FFI RAPPORT

DECISION MAKING IN SIMPLIFIED LAND COMBAT MODELS - On design and implementation of software modules playing the games of Operation Lucid and Operation Opaque

HALCK Ole Martin, SENDSTAD Ole Jakob, BRAATHEN Sverre, DAHL Fredrik A

FFI/RAPPORT-2000/04403

FFISYS/722/161.3

Approved
Kjeller 15 November 2000

Bent Erik Bakken
Director of Research

DECISION MAKING IN SIMPLIFIED LAND COMBAT MODELS - On design and implementation of software modules playing the games of Operation Lucid and Operation Opaque

HALCK Ole Martin, SENDSTAD Ole Jakob,
BRAATHEN Sverre, DAHL Fredrik A

FFI/RAPPORT-2000/04403

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
P O Box 25, NO-2027 Kjeller, Norway

FORSVARETS FORSKNINGSINSTITUTT (FFI)
Norwegian Defence Research Establishment

UNCLASSIFIED

P O BOX 25
N0-2027 KJELLER, NORWAY

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

REPORT DOCUMENTATION PAGE

1) PUBL/REPORT NUMBER FFI/RAPPORT-2000/04403 1a) PROJECT REFERENCE FFISYS/722/161.3	2) SECURITY CLASSIFICATION UNCLASSIFIED 2a) DECLASSIFICATION/DOWNGRADING SCHEDULE -	3) NUMBER OF PAGES 49

4) TITLE
DECISION MAKING IN SIMPLIFIED LAND COMBAT MODELS - On design and implementation of software modules playing the games of Operation Lucid and Operation Opaque

5) NAMES OF AUTHOR(S) IN FULL (surname first)
HALCK Ole Martin, SENDSTAD Ole Jakob, BRAATHEN Sverre, DAHL Fredrik A

6) DISTRIBUTION STATEMENT
Approved for public release. Distribution unlimited. (Offentlig tilgjengelig)

7) INDEXING TERMS

	IN ENGLISH:	IN NORWEGIAN:
a)	Combat modelling	Stridsmodellering
b)	Game theory	Spillteori
c)	Constraint programming	Føringsbasert programmering
d)	Neural nets	Nevrale nett
e)	Fuzzy logic	Fuzzy logikk

THESAURUS REFERENCE:

8) ABSTRACT

We present the work done on the stochastic games Operation Lucid and Operation Opaque during FFI project 722, “Synthetic decision making”. These games, designed as simplified land combat simulation models, are defined and some of their properties described. We give a theoretical and practical treatment of the problem of evaluating performance in these games, including mathematically sound performance measures and a successful method for reducing the effect of stochastic noise in the games. The core of the report consists of a general design based on constraint programming for software agents playing the games of Operation, and two applications of this design, using neural nets and fuzzy logic, respectively. The agent design presented is successful in combining the best points of a brute-force and a more human-like approach to game playing, and makes it possible for software agents to play well in spite of the very high complexity of the games. The applications demonstrate the practical utility of this design. Special issues pertaining to the information imperfection of Operation Opaque are also addressed. Some main conclusions of the work are: 1) Our agent design is useful for applying and combining artificial intelligence techniques. 2) Reinforcement learning algorithms are suitable for learning in this noisy domain, while direct gradient-based parameter optimisation is not. 3) Representation of domain knowledge can significantly improve performance.

9) DATE 15 November 2000	AUTHORIZED BY This page only Bent Erik Bakken	POSITION Director of Research

ISBN-82-464-0463-6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

CONTENTS

		Page
1	INTRODUCTION	7
2	THE GAME ENVIRONMENT	8
2.1	Operation Lucid	9
2.2	Operation Opaque	10
2.3	Interpretation as Combat Model	10
2.4	The Complexity of the Problem	11
3	EVALUATING PERFORMANCE	11
3.1	Theoretical Performance Measures	12
3.1.1	*Geq*: Equity Against Globally Optimising Agent	13
3.1.2	*Peq*: Equity Against Minimax Playing Agent	13
3.2	Estimating Performance by Reference Players	14
3.2.1	Reference Players	15
3.3	Compensating for Randomness	16
3.3.1	Variance Reduction by the Control Variate Method	16
4	A GENERIC PLAYER DESIGN	19
4.1	Two Common Approaches	19
4.1.1	The Constructive Approach	19
4.1.2	The Evaluation Approach	19
4.2	A Modular Mixed Approach	20
5	A MOVE GENERATOR BASED ON CONSTRAINT SATISFACTION	22
5.1	Constraint Satisfaction Problems	23
5.2	Implementation of the Constrained Move Generator	23
5.2.1	Input and Output	23
5.2.2	Formulation of the Constraint Satisfaction Problem	24
5.3	The Constraints	24
5.3.1	Global Constraints: Rules and Heuristics	25
5.3.2	Local Contraints: The Area Language Protocol	26
6	USING THE DESIGN (1): A NEURAL NET AGENT	27
6.1	Constraint Generation	27
6.2	Evaluating Moves by a Neural Network	27
6.3	TD(λ)-learning	28
6.4	Experimental Results	28
6.5	An Improved Constraint Generator	29
7	USING THE DESIGN (2): A FUZZY LOGIC AGENT	29
7.1	The Agent Model	30
7.1.1	Design Principles	30

7.1.2 Implementation 32

7.2 Training the Agent 36
7.2.1 Training Method 36
7.2.2 Experimental Results 37

7.3 Discussion 38

8 DEALING WITH IMPERFECT INFORMATION 39

8.1 An Intelligence Module for Operation Opaque 40

9 FUTURE RESEARCH 41

10 CONCLUSION 42

REFERENCES 45

Distribution list 49

DECISION MAKING IN SIMPLIFIED LAND COMBAT MODELS - On design and implementation of software modules playing the games of Operation Lucid and Operation Opaque

1 INTRODUCTION

This report describes work done by FFI project 722, Synthetic Decision Making, on the games Operation Lucid and Operation Opaque (11)[1]. The goal of this activity was to study decision making in land combat simulation model; these games were defined to represent the essential parts of land combat modelling, without having to deal with the cumbersome and context-specific details present in an actual simulation model. The choice of representing the combat situation by games – more specifically, by two-player zero-sum games – was motivated by the characteristics of such a situation. In most land combat models there are two opposing sides (hence we use two-player games) having no common interest and no incentive to co-operate (hence zero-sum games).

Given this simplified representation of the problem, the problem of modelling decision making turns into the problem of constructing software agents that play the game by observing and acting on the environment. For our modelling purposes, the goal is to construct players that behave like humans do. In general, this is not necessarily the same as behaving “rationally” (in some sense of the word). However, certain properties of games like Operation are such that humans generally play them quite well, at least better than computers. Thus, for the task of constructing Operation-playing software agents, optimising performance can be seen as roughly equivalent to mimicking human game-play.

From the start of artificial intelligence research, game playing has been much studied. Russell and Norvig suggest that “[w]e might say that game playing is to AI as Grand Prix motor racing is to the car industry [...]” (26, p. 141) – implying that game playing is a leading-edge activity within AI, but that the results and solutions from these abstract settings may not always generalise well to real-world problems.

Generally speaking, we can discern a difference between how humans and machines usually play games. Humans tend to plan towards a goal and *construct* their moves according to this (26, p. 140), while computers *evaluate* all available options. The former approach is often associated with “intelligence” as we normally use the term; it requires semantic understanding of the domain, and is difficult to implement well in a computer program. The latter approach requires less previous knowledge, but is usually prone to problems related to computational complexity.

In computer game-play, evaluation is often done by search in game trees – most game-playing research concentrates on this (see e.g. (26, Chapter 5) and the references therein). This is due to certain properties common to most of the games usually studied. In particular, these games

[1] In the following, these games will sometimes jointly be called simply “Operation”.

have trees with a branching factor that is not too large (in chess rarely over 100, for instance); consequently it is feasible to enumerate all legal actions in a given game state and, given efficient pruning methods, to look several moves into the future. This property may be part of the reason why results from AI research in games does not necessarily translate directly to more practically useful knowledge – in the real world, the number of choices may be much greater than is usual in a game. Thus, at the risk of offending avid players of these games, they may often be said to be "toy problems" in the context of real-world applications.

This paper, on the other hand, deals with a game designed to model essential parts of a complex real-world situation; the branching of the game tree is consequently very large – up to the order of 10^{10}. In most game states, the number of legal moves is thus far too large to enumerate exhaustively. In addition to this, the sequence of moves is stochastic, in the sense that it is not known which player gets to move in the next turn. Due to these properties, game-tree search is not applicable to our game, as it is infeasible to build even the first two levels of the game tree. Nevertheless, a division into two general approaches, similar to that mentioned above, is applicable to the problem of selecting which move to make. The central chapters of this report present a design for a game-playing agent which combines the strong points of these two approaches while attempting to diminish the drawbacks of each of them. This design is centred about the formulation of the problem as a *constraint satisfaction problem* (CSP), and the application of constraint programming for solving this problem – see (4) for a review of this field. Constraint programming has proved to be a very useful building block when making game-playing agents for Operation; as we shall see, it facilitates the use of other AI methods – and combinations of AI methods – in the building of agents playing our games.

The report is organised as follows. Chapter 2 is dedicated to the definition, interpretation and complexity of the problem environment – the games of Operation Lucid and Operation Opaque. In Chapter 3, we give a theoretical and practical exposition of the evaluation of agents playing (zero-sum) games in general and Operation in particular. Chapters 4 and 5 are the core chapters in this report[2]: Chapter 4 treats the overall design of our game-playing agents, while Chapter 5 gives a more detailed view of our use of constraint programming in this design. In Chapters 6 and 7 we describe actual agents for Operation Lucid that have been built using the design presented in Chapters 4 and 5. Chapter 8 deals with the special challenges presented by the imperfect information of Operation Opaque, while further research and conclusions are treated in Chapters 9 and 10.

2 THE GAME ENVIRONMENT

Our goal is to construct game-playing software agents. Such an agent should be able to get a game state as input, and, from this state and the rules of the game, generate a single move as output. The move describes where each of the agent's own pieces should be placed when the turn is finished. In this chapter we present the problem environment – the games of Operation Lucid and Operation Opaque – and describe some of the properties that make them both interesting and very challenging. More information on Operation can be found in (11).

[2] A version of these chapters, along with Chapter 6, has been previously published (30).

2.1 Operation Lucid

In short Operation Lucid is a two-person stochastic board game where the two players start off with their pieces in opposing ends of the board (Figure 2.1). All pieces of one side are equivalent, and therefore indistinguishable from each other except by their locations on the board. One player, named Blue, is the attacker. Blue starts the game with fifteen pieces; his aim is to cross the board, break through – or evade – his opponent's defence, and move his pieces off the board into the goal node. The defending player, Red, starts with ten pieces; his task is to hinder Blue from succeeding. The result of the game is the number of pieces that Blue has managed to get across the board and into the goal node; thus, there is no "winner" or "loser" of a single game – the result is a number ranging from 0 to 15. The rest of this section contains the rules of the game.

The game of Operation Lucid is played in 36 turns. At the start of each turn, the right to move pieces is randomly given to either Blue or Red, with equal probabilities. The side winning this draw is allowed to move each piece to one of the neighbouring nodes (that is, a node that is connected to the piece's current node by an edge) or leave it where it is. The side losing the draw, of course, does not get to move any pieces in that turn.

The movement of the pieces is subject to these restrictions:

- When the move is finished, no node can have more than three pieces of the same colour.
- Pieces cannot be moved from nodes where the player is defined as the attacker (see below).
- Blue cannot move out of the goal node.

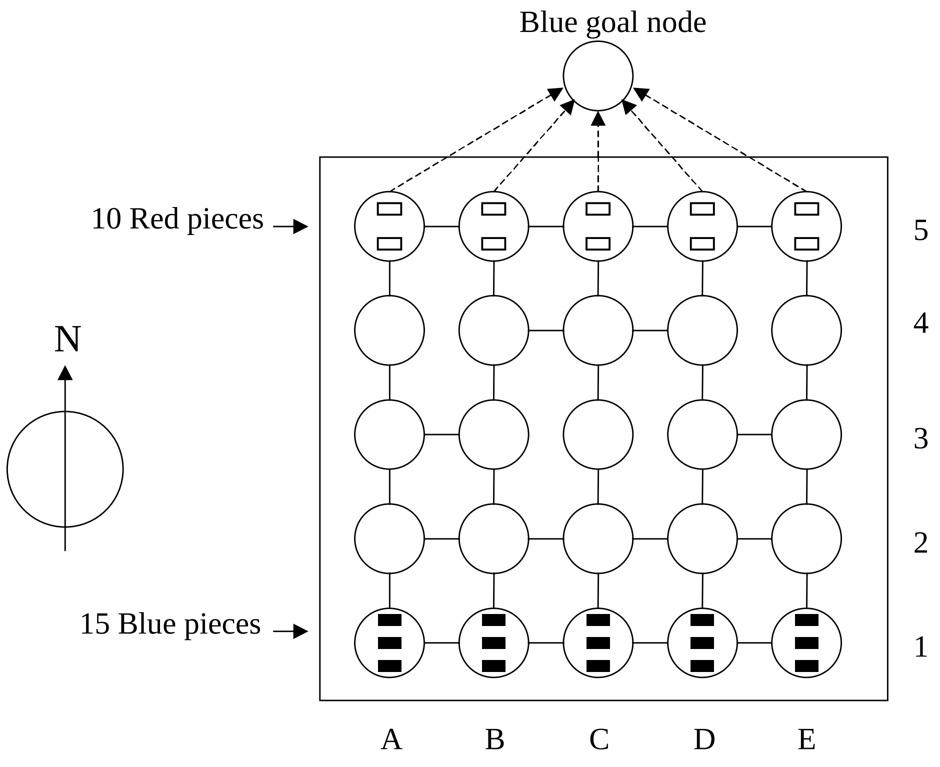


Figure 2.1 The initial set-up for Operation

Whenever Blue and Red pieces are in the same node at the end of a turn, combat ensues, and one of the pieces in that node is lost and taken out of the game. A weighted random draw decides which side loses a piece. In a node having combat, the player last entering is defined as the *attacker* of that node, while the other part is defined as the *defender* of the location. The weighted random draw is specified by the probability that Blue wins, which means that Red loses a piece. This probability is given by the fraction

(Blue strength)/(Blue strength + Red strength),

where a player's *strength* in a node equals the number of own pieces in that node, modified in accordance to two rules. Firstly, the defending part in the node gains one extra point of strength. Secondly, if the Blue player does not have an unbroken path of nodes with only Blue pieces leading from the combat node to one of Blue´s starting positions (a *supply line*), Blue loses one point of strength.

