

## **EN SCHACK AI BASERAD PÅ CASE-BASED REASONING MED GRUNDLIG LIKHET**

## **A CASE-BASED REASONING APPROACH TO A CHESS AI USING SHALLOW SIMILARITY**

Examensarbete inom huvudområdet Datavetenskap  
Grundnivå 30 högskolepoäng  
Vårtermin 2015

Johannes Qvarford

Handledare: Peter Sjöberg  
Examinator: Anders Dahlbom

# Sammanfattning

Schack är ett spel som ofta används för att undersöka olika tekniker inom artificiell intelligens (AI). I det här arbetet ställs frågan om det går att utveckla en AI-agent vars beslutsfattande är baserat på tekniken Case-based Reasoning (CBR) med grundlig likhet som spelar bättre med fallbaser baserade på bättre experter. En AI-agent har utvecklats som spelat ett antal partier mot sig själv med olika fallbaser baserade på olika experter. Efter att ha undersökt resultatet visade de sig att AI-agenten spelar så dåligt att den nästan aldrig lyckades vinna oavsett fallbas, vilket gjorde att det inte gick att rangordna dem efter skicklighet. I framtida arbeten kan andra likheter än grundlig likhet undersökas eller så kan det undersökas om en CBR-baserad schackspelande AI-agent kan spela schack bra.

**Nyckelord:** Schack, Case-based Reasoning, Artificiell intelligens, grundlig likhet.

# Innehållsförteckning

<b>1</b>	<b>Introduktion.....</b>	<b>1</b>
<b>2</b>	<b>Bakgrund.....</b>	<b>2</b>
2.1	Case-based Reasoning .....	2
2.1.1	Representation .....	2
2.1.2	Liknelse .....	3
2.1.3	Hämtning.....	3
2.1.4	Anpassning.....	4
2.1.5	Tidigare arbeten .....	4
2.2	Schack.....	4
2.2.1	Regler .....	4
2.2.2	Elo-rankning .....	7
2.2.3	Portable Game Notation.....	8
2.2.4	Schack AI:s historia.....	8
<b>3</b>	<b>Problemformulering .....</b>	<b>11</b>
3.1	Problemformulering.....	11
3.2	Metodbeskrivning.....	12
<b>4</b>	<b>Implementation .....</b>	<b>14</b>
4.1	Schack.....	14
4.1.1	Parti .....	14
4.1.2	Bräde .....	15
4.1.3	Läge .....	16
4.1.4	Drag.....	16
4.2	CBR.....	18
4.2.1	Problem och lösning.....	18
4.2.2	Fallbas.....	18
4.2.3	CBR-process .....	19
4.2.4	Hämtning.....	19
4.2.5	Anpassning.....	20
<b>5</b>	<b>Utvärdering.....</b>	<b>22</b>
5.1	Om undersökningen.....	22
5.2	Resultat .....	24
5.3	Analys.....	24
5.4	Slutsatser.....	25
<b>6</b>	<b>Avslutande diskussion.....</b>	<b>26</b>
6.1	Sammanfattning.....	26
6.2	Diskussion .....	26
6.3	Framtida arbete .....	28
	<b>Referenser .....</b>	<b>29</b>

Schack är ett spel som fått mycket uppmärksamhet inom forskning för artificiell intelligens. I det här arbetet presenteras hur passande

# 1 Introduktion

Schack är ett spel som fått mycket uppmärksamhet inom forskningsområdet AI. Redan på 50-talet presenterade Shannon (1950) ett förslag på hur en schackspelande AI-agent kunde fungera och sedan dess har många forskningsarbeten dedikerats till att förbättra hans ursprungliga design. Trots alla möjliga förbättringar som presenterats genom åren har den grundläggande tekniken som Shannon föreslog dock förblivit den samma och få utförliga arbeten om alternativa tillvägagångssätt har presenterats (Schaeffer 1991). AI-agenten undersöker vilket drag som är bäst genom att internt utföra alla möjliga kombinationer av nästa  $x$  antal drag, gradera de resulterade lägena efter en heuristik och utvärdera vilket drag som leder till det bästa läget, givet att motståndaren spelar optimalt.

I det här arbetet appliceras en alternativ teknik för att en schackspelande AI-agents beslutstagande som kan vara av användning för schackspel med varierbar svårighetsgrad. Tekniken heter CBR och är en problemlösningsteknik som går ut på att använda tidigare lösta problem för att lösa nya problem. Ett problem tillsammans med sin lösning kallas för ett fall. För att AI-agenten ska kunna ha några tidigare problem med lösningar att hänvisa till, samlas information in t.ex. genom att låta en mänsklig eller artificiell expert lösa ett antal exempelproblem. Problemen med expertens lösningar lagras i en fallbas som AI-agenten kan hänvisa till och söka i när den behöver lösa ett nytt problem. Kopplat till schack kan ett problem vara ett läge och en lösning det drag som ska utföras i läget.

Det har undersökts i det här arbetet om CBR med så kallad grundlig liknelse är passande för en schackspelande AI-agents beslutsfattande. Med passande menas att om AI-agenten använder två fallbaser baserade på två experter, så borde den spela bättre med fallbasen baserad på den bättre spelaren. En AI-agent skapades som kunde basera sina fallbaser på partier spelade av olika bra experter. Undersökningen utfördes genom att låta AI-agenten spela ett antal partier mot sig själv med alla kombinationer av ett antal fallbaser baserade på olika experter, för att se om den vinner oftare med fallbaser baserade på högre rankade spelare. Ett antal partier studerades även för att dra slutsatser om varför den betedde sig som den gjorde.

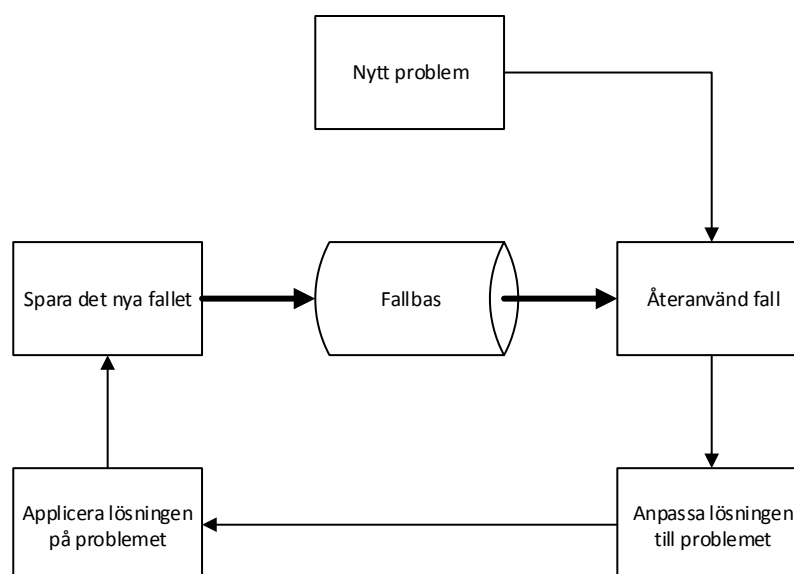
Rapporten är indelad i ett antal sektioner. I sektion 2 presenteras bakgrundsinformation till arbetet. Sektion 3 innehåller problembeskrivningen, den produkt som ska skapas för att besvara den och vilken undersökningsmetod som användes. I sektion 4 presenteras produkten och i sektion 5 utvärderas resultatet av undersökningen. Tills sist diskuteras resultatet i sektion 6.

## 2 Bakgrund

I denna sektion presenteras bakgrundsinformation om ämnen och termer som nämns i arbetet. I sektion 2.1 presenteras CBR och hur arbetet bygger på tidigare forskning kring CBR. Sektion 2.2 innehåller reglerna till schack, termer som ofta används inom schack och schack AI:s historia.

### 2.1 Case-based Reasoning

CBR är en teknik för problemlösning inom AI som är baserad på idén att använda lösningar på tidigare, liknande problem (Richter & Weber 2013). För att använda CBR måste en samling testfall med problem och deras respektive lösningar först samlas in. Hur testfallen samlas in varierar och kan ske genom observation av artificiella eller mänskliga experter. Lösningarna på problemen från experten behöver inte nödvändigtvis uppfylla några korrekthetskrav. Ett problem tillsammans med sin lösning beskrivs som ett fall, och en grupp fall kallas för en fallbas. När ett nytt problem ska lösas kan AI-agentens lösning baseras på hur en expert löste problemet eller ett liknande problem genom att konsultera en fallbas. När det fall har hittats vars problem är mest likt det nya problemet, kan fallets lösning anpassas till det nya problemet. Lösningen kan sedan appliceras. Tillsammans bildar det nya problemet med dess anpassade lösning ett nytt fall, som kan inkluderas i fallbasen och återanvändas. Denna process illustreras i Figur 1.



**Figur 1** Figur över processen för att applicera CBR.

#### 2.1.1 Representation

Problem kan representeras på olika sätt inom CBR, och de passar olika bra beroende på vad för sorts problem som ska lösas. En vanlig representation är en tupel av värden, där varje element överensstämmer med ett attribut (Richter & Weber 2013). Ett värde kan vara sammansatt av flera värden, och kan nästlas till ett arbiträrt djup. Till exempel kan ett värde vara en tupel, vars första element i sin tur kan vara en tupel av ett antal mängder osv.. Som exempel kan det finnas en AI-agent utvecklad för att estimerar priset på begagnade bilar baserat på priset av tidigare sålda begagnade bilar. En bil kan ha flera attribut men vissa attribut kan vara mer relevanta för att identifiera liknande, tidigare sålda bilar. T.ex. kan en

bil representeras med attributen modell, tillverkningsår och mätarställning men sakna attributet färg, för att en bils färg inte har haft en märkbar påverkan på en bils pris tidigare. Representationen illustreras i Figur 2. Andra möjliga representationer av problem är bilder, text och ljud (Richter & Weber 2013). Om det visar sig att bilar som ser liknande ut, har liknande beskrivningar eller låter liknande har en större chans att säljas för liknande pris, så kan det vara en lämplig problemrepresentation.

$([Modell], [Tillverkningsår], [Mätarställning])$

$(Audi\ A7, 2012, 17203\ mil)$

$(Bmw\ 320d, 2013, 707\ mil)$

**Figur 2** Ett exempel på hur en begagnad bil kan representeras som ett problem i CBR och två exempel på begagnade bilar.

### 2.1.2 Liknelse

Richter och Weber (2013) definierar problem som lika om de har liknande lösningar. Det är därför viktigt att identifiera vad som gör problem lika för att hitta liknande och passande lösningar till nya problem. Likhet är i denna benämning inte exakt, utan en grad på en skala. Ett problem kan vara sammansatt av flera värden och det finns olika sätt att beräkna likhet mellan sammansatta värden och primitiva värden.

Två vanliga sätt att beräkna likhet mellan primitiva värden är luddig matchning (engelska: *fuzzy matching*) och avstånd (Richter & Weber 2013). Luddig matchning associerar ett tal till ett par av värden, som är proportionerligt till graden av deras likhet. Avstånd associerar också ett tal till ett par värden, men avståndet är proportionerligt till inversen av graden av deras likhet. Ingen av metoderna är bättre, ibland kan det dock vara enklare att uttrycka likhet på ett eller annat sätt.

Likhet mellan sammansatta värden kan beräknas genom att aggregera deras elements likheter. Två samlingar är då lika om deras element är lika. Hammingsmätning (Richter & Weber 2013) går ut på att mäta likhet mellan två samlingar baserat på antalet element som är exakt lika. Med viktad hammingsmätning värdesätter olika element i samlingen olika, så att två samlingars likhet är mer beroende av vissa elements likhet än andra. Med metrisk likhet (Richter & Weber 2013) summeras istället elementens likhetsgrad. Hammingsmätning kan ge mindre precisa resultat än metrisk likhet, men är mer effektiv att implementera ur en prestandasynpunkt.

### 2.1.3 Hämtning

Syftet med att använda hämtning (engelska: *retrieval*) är att hitta en lösning vars problem är mest likt problemet som ska lösas. För att hämta lösningen används en hämtningsfunktion, vars krav kan variera från system till system. Det kan vara viktigt att den hämtade lösningen tillhör det mest lika problemet, men detta kan påverka systemets effektivitet. För att hitta det mest lika problemet och dess tillhörande lösning används ofta sekventiell sökning (Richter & Weber 2013). Eftersom problem kan likna varandra på olika sätt kan de ordnas efter olika kriterier och saknar en definitiv ordning. Av denna anledning är binärsökning mindre användbar för att söka efter fall. Om likhetsmetoden är prestandatung, kan problemen först gallras med en lättare likhetsmetod och sedan kan den tyngre likhetsmetoden användas på de kvarvarande problemen.

