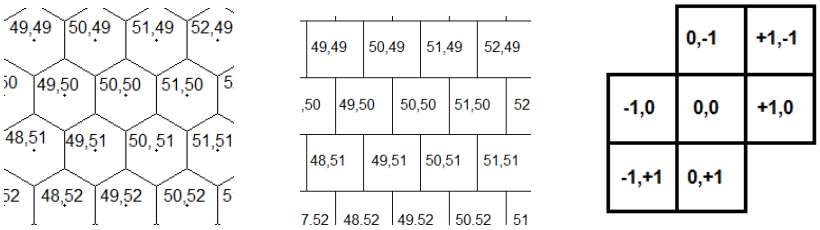
5.1 Piece

Den lägsta klassen är Piece. Piece håller i information såsom vilken spelare den tillhör, vad för slags pjäs det är och vilken position den ligger på. Pjäser kan också låsas positionsmässigt, det vill säga man får inte flytta på dem. Detta för att hantera när en skalbagge ligger ovanpå pjäsen.

Ett första problem som uppstod var hur representationen av den hexagonala spelplanen skulle hanteras. Initiallösningen blev att låta varje pjäs peka på sina granner. Om en pjäs lades till så gjordes det genom att ge en granne som referens samt önskad position bredvid den. Även om det funkade som planerat så gjorde det Piece väldigt svårt att felsöka då systemet saknar grafisk representation av hur pjäserna låg bredvid varandra.

Det andra försök blev att använda kartesiska koordinater. Genom att se på det hexagonala rutnätet som ett förskjutet rutnät bestående av kvadrater så kan man lätt beskriva positionen för en pjäs. En position (x, y) representeras internt som $x \cdot 100 + y$ då det möjliggör lättare jämförelse av positioner. Den första pjäsen får position $(50, 50)$. Värdet 50 baseras på att det är osan-

nolikt att pjäser kommer nå någon av kanterna $(0, y)$ eller $(x, 0)$ och därmed ha sönder brädet.



(a) Hexagonal spelplan (b) Förskjutna kvadrater (c) Förskjutningsmatris

Figur 7: Intern hantering av positioner.

5.2 Board

Board håller i alla pjäser samt information om var på brädet som det för stunden finns pjäser. Informationen består enbart av pjäser som inte har en beetle ovanför sig. Anledningen till detta designval är helt enkelt att se till att enbart aktiva pjäser finns tillgängliga för analyser. Board håller också koll på vems tur det är.

Bräden kan skapas på tre olika sätt.

- En pjäs kan som första pjäs initiera ett bräde.
- En pjäs kan läggas till på ett bräde
- En pjäs kan flyttas på ett bräde

Att skapa ett bräde genom att flytta en pjäs involverar många steg. Om det är en skalbagge som flyttas så måste det undersökas huruvida den var på en annan pjäs och därmed låser upp den. Skalbaggen kan också flyttas ovanpå en annan pjäs som då måste låsas.

5.3 Handler

Varken Piece eller Board har koll på hur pjäserna funkar med undantag för lite grundläggande funktionalitet för att låsa pjäser på en plats. Det är här Handler kommer in i bilden. Den implementerar rörelsemönster för alla de olika pjäserna och tillhandahåller funktioner för att skapa bräden, lista möjliga drag, kontrollera om en pjäs kan tas bort från brädet utan att dela det i tu samt många fler.

Alla rörelsemönster förutom skalbaggens var enkla att implementera. Reglerna för hur den får röra sig kan tyckas enkla men det finns ingen brist på fallgropar. Ett rörelsemönster tar en pjäs och returnerar en lista med alla

positioner som pjäsen kan gå till. Mönstret tar hänsyn till om pjäsen får röra på sig, om den kan flyttas utan att dela på brädet samt var andra pjäser finns.

Genom att gå igenom listan över pjäser och se vilken sorts pjäs det är så skapar Handler en komplett lista över möjliga drag. Beroende på om man vill ha dragen för båda spelarna eller enbart en så går det att ställa in vilka som ska sammanställas och returneras.

Handler används också för att skapa bräden baserat på förflyttningar och tillägg av pjäser.