The game ends when the 36 turns are completed, or when Blue has no pieces left on the board. The result of the game is the number of Blue pieces that have reached the goal node.

2.2 Operation Opaque

The rules of Operation Lucid all apply without modifications to the game of Operation Opaque. The difference between the two games lies in the information presented to the players during the game. In Operation Opaque, enemy pieces are invisible to each player, except for pieces in the same node as or in nodes adjacent to any of the player's own pieces. Thus, Blue (say) only sees Red pieces in nodes that are at most one edge away from his own pieces (regardless of whether the Blue pieces can actually be moved or not). The goal node is not subject to this rule; Red always knows the number of Blue pieces in the goal node, while Blue pieces in the goal node do not yield information on Red pieces in the northernmost row of the board. Both sides at all times know the number of remaining Blue and Red pieces on the board.

Information is presented both before and after combat is resolved. Thus, if a player uses his turn to move a single piece into combat, the player does get information on enemy pieces in the nodes adjacent to the node he has just moved into. If the piece is destroyed in the ensuing battle, the player does not see these pieces any more (though, of course, he knows they are still there until his opponent has made a move).

2.3 Interpretation as Combat Model

The games of Operation have been designed to capture important aspects of military ground combat modelling; in particular, they represent a scenario where the goal of one side is to break through enemy defence to reach a certain location. Movement of the pieces on the board naturally represents the movement of force units in a terrain; the stochastic ordering of Blue and Red moves is intended to capture the uncertainty inherent in manoeuvring in possibly unfamiliar territory.

The rules for determining the result of combat naturally take into account the numerical strength of each side in the area of combat. In addition, they represent the advantage of being the defender of a location; this advantage is due to the defender's opportunity to prepare himself and the environs for resisting attacks. The rule regarding Blue supply lines models the effects of logistics; an invading force will need a functioning line of supplies back to his home base to be able to perform well in enemy territory.

In Operation Opaque, the rule restricting a player's view of the opponent's pieces is intended as a simplified representation of lacking information on the whereabouts of enemy forces. Also, the fact that pieces of one colour are indistinguishable assumes special importance in Operation Opaque, and reflects the problem of identifying an enemy unit when it has been detected.

2.4 The Complexity of the Problem

A seemingly obvious way of playing Operation is by evaluating (in some way) each legal move in the current game state, and then choosing the one with the best evaluation. This would reduce the problem of constructing a player agent to the problem of evaluating moves in given game states. This method is generally not feasible, however, as the number of possible moves in each state tends to be huge. A player may allocate one of at most five actions (stand still or move in either of four directions) to each of at most fifteen pieces, so an upper bound on the number of legal moves is $5^{15} \approx 3 \cdot 10^{10}$. If we assume that a computer generates one thousand possible moves each second (a reasonable assumption according to our experiences), it might take up to year to enumerate all legal moves in one state.

In typical game states the number is usually far lower than this – the player may have fewer than fifteen pieces left, and each of the pieces may not be free to perform all five actions. Also, a lot of the legal moves are equivalent, as all pieces of the same side are interchangeable. From the initial position, for instance, the number of possible non-equivalent moves for Blue is 60,112 (disregarding equivalence by symmetry). In intermediate states of the game the number of legal moves increases quickly, and is generally far too large for an exhaustive enumeration. Thus, we need a player design that is not based on brute-force evaluation of all alternatives. In Chapter 4 of this report, we present such a design.

3 EVALUATING PERFORMANCE

In this chapter, we discuss the matter of evaluating the performance of agents playing Operation. In order to place our discussion on a game-theoretically sound basis, we start by defining some theoretical performance measures. The complexity of Operation prevents us from using these measures directly; consequently, we need a way of estimating the true performance of our agents by actual game-play. In addition to describing the opponents we use for this estimation, we also present a method for diminishing the variability of the outcomes of single games caused by the randomness inherent in the game. For more information on evaluation of game-playing agents, see (14).

3.1 Theoretical Performance Measures

We view the games of Operation as *zero-sum* games, that is, games where one side loses what the other side wins. Specifically, we think of the result that *n* Blue pieces reaches the goal node as Blue winning a utility value of *n* from Red. In this case, Blue's utility from the result is *n*, while Red's is –*n*. Under these assumptions, the minimax theorem of game theory – see e.g. (21) – implies that for each of the games, there exists a so-called *minimax strategy* for each of the players. This is a pair of strategies with the property that neither of the players can benefit from changing strategy. The expected Blue score when these strategies are used is called the *value* of the game. Consequently, the value is the maximum expected score that Blue can guarantee, and the lowest expected Blue score that Red can guarantee.

Some words need to be said about the concept of "strategy" in the previous paragraph. Here, the difference between Operation Lucid and Operation Opaque – that one is perfect-information and the other imperfect-information – is essential. In Operation Lucid, there is no need for either player to act randomly, as all actions are immediately revealed to the opponent. Thus, a strategy for a player is a mapping assigning *a single move* to every possible game state in which this player has the turn. In Operation Opaque, random actions are necessary for optimal play; this means that a strategy is a mapping assigning *a probability distribution over the set of legal moves* to every game state. In this case, the expectation of the score in a game between two strategies is taken both over these probability distributions and over the random draws in the game. In practice, of course, strategies are not explicitly represented as mappings from game states to (probability distributions over) moves; these mappings are implicitly represented through the actual (possibly random) behaviour of the game-playing agents.

The games of Operation are not symmetric, as the sets of possible strategies are different for Blue and Red agents. If we were to demand that an agent should be able to play both Blue and Red, however, we could make the game symmetric. This could be done by defining a full game between two agents as a pair of single games, with each agent playing Blue once and Red once, and summing the results. We would also need a random draw to decide which agent should play which side first. This redefined game is also clearly zero-sum, and the symmetricity guarantees that its value is zero. As most of our agents have been designed to play one side only, we will not follow this procedure here.

Our goal of making agents that play well in the game-theoretic sense deserves some justification, especially as some of our research in the game of Campaign indicates that humans do not necessarily act the way game theory predicts that rational agents should act (1). However, the main source of complexity in that game is of a kind that humans in general do not handle well. The complexity in Operation, on the other hand, the main kind of complexity is such that humans in general outperform computers. Consequently, we conjecture that the correspondence between game theory and good human play is high in Operation, especially Operation Lucid. For more on different types of complexity see (10).

In the following, we describe the performance measures from the point of view of each side. The set of possible Blue strategies will be denoted by *B*, and that of Red strategies by *R*. The function $E : B \times R \to [0,15]$ gives the expected result when agents using strategies in *B* and *R* play against each other.

3.1.1 *Geq*: Equity Against Globally Optimising Agent

The first criterion we present is that of *equity against globally optimizing agent,* abbreviated *Geq*. The *Geq* measure gives the expected outcome for a strategy when playing against the opposing strategy that is most effective against it. Thus, for a given $b \in B$ it is defined as

$$Geq(b) = \inf_{r \in R} \{E(b,r)\}.$$

Correspondingly, *Geq* for a given $r \in R$ is

$$Geq(r) = \inf_{b \in B} \{-E(b,r)\}$$

– since Red's goal is to minimise the outcome. In a game with value *v* for Blue, it is clear that $Geq(b) \leq v$ for all $b \in B$ and $Geq(r) \leq -v$ for all $r \in R.$ The strategy *b* is a minimax strategy if and only if $Geq(b) = v,$ the strategy *r* if and only if $Geq(r) = -v.$

An equivalent way of understanding the *Geq* measure is as the expected outcome of players when playing against a hypothetical benchmark agent that always knows and exploits the strategy of its opponent – that is, a globally optimising agent. Of course, such an agent, which maximises its payoff against an *arbitrary* opponent, cannot belong to *B* or *R,* as it requires information on its opponent as well as on the game state.

The *Geq* criterion is very strict. Its "knowledge" about the player's strategy can be utilised in ways that seem quite unfair, such as using inferior moves to lead the player into states where it has a weakness. Consequently, this criterion may give very low values even for apparently reasonable players, especially in complex games.

3.1.2 *Peq*: Equity Against Minimax Playing Agent

A weaker criterion than *Geq* is that of *equity against minimax playing agent*, or *Peq* (the "P" indicates "perfect play"). It is defined as the expected outcome of a player when playing against an agent which follows a minimax strategy. Thus, if $r^* \in R$ plays according to a minimax strategy, *Peq* for a player $b \in B$ is

$$Peq(b) = E(b, r^*),$$

and if $b^* \in B$ plays a minimax strategy, *Peq* for a Red strategy *r* is

$$Peq(r) = -E(b^*, r).$$

As the minimax player by definition is guaranteed to do as well as possible against its most dangerous opponent, it follows that $Peq(b) \leq v$ and $Peq(r) \leq -v.$ If the agent being evaluated itself plays a minimax strategy, the *Peq* criterion takes the value of the game (*v* for a Blue and –*v* for a Red strategy). However, the implication is not reversed: a sufficient condition for achieving (say) $Peq(b) = v$ is that *b*'s mixed strategy does not include any pure strategy absent from the minimax strategy. Consequently, mixture of pure strategies is not required: any pure strategy represented in the minimax strategy will have a *Peq* equal to the value of the game. This in turn means that the *Peq* measure is too weak to be of much use in imperfect-information games, where minimax strategies are generally necessary. An additional disadvantage with this criterion is that in a game with multiple minimax strategies, the *Peq* values depend on the strategy selected for *b** or *r**.

While *Geq* represents the ultimately devious opponent, *Peq* corresponds to an opponent that plays perfectly in a more defensive way. It never makes any errors and, while punishing inferior pure strategies, never exploits biases in the opponent's strategy mixture. It is clear from the definitions of *Geq* and *Peq* that $Geq(s) \leq Peq(s)$ for any strategy $s \in B \cup R$.

Most of the preceding remarks pertain mostly to imperfect-information games, such as Operation Opaque. In Operation Lucid, mixture of strategies is never necessary due to the perfect-information properties of the game.

3.2 Estimating Performance by Reference Players

The problem with the performance measures defined in the previous section is that they cannot be calculated except in games for which the solution is known. Both varieties of Operation are clearly far too complex to be solved. What, then, is the use of these measures – is it not more natural to measure performance by how agents succeed in actual game-play?

The answer is that this is exactly what we do. However, game-play on its own can also be a misleading guide to true playing strength in the game-theoretic senses defined above. In fact, it is often possible that agents may outperform each other in circle, even in the long run. In this case, it is impossible to rank them in a consistent manner by playing them against each other. The usefulness of the theoretical measures is then as guides to what kind of agents we should use as opponents for the agents to be evaluated.

To estimate the *Peq* ability of agents well by using a benchmark opponent, we would need an agent that plays the solution, or at least plays sufficiently like the solution to rank its opponents correctly. As our goal in the first place is to make agents that plays strategies close to the solution, this is clearly not feasible in any strict sense. Nevertheless, by making reference opponents that play according to the principles of minimax strategies as much as possible, we can hope that these will establish an ordering of agents that corresponds reasonably well to their *Peq* ranking.

The task of estimating the *Geq* measure is even harder. Actually trying to find the most effective counter-strategy against each agent is clearly too demanding, especially when we want to evaluate a multitude of agents. Making a globally optimising agent, that is allowed to peek into any opponent's strategy and use this information to exploit its weaknesses, is even worse – indeed, it is far more difficult than our original task. A possible approach that is feasible to implement, and which we may hope captures some of the strictness of the *Geq* criterion, is to play our agents against a diverse suite of reference opponents, and evaluate them by the score against its most effective opponent.

We implemented a number of simple benchmark players, both for Blue and Red, to act as evaluators for our more complex agents. These reference agents are described in the following.

3.2.1 Reference Players

The simplest players we implemented – one Blue and one Red, called *SimpleBlue* and *SimpleRed* respectively – were made primarily as an exercise in using the software environment. The Blue agent moves every piece north that can be moved north at every turn it wins, while the Red one does nothing but stand still.

Of course, these agents do not play very well by any standard. The Blue player spreads its forces in a way such that every piece is at great risk of being lost in battle, while the Red player never fills holes that might arise in its line of defence. Clearly, we require more sensible behaviour than this for evaluation, so we implemented other, still rather simple, agents for this purpose. These agents, one Blue and two Red, are now described in short.