### **2.1.4 Anpassning**

När det mest lika problemet har upptäckts är det inte säkert att dess lösning är direkt applicerbar på det nya problemet. Lösningen kan därför behöva anpassas. Anpassning sker genom att applicera ett antal regler på lösningen för att få fram en ny lösning. En regel består av ett förvillkor och en handling som ska utföras om förvillkoret är sant. Handlingar delas in i två kategorier: transformationsbaserade och genererande handlingar (Richter & Weber 2013). Transformationsbaserade handlingar utgår från lösningen och byter ut delar av den för att anpassa den till det nya problemet. Genererande handlingar används ibland när lösningen har beräknats från dess tillhörande problem. Om lösningen inte är giltigt för det nuvarande problemet, så används samma teknik som användes för att beräkna lösningen från problemet, med den nya lösningen som grund. Detta är användbart när det är mycket dyrare prestandamässigt att beräkna hela lösningen från början, än att utgå från en nästan giltig lösning. Ett exempel är en färdbeskrivning från en plats till en annan. Det kan då vara enklare att utgå från en färdbeskrivning mellan två städer nära ursprungsplatsen och destinationen än att beräkna en helt ny färdbeskrivning.

### **2.1.5 Tidigare arbeten**

CBR har tidigare applicerats på spel med varierande resultat.

Wender och Watson (2014) har undersökt hur CBR kan användas för att mikrohantera enheter i realtidsstrategispelet (RTS) Warcraft 3. Deras CBR-baserade AI-agent styrde och attackerade med enheter baserat på tidigare liknande situationer. Likhetsmetoden i arbetet var hur nära de respektive enheterna var varandra i situationerna och hur lika deras hälsopoäng var. I arbetet testade de deras CBR-baserade AI-agent på mänskliga spelare och gemförde hur spelets inbyggda AI presterade på sin högsta skicklighetsnivå mot spelarna. De kom fram till att bara de bästa spelarna kunde besegra båda AI-agenterna, men ingen kunde besegra den CBR-baserade AI-agenten utan att förlora några enheter.

Bellamy-McIntyre (2008) presenterade hur CBR kan appliceras för att lära en AI-agent att göra bud i öppningar av bridge. Det upptäcktes dock att AI-agenten inte lyckades göra bra bud med sina givna händer, för att två händer som ansågs lika enligt likhetsmetoden ofta hade olika bud. Bellamy-McIntyre sammanfattade att CBR kanske inte är lämpligt för bridge-AI, eftersom att bridgespelare ofta använder ett regelverk för att bestämma vilket bud de ska göra för en given hand - de baserar inte sina bud på tidigare bud från liknande händer.

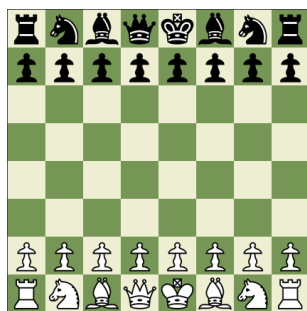
I Rubins (2013) arbete undersöks lämpligheten av att använda CBR för att utveckla en AI-agent som kunde spela olika Texas hold 'em varianter. AI-agenten kunde bestämma vilka bud den skulle ge baserat på vad andra spelare gjort i en pokertävling när de haft liknande kort på hand och på bordet. Den presterade bortom förväntningar och lyckades vinna Annual Computer Poker Competition (ACPC) från 2009 t.o.m. 2012.

## **2.2 Schack**

### **2.2.1 Regler**

Reglerna i schack har utvecklats genom åren och än idag spelas tävlingar med små variationer. Reglerna i denna sektion är baserade på reglerna som schackorganisationen FIDE (World Chess Federation) använder i deras anordnade tävlingar (World Chess Federation 2014c). Schack är ett turbaserat brädspel för två spelare där målet är att besegra sin motståndare.

Spelet utspelar sig på en 8x8-rutors spelplan, där varje spelare kontrollerar varsin armé av spelpjäser – vit och svart. I Figur 3 visas en bild av spelplanen i början av spelet.



**Figur 3** Bild av spelplanen i början av spelet.

Spelarna turas om att flytta spelpjäser i sina arméer. En spelare får bara flytta en spelpjäs per drag. Två pjäser av samma färg får inte ockupera samma ruta. Om en spelare flyttar en av sina spelpjäser på en ruta ockuperad av en motståndarpjäs, så fångas motståndarpjäsen och lämnar spelplanen för resten av partiet. Högst en pjäs i taget får ockupera en ruta, och en pjäs får generellt inte flytta till en ruta om andra pjäser står i vägen till rutan. Om en pjäs kan flytta till specifik ruta betraktas det som att pjäsen hotar rutan, eller pjäsen som står på rutan.

Bonden (♟) kan flytta sig ett steg rakt framåt (sett från den ägande spelarens håll), två steg rakt framåt om det är dess första förflyttning, eller ett steg diagonalt framåt om draget är ett fångande drag. Om en bonde når den sista raden sedd ur ägarens perspektiv, så kan den omvandlas till vilken annan pjäs som helst. Om en spelare flyttar en bonde två steg, så kan bonden betraktas som om den bara tog ett steg, om den fångas av en motståndarbonden nästa drag. Detta kallas *en passant* och är franska för "i förbifarten". En passant illustreras i Figur 4.



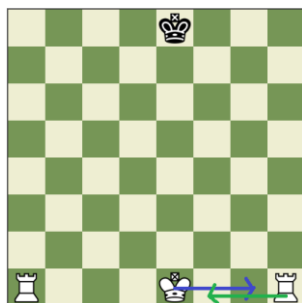
**Figur 4** Bild av en passant. Om vit flyttar sin bonde två rutor framåt kan den svarta bonden fånga den genom att flytta till rutan som den röda pilen indikerar.

Springaren (♘) kan flytta sig två steg horisontellt eller vertikalt, och ett steg på den resterande axeln. Springaren kan flytta till en ruta även om det finns pjäser som blockerar vägen. Löparen (♙) kan röra sig diagonalt. Tornet (♖) kan röra sig horisontellt eller vertikalt. Drottningen (♑) kan antingen röra sig horisontellt, vertikalt eller diagonalt. Kungen (♔) kan röra sig ett steg horisontellt, vertikalt eller diagonalt.

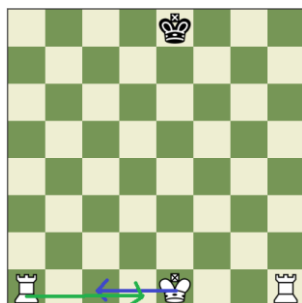
En spelare kan göra så kallad rockad med sin kung och ett torn, om det inte finns några pjäser mellan tornet och kungen, varken tornet eller kungen har flyttats förut, och varken rutan som kungen står på, passerar eller landar på är hotad. Rockaden går till så att kungen flyttas två



steg i tornets riktning, och tornet flyttas i kungens riktning så att den hamnar en ruta på andra sidan av kungens nya position. Figur 5 illustrerar hur detta kan se ut om den vita kungen gör rockad med det närmaste tornet, och Figur 6 illustrerar rockad med tornet längst bort. Detta kallas kort respektive lång rockad.

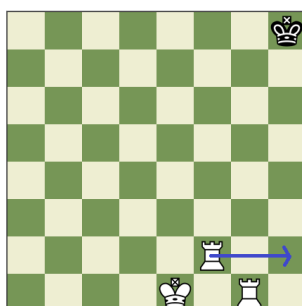


**Figur 5** Bild som visar hur pjäserna flyttas när vit gör kort rockad.



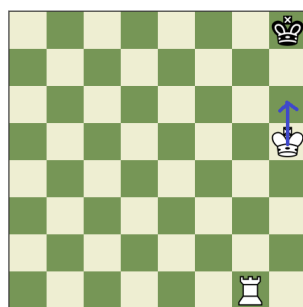
**Figur 6** Bild som visar hur pjäserna flyttas när vit gör lång rockad.

En spelare får aldrig göra ett drag som leder till att motståndaren hotar spelarens kung. Om en spelare gör ett drag så att motståndarens kung hotas kallas det för schack. Detta gäller även om kungen hotas av pjäser som skulle lämna sin kung hotad om de flyttade från sin ruta. Om motståndaren inte kan följa upp med ett drag som försätter kungen ur schack vinner spelaren, vilket kallas för schack matt. Ett exempel av schack matt visas i Figur 7.



**Figur 7** Bild som visar hur vit kan göra schack matt. Kungen hotas av tornet på andra raden, samtidigt som den inte kan flytta sig utan att fortfarande hotas.

Om motståndarens kung inte hotas, men samtidigt inte kan göra något drag utan att kungen hotas så blir det lika, vilket även kallas för remi. Ett exempel av remi visas i Figur 8.



**Figur 8** Bild som visas hur vit kan göra remi. Den svarta kungen hotas inte, men samtidigt kan den inte flytta sig någonstans utan att hotas av tornet eller den vita kungen.

Det finns två regler för att avbryta partier som inte leder mot vinst för någon av spelarna. Den första regeln säger att spelet är oavgjort om ett läge upprepar sig tre gånger under ett parti. Den andra regeln säger att spelet är oavgjort om ingen spelare flyttar en bonde på 50 drag.

### 2.2.2 Elo-rankning

Elo-rankning är ett sätt att ranka schackspelare relativt till varandra, namngett efter dess skapare Arpad Elo (Elo 1978). Rankningssystemet har sedan dess används av schackorganisationer som FIDE (World Chess Federation 2014a). Enligt Elo-rankningssystemet rankas spelare i form av poäng. Rankningar av spelare uppdateras kontinuerligt allt eftersom spelare spelar partier mot varandra i schacktävlingar. Hur mycket en spelares rankning påverkas av vinster och förluster beror på hur hög dess rankning är proportionerligt till dess motståndare, vilket avgör troligheten att en av dem kommer vinna. Troligheten att en spelare kommer besegra en annan spelare är baserat på skillnaden i deras rankning, men de exakta siffrorna är organisationsspecifika. T.ex. så anser FIDE att troligheten är en procenthalt mellan 0 % och 100 % (uttryckt som 0 till 1), och skillnaden i rankning är ett tal mellan -800 och +800 poäng (World Chess Federation 2014a). Om skillnaden i rankning är större eller lägre än +-800 poäng är troligheten 100 % att spelaren med högre rankning vinner.

Hur mycket en spelares rankning påverkas av ett parti beräknas på följande sätt:

- För varje parti får en spelare 1, 1/2 eller 0 poäng om den vann, gjorde remi eller förlorade.
- Troligheten att en spelare skulle vinna (mellan 0 och 1) subtraheras från antalet poäng den fick. Detta tal kan bli 1 som mest och -1 som minst.
- Detta tal multipliceras med en koefficient mellan 10 till 40, baserat på hur många partier spelaren tidigare spelat, dess nuvarande rankning och ålder.
- Det slutgiltiga (möjligen negativa) talet adderas med spelarens nuvarande rankning, för att få dess nya rankning.

Detta system ser till att spelare inte går ner mycket i rankning om de inte förväntas vinna; en spelare kan inte ens minska i rankning om den förlorar mot en spelare med mer än 800 i rankning. Likaså kan en spelare inte gå upp i rankning om den bara besegrar spelare med mycket lägre rankning. Om en spelare gör remi med en motståndare med högre rankning går spelaren upp i rankning och motståndaren ner, vilket leder till att de närmar sig samma rankning om det kontinuerligt gör remi mot varandra.

### 2.2.3 Portable Game Notation

I artikeln *Standard: Portable Game Notation Specification and Implementation Guide* (1994) beskrivs PGN som ett format för att spara och beskriva schackpartier. Formatet blev snabbt populärt och idag finns det tusentals allmänt tillgängliga sparade partier på t.ex. FIDE:s hemsida. Ett PGN-dokument kan innehålla ett antal partier och varje parti innehåller metainformation om partiet och de drag som utfördes i partiet. Informationen kan gälla när eller var partiet spelades och av vilka. Dragen skrivs med algebraisk notation (AN).

AN är en notation som beskriver drag kortfattat till den grad att de inte är tvetydiga. Raderna numreras från vits synvinkel med bokstäver från a till h, och kolumnerna med siffrorna 1 till 8. En ruta på spelplanen kan då beskrivas med dess tillhörande rad och kolumn t.ex. e4 eller a2. Drag har ett prefix med stor bokstav som beskriver vilken sorts pjäs som flyttades. N för springare, R för torn, B för löpare, Q för drottning, K för kung, medan bonde saknar prefix. Detta följs av positionen som pjäsen flyttades till. Exempel: e4, Nf3, Bb5. Om draget är ett fångande drag så sätts ett x framför rutan som pjäsen flyttades till. De drag som leder till schack har ett plustecken som suffix. Kort rockad representeras med "O-O" och lång rockad representeras med "O-O-O".

