Inducing game rules from varying quality game play

Alastair Flynn

May 7, 2020

Contents

Abstract

General Game Playing (GGP) is a framework in which an artificial intelligence program is required to play a variety of games successfully, it acts as a test bed for AI and motivator of research. The AI is given a random game description at runtime (such as checkers or tic tac toe) which it then plays. The framework includes repositories of game rules written in a logic programming language.

The Inductive General Game Playing (IGGP) problem challenges machine learning systems to learn these GGP game rules by watching the game being played. In other words, IGGP is the problem of inducing general game rules from specific game observations. Inductive Logic Programming (ILP), a subsection of ML, has shown to be a promising approach to this problem though it has been demonstrated that it is still a hard problem for ILP systems. In this regard IGGP motivates future research in ILP.

Existing work on IGGP has always assumed that the game player being observed makes random moves. This is not representative of how a human learns to play a game, to learn to play chess we watch someone who is playing to win. With random gameplay a lot situation that would normally be encountered when humans play are not present. It may be the case that some games rules do not come into effect unless the game gets to a certain state such as castling in checkers. Rules of games are designed with good play in mind so it would make sense that a good player of the game would invoke more cases of the rules.

To address this limitation, we analyse the effect of using intelligent verses random gameplay traces as well as the effect of varying the number of traces in the training set.

We use Sancho, the 2014 GGP competition winner, to generate optimal game traces for a large number of games. We then use the ILP systems, Metagol, Aleph and ILASP to induce game rules from the traces. We train and test the systems on combinations of intelligent and random data including a mixture of both. We also vary the volume of training data.

Our results show that whilst some games were learned more effectively in some of the experiments than others no overall trend was statistically significant.

The implications of this work are that varying the quality of training data as described in this paper has strong effects on the accuracy of the learned game rules however one solution does not work for all games.

Chapter 1

Introduction

General Game Playing (GGP) is a framework in which artificial intelligence programs are required to play a large number of games successfully.[21]. Traditionally game playing AI have focused on a single game[41, 23, 39, 44]. Famous AI include programs such as IBMs deep blue which is able to beat grand masters at chess but is completely unable to play checkers [23]. These traditional AI also only do part of the work. A lot of the analysis of the game is often done outside of the system[39]. A more interesting challenge is building AI that can play games without any prior knowledge. In GGP the AI are given the description of the rules of a game at runtime. Games in the framework range greatly in both number of players and complexity; from the single player Eight Puzzle to the six player Chinese Checkers, and from the relatively simple Rock Paper Scissors to Chess[20]. The progress in the field is consolidated annually at the GGP where participants compete to find the best GGP AI[21].

The GGP framework includes a large database of games. In a GGP match games from these databases are selected at random and sent to the competitors[21]. The games are specified in the Game Description Language (GDL), a logic programming language built for describing games as state machines[31]. A logic programming language being any that is mainly based on formal logic, such as Prolog (see section 2.3). An example of GDL rules is given in listing 1.

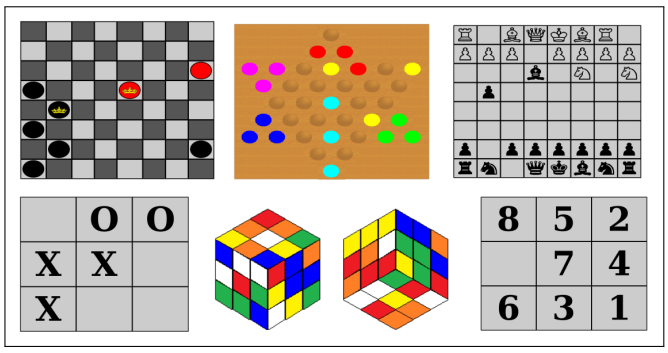


Figure 1.1: A selection of games from the GGP competition. From the top right: *checkers*, *chinese checkers*, *chess*, *tic tac toe*, *rubix cube* and *eight puzzle*

```
(succ 0  1)
(succ 1  2)
(succ 2  3)
(beats scissors paper)
(beats paper stone)
(beats stone scissors)
(<= (legal ?p scissors) (player ?p))
(<= (legal ?p paper) (player ?p))
(<= (legal ?p stone) (player ?p))
(<= (draws ?p) (does ?p ?a) (does ?q ?a) (distinct ?p ?q))
(<= (wins ?p) (does ?p ?a1) (does ?q ?a2) (distinct ?p ?q) (beats ?a1 ?a2))
(<= (loses ?p) (does ?p ?a1) (does ?q ?a2) (distinct ?p ?q) (beats ?a2 ?a1))
```

Listing 1: A sample of rules from the GDL description of Rock Paper Scissors. The ? indicates a variable and <= indicates an implication with the first expression after being the head and the conjugation of the rest making up the body

These GDL game descriptions form the basis for the Inductive General Game Playing (IGGP) problem[11]. The task is an inversion of the GGP problem. Rather than taking game rules and using them to play the game in IGGP the aim is to learn the rules from observations of gameplay, similar to how a human might work out the rules of a game by watching someone play it. Cropper et al. define the IGGP problem in their 2019 paper; given a set of gameplay observations the goal is to induce the rules of the game[11]. The games used in IGGP are a selection of those from the GGP competition problem set, specified in GDL, meaning they are widely varied in complexity. An example IGGP for *rock paper scissors* is given in table 1.1 and the rules given for the GGP game description in table 1.2

An effective way of solving the IGGP problem is a form of machine learning: Inductive Logic Programming (ILP) [11, 33]. In ILP, the learner is tasked with learning logic programs given some background knowledge and a set of values for which the programs are true or false (also expressed in a logical programming language). In the IGGP paper, the authors showed through empirical evaluations that ILP systems achieve the best score in this task compared to other machine learning techniques[11]. The ILP system derives a hypothesis, a logic program that when combined with the background knowledge entails all of the positive and none of the negative examples[33]. In the IGGP paper it is also shown that the problem is hard for current ILP systems, with on average only 40% of the rules being learned by the best performing systems[11].

However, the existing work has limitations. All work so far has assumed that the gameplay being observed is randomly generated[11]. Rather than agents playing to win they simply make moves at random. Often this has the result of the game terminating before it reaches a goal state due to a cap on trace length or sections of the rule set remaining completely unused. In previous work there has also not been any research done to ascertain the effects that increasing the number of game observation has on the quality of the induced rules. It is unknown whether there is a threshold at which new any new game traces introduced are insignificant. In this paper we use the IGGP framework to evaluate the ability of ILP agents to correctly induce the rules of a game given different

BK	E^+	E^-
`(beats scissors paper)`		
`(beats paper stone)`		
`(beats stone scissors)`		`(next (score p1 1))`
		`(next (score p1 2))`
`(succ 0 1)`		`(next (score p1 3))`
`(succ 1 2)`	`(next (score p1 0))`	`(next (score p2 0))`
`(succ 2 3)`	`(next (score p2 1))`	`(next (score p2 2))`
		`(next (score p2 3))`
`(player p1)`	`(next (step 1))`	
`(player p2)`		`(next (step 0))`
		`(next (step 2))`
`(true (step 0))`		`(next (step 3))`
`(true (score p1 0))`		
`(true (score p2 0))`		

Table 1.1: An example of an IGGP task for *Rock Paper Scissors*. The task consists of the background knowledge (BK), the positive examples (E^+) and the negative examples (E^-). Here the predicate to be learned is `next`. We treat the `next score` and `next step` as two separate tasks

```
next_step(N):-
  true_step(M),
  succ(M,N).
next_score(P,N):-
  true_score(P,N),
  draws(P).
next_score(P,N):-
  true_score(P,N),
  loses(P).
next_score(P,N2):-
  true_score(P,N1),
  succ(N2,N1),
  wins(P).
```

```
draws(P):-
  does(P,A),
  does(Q,A),
  distinct(P,Q).
loses(P):-
  does(P,A1),
  does(Q,A2),
  distinct(P,Q),
  beats(A2,A1).
wins(P):-
  does(P,A1),
  does(Q,A2),
  distinct(P,Q),
  beats(A1,A2).
```

Table 1.2: The GGP rules for the `next step` and `next score` predicates. Translated from GDL to Prolog for readability.

sets of gameplay observations - intelligent gameplay verses random gameplay as well as combinations of the two. To generate the intelligent gameplay we use the system Sancho, winner of the 2014 GGP competition[1][43]. Every year the GGP competition includes a 'Carbon versus Silicone' event where humans pit themselves against the winner of the main competition. The machines almost always win[21] so we can assume Sancho has above amateur human level play. We also vary the number of gameplay traces from which the ILP systems learn to determine the optimal number of traces.

It is not obvious which of random or intelligent gameplay will achieve the best results. When learning the rules of chess would a human rather watch moves being made randomly, or a match between two grandmasters? It is not an easy question to answer. Both situations will result in a restricted view of the game, with certain situations never occurring in each one. This is not only a dilemma in the context of learning game rules. For example, teaching a self driving car to navigate roads requires training it on examples of driving. We would clearly not train it on highly intelligent Formula One quality driving and neither would we train it on random movement of the car. The question to be asked is what is the ideal quality level of training data to use to best teach a system the rules you want it to learn. In this paper, we try to help give some insight into this fundamental question. Specifically, we ask the following research questions:

Q1 Does varying the quality of game traces influence the ability for learners to solve the IGGP problems? Specifically, does the quality of game play affect predictive accuracy?