OneAxisBlue:
This Blue agent initially decides on an axis along which to conduct its attack. At its turn, all movable Blue pieces on this axis are pushed forwards, while pieces on other axes are moved towards the attack axis as much as possible – this can clearly only happen on the southernmost row. The strength of this agent lies in its concentration of forces, its weaknesses in the fact that it does not keep a supply line for long, and also that it does not try to evade the defence.

AxesExpRed:
Like the SimpleRed agent, this agent always keeps its pieces on the northernmost row. It is more advanced is in its sideways movements, however, as it seeks to place its pieces where the Blue threat is greatest. For this purpose, it calculates a threat for each of the five northern nodes, which for a given node is a weighted count of the Blue pieces present on that axis. The weights reflect how close the threat is, and are set to 5 for the northernmost row, 4 for the next row, and so on, counting down to 1 for the southernmost row. The agent then attempts to distribute its own pieces on the five axes in a way mirroring the perceived threat. An exception to this rule occurs if there is combat going on the northernmost row. In this case, a simple expert system, made using the ExperTalk framework (2), is used to prevent moves that are clearly disadvantageous for Red's performance in the combat.

DistExpRed:
This agent is identical to AxesExpRed, except for the way in which the threats to the five nodes are calculated. Here, the threat posed to a given northern node by a given Blue piece on the board is $1/d^2$, where *d* is the length of the shortest path from where the Blue piece currently is, through the node we are calculating the threat for, to the goal node. The sum of contributions like this from all Blue pieces on the board – except from those that are in other nodes on the northernmost row – is taken to be the threat to the node in question. Thus, *all* Blue pieces on the four southern rows contribute to *all* the threat estimates. The point of

calculating threat in this way is to allow for the risk that Blue may manoeuvre in order to evade the defence – in contrast, AxesExpRed does not attempt to cover an axis before there is actually a Blue piece on it.

3.3 Compensating for Randomness

In addition to the problems with estimating performance addressed in Section 3.2, there is the added complication of the random draws made during the game, both in deciding whose turn it is and in resolution of combat. These draws may have a large impact on the score in each game. Consequently, it is necessary for any pair of agents to play a number of games in order to get a good estimate of the expected outcome. The number of games needed for a reasonably accurate estimate of expected outcomes in Operation may be inconveniently large; this section describes a method for reducing the number of games required for a given required accuracy.

3.3.1 Variance Reduction by the Control Variate Method

The basic idea for reducing the random noise in the game outcomes is to estimate the influence that the distribution of "luck" between the players has had on the outcome, and adjusting the outcome according to this estimate. Formally, if we let Y be the outcome of a game and l our estimate of the advantage (positive or negative) that Blue has had due to the random draws in the game, the adjusted outcome is $\tilde{Y} = Y - l.$

When we play two agents *B* and *R* against each other, the expected outcome $E(B,R)$ is what we are trying to estimate. If the adjusted outcomes $\tilde{Y}$ are to be of any use, we therefore require that $E(\tilde{Y}) = E(Y),$ which means that we need estimates l with the property $E(l) = 0.$ Here, we define an estimator with this property, based on linear regression.

The two kinds of random draws in Operation are those that decide whose turn it is next and those that determine who loses a piece in a combat node. The assumption behind our simplest estimator is that a large part of the variance in the game outcomes can be described as a linear function of two variables describing the aggregated advantage Blue has had during a game with respect to these two kinds of draws. Thus, we use the model

$$Y = a_0 + a_D D + a_C C + \varepsilon$$

– where *D* represents the luck Blue has had in the draws for turns and *C* his luck in the combat draws, the *a*'s are the coefficients of the linear function, and ε is an error term. The adjusted outcome of the game is then set to

$$\tilde{Y} = Y - a_D D - a_C C.$$

If we construct *D* and *C* such that $E(D) = E(C) = 0,$ this estimator will then be unbiased. Since both sides always have equal probability of winning a turn, this is fulfilled for *D* if we let it be equal to the total number of turns Blue has won throughout the game, minus the number of turns Red has won. The case of *C* is a little more complex, as the probabilities of Blue winning is usually different from ½. If we denote the outcome of a combat draw by $k \in \{0,1\},$

where $k = 1$ if Blue wins the draw and $k = 0$ otherwise, and we let $p \in (0,1)$ be Blue's probability of winning the draw, then $k - p$ is our estimate of Blue's luck. Since

$$E(k - p) = \Pr(k = 1) \cdot (1 - p) + \Pr(k = 0) \cdot (0 - p) = p(1 - p) - (1 - p)p = 0,$$

C is unbiased if we let it be the sum of the $k - p$ values for all the combat draws in the game.

The question now is how to find the values of a_D and a_C. We could use a set of reference agents and use linear regression to determine a_D and a_C once and for all. Although the lack of bias ensures that the expected mean of the adjusted scores would be correct in this case, we would have no guarantee that the variance was actually reduced. This is because just as the expected outcome of a game depends on the agents that are playing, the influence of the draws on the results may also depend on the actual agents used.

A mathematically sound way of determining a_D and a_C for a given pair of players who have played N games is to run a new regression for adjusting each game, where only the other $N - 1$ games are used as data for the regression analysis. When the number of games grows large, however – as it does when we need high accuracy in our score estimates – this approach requires a correspondingly large number of regression analyses. Also, with a large number of games, the error caused by statistical dependency if we simply use all games both for regression and expected score estimation becomes negligible (24). Consequently, this latter method, though erroneous in principle, was adopted as sufficient for our purposes.

The simple model above assumes that winning a draw of a given type is equally useful in any board state, an assumption that is clearly not realistic. Therefore, we extended the model by adding four new random variables, with corresponding coefficients, intended to capture the relative importance of draws.

One of the additional variables, $D^{(S)}$, measures the amount of luck Blue has had in particularly significant turn draws. These significant turn draws occur when Blue has pieces in the northernmost row. These draws may be particularly important for the outcome, as they can determine Red's chances of destroying Blue pieces before they reach their goal. For each of the five northernmost nodes, let s_{node} be the product of the number of Blue pieces in that node and the number of Red pieces in its two neighbouring nodes. If we let D_i be the luck associated with a single turn draw, so that D_i is 1 if Blue wins the turn and –1 if Red wins it, then $D^{(S)}{}_i = S_i D_i$, where S_i is the sum of the five s_{node} values. $D^{(S)}$ is then the sum of all the $D^{(S)}{}_i$'s throughout the game.

The other three new variables measure combat luck in specific situations. All of these are similar to $D^{(S)}$, as for each draw they are multiples of the basic combat luck variable. One, $C^{(S)}$, is somewhat parallel to $D^{(S)}$; the multiplier in this case is the total number of pieces in the combat node if it is on the northernmost row, and zero otherwise. For the variable $C^{(B)}$, the multiplier is simply the number of Blue pieces remaining on the board. Finally, the variable $C^{(G)}$ captures the fact that the importance of a combat draw depends upon the chances the

Blue pieces have of reaching the goal node before 36 turns have passed and the game is over. If Blue is the attacker in a combat node, all Red pieces must be defeated before Blue can exit the node; consequently the number n_G of moves needed to reach the goal equals the sum of the distance to the goal and the number of Red pieces present. (It is assumed that Red neither exits the node himself, nor reinforces the node with additional pieces, making Blue the defender.) The multiplier for this variable is set to zero if n_G is greater than the number n_l of remaining turns, and to $1 - n_G / n_l$ otherwise.

Since, for a given draw, each of these multipliers has a constant value, and the expected value of the contribution of the draw to *D* or *C* is zero, the additional variables defined here will fulfil the no-bias requirement.

As mentioned, the usefulness of these models for variance reduction may depend on the actual players involved. Table 3.1 shows the estimated means along with standard deviations for the agents described in Section 3.2.1, in the three cases of no variance reduction, the simple regression model and the expanded model. For each pairing, 100 games were played.

	Var. red.	**SimpleRed**	**AxesExpRed**	**DistExpRed**
SimpleBlue	**None**	4.39 (0.26)	4.52 (0.30)	3.78 (0.29)
	Simple	4.21 (0.09)	4.11 (0.11)	3.86 (0.09)
	Expanded	4.21 (0.05)	3.98 (0.08)	3.95 (0.07)
OneAxisBlue	**None**	12.76 (0.28)	4.89 (0.33)	5.50 (0.34)
	Simple	12.75 (0.06)	5.21 (0.13)	5.37 (0.14)
	Expanded	12.78 (0.06)	5.31 (0.10)	5.34 (0.11)

Table 3.1 Estimates of expected outcomes, with standard deviations in parentheses, without and with variance reduction

Table 3.2 shows the factor by which the number of games would have to be increased to achieve the same standard deviation, for the following three cases: no variance reduction compared to the simple model, no variance reduction versus the expanded model, and the simple model compared to the expanded model.

	Comparison	**SimpleRed**	**AxesExpRed**	**DistExpRed**
SimpleBlue	**None vs simple**	8.4	7.0	9.8
	None vs expanded	23.4	13.8	17.9
	Simple vs expanded	2.8	2.0	1.8
OneAxisBlue	**None vs simple**	20.2	5.9	5.8
	None vs expanded	24.7	10.4	10.2
	Simple vs expanded	1.2	1.8	1.7

Table 3.2 Multiplicative effect of variance reduction on the number of games necessary for a given standard deviation

The table shows clearly how the effect of the variance reduction varies between pairs of players, although it should be noted that the two "simple" agents are rather trivial, and may not yield very interesting results. Still, it is clear that compared to the relatively low cost of running the variance-reducing algorithms, the gain from using these methods – especially the more complex one – is considerable.

For further reading on variance reduction by control variates, see for instance (24).

4 A GENERIC PLAYER DESIGN

As mentioned in Chapter 1, we can identify two general ways for an agent (human or machine) to select moves in games, which we may call the *constructive approach* and the *evaluation approach*. These approaches, in their pure forms, are both seen to be inappropriate when considering our problem, that of playing Operation Lucid. The main contribution of this chapter and the next is a mixed approach that *combines* these approaches by utilising the valuable properties from each. This approach allows us to create agents that play the game efficiently.

4.1 Two Common Approaches

This section describes the constructive approach and the evaluation approach to game playing, along with their advantages and disadvantages when applied to our problem.

4.1.1 The Constructive Approach

In each state, an agent using the constructive approach forms its next move constructively according to the state and the rules of the game. Typically, this is done by associating actions to properties of the state, or by defining goals and selecting moves which are seen as leading to these goals. This is how humans typically think when playing games of the same class as Operation Lucid (26, p. 140). An example of a constructive software agent is an agent using an expert system for inferring its next action from the current game state.

An advantage of this approach is that it reduces the complexity of the decision-making process: there is no need to consider all possible actions in a given state.

The main disadvantage is that the complexity of the domain itself makes it difficult to define rules or goals with sufficient power of expression to achieve high-performance game play. Obviously it is quite challenging to utilise the potential of the huge set of alternative decisions, *without* addressing the complexity of this decision situation. Thus, constructive agents may typically behave vulnerably, by playing revealing and rigid strategies.

4.1.2 The Evaluation Approach

In its pure form, the evaluation approach is based on evaluating (in some way) all possible actions in each game state the agent encounters, and choosing the action with the highest evaluation. To achieve this, the agent initially performs a search, which generates all possible actions. This search is defined by the current state and the rules of the game. Having the actions at hand, an evaluator assesses them in accordance with some criterion, and the action that is seen as the best one is chosen.

This basic form of the evaluation approach is illustrated in Figure 4.1. The *Move Generator* constructs the set of legal actions using the current situation and rules of the game. The *Move Evaluator* then evaluates these actions according to certain criteria, and the action with the best evaluation is chosen.

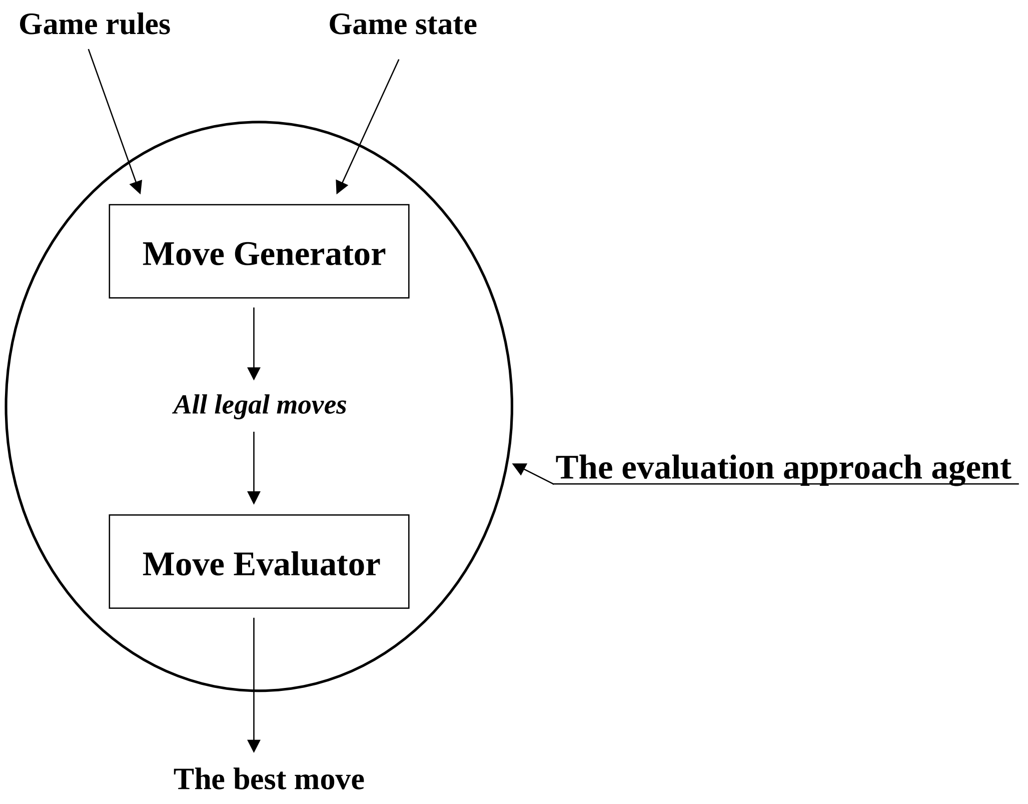


Figure 4.1 Design for an agent using the evaluation approach

This approach, in a more sophisticated form, is often used by computer programs playing games with a reasonably small number of legal actions – chess is a well-known example of such a game. If a large number of possible actions need to be generated and evaluated in each game state, this approach becomes infeasible. As mentioned in Section 2.4, the number of possible actions in a given Operation game state can be of the order 10^{10}, which makes a pure evaluation-based approach too expensive.