5.4 Player

Player tar två strategier och kör dem mot varandra. För att förhindra att matchen aldrig körs klart så begränsas antalet drag med en variabel när matchen börjas. Att köra fast är mest sannolikt om man har någon mindre kraftfull strategi såsom slumpade drag eller en heuristik då de tenderar att göra galna drag. Player returnerar ett värde som påvisar utfallet.

5.5 Felsökning av struktur

För att kontrollera att alla giltiga drag genereras av programmet så laddades textfiler som beskriver matcher från Boardspace [6] ner från internet. Med hjälp av en parser så bröts filen ner till en serie kommandon. Matchen spelades sedan av Handler. Om den inte kände igen ett drag avbröts matchen och situationen undersöktes. På så sätt kunde giltiga drag som ej genererades av Handler läggas till.

För att kontrollera ogiltiga drag så spelades matcher manuellt och vid varje spelplan så skrevs alla potentiella drag ut. Dessa kontrollerades sedan för hand och vid behov korrigerades programmet.

6 Trädalgoritmer

För att impementera AI för brädspel används ofta sökalgoritmer. Algoritmen bygger på att man undersöker dragen som kan uppstå när de två spelarna turas om att göra drag. De söker till ett visst djup där djupet syftar på nivån i trädet man söker på.

Ett exempel på en trädalgoritm är Minmax [7]. Den väljer ut ett drag genom att växla mellan val av drag med minsta värde och högsta värde. I början skapas alla drag och dessa utvärderas av den minimerande spelaren. Den minimerande spelaren skapar i sin tur alla drag som kan uppstå från det nya brädet. Dessa drag skickas vidare till den maximerande spelaren. Processen fortsätter tills det önskade djupet har nåtts varvid dragen tilldelas ett värde av en evalueringsfunktion.

En algoritm som presterar bättre än Minmax är Alpha-Beta Pruning [8]. Den söker likt Minmax i trädet men den beskär delar av det för att snabba upp processen. Detta görs genom att utvärdera värdet för ett returnerat drag och se om det är sämre en ett drag som tidigare utvärderats i den delen av trädet. Om så är fallet avbryter Alpha-Beta sökningen för noden och returnerar dess värde.

```
int minmax(node root, int depth, bool max){
  if(depth == 0 || !root.hasChildren()){
    return eval(node);
 }

  int INFINITY = 1234567890;
  if(max){
    best = -INFINITY;
    for(each child of root){
      val = minmax(child, depth-1, false);
      best = max(val, best);
    }
  }else{
    best = INFINITY;
    for(each child of root) {
      val = minmax(child, depth-1, true);
      best = min(val, best);
    }
  }
  return best;
}

minmax(origin, depth, true) //basanrop
```

Tabell 1: Pseudokod för Minmax

6.1 Evalueringsfunktion

För att implementera sökalgoritmer för träd behövs en evalueringsfunktion. Den tar ett bräde och returnerar ett värde som beskriver hur bra det är för en spelare. Evalueringsfunktionen baserades på enkla koncept som sedan förbättrades i form av en turnering mellan olika versioner.

Slutversionen av evalueraren adderar respektive drar bort poäng för varje pjäs kring fiendedrottningen och vår drottning. Inga extra poäng delas ut baserat på vilka pjäser skalbaggar låst fast. Inga extra poäng ges för antalet pjäser ute på brädet. Pjäserna är viktade och poäng läggs till när en pjäs med högt värde, till exempel en drottning, kan röra på sig. Fientliga pjäser som kan röra på sig ger avdrag.

```
int alphabeta(node root, depth, int alfa, int beta, bool max){
  if(depth == 0 || !root.hasChildren()){
    return eval(node);
  }

  if(max){
    for(each child of root){
      alfa=max(alfa, alphabeta(child,depth-1,alfa,beta,false);
      if(beta<=alfa)
        return alfa;
    }
  }else{
    for(each child of root) {
      beta=min(beta, alpahbeta(child depth-1,alfa,beta,true);
      if(beta<=alfa)
        return alfa;
    }
}
alphabeta(origin, depth, -INFINITY, +INFINITY, true) //basanrop
```

Tabell 2: Pseudokod för Alpha-beta

7 Strategier

Strategierna kan delas in i två kategorier. De som enbart försöker hitta nästa drag baserat på hur de förändrar situationen för stunden samt de som letar många drag framåt för att undersöka nuvarande drag.