Q2 Does varying the amount of game traces influence the ability for learners to solve the IGGP problems? Specifically, does the quality of game play affect predictive accuracy?

Q3 Can we improve the performance of a learner by mixing the quality of traces?

We will train a range of ILP systems that each use a different approach to the problem on different sets of training data. The results for Q1 will be the most interesting as it is not clear what the expected outcome is. Q2 has more of a natural answer, it is generally accepted that for machine learning problems the more training data you have the better the predictive accuracy of the ML system [32]. We ask this question to get some insight into how much the predictive accuracy is affected by the training set size. Often in ILP only a small amount of training data is needed, adding more data may not significantly affect accuracy [33].

The third question is an interesting one. Intuitively greater diversity in the training data should give a result closer to the rule that generated the data. However if a learner is trained on random data and only tested on random data we would expect this to perform better than a learner trained on random data then tested on a mix of optimal and random data. This question thus highlights an issue we face: how do we test the learned game rules?

Ideally the generated rules would be compared directly against the rules in the GDL game descriptions. We would take the generated rule and see for

[1] http://ggp.stanford.edu/iggpc/winners.php accessed 28th April 2020

what percentage of all possible game states the reference rule and the learned rule gave the same output. Unfortunately we do not have the computational resources to do this with a lot of the games having too many possible states such as checkers which has a state-space complexity of roughly $5.0 \cdot 10^{20}$[45] and sudoku which exceeds $6.6 \cdot 10^{21}$[18]. Instead we will test the learned programs on both intelligently generated and randomly generated data of the same quality used in training.

We would expect models trained on the same distribution as they are tested on to perform best since it is generally accepted that the accuracy of a model increases the closer the test data distribution is to the training data distribution [32]. However, Gonzales and Abu-Mostafa[22] suggest that a system trained on a different domain to the one it is tested can outperform a system trained and tested on the same domain. Given a test distribution there exsists a duel distribution that, when used to train, gives better results. The duel distribution gives a lower out-of-sample[2] error than using the test distribution. This duel distribution can be thought of as the point in the input space where the least out-of-sample error occurs[22].

As well as optimising the single training distribution we can take data from multiple distinct distributions. Ben-David et. al. show that training data taken from multiple different domains can in fact give lower error on testing data that traning data taken from any single domain, including the testing domain [4]. It is not clear in our case what selection of training data will result in the most effect learning.

To help answer questions 1-3, we make the following contributions:

Contributions

- We implement a system to play GGP games at (1) random and highly intelligent levels (Chapter 4)
- We transform the GGP traces to IGGP problems (Chapter 4)
- We train the ILP systems Metagol, Aleph and ILASP on different combinations of intelligent and random data as well as testing them on each individually (Chapter 5)
- We train the ILP systems on differing amounts of traces and test to ascertain the effect this has on the accuracy of the predicted results (Chapter 6)

[2]out-of-sample data is data that is not in the training set

Chapter 2

Related work

2.1 GGP

General game playing is a framework for evaluating an agenet's general intelligence accross a wide range of tasks [11, 21]. The idea is that agents are able to accept declarative descriptions of arbitrary games at run time and are able to use such descriptions to play those games effectively. All the games are finite, discrete, deterministic multi-player games of complete information[31]. The games in the GGP competition game set vary in number of players, dimensions and complexity. For example games such as rock paper scissors have 0 dimensions and only 10 rules in the given GGP rule set, more complex games such as checkers has 52 rules and are 2 dimensional. There are also single player games such as Eight Puzzle or Fizz Buzz.

The agents play games selected at random. They are sent a listing of the rules described in the Game Description Language (GDL). GDL is a language based on first order logic (section 2.1.1). The list of games along with their descriptions are available online[1]. Matches take place through an online framework called the Gamemaster. The agents connect to an online Game Manager (part of the Gamemaster) service which arbitrates individual matches. After the players have establish a HTTP connection to the Game Manager one of the players takes the role of game director. The game director then broadcasts the following information to the game manger: 1) the name of a game that is know to the manager 2) the required number of players 3) the amount of time to be given for each tern. The match can then be started by pressing a start button, when pressed the agents will receive a *Start* message including the role of the player (e.g. black or white in chess) and a description of the game in GDL. The Game Manger communicates all instructions to the players through HTTP[21].

In 2005 an annual International General Game Playing Competition (IGGPC) was set up which still runs to this day[26]. Each year hopeful participants pit GGP agents against one another to determine the most effective system. The competitors take part in a series of rounds of increasing complexity. The agent that wins the most games in these rounds is declared victorious. The 2014 winner Sancho is used in this paper to generate optimal game traces for the IGGP task.

[1] http://ggp.stanford.edu/iggpc/

The game descriptions written in GDL used in the GGP competition can be used as example rule sets for systems in Inductive General Game Playing (IGGP) as the output the learners would ideally generate. The descriptions of the games used in GGP are not necessarily minimal so it is possible that an ILP system could generate a more concise rule set than the GGP descriptions.

2.1.1 GDL

Game definition language is the formal language used in the GGP competition to specify the rules of the games[31]. The language is based off a logical programming language *datalog*, a subset of Prolog. To understand the semantics of GDL it helps to first cover logic programming as a field of study.

Logic Programming

Logic programming is a programming paradigm based on formal logic. Programs are made up of facts and rules. Rules are made up of two parts: the head and the body. They can be read as logical implications where the conjunction of all the elements in the body imply the head. The syntax is different for different logical programming languages but the head is usually written before the body in reverse implication. For example `a <= b,c` states that `b` and `c` imply `a`. A fact is simply a rule without a body, that is, a statement that is taken as true. The compiler takes queries and returns weather they are true of false. If there are free variables in the query the compiler assigns them values for which the query is true. Logical programming is good for symbolic non-numeric computation[6]. Prolog is one of the most well know logical programming languages[46] and provides a good introduction to the field. It is discussed in section 2.3. Logical programming is well suited to solving problems that involve well defined objects and relations between them, such as a GGP game.

Usefulness of GDL

The game description language was designed specifically to represent finite, discrete, deterministic multi-player games of complete information. It is suited to specifying game rules because:

- It is a purely declarative language
- It has restrictions to ensure that all questions of logical entailment are decidable
- There are some reserved words (such as *terminal* or *goal*), which tailor the language to the task of defining games

These descriptions define the games in terms of a set of true facts capturing the information needed to give the following predicates:

- The initial game state
- The goal state
- The terminal state

In addition, logical rules are used to describe the following:

- The legal moves for a given player and state
- The next state for a given player state and move
- The termination and goal conditions

The GDL language is an extension of $datalog^{\neg}$, that is datalog with stratified negation[31]. *datalog* allows only rules consisting of a conjunction of positive atoms that imply a single atom. $datalog^{\neg}$ allows for negative as well as positive atoms[1]. GDL also allows for some functional symbols, that is predicates containing other predicates however it restricts to keep the finite model property that is inherent to *datalog*[31].

In GDL variables are written with the symbol `?` before them and rules are written starting with `=>` followed by the head followed by the body predicates which are interpreted as a conjunction. For example

```
(<= (next (cell ?x ?y ?player)) (does ?player (mark ?x ?y)))
```

in first order logic this would be written

$$does(player, (mark(x, y))) \rightarrow cell(x, y, player)$$

GDL also reserves certain words which are interpreted in special ways:

- `true(?f)` - Atom `?f` is true in the current game state
- `does(?r,?m)` - Player `?r` performs action `?m` in the current state
- `next(?f)` - Atom `?f` will be true in the next game state
- `legal(?r,?m)` - Action `?m` is a legal move for player `?r` in the current game state
- `goal(?r,?n)` - Player `?r` performs action `?m` in the current game state
- `terminal` - The current state is terminal
- `init(?f)` - Atom `?f` is true in the initial game state
- `role(?n)` - The Constant `?n` denotes a player
- `distinct(?x,?y)` - `?x` and `?y` are syntactically different

In the IGGP problem the given task is to generate the rules to predict the values for the goal, the next state, the legal states and the terminal predicates.

2.2 IGGP

The Inductive General Game Playing (IGGP) problem is an inversion of the GGP problem. Rather than using game rules to generate gameplay the learner must learn the rules of the game by watching others play. The learner is given a set of game traces and is tasked with using them to induce (learn) the rules of the game that could have produced the traces[11]. IGGP was designed as a way of benchmarking machine learning systems.

The definition of task itself is based on the Inductive Logic Programming problem.

2.2.1 ILP

Inductive logic programming is a form of machine learning that uses logic programming to represent examples, background knowledge, and learned programs[14]. To learn the program is supplied with positive examples, negative examples and the background knowledge. In the general inductive setting we are provided with three languages.