On the other hand, the design shown in Figure 4.1 has the advantage of being *modular*. The agent is decomposed into a module generating the set of moves to be evaluated and a module performing the evaluation itself. Due to this modularity, the agent development is simplified – changing the implementation of one module does not affect the other. If we want to experiment with different designs for one of the subtasks, this is a valuable property of the evaluation approach.

4.2 A Modular Mixed Approach

In designing an agent for playing Operation Lucid, we wish to exploit the desirable properties of each of the basic approaches mentioned, that is, the complexity reduction of the constructive approach and the modularity and expressivity of the evaluation approach. Our solution is to use the evaluation approach as a starting point, but limiting the set of actions generated. This is achieved by splitting the move generator of Figure 4.1 into two parts: a *Constraint Generator* (CG) that constructively generates a high-level description of the moves to be generated, and a *Constrained Move Generator* (CMG) that generates the set of moves fitting this description. The *Move Evaluator* (ME) is retained. This design is shown in Figure 4.2.

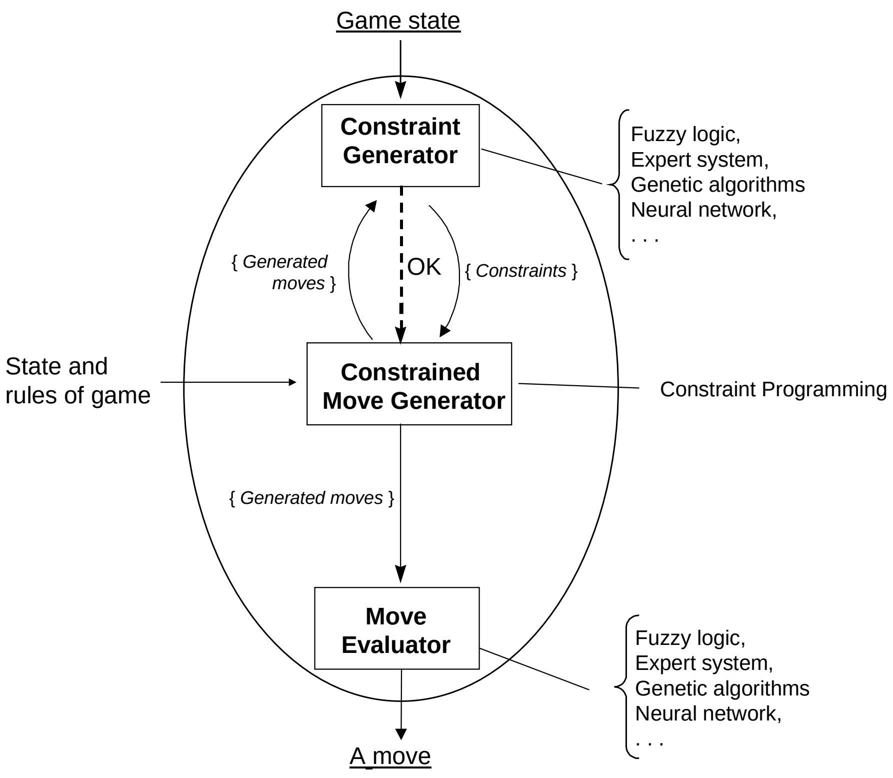


Figure 4.2 Design for an agent using a mixed approach

When the agent in Figure 4.2 is to perform an action, it takes the current game state as its starting point. The CG uses some AI method (as indicated in the figure) to make a set of constraints describing the kind of move it sees as appropriate. The CMG receives these constraints, and generates the complete set of legal moves in accordance with the game rules, the situation description and the constraints from the CG. It then sends this set back to the CG. This feedback allows the CG to check its constraints, and modify them if necessary. If, for instance, the constraints were too strict, so that no legal action exists, this test would force an easing of constraints.

When the CG has accepted the CMG actions, the generated moves are sent to the ME. The ME evaluates each of the candidate actions. The evaluation criterion can be anything from simple heuristics to an advanced AI technique. The output of the agent is the move that is given the highest score.

In this design, a game-playing agent consists of separate modules that may be implemented using different techniques. Thus, the design enables hybrid implementations of agents.

Note that the two common approaches mentioned earlier are included in this general design. If the CG generates a set of constraints describing one single move in detail, only this move will be generated by the CMG, and the ME is made redundant. This amounts to a purely constructive approach. If, on the other hand, the only constraints received by the CMG are those describing the rules of the game, all legal moves will be generated, and we have the pure evaluation-based approach.

As seen from Figure 4.2, applying our design for making an agent calls for the implementation of three modules: the constraint generator, the constrained move generator and the move evaluator. The design obviously does not specify in detail how the CG should act or how the ME should evaluate the moves – these are precisely the tasks which we are interested in studying when creating game-playing agents, and various techniques should therefore be examined. On the other hand, the task of the CMG *is* fully specified, given a well-defined language for specifying constraints. This task can be formulated as follows: Generate all legal moves satisfying the constraint set received from the CG. The fact that the task of the CMG is fully specified leads us to the conclusion that only the efficiency of performing this task separates between CMGs. Thus, provided that we are able to find a sufficiently efficient implementation of this module, we only need *one* CMG common to all agents we create. The implementation of an efficient CMG module is the subject of Chapter 5.

5 A MOVE GENERATOR BASED ON CONSTRAINT SATISFACTION

The modular mixed approach still presents a potential complexity problem: How is the constrained move generator supposed to decide which actions fit the high-level description from the constraint generator – the constraint set – without generating and testing all possible actions? Here we have found *constraint satisfaction problems* (CSP) – see e.g. (12) or (35) – to be a very useful framework.

The task of generating all possible moves can be seen as traversing a search tree, where each leaf node corresponds to a legal action. As the number of legal actions is very large, standard search strategies like breadth-first and depth-first are inappropriate in this context, as these strategies visit *every* node of the search tree. The point of using algorithms solving the CSP is that branches of the tree are cut as soon as a constraint is violated, thus making search feasible.

Despite cutting branches of the tree, the complexity of CSP algorithms can be huge, depending upon the size of some critical parameters – in our problem, the number of pieces is the most significant parameter. In fact, CSPs include as a special case the well-known NP-complete problem 3SAT (26, p. 84), and we can not expect to do better than exponential complexity in the worst case. However, in our context, the significant parameters affecting the complexity of the CSP are bounded, and therefore the complexity is also limited. Still, depending upon the size of our actual parameters, the actual CSP formulations may be quite complex and fast executions of the search are not always to be expected.

In the following section, a general description of CSP is given, before we return to the description of the CSP-based constrained move generator module.

5.1 Constraint Satisfaction Problems

A constraint satisfaction problem is basically a problem formulation involving a set of variables and a set of constraints, where the constraints define the solution space by means of dependencies between subsets of the variables. The variables are usually defined to have finite domains. A solution of the problem is an assignment of a value to each variable, such that no constraints are violated. Algorithms for solving CSPs typically work by sequentially assigning values to the variables and checking for violations of the constraints.

This general solution method outlined above has been refined in algorithms such as *forward checking* (15) and *maintaining arc consistency* (27). Many specific algorithms and applications have been devised within problem domains such as scheduling (25), vehicle routing (8) and timetabling (20); a survey of constraint satisfaction algorithms is given in (19).

5.2 Implementation of the Constrained Move Generator

We now describe how we have applied constraint satisfaction methods to the problem of generating the set of legal moves satisfying a set of constraints from a constraint generator. We start by describing the input and output of the constrained move generator, and proceed to show how the task of the CMG is framed in CSP terms – that is, how the variables and constraints are defined.

5.2.1 Input and Output

The input to the CMG consists of three parts: the constraint set created by the constraint generator (see Figure 4.2), the rules and the current state of the game. The constraint set and constraints are described in more detail in Section 5.3; in particular, Section 5.3.2 defines a protocol the CG uses for communicating a part of the constraint set to the CMG.

Due to the perfect-information property of Operation Lucid – both players know the full state of the game at all times – it is not necessary to include the history of the game in the state description. The state description comprises the position of each piece on the board (including Blue pieces in the goal node), the number of remaining turns, and which side was the last to enter each combat node.

The output of the CMG module is a set of moves. This set represents a subset of the set of all possible moves in the current state – those that satisfy the constraint set. The format of a move is a list of two-tuples. Each tuple describes how one piece is moved, by giving the names of the current node and the one to which the piece is to move. For instance, moving all Red pieces forward from the starting position (see Figure 2.1) is formulated as

$$((A5,A4),(A5,A4),(B5,B4),(B5,B4),(C5,C4),(C5,C4),(D5,D4),(D5,D4),(E5,E4),(E5,E4)).$$

Consequently, the output of the CMG module is a set of lists, each lists having the format seen above.

5.2.2 Formulation of the Constraint Satisfaction Problem

Now we describe how the problem of generating the moves is formulated as a constraint satisfaction problem. Referring to the description of CSPs in Section 5.1, we note that we need to define the variables of the search, the (finite) domains of the variables, and the constraints. The constraints are described in detail in Section 5.3 and will not be commented on here.

We associate one variable to each of the agent's own pieces still on the board, excluding those that have reached the goal node. Each of these variables has a domain consisting of the five values *north, south, east, west,* and *remain*. Thus, the values of a variable correspond to the actions available to the piece in a single move.

The goal of the CSP solution search is to find all ways each variable can be assigned one of the five values, so that the constraints from the CG are satisfied. Using the search tree interpretation in the introduction to this chapter, we see that a node at depth *k* in the search tree corresponds to an assignment of movements to *k* pieces. It follows that the search tree for Blue (Red) is of depth 15 (10) at the start of the game, before any pieces are lost through combat or safely in the goal node.

Having framed our problem as a CSP search, standard techniques for solving CSPs can be applied. We have chosen to use a commercial software product for this task, namely the C++ library *ILOG Solver,* version 4.4 (17).

5.3 The Constraints

The main reason for using constraint programming in this game application is the size of the search space. The main effect of the constraints should be to reduce the search space substantially. How efficient a constraint or combination of constraints "cuts" the search space is not well known before the search is actually done – sometimes the reduction may fail so that the search has to be stopped before completion, at other times no solution may exist (the problem is *over-constrained*). Neither of these two situations is desirable – a manageable, non-empty set of solutions is preferred. A standard approach for achieving this is to consider some of the constraints as *soft*, that is, not necessarily satisfied. This approach, also called *partial constraint satisfaction* (13), allows that only a subset of the constraints are active; this permits additional acceptable value combinations in the solution.

A variant of the partial constraint satisfaction approach is applied here. All moves coming out of the CMG have to satisfy *all* constraints, but if the set of legal moves is too small, the CG adjusts the constraint set by loosening some of the constraints, and posts it to the CMG. This means that the agent does the "soft" part outside of the CSP module itself, by means of the looping between CG and CMG (cf. Figure 4.2).

To avoid time-consuming search, a strategy for the CG is to start with a strict constraint set (generating no solutions), and iteratively loosen the constraint set until a proper set of legal moves arises.

In the following we describe the applied constraints; these are divided into constraints that affect the whole board (Section 5.3.1), and constraints treating sub-areas of the board (Section 5.3.2).

5.3.1 Global Constraints: Rules and Heuristics

Constraints implementing the rules:
The rules of the game represent a minimal reduction of the search space; all CSP searches are constrained by the rules. Three different types of rules constitute the actual constraints:

1. Some nodes do not have edges in all four directions (cf. Figure 2.1).
2. When a piece is an attacker, it is not allowed to move out of the node.
3. A node can maximally contain three pieces of the same side.

The first type is easily implemented by removing subdomains of the variables *before* starting the search. If, for instance, a piece is located in node A4, the directions *west* and *east* are removed from the domain of the corresponding variable.

The second type is implemented in a similar way; the variables belonging to immovable pieces have all values except *remain* removed from their domains.

The last type is an example of typical CSP constraints, as the constraint dynamically affects the domains of the variables *during* the search. For instance, suppose that four Blue pieces is placed around node B2, that is, the nodes A2, B3, C2 and B1 each contains one Blue piece. If the variables of the first three of these pieces are given the values *east*, *south* and *west*, respectively, the node B2 is full, and the value *north* is removed from the domain of the fourth variable.