I de fall då ett drag är tvetydigt, t.ex. om två springare på e4 respektive e6 kan flytta till c5, så följs pjäsbokstaven av radkoordinaten eller kolumnkoordinaten beroende på vilken som kan uttrycka draget unikt (Nee5 är inte unikt i detta fall, medan N4e5 är det). Efter det sista draget i partiet visas resultatet 1-O, O-1, eller 1/2-1/2 om vit vann, förlorade, respektive gjorde remi med svart. I Figur 9 visas ett exempel av ett parti beskrivet i PGN.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia Yugoslavia|JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 {This opening is called the Ruy Lopez.}
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxb3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

**Figur 9** Ett schackparti i PGN-formatet. Notera att numreringen inte ökar för varje drag, utan varje par av drag.

### 2.2.4 Schack AI:s historia

Forskning inom schack AI tog fart efter att Shannon (1950) presenterade ett exempel på en schackspelande dator. Den var utformad så att den kunde bestämma ett drag att utföra baserat på vilket läge spelet var i och vilken färg den hade. Datorn försökte lista ut vilket drag som var bäst genom att internt göra alla möjliga kombinationer av drag ett antal drag i framtiden. Detta liknar förgreningen av ett sökträd, där varje drag är en kant och varje läge är en nod. Efter förgreningen så utvärderades lägena, och det drag som ledde till det garanterat bästa läget valdes. Med garanterat bästa menas att det ledde till det bästa läget givet att

motståndaren spelade så bra som möjligt. Sökalgoritmen som användes för att hitta det garanterat bästa läget kallas för minimax (McKinsey 1952). De flesta lägen går inte exakt att utvärdera om de inte är en vinst, förlust eller remi och därför används en heuristik. Heuristiken kan vara baserat på allt ifrån summan av spelarnas pjäsers värden till om det finns flera bönder av samma färg på samma kolumn (Shannon 1950). Ett läge representeras som en lista av 64 heltal, där varje heltal beskriver vilken pjäs av vilken färg som finns på en viss ruta, alternativt att rutan är tom.

Shannons exempel har förfinats på olika sätt genom åren. Bitboards är ett alternativt sätt att representera lägen i schack som är passande för att snabbt generera vilka drag som kan utföras ett givet läge (Slate & Atkin 1977). Det går ut på att lagra olika information av brädet i listor av 64 bitar, där varje bit representerar om en viss ruta uppfyller ett predikat specifikt för listan. Predikaten är ofta av formen "är rutan hotad av en pjäs av en viss typ" eller "har pjäsen på rutan en viss färg". Anledningen att bitboards är så användbara är att binära operationer mellan bitboards kan ge användbar information snabbt på hårdvara som har inbyggda operationer för 64-bitars register. T.ex. går det att sälla brädet med rutorna som de vita springarna kan hota genom att utföra xor på brädet av vita pjäser (Laramée 2000b). Alfabetasökning är en sökteknik för att undvika expansionen av grenar i ett träd, om det går att lista ut att motståndaren aldrig skulle besöka grenarna för att de inte är fördelaktiga nog (Frey 1983). Sökning används numera inte lika mycket under öppningar för att de har studerats i större utsträckning genom åren och har mer eller mindre "lösts" (Lincke 2001). Likaså har alla möjliga slutspel med ett lågt antal pjäser lösts genom att använda genererade slutspelsdatabaser (Heinz 1999). Trots alla förbättringar går det fortfarande inte att garanterat välja det bästa draget i varje läge med dagens schack AI. Problemet är att söktiden är exponentiellt proportionerlig till djupen som söks i sökträdet. På grund av detta finns det ett väldigt stort antal möjliga lägen att undersöka för ett lågt antal drag. Shannon (1950) estimerar att antalet möjliga lägen är i närheten av  $10^{54}$  och det skulle ta alldeles för lång tid för en dator att undersöka så många lägen.

Schack benämndes ofta förr som "bananflugan av AI" (McCarty 1990). Liknelsen syftar på att bananflugor började användas inom biologisk forskning om genetik för att det var så lätt att föda upp dem. Likaså har schack ansetts vara ett spel med enkla regler och ett tydligt men svårnått mål, som kan användas som testramverk för att undersöka hypoteser (Simon & Chase 1973). Problemet var att forskning inom schack AI fokuserade mer på specifika tekniker för att förbättra Shannons (1950) exempel, medan alternativa tekniker fick mindre uppmärksamhet (Schaeffer 1991; Ensmenger 2012). Därför anser Schaeffer (1991) att schack bör överges som testramverk för AI och fokus bör förflyttas till spelet go. Trots detta har ett antal alternativa tillvägagångssätt presenterats genom åren. Gould & Levison (1991) presenterar i sitt arbete en schackspelande AI-agent vid namn *Morph*. Morph utmärks av att den under ett antal spelade partier lär sig associera mönster av schackformationer med drag som bör utföras. Likt AI-agenten i detta arbete är den baserad på tanken att liknande problem har liknande lösningar. Morph skiljer sig dock från AI-agenten i detta arbete eftersom den börjar med väldigt lite kunskap, och lär sig genom att spela partier. Morph lärde sig att utföra olika attacker, men kunde inte alltid lista ut hur den kunde avsluta dem och vinna de partier den deltog i.

Innan persondatorer blev vanliga byggdes ofta maskiner som endast kunde spela schack. Dessa maskiner behövde en mänsklig operatör som kunde berätta för maskinen vad dess motståndare gjorde för drag och kunde utföra maskinens egna drag (Greenblatt, Eastlake &

Crocker 1969). Idag har schackmaskiner i stort bytts ut mot schackmotorer och operatörer mot grafiska och textbaserade användargränssnitt implementerade som mjukvara på persondatorer. Det finns ett antal kommunikationsprotokoll som används för att kommunicera mellan schackmotorer och användargränssnitt. XBoard (Mann & Muller 2009) är ett kommunikationsprotokoll baserat på användargränssnittet med samma namn och *Universal Chess Interface* (UCI) (Huber 2006) är ett nyare alternativ till XBoard.

Ända sedan schackmaskinen Deep Blue besegrade den dåvarande världsmästaren Kasparov (Campbell, Hoane & Hsu 2002) har schackmotorer och schackmaskiner mer och mer används för att utmana varandra istället för mänskliga spelare. De flesta schackmotorer kan nu för tiden besegra stormästare utan specialanpassad hårdvara. Pocket Fritz 4 är exemplarisk i att den implementerades på en telefon och samtidigt vann Mercusor Cup 2009, en av de enda tävlingarna på senare tid som tillåtit både mänskliga spelare och schackmotorer som deltagare (Chess News 2009). Varje år hålls World Computer Chess Championship (WCCC) av International Computer Game Association (ICGA) som är en tävling för schackmotorer. Tillställningen hålls inte bara för att utse en vinnare, men även för att dela teknisk kunskap med ICGA:s tidsskrift: ICGA Journal, som presenterar analyser av partier mellan spelare i spel som schack och go. Det finns även en annan tävling vid namn Thoresen Chess Engines Competition (TCHE) där de bästa schackmotorerna spelar mot varandra under långa perioder, vilket kan ses live på deras hemsida (<http://tcec.chessdom.com/live.php>). I januari 2015 blev Komodo 8 den nya världsmästaren bland schackmotorer när den uppnådde en Elo-rankning av 3320 poäng och besegrade den tidigare världsmästaren Stockfish 4, som hade en Elo-rankning av 3300 poäng (Anthony 2014). Detta är högt jämfört med den nuvarande mänskliga världsmästaren Magnus Carlsen, som i början av februari 2015 bara hade en Elo-rankning av 2865 poäng (World Chess Federation 2015).

## 3 Problemformulering

### 3.1 Problemformulering

AI för schack är ett forskningsområde som har fått mycket uppmärksamhet. Trots detta är schack ett olöst problem – det finns ännu ingen beprövat teknik att garanterat besegra en motståndare, som samtidigt går att implementera på existerande hårdvara. Det kan därför vara värt att undersöka hur väl alternativa AI tekniker kan appliceras på schack, i hopp om att närma sig en lösning. Det finns flera saker som tyder på att CBR kan vara lämpligt för en schackspelande AI-agents beslutsfattande. CBR är en problemlösningsteknik baserad på att använda lösningar för problem för att lösa liknande problem. Detta innebär att alla möjliga fall inte behöver finnas i dess fallbas för att AI-agenten fortfarande ska kunna lista ut vilket drag som är bäst att utföra i ett visst läge, givet en tillräckligt bra liknelsemetod och anpassningsmetod. Med CBR behöver det därför inte finnas fall för alla möjliga  $10^{54}$  lägen, eftersom flera av dem teoretiskt kan härledas. Det kan även visa sig att en CBR-baserad, schackspelande AI-agent kan spela lika bra som experten dess fallbas är baserade på, vilket kan användas för att utveckla schack-AI med varierbar eller flexibel svårighetsgrad.

I detta arbete har det undersökts om CBR med grundlig (i motsatt till djup) liknelse och anpassning passar för en schackspelande AI-agents beslutsfattande. Med grundlig menas att implementationen av liknelse och anpassning inte kräver djupare kunskaper om schack som vanliga strategier. Ett exempel på grundlig liknelse är om två lägen är lika om de generellt innehåller samma pjäser på samma rutor. Ett exempel på en strategi är skolmatt, där en spelare gör matt under mittspelet genom försöka fånga motståndarens kung när den gjort rockad. Två lägen skulle med djup liknelse kunna anses lika om de båda liknar uppbyggnad mot skolmatt och anpassningen av draget skulle ske så att AI-agenten försöker göra skolmatt. Problemet med att använda denna sorts likhet är att det finns så många olika strategier i schack som kan användas för att avgöra likhet och anpassning. Det begränsar även AI-agenten till strategier som tidigare upptäckts och definierats. Hsu (1991) har även berättat att användandet av rigida strategier gjorde att schackmaskinen *Deep Thought*, föregångaren till *Deep Blue* (Campbell, Hoane & Hsu 2002) spelade sämre. Det är inte säkert att grundlig liknelse ger bättre resultat, men det är ett mindre område och är därför enklare att undersöka till fullo än djup liknelse.

Att grundlig likhet passar för en CBR-baserad schackspelande AI-agents beslutsfattande innebär att AI-agenten generellt spelar bättre med fallbaser baserade på bättre experter. Om AI-agentens prestation med en viss fallbas kallas för ett beteende, kan det även uttryckas som att beteenden baserade på bättre experter bär spela bättre. Om så är fallet skulle det gå att med en liten felmarginal förutse hur bra AI-agenten kommer spela beroende på dess fallbas. Ett exempel där denna information är användbar är ett schackspel som har en AI-agent med justerbar svårighetsgrad. AI-agentens beslutsfattande skulle kunna vara baserad på CBR med grundlig likhet och den skulle kunna använda fallbaser baserade på bättre experter på högre svårighetsgrader.

En spelare anses spela bättre om den har en högre Elo-rankning. Det anses att ju större skillnaden i Elo-rankning är mellan två spelare desto större är sannolikheten att spelaren med högre Elo-rankning skulle vinna ett framtida parti mellan dem. FIDE (2014a) har definierat en tabell för att översätta mellan sannolikhet att vinna och skillnad i Elo-rankning. Ett beteende förväntas dock inte spela lika bra som experten det är baserad på och det är svårt att

förutse hur tabellen skulle kunna översättas från skillnader i Elo-rankning av experterna till skillnader i Elo-rankning för beteendena. T.ex. så anser FIDE (2014a) att om skillnaden i Elo-rankning mellan två spelare är 193 poäng så är det 75 % chans att den högre rankade spelaren kommer vinna i ett framtida parti, vilket inte är något som förväntas gälla för beteendena. Beteendena behöver inte heller uppfylla några skicklighetskrav gentemot Elo-rankade spelare, dvs. de måste inte spela lika bra, bättre eller sämre än en spelare med en viss Elo-rankning. AI-agenten bör dock vinna oftare med fallbasen baserad på den högre rankade spelaren, vilket kräver att beteendena kan spela så bra att de kan besegra varandra. Eftersom Elo-rankningar inte alltid stämmer exakt och ständigt förändras tillåts det att ett beteenden vars expert har lägre Elo-rankning vinner oftare mot ett beteende vars expert har högre Elo-rankning om skillnaden i rankning är mindre än 100 poäng. Detta innebär att i exemplet med programmet som hade en AI-agent med varierbar svårighetsgrad är resultatet av det här arbetet inte nödvändigtvis tillräckligt för att tekniken som undersöks ska vara av användning för programmet. Det är fullt möjligt att beteendena går att rangordna men att de alla spelar sämre än en nybörjare. Det krävs mer forskning om beteenden kan uppnå viss skicklighet som förväntas av folk som spelar mot dem. Problemet är av praktisk karaktär; det är inte relevant att veta om det skulle gå att lösa problemet om det fanns tillräckligt bra och mycket expertdata tillgängligt, bara om det är så i dagsläget. Om grundlig likhet passar för en CBR-baserad schackspelande AI-agents beslutsfattande så borde därför följande gälla:

- Det går att rangordna ett antal beteenden, där ett högre rangordnat beteende vinner oftare mot ett lägre rangordnat beteende givet att den största skillnaden i beteendenas experters Elo-rankningar är minst 100.
- Varje beteendes rangordning stämmer överens med rangordningen av dess experts Elo-rankning relativt de andra experternas Elo-rankningar.