- $\mathcal{L}_O$: the language of observations (positive and negative examples)
- $\mathcal{L}_B$: the language of background knowledge
- $\mathcal{L}_H$: the language of hypotheses

The general inductive problem is as follows: given a consistent set of examples or observations $O \subseteq \mathcal{L}_O$ and consistent background knowledge $B \subseteq \mathcal{L}_B$ find an hypothesis $H \in \mathcal{L}_H$ such that

$$B \wedge H \vDash O$$

[33] That is that the generated hypothesis and the background knowledge imply the positive examples and not imply the negative examples.

Grandparent example

A classic illustration of the problem is the learning of the grandparent relation. Here we want to derive a logical program that given information about who is a parent of whom will be able to tell us the all the people each person is a grandparent of. It needs to capture the idea that if Y is a parent of X and Z a parent of Y then Z is a grandparent of X. We are given the following background knowledge.

$$B = \begin{cases} mother(alice, jane) \\ mother(jane, dave) \\ mother(alice, henry) \\ father(henry, bob) \end{cases}$$

Here we are not directly given the parent relation. This will have to be induced by the learner. The positive and negative observations are given by $O^+ \cap \neg O^- = O$.

$$O^+ = \begin{cases} grandparent(alice, bob) \\ grandparent(alice, dave) \end{cases} \qquad O^- = \begin{cases} grandparent(alice, jane) \\ grandparent(henry, bob) \\ grandparent(dave, henry) \end{cases}$$

The program we induce would hopefully look something like this:

$$H = \begin{cases} grandparent(X, Y) \leftarrow parent(X, Z) \wedge parent(Z, Y) \\ parent(X, Y) \leftarrow mother(X, Y) \\ parent(X, Y) \leftarrow father(X, Y) \end{cases}$$

It should hopefully be clear that this program satisfies $B \wedge H \vDash O$

ILP systems generally regard ILP as a search problem. The search space is the set of well formed hypothesis. Often a set of inference rules are applied to the starting hypothesis, the new hypothesise are then pruned and expanded according to how large $E \subseteq O$ is in $B \wedge H \vDash E$. When all the observations are implied then a correct program has been found.

2.2.2 Back to IGGP

In IGGP we also have the idea of background knowledge and positive or negative observations. The task is given as a triple $\{(B, E^+, E^-)\}$. The positive examples E^+, the negative examples E^- and the background knowledge B. Similar to the Inductive logic programming problem the learner 2.2.1 has to come up with a hypothesis H such that H and the background knowledge imply all the positive examples but none of the negative examples, that is $H \cup B \models E^+$ and $H \cup B \not\models E_i^-$. All three are given in the form of a set of ground atoms, that is they do not contain any free variables.The background knowledge of a task consists of all the rules of the GDL game description that do not concern the predicate being learned. For example in Rock Paper Scissors this would be the facts that give which item beats which (i.e. that scissors beat paper) as can be seen in table 1.1. The IGGP problem itself is described in chapter 3.

The IGGP dataset, also given in the same paper as the definition of the problem. It is a collection of 50 games, specified in GDL. The purpose of this database is to standardises the set of games used in the IGGP problem to allow for results to be easily compared. It is in fact the set of games that this paper is based on. A mechanism is also provided by the authors to turn these GDL game descriptions in the set into new IGGP tasks. This method plays the games randomly to generate the observations. In this paper we modify the mechanism to generate optimal game traces.

To better understand the IGGP problem and the methods of solving it helps to first understand Prolog. The language is used as the basis for a large number of ILP systems[13, 34, 2].

2.3 Prolog

Prolog is one of the most popular and well used logical programming languages. The syntax in Prolog for rules and facts are fairly intuitive. A simple fact might be `son(bob,alice).`. This tells us that the atom `alice` relates the atom `bob` in the `son` relation. A rule is written `parent(X,Y) :- son(Y,X)` which means if we have the relation son of Y and X then X relates Y in parent. The symbol :- is the same as reverse implication. We can query a program in the Prolog environment. If we typed in "`parent(alice,bob).`" then it would return true because we have `son(bob,alice)` and the rule which tells us that if X is a son of Y else then Y is a parent of X so `alice` is a parent of `bob`. Predicates can be conjoined with the comma "," (e.g. `a(X,Y,Z) :- b(X,Y), c(Z,Y)`). Disjunction is expressed with the semicolon ";" (e.g. `a(X,Y,Z) :- b(X,Y) ; c(Z,Y)`[6].

Prolog answers queries using a process know as proof seach and unification. The unification process is similar to logical unification, two terms unify if they are the same term or if they contain variables that can be instantiated with terms in such a way that the new terms are equal. For example the terms

`name(bob).` and `name(X).` will unify with `X = bob`. The Prolog ISO defines the Herbrand Algorithm for unification [24]. A Prolog query is a set of goals. The Prolog System decides weather they are satisfiable or not. When a query is asked the of the Prolog system it executes something similar to the following algorithm. The exact implementations vary among Prolog systems however they generally follow this procedure.

Algorithm 1: Execute Prolog Goals

Input: A list of goals $GoalList = G_1, G_2, ..., G_M$
Function *execute* *(Program, GoalList, Success):*
  if *empty*($GoalList$) **then**
    $Success \leftarrow true$
  end
  while *not* *empty(GoalList)* **do**
    $Goal \leftarrow head(GoalList)$;
    $OtherGoals \leftarrow tail(GoalList)$;
    $Satisfied \leftarrow false$;
    while *not* *Satisfied and there are more clauses in the program* **do**
      Let next clause in the program be $H \vdash B_1, ..., B_n$;
      Construct a variant of this clause $H' \vdash B'_1, ..., B'_n$;
      $match(Goal, H', MatchOK, Instant)$;
      if $MatchOK$ **then**
        $NewGoals \leftarrow append([B'_1, ..., B'_n], OtherGoals)$;
        $NewGoals \leftarrow substitue(Instant, NewGoals)$;
        $execute(Program, NewGoals, Satisfied)$;
      end
    end
    $Success \leftarrow Satisfied$;;
  end

The program listing is searched for a term to unify with. The listing is searched in the order it is written in. When Prolog finds a matching rule it then attempts to sequentially unify the terms of the body using the same method. If the rule has no body then the variables are assigned and the terms unify. If Prolog fails to unify two terms then it backtracks, assigns the last variable a different values. This continues until a proof is found or all possibilities have been exhausted[6]. Prolog forms the basis of two of the ILP systems the we test.

2.4 ILP systems used

We use three ILP systems to compare the effects of different learning data, Metagol, Aleph and ILASP. There are many approaches to the ILP problem[5, 10]; the three systems here all represent different approaches to the problem but certainly do not give a full representation of the techniques available.

2.4.1 Metagol

The Metagol ILP system is a meta-interpreter for Prolog, that is, it is written in the same language is evaluates[9, 12, 13]. Metagol takes positive and negative examples, background knowledge and meta-rules. Meta rules are specific to Metagol. They determin the shape of the induced rules and are used to guide the search for a hypothesis. An example of a metarule is the *chain* rule

$$P(A, B) \leftarrow Q(A, C), R(C, B)$$

The letters P, Q and R represent existentially quantified second order variables, A, B and C are regular Prolog variables. When trying to induce rules the second order variables are substituted for predicates from the background knowledge or the hypothesis itself. To illustrate this consider a metarule being applied when learning the predicate *last(A,B)* where a is a list and b is the last element in it. Given the positive example

```
last([a,l,g,o,r,i,t,h,m],m).
```

As well as the background predicates *reverse/2* and *head/2* the chain rule might be used to derive the rule

```
last(A,B) :- reverse(A,C), head(C,B)
```

1. Select a positive example (an atom) to generalise. If none exists, stop, otherwise proceed to the next step.
2. Try to prove an atom using background knowledge by delegating the proof to Prolog. If successful, go to step 1, otherwise proceed to the next step.
3. Try to unify the atom with the head of a metarule and either choose predicates from the background knowledge that imply the head to fill the body. Try proving the body predicate, if it cannot be proved try different BK[2]. If no BK can be found that prove the positive example then try adding a new invented predicate and attempt to prove this[3].
4. Once you find a metarule substitution that works add it to the program and move to the next atom

The end hypothesis is all the metarule substitutions. It is the checked that the negative atoms are not implied by the hypothesis, if they are a new one is generated. When the hypothesis is combined with the background knowledge the positive examples, but not the negative examples, are implied.

The choice of metarules determines the structure of the hypothesis. Different choice of metarules will allow different hypotheses to be generated. Deciding which metarules to use for a given task is an unsolved problem [9]. For this task a set of 9 derivationally irreducible metarules are used which remain consistent across all tasks.

[2]To prove the body predicate the whole procedure is called again with the body predicates as the positive examples. For example if we had `last([a,b],b)` as our positive example and have tried to use the chain metarule to with reverse and head we would then call the whole procedure again with the positive examples as [reverse([a,b], C),head(C,b)] if this was successful then we continue, otherwise we try different predicates

[3]For example we might replace head with an invented predicate in the previous footnote example

2.4.2 ALEPH

Aleph is an Prolog variant of the ILP system Progol [34]. As input, like any other ILP system, Aleph takes positive and negative examples represented as a set of facts along with the background knowledge. It also requires *mode declarations* and *determinations* which are specific to Aleph. Mode declarations specify the type of the inputs and outputs of each predicate used e.g. `plus(+integer,+integer,-integer)` where `+` signifies an input and `-` an output. Determinations specify which predicates can go in the body of a hypothesis. These determinations take the form of pairs of predicates, the first being the head of the clause and the second a predicate that can appear in its body.