7.1 Random

Random gör precis vad det låter som. Den skapar en komplett lista över var nya pjäser kan placeras ut samt var gamla pjäser kan flyttas. Sedan slumpas ett drag ut ur listan och utförs. Inga begränsningar för hur dumma drag som kan göras finns och strategin kan förlora på eget drag.

Huvudsyftet med strategin är att ha något enkelt för andra strategier att spela mot. En strategi med lite mer vett bör med lätthet kunna slå Random. Om en strategi misslyckas med att slå Random så är det troligt att den har en bug.

7.2 Aggro

Aggro skiljer sig enbart lite från Random. Den sammanställer en lista över att möjliga drag och försöker sedan att få in fler pjäser runt fiendedrottningen. Misslyckas det så slumpar den ett drag.

7.3 Greedy

Greedy skapar en lista över evalueringsfunktionens värdering av drag den kan göra givet ett bräde. Den väljer sedan draget med det högsta värdet och utför det.

7.4 Minmax

Minmax följer till stor delspecifikationen som gavs tidigare. Den enda skillnaden är införandet av slump för att motverka att algoritmen alltid väljer samma drag. Slumpen infördes genom att välja ett slumpad drag med ett värde på minst $best - \delta$. Värdet δ sattes lågt.

7.5 Alpha-Beta pruning

Då Alpha-Beta avslutar en sökning så abrupt så kunde slump inte introduceras på samma sätt. Istället används Knuth Shuffle som blandar om vektorer [9]. Vektorn i fråga är alla de drag som skapas givet ett bräde. Genom att ändra på ordningen på hur dessa evalueras kan olika värden returneras från samma nod.

8 Resultat

För att utvärdera hur de olika algoritmerna presterar har de fått köra matcher mot varandra. Mätningar har också gjorts på den ökade tidsåtgången när djupet ökar för trädalgoritmer.

8.1 Branching factor

Branching factor för ett spel syftar på antalet möjliga drag per position. I schack är det till exempel cirka 35 [10]. Mätningarna för faktorn ligger runt 60-70 i början av spelet för att sedan gå ner till 20-30 när alla pjäser placerats ut.

8.2 Utvärdering av ett drag

För att undersöka Minmax och Alpha-Beta gjordes mätningar på tiden det tar att välja ett drag. Dessutom mättes också antalet noder som undersöks i samband med draget. I tabellen syftar siffran inom parentesen på djupet algortimen använde. Det saknas större mängder data på Minmax(4) och Alpha(5) då dessa tar väldigt lång tid på sig.

8.3 Matchresultat

För att undersöka om extra djup gav bättre resultat testkördes alla algoritmer mot varandra. Varje unikt par körde 100 matcher med en gräns på 200

Algoritm	Tid för ett drag (sek)	Utvärderade noder för ett drag
Minmax(2)	0-1	2.000-4.000
Minmax(3)	30-50	200.000-500.000
Minmax(4)*	2000-4200	1.300.000-5.400.000
Alpha(2)	0-1	200-600
Alpha(3)	2-5	20.000-50.000
Alpha(4)	15-30	100.000-200.000
Alpha(5)*	600-1300	5.000.000-12.000.000

Tabell 3: Data för drag. Algoritmer markerade med * har undermålig data på grund av körtiden.

drag och resultaten sammanställdes i tabellen nedan. I tabellen nedan visas all data.

För att läsa resultaten från en serie matcher väljer man en rad samt en kolumn och utläser de två värdena. Det första värdet är antalet vinster för strategin på raden, det andra värdet strategin på kolumnen.

Exempelvis säger värdet $(9, 75)$ från $(Gr, M2)$ att Greedy vann 9 gånger och Minmax(2) 75 gånger. De resterande 16 matcherna resulterade i slut på drag.