Det går teoretiskt redan att skapa en AI-agent med tidigare beprövade tekniker som spelar schack perfekt; problemet är att det skulle ta flera år att beräkna vilket drag som bör utföras i varje läge. För att CBR ska vara av användning måste därför AI-agenten kunna implementeras på konsumenthårdvara och utföra drag under tidspress likt dagens schackmotorer. Tidskravet som AI-agenten förväntas följa är samma som spelare i FIDE-tävlingar förväntas följa. I FIDE-tävlingar får en schackspelare 90 minuter på sig att utföra sina första 40 drag (World Chess Federation 2014b).

## 3.2 Metodbeskrivning

För att besvara frågeställningen ska ett funktionsbibliotek skapas av en schackspelande AI-agent baserad på CBR, som kan använda olika fallbaser. Anledningen att ett funktionsbibliotek används är att det skulle vara för tidskrävande att för hand analysera den stora mängden expertdata som AI-agenten kommer använda och dra slutsatser om beteendenas prestation.

För att undersöka om beteendenas skicklighet är relativ till skickligheten av deras respektive experter behöver olika beteenden tävla mot varandra. För varje parti får ett beteende poäng gentemot ett annat beteende; 1 om det vann,  $\frac{1}{2}$  om det blev lika och 0 om det förlorade. Efter att alla har spelat ett antal partier mot varandra ska poängen räknas och beteendena rangordnas. Om CBR med grundlig liknelse och anpassning passar för en schackspelande AI-agents beslutsfattande måste följande gälla:

- Det går att rangordna beteendena, där ett högre rangordnat beteende alltid får högre poäng mot ett lägre rangordnat beteende givet att den största skillnaden i beteendenas experters Elo-rankningar är minst 100.
- Varje beteendes rangordning stämmer överens med rangordningen av dess experts Elo-rankning relativt de andra experternas Elo-rankningar.

AI-agenten ska kunna konfigureras med vikter så att olika aspekter av likhet värdesätt mer andra. Ett antal konfigurationer av vikter ska användas för att undersöka om någon eller några aspekter av likhet ger bättre resultat. Det räcker med att en konfiguration av vikter ger ett positivt resultat för att resultat av problemet ska anses som positivt.

Expertdata från olika experter ska komma från drag som experten utfört i olika lägen i tidigare partier. En allmänt tillgänglig databas av sparade schackpartier ska användas som källa för expertdata.

Ett möjligt problem med undersökningen är att det inte är säkert att en delmängd av en spelares historik av spelade schackpartier visar hur den fick sin rankning. Om en större andel av de partier som experten vunnit gentemot de partier som den förlorat skulle finnas tillgängliga kan AI-agenten spela bättre än förväntat med expertens fallbas. Motsatsen är lika möjlig. Det finns även en risk att olika experter har olika många sparade partier procentuellt gentemot hur många partier de spelat för att ha fått sin rankning. Detta skulle kunna ge dem en oproportionerligt mindre fallbas gentemot andra än de teoretiskt skulle kunna haft, än om alla deras partier varit tillgängliga i databasen.

Det finns en risk att det kan ta för lång tid att samla in tillräckligt mycket information i undersökningen för att dra en pålitlig slutsats. Ett alternativt sätt att utföra undersökningen är att låta experter observera några få partier spelade av respektive beteende och ranka dem efter experternas utlåtanden. Denna metod har dock nackdelen att det kan vara svårt att göra expertutlåtanden baserat på så lite information.

Att bara ta hänsyn till resultatet av partierna kan ge en falsk bild av hur väl ett beteende presterar. Det går att basera beteendens rangordning på fler viktiga aspekter som hur bra pjäsbyten den gör (om den offerar en bonde för en drottning t.ex.), hur snabbt den avancerar sin armé eller hur bra den spelar i öppningar, mittspel och slutspel. Av denna anledning kommer flera partier analyseras för att upptäcka mönster i hur AI-agenten beter sig och varför, men det kommer inte påverka deras rangordning i resultatet av undersökningen.



## 4 Implementation

Produkten som beskrevs i metoden har implementerats som ett funktionsbibliotek skrivet i programmeringsspråket C# 4.0 och kompilerat mot plattformen .NET 4.5. Produkten är en schackspelande AI-agent, men att bara presentera hur den är implementerad skulle ge en otydlig bild av sammanhanget den kan användas i och därför presenteras mer än bara AI-agentens implementation. Det delar som presenteras men inte är del av AI-agenten kommer användas i undersökningen, som utförs genom en programkörning.

De flesta datastrukturer i biblioteket kan inte muteras, vilket innebär att för att t.ex. utföra ett drag på ett läge så måste ett nytt läge skapas. Detta har gjort det enklare att skriva korrekt kod och tagit bort många svårigheter med att utföra beräkningar parallellt. Bara på de områden i koden där prestanda är kritiskt tillåts mutation. Enhetstest med testbiblioteket NUnit 3.0 har skrivits för att försäkras om att koden ständigt fungerar, trots förändringar och optimeringar. De flesta testerna berör bara implementationerna av schackalgoritmerna. Testerna garanterar inte att alla algoritmer som de testar korrektheten av fungerar, men det tros vara en liten risk att både algoritmerna och enhetstesten har skrivits fel.

För att beskriva funktioner som används av produkten används matematiska funktionsdefinitioner och signaturer som definierat av Holmos (1960). Funktionssignaturer är av formen  $F: A \rightarrow B$  där  $F$  är namnet på funktionen,  $A$  är domänen (mängd av alla möjliga argument) och  $B$  är målmängden (alla möjliga resultat är en delmängd av målmängden). En konvention i rapporten är att namn på mängder och funktioner börjar med stor bokstav medan element, argument och variabler börjar med liten bokstav.

### 4.1 Schack

I den här sektionen beskrivs implementationen av diverse datastrukturer och algoritmer som krävs för att implementera schackrelaterade delarna av AI-agenten och undersökningen.

Det finns inte många avvägningar för val av schackalgoritmer och datastrukturer. Det går inte att implementera schack mer eller mindre korrekt på grund av dess stränga regler, så de enda avvägningarna att ta hänsyn till är prestanda, minneskrav och enkelhet att implementera, förstå och utbygga. I sin artikelserie beskriver Laramée (2000a; 2000b; 2000c; 2000d; 2000e; 2000f) olika datastrukturer och algoritmer som kan användas för att implementera en schackspelande AI-agent. Det framhävdes av hans presentation att det generellt finns vissa schackrelaterade datastrukturer och algoritmer som är enkla att implementera och andra som kan implementeras mer effektivt. I det här arbetet används främst de enkla datastrukturerna och algoritmerna eftersom det förväntas att AI-agenten hinner bestämma drag med de tidskrav som ställs på den, oavsett val av schackrelaterade algoritmer och datastrukturer. Det innebär inte att implementationerna av de valda algoritmerna inte kan optimeras, bara att produkten antagligen inte kommer hindras av dem för att uppfylla sina prestandakrav. Valen lär ha mindre prestandapåverkan än för andra vanliga schackspelande AI-agenter, eftersom deras flaskhals är generering och utförande av drag i sökträd (Slate & Atkin 1977), medan det förväntades att produktens flaskhals är hämtningen av fall från AI-agentens fallbas.

#### 4.1.1 Parti

Schack är ett spel mellan två agenter som har ett definitivt slut. Agenterna turas om att göra ett beslut i taget i ett parti. Efter varje beslut kan resultatet av partiet vara bestämt eller inte.

Partiet fortsätter tills resultatet är bestämt. Ett resultat är ett element i mängden  $Resultat = \{Obestämt, Lika, VitVinst, SvartVinst\}$ , och resultatet av ett läge kan ges av funktionen  $ResultatAvLäge: Läge \rightarrow Resultat$ . Varje parti börjar alltid i samma läge men ändras beroende på agenternas beslut. Ett drag är ett beslut som kan utföras i ett positivt antal möjliga lägen och mängden av alla drag representeras med symbolen  $Drag$ . Alla drag kan inte utföras i alla lägen, så funktionen  $GiltigaDrag: Läge \rightarrow \{drag_1, drag_2 \dots drag_n\} \subset Drag$  ger en mängd av alla giltiga drag som kan utföras i ett givet läge. Funktionen  $UtförDrag: Läge \times Drag \rightarrow Läge$  kan ge ett läge där ett visst giltigt drag har utförts på det givna läget. Genom att utföra ett drag på det resulterade läget av  $UtförDrag$  och sedan på resultatet av det draget osv. kan ett parti spelas. Agenterna kan ses som funktioner som översätter ett läge till det giltiga drag som agenten anser bör utföras i läget, dvs. funktionssignaturen för agenterna är:  $Agent: Läge \rightarrow Drag$ . Med detta i åtanke kan ett schackparti implementeras med följande algoritm:

1. Initiera det nuvarande läget till det ursprungliga läget och låt den nuvarande agenten vara agenten som spelar vit.
2. Låt den nuvarande agenten besluta ett drag att utföra givet det nuvarande läget.
3. Utför draget på det nuvarande läget.
4. Om resultatet av partiet är obestämt, byt ut den nuvarande agenten mot den andra och gå till steg 2, annars är algoritmen slutförd.

#### 4.1.2 Bräde

Ett schackbräde representeras med en 64-elementslista av bytes, där varje element representerar vad som finns på rutan indikerad av dess plats listan.  $a_1$  är den första rutan,  $b_1$  den andra,  $a_2$  den nionde och  $h_8$  den 64:e, dvs.  $(rad - 1) * 8 + kolumn$  avgör vilken plats i listan som en viss rutas innehåll är lagrat på. Mängden giltiga index för rutor kan då definieras som  $RutIndex = \{n \mid n \in \mathbb{N} \wedge 1 \leq n \leq 64\}$ . Det som kan finnas på en ruta är en pjäs med en färg och typ, eller avsaknaden av en pjäs. För att skilja på olika sorters pjäser och avsaknaden av en pjäs används heltal där varje unikt värde representerar en unik pjäs- och färgkombination alternativt avsaknaden av en pjäs, som Shannon (1950) gjorde i sin beskrivning av en schackmaskin. I det här arbetet är heltalet en byte där den första biten representerar om det finns en pjäs på rutan, de tre följande bitarna representerar pjästypen och den femte biten representerar färgen på pjäsen. Eftersom tre bitar används för att representera typen kan det finnas upp till åtta olika pjästyper, men det krävs bara sex stycken för schack. De är bonde, torn, springare, löpare, drottning och kung, där bonde representeras av  $(000)_2$ , torn representeras av  $(001)_2$  osv.. Mängden  $RutInnehåll$  kan därför beskrivas som alla unika positiva heltal som får plats i fem bitar och bildar det tomma rutinnehållet eller en giltig färg och typ kombination. Notera att det tomma rutinnehållet antar att de resterande bitarna för färg och typ är noll, så det finns bara ett heltal som representerar ett tomt rutinnehåll. Funktionen  $InnehållerPjäs(Rutinnehåll) \rightarrow Boolean$  säger om ett givet rutinnehåll är en pjäs. Funktionen  $PjäsFärg(Rutinnehåll) \rightarrow Färg$  ger färgen för ett rutinnehåll givet att det innehåller en pjäs, där  $Färg = \{Vit, Svart\}$ .  $PjäsTyp(Rutinnehåll) \rightarrow Typ$  ger pjästypen för ett rutinnehåll givet att det innehåller en pjäs, där  $Typ = \{Bonde, Torn, Springare, Löpare, Dam, Kung\}$ . Funktionen  $InnehållPåRuta(Bräde, RutIndex) \rightarrow RutInnehåll$  ger innehållet på en viss ruta för ett visst bräde.

Enligt Chess Programming Wiki (CPW) (2013a) kan vanliga brädrepresentationer delas in i två kategorier: pjäscentrerade och rutcentrerade. Ett exempel på en pjäscentrerad

representation är pjäslistor. Pjäslistor är ett par av listor, en per spelare, som innehåller par av pjästyp och rutindex, som indikerar vilka rutor som respektive spelares pjäser står på (Chess Programming Wiki 2013b). En annan pjäscentrerad representation är bitboards (Slate & Atkin 1977) som presenterades i sektion 2.2.4. En rutbaserad representation är 8x8 Board (Shannon 1950; Chess Programming Wiki 2014a) som representeras med en 64-elementslista av heltal likt den representation som används i det här arbetet. En annan rutbaserad representation är 10x12 Board (Chess Programming Wiki 2014b) som representeras med en 120 elementslista. Anledningen att använda ett större bräde är att springardrag alltid kan utföras även om de landar utanför brädet. Detta innebär att dessa drag kan sällas bort i ett senare skede av draggenereringen, vilket kan ge en prestandavinst över 8x8 Board. Ingen av dessa har en speciell fördel över de andra förutom potentiella minneskrav att lagra och hur effektiv draggenererings- och dragutförandealgoritmen kan implementeras. Som nämnt tidigare förväntas prestandaförbättringarna vara oviktiga för detta arbete och därför valdes 8x8 Board för att den ansågs lättast att implementera.