For each predicate we would like to learn in the IGGP problem we give Aleph the determinations consisting of every target predicate (next,goal and legal) paired with every background predicate (which are specific to each game). Luckily there has been some work to induce mode declarations from the determinations [16] so we do not need to come up with our own mode declarations.

A basic outline of the Aleph algorithm is taken from the aleph website [4]:

1. Select an example to be generalised. If none exist, stop, otherwise proceed to the next step.

2. Construct the most specific clause (also known as the bottom clause [34]) that entails the example selected and is within language restrictions provided.

3. Search for a clause more general than the bottom clause. This step is done by search- ing for some subset of the literals in the bottom clause that has the 'best' score.

4. The clause with the best score is added to the current theory and all the examples made redundant are removed. Return to step 1.

Mode declarations and determinations are used in step 2 of this procedure to bound the hypothesis space. Only predicates that are mentioned in the determinations of the hypothesis and are of the correct type are tried. The bottom clause constructed is the most specific clause that entails the example. Therefore a clause with the same head and any subset of the predicates of the body will be more general than the bottom clause. Aleph only considers these generalisations of this bottom clause. The search space is therefore bounded by 2^n with n being the number of predicates in the bottom clause.

By default Aleph preforms a bounded breath first search on all the possible rules, enumerating shorter clauses before longer clauses. The search is bounded by several parameters such as maximum clause size and maximum proof depth. The best score is selected as the one with the best $P - N$ value where P is the number of positive rules entailed by the hypothesis and N is the number of negative rules entailed. For this paper we will use Aleph with the default settings.

[4] http://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html accessed 26/03/2020

2.4.3 ILASP

ILASP is an ILP system based on Answer Set Programming (ASP). An introduction to ASP can be found here [8]. ILASP uses a subset of ASP that is defined in these papers[27, 29, 28] The ILASP process effectively generates all possible rules of a certain length, turns the problem into an ASP problem that adds a predicate to each rule allowing it to be active or inactive. It then uses the ASP solver clingo[19] to check which rules should be active if the program is to be consistent with the positive and the negation of the negative examples[29, 28]. In this paper we use a version of ILASP based on ILASP2i[30] which was developed with the IGGP problem in mind[11]. As one input ILASP takes a hypothesis search space, i.e. the set of all hypotheses to be considered. This is constructed using the type signatures given for each problem that are provided in the IGGP dataset.

An ILASP task is defined as a tuple $T = \langle B, S_M, E^+E^- \rangle$ where B is the background knowledge, S_M is the hypothesis space, and E^+ and E^- are the positive and negative examples. The ILASP procedure is given in algorithm 2.

Algorithm 2: ILASP outline

$n = 0$;
solutions = [];
while *solutions.empty* **do**
  S^N = all possible hypotheses of length N from S_M ;
  ns = all subsets of S^N that imply E^- (Using an ASP solver);
  vs = the set of rules that for each set of rules in ns imply false if exactly those rules are active;
  solutions = all subsets of S^N that imply E^+ and satisfy vs (using asp solver);
  $n = n + 1$;
end

The approaches used by ILASP have proved to be well suited to the IGGP task [11]. We expect it to do well in the experiments conducted.

Chapter 3

IGGP problem

The IGGP problem is defined in the 2019 paper Inductive General Game Playing [11]. Much like the problem of ILP described in section 2.2.1 the problem setting consists of examples about the truth or falsity of a formula F and a hypothesis H which covers F if H entails F. We assume the languages of:

- $\mathscr{E}$ the language of examples (observations)
- $\mathscr{B}$ the language of background knowledge
- $\mathscr{H}$ the language of hypotheses

Each of these languages can be see as a subset of those defined for the ILP task. These languages are made up only of function free ground atoms[40]. In our experiments we transform data from the GDL descriptions of games in the IGGP dataset and transform it into the languages of $\mathscr{B}$ and $\mathscr{E}$. GDL allows for functions (predicates nested in one another) in rules albeit in a restricted form. For example any atom appearing inside a `true` predicate such as `true(count(9))`. We flatten all of these to single, non nested predicates, i.e. `true_count(9)`. This is needed as not all ILP systems support function symbols. We can therefore assume that both $\mathscr{E}$ and $\mathscr{B}$ are function-free. The language of hypotheses $\mathscr{H}$ can be assumed to consist of datalog programs with stratified negation as described here[38]. Stratified negation is not necessary but in practice allows significantly more concise programs, and thus often makes the learning task computationally easier. We first define an IGGP *input* then use it to define the IGGP *problem*. The IGGP input needs to capture the idea of a set of observations about a single game. The input is based on the general input for the Logical induction problem.

The IGGP Input: An input Δ is a set of triples $\{(B_i, E_i^+, E_i^-)\}_m^{i=0}$ where

- $B_i \subset \mathscr{B}$ represents background knowledge
- $E_i^+ \subseteq \mathscr{E}$ and $E_i^- \subseteq \mathscr{E}$ represent positive and negative examples respectively

The IGGP input composes the IGGP problem as follows:

The IGGP Problem: Given an IGGP input Δ, the IGGP problem is to return a hypothesis $H \in \mathscr{H}$ such that for all $(B_i, E_i^+, E_i^-) \in \Delta$ it holds that $H \cup B_i \models E_i^+$ and $H \cup B_i \not\models E_i^-$.

In this paper we use the IGGP problem to analyse ability of ILP agents to learn from on a range of training data of differing quality. To analyse the ability of the ILP systems to learn we use, for each IGGP task Δ and hypothesis H, a testing set $T = T^+ \cup T^-$ where

- $T^+ \subseteq \mathscr{E}$ and $T^- \subseteq \mathscr{E}$ are the positive and negative testing observations.
- $T^+ \cap \bigcup_{i=0}^{m} E_i^+ = \emptyset$. The positive testing examples are distinct from the positive training examples
- $T^- \cap \bigcup_{i=0}^{m} E_i^- = \emptyset$. The negative testing examples are distinct from the negative training examples

To test the systems we check for each $t^+ \in T^+$ weather

$$H \cup \bigcup_{i=0}^{m} B_i \models t^+$$

and for each $t^- \in T^-$ weather

$$H \cup \bigcup_{i=0}^{m} B_i \not\models t^-$$

The success of the system is defined as the percentage of correctly classified $t \in T$. We use this definition of success to answer the research questions in chapter 1. The experiments preformed to answer these questions are described in chapter 5.

Chapter 4

Generating Traces

In this section we describe the process of generating and transforming game traces to IGGP tasks. The first step is to generate the intelligent and random play. To do this we used the Sancho system[43].

4.1 Sancho

To generate optimal gameplay we decided that the best approach was to use a previous winner of the GGP competition. Since the aim of the competition is to find the program that performs best at the set of games used in the IGGP problem we conclude that there is no better way to generate a comprehensive set of game traces. The winner of the 2014 GGP competition 'Sancho' was selected [1] since they are the most recent winner to have published their code[43]. Some small modifications to the information logged by the game server are made but otherwise the code is unchanged.

The core of algorithm used by Sancho is the Monte Carlo tree search (MCTS).

4.1.1 Monte Carlo Tree Search

Given a game state the basic MCTS will return the most promising next move. The algorithm achieves this by simulating random playouts of the game many times. The technique was developed for computer Go but has since been applied to play a wide range of games effectively including board games and video games[42, 7].

The use of random simulation to evaluate game states is a powerful tool. Information about the game such as heuristics for evaluating non terminal states are not needed at all, the rules of the game are enough on their own. In the case of the GGP problem this is ideal. The rules of the game are only revealed shortly before the game is played meaning deriving effective heuristics is a hard problem.

In our case all games being played are sequential, finite and discrete so we only need to consider MCTS for this case.

The MCTS is a tree search algorithm, the tree being searched is the game tree. A game tree being a tree made up of nodes representing states of the

[1] http://ggp.stanford.edu/iggpc/winners.php accessed on 12/03/2020

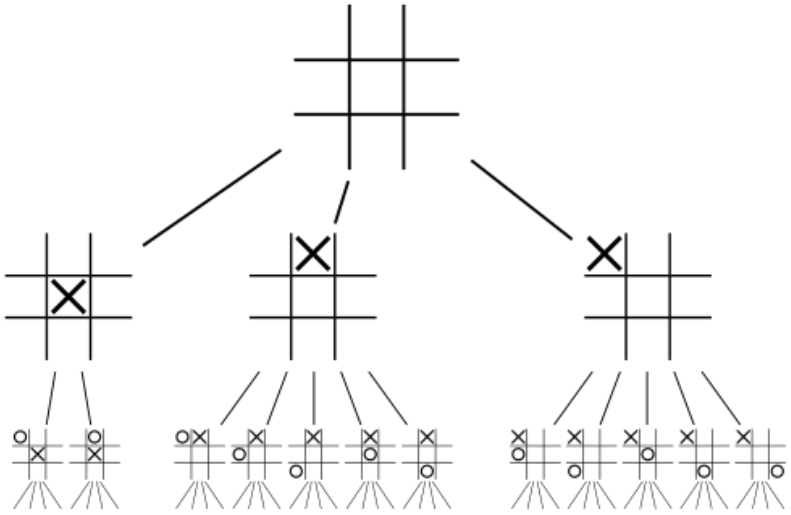


Figure 4.1: A section of a game tree of *tic tac toe* showing some of the possible moves. *Stannered/CC-BY-SA-3.0*

game and the children of each node being all the states that can be moved to from that state (see figure 4.1). The leaves of this tree are the terminal states. The search is a sequence of traversals of the game tree. A *traversal* is a path that starts at the root node and continues down until it reaches a node that has at least one unvisited child (not necessarily a terminal state). One of these unvisited children are then chosen to be the start state for a simulation of the rest of the game. The simulation chooses random moves, playing the game out to a terminal state. The result of the simulation is propagated back from the node it started at all the way to the root node to update statistics attached to each node. These statistics are used to choose future paths to traverse so that more promising moves are investigated with higher probability.