Heuristic constraints:
Human players of Operation Lucid typically develop some rules of thumb when playing the game. The negative version of these rules, that is what to avoid, is utilised to make heuristic constraints, the purpose of which is to reduce the search space as efficiently as possible *without* removing all the “good” moves. The heuristic constraints we have identified are listed below:

1. The pieces should not occupy more than a *limited number* of different nodes when the move is completed.
2. No more than a *limited number* of axes should be used for forward (backward) movements during one move. An *axi*s is a chain of consecutive north–south edges, for example the chain of edges from node B1 to node B5.
3. No more than a *limited* number of pieces may be moved into each of the five directions. The number is set independently for each direction.

These three groups of constraints all affect the domains of the variables dynamically, like the last group of rule-implementing constraints above. Note that despite the intention of not removing all the good moves, this is of course not guaranteed. This is due to the very complex mapping between the action performed and its effect on the result.

All three groups of constraints are parametric, that is, each constraint can be adjusted to remove more or less of the search tree. This is a desirable property when adjusting the number of moves coming out of the CMG by means of the loop in Figure 4.2. All parameters are integer valued and mentioned by the phrase "limited number" in the list above.

5.3.2 Local Contraints: The Area Language Protocol

In addition to global constraints, based on the rules and our practical experience of the rules, we also use local constraints, which are expressed through a protocol, namely the *area language* (ALA) protocol. By means of this protocol a large variety of constraints, and consequently playing styles, can be expressed. Without a general protocol like ALA, making a new agent would mean starting from scratch and having to define and implement new concepts and constraint types similar to the ones in Section 5.3.1.

ALA is a protocol for the communication between the CG and the CMG (cf. Figure 4.2). An ALA constraint specifies the minimal number of pieces required in a sub-area of the board. The syntax of ALA is as follows:

Let M_i denote a sub-area consisting of the set of nodes $\{node_1, node_2, \ldots, node_k\}$. Then a general ALA constraint set reads:

$$\{(M_1, n_1), (M_2, n_2), \ldots, (M_l, n_l)\},$$

where each n_i is an integer. The interpretation of this constraint set is that all moves generated in accordance with this statement needs to have at least n_1 pieces in the set of nodes M_1, at least n_2 pieces in M_2, and so on. An example constraint set is $\{(\{B2, B3, C2\}, 1), (\{E5\}, 3)\}$, the interpretation of which is that there should be at least one piece in one of the nodes B2, B3, and C2, *and* there should be three pieces in node E5. Note that *all* sub-area restrictions have to be satisfied, due to the interpretation (logical AND) of the relationship between the constraints in a constraint set.

It is easily seen that ALA can express *all* possible moves – any move can be specified completely by letting each M_i consist of one node, and setting the corresponding n_i to the number of pieces in the node after the move has been made.

ALA has several advantages, such as expressivity and a well-defined format. Still, we have found that combining ALA with heuristic constraints compares favourably to using only ALA constraints. This is because the heuristic constraints express general knowledge about good play that holds independently of the location on the board. ALA is not appropriate for expressing this independence of location.

6 USING THE DESIGN (1): A NEURAL NET AGENT

This chapter and the next describe our use of the agent design in implementing actual game-playing agents for Operation Lucid. First, we present a pair of agents, one Blue and one Red, using a neural net (NN) trained from self-play by the temporal difference algorithm TD(λ).

6.1 Constraint Generation

The initial CG-modules of the agents are very simple. Only global constraints are used, and they are static, except for the *distribution* constraint, that is, the constraint specifying the maximum number of nodes with pieces. The maximum distribution is set as a function of the number of remaining pieces:

$$\text{Distribution} \le 1 + \frac{\#\text{pieces}}{3}.$$

Experiments have shown that the chosen set of constraints tends to give a reasonable number of moves, typically between 10 and 100, and that they appear to allow at least a few sensible ones. In Section 6.5 we will describe a more complex constraint generator for Blue, designed to mend some weaknesses in the NN evaluator; this improved CG also uses ALA constraints.

If we for a moment view the constraints as game rules, we have defined a simpler version of Operation Lucid. This modified game is in many ways similar to the dice game backgammon, both being stochastic two-player zero-sum games with perfect information, and with similar branching factors. Gerry Tesauro has showed that neural nets trained from self-play using TD-learning can learn to play backgammon at championship level (34). We wish to test if the same algorithm can be used successfully for our simplified version of Operation Lucid. Apart from being an interesting application in its own right, this also serves as a test of how generally applicable TD-learning in stochastic games is. The main difference between backgammon and our simplified Operation Lucid game is that the former has a one-dimensional geometry, while the latter is two-dimensional. Another important difference is that the dice rolls of backgammon often force checkers to certain locations, while the pieces in the present game are not forced in the same way.

6.2 Evaluating Moves by a Neural Network

The NN of our design is used for choosing a move from the list generated by the CMG, placing it in the lower box of Figure 4.2. Rather than evaluating the *moves* as such, the net equivalently evaluates the *positions* – game states – they produce. Because the game has perfect information, each game state can be seen as the starting point of a new game, and therefore has a *value*, according to game theory (21). The value is the expected number of Blue pieces that reach the goal location, given perfect play by both sides. The NN works as a black box function that takes the game state as input and gives an estimate of the value as output. Note that the same NN can be used for playing both sides of the game. Blue chooses moves that maximize the NN output, and Red chooses moves that minimize it.

The NN we use is a simple feed-forward net with one layer of hidden nodes and sigmoid activation functions, and back-propagation of errors as update rule for the weights. This is a “vanilla-flavored” design, which has been shown to work quite well for many different problems. For a general introduction to NNs, one reference is (16).

As shown by Tesauro (34) and others, the input representation fed into the NN can affect the performance significantly. We will not go into the details here, but our input representation contains both “raw” board features (the number of pieces in different locations) and more high-level features of the position. Examples of these are the number of Blue pieces placed on open axes and the number of Blue pieces in defensive combat posture. In total there are 66 input features. Our NN has only one output node, and the design chosen limits the output to the range (0,1). The value of states has the range (0,15), so we multiply the NN output by 15. Our NN has 20 hidden nodes.

6.3 TD(λ)-learning

The basic idea behind temporal difference learning (33) is that the final outcome, and the course of the game, is used to produce feedback signals to the NN for the states visited in the game. A game that has been played to the conclusion gives a sequence of game states. When calculating the feedback signal for one of these states *s*, the algorithm takes a weighted average of the game outcome and the NN’s own evaluation of the states between *s* and the end state. A parameter λ controls how much weight is placed on states that are close to or far from the state *s* in time. Values of λ close to 1 place most of the weight on states close to the end (and the actual outcome), and value close to 0 place most of the weight on states immediately succeeding *s*. A NN trained by TD(λ) works towards predicting the expected end-state value of a stochastic process. The fact that the stochastic process is controlled by the very same NN complicates the picture, but the experience from backgammon suggests that this may not be a big problem.

For our implementation of the training procedure three agents are used: a Blue agent, a Red agent and a Referee agent. The Referee holds the NN, but the Blue and Red agents are granted access to it, and use it in determining their moves. The Referee conducts the game by maintaining the game state, and randomly draws turns and combat outcomes according to the rules. The Referee informs Blue and Red about the current game state, and the side that wins a turn responds with a move. After the game is finished, the Referee updates the NN according to the TD(λ) algorithm. This procedure is then repeated for a large number of games. The starting point of the algorithm is a NN with random weights.

6.4 Experimental Results

Our first experimental results were very negative, as the NN did not appear to learn anything about the game, and the Blue side never got any pieces to the goal location. When inspecting the game logs we quickly found out what was happening; Blue did not move forwards. It moved its pieces to apparently random nodes, and then stopped. However, the TD-algorithm actually did its job correctly, as the NN output was very close to zero. So the NN was indeed able to predict the outcome of the games it played, except that it played the game poorly for one side. As long as there had never been any game with positive feedback, the NN had no means of distinguishing good states from bad. This does not happen in backgammon, because the rules of that game forces the checkers forwards. Our problem was that the algorithm did too little *exploration* of the state-space of the game. We were able to correct this by adding some random noise to the games, sometimes drawing a move at random. Of course this was more necessary in the early phases of training, and the frequency of the randomising procedure was reduced with training time.

With the randomising procedure the results were pretty good. The NN learned that Blue should seek openings in Red's defence, and conversely that Red should cover axes with many Blue pieces. It learned that one should always attack with the maximum number of pieces possible, and that forward movement is more urgent for Blue closer to the end of the game.

We have tested the Blue version of our NN agent against the Red benchmark agent DistExpRed (see Section 3.2.1).. The average performance started at 0, as the initial Blue agent did not even know that it should move forwards. After approximately 10,000 games of training the average performance reached approximately 5.5.

All in all it appeared to play a relatively sound game, with one exception. It never learned that Blue should keep an unbroken supply line to his planned combat nodes. In retrospect this is only natural, as the NN had no input feature telling it if it had these supply lines. In theory a NN should be able to learn this from the raw input features, but connectivity in a graph is a kind of problems that NNs handle poorly. Also a supply line would have to show up by chance first, so that the NN could learn about its importance, and this is very unlikely.

6.5 An Improved Constraint Generator

To correct our Blue agent's problem with supply lines we modified the constraint generator. Simply put, we added a module that first chose a possible supply line, and then posted a constraint in the ALA-syntax that had the desired effect of maintaining this line. Assuming that we wish to have a supply line to the node A5, we can post the constraint set

$$\{(\{A1\},1),(\{A2\},1),(\{A3\},1),(\{A4\},1)\},$$

which means that at least one piece should be allocated to each of the nodes A1 to A4. This improved constraint generator has to loop as indicated in Figure 4.2, because it will sometimes first attempt to produce supply lines that cannot be constructed. The module also considers the number of turns left in the game, as it is less desirable to keep a supply line when there is little time left of the game.

When pitted against DistExpRed, the Blue agent's performance increased to approximately 7, which is a very significant improvement.

7 USING THE DESIGN (2): A FUZZY LOGIC AGENT

Fuzzy logic (FL) is a theory that is applicable to the problem of representing decision situations where complexity and uncertainty are present (18, 37). Fuzzy rule representation and extraction, learning and inference have been discussed in various settings, also applicable to gaming situations (3, 6, 7, 23, 28, 32, 36). Thus, there exists a frame of reference for the application of fuzzy logic to decision making in simulation game models.

In games, fuzzy logic may represent uncertainty of payoffs during play, as well as of the goals of players and of game solution aspects (3, 7, 23, 28, 36). Here we describe an application of the design in Chapter 4 for making a decision agent playing the Red side of Operation Lucid.

7.1 The Agent Model

The CSP/FL design follows the general agent model described in Chapter 4 and Figure 4.2 above, where three modules co-operate in sequence as an agent model. The FL part consists of modules one and three, while module two is the CSP part.

The FL part of the agent model follows a basic design similar to an agent described for the resource allocation game of Campaign (5). In both of these games the fuzzy logic part uses the decision processes and variables of a military C2 headquarters as a model for the basic design. The application of this same basic design model in the two different cases provides a design method that can be used when designing such agents for automatic decision making in other simulation models.

As explained in Chapter 5, the CSP part of the agent is used to generate a reduced set of candidate actions (moves) from the very large set of possible moves, based on constraints generated from the first FL agent module. The candidates in this set are then evaluated in the second FL module to give a ranking of the moves.

7.1.1 Design Principles

As stated, in order to have a general design for a decision agent applicable to a gaming situation, the basic decision processes and concepts of a military headquarters will be used as a model.

In a military decision context a headquarters staff will use functional knowledge based upon situational information to arrive at a chosen plan. A final decision is made by the staff evaluating different possible courses of action (COAs), based upon these higher-level planning directives. These planning and COA processes are adapted as a modelling basis in the design of a Fuzzy Logic Decision Agent.

As decision inputs to the staff processes, in general force “strength” as well as force “concentration” and “time” are used as basic variables of the decision process. The general design for the Fuzzy Logic Decision Agent using this basic model is shown in Figure 7.1.

An intermediate game state is fed into to the PLAN fuzzy rule base with *strength* and *time* as fuzzy input variables. The output from PLAN is the desired own *strength* values as constraint parameters for the left and right halves of the game board (expressed as two ALA areas overlapping in the middle column). For the Red FL Decision Agent a balance of own and opponent *strength* in these halves is the main PLAN objective. From these PLAN constraints the CSP module generates a set of moves, which are then evaluated in the COA fuzzy rule base of the FL agent. This evaluation is based upon the agent’s own *force concentration* characteristics of a move compared to the opponent threat in each of the two Red defensive lines (the two northernmost rows), giving a resulting priority measure for the CSP-generated moves. Normalising the priority measures to sum to 1 over the move set produces the final output, an priority ranking of the move set for the current state. Finally, the Fuzzy Logic Decision Agent selects the move with highest priority as the its move. In Operation Opaque, randomised actions may be required; consequently, this normalised priority measure can be interpreted as an estimated probability distribution function to be used for drawing a move at random.

As seen from the figure, this design follows the general agent model described in Chapter 4, and may be seen as a hybrid combination of two techniques, FL and CSP. This method for constructing such agents may also be useful in other simulation game models where graphs constrain the moves of the game. In such a context an ALA constraint set may be formulated as described for this game with force balancing constraints generated according to objectives of the agent.

In the following sections a detailed implementation for a Red FL Decision Agent is described with variables, membership functions and rulebases defined.