### 4.1.3 Läge

Ett läge i schack representeras med ett bräde och diverse bokföringsinformation för speciella regler. Representationen av bokföringsinformationen är baserad på representationen som Shannon (1950) använde i sitt arbete. Det senaste draget sparas för att avgöra om en passant är giltigt. Det hålls reda på om kungarna och tornen har flyttat på sig för att avgöra om rockad är giltigt. Ett heltal används för att hålla reda på antalet drag sedan en bonde flyttade sig för att upptäcka om ett läge är remi efter 50 drag utan en flyttad bonde. Alla tidigare brädformationer sparas i en ordbok från lägen till antal gånger de skett under partiet för att upptäcka om en brädformation upprepat sig tre gånger vilket skulle innebära remi. Efter att ett drag har utförts indexeras ordboken med den nya brädformationen och ökar antalet kopplat till den. Om det är första gången brädformationen skett sätts antalet till ett istället. Läget håller även reda på färgen av den nuvarande spelaren och byter färg efter varje drag. Med denna information går det att avgöra resultatet i ett visst läge med följande algoritm:

1. Är den nuvarande spelaren i schack och har inga giltiga drag att utföra? I så fall är resultatet att den nuvarande spelaren har förlorat.
2. Annars, är den nuvarande spelaren inte i schack men den har samtidigt inga giltiga drag? I så fall är resultatet remi.
3. Annars, har ingen bonde flyttats på 50 drag? I så fall är resultatet remi.
4. Annars, har det nuvarande brädesformationen uppstått totalt tre gånger under partiet? I så fall är resultatet remi.
5. I annat fall är resultatet obestämt.

### 4.1.4 Drag

Ett drag representeras med två rutindex, det första för att identifiera vilken pjäs på brädet som ska flyttas och det andra för att identifiera vart på brädet pjäsen ska flyttas till. Utöver det kan ett drag innehålla en pjästyp som indikerar vilken pjästyp en bonde ska omvandlas till när den når sista raden sedd från ägarens perspektiv. Funktionen *FrånRuta(Drag) → RutIndex* ger rutindex för pjäsens ruta och funktionen *TillRuta(Drag) → RutIndex* ger rutindex för rutan som pjäsen ska flyttas till. *ÄrOmvandling(Drag) → Boolean* säger om ett drag är en omvandling och *OmvandlingPjästyp(Drag) → Typ* ger pjästypen att omvandlas till för ett drag, givet att draget är en omvandling. För att skapa drag kan funktionerna

*SkapaFrånTillDrag(RutIndex, RutIndex) → Drag*

och

*SkapaFrånTillOmvandlingDrag(RutIndex, RutIndex, Typ) → Drag* användas.

Rockad och en passant representeras med två rutindex likt andra drag, som gör dem unikt identifierbara som rockad och en passant men bara i specifika lägen. En passant representeras som en förflyttning för bonden till rutan som motståndarens bonde hoppade över. Rockad representeras som en förflyttning av kungen två steg åt vänster eller höger för lång respektive kort rockad. Eftersom kungen kan flytta på sig och ersättas med en annan pjäs på dess ursprungliga ruta, kan samma drag vara rockad i ett läge och en vanlig förflyttning eller fångande drag i ett annat. T.ex. om den vita kungen är på e1 kan den göra rockad med draget e1c1; i ett annat läge där det står en vit drottning på e1, tolkas draget istället som en förflyttning av drottningen till c1. I den senast nämnda situationen är rockad dock inte giltigt för att kungen flyttat på sig, så det går att härleda att draget inte är rockad. Funktionen *ÄrRockad(Drag, Läge) → Boolean* säger om ett drag är rockad i ett visst läge och funktionen *RockadSida(Drag, Läge) → RockadSida* ger vilken sida rockaden är på där *RockadSida* = {Vänster, Höger}. Funktionen *ÄrEnPassant(Drag, Läge) → Boolean* säger om ett drag är en passant i ett givet läge. Att låta rockad och en passant dela representation med förflyttningar och fångande drag har haft fördelen att förenkla definitionen av likhet mellan drag, vilket används inom anpassningsfunktionen i sektion 4.2.5.

En trippel av frånrutindex, tillrutindex och möjlig omvandlingstyp är en vanlig representation av drag inom schackprogrammering (Chess Programming Wiki 2014c). Om en rockad- och en passantindikator inte lagras explicit i draget så får det plats i två bytes; de första sex bitarna lagrar frånrutindex, nästa sex lagrar tillrutindex följande tre lagrar omvandlingspjästypen och den sista biten är ledig. Även om det förväntas att många drag kommer behöva lagras i minnet samtidigt när AI-agenten använder stora fallbaser, så lär den totala storleken vara mycket mindre än minnet som krävs för alla bräden. Utöver frånrutindex, tillrutindex och omvandlingstyp kan även pjäsen som flyttades lagras, och pjäsen som eventuellt fångades. På så vis behövs inte tillhörande läge finnas tillgängligt för att få den informationen, vilket kan vara användbart för att ångra drag i produkten, om detta skulle krävas.

För att generera giltiga drag måste följande steg utföras:

1. Sök igenom brädet efter pjäser med samma färg som den nuvarande spelaren.
2. För varje pjäs, placera alla drag som pjäsen kan utföra i en lista, ignorera om det finns pjäser på vägen som blockerar och skulle göra draget ogiltigt.
3. Ta bort de drag som är ogiltiga för att de blockeras av pjäser på vägen.
4. Ta bort de drag som är ogiltiga för att motståndaren kan fånga den nuvarande spelarens kung nästa drag. För att undersöka om så är fallet, utför draget, generera drag för motståndaren men hoppa över steg 4 och 5 och undersök om den nuvarande spelarens kung fångas i ett av dragen. Om så är fallet är draget ogiltigt.
5. Ta bort rockadrag om rutan som kungen måste korsa är hotad av motståndaren. Om det inte är giltigt för kungen att gå till den korsade rutan är rockaden inte giltig heller.
6. De resterande dragen är de giltiga dragen.

Den här draggenereringsalgoritmen är vad CPW definierar som en laglig draggenereringsalgoritm (Chess Programming Wiki 2015). Utöver lagliga draggenereringsalgoritmer finns det pseudolagliga draggenereringsalgoritmer där det inte undersöks om ett drag lämnar kungen hotad, utan det undersöks i dragutförningsalgoritmen.

Det finns optimerade algoritmer för att generera drag i speciella situationer; om t.ex. kungen är hotad är de enda giltiga dragen dem som fångar pjäsen, blockerar den förutsatt att den inte är en springare eller flyttar kungen ur vägen. Eftersom AI-agenten inte genererar många drag i sökträd till skillnad från de flesta schackmotorer så är pseudolaglig draggenerering och flera andra draggenereringstekniker inte användbara. Istället utförs ett drag i taget och varje drag måste vara giltigt. Speciella optimeringar kan göras till algoritmen för att förbättra dess prestanda i vissa situationer, men detta anses inte behövas.

För att utföra en förflyttning eller ett fångade drag så töms källrutan (innehållet byts ut av avsaknaden av pjäs) och destinationsrutan skrivs över med det innehållet som fanns på källrutan. En passant utförs likadant, förutom att rutan som den fångade bonden stod på töms. Rockad utförs som en förflyttning av kungen två steg till vänster och en förflyttning av det vänstra tornet tre steg till höger för lång rockad, eller en förflyttning av kungen två steg till höger och en förflyttning av det högra tornet två steg till vänster för kort rockad. Efter det sker diverse bokföring för att avgöra giltigheten av drag och resultatet av partiet. Det beskrevs mer detaljerat i sektion 4.1.3.

## 4.2 CBR

De CBR-relaterade delarna av produkten beskrivs i denna sektion. De inkluderar domänen av problem som CBR-modellen löser, fallbasrepresentation, översättning från pgn-partier till en fallbas, och en algoritm för att lösa ett problem.

### 4.2.1 Problem och lösning

En essentiell del av CBR-modellen är domänen av problem som försöker besvaras (Richter & Weber 2013). I arbetet behöver den schackspelande AI-agenten lösa problem av formen "Vilket drag ska utföras i ett visst läge?". Det anses då rimligt att ett problem representeras som ett läge och en lösning som ett drag. De är dock inte den enda domän av problem som CBR-modellen kan tänkas användas till. Den kan besvara problem som "vilka x antal drag ska utföras i sekvens av den nuvarande spelaren från ett visst läge?", men det är möjligt att inte alla framtida drag kommer vara giltiga beroende på vad motståndaren gör. Schackpartier påverkas dessutom väldigt mycket beroende på vad motståndaren gör; om motståndaren lämnar sin drottning öppen att fångas är det ofta bra att fånga den, men att motståndaren skulle göra en sådan tabbe är svårt att förutse. Av denna anledning är det inte användbart att beräkna drag i förväg och det tros inte heller replikera avsikterna av experterna på ett bra sätt.

### 4.2.2 Fallbas

Ett fall representeras som ett par av ett problem med dess lösning. Fall lagras sekventiellt som en lista i en fallbas. För att översätta en lista med pgn-partier för en viss expert till en fallbas används följande algoritm:

1. Skapa en tom lista med fall.
2. Ta ett parti från listan av partier. Om det inte finns några partier kvar, använd listan med fall för att bilda en fallbas och avsluta algoritmen.
3. Identifiera vilken av spelarna som är experten genom att undersöka namnet på den vita spelaren och den svarta spelaren.
4. Spela partiet och för varje drag som experten utför, lagra draget och läget det utfördes i som ett fall i listan av fall.
5. Gå till steg två.

Som en optimering utförs algoritmen parallellt där varje tråd samlar fall från ett antal tilldelade partier. En annan optimering är att bara den information i ett läge som används av AI-agenten lagras i fallbasen, vilket är brädet och färgen på spelaren som utförde draget. När algoritmen utförs lagras identiska fall i fallbasen lika många gånger som de uppstår under partierna. Även med optimeringen är resultatet av algoritmen deterministiskt; samma partier i samma ordning ger samma fall i samma ordning. Samma partier i olika ordning ger samma fall i olika ordning. Ordningen påverkar dock inte vilket fall som väljs enligt hämtningsalgoritmen, vilket beskrivs i sektion 4.2.4.

### 4.2.3 CBR-process

Som nämnt i sektion 2.1 finns det en process för att lösa problem med CBR (Richter & Weber 2013). Denna process används även av AI-agenten för att lösa problem. Processen kan beskrivas såhär:

1. Hämta ett fall från fallbasen, där fallets problem är mest likt problemet som ska lösas.
2. Anpassa fallets lösning till problemet som ska lösas.
3. Skapa ett fall av problemet som skulle lösas tillsammans med den anpassade lösningen och lagra det i fallbasen.
4. Den anpassade lösningen är lösningen på problemet.

AI-agenten utför alla steg utom tre, för att det inte ska undersökas i det här arbetet om AI-agenten kan bli bättre på att spela schack allt eftersom den spelar flera partier. I den här sektionen ska steg ett och steg två förklaras djupare.

En lista med 3 vikter används för att konfigurera AI-agentens beslutsfattande. Alla vikter är reella tal inom  $[0,1]$ . Vissa delar av AI-agenten är beroende av slump för att den ska fatta varierade beslut. AI-agenten använder därför en intern pseudoslumpgenerator initierad av ett slumpfrö. Slumpfröet är ett heltal som påverkar sekvensen av slumpmässiga tal som pseudoslumpgeneratoren producerar. Funktionen  $SkapaAiAgent: Fallbas \times ViktLista \times Slumpfrö \rightarrow AiAgent$  skapar en AI-agent (en funktion från läge till drag) givet en fallbas, en viktlista och ett slumpfrö.

### 4.2.4 Hämtning

Hämtning av ett fall från en fallbas går till på följande sätt:

1. Skapa en lista av fall med de mest lika problemen.
2. Ta ett fall från fallbasen som inte undersökts än.
3. Jämför likheten mellan problemet som ska lösas och fallets problem.
4. Om fallet ursprungligen utfördes av en spelare med en färg olik den nuvarande spelaren, gå till steg 8.
5. Om listan är tom, lägg till fallet i listan.
6. Om fallets problem har haft högst likhet hittills, töm listan och lägg till fallet.
7. Om likheten istället är av samma värde som likhet av de mest lika problemen, lägg till fallet i listan.
8. Om det fortfarande finns fall som inte undersökts, gå till steg 2.
9. Generera ett slumpmässigt heltal från och med ett och till och med antalet fall i listan.
10. Indexera listan med det genererade talet; fallet som finns på den platsen är fallet som ska användas.