Exactly how the tree search algorithm chooses new nodes to simulate is where the complexity lies. The node chosen by traversal of the tree should strike a balance between exploiting promising nodes and exploring nodes with few simulations. The Upper Confidence Bound for trees algorithm was designed to do exactly this[25]. The algorithm chooses a child node for each state based on the UTC formula. The formulae can be written as:

$$UTC(v_i) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{ln(N(v_i))}{N(v_i)}}$$

Where we have:

- v_i is node v after move i
- $Q(v)$ is the number of winning simulations that have taken place below it
- $N(v)$ is the number of simulations that have taken place below it

The function is the sum of two components. The first part $\frac{Q(v_i)}{N(v_i)}$ is called the exploitation component. It is a ratio of winning to loosing simulation that resulted from making this move. This encourages traversal of promising nodes that have a high win rate. However, if only this factor was used the algorithm would quickly find a winning node and only explore that hence we need a second component: the exploration factor. This part $c\sqrt{\frac{ln(N(v_i))}{N(v_i)}}$ favours nodes that have not yet been explored. The value c is a constant that balances the two components, it can be adjusted depending on the use case.

Use of MCTS in Sancho

Sancho makes a few modifications to MCTS[2]. The main one being the adding of heuristics. The tree is also replaced with a more general graph which allows for transitions between lines of play without duplication of states. There are also optimisations to increase efficiency.

In GGP matches a period of time before the match is given in which to do pre match calculations. Sancho uses this period to derive basic heuristics and optimise the value of c in the UTC formulae.

A heuristic should take a game state or move and return a value based on how promising that state or move is in relation to the goal state. To take the heuristics into account when choosing the next state to explore each node (state) v is seeded with a heuristic value on creation.

The identification of possible heuristics takes place in two stages. The first of these is static analysis of the game rules. This static analysis identifies possible heuristics that can be applied to the current game. These include things like piece capture: if certain rules indicate capture of a piece these can be selected against. Another is numeric quantity detection: a number in the state acts as a heuristic (like number of coins a player has). The second stage is simulation of the game. Many (possibly tens of thousands) of full simulations of the game are run. After the simulations are complete a correlation coefficient is calculated between each candidate heuristic's observed values and the eventual score achieved in the game. Those heuristics that show correlation above a fixed threshold are then enabled, and will be used during play to guide the exploration of the MCTS.

4.1.2 Other aspects of Sancho

Monte Carlo simulations are used for a large number of the games available however Sancho identifies games that can be solved more efficiently in other ways. Any single player game in the GGP dataset can be analysed using puzzle solving techniques.

Puzzle Solving

A single player game in the GGP dataset is necessarily a deterministic game of complete information. This is due to the constraints of the Game Description Language, it is not possible to express anything more. This gives the useful constraint that any solution found in a playout of the game will always remain

[2]https://sanchoggp.blogspot.com/2014/05/what-is-sancho.html accessed on 15/03/2020

valid since the game is completely deterministic. Sancho identifies these single player games and attempts to derive heuristics. Where an obvious goal state exists a distance metric such as the hamming distance between two states can be applied. A* search can then be used to find optimal solutions. For some games, such as eight puzzle the hamming distance is an admissible heuristic so Sancho finds a provably optimal solution.

Static analysis

Sancho also does static analysis of the game rules. It determines game predicates that if true imply the goal will never be reached in any proceeding state as well as predicates that guarantee that the goal will be reached in a future state. An example of this is a game such as untwisty corridor (the game is effectively a maze where if you immediately lose if you step off a "safe" path). If the safe path is stepped off then the goal can never be reached so any state not on the safe path is avoided.

4.1.3 Sancho and IGGP

Sancho represents the best available automated player of the games in the IGGP dataset. Its level of play for most games is above that of an amateur human as has been shown in the Carbon versus Silicone matches at the GGP competition[21]. Sancho generally provides good quality intelligent play however in some cases it has struggled.

The IGGP dataset used for the experiments contains several games based on game theory. Games such as the Prisoner Dilemma where player can choose to either *cooperate* or *defect*. If both player cooperate then they both get 3 point. If one defects and the other tried to cooperate the cooperator gets 0 points and the defector gets 5. If they both defect both get 0. There are several rounds of this in one game playout. A human playing this might choose to try and cooperate, hoping that that the other player will too however Sancho has been observed to always defect. This result is predictable since this is the only strong Nash equilibrium for this game however it is hard to justify this as "human quality play".

4.2 Generating and transforming the traces

To determine the influence of the quality of training data upon the ability of learners to solve the IGGP problem we initially generate the game traces.

The general game playing community have developed a codebase that provides the basic functionality for hosting a match between GGP agents[3]. This codebase along with the GGP agent Sancho were used to generate the datasets. The codebase includes a very basic random GGP player that will, given a GDL game description, play a random legal move every turn. It uses the GDL description to generate all possible legal moves given the current game state then chooses one uniformly at random. To generate the random traces for a game of x players x random GGP players were pitted against one another. The same

[3]https://github.com/ggp-org/ggp-base

method was used to generate the intelligent traces with Sancho playing instead of the random gamers.

Information about games is represented as sets of ground atoms. These atoms can be divided into two sets, the atoms that can change during the game and the atoms that can not. The set of atoms that can change can again be divided in two, the action atoms A and the state atoms S.

The set S represents all atoms that can change from one state to the other. Examples of elements of this set S might be the state of the board in *tic tac toe* along with which player is to take their turn next:

```
( control noughts )
( cell 1 1 b ) ( cell 1 2 x ) ( cell 1 3 o )
( cell 2 1 x ) ( cell 2 2 o ) ( cell 2 3 o )
( cell 3 1 b ) ( cell 3 2 x ) ( cell 3 3 b )
```

The set A is the set of ground atoms representing the actions that can be taken. The moves that each player makes are taken from this set. For a game such as *eight puzzle* these would consist of the set of atoms:

$$\{\texttt{(move i j)} | 0 < i \leq 9, 0 < j \leq 9\}$$

Each atom represent sliding the tile at (i, j) into the space. For the game *tic tac toe* these atoms would be the set:

$$\{\texttt{(mark i j)} | 0 < i \leq 3, 0 < j \leq 3\}$$

These atoms represent marking the square (i, j) with the players symbol. To represent a player not making a move we also always have $noop \in A$.

Game traces

We modified the GGP matchmaker to log the following information:

- **Game state trace** - The sequence of game states: $states = (s_1, ..., s_n)$ where each $s_i \subseteq S$ is the set of ground atoms true in the i^{th} state.
- **Game roles** - The list of roles of each player in the game: $roles = (r_1, ..., r_k)$ e.g. *noughts* and *crosses* in *tic tac toe*
- **Move trace** - The sequence of moves made by each player after each state: $moves = ((m_{1,r_1}, ..., m_{1,r_k}), ..., (m_{n,r_1}, ..., m_{n,r_k}))$ where $m_{i,r_j} \in A$ is the i^{th} move of player r_j. In games where not all players move every turn then $m_{i,r_j} = noop$ shows that r_j made no move
- **Legal move trace** - The sequence of the legal moves for each player in each state: $legal = ((l_{1,r_1}, ..., l_{1,r_k}), ..., (l_{n,r_1}, ..., l_{n,r_k}))$. $l_{i,r_j} \subseteq A$ is the set of possible moves for r_j in state s_i
- **Goal value trace** - The sequence of goal value for each player on every state: $goals = ((g_{1,r_1}, ..., g_{1,r_k}), ..., (g_{n,r_1}, ..., g_{n,r_k}))$. $g_{i,r_j} \in [0, 100]$ represents how well player r_j has achieved the goal in state s_i. For some games such as eight puzzle this is 0 on every state until the winning state where it becomes 100

This represents all the data needed from the match to generate IGGP tasks. In each state all the positive atoms in S (that is the atoms true in the current state) are arguments of the `true` predicate. For example if at the start of the game the first cell is blank on the board then we would have `( true ( cell 1 1 b ) )` $\in$ s_1. All the positive atoms in A are arguments of the `does` predicate in a similar fashion.