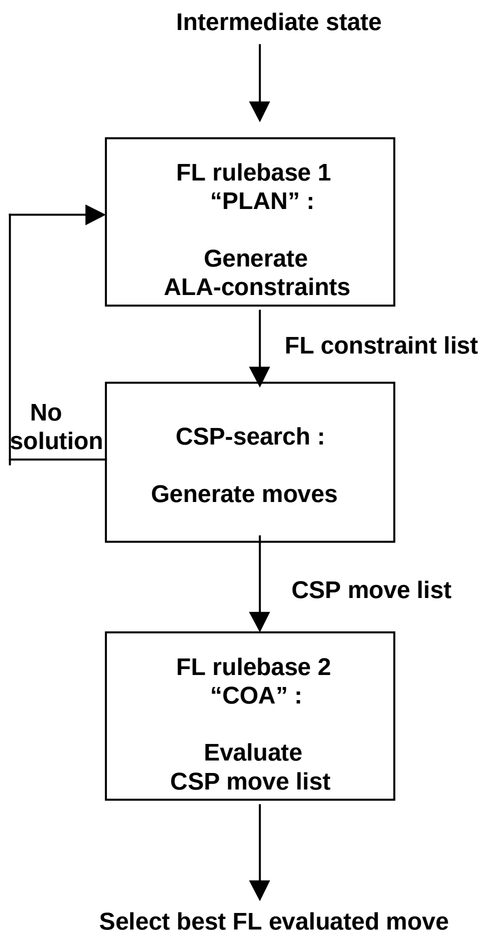


Figure 7.1 Design of the fuzzy logic agent

To simplify the implementation we restrict ourselves to fuzzy sets with triangular membership functions only, even though more complex functional forms are well known (18, 37). Furthermore, we partition the variable domains into triangular partitions (TP) (6, 32), where the maxima (the “midpoints”) of the membership functions divide the domains into a sequence of intervals, such that only two fuzzy sets are active at the same time. Thus, only two rules are active for each variable in the fuzzy rule bases, giving an easy and fast inference process.

7.1.2 Implementation

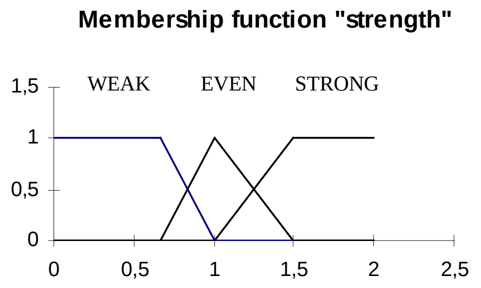
A goal of the design is to use as few fuzzy model variables and sets as possible to reduce the size of the rule bases while still achieving acceptable performance. Therefore, only the two variables *strength* and *time* are used as input to the PLAN rule base. In order to be able to use the same rule bases regardless of game board size, the number of rounds, the number of pieces and the *strength* and *time* variables are reduced to the following two input fuzzy ratio variables s and t:

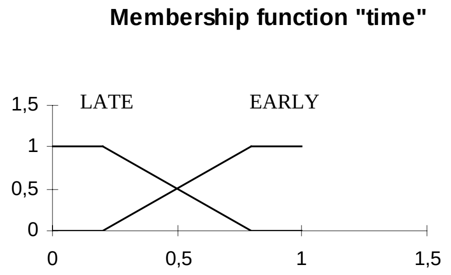
$$s_b = b_l / b_r \ ; \ s_r = r_l / r_r \ ; \ s = s_b / s_r$$
$$t = (n / n_0) - \tau \ ; \ t \in [0,1].$$

Here s_b, s_r are opponent and own force ratios between the left and right ALA half areas of the game board, s is the «strength» expressed as a left/right-side force balance ratio, n_0 is the initial number of rounds, and n is the number of rounds left. Since the game is stochastic with player turns drawn randomly at each round, the time variable τ is an urgency variable as a function of the number of turns won for the opponent: $\tau = w / T$ where w is the number of opponent turns won and T (= 5) is the minimum time taken to traverse the board. This will regulate the fuzzy time ratio variable t according to possible early threats due to opponent luck.

The *strength* variable expresses the relative opponent left and right force-ratios, which gives a balancing signal to the Red player. The input variable *time* is also a ratio variable, and together they are the input to the PLAN rule base, which is valid for any number of pieces and game duration. The PLAN rule base will give a balancing correction as output, to direct the movement of the Red pieces in a generally favourable direction. In this way the role of the PLAN rule base is to act as a high-level decision-making part of the agent, while the COA base evaluates the move set from PLAN in more detail.

The initial FL membership functions for the input variables are shown in Figure 7.2. Three fuzzy sets, *weak*, *even* and *strong*, are defined for the *strength* variable, while only two sets *early* and *late* are initially considered for the *time* variable.





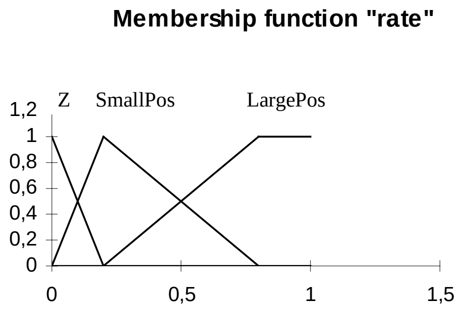


Figure 7.2 *Membership functions for the fuzzy input variables* strength *and* time, *and for the output variable moving* rate *(shown for the positive direction left only, symmetric for the negative direction)*

Both these input variables for the Red player contribute to the moving rate and direction output e from PLAN: early on the Red player may react slower than later, when it is imperative to reach a balancing position fast in order to prevent Blue from moving to the Goal node. Equal balance for Red and Blue gives no signal and Red remains as before. This leads to three magnitudes for the moving rate variable: *zero*, *small* and *large*. Directional information is given as the suffix *pos* (left) or *neg* (right) as shown in Figure 7.2.

The PLAN rule base is shown in Table 7.1, where the FL input variables s and t and the moving rate output variable e are shown. This rule base expresses an error correction logic for balancing Red and Blue forces in each ALA half of the board.

Strength \ Time	**Early**	**Late**
Weak	SmallNeg	LargeNeg
Even	Zero	Zero
Strong	SmallPos	LargePos

Table7.1 *The PLAN FL rulebase with* strength *and* time *as input, and with moving rate as output (left direction is positive).*

Defuzzying the output variable e gives new balanced Red force objectives for the left and right side ALA areas :

$$
\begin{aligned}
&rcorr = e \cdot \left(\frac{s}{s+1} r_r(n) - \frac{1}{s+1} r_l(n) \right) \\
&r_l(n-1) = r_l(n) + rcorr \quad ; \quad r_l(n-1) \in [0, r(n)] \\
&r_r(n-1) = r_r(n) - rcorr \quad ; \quad r_r(n-1) \in [0, r(n)],
\end{aligned}
$$

where $r_l(n), r_r(n)$ are the Red number of pieces in the left and right ALA halves of the game board, $r(n)$ is Red total number of pieces and n is number of rounds left. The correction term in these equations seeks to match the Red and the opponent left/right balance ratios as much as possible.

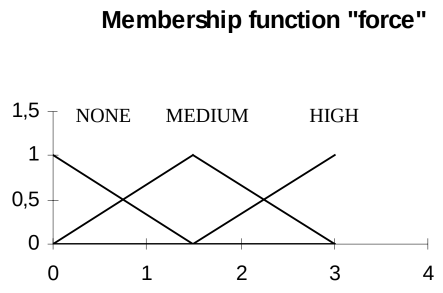
Based upon the new corrected Red left/right balance ratio, the following ALA constraints are given to the CSP module: (LeftArea, $r_l(n-1)$), (RightArea, $r_r(n-1)$). It may happen that the CSP-module gives no moves satisfying these constraints. The FL module then loops a number of times, subtracting one piece from each of these constraints until a valid move set results or a maximum number of iterations is reached (in which case Red does not move). In addition to these ALA constraints some general, parametric constraints are also given to reduce the move set, such as:

- maximum number of moves $\leq$ MaxMoves
- maximum number of pieces moving left/right/south $\leq$ MaxPieces
- maximum number of occupied nodes $\leq$ MaxNodes

– where the right-hand sides are integer parameters (a separate MaxPieces parameter is used for each of the three directions). These constraints together define the PLAN rule base resulting in ALA constraints that the CSP module uses to generate a move set, from which a best move is selected in the following COA part of the agent.

The COA part considers the threat for every move in each of the upper two rows of the game board, since these are considered the main defensive lines for Red. Each row is a separate new ALA area (Row4 and Row5). COA is divided in two separate rule bases where the first, COA1, evaluates the local threat in a single node of each area, and the second rule base, COA2, determines the final ranking based upon the summed threat in each of the two rows. These rule bases allow the Red FL agent to consider the simultaneous deployment priority in two rows and thus gain a possible advantage by cutting Blue's supply line.

The COA1 rule base takes Red and Blue forces $r(n)$, $b(n)$ in a single node as input and gives a local threat (or opponent advantage) estimate $p(n)$ for this node as output. Three fuzzy sets *none*, *medium* and *high* are defined for both of the input force variables, while the threat output in the interval [0,1] is divided into 9 sets P_k, $k = 1,\dots,9.$ The membership functions for these variables are shown in Figure 7.3, where only one of the two force input functions is shown.



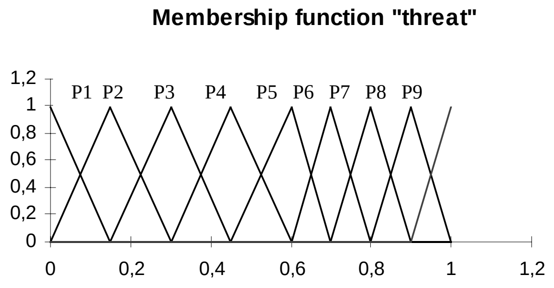


Figure 7.3 *Membership functions for the two fuzzy input variables* force *(the same for both Red and the opponent), and for the output variable* threat *from the COA1 rule base.*

The COA1 rule base is shown in Table 7.2, where the FL input *force* variables and the *threat* output variable P_k are shown. This rule base expresses threat estimate logic for a node relating the input of Red and opponent forces to the threat output variable P_k for the node. Defuzzifying the output P_k from COA1 and summing over the nodes in each of the two ALA rows Row4 and Row5 gives a total threat estimate for each of these rows after normalising by the number of nodes in each row.

Force 1 \ Force 2	**None**	**Medium**	**High**
None	P_6	P_7	P_9
Medium	P_2	P_4	P_8
High	P_1	P_3	P_5

Table 7.2 *The COA1 FL rulebase with forces as input, and with estimated threat in a node as output.*

The COA2 rule base is similar to the one shown for COA1 in Table 7.2, except that the two FL input variables are the Row4 and Row5 summed *threat*s, and the output variable is the resulting ranking or priority Pr in the interval (0,1). This rule base gives the final priority of a move by considering the two upper defensive rows of the Red agent simultaneously. This allows for weighting between both defensive and more offensive types of Red agents to test their resulting performance.

Defuzzifying the output from COA2 for each move in the generated move set and normalising the priorities to sum to one gives the final ranking as basis for selecting the highest value move for the Red agent to play next.

This completes the total design and implementation of the Red FL/CSP agent which consists of seven membership functions used in three rule bases with a total of 24 rules. The membership values shown above are the initial values, and the Red agent parameters may be tuned by optimising its performance in a separate training session.

7.2 Training the Agent

To train a Red agent R, an optimisation procedure against a selected opponent agent B from a fixed set S of opponents is considered. The selection will be based upon the expected outcome $-E(B,R)$ – see Section 3.1 – against the opponent that is considered best (by judgement or tournament results from a number of games):

$$Sinf(R;S) = -\inf_{B \in S}\{E(B,R)\}$$

This will give an approximation to a global equity result for the Red agent, but several problems exist that make such a procedure challenging. First, no global best opponent is known, and thus only an approximate estimate of global equity is measured. Secondly, the game is stochastic, which gives a noisy performance measure that may confuse an optimising training algorithm. Variance reduction methods from a number of games must be used for each expected outcome estimate as described in Chapter 3.3. Thus, compared to a previously described agent training for the resource allocation game Campaign (5), this training process will be much more demanding.

7.2.1 Training Method

As an estimated best opponent, only one agent B_o that initially randomly chooses one of the board columns and then moves with maximal thrust to the Goal along this column – that is, the OneAxisBlue agent of Section 3.2.1 – is selected. This is considered a representative challenge for a Red agent, since lateral movement of Red is necessary and all columns of the game board may be chosen. Force concentration ability is also tested, since the opponent uses maximum thrust (with three pieces) in front whenever possible, which must be matched by Red to achieve a good performance. One weakness of the agent B_o is its lack of a supply line in later rounds of a game.

As described for Campaign (5), the training process is using a Nelder-Mead optimisation algorithm similar to the Matlab function *fmin* or the NAG routine *E04CCF* (22) to tune the Red agent fuzzy parameter values. These membership function parameters are modified based upon minimisation of the opposing number of pieces in the Goal node at the end of a game. As described, this is a noisy output measure, and an expected adjusted value based on variance reduction from a series of games with the same input parameter set is used as optimisation performance measure to try to filter out the noise. However, a complete filtering is not possible, and the optimisation process may therefore be confused by the noise in the output. The number of games chosen for each adjusted mean value is thus a very important parameter of the training process.