Det finns ett antal konsekvenser med denna algoritm. Först och främst väljs ett fall slumpmässigt av ett antal fall vars problem alla har högst likhet. Det innebär att om det finns identiska fall med identiska problem bland de valda fallen, så har de större chans att väljas. Anledningen att denna metod används är att det anses naturligt att om en expert har gjort samma val fler gånger, så är det en större chans att den skulle göra det igen. Den valda likhetsfunktionen ger likhet på en diskret skala, och därför finns ingen risk för flyttalsfel när likheternas värden jämförs. De fall som inte utfördes av en spelare med samma färg som den nuvarande spelaren ignoreras. Anledningen är att fallets drag med stor sannolikhet bara skulle vara vettigt om det var spelarens tur, eftersom den annars inte kan flytta pjäsen i fallets drag. Området som draget utfördes i har även med större sannolikhet flera pjäser av motståndarens färg, vilket gör det svårt att hitta ett passande drag att anpassa till.

Likhet mellan problem dvs. lägen, definieras som den summerade likheten av rutinnehåll på respektive läges bräden. Likhet mellan rutinnehåll är som följer:

- 4 om båda pjäserna inte finns (rutorna är tomma) eller om pjäserna finns och har samma färg och typ.
- 3 om pjäserna finns och har samma färg men inte typ.
- 2 om bara en pjäs finns.
- 1 om pjäserna finns och har samma typ men inte färg.
- 0 om pjäserna har olika färg och typ.

Likheten har alltså en målmängd av diskreta heltal från och med 0 till och med 256 ( $64 \cdot 4$ ). Tanken med denna definition av likhet är att identiska lägen är 100 % lika och lägen som skiljer sig med bara några få flyttade pjäser fortfarande är lika. Lägen är även lika om det generellt men inte alltid finns pjäser av samma färg på i samma platser. Detta innebär att lösningarna, dvs. dragen, i fallen har en större chans att vara giltiga för det nya läget, för att det är en större chans att källrutan har en pjäs med samma färg och typ, att rutorna på vägen är tomma och att destinationsrutan har den förväntade färgen, alternativt att den är tom i båda problemen. I ett fall där en drottning fångar en motståndarlöpare ökar sannolikheten att hämta fallet om drottningen och löparen står på samma rutor i respektive problem och om rutorna mellan dem är tomma i respektive problem. Något som liknelsefunktionen inte tar hänsyn till, är likheter mellan förskjutna lägen, dvs. lägen där en eller flera pjäser i respektive problem inte är på samma rutor, men på närliggande rutor. Denna likhet är dock mycket mer prestandatung; istället för att bara jämföra innehållet på respektive ruta kräver denna likhet en sökning efter en liknande pjäs för varje ruta.

#### 4.2.5 Anpassning

När ett fall har hämtats kan det behöva anpassas till det nuvarande problemet. Eftersom antalet giltiga lösningar alltid är mycket mindre än antalet lösningar som kan representeras, så väljs den giltiga lösning som är mest lik fallets lösning. Det kan vara fördelaktigt att föranpassa lösningen baserat på identifierade skillnader i problemen, innan det jämförs med de giltiga lösningarna. Det är dock svårt att argumentera för vilka skillnader som är relevanta för att anpassningen ska ge en så passande lösning som möjligt. Om problemen t.ex. skiljer sig för att pjäsen som fångas i det fallets läge är förskjuten, bör den fångade pjäsen fortfarande fångas, eller hängde dragets relevans på att den specifika destinationsrutan ockuperades? Av denna anledning används ingen föranpassning.

Likhetsfunktioner och distansfunktioner i den här sektionen har målmängden  $[0, 1]$ . Ett högt tal indikerar hög likhet eller hög distans (låg likhet) beroende på om funktionen är en likhetsfunktion eller en distansfunktion. I denna sektion indikerar invertering att översätta från ett likhetsvärde till ett distansvärde eller tvärt om. T.ex. om likheten är 0,3 skulle distansvärdet vara 0,7 ( $1 - 0,3$ ). I sektionen betyder även normalisering att överföra ett tal från ett intervall  $[0, r]$  till  $[0, 1]$  genom att dividera talet med  $r$ .

Drag liknar varandra efter två olika liknelser: inverterat avstånd och innehåll. Den totala liknelsen är summan av de två viktade liknelserna. Anpassningslikhetsfunktionen har följande signatur:  $Anpassningslikhet: Drag \times Drag \times Bräde \times Bräde \times ViktLista \rightarrow [0, 1]$ . De två dragen i signaturen är ett av de giltiga dragen och fallets drag, bräderna är det nuvarande brädet och fallets läges bräde och viktlistan är AI-agentens viktlista. Om  $d_1$  och  $d_2$  är dragen,  $b_1$  och  $b_2$  är bräderna,  $v$  är viktlistan,  $L$  är avståndslikhetsfunktionen,  $A$  är avståndsfunktionen och  $I$  är innehållsfunktionen kan avståndslikhetsfunktionen beskrivas såhär:  $L(d_1, d_2, b_1, b_2, v) = v_1 * (1 - A(d_1, d_2, v_2)) + (1 - v_1) * I(d_1, d_2, b_1, b_2, v_3)$ .

Avstånd mellan drag är baserat på det normaliserade manhattanavståndet (Cormen, Leiserson, Rivest & Stein 2009) mellan rutorna som pjäserna flyttade ifrån och rutorna de flyttade till. Manhattanavståndet mellan två rutor är antal rader adderat med antalet kolumner som måste korsas för att nå den ena rutan från den andra. Manhattanavståndet har alltså målmängden  $[0, 14]$ , för att 14 är det största manhattanavståndet i schack. Att normalisera manhattanavståndet gör målmängden till  $[0, 1]$ . Vikten av avstånd mellan källrutor respektive destinationsrutor kontrolleras av en vikt. Funktionen har signaturen:  $Avstånd: Drag \times Drag \times Vikt \rightarrow [0, 1]$ . Om  $d_1$  och  $d_2$  är dragen,  $v$  är vikten,  $A$  är avståndsfunktionen,  $FrånRuta$  är  $F$ ,  $TillRuta$  är  $T$  och det normaliserade manhattanavståndet är  $M$  kan funktionen skrivas såhär:  $A(d_1, d_2, v) = v * M(F(d_1), F(d_2)) + (1 - vikt) * M(T(d_1), T(d_2))$ .

Innehåll är baserat på likheten av respektive drags flyttade och eventuellt fångade pjäs. För att se viken pjäs som flyttades och eventuellt fångades i ett drag behövs brädet som draget utfördes på. Likheten mellan potentiella pjäser är samma som användes för likhet mellan rutinnehåll för hämtningslikhet i sektion 4.2.4 och benämns med namnet *RutinnehållsLikhet*. En vikt används för att prioritera likhet på källrutor respektive destinationsrutor. Funktionen har signaturen:  $Innehåll: Drag \times Drag \times Bräde \times Bräde \times Vikt \rightarrow [0, 1]$ . De två dragen i signaturen är dragen som ska jämföras och bräderna är de bräden som respektive drag utfördes på. Om  $d_1$  och  $d_2$  är dragen,  $b_1$  och  $b_2$  är bräderna,  $v$  är vikten, *Rutinnehållslikhet* är  $R$ ,  $FrånRuta$  är  $F$ ,  $TillRuta$  är  $T$ ,  $InnehållPåRuta$  är  $P$  och innehållsfunktionen är  $I$  kan innehållsfunktionen beskrivas såhär:

$$I(d_1, d_2, b_1, b_2, v) = v * R(P(b_1, F(d_1)), P(b_2, F(d_2))) + (1 - v) * R(P(b_1, T(d_1)), P(b_2, T(d_2)))$$

Tanken med denna definition av draglikhet är att försöka identifiera vilket drag som har ett syfte så likt fallets drag som möjligt. Ett drag kan vara viktigt för att en viss pjäs på en viss ruta eller närliggande ruta flyttades, att den flyttades till en viss ruta eller i närheten av en viss ruta, att den flyttade pjäsen var av en viss typ, att den fångade pjäsen var av en viss typ eller någon kombination av dessa kriterier. Likhetsfunktionen ökar därför chansen att ge högre likhet för drag som har liknande syfte.



## 5 Utvärdering

I den här sektionen presenteras undersökningen som har utförts på produkten för att besvara arbetets fråga. I sektion 5.1 presenteras hur undersökningen utfördes, i sektion 5.2 presenteras en sammanställning av resultatet, sektion 5.3 innehåller en analys av resultat och i sektion 5.4 presenteras de slutsatser som kan dras i relation till arbetets fråga.

### 5.1 Om undersökningen

Undersökningen utfördes som en körning av ett datorprogram. Programmet är skrivet i programmeringsspråket C# 4.0 och kompilerat mot plattformen .NET 4.5. Undersökningen utfördes på en dator med en åttakärnig Intel Core i7-3770 processor på 3,40 GHz. Datorn hade tillgång till 8GB RAM och använde operativsystemet Windows 8.1 Pro. Programmet utför ett antal partier där AI-agenten spelar mot sig själv med olika fallbaser ett antal ronder per viktkonfiguration.

Under en körning spelar AI-agenten partier mot sig själv med alla kombinationer av fallbaser, undantaget är att den aldrig spelar ett parti med två identiska fallbaser. När en körning av programmet är klart produceras en mänskligt läsbar pgn-databasfil med information om de utförda partierna. Från ett pgn-formaterat parti går det att se vilka experters fallbaser som användes, deras respektive färger i partiet och datumet när partiet (undersökningen) utfördes. Utöver det går det att se om partiet utfördes under tidskravet, vilken viktkonfiguration som användes och ronden för partiet för den specifika viktkonfigurationen för de specifika fallbaserna för de specifika färgerna. Flera partier kan alltså ha samma rond om en eller båda fallbaserna skiljer sig, viktkonfigurationen skiljer sig eller om samma fallbaser används, men de spelar olika färger. Alla partier för ett visst par och en viss konfiguration grupperas och varje grupp tilldelas en tråd att utföra sina partier på som en optimering.

Fallbaser från sju experter användes i undersökningen baserade på pgn-databasfiler innehållande partier de tidigare spelat. En pgn-databasfil användes per expert och alla filer hämtades från hemsidan [www.chess-db.com](http://www.chess-db.com). Varje spelare på hemsidan har en egen sida där det går att ladda ner en pgn-databasfil av deras spelade partier. En tabell över experterna visas i Tabell 1. Informationen om experterna inklusive pgn-filerna över deras sparade partier hämtades 2015-05-04. Det är värt att notera att skillnaden i Elo-rankning mellan två experter med närliggande Elo-rankning i alla fall är lite högre än 100 poäng.

**Tabell 1** Tabell över experterna som användes i undersökningen.

Namn	Elo-Rankning
Andreas Hirsch	1994
Johan Andersson	2101
Sinan Younus Abdulrazzaq Abaeji	2211
Arturo Vidarte Morales	2320
Joerg Wegerle	2430

Michael Prusikin	2535
Evgeny Postny	2645

Antalet ordnade par som går att bilda från dessa sju experter där varje par innehåller två unika experter är 42 ( $6 \cdot 7$ ). Alla par har en motpart med samma experter i motsatt ordning. Ordningen avgör vilken färg respektive expert tilldelas, den första spelar vit och den andra spelar svart. För varje par spelas tre ronder. Dessa 126 partier ( $42 \cdot 3$ ) spelas för varje viktkonfiguration. För två unika fallbaser spelas alltså totalt sex ronder, där varje fallbas spelar vit i tre ronder och svart de tre andra ronderna. En av fallbaserna skulle då anses vara bättre än den andra om den t.ex. vann fyra partier och förlorade två (4-2 i poäng), men det är även giltigt om t.ex. fem av partierna var remi och ett var en vinst ( $3\frac{1}{2}$ -2 $\frac{1}{2}$  i poäng). För varje parti tilldelas varje fallbas ett unikt slumpfrö. Tanken var att olika ronder mellan samma par fallbaser inte skulle se likadana ut och producera identiska partier. Totalt fem viktkonfigurationer användes, vilket innebär att antalet spelade partier i undersökningen är 630. Viktkonfigurationerna visas i Tabell 2.