4.2.1 Transforming game traces into IGGP tasks

We define a function that transforms the sequences into four separate induction tasks. An induction task is the set of triples $\{(B_i, E_i^+, E_i^-)\}_n^{i=0}$. We first define a function trace that translates the sequences given in the log of the match into a set of pairs $\{(B_i, E_i^+)\}_n^{i=0}$. We then define a function *triple* which gives the the set of triples $\{(B_i, E_i^+, E_i^-)\}_n^{i=0}$ from the set of pairs $\{(B_i, E_i^+)\}_n^{i=0}$.

We flatten the sequences of $(m_{i,r_1}, ..., m_{i,r_k})$ in *moves*, $(l_{1,r_1}, ..., l_{1,r_k})$ in *legals* and $(g_{1,r_1}, ..., g_{1,r_k})$ in *goals* to a set that includes the role name as an extra argument of the predicate (the arity is increased by 1). For example if

$$(m_{i,r_1}, m_{i,r_2}) = \texttt{[(move 2 2), (move 2 3)]}$$

and $r_1 = black$ and $r_2 = white$ then it would be replaced by m_i

$$m_i = \{\texttt{(move black 2 2), (move white 2 3)}\}$$

moves is now a sequence $(m_1, ..., m_n)$. With *legals* a similar intuative flattening procedure is applied

$$\begin{aligned}(l_{i,r_1}, l_{i,r_2}) = \ & [\{\texttt{(move 2 2), (move 1 1), (move 1 3)}\}, \\ & \{\texttt{(move 2 3), (move 1 2)}\}]\end{aligned}$$

and $r_1 = black$ and $r_2 = white$ then it would be replaced by l_i

$$\begin{aligned}l_i = \{&\texttt{(move black 2 2), (move black 1 3),} \\ &\texttt{(move white 2 3), (move white 1 2)}\}\end{aligned}$$

Again we now have that legals is a sequence $(l_1, ..., l_n)$. The same procedure is also applied to *goals* giving us a single flat list with the roles included in each predicate. Since terminal does not contain any extra information no processing at this stage is needed.

Building the traces

The trace function is defined for *legal*, *goal*, *next* and *terminal*. We treat the output of the zip function as a set. We define $S[p/q]$ on a set S for predicates p and q to be the substitution of all instances of q in S for p.

$$\begin{aligned}
\Lambda_{legal} &= trace_{legal}(states, legal) && = \text{zip } states\ legal[\texttt{legal/true}] \\
\Lambda_{goal} &= trace_{goal}(states, goals) && = \text{zip } states\ goal \\
\Lambda_{next} &= trace_{next}(states, moves) && = \text{zip } states\ (\text{tail } states[\texttt{next/true}]) \\
\Lambda_{terminal} &= trace_{terminal}(states) && = \text{zip } states\ t \\
&\quad \textbf{where } t = \text{take } n\ (\text{repeat } \emptyset) && ++\ [\{\texttt{(terminal)}\}] \\
&\quad \textbf{and } n = (\text{length } states) - 1
\end{aligned}$$

The substitutions [`legal/true`] and [`next/true`] ensure that all positive examples in E^+ are in the relevant predicate for training. Before, the list *legal* consisted of the only atoms in the predicate `true`. For the ILP systems to identify this as an example of legal we need to surround these with the `legal` predicate. That is, we need all the `move` atoms in *legal* to be inside the `legal` predicate, e.g.:

```
( legal ( move 1 1 ) )
```

Since the predicates are already in the `true` predicate we only need to substitute one out for the other.

Some of the ILP systems being tested cannot work with function symbols of arity greater than 0. To allow them to operate we merge the function and their arguments into one predicate. For example `( legal ( move 1 1 ) )` becomes `( legal_move 1 1 )` where `legal_move` is a newly formed predicate.

To generate the triples we use the two functions $triples_1$ and $triples_2$. Let $pred\ X\ p$ be the subset of atoms in the set X that use the predicate p.

$$
\begin{aligned}
&triple_1(\Lambda_p) = \text{map } f\ \Lambda \\
&\quad \textbf{where } f\ (B, E^+) = (B, E^+, (pred\ S\ p) - E^+) \\
&triple_2(\Lambda_p) = \text{map } f\ \Lambda \\
&\quad \textbf{where } f\ (B, E^+) = (B, E^+, (pred\ A\ p) - E^+)
\end{aligned}
$$

We generate the IGGP task with $triples_1$ and $triples_2$ as below:

$$\Delta = triples_1(\Lambda_{goal}) \cup triples_1(\Lambda_{terminal}) \cup triples_2(\Lambda_{next}) \cup triples_2(\Lambda_{legal})$$

The IGGP task Δ is set to the ILP systems as described in chapter 5

Chapter 5

Experimental methodology

In this section we describe the exact methods used to run the experiments. Multiple experiments were run. To answer questions Q1 and Q3 experiment E1 was carried out in which three systems were trained on a set of random, intelligent and mixed quality traces. To answer Q2 experiment E2 was carried out in which the systems were trained on a varying number of mixed quality traces.

5.1 Materials

Generating training data

The training data was generated according to the methods described in chapter 4. To generate the game traces the Sancho version 1.61 was used[43]. The only modifications made to this were the changes to the logging system. Extra code was added to record the legal moves, the goal values at each state and the names of the players. Sancho was run through the GGP-base framework, a codebase developed by the general game playing community for hosting matches between GGP agents[1].

To generate traces matches between multiple random players or multiple instances of Sancho were conducted. Each match was run with 30 seconds pre game warm up time and a maximum of 15 seconds per move. The games used are listed in table 5.1. Due to issues of compatibility we were not able to use all the games from the IGGP dataset. Some games caused issues with Sanchos simulations and thus is was not possible to run experiments with them. A total of 36 out of the 50 games in the dataset were used. All games used from the IGGP dataset has a limit on the number of number of moves that can be taken however it varied from game to game. The games always terminate weather they are in the goal state or not.

Due to restraints on computational power 30 random traces and 30 Sancho generated traces were generated. The traces were generated on a machine with 20GB of RAM and an 8 core processor.

[1] https://github.com/ggp-org/ggp-base

alquerque	dont touch	gt ultimatum	pentago
asylum	duikoshi	hex for three	platform jumpers
battle of numbers	eight puzzle	horseshoe	rainbow
breakthrough	farming	hunter	sheep and wolf
buttons and lights	fizz buzz	knights tour	sudoku
centipede	forager	kono	sukoshi
checkers	gt centipede	light board	tic tac toe
coins	gt chicken	multiple buttons and lights	ttcc4
connect4team	gt prisoner	nine board tic tac toe	untwisty corridor

Table 5.1: Games used in the experiments

The ILP systems

The three ILP systems were trained on the random, intelligent and mixed game traces using the same settings as were used in the IGGP paper[11]. These settings are:

- Metagol - Metagol 2.2.3 with YAP 6.2.2. The metarules used can be found in the IGGP code repository.
- Aleph - Aleph 5 with YAP 6.2.2. The default Aleph parameters were used.
- ILASP - A specialised version ILASP* developed for this task was used and can be found in the IGGP code repository. It is based on ILASP2i.

The systems were run concurrently in both training and testing.

5.2 Methods

5.2.1 Training

Each system was given 15 minutes to generate a hypothesis for each predicate. If the predicate was not learned the default hypothesis was *true*.

In **E1** The ILP systems were trained on 8 full game traces. They were each trained on intelligent random and mixed traces. The mixed traces were made up of a 50/50 mix of random and intelligent traces.

In **E2** we trained each system on 8, 16 and 24 mixed traces where each set was made up of a 50/50 mix of random and intelligent traces. It was found that the results for 16 and 24 traces were very similar across all systems so no larger training sets were tested.

In the IGGP paper[11] the IGGP task was defined across four predicates for each game: *goal*, *next*, *terminal* and *legal*. After testing the systems on all four predicates it was found that the terminal predicates did not give an accurate picture into the ability to learn of each system. Each game had a maximum number of moves, these varied from game to game. Generally the randomly generated game traces did not complete the game before the move limit was reached. Since the terminal predicate is true on the last game state in each game and the game state includes a move counter this allowed the systems to simply learn the correlation between the maximum move and the terminal predicate. They would learn the program: `terminal :- true_step(MAX)` where `MAX` is the

maximum number of moves for the game. Since when a game is played randomly the max move count is almost always hit the predicate is often perfectly solved. However if when Sancho plays then game it solves it before the maximum move count is hit it is less likely to induce a correct rule. For this reason the *terminal* predicate was not included in training data.

5.2.2 Testing

Each system was tested on 4 randomly generated and 4 intelligently generated traces. To evaluate the performance of the ILP systems on the training dataset we use two metrics: balanced accuracy and perfectly solved.

Balanced accuracy In the datasets used for testing the ILP systems the vast majority of examples are negative. Balanced accuracy takes this into account when evaluating approaches. The system to be tested is the generated logical hypothesis H which, along with the background knowledge B for the relevant game. The test data is the set of combined positive and negative testing examples $E^+ \cup E^-$. We define the number of positive examples $p = |E^+|$, the number of negative examples $n = |E^-|$, the number of correctly predicted positives as $tp = |\{e \in E^+ | B \cup H \models e\}|$ and the number of correctly predicted negatives as $tn = |\{e \in E^- | B \cup H \not\models e\}|$. The balanced accuracy is subsequently defined as $ba = 100 \cdot (tp/p + tn/n)/2$.