Of course, if the training process is so fast that time spent in training is negligible, then there is no problem. But the considerations above is precisely translating into total computing time for this training process. Some numbers may illustrate this. One game takes roughly 10 seconds on a Pentium II 450 MHz machine with ILOG CSP software (17). To be able to discriminate between outputs differing by 0.1 with 95% confidence, about 2500 games must be run per optimisation iteration. This gives roughly 7 hours per iteration of the optimisation process, or 3–4 trial points per day. Since a training process may use more than 100 iterations to arrive at a near optimum, one sees that noise is a very complicating factor of this game, and in general of training problems this noisy. Furthermore, since we do not know the general properties of the outcome function, global optimisation with random starts should be used when tuning the Red agent. Thus, still another factor must be considered when training for this game.

The following procedure considers these factors when using optimisation to train the Red CSP/FL agent:

(a) Choose a number of random starting values for FL parameters in a first-pass global search phase with 40 games per iteration. Select the best of these for the next phase (b).

(b) Start with the best of (a) in a second-pass optimisation, starting with 400 games per iteration for 100 initial iterations to determine parameter gradients. Continue with 4000 games per iteration for a number of optimising iterations with improved outcome accuracy until no improvement or until a maximum number of iterations is reached.

(c) Compare the result from (b) with a third-pass optimisation as in (b), but now starting with the best human initially determined FL parameters based upon judgement and experience from game plays.

(d) Select the best of (b) and (c) as the current Red CSP/FL tuned agent.

The results of this procedure are described for the Red CSP/FL agent based upon seven random starts for the global first-pass search phase. This phase would benefit greatly by parallel processing, but this has not been used here. The step (b) requires more sequential processing, even though parallel optimisation software certainly would be of benefit to the overall optimisation process.

7.2.2 Experimental Results

Training the Red CSP/FL agent using the procedure above gave the optimisation results shown in Figure 7.4 for step (a). It is seen that even though the results are noisy, one of the first-phase random start runs is a clear best, and this is selected for use in steps (b) and (c) for further optimisation.

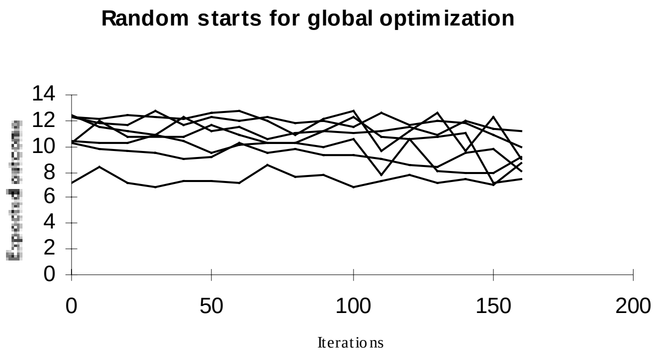


Figure 7.4 Global optimisation with seven random starts for the FL agent using 40 games per iteration (a total of 47600 games are used)

It is also clear that the noisy output makes it difficult for the optimising algorithm to improve when only 40 games per iteration are used.

Continued optimisation in step (b) with the best global run from (a) is shown in Figure 7.5. As seen from the figure no significant improvement is found after more than 150 iterations. It may be that only a local minimum has been found, indicating that the random global search start set is too restricted. In addition, the gradient information after ca 100 iterations may also be noisy due to too few games pr iteration (400), giving errors in the search directions for the rest of the optimising iterations.

The results of step (c) starting with parameters based upon human judgement and experience are also shown in Figure 7.5. It is seen that after an initial gradient search the optimisation gives only a slight improvement of the initial FL player. The results demonstrate fully the difficulties of noisy optimisation. Still, the performance of the trained Red FL agent from steps (a), (b) and (c) is such that only ca 40 % of the opponent force (that is, six pieces) on average reaches its goal node.

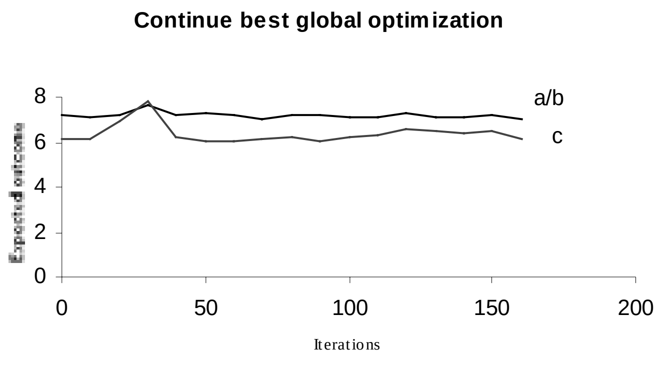


Figure 7.5 Continued optimisation of the best global FL agent using 400 games (100 iterations) and then 4000 games pr iteration (for a total of 140000 games, graph a/b). Also shown are optimisation iterations with the best human-found initial FL parameters using 1000 games pr iteration (for a total of 200000 games, graph c).

7.3 Discussion

A CSP/FL agent following the general design model of Chapter 4, where three modules co-operate in sequence as an agent model, has been described. The FL part of the agent model follows a basic design similar to an agent described for another resource allocation game (5). In both of these games, the fuzzy logic part is using the decision processes and variables of a military C2 headquarters as a model for the basic design. Thus, applying this same basic design model in these two different cases provides a design method that may be considered useful (as a possibility of using the same ideas) when designing such agents for automatic decision making in other, more general simulation game models.

It is simple to construct a set of initial FL rules and membership functions based on this design, and human judgement and experience is also integrated rather easily to form a best human initial agent model as starting point for further training and tuning. However, such training is not easy mainly due to the complexity and stochastic (noisy) outcome of the game. We chose initially to apply a global search from a set of random starting points, with further optimisation based upon a best choice. The optimisation process is shown to be susceptible to the outcome noise, and this translates to high computing time requirements. Given sufficient computing power an optimisation may be possible, but finding a global optimum agent is still a very demanding task.

Even though the training process has proved difficult, the trained Red CSP/FL agent is still having a satisfactory performance, denying on the average 60 % of the opponent forces from reaching their goal.

The resulting CSP/FL agent design is compact and generally applicable to any board size, force level and game duration since ratio variables are used in the design. This is an obvious advantage when considering possible variations of the particular game version described here. Furthermore, the rule bases are small with altogether only 24 rules distributed in three rule bases, and with a total of seven membership functions.

The results are shown for a stochastic game with perfect information. If we insert a possible CSP- or a separate FL- based intelligence module, most of the basic FL agent design may be utilized also for a game with imperfect information, where the move rank priority output function of the agent may be interpreted as a probability function for random draws between moves. This may be a topic for further research, together with possible improvements of the training optimisation process. Finally, as further research an opponent agent may also be constructed, using a similar design as a “mirror” image of the Red agent with an evading strategy, giving opposite error correction signals in the FL PLAN rule base. Thus, similar modules as described may be used for both a Red and a Blue agent with differences only in the final fuzzy tuned parameters.

8 DEALING WITH IMPERFECT INFORMATION

Although there is nothing in the general design in Chapter 4.2 that presupposes perfect information, both of our actual applications of the design are limited to Operation Lucid. The neural network agent evaluates game states based on the location of both its own and its opponent’s pieces, while the fuzzy logic agent uses the opponent’s known force distribution to decide on a desirable positioning of its own pieces. What these applications have in common is then that a decision is taken based on the current state only; this is a valid approach as the game possesses the Markov property.

The imperfect-information property of Operation Opaque, on the other hand, implies that the whole history up to the present state (as seen by one of the agents) has to be taken into account. This chapter describes a way of using the ALA language (Section 5.3.2) and constraint satisfaction programming in the design of an intelligence module[3] for agents playing

[3] The term “intelligence” is here used in its military sense, that is, as an approximate synonym to “information processing”, not to “intellectual capacity”.

Operation Opaque. This module has not at the time of writing progressed past the prototype stage, and further implementation, testing and use is a topic for future research.

8.1 An Intelligence Module for Operation Opaque

In a given observed state of Operation Opaque, the whereabouts of opponent pieces are in general not known. This does not necessarily mean, however, that these pieces could be anywhere apart from the observed nodes. Observations of opponent pieces in earlier stages of the game may be a source of information, as the distance such a piece may have moved since it was observed is constrained by the number of turns the opponent has won. In addition, the initial set-up (Figure 2.1) is known by both sides. This indicates that the question of where the enemy pieces are located can be formulated as a constraint satisfaction problem.

When an agent is to use its information from earlier stages in making a decision, the most relevant basis for this decision is where the opponent's pieces may be at the moment. This information can also be expressed another way, namely as which nodes and sets of nodes may hold a certain number of enemy pieces. The ALA syntax is well suited for expressing queries using the latter formulation, the semantics is then that an ALA constraint set

$$\{(M_1, n_1), (M_2, n_2), \ldots, (M_l, n_l)\}$$

is a query to the intelligence module about whether the opponent may have at least n_1 pieces in the node set $M_1 = \{node_1, node_2, \ldots, node_k\}$ *and* at least n_2 pieces in node set M_2, and so on.

Since observed enemy pieces are indistinguishable, the problem of identity needs to be implicitly addressed by the intelligence module. Consequently, the CSP formulation of the problem includes two kinds of variables. Firstly, there is one set of variables for each opponent piece, where the possible values represent the location of the piece on the board. Secondly, there is one set for each node, where the value represents the number of opponent pieces in that node. (The values of the latter type of variables are calculated from those of the former type.) The set of variables for a piece or a node contains one variable for each turn the opponent has won during the game.

The constraints limiting the values of the variables are also of two types. The first set of constraints is generated from the observations the agent has made about the enemy during the game, including negative observations (nodes without opponent pieces). The known starting position is treated as an observation of this type. The other set of constraints represents the rules of the game, and constrain the possible values for the position of a piece at time *t*, based on its position at time $t-1$ and on any combat that has taken place.

When an ALA query about enemy pieces is passed to the intelligence module, a CSP search tries to find an assignment of the pieces at the current time that does not violate the constraints. If such an assignment is found, the module stops searching and returns a positive reply. If no valid assignment is found, a negative answer is returned.

The full CSP search may be computationally prohibitive. A way of limiting the complexity of the problem at the expense of certainty in the answer is to let the module guess at the probable identity of observed pieces.

The problem remains how to use this type of information in actual agents – this is a topic for future research. If we were able not just to ask the intelligence module whether a given enemy distribution is *possible*, but also to ask for the *most likely* distribution (under some set of assumptions), the agents developed for Operation Lucid could be used with the modification that the most likely distribution is treated as the observed one. This is an even more complex problem than the one considered above, and has not been treated.

Another possible application of the intelligence module in its current design is as a planning tool for Operation agents. In this context, the current state of the agent's own pieces can be entered as an observation, and questions about future possibilities posed to the module. An example query of this type could be "is it possible for me to reach the goal with five pieces before the game ends, given that I win at least half of the turns?"

9 FUTURE RESEARCH

Although we have made considerable progress in our research on decision-making software agents for Operation, there are still plenty of unresolved problems and unexplored territory left to investigate[4]. Here we mention some promising directions for further studies.

A major difficulty for learning algorithms in Operation is the presence of noisy game outcomes due to random draws. The most successful learning procedure used was reinforcement learning by the TD(λ) algorithm, which handles this kind of noise reasonably well. A possible further improvement of this algorithm may be to adopt the principles of variance reduction used in Section 3.3 to give better feedback signals, adjusted for the outcome of the random draws, to the learning algorithm.

The agents considered so far have all been on the "top level", that is, they have been able to directly move their pieces as they like (subject, of course, to the rules of the games). When modelling combat, it seems reasonable that the pieces themselves should also have their own behaviour, since they represent military units. In this context, each side of the game would consist of a collection of agents, each observing, moving in and acting on its environment. A problem that arises when modelling the situation in this way is how to assign credit or blame for good or bad results to the individual agents. A possible starting point for methods and references on this subject is (29).

In addition to the fact that decision making in combat models is distributed, it is also usually hierarchical – there is a hierarchy of decision makers operating on different levels. So far we have addressed only the top level in our work, while the individual units mentioned in the previous paragraph represent the bottom level. The hierarchy may consist a number of such levels, each of which communicates with the levels above and below it. Here, different

[4] The research on Operation will be continued in FFI project 806, "Machine Learning in Simulation".

learning methods may be suitable for learning at the different levels. A useful starting point for research may be (31).

The learning methods addressed in this report depends on the game outcomes only for their feedback. As we have seen, this feedback is noisy, a fact that complicates learning. Another approach to learning good game-play may be to use machine learning techniques for imitating expert behaviour. The work reported in (9) on the game of go indicates that such imitation learning may be useful, at least for certain aspects of the game. A disadvantage of this “data mining” approach is that it presupposes a large number of games played by experts – for Operation, such experts must clearly be human.

10 CONCLUSION

We have presented the work that was done on the games Operation Lucid and Operation Opaque during the FFI project “Synthetic Decision Making”. The main conclusions from our studies can be summarised as follows.

The games of Operation have proved successful as working environment for studying decision making in combat models (Chapters 1–2).
In a “live” combat model, the complexity that a decision-maker needs to handle is typically very large. Some of this complexity is due to details specific to the model itself, and some is common to combat models in general. When studying automatic decision making, the latter type is of special interest, since we seek methods that are broadly applicable. The games of Operation succeed in providing an environment with very high complexity, while still possessing a fairly simple set of well-defined rules. Also, the design of the games has proved robust, in that no changes have been necessary during our work.