**Tabell 2** Tabell över viktkonfigurationerna som användes i undersökningen.

Vikt 1	Vikt 2	Vikt 3
0,5	0,5	0,5
0,8	0,8	0,5
0,8	0,2	0,5
0,2	0,5	0,8
0,2	0,5	0,2

Tanken med den första viktkonfigurationen är att undersöka om alla delar inom draglighet bör ha lika stor vikt. Tanken med de resterande fyra är att undersöka om någon specifik del av draglighet ger bättre resultat. Kom ihåg att den första vikten påverkar om likhet från inverterat avstånd eller likhet från innehåll påverkar den totala likheten, där ett högre tal innebär att inverterat avstånd prioriteras mer. Vikt två påverkar om avstånd mellan källrutor prioriteras mer än avstånd mellan destinationsrutor och vikt tre påverkar om likhet mellan innehåll på källrutor prioriteras över likhet mellan innehåll på destinationsrutor. Fördelningen mellan högt prioriterat (0,8) och lågt prioriterat (0,2) gjordes för att klart prioritera den ena över den andra i de flesta lägen och samtidigt undvika att bara en aspekt av likhet tas hänsyn till. Det anses troligt att en aspekt kan ge ett bättre resultat, men att bara ta hänsyn till den aspekten kan ge ett sämre resultat än att ta lika mycket hänsyn till varje aspekt. Om konfigurationer som bara tar hänsyn till en aspekt ger ett dåligt resultat jämfört med konfigurationen som tar lika mycket hänsyn till alla aspekter kan det vara svårare att avgöra varför: är det för att aspekten inte bör tas hänsyn till eller att det finns andra mindre viktiga aspekter som också behöver tas hänsyn till? Det är möjligt att vissa aspekter av likhet bör ha större vikt än andra, kanske till och med att en aspekt ger överväldigande bättre resultat, men detta kan upptäckas utan att undersöka vad som skulle hända om varje aspekt prioriterades till 100 %.

## 5.2 Resultat

Resultatet från undersökningen innehåller mycket data på grund av det stora antalet partier det är uppbyggt. Av denna anledning visas bara en sammanfattning av resultatet i denna sektion och hela resultatet presenteras i bilaga A. En tabell över resultat för varje konfiguration visas i Tabell 3. Sammanfattningen visar följande för varje konfiguration:

- Om de första 40 dragen för båda fallbaserna utfördes under 90 minuter i alla partier, vilket var tidskravet på produkten.
- Rankningsindex av fallbaserna i den längsta ordningen, där fallbasen baserad på den högst rankade experten har index 1 och den lägst rankade experten har index 7. En ordning är sådan att för all fallbaser i ordningen så är fallbasen antingen sist eller så har den högre poäng jämt emot fallbasen ett steg senare i ordningen.
- Om ordningen stämmer överens med ordningen av fallbasernas respektive experts Elo-rankning.
- Procentuellt antal partier som en av spelarna vann, dvs. som inte slutade i remi.

**Tabell 3** Tabell över resultatet för varje individuell konfiguration.

Konfiguration	Klarade tidskrav	Längsta ordning	Matchar Elo-rankning ordning	% vunna partier
0,5 0,5 0,5	Ja	6, 2	Nej	2
0,8 0,8 0,5	Ja	2, 5, 1, 6, 7	Nej	12
0,8 0,2 0,5	Ja	3, 7, 1, 6	Nej	6
0,2 0,5 0,8	Ja	5, 3, 2, 1 7	Nej	8
0,2 0,5 0,2	Ja	5, 6, 7	Ja	3

## 5.3 Analys

Efter att ha spelat upp ett antal partier med ett program som kan visualisera pgn-formaterade partier upptäcktes det att AI-agenten spelar schack väldigt dåligt. I början av partierna spelar AI-agenten som förväntat, men väldigt snabbt börjar den göra underliga drag. I mittspelet verkade de som om pjäserna flyttades utan rim och reson och nästan ingen av partierna nåddes slutspelet. AI-agenten fångar nästan aldrig pjäser och verkar inte kunna lista ut hur den ska vinna. I de få fall den vinner verkar det mer som en tursam slump än ett resultat av strategiskt tänkande.

AI-agenten presterar så dåligt för att den inte verkar ha något mål att sträva efter; även nybörjare vet t.ex. att det är nästan alltid är fördelaktigt att fånga en högt värderad pjäs om spelaren bara förlorar en lägre värderad pjäs. AI-agenten lämnar dock ofta högt värderade pjäser utan skydd och fångar sällan dessa pjäser som motståndare. AI-agenten verkar inte heller avancera bönder i hopp om att omvandla dem, den försöker inte ta kontroll över mitten av brädet och inte heller flytta ut de högt värderade pjäserna. AI-agenten agerar då och då som den försöker uppfylla någon av dessa mål, men dessa få tillfällen av bra beslut blandas ut av

alla dåliga. AI-agenten vinner ibland, men så sällan att det upplevs som den lika gärna kunnat spela slumpmässiga drag och vunnit lika ofta. De gånger den vinner är det ofta inte svårt för motståndaren att undvika, men eftersom även motståndaren kontrolleras av AI-agenten lyckas den inte göra det.

AI-agenten hamnar ofta i en situation där spelarna fastnar i en loop och flyttar varsin pjäs fram och tillbaka tills samma brädformation upprepat sig tre gånger vilket innebär remi. Detta verkar logiskt givet hur likhet är implementerat; om AI-agenten hamnar i ett läge med samma brädformation som ett tidigare läge så borde AI-agenten välja samma fall från fallbasen och göra samma drag, givet att det inte finns flera fall att välja på. Detta leder dock till att de flesta partier slutar i remi och gör de svårt om omöjligt att rangordna hur bra AI-agenten spelar med respektive fallbas.

Ingen av konfigurationerna verkar ge ett bättre resultat än någon annan, åtminstone inte på ett sätt som går att identifiera från resultatet eftersom så många partier slutar i remi. Oavsett konfiguration klarades tidskraven med galans. Faktum är att hela undersökningen bara tog 20 minuter att utföra. Ett parti kan inte utföras snabbare med flera trådar, men flera partier kan utföras parallellt. Konfiguration två och fyra gav de längsta ordningarna och innehåller det högsta antalet partier som inte slutade i remi. Ordningarna är dock väldigt olika och verkar inte alls vara baserade på fallbasernas respektive experters Elo-rankningar. Den troligaste anledningen att det överhuvudtaget gick att producera så långa ordningar är att tillräckligt många partier inte slutade i remi. Den sista konfigurationen innehåller en ordning som matchar ordningen av de respektive experternas Elo-rankningar, men eftersom den är så kort och väldigt få partier vanns med den konfigurationen så det är svårt att argumentera för att den ger bättre resultat.

## 5.4 Slutsatser

Som nämnt i sektion 3.1 är problemställningen om CBR med grundlig likhet är passande för en CBR-baserad schackspelande AI-agent. Passande definierades som:

- Det går att rangordna ett antal fallbaser där en högre rangordnad fallbas vinner oftare mot en lägre rangordnad fallbas givet att den största skillnaden i fallbasernas experters Elo-rankningar är minst 100 poäng.
- Rangordning av varje fallbas stämmer överens med rangordningen av dess experts Elo-rankning relativt de andra experternas Elo-rankningar.

Givet resultatet i undersökningen anses det att svaret är nekande. Trots att skillnaden i Elo-rankningar mellan närliggande experter aldrig var lägre 100 poäng så gick det inte att få en konsekvent ordning av alla experter med någon av konfigurationerna. De ordningar som producerades med olika konfigurationer matchade heller inte ordningen av experternas Elo-rankningar.

AI-agenten presterar dåligt oavsett fallbas och verkar inte försöka uppfylla några djupare schackrelaterade mål, som att fånga värdefulla pjäser eller vinna partier. På grund av det slutar väldigt många partier i remi och de få gånger AI-agenten vinner med en fallbas liknar det en slump. Det går därför inte att rangordna dem på ett trovärdigt sätt om bara resultatet av partierna tas i beaktning. Även när partierna studeras ger det intrycket att AI-agenten inte spelar noterbart bättre eller sämre med olika fallbaser.

## 6 Avslutande diskussion

### 6.1 Sammanfattning

Arbetet utfördes för att undersöka om en schackspelande AI-agent vars beslutsfattande är baserat CBR med grundlig likhet kan spela bättre med fallbaser baserade på spelare med högre Elo-rankning. Ett funktionsbibliotek av en AI-agent implementerades för att besvara frågan. Undersökningen gick ut på att låta AI-agenten spela mot sig själv ett antal gånger. Samma partier utfördes flera gånger för att undersöka om olika inställningar i AI-agenten påverkade resultatet på ett märkbart sätt. AI-agenten spelade väldigt dåligt och vann nästan aldrig några partier mot sig själv. De få gånger AI-agenten vann verkade det slumpartat med vilken fallbas den vann. Det gick därför inte att rangordna fallbaserna så att fallbaser baserade på högre rankade experter spelade bättre.

### 6.2 Diskussion

Det verkar som om inget arbete tidigare har berört applicering av CBR på schack-AI tidigare och ur den synvinkeln är arbetet unikt. Arbetet utfördes dels för att besvara frågan som ställdes i problemformuleringen men även för att agera som språngbräda för annan forskning runt hur CBR kan appliceras för schack-AI. Det anses värt att undersöka tekniken för att det gett positivt resultat i flera domäner (Wender och Watson 2014; Rubin 2013) även om det inte varit så för alla (Bellamy-McIntyre 2008). Att applicera CBR på just schack ansågs passande eftersom schack flera gånger har använts som testramverk för hypoteser (Simon & Chase 1973) vilket dock avtagit med åren (Schaeffer 1991; Ensmenger 2012). Att inte lämna möjligen användbara tekniker utforskade anses viktigt för att förbättra schack-AI, oavsett om det gäller att skapa så mänsklig schack-AI som möjligt eller så bra schack-AI som möjligt.

Resultatet anses pålitligt givet de praktiska problem som finns i dagsläget. Det är möjligt att resultatet kan ändras i framtiden om ett av problemen har lösts, men det är bara spekulation. Även om inte alla möjliga implementationsval har undersökts för domänen grundlig likhet anses resultatet vara trovärdigt. Anledningen är att de alternativa implementationsvalen inte tillför något som kan bidra med att lösa fundamentala problem med implementationen och bör därför inte ge annat än ett nekande resultat oavsett.

Det finns ett antal alternativa sätt som problemlikhet kunde implementeras inom ramen av grundlig likhet. Två exempel är att förskjuta pjäserna på ett utav fallens bräden och sedan jämföra likhet eller att mäta det genomsnittliga lägsta avståndet från varje ruta på ett falls bräde och den närmaste rutan på det andra fallets bräde som har likadant innehåll. Dessa två alternativ måste dock visa att de kan lösa följande problem för att vara av intresse att undersöka. Anta att det finns en drottning som den nuvarande spelaren kan fånga utan att förlora någon värdefull pjäs och vars fångning inte verkar påverka den nuvarande spelaren negativt i det långa loppet. Även en nybörjare skulle fånga drottningen eftersom det är viktigt i schack att fånga så många värdefulla pjäser som möjligt och samtidigt undvika att ens egna värdefulla pjäser fångas. Varken den nuvarande problemlikheten eller alternativen ger dock en större sannolikhet att välja ett fall med ett läge där en värdefull pjäs fångas utan negativa konsekvenser, än mindre att en lämplig anpassningsmetod kan anpassa fallets drag till exakt rätt drag. Det mest uppenbara vore att ge större likhet mellan lägen där spelaren kan göra fallets drag utan några negativa konsekvenser i den nära framtiden, men denna likhet är utanför grundlig likhet och därför inte intressant att undersöka i detta arbete. Säkerligen är

det möjligt att experten kan ha stött på exakt samma problem eller ett problem som är tillräckligt likt och AI-agenten skulle därför kunna utföra det korrekta draget. Problemet är ovissheten att ett tillräckligt likt problem har skett, eftersom likheten att det går att fånga en pjäs utan negativa konsekvenser är svår att uttrycka med grundlig likhet. AI-agenten kan och bör utföra fel drag ibland eftersom det är mänskligt att fela, men i undersökningen skedde det så ofta att det gav intrycket att AI-agenten spelade sämre än en nybörjare oavsett fallbas. Det kan argumenteras att denna slutsats kunde härledas utan att göra undersökningen i första taget. Slutsatsen hade dock möjligen inte gällt om det praktiska förutsättningarna var annorlunda, t.ex. om det fanns många fler partier tillgängliga för alla experter vilket skulle ge större chans att liknande situationer hade skett för experten. Nu när undersökningen har utförts styrker det uppfattningen att det finns ett fundamentalt problem med grundlig likhet för schackspelande AI-agenter. Det fundamentala problemet med grundlig problemlikhet gör det osannolikt att resultatet kan ändras genom att ändra andra aspekter av implementationen som att undersöka alternativa anpassningsmetoder eller använda utforskade viktkonfigurationer.