Perfectly solved This metric considers the number of predicates for which the ILP system correctly classified all examples. It is equivalent to the number of games with a balanced accuracy score of 100. This metric is important since for all predicates there exists an exact solution (the rules that were used to generate the examples). A system that has correctly predicated 99% of the results is no where near as useful as one that predicts 100%.

In the results section (chapter 6) we present only the aggregate scores in the results since the full results are too large for this paper. To evaluate the systems we compare the average balanced accuracy and the number of perfectly solved games when tested on optimal and random traces.

Chapter 6

Results

We now describe the results of testing the ILP systems. We conducted the experiments in accordance with the methods described in chapter 5.

To recap the research questions are as follows:

- **Q1** - Does varying the quality of game traces influence the ability for learners to solve the IGGP problems? Specifically, does the quality of game play affect predictive accuracy?
- **Q2** - Does varying the amount of game traces influence the ability for learners to solve the IGGP problems? Specifically, does the quality of game play affect predictive accuracy?
- **Q3** - Can we improve the performance of a learner by mixing the quality of traces?

Experiment **E1** answers questions Q1 and Q3 and experiment **E2** answers question Q2.

6.1 Results summary

6.1.1 E1: Varying the quality of game traces

For questions Q1 and Q3 we preform experiment E1 where we train and test each system on random and intelligent traces as well as a 50/50 mixture of both. Table 6.1 shows the average balanced accuracy of each test of the systems in E1, table 6.2 shows the number of games for which the rule was perfectly learned for each test. The the differences for each system in the average balanced accuracy for the different training/testing combinations were unpronounced. To show this we use the χ^2 (chi squared) test to calculate the statistical significance of our results. The biggest difference between any two balanced accuracy testing results between two distributions for a single system was for Aleph where it was trained on Sancho traces and tested on Random (balanced accuracy of 65) and tested on Sancho (balanced accuracy of 72). The χ^2 test was performed on these two scores. The result was a p-value of 0.6723. This value is well above any reasonable threshold for statistical significance meaning the results can be viewed as not significant. Despite this there are still differences worth talking about between the different results.

Balanced Accuracy						
Systems	Random		Sancho		Mixed	
	Random	Sancho	Random	Sancho	Random	Sancho
Metagol	51	51	51	51	51	51
Aleph	66	65	65	72	67	66
ILASP	72	72	71	72	73	73

Table 6.1: The average balanced accuracies across all games and predicates for each system in E1. Each row gives the balanced accuracies of one system. Each was trained on random, Sancho generated and mixed traces then tested against random and Sancho generated. The top of the two header rows gives the training distribution and the lower gives the test distribution

Perfectly Solved						
Systems	Random		Sancho		Mixed	
	Random	Sancho	Random	Sancho	Random	Sancho
Metagol	0	0	0	0	0	0
Aleph	1	0	0	4	1	0
ILASP	7	8	5	10	5	8

Table 6.2: The number of perfectly solved scores for each system trained and tested as described in E1

Aleph

Throughout all experiments conducted the programs learned by Aleph consisted almost entirely of a disjunction of all cases seen in testing. For example when learning the *goal* predicate it relates every board state it sees with the goal value it has and adds it to the generated program. It appears that the most general clause (see section 2.4.2) that could be found for a lot of the examples was also the most specific clause. This simplistic method resulted in programs that performed better when the training and testing data was closely matched. This can be seen in the results for aleph when trained and tested on Sancho generated traces. For some games all Sancho generated traces were the same, in FizzBuzz Sancho always says the correct thing and in the Prisoner Dilemma[1] both Sancho players always defect. This allowed aleph to achieve a perfect score on the *next* predicate. In cases where the training data is diverse Aleph learns a program that over approximates the solution doing well on the tests with positive examples and badly on the tests with negative examples.

Generally Aleph was best at learning the *goal* predicate. Unlike *next* which takes a game state and player moves as input the *goal* predicate only takes the state (or less). This means that the input space was smaller and thus the the example inputs in the training came up more often in the testing. A lot of games only use a *goal* predicate that takes the role and associates it to the current goal value of that role despite it actually depending on the state which is included in the background knowledge. Compare this to the *legal* predicate which does explicitly take the state as input. The programs Aleph learns associate the goal

[1]Called gt_prisoner in the set of games. The players can choose to cooperate or defect, they each get three points if they cooperate but if one defects they get 5 points and the other gets none. If they both defect they both get 0.

only with the player role and give no regard to the actual state of the game since the program made up of facts. For example it might learn:

```
goal(red,0).
goal(black,0).
goal(black,8).
goal(red,8).
```

for a game where at some point red and black both score 0 and 8. In tests this gives the impression of aleph doing relatively well when in fact the program learned does not at all capture any aspects of the original game rule.

Aleph did not do particularly better when trained on the mixed training set. It achieved similar scores to the other tests apart from when it was trained and tested on optimal traces.

Metagol

A limitation with the Metagol system is that if it cannot learn a rule that does covers 100% of the positive training examples and 0% of the testing examples it will not output any rule at all. With the time limitations placed on the experiments due to limits in computational resources Metagol rarely managed this. If some rule contains a special case such as castling in checkers Metagol may have learned the general *next* predicate but would not give any output since this case was not covered. The search space scales exponentially with the size of the rules to be learned.

Metagol never learned any *legal* or *goal* predicates. The *goal* predicate is often relatively complex. It can rely on game concepts which would be obvious to humans but hard to learn for an ML system. For example the idea of three of the same symbol in a row, column or diagonal. There might not be enough training examples to guide Metagol to concepts such as these. The *legal* predicate is usually even more complex than the goal, however in some cases it the possible legal moves were constant across all states. It is possible that the metarules supplied to Metagol for this experiment were not ideal for this task.

Where Metagol learned the *next* predicate it often only managed the simple parts of it. The predicate is split up into different parts such as *next_state* and *next_step* to reflect the different predicates that change each move. If one of the predicates was *next_step(N)* where N increased by 1 each turn Metagol managed to learn that however it failed to produce any program that was more than two or three lines long.

ILASP

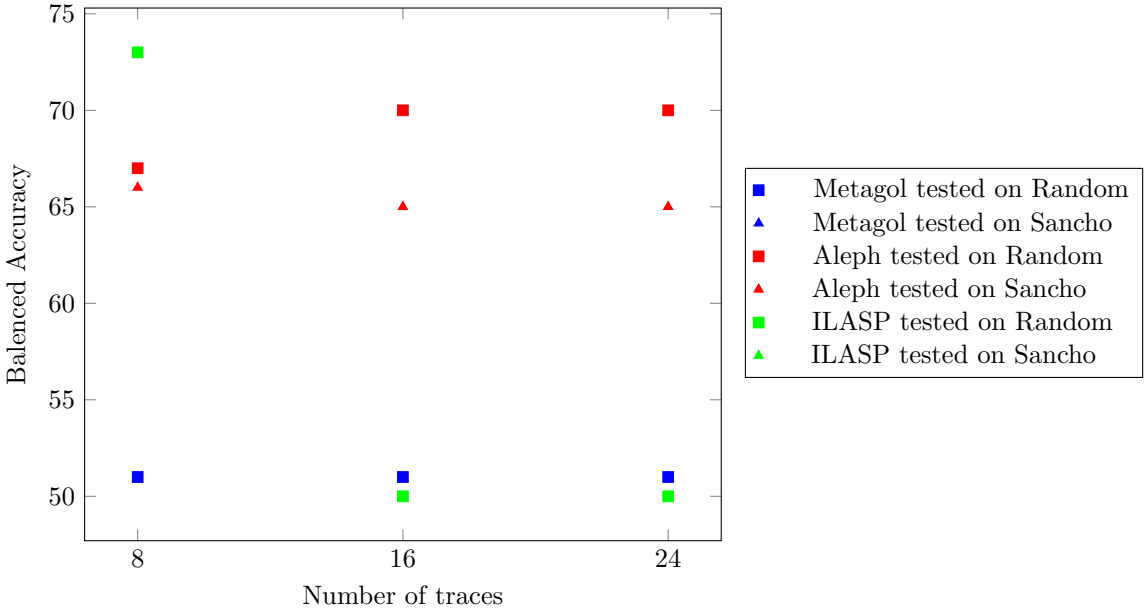
ILASP performed by far the best out of the three systems. The differences in results between the systems trained on Sancho and random data was not large. Out of them scored the best when trained and tested on Sancho traces presumably for the same reasons that Aleph did (the testing data was very similar to training data). The margin by which this scored better however was almost negligible, there was generally less than 2% difference between the balanced accuracies for each predicate of each game.

ILASP scored several pointed better across the board when trained on the mixed data. ILASP benefited the most from the combination of training data

out of all the systems. It scored better on the *goal* and *next* predicates but not on the *legal* predicate (compared to the Sancho and random trained systems). This makes sense because the legal predicate changes the least between the optimal and random traces, the same complexity of rule still generally has to be learned in each. The *next* predicate for *tic tac toe* was learned a lot better by ILASP on the mixed data. It is unclear exactly why this is the case but it does appearer ILASP learns better with greater diversity of examples.