	Ra	Ag	Gr	M2	M3	A2	A3	A4
Ra	x	0, 36	0, 99	0, 94	0, 98	0, 85	0, 99	0, 87
Ag	36, 0	x	9, 26	0, 11	1, 9	0, 12	3, 10	1, 12
Gr	99, 0	26, 9	x	9, 75	12, 69	15, 51	4, 74	10, 76
M2	94, 0	11, 0	75, 9	x	4, 78	13, 19	6, 76	8, 55
M3	98, 0	9, 1	69, 12	78, 4	x	39, 6	18, 41	20, 52
A2	85, 0	12, 0	51, 15	19, 13	6, 39	x	22, 35	0, 27
A3	99, 0	10, 3	74, 4	76, 6	41, 18	35, 22	x	41, 57
A4	87, 0	12, 1	76, 10	55, 8	52, 20	27, 0	57, 41	x

Tabell 4: Resultat från matcher.

9 Diskussion

Basstrukturen är en stor faktor som styr hur snabbt strategier kan utvärderas. Den är inte tidsoptimerad utan bygger mest på tanken att det ska fungera korrekt hellre än gå snabbt. Strukturen kan mycket väl innehålla buggar som gjort att Hive testats med felaktiga regler

Evalueringsfunktionen som valdes var vinnaren i en liten grupp med funktioner. Med mer tid tillgängligt hade en vinnare kunnat korats bland tusen-

tals olika versioner med modifierade viktningar för de olika pjäserna och koncepten.

Testerna kördes på tre av KTH:s beräkningsmaskiner i flera dagar. De matcher som tog längst tid på sig krävde över två dygn för att köra klart. Förhoppningen var att köra tusentals matcher per strategi men det hade tagit flera veckor.

Alpha-Beta och Minmax med stort djup slog versioner med mindre djup, något man kan förvänta sig av trädalgoritmer. Alpha(4) var inte lika överlägsen som förväntat utan slog med nöd och näppe Alpha(3).

Aggro var involverad i en stor mängd matcher som slutade i att dragen tog slut. Troligen innehåller den en bug som ändrar på spelplanen på ett sätt den inte får göra.

Bristen på möjlighet att inspektera matcher grafiskt är ett stort problem när man utvecklar och testar AI. Även om det hade varit fullt möjligt att utveckla ett system baserat på ASCII hade det troligen tagit upp för mycket värdefull tid.

10 Slutsatser

Algoritmer som baseras på sökningar i träden över potentiella drag är lämpliga för att implementera strategier för Hive. Bassystemet måste vara snabbt och mycket tid måste avsättas för att hitta en stark evalueringsfunktion. Algoritmer såsom Greedy kan ge acceptabla resultat om tid är en begränsande faktor.

Alpha-Beta med djup tre-fyra hittar drag på relativt kort tid och tordes kunnas användas som AI mot spelare på lägre nivå.

Referenser

[1] "Board game." http://en.wikipedia.org/wiki/Board_game. Senast besökt 13 April 2014.

[2] "The turk." http://en.wikipedia.org/wiki/The_Turk. Senast besökt 13 April 2014.

[3] "Ibm research | deep blue | overview." https://www.research.ibm.com/deepblue/meet/html/d.3.2.html. Senast besökt 13 April 2014.

[4] "Go(game)." http://en.wikipedia.org/wiki/Go_(board_game). Senast besökt 13 April 2014.

[5] "Hive - a game buzzing with possibilities." http://www.gen42.com/downloads/rules/Hive_Rules.pdf. Senast besökt 13 April 2014.

[6] "Hive at boardspace.net." http://www.boardspace.net/english/about_hive.html. Senast besökt 13 April 2014.

[7] "Minimax." http://en.wikipedia.org/wiki/Minimax. Senast besökt 13 April 2014.

[8] "Alpha-beta pruning." http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. Senast besökt 13 April 2014.

[9] "Knuth shuffle." http://en.wikipedia.org/wiki/Knuth_shuffle. Senast besökt 13 April 2014.

[10] "chessprogramming - branching factor." https://chessprogramming.wikispaces.com/Branching+Factor. Senast besökt 13 April 2014.