De expertdata som användes i undersökningen är inte perfekt. Pgn-databasfilerna som finns tillgängliga för nedladdningar på [www.chess-db.com](http://www.chess-db.com) innehåller antagligen inte alla partier som experterna har spelat, och olika partier har spelats vid olika tillfällen när experterna var olika bra. Om en expert har Elo-rankning  $x$  och spelade på ett visst sätt när den hade en lägre Elo-rankning  $y$ , bör dessa beslut beaktas? Trots allt, experten kanske fick Elo-rankning  $x$  för att den slutade göra de misstag den gjorde när den hade Elo-rankning  $y$ . Det finns dock inte möjlighet att vara kräsen eftersom det finns så få partier tillgängliga för varje spelare och antalet minskar kraftigt med spelarnas rankning. För spelare runt Elo-rankning 2000 poäng finns det ofta bara 50-100 tillgängliga partier, medan det finns över 1000 partier tillgängliga för Evgeny Postny som har en Elo-rankning av 2645 poäng. Detta kan dock vara en konsekvens av att högre rankade spelare har spelat flera partier. Eftersom det inte finns något självklart sätt att få ut en spelares sanna potential ur en delmängd av dess spelade partier förblir bristande expertdata ett problem för att kunna rangordna experters fallbaser efter experternas Elo-rankningar, speciellt för experter med Elo-rankningar lägre än 2000 poäng.

Arbetet är av samhällsnytta för att det visar att CBR med grundlig likhet inte alltid är passande för schackspelande AI-agenter beslutsfattande. Detta kan hjälpa folk att ta informerade beslut av vilka tekniker de ska använda och hur de kan implementeras när de skapar schackspelande AI-agenter. Det kan även användas som grund i andra arbeten för att undersöka liknande problem som kan leda till bättre schack AI. Det kanske visar sig att någon form av CBR är passande för schack-AI och det här arbetet kan visa vad som inte fungerar, vilket leder folk till att undersöka andra sätt att applicera CBR. Arbetet kan även användas för att härleda praktikaliteten av grundlig likhet i andra domäner. Om det t.ex. visar sig att typen av beslut som görs i spelet go liknar de som görs i schack kan det kanske härledas att grundlig likhet inte passar för go. Arbetet användbarhet för samhället sträcker sig inte därför bara till gränserna av schack, utan till många andra brädspel.

För att göra det så lätt som möjligt att upprepa arbetet presenterades den relevanta hårdvaran och mjukvaran som användes i artefaktens och undersökningens implementations i sektion 5.1. Det finns en del skillnader mellan .NET implementationer och versioner i deras implementation av pseudoslumpgenerering och det sättet som fallbaser tilldelas slumpfrön för varje grupp av partier är inte ordentligt dokumenterat i rapporten. Det är därför möjligt att resultatet kan skilja sig lite vid återupprepning, men troligen ge ett liknande resultat.

Undersökningen har utförts på experter som endast valdes för deras Elo-rankningar. Experterna har inte kontaktats för explicit samtycke att deras partier kan användas i undersökningen. För att ha kunnat delta i diverse tävlingar som partierna utfördes i måste experterna dock ha tillåtit att arrangörerna dokumenterat, sparat och eventuellt publicerat partierna. De partier som användes i undersökningen har varit publikt tillgängliga och därför borde experterna inte ha något emot att deras partier användes i undersökningen.

### 6.3 Framtida arbete

I arbetet finns det ett antal olika implementationsval i domänen av frågan som inte undersökts. T.ex. så undersöks inte problemlikheten att förskjuta en av lägenas brädes pjäser ett steg åt ett visst håll och sedan jämföra likheten mellan dem med nuvarande problemlikhet. Ett annat exempel på problemlikhet är att för varje ruta på ett av problemens bräden beräkna inversavståndet till närmaste ruta på det andra brädet som har samma eller tillräckligt likt innehåll och sedan summera alla inversavstånd. En annan sak som kan varieras är antalet fall som hämtas från fallbasen under hämtningen. De mest lika fallen kan hämtas och det fall vars drag har högst likhet med ett av dragen väljs, medräknat straffpoäng om fallets problem har låg likhet med det nuvarande problemet gentemot de andra valda fallens problem. Det kan undersökas om det går att extrahera mer information från en pgn-databas än bara en samling lägen och dragen som utfördes i varje parti. T.ex. om experten i ett fall fångar en drottning utan negativa konsekvenser kan många fler fall med olika bräden och samma drag genereras för att öka chansen att rätt drag väljs när AI-agenten hamnar i en liknande situation. Med alla dessa val måste det finnas många fler vikter och viktkonfigurationer som kan undersökas.

Eftersom det anses troligt att resultatet av arbetet inte kan påverkas genom att ändra implementationen inom problemets domän, bör framtida arbete främst baseras på alternativa närliggande problem. Undersökningen visade att implementationen inte var dyr att implementera ur prestandasynpunkt, vilket gör att det finns gott om rum i ett framtida arbete att använda dyrare algoritmer.

Istället för grundlig likhet kan mer komplicerade och omfattande likheter undersökas. Detta skulle vara av intresse för folk som vill implementera schackprogram med en AI-agent vars skicklighetsnivå går att justera. Istället för att försöka rangordna fallbasernas skicklighet efter deras respektive experters skicklighet kan syftet med AI-agent vara att spela så bra som möjligt, eller att spela så mänskligt som möjligt. Detta är av intresse för folk som studerar schack, AI eller schacks relation till AI, folk som implementerar schackmotorer och folk som underhålls av att se på partier spelade av experter. För att uppnå detta går det även att undersöka hur CBR med grundlig likhet eller annan likhet för schack fungerar tillsammans med andra beprövade tekniker; kanske kan CBR komma en rimlig bit på vägen till att hitta rätt drag men behöver kompletteras med en mer exakt men kanske prestandadyr teknik? Det går möjligtvis att identifiera mönster i hur experterna spelar och skapa en AI-agent som kan uppfatta och utföra dessa mönster med detta arbete som grund. Tekningen Case-based Planning (CBP) (Spalazzi 2001) kan vara av användning för att undersöka detta.

# Referenser

- Anthony, S. (2014). *A new (Computer) Chess Champion is Crowned, and the Continued Demise of Human Grandmasters*. <http://www.extremetech.com/extreme/196554-a-new-computer-chess-champion-is-crowned-and-the-continued-demise-of-human-grandmasters> [2015-03-19]
- Bellamy-McIntyre, J. (2008). *Applying Case-Based Reasoning to the Game of Bridge*. Diss. University of Auckland, New Zealand.
- Campbell, M., Hoane, A. J. & Hsu, F. H. (2002). Deep Blue. *Artificial intelligence*, 134(1), ss. 57-83.
- Chess News (2009). *Breakthrough Performance by Pocket Fritz 4 in Buenos Aires*. <http://en.chessbase.com/post/breakthrough-performance-by-pocket-fritz-4-in-buenos-aires> [2015-03-19]
- Chess Programming Wiki (2013a). Board Representation. <https://chessprogramming.wikispaces.com/Board+Representation> [2015-04-30]
- Chess Programming Wiki (2013b). Piece-Lists. [https://chessprogramming.wikispaces.com/Board+ Piece-Lists](https://chessprogramming.wikispaces.com/Board+Piece-Lists) [2015-04-30]
- Chess Programming Wiki (2014a). 8x8 Board. <https://chessprogramming.wikispaces.com/8x8+Board> [2015-05-04]
- Chess Programming Wiki (2014b). 10x12 Board. <https://chessprogramming.wikispaces.com/10x12+Board> [2015-05-04]
- Chess Programming Wiki (2014c). Encoding Moves. <https://chessprogramming.wikispaces.com/Encoding+Moves> [2015-05-04]
- Chess Programming Wiki (2015). Generating Moves. <https://chessprogramming.wikispaces.com/Move+Generation> [2015-05-04]
- Cormen, T. H., Leirson, C. H., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms (3rd ed.)*. London: MIT Press.
- Elo, A. E. (1978). *The Rating of Chess Players, Past and Present*. New York: Arco Publishing.
- Ensmenger, N. (2012). Is Chess the Drosophilia of Artificial Intelligence? A Social History of an Algorithm. *Social Studies of Science*, 42(1) ss. 5-30.
- Frey, P. W. (1983). The Alpha-Beta Algorithm: Incremental Updating, Well-Behaved Evaluation Functions, and Non-Speculative Forward Pruning. *Computer Game-Playing*. ss. 285-289. Chichester: Ellis Horwood Limited Publishers.
- Gould, J. & Levinson, R. A. (1991). *Method integration for experience-based learning*. California: University of California.
- Greenblatt, R. D., Eastlake, E. D. & Crocker, D. S. (1969). *The Greenblatt Chess Program*. <http://hdl.handle.net/1721.1/6176> [2015-02-08]



- Heinz, E. A. (1999). Endgame databases and efficient index schemes. *ICCA Journal*, 22(1), ss. 22-32.
- Holmos, P. R. (1960). *Naive Set Theory*. New York: Litton Educational Publishing, Inc.
- Hsu, F. (1991). "Expert Inputs" are sometimes harmful. I *IJCAI-91, Proceedings of the Twelfth International Conference on Artificial Intelligence*. Sydney, Australia.
- Huber, R. (2006). *Description of the universal chess interface (UCI)*. <http://download.shredderchess.com/div/uci.zip> [2015-02-08]
- Laramée, F. D. (2000a). *Chess Programming Part I: Getting Started*. [http://www.gamedev.net/page/resources/\\_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014](http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014) [2015-03-30]
- Laramée, F. D. (2000b). *Chess Programming Part II: Data Structures*. [http://www.gamedev.net/page/resources/\\_/technical/artificial-intelligence/chess-programming-part-ii-data-structures-r1046](http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-ii-data-structures-r1046) [2015-03-19]
- Laramée, F. D. (2000c). *Chess Programming Part III: Move Generation*. [http://www.gamedev.net/page/resources/\\_/technical/artificial-intelligence/chess-programming-part-iii-move-generation-r1126](http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-iii-move-generation-r1126) [2015-03-30]
- Laramée, F. D. (2000d). *Chess Programming Part IV: Basic Search*. [http://www.gamedev.net/page/resources/\\_/technical/artificial-intelligence/chess-programming-part-iv-basic-search-r1171](http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-iv-basic-search-r1171) [2015-03-30]
- Laramée, F. D. (2000e). *Chess Programming Part V: Advanced Search*. [http://www.gamedev.net/page/resources/\\_/technical/artificial-intelligence/chess-programming-part-v-advanced-search-r1197](http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-v-advanced-search-r1197) [2015-03-30]
- Laramée, F. D. (2000f). *Chess Programming Part VI: Evaluation Functions*. [http://www.gamedev.net/page/resources/\\_/technical/artificial-intelligence/chess-programming-part-vi-evaluation-functions-r1208](http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-vi-evaluation-functions-r1208) [2015-03-30]
- Linke, T. R. (2001). Strategies for the Automatic Construction of Opening Books. I *CG 2000, Second International Conference*. Hamamatsu, Japan 26-28 oktober 2000, ss. 74-86. DOI: 10.1007/3-540-45579-5\_5
- Mann, T. & Muller, H. G. (2009). *Chess Engine Communication Protocol*. <http://www.gnu.org/software/xboard/engine-intf.html> [2015-02-09]
- McCarty, J. (1990). Chess as the Drosophilia of AI. *Computers, Chess, and Cognition*, ss. 227-237. New York: Springer.
- McKinsey, J. C. C. (1952). *Introduction to the Theory of Games*. Santa Monica: The Rand Corporation.
- Rubin, J. (2013). *On the Construction, Maintenance and Analysis of Case-Based Strategies in Computer Poker*. Diss. University of Auckland, New Zealand.
- Richter, M. M., & Weber, R. O. (2013). *Case-Based Reasoning: A Textbook*. Berlin: Springer.

- Schaeffer, J. (1991). Computer Chess: Science of Engineering? I *IJCAI-91, Proceedings of the Twelfth International Conference on Artificial Intelligence*. Sydney, Australia.
- Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(314).
- Simon, H. A. & Chase, W. G. (1973) Skill in Chess. *American Scientist*, 61(4), ss. 393-403.
- Spalazzi, L. (2001). A Survey on Case-Based Planning. *Artificial Intelligence Review*, 16(1), ss. 3–36.
- Standard: Portable Game Notation Specification and Implementation Guide* (1994). <http://www6.chessclub.com/help/PGN-spec> [2015-02-16]
- Wender, S. & Watson, I. (2014). *Integrating Case-Based Reasoning with Reinforcement Learning for Real-Time Strategy Game Micromanagement*. Diss. University of Auckland, New Zealand.
- World Chess Federation (2014a). *FIDE Rating Regulations Effective From 1 July 2014*. <http://www.fide.com/fide/handbook.html?id=172&view=article> [2015-02-09]
- World Chess Federation (2014b). *General Rules and Recommendations for Tournaments: Time Control*. [www.fide.com/component/handbook/?id=39&view=category](http://www.fide.com/component/handbook/?id=39&view=category) [2015-02-09]
- World Chess Federation (2014c). *Laws of Chess: For competitions Starting On or After 1 July 2014*. <http://www.fide.com/fide/handbook.html?id=171&view=article> [2015-02-09]
- World Chess Federation (2015). *Top 100 Players February 2015 – Archive*. <https://ratings.fide.com/toparc.phtml?cod=341> [2015-03-19]