6.1.2 E2: Varying the amount of game traces

In experiment E2 each system was trained three times on mixed traces. The first time with a total of 8 traces, the second with 16 traces and the third with 24 traces. When designing this experiment it was assumed that the more traces they were given to train on the better the systems would perform. However, as can been see in the scatter graph of the results this was not the case.



When ILASP and Metagol have not had enough time to learn a predicate they output nothing. Both systems scale exponentially in time taken according to the size of the input. Whilst for Metagol it didn't have time even for the 8 traces ILASP ran out of time only on the 16 and 24 traces training sets. This had the effect for ILASP of a large drop in effectiveness. For 16 and 24 the default program consisting of just *true* was used. This gave the balanced accuracy of 50 for each predicate. Metagol still managed to learn the simple predicates that it had managed on the 8 trace training set however did not increase at all in effectiveness over the rest of the experiment.

Unlike the other systems Aleph was consistently able to process the entire input since its method of learning is less computationally intense. When tested on the random traces aleph did better with the increased number of traces. This was most likely due to the fact that more cases were encountered in the training data. Since aleph usually ends up putting the cases it encounters straight into the program this simply meant there was a greater number of situations in the

testing set that aleph had seen before and was thus correctly able to classify. When tested on the optimal traces aleph got worse. This is possibly due to the maximum size of aleph programs taking effect. The clauses relevant to the optimal traces may be pushed out of the program by the vast number of clauses taken from the random gameplay examples.

If this experiment were to be conducted again much more enlightening results than these could be obtained by increasing the time given to each system to learn each predicate or even using a system with greater computational power to run the experiments.

Chapter 7

Conclusions

In this paper we have compared the ability of ILP systems to learn the rules of a game from observed gameplay in the IGGP framework on a range of training datasets. Primarily three ILP systems have been trained on random and intelligent gameplay as generated by the Sancho system. The generated hypotheses of the learners have been tested on random and intelligent gameplay from the same distributions as the training data. Differences in the effectiveness of the learned programs in correctly classifying legal and illegal gameplay were observed however there was not enough statistical significance in the results to disprove the null hypothesis. The systems were also trained on a mixture of optimal and random traces with similar results.

To test the significance of the size of the training dataset on the ability of the ILP systems to learn the systems were trained on varying numbers of game playouts. It was a training dataset of above 16 game traces had little effect on the ability to learn of the systems tested.

Limitations

There are several things that could be improved on in this paper.

Computational resources The computational resources and thus the time given for each system to learn the rules were severely limited for the experiments in this paper. If they were to be conducted again with greater computational resource with more time given to each system more significant results may be achieved

ILP systems In this paper we only test on three ILP systems, it is clear a more representative result could be obtained by testing on more systems. Techniques such as probabilistic[3, 37] and interactive[36] learning are not used by any of the systems tested here. ILASP is the only system designed to handle noisy data[30], it would be interesting to try others with this approach[35, 17]. The systems used also have many customisable settings for example Metagols metarules which have major implications for the programs learnt[15]. Aleph has many different search methods on the hypothesis space many of which may yield better results than the default algorithm used in this paper.

Generating traces Whilst Sancho generates good examples of intelligent play it would be interesting to see traces generated by other general game playing

methods[47, 26]. For some games there exists a provably optimal set of moves in some cases such as for eight puzzle Sancho generates these moves (see section 4.1) however it does not for all such games[39]. Future research could compare the effects on the learned hypotheses when trained on optimal data. It may also be more insightful to look at the difference between human generated data and random or optimal.

Other machine learning systems There exists a huge variety of machine learning systems other than ILP. It is possible some of these may exhibit more of a bias toward one of the training sets. Testing a neural network or genetic algorithm based approach may provide insightful results.

Intelligent verses random in other contexts There are plenty of other domains other than game playing in which datasets can be considered intelligent or random. An example of this might be training a car to drive where you could train it on different quality levels of driving or training a facial recognition system on poor or high quality photos. These would prove an interesting area to study.

How do humans compare We have spent a lot of this paper investigating whether machines learn better on intelligent gameplay or random gameplay with the assumption that a human would learn better from watching another human play than random legal moves. Whilst in reality we might generally choose to watch a human playing when trying to learn a game rather than observing random legal moves it may not be the most effective way. Any real world example of this is almost always accompanied by additional information. For example humans would often have access to some description of the rules or someone explaining the edge cases that may not occur. It is hard to compare the how humans learn to how machines learn since the extra context a human is given around a game is so much greater than that of a machine which gets very little. It would be interesting to see how humans would fare in similar experiments to those in this paper. The experiment would have to include games unknown to the participant which preferably would not share too many common features with know games. Looking into this question would give new insights into the results of this paper. As with the similar question posed in this paper the experiment does not have clear predicted outcome.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] John Ahlgren and Shiu Yin Yuen. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Mach. Learn. Res.*, 14(1):3649–3682, 2013.

[3] Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory Pract. Log. Program.*, 15(2):169–212, 2015.

[4] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine Learning*, 79(1-2):151–175, 2010.

[5] Svetla Boytcheva. Overview of inductive logic programming (ilp) systems. 01 2002.

[6] Ivan Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012.

[7] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, October 22-24, 2008, Stanford, California, USA*, 2008.

[8] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, pages 91–97, 2011.

[9] Andrew Cropper. *Efficiently learning efficient programs*. PhD thesis, Imperial College London, UK, 2017.

[10] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. *CoRR*, abs/2002.11002, 2020.

[11] Andrew Cropper, Richard Evans, and Mark Law. Inductive general game playing. *CoRR*, abs/1906.09627, 2019.

[12] Andrew Cropper, Rolf Morel, and Stephen H. Muggleton. Learning higher-order logic programs. *CoRR*, abs/1907.10953, 2019.

[13] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.

[14] Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. *Machine Learning*, 108(7):1063–1083, 2019.

[15] Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *CoRR*, abs/1907.10952, 2019.

[16] Arun Sharma Eric McCreath. Extraction of meta-knowledge to restrict the hypothesis space for ilp systems. In *Proc. of the 8th Australian Joint Conf. on AI*, 1995.

[17] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.

[18] Bertram Felgenhauer and Frazer Jarvis. Mathematics of sudoku i. *Mathematical Spectrum*, 39, 01 2006.

[19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Patrick Lühne, Philipp Obermeier, Max Ostrowski, Javier Romero, Torsten Schaub, Sebastian Schellhorn, and Philipp Wanko. The potsdam answer set solving collection 5.0. *KI*, 32(2-3):181–182, 2018.

[20] Michael Genesereth. Gdl website.

[21] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[22] Carlos R. González and Yaser S. Abu-Mostafa. Mismatched training and test distributions can outperform matched ones. *Neural Computation*, 27(2):365–387, 2015.

[23] Feng-Hsiung Hsu. Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–81, 1999.

[24] ISO/IEC. Information technology programming languages prolog part 1: General core. ISO 13211-1:1995, International Organization for Standardization, Geneva, Switzerland, 1995.

[25] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pages 282–293, 2006.

[26] Jakub Kowalski and Marek Szykula. Experimental studies in general game playing: An experience report. *CoRR*, abs/2003.03410, 2020.

[27] Mark Law. Ilasp manuel. Technical report.

[28] Mark Law. *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK, 2018.

[29] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Logics in Artificial Intelligence - 14th European Conference, 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, pages 311–325, 2014.

[30] Mark Law, Alessandra Russo, and Krysia Broda. Iterative learning of answer set programs from context dependent examples. *Theory Pract. Log. Program.*, 16(5-6):834–848, 2016.

[31] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification, 2006.

[32] Tom M. Mitchell. *Machine learning, International Edition.* McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[33] Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991.

[34] Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995.

[35] Andrej Oblak and Ivan Bratko. Learning from noisy data using a non-covering ILP algorithm. In Paolo Frasconi and Francesca A. Lisi, editors, *Inductive Logic Programming - 20th International Conference, ILP 2010, Florence, Italy, June 27-30, 2010. Revised Papers*, volume 6489 of *Lecture Notes in Computer Science*, pages 190–197. Springer, 2010.

[36] Luc De Raedt and Maurice Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Mach. Learn.*, 8:107–150, 1992.

[37] Luc De Raedt, Anton Dries, Ingo Thon, Guy Van den Broeck, and Mathias Verbeke. Inducing probabilistic relational rules from probabilistic examples. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1835–1843. AAAI Press, 2015.

[38] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, 1994.

[39] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[40] Uwe Schoning. *Logic for Computer Scientists.* Birkhäuser Basel, 2008.

[41] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[42] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[43] Andrew Rose Steve Draper. Sancho ggp player. https://github.com/SanchoGGP/ggp-base, 2015.

[44] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[45] Jan-Jaap van Horssen. Complexity of checkers and draughts on different board sizes. *ICGA Journal*, 40(4):341–352, 2018.

[46] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swiprolog. *CoRR*, abs/1011.5332, 2010.

[47] Maciej wiechowski, HyunSoo Park, Jacek Madziuk, and Kyung-Joong Kim. Recent advances in general game playing. *TheScientificWorldJournal*, 2015:986262, 09 2015.