Many techniques that have traditionally been used in AI research on games are not useful in Operation (Chapter 1; Section 4.1).
In previous research on artificial intelligence for game playing, focus has mainly been placed on games of perfect information with low complexity compared to Operation. In these other games, brute-force searches looking several moves into the future have been possible, at least when effective pruning techniques such as alpha-beta search have been used. The information imperfection of Operation Opaque renders such search techniques meaningless. For the perfect-information Operation Lucid they are at least well-defined, but the complexity of the game renders them useless in practice.

When an agent’s game-play is to be evaluated, the chosen evaluation criterion is of great importance (Sections 3.1–2).
It seems intuitive that there should exist an evaluation criterion ranking strategies in a consistent way, that is, such that higher-ranked strategies on average will beat lower-ranked ones. This intuition does not hold, as there may well be strategies that beat each other in circle. For mathematically sound criteria we need to turn to game theory. Since these criteria are generally not possible to employ in practice, we must resort to evaluating agents by playing

them against other agents. Still, the properties of the “correct” evaluation criteria should be kept in mind when using this mode of evaluation.

Our variance reduction techniques have been valuable in reducing the number of games needed for satisfactorily accurate estimates of expected outcomes (Section 3.3).
The random draws in Operation have great influence on the game results. In order to gain statistically significant estimates of the expected outcome between two strategies, it is necessary to play a number of games. Since each game takes some time to complete, methods that reduce the necessary number of games for the required accuracy are valuable. By applying domain knowledge and the method of variance reduction by control variates, we were able to reduce the required number of games with factors of more than ten for our non-trivial reference agents.

Humans and computers generally use different approaches to selecting moves. Both of these approaches have serious weaknesses when applied to games like Operation (Section 4.1).
When playing games, humans mainly plan forward in time and construct moves, while computers generate all possible moves and evaluate them to find the best one. The constructive approach, when applied to computers, has the disadvantage that the complexity of the domain makes it difficult to define construction rules with sufficient power of expression to achieve high-performance game-play. The evaluation approach, on the other hand, presupposes that all legal moves can be enumerated in a reasonably short time; this assumption does not hold for Operation.

Our CSP-based agent design has been successful in combining the strengths of the two basic playing approaches (Chapters 4).
A strength of the constructive approach is that it reduces complexity; a strength of the evaluation approach is that it divides its task into two clearly-defined subtasks: generating the legal moves and choosing between them. Our CSP-based agent design combines modularity and reduction of complexity by using constraints to implicitly describe a *limited set* of candidate moves to be evaluated.

Our applications using neural nets and fuzzy logic illustrate how the design is general enough to be useful in agents working in quite different ways. (Chapters 6–7).
Both the neural net agents of Chapter 6 and the fuzzy logic agent of Chapter 7 use our CSP-based agent design. In the basic neural net agents, a minimum of domain knowledge was built in. The fuzzy logic agent, on the other hand, was designed to resemble a military staff headquarters – here, knowledge of the domain was clearly important.

Knowledge-intensive methods can be useful for quickly developing agents of reasonable quality (Sections 3.2.1 and 7.2).
The very simple Blue agent OneAxisBlue, built entirely on the principle of force concentration, achieved results comparable to the basic Blue neural net. The fuzzy logic agent where the fuzzy parameters were set by a human also achieved decent results.

Noisy results hinder optimisation of parameters directly from the game outcomes. Reinforcement learning algorithms like TD(λ) may overcome this in the perfect-information case (Sections 7.2 and 6.3–4).
The training of the fuzzy logic agent by optimising its parameters from game outcomes using standard gradient-based methods were largely unsuccessful, due to the high amount of noise in the results. The basic Blue neural net, on the other hand, used the reinforcement learning algorithm TD(λ) for training. Although its game-play did have some weaknesses, the progress made during training was encouraging. Neural nets work well with reinforcement learning; this conclusion is also supported by other work done in the project (10).

Adding some domain knowledge to an agent otherwise lacking in prior knowledge may significantly improve performance (Sections 6.4–5).
The neural net learning algorithm of Chapter 6 was not able to identify valid supply lines and learn that this was an advantageous feature, because very few of the states it would encounter – by chance or on purpose – would have one. The Blue neural net significantly improved when this knowledge was added, and it was told to keep a supply line for the early and middle parts of the game.

The ALA language, originally made for defining local constraints, appears to be a versatile tool (Section 5.3.2 and Chapter 8).
Although the area language (ALA) protocol was designed to express constraints describing the minimum number of pieces to be placed in sets of nodes, preliminary work on an intelligence module for Operation Opaque indicates that it may be applied in other contexts as well.

We have only scratched the surface yet (Chapter 9).
Although considerable work has been done, there is still much left to explore. Possible directions for future research include more robust reinforcement learning algorithms, multi-agent architectures, hierarchical learning and imitation of expert behaviour.

References

(1) Bakken B T, Dahl F A (2000): An empirical study of decision making and learning in a complex two-person zero-sum game with imperfect information, FFI/NOTAT-2000/03919, Norwegian Defence Research Establishment.

(2) Bergli J (1998): ExperTalk FFI-versjon 1.0 – Teknisk dokumentasjon og brukerveiledning (ExperTalk FFI-version 1.0 – Technical documentation and user manual), FFI/NOTAT-98/04152, Norwegian Defence Research Establishment (in Norwegian).

(3) Borges P S S, Pacheco R C S, Barcia R M, Khator S K (1997): A fuzzy approach to the prisoner's dilemma, *Biosystems* **41**, 127–137.

(4) Brailsford S C, Potts C N, Smith B M (1999), Constraint satisfaction problems: Algorithms and applications, *Eur J Op Res* **119**, 557–581.

(5) Braathen S (1999): A Fuzzy Logic Decision Agent for automatic decision making in a zero-sum game, FFI/NOTAT-99/00877, Norwegian Defence Research Establishment.

(6) Butnario D, Klement E P (1993): Triangular Norm-Based Measures and Games with Fuzzy Coalitions, Kluwer, Dordrecht, The Netherlands.

(7) Campos L, Gonzales A, Vila M-A (1992): On the use of the ranking function approach to solve fuzzy matrix games in a direct way, *Fuzzy Sets and Systems* **49**, 193–203.

(8) Christodoulou N, Wallace M, Kuchenhoff V (1994), Constraint logic programming and its application to fleet scheduling, *Information and Decision Technologies* **19**, 135–144.

(9) Dahl F A (1999): Honte, a go-playing program using neural nets. In: *Proceedings of the ICML-99 Workshop on Machine Learning in Game Playing* (Eds J Fürnkranz, M Kubat), Jožef Stefan Institute, Ljubljana, Slovenia.

(10) Dahl F A (2000): Artificial intelligence and human behaviour in simulations – Final report from FFI-project 722 "Synthetic Decision Making", FFI/RAPPORT-2000/04395, Norwegian Defence Research Establishment.

(11) Dahl F A, Halck O M (1998), Three games designed for the study of human and automated decision making. Definitions and properties of the games Campaign, Operation Lucid and Operation Opaque. FFI/RAPPORT-98/02799, Norwegian Defence Research Establishment.

(12) Freuder E C, Mackworth A (1994), *Constraint-Based Reasoning*, MIT Press, Cambridge, Mass.

(13) Freuder E C, Wallace R J (1992), Partial constraint satisfaction, *Artificial Intelligence* **58**, 21–70.

(14) Halck O M, Dahl F A (1999): On classification of games and evaluation of players – with some sweeping generalizations about the literature – A paper presented at the ICML-99 Workshop on Machine Learning in Game Playing, FFI/NOTAT-99/04875, Norwegian Defence Research Establishment.

(15) Haralick R, Elliott G (1980), Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14**, 263–313.

(16) Hassoun M H (1995), *Fundamentals of Artificial Neural Networks*, MIT Press, Cambridge, Mass.

(17) ILOG (1999), *Ilog Solver 4.4, User´s manual.*

(18) Kosko B (1997): Fuzzy Engineering, Prentice-Hall, Upper Saddle River, NJ.

(19) Kumar V (1992), *Algorithms for Constraint Satisfaction Problems: A Survey*, Department of Computer Sciences, University of Minnesota.

(20) Lajos G (1995), Complete university modular timetabling using constraint logic programming. In: *First International Conference on Practice and Theory of Automated Timetabling* (Eds E Burke, P Ross), *Lecture Notes in Computer Science vol. 1153,* Springer, Berlin, 146–161.

(21) Luce R D, Raiffa H (1957), *Games and Decisions*, Wiley, New York.

(22) Mathworks (1995): MATLAB User's Guide, Mathworks, Natick, Mass.

(23) Mohanty, B K (1994): A procedure for measuring uncertainties due to lack of information in a fuzzy game theory problem, *Int J Systems Sci* **25**, 12, 2309–2317.

(24) Nelson B L (1990): Control variate remedies, *Op Res* **38**, 6, 974–992.

(25) Nuijten W P M, Aarts E H L (1996), A computational study of constraint satisfaction for multiple capacitated job shop scheduling, *Eur J Op Res* **90**, 269–284.

(26) Russell S J, Norvig P (1995), *Artificial Intelligence. A Modern Approach*, Prentice Hall, Upper Saddle River, NJ.

(27) Sabin D, Freuder E C (1994), Contradicting conventional wisdom in constraint satisfaction. In: *Proceedings of European Conference on Artificial Intelligence (ECAI-94)* (Ed A G Cohn), Wiley, Chichester, UK, 125–129.

(28) Sakawa M, Nishizaki I (1994): Max-min solutions for fuzzy multiobjective matrix games, *Fuzzy Sets and Systems* **67**, 53–69.

(29) Schneider J, Wong W-K, Moore A, Riedmiller M (1999): Distributed value functions. In: *Machine Learning. Proceedings of the 16th International Conference (ICML '99)* (Eds I Bratko, S Džeroski), Morgan Kaufmann, San Francisco, California, 371–378.

(30) Sendstad O J, Halck O M, Dahl F A (2000): A constraint-based agent design for playing a highly complex game – A paper presented at the PACLP 2000 conference, FFI/NOTAT-2000/01091, Norwegian Defence Research Establishment.

(31) Stone P, Veloso M (2000): Layered learning. In: *Machine Learning: ECML 2000. Proceedings of the 11th European Conference on Machine Learning. Lecture Notes in Computer Science vol. 1810* (Eds R López de Mántaras, E Plaza), Springer-Verlag, Berlin–Heidelberg–New York, 369–381.

(32) Sudkamp T, Hammell R J II (1994): Interpolation, completion, and learning fuzzy rules, *IEEE Trans Syst, Man, Cybern* **24**, 2, 332–342.

(33) Sutton R S (1988), Learning to predict by the method of temporal differences, *Machine Learning* **3**, 9–44.

(34) Tesauro G J (1992), Practical issues in temporal difference learning, *Machine Learning* **8**, 257–277.

(35) Tsang E (1993), *Foundations of Constraint Satisfaction*, Academic Press, London, UK.

(36) Zhang Y-Q, Kandel A (1997): An efficient hybrid direct-vague fuzzy moves system using fuzzy-rules-based precise rules, *Expert Systems With Applications* **13**, 3, 179–189.

(37) Zimmermann H J (1996): Fuzzy Set Theory – and Its Applications, third edition, Kluwer, Dordrecht, The Netherlands.

DISTRIBUTION LIST

FFISYS **Dato:** 15 november 2000

RAPPORTTYPE (KRYSS AV) [✓ X] RAPP ☐ NOTAT ☐ RR	RAPPORT NR. 2000/04403	REFERANSE FFISYS/722/161.3	RAPPORTENS DATO 15 november 2000
RAPPORTENS BESKYTTELSESGRAD Unclassified		ANTALL EKS UTSTEDT 37	ANTALL SIDER 49
RAPPORTENS TITTEL DECISION MAKING IN SIMPLIFIED LAND COMBAT MODELS - On design and implementation of software modules playing the games of Operation Lucid and Operation Opaque		FORFATTER(E) HALCK Ole Martin, SENDSTAD Ole Jakob, BRAATHEN Sverre, DAHL Fredrik A	
FORDELING GODKJENT AV FORSKNINGSSJEF:		FORDELING GODKJENT AV AVDELINGSSJEF:	

EKSTERN FORDELING

ANTALL	EKS NR	TIL
1		Prof H R Jervell Inst for lingvistiske fag Universitetet i Oslo Pb 1102 Blindern 0317 Oslo
1		Prof O Hallingstad UniK Pb 70 2027 Kjeller
1		Prof II R A Fjellheim UniK Pb 70 2027 Kjeller
1		Prof T Lensberg Inst for samfunnsøkonomi Norges handelshøgskole Hellev 30 5035 Bergen
1		F aman J Frihagen Krigsskolen Pb 42 Linderud 0517 Oslo
1		KK J K Nyhus Forsvarets stabsskole Oslo mil/Akershus 0015 Oslo
		www.ffi.no

INTERN FORDELING

ANTALL	EKS NR	TIL
14		FFI-Bibl
1		Adm direktør/stabssjef
1		FFIE
4		FFISYS
1		FFIBM
1		R H Solstrand, FFISYS
1		B E Bakken, FFISYS
1		J E Torp, FFISYS
1		F A Dahl, FFISYS
1		B T Bakken, FFISYS
1		S Braathen, FFISYS
1		O M Halck, FFISYS
1		O J Sendstad, FFISYS
1		K A Veum, FFIE
1		H O Sundfør, FFISYS
		FFI-veven